steno3d Documentation

Release 0.3.2

ARANZ Geo Limited

Jun 07, 2017

Contents

Welcome to the Python client library for Steno3D by ARANZ Geo Limited. Explore and collaborate on your 3D data!

CHAPTER 1

Demo Video

CHAPTER 2

Quickstart

If you have not yet installed Steno3D, simply

```
pip install steno3d
```

You also need to sign up for a Steno3D account. From there, you can request a developer API key.

At that point, you can

```
import steno3d
steno3d.login()
```

then start building your 3D project. API documentation is available on online. Tutorials and Examples are available as Jupyter notebooks. Class documentation can also be accessed in the IPython environment with *?* and tab completion. The latest version of Steno3D is 0.3.2. Detailed release notes are available on github.

Contents

Getting Started

Get up and running with Steno3D! This page contains resources for installing Steno3D, trying out a sample project yourself, and exploring public projects.

- *Install Steno3D*
- *A First Project*
- *Explore Steno3D*

If you run into issues: report them on github.

Install Steno3D

Want to start using Steno3D with your own data? It is available on pip:

```
pip install steno3d
```

or install from source:

```
git clone https://github.com/3ptscience/steno3dpy.git
python setup.py install
```

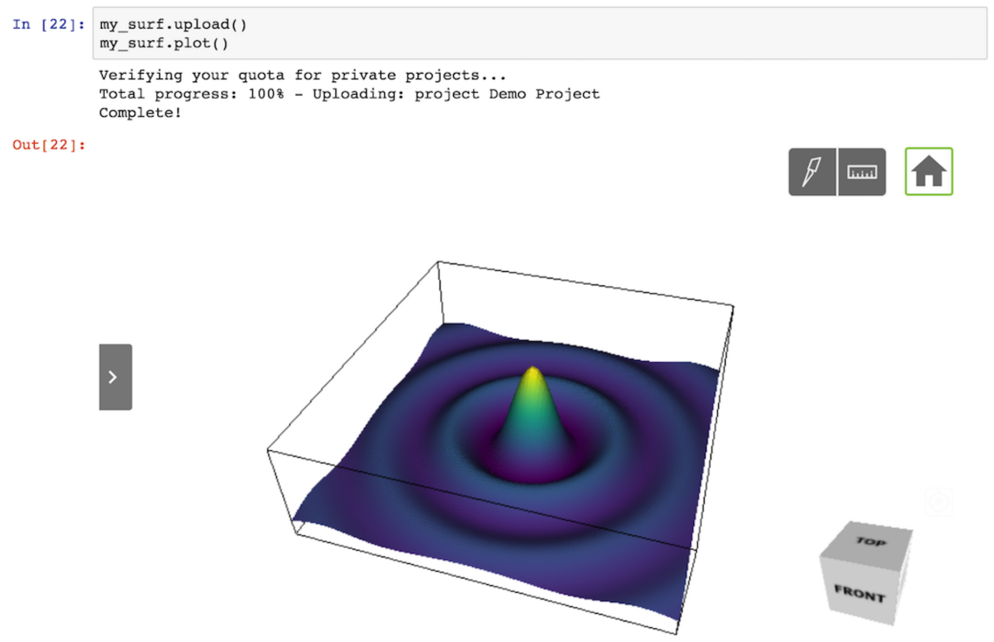
Example Jupyter notebooks can be cloned or you can follow along online with the First Project below

```
git clone https://github.com/3ptscience/steno3d-notebooks.git
```

A First Project

Let's get started using Steno3D. The following demo project is available online in a Jupyter notebook at mybinder.org, no installation required.

Here, we will create a public project containing a surface, upload it, and explore it with Steno3D!



```
In [22]: my_surf.upload()
         my_surf.plot()

         Verifying your quota for private projects...
         Total progress: 100% - Uploading: project Demo Project
         Complete!

Out[22]:
```

Start by importing Steno3D. If you are using the online notebooks your environment should already be set up; otherwise, Steno3D is *easy to install*.

```
>> import steno3d
```

Log In

Next, you need to login using your API developer key. If you do not have a Steno3D account, you can sign up and request a developer key associated with your account.

Then, login using this key within Python. You need to do this step even if you are logged in to steno3d.com; the developer key and your website login are separate.

```
>> steno3d.login('this-is-a-demo-key')
```

Note: By default, your developer key will be saved locally to default credentials file *~/.steno3d_client/credentials*. This allows future logins without manually entering your key:

```
>> steno3d.login()
```

You may specify a different credentials file with:

```
>> steno3d.login('this-is-a-demo-key', credentials_file='/path/to/file')
```

or you may opt to not save the developer key at all with:

```
>> steno3d.login('this-is-a-demo-key', skip_credentials=True)
```

If you ever lose your key, you can always generate a new one.

Create Resources

We start by creating a project

```
>> my_proj = steno3d.Project(
       title='Demo Project',
       description='My first project',
       public=True
   )
```

Here, we will create a topographic surface of a sinc function. We will use numpy to do this.

```
>> import numpy as np
>> topo = lambda X, Y: 50*np.sinc(np.sqrt(X**2. + Y**2.)/20.)
```

Next, we define our x and y coordinates to make the mesh

```
>> x = np.linspace(-100, 100., num=100.)
>> y = np.linspace(-100., 100., num=100.)
>> my_mesh = steno3d.Mesh2DGrid(
       h1=np.diff(x),
       h2=np.diff(y),
       O=np.r_[-100.,-100.,0.]
   )
```

and define the Z vertex topography of the mesh.

```
>> X, Y = np.meshgrid(x, y, indexing='ij')
>> Z = topo(X, Y)
>> my_mesh.Z = Z.flatten()
```

Right now, we have a 2D mesh. Let's create a surface with this mesh geometry.

```
>> my_surf = steno3d.Surface(
       project=my_proj,
       mesh=my_mesh
   )
```

```
>> my_surf.title = 'Sinc Surface'
>> my_surf.description = '3D rendering of sinc function in Steno3D'
```

You may want to put data on the mesh. In this case, we assign topography (same as the Z-values of the mesh) as data on the nodes of the mesh

```
>> my_topo_data = steno3d.DataArray(
       title='Sinc function topography',
       array=my_mesh.Z
   )
>> my_surf.data = [dict(
       location='N',
       data=my_topo_data
   )]
```

Upload

In order to view our 3D data, we first need to upload it. Prior to uploading, you can check that all required parameters are set and valid

```
>> my_surf.validate()
```

and then upload the surface.

```
>> my_surf.upload()
```

This will return a URL where you can view it.

View

There are two options for viewing, if you are using the Jupyter notebook you can plot the surface inline. This allows you to inspect it and make sure it is constructed correctly.

