voxelmap

Release 3.5

Andrew Garcia, Ph.D.

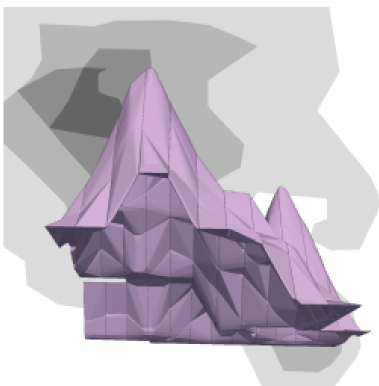
Mar 31, 2023

CONTENTS

CHAPTER

ONE

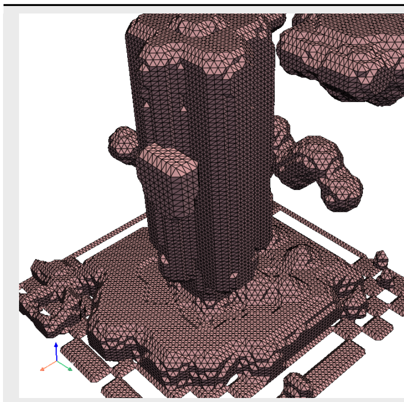
LET'S MAKE 3-D MODELS WITH PYTHON!



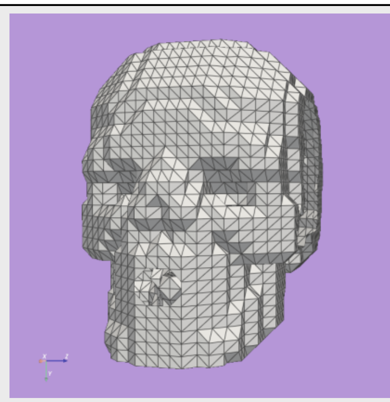
Ever wanted to make simple 3-D models from numpy arrays? Now you can do that with voxelmap ! **Voxelmap** is a Python library for making voxel and three-dimensional models from NumPy arrays. It was initially made to streamline 3-D voxel modeling by assigning each integer in an array to a voxel. Now,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

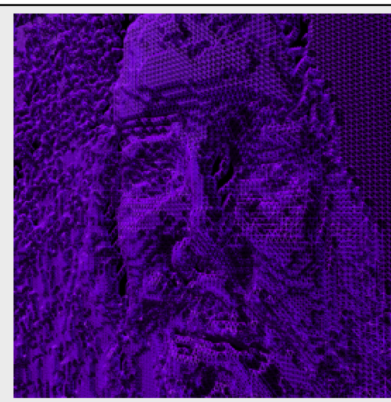
1.1 Examples



Island Mesh



Skull Mesh



ImageMesh Portrait

1.2 Colab Notebook

We also offer an interactive tutorial through a Colab notebook, click below:

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 Draw voxels from an integer array

Voxelmap was originally made to handle third-order integer arrays of the form `np.array((int,int,int))` as blueprints to 3-D voxel models.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, the non-zero integer values 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 voxel color and transparencies may be added or modified to the `hashblocks` map with the `hashblocksAdd` method.

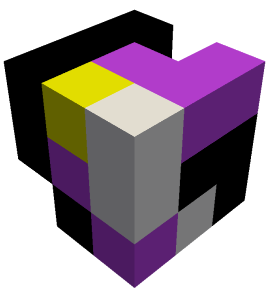
```
import voxelmap as vxm
import numpy as np

#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',
```

(continues on next page)

(continued from previous page)

```
         'yellow', '#000000', 'white', 'k', '#c745f8']
for i in range(9):
   model.hashblocksAdd(i+1, colors[i])

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

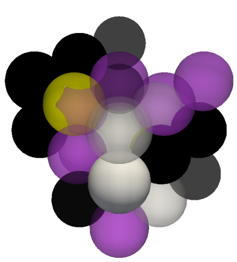
```
>>> [Out]
[[[3 8 5]
  [0 2 6]
  [2 2 7]]
 [[8 3 6]
  [7 2 0]
  [2 2 1]]
 [[9 2 4]
  [8 5 7]
  [8 9 8]]]
```



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.hashblocksAdd(i+1,colors[i],alphas[i])
model.draw('custom', geometry='particles', background_color='#ffffff')
```



2.1.3 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.

