Entwine Documentation

Release 2.2.0

Connor Manning

Apr 11, 2024

CONTENTS

Entwine is a data organization library for massive point clouds, designed to conquer datasets of trillions of points as well as desktop-scale point clouds. Entwine can index anything that is PDAL-readable, and can read/write to a variety of sources like S3 or Dropbox. Builds are completely lossless, so no points, metadata, or precision will be discarded even for terabyte-scale datasets.

Check out the client demos, showcasing Entwine output with Potree, Plasio, and Cesium clients.

Entwine is available under the LGPL License.

CHAPTER

ONE

NEWS

1.1 2019-08-30

Watch Connor Manning present "Continental scale point cloud data management with Entwine" (slides) at FOSS4G 2019 in Bucharest, Romania.

Links:

- Entwine
- PDAL
- EPT Tools
- USGS lidar
- Potree/Entwine demo
- Cesium/Entwine demo

1.2 2019-07-23

Entwine 2.1 is now released. See *Download* to obtain a copy of the source code.

1.3 2018-12-18

Entwine 2.0 is now released. See *Download* to obtain a copy of the source code.

1.4 2017-12-01

See Connor Manning present "Trillions of points - spatial indexing, organization, and exploitation of massive point clouds" at FOSS4G 2017 in Boston, MA USA in August 2017.

1.4.1 Quickstart

Getting started with Entwine is easy with Conda. Let's use Entwine to fetch and organize some public data, visualize it in our browser, and extract a quick elevation model and hillshade from it.

Installation

First, install Miniconda (or full Anaconda if you prefer) by downloading and running the install script for your platform. Then, run a shell and create an Entwine environment:

```
conda create -n entwine -c conda-forge entwine
```

This will create a new Conda environment, add the Conda Forge catalog to it, and install the Entwine package from the Conda Forge catalog.

Activate the `entwine` environment to use it:

```
conda activate entwine
```

Building the data

Make an Entwine data directory at `entwine` and use Entwine to fetch the Red Rocks Amphitheatre dataset from the internet and build an EPT dataset of it:

```
entwine build -i https://data.entwine.io/red-rocks.laz -o ~/entwine/red-rocks
```

Now we have our output at `~/entwine/red-rocks`. We could have also passed a directory like `-i ~/nyc/` to index multiple files. Now we can statically serve the `entwine` directory with an HTTP server and visualize it with the WebGL-based Potree and Plasio projects.

Viewing the data

Conda makes it easy to grab other things, and so we'll grab NodeJS and install `http-server` from it:

```
conda install nodejs -y
npm install http-server -g
http-server entwine -p 8080 --cors
```

Note: We need to set the `--cors` option to allow our localhost HTTP server to serve data to the remote Potree/Plasio pages.

With the server running, we can visit special Potree or Plasio URLs that allow you to take in localhost URLs and visualize them:

- Potree view
- Plasio view

Processing with PDAL

We can also use the PDAL EPT reader to create an elevation model of the data. This can be done over HTTP or the local filesystem. Use PDAL to translate the service to a GeoTIFF using the GDAL writer driver:

```
pdal translate ept://entwine/red-rocks red-rocks-dtm.tif --writers.gdal.resolution=2.0
```

That doesn't give us much to see, so let's create a hillshade using gdaldem:

```
gdaldem hillshade red-rocks-dtm.tif hillshade.png
```