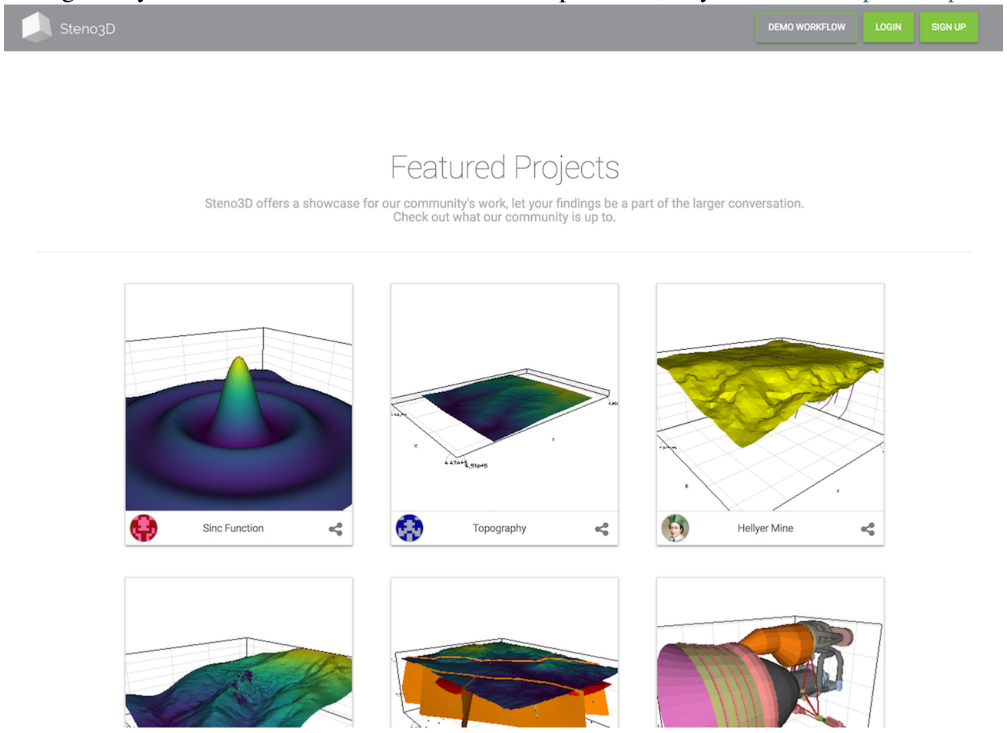
```
>> my_surf.plot()
```

Once you are happy with your upload, use the project URL to view, explore, and share the project on steno3d.com.

```
>> print(my_proj.url)
```

Explore Steno3D

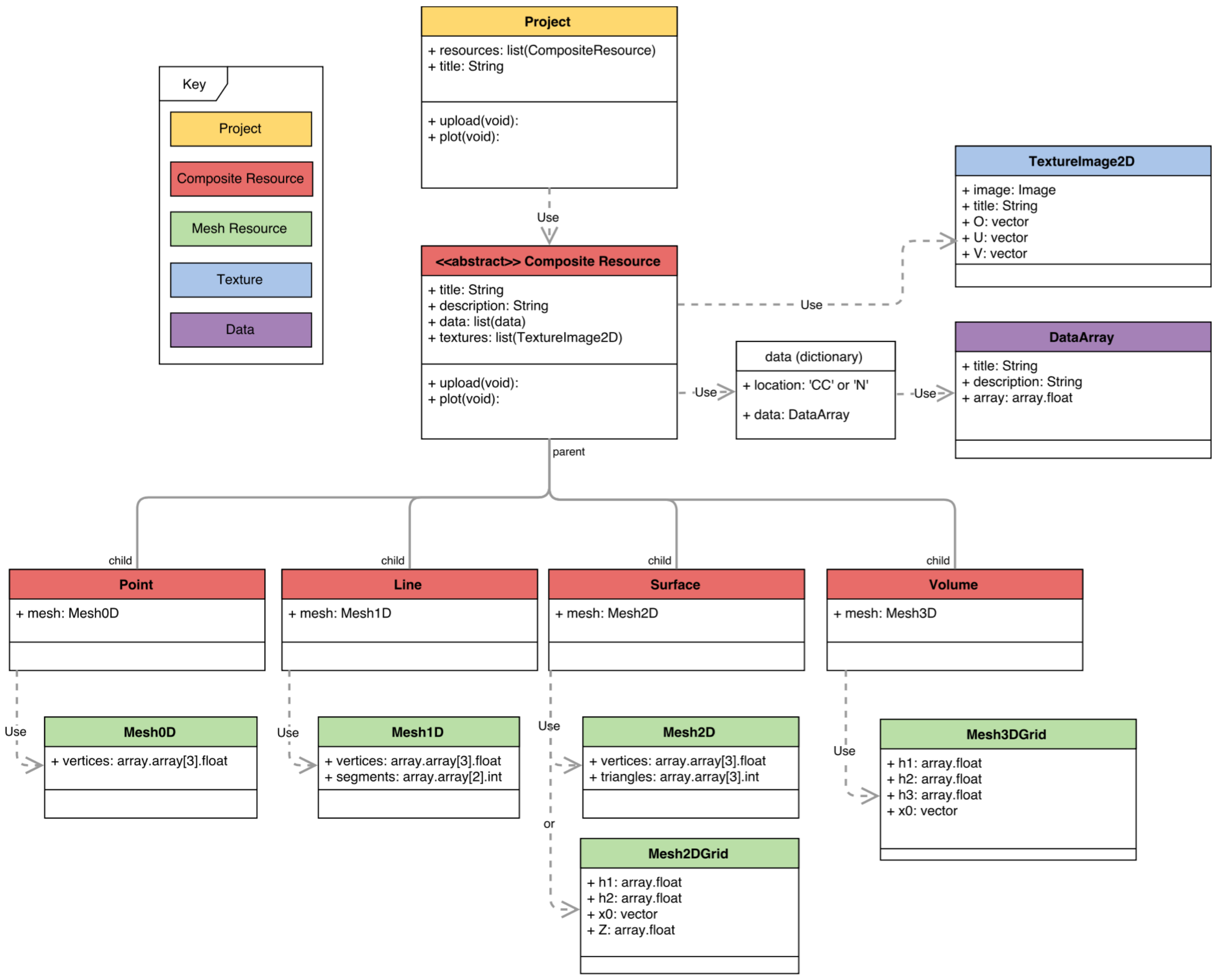
To give you a flavor of Steno3D's capabilities, you can explore public Steno3D projects


Steno3D
DEMO WORKFLOW
LOGIN
SIGN UP
Featured Projects
Steno3D offers a showcase for our community's work, let your findings be a part of the larger conversation. Check out what our community is up to.
Sinc Function
Topography
Hellyer Mine

What is Steno3D

Steno3D is designed for you to explore and share your 3D data. Below is a visual overview of this library. An interactive tutorial is also available online as a Jupyter notebook

Contents



Projects

Projects are groupings of associated resources. Viewing a project allows the user to interact with multiple resources at once.

Resources

Within Steno3D, any object that can be created and uploaded is a resource. This includes basic component structures such as data, meshes, and textures as well as more complex objects like points, lines, surfaces, and volumes. Once in the database, resources are static.

Points, Lines, Surfaces, Volumes

Points, *lines*, *surfaces*, and *volumes* are zero-, one-, two-, and three-dimensional resources, respectively. These resources are composites of pointers to other resources. Specifically, they must contain a mesh resource of the appropriate dimensionality to describe the geometry. They may also contain a number of data or texture resources that correspond to the mesh.

Vectors

Vectors are also a composite resesource. They use the same mesh as points but also include vectors at each point.