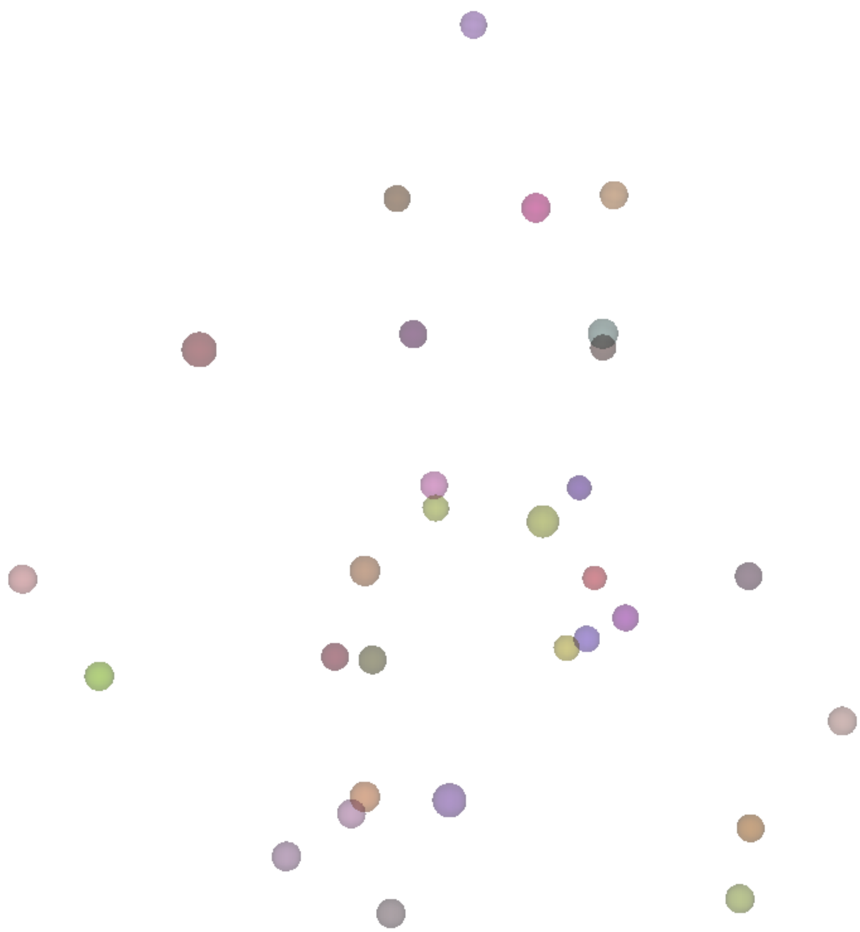
```
import voxelmap as vxm
import numpy as np

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
```



2.1.4 Get files for below examples

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

LAND IMAGE (.png)

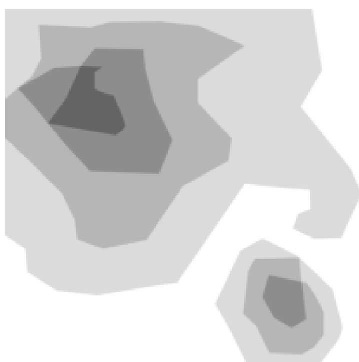
DOG MODEL (.txt)

ISLAND MODEL (.txt)

2.1.5 3-D Mapping of an Image

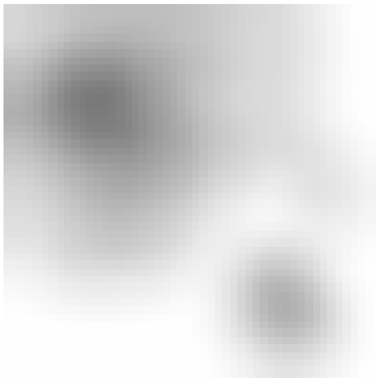
Here we map the synthetic topography image land.png we just downloaded to 3-D using the `map3d` method from the `voxelmap.Image` class.

