Mapcrafter Documentation

Release 2.4

Moritz Hilscher

Jul 31, 2017

Contents

CHAPTER 1

Welcome

Mapcrafter is a high performance Minecraft map renderer written in C++. It renders Minecraft worlds to a bunch of images which are viewable like a Google Map in any webbrowser using Leaflet.js.

It runs on Linux and other Unix-like operating systems like Mac OS and has also an experimental support for Windows (see *Windows*). The renderer works with the Anvil world format and the Minecraft 1.6 resource packs.

Mapcrafter is free software and available under the GPL license. You can access the latest source code of Mapcrafter on GitHub: https://github.com/mapcrafter/mapcrafter

There are a few example maps of the renderer on the GitHub Wiki. Please feel free to add your own map to this list.

CHAPTER 2

Features

- **Web output:** Render your Minecraft worlds to maps viewable in any webbrowser!
- **Different render views:** Choose between different perspectives to render your world from! A 2D topdown and a 3D isometric render view are available!
- **Different rotations:** Choose from four different rotations to render your worlds from!
- **Different render modes:** Choose between different render modes like day, night and cave for your maps!
- **Different overlays:** Show additional information on your map! For example: Where can slimes spawn? Where can monsters spawn at night?
- **Configuration files:** Highly-customizable which worlds are rendered with which render view and other render parameters!
- **Markers:** Automatically generated markers from your Minecraft world data!
- **Other stuff:** Biome colors, incremental rendering, multithreading

CHAPTER 3

Help

Read *Using Mapcrafter* to get a first insight how to use the renderer. You can find a detailed documentation about the render configuration file format in *Configuration File Format*.

If you find bugs or problems when using Mapcrafter or if you have ideas for new features, then please feel free to add an issue to the GitHub issue tracker.

You can contact me in IRC (#mapcrafter on Freenode). Use the webclient if you are new to IRC. I will be there most of the time, but please bear in mind that I can't be available all the time. If I'm not there, wait some time or try another time of the day.

CHAPTER 4

Documentation Contents

Installation

Requirements

- A Linux-based or Mac operating system would be good, building the renderer on Windows is possible but not easy.
- A decent C++ compiler (preferable gcc >= 4.4, or clang), CMake and make to build Mapcrafter.
- Some libraries:
  - libpng
  - libjpeg (but you should use libjpeg-turbo as drop in replacement)
  - libboost-iostreams
  - libboost-system
  - libboost-filesystem (>= 1.42)
  - libboost-program-options
  - (libboost-test if you want to use the tests)
- For your Minecraft worlds:
  - Anvil world format
  - Minecraft 1.6 resource packs

Building from Source

General Instructions

At first you have to get the source code of Mapcrafter. Clone it directly from GitHub if you want the newest version:

```
git clone https://github.com/mapcrafter/mapcrafter.git
```

Make sure you have all requirements installed. If you are on a Debian-like Linux system, you can install these packages with apt:

```
sudo apt-get install libpng-dev libjpeg-dev libboost-iostreams-dev \
libboost-system-dev libboost-filesystem-dev libboost-program-options-dev \
build-essential cmake
```

If you are on an RPM based system such as Fedora, you can install these packages with yum:

```
sudo yum install boost-devel libjpeg-devel libpng-devel gcc-c++ make cmake
```

Then you can go into the directory with the Mapcrafter source (for example `mapcrafter/`, not `mapcrafter/src/`) and build it with the following commands:

```
cmake .
make
```

If everything works, you should have an executable file `mapcrafter` in the `src/` directory.

You can now install Mapcrafter system-wide for all users if you want:

```
sudo make install
```

If you get an error concerning `libmapcraftercore.so` not found, you have to run `ldconfig` (as root).

On CentOS and other RHEL and Fedora derived distributions, you may have to add `/usr/local/lib` and `/usr/local/lib64` to `/etc/ld.so.conf.d/usrlocal.conf` and run `ldconfig -v`

Don't forget that you still have to install the texture files needed for Mapcrafter. If you install the texture files to `src/data/textures`, they will be copied to a path Mapcrafter will automatically detect when installing Mapcrafter with `make install`.

FreeBSD 10

Mapcrafter builds fine on FreeBSD 10, 9 is not tested but could also build there.

For this guide we will be using ports, but could work with packages from pkgng (untested).

First step is to install prerequisites:

```
cd /usr/ports/devel/git
make install clean; rehash
cd /usr/ports/devel/boost-all
make install clean; rehash
cd /usr/ports/devel/cmake
make install clean; rehash
cd /usr/ports/misc/compat8x
make install clean; rehash
cd /usr/ports/graphics/png
make install clean; rehash
```

Or if you got portmaster installed:

```
portmaster devel/git devel/boost-all devel/cmake misc/compat8x graphics/png
```

Once this is done compiling (takes a long time), you can go ahead with the normal steps:

```
git clone https://github.com/mapcrafter/mapcrafter.git
cd mapcrafter
cmake .
make
```

Mac OS X

Currently there are no pre built packages available for Mac OS X but building it is relatively simple.

Prerequisites:

- Xcode
- Homebrew or Macports

Depending on your version of OS X you may or may not have git installed. Starting from 10.9 Mavericks git is installed with Xcode, if you got 10.8 Mountain Lion or older, you must install command line tools from Xcode and run the following command:

```
brew install git
```

On 10.9 Mavericks systems you will have to run the following command after you've installed Xcode:

```
xcode-select --install
```

and select install in the window that pops up, and accept the EULA.

