# voxelmap

***Release 4.2***

**Andrew Garcia, Ph.D.**

**May 04, 2023**

# CONTENTS

CHAPTER

# ONE

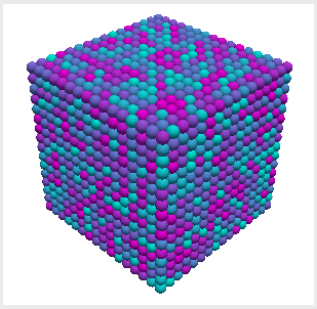
# LET'S MAKE 3-D MODELS WITH PYTHON!

Fig. 1: Check out voxelmap's open-source on GitHub

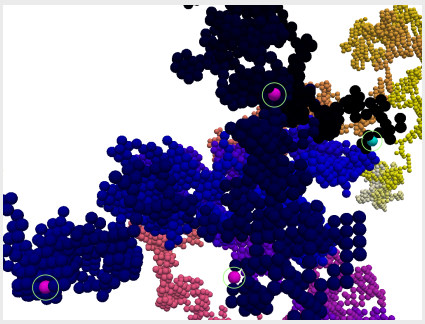
Ever wanted to make simple 3-D models from numpy arrays? Now you can do that with voxelmap ! **Voxelmap** is a Python library for making voxel and three-dimensional models from NumPy arrays. It was initially made to streamline 3-D voxel modeling by assigning each integer in an array to a voxel. Now, methods are being developed for mesh representations, such as **ImageMesh**, voxel-to-mesh transformation and vice-versa.

Check out the *Usage* section for further information, including how to *Installation* the project. For some quick examples / templates, check out the next section.

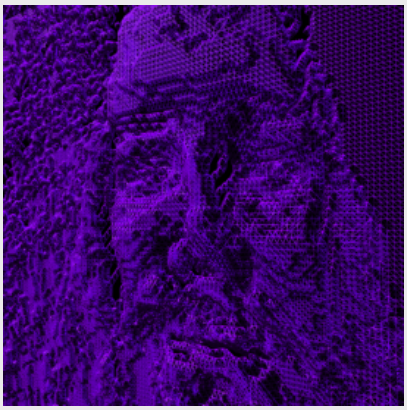
## 1.1 Key Features



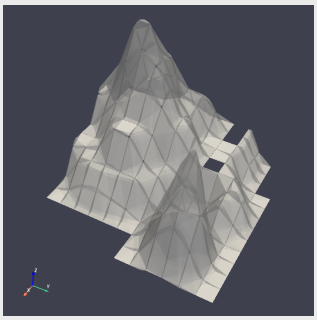
*Draw voxels from an integer array*



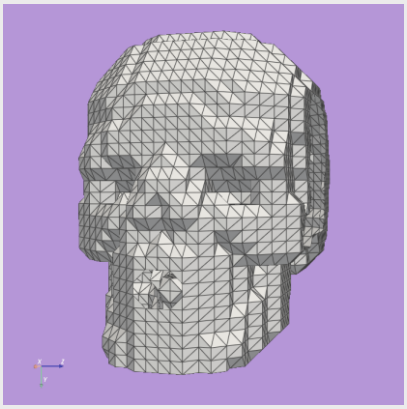
*Colormap Block Coloring*



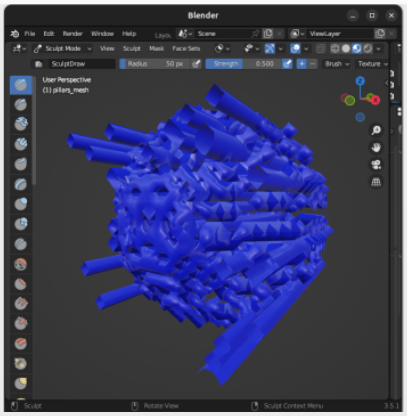
*3-D Reconstruction*



*Low-Poly 3-D Mesh Mapping from Image*



*Voxel Model to 3-D Mesh*




*3-D Model Numpy Manipulation and Blender Integration*

## 1.2 Colab Notebook

While we offer an interactive tutorial via a Colab notebook, we recommend using this site as your primary guide instead. Please note that the Colab notebook is currently out-of-date and may not reflect the most recent features and functionality of Voxelmap. We are continually updating our website to ensure the latest information and resources are available to our users.

**Note:** This project is under active development.

CHAPTER

TWO

# CONTENTS

## 2.1 Usage

### 2.1.1 Installation

It is recommended you use voxelmap through a virtual environment. You may follow the below simple protocol to create the virtual environment, run it, and install the package there:

```
$ virtualenv venv
$ source venv/bin/activate
(.venv) $ pip install voxelmap
```

To exit the virtual environment, simply type `deactivate`. To access it at any other time again, enter with the above `source` command.

### 2.1.2 The `hashblocks` Constructor Variable

**Voxelmap** was originally made to handle third-order integer arrays of the form `np.array((int,int,int))` as blueprints to 3-D voxel models.

While **"0"** integers are used to represent empty space, **non-zero integers** are used to define a distinct voxel type and thus, they are used as keys for such voxel type to be mapped to a specific color and `alpha` transparency. These keys are stored in a map (also known as "dictionary") internal to the `voxelmap.Model` class called `hashblocks`.

The `hashblocks` dictionary contains an entry for each integer key, where the corresponding value is a list. This list contains two elements: a string that represents the color in either the hex format (#rrggbb) or as a color label (e.g., 'red') as the first element, and a float between 0 and 1 that represents the alpha transparency as the second element. The dictionary's structure and call method are shown below:

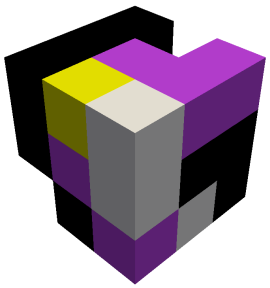
```
import voxelmap as vxm

model = vxm.Model()

model.hashblocks = {
   key_1 (int): ['#rrggbb' (string) ,alpha (float)]
   key_2 (int): ['#rrggbb' (string) ,alpha (float)]
   .
   .
   .
}
```

### 2.1.3 Draw voxels from an integer array

The voxel color and transparencies may be added or modified to the `hashblocks` map with the `hashblocks_add` method.

```
import voxelmap as vxm
import numpy as np

#make a 3x3x3 integer array with random values between 0 and 9
array = np.random.randint(0, 10, (3, 3, 3))
print(array)

#incorporate array to Model structure
model = vxm.Model(array)

#add voxel colors and alpha-transparency for integer values 0 - 9 (needed for `custom`␣
↪coloring)
colors = ['#ffffff', 'black', '#ffffff', 'k',
          'yellow', '#000000', 'white', 'k', '#c745f8']
for i in range(9):
   model.hashblocks_add(i+1, colors[i])

#draw array as a voxel model with `custom` coloring scheme
model.draw('custom', background_color='#ffffff')
```

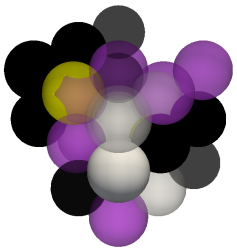
```
>>> [Out]
[[[3 8 5]
  [0 2 6]
  [2 2 7]]
 [[8 3 6]
  [7 2 0]
  [2 2 1]]
 [[9 2 4]
  [8 5 7]
  [8 9 8]]]
```



#### With particles geometry and user-defined `alpha` transparencies

The new version of voxelmap now has a `geometry` kwarg for the `Model.draw()` method where the voxel geometry can be chosen between *voxels* and *particles* form. Below we change it to *particles* to represent the voxels above as spherical objects. In addition, we declare different transparencies of the different voxel-item types:

```
alphas = [0.8,1,0.5,1,0.75,0.5,1.0,0.8,0.6]
for i in range(9):
model.hashblocks_add(i+1,colors[i],alphas[i])
model.draw('custom', geometry='particles', background_color='#ffffff')
```



### 2.1.4 Draw voxels from coordinate arrays

**Voxelmap** may also draw a voxel model from an array which defines the coordinates for each of the voxels to be drawn in x y and z space.

The internal variable `data.xyz` will thus take a third-order array where the rows are the number of voxels and the columns are the 3 coordinates for the x,y,z axis. Another internal input, `data.rgb`, can be used to define the colors for each of the voxels in the `data.xyz` object in `'xxxxxx'` hex format (i.e. `'ffffff'` for white).

The algorithm will also work for negative coordinates, as it is shown in the example below.

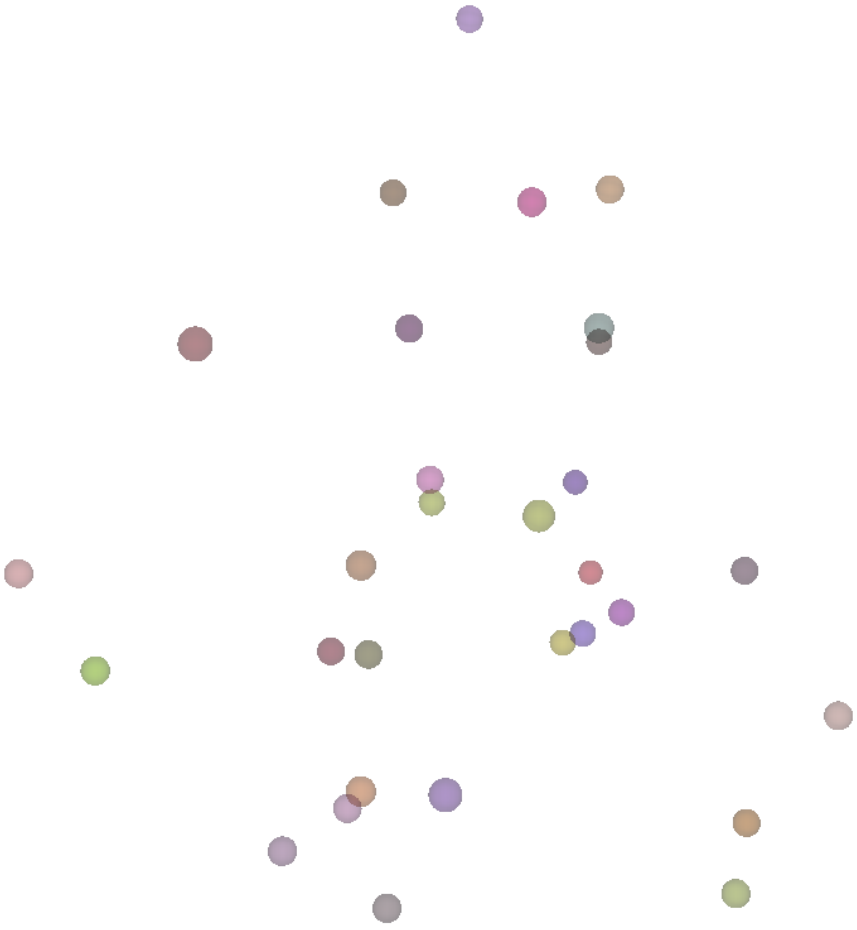
```
import voxelmap as vxm
import numpy as np

cubes = vxm.Model()
num_voxels = 30
cubes.XYZ = np.random.randint(-1,1,(num_voxels,3))+np.random.random((num_voxels,3))
↪    # random x,y,z locs for 10 voxels
cubes.RGB = [ hex(np.random.randint(0.5e7,1.5e7))[2:] for i in range(num_voxels) ]   #␣
↪define random colors for the 10 voxels
cubes.sparsity = 5
                                                    # spaces out coordinates
cubes.load(coords=True)
cubes.hashblocks

for i in cubes.hashblocks:
   cubes.hashblocks[i][1] = 0.30     # update all voxel alphas (transparency) to 0.3

# print(cubes.XYZ)                                  # print the xyz coordinate data
cubes.draw('custom',geometry='particles', background_color='#ffffff',window_size=[416,␣
↪416])                              # draw the model from that data
```

```
>>> [Out]
Color list built from file!
Model().hashblocks =
 {1: ['#4db692', 1], 2: ['#564bfb', 1], 3: ['#5915c1', 1], 4: ['#6283df', 1], 5: ['
↪#6e5722', 1], 6: ['#6eebc3', 1], 7: ['#70cffa', 1], 8: ['#787ea7', 1], 9: ['#813c5b',␣
↪1], 10: ['#8906d7', 1], 11: ['#8a871d', 1], 12: ['#8ba24f', 1], 13: ['#930979', 1],␣
↪14: ['#932fde', 1], 15: ['#964c67', 1], 16: ['#9bafea', 1], 17: ['#9c248b', 1], 18: ['
↪#9e5fff', 1], 19: ['#a2183b', 1], 20: ['#a248a6', 1], 21: ['#a63265', 1], 22: ['#a6c6a1
↪', 1], 23: ['#aa381b', 1], 24: ['#ae9c6a', 1], 25: ['#b58c2c', 1], 26: ['#c114a1', 1],␣
↪27: ['#c618df', 1], 28: ['#d15d6e', 1], 29: ['#da6f7d', 1], 30: ['#e36ff6', 1]}
```

