Final Project : Volumetric Surface Reconstruction

by Spencer Van Leeuwen

In [1]:
```
%matplotlib inline

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import mayavi.mlab

import numpy as np
import math
import matplotlib.pyplot as plt
import maxflow
from skimage import img_as_ubyte
from skimage.color import rgb2grey
from skimage.color import rgb2hsv
```

Part 1 : Segmentation

Loading the dataset

First, we start by loading the dataset. I will be using the temple dataset by Steven Seitz *et al.* (http://grail.cs.washington.edu/projects/mview/). All data is stored in the "temple/" directory. It contains a set of photos of a sculpture taken from different angles.

A sample image is provided below.

In "temple/temple_par.txt", we have a parameters file that starts with the number of images in the folder. Every subsequent line has the following format:

```
file_name k11 k12 k13 k21 k22 k23 k31 k32 k33 r11 r12 r13 r21 r22 r23 r31 r3
2 r33 t1 t2 t3
```

where the projection matrix of the given camera is $P = K\ [R \mid T]$.

In [2]:
```
img_dir = 'temple/'
file_root = 'temple'
params = []

with open(img_dir + file_root + '_par.txt', 'r') as f:
    for line in f:
        params.append(line.split())

num_images = params[0][0]
params = params[1:]

images = []

for par in params:
    images.append(plt.imread(img_dir + par[0]))
```

Segmentation of the images

I have altered the graph cuts code that I submitted for Assignment 3 so that it no longer required user input. Instead, I initialize it by selecting high-intensity pixels as object pixels. In particular, I convert the image to greyscale and take the pixels in the 80th percentile of intensity. K-means with $k = 2$ was giving me trouble. K-means could probably be used if we wanted something more general, but I figure that I might as well take advantage of the simplicity of the dataset.

Recall that in Assignment 3, we used k-means to segment the colours then compute the t-links. As an alternative, we were allowed to use colour histogram binning. I started off by using k-means since I already had it implemented from Assignment 3. However, I found that the randomness made it unreliable. In particular, since there is no longer an interactive component, poor groupings by k-means were leading to the algorithm including the table cloth as part of the object. I couldn't find values for sigma and the regulizer that would mitigate this problem. However, I have altered the algorithm to use colour histogram binning and this solved the problem.

Now, we are able to iteratively re-compute the weights of the t-links depending on the likelihood of an object or background pixel being in a given bin. I iterate until < 1% of the pixels change their label from one iteration to the next.

In [3]:
```
class MyGraphCuts:
    bgr_value = 0
    obj_value = 1
    none_value = 2

    def __init__(self, img, sigma=1, regularizer=1, bins_per_channel=
10, init_percentile=90):
        self.num_rows = img.shape[0]
        self.num_cols = img.shape[1]
        self.shape = img.shape[:2]

        self.img = img.copy().astype('d')

        self.sigma = sigma
        self.regularizer = regularizer
        self.bins_per_channel = bins_per_channel

        self.max_weight = 2**5-1
        self.t_inf = max(4*self.max_weight, regularizer*4*self.max_we
ight)

        self.label_mask = self.compute_label_mask()
        self.seed_mask = self.high_intensity_mask(init_percentile)

    # Returns pixels in the 90th percentile of intensity
    def high_intensity_mask(self, p):
        grey_img = rgb2grey(self.img)
        bright_pixels = np.percentile(grey_img, p)

        return (np.sum(self.img, axis=2) > bright_pixels).astype('i')

    # Link weights are regularized to be in the range [0,max_weight].
 This ensures that edge weights
    # are small integers in order to take advantage of optimizations
 in the graph cut library
    def regularize(self, weights, n_link=True):
        min_weight = np.amin(weights)
        max_weight = np.amax(weights)

        weights[:,:] = (self.regularizer if n_link else 1) * \
                        (self.max_weight) * (weights - min_weight)/ab
s(max_weight - min_weight)

    # n-link weights are computed depending on the difference of pixe
l intensities
    def compute_grid_weights(self):
        hor_weights = np.zeros(self.shape, 'd')
        hor_weights[:,:-1] = np.exp(- (np.linalg.norm(self.img[:,:-1]
 - self.img[:,1:], axis=2)**2 / self.sigma**2))
        self.regularize(hor_weights)

        vert_weights = np.zeros(self.shape, 'd')
        vert_weights[:-1,:] = np.exp(- (np.linalg.norm(self.img[:-1
,:] - self.img[1:,:], axis=2)**2 / self.sigma**2))
        self.regularize(vert_weights)
```

```
        return (hor_weights, vert_weights)

    # We use a 4-connected grid connecting the pixels
    def add_grid_edges(self, g, nodeids):
        hor_struct = np.array([[0, 0, 0],
                               [0, 0, 1],
                               [0, 0, 0]])

        vert_struct = np.array([[0, 0, 0],
                                [0, 0, 0],
                                [0, 1, 0]])

        hor_weights, vert_weights = self.compute_grid_weights()

        g.add_grid_edges(nodeids, weights=hor_weights, structure=hor_
struct, symmetric=True)
        g.add_grid_edges(nodeids, weights=vert_weights, structure=ver
t_struct, symmetric=True)

    # Compute t-links depending on the likelihood of a pixel's label
 (object or background)
    # belonging to a given colour histogram bin
    def compute_tlinks(self, seed_mask):
        s_tlinks = np.zeros(self.shape) # background
        t_tlinks = np.zeros(self.shape) # object

        bgr_indices = seed_mask == self.bgr_value
        obj_indices = seed_mask == self.obj_value

        bgr_labels = self.label_mask[seed_mask == self.bgr_value]
        obj_labels = self.label_mask[seed_mask == self.obj_value]

        bgr_size = np.sum((bgr_indices).astype('d'))
        obj_size = np.sum((obj_indices).astype('d'))

        bgr_likelihoods = np.zeros(self.shape, 'd')
        obj_likelihoods = np.zeros(self.shape, 'd')

        for label in range(self.bins_per_channel**3):
            if bgr_size > 0:
                bgr_cluster_size = np.sum((bgr_labels == label).astyp
e('d'))
                bgr_cluster_size = max(bgr_cluster_size, 1)
                bgr_likelihoods[self.label_mask == label] = bgr_clust
er_size / bgr_size

            if obj_size > 0:
                obj_cluster_size = np.sum((obj_labels == label).astyp
e('d'))
                obj_cluster_size = max(obj_cluster_size, 1)
                obj_likelihoods[self.label_mask == label] = obj_clust
er_size / obj_size

        s_tlinks = -np.log(obj_likelihoods)
        t_tlinks = -np.log(bgr_likelihoods)

        self.regularize(s_tlinks, False)
```

```
        self.regularize(t_tlinks, False)

        return (s_tlinks, t_tlinks)

    # Greyscale function to evaluate which pixels in img have intensi
ty in
    # the percentile range [lower, upper)
    def get_pixels_in_range(self, img, lower, upper):
        lower_bound = np.percentile(img, lower)

        if abs(upper - 100) < 0.01:
            upper_bound = np.max(img) + 1
        else:
            upper_bound = np.percentile(img, upper)

        return np.logical_and(img >= lower_bound, img < upper_bound)

    # Compute the label_mask based on a colour histogram
    def compute_label_mask(self):
        bins = self.bins_per_channel

        red_img = self.img[:,:,0]
        green_img = self.img[:,:,1]
        blue_img = self.img[:,:,2]

        label_mask = np.full(self.shape, bins**3)
        voxel_width = 100.0 / bins

        for r in range(bins):
            r_lower = voxel_width * r
            r_upper = voxel_width * (r+1)

            r_in_range = self.get_pixels_in_range(red_img, r_lower, r
_upper)

            for g in range(bins):
                g_lower = voxel_width * g
                g_upper = voxel_width * (g+1)

                g_in_range = self.get_pixels_in_range(green_img, g_lo
wer, g_upper)

                for b in range(bins):
                    b_lower = voxel_width * b
                    b_upper = voxel_width * (b+1)

                    b_in_range = self.get_pixels_in_range(blue_img, b
_lower, b_upper)

                    in_bin = np.logical_and(r_in_range, g_in_range, b
_in_range)
                    index = b + bins*g + (bins**2)*r

                    label_mask[in_bin] = index

        self.k = bins**3
        return label_mask
```

```
    # This is the "main" function. The framework of the algorithm is
 performed here.
    def compute_labels(self):
        seed_mask = self.seed_mask

        epsilon = 0.01
        delta = 1

        while delta > epsilon:
            g = maxflow.GraphInt()
            nodeids = g.add_grid_nodes(self.shape)

            # Add weighted grid edges (n-links)

            self.add_grid_edges(g, nodeids)

            # Compute weights for colour consistency t-links

            s_tlinks, t_tlinks = self.compute_tlinks(seed_mask)

            # Add t-links to graph

            g.add_grid_tedges(nodeids, s_tlinks, t_tlinks)

            # Compute min cut segments
            g.maxflow()
            sgm = g.get_grid_segments(nodeids)

            label_mask = np.full(self.shape, self.none_value, dtype=
'uint8')
            label_mask[sgm] = self.bgr_value
            label_mask[np.logical_not(sgm)] = self.obj_value

            # Compute delta to see if the algorithm should iterate ag
ain
            changed_pixels = (seed_mask != label_mask)
            img_size = self.shape[0]*self.shape[1]

            delta = np.sum(changed_pixels) / img_size

            seed_mask = label_mask

        return label_mask
```

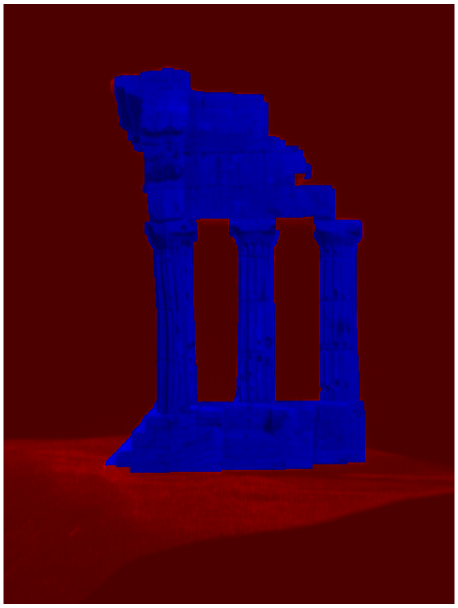
Run segmentation algorithm

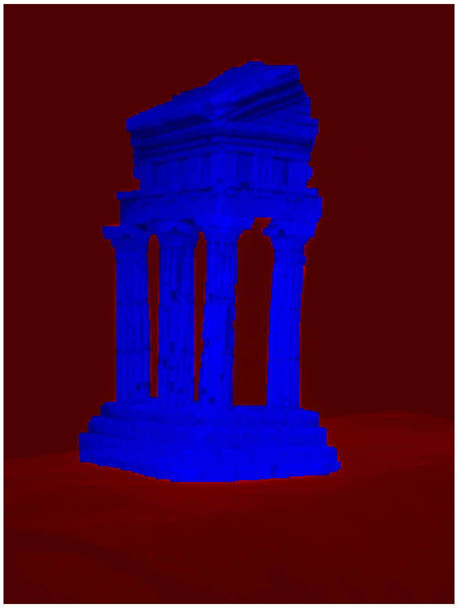
Here, we run the segmentation algorithm and save each mask in a separate directory. Greyscale images with the masks overlayed are also saved. This allows us to inspect the accuracy of the segmentation.

