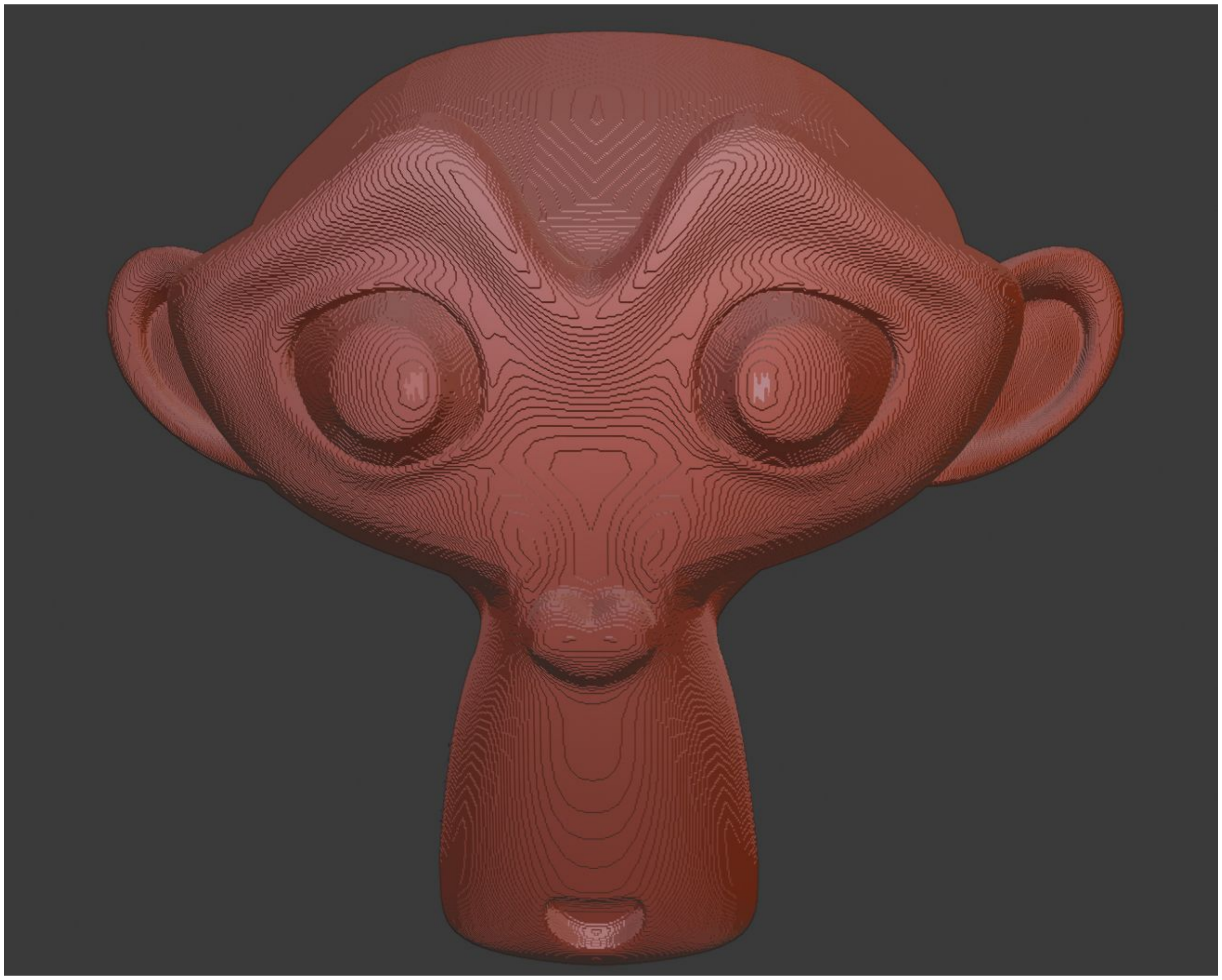


CPSC 589 Modelling Project

Generating 3D Models from Orthogonal Depth Maps

12.07.2019

—

Sebastian Kopacz

Cassandra Lui

Chris Mossman

Overview

This project focuses on creating objects from six perpendicular depth maps. From these six reference images, our program is able to create a base object, which the user can then interact with and modify. This idea was inspired by several other existing tools and methodologies, such as ZBrush's ShadowBox tool, photogrammetry (such as AliceVision's MeshRoom), and Luke Olsen's image-assisted sketch-based modeling. Using a modification of these ideas, we have developed a new tool, scripted in Python and implemented as a Blender 2.8 add-on.

Problem Statement

When modelling 3D objects, it typically helps to use 2D reference images from multiple views. An ideal advancement of this technique would be automating, even slightly, the process of creating an object from such reference images. We hoped to achieve a variant of this using depth maps as the reference images. As far as our research into alternatives, no tool for this currently enjoys widespread usage. Similar tools exist, but include limitations.

As one example, ZBrush's ShadowBox tool uses pure black/white "shadow" images rather than depth maps. Many objects, such as spheres, cannot be modeled this way; if one were to draw a circle as the sphere's "shadow" on each axis, the resulting object would be an intersection of three cylinders, which generates a Steinmetz solid, not a sphere.