### Increase sparsity

The *sparsity* variable will extend the distance from all voxels at the expense of increased memory.

```
cubes.sparsity = 12                                                    # spaces out␣
↪coordinates
cubes.load(coords=True)
for i in cubes.hashblocks:
   cubes.hashblocks[i][1] = 0.30     # update all voxel alphas (transparency) to 0.3

cubes.draw('custom', geometry='particles', background_color='#ffffff',window_size=[1000,␣
↪1000])                            # draw the model from that data
```



### 2.1.5 Colormap Block Coloring

The coloring kwarg for the draw method now has a `'cmap'` string option to assign colors from a colormap to the defined voxel types (i.e. the non-zero integers in the 3-D arrays). Download the RANDOMWALK.JSON file and save it in the same directory where you are running these examples. If you inspect the .json file, you'll see the following structure:

```
{
   "hashblocks": {},
   "size": [300, 300, 300],
   "coords": [
      [146, 149, 152],
      [146, 150, 152],
```

(continues on next page)

(continued from previous page)

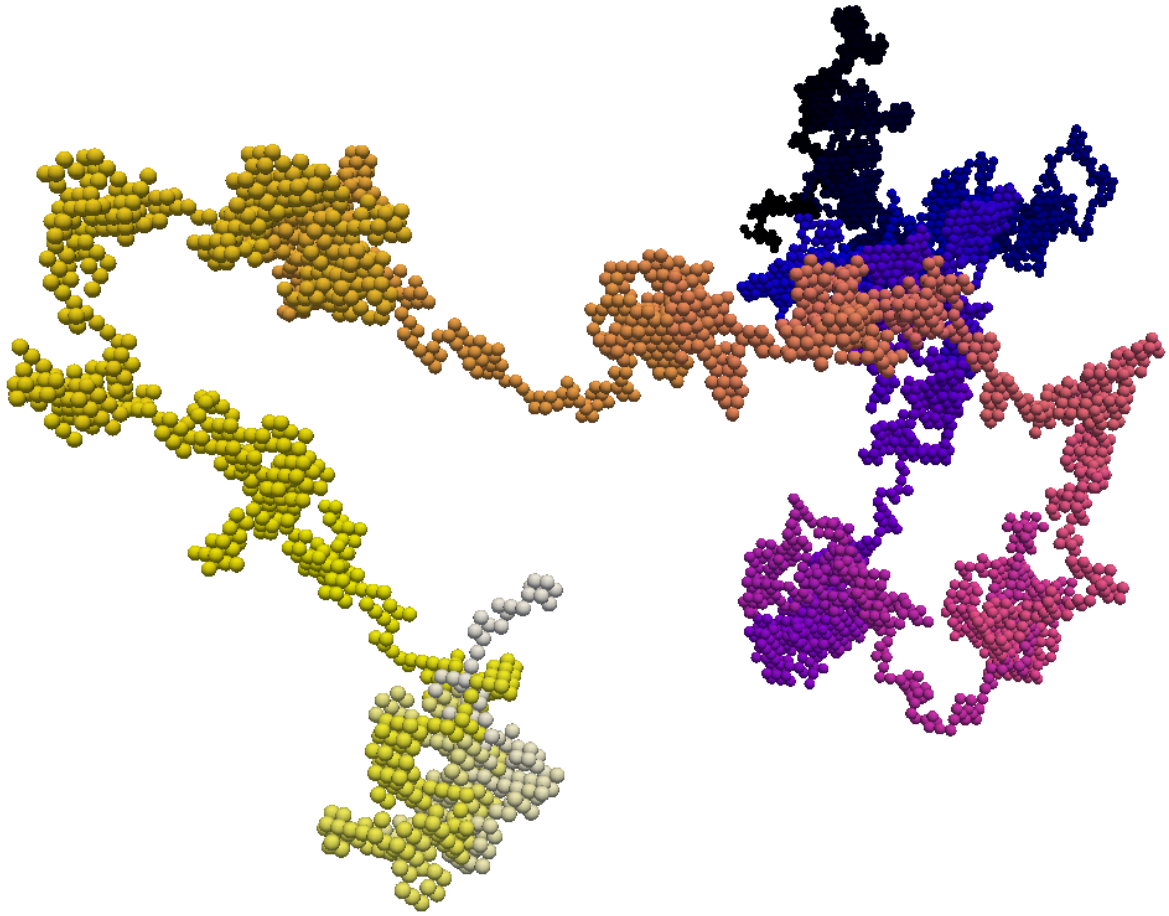
```
      [147, 148, 153],
      [147, 148, 154],
      .
      .
      .
      [197, 142, 132],
      [197, 143, 132]
   ],
   "val": [7, 6, 24, 25, . . ., 3182, 3183]
}
```

The file contains information about a 3-D array, including its dimensions (specified by the `size` key), the 3-D coordinates of its non-zero integers (specified by the `coords` key), and the corresponding integer values (specified by the `val` key). It's worth noting that the hashblocks dictionary is currently empty.

When using the 'cmap:' option to color the array, the **hashblocks** dictionary is built based on a linear relation between the chosen colormap and the values of the integers in the array, similar to how a gradient coloring would work. In the code block below, the RANDOMWALK.JSON model is drawn using the **gnuplot2** colormap with an **alpha** of 1.

Note that this file generates a model that requires a substantial amount of memory, which may result in a longer rendering time.

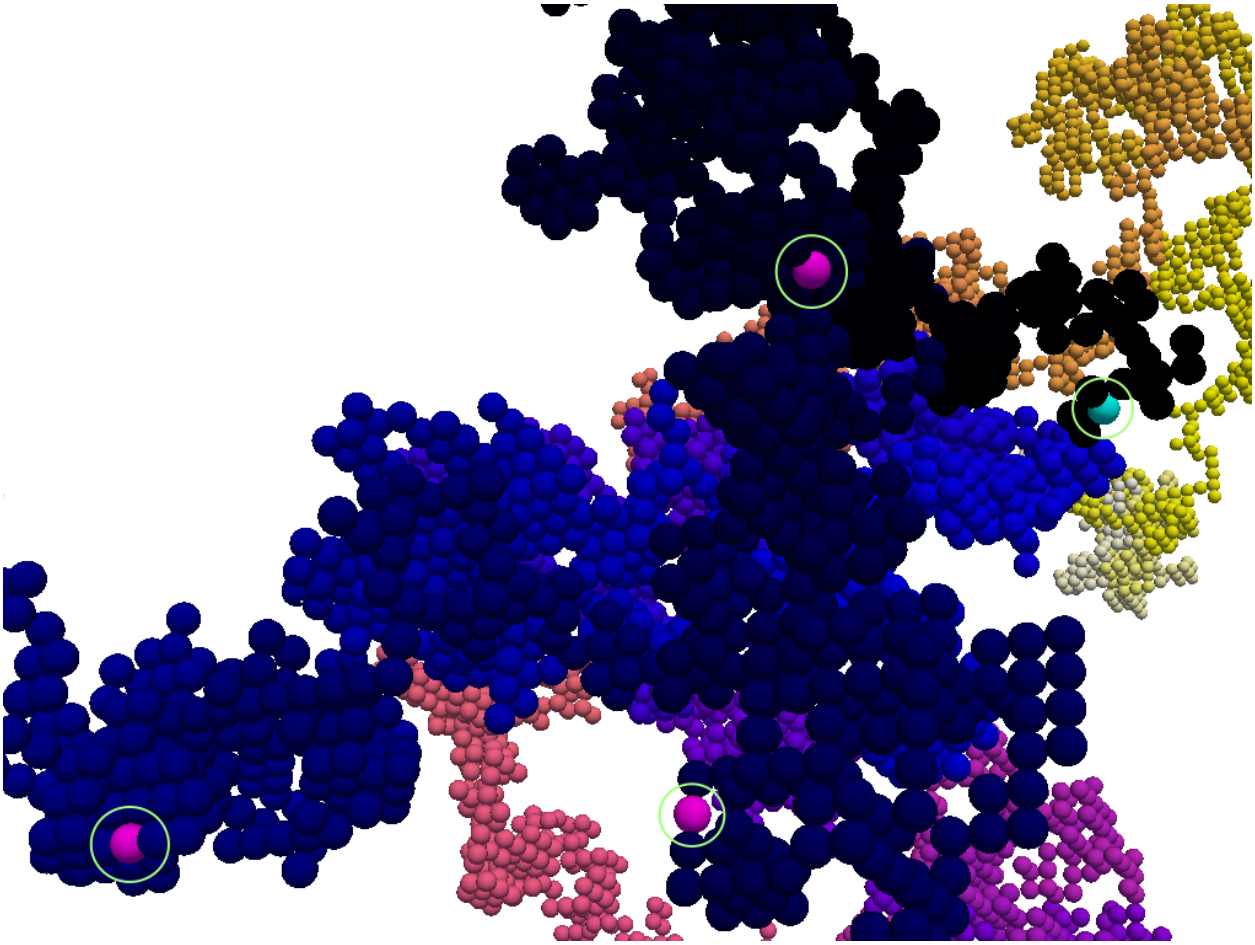
```
model = vxm.Model()

model.load('randomwalk.json')
model.draw(coloring='cmap: gnuplot2, alpha:1',geometry='particles',background_color='w')
```



### Colormap Block Coloring with Integer Tagging

The **hashblocks** map can be used in combination with the 'cmap' coloring option to selectively color-tag specific voxels based on their type. To accomplish this, we can modify the previous code block by adding a model.hashblocks declaration after model.load(). This will allow us to color the voxels represented by the integers 743, 500, and 256 in magenta, and the voxel represented by integer 2 in cyan.

```
model = vxm.Model()

model.load('randomwalk.json')

# add the hashblocks voxel assignments
model.hashblocks = {
  743: ['magenta',1],
  500: ['magenta',1],
  256: ['magenta',1],
  2: ['cyan',1]
}

model.draw(coloring='cmap: gnuplot2, alpha:1',geometry='particles',background_color='w')
```



The ability to use the hashblocks map with the 'cmap' coloring option can be a valuable feature when we need to represent multiple relationships simultaneously. For example, we can use a gradient of values described by the colormap to color the array, while simultaneously highlighting specific voxels with hashblocks. This technique can have numerous applications in fields such as 3-D modeling, medical imaging, and coarse-grained molecular modeling, among others.