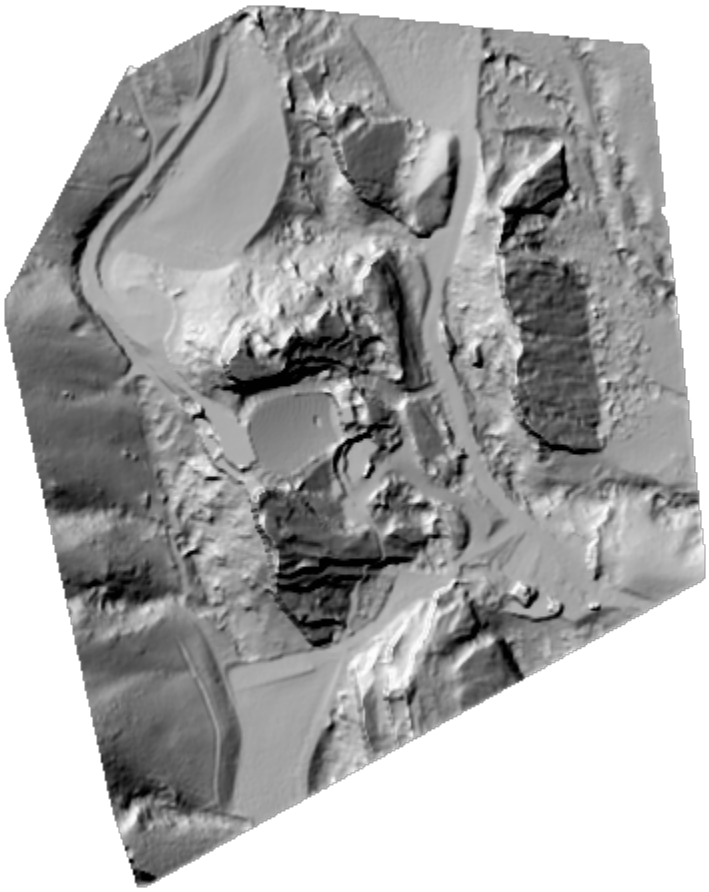

Fig. 1: Red Rocks Amphitheater (data from DroneMapper).

See also:

For further information about how to configure Entwine - like reprojecting data, using configuration files and templates, enabling S3 capabilities, and producing Cesium 3D Tiles output - see the Configuration section.

1.4.2 Download

Current Release

- **2021-08-04** entwine-2.2.0-src.tar.gz (release notes)

Previous Releases

- **2019-07-23** entwine-2.1.0-src.tar.gz
- **2018-12-18** entwine-2.0.0-src.tar.gz
- **2018-07-03** entwine-1.3.0-src.tar.gz
- **2017-12-04** entwine-1.2.0-src.tar.gz
- **2017-05-15** entwine-1.1.0-src.tar.gz
- **2017-05-15** entwine-1.0.0-src.tar.gz

Development Source

The main repository for Entwine is located on github at https://github.com/connormanning/entwine.

You can obtain a copy of the active source code by issuing the following command:

```
git clone https://github.com/connormanning/entwine.git
```

Binaries

Conda

The fastest way to get going with Entwine is to use the Conda build. See the *Quickstart* for more information. The Conda build will always contain the most recent released version of Entwine.

1.4.3 Configuration

Entwine provides 4 sub-commands for indexing point cloud data:

These commands are invoked via the command line as:

```
entwine <command> <arguments>
```

Although most options to entwine commands are configurable via command line, each command accepts configuration via JSON. A configuration file may be specified with the `-c` command line argument.

Command line argument settings are applied in order, so earlier settings can be overwritten by later-specified settings. This includes configuration file arguments, allowing them to be used as templates for common settings that may be overwritten with command line options.

Internally, Entwine CLI invocation builds a JSON configuration that is passed along to the corresponding sub-command, so CLI arguments and their equivalent JSON configuration formats will be described for each command. For example, with configuration file `config.json`:

```
{
    "input": "~/data/chicago.laz",
    "output": "~/entwine/chicago"
}
```

The following Entwine invocations are equivalent:

```
entwine build -i ~/data/chicago.laz -o ~/entwine/chicago
entwine build -c config.json
```

Throughout Entwine, a wide variety of point cloud formats are supported as input data, as any PDAL-readable format may be indexed. Paths are not required to be local filesystem paths - they may be local, S3, GCS, Dropbox, or any other Arbiter-readable format.

Each command accepts some common options, detailed at *common*.