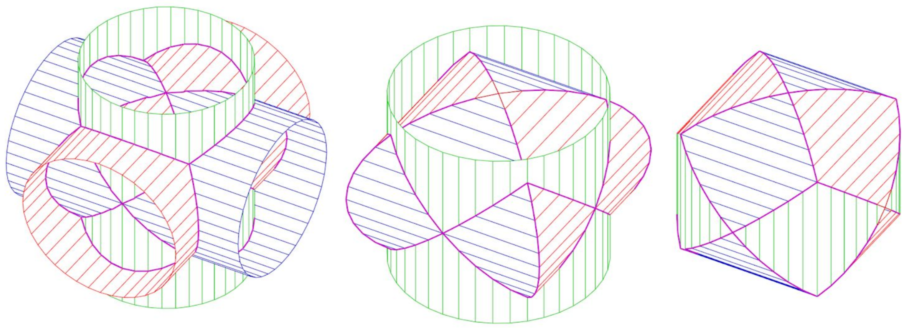


Fig 1.0

A Steinmentz Solid, which is what would result when trying to recreate a sphere from just the outlines from six angles. Image source: Ag2gaeh. (2017)

Another technique to create 3D models from a set of reference images is photogrammetry, which is used by tools such as AliceVision's MeshRoom tool. While the results are visually impressive, it is difficult to modify the automatically-generated result and the process requires a large number of reference images, among other limitations.

Goals

I. Object Creation from Multiple Depth Maps: Creation of an object through our developed program, deriving the object from six depth maps.

II. User Interaction: After creation of said object, we wanted to allow the user to interact and modify the object, either through mouse clicks or sketching.

   A. Though a method to modify the mesh through sketch-based manipulation of the depth maps in Blender has been planned, we ultimately decided to focus on the automation portions of the project given time constraints.

Technical Specifications

As this is an addon written for the Blender environment, you must have the 2.8 version of Blender installed.

When running the scripts in Blender as an add-on, the resolution of the given six images will affect the number of points and edges that the program will draw. Running this add-on on a computer without a sufficient graphics card or memory will sometimes result in Blender stalling for a long period of time until the model can be rendered, or in other cases, freezing or crashing completely. This is mostly unavoidable. We recommend setting the "Max width along any axis" parameter to a low value to start, and then gradually incrementing it over multiple tests until you feel your device could not handle anything greater.

Image Specifications

The user must input depth maps, where darker pixels indicate closer geometry and where the difference in depth in the real life object is linearly related to the difference in darkness of the respective image pixels. The depth maps which are provided by the user must be of equal scale, resolution and format, with the background set to pure transparency. While a more user-friendly version of this add-on would perhaps scale images automatically or try to detect and remove the background, we feel these tasks are not too frustrating to perform manually compared to the others we have automated; Therefore, we have put them aside for the time being. Due to the way we had to generate our test cases, .exr files are recommended, as other file formats have not been tested.

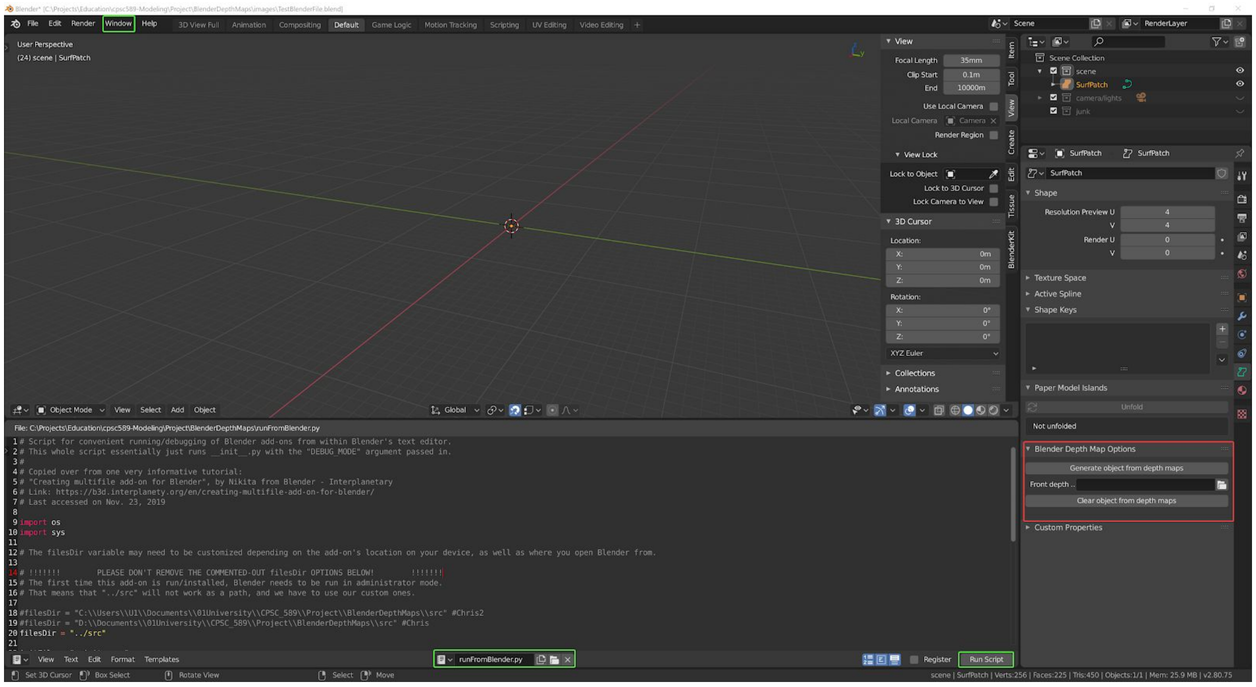
The 6 different images are to represent different views of the object to be rendered, as shown in the examples below. The images can be named as pleased, though including the numeric code or names “front”, “left”, etc. in the image names will assist the Blender add-on in auto-filling the other depth map filenames once you select the front face’s file.

1) 0 to denote the front facing image
2) 1 to denote the left side of the model
3) 2 to denote the back of the model (the farthest area)
4) 3 to denote the right side of the model
5) 4 to denote the top face of the model
6) 5 to denote the bottom face of the model

Installation Process and Usage Steps

1) Open Blender 2.8 using administrator mode, optionally load the included blender file (TestBlenderFile.blend), and run the installExternal.py script from within blender.
2) Next close blender and open the TestBlenderFile.blend which should be 1 folder down from runFromBlender.py.
3) Run the runFromBlender.py script from within blender to add the needed UI functionality.
    a. Optional: Toggle the system console (Window -> Toggle System Console).