### 2.1.6 3-D Mapping of an Image

Here we map the synthetic topography LAND IMAGE (.png) to a 3-D model using the `ImageMap` method from the `voxelmap.Model` class.

```
#import packages
import cv2
import matplotlib.pyplot as plt

plt.imshow(cv2.imread('docs/img/land.png'))      # display fake land topography .png␣
↪file as plot
plt.axis('off')
plt.show()

#import packages
import numpy as np
from matplotlib import cm
```

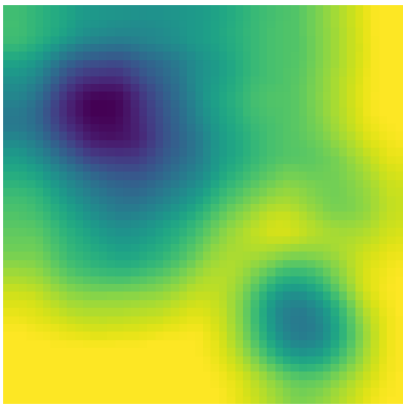
(continues on next page)

(continued from previous page)

```
model = vxm.Model(file='docs/img/land.png')            # incorporate fake land␣
↪topography .png file to voxelmap.Image class
print(model.array.shape)
```

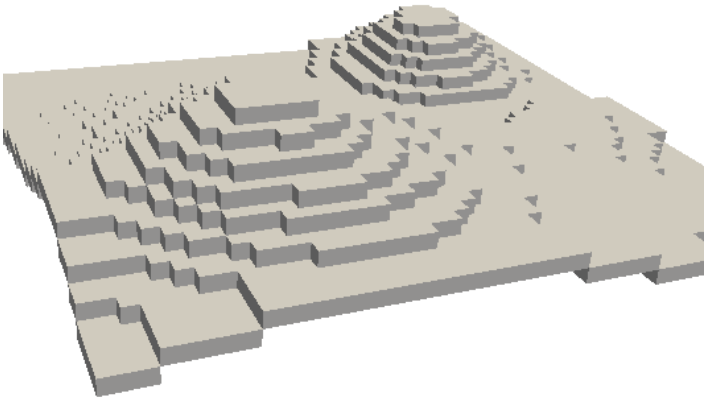
The image is then resized for the voxel draw with the matplotlib method i.e. `Model().draw_mpl`. This is done with `cv2.resize`, resizing the image from 1060x1060 to 50x50. After resizing, we convolve the image to obtain a less sharp color shift between the different gray regions with the `cv2.blur` method:

```
model.array = cv2.resize(model.array, (50,50), interpolation = cv2.INTER_AREA)
print(model.array.shape)

model.array = cv2.blur(model.array,(10,10))    # blur the image for realiztic topography␣
↪levels
plt.imshow(model.array)      # display fake land topography .png file as plot
plt.axis('off')
plt.show()
```



After this treatment, the resized and blurred image is mapped to a 3-D voxel model using the *ImageMap* method from the *Model* class:

```
model.array = model.ImageMap(12)              # mapped to 3d with a depth of 12 voxels
print(model.array.shape)

model.draw('none',background_color='#ffffff')
```



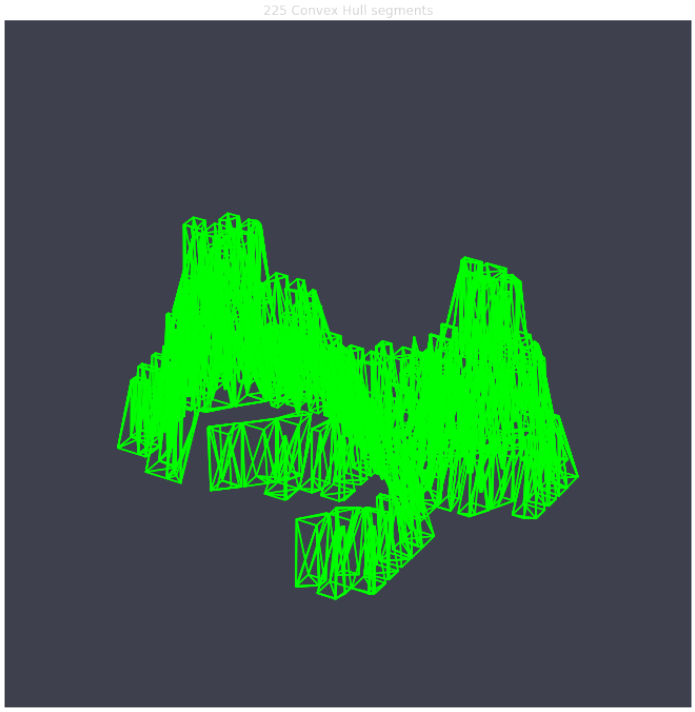
### 2.1.7 Low-Poly 3-D Mesh Mapping from Image

The ImageMesh method creates a low-poly mesh model from an Image using an algorithm developed by Andrew Garcia where 3-D convex hull is performed on separate "cuts" or sectors from the image (see: *ImageMesh : A Convex Hull based 3D Reconstruction Method*).

This can decrease the size of the 3-D model and the runtime to generate it significantly, making the runtime proportional to the number of sectors rather than the number of pixels. Sectors are quantified with the L_sectors kwarg, which is the length scale for the number of sectors in the grid.

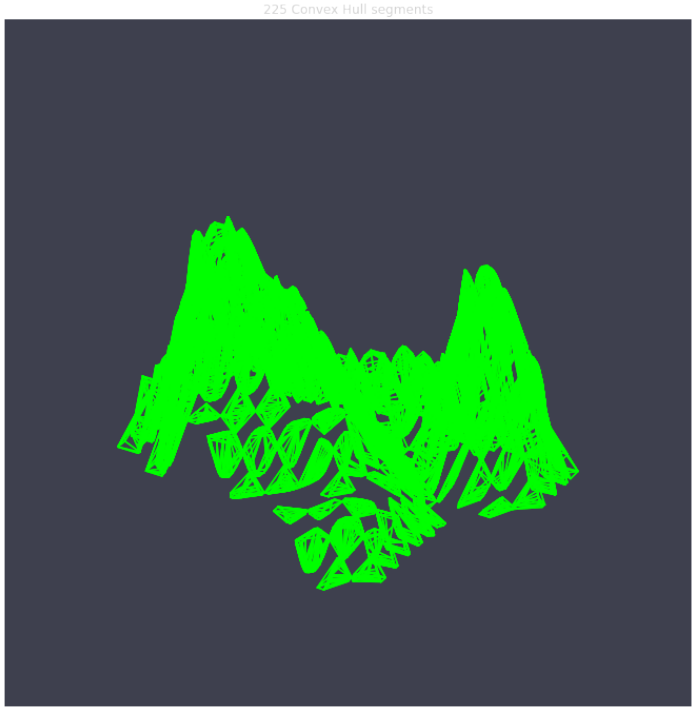
We can see that the mesh model can be calculated and drawn with matplotlib `plot=mpl` option even from a large image of 1060x1060 without resizing:

```
import voxelmap as vxm
import cv2

model = vxm.Model(file='docs/img/land.png')    # incorporate fake land topography .
↪png file

print(model.array.shape)

model.ImageMesh(out_file='scene.obj', L_sectors = 15, trace_min=5, rel_depth = 20,␣
↪figsize=(15,12), plot='mpl')
```



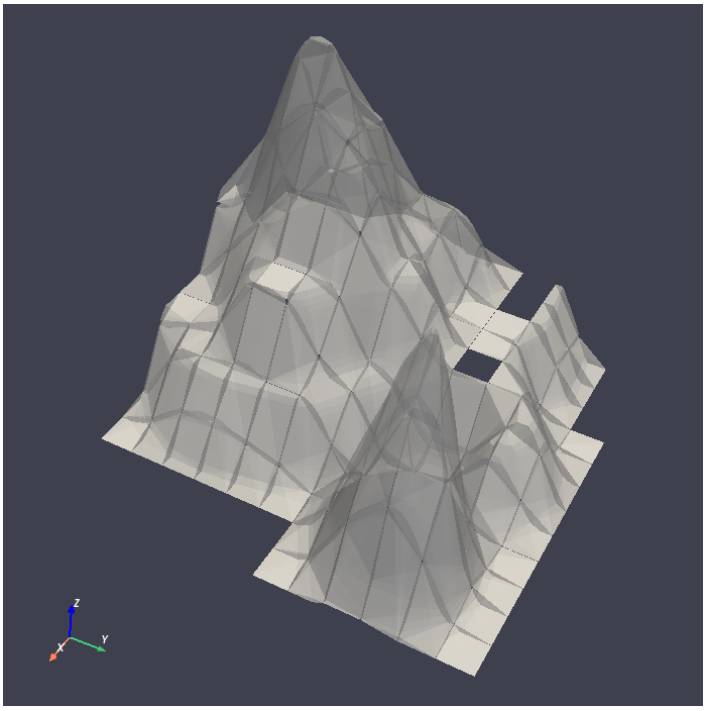

This `ImageMesh` transformation is also tested with a blurred version of the image with `cv2.blur`. A more smooth low-poly 3-D mesh is generated with this additional treatment. The topography seems more realistic:

```
model.array = cv2.blur(model.array,(60,60))    # blur the image for realiztic topography␣
↪levels
model.ImageMesh(out_file='scene.obj', L_sectors = 15, trace_min=5, rel_depth = 20,␣
↪figsize=(15,12), plot='mpl')
```



For a more customizable OpenGL rendering, `img.MeshView()` may be used on the above image:

```
import voxelmap as vxm
import numpy as np
import cv2 as cv

model = vxm.Model(file='docs/img/land.png')           # incorporate fake land topography␣
↪.png file
model.array = cv.blur(model.array,(100,100))    # blur the image for realistic␣
↪topography levels

# model.make()                                  # resized to 1.0x original size i.e. not␣
↪resized (default)

model.ImageMesh('land.obj',  12, 14, 1, False, figsize=(10,10))

model.MeshView( alpha=0.7,background_color='#3e404e',color='white',viewport=(700, 700))
```



### 2.1.8 MarchingMesh : Turning Voxel Models to 3-D Mesh Representations

Click on the links below to save the files in the same directory you are running these examples:

DOG MODEL (.txt)

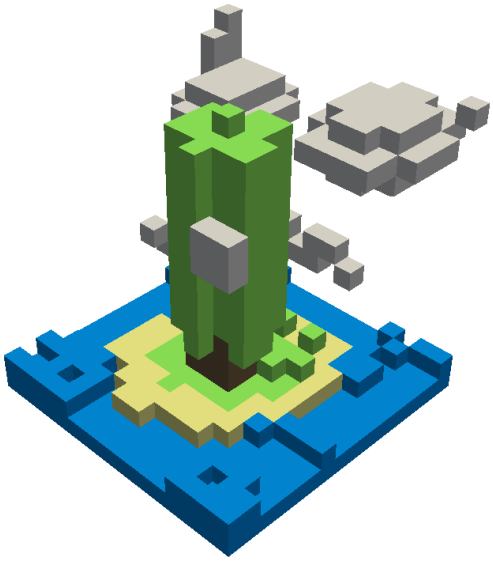
ISLAND MODEL (.txt)

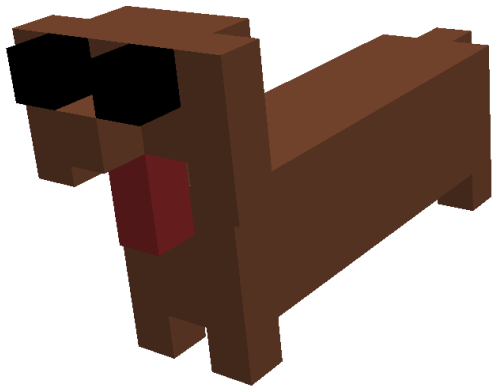
The `.txt` files you downloaded were exported from Goxel projects.

Goxel is an open-source and cross-platform voxel editor which facilitates the graphical creation of voxel models. More information by clicking the icon link below.