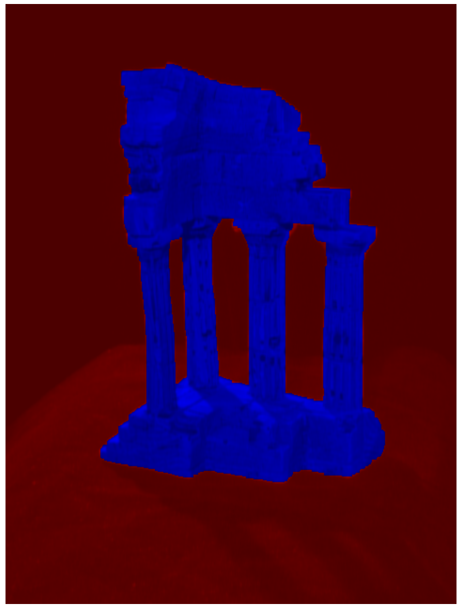
In [4]:
```
def print_progress(index, length, current):
    progress = (((float)(index) / length) * 10)
    if ((int)(progress) > current):
        print (str)((int)(progress)*10) + ('% complete' if index == 0
 else '%')
        return 1
    return 0

def save_mask(img, par, mask_dir):
    mask_img_dir = 'temple_mask_images/'
    mask_file = mask_dir + par[0]
    mask = np.load(mask_file + '.npy')

    img_grey = rgb2grey(img) * 0.7 + 0.3

    mask_img = np.zeros(img.shape)
    mask_img[mask == 1, 0] = img_grey[mask == 1]
    mask_img[mask == 0, 2] = img_grey[mask == 0]

    plt.imsave(mask_img_dir + par[0], mask_img)

def run_segmentation():
    mask_dir = 'temple_mask/'

    j = -1

    a,b = 0,len(images)

    for img, par, i in zip(images[a:b], params[a:b], range(len(images
[a:b]))):
        app = MyGraphCuts(img, sigma=0.1, regularizer=300, bins_per_c
hannel=6)

        filename = mask_dir + par[0]
        np.save(filename, app.compute_labels())

        j += print_progress(i, len(images[a:b]), j)
        save_mask(img, par, mask_dir)

    print 'done'

# The line below is commented out because the masks are saved,
# so we don't need to re-run the segmentation after restarting the ke
rnel

# run_segmentation()
```

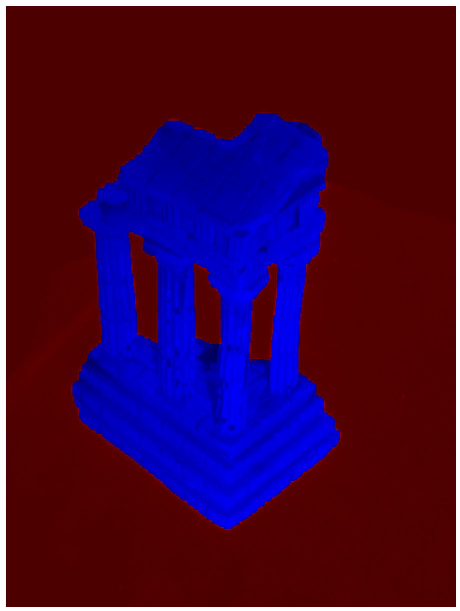
Results

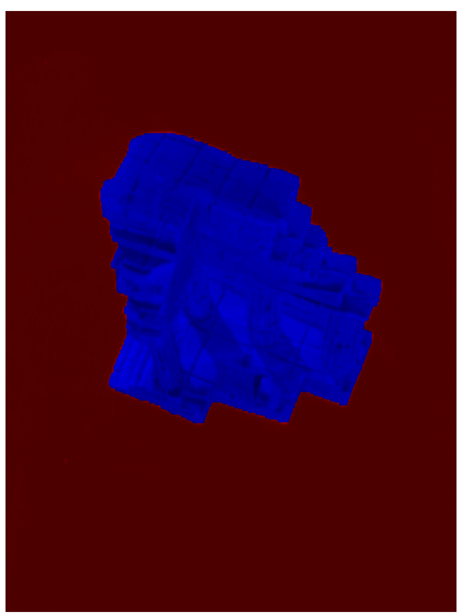
Here, we present the results of our segmentation. Note that the images were rotated 90 degrees when provided by Steven Seitz *et al.* I have rotated them upright for presentation purposes.





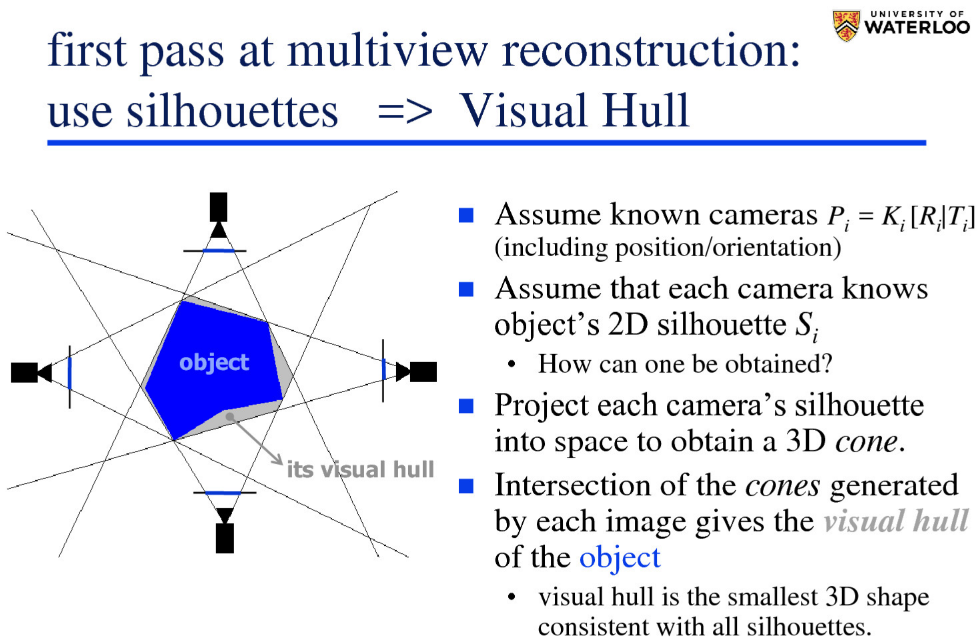






Part 2: Visual Hull

Here is a visual description of what we do in this section from the lecture notes:



Setting up the grid

Here, we set up a 3D grid. The grid will subdivide the bounding box for the object into voxels. As such, it will have the same world position and dimensions as the bounding box of the object.

In [5]:
```
class VoxelGrid:
    def __init__(self, bound_min, bound_max, res):
        self.bound_min = np.array(bound_min, dtype='f')
        self.bound_max = np.array(bound_max, dtype='f')

        # res is the resolution along the longest axis of the boundin
g box
        # The width of each cubic voxel is calculated accordingly
        self.v_width = np.max(np.array((bound_max - bound_min) / res,
 dtype='f'))
        self.shape = np.ceil(((bound_max - bound_min) / self.v_width
)).astype('i')

        self.obj_mask = np.full(self.shape, True)
        self.init_center_positions()

        # Pixel dimensions of images from dataset
        self.img_dim = (480, 640)

    def get_voxels(self):
        return self.obj_mask

    def get_positions(self):
        return self.centers

    def init_center_positions(self):
        # Center of (0,0,0)
        min_pos = self.bound_min + 0.5*self.v_width

        # Make 3D grid where each cell contains (x, y, z) position
        pos_grid = np.stack(np.mgrid[ :self.shape[0], :self.shape[1],
 :self.shape[2]].astype('f'), -1)

        # Add multiples of voxel width to the center of (0,0,0)
        pos_grid = min_pos + (pos_grid * self.v_width)

        # Concatenate ones to make homogeneous coordinates
        pos_grid = np.concatenate((pos_grid, np.ones(np.append(self.s
hape[:3], 1), dtype='f')), axis=3)

        # Transpose position vectors to column vectors
        pos_grid = pos_grid.reshape(np.append(pos_grid.shape, 1))

        self.centers = pos_grid

    def visual_hull(self, images, params, indices):
        size = len(images)
        j = -1

        for img, par, i in zip(images, params, range(len(images))):
            self.cull_projection(img, par)

            if (int)(i / (float)(size) * 10) > j:
                j += print_progress(i, size, j)

    def get_proj_mat(self, par):
```

```
        K = np.reshape(par[1:10], (3,3)).astype('f')
        R = np.reshape(par[10:19], (3,3)).astype('f')
        T = np.reshape(par[19:22], (3,1)).astype('f')

        return K.dot( np.append(R,T, axis=1) )

    def compute_pixel_projections(self, par):
        P = self.get_proj_mat(par)

        # Project the center of each voxel onto the image plane
        proj = (P.dot(self.centers)).transpose((1, 2, 3, 0, 4))

        # Normalize the image coordinates and truncate them to get pi
xel indices
        proj = np.concatenate(((proj[:,:,:,0]/proj[:,:,:,2]).reshape(
np.append(self.shape,1)),
                                   (proj[:,:,:,1]/proj[:,:,:,2]).resh
ape(np.append(self.shape,1))), \
                              axis=3).astype('i')

        proj[:,:,:,0] = np.clip(proj[:,:,:,0], 0, self.img_dim[1]-1)
        proj[:,:,:,1] = np.clip(proj[:,:,:,1], 0, self.img_dim[0]-1)

        # Images coordinates are (x,y), so we flip x and y
        proj_trans = np.empty(proj.shape, dtype='i')
        proj_trans[:,:,:,0] = proj[:,:,:,1]
        proj_trans[:,:,:,1] = proj[:,:,:,0]

        return proj_trans

    def cull_projection(self, img, par):
        img_mask = self.load_mask(par)

        proj = self.compute_pixel_projections(par)
        #valid = np.logical_and(proj[:,:,:,1] >=0, np.logical_and(pro
j[:,:,:,1] < img.shape[0], \
        #                      np.logical_and(proj[:,:,:,0] >= 0, pr
oj[:,:,:,0] < img.shape[1])))

        #self.obj_mask[valid] = np.logical_and(self.obj_mask[valid],
 img_mask[proj[valid][:,1], proj[valid][:,0]] == 0)
        self.obj_mask = np.logical_and(self.obj_mask, img_mask[proj
[:,:,:,0], proj[:,:,:,1]] == 0)

    def load_mask(self, par):
        mask_dir = 'temple_mask/'
        img_name = par[0]

        return np.load(mask_dir + img_name + '.npy')
```

Initializing the grid

The parameters of the bounding box used below were provided in 'temple/README.txt' alongside the dataset.

In [6]:
```
# Set up the voxel grid
bound_min = np.array([-0.054568, 0.001728, -0.042945])
bound_max = np.array([0.047855, 0.161892, 0.032236])

grid_resolution = 2**9

def init_grid():
    return VoxelGrid(bound_min, bound_max, grid_resolution)

grid = init_grid()
```

Running the visual hull

After running the graph cuts, some of the images had object pixels marked as background pixels. This could have been solved by:

- playing with image-independent parameters
- a more robust algorithm
- a semi-supervised algorithm
- simply discard undesirable images

I have opted to discard the undesirable images. This is partly due to time constraints and partly because the project specifications explicitly said that we were allowed to use a subset of the images. That being said, you will see in the cell below that the number of remaining images is still reasonable.