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

plt.imshow(cv2.imread('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

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



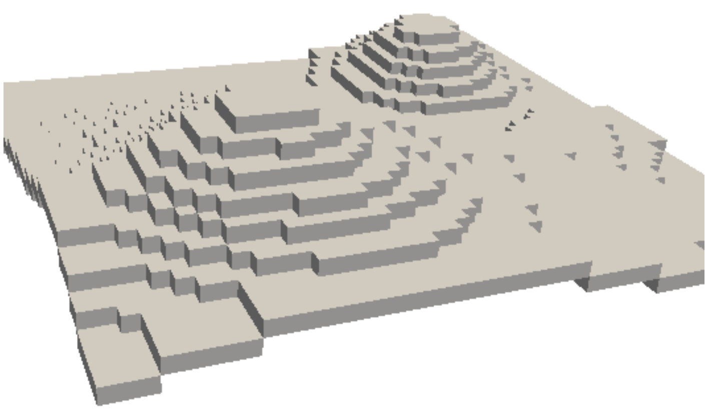
The image is then resized for the voxel draw with the matplotlib method i.e. `Model().draw_mpl`. This is done with `cv2.resize`, resizing the image from 1060x1060 to 50x50. After resizing, we convolve the image to obtain a less sharp color shift between the different gray regions with the `cv2.blur`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

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

img.array = cv2.blur(img.array,(10,10))     # blur the image for realiztic topography␣
↪levels
plt.imshow(img.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 *Image* class:

```
mapped_img = img.ImageMap(12)              # mapped to 3d with a depth of 12 voxels
print(mapped_img.shape)
model = vxm.Model(mapped_img)
model.array = np.flip(np.transpose(model.array))

model.alphacm = 0.5
model.draw('none',background_color='#ffffff')
```



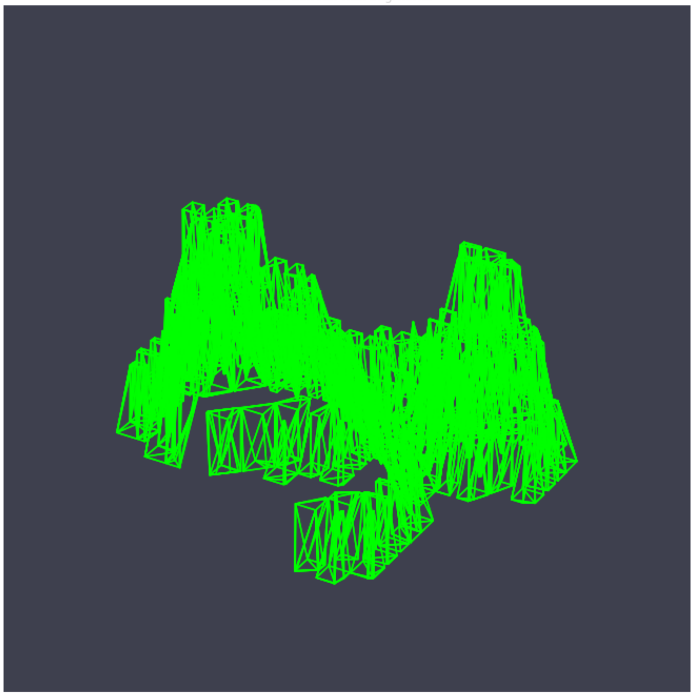
2.1.6 ImageMesh : 3-D Mesh Mapping from Image

This 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.