First you will have to clone the latest Mapcrafter source by running:

```
git clone https://github.com/mapcrafter/mapcrafter.git
```

After this, install the dependencies using brew:

```
brew install boost libpng cmake libjpeg-turbo
```

Or install the dependencies using port:

```
port install boost libpng cmake libjpeg-turbo
```

Once you have run this, you should have a working build system for Mapcrafter:

```
cd mapcrafter
cmake .
make
```

This will build Mapcrafter and put the ready to use binary in the `src/` directory.

Note: With homebrew you will have to run the following CMake command:

```
cmake . -DJPEG_INCLUDE_DIR=/usr/local/opt/jpeg-turbo/include/ -DJPEG_LIBRARY=/usr/
↪local/opt/jpeg-turbo/lib/libjpeg.dylib
```

Arch Linux

If you are running Arch Linux as operating system, you can install Mapcrafter from the AUR.

Debian Packages

If you are running Debian or Ubuntu, you can use the already built Mapcrafter Debian packages.

If you are using Debian, run the following commands in a shell:

```
echo "deb http://packages.mapcrafter.org/debian $(lsb_release -sc) main" | sudo tee /
↪etc/apt/sources.list.d/mapcrafter.list
sudo wget -O /etc/apt/trusted.gpg.d/mapcrafter.gpg http://packages.mapcrafter.org/
↪debian/keyring.gpg
```

If you are using Ubuntu, run the following commands in a shell:

```
echo "deb http://packages.mapcrafter.org/ubuntu $(lsb_release -sc) main" | sudo tee /
↪etc/apt/sources.list.d/mapcrafter.list
sudo wget -O /etc/apt/trusted.gpg.d/mapcrafter.gpg http://packages.mapcrafter.org/
↪ubuntu/keyring.gpg
```

The commands above add the Mapcrafter Debian package repository to your package manager and import the public key which was used to sign the packages.

Now you can run `sudo apt-get update` to tell your package manager about the sources and `sudo apt-get install mapcrafter` to install Mapcrafter. During this process it will automatically download a temporary Minecraft Jar file and unpack required texture files.

Ubuntu Vivid Vervet (15.04), Trusty Tahr (14.04 LTS), Precise Pangolin (12.04 LTS), Debian Jessie (stable) and Wheezy (oldstable) i386/amd64 are supported at the moment. You have to build Mapcrafter from source if you are using another distribution / version. If you think that there is an important distribution / version missing, please contact me.

There is also a "nightly channel" of packages built every night from the newest source code (`nightly` instead of `main` sources list file). Those packages are primarily built to make sure that no build problems on the different platforms arise while doing development work on Mapcrafter.

Windows

You can download prebuilt packages for Windows from mapcrafter.org:

http://mapcrafter.org/downloads

Mapcrafter for Windows is cross-compiled on Linux using mingw-w64. You can find the CMake toolchain files on GitHub if you want to build it on your own:

https://github.com/mapcrafter/mapcrafter-buildfiles/tree/master/windows-cross

Having all the depedencies ready is a bit complicated, that's why I'm using Arch Linux which has AUR packages for mingw-w64 and all the required libraries.

Resources and Textures

Note: You don't need to install the Minecraft texture files manually if you installed Mapcrafter from the AUR or with the Debian package.

Mapcrafter needs some resources to render maps: Minecraft texture files and some template files for the web output.

There are different directories Mapcrafter searches these files:

1. $HOME/.mapcrafter
2. $PREFIX/share/mapcrafter
3. $MAPCRAFTER/data

$HOME is your home directory (usually /home/<username>). $PREFIX is the directory where Mapcrafter is installed (mostly /usr or /usr/local, if installed via Debian package or make install). $MAPCRAFTER is the directory of the Mapcrafter executable. The third path is used if you built Mapcrafter from source and run it directly without installing.

The template and texture files in these resource directories are expected by the renderer in template/, the texture files in textures/.

You can get the paths to the resource directories of Mapcrafter by running mapcrafter --find-resources. For example, when I installed the Debian package:

```
$ mapcrafter --find-resources
Your home directory: /home/moritz
Mapcrafter binary: /usr/bin/mapcrafter
Resource directories:
  1. /home/moritz/.mapcrafter
  2. /usr/share/mapcrafter
Template directories:
  1. /usr/share/mapcrafter/template
Texture directories:
  1. /usr/share/mapcrafter/textures
Logging configuration file:
  1. /etc/mapcrafter/logging.conf
```

You can see that Mapcrafter found a resource directory in the home directory but no template/ or textures/ directory in it. So it's just using the template and texture directories in /usr/share/mapcrafter. The numbers in front of the paths are the order Mapcrafter is using these directories. If you want to overwrite the default textures, you can just create a new texture directory .mapcrafter/textures in your home directory.

Now you have to install the Minecraft texture files. You need the following files in your texture directory:

- entity/chest/normal.png
- entity/chest/normal_double.png
- entity/chest/ender.png
- entity/chest/trapped.png
- entity/chest/trapped_double.png
- colormap/foliage.png
- colormap/grass.png
- blocks/ with block texture files
- endportal.png

You can get those files from your Minecraft Jar file (default textures) or from another resource pack. To extract these texture files there is a python script mapcrafter_textures.py (src/tools/mapcrafter_textures.py in the Mapcrafter source if you didn't install Mapcrafter on your system). Run the python script with the Minecraft Jar file and the texture directory as arguments:

```
mapcrafter_textures.py /path/to/my/minecraft/jar/1.8.jar /my/texture/directory
```

You will probably find your Minecraft Jar file in ~/.minecraft/versions/%version%/%version%.jar.