Build

The `build` command is used to generate an *Entwine Point Tile* (EPT) dataset from point cloud data.

input

The point cloud data paths to be indexed. This may be a string, as in:

```
{ "input": "~/data/autzen.laz" }
```

This string may be:

- a file path: `~/data/autzen.laz` or `s3://entwine.io/sample-data/red-rocks.laz`
- a directory (non-recursive): `~/data` or `~/data/*`
- a recursive directory: `~/data/**`
- an info directory path: `~/entwine/info/autzen-files/`
- an info output file: `~/entwine/info/autzen-files/1.json`

This field may also be a JSON array of multiples of each of the above strings:

```
{ "input": ["autzen.laz", "~/data/"] }
```

Paths that do not contain PDAL-readable file extensions will be silently ignored.

output

A directory for Entwine to write its EPT output. May be local or remote.

tmp

A local directory for Entwine's temporary data.

srs

Specification for the output coordinate system. Setting this value does not invoke a reprojection, it simply sets the `srs` field in the resulting EPT metadata.

If input files have coordinate systems specified (and they all match), then this will typically be inferred from the files themselves.

reprojection

Coordinate system reprojection specification. Specified as a JSON object with up to 3 keys.

If only the output projection is specified, then the input coordinate system will be inferred from the file headers. If no coordinate system information can be found for a given file, then this file will not be inserted.

```
{ "reprojection": { "out": "EPSG:3857" } }
```

An input SRS may also be specified, which will be overridden by SRS information determined from file headers.

```
{
    "reprojection": {
        "in": "EPSG:26915",
        "out": "EPSG:3857"
    }
}
```

To force an input SRS that overrides any file header information, the `hammer` key should be set to `true.

```
{
    "reprojection": {
        "in": "EPSG:26915",
        "out": "EPSG:3857" ,
        "hammer": true
    }
}
```

When using this option, the `output` value will be set as the coordinate system in the resulting EPT metadata, so the `srs` option does not need to be specified.

threads

Number of threads for parallelization. By default, a third of these threads will be allocated to point insertion and the rest will perform serialization work.

```
{ "threads": 9 }
```

This field may also be an array of two numbers explicitly setting the number of worker threads and serialization threads, with the worker threads specified first.

```
{ "threads": [2, 7] }
```

force

By default, if an Entwine index already exists at the `output` path, any new files from the `input` will be added to the existing index. To force a new index instead, this field may be set to `true`.

```
{ "force": true }
```

dataType

Specification for the output storage type for point cloud data. Currently acceptable values are `laszip`, `zstandard`, and `binary`. For a `binary` selection, data is laid out according to the *schema*. Zstandard data consists of binary data according to the *schema* that is then compressed with Zstandard compression.

```
{ "dataType": "laszip" }
```

hierarchyType

Specification for the hierarchy storage format. Hierarchy information is always stored as JSON, but this field may indicate compression. Currently acceptable values are `json` and `gzip`.

```
{ "hierarchyType": "json" }
```

span

Number of voxels in each spatial dimension which defines the grid size of the octree. For example, a `span` value of `256` results in a `256 * 256 * 256` cubic resolution.

allowOriginId

For lossless capability, Entwine inserts an OriginId dimension tracking each point back to their original source file. If this value is present and set to `false`, this behavior will be disabled.

bounds

Total bounds for all points to be index. These bounds are final, in that they may not be expanded later after indexing has begun. Typically this field does not need to be supplied as it will be inferred from the data itself. This field is specified as an array of the format `[xmin, ymin, zmin, xmax, ymax, zmax]`.

```
{ "bounds": [0, 500, 30, 800, 1300, 50] }
```

schema

An array of objects representing the dimensions to be stored in the output. Each dimension is specified with a string `name`, a string `type`, and a string `size`. Typically this field does not need to be specified as it will be inferred from the data itself.

Valid `type` values are: `signed`, `unsigned`, and `float`.