Meshes

Mesh resources define spatial structure. Meshes contain nodes and cells (except 0-D which only has nodes). Some types of meshes are built by defining nodes and cells explicitly; others structured meshes are defined more simply, for example 2D grids are constructed from two vectors of cell widths.

Data

Data resources define values that correspond to mesh locations. Data resources are tied to mesh resources within a Point, Line, Surface, or Volume. Data location must be specified as node or cell center, and the length of the data array must equal the number of nodes or cell centers in the associated mesh.

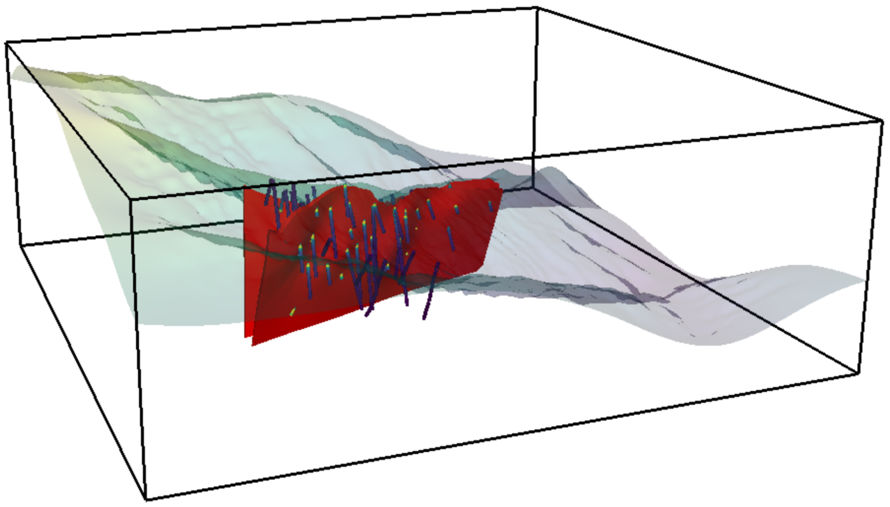
Textures

Texture resources also define values in space. However, they differ from data resources because they do not need to correspond to specific mesh locations. Instead, they are continuous within a domain so values at nodes or cell centers can be extracted. Example textures include 2- or 3-D images or functions dependent on spatial location.

API

The Steno3D API contains tools for making *Projects* and creating and adding *Resources*. Checkout the *Getting Started* for a reference of how to get up and running and *What is Steno3D* for an overview of the elements of Steno3D.

Project API



Steno3D projects organize *resources* so they can be viewed. The steps to construct the project pictured above can be found online in the example notebooks.

class `steno3d.project.`**`Project`**(***metadata*)

Steno3D top-level project

Required Properties:

- **public** (`Bool`): Public visibility of project, a boolean, Default: False
- **resources** (*`a list of CompositeResource`*): Project Resources, a list (each item is an instance of CompositeResource)

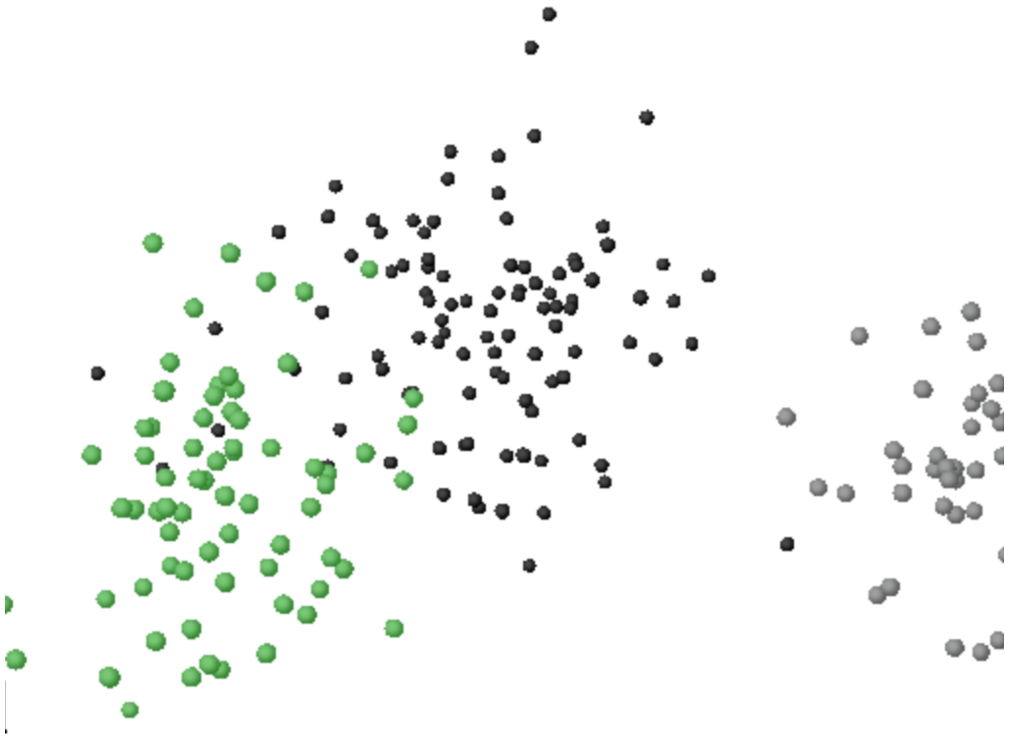
Optional Properties:

- **description** (`String`): Description of the model., a unicode string
- **title** (`String`): Title of the model., a unicode string

Resource API

Resources are elements that you can view. They require a mesh which defines locations in 3D space and may have data and textures.

Point



Steno3D Points are 0D resources. The steps to construct the point resource pictured above can be found online in the example notebooks.

class `steno3d.point.`**`Point`**(*project=None*, ***kwargs*)

Contains all the information about a 0D point cloud

Required Properties:

- **mesh** (*`Mesh0D`*): Mesh, an instance of Mesh0D, Default: new instance of Mesh0D

- **opts** (_`PointOptions`): Options, an instance of _PointOptions, Default: new instance of _PointOptions
- **project** (`a list of Project`): Project containing the resource, a list (each item is an instance of Project)

Optional Properties:

- **data** (`a list of _PointBinder`): Data, a list (each item is an instance of _PointBinder)
- **description** (`String`): Description of the model., a unicode string
- **textures** (`a list of Texture2DImage`): Textures, a list (each item is an instance of Texture2DImage)
- **title** (`String`): Title of the model., a unicode string

Meshes

class `steno3d.point.`**`Mesh0D`**`(**metadata)`
Contains spatial information of a 0D point cloud.

Required Properties:

- **opts** (_`Mesh0DOptions`): Mesh0D Options, an instance of _Mesh0DOptions, Default: new instance of _Mesh0DOptions
- **vertices** (`Array`): Point locations, a list or numpy array of <type 'float'> with shape (*, 3)

Optional Properties:

- **description** (`String`): Description of the model., a unicode string
- **title** (`String`): Title of the model., a unicode string

Data

The intended method of binding data to points is simply using a dictionary containing location (nodes/vertices, 'N', is the only available location for points) and data, a *DataArray*.

```
>> ...
>> my_point = steno3d.Point(...)
>> ...
>> my_data = steno3d.DataArray(
       title='Six Numbers',
       array=[0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
   )
>> my_point.data = [dict(
       location='N',
       data=my_data
   )]
```

Under the surface, this dictionary becomes a `_PointBinder`.