We first load those `.txt` files with the below voxelmap methods:

```
import voxelmap as vxm
import numpy as np

'''process argisle.txt from Goxel'''
theIsland = vxm.Model()
theIsland.load('argisle.txt')
theIsland.array = np.transpose(theIsland.array,(2,1,0))    #rotate island
theIsland.draw('custom',background_color='white')

'''process dog.txt from Goxel'''
Dog = vxm.Model()
Dog.load('dog.txt')
Dog.array = np.transpose(Dog.array,(2,1,0))     #rotate dog
Dog.draw('custom',background_color='white')
```





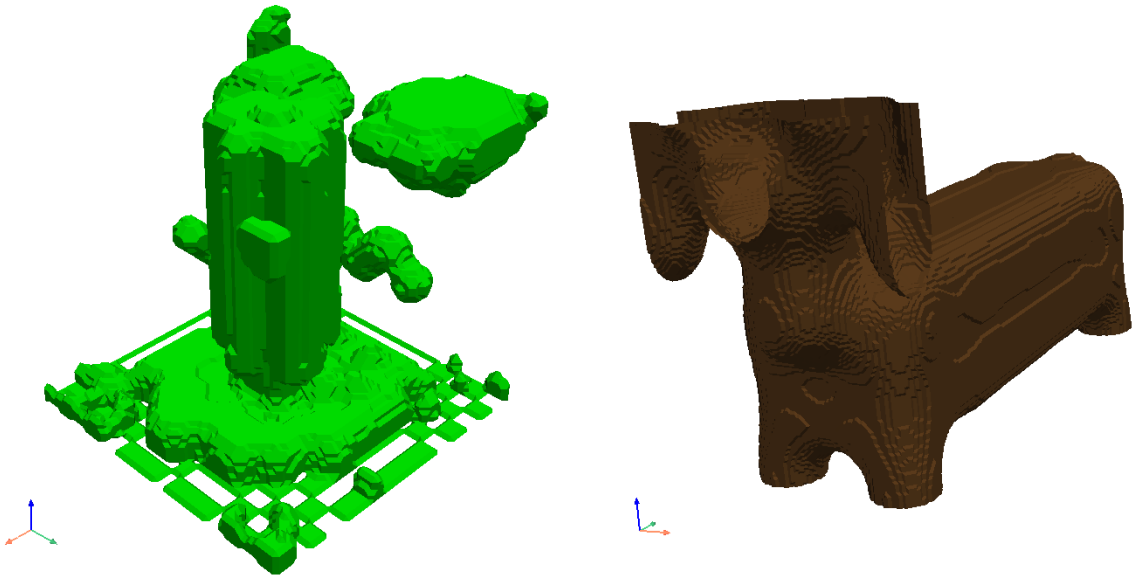
The voxel models can be transformed to 3D mesh representations with voxelmap's `Model().MarchingMesh` method, which uses *Marching Cubes* from the `scikit-image` Python library.

```
'''MarchingMesh on island model'''
theIsland.array = vxm.resize_array(theIsland.array,(5,5,5)) #make array larger before␣
↪mesh transformation
theIsland.MarchingMesh()
theIsland.MeshView(color='lime',wireframe=False,background_color='white',alpha=1,
↪viewport=[700,700])

'''MarchingMesh on dog model'''
Dog.array = vxm.resize_array(Dog.array,(20,20,20)) #make array larger before mesh␣
```

(continues on next page)

(continued from previous page)

```
↪transformation
Dog.MarchingMesh()
Dog.MeshView(color='brown',wireframe=False,background_color='white',alpha=1,
↪viewport=[700,700])
```



Notice the `self.array` arrays were resized in both objects with the global `voxelmap.resize_array` method. This was done to avoid the formation of voids that you still see on the dog mesh above. The `MarchingMesh` method has a current limitation on small voxel models with low detail. It is not perfect, but this is an open-source package and it can always be developed further by the maintainer and/or other collaborators.

### 2.1.9 Wavefront (.obj) file to 3-D Sparse Array

Voxelmap provides various features for converting Wavefront .obj files to its 3-D sparse arrays. However, the most straightforward approach is to use the global **voxelmap.objcast** method.

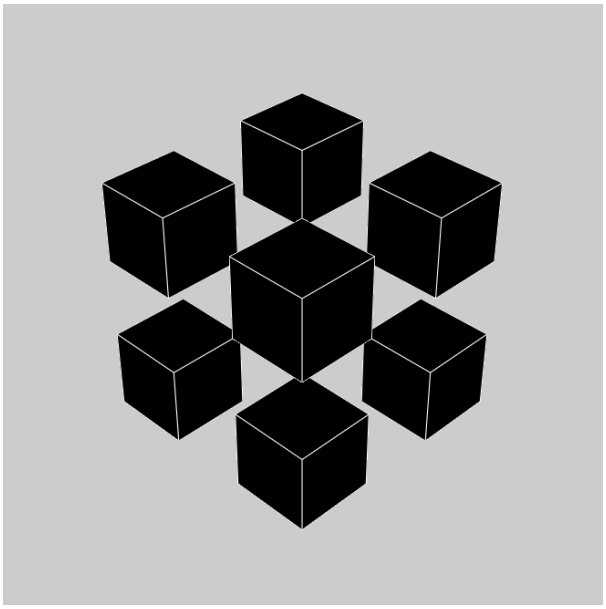
#### Cube Model

To begin, let's download the simple_cube.obj model file. If you take a look at the file, you'll see that it specifies the z, y, and x coordinates of each vertex. However, in order to convert this model to a voxel-based representation, we need to define the spacing between points, since we're transforming from continuous coordinates to the discrete dimensions of a tensor.

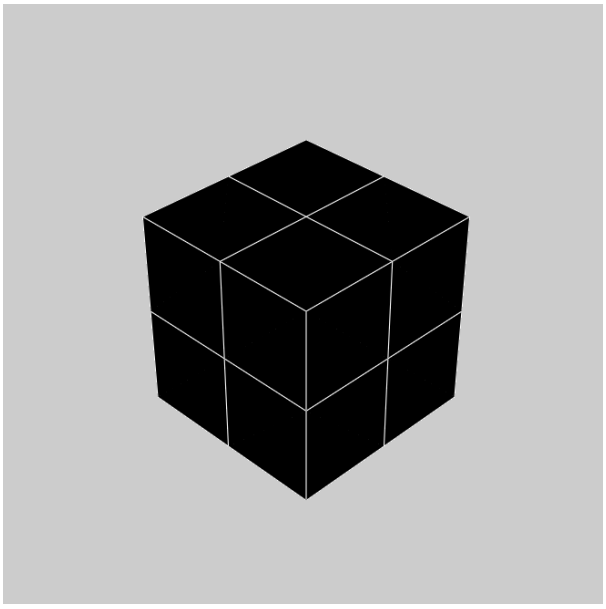
To determine how the spacing affects the transformation from .obj to sparse array, we've provided a function below:

```
def draw_cube(spacing):

   array = vxm.objcast('simple_cube.obj',spacing) # Cast obj file as a point-cloud 3-D␣
↪numpy array
   model = vxm.Model(array)
   model.draw(coloring='custom: black',wireframe=True,wireframe_color='w',background_
↪color='#000000',voxel_spacing=(1,1,1))
```

```
>>> draw_cube(spacing=1)
(see below)
```



```
>>> draw_cube(spacing=0.5)
(see below)
```



We can see a fractional spacing shows a more accurate model of a 2x2x2 supercube.

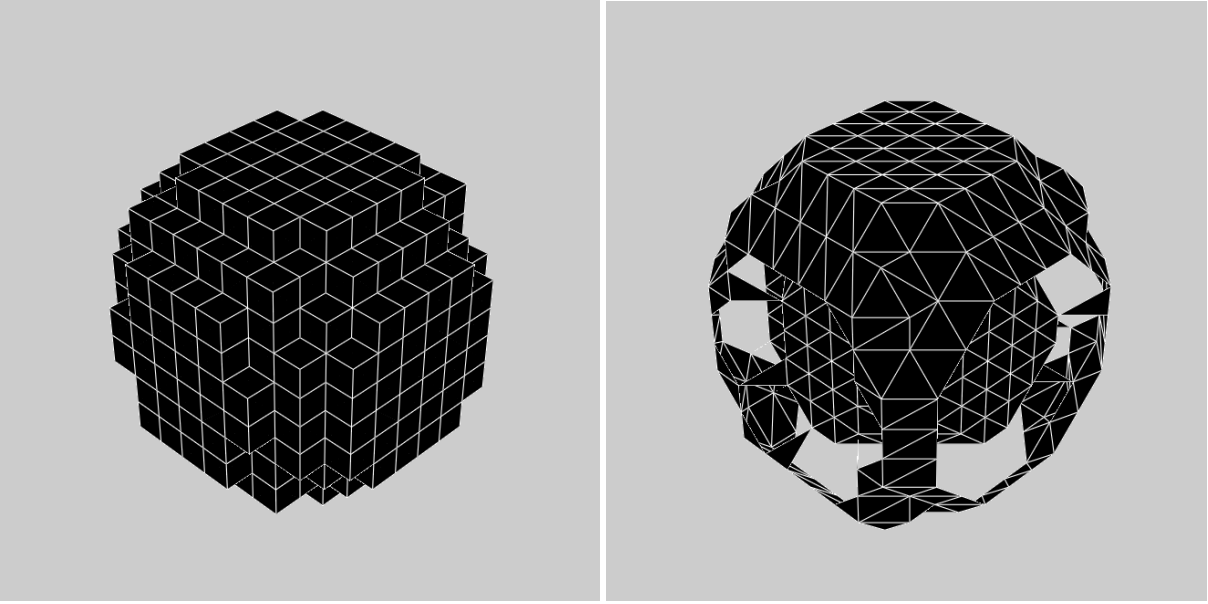
### Sphere Model

This second example demonstrates how voxelmap can be utilized to convert the sphere.obj MODEL into a discrete voxel array, which can then be triangulated using the **MarchingMesh** local method.

```
def sphere_ptcloud(spacing):

   # Draw as point cloud of voxels
   array = vxm.objcast('sphere.obj',spacing) # Cast obj file as a point-cloud 3-D numpy␣
↪array
   model = vxm.Model(array)
   model.draw(coloring='custom: black',wireframe=True,wireframe_color='w',background_
↪color='#000000',voxel_spacing=(1,1,1))

   # Draw as triangulated surface after applying Marching Cubes
   model.objfile= f"scene_marchingmesh{spacing}.obj"
   model.MarchingMesh()
   model.MeshView(wireframe=True,background_color='k',alpha=1)
```

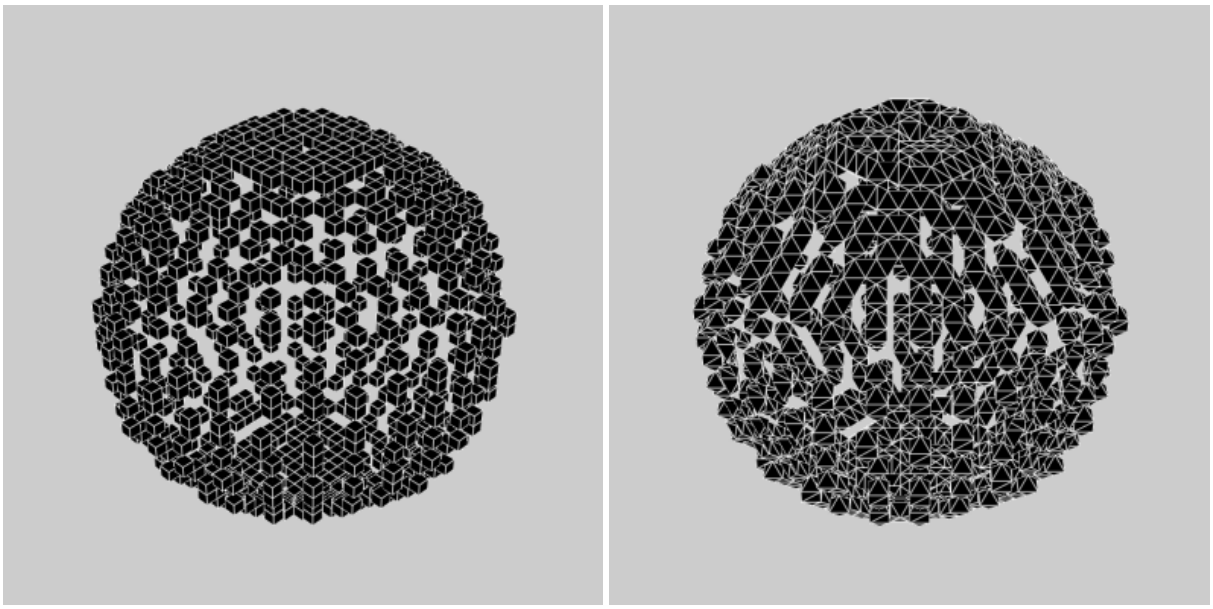
The spacing parameter is crucial when working with the sphere.obj model, which has fractional coordinates between 0 and 1. The transformation from continuous coordinates to a discrete tensor space involves floor-dividing the coordinates, so using a spacing of 1 is likely to result in an unhelpful sparse 3-D tensor for voxel point cloud modeling. To address this issue, we initially set the spacing to 10 and obtained the following results.

```
>>> sphere_ptcloud(spacing=10)
(see below)
```