Size values are the number of bytes used for each dimension. For example, an `unsigned` type with size 2 is capable of storing any `uint16` value. Likewise, n `unsigned` type with size 4 is capable of storing any `uint32`.

```
{
    "schema": [
        { "name": "X", "type": "unsigned", "size": 4 },
        { "name": "Y", "type": "unsigned", "size": 4 },
        { "name": "Z", "type": "unsigned", "size": 4 },
        { "name": "Intensity", "type": "int8", "size": 1 }
    ]
}
```

trustHeaders

By default, file headers for point cloud formats that contain information like number of points and bounds are considered trustworthy. If file headers are known to be incorrect, this value can be set to `false` to require a deep scan of all the points in each file.

absolute

Scaled values at a fixed precision are preferred by Entwine (and required for the `laszip` *dataType*). To use absolute double-precision values for XYZ instead, this value may be set to `true`.

scale

A scale factor for the spatial coordinates of the output. An offset will be determined automatically. May be a number like `0.01`, or a 3-length array of numbers for non-uniform scaling.

```
{ "scale": 0.01 }
```

```
{ "scale": [0.01, 0.01, 0.025] }
```

run

If a build should not run to completion of all input files, a `run` count may be specified to run a fixed maximum number of files. The build may be continued by providing the same `output` value to a later build.

```
{ "run": 25 }
```

subset

Entwine builds may be split into multiple subset tasks, and then be merged later with the *merge* command. Subset builds must contain exactly the same configuration aside from this `subset` field.

Subsets are specified with a 1-based `id` for the task ID and an `of` key for the total number of tasks. The total number of tasks must be a power of 4.

```
{ "subset": { "id": 1, "of": 16 } }
```

overflowDepth

There may be performance benefits by not allowing nodes near the top of the octree to contain overflow. The depth at which overflow may begin is specified by this parameter.

overflowThreshold

For nodes at depths of at least the `overflowDepth`, this parameter specifies the threshold at which they will split into bisected child nodes.

maxNodeSize

A soft limit on the maximum number of points that may be stored in a data node. This limit is only applicable to points that are "overflow" for a node - so points that fit natively in the `span * span * span` grid can grow beyond this size.

minNodeSize

A limit on the minimum number of points that may reside in a dedicated node. For would-be nodes containing less than this number, they will be grouped in with their parent node.

cacheSize

When data nodes have not been touched recently during point insertion, they are eligible for serialization. This parameter specifies the number of unused nodes that may be held in memory before serialization, so that if they are used again soon enough they won't need to be serialized and then reawakened from remote storage.

hierarchyStep

For large datasets with lots of data files, the hierarchy describing the octree layout is split up to avoid large downloads. This value describes the depth modulo at which hierarchy files are split up into child files. In general, this should be set only for testing purposes as Entwine will heuristically determine a value if the output hierarchy is large enough to warrant splitting.

Info

The `info` command is used to aggregate information about unindexed point cloud data prior to building an Entwine Point Tile dataset.

Most options here are common to `build` and perform exactly the same function in the `info` command, aside from `output`, described below.

output (info)

The `output` is a directory path to write detailed per-file metadata. This directory may then be used as the `input` for a *build* command.

Merge

The `merge` command is used to combine *subset* builds into a full Entwine Point Tile dataset. All subsets must be completed.

Note: This command is **not** used to merge unrelated EPT datasets.

output (merge)

The output path must be a directory containing `n` completed subset builds, where `n` is the `of` value from the subset specification.

Common

verbose

Defaults to `false`, and setting to `true` will enable a more verbose output to STDOUT.

arbiter

This value may be set to an object representing settings for remote file access. Amazon S3, Google Cloud Storage, and Dropbox settings can be placed here to be passed along to Arbiter. Some examples follow.

Enable Amazon S3 server-side encryption for the default profile:

```
{ "arbiter": {
    "s3": {
        "sse": true
    }
} }
```

Enable IO between multiple S3 buckets with different authentication settings. Profiles other than `default` must use prefixed paths of the form `profile@s3://<path>`, for example `second@s3://lidar-data/usa`:

```
{ "arbiter": {
    "s3": [
        {
            "profile": "default",
            "access": "<access key here>",
            "secret": "<secret key here>"
        },
        {
            "profile": "second",
            "access": "<access key here>",
            "secret": "<secret key here>",
            "region": "eu-central-1",
            "sse": true
        }
    ]
} }
```

Setting the S3 profile is also accessible via command line with `--profile <profile>`, and server-side encryption can be enabled by using `--sse`.

