Bravo Documentation

Release 2.0

Corbin Simpson, Derrick Dymock, & Justin Noah

September 08, 2014

Contents

1 A high-level introduction **3**
- 1.1 Similar and different . . . 3
- 1.2 Current state . . . 3
- 1.3 Project licensing . . . 3
- 1.4 Q & A . . . 3
- 1.5 Credits . . . 4

2 Features **5**
- 2.1 Standard features . . . 5
- 2.2 Extended features . . . 7

3 Differences vs. vanilla Minecraft Server **9**
- 3.1 Responsiveness . . . 9
- 3.2 Chunks . . . 9
- 3.3 Inventory . . . 9
- 3.4 Minecarts . . . 10

4 Philosophy **11**
- 4.1 Design Decisions . . . 11
- 4.2 Versioning . . . 13

5 Administrator Topics **15**
- 5.1 How to administer Bravo . . . 15
- 5.2 Plugins . . . 17
- 5.3 Troubleshooting . . . 20
- 5.4 Web Service . . . 21

6 Developer Topics **23**
- 6.1 Extending Bravo . . . 23
- 6.2 Noise . . . 24
- 6.3 Core . . . 25
- 6.4 Auxiliary . . . 52
- 6.5 Tools . . . 58

7 Indices and tables **61**

Python Module Index **63**

Bravo is an elegant, speedy, and extensible implementation of the Minecraft Alpha/Beta protocol. Only the server side is implemented. The following introductory topics provide a better look at the project, its goals, and current capabilities.

CHAPTER 1

A high-level introduction

Bravo is an open source, reverse-engineered implementation of Minecraft's server application. Two of the major building blocks are Python and Twisted, but you need not be familiar with either to run, administer, and play on a Bravo-based server.

1.1 Similar and different

While one of the goals of Bravo is to be roughly on par with the standard, "Notchian" Minecraft server, Bravo does change and improve things for the better, where appropriate. See *Differences vs. vanilla Minecraft Server* for more details.

Some of the more positive hilights include:

- More responsiveness with higher populations.
- Much less memory and bandwidth consumption.
- Better inventory system that avoids some bugs found in the standard server.

1.2 Current state

Bravo is currently in heavy development. While it is probably safe to run creative games, we lack some elements needed for Survival-Multiplayer. Take a look at *Features* to get an idea of where we currently stand.

We encourage the curious to investigate for themselves, and post any bugs, questions, or ideas you may have to our issue tracker.

1.3 Project licensing

Bravo is MIT/X11-licensed. A copy of the license is included in the `LICENSE` file in the repository or distribution. This extremely permissive license gives you all of the flexibility you could ever want.

1.4 Q & A

Why are you doing this? What's wrong with the official Alpha/Beta server?

Plenty. The biggest architectural mistake is the choice of dozens of threads instead of NIO and an asynchronous event-driven model, but there are other problems as well. Additionally, the offical server development team has recently moved to remove all other servers as options for people wishing to deploy servers. We don't approve of that.

Are you implying that the official Alpha server is bad?

Yes. As previous versions of this FAQ have stated, Notch is a cool guy, but the official server is bad.

Are you going to make an open-source client? That would be awesome!

The server is free, but the client is not. Accordingly, we are not pursuing an open-source client at this time. If you want to play Alpha, you should pay for it. There's already enough Minecraft piracy going on; we don't feel like being part of the problem. That said, Bravo's packet parser and networking tools could be used in a client; the license permits it, after all.

Where did the docs go?

We contribute to the Minecraft Collective's wiki at http://mc.kev009.com/wiki/ now, since it allows us to share data faster. All general Minecraft data goes to that wiki. Bravo-specific docs are shipped in ReST form, and a processed Sphinx version is available online at http://bravo.readthedocs.org/.

Why did you make design decision <X>?

There's an entire page dedicated to this in the documentation. Look at docs/philosophy.rst or *Philosophy*.

It doesn't install? Okay, maybe it installed, but I'm having issues!

On Freenode IRC (irc.freenode.net), #bravoserver is dedicated to Bravo development and assistance, and #mcdevs is a more general channel for all custom Minecraft development. You can generally get help from those channels. If you think you have found a bug, you can directly report it on the Github issue tracker as well.

Please, please, please read the installation instructions in the README first, as well as the comments in bravo.ini.example. I did not type them out so that they could be ignored. :3

1.5 Credits

Who are you guys, anyway?

Corbin Simpson (MostAwesomeDude/simpson) is the main coder. Derrick Dymock (Ac-town) is the visionary and provider of network traffic dumps. Ben Kero and Mark Harris are the reluctant testers and bug-reporters. The Minecraft Coalition has been an invaluable forum for discussion.

CHAPTER 2

Features

Bravo's extensible design means that there are many different plugins and features. Since most servers do not have an extensive or exhaustive list of the various plugins that they include, one is provided here for Bravo.

2.1 Standard features

These features are found in official, Mojang-sponsored, unmodified servers.

2.1.1 Console

Bravo provides a small, plain console suitable for piping input and output, as well as interactive sessions.

2.1.2 Login

Bravo supports the two login methods supported by the Mojang-sponsored client: offline authentication and online authentication.

2.1.3 Geometry

Bravo understands how to manipulate and transfer geometry. In addition, Bravo can read and write the Alpha NBT and Beta MCR disk formats.

2.1.4 Time

Bravo fully implements the in-game day and night. Bravo's days are exactly 20 minutes long.

2.1.5 Entities

Bravo understands the concept of entities, and is able to track the following kinds of entities:

- Mobs
- Paintings
- Pickups

- Players
- Tiles

Mobs

Bravo understands the following mobs:

- Chickens/ducks (“Chucks”)
- Cows
- Creepers
- Ghasts
- Giant zombies
- Pigs
- Sheep
- Skeletons
- Slimes
- Spiders
- Squids
- Wolves
- Zombie pigmen
- Zombies

Tiles

Bravo understands the following tiles:

- Chests
- Furnaces
- Mob spawners
- Music blocks
- Signs

2.1.6 Inventory

Bravo provides server-side inventory handling.

2.1.7 Physics

Bravo simulates physics, including the behaviors of sand, gravel, water and lava, and redstone.

2.2 Extended features

Bravo provides many things not in other servers. While a strict comparison of other open-source servers is impossible due to the speedy rate at which they are changing, the features that separate Bravo from the Mojang-sponsored server are listed here.

2.2.1 Console

Bravo ships with a fancy console which supports readline-like editing features.

2.2.2 Time

Bravo implements an in-game year of 360 in-game days.

2.2.3 Plugins

Bravo supports several different types of plugins. For more information, see *Plugins*.

CHAPTER 3

Differences vs. vanilla Minecraft Server

Bravo was written from the ground up and doesn't inherit code from any other Minecraft project. This means that it sometimes behaves very differently, in subtle and obvious ways, from other servers.

The "Notchian" server is the server authored by Notch and distributed by Mojang as a companion to the Mojang-sponsored client.

3.1 Responsiveness

Bravo is occasionally perceived to be "lighter" or "snappier" compared to the Notchian server. Reports of feeling like players are moving faster than normal are also common. The root cause is simple: Bravo is quicker to respond to clients than the Notchian server. This is normal, expected, and not currently planned to be fixed.

3.2 Chunks

The Notchian server maintains a floating pattern above players, centered on the chunk the player is standing in. This pattern is always a square of chunks, 21 chunks to a side. This results in a total of 441 chunks being deployed to the client at any one time. All 441 chunks are deployed before the client is permitted to interact with the world.

Bravo does something slightly different; while Bravo also has a floating pattern above each of its players, the pattern is a circle with the same diameter as the Notchian server's square. This effectively results in a circle of 315 chunks deployed to the client; a savings of nearly 30% in memory and bandwidth for chunks. Additionally, only the 50 closest chunks are deployed before the client is spawned and permitted to interact with the world.

3.3 Inventory

The Notchian viewpoint of items in the inventory is as a list of slots. Each slot holds an item, identified by a single number, and can hold 1 to 64 instances of that item. Some items can be damaged. Some items are completely different depending on their damage.

Bravo views item identifiers as a composite key of a primary and secondary identifier. In this scheme, items with identical primary keys and different secondary keys are properly segregated, and item damage is stored as the secondary key, keeping items with differing amounts of damage from occupying the same slot. This avoids an entire class of bugs, where items can be stacked and unstacked to change the amount of damage on them, which have historically plagued the Notchian codebase.

3.4 Minecarts

Bravo permits minecart tracks to be placed on glass.

CHAPTER 4

Philosophy

4.1 Design Decisions

A **design decision** is a core component of building a large piece of software. Roughly stated, it is a choice to use a certain language, library, or methodology when constructing software. Design decisions can be metaphysical, and affect other design decisions. This is merely a way of talking formally and reasonably about choices made in producing Bravo.

This section is largely dedicated to members of the community that have decided that things in Bravo are done incorrectly. While we agree with the need of the community to constructively criticize itself, some things are not worth debating again.

4.1.1 Python

Python is occasionally seen as slow compared to statically typed languages. Some benchmarks certainly are very unflattering to Python, but we feel that there are several advantages to Python which are too important to sacrifice:

- Rapid prototyping
- Algorithmic simplicity
- Simple types
- Twisted

Additionally, with the advent of PyPy, the question of whether a full-fledged Python application is too slow for consumer hardware is rapidly fading.

Compared to Other Languages

C++

Mineserver was a cool attempt to write a custom server in C++. It still receives occasional updates, but never attempted the more ambitious features.

Haskell

The Bravo team attempted to port Bravo to Haskell. It was unsuccessful. Haskell does not yet have a mature library for creating massively event-driven network servers.

No Extension Modules

There are several good reasons to not ship "extension modules," pieces of code written in Fortran, C, or C++ which are compiled and dynamically linked against the CPython extension API. Some of them are:

Portability Python and C have different scopes of portability, and the scope of the C API for Python is limited practically to CPython. Each module we depend on externally has the potential to reduce the number of platforms we can support.

Maintainability C is not maintainable on the same scale as Python, even with (and, some would argue, especially with) the extremely structured syntax required to interface with the C API for Python. Cython is maintainable, but does not solve the other problems.

Dependencies Somebody has to provide binary versions of the modules for all the people without compilers. Practically, this does mean that Win32 users need to have binaries provided for them, as long as our thin veneer of Win32 compatibility holds up.

Forward-compatibility Frankly, extension modules are forever incompatible with the spirit of PyPy, and require, at bare minimum, a recompile and prayer before they'll cooperate. We support running Bravo on PyPy, and on this alone, we wish to not depend on them.

Frankly, most extension modules aren't worth this trouble. Extension modules which are well-tested, ubiquitous, and actively maintained, are generally going to be favored more than extensions which break, are hard to obtain or compile, or are derelict.

Twisted

Apparently, in this day and age, people are still of the opinion that Twisted is too big and not necessary for speedy, relatively bug-free networking. Nothing written here will convince these people; so, instead, I offer this promise: If anybody contributes a patch which makes Bravo not depend on Twisted, does not degrade its performance measureably, and does not break any part of Bravo, then I will acknowledge and apply it.

