Lab 6
Three-Dimensional Image Processing

Jonathan Ananda Nusantara
jan265

Three-Dimensional Filtering:

In the first section, we were exploring how 3D image filtering works. In the beginning, the script "t1make" was ran, which outputs a sequence of image slices t1.vx and its 3D representation t1.vd. The content of the script is as below:

```
bbx=0,20,0,20,0,20
vgen3d e=12,12,12 bb=$bbx tr=10,10,10 of=t1-tmp1
vgen3d p=3,3,15 bb=$bbx tr=10,10,10 of=t1-tmp2
vop -or t1-tmp1 t1-tmp2 | vdim -c -o t1.vx
v3pol -t t1.vx -o t1.vd
rm t1-tmp1 t1-tmp2
```

The command "vgen3d" was executed twice to generate two 3D objects. The first is a 3D arbitrary ellipsoid (using the input e), with dimension 20x20x20 and translated by 10 pixels in each axis to set it in the middle. The other 3D image created is an elliptic cylinder with similar parameters. These two 3D shapes are then passed to the "vop" command, which uses the "or" operation on the two inputs, and to "vdim". This means that the output pixels will cover a combination of pixels from the two shapes, just like how an "or" operator works. The output is again passed to "vdim" in order to get the frame markers using the "-c" flag. The final output is a 3D image in the format of 2D image sequences "t1.vx". This output is then passed to "v3d" command, which is used to convert the 2D image set to 3D polygon.

The output "t1.vx" can be viewed in vview as an image sequence while "t1.vd" can be viewed using "v3d" command to have an animated and interactive 3D. For display purposes, "t1.vx" is passed to vtile to generate a tile image.Below are the two outputs:

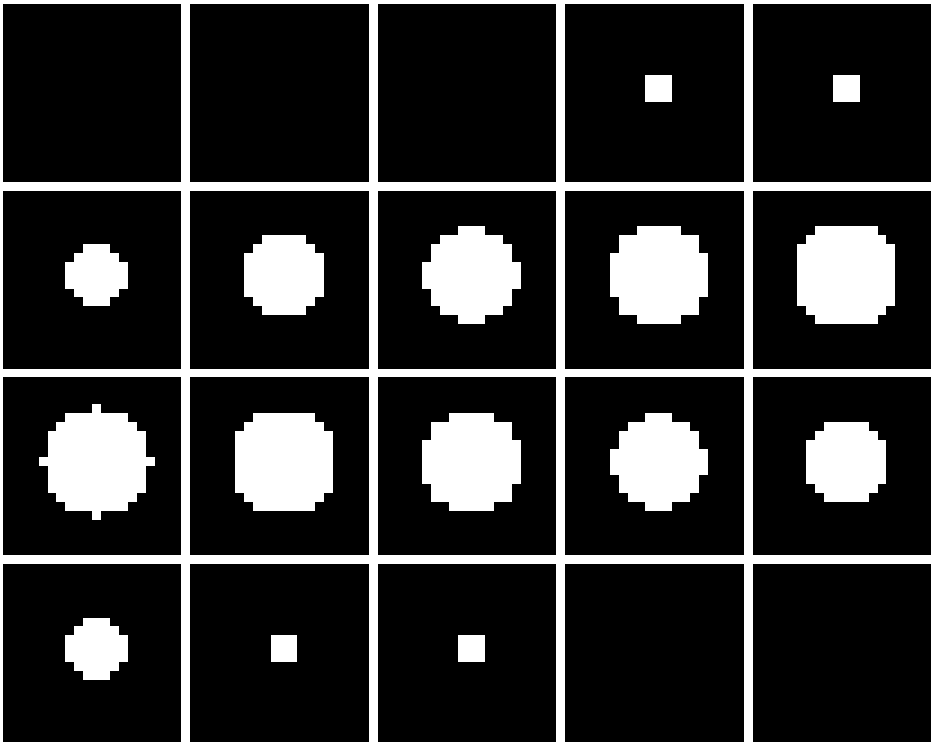

Figure 1 - Tile image of “t1.vx” image sequence.

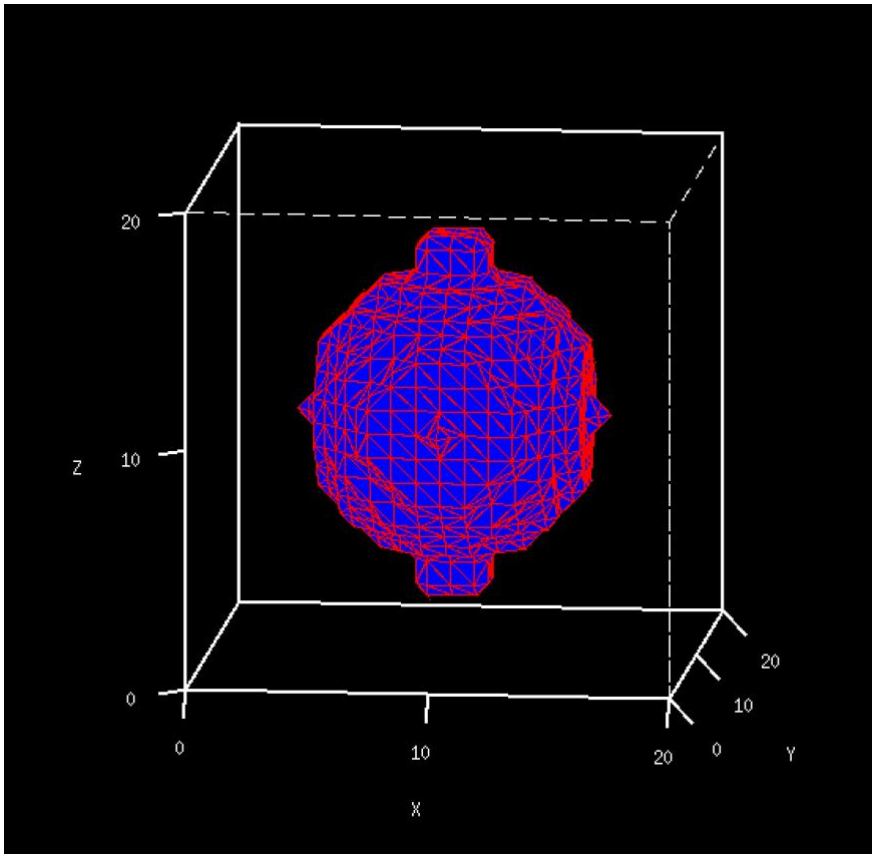

Figure 2 - “t1.vd” displayed using v3d.