Miscellaneous

S3

Entwine can read and write S3 paths. The simplest way to make use of this functionality is to install AWSCLI and run `aws configure`, which will write credentials to `~/.aws`.

If you're using Docker, you'll need to map that directory as a volume. Entwine's Docker container runs as user `root`, so that mapping is as simple as adding `-v ~/.aws:/root/.aws` to your `docker run` invocation.

Cesium

Creating 3D Tiles point cloud datasets for display in Cesium is a two-step process.

First, an Entwine Point Tile datset must be created with an output projection of earth-centered earth-fixed, i.e. `EPSG:4978`:

```
mkdir ~/entwine
docker run -it -v ~/entwine:/entwine connormanning/entwine build \
    -i https://entwine.io/sample-data/autzen.laz \
    -o /entwine/autzen-ecef \
    -r EPSG:4978
```

Then, `entwine convert` must be run to create a 3D Tiles tileset:

```
docker run -it -v ~/entwine:/entwine connormanning/entwine convert \
    -i /entwine/autzen-ecef \
    -o /entwine/cesium/autzen
```

Statically serve the tileset locally:

```
docker run -it -v ~/entwine/cesium:/var/www -p 8080:8080 \
    connormanning/http-server
```

And browse the tileset with Cesium.

1.4.4 Entwine Point Tile

Entwine Point Tile (EPT) is a simple and flexible octree-based storage format for point cloud data. This document is a working draft of the format.

The organization of and EPT dataset contains JSON metadata portions as well as binary point data. This structure for a small dataset may look like this:

```
├── ept.json
├── ept-data
│   └── 0-0-0-0.laz
├── ept-hierarchy
│   └── 0-0-0-0.json
└── ept-sources
    ├── list.json
    └── 0.json
```

These files and directories are described below.

ept.json

The core metadata required to interpret the contents of an EPT dataset. An example file might look like this:


```
{
    "bounds": [634962.0, 848881.0, -1818.0, 639620.0, 853539.0, 2840.0],
    "boundsConforming": [635577.0, 848882.0, 406.0, 639004.0, 853538.0, 616.0],
    "dataType": "laszip",
    "hierarchyType": "json",
    "points": 10653336,
    "schema": [
        { "name": "X", "type": "signed", "size": 4, "scale": 0.01, "offset": 637291.0 },
        { "name": "Y", "type": "signed", "size": 4, "scale": 0.01, "offset": 851210.0 },
        { "name": "Z", "type": "signed", "size": 4, "scale": 0.01, "offset": 511.0 },
        { "name": "Intensity", "type": "unsigned", "size": 2 },
        { "name": "ReturnNumber", "type": "unsigned", "size": 1 },
        { "name": "NumberOfReturns", "type": "unsigned", "size": 1 },
        { "name": "ScanDirectionFlag", "type": "unsigned", "size": 1 },
        { "name": "EdgeOfFlightLine", "type": "unsigned", "size": 1 },
        { "name": "Classification", "type": "unsigned", "size": 1 },
        { "name": "ScanAngleRank", "type": "float", "size": 4 },
        { "name": "UserData", "type": "unsigned", "size": 1 },
        { "name": "PointSourceId", "type": "unsigned", "size": 2 },
        { "name": "GpsTime", "type": "float", "size": 8 },
        { "name": "Red", "type": "unsigned", "size": 2 },
        { "name": "Green", "type": "unsigned", "size": 2 },
        { "name": "Blue", "type": "unsigned", "size": 2 },
        { "name": "OriginId", "type": "unsigned", "size": 4 }
    ],
    "span" : 256,
    "srs": {
        "authority": "EPSG",
        "horizontal": "3857",
        "vertical": "5703",
        "wkt": "PROJCS[\"WGS 84 ... AUTHORITY[\"EPSG\",\"3857\"]]"
    },
    "version" : "1.1.0"
}
```


bounds

An array of 6 numbers of the format `[xmin, ymin, zmin, xmax, ymax, zmax]` describing the cubic bounds of the octree indexing structure. This value is always in native coordinate space, so any `scale` or `offset` values will not have been applied. This value is presented in the coordinate system matching the `srs` value.

