# Overviewer Documentation

***Release 0.13***

**The Overviewer Team**

**Jun 10, 2021**

# Contents

See also the Github Homepage and the Updates Blog, and follow us on our Twitter account.

# CHAPTER 1

# Introduction

The Minecraft Overviewer is a command-line tool for rendering high-resolution maps of Minecraft Java Edition worlds. It generates a set of static html and image files and uses Leaflet to display a nice interactive map.

The Overviewer has been in active development for several years and has many features, including day and night lighting, cave rendering, mineral overlays, and many plugins for even more features! It is written mostly in Python with critical sections in C as an extension module.

For a simple example of what your renders will look like, head over to The "Exmaple" Map. For more user-contributed examples, see The Example Wiki Page.

# CHAPTER 2

# Documentation Contents

## 2.1 Installing

This page is for installing the pre-compiled binary versions of the Overviewer. If you want to build the Overviewer from source yourself, head to *Building*. If you have already built The Overviewer, proceed to *Running the Overviewer*.

The latest prebuilt packages for various systems will always be found at the Overviewer Downloads page.

### 2.1.1 Windows

Running Windows and don't want to compile the Overviewer? You've come to the right place!

1. Head to the Downloads page and download the most recent Windows download for your architecture (32 or 64 bit).
2. For 32 bit you may need to install the VC++ 2008 (x86) and VC++ 2010 (x86) redistributables.

   For 64 bit, you'll want these instead: VC++ 2008 (x64) and VC++ 2010 (x64)
3. That's it! Proceed with instructions on *Running the Overviewer*.

### 2.1.2 Debian / Ubuntu

We provide an APT repository with pre-built Overviewer packages for Debian and Ubuntu users. To do this, add the following line to your `/etc/apt/sources.list`

```
deb https://overviewer.org/debian ./
```

Note that you will need to have the `apt-transport-https` package installed for this source to work.

Our APT repository is signed. To install the key (and allow for automatic updates), run

```
wget -O - https://overviewer.org/debian/overviewer.gpg.asc | sudo apt-key add -
```

Then run `apt-get update` and `apt-get install minecraft-overviewer` and you're all set! See you at the *Running the Overviewer* page!

### 2.1.3 CentOS / RHEL / Fedora

#### Prerequisites for CentOS/RHEL 7

Enable EPEL to get a release of Python 3:

```
yum install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
```

The official instructions also recommend enabling a few additional repositories, as some EPEL packages may depend on them. **However, this is only relevant if you are not using CentOS**:

```
subscription-manager repos --enable "rhel-*-optional-rpms" --enable "rhel-*-extras-
↪rpms"  --enable "rhel-ha-for-rhel-*-server-rpms"
```

#### Installing the Overviewer

We provide a RPM repository with pre-built packages for users on RPM-based distros. To add the Overviewer repository to YUM, just run

```
wget -O /etc/yum.repos.d/overviewer.repo https://overviewer.org/rpms/overviewer.repo
```

Then to install Overviewer run

```
yum install Minecraft-Overviewer
```

After that head to the *Running the Overviewer* page!

## 2.2 Building the Overviewer from Source

These instructions are for building the C extension for Overviewer. Once you have finished with these instructions, head to *Running the Overviewer*.

**Note:** Pre-built Windows and Debian executables are available on the *Installing* page. These kits already contain the compiled code and require no further setup, so you can skip to the next section of the docs: *Running the Overviewer*.

### 2.2.1 Get The Source

First step: download the platform-independent source! Either clone with Git (recommended if you know Git) or download the most recent snapshot:

- Git URL to clone: `git://github.com/overviewer/Minecraft-Overviewer.git`
- Download most recent tar archive
- Download most recent zip archive

Once you have the source, see below for instructions on building for your system.

## 2.2.2 Build Instructions For Various Operating Systems

### Windows Build Instructions

First, you'll need a compiler. You can either use Visual Studio, or cygwin/mingw. The free Visual Studio Community is okay. You will need to select the "Desktop Development with C++" WORKLOAD. Microsoft has been changing up the names on this with the "Community" edition of Visual Studio. If nothing else works, just install every Individual Visual C++ component you can find :)

### Prerequisites

You will need the following:

- Python 3.x
- A copy of the Pillow sources.
- The Pillow Extension for Python.
- The Numpy Extension for Python.
- The extensions can be installed via:

```
c:\python37\python.exe -m pip -U numpy pillow
```

### Building with Visual Studio

1. Get the latest Overviewer source code as per above.
2. From the Start menu, navigate to 'Visual Studio 2017' and open the **'Developer Command Prompt for VS 2017'** (*or whatever year*) shortcut. A regular command or powershell prompt will *NOT* work for this.
3. cd to the folder containing the Overviewer source code.
4. Copy Imaging.h and ImPlatform.h from your Pillow sources into the current working directory.
5. First try a build:

```
c:\python37\python setup.py build
```

If you encounter the following errors:

```
error: Unable to find vcvarsall.bat
```

then try the following:

```
set DISTUTILS_USE_SDK=1
set MSSdk=1
c:\python37\python setup.py build
```

If the build was successful, there should be a c_overviewer.pyd file in your current working directory.

### Building with mingw-w64 and msys2

This is the recommended way to build on Windows without MSVC.

1. Install msys2 by following **all** the instructions on the msys2 installation page.
2. Install the dependencies:

   ```
   pacman -S git mingw-w64-x86_64-python3-numpy mingw-w64-x86_64-python3-Pillow␣
   ↪mingw-w64-x86_64-python3 mingw-w64-x86_64-toolchain
   ```

3. Clone the Minecraft-Overviewer git repository:

   ```
   git clone https://github.com/overviewer/Minecraft-Overviewer.git
   ```

   The source code will be downloaded to your msys2 home directory, e.g. `C:\msys2\home\Potato\Minecraft-Overviewer`

4. Close the msys2 shell. Instead, open the MinGW64 shell.
5. Build the Overviewer by changing your current working directory into the source directory and executing the build script:

   ```
   cd Minecraft-Overviewer
   python3 setup.py build
   ```

After it finishes, you should now be able to execute `overviewer.py` from the MINGW64 shell.

### Building with mingw

1. Open a MinGW shell.
2. cd to the Overviewer directory.
3. Copy Imaging.h and ImPlatform.h from your Pillow sources into the current working directory.
4. Build:

   ```
   python3 setup.py build --compiler=mingw32
   ```

If the build fails with complaints about `-mno-cygwin`, open the file `Lib/distutils/cygwincompiler.py` in an editor of your choice, and remove all mentions of `-mno-cygwin`. This is a bug in distutils, filed as Issue 12641.

### Linux

You will need the gcc compiler and a working build environment. On Ubuntu and Debian, this can be done by installing the `build-essential` package.

You will need the following packages on Debian-derived distributions (e.g. Ubuntu):

- python3-pil or python3-pillow (the latter is usually aliased to the former)
- python3-dev
- python3-numpy

**Note:** If you choose to install pillow through pip instead of your distribution's package manager, you won't get the pillow headers which Overviewer requires to build its C extension. In that case, you should manually download the header files specific to the version of pillow you installed, and point at them with the `PIL_INCLUDE_DIR` environment variable. A version mismatch between the installed pillow library and the headers can lead to segfaults while running Overviewer due to an ABI mismatch.

Then to build:

```
python3 setup.py build
```

At this point, you can run `./overviewer.py` from the current directory, so to run it you'll have to be in this directory and run `./overviewer.py` or provide the the full path to `overviewer.py`. Another option would be to add this directory to your `$PATH`. Note that there is a `python3 setup.py install` step that you can run which will install things into `/usr/local/bin`, but this is strongly not recommended as it might conflict with other installs of Overviewer.

### macOS

1. Install the Xcode Command Line Tools by running the following command in a terminal (located in your /Applications/Utilities folder):

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

2. Install Python 3 if you don't already have it, for example from the official Python website.
3. Install PIP, e.g. with:

   ```
   sudo easy_install pip
   ```

4. Install Pillow (overviewer needs PIL, Pillow is a fork of PIL that provides the same functionality):

   ```
   pip install Pillow
   ```

5. Download the Pillow source files from https://github.com/python-pillow/Pillow/releases/latest and unpack the tar.gz file and move it to a directory you can remember
6. Download the Minercaft Overviewer source-code from https://overviewer.org/builds/overviewer-latest.tar.gz
7. Extract overviewer-[Version].tar.gz and move it to a directory you can remember
8. Go into your Pillow-[Version] folder and navigate to the /src/libImaging directory
9. Drag the following files from the Pillow-[Version]/src/libImaging folder to your overviewer-[Version] folder:

- `Imaging.h`

- `ImagingUtils.h`
- `ImPlatform.h`

1. Make sure your installation of Python 3 is in `$PATH`
2. In a terminal, change your current working directory to your overviewer-[Version] folder (e.g. by using `cd Desktop/overviewer-[Version]`)
3. Build:

   ```
   python3 setup.py build
   ```

You should now be able to run Overviewer with `./overviewer.py` inside of the Overviewer directory.

## 2.3 Running the Overviewer

### 2.3.1 Rendering your First Map

Overviewer is a command-line application, and so it needs to be run from the command line. If you installed Overviewer from a package manager, the command is `overviewer.py`. If you downloaded it manually, open a terminal window and navigate to wherever you downloaded Overviewer. For pre-compiled Windows builds, the command is `overviewer.exe`. For other systems, it's `overviewer.py`.

What follows in this section is a few examples to get you started. For full usage, see the *Usage* section.

So, let's render your first map! Let's say you want to render your single player world called "My World". Let's also say you want to save it c:mcmap. You would type into your command prompt the following:

```
overviewer.exe "My World" c:\mcmap
```

If you're on Linux or a Mac, you could do something like one of the following:

```
overviewer.py "My World" /home/username/mcmap
```

or

```
overviewer.py "My World" /Users/username/mcmap
```

Those will look for a single player world by that name. You can also specify the path to the world you want to render. This is useful for rendering servers.

Let's say you have a server installed in /home/username/mcserver. This command will render the default dimension (in the case of Bukkit multiworld servers, the default world is used. You can also specify the directory to the specific world you want to render).

```
overviewer.py /home/username/mcserver /home/username/mcmap
```

After you enter one of the commands, The Overviewer should start rendering your map. When the render is done, open up *index.html* using your web-browser of choice. Pretty cool, huh? You can even upload this map to a web server to share with others! Simply upload the entire folder to a web server and point your users to index.html!

Incremental updates are just as easy, and a lot faster. If you go and change something inside your world, run the command again and The Overviewer will automatically re-render only what's needed.

### Specifying a different rendermode

There are a few built-in rendermodes for you to choose from. Each will render your map differently. For example, if you want smooth lighting (which looks really good), you would add `--rendermodes=smooth-lighting` to your command. e.g.

```
overviewer.py --rendermodes=smooth-lighting /home/username/mcserver /home/username/
↪mcmap
```

The rendermodes you have to choose from are:

- normal (the default)
- lighting
- smooth-lighting
- cave

You can specify more than one. Just separate them with a comma!

## 2.3.2 Usage

For this section, we assume the executable is `overviewer.py`. Replace that with `overviewer.exe` for windows.

Overviewer usage:

```
overviewer.py [--rendermodes=...] [options] <World> <Output Dir>
overviewer.py --config=<config file> [options]
```

The first form is for basic or quick renderings without having to create a config file. It is intentionally limited because the amount of configuration was becoming unmanageable for the command line.

The second, preferred usage involves creating a configuration file which specifies all the options including what to render, where to place the output, and all the settings. See *The Configuration File* for details on that.

For example, on Windows if your Minecraft server runs out of `c:\server\` and you want to put the rendered map in `c:\mcmap\`, run this:

```
overviewer.exe c:\server\world c:\mcmap
```

For Mac or Linux builds from source, you would run something like this with the current directory in the top level of the source tree:

```
./overviewer.py /opt/minecraft/server/world /opt/minecraft/mcmap
```

The first render can take a while, depending on the size of your world.

### Options

The following options change the way The Overviewer generates or updates the map, and are intended to be things you only have to use in special situations. You should not normally have to specify these options; the default is typically correct.

**`--no-tile-checks`**

With this option, The Overviewer will determine which tiles to render by looking at the saved last-render timestamp and comparing it to the last-modified time of the chunks of the world. It builds a tree of tiles that need updating and renders only those tiles.

This option does not do *any* checking of tile mtimes on disk, and thus is the cheapest option: only rendering what needs updating while minimising disk IO.

The caveat is that the *only* thing to trigger a tile update is if Minecraft updates a chunk. Any other reason a tile may have for needing re-rendering is not detected. This means that changes in your render configuration will not be reflected in your world except in updated chunks. It could also cause problems if the system clock of the machine running Minecraft is not stable.

**This option is the default** unless *--forcerender* or *--check-tiles* is in effect. This option conflicts with *--forcerender* and *--check-tiles*.

**--check-tiles**

Forces The Overviewer to check each tile on disk and check to make sure it is up to date. This also checks for tiles that shouldn't exist and deletes them.

This is functionally equivalent to *--no-tile-checks* with the difference that each tile is individually checked. It is therefore useful if the tiles are not consistent with the last-render timestamp that is automatically stored. This option was designed to handle the case where the last render was interrupted – some tiles have been updated but others haven't, so each one is checked before it is rendered.

This is slightly slower than *--no-tile-checks* due to the additional disk-io involved in reading tile mtimes from the filesystem

Since this option also checks for erroneous tiles, **It is also useful after you delete sections of your map, e.g. with worldedit, to delete tiles that should no longer exist.** Overviewer greatly overestimates tiles to be rendered and time needed to complete.

The caveats with this option are the same as for *--no-tile-checks* with the additional caveat that tile timestamps in the filesystem must be preserved. If you copy tiles or make changes to them with an external tool that modifies mtimes of tiles, it could cause problems with this option.

This option is automatically activated when The Overviewer detects the last render was interrupted midway through. This option conflicts with *--forcerender* and *--no-tile-checks*

**--forcerender**

Forces The Overviewer to re-render every tile regardless of whether it thinks it needs updating or not. It does no tile mtime checks, and therefore ignores the last render time of the world, the last modification times of each chunk, and the filesystem mtimes of each tile. It unconditionally renders every tile that exists.

The caveat with this option is that it does *no* checks, period. Meaning it will not detect tiles that do exist, but shouldn't (this can happen if your world shrinks for some reason. For that specific case, *--check-tiles* is actually the appropriate mode).

This option is useful if you have changed a render setting and wish to re-render every tile with the new settings.

This option is automatically activated for first-time renders. This option conflicts with *--check-tiles* and *--no-tile-checks*

**--genpoi**

**Note:** Don't use this flag without first reading *Signs and Markers*!

Generates the POI markers for your map. This option does not do any tile/map generation, and ONLY generates markers. See *Signs and Markers* on how to configure POI options.

**-p** <procs>, **--processes** <procs>

This specifies the number of worker processes to spawn on the local machine to do work. It defaults to the number of CPU cores you have, if not specified.

This option can also be specified in the config file as *processes*

**--skip-scan**

**Note:** Don't use this flag without first reading *Signs and Markers*!

When generating POI markers, this option prevents scanning for entities and tile entities, and only creates the markers specified in the config file. This considerably speeds up the POI marker generation process if no entities or tile entities are being used for POI markers. See *Signs and Markers* on how to configure POI options.

**-v, --verbose**
Activate a more verbose logging format and turn on debugging output. This can be quite noisy but also gives a lot more info on what The Overviewer is doing.

**-q, --quiet**
Turns off one level of logging for quieter output. You can specify this more than once. One `-q` will suppress all INFO lines. Two will suppress all INFO and WARNING lines. And so on for ERROR and CRITICAL log messages.

If *`--verbose`* is given, then the first `-q` will counteract the DEBUG lines, but not the more verbose logging format. Thus, you can specify `-v -q` to get only INFO logs and higher (no DEBUG) but with the more verbose logging format.

**--update-web-assets**
Update web assets, including custom assets, without starting a render. This won't update overviewerConfig.js, but will recreate overviewer.js

### 2.3.3 Installing the Textures

**Note:** This procedure has changed with Minecraft 1.6's Resource Pack update. The latest versions of Overviewer are not compatible with Minecraft 1.5 client resources.

If Overviewer is running on a machine with the Minecraft client installed, it will automatically use the default textures from Minecraft.

**Note:** Overviewer will only search for installed client *release* versions, not snapshots. If you want to use a snapshot client jar for the textures, you must specify it manually with the *texturepath* option.

If, however, you're running on a machine without the Minecraft client installed, or if you want to use different textures, you will need to provide the textures manually. This is common for servers.

If you want or need to provide your own textures, you have several options:

- The easy solution is to download the latest client jar to the location the launcher would normally install it. Overviewer will find it and use it.

  You can use the following commands to download the client jar on Linux or Mac. Run the first line in a terminal, changing the version string to the latest as appropriate (these docs may not always be updated to reflect the latest). Then paste the second line into your terminal to create directories if necessary. Then paste the third line into your terminal to download the latest version. `${VERSION}` will be replaced by the actual version string from the first line. These 3 lines can be included in a shell script prior to map generation to ensure the proper textures are always downloaded.

```
VERSION=1.16.1
mkdir -p ~/.minecraft/versions/${VERSION}/
wget https://overviewer.org/textures/${VERSION} -O ~/.minecraft/versions/$
↪{VERSION}/${VERSION}.jar
```

If that's too confusing for you, then just take this single line and paste it into a terminal to get 1.16 textures:

```
wget https://overviewer.org/textures/1.16.1 -O ~/.minecraft/versions/1.16.1/1.16.
↪1.jar
```

- You can also just run the launcher to install the client.
- You can transfer the client jar to the correct place manually, from a computer that does have the client, to your server. The correct places are:
  - For Linux: `~/.minecraft/versions/<version>/<version>.jar`
  - For Mac: `~/Library/Application Support/minecraft/versions/<version>/<version>.jar`
  - For Windows: `%APPDATA%/.minecraft/versions/<version>/<version>.jar`
- You can download and use a custom resource pack. Download the resource pack file and specify the path to it with the *texturepath* option.

### If you copy your world before you render it

The important thing to be careful about when copying world files to another location is file modification times, which Overviewer uses to figure out what parts of the map need updating. If you do a straight copy, usually this will update the modification times on all the copied files, causing Overviewer to re-render the entire map. To copy files on Unix, while keeping these modification times intact, use `cp -p`. For people who render from backups, GNU `tar` automatically handles modification times correctly. `rsync -a --delete` will handle this correctly as well. If you use some other tool, you'll have to figure out how to do this yourself.

### HTTPS support

In order to support displaying maps over HTTPS, Overviewer loads the Google maps API and JQuery over HTTPS. This avoids security warnings for HTTPS sites, and is not expected to cause problems for users.

If this change causes problems, take a look at the *custom web assets* option. This allows you to provide a custom index.html which loads the required Javascript libraries over HTTP.

## 2.4 The Configuration File

Using a configuration file is now the preferred way of running The Overviewer. You will need to create a blank file and specify it when running The Overviewer like this:

```
overviewer.py --config=path/to/my_configfile
```

The config file is formatted in Python syntax. If you aren't familiar with Python, don't worry, it's pretty simple. Just follow the examples.

**Note:** You should *always* use forward slashes ("/"), even on Windows. This is required because the backslash ("") has special meaning in Python.

### 2.4.1 Examples

The following examples should give you an idea of what a configuration file looks like, and also teach you some neat tricks.

#### A Simple Example

```
worlds["My world"] = "/home/username/server/world"

renders["normalrender"] = {
    "world": "My world",
    "title": "Normal Render of My World",
}

outputdir = "/home/username/mcmap"
```

This defines a single world, and a single render of that world. You can see there are two main sections.

**The `worlds` dictionary** Define items in the `worlds` dictionary as shown to tell The Overviewer where to find your worlds. The keys to this dictionary ("My world" in the example) is a name you give, and is referenced later in the render dictionary. If you want to render more than one world, you would put more lines like this one. Otherwise, one is sufficient.

**The `renders` dictionary** Each item here declares a "render" which is a map of one dimension of one world rendered with the given options. If you declare more than one render, then you will get a dropdown box to choose which map you want to look at when viewing the maps.

You are free to declare as many renders as you want with whatever options you want. For example, you are allowed to render multiple worlds, or even render the same world multiple times with different options.

**Note:** Since this is Python syntax, keep in mind you need to put quotation marks around your strings. `worlds[My world]` will not work. It must be `worlds["My world"]`

#### A more complicated example

```
worlds["survival"] = "/home/username/server/survivalworld"
worlds["creative"] = "/home/username/server/creativeworld"

renders["survivalday"] = {
    "world": "survival",
    "title": "Survival Daytime",
    "rendermode": smooth_lighting,
    "dimension": "overworld",
}

renders["survivalnight"] = {
    "world": "survival",
```

(continues on next page)

(continued from previous page)

```
    "title": "Survival Nighttime",
    "rendermode": smooth_night,
    "dimension": "overworld",
}

renders["survivalnether"] = {
    "world": "survival",
    "title": "Survival Nether",
    "rendermode": nether_smooth_lighting,
    "dimension": "nether",
}

renders["survivalnethersouth"] = {
    "world": "survival",
    "title": "Survival Nether",
    "rendermode": nether_smooth_lighting,
    "dimension": "nether",
    "northdirection" : "lower-right",
}

renders["creative"] = {
    "world": "creative",
    "title": "Creative",
    "rendermode": smooth_lighting,
    "dimension": "overworld",
}

outputdir = "/home/username/mcmap"
texturepath = "/home/username/my_texture_pack.zip"
```

This config defines four maps for render. Two of them are of the survival world's overworld, one is for the survival's nether, and one is for the creative world.

Notice here we explicitly set the dimension property on each render. If dimension is not specified, the default or overworld dimension is used. It is necessary e.g. for the nether render.

Also note here we specify some different rendermodes. A rendermode refers to how the map is rendered. The Overviewer can render a map in many different ways, and there are many preset rendermodes, and you can even create your own (more on that later).

And finally, note the usage of the `texturepath` option. This specifies a texture pack (also called a resource pack) to use for the rendering. Also note that it is set at the top level of the config file, and therefore applies to every render. It could be set on individual renders to apply to just those renders.

**Note:** See the `sample_config.py` file included in the repository for another example.

### A dynamic config file

It might be handy to dynamically retrieve parameters. For instance, if you periodically render your last map backup which is located in a timestamped directory, it is not convenient to edit the config file each time to fit the new directory name.

Using environment variables, you can easily retrieve a parameter which has been set by, for instance, your map backup script. In this example, Overviewer is called from a *bash* script, but it can be done from other shell scripts and languages.

```
#!/bin/bash

## Add these lines to your bash script

# Setting up an environment variable that child processes will inherit.
# In this example, the map's path is not static and depends on the
# previously set $timestamp var.
MYWORLD_DIR=/path/to/map/backup/$timestamp/YourWorld
export MYWORLD_DIR

# Running the Overviewer
overviewer.py --config=/path/to/yourConfig.py
```

**Note:** The environment variable will only be local to the process and its child processes. The Overviewer, when run by the script, will be able to access the variable since it becomes a child process.

```
## A config file example

# Importing the os python module
import os

# Retrieving the environment variable set up by the bash script
worlds["My world"] = os.environ['MYWORLD_DIR']

renders["normalrender"] = {
    "world": "My world",
    "title": "Normal Render of My World",
}

outputdir = "/home/username/mcmap"
```

### 2.4.2 Config File Specifications

The config file is a python file and is parsed with python's execfile() builtin. This means you can put arbitrary logic in this file. The Overviewer gives the execution of the file a local dict with a few pre-defined items (everything in the overviewer_core.rendermodes module).

If the above doesn't make sense, just know that items in the config file take the form `key = value`. Two items take a different form:, `worlds` and `renders`, which are described below.

#### General

**`worlds`** This is pre-defined as an empty dictionary. The config file is expected to add at least one item to it.

Keys are arbitrary strings used to identify the worlds in the `renders` dictionary.

Values are paths to worlds (directories with a level.dat)

e.g.:

```
worlds['myworld'] = "/path/to/myworld"
```

**You must specify at least one world**

*Reminder*: Always use forward slashes ("/"), even on Windows.

**renders** This is also pre-defined as an empty dictionary. The config file is expected to add at least one item to it. By default, it is an ordered dictionary; the order you add entries to it will determine the default render in the output map and the order the buttons appear in the map UI.

Keys are strings that are used as the identifier for this render in the javascript, and also as the directory name for the tiles, but it's essentially up to you. It thus is recommended to make it a string with no spaces or special characters, only alphanumeric characters.