In [7]:
```
def run_visual_hull():
    # Run the visual hull using a subset of the sample images
    sample_indices = [0,2,3,4,5,8,9,10,11,12,14,15,16,17,18,19,20,
                       21,22,23,31,32,33,34,35,36,37,38,39,40,41,42,
                       43,48,49,50,51,52,53,54,61,62,63,64,65,66,67,
                       68,74,76,77,78,79,80,84,85,86,87,88,89,
                       96,97,98,99,100,101,102,103,104,105,106,107,
                      108,109,116,117,118,119,120,121,122,123,124,125,
126,
                      130,131,132,134,138,139,140,142,144,145,146,147,
                      148,149,150,151,152,153,154,155,156,157,158,159,
160,161,
                      163,165,166,167,168,169,170,171,172,174,175,176,
177,178,
                      180,181,182,183,184,185,186,187,188,189,190,191,
192,
                      194,195,196,197,198,199,200,201,202,203,204,205,
206,
                      207,208,209,210,211,212,213,214,215,216,217,218,
219,
                      221,222,223,224,225,226,227,228,229,230,231,232,
                      233,234,235,236,237,239,240,241,242,244,246,247,
                      248,249,250,251,252,253,254,255,262,263,264,265,
                      268,271,273,274,275,279,286,287,288,293,296,297,
                      303,306,308,311]

    sample_images = np.array(images)[sample_indices]
    sample_params = np.array(params)[sample_indices]

    grid.visual_hull(sample_images, sample_params, sample_indices)
    print 'done'

# run_visual_hull()
```

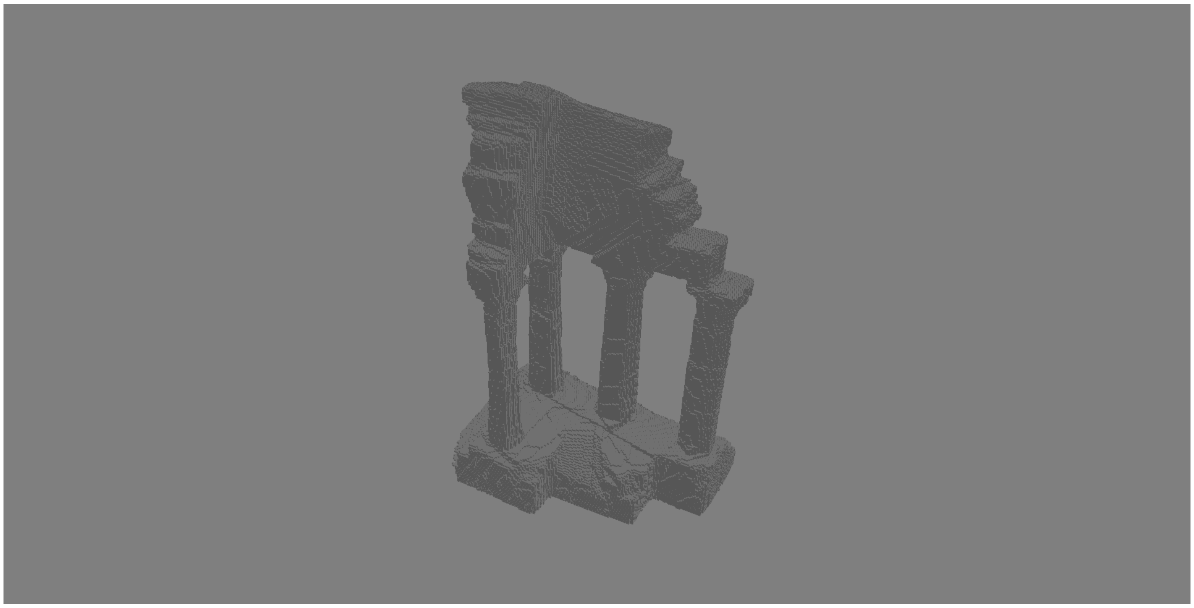
Rendering the voxel grid

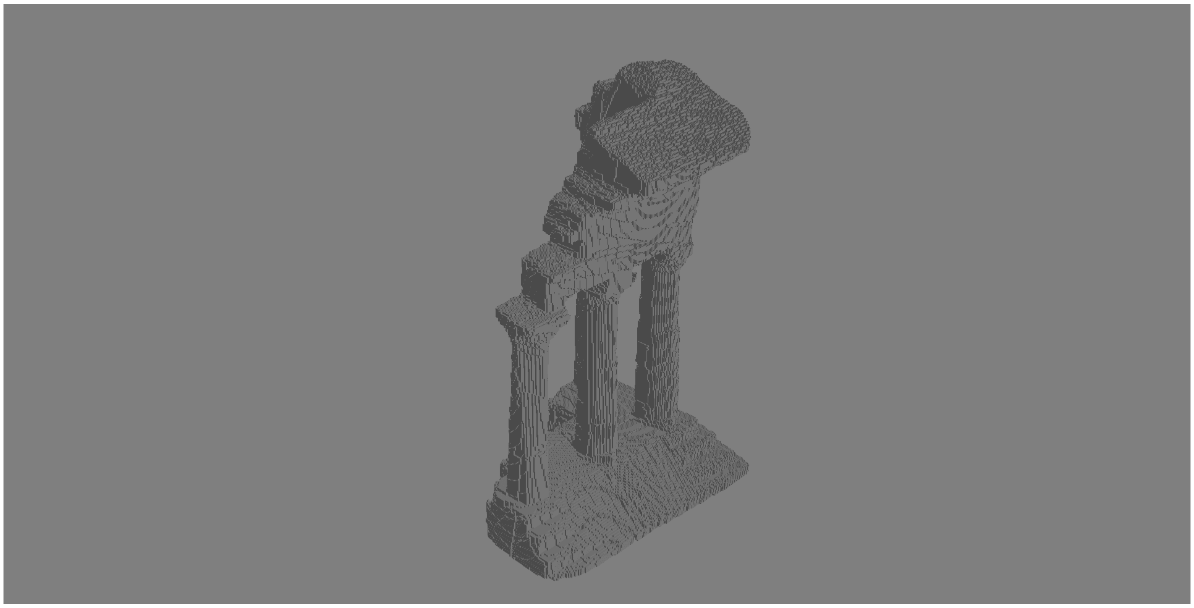
Here, the object's voxels are rendered using Mayavi.

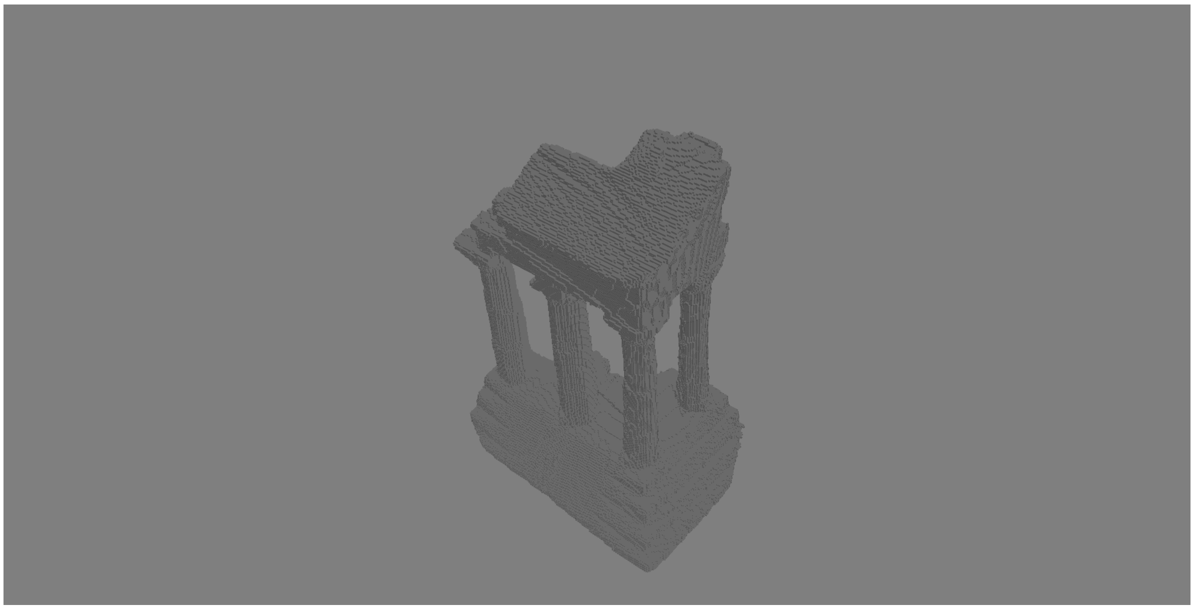
In [8]:
```
def plot_voxels(grid):
    voxels = grid.get_voxels()

    # Transpose and flip axes to get the proper view
    voxels = voxels.transpose((0,2,1))[:,::-1,:]

    xx, yy, zz = np.where(voxels == 1)
    mayavi.mlab.points3d(xx,yy,zz,
                         mode='cube',
                         color=(0.5,0.5,0.5),
                         scale_factor=1)
    mayavi.mlab.show()

# plot_voxels(grid)
```

Results

Here, we can see the results of our visual hull algorithm. This library uses flat shading for different faces of each voxel. Since all voxels are aligned (rotationally), the camera has been positioned carefully so that we can see the details of the model. If the camera was near perpendicular to any of the voxel faces, the object appeared as a blob (a shape shaded with a single colour). Even some of the provided images are better than others due to this limitation. Ideally, we would use a surface interpolation algorithm (e.g., marching cubes) and lambertian shading (although, that would require substantial effort and is beyond the scope of this project).

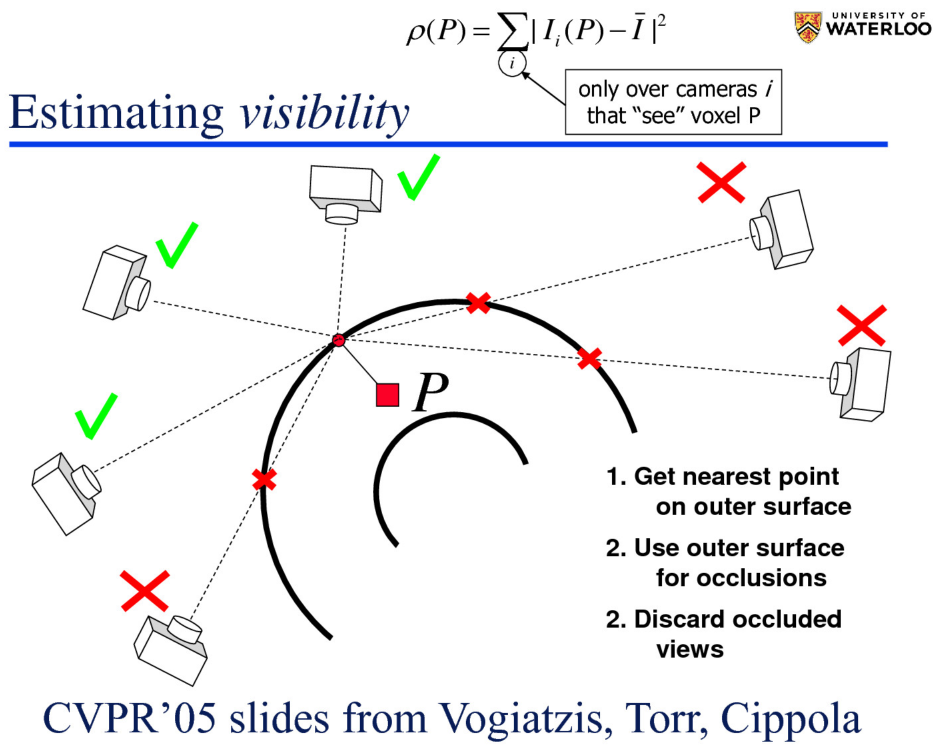






Part 3: Photoconsistency

To start, we need to estimate the visibility of any given voxel. For illustration purposes, here is the corresponding course lecture slide:



Signed distance function

First, we need to start by getting the closest point on the surface to all voxels belonging to the visual hull. To accomplish this, we begin by computing the signed distance function of the grid. This is computed using the fast sweeping method described by Bridson:

```
Bridson, Robert. Fluid simulation for computer graphics. AK Peters/CRC Pres
s, 2015.
```

In particular, we will set the distance of all surface voxels to 0. We will then sweep along each of the axes and compute the distance of adjacent cells, replacing the current distance to the surface if the new distance is smaller. Since the distances along any given axis may change as we update the grid, we will perform this sweep multiple times (Bridson recommends twice). Note, the voxels inside of the object will be given negative distances so that derivatives point towards the surface.