From the two figures above, we can see that the object is very coarse. We then experimented on how to smooth the surface. The two commands below were run:

```
v3pfilt -a t1.vd -o t1f.vd
v3d t1f.vd cf=v3dcf &
```

The command “v3pfilt” is a 3D surface smoothing command, which is taking “t1.vd” as input and output “t1f.vd”. This output is then again displayed with the “v3d” command which now uses the input options/settings file “v3dcf”. The displayed 3D output is as below:

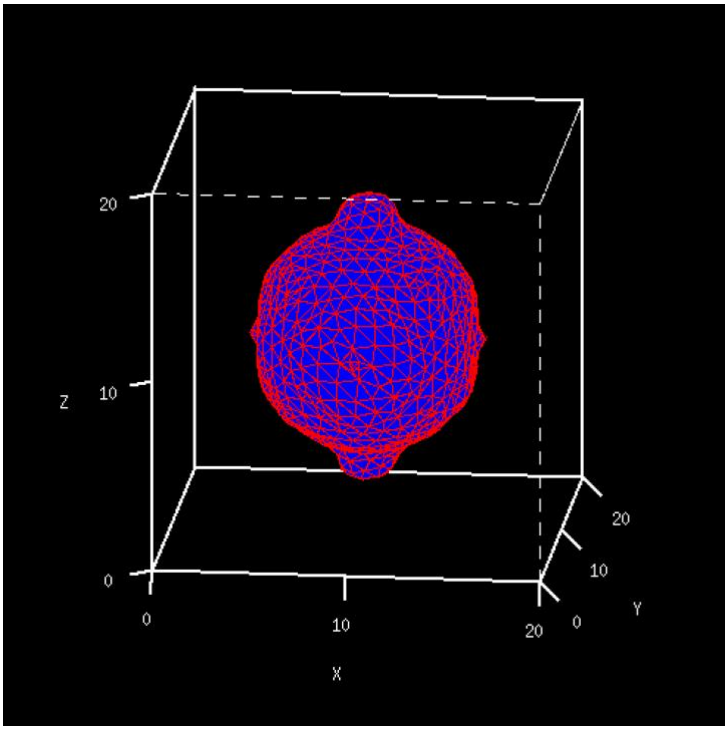

Figure 3 - “t1f.vd” displayed using v3d.

From the above figure, we can easily differentiate the smoothed output “t1f.vd” to “t1.vd”. In the original image, we mentioned that the surface is rough (with pointy edges). On the other hand, “t1f.vd” shown in Figure 3 has a very smooth surface. For example, if we take a closer look at the protrusions located at the top and bottom in Figure 3, we can see its smoothness. This is different when compared to the protrusions in Figure 2. Another noticeable difference is how the shape in Figure 3 is closer to being a perfect smooth circle, compared to the object in Figure 2. This proves how a surface smoothing will affect the object/image.

A smoothed 3D object would look nicer in humans’ eye. However, we would not want to always perform this action on polygons. This is because when we are smoothing, information is actually lost. As a result, one must decide on how much smoothing is to be performed on the object, and if we do not want to lose any information, then smoothing should not be done.

Next, the 3D morphological filtering was explored. The goal is to remove the protrusions that are located at the top and bottom of the object. The following commands were executed:

```
vmorph t1.vx -ed t=s s=3,3,3 | vdim -c  of=n3.vx
v3pol -t if=n3.vx of=n3.vd
```

The “vmorph” command is used to perform morphological filtering, the “-ed” flag meant morphological opening was chosen, “t=s” meant the type of filter is a spherical/ellipsoidal kernel, and parameter “s” is defined to have a 3x3x3 kernel. The output “n3.vd” is displayed using “v3d” as below:

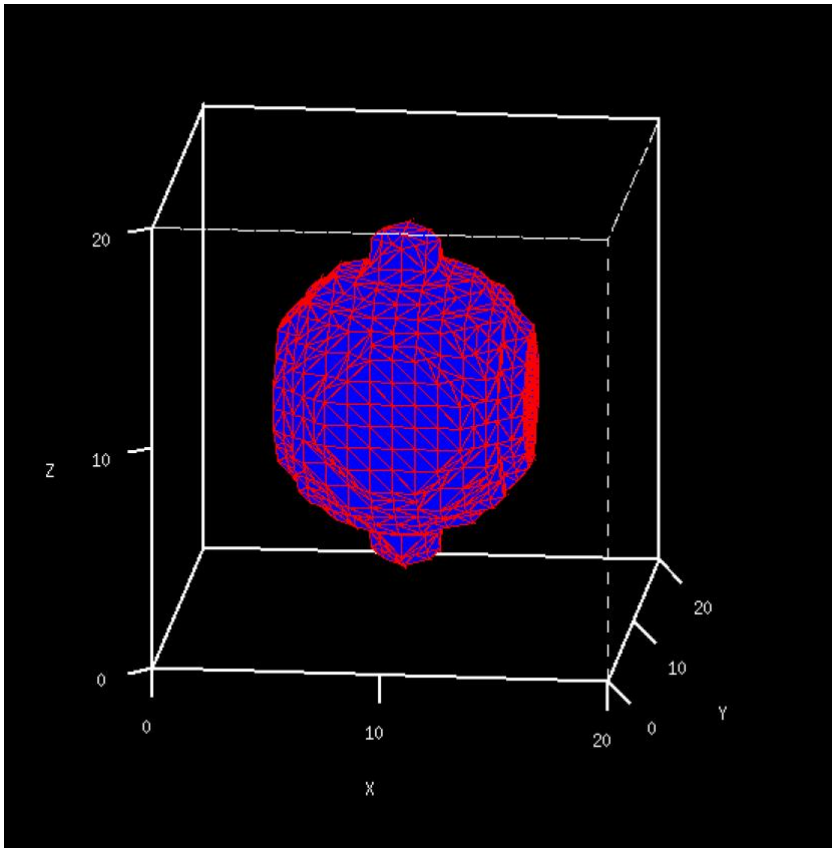

Figure 4 - “n3.vd” displayed using v3d.