4) Select the NURBS surface in the scene, if it's not already selected. The way Blender add-ons work in 2.8 practically require being tied to some object in the scene, so we had chosen a NURBS surface to reflect our ultimate goal of producing a NURBS surface, even though that ultimately didn't become realized within the time constraints.
5) If done successfully the Blender Depth Map Options tab should become available as shown in figure 1.1.
6) Load the first (i.e. "front", or "0") exr file from whichever set of 6 images you wish to use, and the other images in the set should be loaded automatically. If the others are incorrect, you can change them one at a time.
7) Select 'Voxels', 'Marching Cubes', or 'Point Cloud' and click 'Generate object from depth maps' to run the program.
8) This will likely take a while and progress will be printed to the system console (step 3a).
9) The generated object will be relatively large so just zoom out a bit until it is in view.
10) To delete objects, we recommend using Ctrl-Z instead of Delete, otherwise Blender does not seem to free up the memory.


File: C:\Projects\Education\cpsc589-Modeling\Project\BlenderDepthMaps\runFromBlender.py

```
# Script for convenient running/debugging of Blender add-ons from within Blender's text editor.
# This whole script essentially just runs __init__.py with the "DEBUG_MODE" argument passed in.
#
# Copied over from one very informative tutorial:
# "Creating multifile add-on for Blender", by Nikita from Blender - Interplanetary
# Link: https://b3d.interplanety.org/en/creating-multifile-add-on-for-blender/
# Last accessed on Nov. 23, 2019

import os
import sys

# The filesDir variable may need to be customized depending on the add-on's location on your device, as well as where you open Blender from.

# !!!!!!!        PLEASE DON'T REMOVE THE COMMENTED-OUT filesDir OPTIONS BELOW!        !!!!!!!
# The first time this add-on is run/installed, Blender needs to be run in administrator mode.
# That means that "../src" will not work as a path, and we have to use our custom ones.

#filesDir = "C:\\Users\\U1\\Documents\\01University\\CPSC_589\\Project\\BlenderDepthMaps\\src" #Chris2
#filesDir = "D:\\Documents\\01University\\CPSC_589\\Project\\BlenderDepthMaps\\src" #Chris
filesDir = "../src"
```

Blender Depth Map Options
Generate object from depth maps
Front depth ..
Clear object from depth maps
runFromBlender.py
Run Script
Window

Figure 1.1

Regions marked in green are needed in step 3, region marked in red indicates step 3 was successful.

Torus Example

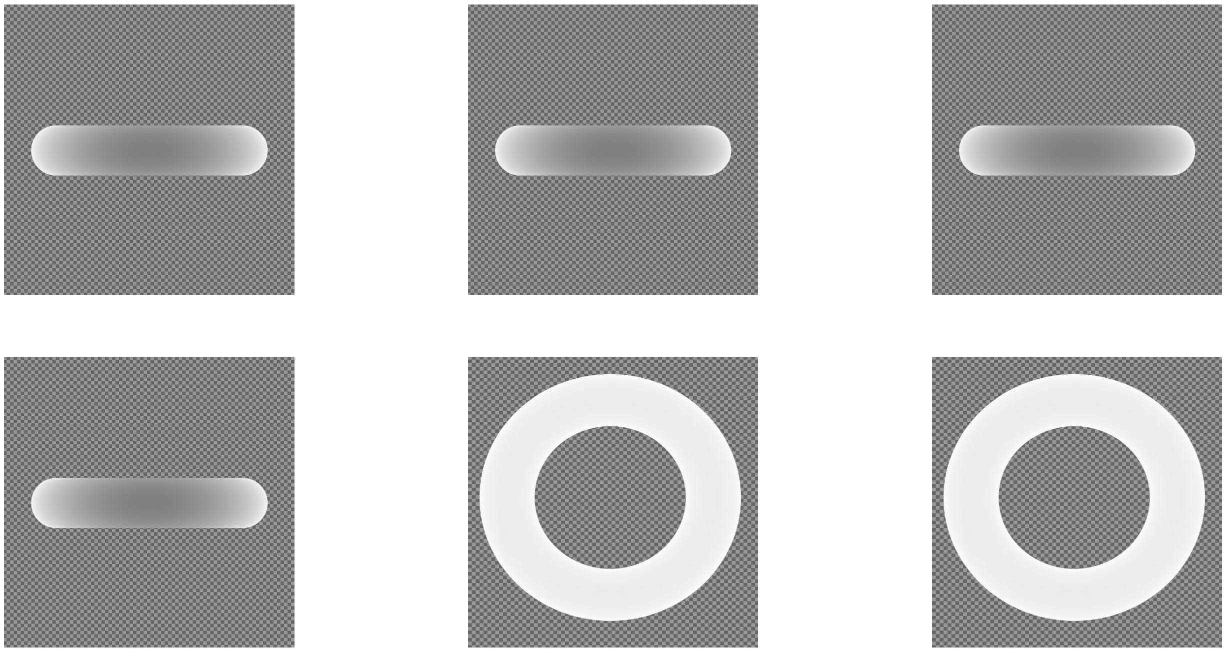


Figure 1.2

The six faces shown here represent the 6 views of the torus to be generated, namely the left, right, front, back faces then the top and bottom faces.