Using Mapcrafter

First Rendered World

At first you have to create a configuration file like this:

```
output_dir = output

[world:myworld]
input_dir = worlds/myworld

[map:map_myworld]
name = My World
world = myworld
```

In the configuration file you define which worlds the renderer should render. In this example is defined that the renderer should render the world in the directory `worlds/myworld/` as the map `map_myworld` into the output directory `output/`. All relative paths in configuration files are relative to the path of the configuration file.

Now it's time to render your first world:

```
mapcrafter -c render.conf
```

To improve the performance you can also render the map with multiple threads:

```
mapcrafter -c render.conf -j 2
```

2 is here the number of threads the renderer uses. You should use the count of your CPU cores. With increasing thread count I/O (reading the world, writing the rendered tiles to disk) mostly becomes the bottleneck so using more threads than CPU cores is not useful.

You can see your rendered map by opening the `index.html` file in the output directory with your webbrowser.

For more information about rendering maps see *Configuration File Format* and the next section about command line options.

Command Line Options

Here is a list of available command line options:

General options

-h, --help
: Shows a help about the command line options.

-v, --version
: Shows the version of Mapcrafter.

Logging/output options

--logging-config `<file>`
: This option sets the global logging configuration file Mapcrafter's logging facility is using. You do not necessarily need to specify a logging configuration file, Mapcrafter is trying to determine it automatically.

--color <colored>
: This option specifies whether Mapcrafter's logging facility should use a colored terminal output for special messages (warnings/errors). Possible options are `true`, `false` or `auto` (default). `auto` means that terminal colors are enabled if the output is connected to a tty (and not piped to a file for example).

-b, --batch
: This option deactivates the animated progress bar and enables the progress logger instead as terminal output. It is also automatically enabled if the output is not connected to a tty (piped to a file for example).

Renderer options

--find-resources
: Shows the resource directories of Mapcrafter. See also *Resources and Textures*.

-c <file>, **--config** <file>
: This is the path to the configuration file to use when rendering and is **required**.

-s <maps>, **--render-skip** <maps>
: You can specify maps the renderer should skip when rendering. This is a space-separated list of map names (the map section names from the configuration file). You can also specify the rotation of the maps to skip by adding a `:` and the short name of the rotation (`tl`, `tr`, `br`, `bl`).

  For example: `-s world world2` or `-s world:tl world:bl world2:bl world3`.

-r, --render-reset
: This option skips all maps and renders only the maps you explicitly specify with `-a` or `-f`.

  Note: This option is useful if you want to update only the template of your rendered map:

  ```
  mapcrafter -c render.conf -r
  ```

-a <maps>, **--render-auto** <maps>
: You can specify maps the renderer should render automatically. This means that the renderer renders the map incrementally, if something was already rendered, or renders the map completely, if this is the first rendering. Per default the renderer renders all maps automatically. See `--render-skip` for the format to specify maps.

-f <maps>, **--render-force** <maps>
: You can specify maps the renderer should render completely. This means that the renderer renders all tiles, not just the tiles, which might have changed. See `--render-skip` for the format to specify maps.

-F, --render-force-all
: This option is similar to the `-f` option, but it makes Mapcrafter force-render all maps.

-j <number>, **--jobs** <number>
: This is the count of threads to use (defaults to one), when rendering the map. Using as much threads as CPU cores you have is good, but the rendering performance also depends heavily on your disk. You can render the map to a solid state disk or a ramdisk to improve the performance.

  Every thread needs around 150MB ram.

Configuration File Format

To tell the Mapcrafter which maps to render, simple INI-like configuration files are used. With configuration files it is possible to render maps with multiple rotations and render modes into one output file.

A First Example

Here is a simple example of a configuration file (let's call it `render.conf`):

```
output_dir = myworld_mapcrafter

[world:myworld]
input_dir = worlds/myworld

[map:myworld_isometric_day]
world = myworld
```

As you can see the configuration files consist of different types of sections (e.g. `[section]`) and containing assignments of configuration options to specific values (e.g. `key = value`). The sections have their names in square brackets, where the prefix with the colon shows the type of the section.

There are three types (actually four, but more about that later) of sections:

- World sections (e.g. sections starting with `world:`)
- Map sections (e.g. sections starting with `map:`)
- Marker sections (e.g. sections starting with `marker:`, also see *Markers*)

Every world section represents a Minecraft world you want to render and needs a directory where it can find the Minecraft world (`input_dir` of the world section `myworld` in the example above).

Every map section represents an actual rendered map of a Minecraft world. You can specify things like rotation of the world, render view, render mode, texture pack, texture size, etc. for each map.

In this example you can see that we have a world `myworld` in the directory `worlds/myworld/` which is rendered as the map `myworld_isometric_day`. The directory `output/` is set as output directory. After the rendering you can open the `index.html` file in this directory and view your rendered map.

As you can see the configuration option `output_dir` is not contained in any section - it's in the so called *root section*. That's because all maps are rendered into this directory and viewable via one `index.html` file, so the `output_dir` option is the same for all maps in this configuration file.

Also keep in mind that you can choose the section names (but not the section types!) on your own, though it is recommended to use some kind of a fixed format (for example `<world name>_<render view>_<render mode>` for maps) to keep things consistent.

Let's have a look at a more advanced configuration file.