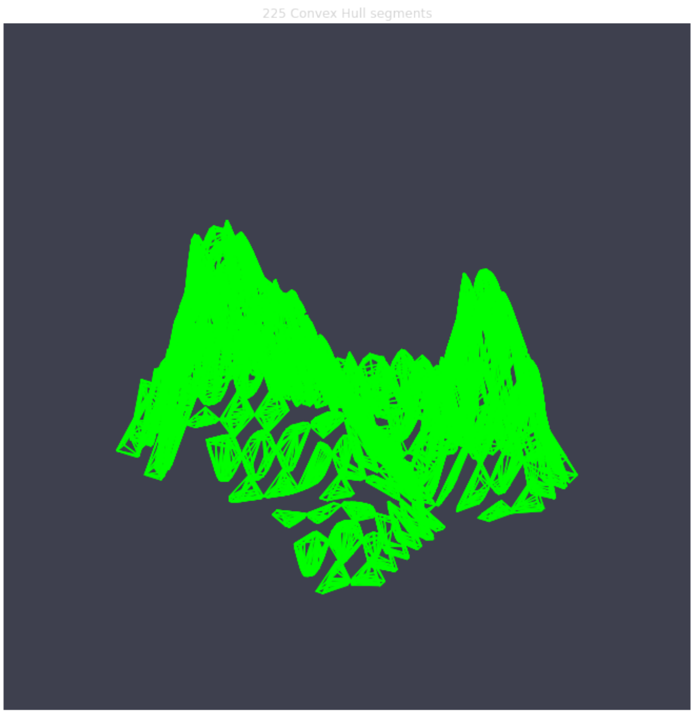
We can see that the mesh model can be calculated and drawn with matplotlib `plot=mpl` option even from a large image of 1060x1060 without resizing:

```
import voxelmap as vxm
import cv2

img = vxm.Image('land.png')   # incorporate fake land topography .png file

print(img.array.shape)

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



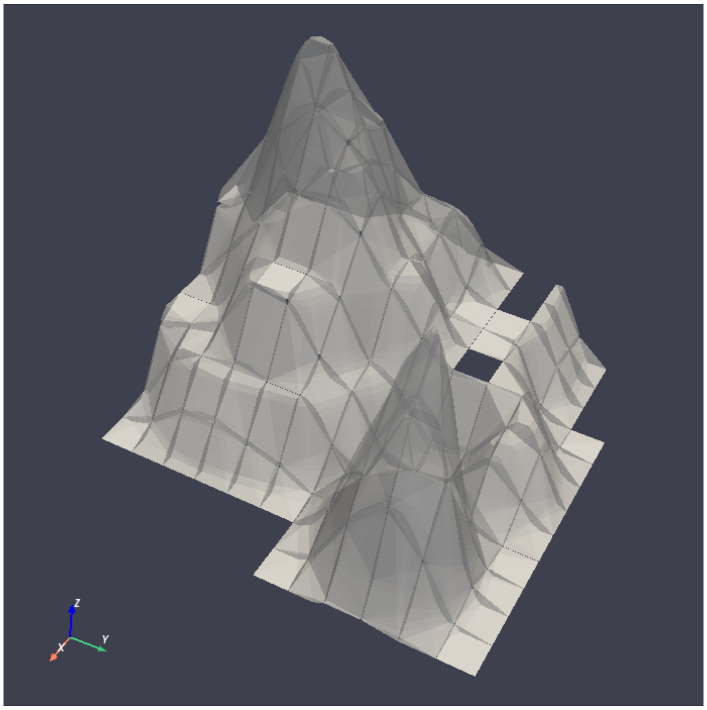
This `ImageMesh` transformation is also tested with a blurred version of the image with `cv2.blur`. A more smooth low-poly 3-D mesh is generated with this additional treatment. The topography seems more realistic:

```
img.array = cv2.blur(img.array,(60,60))    # blur the image for realiztic topography␣
↪levels
img.ImageMesh(out_file='model.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

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

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

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

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

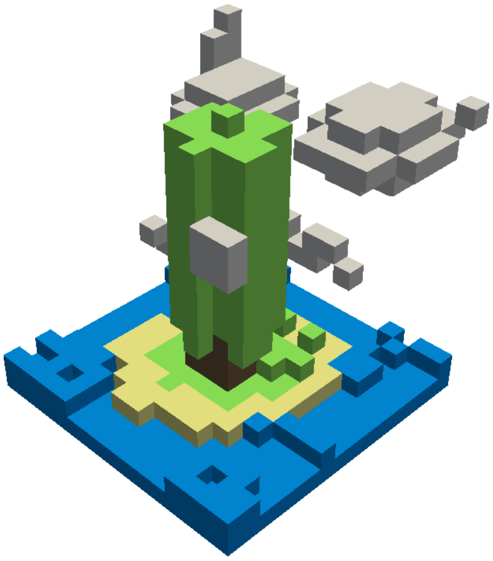
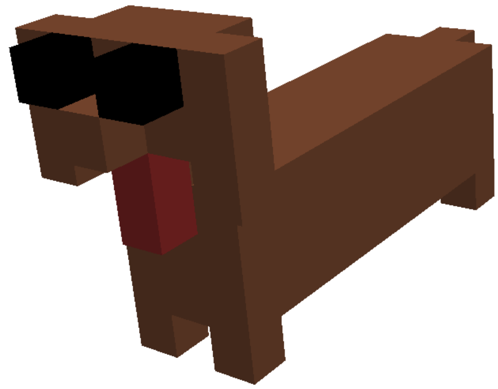


2.1.7 MarchingMesh : Turning Voxel Models to 3-D Mesh Representations

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.