Values are dictionaries specifying the configuration for the render. Each of these render dictionaries maps strings naming configuration options to their values. Valid keys and their values are listed in the *Render Dictionary Keys* section.

e.g.:

```
renders['myrender'] = {
        'world': 'myworld',
        'title': 'Minecraft Server Title',
        }
```

**You must specify at least one render**

**outputdir = "<output directory path>"** This is the path to the output directory where the rendered tiles will be saved.

e.g.:

```
outputdir = "/path/to/output"
```

*Reminder*: Always use forward slashes ("/"), even on Windows.

**Required**

**processes = num_procs** This specifies the number of worker processes to spawn on the local machine to do work. It defaults to the number of CPU cores you have, if not specified.

This can also be specified with `--processes`

e.g.:

```
processes = 2
```

## Observers

**observer = <observer object>** This lets you configure how the progress of the render is reported. The default is to display a progress bar, unless run on Windows or with stderr redirected to a file. The default value will probably be fine for most people, but advanced users may want to make their own progress reporter (for a web service or something like that) or you may want to force a particular observer to be used. The observer object is expected to have at least `start`, `add`, `update`, and `finish` methods.

If you want to specify an observer manually, try something like:

```
from .observer import ProgressBarObserver
observer = ProgressBarObserver()
```

There are currently three observers available: `LoggingObserver`, `ProgressBarObserver` and `JSObserver`.

**LoggingObserver** This gives the normal/older style output and is the default when output is redirected to a file or when running on Windows

**ProgressBarObserver** This is used by default when the output is a terminal. Displays a text based progress bar and some statistics.

**JSObserver(outputdir[, minrefresh][, messages])** This will display render progress on the output map in the bottom right corner of the screen. `JSObserver`.

- **outputdir="<output directory path"** Path to overviewer output directory. For simplicity, specify this as `outputdir=outputdir` and place this line after setting `outputdir = "<output directory path>"`.

  **Required**

- **minrefresh=<seconds>** Progress information won't be written to file or requested by your web browser more frequently than this interval.

- **messages=dict(totalTiles=<string>, renderCompleted=<string>, renderProgress=<str**
  Customises messages displayed in browser. All three messages must be defined similar to the following:

  – `totalTiles="Rendering %d tiles"` The `%d` format string will be replaced with the total number of tiles to be rendered.

  – `renderCompleted="Render completed in %02d:%02d:%02d"` The three format strings will be replaced with the number of hours. minutes and seconds taken to complete this render.

  – `renderProgress="Rendered %d of %d tiles (%d%% ETA:%s)""` The four format strings will be replaced with the number of tiles completed, the total number of tiles, the percentage complete, and the ETA.

  Format strings are explained here: http://docs.python.org/library/stdtypes.html#string-formatting All format strings must be present in your custom messages.

```
from .observer import JSObserver
observer = JSObserver(outputdir, 10)
```

**MultiplexingObserver(Observer[, Observer[, Observer ...]])** This observer will send the progress information to all Observers passed to it.

- All Observers passed must implement the full Observer interface.

```
## An example that updates both a LoggingObserver and a JSObserver
# Import the Observers
from .observer import MultiplexingObserver, LoggingObserver, JSObserver

# Construct the LoggingObserver
loggingObserver = LoggingObserver()

# Construct a basic JSObserver
jsObserver = JSObserver(outputdir) # This assumes you have set the outputdir␣
↪previous to this line

# Set the observer to a MultiplexingObserver
observer = MultiplexingObserver(loggingObserver, jsObserver)
```

**ServerAnnounceObserver(target, pct_interval)** This Observer will send its progress and status to a Minecraft server via `target` with a Minecraft `say` command.

- **target=<file handle to write to>** Either a FIFO file or stdin. Progress and status messages will be written to this handle.

  **Required**

- **`pct_interval=<update rate, in percent>`** Progress and status messages will not be written more often than this value. E.g., a value of `1` will make the ServerAnnounceObserver write to its target once for every 1% of progress.

  **Required**

**`RConObserver(target, password[, port][, pct_interval])`** This Observer will announce render progress with the server's `say` command through RCon.

- **`target=<address>`** Address of the target Minecraft server.

  **Required**

- **`password=<rcon password>`** The server's rcon password.

  **Required**

- **`port=<port number>`** Port on which the Minecraft server listens for incoming RCon connections.

  **Default:** `25575`

- **`pct_interval=<update rate, in percent>`** Percentage interval in which the progress should be announced, the same as for `ServerAnnounceObserver`.

  **Default:** `10`

### Custom web assets

**`customwebassets = "<path to custom web assets>"`** This option allows you to specify a directory containing custom web assets to be copied to the output directory. Any files in the custom web assets directory overwrite the default files.

If you are providing a custom index.html, the following strings will be replaced:

- `{title}` Will be replaced by 'Minecraft Overviewer'
- `{time}` Will be replaced by the current date and time when the world is rendered e.g. 'Sun, 12 Aug 2012 15:25:40 BST'
- `{version}` Will be replaced by the version of Overviewer used e.g. '0.9.276 (5ff9c50)'

### Render Dictionary Keys

The render dictionary is a dictionary mapping configuration key strings to values. The valid configuration keys are listed below.

**Note:** Any of these items can be specified at the top level of the config file to set the default for every render. For example, this line at the top of the config file will set the world for every render to 'myworld' if no world is specified:

```
world = 'myworld'
```

Then you don't need to specify a `world` key in the render dictionaries:

```
renders['arender'] = {
        'title': 'This render doesn't explicitly declare a world!',
        }
```

### General

**`world`** Specifies which world this render corresponds to. Its value should be a string from the appropriate key in the worlds dictionary.

**Required**

**`title`** This is the display name used in the user interface. Set this to whatever you want to see displayed in the Map Type control (the buttons in the upper- right).

**Required**

**`dimension`** Specified which dimension of the world should be rendered. Each Minecraft world has by default 3 dimensions: The Overworld, The Nether, and The End. Bukkit servers are a bit more complicated, typically worlds only have a single dimension, in which case you can leave this option off.

The value should be a string. It should either be one of "overworld", "nether", "end", or the directory name of the dimension within the world. e.g. "DIM-1"

**Note:** If you choose to render your nether dimension, you must also use a nether *rendermode*. Otherwise you'll just end up rendering the nether's ceiling.

**Note:** For the end, you will most likely want to turn down the strength of the shadows, as you'd otherwise end up with a very dark result.

e.g.:

```
end_lighting = [Base(), EdgeLines(), Lighting(strength=0.5)]
end_smooth_lighting = [Base(), EdgeLines(), SmoothLighting(strength=0.5)]
```

**Default:** `"overworld"`

### Rendering

**`rendermode`** This is which rendermode to use for this render. There are many rendermodes to choose from. This can either be a rendermode object, or a string, in which case the rendermode object by that name is used.

e.g.:

```
"rendermode": "normal",
```

Here are the rendermodes and what they do:

**`"normal"`** A normal render with no lighting. This is the fastest option.

**`"lighting"`** A render with per-block lighting, which looks similar to Minecraft without smooth lighting turned on. This is slightly slower than the normal mode.

**`"smooth_lighting"`** A render with smooth lighting, which looks similar to Minecraft with smooth lighting turned on.

*This option looks the best* but is also the slowest.

**`"night"`** A "nighttime" render with blocky lighting.

**`"smooth_night"`** A "nighttime" render with smooth lighting

**"nether"** A normal lighting render of the nether. You can apply this to any render, not just nether dimensions. The only difference between this and normal is that the ceiling is stripped off, so you can actually see inside.

> **Note:** Selecting this rendermode doesn't automatically render your nether dimension. Be sure to also set the *dimension* option to 'nether'.

**"nether_lighting"** Similar to "nether" but with blocky lighting.

**"nether_smooth_lighting"** Similar to "nether" but with smooth lighting.

**"cave"** A cave render with depth tinting (blocks are tinted with a color dependent on their depth, so it's easier to tell overlapping caves apart)

**Default:** `"normal"`

> **Note:** The value for the 'rendermode' key can be either a *string* or *rendermode object* (strings simply name one of the built-in rendermode objects). The actual object type is a list of *rendermode primitive* objects. See *Custom Rendermodes and Rendermode Primitives* for more information.

**northdirection** This is direction or viewpoint angle with which north will be rendered. This north direction will match the established north direction in the game where the sun rises in the east and sets in the west.

Here are the valid north directions:

- `"upper-left"`
- `"upper-right"`
- `"lower-left"`
- `"lower-right"`

**Default:** `"upper-left"`

**overlay** This specifies which renders that this render will be displayed on top of. It should be a list of other renders. If this option is confusing, think of this option's name as "overlay_on_to".

If you leave this as an empty list, this overlay will be displayed on top of all renders for the same world/dimension as this one.

As an example, let's assume you have two renders, one called "day" and one called "night". You want to create a Biome Overlay to be displayed on top of the "day" render. Your config file might look like this:

> **Note:** When 'overlay' is used the `imgformat` must be set to a transparent image format like `"png"`. Otherwise the overlay is rendered without transparency and the render underneath will not show.

```
outputdir = "output_dir"


worlds["exmaple"] = "exmaple"

renders['day'] = {
    'world': 'exmaple',
    'rendermode': 'smooth_lighting',
    'title': "Daytime Render",
}
```

(continues on next page)

(continued from previous page)

```
renders['night'] = {
    'world': 'exmaple',
    'rendermode': 'night',
    'title': "Night Render",
}

renders['biomeover'] = {
    'world': 'exmaple',
    'rendermode': [ClearBase(), BiomeOverlay()],
    'title': "Biome Coloring Overlay",
    'overlay': ['day']
}
```

**Default:** `[]` (an empty list)

**`texturepath`** This is a where a specific texture or resource pack can be found to use during this render. It can be a path to either a folder or a zip/jar file containing the texture resources. If specifying a folder, this option should point to a directory that *contains* the assets/ directory (it should not point to the assets directory directly or any one particular texture image).

Its value should be a string: the path on the filesystem to the resource pack.

**`crop`** You can use this to render one or more small subsets of your map. The format of an individual crop zone is (min x, min z, max x, max z); if you wish to specify multiple crop zones, you may do so by specifying a list of crop zones, i.e. [(min x1, min z1, max x1, max z1), (min x2, min z2, max x2, max z2)]

The coordinates are block coordinates. The same you get with the debug menu in-game and the coordinates shown when you view a map.

Example that only renders a 1000 by 1000 square of land about the origin:

```
renders['myrender'] = {
        'world': 'myworld',
        'title': "Cropped Example",
        'crop': (-500, -500, 500, 500),
}
```

Example that renders two 500 by 500 squares of land:

```
renders['myrender'] = {
        'world': 'myworld',
        'title': "Multi cropped Example",
        'crop': [(-500, -500, 0, 0), (0, 0, 500, 500)]
}
```

This option performs a similar function to the old `--regionlist` option (which no longer exists). It is useful for example if someone has wandered really far off and made your map too large. You can set the crop for the largest map you want to render (perhaps `(-10000,-10000,10000,10000)`). It could also be used to define a really small render showing off one particular feature, perhaps from multiple angles.

> **Warning:** If you decide to change the bounds on a render, you may find it produces unexpected results. It is recommended to not change the crop settings once it has been rendered once.
>
> For an expansion to the bounds, because chunks in the new bounds have the same mtime as the old, tiles will not automatically be updated, leaving strange artifacts along the old border. You may need to use *`--forcerender`* to force those tiles to update. (You can use the `forcerender` option on just one render by adding `'forcerender': True` to that render's configuration)

For reductions to the bounds, you will need to render your map at least once with the `--check-tiles` mode activated, and then once with the `--forcerender` option. The first run will go and delete tiles that should no longer exist, while the second will render the tiles around the edge properly. Also see *this faq entry*.

Sorry there's no better way to handle these cases at the moment. It's a tricky problem and nobody has devoted the effort to solve it yet.

## Image options

**imgformat** This is which image format to render the tiles into. Its value should be a string containing "png", "jpg", "jpeg" or "webp".

**Note:** For WebP, your PIL/Pillow needs to be built with WebP support. Do keep in mind that not all browsers support WebP images.

**Default:** `"png"`

**imgquality** This is the image quality used when saving the tiles into the JPEG or WebP image format. Its value should be an integer between 0 and 100.

For WebP images in lossless mode, it determines how much effort is spent on compressing the image.

**Default:** `95`

**imglossless** Determines whether a WebP image is saved in lossless or lossy mode. Has no effect on other image formats.

**Default:** `True`

`optimizeimg`

**Warning:** Using image optimizers will increase render times significantly.

**Note:** With the port to Python 3, the import line has changed. Prefix the `optimizeimages` module with a period, so `from .optimizeimages import foo, bar`.

This option specifies which additional tools overviewer should use to optimize the filesize of rendered tiles. The tools used must be placed somewhere where overviewer can find them, for example the "PATH" environment variable or a directory like /usr/bin.

The option is a list of Optimizer objects, which are then executed in the order in which they're specified:

```
# Import the optimizers we need
from .optimizeimages import pngnq, optipng

worlds["world"] = "/path/to/world"

renders["daytime"] = {
    "world":"world",
    "title":"day",
    "rendermode":smooth_lighting,
```

(continues on next page)

(continued from previous page)

```
    "optimizeimg":[pngnq(sampling=1), optipng(olevel=3)],
}
```

**Note:** Don't forget to import the optimizers you use in your config file, as shown in the example above.

Here is a list of supported image optimization programs:

**pngnq** pngnq quantizes 32-bit RGBA images into 8-bit RGBA palette PNGs. This is lossy, but reduces filesize significantly. Available settings:

**sampling** An integer between `1` and `10`, `1` samples all pixels, is slow and yields the best quality. Higher values sample less of the image, which makes the process faster, but less accurate.

**Default:** `3`

**dither** Either the string `"n"` for no dithering, or `"f"` for Floyd Steinberg dithering. Dithering helps eliminate colorbanding, sometimes increasing visual quality.

**Warning:** With pngnq version 1.0 (which is what Ubuntu 12.04 ships), the dithering option is broken. Only the default, no dithering, can be specified on those systems.

**Default:** `"n"`

**Warning:** Because of several PIL bugs, only the most zoomed in level has transparency when using pngnq. The other zoom levels have all transparency replaced by black. This is *not* pngnq's fault, as pngnq supports multiple levels of transparency just fine, it's PIL's fault for not even reading indexed PNGs correctly.

**optipng** optipng tunes the deflate algorithm and removes unneeded channels from the PNG, producing a smaller, lossless output image. It was inspired by pngcrush. Available settings:

**olevel** An integer between `0` (few optimizations) and `7` (many optimizations). The default should be satisfactory for everyone, higher levels than the default see almost no benefit.

**Default:** `2`

**oxipng** An optipng replacement written in Rust. Works much like optipng.

**olevel** An integer between `0` (few optimizations) and `6` (many optimizations). The default should be satisfactory for everyone, higher levels than the default see almost no benefit.

**Default:** `2`

**threads** An integer specifying how many threads per process to use. Note that Overviewer spawns one oxipng process per Overviewer worker process, so increasing this value if you already have one Overviewer process per CPU thread makes little sense, and can actually slow down the rendering.

**Default:** `1`

**pngcrush** pngcrush, like optipng, is a lossless PNG recompressor. If you are able to do so, it is recommended to use optipng instead, as it generally yields better results in less time. Available settings:

**brute** Either `True` or `False`. Cycles through all compression methods, and is very slow.

**Note:** There is practically no reason to ever use this. optipng will beat pngcrush, and throwing more CPU time at pngcrush most likely won't help. If you think you need this option, then you are most likely wrong.

**Default:** `False`

**`jpegoptim`** jpegoptim can do both lossy and lossless JPEG optimisation. If no options are specified, jpegoptim will only do lossless optimisations. Available settings:

**`quality`** A number between 0 and 100 that corresponds to the jpeg quality level. If the input image has a lower quality specified than the output image, jpegoptim will only do lossless optimisations.

If this option is specified and the above condition does not apply, jpegoptim will do lossy optimisation.

**Default:** `None` *(= Unspecified)*

**`target_size`** Either a percentage of the original filesize (e.g. `"50%"`) or a target filesize in kilobytes (e.g. `15`). jpegoptim will then try to reach this as its target size.

If specified, jpegoptim will do lossy optimisation.

**Warning:** This appears to have a greater performance impact than just setting `quality`. Unless predictable filesizes are a thing you need, you should probably use `quality` instead.

**Default:** `None` *(= Unspecified)*

**Default:** `[]`

## Zoom

These options control the zooming behavior in the JavaScript output.

**`defaultzoom`** This value specifies the default zoom level that the map will be opened with. It has to be greater than 0, which corresponds to the most zoomed-out level. If you use `minzoom` or `maxzoom`, it should be between those two.

**Default:** `1`

**`maxzoom`** This specifies the maximum, closest in zoom allowed by the zoom control on the web page. This is relative to 0, the farthest-out image, so setting this to 8 will allow you to zoom in at most 8 times. This is *not* relative to `minzoom`, so setting `minzoom` will shave off even more levels. If you wish to specify how many zoom levels to leave off, instead of how many total to use, use a negative number here. For example, setting this to -2 will disable the two most zoomed-in levels.

**Note:** This does not change the number of zoom levels rendered, but allows you to neglect uploading the larger and more detailed zoom levels if bandwidth usage is an issue.

**Default:** Automatically set to most detailed zoom level

**`minzoom`** This specifies the minimum, farthest away zoom allowed by the zoom control on the web page. For example, setting this to 2 will disable the two most zoomed-out levels.

**Note:** This does not change the number of zoom levels rendered, but allows you to have control over the number of zoom levels accessible via the slider control.

**Default:** 0 (zero, which does not disable any zoom levels)

### Other HTML/JS output options

**`showlocationmarker`** Allows you to specify whether to show the location marker when accessing a URL with coordinates specified.

**Default:** `True`

**`base`** Allows you to specify a remote location for the tile folder, useful if you rsync your map's images to a remote server. Leave a trailing slash and point to the location that contains the tile folders for each render, not the tiles folder itself. For example, if the tile images start at http://domain.com/map/world_day/ you want to set this to http://domain.com/map/

**`markers`** This controls the display of markers, signs, and other points of interest in the output HTML. It should be a list of dictionaries.

**Note:** Setting this configuration option alone does nothing. In order to get markers and signs on our map, you must also run the genPO script. See the *Signs and markers* section for more details and documenation.

**Default:** `[]` (an empty list)

**`poititle`** This controls the display name of the POI/marker dropdown control.

**Default:** "Signs"

**`showspawn`** This is a boolean, and defaults to `True`. If set to `False`, then the spawn icon will not be displayed on the rendered map.

**`bgcolor`** This is the background color to be displayed behind the map. Its value should be either a string in the standard HTML color syntax or a 4-tuple in the format of (r,b,g,a). The alpha entry should be set to 0.

**Default:** `#1a1a1a`

**`center`** This is allows you to specify a list or a tuple of Minecraft world coordinates that should be used as the map's default center, e.g. `[800, 64, -334]`.

You may also specify only two coordinates, in case they will be interpreted as X and Z coordinates, and Y is assumed to be `64` (sea level).

**Default:** The coordinates of your spawn, or `[0, 64, 0]` if the regionset has no spawn.

### Map update behavior

**`rerenderprob`** This is the probability that a tile will be rerendered even though there may have been no changes to any blocks within that tile. Its value should be a floating point number between 0.0 and 1.0.

**Default:** `0`

**`forcerender`** This is a boolean. If set to `True` (or any non-false value) then this render will unconditionally re-render every tile regardless of whether it actually needs updating or not.

The *--forcerender* command line option acts similarly, but with one important difference. Say you have 3 renders defined in your configuration file. If you use *--forcerender*, then all 3 of those renders get re-rendered completely. However, if you just need one of them re-rendered, that's unnecessary extra work.

If you set `'forcerender': True,` on just one of those renders, then just that one gets re-rendered completely. The other two render normally (only tiles that need updating are rendered).

You probably don't want to leave this option in your config file, it is intended to be used temporarily, such as after a setting change, to re-render the entire map with new settings. If you leave it in, then Overviewer will end up doing a lot of unnecessary work rendering parts of your map that may not have changed.

Example:

```
renders['myrender'] = {
        'world': 'myworld',
        'title': "Forced Example",
        'forcerender': True,
}
```

**`renderchecks`** This is an integer, and functions as a more complex form of `forcerender`. Setting it to 1 enables *--check-tiles* mode, setting it to 2 enables *--forcerender*, and 3 tells Overviewer to keep this particular render in the output, but otherwise don't update it. It defaults to 0, which is the usual update checking mode.

**`changelist`** This is a string. It names a file where it will write out, one per line, the path to tiles that have been updated. You can specify the same file for multiple (or all) renders and they will all be written to the same file. The file is cleared when The Overviewer starts.

This option is useful in conjunction with a simple upload script, to upload the files that have changed.

**Warning:** A solution like `rsync -a --delete` is much better because it also watches for tiles that should be *deleted*, which is impossible to convey with the changelist option. If your map ever shrinks or you've removed some tiles, you may need to do some manual deletion on the remote side.

### 2.4.3 Custom Rendermodes and Rendermode Primitives

We have generalized the rendering system. Every rendermode is made up of a sequence of *rendermode primitives*. These primitives add some functionality to the render, and stacked together, form a functional rendermode. Some rendermode primitives have options you can change. You are free to create your own rendermodes by defining a list of rendermode primitives.

There are 9 rendermode primitives. Each has a helper class defined in overviewer_core.rendermodes, and a section of C code in the C extension.

A list of rendermode primitives defines a rendermode. During rendering, each rendermode primitive is applied in sequence. For example, the lighting rendermode consists of the primitives "Base" and "Lighting". The Base primitive draws the blocks with no lighting, and determines which blocks are occluded (hidden). The Lighting primitive then draws the appropriate shading on each block.

More specifically, each primitive defines a draw() and an is_occluded() function. A block is rendered if none of the primitives determine the block is occluded. A block is rendered by applying each primitives' draw() function in sequence.

## The Rendermode Primitives

**Base** This is the base of all non-overlay rendermodes. It renders each block according to its defined texture, and applies basic occluding to hidden blocks.

**Options**

**biomes** Whether to render biome coloring or not. Default: True.

Set to False to disable biomes:

```
nobiome_smooth_lighting = [Base(biomes=False), EdgeLines(), SmoothLighting()]
```

**Nether** This doesn't affect the drawing, but occludes blocks that are connected to the ceiling.

**HeightFading** Draws a colored overlay on the blocks that fades them out according to their height.

**Options**

**sealevel** sealevel of the word you're rendering. Note that the default, 128, is usually *incorrect* for most worlds. You should probably set this to 64. Default: 128

**Depth** Only renders blocks between the specified min and max heights.

**Options**

**min** lowest level of blocks to render. Default: 0

**max** highest level of blocks to render. Default: 255

**Exposed** Only renders blocks that are exposed (adjacent to a transparent block).

**Options**

**mode** when set to 1, inverts the render mode, only drawing unexposed blocks. Default: 0

**NoFluids** Don't render fluid blocks (water, lava).

**EdgeLines** Draw edge lines on the back side of blocks, to help distinguish them from the background.

**Options**

**opacity** The darkness of the edge lines, from 0.0 to 1.0. Default: 0.15

**Cave** Occlude blocks that are in direct sunlight, effectively rendering only caves.

**Options**

**only_lit** Only render lit caves. Default: False

**Hide** Hide blocks based on blockid. Blocks hidden in this way will be treated exactly the same as air.

**Options**

**blocks** A list of block ids, or (blockid, data) tuples to hide.

**DepthTinting** Tint blocks a color according to their depth (height) from bedrock. Useful mainly for cave renders.

**Lighting** Applies lighting to each block.

**Options**

**strength** how dark to make the shadows. from 0.0 to 1.0. Default: 1.0

**night** whether to use nighttime skylight settings. Default: False

**color** whether to use colored light. Default: False

**SmoothLighting** Applies smooth lighting to each block.

**Options**

(same as Lighting)

**ClearBase** Forces the background to be transparent. Use this in place of Base for rendering pure overlays.

**SpawnOverlay** Color the map red in areas where monsters can spawn. Either use this on top of other modes, or on top of ClearBase to create a pure overlay.

**Options**

**overlay_color** custom color for the overlay in the format (r,g,b,a). If not defined a red color is used.

**SlimeOverlay** Color the map green in chunks where slimes can spawn. Either use this on top of other modes, or on top of ClearBase to create a pure overlay.

**Options**

**overlay_color** custom color for the overlay in the format (r,g,b,a). If not defined a green color is used.

**MineralOverlay** Color the map according to what minerals can be found underneath. Either use this on top of other modes, or on top of ClearBase to create a pure overlay.