In the above figure, we can still see the protrusions on the top and bottom of the object. However, we can see the small protrusions on the side of the object are now gone. This means that the kernel that was used is too small.

Additional experimentations were done in order to remove the protrusions using “vmorph”. The parameter “s” was incrementally increased during the experiment. At the end, it was found that when the kernel is of dimension 8x8x8, the top and bottom protrusions were able to be removed. The output is as below:

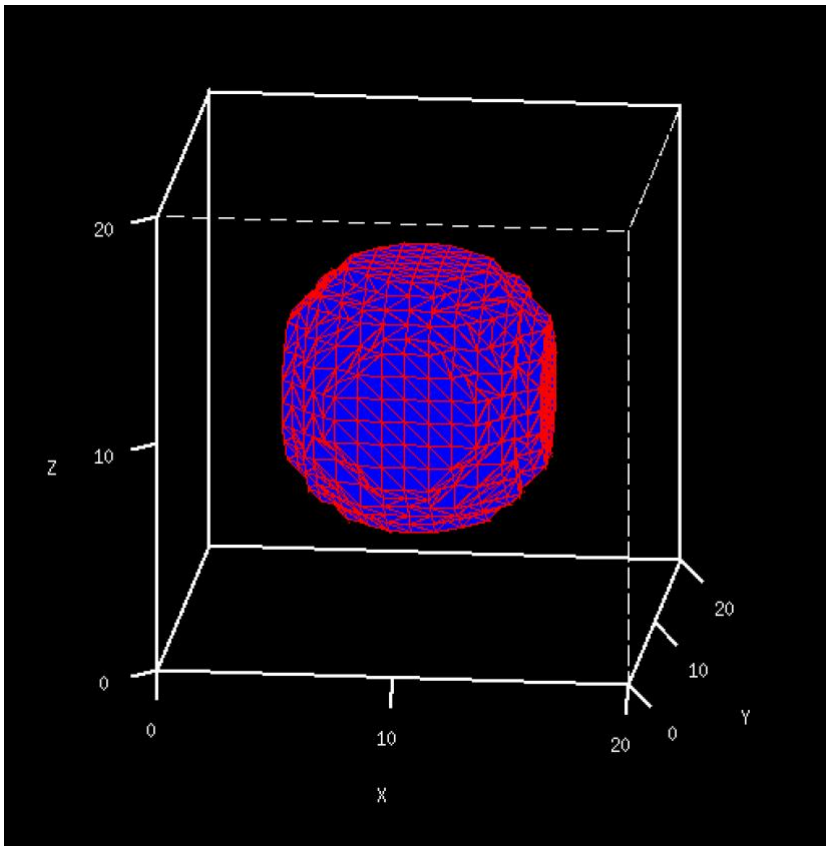

Figure 5 - Object “t1.vx” with removed protrusions using “vmorph”.

In Figure 5, we can see how the protrusions were able to be removed when using a bigger size kernel for the morphological filter. However, the problem with removing the protrusion with morphological filtering is that we cannot choose specific sections to process or filter.. As a result, other protrusions in the object that we may not want to remove will also be removed if the kernel size is enough to remove them (such as in the case of using kernel 8x8x8, where smaller protrusions were removed).

From this section, we have learned how to create 3D test images and how it can be displayed as either a sequence of 2D images or as an animated 3D display. The image is first used to test the effects of smoothing, which may lose some information when too much smoothing is performed. The image is also used to explore morphological smoothing and how it can be used to remove protrusions on objects.

Three-Dimensional Edge Detector:

In this section, the 3D edge detector algorithm was explored. Based on the v3dmeanpy program that was used to perform mean filtering on 3D image, a 3D edge operator program was created. The program is as below:


```
#!/usr/bin/env python
#### v3dedge a stand-alone python edge detector program for 3D

import sys
from numpy import *
from v4 import vx
import math

# Sobel separable matrix: h and h'
h = [1, 2, 1]
h_p = [1, 0, -1]

# Initialize all mask to be of 3x3x3 zero arrays
mask_x = [[[0,0,0], [0,0,0], [0,0,0]],
          [[0,0,0], [0,0,0], [0,0,0]],
          [[0,0,0], [0,0,0], [0,0,0]]]
mask_y = [[[0,0,0], [0,0,0], [0,0,0]],
          [[0,0,0], [0,0,0], [0,0,0]],
          [[0,0,0], [0,0,0], [0,0,0]]]
mask_z = [[[0,0,0], [0,0,0], [0,0,0]],
          [[0,0,0], [0,0,0], [0,0,0]],
          [[0,0,0], [0,0,0], [0,0,0]]]

# Create sobel kernel using loop
for i in range(3):
    for j in range(3):
        for k in range (3):
            mask_x[i][j][k] = h_p[i] * h[j] * h[k]
            mask_y[i][j][k] = h[i] * h_p[j] * h[k]
            mask_z[i][j][k] = h[i] * h[j] * h_p[k]
#print(mask_x)
#print(mask_y)
#print(mask_z)

# compute 3 x 3 x 3 edge detector
def v3dedge ( img ) :
 im = img.i
 tmimage = vx.Vx( img )
 tmimage.embedim((1,1,1,1,1,1))
 tm = tmimage.i

 magnitude = 0

 for z in range(im.shape[0]):
    for y in range(im.shape[1]):
        for x in range(im.shape[1]):
            sum_x = 0
            sum_y = 0
            sum_z = 0
            for zz in (0, 1, 2):
                for yy in (0, 1, 2):
                   for xx in (0, 1, 2):
                      sum_x += mask_x[zz][yy][xx] * tm[z + zz][y + yy][x+xx]
```

```
                        sum_y += mask_y[zz][yy][xx] * tm[z + zz][y + yy][x+xx]
                        sum_z += mask_z[zz][yy][xx] * tm[z + zz][y + yy][x+xx]
            sum_x = sum_x / 27
            sum_y = sum_y / 27
            sum_z = sum_z / 27
            magnitude = math.sqrt(sum_x*sum_x + sum_y*sum_y + sum_z*sum_z)
            # print(magnitude)
            if magnitude > 25:
                im[z,y,x] = 200
            else:
                im[z,y,x] = 0

## stand-alone wrapper
of=' '
vxif=' '
clist = vx.vxparse(sys.argv,  "if= of= -v  - ")
exec (clist )

if 'OPT' in locals():
   print ("v3dedge V4 3D python example program")
   print ("if= input file")
   print ("of= output file")
   print ("[-v] verbose mode")
   exit(0)
optv = 'OPTv' in locals()

ximage = vx.Vx( vxif )    ;#read image
v3dedge(ximage)           ;#process image
if optv:
   print (ximage.i)       ;# for very small images
ximage.write(of)          ;# Write the result file

```

There are several changes made to the program v3dmeanpy to make it into an edge detection program. First of all, it was chosen that the Sobel edge operator algorithm will be used. The sobel separable matrix is first defined as h and h'. We know that to calculate sobel kernel with respect to x, it will be equivalent to h'(x) * h(y) * h(z). These calculations are made for each index in the three sobel kernels for the different axis..

Next, based on the program in v3dmeanpy, we would perform convolution between the kernel and the region of image. Thus, we will calculate the sum of the calculation of each axis in each pixel, then normalize it over the number of pixels (to prevent pixel value going to above 255), and then find the magnitude. By definition , the magnitude is the square root of the squared sum of each axis. If the magnitude is above a threshold that was set as 25, the output pixel will have a high pixel value of 200, else it will have a pixel value of 0. From this calculation, the high valued pixel will be the edge of the original image.

In order to test the program, some test images were created based on the "t1.vx". The following commands were executed:

```
vpix t1.vx lo=100 hi=200 th=1 of=t1g.vx
vpix t1g.vx gn=25 of=t1n.vx
```

The first line above uses the command “vpix” to perform a threshold on the image “t1.vx” with a threshold value of 1. The “t1g.vx” would have a high pixel value of 200 and a low pixel value of 100. This means that the difference between high and low valued pixels are less intense. The second line took in this thresholded output and applied a gaussian noise to it, outputting image “t1n.vx”. The output images are as below:

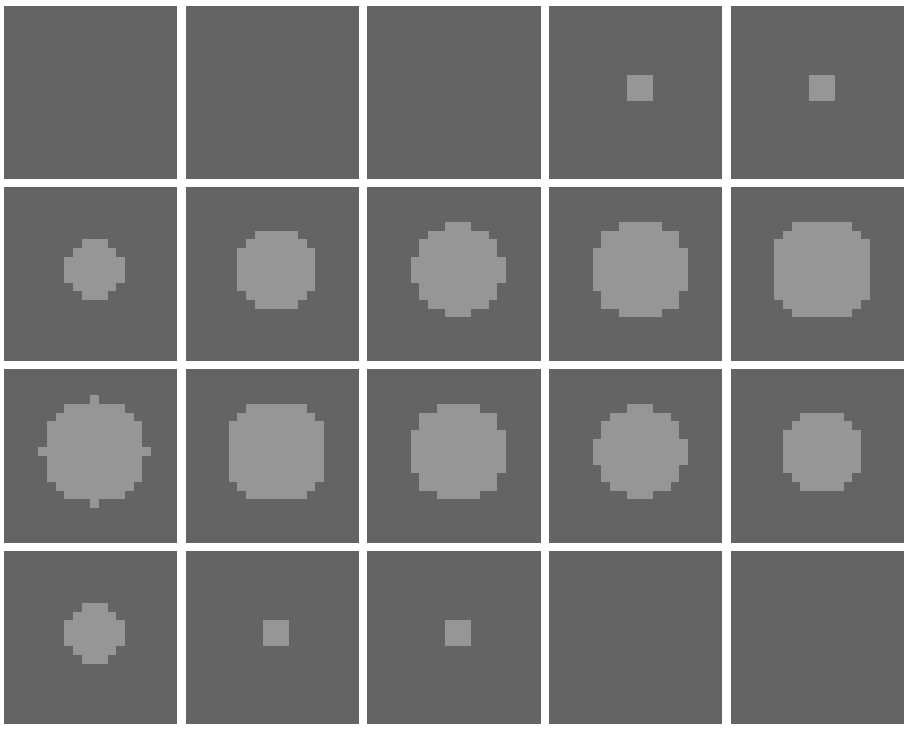

Figure 6 - Tile image of “t1g.vx” image sequence.