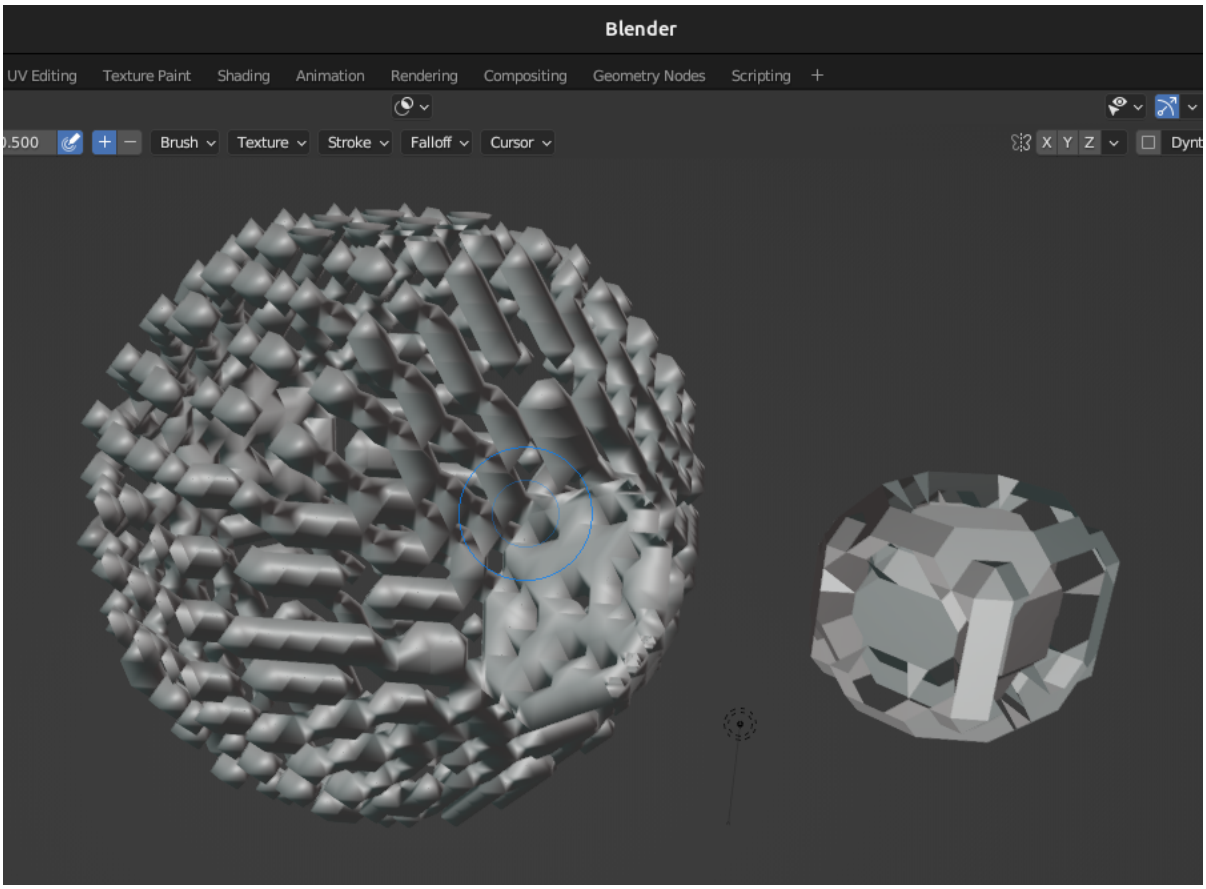
It is important to note that the spacing parameter plays a crucial role in transforming the .obj model to a discrete tensor space, especially when the model has fractional coordinates between 0 and 1. Setting a spacing value of 1 might not produce a useful sparse 3-D tensor for voxel point cloud modeling. In this example with the sphere model, we set the spacing to 10 and obtained satisfactory results.

```
>>> sphere_ptcloud(spacing=30)
(see below)
```



The MarchingMesh method typically output a Wavefront (.obj) file named `scene.obj` as default, but users can specify a different name by changing the objfile variable in the constructor of the **voxelmap.Model** class. Once the file is generated, it can be edited in software like Blender by importing it.

The above commands generated 2 MarchingMesh .obj files, `scene_marchingmesh10.obj` and `scene_marchingmesh30.obj` for the different spacings chosen. These files can be imported simultaneously to a Blender project, and they look like this:



The larger spacing between voxels in a sphere can be observed to result in a larger overall size of the sphere.

### 3-D Model Numpy Manipulation and Blender Integration

What if we wanted to make a modification to the .obj file with Numpy and then save the new modified model as an .obj file for additional treatment in other software like Blender? Here we show a use case for that. Let's take the above **sphere_ptcloud()** function and make the following changes:

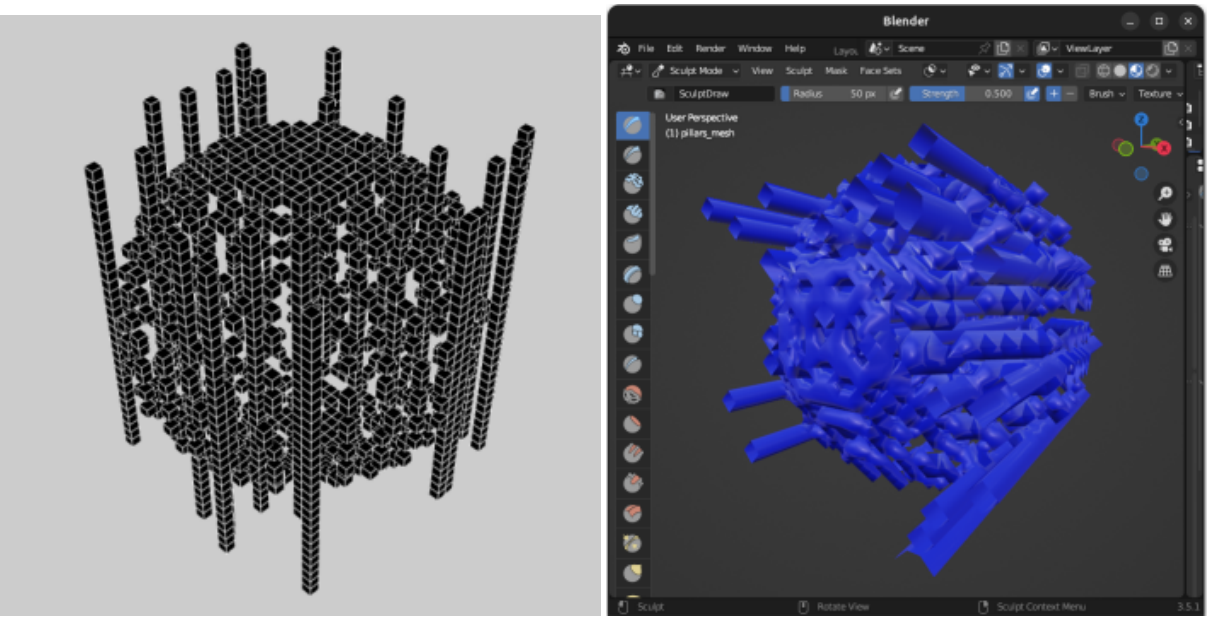
```
def sphere_skewer():

   array = vxm.objcast('sphere.obj',30) # Cast obj file as a point-cloud 3-D numpy array

   #Numpy manipulation
   for i in range(30):
       x,y = np.random.randint(0,np.min(array.shape),2)
       array[x,y,:] = 1

   vxm.Model(array).save('pillars.obj')    #save the array as an .obj file

   model.objfile= f"pillars_mesh.obj"
   model.MarchingMesh()
   model.MeshView(wireframe=True,background_color='k',alpha=1)
```

The above code block loads the `sphere.obj` file, and casts it as a point-cloud 3D numpy array using the **objcast** method. It then makes 30 pillars in random x-y coordinates using Numpy. After this, the modified Numpy array containing the voxel model can be saved back to an .obj format as a NEW FILE using the local **save** method of the **voxelmap.Model** class.

The new `pillars.obj` file can be viewed with voxelmap with the below command

```
>>> vxm.MeshView('pillars.obj',alpha=1,wireframe=True)  # load for view with the global␣
↪MeshView method
(see below)
```

And with Blender importing `pillars_mesh.obj` with the Import tool. Below are the outputs from the voxelmap (left) and the Blender (right) approach:



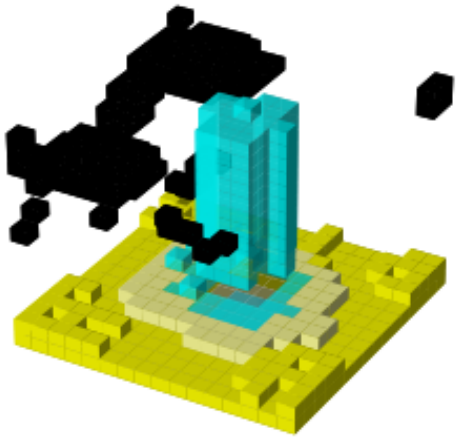

### 2.1.10 3-D Voxel Model Reprocessing

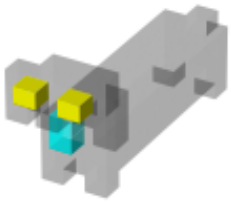
Here we do some reprocessing of the above *voxel* models. Note that here we use the `draw_mpl` method, which is voxelmap's legacy method for voxel modeling and not its state-of-the-art. For faster and higher quality graphics with more kwargs / drawing options, use voxelmap's `draw` method instead.

#### Re-color with custom colors

#### using the `hashblocks_add()` method

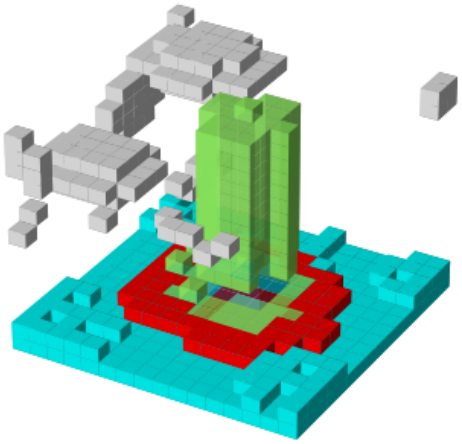
```
theIsland.hashblocks_add(1,'yellow',1)
theIsland.hashblocks_add(2,'#333333',0.2)
theIsland.hashblocks_add(3,'cyan',0.75)
theIsland.hashblocks_add(4,'#000000')

theIsland.draw_mpl('custom',figsize=(5,5))

Dog.hashblocks = theIsland.hashblocks
print('black dog, yellow eyes, cyan tongue')
Dog.draw_mpl('custom',figsize=(5,5))
```





### defining them directly in the hashblocks dictionary

```
theIsland.hashblocks = {
      1: ['cyan', 1],
      2: ['#0197fd', 0.25],
      3: ['#98fc66', 0.78],
      4: ['#eeeeee', 1],
      5: ['red', 1]}

theIsland.draw_mpl('custom',figsize=(7,7))
```



### Save and Load Methods for voxelmap Model objects

#### Save the ghost dog model

If you'd like to save an array with customized color assignments, you may do so now with the `Model().save()` method. This method saves the array data as a DOK hashmap and integrates this DOK hashmap with the Model.hashblocks color information in a higher-order JSON file format:

```
#re-define colors for a ghost dog
Dog.hashblocks = {
      1: ['cyan', 1],
      2: ['#0197fd', 0.25],
      3: ['#98fc66', 0.78],
      4: ['#eeeeee', 1]}

#save
Dog.save('ghostdog.json')
```

#### Load ghost dog model

The `Model().load()` method processes the array and color information to a blank Model object. To load this data into a "blank slate" and re-draw it, type the following:

```
# defines a blank model
blank = vxm.Model()
print(blank.array)
print(blank.hashblocks)

blank.load('ghostdog.json')

print(blank.array[0].shape)
```

(continues on next page)

(continued from previous page)

```
print(blank.hashblocks)
blank.draw_mpl('custom',figsize=(7,7))
```

## 2.2 API Reference

### 2.2.1 Global Methods

As the methods are several, below are only listed the most pertinent global methods of voxelmap, in order of the lowest level to highest level of applications to 3-D modeling operations.and classified in sub-sections:

#### Special

**class** voxelmap.**objcast**(*filename='scene.obj'*, *spacing=1*)

Converts a Wavefront .obj file into a sparse, third-order (3-D) NumPy array to represent a point cloud model.