**Options**

**minerals** A list of (blockid, (r, g, b)) tuples to use as colors. If not provided, a default list of common minerals is used.

Example:

```
MineralOverlay(minerals=[(64,(255,255,0)), (13,(127,0,127))])
```

**StructureOverlay** Color the map according to patterns of blocks. With this rail overlays or overlays for other small structures can be realized. It can also be a MineralOverlay with alpha support.

This Overlay colors according to a patterns that are specified as multiple tuples of the form `(relx, rely, relz, blockid)`. So by specifying `(0, -1, 0, 4)` the block below the current one has to be a cobblestone.

One color is then specified as `((relblockid1, relblockid2, ...), (r, g, b, a))` where the `relblockid*` are relative coordinates and the blockid as specified above. The `relblockid*` must match all at the same time for the color to apply.

Example:

```
StructureOverlay(structures=[(((0, 0, 0, 66), (0, -1, 0, 4)), (255, 0, 0, 255)),
                             (((0, 0, 0, 27), (0, -1, 0, 4)), (0, 255, 0, 255))])
```

In this example all rails(66) on top of cobblestone are rendered in pure red. And all powerrails(27) are rendered in green.

If `structures` is not provided, a default rail coloring is used.

**BiomeOverlay** Color the map according to the biome at that point. Either use on top of other modes or on top of ClearBase to create a pure overlay.

**Options**

**biomes** A list of ("biome name", (r, g, b)) tuples to use as colors. Any biome not specified won't be highlighted. If not provided then a default list of biomes and colors is used.

Example:

```
BiomeOverlay(biomes=[("Forest", (0, 255, 0)), ("Desert", (255, 0, 0))])
```

**HeatmapOverlay** Color the map according to when a chunk was last visited. The color for Timestamps between t_invisible and t_full will be interpolated between 0 and 255. This RenderPrimitive might require use of the forcerender option. Otherwise the Overlay might not get updated for not visited chunks (resulting in them always being the brightest color, as if recently visited).

**Options**

**t_invisible** The timestamp when the overlay will get invisible. The default is 30 days ago.

**t_now** The timestamp when the overlay will be fully visible. The default is today.

Example:

```
HeatmapOverlay(
    t_invisible=int((t_now - timedelta(days=2)).timestamp()),
    t_full=int(t_now.timestamp()),
)
```

### Defining Custom Rendermodes

Each rendermode primitive listed above is a Python *class* that is automatically imported in the context of the config file (They come from overviewer_core.rendermodes). To define your own rendermode, simply define a list of rendermode primitive *objects* like so:

```
my_rendermode = [Base(), EdgeLines(), SmoothLighting()]
```

If you want to specify any options, they go as parameters to the rendermode primitive object's constructor:

```
my_rendermode = [Base(), EdgeLines(opacity=0.2),
        SmoothLighting(strength=0.5, color=True)]
```

Then you can use your new rendermode in your render definitions:

```
renders["survivalday"] = {
    "world": "survival",
    "title": "Survival Daytime",
    "rendermode": my_rendermode,
    "dimension": "overworld",
}
```

Note the lack of quotes around `my_rendermode`. This is necessary since you are referencing the previously defined list, not one of the built-in rendermodes.

### Built-in Rendermodes

The built-in rendermodes are nothing but pre-defined lists of rendermode primitives for your convenience. Here are their definitions:

```
normal = [Base(), EdgeLines()]
lighting = [Base(), EdgeLines(), Lighting()]
smooth_lighting = [Base(), EdgeLines(), SmoothLighting()]
night = [Base(), EdgeLines(), Lighting(night=True)]
smooth_night = [Base(), EdgeLines(), SmoothLighting(night=True)]
```

(continues on next page)

(continued from previous page)

```
nether = [Base(), EdgeLines(), Nether()]
nether_lighting = [Base(), EdgeLines(), Nether(), Lighting()]
nether_smooth_lighting = [Base(), EdgeLines(), Nether(), SmoothLighting()]
cave = [Base(), EdgeLines(), Cave(), DepthTinting()]
```

## 2.5 Signs and Markers

The Overviewer can display signs, markers, and other points of interest on your map. This works a little differently than it has in the past, so be sure to read these docs carefully.

In these docs, we use the term POI (or point of interest) to refer to entities and tileentities.

### 2.5.1 Configuration File

#### Filter Functions

A filter function is a python function that is used to figure out if a given POI should be part of a markerSet or not, and to control how it is displayed. The function should accept one argument (a dictionary, also know as an associative array), and return a string representing the text to be displayed. For example:

```
def signFilter(poi):
    if poi['id'] == 'Sign' or poi['id'] == 'minecraft:sign':
        return "\n".join([poi['Text1'], poi['Text2'], poi['Text3'], poi['Text4']])
```

**Note:** This example is intended as a teaching aid and does not escape HTML, so if you are concerned that your Minecraft players will put HTML/JS into their signs, see below for a version that does do escaping.

If a POI doesn't match, the filter can return None (which is the default if a python functions runs off the end without an explicit 'return').

The single argument will either a TileEntity, or an Entity taken directly from the chunk file. It could also be a special entity representing a player's location or a player's spawn. See below for more details.

In this example, this function returns all 4 lines from the sign if the entity is a sign. For more information of TileEntities and Entities, see the Chunk Format page on the Minecraft Wiki.

A more complicated filter function can construct a more customized display text:

```
def chestFilter(poi):
    if poi['id'] == "Chest" or poi['id'] == 'minecraft:chest':
        return "Chest with %d items" % len(poi.get('Items', []))
```

It is also possible to return a tuple from the filter function to specify a hovertext different from the text displayed in the info window. The first entry of the tuple will be used as the hover text, the second will be used as the info window content:

```
def chestFilter(poi):
    if poi['id'] == "Chest" or poi['id'] == 'minecraft:chest':
        return ("Chest", "Chest with %d items" % len(poi.get('Items', [])))
```

Additionally, you can filter based on other Block Entity data by including references to other Minecraft Block Entity fields. For instance, you can filter out world-generated lootable chests that have not yet been opened by players by filtering out chests that still have loot tables:

```
def chestFilter(poi):
    if poi['id'] == "Chest" or poi['id'] == 'minecraft:chest':
        if "LootTable" in poi:
            return None
        else:
            return ("Chest", "Chest with %d items" % len(poi.get('Items', [])))
```

Because of the way the config file is loaded, if you need to import a function or module for use in your filter function, you need to explicitly load it into the global namespace:

```
global escape
from html import escape
def signFilter(poi):
    if poi['id'] == 'Sign' or poi['id'] == 'minecraft:sign':
        return escape("\n".join([poi['Text1'], poi['Text2'], poi['Text3'], poi['Text4
↪']]))
```

Since writing these filters can be a little tedious, a set of predefined filters functions are provided. See the *Predefined Filter Functions* section for details.

### Special POIs

There are currently two special types of POIs. They each have a special id:

**PlayerSpawn** Used to indicate the spawn location of a player. The player's name is set in the `EntityId` key, and the location is in the x,y,z keys.

**Player** Used to indicate the last known location of a player. The player's name is set in the `EntityId` key, and the location is in the x,y,z keys.

**Note:** The player location is taken from level.dat (in the case of a single-player world) or the player.dat files (in the case of a multi-player server). The locations are only written to these files when the world is saved, so this won't give you real-time player location information.

Here's an example that displays icons for each player:

```
def playerIcons(poi):
    if poi['id'] == 'Player':
        poi['icon'] = "https://overviewer.org/avatar/%s" % poi['EntityId']
        return "Last known location for %s" % poi['EntityId']
```

Note how each POI can get a different icon by setting `poi['icon']`. These icons must exist in either the output folder, or in your custom web assets folder. If the icon file does not exist in the correct location, your markers will be shown without an icon - making them invisible!

### Manual POIs

It is also possible to manually define markers. Each render can have a render dictionary key called `manualpois`, which is a list of dicts. Each dict represents a marker, and is required to have at least the attributes `x`, `y`, `z` and `id`, with the coordinates being Minecraft world coordinates. (i.e. what you see in-game when you press F3)

An example which adds two POIs with the id “town”, and then uses a filter function to filter for them:

```
def townFilter(poi):
    if poi['id'] == 'Town':
        return poi['name']


renders['myrender'] = {
    'world':'myworld',
    'title':'Example',
    'manualpois':[
                    {'id':'Town',
                     'x':200,
                     'y':64,
                     'z':200,
                     'name':'Foo'},
                    {'id':'Town',
                     'x':-300,
                     'y':85,
                     'z':-234,
                     'name':'Bar'}],
    'markers': [dict(name="Towns", filterFunction=townFilter)],
}
```

Here is a more complex example where not every marker of a certain id has a certain key:

```
def townFilter(poi):
    if poi['id'] == 'Town':
        try:
            return (poi['name'], poi['description'])
        except KeyError:
            return poi['name'] + '\n'


renders['myrender'] = {
    'world':'myworld',
    'title':'Example',
    'manualpois':[
                    {'id':'Town',
                     'x':200,
                     'y':64,
                     'z':200,
                     'name':'Foo',
                     'description':'Best place to eat hamburgers'},
                    {'id':'Town',
                     'x':-300,
                     'y':85,
                     'z':-234,
                     'name':'Bar'}],
    'markers': [dict(name="Towns", filterFunction=townFilter, icon="markers/marker_
↪town.png")],
    ### Note: The 'icon' parameter allows you to specify a custom icon, as per
    ###       standard markers. This icon must exist in the same folder as your
    ###       custom webassets or in the same folder as the generated index.html
    ###       in this case, we use the marker_town.png icon which comes with
    ###       the Overviewer by default, located in a subdirectory of web_assets.
}
```

### Render Dictionary Key

Each render can specify a list of zero or more filter functions. Each of these filter functions become a selectable item in the 'Signs' drop-down menu in the rendered map. Previously, this used to be a list of functions. Now it is a list of dictionaries. For example:

```
renders['myrender'] = {
        'world': 'myworld',
        'title': "Example",
        'markers': [dict(name="All signs", filterFunction=signFilter),
                    dict(name="Chests", filterFunction=chestFilter, icon="chest.png",␣
↪createInfoWindow=False)]
}
```

The following keys are accepted in the marker dictionary:

**name** This is the text that is displayed in the 'Signs' dropdown.

**filterFunction** This is the filter function. It must accept at least 1 argument (the POI to filter), and it must return either None or a string.

**icon** Optional. Specifies the icon to use for POIs in this group. If omitted, it defaults to a signpost icon. Note that each POI can have different icon by setting the key 'icon' on the POI itself. (this can be done by modifying the POI in the filter function. See the example above)

**createInfoWindow** Optional. Specifies whether or not the icon displays an info window on click. Defaults to True

**showIconInLegend** Optional. Specifies whether or not the icon is displayed in the legend. Defaults to False

**checked** Optional. Specifies whether or not this marker group will be checked(visible) by default when the map loads. Defaults to False

## 2.5.2 Generating the POI Markers

**Note:** Markers will not be updated or added during a regular overviewer.py map render! You must use one of the following options to generate your markers.

### The –genpoi option

Running overviewer.py with the *--genpoi* option flag will generate your POI markers. For example:

```
/path/to/overviewer.py --config /path/to/your/config/file.conf --genpoi
```

**Note:** A –genpoi run will NOT generate a map render, it will only generate markers.

If all went well, you will see a "Markers" button in the upper-right corner of your map.

### genPOI.py

The genPOI.py script is also provided, and can be used directly. For example:

```
/path/to/overviewer/genpoi.py --config=/path/to/your/config.file
```

This will generate the necessary JavaScript files needed in your config file's outputdir.

#### Options

genPOI comes with a few options of its own.

**-c** <file>, **--config**=<file>
: The config file to use for the genPOI operation. This must be the same config file that you use for your normal rendering runs.

**-q, --quiet**
: Outputs less information onto the terminal while running.

**--skip-scan**
: Skip scanning the world for entities and tile entities. Useful if you only want to generate markers for players or through manual POIs, as you can speed up the genPOI operation considerably.

**--skip-players**
: Skip reading and retrieving player data during genPOI runs. This is useful if you don't plan on generating markers for the player locations.

### 2.5.3 Predefined Filter Functions

TODO write some filter functions, then document them here

### 2.5.4 Marker Icons Overviewer ships by default

Overviewer comes with multiple small icons that you can use for your markers. You can find them in the `overviewer_core/data/web_assets/markers` directory.

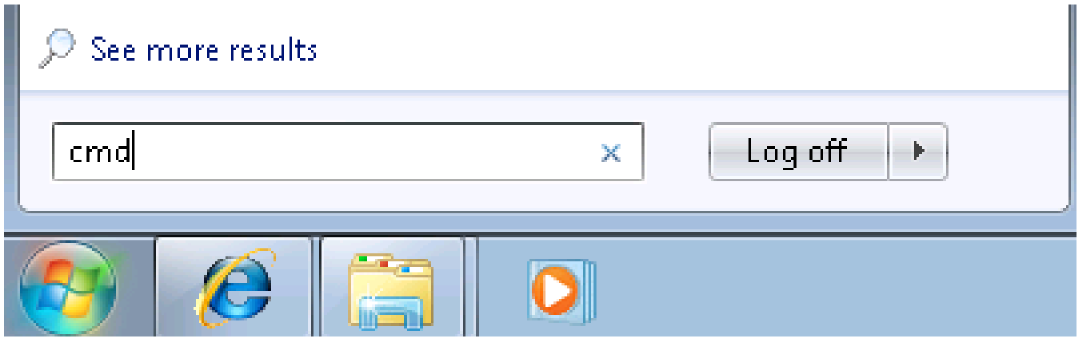
If you want to make your own in the same style, you can use the provided `marker_base_plain.svg` and `marker_base_plain_red.svg` as template, with a vector editing software such as Inkscape.

## 2.6 Windows Newbie Guide

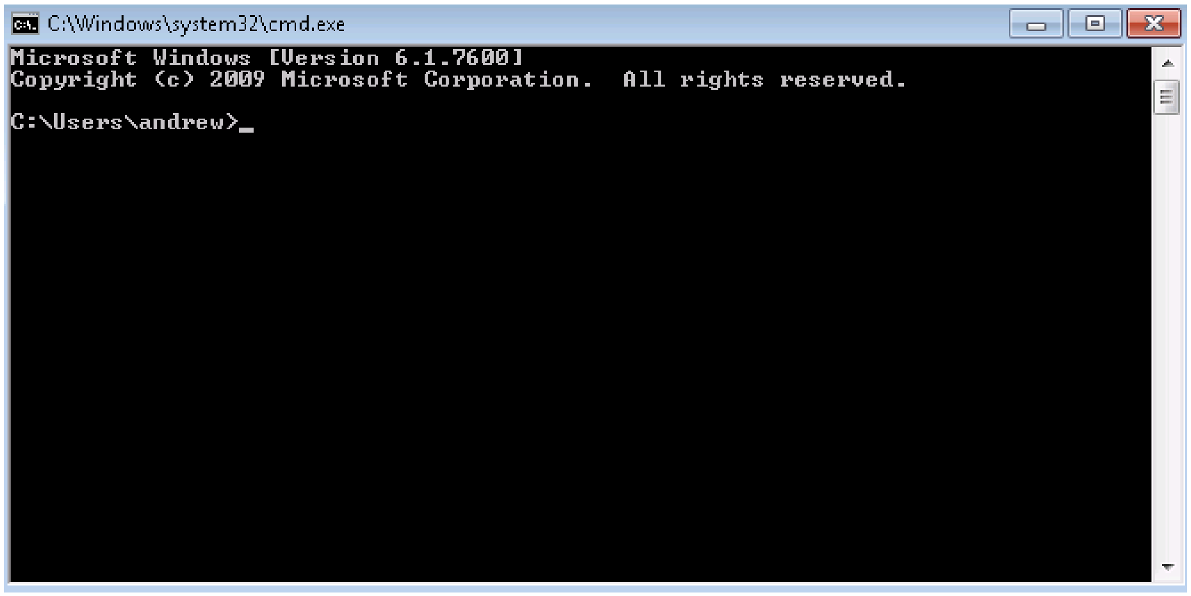
If you're running Windows and aren't as familiar with the Windows command prompt as the rest of the documentation assumes you are, this page is for you!

The Overviewer is a *command line* tool, which means you will need to use the command line to run it.

**First step: Open the command line.** Open your Start menu and type in the box 'cmd' and press enter. If you're running XP you'll go to the "run" option instead and then type 'cmd' and press enter.



This should bring up the *command prompt*, a black window with a prompt where you can type *commands*. The prompt part will probably look something like `C:\Users\andrew>` followed by a cursor where you type your commands.



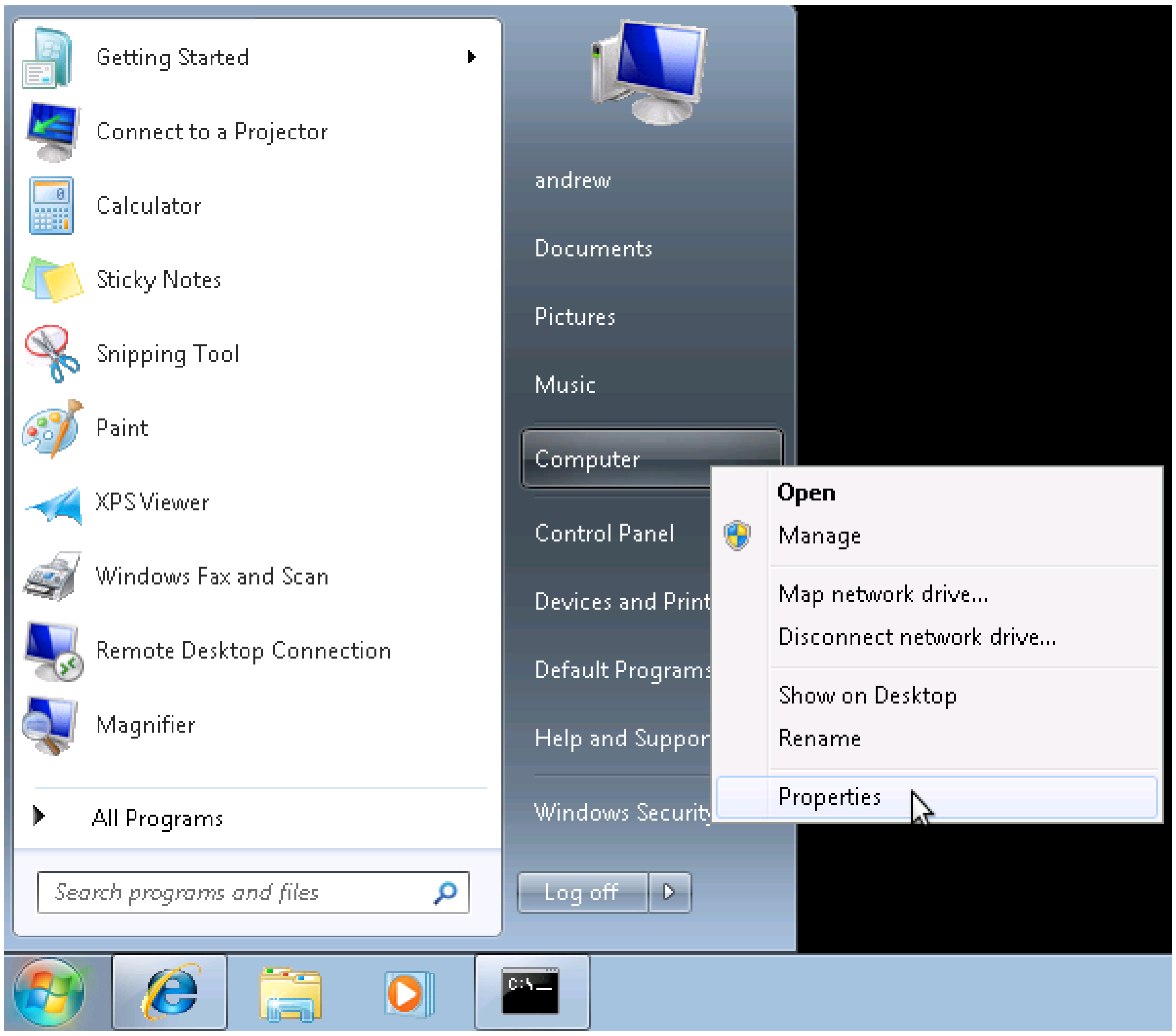
```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\andrew>_
```

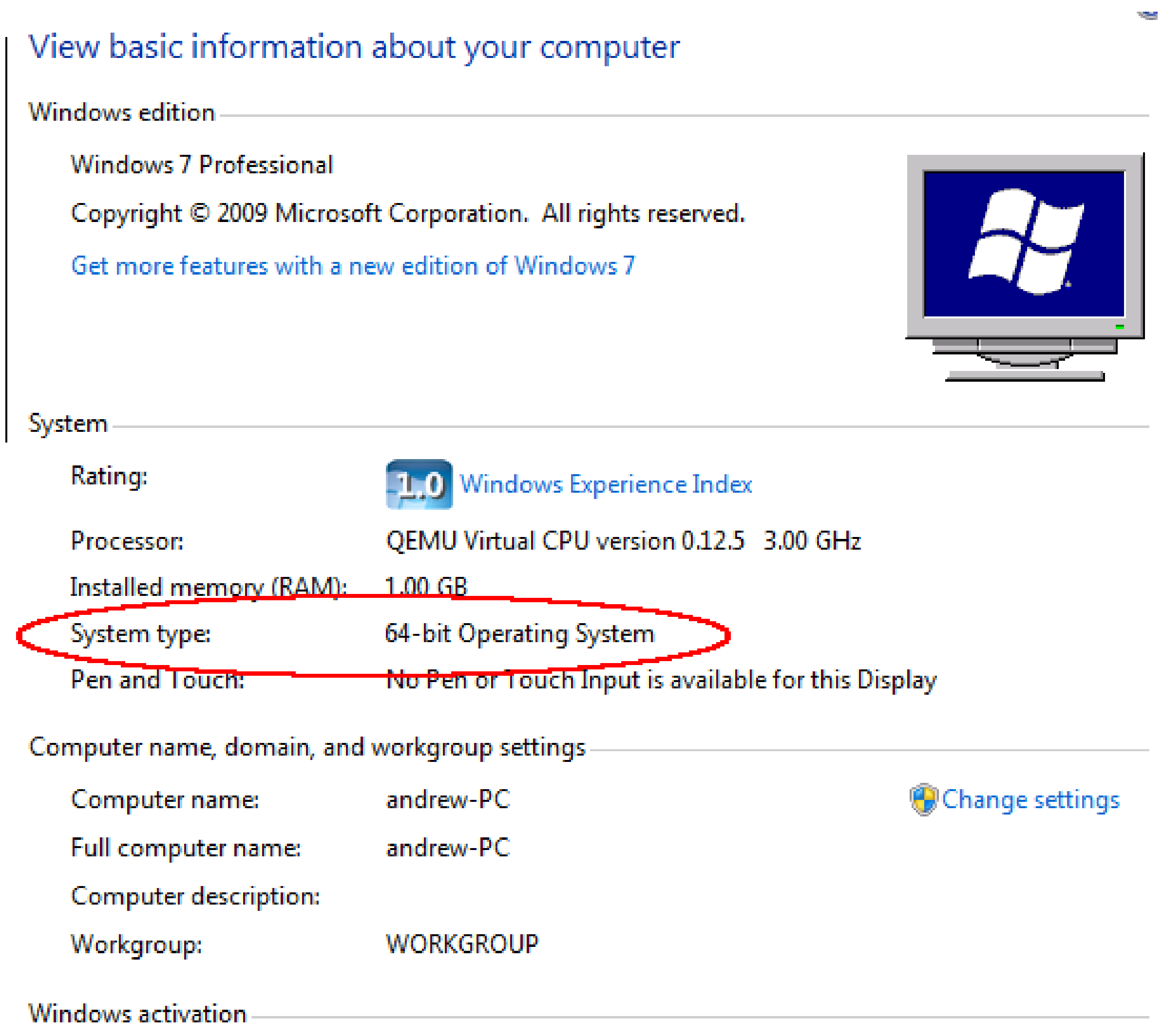
Leave this window open and move on to step 2.

Now that you know how to open a command line, and haven't been scared off yet, the next step is to download the latest Overviewer.

**Step 2: Download Overviewer** Go to the Downloads Page and download the *latest* version for your architecture, either 32 bit or 64 bit.

*This is important. If you don't know which to choose, 32 or 64,* then you can find out by clicking on the start menu, *right clicking* on the "Computer" icon or "My Computer" icon (depending on your version of Windows), and then selecting "Properties." Somewhere among the information about your computer it should tell you if you're running a *32 bit operating system* or *64 bit operating system*.