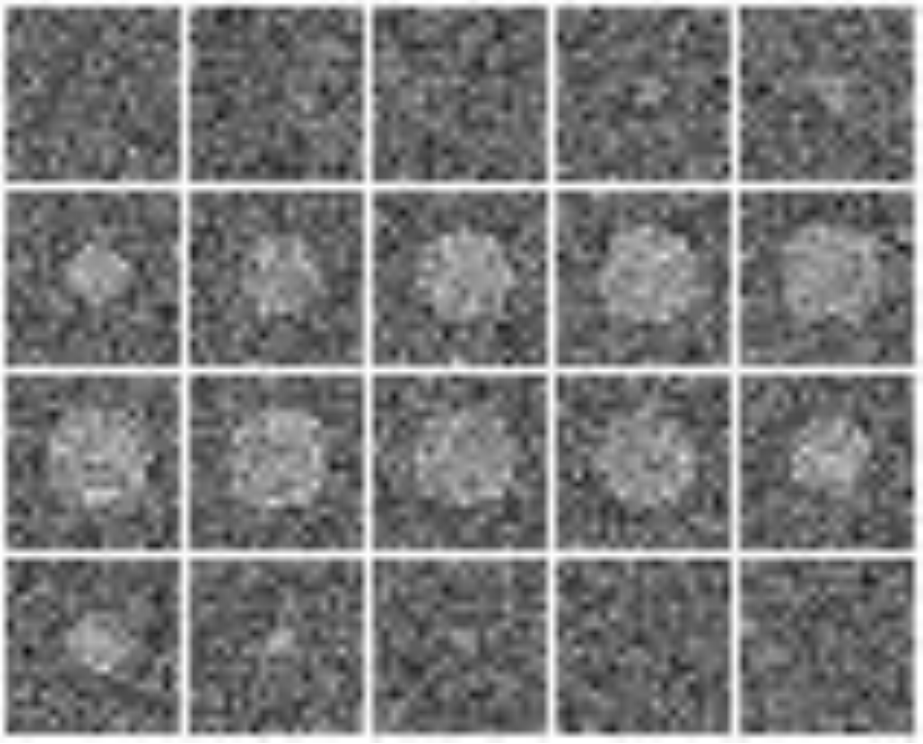

Figure 7 - Tile image of “t1n.vx” image sequence.

We can see how the foreground-background in Figure 6 is less contrasting compared to the original image in Figure 1. In Figure 7, we were not able to clearly notice the object due to the noise that was introduced.

The “v3dedge” program is first used to detect the edges in “t1g.vx” by running the following command:

```
python v3dedge t1g.vx of=t1g-edge.vx
```

The output image looks good, except for the detected edge on the border of the image. This is caused by how the input image is padded with zeros in the border, which results in an edge detected because the background in image “t1g.vx” is of pixel value 100 and is pretty contrasting to pixel value 0. This flawed edge is removed from the output image by clipping 1 pixel of the borders using “vclip” and the output image will by 18x18, with 20 image slices. Then, for additional processing, the “vthresh” command is used to automatically threshold the output, which would provide a better visualization. The output is then tiled using “vtile” as shown below:

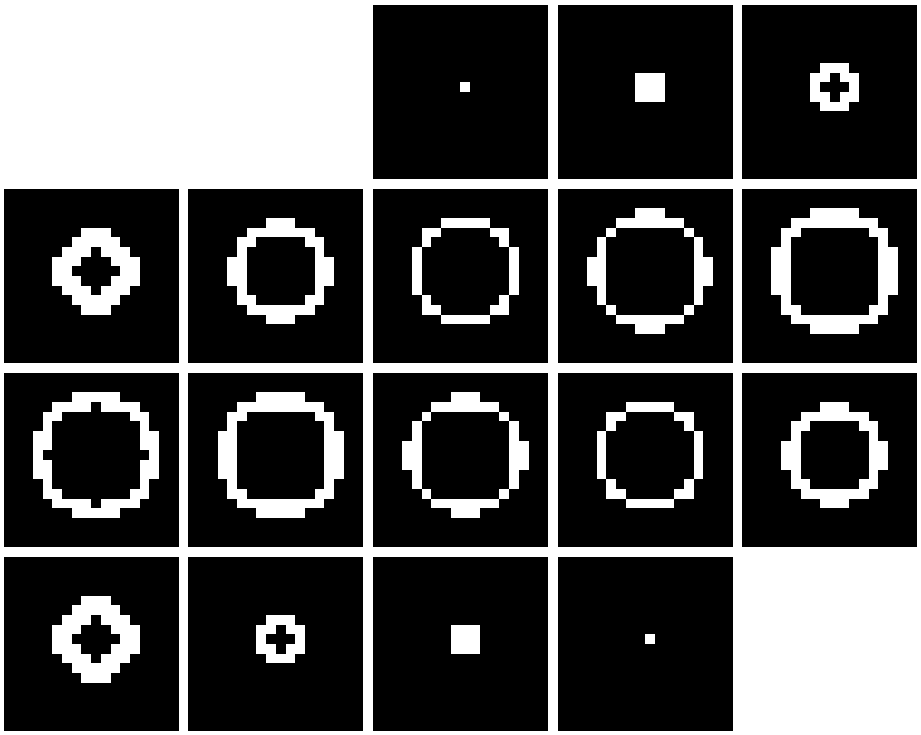

Figure 8 - Tile image of “t1g-edge.vx” image sequence.

From the image above, we can learn about the performance of edge detection of the original image in Figure 6. When the object/region is small, such as in image slices 3-6 and 16-19, we can see that the edge is not represented well. This is because some of the regions are too small, and when the edge is detected, it will be too thick. Worse, several edges detected may overlap and result is flawed, such as in slice 4, where the original shape is a square and the edge is also a square. On the other hand, when the region is bigger, we can see a good performance in edge detection, such as from image slices 6-15. The edges are able to be fully connected to each other (when using 8-connected pixels in foreground).