boundsConforming

An array of 6 numbers of the format `[xmin, ymin, zmin, xmax, ymax, zmax]` describing the narrowest bounds conforming to the maximal extents of the data. This value is always in native coordinate space, so any `scale` or `offset` values will not have been applied. This value is presented in the coordinate system matching the `srs` value.

dataType

A string describing the binary encoding of the tiled point cloud data. See the `Point cloud data` section. Possible values:

- `laszip`: Point cloud files are LASzip compressed, with file extension `.laz`.
- `binary`: Point cloud files are stored as uncompressed binary data in the format matching the `schema`, with file extension `.bin`.
- `zstandard`: Point cloud files are stored as compressed binary data (using Zstandard compression) in the format matching the `schema`, with file extension `.zst`.

hierarchyType

A string describing the encoding of the hierarchy information. See the `Hierarchy` section. The hierarchy itself is always represented as JSON, but this value may indicate a compression method for this JSON. Possible values:

- `json`: Hierarchy is stored uncompressed with file extension `.json`.
- `gzip`: Hierarchy is stored as Gzip compressed JSON with file extension `.json.gz`.

points

A number indicating the total number of points indexed into this EPT dataset.

schema

An array of objects that represent the indexed dimensions - every dimension has a `name`, `type`, and `size`.

Known dimension names are mapped to PDAL well known dimension list, but arbitrary names may exist in addition to these prescribed dimensions. Sizes are specified in 8-bit bytes, and all dimension attributes must be of integral byte size.

There are 10 prescribed combinations of `type` and `size` representing primitive types available to most programming languages.

Attributes of other application-defined types (e.g. `string`) are possible as long as their `type` appropriately specifies their size in bytes.

For a `dataType` of `binary`, the `schema` provides information on the binary contents of each file. However for `laszip` data, the file should be parsed according to its header, as individual LASzip formats may combine dimension values. For example, for point format IDs 0-4, `ReturnNumber`, `NumberOfReturns`, `ScanDirectionFlag`, and `EdgeOfFlightLine` dimensions are bit-packed into a single byte.

In addition to the required `name`, `type`, and `size` specifications, attributes may also contain optional `scale` and/or `offset` values. These options specify that the absolutely positioned value of a given attribute should be computed as `read_value * scale + offset`. If `scale` does not exist then it is implicitly `1.0`. If `offset` does not exist it is implicitly `0.0`. This is commonly used to provide a fixed precision to the `X`, `Y`, and `Z` spatial dimensions.

span

This value represents the span of voxels in each spatial dimension defining the grid size used for the octree. This value must be a power of `2`.

For example, a `span` of `256` means that the root volume of the octree consists of the `bounds` cube split into a voxelized resolution of `256 * 256 * 256`.

For aerial LiDAR data which tends to be much denser in its X and Y ranges than the Z range, for example, this would loosely correspond to a practical resolution of `256 * 256` points per volume.

Because the `span` is constant throughout an entire EPT dataset, but each subsequent depth bisects the cubic volume of the previous, each increase in depth effectively doubles the resolution in each spatial dimension.

srs

An object describing the spatial reference system for this indexed dataset, or may not exist if a spatial reference could not be determined and was not set manually. In this object there are string keys with string values of the following descriptions:

- `authority`: Typically `"EPSG"` (if present), this value represents the authority for horizontal code as well as the vertical code if one is present.
- `horizontal`: Horizontal coordinate system code with respect to the given `authority`. If present, `authority` must exist.
- `vertical`: Vertical coordinate system code with respect to the given `authority`. If present, both `authority` and `horizontal` must exist.
- `wkt`: A WKT specification of the spatial reference, if available.

For a valid `srs` specification: if one of either `authority` or `horizontal` exists, then the other must also exist. If `vertical` exists, then both `authority` and `horizontal` must exist.