Getting Started
Connect to a Projector
Calculator
Sticky Notes
Snipping Tool
Paint
XPS Viewer
Windows Fax and Scan
Remote Desktop Connection
Magnifier
All Programs
Search programs and files
andrew
Documents
Pictures
Music
Computer
Control Panel
Devices and Print
Default Programs
Help and Suppor
Windows Security
Log off
Open
Manage
Map network drive...
Disconnect network drive...
Show on Desktop
Rename
Properties



View basic information about your computer

Windows edition

Windows 7 Professional

Copyright © 2009 Microsoft Corporation. All rights reserved.

Get more features with a new edition of Windows 7

System

| | |
|---|---|
| Rating: | 1.0 Windows Experience Index |
| Processor: | QEMU Virtual CPU version 0.12.5 3.00 GHz |
| Installed memory (RAM): | 1.00 GB |
| System type: | 64-bit Operating System |
| Pen and Touch: | No Pen or Touch Input is available for this Display |

Computer name, domain, and workgroup settings

| | | |
|---|---|---|
| Computer name: | andrew-PC | Change settings |
| Full computer name: | andrew-PC | |
| Computer description: | | |
| Workgroup: | WORKGROUP | |

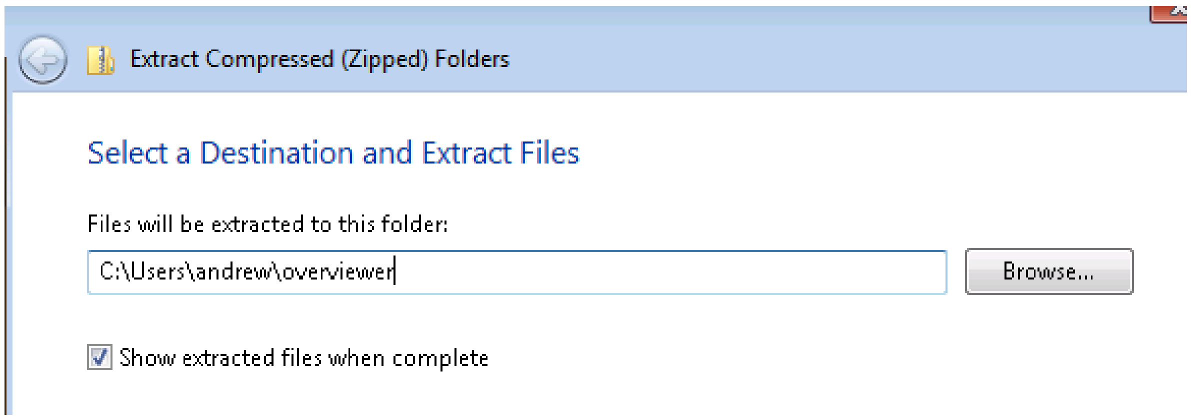
Windows activation

Once you know if your computer is 32 or 64 bit, go and download the latest version. We make small changes all the time, and a new version is uploaded to that page for every change we make. It's usually best to just get the latest.

Okay, you've got a command prompt open. You've got The Overviewer downloaded. We're half way there!

**Step 3: Extract the Overviewer zip you downloaded.** This is easy. I assume you know how to unzip things. Unzip the contents to somewhere you can find easily. You'll need to find it in the command prompt. It may help to leave the window with the unzipped contents open so you can remind yourself where it is.

*Keep all those files together!* They're all needed to run The Overviewer.



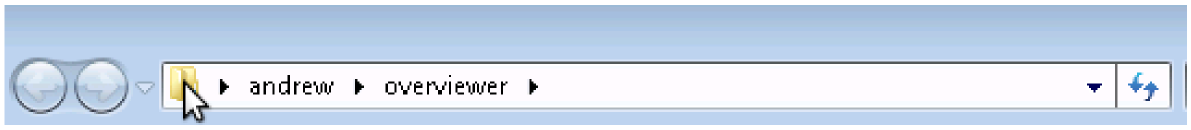
Select a Destination and Extract Files

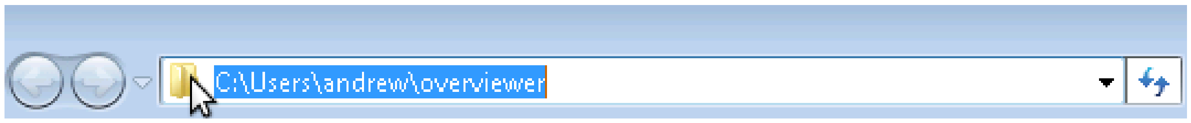
Files will be extracted to this folder:

C:\Users\andrew\overviewer

Show extracted files when complete: ✓

**Step 4: Change directory in command prompt to the location of overviewer.exe** You remember the location of the files you just extracted? Windows doesn't always make it easy. Here's how in windows 7: just click on the little icon to the left of the directory name.

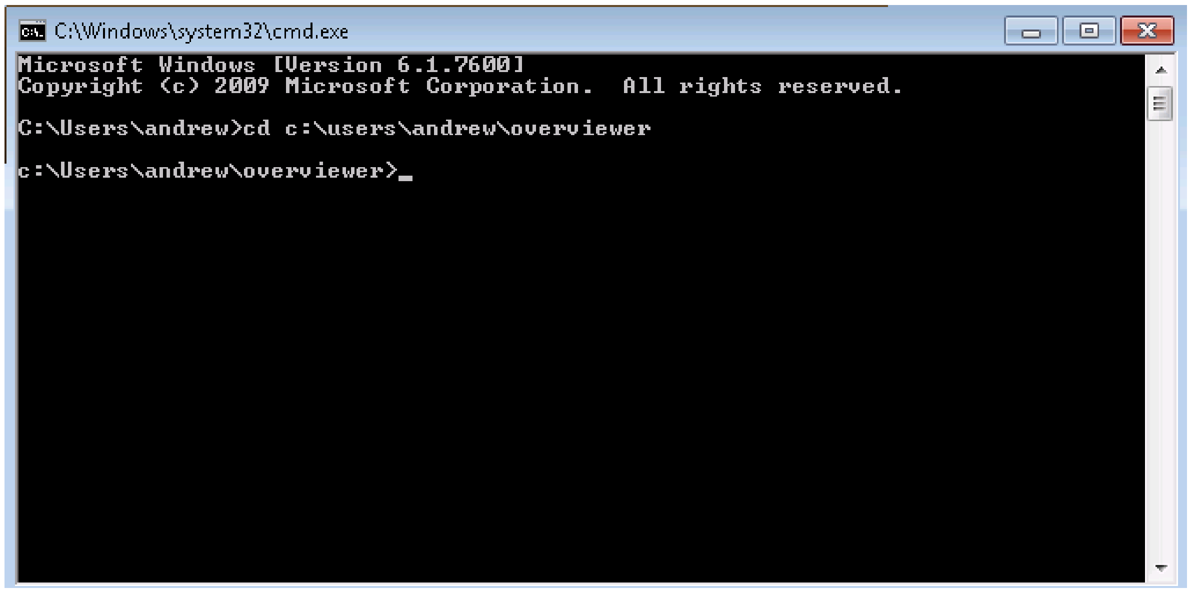




Got the location? Good. We're going to *change directory* to that directory with the command prompt. Bring the command prompt window back up. The command we're going to use is called `cd`, it stands for . . . *change directory*!

I'm going to illustrate this with an example. Let's say you extracted Overviewer to the directory `c:\users\andrew\overviewer`. Here is exactly what you'll type into the command prompt and then press enter:

```
cd c:\users\andrew\overviewer
```



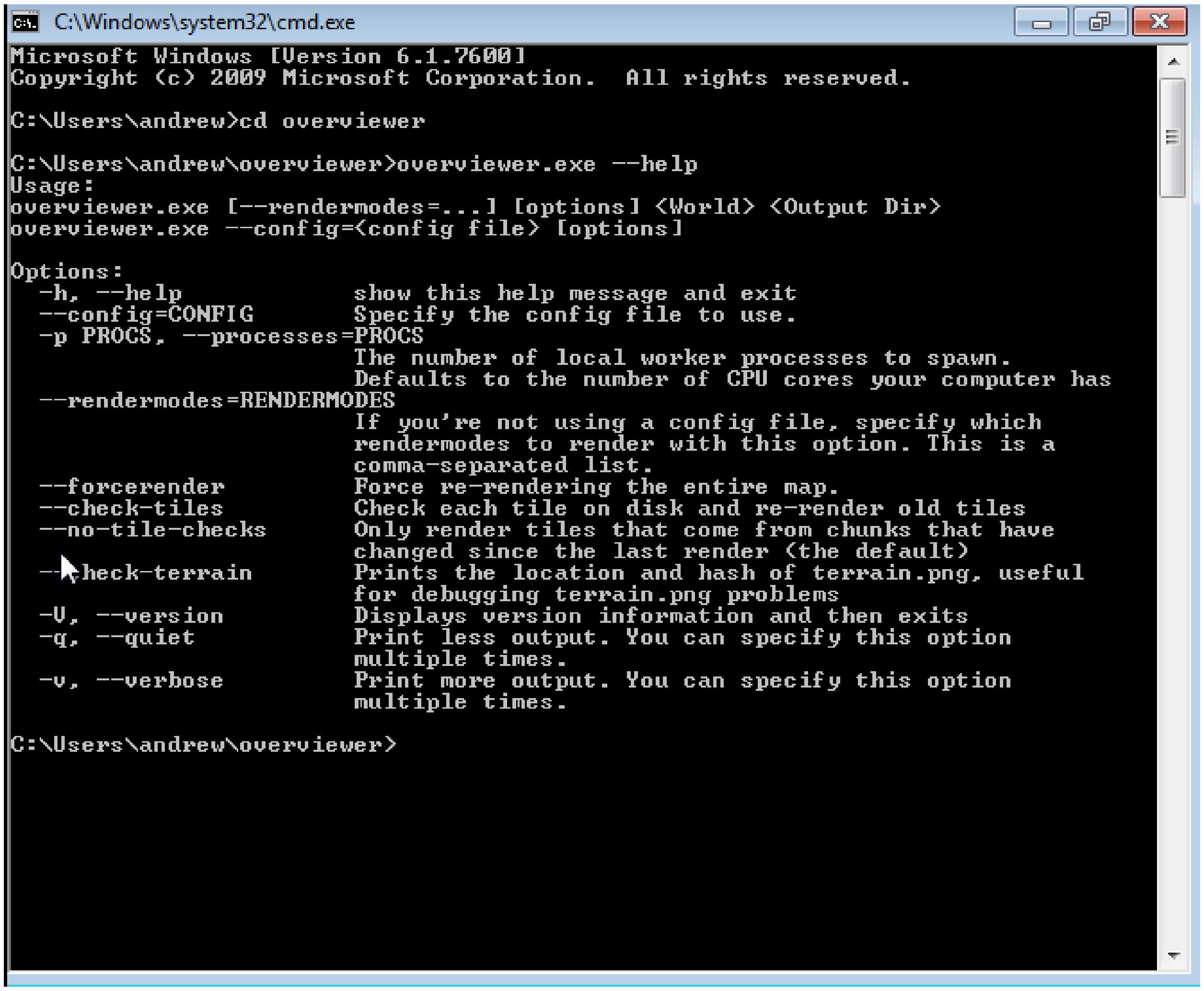
```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\andrew>cd c:\users\andrew\overviewer

c:\Users\andrew\overviewer>_
```

Okay, did it work? Your command *prompt* should now have the *current working directory* in it. If your prompt changed to the directory that you just cd'd to, then your current directory changed successfully! You're ready for the next step!

Okay before we actually run Overviewer for real, let's do a checkpoint. You should have *cd*'d to the directory where overviewer.exe is. To test, type this in and you should see the help text print out:

```
overviewer.exe --help
```

note the two hyphens before "help". You should see something like this:



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\andrew>cd overviewer

C:\Users\andrew\overviewer>overviewer.exe --help
Usage:
overviewer.exe [--rendermodes=...] [options] <World> <Output Dir>
overviewer.exe --config=<config file> [options]

Options:
  -h, --help            show this help message and exit
  --config=CONFIG       Specify the config file to use.
  -p PROCS, --processes=PROCS
                        The number of local worker processes to spawn.
                        Defaults to the number of CPU cores your computer has
  --rendermodes=RENDERMODES
                        If you're not using a config file, specify which
                        rendermodes to render with this option. This is a
                        comma-separated list.
  --forcerender         Force re-rendering the entire map.
  --check-tiles         Check each tile on disk and re-render old tiles
  --no-tile-checks      Only render tiles that come from chunks that have
                        changed since the last render (the default)
  --check-terrain       Prints the location and hash of terrain.png, useful
                        for debugging terrain.png problems
  -V, --version         Displays version information and then exits
  -q, --quiet           Print less output. You can specify this option
                        multiple times.
  -v, --verbose         Print more output. You can specify this option
                        multiple times.

C:\Users\andrew\overviewer>
```

The help text displays the *usage* of overviewer.exe, or the parameters it takes to run it. It's kind of long, I had to make my window larger to show it all.

```
Usage:
overviewer.exe [--rendermodes=...] [options] <World> <Output Dir>
```

Command line tool usage convention says that items in [square brackets] are *optional*, while items in <angled brackets> are *required*.

**Step 5 Render a map!** Okay, so to render a map, you have to run `overviewer.exe` with two *parameters*: the *world path* and a *destination directory*.

Let's say you have a world named "Singleplayer world" and you want to put the tiles into a directory on your desktop. Singleplayer worlds are stored on your hard drive at a location called `%appdata%\.minecraft\saves`. Try typing this into the command prompt:

```
overviewer.exe "%appdata%\.minecraft\saves\Singleplayer World" ␣
↪c:\users\andrew\desktop\mymap
```

**Note:** You must use quotation marks around a path that has spaces in it.

**Note:** `%appdata%` is a special windows "variable" that refers to the location on your drive where applications can store their data. Typing `%appdata%` instead of the full path is a convenient shortcut.

If everything went according to plan, The Overviewer should now be churning away furiously on your world, rendering thousands of image files that compose a map of your world.

When it's done, open up the file `index.html` in a web browser and you should see your map!

I hope this has been enough to get some of you Windows noobs started on The Overviewer. Sorry there's no easy-to-use graphical interface right now. We want to make one, we really do, but we haven't had the time and the talent to do so yet.

The preferred way to run The Overviewer is with a *configuration file*. Without one, you can only do the most basic of renders. Once you're ready, head to the *The Configuration File* page to see what else The Overviewer can do. And as always, feel free to drop by in IRC if you have any questions! We're glad to help!

### 2.6.1 Common Pitfalls

- Wrong working directory:

```
"overviewer.exe" is not recognised as an internal or external
command, operable program, or batch file.
```

  This is a common mistake to make, especially for people unfamiliar with the command line. This happens if your current working directory does not contain overviewer.exe. This is likely because you've forgot to change the working directory to the directory you have unzipped overviewer into. Re-read Step 4 for instructions on how to do that.

- Overviewer is on a different drive than C:

  You may have Overviewer located on a different partition than C:, and for some odd reason the windows command line does not accept "cd D:" as a way to switch partitions. To do this, you have to just type the drive letter followed by a colon:

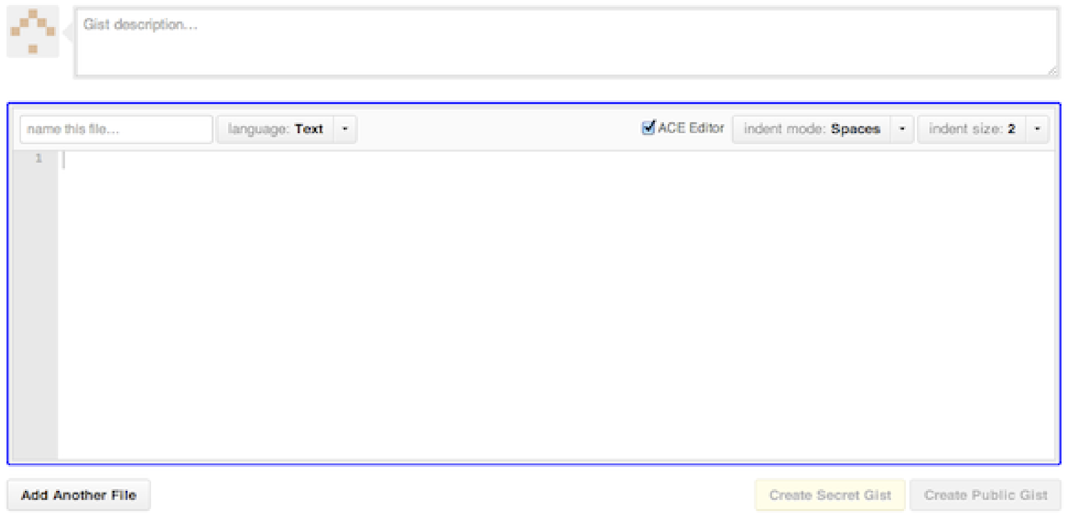
```
D:
```

  This should switch your current working directory to D:

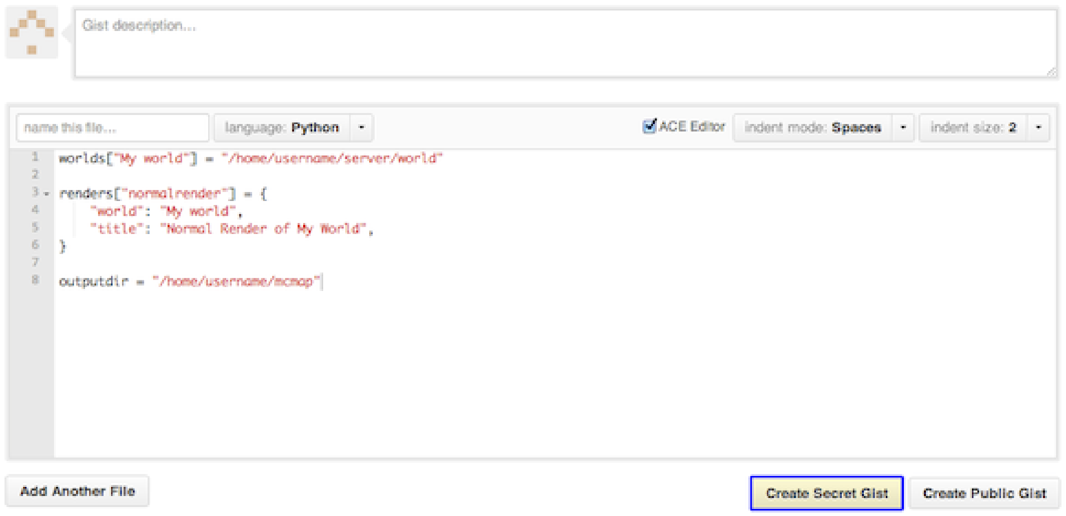
### 2.6.2 Using GitHub Gist

Sometimes, when helping people with issues with Overviewer, we'll often ask to see the config file you're using, or, if there was an Overviewer error, a full copy of an error message. Unfortunately, IRC is not a good way to send large amounts of text. So we often ask users to create a Gist containing the text we want to see. Sites like these are also called Pastebins, and you are welcome to use your favorite pastebin site, if you'd like.

- First, go to http://gist.github.com/
- Second, paste your text into the primary text entry area:

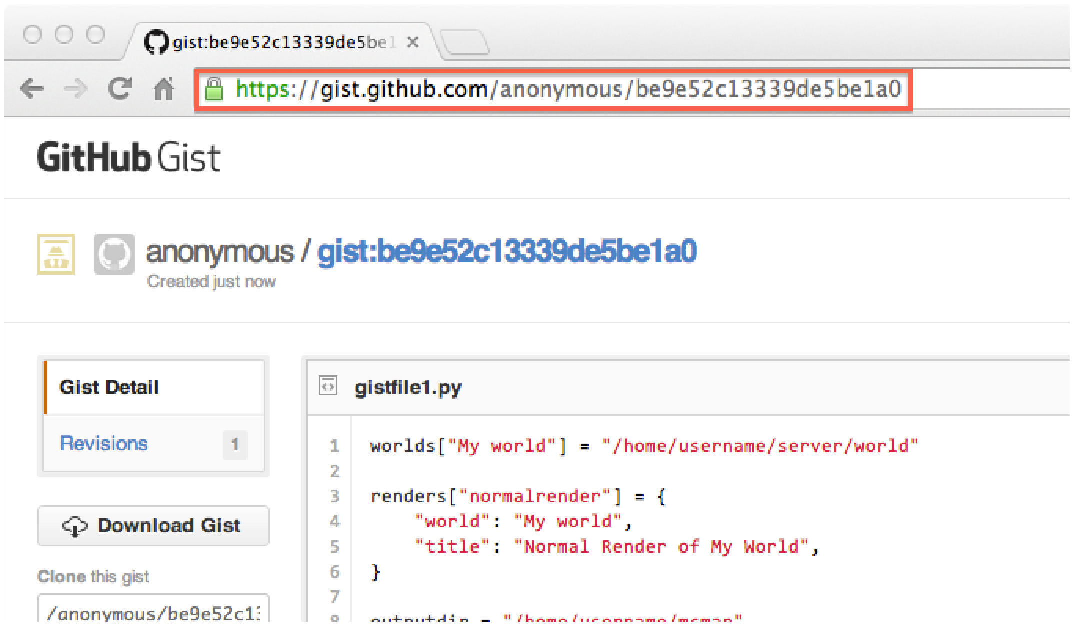


- Third, click the 'Create Secret Gist' button. A secret gist means that only someone with the exact URL can view your gist.



```
worlds["My world"] = "/home/username/server/world"

renders["normalrender"] = {
    "world": "My world",
    "title": "Normal Render of My World",
}

outputdir = "/home/username/mcmap"
```

- Finally, send us the URL. This will let us easily view your properly formatted Gist.