4.1.2 No Threads

Threads are evil. They are not an effective concurrency model in most cases. Tests done with offloading various parts of Bravo's CPU-bound tasks to threads have shown that threads are a liability in most cases, enforcing locking overhead while providing little to no actual benefit in terms of speed and latency.

However, as a concession to the CPU-centric nature of geometry generation, Bravo will offload all geometry generation to separate processes when Ampoule is available and enabled in its configuration file, which does yield massive improvements to server interactivity.

4.1.3 Extreme Extensibility

Bravo is remarkably extensible. Pieces of functionality that are considered essential or "core" are treated as plugins and dynamically loaded on server startup. Actual services are dynamically started and stopped as needed. Bravo's core does not even provide Minecraft services by default.

The reason for this extreme plugin approach is that Bravo was designed to be easily totally convertible; in theory, a proper set of configuration files and external plugins can completely change Bravo's behavior.

4.2 Versioning

Bravo's version numbers are not very complex. Here's a quick breakdown.

Major version numbers indicate the core structure of Bravo. A major version bump probably means that lots of modules changed names, or that something significant was added. In practice, this probably means that an entirely new set of protocols was added. (The next major version bump will probably be for InfiniCraft support.)

Minor version numbers are for changes to interfaces or any other change which means that external code relying on Bravo's API will have to be updated.

Patchlevel version numbers aren't currently used, but probably will signify that the release is a bugfix-only release with no significant change in functionality.

The hope of all of this is that, given a series of releases with the same major and minor, plugins do not have to be changed.

CHAPTER 5

Administrator Topics

The following topics are meant for those wishing to run a Bravo server. Topics such as installation, configuration, and troubleshooting are covered here. No software development background is necessary.

5.1 How to administer Bravo

While Bravo is not a massively complex piece of software on its own, the plugins and features that are available in Bravo can be overwhelming and daunting. This page is a short but comprehensive overview for new administrators looking to set up and run Bravo instances.

5.1.1 Configuration

Bravo uses a single configuration file, bravo.ini, for all of its settings. The file is in standard INI format. Note that this is not the extended INI format of Windows 32-bit configuration settings, nor the format of PHP's configuration files. Specifically, bravo.ini is parsed and written using Python's `ConfigParser` class.

An example configuration file is provided as `bravo.ini.example`, and is a good starting point for new configurations.

`bravo.ini` should live in one of three locations:

1. /etc/bravo
2. ~/.bravo
3. The working directory

All three locations will be checked, in that order, and more-recently-loaded configurations will override configurations in previous directories. For sanity purposes, it is highly encouraged to either use `/etc/bravo` if running as root, or `~/.bravo` if running as a normal user.

The configuration file is divided up into **sections**. Each section starts with a name, like `[section name]`, and only ends when another section starts, or at the end of the file.

A note on lists

Bravo uses long lists of named plugins, and has special facilities for handling them.

If an option takes a list of choices, then the choices should be comma-separated. They may be on the same line, or multiple lines; spaces do not matter much. (As an aside, spaces matter *inside* plugin names, but Bravo's plugin collection uses only underscores, not spaces, so this should not matter. If it does, bug your plugin authors to fix their code.)

Additionally, to simplify plugin naming, many plugin configuration options support **wildcards**. Currently, the "*" wildcard is supported. A "*" anywhere in an option list will be internally expanded to *all* of the available choices for that option.

The special notation "-" before a name will forcibly remove that name from a list.

Putting everything together, an example set of configurations might look like this:

```
some_option = first, second, third
some_newline_option = first, second,
    third, fourth
some_wildcard_option = *
some_picky_option = *, -fifth
another_picky_option = -fifth, -sixth, *
a_weird_but_valid_option = seventh, -seventh
```

General settings

These settings apply to all of Bravo. This section is named `[bravo]`.

fancy_console Whether to enable the fancy console in standalone mode. This setting will be overridden if the fancy console cannot be set up; e.g. on Win32 systems.

ampoule Whether asynchronous chunk generators will be used. This can result in massive improvements to Bravo's latency and responsiveness, and defaults to enabled. This setting will be overridden if Ampoule cannot be found.

World settings

These settings only apply to a specific world. Worlds are created by starting the section of the configuration with "world"; an example world section might start with `[world example]`.

port Which port to run on. Must be a number between 0 and 65535. Note that ports below 1024 are typically privileged and cannot be bound by non-root users.

host The hostname to bind to. Defaults to no hostname, which is usually correct for most people. If you don't know what this is, you don't need it.

url The path to the folder to use for loading and saving world data. Must be a valid URL.

serializer Which serializer to use for saving worlds. Currently, the "anvil" serializers is provided for compatibility with modern MC clients and servers.

seed A numeric seed to use for terrain generation. If omitted, the seed will be generated when the world is created. This option only affects new worlds; existing worlds already have a seed.

5.1.2 Plugin Data Files

Plugins have a standardized per-world storage. Only a few of the plugins that ship with Bravo use this storage. Each plugin has complete autonomy over its data files, but the file name varies depending on the serializer used to store the world. For example, when using the Alpha and Beta world serializers, the file name is <plugin>.dat, where <plugin> is the name of the plugin.

Bravo worlds have per-world IP ban lists. The IP ban lists are stored under the plugin name "banned_ips", with one IP address per line.

Warps and homes are stored in hey0 CSV format, in “warps” and “homes”.

5.2 Plugins

Bravo is highly configurable and extensible. The plugins shipped with Bravo are listed here, for convenience.

5.2.1 Packs

Beta

The Beta plugin pack, called “beta”, provides all of Bravo’s Beta compatibility in one single line of configuration.

5.2.2 Terrain generators

The following terrain generators may be added to the `generators` setting in your `bravo.ini` under the `[world]` section. The order in which these appear in the list is not important.

Beaches

Generates simple beaches.

Beaches are areas of sand around bodies of water. This generator will form beaches near all bodies of water regardless of size or composition; it will form beaches at large seashores and frozen lakes. It will even place beaches on one-block puddles.

Boring

Generates boring slabs of flat stone.

Grass

Grows grass on exposed dirt.

Caves

Carves caves and seams out of terrain.

Cliffs

Generates sheer cliffs.

Complex

Generates islands of stone and other ridiculous things.

Erosion

Erodes stone surfaces into dirt.

Float

Rips chunks out of the map, to create surreal chunks of floating land.

Safety

Generates terrain features essential for the safety of clients, such as the indestructible bedrock at Y = 0.

Warning: Removing this generator will permit players to dig through the bottom of the world.

Simplex

Generates organic-looking, continuously smooth terrain.

Saplings

Plants saplings at relatively silly places around the map.

Note: This generator only places saplings, and is not responsible for the growth of trees over time. The `trees` automaton should be used for ensuring that trees will grow.

Ore

Places ores and clay.

Watertable

Creates a flat water table half-way up the map (Y = 64).

5.2.3 Automatons

Automatons are simple tasks which examine and update the world as the world loads and displays data to players. They are able to do periodic or delayed work to keep the world properly. (The mental image of small robotic gardeners roving across the hills and valleys trimming grass and dusting trees is quite compelling and adorable!)

Automatons marked with (Beta) provide Beta compatibility and should probably be enabled.

- **lava**: Enable physics for placed lava springs. (Beta)
- **trees**: Turn planted saplings into trees. (Beta)
- **water**: Enable physics for placed water springs. (Beta)

5.2.4 Seasons

Bravo's years are 360 days long, with each day being 20 minutes long. For those who would like seasons, the following seasons be added to the `seasons` setting in your `bravo.ini` under the `[world]` section.

Winter

Causes water to freeze, and snow to be placed on certain block types. Winter starts on the first day of the year.

Spring

Thaws frozen water and removes snow as that was placed during Winter. Spring starts on the 90th day of the the year.

5.2.5 Hooks

Hooks are small pluggable pieces of code used to add event-driven functionality to Bravo.

Build hooks

Hooks marked with (Beta) provide Beta compatibility and should probably be enabled.

- **alpha_sand_gravel**: Make sand and gravel fall as if affected by gravity. (Beta)
- **bravo_snow**: Make snow fall as if affected by gravity.
- **build**: Enable placement of blocks from inventory onto the terrain. (Beta)
- **redstone**: Enable physics for placed redstone. (Beta)
- **tile**: Register tiles. Required for signs, furnaces, chests, etc. (Beta)
- **tracks**: Align minecart tracks. (Beta)

Dig hooks

- **alpha_sand_gravel**: Make sand and gravel fall as if affected by gravity. (Beta)
- **alpha_snow**: Destroy snow when it is dug or otherwise disturbed. (Beta)
- **bravo_snow**: Make snow fall as if affected by gravity.
- **give**: Spawn pickups for blocks and items destroyed by digging. (Beta)
- **lava**: Enable physics for lava. (Beta)
- **redstone**: Enable physics for redstone. (Beta)
- **torch**: Destroy torches that are not attached to walls or floors. (Beta)
- **tracks**: Align minecart tracks. (Beta)
- **water**: Enable physics for water. (Beta)

5.3 Troubleshooting

5.3.1 Configuring

When I connect to the server, the client gets an "End of Stream" error and the server log says something about "ConsoleRPCProtocol".

You are connecting to the wrong port.

Bravo always runs an RPC console by default. This console isn't directly accessible from clients. In order to connect a client, you must configure a world and connect to that world. See the example bravo.ini configuration file for an example of how to configure a world.

My world is snowy. I didn't want this.

In bravo.ini, change your `seasons` list to exclude winter. A possible incantation might be the following:

```
seasons = *, -winter
```

5.3.2 Errors

I get lots of RuntimeErrors from Exocet.

Upgrade to a newer Bravo which doesn't use Exocet.

I have an error involving construct!

Install Construct. It is a required package.

I have an error involving JSON!

If you update to a newer Bravo, you won't need JSON support.

I have an error involving IRC/AMP/ListOf!

Your Twisted is too old. You really do need Twisted 11.0 or newer.

I have an error ``TypeError: an integer is required`` when starting Bravo!

Your Twisted is too old. You *really* do need Twisted 11.0 or newer.

I am running as root on a Unix system and twistd cannot find ``bravo.service``. What's going on?

For security reasons, twistd doesn't look in non-system directories as root. If you insist on running as root, try an incantation like the following, setting `PYTHONPATH`:

```
# PYTHONPATH=. twistd -n bravo
```

But seriously, stop running as root.

5.3.3 Help!

If you are having a hard time figuring something out, encountered a bug, or have ideas, feel free to reach out to the community in one of several different ways:

- **IRC:** #bravoserver on FreeNode
- Post to our issue tracker.
- Speak up over our mailing list.

5.4 Web Service

Bravo comes with a simple web service which can be used to monitor the status of your server.