The `srs` specification may be an empty object.

version

Version string in the form of `<major>.<minor>.<patch>`.

ept-data

The point cloud data itself is arranged in a 3D analogous manner to slippy map tiling schemes. The filename scheme `Zoom-X-Y` is expanded to three dimensions as `Depth-X-Y-Z`. As opposed to raster tiling schemes where coarser-resolution data is replaced as its sub-tiles are traversed, the point cloud data is instead additive, meaning that full-resolution is obtained by the accumulated traversal from depth `0` to the last depth containing a positive point count for the selected area.

The root volume is always named `0-0-0-0`. This volume always represents the volume of the cubic `bounds` value from `ept.json` split in each dimension by `span`. So a `bounds` with a volume of 256 cubic meters, with a `span` value of `256`, corresponds to a root node with a resolution of up to 1 point per cubic meter.

Each node at depth `D` may have up to 8 child nodes at depth `D + 1` which represent bisected sub-volumes. To zoom in, depth `D` is incremented by one, and each of `X`, `Y`, and `Z` are doubled and then possibly incremented by one. Coordinates are treated as Cartesian during this bisection, so `X -> 2 * X` represents the sub-volume where the new maximum `X`

value is the midpoint from its parent, and `X -> 2 * X + 1` represents the sub-volume where the new minimum `X` value is the midpoint from its parent.

In web-mercator, for example, where `X` increases going eastward and `Y` increases going northward, a traversal from `0-0-0-0` to `1-1-0-1` represents a traversal to the east/south/down sub-volume.

There is no fixed maximum resolution depth, instead the tiles must be traversed until no more data exists. For look-ahead capability, see `ept-hierarchy`.

Note that unlike raster tiling schemes, where lower-resolution data is replaced by higher-resolution data during traversal, EPT is an additive scheme rather than a replacement scheme. So data at lower resolution nodes is not duplicated at higher resolution nodes, and full resolution for a given area is found via the aggregation of all the points from the highest resolution leaf node with all overlapping lower resolution nodes.

ept-hierarchy

The hierarchy section contains information about what nodes exist and how many points they contain. The file format is simple JSON object, with string keys of `D-X-Y-Z` mapping to a point count for the corresponding file. The root file of the hierarchy data exists at `ept-hierarchy/0-0-0-0.json`. For example:

```
{
    "0-0-0-0": 65341,
        "1-0-0-0": 438,
            "2-0-1-0": 322,
        "1-0-0-1": 56209,
            "2-0-1-2": 4332,
            "2-1-1-2": 20300,
            "2-1-1-3": 64020,
                "3-2-3-6": 32004,
                    "4-4-6-12": 1500,
                    "4-5-6-13": 2400,
                "3-3-3-7": 542,
        "1-0-1-0": 30390,
            "2-1-2-0": 2300,
        "1-1-1-1": 2303
}
```

Note that this sample is visually arranged hierarchically for clarity, which is not the case in practice. In this example, the root node `0-0-0-0` contains 65341 points. Nodes with zero points are never included in the hierarchy, so the absence of any child key indicates a leaf node (for example `4-4-6-12` above).

Hierarchy nodes must always contain a positive value with the sole exception of the sentinel value `-1`. This value indicates that the number of points indicator value for this node resides in a subtree in its own file. For example:

`ept-hierarchy/0-0-0-0.json`

```
{
    "0-0-0-0": 65341,
        "1-0-0-0": 438,
            "2-0-1-0": 322,
        "1-0-0-1": 56209,
            "2-0-1-2": 4332,
            "2-1-1-2": 20300,
            "2-1-1-3": 64020,
                "3-2-3-6": -1,
```

(continues on next page)