Next, the “v3dedge” program is tested on the noise-added image “t1n.vx”. The tiled output image is as below:

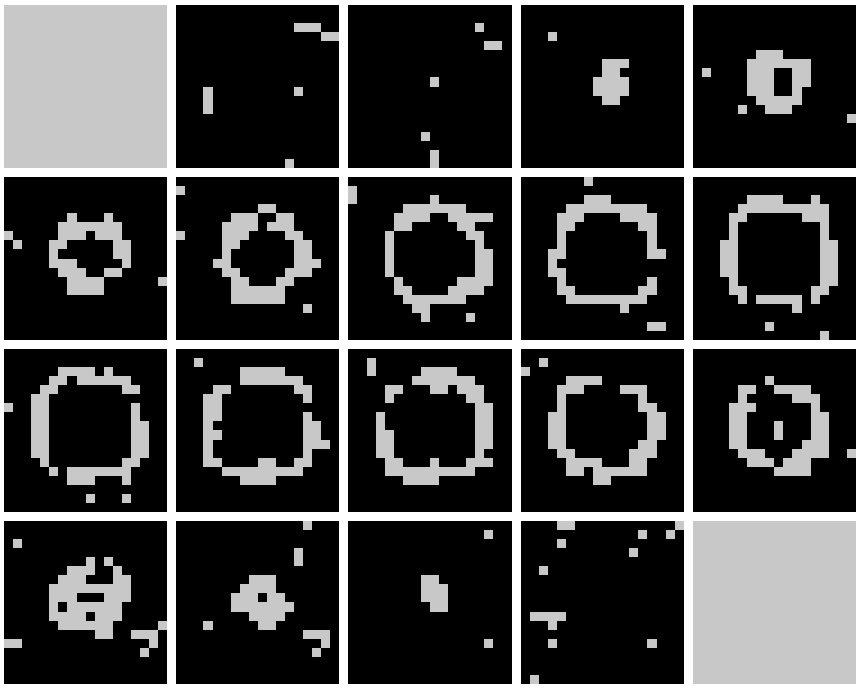

Figure 9 - Tile image of “t1n-edge.vx” image sequence.

From the image above, we can immediately see how the program has a poorer performance when performing edge detection in the “t1n.vx” image compared to “t1g.vx”. This is because of the noise that is added to the image. In the surrounding pixels of the detected edge in Figure 9, we can see some false edges detected (the dots), which is due to the noise. Additionally, we can see how some of the edges of the object were not connected and thus the edges are not a closed boundary. The figure above also has not been threshold and so the result can still be improved.

A simple program is created to try and remove the single-pixel noise. The code snippet is as below:

```
inimage = vx.Vx( vxif )
im = inimage.i
tmimage = vx.Vx( inimage )
tmimage.embedim((1,1,1,1,1,1))
tm = tmimage.i

# Clear image for output
for y in range(im.shape[0]):
    for x in range(im.shape[1]):
        im[y,x] = 0
for z in range(im.shape[0]):
    for y in range(im.shape[1]):
        for x in range(im.shape[2]):
            if tm[z+1,y+1,x+1] > 0 and (tm[z+1,y+2,x+1] > 0 or tm[z+1,y,x+1] > 0 or tm[z+1,y+1,x] > 0 or tm[z+1,y+1,x+2] > 0):
                im[z,y,x] = 255
            else:
                im[z,y,x] = 0

inimage.write(of)
```

Figure 10 – Code snippet of the simple program to remove noise.

The image in Figure 9 is first thresholded with the “vthresh” command for automatic thresholding. Then, the process above is used for the output in Figure 9. As a result, some of the noise was able to be removed:

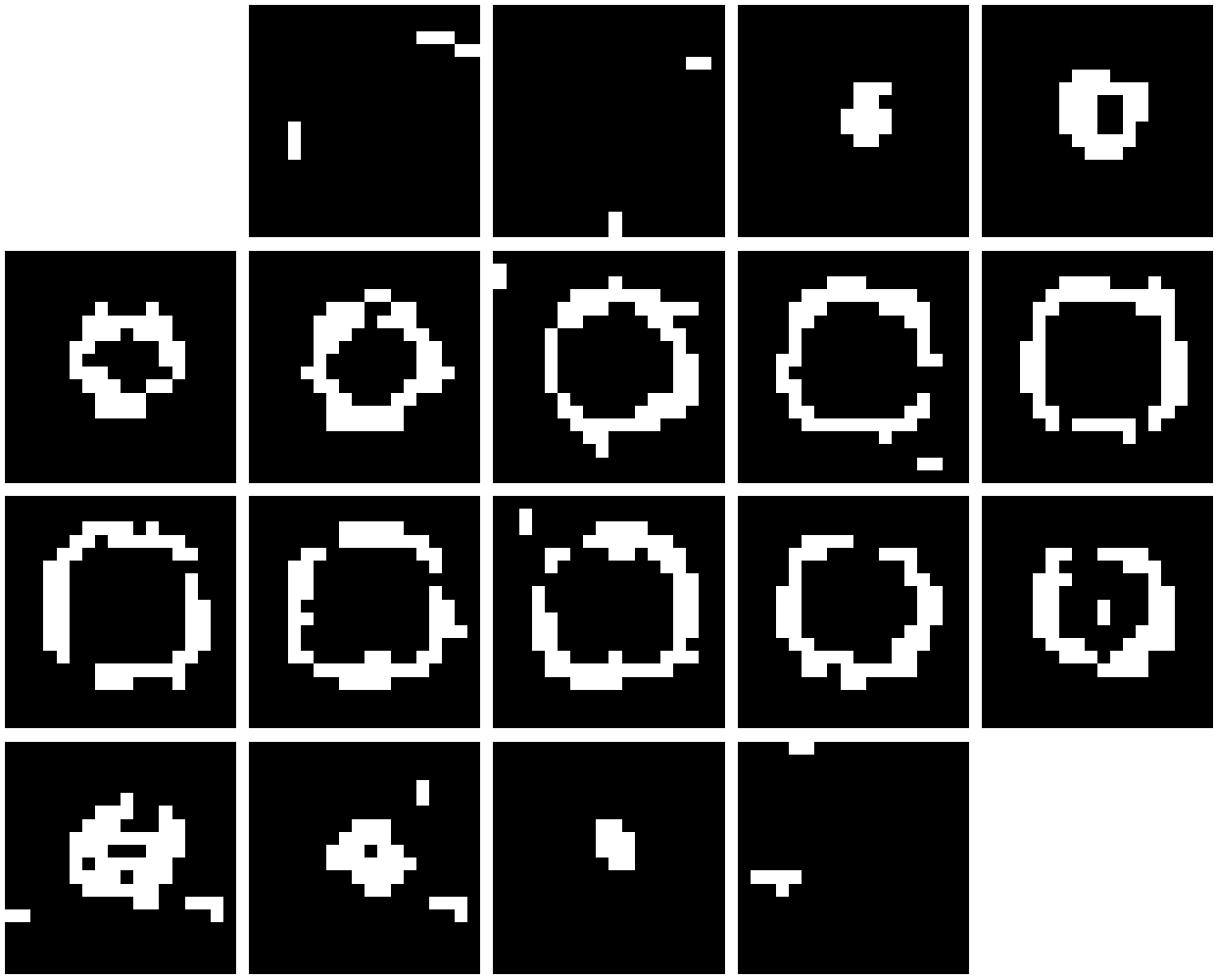

Figure 11 - “t1n-edge.vx” image sequence with removed noise.

From Figure 11, we can see that with the program that is written above, some of the single-pixel noise are now removed. However, we can see some noise still exist in the figure above. A more advanced program will need to be designed in order to filter all of the noise above. We have learned and proved how edge detection will not work well in an image with noise. That is why it is highly recommended to first perform a Low Pass-Filter to remove the noise in an image before detecting the edges. Additionally, more-advanced post processing may also be done to connect the appropriate edges that were not connected, but should have been.

Three-Dimensional Segmentation Evaluation:

In this final section of the lab, we will be exploring how 3D segmentation may be evaluated in respect to the ground truth. Firstly, a manual segmentation is performed on a 3D lung image. Using vdview, we can view the 2D image slices representation of the 3D lung image. Using the boundary marking tool, new points and boundaries were outlined on a pulmonary nodule. The boundaries were then saved as “ctimiage.2.vxa”. Then, a script called “analseg” was executed. The commands were:

```
vrdiff if=ctimage.2.vxa bf=nodseg.vx -cp of=compare.vx
ig=ctimage.vs os=compare.txt
vclip f=10,29 compare.vx | vtile n=5,4 of=dcompare.vx
varend ctimage.2.vxa -b -f | v3pad x=1 y=1 z=1 |vdim -c
of=mregion.vx
v3pol -t mregion.vx | v3pfilt of=mregion.vd
v3view mregion.vd of=dmregion.vx
v3pol -t nodseg.vx | v3pfilt of=nodseg.vd
v3view nodseg.vd of=dnodseg.vx
```

In the first line, the command “vrdiff” was executed to output the statistics and visualization of the image difference based in the input boundaries “ctimage.2.vxa”, compared to the ground truth “nodseg.vx”. The “-cp” flag was used to clip the original image to the size of the comparison image. The “ctimage.vs” was supplied to provide the original image, and the statistical result is output as a text file “compare.txt” while the output image is as “compare.vx”. Since many of the image slices are not relevant, the useful range is clipped using “vcip” then output as a tiled image using “vtile”.

Next, we want to provide a 3D visualization of the boundary that we have created. This is done by using a “varend” command that will trace and fill the boundaries, then adding a zero padding using “v3pad” and having a frame marker for it using “vdim”. The output is named “mregion.vx”. In the final 4 lines, the two images that we will be comparing “mregion.vx” (my segmentation) and “nodseg.vx” (ground-truth) are processed using “v3pol” to convert image to 3D polygon and following marching cube algorithm. Then, “v3pfilt” was used to filter and smooth the surface. Finally, “v3view” command was used to provide a light-shaded representation of the 3D image.

The output images are as below:

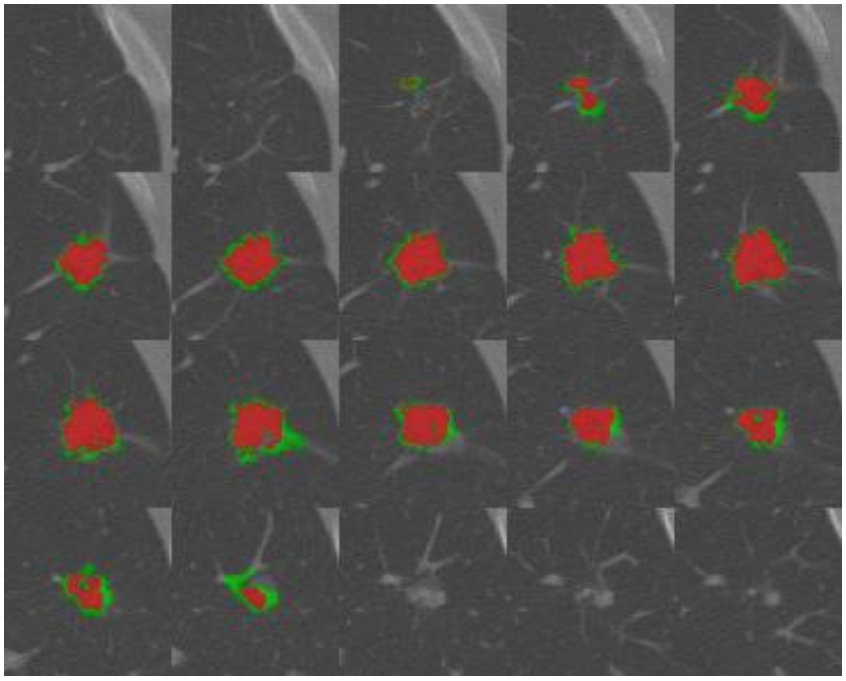

Figure 12 - Image slices of dcompare.vx

In Figure 12, it shows the slices of images where the pulmonary nodules exist. The region shown in red represents the “ground-truth” while the region in green represents the manual segmentation done by me. In every image, the green region is always overlapped by the red region. We can see how in all images where pulmonary nodules exist, there is always an overcovered region, where some green boundaries are not red. However, there are no undercovered regions, which is when a red boundary is not green. Overall, it is great that I was able to segment the image where a pulmonary nodule exists. Additionally, the segmentation region that I have chosen always covers the entire ground -truth. One room for improvement is to be more tight with the boundaries, so that there will be no overcovered region. This will help to be more accurate and a more perfect segmentation.