Binding data to Points requires the data array to correspond to mesh vertices.

class `steno3d.point.`**`_PointBinder`**`(**metadata)`
Contains the data on a 0D point cloud

Required Properties:

- **data** (`DataArray`): Data, an instance of DataArray, Default: new instance of DataArray

- **location** (`StringChoice`): Location of the data on mesh, any of "N", Default: N

Options

Similar to data, options are intended to be constructed simply as a dictionary.

```
>> ...
>> my_point = steno3d.Point(...)
>> ...
>> my_point.opts = dict(
       color='red',
       opacity=0.75
   )
```

This dictionary then becomes *_PointOptions*.

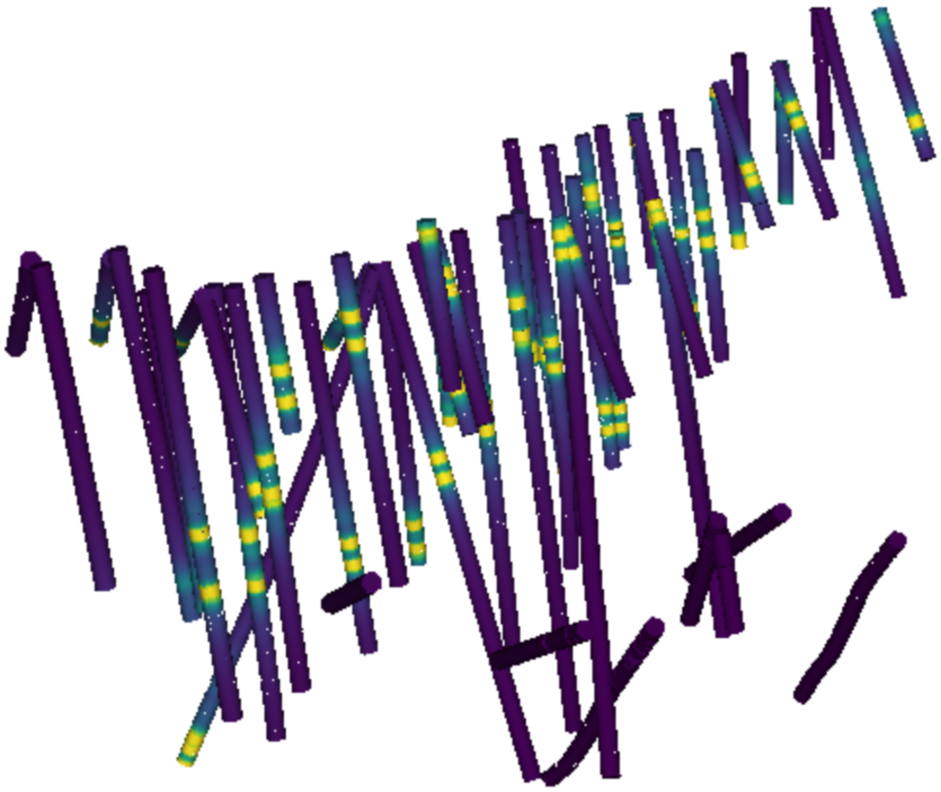
class `steno3d.point.`**`_PointOptions`**`(`*****metadata***`)`

Optional Properties:

- **color** (`Color`): Solid color, a color, Default: random
- **opacity** (`Float`): Opacity, a float in range [0.0, 1.0], Default: 1.0

class `steno3d.point.`**`_Mesh0DOptions`**`(`*****metadata***`)`

Line



Steno3D Lines are 1D resources. The steps to construct the line resource pictured above can be found online in the example notebooks.

class `steno3d.line.`**`Line`**(*project=None*, ***kwargs*)

Contains all the information about a 1D line set

Required Properties:

- **mesh** (*`Mesh1D`*): Mesh, an instance of Mesh1D, Default: new instance of Mesh1D
- **opts** (*`_LineOptions`*): Options, an instance of _LineOptions, Default: new instance of _LineOptions
- **project** (*`a list of Project`*): Project containing the resource, a list (each item is an instance of Project)

Optional Properties:

- **data** (*`a list of _LineBinder`*): Data, a list (each item is an instance of _LineBinder)
- **description** (`String`): Description of the model., a unicode string
- **title** (`String`): Title of the model., a unicode string

Meshes

class `steno3d.line.`**`Mesh1D`**(***metadata*)

Contains spatial information of a 1D line set

Required Properties:

- **opts** (*`_Mesh1DOptions`*): Options, an instance of _Mesh1DOptions, Default: new instance of _Mesh1DOptions
- **segments** (`Array`): Segment endpoint indices, a list or numpy array of <type 'int'> with shape (*, 2)
- **vertices** (`Array`): Mesh vertices, a list or numpy array of <type 'float'> with shape (*, 3)

Optional Properties:

- **description** (`String`): Description of the model., a unicode string
- **title** (`String`): Title of the model., a unicode string

Data

The intended method of binding data to lines is simply using a dictionary containing location (either nodes/vertices, 'N', or cell centers/segments, 'CC') and data, a *DataArray*.

```
>> ...
>> my_line = steno3d.Line(...)
>> ...
>> my_data = steno3d.DataArray(
       title='Six Numbers',
       array=[0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
   )
>> my_line.data = [dict(
       location='N',
       data=my_data
   )]
```

Under the surface, this dictionary becomes a `_LineBinder`.

Binding data to Lines requires the data array to correspond to mesh vertices or mesh segments, for node and cell-center data, respectively.

class `steno3d.line.`**`_LineBinder`**`(`***metadata*`)`

Contains the data on a 1D line set with location information

Required Properties:

- **data** (*`DataArray`*): Data, an instance of DataArray, Default: new instance of DataArray
- **location** (`StringChoice`): Location of the data on mesh, either "CC" or "N"

Options

Similar to data, options are intended to be constructed simply as a dictionary.

```
>> ...
>> my_line = steno3d.Line(...)
>> ...
>> my_line.opts = dict(
       color='red',
       opacity=0.75
   )
```

This dictionary then becomes *_LineOptions*.

class `steno3d.line.`**`_LineOptions`**`(`***metadata*`)`

Optional Properties:

- **color** (`Color`): Solid color, a color, Default: random
- **opacity** (`Float`): Opacity, a float in range [0.0, 1.0], Default: 1.0

class `steno3d.line.`**`_Mesh1DOptions`**`(`***metadata*`)`

Optional Properties:

- **view_type** (`StringChoice`): Display 1D lines or tubes/boreholes/extruded lines, either "line" or "tube", Default: line

Surface

Steno3D Surfaces are 2D resources. The steps to construct the surface resource pictured above can be found online in the example notebooks.

class `steno3d.surface.`**`Surface`**(*project=None*, ***kwargs*)

Contains all the information about a 2D surface

Required Properties:

- **mesh** (*`Mesh2D`*, *`Mesh2DGrid`*): Mesh, an instance of Mesh2D or an instance of Mesh2DGrid, Default: new instance of Mesh2D
- **opts** (*`_SurfaceOptions`*): Options, an instance of _SurfaceOptions, Default: new instance of _SurfaceOptions
- **project** (*`a list of Project`*): Project containing the resource, a list (each item is an instance of Project)

Optional Properties:

- **data** (*`a list of _SurfaceBinder`*): Data, a list (each item is an instance of _SurfaceBinder)
- **description** (`String`): Description of the model., a unicode string
- **textures** (*`a list of Texture2DImage`*): Textures, a list (each item is an instance of Texture2DImage)
- **title** (`String`): Title of the model., a unicode string

Meshes

class `steno3d.surface.`**`Mesh2D`**(***metadata*)

class steno3d.Mesh2D

Contains spatial information about a 2D surface defined by triangular faces.