```
worlds["My world"] = "/home/username/server/world"

renders["normalrender"] = {
    "world": "My world",
    "title": "Normal Render of My World",
}

outputdir = "/home/username/mcmap"
```

## 2.7 Frequently Asked Questions

### 2.7.1 General Questions

#### Does the Overviewer work with mod blocks?

The Overviewer will render the world, but none of the blocks added by mods will be visible. Currently, the blocks Overviewer supports are hardcoded, and because there is no official Minecraft modding API as of the time of writing, supporting mod blocks is not trivial.

### Can I view Overviewer maps without having an internet connection?

Yes, absolutely. The Overviewer switched away from the Google Maps API and now uses Leaflet. All files which Overviewer needs are included in the output, so even if you have no internet connection, you will still be able to view the map without any issues.

### When my map expands, I see remnants of another zoom level.

When your map expands ("Your map seems to have expanded beyond its previous bounds") you may see tiles at a zoom level that shouldn't be there, usually around the borders. This is probably not a bug, but is typically caused by copying the map tiles from their render destination to another location (such as a web server).

When you're copying the rendered map, you need to be sure files that *don't* exist in the source are *deleted* in the destination.

Explanation: When Overviewer re-arranges tiles to make room for another zoom level, it moves some tiles at a particular zoom level and places them at a higher zoom level. The tiles that used to be at that zoom level should no longer exist there, but if you're copying tiles, there is no mechanism to *delete* those files at the copy destination.

If that explanation doesn't make full sense, then just know that you must do one of the following:

- Render the tiles directly to the destination.
- Copy the tiles from the render destination in a way that deletes extra files, such as using `rsync` with `--delete`.
- Erase and re-copy the files at the final destination when the map expands. Map expansions double the width and height of the map, so you will eventually hit a map size that is unlikely to need another level.

### You've added a new feature or changed textures, but it's not showing up on my map!

Some new features will only show up in newly-rendered areas. Use the `--forcerender` option to update the entire map. If you have a really large map and don't want to re-render everything, take a look at the *rerenderprob* configuration option.

### The background color of the map is black, and I don't like it!

You can change the background color by specifying a new one in the configuration file. See the *The Configuration File* page for more details.

### I downloaded the Windows version but when I double-click it, the window closes real fast.

The Overviewer is a command line program and must be run from a command line. It is necessary to become at least a little familiar with a command line to run The Overviewer (if you have no interest in this, perhaps this isn't the mapping program for you). A brief guide is provided on the *Windows Newbie Guide* page.

Unfortunately, A full tutorial of the Windows command line is out of scope for this documentation; consult the almighty Google for tutorials and information on the Windows command line. (If you would like to contribute a short tutorial to these docs, please do!)

Batch files are another easy way to run the Overviewer without messing with command lines, but information on how to do this has also not been written.

On a related note, we also welcome contributions for a graphical interface for the Overviewer.

### The Overviewer is eating up all my memory!

We have written The Overviewer with memory efficiency in mind. On even the largest worlds we have at our disposal to test with, it should not be taking more than a gigabyte or two. It varies of course, that number is only an estimate, but most computers with a reasonable amount of RAM should run just fine.

If you are seeing exorbitant memory usage, then it is likely either a bug or a subtly corrupted world. Please file an issue or come talk to us on IRC so we can take a look! See *Help*.

### How can I log The Overviewer's output to a file?

If you are on a UNIX-like system like MacOSX or Linux, you can use shell redirection to write the output into a file:

```
overviewer.py --config=myconfig.py > renderlog.log 2>&1
```

What this does is redirect the previous commands standard output to the file "renderlog.log", and redirect the standard error to the standard output. The file will be overwritten each time you run this command line; to simply append the output to the file, use two greater than signs:

```
overviewer.py --config=myconfig.py >> renderlog.log 2>&1
```

### I've deleted some sections of my world, but they still appear in the map.

Okay, so making edits to your world in e.g. worldedit has some caveats, especially regarding deleting sections of your world.

This faq also applies to using the *crop* option.

Under normal operation with vanilla Minecraft and no external tools fiddling with the world, Overviewer performs correctly, rendering areas that have changed, and everything is good.

Often with servers one user will travel reeeeally far out and cause a lot of extra work for the server and for The Overviewer, so you may be tempted to delete parts of your map. This can cause problems, so read on to learn what you can do about it.

First some explanation: Until recently (Mid May 2012) The Overviewer did not have any facility for detecting parts of the map that should no longer exist. Remember that the map is split into small tiles. When Overviewer starts up, the first thing it does is calculate which tiles should exist and which should be updated. This means it does not check or even look at tiles that should not exist. This means that parts of your world which have been deleted will hang around on your map because Overviewer won't even look at those tiles and notice they shouldn't be there. You may even see strange artifacts around the border as tiles that should exist get updated.

Now, with the `--check-tiles` option, The Overviewer *will* look for and remove tiles that should no longer exist. So you can render your map once with that option and all those extra tiles will get removed automatically. However, this is only half of the solution. The other half is making sure the tiles along the border are re-rendered, or else it will look like your map is being cut off.

Explanation: The tiles next to the ones that were removed are tiles that should continue to exist, but parts of them have chunks that no longer exist. Those tiles then should be re-rendered to show that. However, since tile updates are triggered by the chunk last-modified timestamp changing, and the chunks that still exist have *not* been updated, those tiles will not get re-rendered.

The consequence of this is that your map will end up looking cut-off around the new borders that were created by the parts you deleted. You can fix this one of two ways.

1. You can run a render with `--forcerender`. This has the unfortunate side-effect of re-rendering *everything* and doing much more work than is necessary.

2. Manually navigate the tile directory hierarchy and manually delete tiles along the edge. Then run once again with `--check-tiles` to re-render the tiles you just deleted. This may not be as bad as it seems. Remember each zoom level divides the world into 4 quadrants: 0, 1, 2, and 3 are the upper left, upper right, lower left, and lower right. It shouldn't be too hard to navigate it manually to find the parts of the map that need re-generating.
3. The third non-option is to not worry about it. The problem will fix itself if people explore near there, because that will force that part of the map to update.

### My map is zoomed out so far that it looks (almost) blank.

We see this quite a bit, and seems to stem from a bug in the Minecraft terrain generation.

Explanation: Minecraft generates chunks of your world as it needs them. When Overviewer goes to render your map, it looks at how big the world is, and calculates how big the maps needs to be in order to fit it all in. Occasionally, we see that Minecraft has generated a few chunks of the world extremely far away from the main part of the world. These erroneous chunks have most likely not been explored*[0] and should not exist.

There are two solutions. The preferred is to delete the offending chunks. Open up your region folder of your world and look at the region file names. They are numbered `r.##.##.mcr` where `##` is a number. The two numbers indicate the coordinates of that region file. Look for region files with coordinates much larger in magnitude than any others. Most likely you will find around 1–3 region files with coordinates much larger than any others. Delete or otherwise remove those files, and re-render your map.

The other option is to use the *crop* option to tell Overviewer not to render all of your map, but instead to only render the specified region.

As always, if you need assistance, come chat with us on *irc*.

### I want to put manual POI definitions or other parts of my config into a separate file.

This can be achieved by creating a module and then importing it in your config. First, create a file containing your markers definitions. We'll call it `manualmarkers.py`.

```
mymarkers = [{'id':'town', 'x':200, 'y':64, 'z':-400, 'name':'Pillowcastle'},
             {'id':'town', 'x':500, 'y':70, 'z': 100, 'name':'brownotopia' }]
```

The final step is to import the very basic module you've just created into your config. In your config, do the following

```
import sys
sys.path.append("/wherever/your/manualmarkers/is/")  # Replace this with your path to␣
↪manualmarkers.py,
                                                     # so python can find it

from manualmarkers import *                          # import our markers

# all the usual config stuff goes here

renders["myrender"] = {
    "title" : "foo",
    "world" : "someworld",
    "manualpois" : mymarkers,                          # IMPORTANT! Variable name from␣
↪manualmarkers.py
    # and here goes the list of the filters, etc.
}
```

Now you should be all set.

[0] They could also have been triggered by an accidental teleport where the coordinates were typed in manually.

## 2.8 Contributing

In this page, you'll be given some pointers on how to start contributing to the Minecraft-Overviewer project. This is useful for people who want to help develop the Overviewer, but don't quite know where to start.

This page is mostly focused on where to look for things and how to get your changes back into the project, for help on how to compile the Overviewer, check *Building*.

### 2.8.1 Prerequisites

Ideally you're familiar with Python (Overviewer uses Python 3), and know the basics of Git. Both have various very good resources online that help you in learning them, but the best way of learning is always to use them in the real world, so don't hesitate to jump right in after having a basic grasp and ask questions along the way.

Additionally, some parts of Overviewer are written in C, though unless you're interested in the drawing and compositing routines or the rendermodes, you don't need to know C.

Last but not least, some of the Overviewer's code is written in JavaScript, namely the part that runs in your browser when you view the map.

### 2.8.2 Acquiring the Source Code

First, you'll need to get the Overviewer source code. We do version management of code through Git, which allows multiple people to work on the code at the same time. Naturally, this means you'll also be getting the source code through Git. For this to work, you'll have to install Git on your computer.

Our source code is hosted on GitHub, so it's a good idea to make an account there if you don't already have one.

This page won't go into the details of how to use Git, but it'll give you some advice on how your workflow should be to avoid some trouble.

### 2.8.3 Finding Your Way around the Code Base

At first glance, all the code can be a bit overwhelming. So here's a quick overview of the important parts.

- `setup.py` is the build script. If you need to make any changes to how the Overviewer is built, you'll want to look there.
- `overviewer.py` is the entry-point of the application. It imports all the other functionality, and does the command line parsing.
- `overviewer_core/` is the directory where the vast majority of the Overviewer's functionality is. More on that below.
- `overviewer_core/aux_files/genPOI.py` is where the genPOI functionality is implemented. If you're looking into changing the way markers are generated, look there.
- `overviewer_core/src/` is the directory for all the files that are part of Overviewer's C extension. This includes things such as rendermodes, which are stored in the `primitives` sub-directory.
- `overviewer_core/data/` mostly contains the parts that make up Overviewer's web front-end, with `js_src` containing the JS files and `web_assets` containing the `index.html`, CSS files and image files such as icons or the compass.
- `docs/` contains the documentation, which can be built with the included Makefile if you have sphinx installed.

### overviewer_core

Let's take a closer look at the `overviewer_core/` directory:

- `assetmanager.py` controls how the HTML and JS output are written out, as well as the `overviewerConfig.js` format.
- `cache.py` implements a Least-Recently-Used (LRU) cache, which is used for caching chunks in memory as the rendering happens.
- `config_parser.py` contains some code that sets up how the config is parsed, but is not really involved in the definitions of individual settings therein.
- `dispatcher.py` is the code that sets up multiprocessing, so Overviewer can use all available CPU threads on a machine.
- `files.py` implements helpful routines which allow you to determine whether some file operations such as replacing a file work in a given directory, and also implements the `FileReplacer` class which can then safely replace a file given the capabilities of the filesystem.
- `items.py` is a remnant of the past and entirely unused.
- `logger.py` sets up and implements Overviewer's logging facilities.
- `nbt.py` contains the code that is used to parse the Minecraft NBT file structure.
- `observer.py` defines all the observers that are available. If you want to add a new observer, this is the place where you'll want to look.
- `optimizeimages.py` defines all the optimizeimg tools and how they're called.
- `progressbar.py` implements the fancy progress bar that the Overviewer has.
- `rcon.py` implements an rcon client for the Minecraft server, used by the RConObserver.
- `rendermodes.py` contains definitions and glue code for the rendermodes in the C extension.
- `settingsDefinitions.py` includes all definitions for the Overviewer configuration file. If you want to add a new configuration option, this is where you'll want to start.
- `settingsValidators.py` contains validation code for the settings definitions, which ensures that the values are all good.
- `signals.py` is multiprocessing communication code. Scary stuff.
- `textures.py` contains all the block definitions and how Overviewer should render them. If you want to add a new block to the Overviewer, this is where you'll want to do it. Additionally, this code also controls how the textures are loaded.
- `tileset.py` contains code that maps a render dict entry to the output tiled image structure.
- `util.py` contains random utility code that has no home anywhere else.
- `world.py` is a whole lot of code that does things like choosing which chunks to load and to cache, and general functionality revolving around the concept of Minecraft worlds.

### docs

The documentation is written in reStructuredText, a markup format. It can be compiled into an HTML output using the Makefile in the `docs/` subtree by typing `make`. You'll need to have sphinx installed for this to work.

The theme that will be used in the locally generated HTML is different than what is used on http://docs.overviewer.org. However, it should still be sufficient to get a good idea of how your changes will end up looking like when they're on the main docs page.

### 2.8.4 Code Style

To be honest, currently the Overviewer's codebase is a bit of a mess. There is no consistent code style in use right now. However, it's probably a good idea to stick to PEP8 when writing new code. If you're refactoring old code, it would be great if you were to fix it to make it PEP8 compliant as well.

To check whether the code is PEP8 compliant, you can use pycodestyle. You can easily install it with pip by using `pip3 install pycodestyle`.

### 2.8.5 Example Scenarios

This section will demonstrate by example how a few possible contributions might be made. These serve as guidelines on how to quickly get started if you're interested in doing a specific task that many others before you have done too in some other form.

#### Adding a Block

Let's assume you want to add support for a new block to the Overviewer. This is probably one of the most common ways people start contributing to the project, as all blocks in the Overviewer are currently hardcoded and code to handle them needs to be added by hand.

The place to look here is `textures.py`. It contains the block definitions, which are assisted by Python decorators, which make it quite a bit simpler to add new blocks.

The big decorator in question is `@material`, which takes arguments such as the `blockid` (a list of block IDs this block definition should handle), and `data` (a list of possible data values for this block). Additionally, it can also take various additional arguments for the different block properties, such as `solid=True` to indicate that the block is a solid block.

#### Simple Solid 6-Sided Block

A lot of times, new blocks are basically just your standard full-height block with a new texture. For a block this simple, we don't even really need to use the material decorator. As an example, check out the definition of the coal block:

```
block(blockid=173, top_image="assets/minecraft/textures/blocks/coal_block.png")
```

#### Block with a Different Top

Another common theme is a block where the top is a different texture than the sides. Here we use the `@material` decorator to create the jukebox block:

```
@material(blockid=84, data=range(16), solid=True)
def jukebox(self, blockid, data):
    return self.build_block(self.load_image_texture("assets/minecraft/textures/blocks/
↪jukebox_top.png"), self.load_image_texture("assets/minecraft/textures/blocks/
↪noteblock.png"))
```

As you can see, we define a method called `jukebox`, taking the parameters `blockid` and `data`, decorated by a decorator stating that the following definition is a material with a `blockid` of `84` and a data value range from `0` to `15` (or `range(16)`), which we won't use as it doesn't affect the rendering of the block. We also specify that the block is solid.

Inside the method, we then return the return value of `self.build_block()`, which is a helper method that takes a texture for the top and a texture for the side as its arguments.

#### Block with Variable Colors

Occasionally, blocks can have colors stored in their data values. `textures.py` includes an easy mapping list, called `color_map`, to map between data values and Minecraft color names. Let's take stained hardened clay as an example of how this is used:

```
@material(blockid=159, data=range(16), solid=True)
def stained_clay(self, blockid, data):
    texture = self.load_image_texture("assets/minecraft/textures/blocks/hardened_clay_
↪stained_%s.png" % color_map[data])