Implementation note

In order to make the project readable in a linear fashion, I am going to be defining functions then binding them to the VoxelGrid class. Note that this is the same as defining the function inside the class.

In [9]:
```
# Define VoxelGrid class functions
def init_distances(self):
    # Set distance for all voxels to minus functional infinity
    func_inf = np.max(self.bound_max - self.bound_min)
    self.dist = np.full(self.shape, -func_inf, dtype='f')

    # Set outside values to the voxel width. This will be needed to g
et the correct gradient at the surface.
    self.dist[self.obj_mask == False] = self.v_width

    # Set surface voxels to 0
    self.set_boundary_voxels()

# Look at the outer layer of the bounding box and set a voxel as boun
dary (distance = 0)
# if it belongs to the object
def set_boundary_along_border(self):
    (self.dist[0, :, :])[self.obj_mask[0, :, :]] = 0
    (self.dist[self.shape[0]-1, :, :])[self.obj_mask[self.shape[0]-1,
 :, :]] = 0

    (self.dist[:, 0, :])[self.obj_mask[:, 0, :]] = 0
    (self.dist[:, self.shape[1]-1, :])[self.obj_mask[:, self.shape[1]
-1, :]] = 0

    (self.dist[:, :, 0])[self.obj_mask[:, :, 0]] = 0
    (self.dist[:, :, self.shape[2]-1])[self.obj_mask[:, :, self.shape
[2]-1]] = 0

# Mark each object voxel as a boundary voxel (distance = 0) if one of
 its adjacent voxels is outer
def set_boundary_voxels(self):

    # Start with border voxels. They are on the boundary if they belo
ng to the object
    self.set_boundary_along_border()

    # Check left voxel
    is_boundary = np.logical_and(self.obj_mask[1:, :, :],
                                 self.obj_mask[:-1, :, :] == False)
    (self.dist[1:self.shape[0], :, :])[is_boundary] = 0

    # Check right
    is_boundary = np.logical_and(self.obj_mask[:-1, :, :],
                                 self.obj_mask[1:, :, :] == False)
    (self.dist[:self.shape[0]-1, :, :])[is_boundary] = 0

    # Check down
    is_boundary = np.logical_and(self.obj_mask[:, 1:, :],
                                 self.obj_mask[:, :-1, :] == False)
    (self.dist[:, 1:self.shape[1], :])[is_boundary] = 0

    # Check up
    is_boundary = np.logical_and(self.obj_mask[:, :-1, :],
                                 self.obj_mask[:, 1:, :] == False)
    (self.dist[:, :self.shape[1]-1, :])[is_boundary] = 0
```

```
    # Check in front
    is_boundary = np.logical_and(self.obj_mask[:, :, 1:],
                                 self.obj_mask[:, :, :-1] == False)
    (self.dist[:, :, 1:self.shape[2]])[is_boundary] = 0

    # Check behind
    is_boundary = np.logical_and(self.obj_mask[:, :, :-1],
                                 self.obj_mask[:, :, 1:] == False)
    (self.dist[:, :, :self.shape[2]-1])[is_boundary] = 0

# After initializing the boundaries, we can calculate distances for t
he object voxels
def fast_plane_sweep(self, num_iterations=2):

    for it in range(num_iterations):

        # Sweep x-axis increasing
        for i in range(self.shape[0])[1:-1]:
            obj_voxels = self.obj_mask[i,:,:]
            self.dist[i][obj_voxels] = np.maximum(self.dist[i][obj_vo
xels],
                                                  self.dist[i-1][obj_v
oxels] - self.v_width)

        # Sweep x-axis decreasing
        for i in range(self.shape[0])[::-1][1:-1]:
            obj_voxels = self.obj_mask[i,:,:]
            self.dist[i][obj_voxels] = np.maximum(self.dist[i][obj_vo
xels],
                                                  self.dist[i+1][obj_v
oxels] - self.v_width)

        # Sweep y-axis increasing
        for j in range(self.shape[1])[1:-1]:
            obj_voxels = self.obj_mask[:,j,:]
            self.dist[:,j,:][obj_voxels] = np.maximum(self.dist[:,j
,:][obj_voxels],
                                                  self.dist[:, j-1, :]
[obj_voxels] - self.v_width)

        # Sweep y-axis decreasing
        for j in range(self.shape[1])[::-1][1:-1]:
            obj_voxels = self.obj_mask[:,j,:]
            self.dist[:,j,:][obj_voxels] = np.maximum(self.dist[:,j
,:][obj_voxels],
                                                  self.dist[:, j+1, :]
[obj_voxels] - self.v_width)

        # Sweep z-axis increasing
        for k in range(self.shape[2])[1:-1]:
            obj_voxels = self.obj_mask[:, :, k]
            self.dist[:,:,k][obj_voxels] = np.maximum(self.dist[:,:,k
][obj_voxels],
                                                  self.dist[:, :, k-1]
[obj_voxels] - self.v_width)
```

```
        # Sweep z-axis decreasing
        for k in range(self.shape[2])[::-1][1:-1]:
            obj_voxels = self.obj_mask[:, :, k]
            self.dist[:,:,k][obj_voxels] = np.maximum(self.dist[:,:,k
][obj_voxels],
                                                       self.dist[:, :, k+1]
[obj_voxels] - self.v_width)

# Add padding to the distance function. This allows us to get the cor
rect gradient for surface
# object voxels on the boundary of the grid
def add_dist_padding(self):
    old_dist = self.dist.copy()

    self.dist = np.full( np.array(self.dist.shape) + 2, self.v_width,
 dtype='f')
    self.dist[1:-1,1:-1,1:-1] = old_dist

# Bind VoxelGrid class functions
VoxelGrid.init_distances = init_distances
VoxelGrid.set_boundary_along_border = set_boundary_along_border
VoxelGrid.set_boundary_voxels = set_boundary_voxels
VoxelGrid.fast_plane_sweep = fast_plane_sweep
VoxelGrid.add_dist_padding = add_dist_padding

# Call VoxelGrid class functions

# grid.init_distances()
# grid.fast_plane_sweep()
# grid.add_dist_padding()

print 'done'
```

```
done
```

Gradient of the signed distance function

Taking the derivative of the signed distance function gives us the direction of the closest point on the surface.

For the derivative, we will be using central differences.

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h}$$

Note that the derivatives will be computed for points between voxels (i.e., the center of the voxel faces), so it may resemble to be forward differences when examining the code.

Then, we will compute the gradient at the grid centers. This will essentially average derivatives on either face of a voxel and combine everything into a single structure.

In [10]:
```
def compute_derivatives(self):
    self.diff_x = (self.dist[1:,:,:] - self.dist[:-1,:,:]) / self.v_w
idth
    self.diff_y = (self.dist[:,1:,:] - self.dist[:,:-1,:]) / self.v_w
idth
    self.diff_z = (self.dist[:,:,1:] - self.dist[:,:,:-1]) / self.v_w
idth
    return

# Compute the gradient and normalize it so that it has unit length
# Note that the returned gradient has no padding (i.e., it has the sa
me shape as obj_mask, not dist)
def compute_gradient(self):
    grad_x = (self.diff_x[ :-1,1:-1,1:-1] + self.diff_x[1:  ,1:-1,1:-
1]) / 2.0
    grad_y = (self.diff_y[1:-1, :-1,1:-1] + self.diff_y[1:-1,1:  ,1:-
1]) / 2.0
    grad_z = (self.diff_z[1:-1,1:-1, :-1] + self.diff_z[1:-1,1:-1,1:
 ]) / 2.0
    self.gradient = np.concatenate((grad_x.reshape(grad_x.shape + (1
,)),
                                    grad_y.reshape(grad_y.shape + (1
,)),
                                    grad_z.reshape(grad_z.shape + (1
,))),
                                  axis=3)

    gradient_lengths = np.linalg.norm(self.gradient, axis=3)
    non_zero = gradient_lengths != 0

    gradient_lengths = np.concatenate((gradient_lengths.reshape((grad
ient_lengths.shape + (1,))),
                                      gradient_lengths.reshape((grad
ient_lengths.shape + (1,))),
                                      gradient_lengths.reshape((grad
ient_lengths.shape + (1,)))),
                                     axis=3)

    self.gradient[non_zero] = self.gradient[non_zero] / gradient_leng
ths[non_zero]

# Bind VoxelGrid class functions
VoxelGrid.compute_derivatives = compute_derivatives
VoxelGrid.compute_gradient = compute_gradient

# Call VoxelGrid class functions

# grid.compute_derivatives()
# grid.compute_gradient()

print 'done'
```

done

Closest surface point

We now have the distance from each voxel center to the surface and the direction of the surface (i.e., the gradient of the signed distance function). This means that we can easily estimate the closest surface point. Since we have already lost accuracy by discretizing the space, we will round the surface point to the nearest voxel center so that we have access to its gradient.

In [11]:
```
# Now that we have computed gradients, we no longer require padding.
 We will remove it so that the shape
# of self.dist matches the shape of self.gradient
def remove_dist_padding(self):
    if self.dist.shape[:3] != self.gradient.shape[:3]: # only for deb
ugging in case function is called twice
        self.dist = self.dist[1:-1,1:-1,1:-1]

# Each voxel will contain the index of the closest surface voxel
def compute_closest_surface_point(self):
    # Compute estimated closest surface point
    distance = np.concatenate((self.dist.reshape(self.dist.shape + (1
,)),
                               self.dist.reshape(self.dist.shape + (1
,)),
                               self.dist.reshape(self.dist.shape + (1
,))),
                              axis=3)

    surface_points =  self.centers[:,:,:,:3,0] + self.gradient * dist
ance

    # Convert to index of the voxel containing the point
    self.surface_index = ((surface_points - self.bound_min ) / self.v
_width).astype('i')

# Bind VoxelGrid class functions
VoxelGrid.remove_dist_padding = remove_dist_padding
VoxelGrid.compute_closest_surface_point = compute_closest_surface_poi
nt

# Call VoxelGrid class functions

# grid.remove_dist_padding()
# grid.compute_closest_surface_point()

print 'done'
```

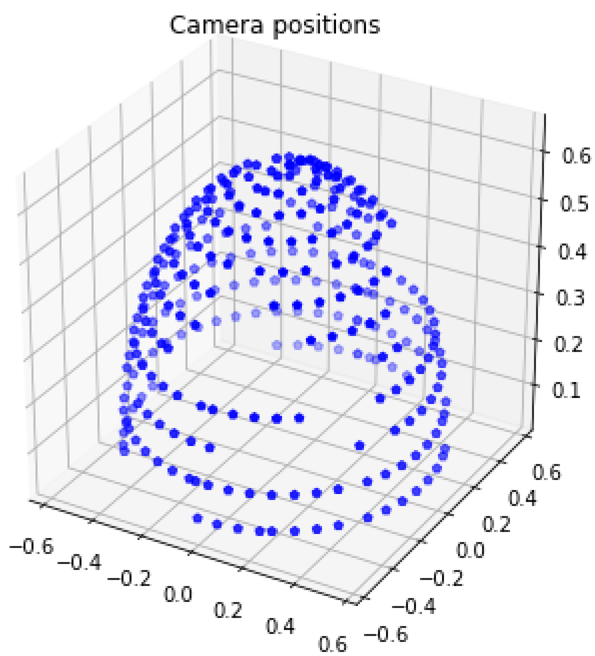
```
done
```

Camera positions

We can compute the position of a given camera center as $R^{-1}(-T)$, as seen in Assignment 2.