Required Properties:

- **opts** (*`_Mesh2DOptions`*): Mesh2D Options, an instance of _Mesh2DOptions, Default: new instance of _Mesh2DOptions

- **triangles** (Array): Mesh triangle vertex indices, a list or numpy array of <type 'int'> with shape (*, 3)
- **vertices** (Array): Mesh vertices, a list or numpy array of <type 'float'> with shape (*, 3)

Optional Properties:

- **description** (String): Description of the model., a unicode string
- **title** (String): Title of the model., a unicode string

class `steno3d.surface.`**`Mesh2DGrid`**`(`***metadata*`)`

Contains spatial information of a 2D grid.

Required Properties:

- **O** (Vector3): Origin vector, a 3D Vector of <type 'float'> with shape (3), Default: [0.0, 0.0, 0.0]
- **U** (Vector3): Orientation of h1 axis, a 3D Vector of <type 'float'> with shape (3), Default: X
- **V** (Vector3): Orientation of h2 axis, a 3D Vector of <type 'float'> with shape (3), Default: Y
- **h1** (Array): Grid cell widths, U-direction, a list or numpy array of <type 'float'>, <type 'int'> with shape (*)
- **h2** (Array): Grid cell widths, V-direction, a list or numpy array of <type 'float'>, <type 'int'> with shape (*)
- **opts** (*_Mesh2DOptions*): Mesh2D Options, an instance of _Mesh2DOptions, Default: new instance of _Mesh2DOptions

Optional Properties:

- **Z** (Array): Node topography, a list or numpy array of <type 'float'> with shape (*)
- **description** (String): Description of the model., a unicode string
- **title** (String): Title of the model., a unicode string

Other Properties:

- **x0**: This property has been renamed 'O' and may be removed in the future.

Data

The intended method of binding data to surfaces is simply using a dictionary containing location (either nodes/vertices, 'N', or cell centers/faces, 'CC') and data, a *DataArray*.

```
>> ...
>> my_surf = steno3d.Surface(...)
>> ...
>> my_data = steno3d.DataArray(
       title='Six Numbers',
       array=[0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
   )
>> my_surf.data = [dict(
       location='N',
       data=my_data
   )]
```

Under the surface, this dictionary becomes a `_SurfaceBinder`.

Binding data to a Surface using Mesh2D requires the data array to correspond to mesh vertices or mesh triangles, for node and cell-center data, respectively. When binding data to a Surface using Mesh2DGrid, you may specify data order; the default is C-style, row-major ordering, but Fortran-style, column-major ordering is also available. For more details see the *DataArray documentation*.

class `steno3d.surface.`**`_SurfaceBinder`**`(`***metadata*`)`

Contains the data on a 2D surface with location information

Required Properties:

- **data** (`DataArray`): Data, an instance of DataArray, Default: new instance of DataArray
- **location** (`StringChoice`): Location of the data on mesh, either "CC" or "N"

Options

Similar to data, options are intended to be constructed simply as a dictionary.

```
>> ...
>> my_surf = steno3d.Surface(...)
>> ...
>> my_surf.opts = dict(
        color='red',
        opacity=0.75
   )
```

This dictionary then becomes *_SurfaceOptions*.

class `steno3d.surface.`**`_SurfaceOptions`**`(`***metadata*`)`

Optional Properties:

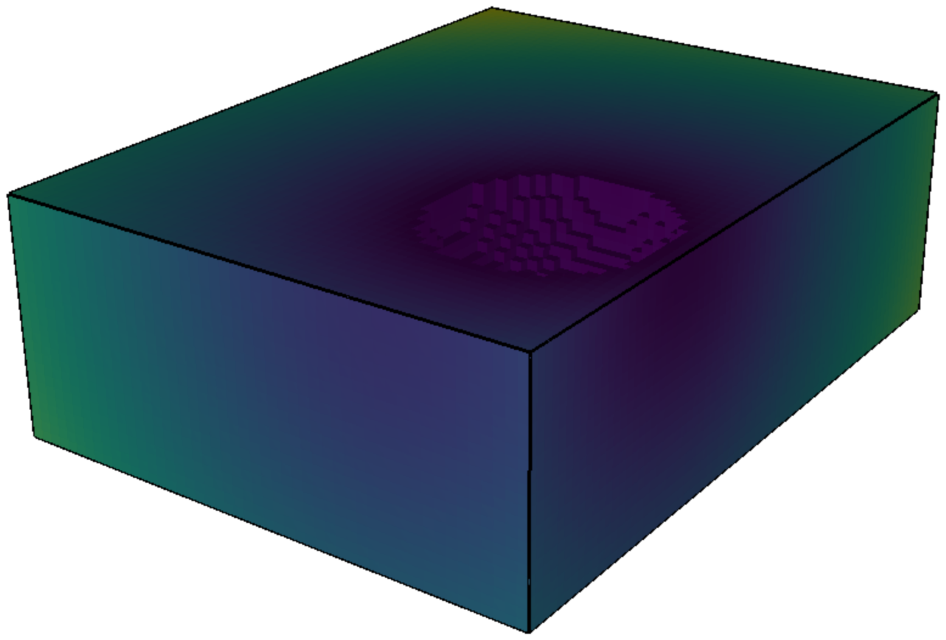
- **color** (`Color`): Solid color, a color, Default: random
- **opacity** (`Float`): Opacity, a float in range [0.0, 1.0], Default: 1.0

class `steno3d.surface.`**`_Mesh2DOptions`**`(`***metadata*`)`

Optional Properties:

- **wireframe** (`Bool`): Wireframe, a boolean, Default: False

Volume



Steno3D Volumes are 3D resources. The steps to construct the volume resource pictured above can be found online in the example notebooks.

class `steno3d.volume.`**`Volume`**(*project=None*, ***kwargs*)

Contains all the information about a 3D volume

Required Properties:

- **mesh** (*Mesh3DGrid*): Mesh, an instance of Mesh3DGrid, Default: new instance of Mesh3DGrid
- **opts** (*_VolumeOptions*): Options, an instance of _VolumeOptions, Default: new instance of _VolumeOptions
- **project** (*a list of Project*): Project containing the resource, a list (each item is an instance of Project)

Optional Properties:

- **data** (*a list of _VolumeBinder*): Data, a list (each item is an instance of _VolumeBinder)
- **description** (`String`): Description of the model., a unicode string
- **title** (`String`): Title of the model., a unicode string

Meshes

class `steno3d.volume.`**`Mesh3DGrid`**(***metadata*)

Contains spatial information of a 3D grid volume.