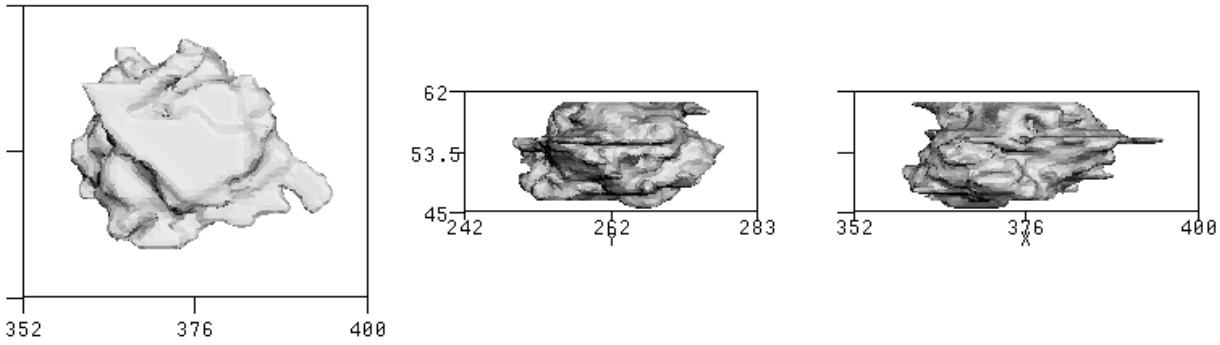

Figure 13 - 3D light-shaded representation of mregion.vd

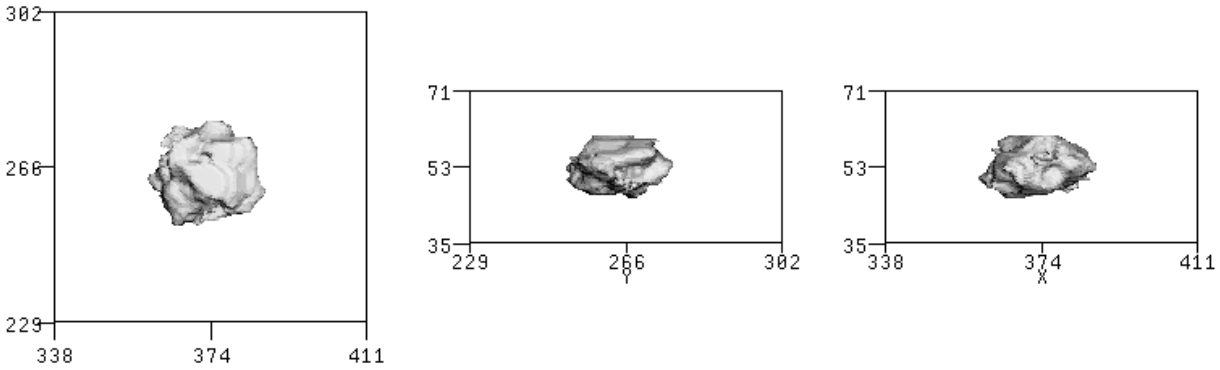

Figure 14 - 3D light-shaded representation of nodseg.vd

Next, we are comparing the 3D light-shaded representation of my segmentation result in Figure 13 and the ground-truth in Figure 14. They are represented from viewing at 3 different angles of the different axes. Overall, the result of my segmentation has a pretty consistent shape with the ground truth. The primary difference is that the 3D model from my segmentation result is bigger, due to the overcovering explained above. Additionally, the edges in the model seemed to be sharper compared to the ground-truth representation. However, Figure 13 and 14 is pretty similar and I would say that I have done a good performance in segmenting the pulmonary nodule region.

Finally, we will look into the compare.txt text file. This includes the statistics and comparisons between my segmentation results and the ground truth. The result is as below:

```
apcnt 5367
bpcnt 3849
overlap     3810
anotb 1557
bnota 39
union 5406
diff  1596
tp    3810
tn    186438
fp    1557
fn    39
fpc   0.58535
sim   0.70477
sens  0.98987
spec  0.99172
jin   0.70477
dsc   0.82682
```

First, we can take a look at the fpc value which is 0.58535. Since FPC = (groundt truth - (overcovered + undercovered))/(ground truth), which also stands for Fraction of Pixels Correct, this means that the amount of mistakes that I made (either overcovered or undercovered) is almost half of the value of the ground truth, 41.465% to be exact. Looking at this number, it turns out that my segmentation is not too good after all. Second, we look at the Jaccard index which has a value of 0.70477. The Jaccard index will show us how similar the region that I segment is with the ground truth. From the range of 0 to 1, I scored around 0.7. It is not perfect, but it is a good enough number to show that my performance is acceptable. We also look into the Dice coefficient, which is scored as 0.82682. Similar to Jaccard, it gives us the result of the amount of overlaps divided by the total area. The number means that around 82% of the region is right and overlaps. We can make different conclusions based on the numbers above, where my performance would look bad based on the FPC value, but looks good with the Dice Coefficient value. Depending on the regions or evaluation goal, we can choose on which measurements to follow.

Overall, this section taught us on how we can create boundaries in the image in VisionX and create 3D renderings of the image. More importantly, we learned how two three-dimensional segmentation can be compared. The regions can be compared from a 2D image slice or a 3D representation, which would provide different information about the performance. Additionally, a report of calculations of statistics can also be created. This report includes things like Jaccard index or FPC. This is important because by only looking at images, sometimes it is difficult to determine the segmentation performance. However, when numbers are shown, we can better understand and evaluate the performance of segmentation.