A More Advanced Example

```
output_dir = output

[global:map]
world = world
render_view = isometric
render_mode = daylight
rotations = top-left bottom-right
texture_size = 12

[world:world]
input_dir = worlds/world

[world:creative]
```

```
input_dir = worlds/creative

[map:world_isometric_day]
name = Normal World - Day

[map:world_isometric_night]
name = Normal World - Night
render_mode = nightlight

[map:world_isometric_cave]
name = Normal World - Cave
render_mode = cave

[map:world_topdown_day]
name = Normal World - Topdown overview
render_view = topdown
texture_size = 6
texture_blur = 2
tile_width = 3

[map:creative_isometric_day]
name = Creative World - Day
world = creative
render_mode = daylight
rotations = top-left top-right bottom-right bottom-left
texture_dir = textures/special_textures
texture_size = 16

[map:creative_isometric_night]
name = Creative World - Night
world = creative
render_mode = nightlight
rotations = top-left top-right bottom-right bottom-left
texture_dir = textures/special_textures
texture_size = 16
```

Here we have some more worlds and maps defined. We have a "normal" world which is rendered with the day, night, cave render mode, and also with the top view and a lower texture size as overview map. Also we have a "creative" world which is rendered with a special texture pack, higher texture size and all available world rotations with the day and night render mode (super fancy!).

As you can see there is a new section `global:map`. This section is used to set default values for all map sections. Because of this in this example every map has the world `world`, the 3D isometric render view, the daylight render mode, the world rotations top-left and top-right and the 12px texture size as default. Of course you can overwrite these settings in every map section. There is also a global section `global:world` for worlds, but at the moment there is only one configuration option for worlds (`input_dir`), so it doesn't make much sense setting a default value here.

Furthermore every map has as option `name` a name which is used in the web interface of the output HTML-File. This can be anything suitable to identify this map. In contrast to that the world and map names in the sections are used for internal representation and therefore should be unique and contain only alphanumeric chars and underscores.

When you have now your configuration file you can render your worlds with (see *Command Line Options* for more options and usage):

```
mapcrafter -c render.conf
```

There are tons of other options to customize your rendered maps. Before a reference of all available options, here is a quick overview of interesting things you can do:

- Default view / zoom level / rotation in web interface
- World cropping (only render specific parts of your world)
- Block mask (skip rendering / render only specific types blocks)
- Different render views, render modes, overlays
- Use custom texture packs, texture sizes, apply a blur effect to textures
- Custom tile widths
- Different image formats
- Custom lighting intensity

Available Options

General Options

Note: These options are relevant for all worlds and maps, so you have to put them in the header before the first section starts

`output_dir = <directory>`

Required

This is the directory where Mapcrafter saves the rendered map. Every time you render your map the renderer copies the template files into this directory and overwrites them, if they already exist. The renderer creates an `index.html` file you can open with your webbrowser. If you want to customize this HTML-File, you should do this directly in the template (see `template_dir`) because this file is overwritten every time you render the map.

`template_dir = <directory>`

Default: default template directory (see *Resources and Textures*)

This is the directory with the web template files. The renderer copies all files, which are in this directory, to the output directory and replaces the variables in the `index.html` file. The `index.html` file is also the file in the output directory you can open with your webbrowser after the rendering.

`background_color = <hex color>`

Default: `#DDDDDD`

This is the background color of your rendered map. You have to specify it like an HTML hex color (`#rrggbb`).

The background color of the map is set with a CSS option in the template. Because the JPEG image format does not support transparency and some tiles are not completely used, you have to re-render your maps which use JPEGs if you change the background color.

World Options

Note: These options are for the worlds. You can specify them in the world sections (the ones starting with world:) or you can specify them in the global:world section. If you specify them in the global section, these options are default values and inherited into the world sections if you do not overwrite them.

`input_dir = <directory>`

Required

This is the directory of your Minecraft world. The directory should contain a directory `region/` with the .mca region files.

`dimension = nether|overworld|end`

Default: `overworld`

You can specify with this option the dimension of the world Mapcrafter should render. If you choose The Nether or The End, Mapcrafter will automagically detect the corresponding region directory. It will try the Bukkit region directory (for example `myworld_nether/DIM-1/region`) first and then the directory of a normal vanilla server/client (for example `myworld/DIM-1/region`).

Note: If you want to render The Nether and want to see something, you should use the cave render mode or use the crop_max_y option to remove the top bedrock layers.

`world_name = <name>`

Default: `<name of the world section>`

This is another name of the world, the name of the world the server uses. You don't usually need to specify this manually unless your server uses different world names and you want to use the mapcrafter-playermarkers script.

`default_view = <x>,<z>,<y>`

Default: Center of the map

You can specify the default center of the map with this option. Just specify a position in your Minecraft world you want as center when you open the map.

`default_zoom = <zoomlevel>`

Default: `0`

This is the default zoom level shown when you open the map. The default zoom level is 0 (completely zoomed out) and the maximum zoom level (completely zoomed in) is the one Mapcrafter shows when rendering your map.

`default_rotation = top-left|top-right|bottom-right|bottom-left`

Default: First available rotation of the map

This is the default rotation shown when you open the map. You can specify one of the four available rotations. If a map doesn't have this rotation, the first available rotation will be shown.

By using the following options you can crop your world and render only a specific part of it. With these two options you can skip blocks above or below a specific level:

`crop_min_y = <number>`

Default: -infinity

This is the minimum y-coordinate of blocks Mapcrafter will render.