Required Properties:

- **O** (`Vector3`): Origin vector, a 3D Vector of <type 'float'> with shape (3), Default: [0.0, 0.0, 0.0]
- **U** (`Vector3`): Orientation of h1 axis, a 3D Vector of <type 'float'> with shape (3), Default: X

- **V** (Vector3): Orientation of h2 axis, a 3D Vector of <type 'float'> with shape (3), Default: Y
- **W** (Vector3): Orientation of h3 axis, a 3D Vector of <type 'float'> with shape (3), Default: Z
- **h1** (Array): Tensor cell widths, x-direction, a list or numpy array of <type 'float'>, <type 'int'> with shape (*)
- **h2** (Array): Tensor cell widths, y-direction, a list or numpy array of <type 'float'>, <type 'int'> with shape (*)
- **h3** (Array): Tensor cell widths, z-direction, a list or numpy array of <type 'float'>, <type 'int'> with shape (*)
- **opts** (*_Mesh3DOptions*): Mesh3D Options, an instance of _Mesh3DOptions, Default: new instance of _Mesh3DOptions

Optional Properties:

- **description** (String): Description of the model., a unicode string
- **title** (String): Title of the model., a unicode string

Other Properties:

- **x0**: This property has been renamed 'O' and may be removed in the future.

Data

The intended method of binding data to volumes is simply using a dictionary containing location (cell centers, 'CC', is the only available location for volumes) and data, a *DataArray*.

```
>> ...
>> my_volume = steno3d.Volume(...)
>> ...
>> my_data = steno3d.DataArray(
       title='Six Numbers',
       array=[0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
   )
>> my_volume.data = [dict(
       location='N',
       data=my_data
   )]
```

Under the surface, this dictionary becomes a `_VolumeBinder`.

When binding data to a Volume, you may specify data order; the default is C-style, row-major ordering, but Fortran-style, column-major ordering is also available. For more details see the *DataArray documentation*.

class `steno3d.volume.`**`_VolumeBinder`**`(**metadata)`
Contains the data on a 3D volume with location information

Required Properties:

- **data** (*DataArray*): Data, an instance of DataArray, Default: new instance of DataArray
- **location** (StringChoice): Location of the data on mesh, any of "CC", Default: CC

Options

Similar to data, options are intended to be constructed simply as a dictionary.

```
>> ...
>> my_volume = steno3d.Volume(...)
>> ...
>> my_volume.opts = dict(
       color='red',
       opacity=0.75
   )
```

This dictionary then becomes *_VolumeOptions*.

class `steno3d.volume.`**`_VolumeOptions`**(***metadata*)

Optional Properties:

- **color** (`Color`): Solid color, a color, Default: random
- **opacity** (`Float`): Opacity, a float in range [0.0, 1.0], Default: 1.0

class `steno3d.volume.`**`_Mesh3DOptions`**(***metadata*)

Optional Properties:

- **wireframe** (`Bool`): Wireframe, a boolean, Default: False

Vector

Steno3D Vectors are points in space with an associated direction. Currently, vector magnitude is unsupported, only direction.

class `steno3d.vector.`**`Vector`**(*project=None*, ***kwargs*)

Contains all the information about a vector field

Required Properties:

- **mesh** (*`Mesh0D`*): Mesh, an instance of Mesh0D, Default: new instance of Mesh0D
- **opts** (*`_VectorOptions`*): Options, an instance of _VectorOptions, Default: new instance of _VectorOptions
- **project** (*`a list of Project`*): Project containing the resource, a list (each item is an instance of Project)
- **vectors** (`Array`): Vector, a list or numpy array of <type 'float'> with shape (*, 3)

Optional Properties:

- **data** (*`a list of _PointBinder`*): Data, a list (each item is an instance of _PointBinder)
- **description** (`String`): Description of the model., a unicode string
- **title** (`String`): Title of the model., a unicode string

Meshes

Vectors use the same mesh as Points.

class `steno3d.point.`**`Mesh0D`**(***metadata*)

Contains spatial information of a 0D point cloud.

Required Properties:

- **opts** (*`_Mesh0DOptions`*): Mesh0D Options, an instance of _Mesh0DOptions, Default: new instance of _Mesh0DOptions

- **vertices** (`Array`): Point locations, a list or numpy array of <type 'float'> with shape (*, 3)

Optional Properties:

- **description** (`String`): Description of the model., a unicode string
- **title** (`String`): Title of the model., a unicode string

Data

The intended method of binding data to points is simply using a dictionary containing location (nodes/vertices, 'N', is the only available location for vectors) and data, a *DataArray*.

```
>> ...
>> my_vector = steno3d.Vector(...)
>> ...
>> my_data = steno3d.DataArray(
       title='Six Numbers',
       array=[0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
   )
>> my_vector.data = [dict(
       location='N',
       data=my_data
   )]
```

Under the surface, this dictionary becomes a `_PointBinder` since Vectors use the same mesh as Points.

Binding data to Vectors requires the data array to correspond to mesh vertices.

class `steno3d.point.`**`_PointBinder`**(***metadata*)

Contains the data on a 0D point cloud

Required Properties:

- **data** (*`DataArray`*): Data, an instance of DataArray, Default: new instance of DataArray
- **location** (`StringChoice`): Location of the data on mesh, any of "N", Default: N

Options

Similar to data, options are intended to be constructed simply as a dictionary.

```
>> ...
>> my_vector = steno3d.Vector(...)
>> ...
>> my_vector.opts = dict(
       color='red',
       opacity=0.75
   )
```

This dictionary then becomes *_VectorOptions*.

class `steno3d.vector.`**`_VectorOptions`**(***metadata*)

Optional Properties:

- **color** (`Color`): Solid color, a color, Default: random
- **opacity** (`Float`): Opacity, a float in range [0.0, 1.0], Default: 1.0

class `steno3d.point.`**`_Mesh0DOptions`**(***metadata*)

Data

In Steno3D, binding data to resources requires both a DataArray and a data binder. These binders are documented within each composite resource. The binders provide information about where the data maps to the mesh. In most cases the available mesh locations are ‘N’, nodes, and ‘CC’, cell centers. The easiest way to do this binding is to use a dictionary with entries ‘location’ (either ‘N’ or ‘CC’) and ‘data’ (the DataArray).