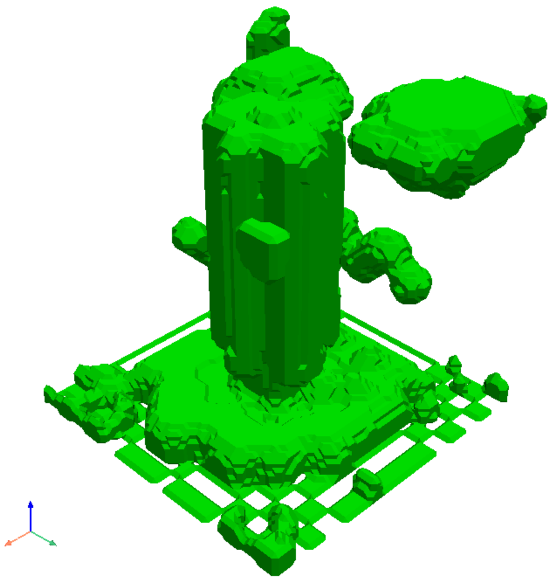
We first load those `.txt` files with the below voxelmap methods:

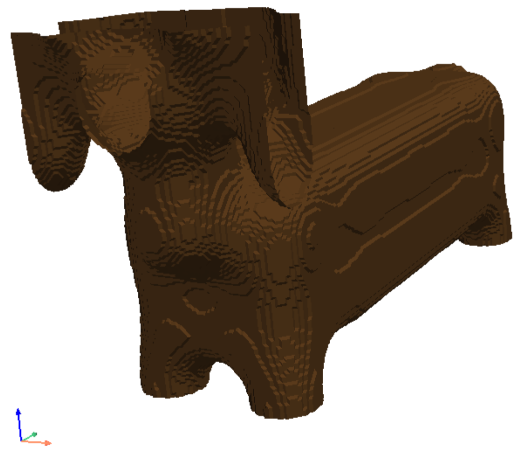
```
import voxelmap as vxm
import numpy as np

'''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␣
↪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.

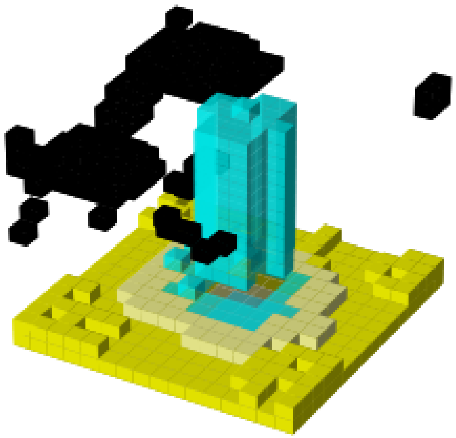
2.1.8 3-D Voxel Model Reprocessing

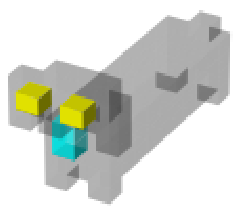
Here we do some reprocessing of the above *voxel* models. Note that here we use the `draw_mpl` method, which is `voxelmap`'s legacy method for voxel modeling and not its state-of-the-art. For faster and higher quality graphics with more kwargs / drawing options, use `voxelmap`'s `draw` method instead.

Re-color with custom colors

using the `hashblocksAdd()` method

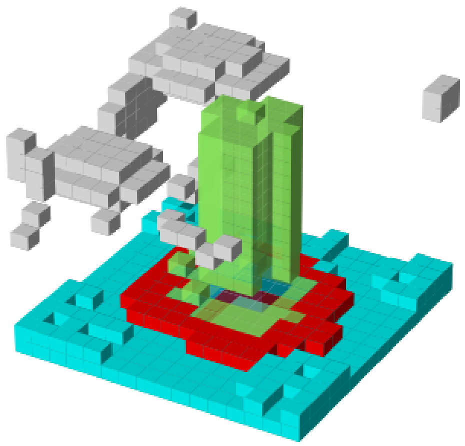
```
theIsland.hashblocksAdd(1,'yellow',1)
theIsland.hashblocksAdd(2,'#333333',0.2)
theIsland.hashblocksAdd(3,'cyan',0.75)
theIsland.hashblocksAdd(4,'#000000')