`crop_max_y = <number>`

Default: infinity

This is the maximum y-coordinate of blocks Mapcrafter will render.

Furthermore there are two different types of world cropping:

1. Rectangular cropping:

- You can specify limits for the x- and z-coordinates. The renderer will render only blocks contained in these boundaries. You can use the following options whereas all options are optional and default to infinite (or -infinite for minimum limits):
  - `crop_min_x` (minimum limit of x-coordinate)
  - `crop_max_x` (maximum limit of x-coordinate)
  - `crop_min_z` (minimum limit of z-coordinate)
  - `crop_max_z` (maximum limit of z-coordinate)

2. Circular cropping:

- You can specify a block position as center and a radius. The renderer will render only blocks contained in this circle:
  - `crop_center_x` (**required**, x-coordinate of the center)
  - `crop_center_z` (**required**, z-coordinate of the center)
  - `crop_radius` (**required**, radius of the circle)

Note: The renderer automatically centers circular cropped worlds and rectangular cropped worlds which have all four limits specified so the maximum zoom level of the rendered map does not unnecessarily become as high as the original map.

Changing the center of an already rendered map is complicated and therefore not supported by the renderer. Due to that you should completely rerender the map when you want to change the boundaries of a cropped world. This also means that you should delete the already rendered map (delete <output_dir>/<map_name>).

The provided options for world cropping are very versatile as you can see with the next two options:

`crop_unpopulated_chunks = true|false`

Default: `false`

If you are bored of the chunks with unpopulated terrain at the edges of your world, e.g. no trees, ores and other structures, you can skip rendering them with this option. If you are afraid someone might use this to find rare ores such as Diamond or Emerald, you should not enable this option.

`block_mask = <block mask>`

Default: *show all blocks*

With the block mask option it is possible to hide or shown only specific blocks. The block mask is a space separated list of block groups you want to hide/show. If a `!` precedes a block group, all blocks of this block group are hidden, otherwise they are shown. Per default, all blocks are shown. Possible block groups are:

- All blocks:
  - `*`
- A single block (independent of block data):
  - `[blockid]`
- A single block with specific block data:
  - `[blockid]:[blockdata]`
- A range of blocks:

  - `[blockid1]-[blockid2]`
- All blocks with a specific id and `(block data & bitmask) == specified data`:
  - `[blockid]:[blockdata]b[bitmask]`

For example:

- Hide all blocks except blocks with id 1,7,8,9 or id 3 / data 2:
  - `!* 1 3:2 7-9`
- Show all blocks except jungle wood and jungle leaves:
  - `!17:3b3 !18:3b3`
  - Jungle wood and jungle leaves have id 17 and 18 and use data value 3 for first two bits (bitmask 3 = 0b11)
  - other bits are used otherwise -> ignore all those bits

Map Options

Note: These options are for the maps. You can specify them in the map sections (the ones starting with map:) or you can specify them in the global:map section. If you specify them in the global section, these options are default values and inherited into the map sections if you do not overwrite them.

`name = <name>`

Default: `<name of the section>`

This is the name for the rendered map. You will see this name in the output file, so you should use here an human-readable name. The belonging configuration section to this map has also a name (in square brackets). Since the name of the section is used for internal representation, the name of the section should be unique and you should only use alphanumeric chars.

`render_view = isometric|topdown`

Default: `isometric`

This is the view that your world is rendered from. You can choose from different render views:

`isometric` A 3D isometric view looking at north-east, north-west, south-west or south-east (depending on the rotation of the world).

`topdown` A simple 2D top view.

`render_mode = plain|daylight|nightlight|cave`

Default: `daylight`

This is the render mode to use when rendering the world. Possible render modes are:

`plain` Plain render mode without lighting or other special magic.

`daylight` Renders the world with lighting.

`nightlight` Like `daylight`, but renders at night.

`cave` Renders only caves and colors blocks depending on their height to make them easier to recognize.

Note: The old option name `rendermode` is still available, but deprecated. Therefore you can still use it in old configuration files, but Mapcrafter will show a warning.

`overlay = slime|spawnday|spawnnight`

Default: `none`

Additionally to a render mode, you can specify an overlay. An overlay is a special render mode that is rendered on top of your map and the selected render mode. The following overlays are used to show some interesting additional data extracted from the Minecraft world data:

`none` Empty overlay.

`slime` Highlights the chunks where slimes can spawn.

`spawnday` Shows where monsters can spawn at day.

`spawnnight` Shows where monsters can spawn at night.

At the moment there is only one overlay per map section allowed because the overlay is rendered just like a render mode on top of the world. If you want to render multiple overlays, you need multiple map sections. This behavior might change in future Mapcrafter versions so you will be able to dynamically switch multiple overlays on and off in the web interface.

`rotations = [top-left] [top-right] [bottom-right] [bottom-left]`

Default: `top-left`

This is a list of directions to render the world from. You can rotate the world by n*90 degrees. Later in the output file you can interactively rotate your world. Possible values for this space-separated list are: `top-left`, `top-right`, `bottom-right`, `bottom-left`.

Top left means that north is on the top left side on the map (same thing for other directions).

`texture_dir = <directory>`

Default: default texture directory (see *Resources and Textures*)

This is the directory with the Minecraft Texture files. The renderer works with the Minecraft 1.6 resource pack file format. You need here:

- directory `chest/` with normal.png, normal_double.png and ender.png
- directory `colormap/` with foliage.png and grass.png
- directory `blocks/` from your texture pack
- endportal.png

See also *Resources and Textures* to see how to get these files.