5.4.1 Configuration

Only one web service can be defined; it uses the configuration key `[web]` and has only one parameter, `port`, specifying the port on which to listen. An example configuration snippet might look like this:

```
[web]
port = 8080
```

CHAPTER 6

Developer Topics

The following topics are of general use to those wishing to modify or understand the Bravo source code. These topics are completely unecessary for those who are only interested in running or administering a Bravo server.

6.1 Extending Bravo

Bravo is designed to be highly extensible. This document is a short guide to the basics of writing extension code for Bravo.

6.1.1 Asynchronous Ideas

Bravo, being built on Twisted, has inherited most of the concepts of asynchronous control flow from Twisted, and uses them liberally. Nearly every plugin method is permitted to return a Deferred in place of their actual return value.

6.1.2 The Good, the Bad, and the Ugly

There are a lot of modules in the standard library. Some of them should not be used in Bravo.

The following modules are blacklisted because they conflict with, or are slow compared to, Twisted's own systems:

These modules are bad. All of them duplicate functionality available in Twisted, and do it in ways that can interfere with Twisted's ability to do things in a speedy manner. Do not use them under any circumstances.

- `asyncore`
- `multiprocessing`
- `socket`
- `subprocess`
- `thread`
- `threading`

These modules are ugly. They can quite easily corrupt memory or cause server crashes, and should be used with extreme caution and very good reasons. If you don't know exactly what you are doing, don't use these.

- `ctypes`
- `gc`
- `imp`

- inspect

6.1.3 Parameters

Hooks should accept a single named parameter, `factory`, which will be provided when the hook is loaded.

6.1.4 The Flexibility of Commands

Bravo's command interface is designed to feel like a regular class instead of a specialized plugin, while still providing lots of flexibility to authors. Let's look at a simple plugin:

```
class Hello(object):
    """
    Say hello to the world.
    """

    implements(IChatCommand)

    def chat_command(self, username, parameters):
        greeting = "Hello, %s!" % username
        yield greeting

    name = "hello"
    aliases = tuple()
    usage = ""
```

This command is a simple greeter which merely echoes a salutation to its caller. It is an `IChatCommand`, so it only works in the in-game chat, but that should not be a problem, since there is an internal, invisible adaptation from `IChatCommand` to `IConsoleCommand`. This means that chat commands are also valid console commands, without any action on your part! Pretty cool, huh?

So, how does this plugin actually work? Well, nearly every line of this plugin is required. The first thing you'll notice is that this plugin has a class docstring. Docstrings on commands are required; the docstring is used to provide help text. As with all chat commands, this plugin `implements(IChatCommand)`, which lets it be discovered as a command.

The plugin implements the required `chat_command(username, parameters)`, which will be called when a player uses the command. An interesting thing to note is that this plugin yields its return value; commands may return any iterable of lines, including a generator!

Finally, the plugin finishes with more required interface attributes: a name which will be used to call the command, a (possibly empty) list of aliases which can also be used to call the command, and a (possibly empty) usage string.

6.2 Noise

Bravo, like all Minecraft terrain generators, relies heavily on randomness to generate its terrain. In order to understand some of the design decisions in the terrain generator, it is required to understand noise and its various properties.

6.2.1 Probability

Noise's probability distribution is not even, equal, or normal. It *is* symmetric about 0, meaning that the absolute value of noise has all of the same relative probabilities as the entire range of noise.

When binned into a histogram with 100 bins, a few bins become very large.

Bin	Probability
0.00	2.6150%
0.49	2.2262%
0.59	1.8274%
0.43	1.8248%
0.42	1.7888%
0.58	1.5939%
0.48	1.5194%
0.41	1.5118%
0.18	1.4715%
0.24	1.4366%
0.54	1.4072%
0.22	1.3825%
0.50	1.3786%
0.44	1.3696%
0.26	1.3680%

6.3 Core

These modules comprise the core functionality of Bravo.

6.3.1 `beta` – Minecraft Beta

Protocols and factories for Minecraft Beta, the Mojang-authored Minecraft which everybody knows and loves.

Packets

`bravo.beta.packets.`**`IPacket`**(*object*)
Interface for packets.

class `bravo.beta.packets.`**`Metadata`**
Bases: `tuple`

Metadata(type, value)

`type`
Alias for field number 0

`value`
Alias for field number 1

`bravo.beta.packets.`**`Speed`**
alias of `speed`

`bravo.beta.packets.`**`make_error_packet`**(*message*)
Convenience method to generate an error packet bytestream.

`bravo.beta.packets.`**`make_packet`**(*packet*, **args*, ***kwargs*)
Constructs a packet bytestream from a packet header and payload.

The payload should be passed as keyword arguments. Additional containers or dictionaries to be added to the payload may be passed positionally, as well.

`bravo.beta.packets.`**`parse_packets`**`(`*bytestream*`)`

Opportunistically parse out as many packets as possible from a raw bytestream.

Returns a tuple containing a list of unpacked packet containers, and any leftover unparseable bytes.

`bravo.beta.packets.`**`parse_packets_incrementally`**`(`*bytestream*`)`

Parse out packets one-by-one, yielding a tuple of packet header and packet payload.

This function returns a generator.

This function will yield all valid packets in the bytestream up to the first invalid packet.

Returns a generator yielding tuples of headers and payloads

`bravo.beta.packets.`**`simple`**`(`*name*, *fmt*, **args*`)`

Make a customized namedtuple representing a simple, primitive packet.

Protocols

class `bravo.beta.protocol.`**`BetaProxyProtocol`**

Bases: `bravo.beta.protocol.BetaServerProtocol`

A `BetaServerProtocol` that proxies for an InfiniCraft client.

`add_node``(`*address*, *port*`)`

Add a new node to this client.

class `bravo.beta.protocol.`**`BetaServerProtocol`**

Bases: `object`, `twisted.internet.protocol.Protocol`, `twisted.protocols.policies.TimeoutMixin`

The Minecraft Alpha/Beta server protocol.

This class is mostly designed to be a skeleton for featureful clients. It tries hard to not step on the toes of potential subclasses.

`action``(`*container*`)`

Hook for action packets.

`animate``(`*container*`)`

Hook for animate packets.

`ascend``(`*count*`)`

Ascend to the next XZ-plane.

`count` is the number of ascensions to perform, and may be zero in order to force this player to not be standing inside a block.

Returns bool of whether the ascension was successful

This client must be located for this method to have any effect.

`authenticated``()`

Called when the client has successfully authenticated with the server.

`build``(`*container*`)`

Hook for build packets.

`chat``(`*container*`)`

Hook for chat packets.

`client_settings``(`*container*`)`

Hook for interaction setting packets.

`complete`(*container*)
Hook for tab-completion packets.

`digging`(*container*)
Hook for digging packets.

`equip`(*container*)
Hook for equip packets.

`error`(*message*)
Error out.

This method sends `message` to the client as a descriptive error message, then closes the connection.

`grounded`(*container*)
Hook for grounded packets.

`handshake`(*container*)
Hook for handshake packets.

Override this to customize how logins are handled. By default, this method will only confirm that the negotiated wire protocol is the correct version, copy data out of the packet and onto the protocol, and then run the `authenticated` callback.

This method will call the `pre_handshake` method hook prior to logging in the client.

`location_packet`(*container*)
Hook for location packets.

`orientation`(*container*)
Hook for orientation packets.

`orientation_changed`()
Called when the client moves.

This callback is only for orientation, not position.

`pickup`(*container*)
Hook for pickup packets.

`ping`(*container*)
Hook for ping packets.

By default, this hook will examine the timestamps on incoming pings, and use them to estimate the current latency of the connected client.

`play_notes`(*notes*)
Play some music.

Send a sequence of notes to the player. `notes` is a finite iterable of pairs of instruments and pitches.

There is no way to time notes; if staggered playback is desired (and it usually is!), then `play_notes()` should be called repeatedly at the appropriate times.

This method turns the block beneath the player into a note block, plays the requested notes through it, then turns it back into the original block, all without actually modifying the chunk.

`poll`(*container*)
Hook for poll packets.

By default, queries the parent factory for some data, and replays it in a specific format to the requester. The connection is then closed at both ends. This functionality is used by Beta 1.8 clients to poll servers for status.

position (*container*)
Hook for position packets.

position_changed ()
Called when the client moves.

This callback is only for position, not orientation.

pre_handshake ()
Whether this client should be logged in.

quit (*container*)
Hook for quit packets.

By default, merely logs the quit message and drops the connection.

Even if the connection is not dropped, it will be lost anyway since the client will close the connection. It's better to explicitly let it go here than to have zombie protocols.

respawn (*container*)
Hook for respawn packets.

send_chat (*message*)
Send a chat message back to the client.

settings_packet (*container*)
Hook for presentation setting packets.

sign (*container*)
Hook for sign packets.

update_location ()
Send this client's location to the client.

Also let other clients know where this client is.

update_ping ()
Send a keepalive to the client.

use (*container*)
Hook for use packets.

wacknowledge (*container*)
Hook for wacknowledge packets.

waction (*container*)
Hook for waction packets.

wclose (*container*)
Hook for wclose packets.

wcreative (*container*)
Hook for creative inventory action packets.

write_packet (*header*, ***payload*)
Send a packet to the client.

class `bravo.beta.protocol.`**BravoProtocol** (*config*, *name*)
Bases: `bravo.beta.protocol.BetaServerProtocol`

A `BetaServerProtocol` suitable for serving MC worlds to clients.

This protocol really does need to be hooked up with a `BravoFactory` or something very much like it.

build(**args*, ***kwargs*)
Handle a build packet.

Several things must happen. First, the packet's contents need to be examined to ensure that the packet is valid. A check is done to see if the packet is opening a windowed object. If not, then a build is run.

complete(*container*)
Attempt to tab-complete user names.

connectionLost(*reason=<twisted.python.failure.Failure <class 'twisted.internet.error.ConnectionDone'>>*)
Cleanup after a lost connection.

Most of the time, these connections are lost cleanly; we don't have any cleanup to do in the unclean case since clients don't have any kind of pending state which must be recovered.

Remember, the connection can be lost before identification and authentication, so `self.username` and `self.player` can be None.

enable_chunk(*x*, *z*)
Request a chunk.

This function will asynchronously obtain the chunk, and send it on the wire.

Returns *Deferred* that will be fired when the chunk is obtained, with no arguments

entities_near(*radius*)
Obtain the entities within a radius of this player.

Radius is measured in blocks.

pre_handshake()
Set up username and get going.

run_dig_hooks(*chunk*, *coords*, *block*)
Destroy a block and run the post-destroy dig hooks.

send_initial_chunk_and_location()
Send the initial chunks and location.

This method sends more than one chunk; since Beta 1.2, it must send nearly fifty chunks before the location can be safely sent.

settings_packet(*container*)
Acknowledge a change of settings and update chunk distance.

use(*container*)
For each entity in proximity (4 blocks), check if it is the target of this packet and call all hooks that stated interested in this type.

wcreative(*container*)
A slot was altered in creative mode.

class `bravo.beta.protocol.`**KickedProtocol**(*reason=None*)
Bases: `bravo.beta.protocol.BetaServerProtocol`

A very simple Beta protocol that helps enforce IP bans, Max Connections, and Max Connections Per IP.