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

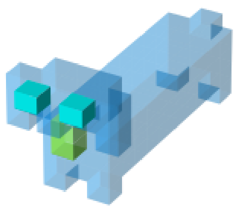
The `Model().load()` method processes the array and color information to a blank Model object. To load this data into a "blank slate" and re-draw it, type the following:

```
# 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:

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='model.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='model.obj'*, *wireframe=False*, *color='pink'*, *alpha=0.5*, *background_color='#333333'*, *viewport=[1024, 768]*)

Triangulated mesh view with PyVista (GLOBAL)

Parameters

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

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

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

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

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=[])`

`MarchingMesh(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 3D mapped image

Parameters

voxel_depth
[int] depth of 3-D mapped image on number of voxels

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(*wireframe=False*, *color='pink'*, *alpha=0.5*, *background_color='#333333'*, *viewport=[1024, 768]*)

Triangulated mesh view with PyVista

Parameters

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

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

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

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

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*, *window_size=[1024, 768]*, *voxel_spacing=(1, 1, 1)*)

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
- '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

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)).

`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)

`hashblocksAdd`(*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='voxeldata.json'*, *coords=False*)

Load to Model object.

Parameters

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

`save`(*filename='voxeldata.json'*)

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

Parameters

filename: string
name of file (e.g. 'voxeldata.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)

2.2.3 Local Methods to Image class

`class voxelmap.Image`(*file=''*)

`ImageMap`(*depth=5*)

Map image to 3-D array

Parameters

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

`ImageMesh`(*out_file='model.obj'*, *L_sectors=4*, *rel_depth=0.5*, *trace_min=5*, *plot=True*, *figsize=(4.8, 4.8)*, *verbose=False*)

3-D triangulation of 2-D images 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: 5)

plot: bool / str
plots a preliminary 3-D triangulated image if True [with PyVista (& with matplotlib if plot = 'img')

`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 3D-mapped image

Parameters

voxel_depth
[int] depth of 3-D mapped image on number of voxels

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`(*wireframe=False*, *color='pink'*, *alpha=0.5*, *background_color='#333333'*, *viewport=[1024, 768]*)

MeshView: triangulated mesh view with PyVista

Parameters

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

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

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

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

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)

`make`()

Turn image into intensity matrix i.e. matrix with pixel intensities

`resize`(*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)

voxelmap

voxelmap

CHAPTER

THREE

EXAMPLES

3.1 Island Mesh

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

```
#argisle.txt
import voxelmap as vxm
import numpy as np

model= vxm.Model()
model.load('argisle.txt')
model.array = np.transpose(model.array,(2,1,0))      #rotate dog
model.draw('custom',background_color='white')

'to convert to mesh'
model.array = vxm.resize_array(model.array,(5,5,5)) #make array larger before mesh␣
↪transformation
model.MarchingMesh()
model.MeshView(color='pink',wireframe=True,background_color='white',alpha=1,
↪viewport=[700,700])
```

3.2 Skull 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'
```

(continues on next page)

(continued from previous page)

```
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])
```

3.3 ImageMesh Portrait

This script makes the 3-D model file (.obj format) from the below image. The .obj file made in the last line may then be imported to a graphic editing software such as Blender or viewed with `voxelmap` i.e. adding `img.MeshView()` after the last line

```
#galactic.py
import voxelmap as vxm

img = vxm.Image('galactic.png')        # load image
img.objfile = 'galactic.obj'           # set name of 3-D model file (.obj) to be made
img.resize(0.25)                       # resize image to 25% its original size for␣
↪feasible file size rendering
img.MarchingMesh(25)                   # make 3-D model from image.
```

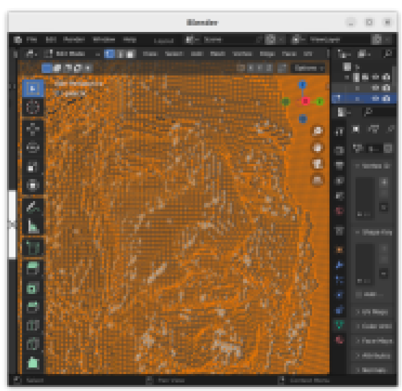
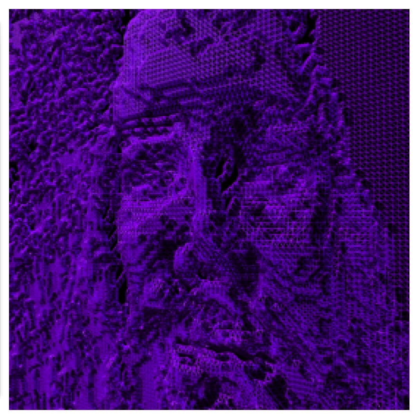




*Photo used with permission from the @galacticeffect.