Additionally, a resource can have any number of associated data arrays; simply provide a list of these data binder dictionaries. Here is a code snippet to show data binding in action; this assumes the surface contains a mesh with 9 vertices and 4 faces (ie a 2x2 square grid).

```
>> ...
>> my_surface = steno3d.Surface(...)
>> ...
>> my_node_data = steno3d.DataArray(
       title='Nine Numbers',
       array=[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0]
   )
>> my_face_data = steno3d.DataArray(
       title='Four Numbers',
       array=[0.0, 1.0, 2.0, 3.0]
   )
>> my_surface.data = [
       dict(
           location='N',
           data=my_node_data
       ),
       dict(
           location='CC',
           data=my_face_data
        )
   ]
```

Mapping data to a mesh is staightforward for unstructured meshes (those defined by vertices, segments, triangles, etc); the order of the data array simply corresponds to the order of the associated mesh parameter. For grid meshes, however, mapping 1D data array to the 2D or 3D grid requires correctly ordered unwrapping. The default is C-style, row-major ordering, `order='c'`. To align data this way, you may start with a numpy array that is size (x, y) for 2D data or size (x, y, z) for 3D data then use numpy’s `flatten()` function with default order ‘C’. Alternatively, if your data uses Fortran- or Matlab-style, column-major ordering, you may specify data `order='f'`. For in-depth examples of binding data to resources please refer to the example notebooks.

class `steno3d.data.`**`DataArray`**(*array=None*, ***kwargs*)
Data array with unique values at every point in the mesh

Required Properties:

- **array** (`Array`): Data, unique values at every point in the mesh, a list or numpy array of <type ‘float’>, <type ‘int’> with shape (*)
- **order** (`StringChoice`): Data array order, for data on grid meshes, either “c” or “f”, Default: c

Optional Properties:

- **colormap** (`a list of Color`): Colormap applied to data range or categories, a list (each item is a color) with length between 1 and 256
- **description** (`String`): Description of the model., a unicode string
- **title** (`String`): Title of the model., a unicode string

class `steno3d.data.`**`DataCategory`**(*array=None*, ***kwargs*)

Data array with indices and corresponding string categories

For locations with no data, use -1 for index. If colormap is unspecified, colors will be randomized.

Required Properties:

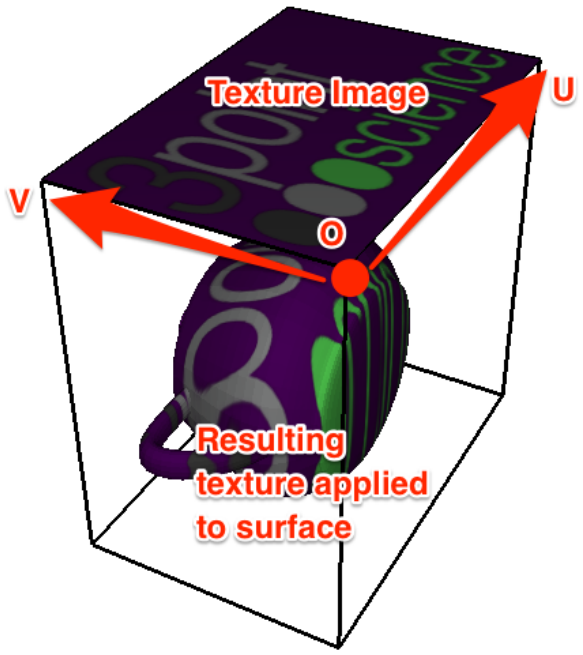
- **array** (`Array`): Category index values at every point in the mesh, a list or numpy array of <type 'int'> with shape (*)
- **order** (`StringChoice`): Data array order, for data on grid meshes, either "c" or "f", Default: c

Optional Properties:

- **categories** (`a list of String`): List of string categories, a list (each item is a unicode string) with length between 1 and 256
- **colormap** (`a list of Color`): Colormap applied to data range or categories, a list (each item is a color) with length between 1 and 256
- **description** (`String`): Description of the model., a unicode string
- **title** (`String`): Title of the model., a unicode string

Texture

In Steno3D, textures are data that exist in space and are mapped to their corresponding resources. Unlike data, they do not need to correspond to mesh nodes or cell centers. This image shows how textures are mapped to a surface. Their position is defined by an origin, O, and axis vectors, U and V, then they are mapped laterally to the resource position.



Like data, multiple textures can be applied to a resource. Simply provide a list of textures.

```
>> ...
>> my_surface = steno3d.Surface(...)
>> ...
>> my_tex_1 = steno3d.Texture2DImage(
       O=[0.0, 0.0, 0.0],
       U=[1.0, 0.0, 0.0],
       V=[0.0, 1.0, 0.0],
       image='image1.png'
   )
>> my_tex_2 = steno3d.Texture2DImage(
       O=[0.0, 0.0, 0.0],
       U=[1.0, 0.0, 0.0],
       V=[0.0, 0.0, 1.0],
       image='image2.png'
   )
>> my_surface.textures = [
       my_tex_1,
       my_tex_2
   ]
```

class `steno3d.texture.`**`Texture2DImage`**`(`***metadata*`)`

Contains an image that can be mapped to a 2D surface

Required Properties:

- **O** (`Vector3`): Origin of the texture, a 3D Vector of <type 'float'> with shape (3)
- **U** (`Vector3`): U axis of the texture, a 3D Vector of <type 'float'> with shape (3)
- **V** (`Vector3`): V axis of the texture, a 3D Vector of <type 'float'> with shape (3)
- **image** (`ImagePNG`): Image file, a PNG image file, valid modes include (u'ab+', u'rb+', u'wb+', u'rb')

Optional Properties:

- **description** (`String`): Description of the model., a unicode string
- **title** (`String`): Title of the model., a unicode string

Base Resource

class `steno3d.base.`**`UserContent`**(***metadata*)

Base class for everything user creates and uploads in steno3d

Optional Properties:

- **description** (`String`): Description of the model., a unicode string
- **title** (`String`): Title of the model., a unicode string

class `steno3d.base.`**`CompositeResource`**(*project=None*, ***kwargs*)

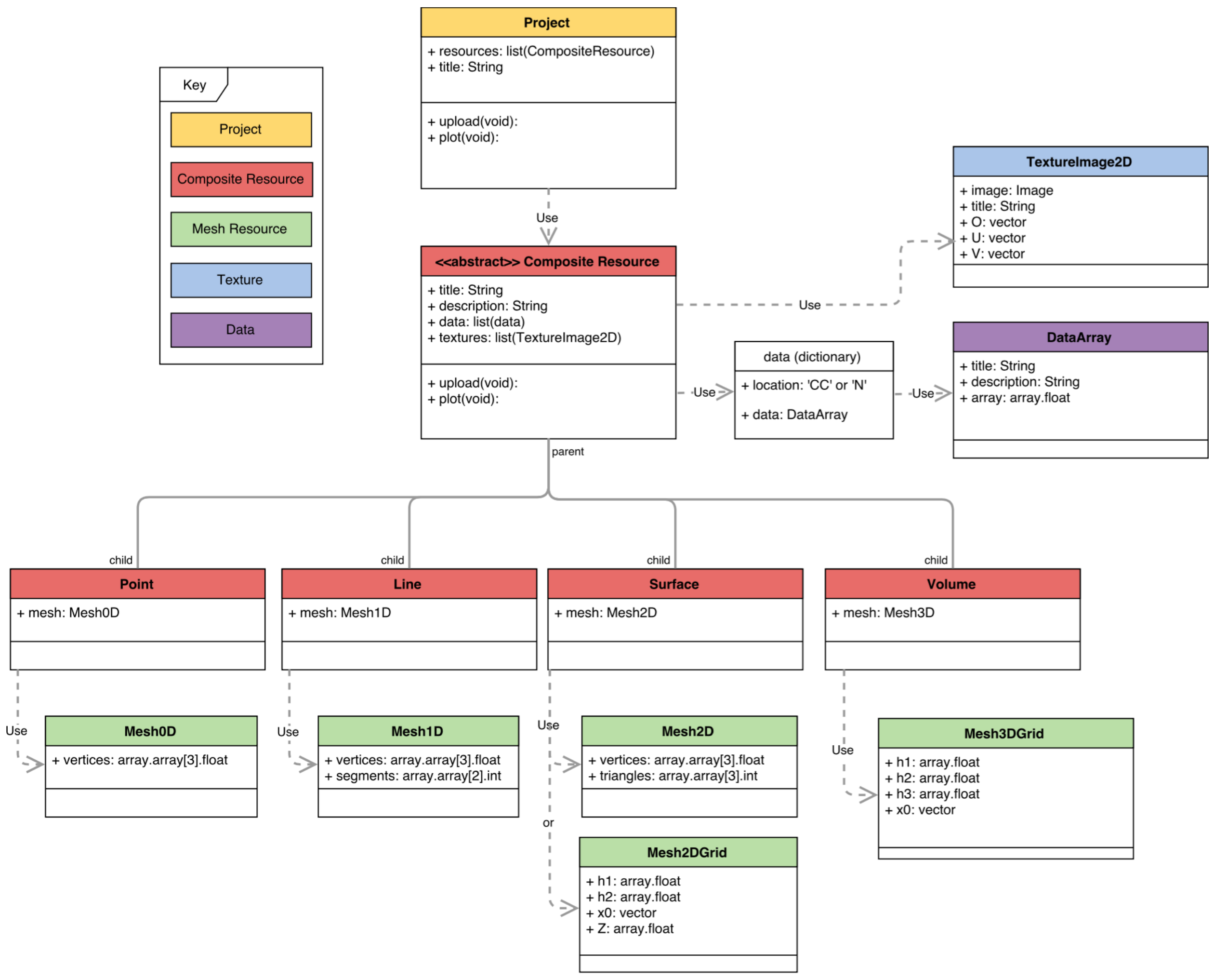
A composite resource that stores references to lower-level objects.