3DBenchy Example

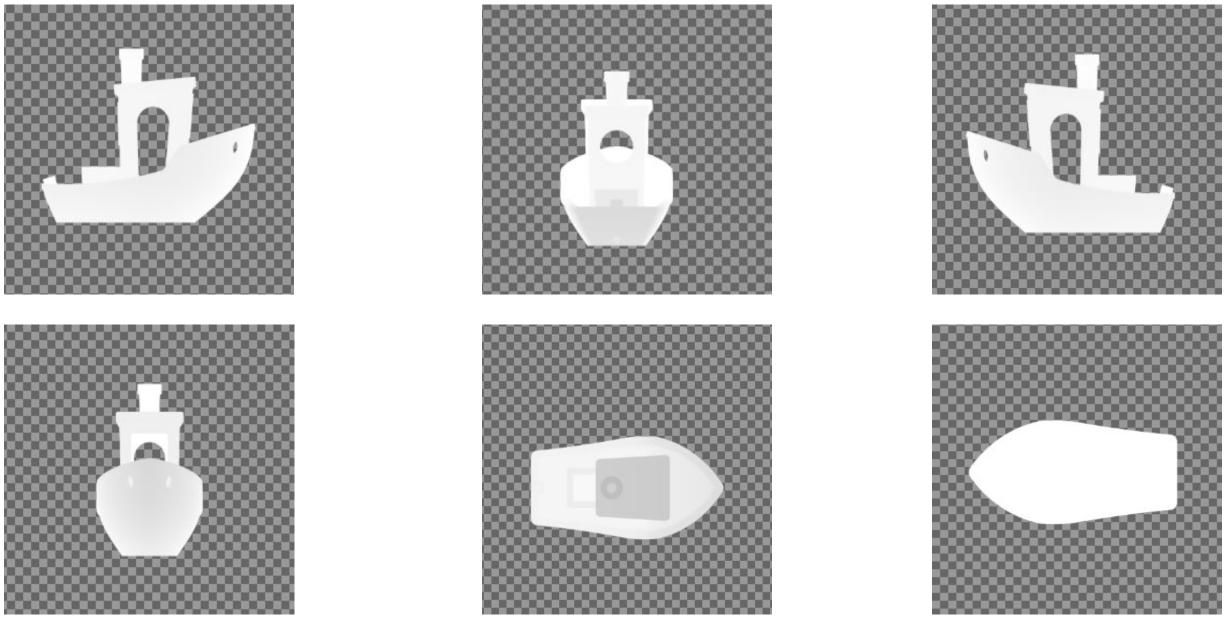


Figure 1.3

For more complicated models, each face shown above has a distinct range of values.

Methodology /Programming Environment

Since Blender supports most modeling workflows, we wanted to develop our project as an add-on for it, as that would increase our project's overall utility. This decision forces us to use Python, since that is what Blender's API is written in. Numpy is used to efficiently manipulate the images and the arrays of points and voxel data, while scikit-image is used to efficiently calculate the marching cube geometry from the voxel input.

There are several approaches we could use to create the models, but converting depth maps directly into surfaces seems to be the simplest option. So, we decided to let users provide depth maps to Blender and then convert those depth maps into a 3D model.

Expected Results (From the Beginning of the Project)

Our goal was a Blender add-on which not only converts depth maps into 3D models, but also allows for easy creation of the necessary depth maps, either through a UI similar to ShadowBox or by converting normal photos into depth maps. Due to time constraints, however, the depth map creation portion of the project was deemphasized to devote more time to automating the object creation from said depth maps. Conversion of depth maps to 3D models is the primary focus of this project.

Process and Steps

Beginning Assumptions:

As stated in the image specifications, the images that the program handles must be of approximately equal scale, format, and type.

Overview Step-Through:

1) The algorithm first crops to the bounding box of each image; where each image's boundary should contain an opaque pixel between said boundary and the image.
2) From this boundary image, we then analyze each image to determine the highest value pixel and the lowest pixel located in the image.

3) Once these float values are determined, the next step is to figure out how far the pixels are from the edge, i.e. how many cells "deep" the highest and lowest pixels each should be. This process is detailed later on in the document.
4) A simple array of 3D points is created by mapping each pixel's depth value to a coordinate using the results from step 3. If the user selected the "voxel" representation for their image the process stops here, and step 5 is skipped.
5) Otherwise, a 3D array is generated where each cell contains "1" for cells outside the mesh, "0" for boundary cells, and "-1" for cells inside the mesh. This process is slightly slower, as we must essentially iterate over each pixel in each image and, in addition to setting the cell that corresponds to the pixel's depth in the 3D array to "0", we must all set all those "in front of" this cell to "1". If the image's pixel is transparent, we know that the entire range of cells through the 3D array lining up with this pixel should be set to "1". The 3D array is initialized with "-1" in each cell, so all remaining cells will keep this value and be considered as "inside" the mesh.
6) Geometry is then created and displayed, either using voxels or marching cubes.

Height Range Calculation Algorithm

Consider the darkest/highest pixel in each row of an image. By virtue of being the furthest distance from the centre of the object, these pixels will also be visible from the side. Therefore, we know that neighbouring image's outline must match up with the highest pixels in the original depth map. Similar logic applies to the highest pixels in each column.