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

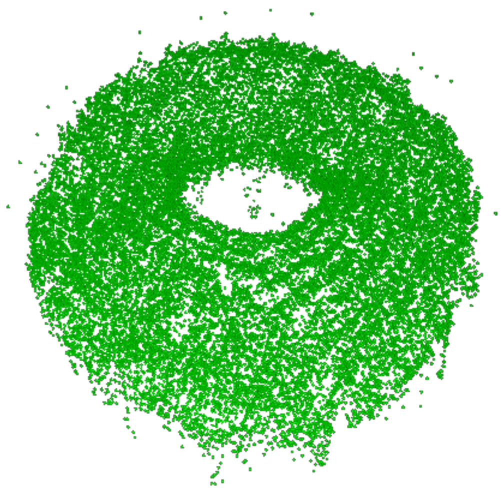


Fig. 2: 3-D reconstruction of donut image to voxel cloud done by transforming relative pixel intensities to 3-D depth with the ImageMap method.

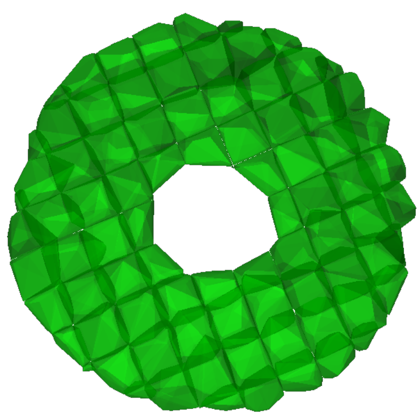


Fig. 3: 3-D reconstruction of donut image with the ImageMesh method.

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.

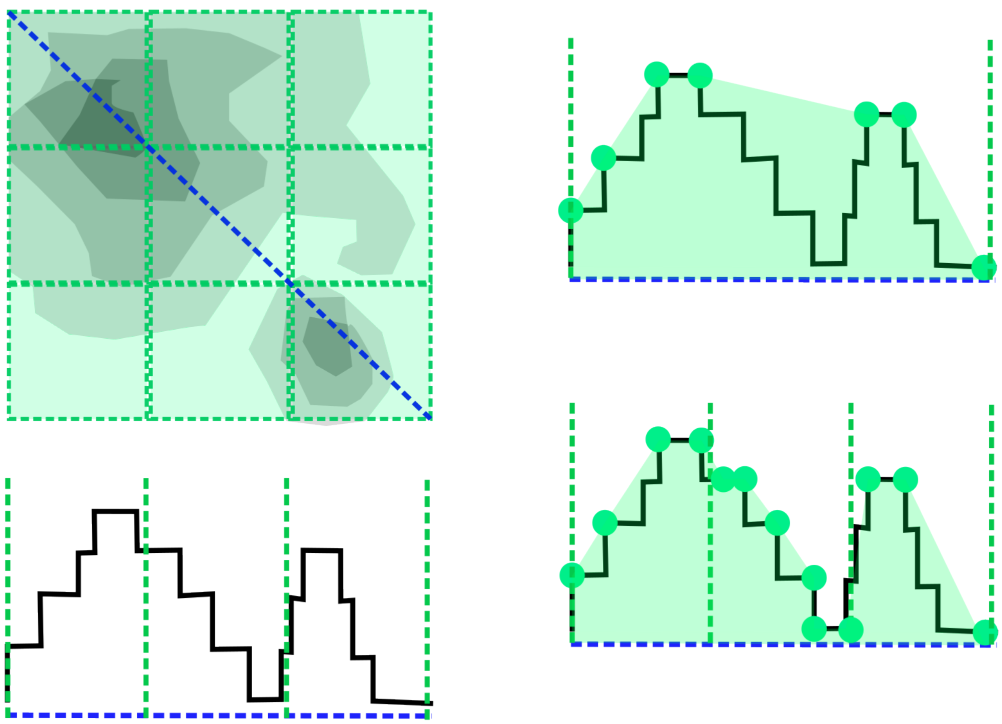


Fig. 4: Schematic representation of the performance of ImageMesh with more segments.