Note that even though we did not consider all cameras during the visual hull due to non-ideal segmentations in some images, we can still use all cameras for photoconsistency (and we will!).

In [12]:
```
def compute_camera_positions():
    cam_pos = []

    for par in params:
        R = np.reshape(par[10:19], (3,3)).astype('f')
        T = np.reshape(par[19:22], (3,1)).astype('f')

        cam_pos.append(np.linalg.inv(R).dot(-T))

    return cam_pos

cam_pos = np.array(compute_camera_positions())

fig = plt.figure(figsize = (6, 6))
ax = plt.subplot(111, projection='3d')

plt.title('Camera positions')
ax.scatter(cam_pos[:,0],cam_pos[:,2],cam_pos[:,1], c='b', marker='p')

plt.show()
```



Occlusion of surface points

While the gradient of the inner voxels points to towards the surface, the gradients of the surface voxels correspond to the normal of the surface. This means that we can compute whether a camera can see a point on the surface using two tests:

- If the dot product of the direction to the camera from the surface $\overrightarrow{d_{cs}}$ with the surface normal $\vec{n}$ is greater than 0 (i.e., $\overrightarrow{d_{cs}} \cdot \vec{n} > 0$), then the point is (potentially) visible.
- If the dot product was positive, cast a ray from the surface. If the ray intersects the object, the view is occluded.

Given the setup of the dataset, we are guaranteed that the object is not partially outside of the image. This means that only an occlusion test to determine is necessary (i.e., no clipping/culling is required).

For the ray casting, we will use a fast grid traversal algorithm by Amanatides and Woo:

```
Amanatides, John, and Andrew Woo. "A fast voxel traversal algorithm for ray
 tracing." Eurographics. Vol. 87. No. 3. 1987.
```

In [13]:
```
# Computes visibility for all object voxels

def compute_grid_occlusion(self, camera):
    grid_is_visible = np.full(self.shape, False, dtype='uint8')

    is_visible, path = self.occlusion_test(camera.flatten())
    grid_is_visible[self.obj_mask] = is_visible

    return (grid_is_visible, path)


# The framework of the occlusion test

def occlusion_test(self, camera):
    # Test if dot product is less than 0 for visibility
    is_visible = self.angle_check(camera)

    visible_points = self.surface_points[is_visible]
    visible_pos = self.surface_world[is_visible]

    paths = []
    hits = []

    vis_ind = np.where(is_visible)[0]

    # If a voxel passes the dot product text, cast a ray
    for index, voxel, pos in zip(vis_ind, visible_points, visible_pos
):
        direction = camera - pos
        distance = np.linalg.norm(direction)

        ray =  direction / distance

        hit, path = self.cast_ray(ray, voxel)
        is_visible[index] = not hit

        paths.append(path)
        hits.append(hit)
        #if hit:
            #print 'camera: ' + (str)(camera)
            #print 'pos: ' + (str)(pos)
            #print 'direction: ' + (str)(ray)
            #return (is_visible, path)

    return (is_visible, (paths, hits))


# Take the dot product of all surface points with the ray from the po
int to the camera.
# Visible points have value greater than or equal to zero

def angle_check(self, camera):
    sw = self.surface_world
    sp = self.surface_points
```

```
    return np.sum((camera - sw) * (self.gradient[sp[:,0], sp[:,1], sp
[:,2]]), axis=1) > 0
```

In [14]:
```
# Check if the ray is occluded by the object using raytracing. This i
s the algorithm by Amanatides and Woo.

def cast_ray(self, direct, voxel):
    delta_x = abs(self.v_width / direct[0])
    delta_y = abs(self.v_width / direct[1])
    delta_z = abs(self.v_width / direct[2])

    max_x = delta_x / 2.0
    max_y = delta_y / 2.0
    max_z = delta_z / 2.0

    i, j, k = voxel

    step_x, step_y, step_z = (direct / np.abs(direct)).astype('i')

    # This counter allows for a margin of error when computing estima
ted surface points
    counter = np.zeros((3), dtype='uint8')

    path = []

    while True:
        path.append(self.centers[i, j, k, :, 0])

        if max_x < max_y:
            if max_x < max_z:
                i += step_x
                counter[0] += 1

                if i < 0 or i >= self.shape[0]:
                    return (False, path)

                max_x = max_x + delta_x
            else:
                k += step_z
                counter[2] += 1

                if k < 0 or k >= self.shape[2]:
                    return (False, path)

                max_z = max_z + delta_z
        else:
            if max_y < max_z:
                j += step_y
                counter[1] += 1

                if j < 0 or j >= self.shape[1]:
                    return (False, path)

                max_y = max_y + delta_y
            else:
                k += step_z
                counter[2] += 1

                if k < 0 or k >= self.shape[2]:
```

```
                    return (False, path)

                max_z = max_z + delta_z

        if self.obj_mask[i,j,k] and np.max(counter) > 1:
            return (True, path)

    print "ERROR: code should not reach this point"
```

In [15]:
```
# Declare array containing the index of all surface points and the wo
rld coordinates of those points

def init_surface_points(self):
    self.surface_points = self.surface_index[self.obj_mask]

    sp = self.surface_points
    self.surface_world = self.centers[sp[:,0], sp[:,1], sp[:,2], :3,
0]
```

In [16]:
```
# Bind VoxelGrid class functions

VoxelGrid.init_surface_points = init_surface_points
VoxelGrid.angle_check = angle_check
VoxelGrid.occlusion_test = occlusion_test
VoxelGrid.cast_ray = cast_ray
VoxelGrid.compute_grid_occlusion = compute_grid_occlusion
```

In [17]:
```
# Call VoxelGrid class functions

# grid.init_surface_points()

print 'done'
```

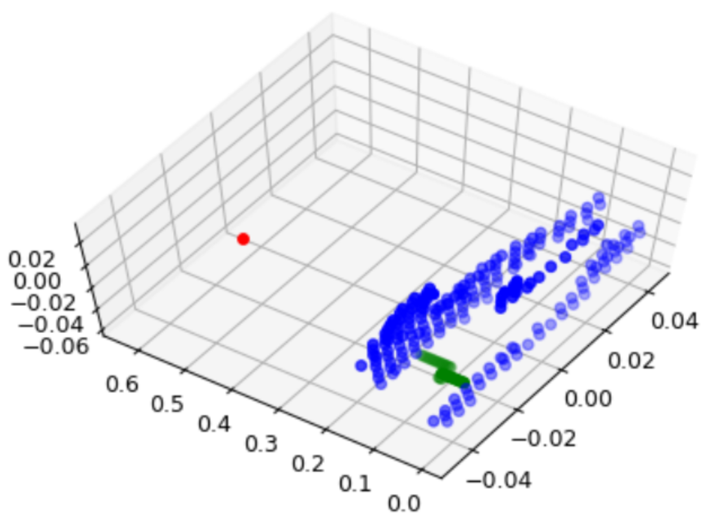
done

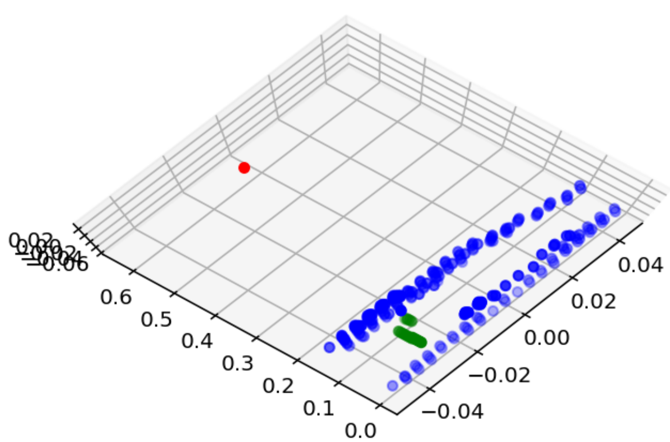
Occlusion results

Although I don't call the occlusion code until the next section, it seemed that this was an appropriate place to put some intermediate results. They are not really comprehensive. They are just sanity checks because I had some bugs and needed visual output to fix them. Due to this fact, I decided to use a low resolution for these plots so that I could iterate quickly.

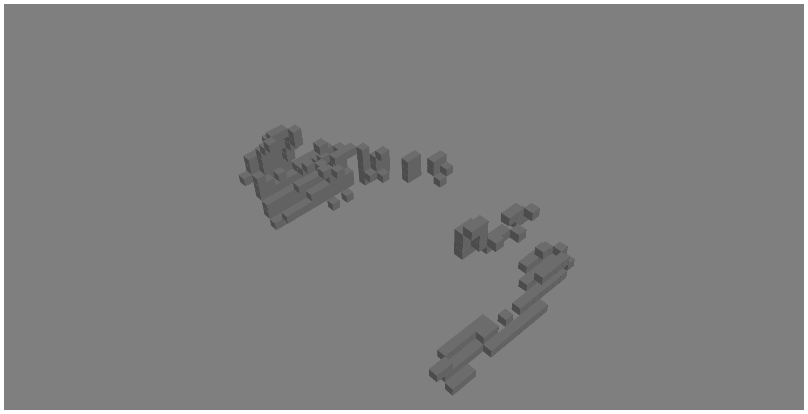
We are going to consider one of the datasets where the camera is above the structure. For the two figures below, the camera is red; the points visible by the camera are blue and the raytracing path is green.

In both cases, the path encounters an object voxel and terminates, so the source voxel is not blue (i.e., not visible).





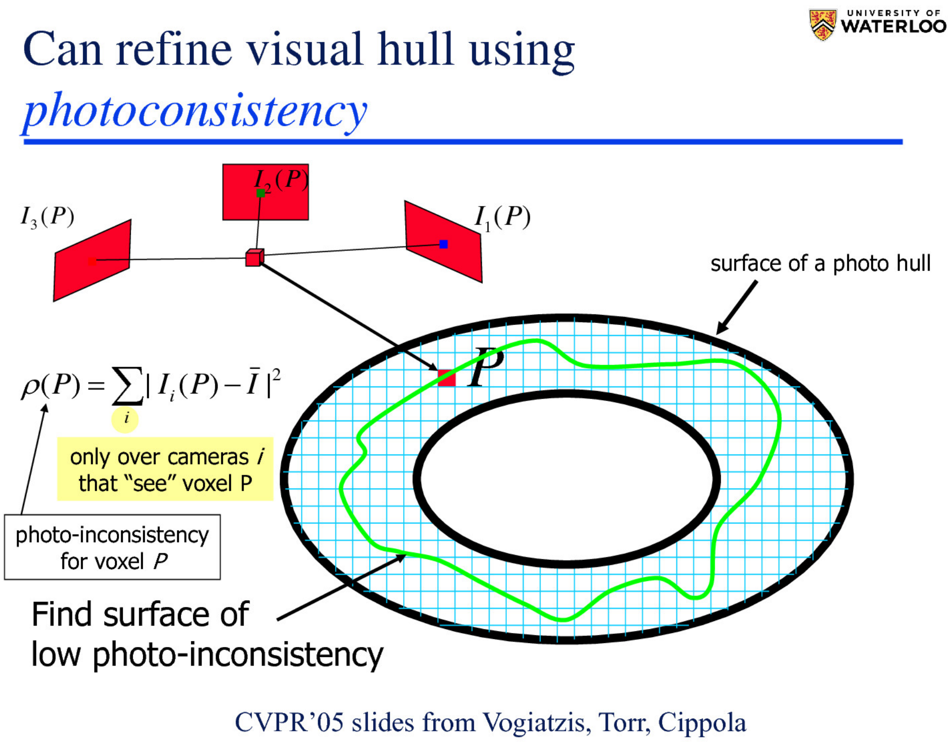
Here, we see the 3D result for the same camera. I attempted to orient the model in the same way as the first of the two images above.