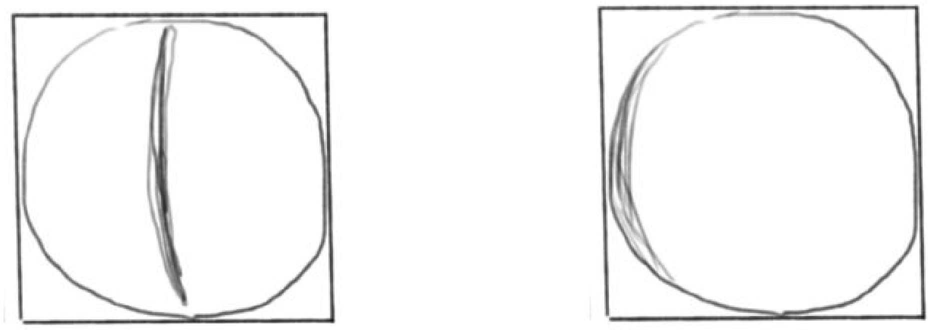
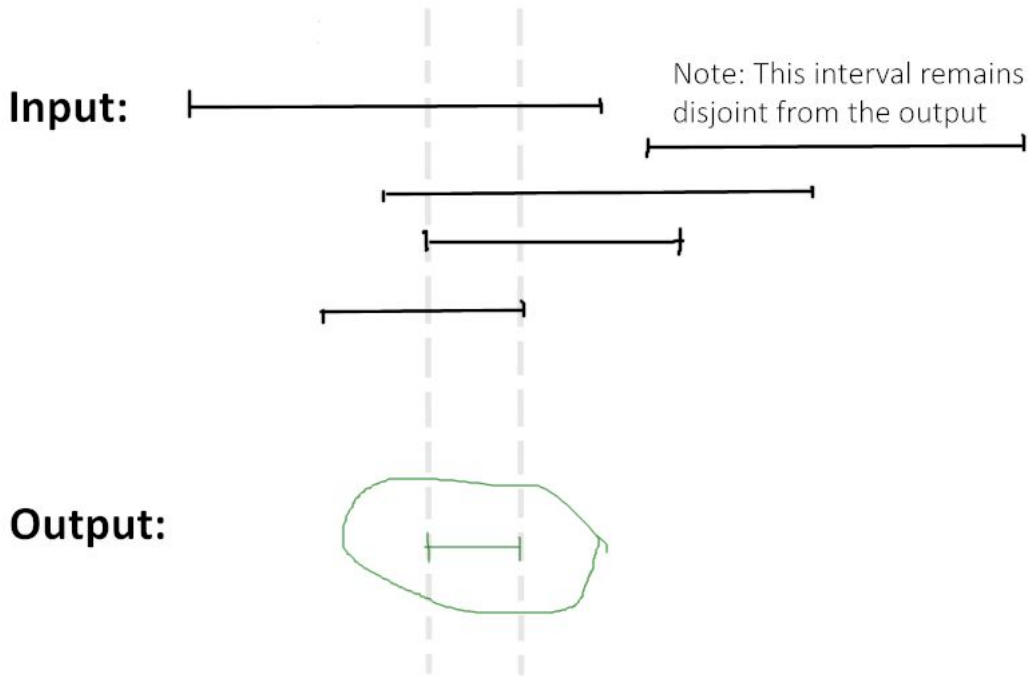


Fig 1.4

A simple sphere example. We know the darkest (nearest) pixels in the image on the left must line up with the outline of its "neighbour" depth map, the image on the right.

With this in mind, we can calculate the height range multiplier that would take each of these highest pixels to the outline of the neighbouring image in the same row. Since we are dealing with height maps, which have discrete pixels in them, there is in fact a small interval of multipliers that would work. After all, a discrepancy of 0.01 units, for example, would still result in the same voxel placement, excepting the rare case of rounding going from 0.49 to 0.5 or vice versa. In particular, we allow a tolerance of 1.49 voxel units on each side of our outline location; 0.49 to represent ending up in the same voxel, and an additional 1 unit to account for semi-transparent outline pixels, which may or may not represent the true edge of an object.

From here, we get an acceptable interval for each row in each neighbouring side image. Likewise, we can do the same for highest pixels in each column and the neighbouring top/bottom images. Ideally, all of these intervals would intersect, but imperfections may result in this not being the case. Therefore, we use Marzullo's algorithm to output the optimal relaxed intersection, then choose the midpoint from this result. A diagram illustrating an example of Marzullo's algorithm can be found below, and a description of how it works can be found in multiple places online, including Keith Marzullo's Ph.D. dissertation and Wikipedia.



Noted Issues through Progress

As we developed this code, we ran into several issues, documented below.

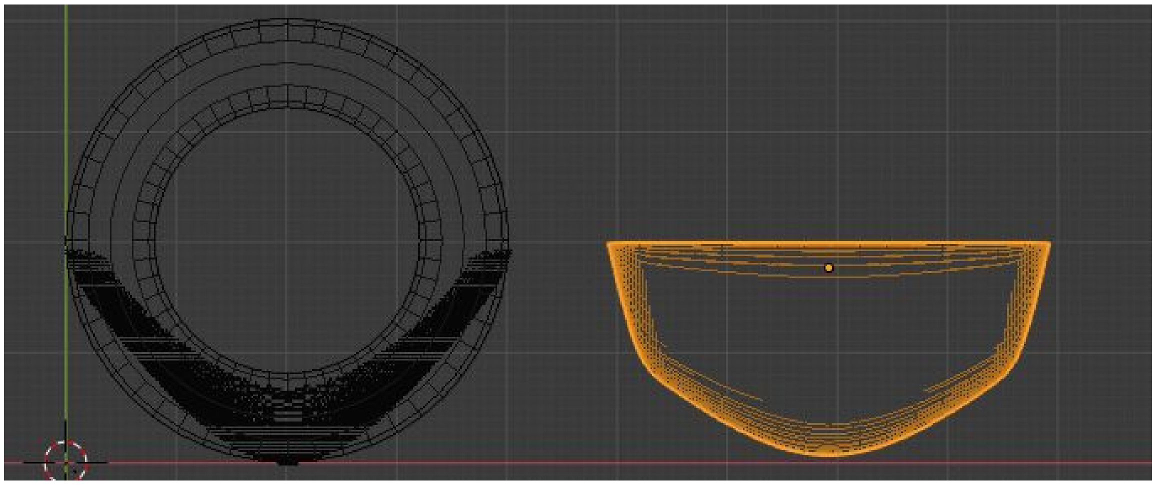


Fig 1.5

The generated face (left) from the script , and a simple single-image height map result (right) did not "curve" enough to properly form the torus when we used our first height maps.

Gamma Correction Issue

In our original depth map test cases (themselves generated using Blender), we noticed that curved objects were being "warped" in an unnatural way. At first, we thought this may be our algorithm, but upon generating a simple height map from a single depth map using Blender's displacement modifier (the orange object in Fig 1.4), it became clear that the depth maps themselves were an issue.