This protocol disconnects people as soon as they connect, with a helpful message.

Factories

class `bravo.beta.factory.`**BravoFactory**(*config*, *name*)
Bases: `twisted.internet.protocol.Factory`

A `Factory` that creates `BravoProtocol` objects when connected to.

Create a factory and world.

`name` is the string used to look up factory-specific settings from the configuration.

Parameters **name** (*str*) – internal name of this factory

`broadcast` (*packet*)
Broadcast a packet to all connected players.

`broadcast_for_chunk` (*packet*, *x*, *z*)
Broadcast a packet to all players that have a certain chunk loaded.

x and *z* are chunk coordinates, not block coordinates.

`broadcast_for_others` (*packet*, *protocol*)
Broadcast a packet to all players except the originating player.

Useful for certain packets like player entity spawns which should never be reflexive.

`buildProtocol` (*addr*)
Create a protocol.

This overriden method provides early player entity registration, as a solution to the username/entity race that occurs on login.

`chat` (*message*)
Relay chat messages.

Chat messages are sent to all connected clients, as well as to anybody consuming this factory.

`create_entity` (*x*, *y*, *z*, *name*, ***kwargs*)
Spawn an entirely new entity at the specified block coordinates.

Handles entity registration as well as instantiation.

`destroy_entity` (*entity*)
Destroy an entity.

The factory doesn't have to know about entities, but it is a good place to put this logic.

`flush_all_chunks` ()
Flush any damage anywhere in this world to all players.

This is a sledgehammer which should be used sparingly at best, and is only well-suited to plugins which touch multiple chunks at once.

In other words, if I catch you using this in your plugin needlessly, I'm gonna have a chat with you.

`flush_chunk` (*chunk*)
Flush a damaged chunk to all players that have it loaded.

`give` (*coords*, *block*, *quantity*)
Spawn a pickup at the specified coordinates.

The coordinates need to be in pixels, not blocks.

If the size of the stack is too big, multiple stacks will be dropped.

Parameters

- **coords** (*tuple*) – coordinates, in pixels
- **block** (*tuple*) – key of block or item to drop
- **quantity** (*int*) – number of blocks to drop in the stack

`players_near`(*player*, *radius*)
Obtain other players within a radius of a given player.

Radius is measured in blocks.

`protocol`
alias of `BravoProtocol`

`register_entity`(*entity*)
Registers an entity with this factory.

Registration is perhaps too fancy of a name; this method merely makes sure that the entity has a unique and usable entity ID. In particular, this method does *not* make the entity attached to the world, or advertise its existence.

`register_plugins`()
Setup plugin hooks.

`scan_chunk`(*chunk*)
Tell automatons about this chunk.

`set_username`(*protocol*, *username*)
Attempt to set a new username for a protocol.

Returns whether the username was changed

`stopFactory`()
Called before factory stops listening on ports. Used to perform shutdown tasks.

`teardown_protocol`(*protocol*)
Do internal bookkeeping on behalf of a protocol which has been disconnected.

Did you know that "bookkeeping" is one of the few words in English which has three pairs of double letters in a row?

`update_season`()
Update the world's season.

`update_time`()
Update the in-game timer.

The timer goes from 0 to 24000, both of which are high noon. The clock increments by 20 every second. Days are 20 minutes long.

The day clock is incremented every in-game day, which is every 20 minutes. The day clock goes from 0 to 360, which works out to a reset once every 5 days. This is a Babylonian in-game year.

6.3.2 `blocks` – Block descriptions

The `blocks` module contains descriptions of blocks.

class `bravo.blocks.`**`Block`**(*slot*, *name*, *secondary=0*, *drop=None*, *replace=0*, *ratio=1*, *quantity=1*, *dim=16*, *breakable=True*, *orientation=None*, *vanishes=False*)
Bases: `object`

A model for a block.

There are lots of rules and properties specific to different types of blocks. This class encapsulates those properties in a singleton-style interface, allowing many blocks to be referenced in one location.

The basic idea of this class is to provide some centralized data and information about blocks, in order to abstract away as many special cases as possible. In general, if several blocks all have some special behavior, then it may be worthwhile to store data describing that behavior on this class rather than special-casing it in multiple places.

Parameters

- **slot** (*int*) – The index of this block. Must be globally unique.
- **name** (*str*) – A common name for this block.
- **secondary** (*int*) – The metadata/damage/secondary attribute for this block. Defaults to zero.
- **drop** (*tuple*) – The type of block that should be dropped when an instance of this block is destroyed. Defaults to the block value, to drop instances of this same type of block. To indicate that this block does not drop anything, set to air (0, 0).
- **replace** (*int*) – The type of block to place in the map when instances of this block are destroyed. Defaults to air.
- **ratio** (*float*) – The probability of this block dropping a block on destruction.
- **quantity** (*int*) – The number of blocks dropped when this block is destroyed.
- **dim** (*int*) – How much light dims when passing through this kind of block. Defaults to 16 = opaque block.
- **breakable** (*bool*) – Whether this block is diggable, breakable, bombable, explodeable, etc. Only a few blocks actually genuinely cannot be broken, so the default is True.
- **orientation** (*tuple*) – The orientation data for a block. See `orientable()` for an explanation. The data should be in standard face order.
- **vanishes** (*bool*) – Whether this block vanishes, or is replaced by, another block when built upon.

`face`(*metadata*)

Retrieve the face for given metadata corresponding to an orientation, or None if the metadata is invalid for this block.

This method only returns valid data for orientable blocks; check `orientable()` first.

`orientable`()

Whether this block can be oriented.

Orientable blocks are positioned according to the face on which they are built. They may not be buildable on all faces. Blocks are only orientable if their metadata can be used to directly and uniquely determine the face against which they were built.

Ladders are orientable, signposts are not.

Return type bool

Returns True if this block can be oriented, False if not.

`orientation`(*face*)

Retrieve the metadata for a certain orientation, or None if this block cannot be built against the given face.

This method only returns valid data for orientable blocks; check `orientable()` first.

class `bravo.blocks.`**`Item`**(*slot*, *name*, *secondary=0*)

Bases: `object`

An item.

`bravo.blocks.`**`armor_boots`** **= (301, 305, 309, 313, 317)**

List of slots of boots.

`bravo.blocks.`**`armor_chestplates`** **= (299, 303, 307, 311, 315)**

List of slots of chestplates.

Note that slot 303 (chainmail chestplate) is a chestplate, even though it is not normally obtainable.

`bravo.blocks.`**`armor_helmets`** **= (86, 298, 302, 306, 310, 314)**
List of slots of helmets.

Note that slot 86 (pumpkin) is a helmet.

`bravo.blocks.`**`armor_leggings`** **= (300, 304, 308, 312, 316)**
List of slots of leggings.

`bravo.blocks.`**`blocks`** **= {0: Block((0, 0) ‘air’: unbreakable, transparent), 1: Block((1, 0) ‘stone’: drops 1 (key (4, 0), rate 1**
A dictionary of `Block` objects.

This dictionary can be indexed by slot number or block name.

`bravo.blocks.`**`items`** **= {‘’: Item((398, 0) ‘’), ‘wooden-door’: Item((324, 0) ‘wooden-door’), 379: Item((379, 0) ‘’), ‘emerald’**
A dictionary of `Item` objects.

This dictionary can be indexed by slot number or block name.

`bravo.blocks.`**`parse_block`**(*block*)
Get the key for a given block/item.

`bravo.blocks.`**`unstackable`** **= (268, 269, 270)**
List of fuel blocks and items maped to burn time

6.3.3 `chunk` – Chunk data structures

The `chunk` module holds the data structures required to track and update block data in chunks.

class `bravo.chunk.`**`Chunk`**(*x*, *z*)
Bases: `object`

A chunk of blocks.

Chunks are large pieces of world geometry (block data). The blocks, light maps, and associated metadata are stored in chunks. Chunks are always measured 16xCHUNK_HEIGHTx16 and are aligned on 16x16 boundaries in the xz-plane.

Variables

- **dirty** (*bool*) – Whether this chunk needs to be flushed to disk.
- **populated** (*bool*) – Whether this chunk has had its initial block data filled out.
- **heightmap** (*array.array*) – Tracks the tallest block in each xz-column.
- **all_damaged** (*bool*) – Flag for forcing the entire chunk to be damaged. This is for efficiency; past a certain point, it is not efficient to batch block updates or track damage. Heavily damaged chunks have their damage represented as a complete resend of the entire chunk.

Parameters

- **x** (*int*) – X coordinate in chunk coords
- **z** (*int*) – Z coordinate in chunk coords

`clear_damage`()
Clear this chunk’s damage.

`damage`(*coords*)
Record damage on this chunk.

destroy(*chunk*, *coords*, **args*, ***kwargs*)

Destroy the block at the given coordinates.

This may or may not set the block to be full of air; it uses the block's preferred replacement. For example, ice generally turns to water when destroyed.

This is safe as a no-op; for example, destroying a block of air with no metadata is not going to cause state changes.

Parameters **coords** (*tuple*) – coordinate triplet

dirtied = None

Optional hook to be called when this chunk becomes dirty.

get_block(*chunk*, *coords*, **args*, ***kwargs*)

Look up a block value.

Parameters **coords** (*tuple*) – coordinate triplet

Return type int

Returns int representing block type

get_damage_packet()

Make a packet representing the current damage on this chunk.

This method is not private, but some care should be taken with it, since it wraps some fairly cryptic internal data structures.

If this chunk is currently undamaged, this method will return an empty string, which should be safe to treat as a packet. Please check with *is_damaged()* before doing this if you need to optimize this case.

To avoid extra overhead, this method should really be used in conjunction with *Factory.broadcast_for_chunk()*.

Do not forget to clear this chunk's damage! Callers are responsible for doing this.

```
>>> packet = chunk.get_damage_packet()
>>> factory.broadcast_for_chunk(packet, chunk.x, chunk.z)
>>> chunk.clear_damage()
```

Return type str

Returns String representation of the packet.

get_metadata(*chunk*, *coords*, **args*, ***kwargs*)

Look up metadata.

Parameters **coords** (*tuple*) – coordinate triplet

Return type int

get_skylight(*chunk*, *coords*, **args*, ***kwargs*)

Look up skylight value.

Parameters **coords** (*tuple*) – coordinate triplet

Return type int

height_at(*x*, *z*)

Get the height of an xz-column of blocks.

Parameters

- **x** (*int*) – X coordinate

- **z** (*int*) – Z coordinate

Return type int

Returns The height of the given column of blocks.

is_damaged()

Determine whether any damage is pending on this chunk.

Return type bool

Returns True if any damage is pending on this chunk, False if not.

regenerate()

Regenerate all auxiliary tables.

regenerate_heightmap()

Regenerate the height map array.

The height map is merely the position of the tallest block in any xz-column.

regenerate_skylight()

Regenerate the ambient light map.