As we can see, the pillars and most of the base are occluded by the top of the temple. This is the expected behaviour.

Photoconsistency calculation

Next, we need to compute the photoconsistency for each of the voxels. This is illustrated in the following course lecture slide:



In [18]:
```
# %matplotlib notebook

def compute_photoconsistency(self, cameras, params):
    self.ph_con = np.full(self.shape, 0, dtype='f')

    # Store visibility of surface points for all cameras
    print "Computing occlusions"
    vis_arr = []
    j = 0

    for i, cam in enumerate(cameras):
        is_visible, path = self.compute_grid_occlusion(cam)

        vis_arr.append(np.where(is_visible))

        j += print_progress(i, len(cameras), j)

        # The below code was used for the occlusion results plotting

        #indices = np.where(is_visible == 1)
        #points = self.centers[indices[0], indices[1], indices[2], :
3, 0]

        #if i == 218:
        #    xx, yy, zz = np.where(is_visible == 1)
        #    mayavi.mlab.points3d(xx,yy,zz,
        #                 mode='cube',
        #                 color=(0.5,0.5,0.5),
        #                 scale_factor=1)
        #    mayavi.mlab.show()

        # return (cam, points, path)

    # Compute intensity for each (voxel,camera) pair
    print "\nComputing intensities"
    intensities = []
    j = 0

    for i, cam, par, vis in zip(range(len(cameras)), cameras, params,
 vis_arr):
        img = plt.imread(img_dir + par[0])
        pixel_coords = self.compute_pixel_projections(par)[vis[0], vi
s[1], vis[2]]
        intensities.append(img[pixel_coords[:,0], pixel_coords[:,1]])

        j += print_progress(i, len(cameras), j)

    # Compute the mean intensity for each voxel
    intensity_sum = np.zeros(np.append(self.shape, 3), dtype='f')

    for vis, intensity in zip(vis_arr, intensities):
        intensity_sum[vis[0], vis[1], vis[2]] += intensity

    mean_intensity = intensity_sum / len(cameras)

    self.photo_incon = np.zeros(self.shape, dtype='f')
```

```
    # Compute photoinconsistency sum
    for vis, intensity in zip(vis_arr, intensities):
        self.photo_incon[vis[0], vis[1], vis[2]] += \
                np.linalg.norm(intensity - mean_intensity[vis[0], vis
[1], vis[2]], axis=1)**2
```

In [19]:
```
# Bind VoxelGrid class functions

VoxelGrid.compute_photoconsistency = compute_photoconsistency
```

In [20]:
```
# Call VoxelGrid class functions

# grid.compute_photoconsistency(cam_pos, params)

print 'done'
```

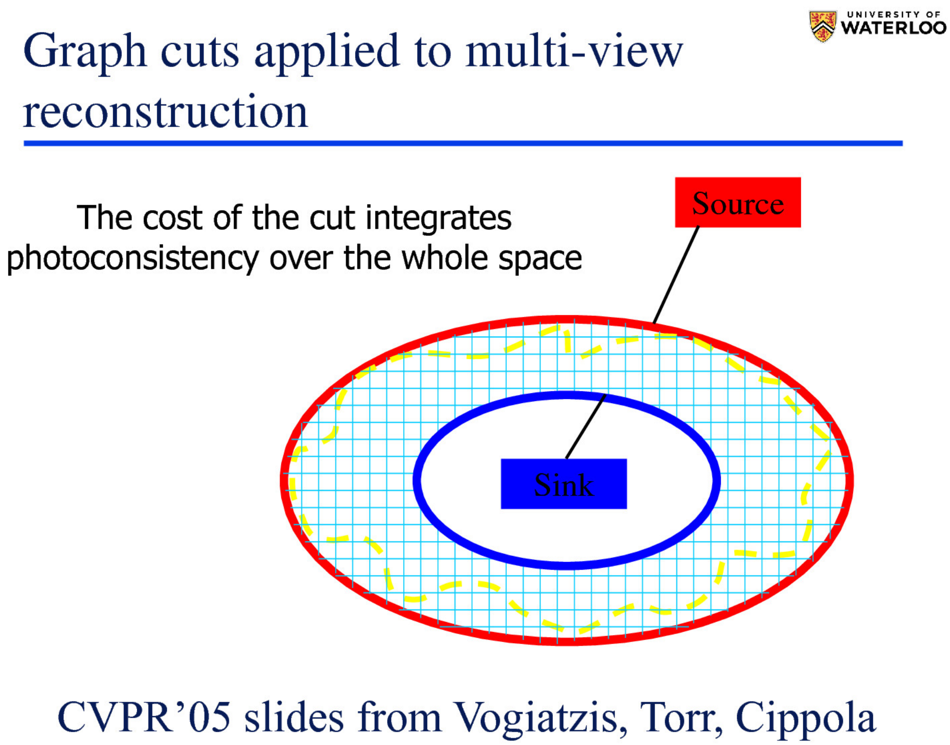
```
done
```

In [21]:
```
# Save important data since the raytracer is slow

save = False

if save:
    np.save('voxel_data/grid_shape', grid.shape)
    np.save('voxel_data/grid_dist', grid.dist)
    np.save('voxel_data/grid_photo_incon', grid.photo_incon)
    np.save('voxel_data/grid_v_width', grid.v_width)
```

In [22]:
```
# Code below was used to generate occlusion plots.

# cam, points, paths = grid.compute_photoconsistency(cam_pos, params)

# hits = paths[1]
# paths = paths[0]

def plot_occlusion_path():
    ind = 120

    path = np.array(paths[ind])

    fig = plt.figure()
    ax = plt.subplot(111,projection='3d')

    ax.scatter(cam[0], cam[1], cam[2], c='r')
    ax.scatter(path[:,0], path[:,1], path[:,2], c='g')
    ax.scatter(points[:,0], points[:,1], points[:,2], c='b')

    plt.show()

def plot_visibility_3d():
    xx, yy, zz = np.where(is_visible == 1)
    mayavi.mlab.points3d(xx,yy,zz,
                 mode='cube',
                 color=(0.5,0.5,0.5),
                 scale_factor=1)
    mayavi.mlab.show()
```

Graph cut

Finally, we need to perform the graph cut using the photo-inconsistencies as edge weights. This is illustrated in the following course lecture slide:



For the source edges, we will simply connect the source to all non-object voxels with (functionally) infinite weight. In order to decide which voxels to connect to the sink, we will use the signed distance function. A voxel will belong to the sink if it is at least 5 times the voxel width away from the surface, each edge having (functionally) infinite weight. For object nodes, incoming edges for any given voxel will have weight corresponding to its photoinconsistency sum. Note that the directed edges are not symmetric according to this description.

I have decided to square the photoconsistency sum before assigning edge weights (as suggested by the project specifications). I found that using un-squared values ensured that the graph would only classify voxels connected to the sink as object voxels (i.e., it was minimizing the allowable surface area). However, this was resolved by using the squared values. I also tried exponential values, but this gave the same results as quadratic for this dataset.

In [23]:
```
# Load grid data so we don't need to perform raytracing again

load = True

if load == True:
    grid_shape = np.load('voxel_data/grid_shape.npy')
    grid_dist = np.load('voxel_data/grid_dist.npy')
    grid_photo_incon = np.load('voxel_data/grid_photo_incon.npy')
    grid_v_width = np.load('voxel_data/grid_v_width.npy')

else:
    grid_shape = grid.shape
    grid_dist = grid.dist
    grid_photo_incon = grid.photo_incon
    grid_v_width = grid.v_width
```

In [24]:
```
class ReconstructionGraphCuts:

    def __init__(self):
        self.g = maxflow.GraphInt()

        self.add_nodes()

        self.max_weight = 2**8-1
        self.func_inf = self.max_weight * np.prod(grid_shape) + 1

        self.photo_incon = grid_photo_incon**2

        self.lower_bound = min(0, np.min(grid_photo_incon))
        self.upper_bound = np.max(grid_photo_incon)

        self.add_node_edges()
        self.add_st_edges()

    def add_nodes(self):
        s = grid_shape
        self.nodeids = self.g.add_grid_nodes((s[0], s[1], s[2]))

    # Link weights are regularized to be in the range [0,max_weight].
 This ensures that edge weights
    # are small integers in order to take advantage of optimizations
 in the maxflow library
    def regularize(self, weights):
        reg_weights = np.empty(weights.shape, dtype='i')
        reg_weights[:,:,:] = (self.max_weight) * \
                                (weights - self.lower_bound)/abs(self
.upper_bound - self.lower_bound)

        return reg_weights

    def add_right_edges(self):
        structure = np.zeros((3,3,3), dtype='i')
        structure[2,1,1] = 1

        # Set edge weights to photoinconsistency
        weights = np.zeros(grid_shape, dtype='f')
        weights[:-1,:,:] = self.photo_incon[1:,:,:]

        background_voxels = np.logical_not(grid_photo_incon > 0)

        # Regularize weights to small integers
        weights = self.regularize(weights)

        # Set links to non-object voxels as infinite
        (weights[:-1,:,:])[background_voxels[1:,:,:]] = self.func_inf

        self.g.add_grid_edges(self.nodeids, weights=weights, structur
e=structure, symmetric=False)

    def add_left_edges(self):
        structure = np.zeros((3,3,3), dtype='i')
        structure[0,1,1] = 1
```

```
        # Set edge weights to photoinconsistency
        weights = np.zeros(grid_shape, dtype='f')
        weights[1:,:,:] = self.photo_incon[:-1,:,:]

        background_voxels = np.logical_not(grid_photo_incon > 0)

        # Regularize weights to small integers
        weights = self.regularize(weights)

        # Set links to non-object voxels as infinite
        (weights[1:,:,:])[background_voxels[:-1,:,:]] = self.func_inf

        self.g.add_grid_edges(self.nodeids, weights=weights, structur
e=structure, symmetric=False)

    def add_up_edges(self):
        structure = np.zeros((3,3,3), dtype='i')
        structure[1,2,1] = 1

        # Set edge weights to photoinconsistency
        weights = np.zeros(grid_shape, dtype='f')
        weights[:,:-1,:] = self.photo_incon[:,1:,:]

        background_voxels = np.logical_not(grid_photo_incon > 0)

        weights = self.regularize(weights)

        # Set links to non-object voxels as infinite
        (weights[:,:-1,:])[background_voxels[:,1:,:]] = self.func_inf

        self.g.add_grid_edges(self.nodeids, weights=weights, structur
e=structure, symmetric=False)

    def add_down_edges(self):
        structure = np.zeros((3,3,3), dtype='i')
        structure[1,0,1] = 1

        # Set edge weights to photoinconsistency
        weights = np.zeros(grid_shape, dtype='f')
        weights[:,1:,:] = self.photo_incon[:,:-1,:]

        background_voxels = np.logical_not(grid_photo_incon > 0)

        weights = self.regularize(weights)

        # Set links to non-object voxels as infinite
        (weights[:,1:,:])[background_voxels[:,:-1,:]] = self.func_inf

        self.g.add_grid_edges(self.nodeids, weights=weights, structur
e=structure, symmetric=False)

    def add_infront_edges(self):
        structure = np.zeros((3,3,3), dtype='i')
        structure[1,1,2] = 1

        # Set edge weights to photoinconsistency
```

```
        weights = np.zeros(grid_shape, dtype='f')
        weights[:,:,:-1] = self.photo_incon[:,:,1:]

        weights = self.regularize(weights)

        background_voxels = np.logical_not(grid_photo_incon > 0)

        # Set links to non-object voxels as infinite
        (weights[:,:,:-1])[background_voxels[:,:,1:]] = self.func_inf

        self.g.add_grid_edges(self.nodeids, weights=weights, structur
e=structure, symmetric=False)

    def add_behind_edges(self):
        structure = np.zeros((3,3,3), dtype='i')
        structure[1,1,0] = 1

        # Set edge weights to photoinconsistency
        weights = np.zeros(grid_shape, dtype='f')
        weights[:,:,1:] = self.photo_incon[:,:,:-1]

        weights = self.regularize(weights)

        background_voxels = np.logical_not(grid_photo_incon > 0)

        # Set links to non-object voxels as infinite
        (weights[:,:,1:])[background_voxels[:,:,:-1]] = self.func_inf

        self.g.add_grid_edges(self.nodeids, weights=weights, structur
e=structure, symmetric=False)

    def add_node_edges(self):
        self.add_right_edges()
        self.add_left_edges()
        self.add_up_edges()
        self.add_down_edges()
        self.add_behind_edges()
        self.add_infront_edges()

    def add_st_edges(self):
        source_weights = np.zeros(grid_shape, dtype='f')
        sink_weights = np.zeros(grid_shape, dtype='f')

        sdf = grid_dist

        # Compute source weights
        boundary_voxels = (sdf == grid_v_width)
        source_weights[boundary_voxels] = self.func_inf

        # Compute sink weights
        sink_voxels = sdf < grid_v_width * -5
        sink_weights[sink_voxels] = self.func_inf

        # Add terminal edges
        self.g.add_grid_tedges(self.nodeids, source_weights, sink_wei
ghts)
```

```
    def run(self):
        self.g.maxflow()

    def get_segments(self):
        return self.g.get_grid_segments(self.nodeids)
```