(continued from previous page)

```
                "3-3-3-7": -1,
        "1-0-1-0": 30390,
            "2-1-2-0": 2300,
        "1-1-1-1": 2303
}
```

ept-hierarchy/3-2-3-6.json

```
{
    "3-2-3-6": 32004,
        "4-4-6-12": 1500,
        "4-5-6-13": 2400
}
```

ept-hierarchy/3-3-3-7.json

```
{
    "3-3-3-7": 542
}
```

ept-sources

Sparse input data source information is stored in an array at `ept-sources/manifest.json`. This contains an array of JSON objects representing a metadata summary for each input source. This array may potentially be an empty array if this information is not stored. If an `OriginId` dimension exists in the `schema`, then each item's position in this array maps to its `OriginId` value in the EPT dataset, starting from `0` at the first position in the array. An sample `manifest.json` file may look like this:

```
[
    {
        "path": "autzen-low.laz",
        "bounds": [635577.0, 848882.0, 406.0, 639004.0, 853538.0, 511.0],
        "inserted": true,
        "points": 6000 ,
        "metadataPath": "autzen-low.json"
    },
    {
        "path": "autzen-high.laz",
        "bounds": [635577.0, 848882.0, 511.0, 639004.0, 853538.0, 616.0],
        "inserted": true,
        "points": 4000,
        "metadataPath": "autzen-high.json"
    }
]
```

bounds

A source object must contain a bounds which is an array of 6 numbers of the format `[xmin, ymin, zmin, xmax, ymax, zmax]`.

error

A string value representing an error which occurred during the insertion of this file. For successfully inserted files, this key must be absent.

inserted

A flag denoting whether this file has been inserted into the EPT dataset. In general this should always be `true` for public-facing EPT datasets, and will be false for builds which are only partially complete.

metadataPath

A source object may optionally contain a string URL which points to a file, relative to the `ept-sources/` location, which contains more thorough metadata for this source. If present, this URL must end in `.json` and this file must exist in JSON format. The format of this metadata file is discussed more fully in the *Source metadata* section, below.

points

The number of points inserted from this file.

Source metadata

To be lossless and facilitate a full reconstitution of original source data files from an EPT index, the full metadata for each input file may be retained. For each listed source in `ept-sources/manifest.json` containing a `metadataPath`, this URL points to a JSON file containing the associated metadata for that source.

This file contains a JSON object which may contain `bounds`, `path`, `points`, `schema`, and `srs` keys. These keys have the same meanings and range of values described above, but are expressed here on a per-source basis rather than per-EPT dataset.

In addition to the keys described above, each metadata object can also contain a `metadata` key, which maps to a JSON object representing arbitrary metadata for this source.

With the `ept-sources/list.json` file described above, a corresponding `ept-sources/autzen-high.json` file might look something like:

```
{
    "path": "autzen-high.laz",
    "bounds": [635577.0, 848882.0, 511.0, 639004.0, 853538.0, 616.0],
    "points" : 120000,
    "srs" :
    {
        "authority" : "EPSG",
        "horizontal" : "3857",
        "wkt": "PROJCS[\"WGS 84 ... AUTHORITY[\"EPSG\",\"3857\"]]"
```

(continues on next page)

(continued from previous page)

```
    },
    "metadata" : {
        "key": "value",
        "sofware_id": "PDAL",
        "version": 58,
        "something_else": -1
    },
    "schema": [] // Omitted for brevity.
}
```

1.4.5 Community

Entwine's community interacts through Gitter, GitHub, and a mailing list. Please feel welcome to ask questions and participate in all of the venues. The Gitter channels are for real-time chat activities. The GitHub communication channel is for development activities, bug reports, and testing. The mailing list communication channel is for general questions, development discussion, and feedback. The mailing list and Gitter channels are shared with those of PDAL.

Gitter

Entwine developers and some users are active on Gitter, which can be used for asking questions in a real-time chat. Gitter uses your GitHub credentials for access, so you will need an account to get started.

GitHub

Visit http://github.com/connormanning/entwine to file issues you might be having with the software. GitHub is also where you can obtain a current development version of the software in the git version control system.

Mailing List

Developers and users of Entwine participate on the PDAL mailing list. It is OK to ask questions about how to use Entwine, how to integrate Entwine into your own software, and report issues that you might have.

http://lists.osgeo.org/mailman/listinfo/pdal

Note: Please remember that an email to the PDAL list is going to hundreds of individuals. Do your diligence the best you can on your question before asking, but don't be afraid to ask.

1.4.6 Presentations

- FOSS4G 2017 (slides)
- FOSS4G 2016 (slides)