Each block's individual light comes from two sources. The ambient light comes from the sky.

The height map must be valid for this method to produce valid results.

save_to_packet()

Generate a chunk packet.

sed(*search*, *replace*)

Execute a search and replace on all blocks in this chunk.

Named after the ubiquitous Unix tool. Does a semantic s/search/replace/g on this chunk's blocks.

Parameters

- **search** (*int*) – block to find
- **replace** (*int*) – block to use as a replacement

set_block(*chunk*, *coords*, **args*, ***kwargs*)

Update a block value.

Parameters

- **coords** (*tuple*) – coordinate triplet
- **block** (*int*) – block type

set_metadata(*chunk*, *coords*, **args*, ***kwargs*)

Update metadata.

Parameters

- **coords** (*tuple*) – coordinate triplet
- **metadata** (*int*) –

set_skylight(*chunk*, *coords*, **args*, ***kwargs*)

Update skylight value.

Parameters

- **coords** (*tuple*) – coordinate triplet
- **metadata** (*int*) –

exception `bravo.chunk.`**`ChunkWarning`**
Bases: `exceptions.Warning`

Somebody did something inappropriate to this chunk, but it probably isn't lethal, so the chunk is issuing a warning instead of an exception.

`bravo.chunk.`**`check_bounds`**(*f*)
Decorate a function or method to have its first positional argument be treated as an (x, y, z) tuple which must fit inside chunk boundaries of 16, CHUNK_HEIGHT, and 16, respectively.

A warning will be raised if the bounds check fails.

`bravo.chunk.`**`ci`**(*x*, *y*, *z*)
Turn an (x, y, z) tuple into a chunk index.

This is really a macro and not a function, but Python doesn't know the difference. Hopefully this is faster on PyPy than on CPython.

`bravo.chunk.`**`composite_glow`**(*target*, *strength*, *x*, *y*, *z*)
Composite a light source onto a lightmap.

The exact operation is not quite unlike an add.

`bravo.chunk.`**`iter_neighbors`**(*coords*)
Iterate over the chunk-local coordinates surrounding the given coordinates.

All coordinates are chunk-local.

Coordinates which are not valid chunk-local coordinates will not be generated.

`bravo.chunk.`**`make_glows`**()
Set up glow tables.

These tables provide glow maps for illuminated points.

`bravo.chunk.`**`neighboring_light`**(*glow*, *block*)
Calculate the amount of light that should be shone on a block.

`glow` is the brighest neighboring light. `block` is the slot of the block being illuminated.

The return value is always a valid light value.

`bravo.chunk.`**`segment_array`**(*a*)
Chop up a chunk-sized array into sixteen components.

The chops are done in order to produce the smaller chunks preferred by modern clients.

6.3.4 `entity` – Entities

The `entity` module contains entity classes.

class `bravo.entity.`**`Chest`**(**args*, ***kwargs*)
Bases: `bravo.entity.Tile`

A tile that holds items.

class `bravo.entity.`**`Chuck`**(***kwargs*)
Bases: `bravo.entity.Mob`

A cross between a duck and a chicken.

Create a mob.

This method calls super().

class `bravo.entity.`**`Cow`**(***kwargs*)

Bases: `bravo.entity.Mob`

Large, four-legged milk containers.

Create a mob.

This method calls super().

class `bravo.entity.`**`Creeper`**(*aura=False*, ***kwargs*)

Bases: `bravo.entity.Mob`

A creeper.

Create a creeper.

This method calls super()

class `bravo.entity.`**`Entity`**(*location=None*, *eid=0*, ***kwargs*)

Bases: `object`

Class representing an entity.

Entities are simply dynamic in-game objects. Plain entities are not very interesting.

Create an entity.

This method calls super().

class `bravo.entity.`**`Furnace`**(**args*, ***kwargs*)

Bases: `bravo.entity.Tile`

A tile that converts items to other items, using specific items as fuel.

`burn`(*ticks*)

The main furnace loop.

Parameters **ticks** (*int*) – number of furnace iterations to perform

`can_craft`()

Determine whether this furnace is capable of outputting items.

Note that this is independent of whether the furnace is fueled.

Returns bool

`changed`(*factory*, *coords*)

Called from outside by event handler to inform the tile that the content was changed. If the furnace meet the requirements the method starts `burn` process. The `burn` stops the looping call when it's out of fuel or no need to burn more.

We get furnace coords from outer side as the tile does not know about own chunk. If self.chunk is implemented the parameter can be removed and self.coords will be:

```
>>> self.coords = self.chunk.x, self.x, self.chunk.z, self.z, self.y
```

Parameters

- **factory** (*BravoFactory*) – The factory
- **coords** (*tuple*) – (bigx, smallx, bigz, smallz, y) - coords of this furnace

`has_fuel`()

Determine whether this furnace is fueled.

Returns bool

class `bravo.entity.`**`Ghast`**(***kwargs*)

Bases: `bravo.entity.Mob`

A very melancholy ghost.

Create a mob.

This method calls super().

class `bravo.entity.`**`GiantZombie`**(***kwargs*)

Bases: `bravo.entity.Mob`

Like a regular zombie, but far larger.

Create a mob.

This method calls super().

class `bravo.entity.`**`Mob`**(***kwargs*)

Bases: `bravo.entity.Entity`

A creature.

Create a mob.

This method calls super().

`name` **= 'Mob'**

The name of this mob.

Names are used to identify mobs during serialization, just like for all other entities.

This mob might not be serialized if this name is not overriden.

`run`()

Start this mob's update loop.

`save_to_packet`()

Create a "mob" packet representing this entity.

`update`()

Update this mob's location with respect to a factory.

`update_metadata`()

Overrideable hook for general metadata updates.

This method is necessary because metadata generally only needs to be updated prior to certain events, not necessarily in response to external events.

This hook will always be called prior to saving this mob's data for serialization or wire transfer.

class `bravo.entity.`**`MobSpawner`**(*x*, *y*, *z*)

Bases: `bravo.entity.Tile`

A tile that spawns mobs.

class `bravo.entity.`**`Music`**(*x*, *y*, *z*)

Bases: `bravo.entity.Tile`

A tile which produces a pitch when whacked.

class `bravo.entity.`**`Painting`**(*face='+x'*, *motive=''*, ***kwargs*)

Bases: `bravo.entity.Entity`

A painting on a wall.

Create a painting.

This method calls super().

save_to_packet()
Create a "painting" packet representing this entity.

class bravo.entity.**Pickup**(*item=(0, 0)*, *quantity=1*, ***kwargs*)
Bases: bravo.entity.Entity

Class representing a dropped block or item.

For historical and sanity reasons, this class is called Pickup, even though its entity name is "Item."

Create a pickup.

This method calls super().

save_to_packet()
Create a "pickup" packet representing this entity.

class bravo.entity.**Pig**(*saddle=False*, ***kwargs*)
Bases: bravo.entity.Mob

A provider of bacon and piggyback rides.

Create a pig.

This method calls super().

class bravo.entity.**Player**(*username=''*, ***kwargs*)
Bases: bravo.entity.Entity

A player entity.

Create a player.

This method calls super().

save_equipment_to_packet()
Creates packets that include the equipment of the player. Equipment is the item the player holds and all 4 armor parts.

save_to_packet()
Create a "player" packet representing this entity.

class bravo.entity.**Sheep**(*sheared=False*, *color=0*, ***kwargs*)
Bases: bravo.entity.Mob

A woolly mob.

Create a sheep.

This method calls super().

class bravo.entity.**Sign**(**args*, ***kwargs*)
Bases: bravo.entity.Tile

A tile that stores text.

class bravo.entity.**Skeleton**(***kwargs*)
Bases: bravo.entity.Mob

An archer skeleton.

Create a mob.

This method calls super().

class `bravo.entity.`**`Slime`**(*size=1*, ***kwargs*)
    Bases: `bravo.entity.Mob`

    A gelatinous blob.

    Create a slime.

    This method calls super().

class `bravo.entity.`**`Spider`**(***kwargs*)
    Bases: `bravo.entity.Mob`

    A spider.

    Create a mob.

    This method calls super().

class `bravo.entity.`**`Squid`**(***kwargs*)
    Bases: `bravo.entity.Mob`

    An aquatic source of ink.

    Create a mob.

    This method calls super().

class `bravo.entity.`**`Tile`**(*x*, *y*, *z*)
    Bases: `object`

    An entity that is also a block.

    Or, perhaps more correctly, a block that is also an entity.

class `bravo.entity.`**`Wolf`**(*owner=None*, *angry=False*, *sitting=False*, ***kwargs*)
    Bases: `bravo.entity.Mob`

    A wolf.

    Create a wolf.

    This method calls super().

class `bravo.entity.`**`Zombie`**(***kwargs*)
    Bases: `bravo.entity.Mob`

    A zombie.

    Create a mob.

    This method calls super().

class `bravo.entity.`**`ZombiePigman`**(***kwargs*)
    Bases: `bravo.entity.Mob`

    A zombie pigman.

    Create a mob.

    This method calls super().

6.3.5 `furnace` – Furnace Tile

The `Furnace` tile has method `changed(factory, coords)` where `coords` is tuple `(bigx, smallx, bigz, smallz, y)` - coordinates of the furnace which inventory was updated.

```
# inform content of furnace was probably changed
d = factory.world.request_chunk(bigx, bigz)
@d.addCallback
def on_change(chunk):
    furnace = self.get_furnace_tile(chunk, (x, y, z))
    if furnace is not None:
        furnace.changed(factory, coords)
```

`Furnace.changed()` method checks if current furnace shall start to burn: it must have source item, fuel and must have valid recipe. If it meets the requirements `Furnace` schedules `burn()` method with LoopingCall for every .5 second.

At every `burn()` call it:

1. increases cooktime timer and checks if item shall be crafted on this iteration;
2. decreases fuel counter and burns next fuel item if needed;
3. if there is no need to burn next fuel item because crafted slot is full or source slot is empty it stops the LoopingCall;
4. sends progress bars updates to all players that have this furnace's window opened.

6.3.6 `ibravo` – Interfaces

The `ibravo` module holds the interfaces required to implement plugins and hooks.

Interface Bases

These are the base interface classes for Bravo. Plugin developers probably will not inherit from these; they are used purely to express common plugin functionality.

class `bravo.ibravo.`**`IBravoPlugin`**

Bases: `zope.interface.Interface`

Interface for plugins.

This interface stores common metadata used during plugin discovery.

class `bravo.ibravo.`**`ISortedPlugin`**

Bases: `bravo.ibravo.IBravoPlugin`

Parent interface for sorted plugins.

Sorted plugins have an innate and automatic ordering inside lists thanks to the ability to advertise their dependencies.

Plugins

class `bravo.ibravo.`**`IAutomaton`**

Bases: `bravo.ibravo.IBravoPlugin`

An automaton.