    return self.build_block(texture,texture)
```

As you can see, we specify that the block has 16 data values, then depending on the data value we load the right block texture by looking up the color name in the `color_map` list, formatting a string for the filename with it.

### 2.8.6 Good Git Practices

How you structure your Git workflow is ultimately up to you, but here are a few recommendations to make your life and the life of the people who want to merge your pull requests easier.

- **Commit your changes in a separate branch, and then submit a pull request from that branch.** This makes it easier for you to rebase your changes, and allows you to keep your repository's master branch in-sync with our master branch, so you can easily split off a new branch from master if you want to develop a new change while your old change still isn't merged into the master.
- **Format your commit messages properly.** The first line should be a 50 character long summary of the change the commit makes, in present tense, e.g. "Add a spinner to the progress bar". This should be followed by a blank line, and a longer explanation of the change the commit actually does, wrapped at 72 characters.
- **Don't merge master into your branch.** If you plan on submitting a change as a pull request and the master branch has moved in the meantime, then don't merge the master branch into the branch of your pull request. Instead, rebase your branch on top of the updated master.
- **Keep commits logically separated.** Don't try to cram unrelated changes into just one commit unless it's a commit full of small fixes. If you find yourself struggling to keep the commit summary below 50 characters, and find yourself using the word "and" in it, rethink whether the changes you're making should be just one commit.

It's also a good idea to look at the output of `git diff` before committing a change, to make sure nothing was unintentionally changed in the file where you weren't expecting it. `git diff` will also highlight blank lines with spaces in them with a solid red background.

### 2.8.7 Talking with other Developers

Occasionally, the issue tracker simply doesn't cut it. You need to talk with another developer, maybe to brainstorm a new feature or ask a question about the code. For this, we have an IRC channel on Libera.Chat, which allows you to talk with other developers that are on the IRC channel in real-time.

Since most developers have jobs or are in college or university, it may sometimes take a few moments to get a reply. So it's useful to stick around and wait for someone who can help you to be around.

## 2.9 Design Documentation

So you'd like a technical overview of how The Overviewer works, huh? You've come to the right place!

This document's scope does not cover the details of the code. The code is fairly well commented and not difficult to understand. Instead, this document is intended to give an explanation to how the Overviewer was designed, why certain decisions were made, and how all the pieces fit together. Think of this document as commenting on how all the high level pieces of the code work.

This document is probably a good read to anyone that wants to get involved in Overviewer development.

So let's get started!

**Note:** This page is continually under construction.

**Contents**

### 2.9.1 Background Info

The Overviewer's task is to take Minecraft worlds and render them into a set of tiles that can be displayed with a Leaflet interface. This section goes over how Minecraft worlds work and are stored.

A Minecraft world extends indefinitely along the two horizontal axes, and are exactly 256 units high. Minecraft worlds are made of voxels (volumetric pixels), hereby called "blocks", where each block in the world's grid has a type that determines what it is (grass, stone, . . . ). This makes worlds relatively uncomplicated to render, the Overviewer simply determines what blocks to draw and where. Since everything in Minecraft is aligned to a strict grid, placement and rendering decisions are completely deterministic and can be performed iteratively.

The coordinate system for Minecraft has three axes. The X and Z axes are the horizontal axes. They extend indefinitely towards both positive and negative infinity. (There are practical limits, but no theoretical limits). The Y axis extends from 0 to 255, which corresponds with the world height limit. Each block in Minecraft has a coordinate address, e.g. the block at 15,78,-35 refers to 15 along the X axis, -35 along the Z axis, and 78 units up from bedrock.

The world is organized in a three-layer hierarchy. At the finest level are the blocks (voxels). A 16x16x16 array of blocks is called a *chunk section*. A vertical column of 16 chunk sections makes a *chunk*. A chunk is therefore a 16 by 16 area of the world that extends from bedrock to sky. In other words, a 16 by 256 by 16 "chunk" of the world. A 32 by 32 area of chunks is called a *region*. Regions are stored on disk one per file.

While blocks have a global coordinate (the ones you see in the debug output in-game), they also have a local coordinate within a chunk section and within a chunk. Also, chunks and regions have their own coordinates to identify themselves. To find which chunk a block is in, simply divide its coordinates by 16 and take the floor. To find which region a chunk is in, divide the chunk coordinates by 32 and take the floor. To find which chunk section a block is in, take its Y coordinate and floor-divide by 16.

Minecraft worlds are generated on-the-fly by the chunk. This means not all chunks will exist, and not all sections within a chunk will exist. There is no pattern to which chunks are generated, the game generates them as needed as players explore the area. A region file may not exist if none of its chunks exist.

### 2.9.2 Overviewer at a High Level

Minecraft worlds are rendered in an approximated Isometric projection at an oblique angle. In the original design, the projection acts as if your eye is infinitely far away looking down at the world at a 45 degree angle in the South-East direction (now, the world can be rendered at any of the 4 oblique directions).

The Overviewer is a sprite-based renderer. Each block type corresponds to a pre-rendered sprite (a small image). The basic idea is to iterate over the blocks of the world and draw these sprites to the appropriate location on the map.

These are the high-level tasks The Overviewer must perform in rendering a map:

1. Render each block sprite from the textures
2. Scan the chunks of the world and determine which tiles need rendering
3. Render a chunk by drawing the appropriate blocks sprites on an image
4. Render a tile of the map from several chunk images
5. Compose the lower-zoom tiles from the higher-zoom tiles

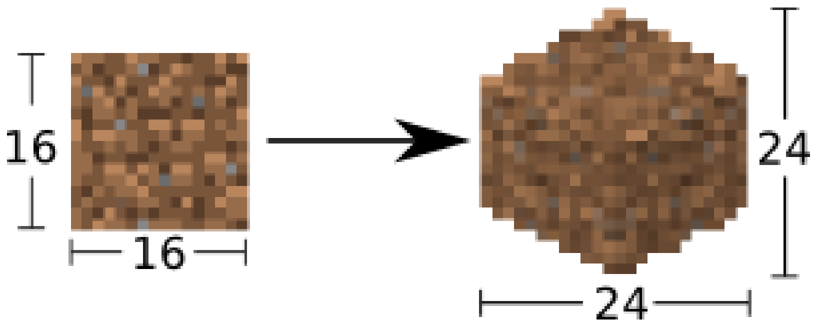
The next sections will go over how these tasks work.

### 2.9.3 Block Rendering

The first step is rendering the block sprites from the textures. Each block is "built" from its textures into an image of a cube and cached in a `Textures` object.

Textures come from files inside of a "textures" folder. If the file is square (has equal width and height dimensions), it is scaled down to 16 x 16 pixels. Non-square images are used with animated textures. In this case, the first frame of the animated texture is used, and also scaled to a 16 by 16 image. In order to draw a cube out of the textures, an affine transformation is applied to the images for the top and sides of the cube in order to transform it to the appropriate perspective.

**Note:** This section goes over the simple case for a regular cube, which are most of the blocks in Minecraft. There are lots of irregular blocks that aren't cubes (fences, torches, doors) which require custom rendering. Irregular blocks are not covered by this design document. Each type of block has its own function in `overviewer_core.textures` that defines how to render it.



Every block sprite is exactly 24 by 24 pixels in size. This particular size for the cubes was chosen for an important reason: 24 is divisible by 2 and by 4. This makes placement much easier. E.g. in order to draw two cubes that are next to each other in the world, one is drawn exactly 12 pixels over and 6 pixels down from the other. All placements of the cubes happen on exact pixel boundaries and no further resolution is lost beyond the initial transformations. (This advantage will become clear in the *Block Positioning* section; all offsets are a nice even 6, 12, or 24 pixels)

A cube sprite is built in two stages. First, the texture is transformed for the top of the cube. Then the texture is transformed for the left side of the cube, which is mirrored for the right side of the cube.

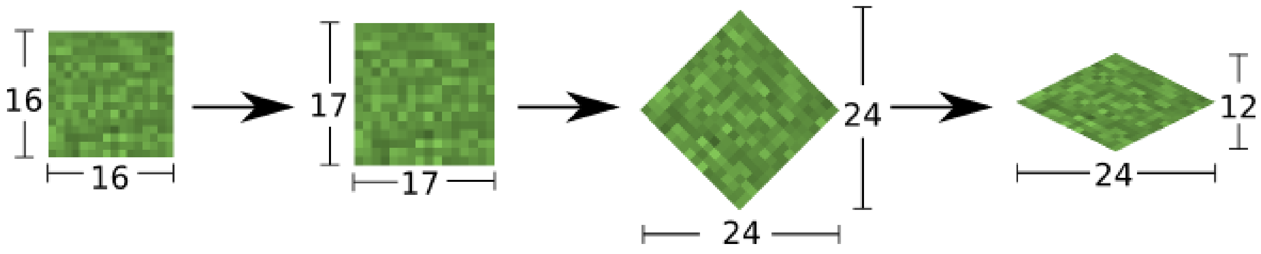
### Top Transformation

The transformation for the top face of the cube is a simple affine transformation from the original square texture. It is actually several affine transformations: a re-size, a rotation, and a scaling; but since multiple affine transformations can be chained together simply by multiplying the transformation matrices together, only one transformation is actually done.

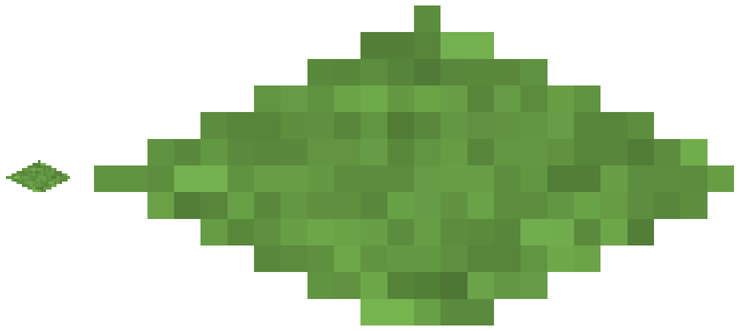
This can be seen in the function `overviewer_core.textures.transform_image()`. It performs three steps:

1. The texture is re-sized to 17 by 17 pixels. This is done because the diagonal of a square with sides 17 is approximately 24, which is the target size for the bounding box of the cube image. So when it's rotated, it will be the correct width. (Better to scale it now than after we rotate it)