Our original depth maps, rendered to .png format, contained gamma correction that interfered with the linearity between the height values recorded. Upon recreating our depth maps by outputting to an .exr format, which did not perform gamma correction, this issue was resolved, as could be seen when we again tested the depth map using Blender's displacement modifier.

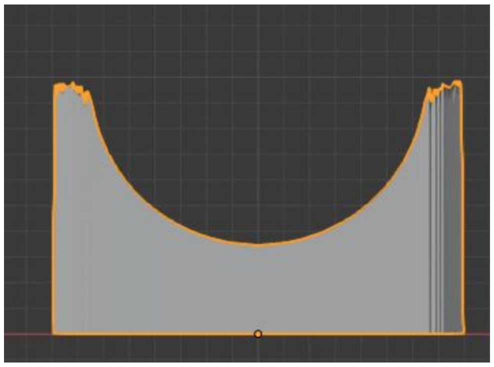


Fig 1.6

Changing the “filtering” of gamma values to invoke a change within the curve

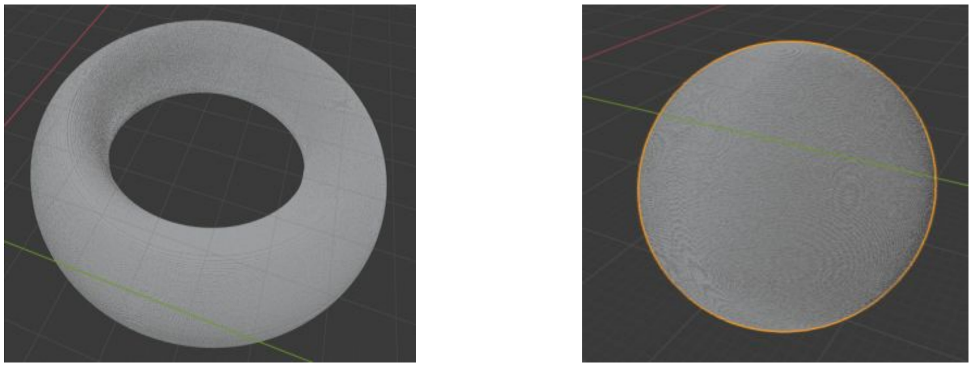


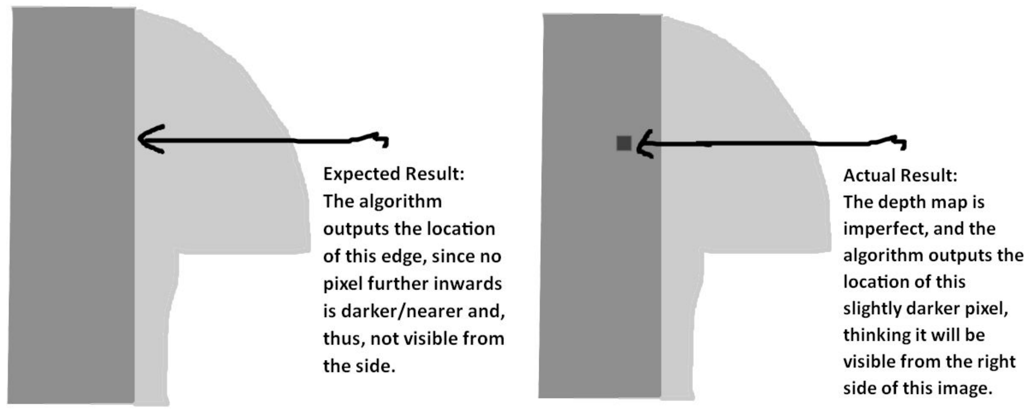
Fig.1.7 and 1.8

Generated Torus and Sphere

Height Range Calculation Issues

In one of our initial height range calculation algorithms, instead of looking at matching the outline of the object seen in neighbour images, we first tried to find the lowest pixel on the edge of the object, and match it up with the pixel furthest-away in that “direction” in the corresponding neighbouring image.

This worked well for some shapes, but for flat surfaces, minor variations in pixel values could make a single pixel partway through the flat face slightly darker than the ones elsewhere. The algorithm would then think that this pixel was higher than the others on the flat surface, and thereby must be the one that was visible from the neighbouring view. A simplified, extreme example of this can be seen in the following image. In this example, the reference image (the one pictured) borders the image whose height range we’re calculating on the right, which means we traverse through the image from the right, the direction we’d “see” it from the original.


Expected Result:
The algorithm outputs the location of this edge, since no pixel further inwards is darker/nearer and, thus, not visible from the side.
Actual Result:
The depth map is imperfect, and the algorithm outputs the location of this slightly darker pixel, thinking it will be visible from the right side of this image.

The result of the old algorithm on one of our actual test cases:

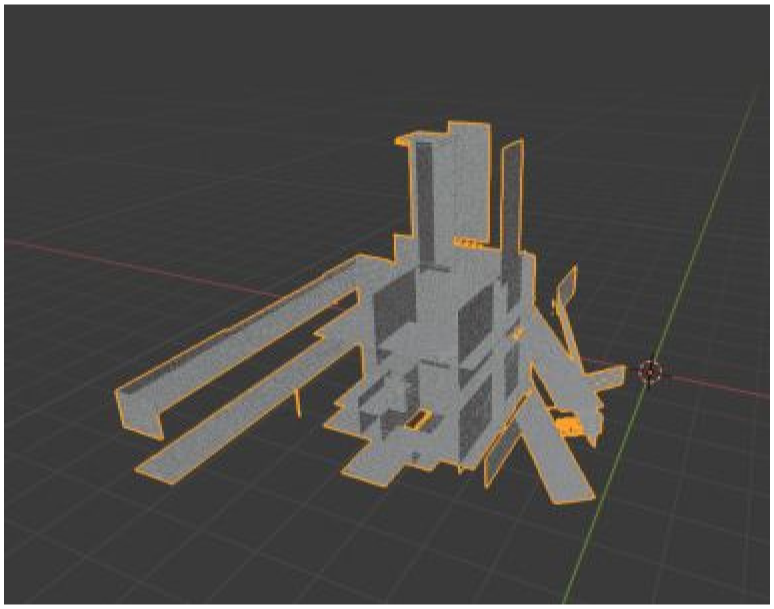


Fig. 1.9

Attempted "Boxes" Test Case Generation

After trying various alterations to our algorithm, including one similar to our final algorithm but using statistics instead of interval intersection, we found our current approach, which generated the correct voxels for all of our test cases.