##### Parameters

**filename**
: [str ] Path to the .obj file. Defaults to 'scene.obj'.

**spacing**
: [float] Distance between points in the point cloud. Can be adjusted to create a denser or sparser point cloud. Defaults to 1.

**class** voxelmap.**matrix_toXY**(*array*, *mult*)

Converts a 2D numpy array into an XYv coordinate matrix where v is the corresponding element in every x-y coordinate.

#### Parameters

**array**
[np.array(int,int)] The 2D numpy array to be converted to an XYv coordinate matrix.

**mult**
[int or float] The multiplication factor to be applied to the elements in the matrix.

class voxelmap.**tensor_toXYZ**(*tensor*, *mult*)

Converts a 3D numpy array (tensor) into an XYZ coordinate matrix.

#### Parameters

**tensor**
[np.array(int,int,int)] The 3D array (tensor) to be converted to XYZ coordinates.

**mult**
[int or float] The multiplication factor to be applied to all the coordinates.

### Load and Save

class voxelmap.**load_array**(*filename*)

Loads a pickled numpy array with *filename* name

class voxelmap.**save_array**(*array*, *filename*)

Saves an *array* array with *filename* name using the pickle module

class voxelmap.**tojson**(*filename*, *array*, *hashblocks={}*)

Save 3-D array and *hashblocks* color mapping as JSON file

class voxelmap.**load_from_json**(*filename*)

Load JSON file to object

### Array Manipulation

class voxelmap.**resize_array**(*array*, *mult=(2, 2, 2)*)

Resizes a three-dimensional array by the three dim factors specified by *mult* tuple. Converts to sparse array of 0s and 1s

#### Parameters

**array**
[np.array(int,int,int)] array to resize

**mult: tuple(float,float,float)**
depth length width factors to resize array with. e.g 2,2,2 resizes array to double its size in all dims

class voxelmap.**roughen**(*array*, *kernel_level=1*)

Makes a 3d array model rougher by a special convolution operation. Uses *voxelmap.random_kernel_convolve*.

### Parameters

**array**
[np.array(int,int,int)] array to *roughen up*

**kernel_level: int**
length scale (size) of random kernels used. The smallest scale (=1) gives the roughest transformation.

class voxelmap.**random_kernel_convolve**(*array*, *kernel*, *random_bounds=(-10, 10)*)

Applies a three-dimensional convolution with a randomly-mutating *kernel* on a 3-D *array* which changes for every array site when random_bounds are set to tuple. If random_bounds are set to False, convolution occurs in constant mode for the specified kernel.

### Parameters

**array**
[np.array(int,int,int)] array to convolve

**kernel: np.array(int,int,int)**
kernel to use for convolution. If random_bounds are set to tuple, only the kernel's shape is used to specify the random_kernels

**random_bounds**
[tuple(int,int) OR bool] see above explanation.

## Mapping

class voxelmap.**MarchingMesh**(*array*, *out_file='scene.obj'*, *level=0*, *spacing=(1.0, 1.0, 1.0)*, *gradient_direction='descent'*, *step_size=1*, *allow_degenerate=True*, *method='lewiner'*, *mask=None*, *plot=False*, *figsize=(4.8, 4.8)*)

Marching cubes on sparse 3-D integer *voxelmap* arrays (GLOBAL)

### Parameters

**array: np.array((int/float,int/float,int/float))**
3-D array for which to run the marching cubes algorithm

**out_file**
[str] name and/or path for Wavefront .obj file output. This is the common format for OpenGL 3-D model files (default: model.obj)

**level**
[float, optional] Contour value to search for isosurfaces in *volume*. If not given or None, the average of the min and max of vol is used.

**spacing**
[length-3 tuple of floats, optional] Voxel spacing in spatial dimensions corresponding to numpy array indexing dimensions (M, N, P) as in *volume*.

**gradient_direction**
[string, optional] Controls if the mesh was generated from an isosurface with gradient descent toward objects of interest (the default), or the opposite, considering the *left-hand* rule. The two options are: – 'descent' : Object was greater than exterior – 'ascent' : Exterior was greater than object

**step_size**
[int, optional] Step size in voxels. Default 1. Larger steps yield faster but coarser results. The result will always be topologically correct though.

**allow_degenerate**
[bool, optional] Whether to allow degenerate (i.e. zero-area) triangles in the end-result. Default True. If False, degenerate triangles are removed, at the cost of making the algorithm slower.

**method: str, optional**
One of 'lewiner', 'lorensen' or '_lorensen'. Specify which of Lewiner et al. or Lorensen et al. method will be used. The '_lorensen' flag correspond to an old implementation that will be deprecated in version 0.19.

**mask**
[(M, N, P) array, optional] Boolean array. The marching cube algorithm will be computed only on True elements. This will save computational time when interfaces are located within certain region of the volume M, N, P-e.g. the top half of the cube-and also allow to compute finite surfaces-i.e. open surfaces that do not end at the border of the cube.

**plot: bool**
plots a preliminary 3-D triangulated image if True

`class` voxelmap.**`MeshView`**(*objfile='scene.obj'*, *color='black'*, *alpha=1*, *wireframe=True*, *wireframe_color='white'*, *background_color='#cccccc'*, *viewport=[1024, 768]*)

Triangulated mesh view with PyVista (GLOBAL)

### Parameters

**objfile: string**
.obj file to process with MeshView [in GLOBAL function only]

**color**
[string / hexadecimal] mesh color. default: 'pink'

**alpha**
[float] opacity transparency range: 0 - 1.0. Default: 0.5

**wireframe: bool**
Represent mesh as wireframe instead of solid polyhedron if True (default: False).

**wireframe_color: string / hex**
edges or wireframe colors

**background_color**
[string / hexadecimal] color of background. default: 'pink'

**viewport**
[(int,int)] viewport / screen (width, height) for display window (default: 80% your screen's width & height)

### 2.2.2 Local Methods to Model class

**class** voxelmap.**Model**(*array=[]*, *file=''*)

**__init__**(*array=[]*, *file=''*)

Model structure. Calls *3-D* array to process into 3-D model.

#### Parameters

**array**
[np.array(int)] array of the third-order populated with discrete, non-zero integers which may represent a voxel block type

**file**
[str] file name and/or path for image file

**hashblocks**
[dict[int]{str, float }] a dictionary for which the keys are the integer values on the discrete arrays (above) an the values are the color (str) for the specific key and the alpha for the voxel object (float)

**colormap**
[< matplotlib.cm object >] colormap to use for voxel coloring if coloring kwarg in Model.draw method is not voxels. Default: cm.cool

**alphacm**
[float] alpha transparency of voxels if colormap option is chosen. default: opaque colormap (alpha=1)

– FOR FILE PROCESSING –

**file**
[str] file name a/or path for goxel txt file

– FOR XYZ COORDINATE ARRAY PROCESSING –

**XYZ**
[np.array(float )] an array containing the x,y,z coordinates of shape *number_voxel-locations* x 3 [for each x y z]

**RGB**
[list[str] ] a list for the colors of every voxels in xyz array (length: *number_voxel-locations*)

**sparsity**
[float] a factor to separate the relative distance between each voxel (default:10.0 [> 50.0 may have memory limitations])

**ImageMap**(*depth=5*, *out_file='scene.obj'*, *plot=False*)

Map image or 2-D array (matrix) to 3-D array

### Parameters

**depth**
[int] depth in number of voxels (default = 5 voxels)

**out_file**
[str] name and/or path for Wavefront .obj file output. This is the common format for OpenGL 3-D model files (default: model.obj)

**plot: bool / str**
plots a preliminary 3-D triangulated image if True with PyVista. For more plotting options, plot by calling MeshView separately.

**ImageMesh**(*out_file='scene.obj'*, *L_sectors=4*, *rel_depth=0.5*, *trace_min=1*, *plot=True*, *figsize=(4.8, 4.8)*, *verbose=False*)

3-D triangulation of 2-D images / 2-D arrays (matrices) with a Convex Hull algorithm (Andrew Garcia, 2022)

### Parameters

**out_file**
[str] name and/or path for Wavefront .obj file output. This is the common format for OpenGL 3-D model files (default: model.obj)

**L_sectors: int**
length scale of Convex Hull segments in sector grid, e.g. L_sectors = 4 makes a triangulation of 4 x 4 Convex Hull segments

**rel_depth: float**
relative depth of 3-D model with respect to the image's intensity magnitudes (default: 0.50)

**trace_min: int**
minimum number of points in different z-levels to triangulate per sector (default: 1)

**plot: bool / str**
plots a preliminary 3-D triangulated image if True [with PyVista (& with matplotlib if plot = 'img'). For more plotting options, plot with Meshview instead.

**MarchingMesh**(*voxel_depth=12*, *level=0*, *spacing=(1.0, 1.0, 1.0)*, *gradient_direction='descent'*, *step_size=1*, *allow_degenerate=True*, *method='lewiner'*, *mask=None*, *plot=False*, *figsize=(4.8, 4.8)*)

Marching cubes on 3-D mapped image or 3-D array

### Parameters

**voxel_depth**
[int (optional)] depth of 3-D mapped image on number of voxels (if file [image] is to be processed / i.e. specified)

**level**
[float, optional] Contour value to search for isosurfaces in *volume*. If not given or None, the average of the min and max of vol is used.

**spacing**
[length-3 tuple of floats, optional] Voxel spacing in spatial dimensions corresponding to numpy array indexing dimensions (M, N, P) as in *volume*.

**gradient_direction**
[string, optional] Controls if the mesh was generated from an isosurface with gradient descent toward objects of interest (the default), or the opposite, considering the *left-hand* rule. The two options are: – 'descent' : Object was greater than exterior – 'ascent' : Exterior was greater than object

**step_size**
[int, optional] Step size in voxels. Default 1. Larger steps yield faster but coarser results. The result will always be topologically correct though.

**allow_degenerate**
[bool, optional] Whether to allow degenerate (i.e. zero-area) triangles in the end-result. Default True. If False, degenerate triangles are removed, at the cost of making the algorithm slower.

**method: str, optional**
One of 'lewiner', 'lorensen' or '_lorensen'. Specify which of Lewiner et al. or Lorensen et al. method will be used. The '_lorensen' flag correspond to an old implementation that will be deprecated in version 0.19.

**mask**
[(M, N, P) array, optional] Boolean array. The marching cube algorithm will be computed only on True elements. This will save computational time when interfaces are located within certain region of the volume M, N, P-e.g. the top half of the cube-and also allow to compute finite surfaces-i.e. open surfaces that do not end at the border of the cube.

**plot: bool**
plots a preliminary 3-D triangulated image if True

`MeshView`(*color='black'*, *alpha=1*, *wireframe=True*, *wireframe_color='white'*, *background_color='#ffffff'*, *viewport=[1024, 768]*)

Triangulated mesh view with PyVista

### Parameters

**objfile: string**
.obj file to process with MeshView [in GLOBAL function only]

**color**
[string / hexadecimal] mesh color. default: 'pink'

**alpha**
[float] opacity transparency range: 0 - 1.0. Default: 0.5

**wireframe: bool**
Represent mesh as wireframe instead of solid polyhedron if True (default: False).

**wireframe_color: string / hex**
edges or wireframe colors

**background_color**
[string / hexadecimal] color of background. default: 'pink'

**viewport**
[(int,int)] viewport / screen (width, height) for display window (default: 80% your screen's width & height)

`build`()

Builds voxel model structure from python numpy array

**draw**(*coloring='none'*, *geometry='voxels'*, *scalars=''*, *background_color='#cccccc'*, *wireframe=False*, *wireframe_color='k'*, *window_size=[1024, 768]*, *voxel_spacing=(1, 1, 1)*, *show=True*)

Draws voxel model after building it with the provided *array* with PyVista library

### Parameters

**coloring: string**

**voxel coloring scheme**

- 'custom' –> colors voxel model based on the provided keys to its array integers, defined in the *hashblocks* variable from the *Model* class
- 'custom: #8599A6' –> color all voxel types with the #8599A6 hex color (bluish dark gray) and an alpha transparency of 1.0 (default)
- 'custom: red, alpha: 0.24' –> color all voxel types red and with an alpha transparency of 0.24
- 'cmap : {colormap name}' : colors voxel model based on a colormap; assigns colors from the chosen colormap to the defined array integers
- 'cmap : viridis' –> colormap voxel assignment with the viridis colormap
- 'cmap : hot', alpha: 0.56 –> voxel assignment with the hot colormap with an alpha transparency of 0.56
- 'none' –> no coloring
- 'cool' cool colormap
- 'fire' fire colormap
- and so on...