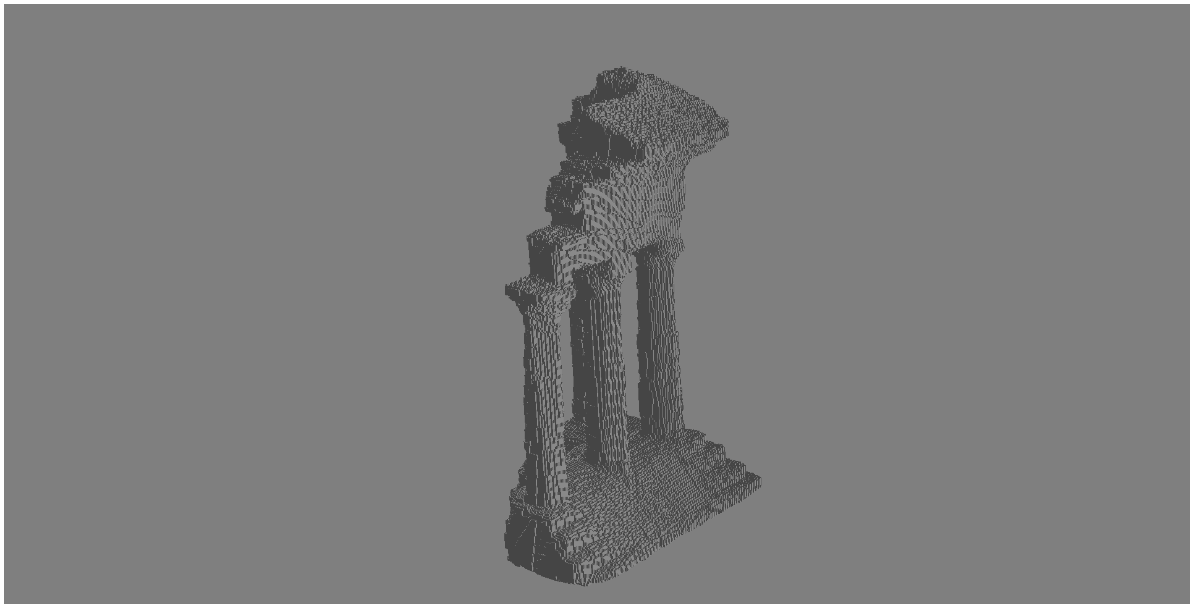
In [25]:
```
print 'start'

# Run graph cut

# rgc = ReconstructionGraphCuts()
# rgc.run()

print 'done'
```

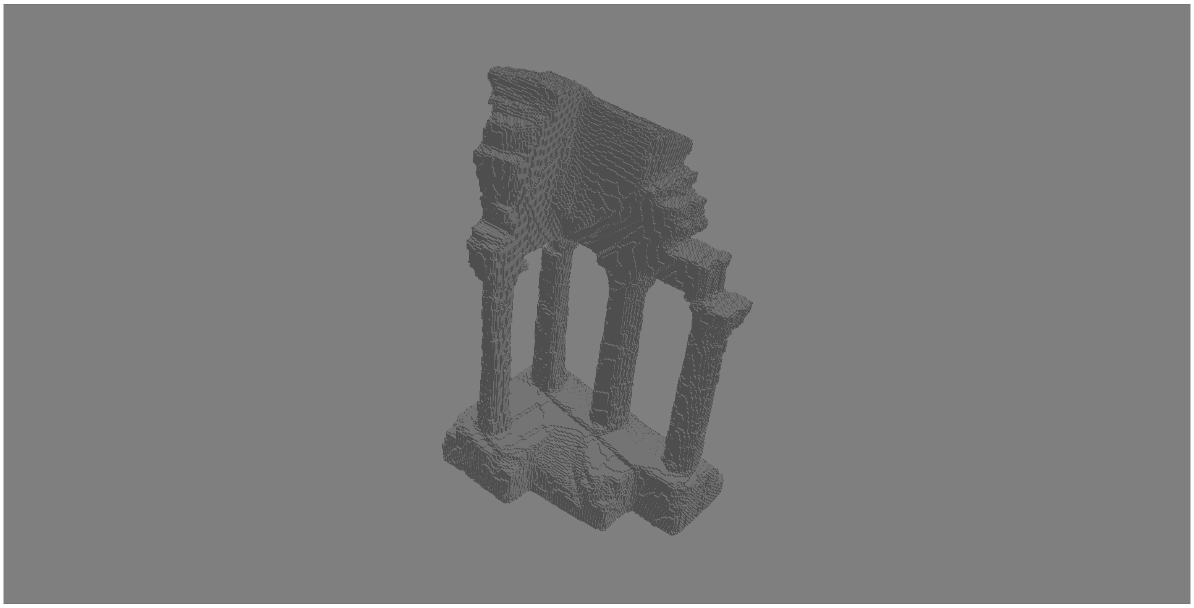
```
start
done
```

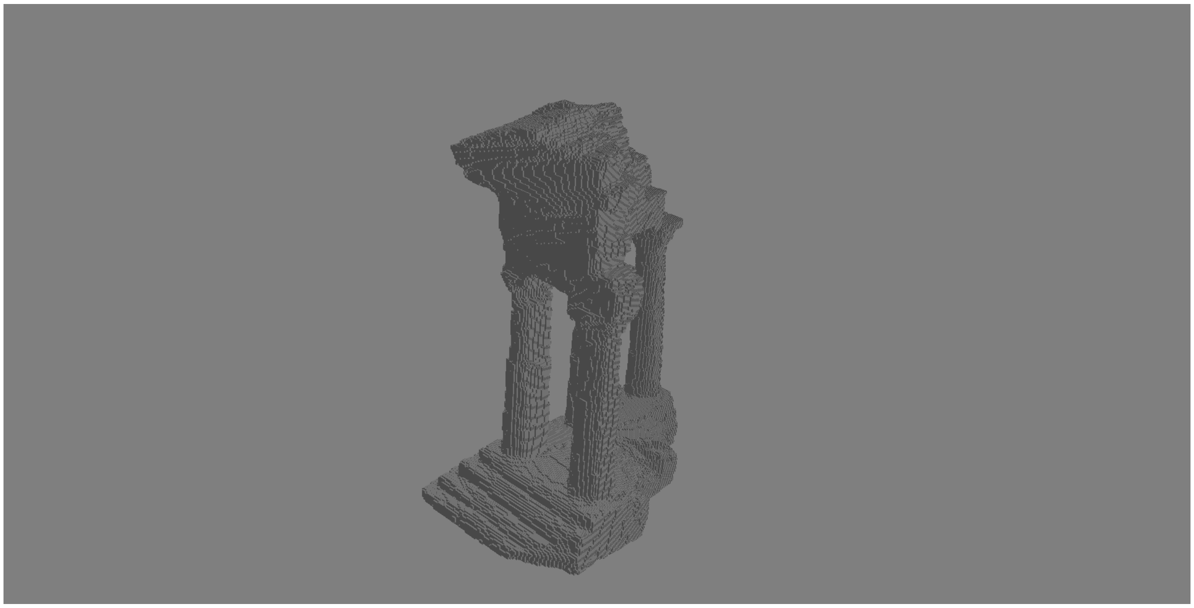
Results

Let's see the reconstructed surface using our photoconsistency graph cut.