Automatons are given blocks from chunks which interest them, and may do processing on those blocks.

class `bravo.ibravo.`**`IChatCommand`**

Bases: `bravo.ibravo.ICommand`

Interface for chat commands.

Chat commands are invoked from the chat inside clients, so they are always called by a specific client.

This interface is specifically designed to exist comfortably side-by-side with *IConsoleCommand*.

class `bravo.ibravo.`**`IConsoleCommand`**

Bases: `bravo.ibravo.ICommand`

Interface for console commands.

Console commands are invoked from a console or some other location with two defining attributes: Access restricted to superusers, and no user issuing the command. As such, no access control list applies to them, but they must be given usernames to operate on explicitly.

class `bravo.ibravo.`**`IRecipe`**

Bases: `bravo.ibravo.IBravoPlugin`

A description for creating materials from other materials.

class `bravo.ibravo.`**`ISeason`**

Bases: `bravo.ibravo.IBravoPlugin`

Seasons are transformational stages run during certain days to emulate an environment.

class `bravo.ibravo.`**`ISerializer`**

Bases: `bravo.ibravo.IBravoPlugin`

Class that understands how to serialize several different kinds of objects to and from disk-friendly formats.

Implementors of this interface are expected to provide a uniform implementation of their serialization technique.

class `bravo.ibravo.`**`ITerrainGenerator`**

Bases: `bravo.ibravo.ISortedPlugin`

Interface for terrain generators.

class `bravo.ibravo.`**`IWorldResource`**

Bases: `bravo.ibravo.IBravoPlugin`, `twisted.web.resource.IResource`

Interface for a world specific web resource.

Hooks

class `bravo.ibravo.`**`IPreBuildHook`**

Bases: `bravo.ibravo.ISortedPlugin`

Hook for actions to be taken before a block is placed.

class `bravo.ibravo.`**`IPostBuildHook`**

Bases: `bravo.ibravo.ISortedPlugin`

Hook for actions to be taken after a block is placed.

class `bravo.ibravo.`**`IDigHook`**

Bases: `bravo.ibravo.ISortedPlugin`

Hook for actions to be taken after a block is dug up.

class `bravo.ibravo.`**`ISignHook`**

Bases: `bravo.ibravo.ISortedPlugin`

Hook for actions to be taken after a sign is updated.

This hook fires both on sign creation and sign editing.

class `bravo.ibravo.`**`IUseHook`**

Bases: `bravo.ibravo.ISortedPlugin`

Hook for actions to be taken when a player interacts with an entity.

Each plugin needs to specify a list of entity types it is interested in in advance, and it will only be called for those.

6.3.7 `infini` – InfiniCraft

Protocols and factories for InfiniCraft.

Packets

`bravo.infini.packets.`**`InfiniPacket`**(*name*, *identifier*, *subconstruct*)

Common header structure for packets.

This is possibly not the best way to go about building these kinds of things.

`bravo.infini.packets.`**`String`**(*name*)

UTF-8 length-prefixed string.

`bravo.infini.packets.`**`make_packet`**(*packet*, **args*, ***kwargs*)

Constructs a packet bytestream from a packet header and payload.

The payload should be passed as keyword arguments. Additional containers or dictionaries to be added to the payload may be passed positionally, as well.

Protocols

Factories

class `bravo.infini.factory.`**`InfiniClientFactory`**(*config*, *name*)

Bases: `twisted.internet.protocol.Factory`

A `Factory` that serves as an InfiniCraft client.

class `bravo.infini.factory.`**`InfiniNodeFactory`**(*config*, *name*)

Bases: `twisted.internet.protocol.Factory`

A `Factory` that serves as an InfiniCraft node.

6.3.8 `inventory` – Inventories

The `inventory` module contains all kinds of windows and window parts like inventory, crafting and storage slots.

Generally to create a window you must create a `Window` object (of specific class derived from `Window`) and pass arguments like: window ID, player's inventory, slot's or tile's inventory, coordinates etc.

Generic construction (never use in your code :)

```
window = Window( id, Inventory(), Workbench(), ...)
```

Please note that player's inventory window is a special case. It is created when user logins and stays always opened. You probably will never have to create it.

```
def authenticated(self):
    BetaServerProtocol.authenticated(self)

    # Init player, and copy data into it.
    self.player = yield self.factory.world.load_player(self.username)
    ...
    # Init players' inventory window.
    self.inventory = InventoryWindow(self.player.inventory)
    ...
```

Every windows have own class. For instanse, to create a workbench window:

```
i = WorkbenchWindow(self.wid, self.player.inventory)
```

Furnace:

```
bigx, smallx, bigz, smallz, y = coords
furnace = self.chunks[x, y].tiles[(smallx, y, smallz)]
window = FurnaceWindow(self.wid, self.player.inventory, furnace.inventory, coords)
```

class `bravo.inventory.`**`Inventory`**

Bases: `bravo.inventory.SerializableSlots`

The class represents Player's inventory

`add`(*item*, *quantity*)

Attempt to add an item to the inventory.

Parameters **item** (*tuple*) – a key representing the item

Returns quantity of items that did not fit inventory

`consume`(*item*, *index*)

Attempt to remove a used holdable from the inventory.

A return value of `False` indicates that there were no holdables of the given type and slot to consume.

Parameters

- **item** (*tuple*) – a key representing the type of the item
- **slot** (*int*) – which slot was selected

Returns whether the item was successfully removed

`select_armor`(*index*, *alternate*, *shift*, *selected=None*)

Handle a slot selection on an armor slot.

Returns tuple (True/False, new selection)

class `bravo.inventory.`**`SerializableSlots`**

Bases: `object`

Base class for all slots configurations

class `bravo.inventory.slots.`**`Crafting`**

Bases: `bravo.inventory.slots.SlotsSet`

Base crafting class. Never shall be instantiated directly.

check_recipes()
See if the crafting table matches any recipes.

Returns None

close(*wid*)
Clear crafting areas and return items to drop and packets to send to client

reduce_recipe()
Reduce a crafting table according to a recipe.

This function returns None; the crafting table is modified in-place.

This function assumes that the recipe already fits the crafting table and will not do additional checks to verify this assumption.

select_crafted(*index*, *alternate*, *shift*, *selected=None*)
Handle a slot selection on a crafted output.

Parameters

- **index** – index of the selection
- **alternate** – whether this was an alternate selection
- **shift** – whether this was a shifted selection
- **selected** – the current selection

Returns a tuple of a bool indicating whether the selection was valid, and the newly selected slot

class `bravo.inventory.slots.`**LargeChestStorage**(*chest1*, *chest2*)
Bases: `bravo.inventory.slots.SlotsSet`

LargeChest is a wrapper around 2 ChestStorages

class `bravo.inventory.slots.`**SlotsSet**
Bases: `bravo.inventory.SerializableSlots`

Base calss for different slot configurations except player's inventory

class `bravo.inventory.windows.`**InventoryWindow**(*inventory*)
Bases: `bravo.inventory.windows.Window`

Special case of window - player's inventory window

creative(*slot*, *primary*, *secondary*, *quantity*)
Process inventory changes made in creative mode

class `bravo.inventory.windows.`**SharedWindow**(*wid*, *inventory*, *slots*, *coords*)
Bases: `bravo.inventory.windows.Window`

Base class for all windows with shared containers (like chests, furnace and dispenser)

Parameters

- **wid** (*int*) – window ID
- **inventory** (*Inventory*) – player's inventory object
- **tile** (*Tile*) – tile object
- **coords** (*tuple*) – world coords of the tile (bigx, smallx, bigz, smallz, y)

packets_for_dirty(*dirty_slots*)
Generate update packets for dirty usually privided by another window (sic!)

class `bravo.inventory.windows.`**`Window`**(*wid*, *inventory*, *slots*)

Bases: `bravo.inventory.SerializableSlots`

Item manager

The `Window` covers all kinds of inventory and crafting windows, ranging from user inventories to furnaces and workbenches.

The `Window` agregates player's inventory and other crafting/storage slots as building blocks of the window.

Parameters

- **wid** (*int*) – window ID
- **inventory** (*Inventory*) – player's inventory object
- **slots** (*SlotsSet*) – other window slots

`close`()

Clear crafting areas and return items to drop and packets to send to client

`container_for_slot`(*slot*)

Retrieve the table and index for a given slot.

There is an isomorphism here which allows all of the tables of this `Window` to be viewed as a single large table of slots.

`load_from_packet`(*container*)

Load data from a packet container.

`select`(*slot*, *alternate=False*, *shift=False*)

Handle a slot selection.

This method implements the basic public interface for interacting with `Inventory` objects. It is directly equivalent to mouse clicks made upon slots.

Parameters

- **slot** (*int*) – which slot was selected
- **alternate** (*bool*) – whether the selection is alternate; e.g., if it was done with a right-click
- **shift** (*bool*) – whether the shift key is toogled

`select_stack`(*container*, *index*)

Handle stacking of items (Shift + RMB/LMB)

`slot_for_container`(*table*, *index*)

Retrieve slot number for given table and index.

6.3.9 `location` – Locations

The `location` module contains objects for tracking and analyzing locations.

class `bravo.location.`**`Location`**

Bases: `object`

The position and orientation of an entity.

classmethod **`at_block`**(*x*, *y*, *z*)

Pinpoint a location at a certain block.

This constructor is intended to aid in pinpointing locations at a specific block rather than forcing users to do the pixel<->block maths themselves. Admittedly, the maths in question aren't hard, but there's no reason to avoid this encapsulation.

clamp()
Force this location to be sane.

Forces the position and orientation to be sane, then fixes up location-specific things, like stance.

Returns bool indicating whether this location had to be altered

distance(*other*)
Return the distance between this location and another location.

in_front_of(*distance*)
Return a `Location` a certain number of blocks in front of this position.

The orientation of the returned location is identical to this position's orientation.

Parameters **distance** (*int*) – the number of blocks by which to offset this position

save_to_packet()
Returns a position/look/grounded packet.

class bravo.location.**Orientation**
Bases: `bravo.location.Orientation`

The angles corresponding to the heading of an entity.

Theta and phi are very much like the theta and phi of spherical coordinates, except that phi's zero is perpendicular to the XZ-plane rather than pointing straight up or straight down.

Orientation is stored in floating-point radians, for simplicity of computation. Unfortunately, no wire protocol speaks radians, so several conversion methods are provided for sanity and convenience.

The `from_degs()` and `to_degs()` methods provide integer degrees. This form is called "yaw and pitch" by protocol documentation.

classmethod from_degs(*yaw*, *pitch*)
Create an `Orientation` from integer degrees.

to_degs()
Return this orientation as integer degrees.

to_fracs()
Return this orientation as fractions of a byte.

class bravo.location.**Position**
Bases: `bravo.location.Position`

The coordinates pointing to an entity.

Positions are *always* stored as integer absolute pixel coordinates.

distance(*other*)
Return the distance between this position and another, in absolute pixels.

classmethod from_player(*x*, *y*, *z*)
Create a `Position` from floating-point block coordinates.

heading(*other*)
Return the heading from this position to another, in radians.