**geometry: string**
voxel geometry. Choose voxels to have a box geometry with geometry='voxels' or spherical one with geometry='particles'

**scalars**
[list] list of scalars for cmap coloring scheme

**background_color**
[string / hex] background color of pyvista plot

**wireframe: bool**
Represent mesh as wireframe instead of solid polyhedron if True (default: False).

**wireframe_color: string / hex**
edges or wireframe colors

**window_size**
[(float,float)] defines plot window dimensions. Defaults to [1024, 768], unless set differently in the relevant theme's window_size property [pyvista.Plotter]

**voxel_spacing**
[(float,float,float)] changes voxel spacing by defining length scales of x y and z directions (default:(1,1,1)).

**show**
[bool] Display Pyvista 3-D render of drawn 3-D model if True (default: True)

`draw_mpl`(*coloring='custom'*, *edgecolors=None*, *figsize=(6.4, 4.8)*, *axis3don=False*)

Draws voxel model after building it with the provided *array* (Matplotlib version. For faster graphics, try the `draw()` method (uses PyVista)).

### Parameters

**coloring: string**

**voxel coloring scheme**

- 'custom' –> colors voxel model based on the provided keys to its array integers, defined in the *hashblocks* variable from the *Model* class
- 'custom: #8599A6' –> color all voxel types with the #8599A6 hex color (bluish dark gray) and an alpha transparency of 1.0 (default)
- 'custom: red, alpha: 0.24' –> color all voxel types red and with an alpha transparency of 0.2
- 'nuclear' colors model radially, from center to exterior
- 'linear' colors voxel model vertically, top to bottom.

**edgecolors: string/hex**
edge color of voxels (default: None)

**figsize**
[(float,float)] defines plot window dimensions. From matplotlib.pyplot.figure(figsize) kwarg.

**axis3don: bool**
defines presence of 3D axis in voxel model plot (Default: False)

`hashblocks_add`(*key*, *color*, *alpha=1*)

Make your own 3-D colormap option. Adds to hashblocks dictionary.

### Parameters

**key**
[int ] array value to color as voxel

**color**
[str] color of voxel with corresponding *key* index (either in hexanumerical # format or default python color string)

**alpha**
[float, optional] transparency index (0 -> transparent; 1 -> opaque; default = 1.0)

`load`(*filename='scene.json'*, *coords=False*)

Load to Model object.

#### Parameters

**filename: string (.json or .txt extensions (see above))**
name of file to be loaded (e.g 'scene.json')

**coords: bool**
loads and processes self.XYZ, self.RGB, and self.sparsity = 10.0 (see Model class desc above) to Model if True. This boolean overrides filename loader option.

`make_intensity()`

Turn image into intensity matrix i.e. matrix with pixel intensities. Outputs self.array as mutable matrix to contain relative pixel intensities in int levels [for file if specified]

`resize_intensity(res=1.0, res_interp=3)`

Resize the intensity matrix of the provided image.

#### Parameters

**res**
[float, optional] relative resizing percentage as *x* times the original (default 1.0 [1.0x original dimensions])

**res_interp: object, optional**
cv2 interpolation function for resizing (default cv2.INTER_AREA)

`save(filename='scene.json')`

Save sparse array + color assignments Model data as a dictionary of keys (DOK) JSON file

#### Parameters

**filename: string**
name of file (e.g. 'scene.json') Data types: .json -> voxel data represented as (DOK) JSON file .txt -> voxel data represented x,y,z,rgb matrix in .txt file (see Goxel .txt imports)

*voxelmap*

## voxelmap

CHAPTER

THREE

# EXAMPLES

## 3.1 Voxel Model to 3-D Mesh

.txt file source: https://raw.githubusercontent.com/andrewrgarcia/voxelmap/main/model_files/skull.txt

```
#skull.py
import voxelmap as vxm

model = vxm.Model()

model.load('extra/skull.txt')

arr = model.array


model.array = model.array[::-1]

'draw in standard voxel form'
model.draw('voxels',wireframe=True, background_color='#3e404e',window_size=[700,700])

'to convert to mesh'
model.MarchingMesh()
model.MeshView(wireframe=True,alpha=1,color=True,background_color='#b064fd',
↪viewport=[700,700])
```

## 3.2 3-D Reconstruction

You can easily create stunning 3D models from images with voxelmap! This software package provides powerful geometric algorithms designed to reconstruct images, with a primary focus on generating very simple depth fields from pixel intensity values. These algorithms include ImageMesh, which produces compact, low-poly models from images, and the well-known MarchingMesh algorithm, which creates detailed mesh representations at a higher computational cost.

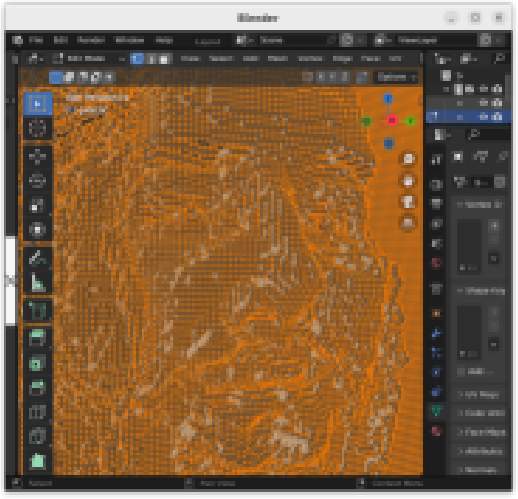
With just a few lines of code, you can use this package to transform 2D images into high-quality, immersive 3D models in the widely-used .obj format.

*Photo used with permission from the @galacticeffect. Check out his music available at Spotify

In the example below, the above artist portrait is loaded as a file to the Model class to create a 3-D model with the local MarchingMesh method, which generates an .obj file in the process:

```
#galactic.py
import voxelmap as vxm

model = vxm.Model(file='galactic.png')       # load image
model.objfile = 'galactic.obj'           # set name of 3-D model file (.obj) to be made
model.MarchingMesh(25)                   # make 3-D model from image.
```

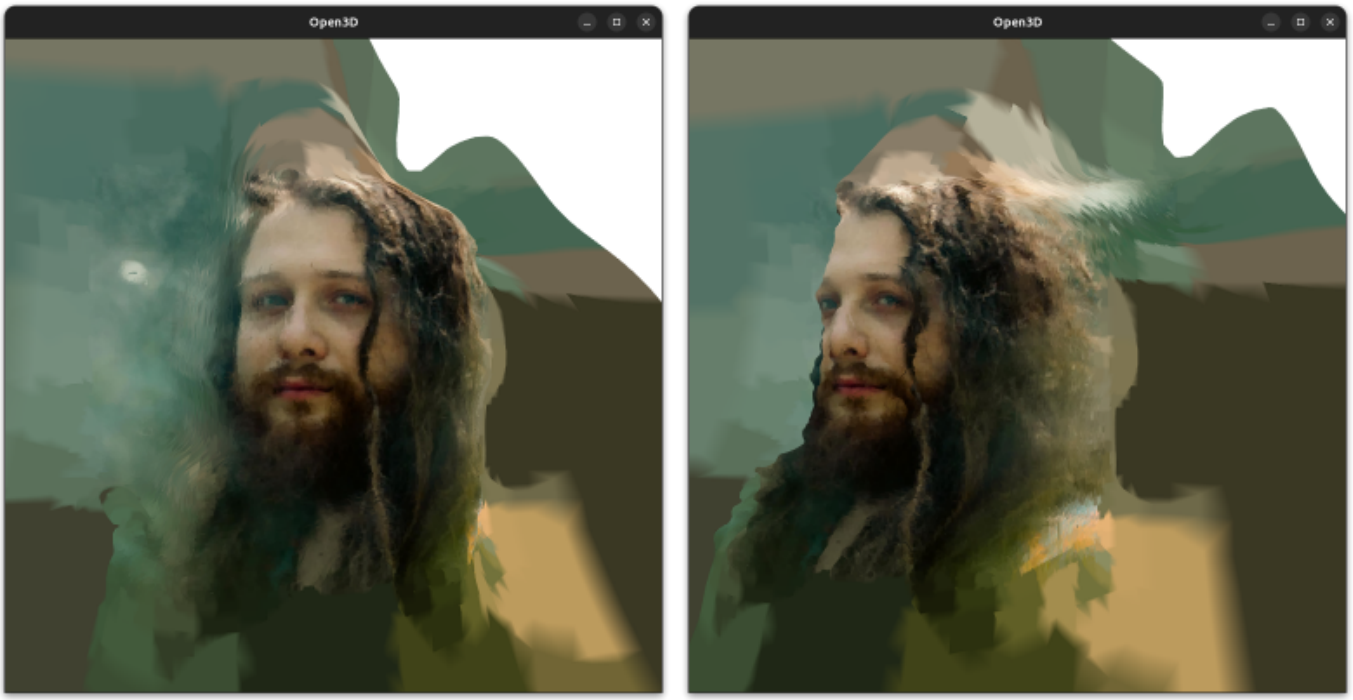
Once you have the .obj file, you can import it into popular graphic editing software such as Blender for further customization and rendering. Alternatively, you can visualize the 3D model using the voxelmap library by calling the local **model.MeshView()** method after the last line of the script.



Unleash your creativity with this user-friendly protocol for 3D reconstruction from images.

### 3.2.1 Coming Soon : AI-based 3-D Reconstruction

With voxelmap, we are committed to continually improving our project with the latest technological advancements. As part of our ongoing development, we are planning to introduce more advanced methods for 3-D reconstruction using artificial intelligence in a future release, version 5.0.




These new method(s) will provide higher quality 3-D reconstruction, and enable users to go from image to 3-D model with just one line of code. Our aim is to democratize the 3-D modeling process and make it more accessible to everyone.

CHAPTER

FOUR

# WHITEPAPERS

## 4.1 ImageMesh : A Convex Hull based 3D Reconstruction Method

**Andrew R. Garcia**

garcia.gtr@gmail.com

### 4.1.1 Abstract

ImageMesh is a method available in versions $\geq$ 2.0 of the voxelmap Python library that generates 3D models from images using Convex Hull in 3-D to enclose external points obtained from a series of partitioned point clouds. These point clouds are generated by assigning the relative pixel intensities from the partitioned images as the depth dimension to the points. In this paper, we describe the limitations of the original ImageMesh method and the quick solution we have implemented to address them. Additionally, we introduce MeshView, a Python visualization tool developed in tandem with ImageMesh that provides a convenient way to visualize the 3D models generated by ImageMesh. Finally, we discuss the GPU memory space complexity of both methods.

### 4.1.2 Introduction

Fig. 1: An image of a crochet donut

ImageMesh is a 3D reconstruction method. It utilizes Convex Hull in 3-D to enclose external points obtained from a series of partitioned point clouds, which are generated by assigning the relative pixel intensities from the partitioned

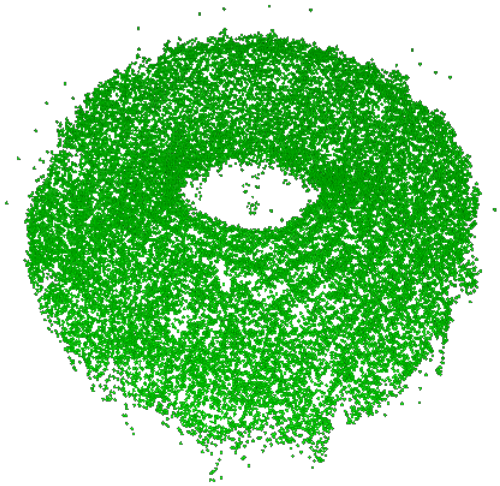


Fig. 2: 3-D reconstruction of donut image to voxel cloud done by transforming relative pixel intensities to 3-D depth with the ImageMap method.