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

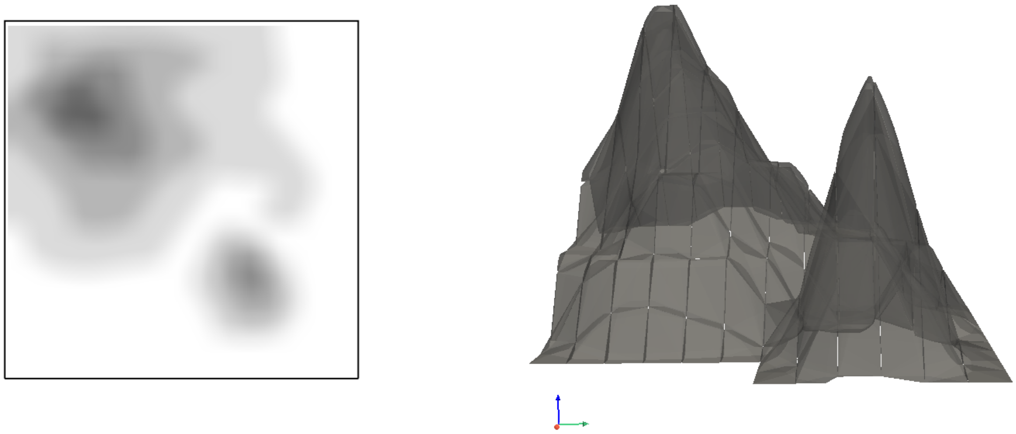


Fig. 5: Blurring of former image to create a smoother 3-D texture with ImageMesh.

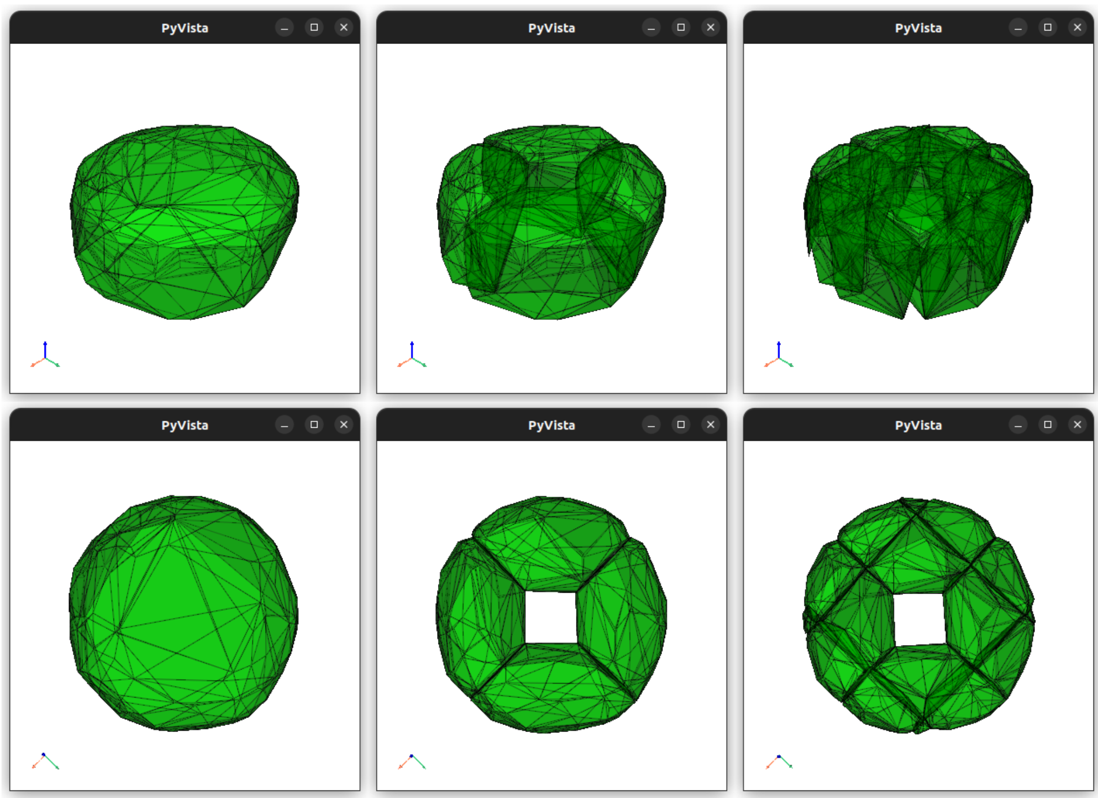


Fig. 6: The graphical effect of increasing the number of 3-D Convex Hull (CH) sectors in each column. The left column shows 1 CH sector, the middle column shows 4 CH sectors, and the right column shows 16 CH sectors.

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

V

INDEX