This is a wrapper for the common atan2() expression found in games, meant to help encapsulate semantics and keep copy-paste errors from happening.

to_block()
Return this position as block coordinates.

`to_player`()
Return this position as floating-point block coordinates.

6.3.10 `plugin` – Plugin loader

The `plugin` module implements a sophisticated, featureful plugin loader with interface-based discovery.

`bravo.plugin.`**`add_plugin_edges`**(*d*)
Mirror edges to all plugins in a dictionary.

`bravo.plugin.`**`expand_names`**(*plugins*, *names*)
Given a list of names, expand wildcards and discard disabled names.

Used to implement * and - options in plugin lists.

Parameters

- **plugins** (*dict*) – plugins to use for expansion
- **names** (*list*) – names to examine

Returns a list of filtered plugin names

`bravo.plugin.`**`get_plugins`**(*interface*, *package*)
Lazily find objects in a package which implement a given interface.

This is a rewrite of Twisted's `twisted.plugin.getPlugins` which searches for implementations of interfaces rather than providers.

Parameters

- **interface** (*interface*) – the interface to match against
- **package** (*str*) – the name of the package to search

`bravo.plugin.`**`retrieve_named_plugins`**(*interface*, *names*, ***kwargs*)
Look up a list of plugins by name.

Plugins are returned in the same order as their names.

Parameters

- **interface** (*interface*) – the interface to use
- **names** (*list*) – plugins to find
- **parameters** (*dict*) – parameters to pass into the plugins

Returns a list of plugins

Raises PluginException no plugins could be found for the given interface

`bravo.plugin.`**`retrieve_plugins`**(*interface*, ***kwargs*)
Look up all plugins for a certain interface.

If the plugin cache is enabled, this function will not attempt to reload plugins from disk or discover new plugins.

Parameters

- **interface** (*interface*) – the interface to use
- **parameters** (*dict*) – parameters to pass into the plugins

Returns a dict of plugins, keyed by name

Raises PluginException no plugins could be found for the given interface

`bravo.plugin.`**`retrieve_sorted_plugins`**(*interface*, *names*, ***kwargs*)
Look up a list of plugins, sorted by interdependencies.

Parameters **parameters** (*dict*) – parameters to pass into the plugins

`bravo.plugin.`**`sort_plugins`**(*plugins*)
Make a sorted list of plugins by dependency.

If the list cannot be arranged into a DAG, an error will be raised. This usually means that a cyclic dependency was found.

Raises PluginException cyclic dependency detected

`bravo.plugin.`**`verify_plugin`**(*interface*, *plugin*)
Plugin interface verification.

This function will call `verifyObject()` and `validateInvariants()` on the plugins passed to it.

The primary purpose of this wrapper is to do logging, but it also permits code to be slightly cleaner, easier to test, and callable from other modules.

6.3.11 `stdio` – Console support

The `stdio` module provides a non-blocking, interactive console for administration, diagnostics, and debugging of running servers.

class `bravo.stdio.`**`AMPGateway`**(*host*, *port=25600*)
Bases: `object`

Wrapper around the logical implementation of a console.

`call`(*command*, *params*)
Run a command.

This is the client-side implementation; it wraps a few things to protect the console from raw logic and the server from builtin commands.

`connect`()
Connect this gateway to a remote Bravo server.

Returns a Deferred that will fire when connected, or fail if the connection cannot be established.

class `bravo.stdio.`**`BravoConsole`**(*ag*)
Bases: `twisted.protocols.basic.LineReceiver`

A console for things not quite as awesome as TTYs.

This console is extremely well-suited to Win32.

class `bravo.stdio.`**`BravoManhole`**(*factory*, **args*, ***kwargs*)
Bases: `twisted.conch.manhole.Manhole`

A console for TTYs.

6.3.12 `world` – Worlds

class `bravo.world.`**`ChunkCache`**
Bases: `object`

A cache which holds references to all chunks which should be held in memory.

This cache remembers chunks that were recently used, that are in permanent residency, and so forth. Its exact caching algorithm is currently null.

When chunks dirty themselves, they are expected to notify the cache, which will then schedule an eviction for the chunk.

exception `bravo.world.`**`ImpossibleCoordinates`**

Bases: `exceptions.Exception`

A coordinate could not ever be valid.

class `bravo.world.`**`World`**(*config*, *name*)

Bases: `object`

Object representing a world on disk.

Worlds are composed of levels and chunks, each of which corresponds to exactly one file on disk. Worlds also contain saved player data.

Parameters

name [str] The configuration key to use to look up configuration data.

`async` **= False**

Whether this world is using multiprocessing methods to generate geometry.

`connect`()

Connect to the world.

`destroy`(*coords*, **args*, ***kwargs*)

Destroy a block in an unknown chunk.

Returns a `Deferred` that will fire on completion

`dimension` **= 'earth'**

The world dimension. Valid values are earth, sky, and nether.

`enable_cache`(*size*)

Set the permanent cache size.

Changing the size of the cache sets off a series of events which will empty or fill the cache to make it the proper size.

For reference, 3 is a large-enough size to completely satisfy the Notchian client's login demands. 10 is enough to completely fill the Notchian client's chunk buffer.

Parameters **size** (*int*) – The taxicab radius of the cache, in chunks

Returns A `Deferred` which will fire when the cache has been

adjusted.

`factory` **= None**

The factory managing this world.

Worlds do not need to be owned by a factory, but will not callback to surrounding objects without an owner.

`flush_chunk`()

Flush a dirty chunk.

This method will always block when there are dirty chunks.

`get_block`(*coords*, **args*, ***kwargs*)

Get a block from an unknown chunk.

Returns a `Deferred` with the requested value

`get_metadata`(*coords*, **args*, ***kwargs*)
Get a block's metadata from an unknown chunk.

Returns a `Deferred` with the requested value

`level` = **Level(seed=0, spawn=(0, 0, 0), time=0)**
The initial level data.

`load_player`(*username*)
Retrieve player data.

Returns a `Deferred` that will be fired with a `Player`

`mark_dirty`(*coords*, **args*, ***kwargs*)
Mark an unknown chunk dirty.

Returns a `Deferred` that will fire on completion

`postprocess_chunk`(*chunk*)
Do a series of final steps to bring a chunk into the world.

This method might be called multiple times on a chunk, but it should not be harmful to do so.

`request_chunk`(**args*, ***kwargs*)
Request a `Chunk` to be delivered later.

Returns `Deferred` that will be called with the `Chunk`

`save_chunk`(*chunk*)
Write a chunk to the serializer.

Note that this method does nothing when the given chunk is not dirty or saving is off!

Returns A `Deferred` which will fire after the chunk has been

saved with the chunk.

`save_off`()
Disable saving to disk.

This is useful for accessing the world on disk without Bravo interfering, for backing up the world.

`save_on`()
Enable saving to disk.

`saving` = **True**
Whether objects belonging to this world may be written out to disk.

`set_block`(*coords*, **args*, ***kwargs*)
Set a block in an unknown chunk.

Returns a `Deferred` that will fire on completion

`set_metadata`(*coords*, **args*, ***kwargs*)
Set a block's metadata in an unknown chunk.

Returns a `Deferred` that will fire on completion

`start`()
Start managing a world.

Connect to the world and turn on all of the timed actions which continuously manage the world.

stop(**args*, ***kwargs*)
Stop managing the world.

This can be a time-consuming, blocking operation, while the world's data is serialized.

Note to callers: If you want the world time to be accurate, don't forget to write it back before calling this method!

Returns A `Deferred` that fires after the world has stopped.

sync_destroy(*coords*, **args*, ***kwargs*)
Destroy a block in an unknown chunk.

Returns None

sync_get_block(*coords*, **args*, ***kwargs*)
Get a block from an unknown chunk.

Returns the requested block

sync_get_metadata(*coords*, **args*, ***kwargs*)
Get a block's metadata from an unknown chunk.

Returns the requested metadata

sync_mark_dirty(*coords*, **args*, ***kwargs*)
Mark an unknown chunk dirty.

Returns None

sync_request_chunk(*coords*, **args*, ***kwargs*)
Get an unknown chunk.

Returns the requested `Chunk`

sync_set_block(*coords*, **args*, ***kwargs*)
Set a block in an unknown chunk.

Returns None

sync_set_metadata(*coords*, **args*, ***kwargs*)
Set a block's metadata in an unknown chunk.

Returns None

`bravo.world.`**coords_to_chunk**(*f*)
Automatically look up the chunk for the coordinates, and convert world coordinates to chunk coordinates.

`bravo.world.`**sync_coords_to_chunk**(*f*)
Either get a chunk for the coordinates, or raise an exception.

6.4 Auxiliary

Modules which do not contribute directly to the functionality of Bravo.

6.4.1 `simplex` – Simplex noise generation

`bravo.simplex.`**dot2**(*u*, *v*)
Dot product of two 2-dimensional vectors.

`bravo.simplex.`**`dot3`**`(`*u*, *v*`)`

Dot product of two 3-dimensional vectors.

`bravo.simplex.`**`octaves2`**`(`*x*, *y*, *count*`)`

Generate fractal octaves of noise.

Summing increasingly scaled amounts of noise with itself creates fractal clouds of noise.

Parameters

- **x** (*int*) – X coordinate
- **y** (*int*) – Y coordinate
- **count** (*int*) – number of octaves

Returns Scaled fractal noise

`bravo.simplex.`**`octaves3`**`(`*x*, *y*, *z*, *count*`)`

Generate fractal octaves of noise.

Parameters

- **x** (*int*) – X coordinate
- **y** (*int*) – Y coordinate
- **z** (*int*) – Z coordinate
- **count** (*int*) – number of octaves

Returns Scaled fractal noise

`bravo.simplex.`**`offset2`**`(`*x*, *y*, *xoffset*, *yoffset*, *octaves=1*`)`

Generate an offset noise difference field.

Parameters

- **x** (*int*) – X coordinate
- **y** (*int*) – Y coordinate
- **xoffset** (*int*) – X offset
- **yoffset** (*int*) – Y offset

Returns Difference of noises

`bravo.simplex.`**`reseed`**`(`*seed*`)`

Reseed the simplex gradient field.

`bravo.simplex.`**`set_seed`**`(`*seed*`)`

Set the current seed.

`bravo.simplex.`**`simplex2`**`(`*x*, *y*`)`

Generate simplex noise at the given coordinates.

This particular implementation has very high chaotic features at normal resolution; zooming in by a factor of 16x to 256x is going to yield more pleasing results for most applications.

The gradient field must be seeded prior to calling this function; call `reseed()` first.

Parameters

- **x** (*int*) – X coordinate
- **y** (*int*) – Y coordinate

Returns simplex noise

Raises Exception the gradient field is not seeded

`bravo.simplex.`**`simplex3`**`(`*x*, *y*, *z*`)`

Generate simplex noise at the given coordinates.

This is a 3-dimensional flavor of `simplex2()`; all of the same caveats apply.

The gradient field must be seeded prior to calling this function; call `reseed()` first.