`texture_size = <number>`

Default: `12`

This is the size (in pixels) of the block textures. The default texture size is 12px (16px is the size of the default Minecraft Textures).

The size of a tile is `32 * texture_size`, so the higher the texture size, the more image data the renderer has to process. If you want a high detail, use texture size 16, but texture size 12 looks still good and is faster to render.

`texture_blur = <number>`

Default: `0`

You can apply a simple blur filter with a radius of `<number>` pixels to the texture images. This might be useful if you are using a very low texture size because areas with their blocks sometimes look a bit "tiled".

`water_opacity = <number>`

Default: `1.0`

With a factor from 0.0 to 1.0 you can modify the opacity of the used water texture before your map is rendered. 0 means that it is completely transparent and 1 means that the original opacity of the texture is kept. Also have a look at the `lighting_water_intensity` option.

Note: Don't actually set the water opacity to 0.0, that's a bad idea regarding performance. If you don't want to render water, have a look at the `block_mask` option.

`tile_width = <number>`

Default: `1`

This is a factor that is applied to the tile size. Every (square) tile is usually one chunk wide, but you can increase that size. The wider a tile is, the more blocks it contains and the longer it takes to render a tile, but the less tiles are to render overall and the less overhead there is when writing the tile images. Use this if your texture size is small and you want to prevent that a lot of very small tiles are rendered.

`image_format = png|jpeg`

Default: `png`

This is the image format the renderer uses for the tile images. You can render your maps to PNGs or to JPEGs. PNGs are losless, JPEGs are faster to write and need less disk space. Also consider the `png_indexed` and `jpeg_quality` options.

`png_indexed = true|false`

Default: `false`

With this option you can make the renderer write indexed PNGs. Indexed PNGs are using a color table with 256 colors (which is usually enough for this kind of images) instead of writing the RGBA values for every pixel. Like using JPEGs, this is another way of drastically reducing the needed disk space of the rendered images.

`jpeg_quality = <number between 0 and 100>`

Default: `85`

This is the quality to use for the JPEGs. It should be a number between 0 and 100, where 0 is the worst quality which needs the least disk space and 100 is the best quality which needs the most disk space.

`lighting_intensity = <number>`

Default: `1.0`

This is the lighting intensity, i.e. the strength the renderer applies the lighting to the rendered map. You can specify a value from 0.0 to 1.0, where 1.0 means full lighting and 0.0 means no lighting.

`lighting_water_intensity = <number>`

Default: `1.0`

This is like the normal lighting intensity option, but used for blocks that are under water. Usually the effect of opaque looking deep water is created by rendering just the top water layer and then applying the lighting

effect on the (dark) floor of the water. By decreasing the lighting intensity for blocks under water you can make the water look "more transparent". Use this option together with the `water_opacity` option. You might have to play around with this to find a configuration that you like. For me `water_opacity=0.75` and `lighting_water_intensity=0.6` didn't look bad.

`render_unknown_blocks = true|false`

Default: `false`

With this option the renderer renders unknown blocks as red blocks (for debugging purposes).

`render_leaves_transparent = true|false`

Default: `true`

You can specifiy this to use the transparent leaf textures instead of the opaque textures. Using transparent leaf textures can make the renderer a bit slower because the renderer also has to scan the blocks after the leaves to the ground.

`render_biomes = true|false`

Default: `true`

This setting makes the renderer to use the original biome colors for blocks like grass and leaves.

`use_image_mtimes = true|false`

Default: `true`

This setting specifies the way the renderer should check if tiles are required when rendering incremental. Different behaviors are:

Use the tile image modification times (`true`): The renderer checks the modification times of the already rendered tile images. All tiles whoose chunk timestamps are newer than this modification time are required.

Use the time of the last rendering (`false`): The renderer saves the time of the last rendering. All tiles whoose chunk timestamps are newer than this last-render-time are required.

Marker Options

Note: These options are for the marker groups. You can specify them in the marker sections (the ones starting with marker:) or you can specify them in the global:marker section. If you specify them in the global section, these options are default values and inherited into the marker sections if you do not overwrite them.

`name = <name>`

Default: *Name of the section*

This is the name of the marker group. You can use a human-readable name since this name is displayed in the webinterface.

`prefix = <prefix>`

Default: *Empty*

This is the prefix a sign must have to be recognized as marker of this marker group. Example: If you choose `[home]` as prefix, all signs whose text starts with `[home]` are displayed as markers of this group.

`postfix = <postfix>`

Default: *Empty*

This is the postfix a sign must have to be recognized as marker of this marker group.

Note: Note that prefix and postfix may not overlap in the text sign to be matched. Example: If you have prefix `foo` and postfix `oo bar` and your sign text says `foo bar`, it won't be matched. A sign with text `foo ooaoo bar` would be matched.

`title_format = <format>`

Default: `%(text)`

You can change the title used for markers (the name shown when you hover over a marker) by using different placeholders:

Placeholder	Meaning
`%(text)`	Complete text of the sign without the prefix/postfix.
`%(prefix)`	Configured prefix of this marker group.
`%(postfix)`	Configured postfix of this marker group.
`%(textp)`	Complete text of the sign with the prefix/postfix.
`%(line1)`	First line of the sign.
`%(line2)`	Second line of the sign.
`%(line3)`	Third line of the sign.
`%(line4)`	Fourth line of the sign.
`%(x)`	X coordinate of the sign position.
`%(z)`	Z coordinate of the sign position.
`%(y)`	Y coordinate of the sign position.