Other Issues

We also found the following: images that were extremely large either took forever to filter in, or made Blender crash. Since the crash would come when Blender tried to render our object, and not during the data generation, this was not an issue that was in our control or that we could fix by improving the efficiency of our software. As a result, we implemented a feature that would let the user set a maximum width for each dimension of the object, to scale the images down by before creating the mesh.

Noise

Moving onto larger images (.exr) files, we tested the program on other complicated models, such as the dragon. There was some minor noise, which we have since introduced a simple algorithm (looking at k nearest neighbours) to remove.

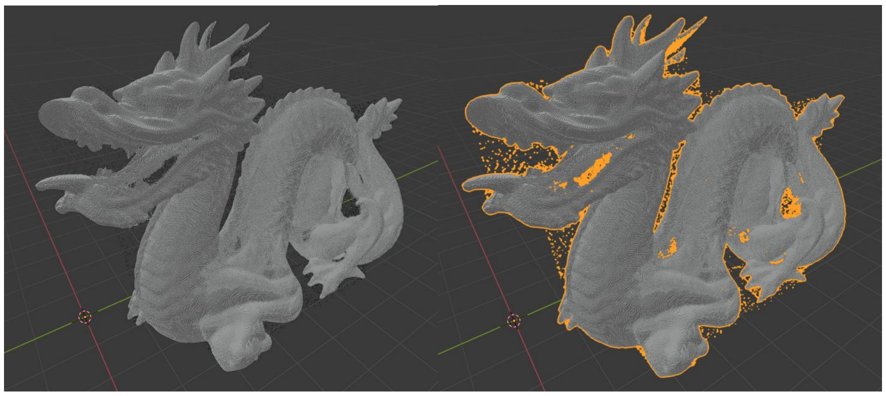


Fig 2.0a

Dragon Model, Noise is visible on the right model.

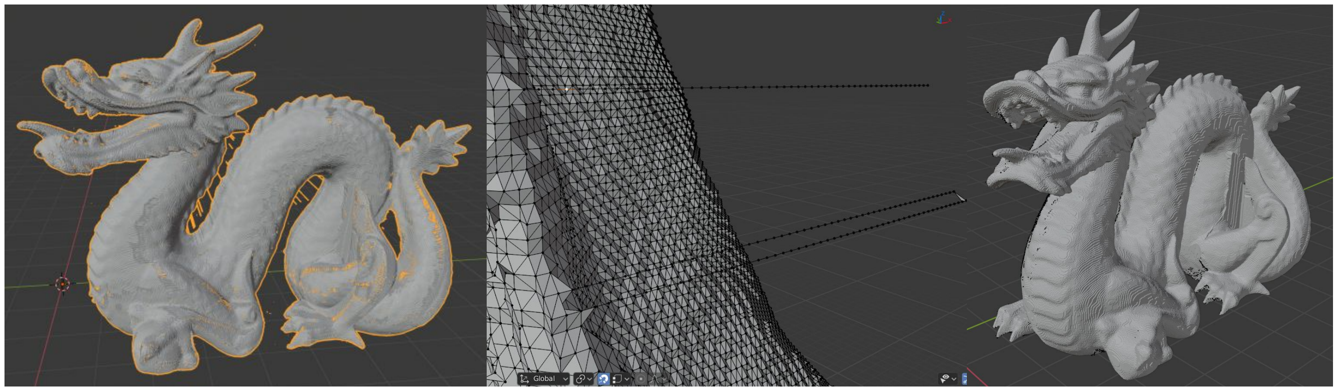


Fig 2.0b

Noise sometimes caused non-manifold lines to form but those could be automatically removed from the mesh.

Marching Cube Generation Issue for Menger Sponge Test Case

During implementation of Marching Cubes, we encountered an odd situation with our Menger Sponge test case. While the edges were placed at the correct height, the flat faces were pushed inwards slightly on each face. This is an issue that we, unfortunately, could not understand and fix in time for our submission, since the Menger Sponge was one of our last test cases, which we tried for fun, thinking that surely such a simple shape would work and look cool. Especially perplexing is the fact that the voxel representation is perfectly accurate, and that other test cases with flat surfaces also behaved just fine. This is an issue we will have to investigate upon further work on this add-on.