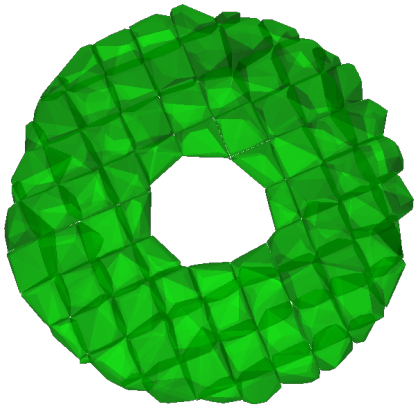


Fig. 3: 3-D reconstruction of donut image with the ImageMesh method.

images as the depth dimension to the point clouds. ImageMesh and ImageMap methods utilize the CCIR 601 luma protocol to transform input images into 2-dimensional "intensity" matrices that represent relative grayscale values. These intensity values are then mapped to the depth dimension to form third-order arrays. ImageMesh generates 3-D model .obj files from images by drawing the smallest collection of triangular polygons that satisfies this rule in three-dimensional space. In simpler terms, it's like placing a fabric over a 3-D object. However, it is important to note that a single convex hull operation may fail to draw a structure with holes or bumps, which are essentially 2-D local extrema.

To address this limitation, we have implemented a quick solution that involves partitioning the input image into multiple segments and placing 3-D convex hull "sheets" on each segment to perform triangulation. Figure 3 shows how this partitioning can process more local extrema with the 3-D Convex Hull method and create more realistic 3-D reconstructions.

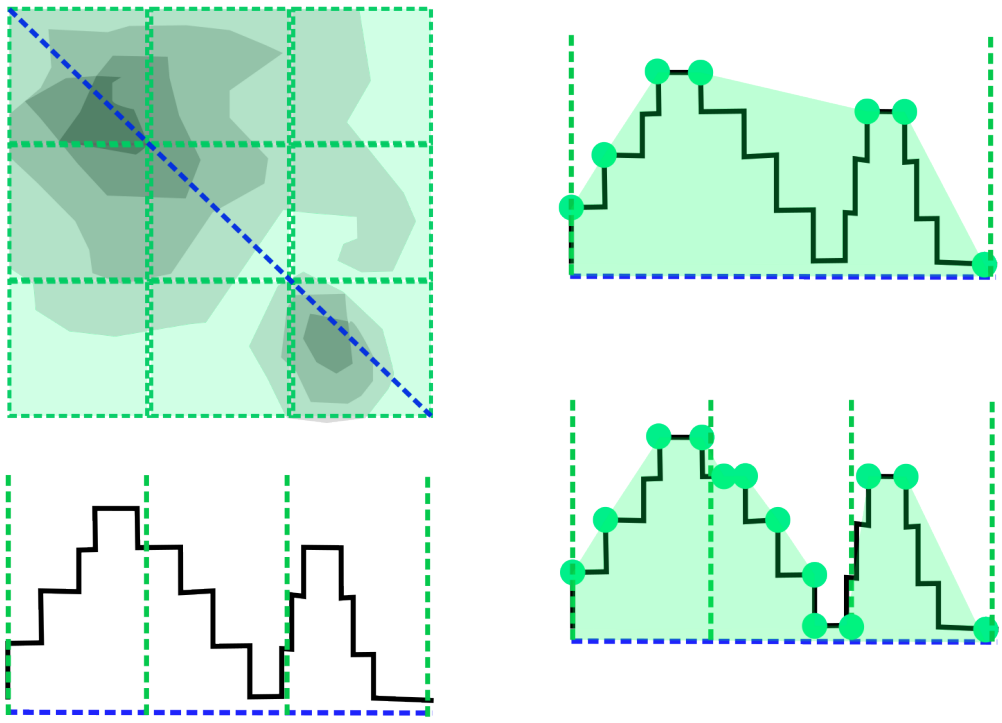


Fig. 4: Schematic representation of the performance of ImageMesh with more segments.

In the Figure 4, blurring is performed to create a more realistic 3-D texture with the ImageMesh method. Blurring an image can be thought of as a process of smoothing out the details in the image by reducing the contrast between adjacent pixels. This results in a 3-D effect where the image appears to have a more uniform and continuous surface. Instead of sharp edges and abrupt changes in color or texture, blurred images have a softer and more gradual transition between different parts of the image.

Complementing ImageMesh is MeshView, a Python visualization tool developed in tandem to provide a convenient way to visualize the 3D models generated by ImageMesh. MeshView is capable of loading the .obj files generated by ImageMesh and rendering them in a PyVista VTK window, allowing for interactive 3D visualization of the models

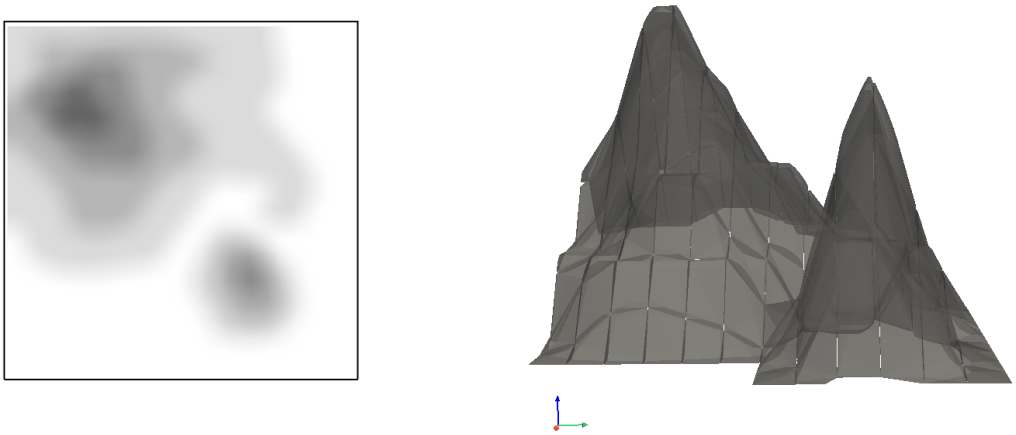


Fig. 5: Blurring of former image to create a smoother 3-D texture with ImageMesh.

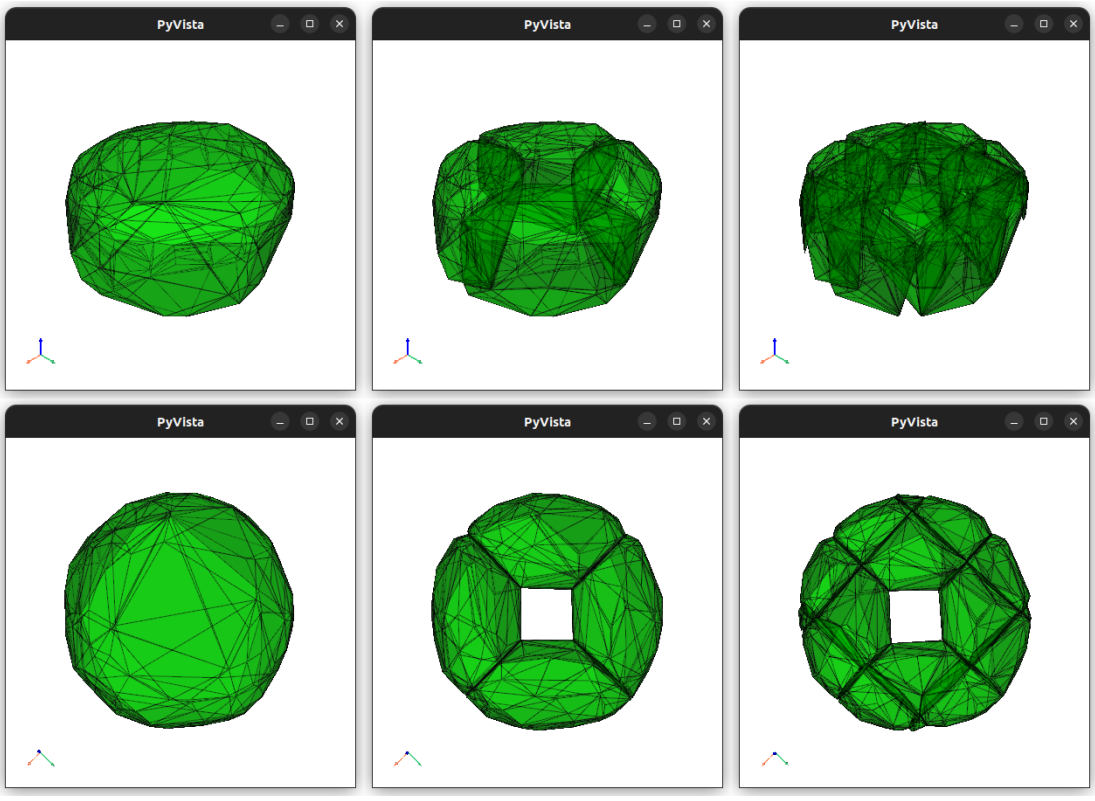



Fig. 6: The graphical effect of increasing the number of 3-D Convex Hull (CH) sectors in each column. The left column shows 1 CH sector, the middle column shows 4 CH sectors, and the right column shows 16 CH sectors.

### 4.1.3 Time Complexity

When using a voxel-based approach for 3-D reconstruction from an image, that is, converting the 2-D image to a voxel cloud, the time complexity for converting the 2-D image to a voxel cloud is $\mathcal{O}(whd)$, where $w$ and $h$ are the width and height of the image, and $d$ is the depth of the voxel cloud. In comparison, the ImageMesh method for 3-D reconstruction involves 3 main steps which have its own computational complexity. In the next paragraphs we elaborate on the steps for this method and their associated time complexities.

In the first step, the image is sliced into equally-sized sectors: $\mathcal{O}(wh)$, where $w$ is the width of the image and $h$ is its height. After this, each of the 2-D sectors are mapped to 3-D point clouds, with a time complexity $\mathcal{O}(wh)$, where w and h are the width and height of the image.

3-D Convex Hull is then performed on each of these point clouds with $\mathcal{O}(n \log n)$, where $n$ is the number of points in each 3-D point cloud. For this last step, we use the `scipy.spatial.ConvexHull` algorithm for computing the convex hull of the 3-D point clouds. Th algorithm uses a divide-and-conquer approach based on the QuickHull algorithm, which has an average case time complexity of $\mathcal{O}(n \log n)$. This makes the `scipy.spatial.ConvexHull` algorithm efficient for computing the convex hull of a 3-D point cloud, especially for larger datasets. However, the time complexity for the worst-case scenario of QuickHull is $\mathcal{O}(n^2)$, which means that in some cases, the `scipy.spatial.ConvexHull` algorithm may take longer to run.

Therefore, the overall time complexity of ImageMesh can be estimated as $\mathcal{O}(whn \log n)$, where $w$ and $h$ are the width and height of the image, and $n$ is the number of points in each 3-D point cloud.

When considering the time complexity of the two methods, it is clear that ImageMesh, a method which generates a single mesh with multiple steps, has a higher computational cost than the voxel-based approac, a method which generates multiple voxel meshes. However, it is important to note that the graphical manipulation of the single mesh is superior to the latter method. This is because a single mesh can update all its defined vertices more efficiently than updating multiple voxel meshes.

As a result, ImageMesh is more suitable for applications that require real-time graphical manipulation or rendering, where the speed of the graphics processing is crucial. On the other hand, the voxel mesh method may be more appropriate for applications where the accuracy of the reconstruction is more important than the graphical performance, and where the computational cost can be distributed over multiple processors or computer nodes.

### 4.1.4 Conclusion

ImageMesh, coupled with MeshView, provides a powerful and efficient 3D reconstruction method. By implementing the Convex Hull-based method with partitioning, we have addressed the limitation of a single convex hull operation and increased the resolution of the reconstructed 3D models. With the reduction in GPU space complexity, the new method has become more practical for real-world applications.

# PYTHON MODULE INDEX

# INDEX