The title of markers defaults to the text (without the prefix/postfix) of the belonging sign, e.g. the placeholder `%(text)`.

You can use different placeholders and other text in this format string as well, for example `Marker at x=%(x), y=%(y), z=%(z):  %(text)`.

`text_format = <format>`

Default: *Format of the title*

You can change the text shown in the marker popup windows as well. You can use the same placeholders you can use for the marker title.

`icon = <icon>`

Default: *Default Leaflet marker icon*

This is the icon used for the markers of this marker group. You do not necessarily need to specify a custom icon, you can also use the default icon.

You can put your own icons into the `static/markers/` directory of your template directory. Then you only need to specify the filename of the icon, the path `static/markers/` is automatically prepended. You should also specify the size of your custom icon.

`icon_size = <size>`

Default: `[24, 24]`

This is the size of your icon. Specify it like `[width, height]`. The icon size defaults to 24x24 pixels.

`match_empty = true|false`

Default: `false`

This option specifies whether empty signs can be matched as markers. You have to set this to `true` if you set the prefix to an empty string to show all remaining unmatched signs as markers and if you want to show even empty signs as markers.

`show_default = true|false`

Default: `true`

With this option you can hide a marker group in the web interface by default.

Logging

Mapcrafter has its own logging facility which is configurable with a global logging configuration file as well as with the normal render configuration files. You can configure Mapcrafter to log its output into a log file or a local syslog daemon.

You can find your global logging configuration file with the `mapcrafter --find-resources` command, but it's usually installed as `/etc/mapcrafter/logging.conf` or directly available as `mapcrafter/src/logging.conf` (if Mapcrafter is not system/user-wide installed).

Here is a very simple example of a logging configuration file:

```
[log:output]
type = output
verbosity = INFO

[log:file]
type = file
verbosity = INFO
file = /var/log/mapcrafter.log

[log:syslog]
type = syslog
verbosity = INFO
```

Every log section (prefixed with `log:`) configures one log sink. You can configure the logging facility with those log sections in the global logging configuration file, but you can also use those log sections in the normal render configuration files. The log sections in the normal render configuration files are used after the ones in the global logging configuration file, so you can use them to overwrite the global logging configuration.

The names of the log sections are not relevant because you specify the type of the log sink with the `type` option. An exception of this are file log sinks. You should make sure that you do not use the same section name for file log sinks multiple times because they are used for internal representation.

Available options

General options

The following options are relevant for all log sink types.

`type = <type>`

Required

This is the type of the log sink you want to configure. Available types are:

output This is the default log output Mapcrafter shows when you run it. It is always enabled by default.

file This sink writes all log output into a log file.

syslog This sink sends all log output to the local syslog daemon.

`verbosity = <verbosity>`

Default: `INFO`

This is the verbosity of the log sink, i.e. the minimum log level a message must have to be handled by the log sink. Available log levels are (according to RFC 5424):

- `DEBUG`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `FATAL`, `ALERT`, `EMERGENCY`

`log_progress = true|false`

Default: `true` (except output log)

This option specifies whether the log sink should log progress messages. It is disabled by default for the output log because it is already using the animated progress bar. If you enable the `--batch` mode, this is also enabled for the output log and the animated progress bar is not shown.

Output and file log sink options

The following options are only relevant for the output and file log sinks.

`format = <format>`

Default: `%(date) [%(level)] [%(logger)] %(message)`

This is the format log messages are formatted with. You can use the following placeholders to specify the format of the log messages:

Placeholder	Meaning
`%(date)`	Current date formatted with date format.
`%(level)`	Log level of the logged message.
`%(logger)`	Logger used to log this message (usually `default` or `progress`).
`%(file)`	Source file name where this message was logged.
`%(line)`	Source file line number where this message was logged.
`%(message)`	The actually logged message.

`date_format = <dateformat>`

Default: `%Y-%m-%d %H:%M:%S`

This is the format the `%(date)` field is formatted with. Internally the std::strftime function is used to format the date field, so have a look at its documentation for the available placeholders.

File log options

The following option is only relevant for the file log sink.

`file = <file>`

Required

This option specifies the file the file log sink should output the log messages to.

Markers

Mapcrafter allows you to add different markers easily to your rendered maps. Markers are organized in marker groups, this allows you show and hide different marker groups on the rendered map.

Automatically Generated Markers

Mapcrafter is able to automatically generate markers from specific signs in your Minecraft world.

A special marker section type is used to configure automatically generated marker groups. Here is an example:

```
[marker:homes]
name = Homes
prefix = [home]
icon = home.png
icon_size = [32, 32]
```

This section defines a marker group showing different homes in your Minecraft world. Every sign that starts with the prefix `[home]` is shown on the map as marker of this marker group.

See *Marker Options* for a reference of marker section options.

To automatically generate these markers, use the `mapcrafter_markers` program with your configuration file:

```
mapcrafter_markers -c render.conf
```

This program generates your defined marker groups and writes them to a `markers-generated.js` file in your output directory. You do not need to worry about manually specified markers being overwritten.

If you have a very big world and want some progress information, use the verbose flag:

```
mapcrafter_markers -v -c render.conf
```

Manually Specifying Markers

Of course it is still possible to add markers manually to your map. You can do this by editing the `markers.js` file in your output directory (it is not overwritten by Mapcrafter if it already exists). The `markers.js` file is a Javascript file which is included by the web interface and contains definitions for the map markers.