2. The image is rotated 45 degrees about its center.
3. The image is scaled on the vertical axis by a factor of 1/2.

This produces an image of size 24 by 12 as seen in the following sequence.



The final image, shown below, becomes the top of the cube.

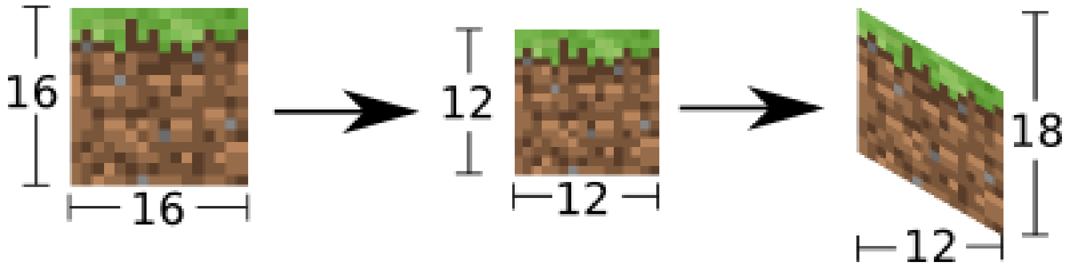


On the left is what will become the top of the block at actual size after the transformation, the right is the same but blown up by a factor of 10 with no interpolation to show the pixels.

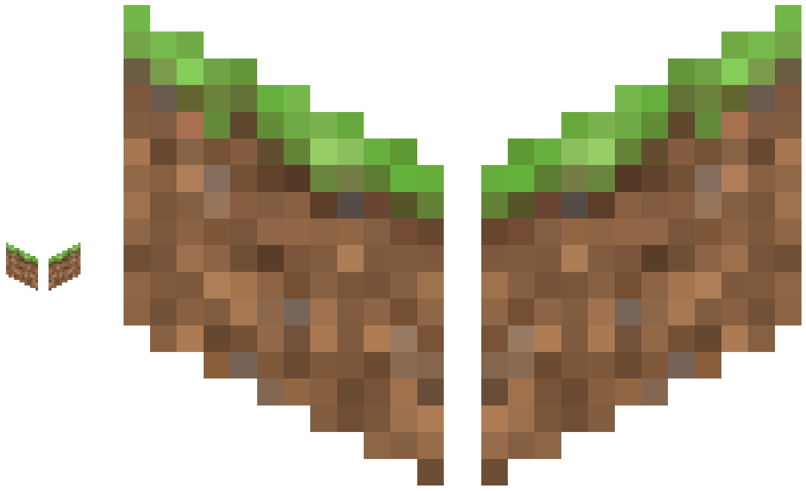
### Side Transformation

The texture square is transformed for the sides of the cube in the `textures.transform_image_side()` function. This is another affine transformation, but this time only two transformations are done: a re-size and a shear.

1. First the texture is re-sized to 12 by 12 pixels. This is half the width of 24 so it will have the correct width after the shear.
2. The 12 by 12 square is sheared by a factor of 1.5 in the Y direction, producing an image that is bounded by a 12 by 18 pixel square.



This image is simply flipped along the horizontal axis for the other visible side of the cube.

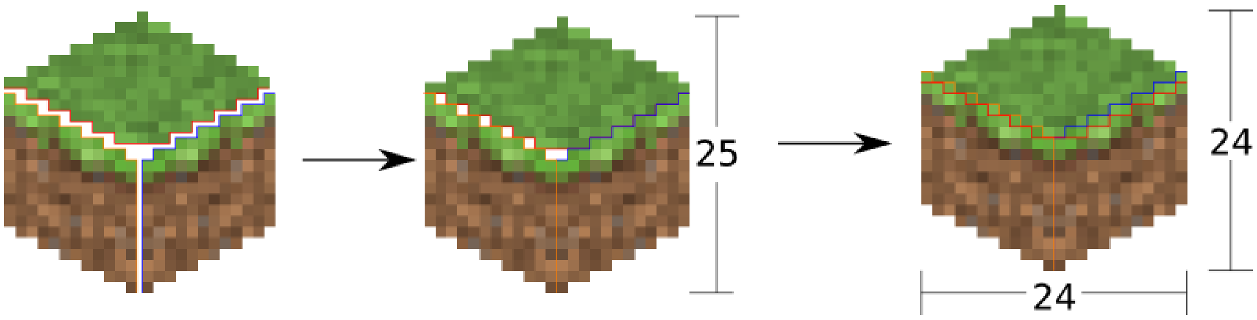


Again, shown on the left are the two sides of the block at actual size, the right is scaled with no interpolation by a factor of 10 to show the pixels.

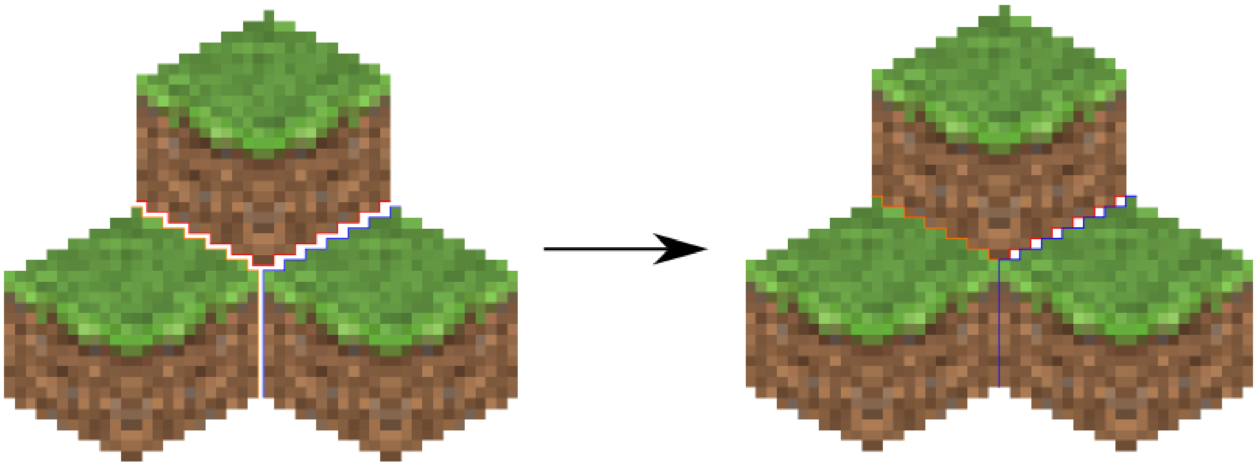
### An Entire Cube

These three images, the top and two sides, are pasted into a single 24 by 24 pixel image to get the cube sprite, as shown.

However, notice from the middle of the three images in the sequence below that the images as transformed don't fit together exactly. There is some overlap when put in the 24 by 24 box in which they must fit.

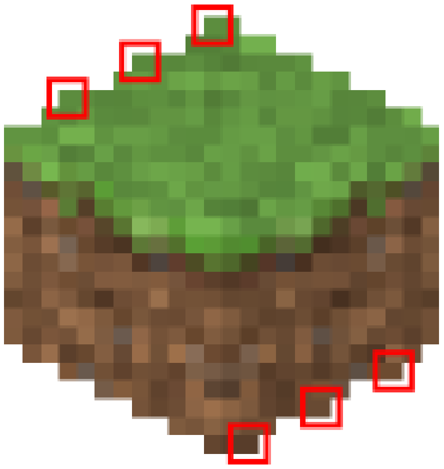


There is one more complication. The cubes don't tessellate perfectly. A six pixel gap is left between the lower-right border and upper-left border of blocks in this arrangement:



The solution is to manually touch up those 6 pixels. 3 pixels are added on the upper left of each cube, 3 on the lower right. Therefore, they all line up perfectly!

This is done at the end of `Textures.build_block()`



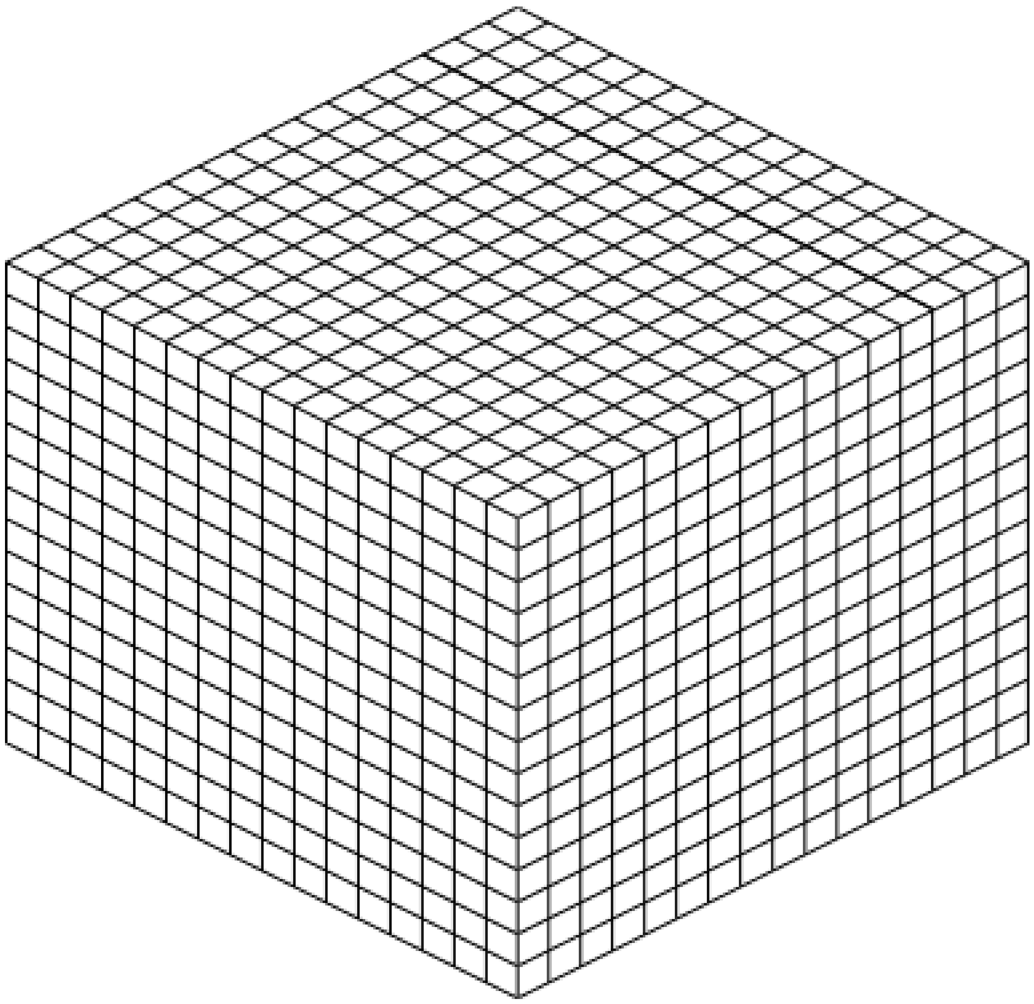
### 2.9.4 Chunk Rendering

With these cube sprites, we can draw them together to start constructing the world. The renderer renders a single chunk section (a 16 by 16 by 16 group of blocks) at a time.

This section of the design doc goes over how to draw the cube sprites together to draw an entire chunk section.

How big is a chunk section going to be? A chunk section is a cube of 16x16x16 blocks.

Rendered at the appropriate perspective, we'll have a cube made up of 4096 smaller cubes, like this:

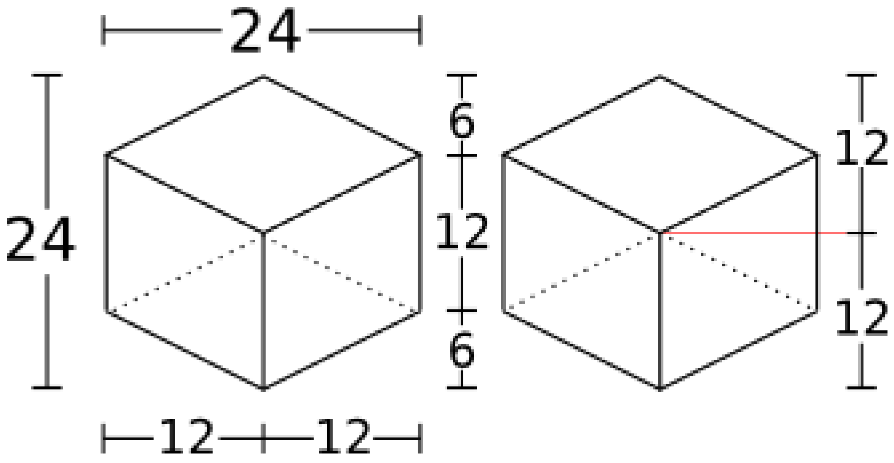


Each of those cubes shown is where one of the pre-rendered block sprites gets pasted; the entire thing is a chunk section. The renderer iterates over a chunk layer-at-a-time from bottom to top, drawing the sprites. The order is important so that the it gets drawn correctly. Obviously if a sprite in the back is pasted on the image after the sprites in the front are drawn, it will be drawn on top of everything instead of behind.

#### Block Positioning

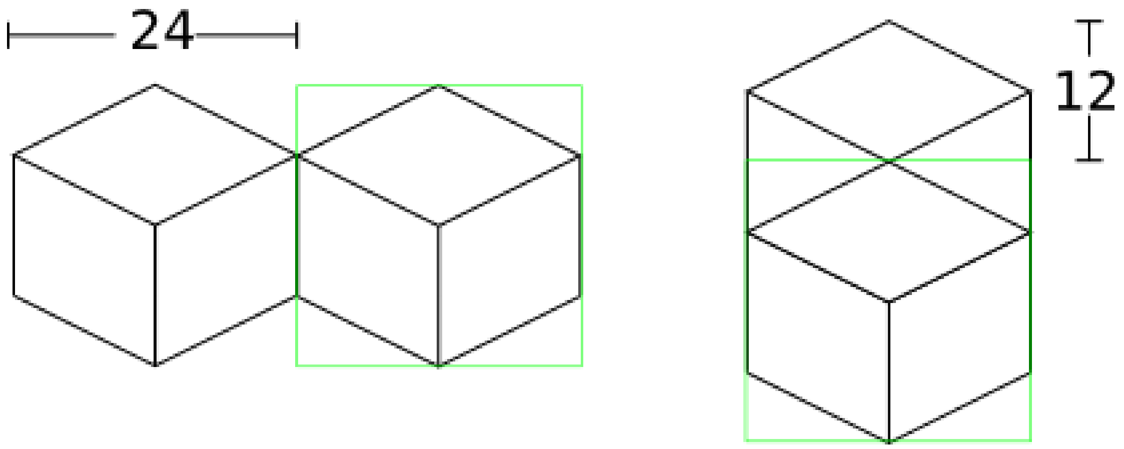
A single block is a 24 by 24 pixel image. Before we can construct a chunk section out of individual blocks, we must figure out how to position neighboring blocks.

First, to review, these are the measurements of a block sprite:

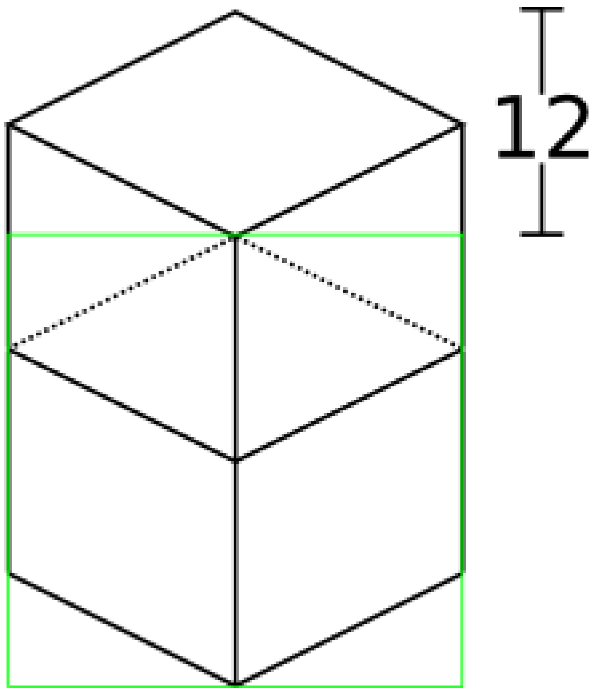


- The image is bounded by a 24 by 24 pixel square.
- The side vertical edges are 12 pixels high.
- The top (and bottom) face of the block takes 12 vertical pixels (and 24 horizontal pixels).
- The edges of the top and bottom of the block take up 6 vertical pixels and 12 horizontal pixels each.

Two blocks that are neighbors after projection to the image (diagonally neighboring in the world) have a horizontal offset of 24 pixels from each other, as shown below on the left. This is mostly trivial, since the images don't overlap at all. Two blocks in the same configuration but rotated 90 degrees have some overlap as shown on the right, and are only vertically offset by 12 pixels.

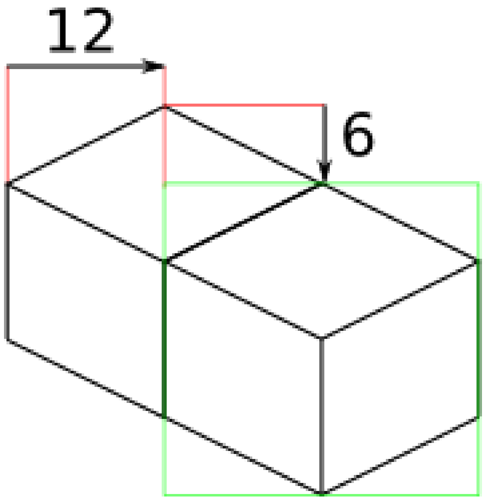


Now for something slightly less intuitive: two blocks that are stacked on top of each other in the world. One is rendered lower on the vertical axis of the image, but by how much?



Interestingly enough, due to the projection, this is exactly the same offset as the situation above for diagonally neighboring blocks. The block outlined in green is drawn 12 pixels below the other one. Only the order that the blocks are drawn is different.

And finally, what about blocks that are next to each other in the world — diagonally next to each other in the image?

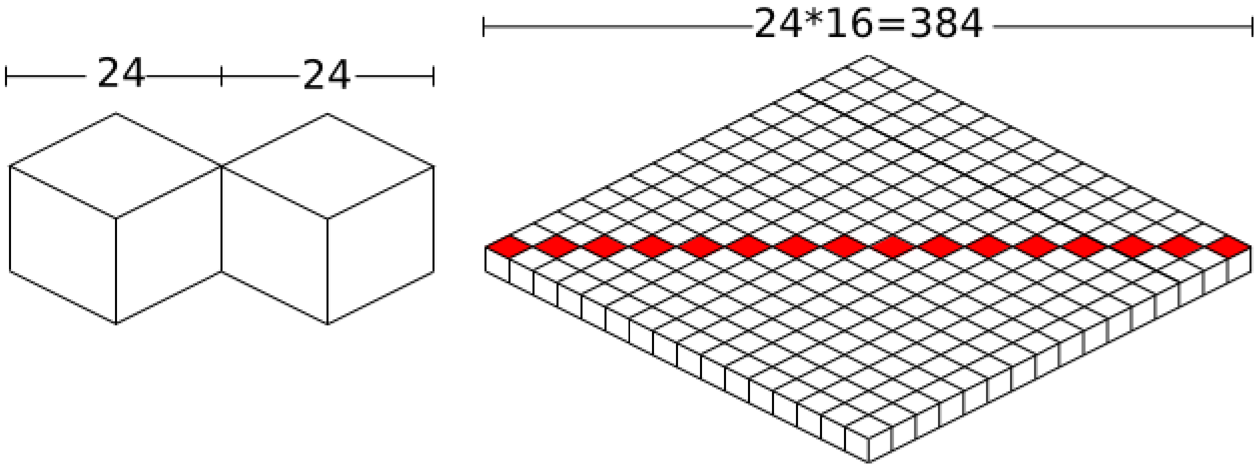


The block outlined in green is offset on the horizontal axis by half the block width, or 12 pixels. It is offset on the vertical axis by half the height of the block's top, or 6 pixels. For the other 3 directions this could go, the directions of the offsets are changed, but the amounts are the same.

## The size of a chunk

Now that we know how to place blocks relative to each other, we can begin to construct an entire chunk section.

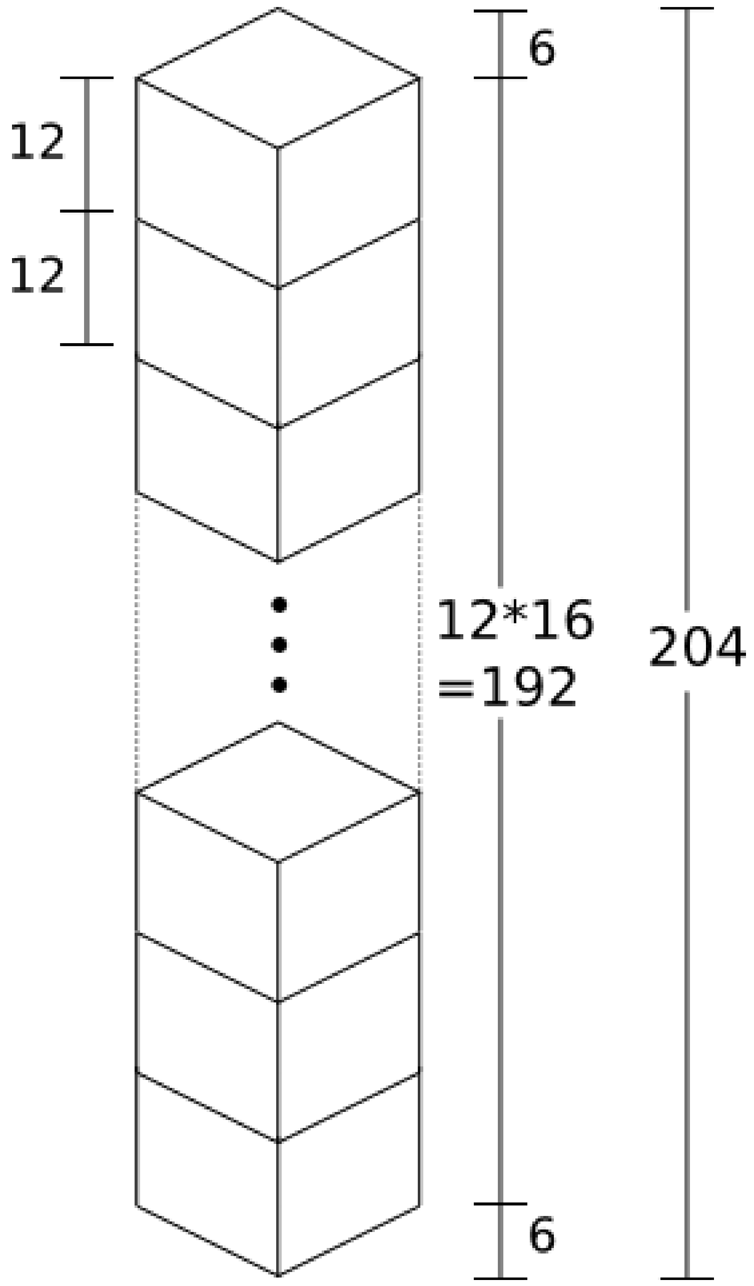
Since the block sprites are 24 by 24 pixels, and the diagonal of the 16 by 16 grid is 16 squares, the width of one rendered chunk section will be 384 pixels. Just considering the top layer of blocks within a section:



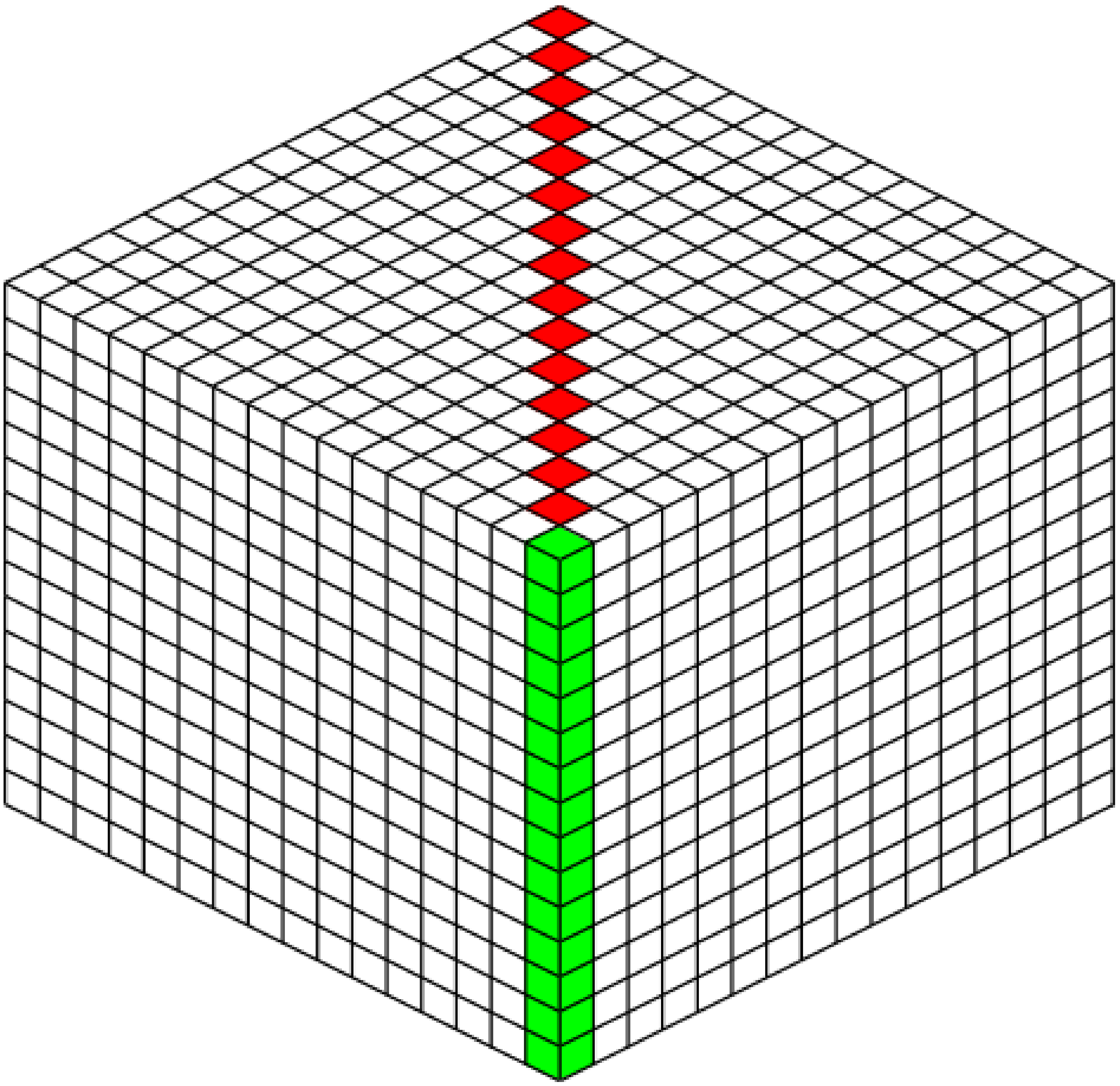
Since blocks next to each other in the same "diagonal row" are offset by 24 pixels, this is trivially calculated.

The height is a bit more tricky to calculate. Let's start by calculating the height of a single stack of 16 blocks.

The non-overlapping edge of each block sprite is 12 pixels high. Since there are 16 blocks in this stack, that's 192 pixels high. There are also 6 additional pixels at the top and bottom of the stack as shown, giving a total height of 204 pixels.



But that’s just for one column of blocks. What about the entire chunk section? Take a look at this diagram:

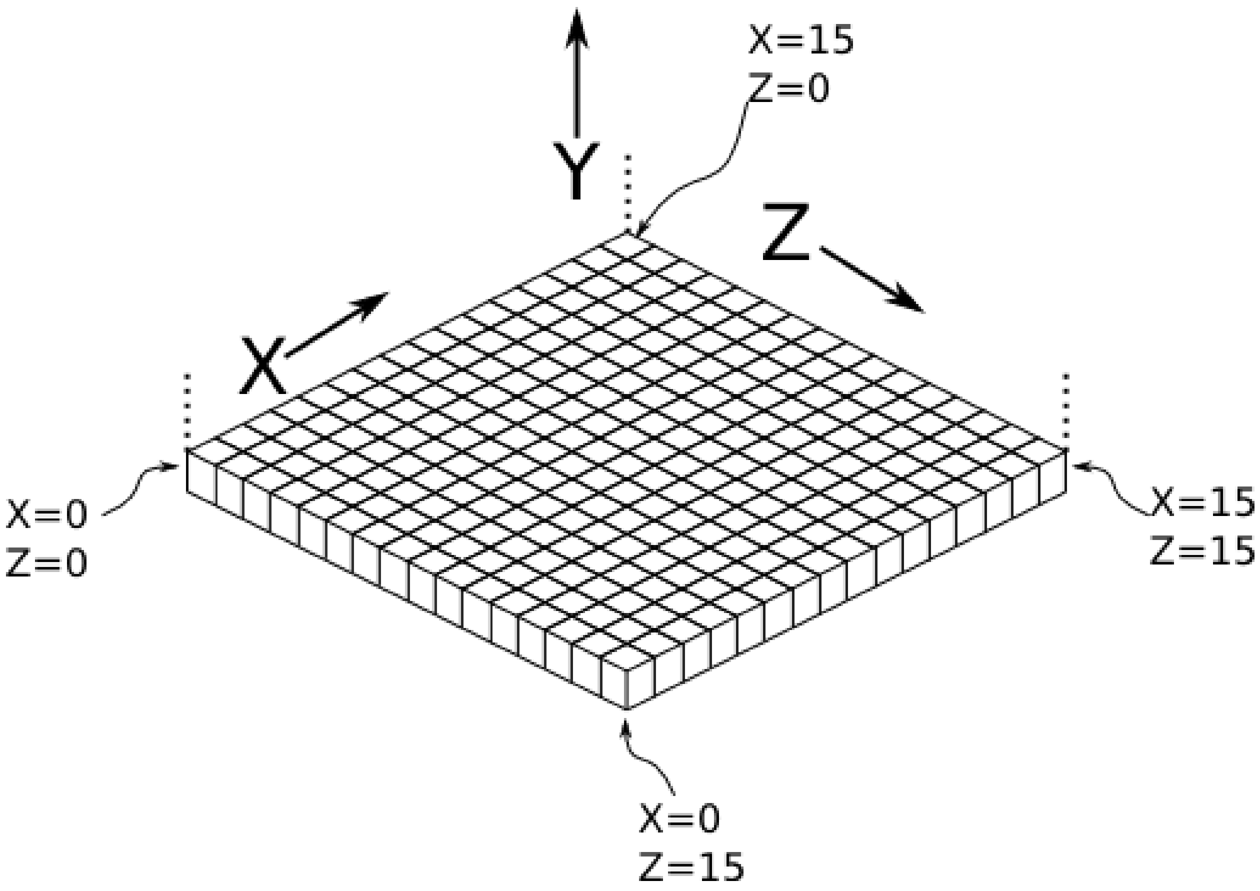


The green highlighted blocks are the stack we calculated just above and have a height of 204 pixels. The red highlighted blocks each take 12 pixels of vertical space on the image, and there are 15 of them. So 204 + 12*15 is 384 pixels.

So the total size of a chunk section in pixels is 384 wide by 384 tall.

### Assembling a Chunk

Now that we know how to place blocks, here's how they are arranged to form an entire chunk section. The coordinate system is arranged as shown, with the origin being at the left corner.



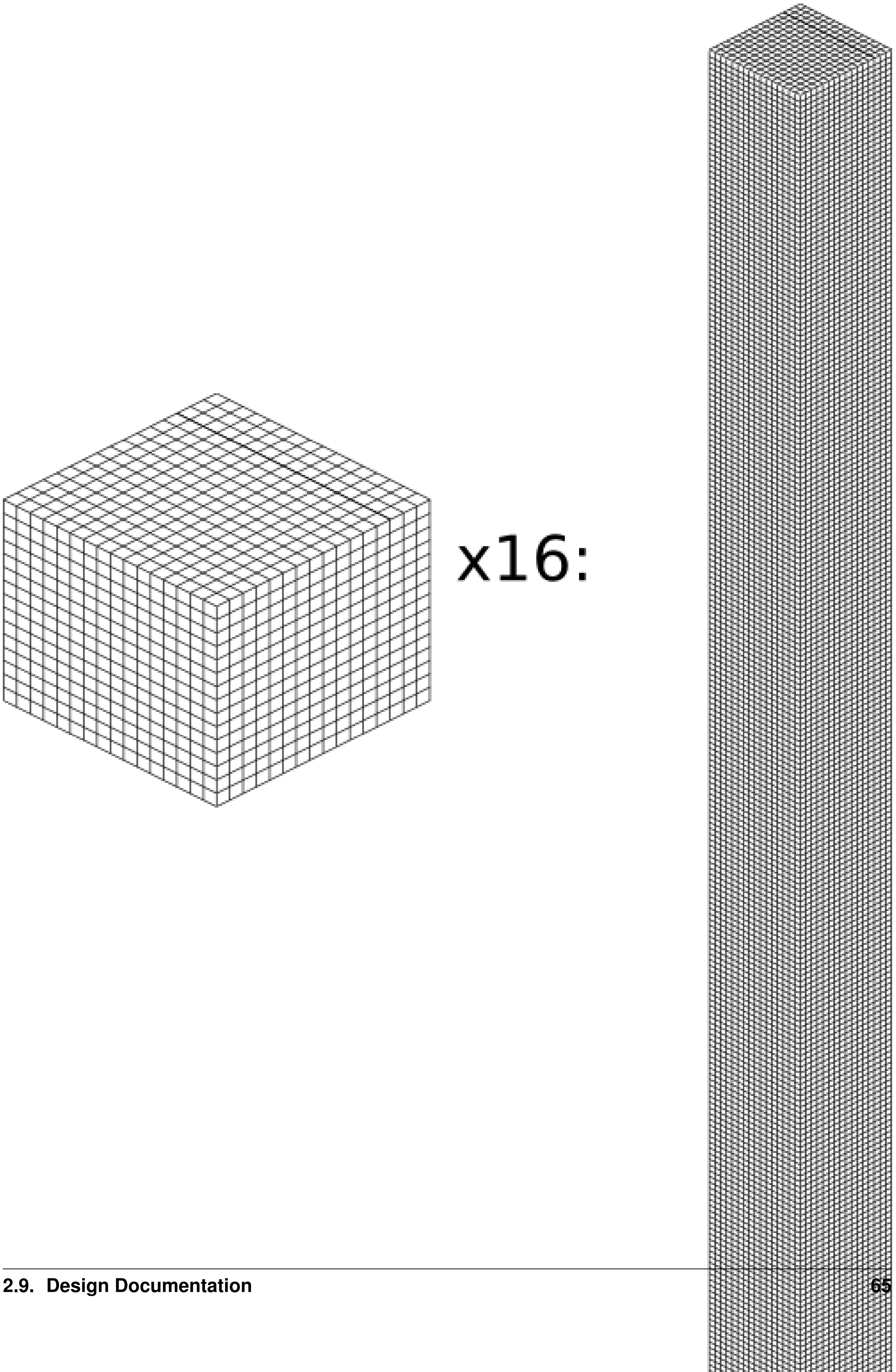
To ensure that block closer to the viewer are drawn on top while blocks that should be obstructed are drawn are hidden, the blocks are drawn one layer at a time from bottom to top (Y=0 to Y=15) and from back to front.

From the data file on disk, block information in a chunk is a three-dimensional array of bytes, each representing a block id. The process of assembling a chunk is simply a matter of iterating over this array, reading the blockid values, looking up the appropriate sprite, and pasting it on the chunk image at the appropriate location.

### 2.9.5 Chunk Placement

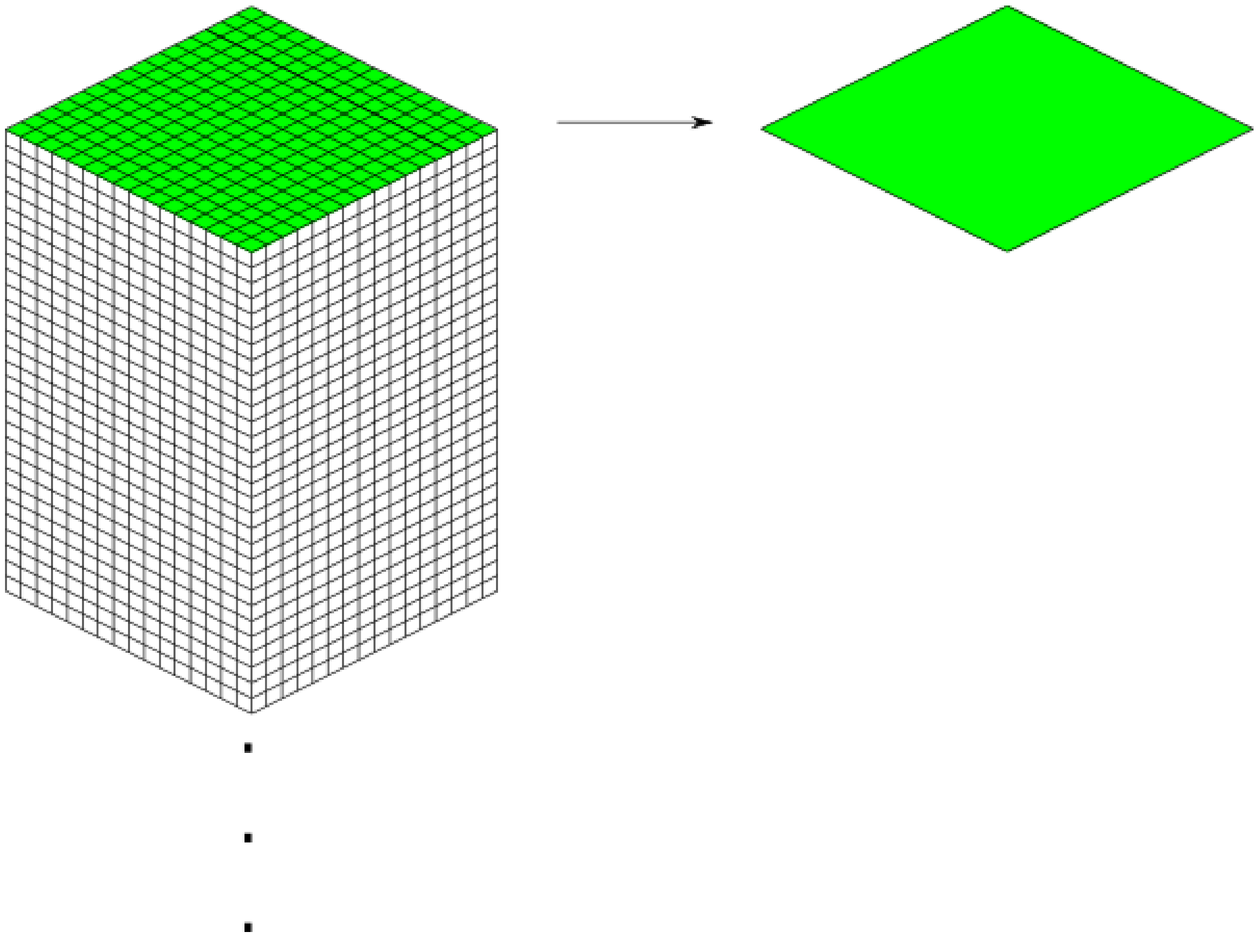
Now that we know how to draw a single chunk, let's move on to how to place chunks relative to each other.

Before we get started, let's take a moment to remember that one chunk section is only 1/16th of a chunk:


x16:

A chunk is 16 chunk sections stacked together.

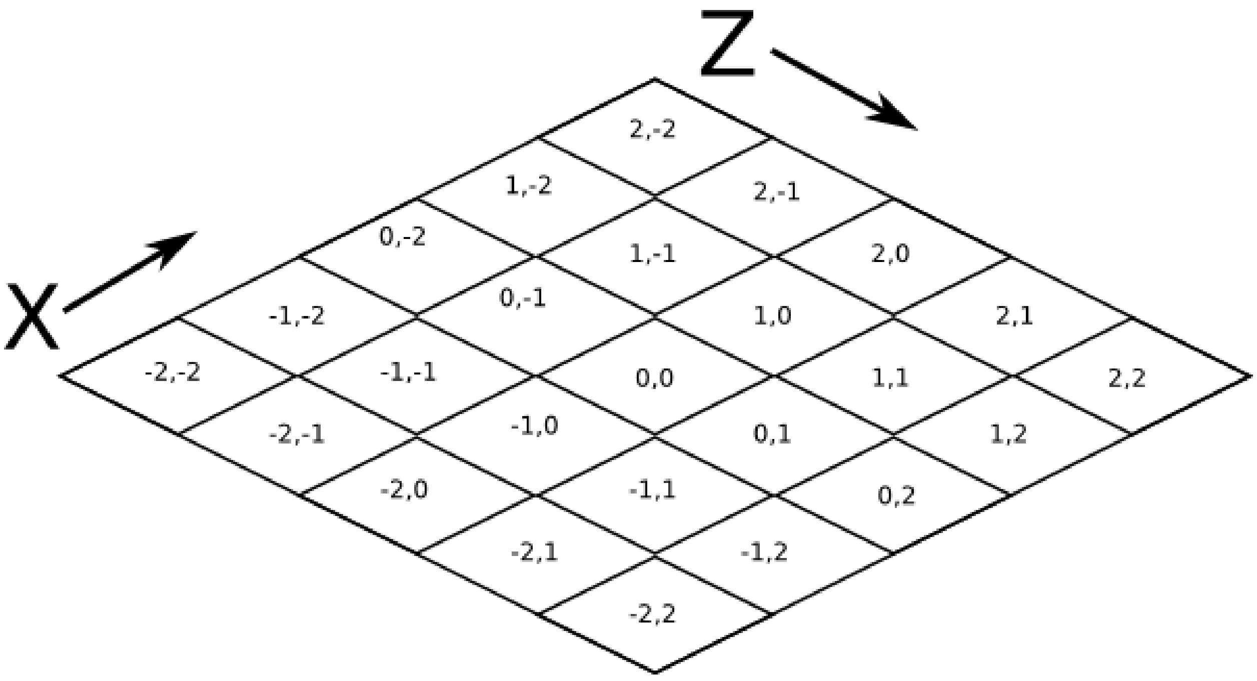
Since this is pretty tall, the diagrams in this section are simplified to only show the *top face* of a chunk, as shown in green here:



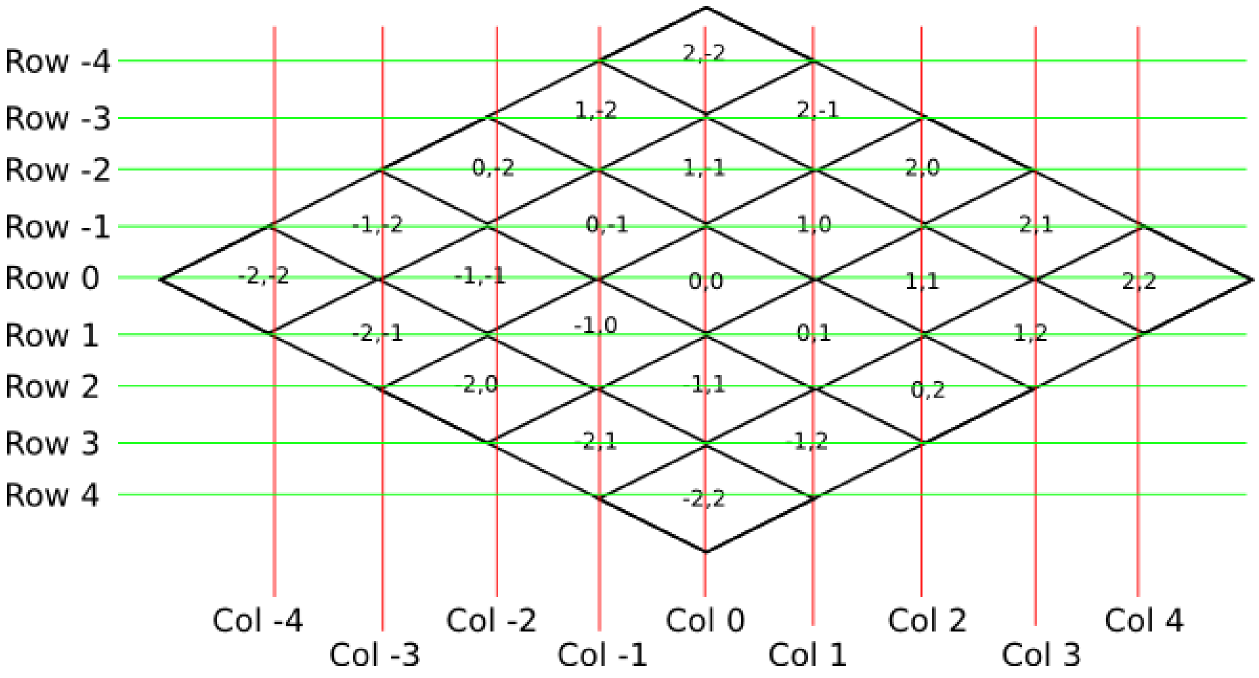
This makes it easier and less cumbersome to describe how to place chunks together on a tile. Just remember that chunks are actually very tall and extend down far beyond the drawn diamonds in these diagrams.

### Chunk Addressing

Chunks in Minecraft have an X,Z address, with the origin at 0,0 and extending to positive and negative infinity on both axes (Recall from the introduction that chunk addresses are simply the block addresses divided by 16). Since we're going to render at a diagonal perspective, it is convenient to perform a change of coordinate system. For that, we translate X,Z coordinates into column,row coordinates. Consider this grid showing 25 chunks around the origin. They are labeled with their X,Z chunk addresses.



Now, we want to transform each chunk to a row/column address as shown here:



So the chunk at address 0,0 would be at col 0, row 0; while the chunk at address 1,1 would be at col 2, row 0. The intersection of the red and green lines addresses the chunk in col,row format.

**Note:** As a consequence of this addressing scheme, there is no chunk at e.g. column 1 row 0. There are some col,row addresses that lie between chunks, and therefore do not correspond to a chunk. (as can be seen where the red/green lines intersect at a chunk boundary instead of the middle of a chunk).

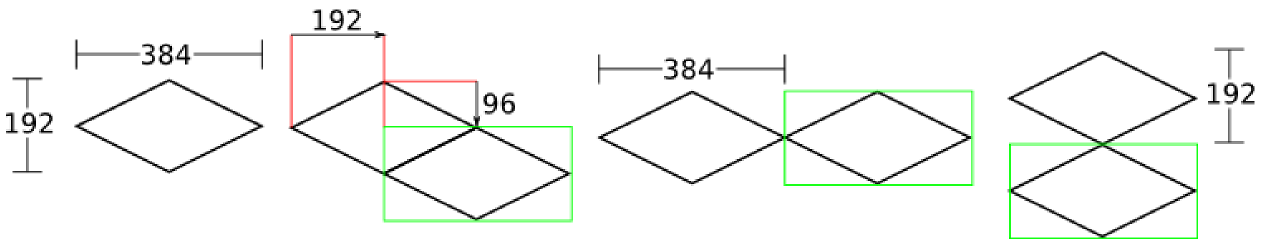
So how does one translate between them? It turns out that a chunk's column address is simply the sum of the X and the Z coordinate, while the row is the difference. Try it!