Required Properties:

- **project** (*`a list of UserContent`*): Project containing the resource, a list (each item is an instance of UserContent)

Optional Properties:

- **description** (`String`): Description of the model., a unicode string
- **title** (`String`): Title of the model., a unicode string



Parsers

The Steno3D Python client is designed to support developers by providing an object-oriented, file-type agnostic environment to work with. However, it contains the infrastructure for parsers as plugin modules to suppport file-heavy workflows.

- *Installing and Using Parsers*
- *Links to Parsers*
- *Contributing*

Installing and Using Parsers

An example parser for Wavefront .obj files is available on pip:

```
pip install steno3d_obj
```

or install from source:

```
git clone https://github.com/3ptscience/steno3d-obj.git
python setup.py install
```

Usage of all parsers should be the same as the obj parser. Upon import, the parser is added to the Steno3D namespace and may be instantiated with the name of the input file.

```
>> import steno3d
>> import steno3d_obj
>> obj_parser = steno3d.parsers.obj('/path/to/input/file.obj')
```

If you have many different parsers imported, you may also let Steno3D select the appropriate parser based on file extension using:

```
>> obj_parser = steno3d.parsers.AllParsers('/path/to/input/file.obj')
```

Then, to parse the file into a new Steno3D project:

```
>> (obj_proj,) = obj_parser.parse()
```

You may also parse the file objects directly into an existing Steno3d project.

```
>> my_proj = steno3d.Project(
       title='OBJ File Project'
   )
>> obj_parser.parse(my_proj)
```

Links to Parsers

- obj parser for Wavefront .obj files (github, pip)
- stl parser for binary and ASCII stereolithography .stl files (github, pip)
- Surfer grd file parser for Surfer 6 & 7 binary and ASCII .grd files (github, pip)

Contributing

If there is a 3D file type you would like to see supported or an existing parser you would like expanded, please contribute! The basic guidelines for our parsers are

1. Easy to use by following the steps described *above*
2. Well-documented coverage of the file type (not necessarily supporting every feature, but describing what is supported and raising non-cryptic warnings or errors for unsupported features)
3. Open source, under the MIT license if possible
4. PEP 8 compliant (aside from parser class names) and Python 2/3 compatible

Additional details for writing a parser follow *below*. If you are interested in a new parser parser or have feedback about an existing parser but are unable to contribute, please at least submit an issue to steno3d or the specific parser's github page.

Implementation

The following steps describe how to implement a Steno3D parser. Please refer to the obj parser source code as an example.

Class Definition

Parser classes must inherit `BaseParser` and they must have a tuple of supported extensions:

```
...
import steno3d


class obj(steno3d.parsers.BaseParser):
    """class obj

    Parser class for Wavefront .obj ASCII object files
    """

    extensions = ('obj',)
    ...
```

Doing this adds the parser to the `steno3d.parsers` namespace, adds the extension to the steno3d supported extensions, and ensures that files have the appropriate extension.

In this example, the lowercase class names deviates from PEP 8 style. However, we break this rule to allow for symmetry between class names and file extensions.

Initialization

Initialization is handled by the `BaseParser` `__init__` function. The only required parameter is the file name. Therefore, `self.file_name` is available to any function defined in your parser. There are two initialization hooks:

```
def _validate_file(self, file_name):
    """function _validate_file

    Input:
        file_name - The file to be validated

    Output:
        validated file_name

    _validate_file verifies the file exists and the extension matches
    the parser extension(s) before proceeding. This hook can be
    overwritten to perform different file checks or remove the checks
    entirely as long as it returns the file_name.
    """
```

and

```
def _initialize(self):
    """function _initialize

    _initialize is a hook that is called during parser __init__
    after _validate_file. It can be overwritten to perform any
    additional startup tasks
    """
```

parse()

This function is what the user will call to parse their file, `self.file_name`. The output should be a tuple of Steno3D Projects. It is recommended to allow a Steno3d Project as input so files can be parsed directly into an existing Project. However this behavior is not required if it does not make sense for a certain file type.

Any errors encountered during parsing should raise a `steno3d.parsers.ParseError` with a descriptive error message. This may include unsupported features, unrecognized features, incorrect syntax in the input file, invalid geometry extracted from the file, etc.

Beyond that, the parse function may use anything else necessary to read the file such as helper functions, additional classes you define, or other imported modules.

AllParsers

If a parser class is defined correctly, it will automatically become available to `steno3d.parsers.AllParsers` with its corresponding extension. However, if you are making a large library of related parsers, you may wish to define your own class similar to AllParsers internal to your library. To do this, simply define a class that inherits AllParsers and contains a dictionary of extensions and appropriate parser:

```
class ex1(steno3d.parsers.BaseParser):
    extensions = ('ex1',)
    ...

class ex2(steno3d.parsers.BaseParser):
    extensions = ('ex2',)
    ...

class ex3(steno3d.parsers.BaseParser):
    extensions = ('ex3',)
    ...

class exN(steno3d.parsers.AllParsers):
    extensions = {
        'ex1': ex1,
        'ex2': ex2,
        'ex3': ex3
    }
```

You can then use this as:

```
>> ex1_parser = steno3d.parsers.exN('file.ex1')
>> ex2_parser = steno3d.parsers.exN('file.ex2')
>> ex3_parser = steno3d.parsers.exN('file.ex3')
```

If you run into issues, report them on github.

Release Notes

The latest version of Steno3D is 0.3.2. Detailed release notes are available on github.

Index

- genindex

Index