Here is an example `markers.js` file:

```
// Add your own markers to this file.

var MAPCRAFTER_MARKERS = [
    // just one example marker group
    {
        // id of the marker group, without spaces/other special chars
        "id" : "signs",
        // name of the marker group, displayed in the webinterface
        "name" : "Signs",
        // icon of the markers belonging to that group (optional)
        "icon" : "sign.png",
        // size of that icon
        "iconSize" : [32, 32],
        // whether this marker group is shown by default (optional)
        "showDefault" : true,
```

```
        // markers of this marker group...
        "markers" : {
            // ...in the world "world"
            "world" : [
                // example marker, pretty format:
                {
                    // position ([x, z, y])
                    "pos" : [35, -21, 64],
                    // title when you hover over the marker
                    "title" : "Sign1",
                    // text in the marker popup window
                    "text" : "Hello."
                    // override the icon of a single marker (optional)
                    "icon" : "player.png",
                    // override the size of the marker icon (optional)
                    "iconSize" : [16, 32]
                },
                // more markers:
                {"pos" : [100, 100, 64], "title" : "Test1"},
                {"pos" : [100, 200, 64], "title" : "Test2"},
                {"pos" : [500, 30, 64], "title" : "Test2"},
            ],
        },
    },

    // another marker group
    {
        "id" : "homes",
        "name" : "Homes",
        "icon" : "home.png",
        "iconSize" : [32, 32],
        "markers" : {
            "world" : [
                {"pos" : [42, 73, 64], "title" : "Steve's home"},
            ],
            "world2" : [
                {"pos" : [73, 42, 64], "title" : "Steve's other home"},
            ],
        },
    },
];
```

As you can see there is a bit Javascript syntax involved here. Do not forget quotation marks around strings or the commas after array elements. The lines starting with a `//` are comments and ignored by Javascript.

The file has a Javascript-Array called `MAPCRAFTER_MARKERS` which contains the different marker groups. The elements are associative Javascript-Arrays and contain the options of the different marker groups.

These options are similar to the marker section configuration options. Every marker group has an unique ID and a name displayed in the web interface. You can also use an icon with a specific size (optional).

The actual markers are specified per world in an associative array with the name `markers`. You have to use as world name your world section name.

The definition of markers is also done with associative arrays:

```
{"pos" : [42, 73, 64], "title" : "Steve's home"},
```

Here you can see a simple marker with the title `Steve's home` and the position `42, 73, 64`. The position is

always specified as array in the form of `[x, z, y]` (x, z and then y because x and z are the horizontal axes and y is the vertical axis).

Here are the available options for the markers:

`pos`

Required

This is the position of the marker in the form of `[x, z, y]`. Example: `[12, 34, 64]`

`title`

Required

This is the title of the marker you can see when you hover over the marker.

`text`

Default: *Title of the marker*

This is the text of the marker popup window. If you do not specify a text, the title of the marker is used as text.

`icon`

Default: *Group icon*

An override for the icon for this specific marker. If you do not specify an icon, the icon set at the group level is used. Or, if there is no group-level icon, the default icon is used.

This option may be used independently of the marker icon size override.

`iconSize`

Default: *Group icon size*

An override for the size of the icon for this specific marker. If you do not specify a size, the icon size set at the group level is used. Or, if there is no group-level icon size, the default icon size is used.

This option may be used independently of the marker icon override.

Custom Leaflet Marker Objects

Furthermore you can customize your markers by specifying a function which creates the actual Leaflet marker objects with the marker data. This function is called for every marker in the marker group and should return a marker-like object displayable by Leaflet. Please have a look at the Leaflet API to find out what you can do with Leaflet:

Here is a simple example which shows two areas on the map:

```
{
    "id" : "test",
    "name" : "Test",
    "createMarker" : function(ui, groupInfo, markerInfo) {
        var latlngs = [];
        // use the ui.mcToLatLng-function to convert Minecraft coords to LatLngs
        latlngs.push(ui.mcToLatLng(markerInfo.p1[0], markerInfo.p1[1], 64));
        latlngs.push(ui.mcToLatLng(markerInfo.p2[0], markerInfo.p2[1], 64));
        latlngs.push(ui.mcToLatLng(markerInfo.p3[0], markerInfo.p3[1], 64));
        latlngs.push(ui.mcToLatLng(markerInfo.p4[0], markerInfo.p4[1], 64));
        latlngs.push(ui.mcToLatLng(markerInfo.p1[0], markerInfo.p1[1], 64));

        return L.polyline(latlngs, {"color" : markerInfo.color});
    },
```

```
    "markers" : {
        "world" : [
            {
                "p1" : [42, 0],
                "p2" : [0, 0],
                "p3" : [0, 42],
                "p4" : [42, 42],
                "color" : "red",
            },
            {
                "p1" : [73, -42],
                "p2" : [-42, -42],
                "p3" : [-42, 73],
                "p4" : [73, 73],
                "color" : "yellow",
            },
        ],
    },
},
```

As you can see you can use the `ui.mcToLatLng` method to convert Minecraft coordinates (x, z and then y) to Leaflet latitude/longitute coordinates. You can also use arbitrary data in the associative marker arrays and access them with the `markerInfo` parameter of your function (same with `groupInfo` and the fields of the marker group).

Minecraft Server

If you want player markers from your Minecraft Server on your map, please have a look at the mapcrafter-playermarkers project.

The plugin adds to your map animated markers of the players on your Minecraft Server.

CHAPTER 5

Indices and tables

- genindex
- search

Index

Symbols

C