```
col = X + Z
row = Z - X

X = (col - row) / 2
Z = (col + row) / 2
```

### Chunk Positioning

This section will seem very familiar to the block positioning. In fact, it is exactly the same but with different numbers (because blocks and chunk sections have the exact same proportions), so let's speed through this.

A chunk's top face is 384 pixels wide by 192 pixels tall. They therefore have these offsets from their neighbors:



## 2.9.6 Tile Rendering

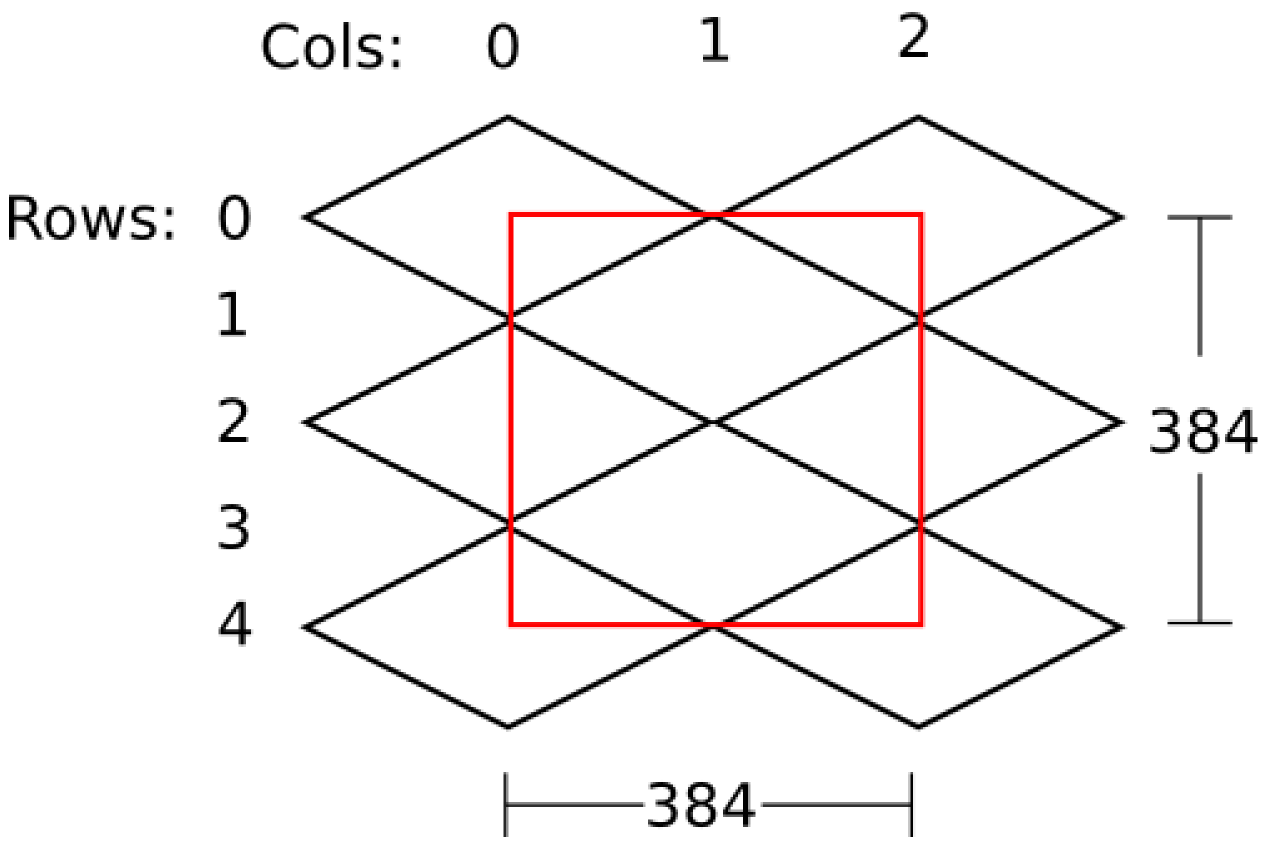
Now that we know how to translate chunk coordinates to col/row coordinates, and know how to calculate the offset from the origin on the final image, we could easily draw the chunks in one large image. However, for large worlds, that would quickly become too much data to handle at once. (Early versions of the Overviewer did this, but the large, unwieldy images quickly motivated the development of rendering to individual tiles).

Hence choosing a technology like Google Maps or Leaflet, which draws small tiles together to make it look like one large image, lets rendering even the largest worlds possible. The Overviewer can draw each tile separately and not have to load the entire map into memory at once. The next sections describe how to determine which chunks to render in which tiles, and how to reason about tile  chunk mappings.

### Tile Layout

Instead of rendering to one large image, chunks are rendered to small tiles. Only a handful of chunks need to be rendered into each tile. The downside is that chunks must be rendered multiple times for each tile they appear in, but the upside is that arbitrarily sized maps can be viewed.

The Overviewer uses a tile size of 384 by 384 pixels. This is the same as the size of a chunk section and is no coincidence. Just considering the top face of a chunk, the 8 chunks directly below it get rendered into a tile in this configuration:



---

**Note:** Don't forget that chunks are tall, so many more than 8 chunks get rendered into this tile. If you think about it, chunks from the rows *above* the ones in that diagram may have blocks that fall into this tile, since the diamonds in the diagram correspond to the *tops* of the chunks, and chunks extend *down*.

---

**Note:** This is an important diagram and we'll be coming back to it. Make sure it makes sense. As a side note, if anything in this document *doesn't* make sense, please let us know in IRC or by filing an issue. I want these docs to be as clear as possible!

---

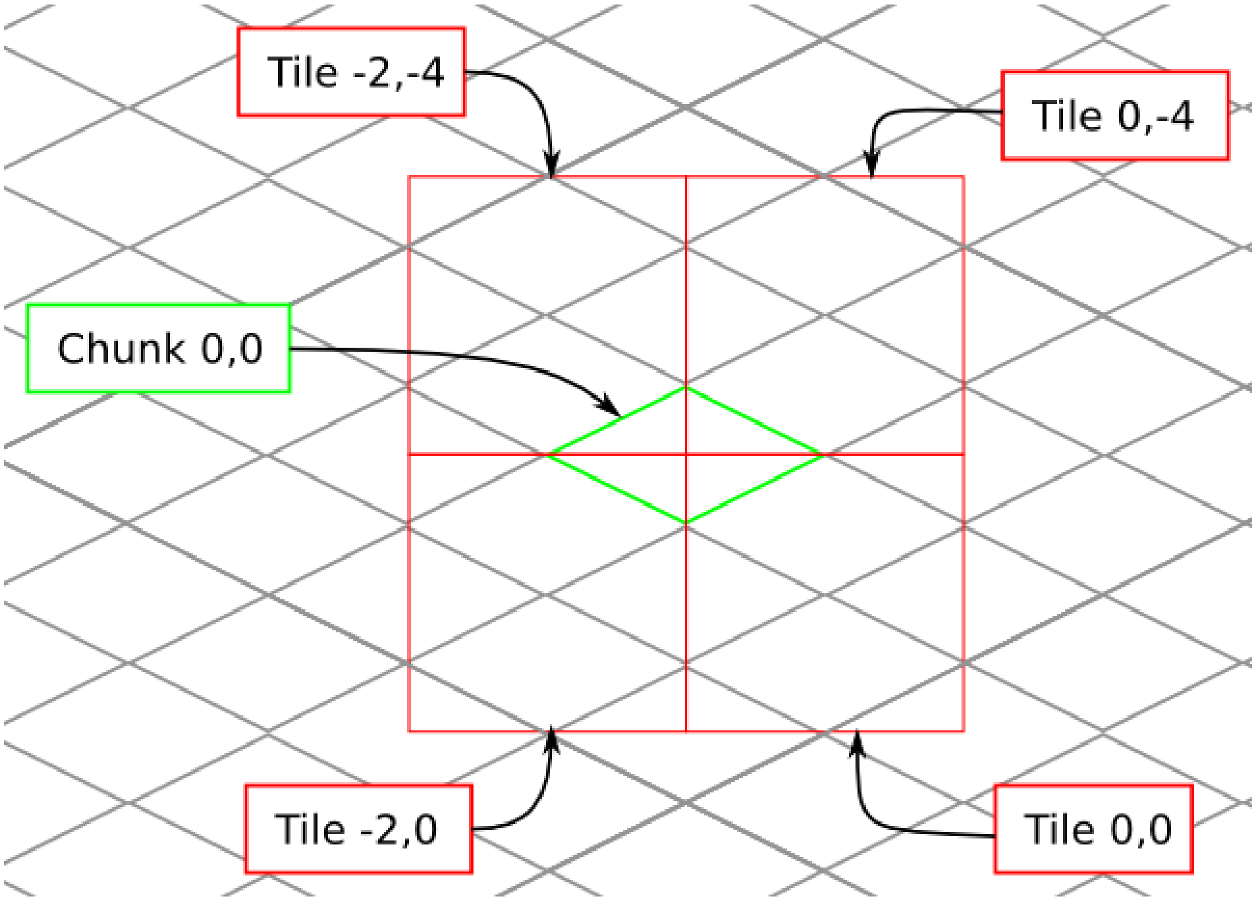
So the overall strategy is to convert all chunks into diagonal col,row coordinates, then for each tile decide which chunks belong in it, then render them in the appropriate place on the tile.

The rendering routines are actually passed a range of chunks to render, e.g. rows 4-6, cols 20-24. The lower bound col,row chunk given in the range is rendered at position 0,0 in the diagram above. That is, at offset -192,-96 pixels.

The rendering routines takes the given range of columns and rows, converts it back into chunk coordinates, and renders the given 8 chunks plus all chunks from the 16 rows above the given range (see the note below). The chunks are positioned correctly with the above positioning rules, so any chunks that are out of the bounds get rendered off the tile and don't affect the final image. (There is therefore no penalty for rendering out-of-bounds chunks for a tile except increased processing)

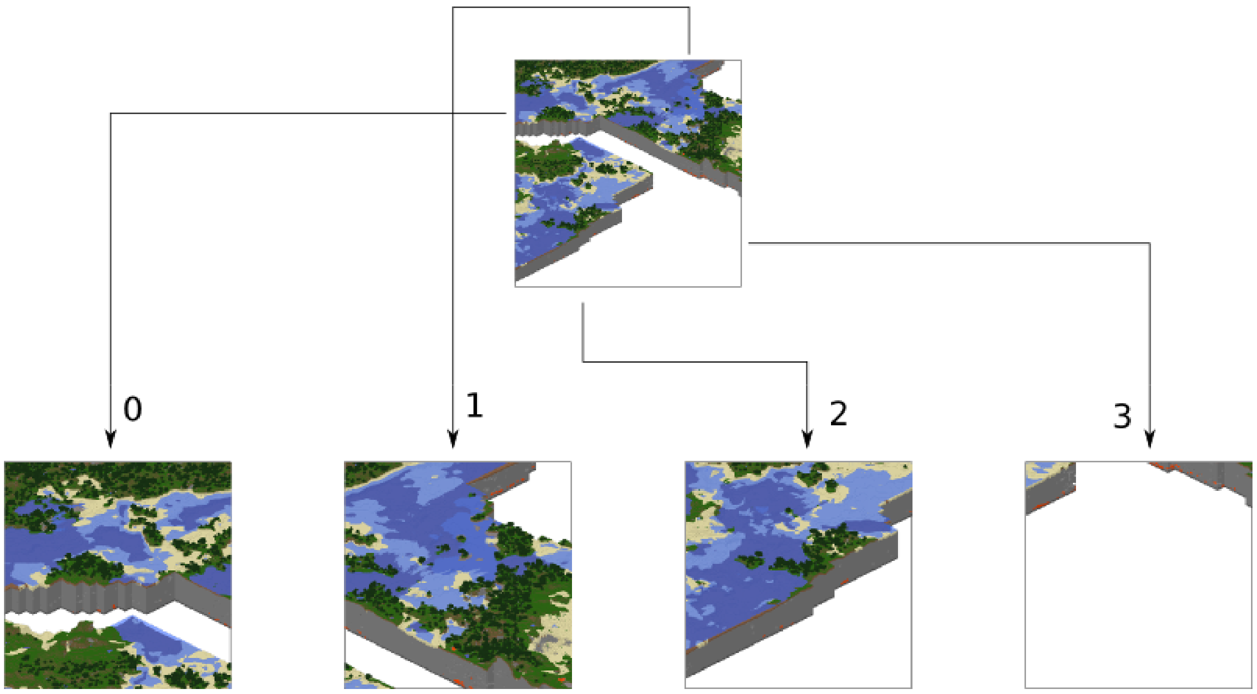
Since every other column of chunks is half-way in two tiles, they must be rendered twice. Each neighboring tile is therefore only 2 columns over, not 3 as one may suspect at first. Same goes for the rows: The next tile down is 4 rows down, not 5.

To further illustrate this point, here are four tiles arranged on the grid of chunks. Notice how the tiles are addressed by the col,row of the chunk in the upper-left corner. Also notice how neighboring tiles are 2 columns apart but 4 rows apart.



### 2.9.7 Quadtrees

Tiles are rendered and stored in a quadtree on disk. Each node is a tile of the world, and each node has four children representing a zoomed-in tile of the four quadrants.



The tree is generated from the bottom-up. The highest zoom level is rendered directly from the chunks and the blocks, then four of those rendered tiles are shrunk and concatenated to get the next zoom level. The tree is built up in this way until the entire world is compressed down to a single tile.

We've already seen how tiles can be identified by the column,row range of the chunks that make up the tile. More precisely, since tiles are always the same size, the chunk that goes in the tile's 0,0 col,row slot identifies the tile.

Now, tiles are also identified by their path in the quadtree. For example, `3/0/0/1/1/2.png` refers to the tile starting at the base, under the third quadrant, then the 0th quadrant, then the 0th, and so fourth.

### Quadtree Size

The size of the quadtree must be known before it's generated, that way the code knows where to save the images. This is easily calculated from a few realizations. Each depth in the quadtree doubles the number of tiles in each dimension, or, quadruples the total tiles. While there is only one tile at level 0, there are four at level 1, 16 at level 2, and 4^n at level n.

To find how deep the quadtree must be, we look at the size of the world. First find the maximum and minimum row and column of the chunks. Just looking at columns, let's say the maximum column is 82 and the minimum column is -136. A zoom level of 6 will be 2^6 tile across and 2^6 tiles high at the highest level.

Since horizontally tiles are two chunks wide, multiply 2^6 by 2 to get the total diameter of this map in chunks: 2*2^6. Is this wide enough for our map?

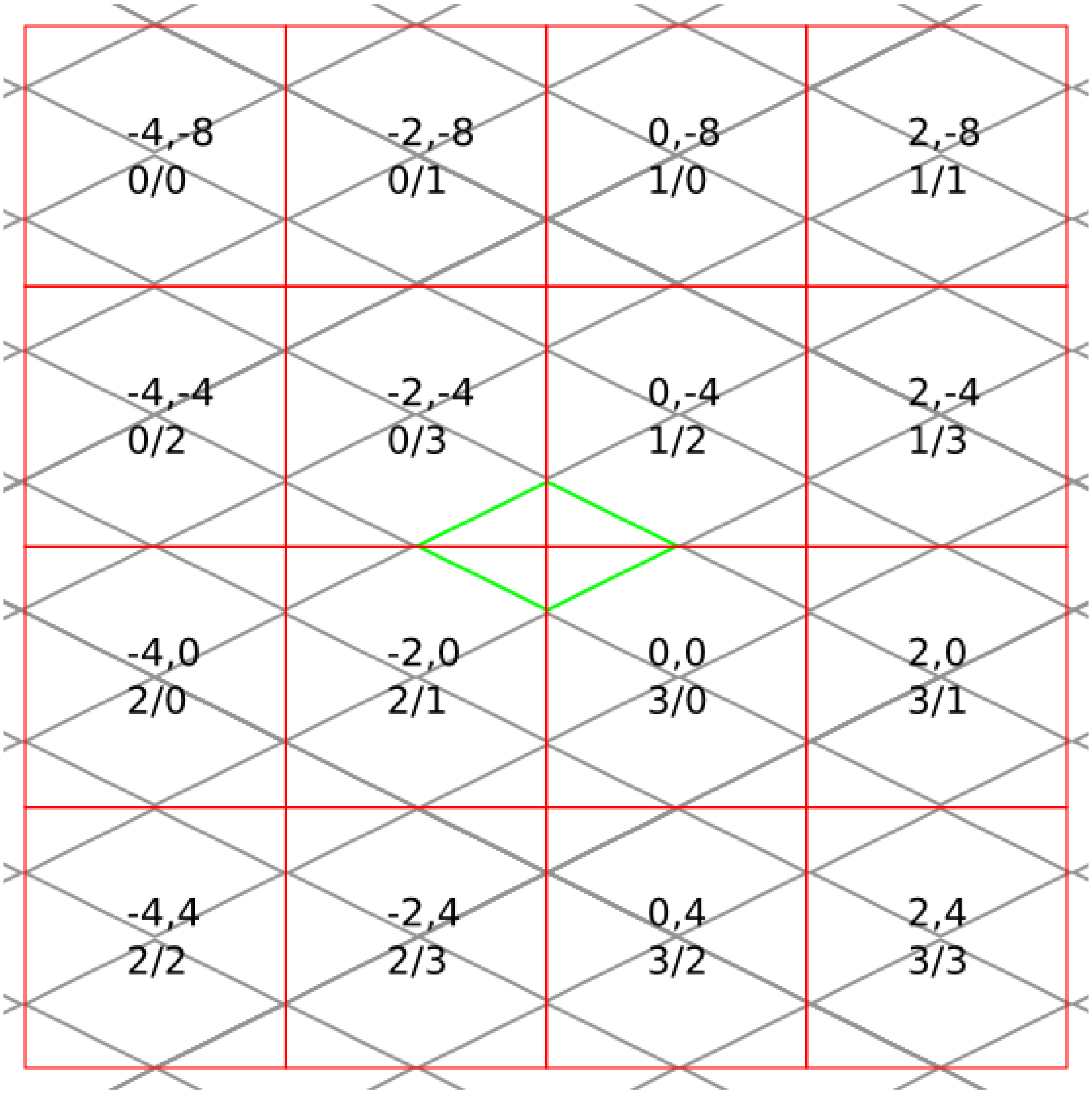
It turns out it isn't (2*2^6=128, 136+82=218). A zoom level of 7 is 2^7 tiles across, or 2*2^7 chunks across. This turns out is wide enough (2*2^7 = 256), however, Overviewer maps are always centered at point 0,0 in the world. This is so tiles will always line up no mater how the map may expand in the future.

So zoom level 7 is *not* enough because, while the chunk diameter is wide enough, it only extends half that far from the origin. The chunk *radius* is 2^7 (half the diameter) and 2^7=128 is not wide enough for the minimum column at absolute position 136.

So this example requires zoom level 8 (at least in the horizontal direction. The vertical direction must also be checked).

### Quadtree Paths

To illustrate the relationship between tile col,row addresses and their path, consider these 16 tiles from a depth 2 quadtree:



The top address in each tile is the col,row address, where the chunk outlined in green in the center is at 0,0. The lower address in each tile is the path. The first number indicates which quadrant the tile is in overall, and the second is which quadrant within the first one.

#### get_range_by_path

### 2.9.8 Reading the Data Files

### 2.9.9 Image Composition

### 2.9.10 Multiprocessing

### 2.9.11 Caching

### 2.9.12 Lighting

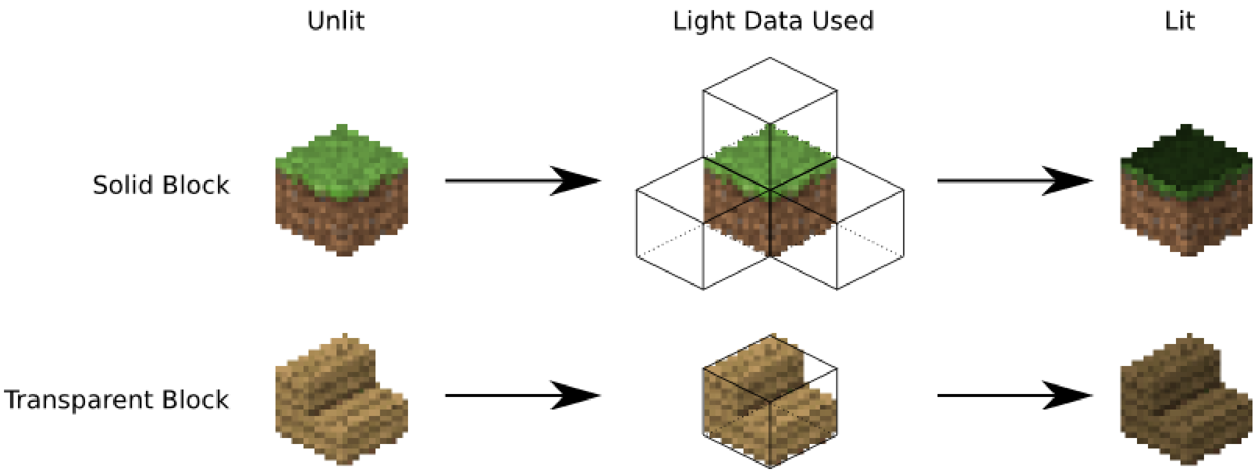
Minecraft stores precomputed lighting information in the chunk files themselves, so rendering shadows on the map is a simple matter of interpreting this data, then adding a few extra steps to the render process. These few extra steps may be found in `rendermode-lighting.c` or `rendermode-smooth-lighting.c`, depending on the exact method used.

Each chunk contains two lighting arrays, each of which contains one value between 0 and 15 for each block. These two arrays are the BlockLight array, containing light received from other blocks, and the SkyLight array, containing light received from the sky. Storing these two seperately makes it easier to switch between daytime and nighttime. To turn these two values into one value between 0 and 1 representing how much light there is in a block, we use the following equation (where $l_b$ and $l_s$ are the block light and sky light values, respectively):

$$c = 0.8^{15-\min(l_b,l_s)}$$

For night lighting, the sky light values are shifted down by 11 before this lighting coefficient is calculated.

Each block of light data applies to all the block faces that touch it. So, each solid block doesn't receive lighting from the block it's in, but from the three blocks it touches above, to the left, and to the right. For transparent blocks with potentially strange shapes, lighting is approximated by using the local block lighting on the entire image.

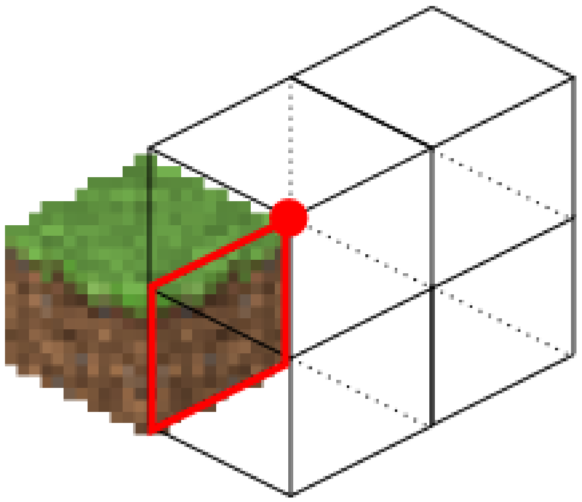


For some blocks, notably half-steps and stairs, Minecraft doesn't generate valid lighting data in the local block like it does for all other transparent blocks. In these cases, the lighting data is estimated by averaging data from nearby blocks. This is not an ideal solution, but it produces acceptable results in almost all cases.

### Smooth Lighting

In the smooth-lighting rendermode, solid blocks are lit per-vertex instead of per-face. This is done by covering all three faces with a quadralateral where each corner has a lighting value associated with it. These lighting values are then smoothly interpolated across the entire face.

To calculate these values on each corner, we look at lighting data in the 8 blocks surrounding the corner, and ignore the 4 blocks behind the face the corner belongs to. We then calculate the lighting coefficient for all 4 remaining blocks as normal, and average them to obtain the coefficient for the corner. This is repeated for all 4 corners on a given face, and for all visible faces.



The ambient occlusion effect so strongly associated with smooth lighting in-game is a side effect of this method. Since solid blocks have both light values set to 0, the lighting coefficient is very close to 0. For vertices in corners, at least 1 (or more) of the 4 averaged lighting values is therefore 0, dragging the average down, and creating the "dark corners" effect.

## 2.9.13 Cave Mode

# CHAPTER 3

# Features

- Renders high resolution images of your world, lets you "deep zoom" and see details!
- Gloriously awesome smooth lighting is here! (*rendermode* name is `smooth_lighting`)
- Customizable textures! Pulls textures straight from your installed texture pack!
- Choose from four rendering angles.
- Generates a Leaflet powered map!
- Runs on Linux, Windows, and Mac platforms!
- Renders efficiently in parallel, using as many simultaneous processes as you want!
- *Only* requires: Python, Numpy, and PIL (all of which are included in the Windows download!)
- Utilizes caching to speed up subsequent renderings of your world. Only parts that need re-rendering are re-rendered.
- Throw the output directory up on a web server to share your Minecraft world with the internet!
- Run The Overviewer from a command line or on a cron schedule for constantly updated maps! Run it for your Minecraft server world to provide your users with a detailed map!
- Supports Nether and The End dimensions!
- Built-in support for Biomes!

## 3.1 What The Overviewer is not

Full disclosure disclaimers of what The Overviewer is *not*.

- It does not run fast. Because of the high level of detail, initial renders of a world can take some time. Expect minutes for medium worlds, hours for large to huge worlds. Subsequent renders are *much* faster due to the caching.

  Also note that speed is improving all the time. We continually make efficiency improvements to The Overviewer. Besides, for the level of detail provided, our users consider it worth the time!

- The Overviewer is not targeted at end users. We mainly see Overviewer fitting in best with server operators, rendering their server's map for all users to view.

  You are welcome to use The Overviewer for your single player worlds, and it will work just fine. However, since the only interface is currently command line based, you will need to know a bit about the command line in order to operate The Overviewer.
- The Overviewer does not support Bedrock/Win10/Portable Edition worlds.
- The Overviewer is not a potato.

# CHAPTER 4

# Requirements

This is a quick list of what's required to run The Overviewer. It runs on Windows, Mac, and Linux as long as you have these software packages installed:

- Python 3.4 or above (we are no longer compatible with Python 2.x)
- PIL (Python Imaging Library) or Pillow
- Numpy
- Either a Minecraft Client installed or a textures/ folder for the textures (possibly from a texturepack)

The first three are included in the Windows download. Also, there are additional requirements for compiling it (like a compiler). More details are available in either the *Building* or *Installing* pages.

CHAPTER 5

# Getting Started

The Overviewer works with Linux, Mac, and Windows! We provide Windows and Debian built executables for your convenience. Find them as well as the full sources on our Github Homepage.

**If you are running Windows, Debian, or Ubuntu and would like the pre-built packages and don't want to have to compile anything yourself**, head to the *Installing* page.

**Running Windows and not familiar with the command line?** Head to the *Windows Newbie Guide* page.

**If you would like to build the Overviewer from source yourself (it's not that bad)**, head to the *Building* page.

**For all other platforms** you will need to build it yourself. *Building the Overviewer from Source*.

**After you have The Overviewer built/installed** see *Running the Overviewer* and *The Configuration File*.

# CHAPTER 6

# Help

**IF YOU NEED HELP COMPILING OR RUNNING THE OVERVIEWER** feel free to chat with us live in IRC: #overviewer on Libera.Chat. There's usually someone on there that can help you out. Not familiar with IRC? Use the web client. (If there's no immediate response, wait around or try a different time of day; we have to sleep sometime)

Also check our *Frequently Asked Questions* page.

If you think you've found a bug or other issue, file an issue on our Issue Tracker. Filing or commenting on an issue sends a notice to our IRC channel, so the response time is often very good!

# CHAPTER 7

# Indices and tables

- genindex
- modindex
- search

# Index