Parameters

- **x** (*int*) – X coordinate
- **y** (*int*) – Y coordinate
- **z** (*int*) – Z coordinate

Returns simplex noise

Raises Exception the gradient field is not seeded or you broke the function somehow

6.4.2 `utilities` – Helper functions

The `utilities` package is the standard home for shared functions which many modules may use. The spirit of `utilities` is also to isolate sections of critical code so that unit tests can be used to ensure a minimum of bugginess.

Automaton Helpers

`bravo.utilities.automatic.`**`column_scan`**`(`*automaton*, *chunk*`)`

Utility function which provides a chunk scanner which only examines the tallest blocks in the chunk. This can be useful for automatons which only care about sunlit or elevated areas.

This method can be used directly in automaton classes to provide *scan()*.

`bravo.utilities.automatic.`**`naive_scan`**`(`*automaton*, *chunk*`)`

Utility function which can be used to implement a naive, slow, but thorough chunk scan for automatons.

This method is designed to be directly useable on automaton classes to provide the *scan()* interface.

This function depends on implementation details of `Chunk`.

Chat Formatting

Colorizers.

`bravo.utilities.chat.`**`complete`**`(`*sentence*, *possibilities*`)`

Perform completion on a string using a list of possible strings.

Returns a single string containing all possibilities.

`bravo.utilities.chat.`**`sanitize_chat`**`(`*s*`)`

Verify that the given chat string is safe to send to Notchian recepients.

`bravo.utilities.chat.`**`username_alternatives`**`(`*n*`)`

Permute a username through several common alternative-finding algorithms.

Coordinate Handling

Utilities for coordinate handling and munging.

`bravo.utilities.coords.`**`CHUNK_HEIGHT`** **= 256**

The total height of chunks.

`bravo.utilities.coords.`**`XZ`** **= [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 9), (0, 10), (0, 11), (0, 12), (0, 1**

The xz-coords for a chunk.

`bravo.utilities.coords.`**`adjust_coords_for_face`**`(`*coords*, *face*`)`

Adjust a set of coords according to a face.

The face is a standard string descriptor, such as “+x”.

The “noop” face is supported.

`bravo.utilities.coords.`**`iterchunk`**`()`

Yield an iterable of x, z, y coordinates for an entire chunk.

`bravo.utilities.coords.`**`itercube`**`(`*x*, *y*, *z*, *r*`)`

Yield an iterable of coordinates in a cube around a given block.

Coordinates with invalid Y values are discarded automatically.

`bravo.utilities.coords.`**`iterneighbors`**`(`*x*, *y*, *z*`)`

Yield an iterable of neighboring block coordinates.

The first item in the iterable is the original coordinates.

Coordinates with invalid Y values are discarded automatically.

`bravo.utilities.coords.`**`polar_round_vector`**`(`*vector*`)`

Rounds a vector towards zero

`bravo.utilities.coords.`**`split_coords`**`(`*x*, *z*`)`

Split a pair of coordinates into chunk and subchunk coordinates.

Parameters

- **x** (*int*) – the X coordinate
- **z** (*int*) – the Z coordinate

Returns a tuple of the X chunk, X subchunk, Z chunk, and Z subchunk

`bravo.utilities.coords.`**`taxicab2`**`(`*x1*, *y1*, *x2*, *y2*`)`

Return the taxicab distance between two blocks.

`bravo.utilities.coords.`**`taxicab3`**`(`*x1*, *y1*, *z1*, *x2*, *y2*, *z2*`)`

Return the taxicab distance between two blocks, in three dimensions.

Data Packing

More affectionately known as “bit-twiddling.”

`bravo.utilities.bits.`**`grouper`**`(`*n*, *iterable*, *fillvalue=None*`)`

grouper(3, ‘ABCDEFG’, ‘x’) –> ABC DEF Gxx

`bravo.utilities.bits.`**`pack_nibbles`**`(`*a*`)`

Pack pairs of nibbles into bytes.

Bytes are returned as characters.

Parameters **a** (*array*) – nibbles to pack

Returns packed nibbles as a string of bytes

`bravo.utilities.bits.`**`unpack_nibbles`**`(`*l*`)`

Unpack bytes into pairs of nibbles.

Nibbles are half-byte quantities. The nibbles unpacked by this function are returned as unsigned numeric values.

```
>>> unpack_nibbles("a")
[6, 1]
>>> unpack_nibbles("nibbles")
[6, 14, 6, 9, 6, 2, 6, 2, 6, 12, 6, 5, 7, 3]
```

Parameters **l** (*list*) – bytes

Returns list of nibbles

Decorators

General decorators for a variety of purposes.

`bravo.utilities.decos.`**`timed`**`(`*f*`)`

Print out timing statistics on a given callable.

Intended largely for debugging; keep this in the tree for profiling even if it's not currently wired up.

Geometry

Simple pixel graphics helpers.

`bravo.utilities.geometry.`**`gen_close_point`**`(`*point1*, *point2*`)`

Retrieve the first integer set of coordinates on the line from the first point to the second point.

The set of coordinates corresponding to the first point will not be retrieved.

`bravo.utilities.geometry.`**`gen_line_covered`**`(`*point1*, *point2*`)`

This is Bresenham's algorithm with a little twist: *all* the blocks that intersect with the line are yielded.

`bravo.utilities.geometry.`**`gen_line_simple`**`(`*point1*, *point2*`)`

An adaptation of Bresenham's line algorithm in three dimensions.

This function returns an iterable of integer coordinates along the line from the first point to the second point. No points are omitted.

Scheduling

class `bravo.utilities.temporal.`**`PendingEvent`**

Bases: `object`

An event which will happen at some point.

Structurally, this could be thought of as a poor man's upside-down DeferredList; it turns a single callback/errback into a broadcast which fires many multiple Deferreds.

This code came from Epsilon and should go into Twisted at some point.

`bravo.utilities.temporal.`**`split_time`**`(`*timestamp*`)`

Turn an MC timestamp into hours and minutes.

The time is calculated by interpolating the MC clock over the standard 24-hour clock.

Parameters **timestamp** (*int*) – MC timestamp, in the range 0-24000

Returns a tuple of hours and minutes on the 24-hour clock

`bravo.utilities.temporal.`**`timestamp_from_clock`**`(`*clock*`)`

Craft an int-sized timestamp from a clock.

More precisely, the size of the timestamp is 4 bytes, and the clock must be an implementor of IReactorTime. twisted.internet.reactor and twisted.internet.task.Clock are the primary suspects.

This function's timestamps are millisecond-accurate.

Spatial Hashes

class `bravo.utilities.spatial.`**`Block2DSpatialDict`**

Bases: `bravo.utilities.spatial.SpatialDict`

Class for tracking blocks in the XZ-plane.

`key_for_bucket``(`*key*`)`

Partition keys into chunk-sized buckets.

`keys_near``(`*key*, *radius*`)`

Get all bucket keys "near" this key.

This method may return a generator.

class `bravo.utilities.spatial.`**`Block3DSpatialDict`**

Bases: `bravo.utilities.spatial.SpatialDict`

Class for tracking blocks in the XZ-plane.

`key_for_bucket``(`*key*`)`

Partition keys into chunk-sized buckets.

`keys_near``(`*key*, *radius*`)`

Get all bucket keys "near" this key.

This method may return a generator.

class `bravo.utilities.spatial.`**`SpatialDict`**

Bases: `object`, `UserDict.DictMixin`

A spatial dictionary, for accelerating spatial lookups.

This particular class is a template for specific spatial dictionaries; in order to make it work, subclass it and add `key_for_bucket()`.

`iteritemsnear``(`*key*, *radius*`)`

A version of `iteritems()` that filters based on the distance from a given key.

The key does not need to actually be in the dictionary.

`iterkeys``()`

Yield all the keys.

`iterkeysnear``(`*key*, *radius*`)`

Yield all of the keys within a certain radius of this key.

`itervaluesnear``(`*key*, *radius*`)`

Yield all of the values within a certain radius of this key.

`keys``()`

Get a list of all keys in the dictionary.

Trigonometry

`bravo.utilities.maths.`**`circling`**`(`*x*, *y*, *r*`)`
Generate the points of the filled integral circle of the given radius around the given coordinates.

`bravo.utilities.maths.`**`clamp`**`(`*x*, *low*, *high*`)`
Clamp or saturate a number to be no lower than a minimum and no higher than a maximum.

Implemented as its own function simply because it's so easy to mess up when open-coded.

`bravo.utilities.maths.`**`dist`**`(`*first*, *second*`)`
Calculate the distance from one point to another.

`bravo.utilities.maths.`**`morton2`**`(`*x*, *y*`)`
Create a Morton number by interleaving the bits of two numbers.

This can be used to map 2D coordinates into the integers.

Inputs will be masked off to 16 bits, unsigned.

`bravo.utilities.maths.`**`rotated_cosine`**`(`*x*, *y*, *theta*, *lambd*`)`
Evaluate a rotated 3D sinusoidal wave at a given point, angle, and wavelength.

The function used is:

$$f(x,y) = -\cos((x\cos\theta - y\sin\theta)/\lambda)/2 + 1$$

This function has a handful of useful properties; it has a local minimum at f(0, 0) and oscillates infinitely betwen 0 and 1.

Parameters

- **x** (*float*) – X coordinate
- **y** (*float*) – Y coordinate
- **theta** (*float*) – angle of rotation
- **lambda** (*float*) – wavelength

Returns float of f(x, y)

`bravo.utilities.maths.`**`sorted_by_distance`**`(`*iterable*, *x*, *y*`)`
Like `sorted()`, but by distance to the given coordinates.

6.5 Tools

A handful of utilities are distributed with Bravo, in the tools directory.

6.5.1 Chunkbench

Chunkbench is a script that tests terrain generation speed.

6.5.2 Jsondump

Jsondump pretty-prints a JSON file.

6.5.3 NBTdump

NBTdump pretty-prints an NBT file.

6.5.4 Noiseview

Noiseview creates a picture of simplex noise, using Bravo's builtin noise generator.

6.5.5 parser-cli

parser-cli parses and pretty-prints raw Alpha packets.

CHAPTER 7

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index

b

bravo.beta.factory, 29
bravo.beta.packets, 25
bravo.beta.protocol, 26
bravo.blocks, 31
bravo.chunk, 33
bravo.entity, 36
bravo.infini.factory, 43
bravo.infini.packets, 43
bravo.infini.protocol, 43
bravo.inventory, 44
bravo.inventory.slots, 44
bravo.inventory.windows, 45
bravo.location, 46
bravo.plugin, 48
bravo.simplex, 52
bravo.stdio, 49
bravo.utilities.automatic, 54
bravo.utilities.bits, 55
bravo.utilities.chat, 54
bravo.utilities.coords, 55
bravo.utilities.decos, 56
bravo.utilities.geometry, 56
bravo.utilities.maths, 58
bravo.utilities.spatial, 57
bravo.utilities.temporal, 56
bravo.world, 49