In [26]:
```
def plot_reconstruction():
    xx, yy, zz = np.where(rgc.get_segments() == 1)

    mayavi.mlab.points3d(xx,yy,zz,
                 mode='cube',
                 color=(0.5,0.5,0.5),
                 scale_factor=1)

    mayavi.mlab.show()

# plot_reconstruction()
```



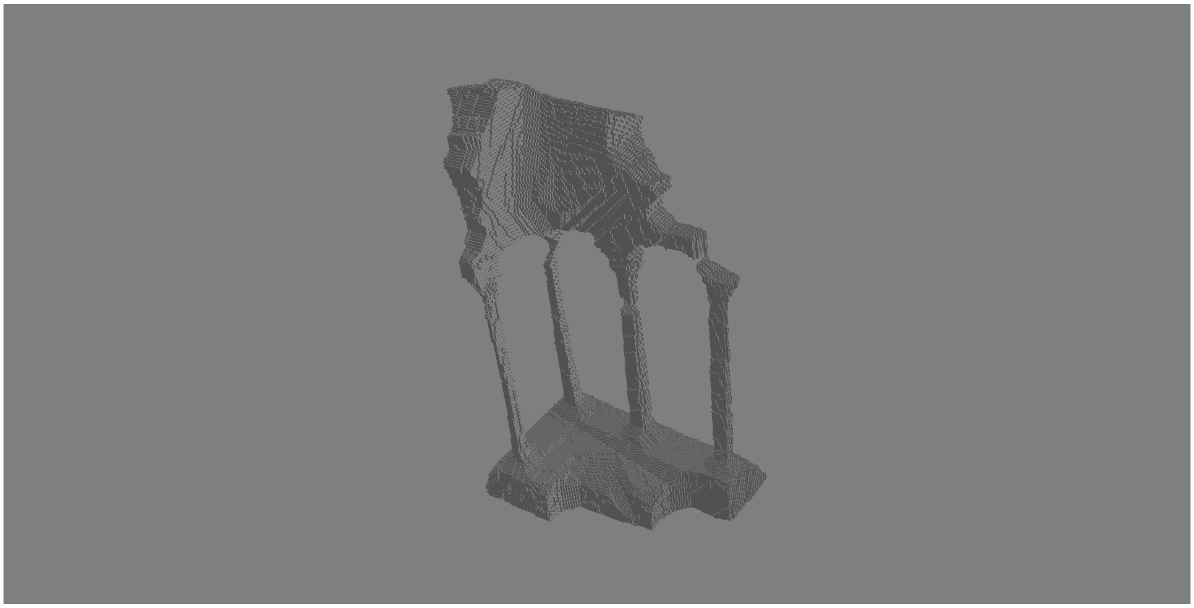


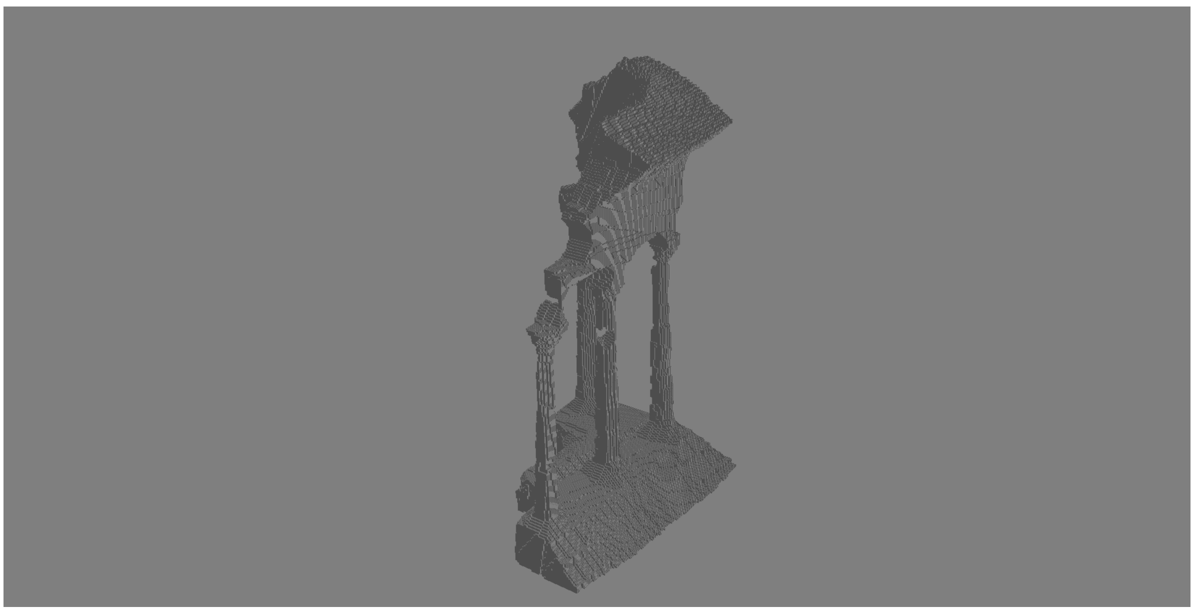


I am not sure whether to blame the dataset or the resolution, but it is honestly hard to tell whether this is the "correct" surface reconstruction. In my opinion the roof of the temple looks more crisp (less blob-like) than the surface provided by the visual hull. However, this is subjective, so let's try to systematically demonstrate that the algorithm is working.

Let's start by rendering just the voxels connected to the sink.

In [27]:
```
def plot_sink():
    xx, yy, zz = np.where(grid_dist < grid_v_width * -5)

    mayavi.mlab.points3d(xx,yy,zz,
                mode='cube',
                color=(0.5,0.5,0.5),
                scale_factor=1)

    mayavi.mlab.show()

# plot_sink()
```



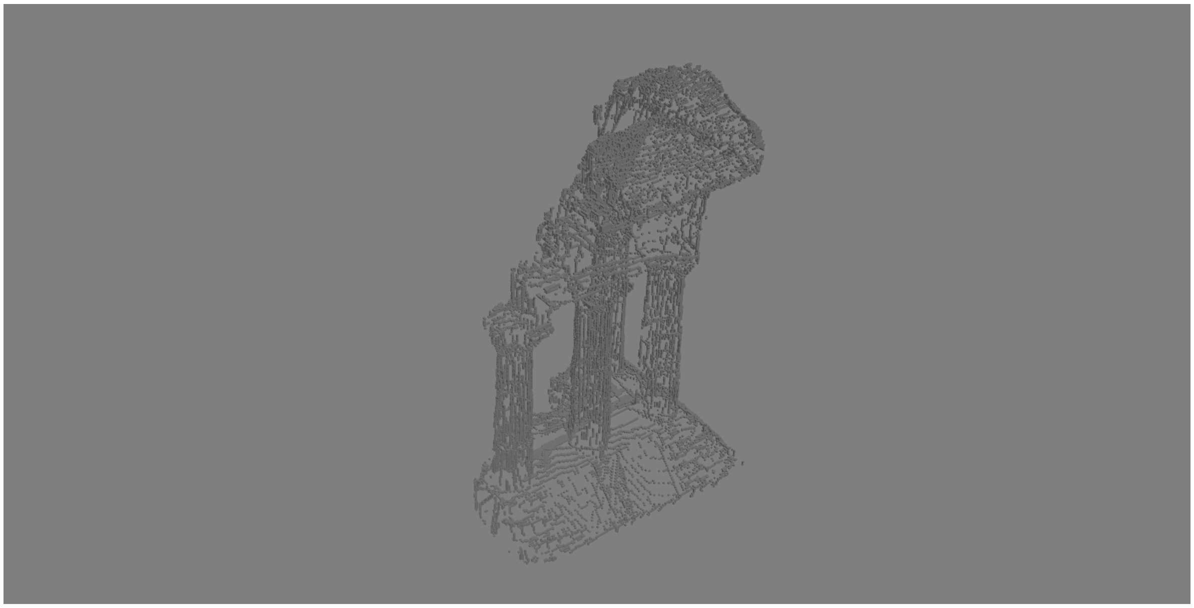


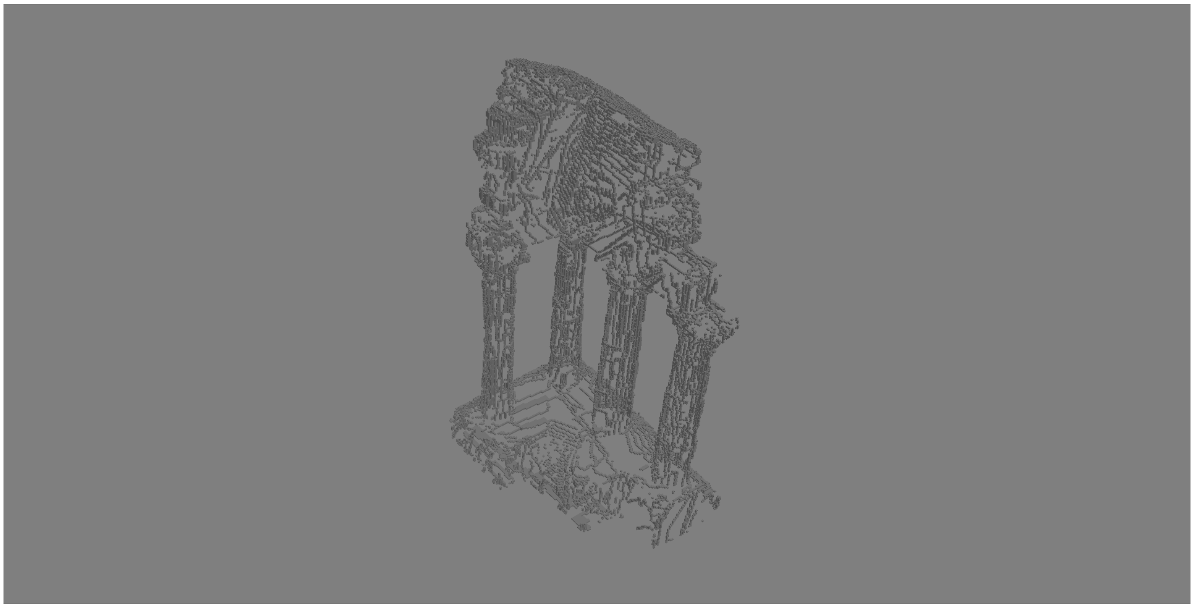
As we can see, the sink is simply connected to the deeply embedded voxels. The columns of the temple are substantially narrower than those belonging to the full object (i.e., compared to images presented as results to the visual hull).

Most importantly, we can see that the reconstructed surface is not equivalent to the surface given solely by the voxels connected to the sink.

Now, if we can also show that our final result is not equivalent to that provided by the visual hull, then we know that the graph cut is doing its job and selectively removing voxels from the 3D model. So, let us see what happens if we plot the voxels that belong to the visual hull but do not belong to the surface produced by the graph cut.

In [28]:
```
def plot_removed_voxels():
    xx, yy, zz = np.where(np.logical_and(grid_dist <= 0, np.logical_not(rgc.get_segments() == 1)))

    mayavi.mlab.points3d(xx,yy,zz,
                 mode='cube',
                 color=(0.5,0.5,0.5),
                 scale_factor=1)

    mayavi.mlab.show()

# plot_removed_voxels()
```





As we can see, the algorithm is clearly removing voxels from the visual hull to provide the final surface reconstruction. This confirms that our algorithm is indeed working and it is selectively removing voxels with poor photoconsistency. So, we have succeeded!

Lessons from the project

Here, I just wanted to make some notes about some shortcomings of the methodology used in this project in case I decide to pursue something along these lines in the future.

When performing the segmentation for the visual hull, it is important to have conservative segmentation (assuming that you plan to use a photoconsistency graph cut afterwards). This is because the photoconsistency approach, as it has been described and implemented, cannot restore voxels after they have been culled by the visual hull. So if the segmentation algorithm incorrectly classifies an object pixel as a background pixel, there will be a hole in the final model.

When testing the photoconsistency graph cut, it is good if the visual hull has performed poorly and a lot of the features are missing. This is of course resolution dependent (i.e., the visual hull needs to perform even worse for lower resolutions). The reason for this is because it is difficult to tell if the photoconsistency graph cut has improved the model if the visual hull has already provided a surface that is indistinguishable from correct.

The final lesson is to NEVER implement a raytracer in python!!! Even at the final resolutions that I used in this report, the raytracing took 3 hours. I tried to double the resolution and run it overnight. It completed after 10 hours, but I did not save the data correctly so I was unfortunately stuck with the lower resolution images.

References

The approach used for each step of this project was my own (following the course slides). I did not reference any research paper or textbook describing how the volumetric surface reconstruction should be implemented. The resources that I used were mainly simple algorithms for 3D grids that I couldn't remember the details of.

Here are my references:

- Dataset by Steven Seitz *et al.* (http://grail.cs.washington.edu/projects/mview/)
- Bridson, Robert. *Fluid simulation for computer graphics*. AK Peters/CRC Press, 2015.
- Amanatides, John, and Andrew Woo. "A fast voxel traversal algorithm for ray tracing." *Eurographics*. Vol. 87. No. 3. 1987.
- Lecture 9 of the course slides