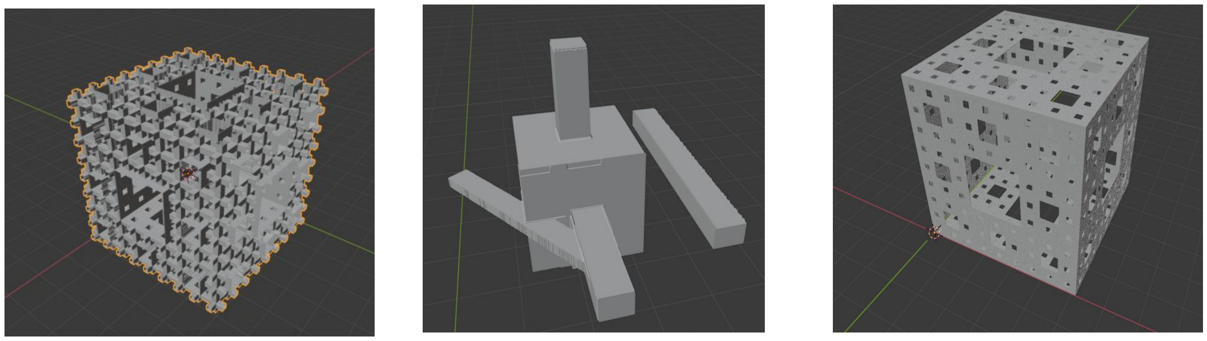


Fig 2.1

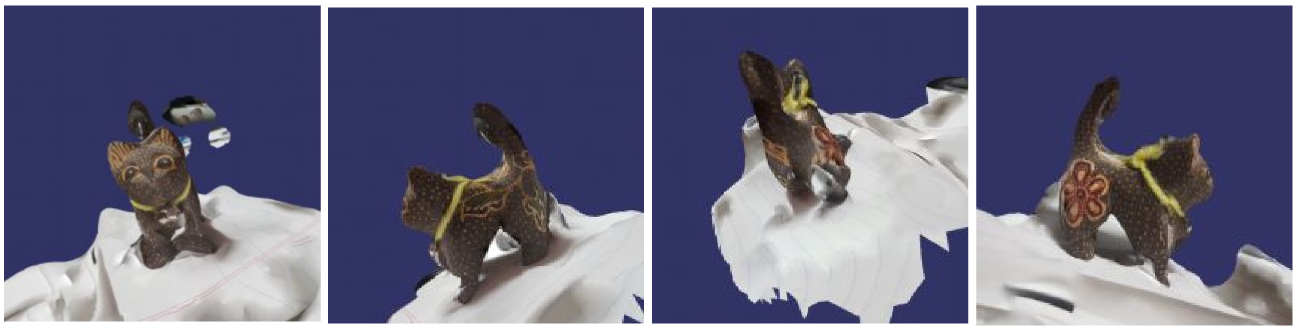
Surfaces are pushed in on the Menger Sponge (left), yet when the similar "Boxes" model (centre) is run through the same algorithm, it generates a model that is completely fine. The voxel version (right) is also mysteriously unaffected.

Comparison to an Existing Application

To get a better understanding of how our program's results compared to that of something already publicly used, we downloaded an application called SCANN3D to see what kind of model it would produce. In this way, we could also see what new approaches/ further enhancements we could then make to our own program.

Example Model of Cat

SCANN3D works by taking multiple pictures around the item in question, requiring a minimum of at least to build a proper model from them. However, as there can be a lot of interference in the real world, such as lighting, there ended up being some interfering meshes being generated, as can be seen below.



Final Results

Torus

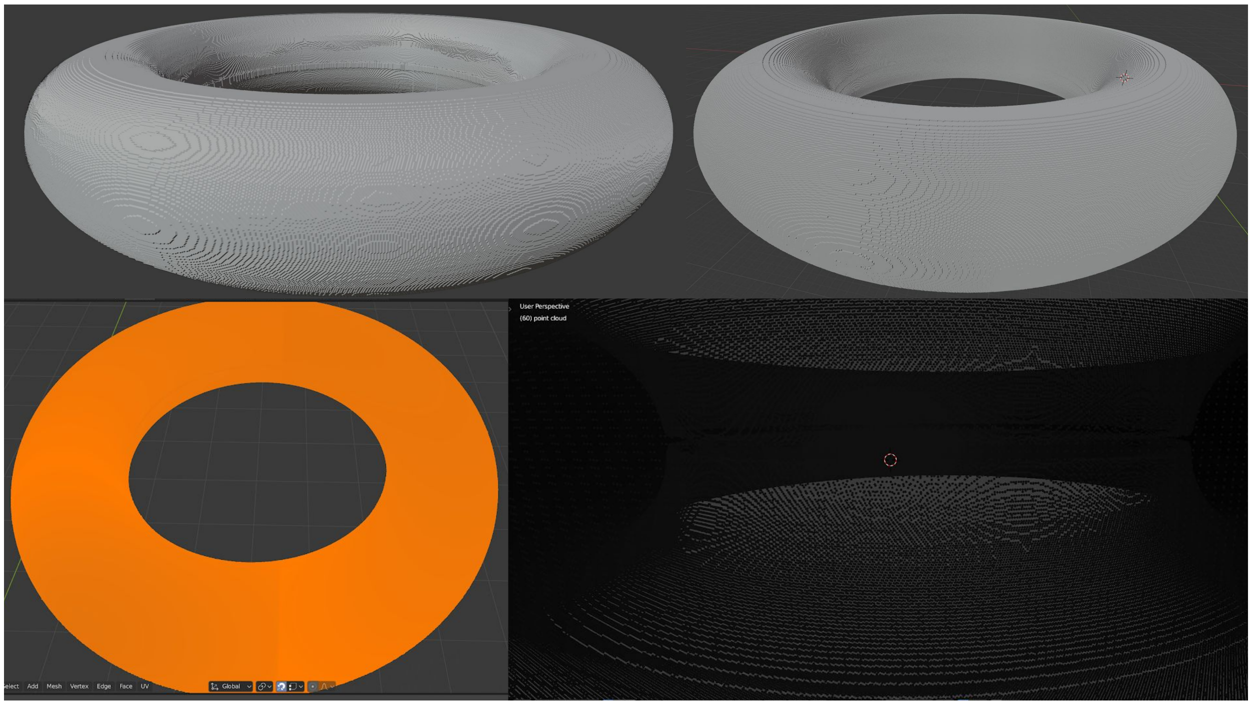


Fig 2.2

Torus: Top left is marching cubes. Top right is voxels. Bottom left and bottom right are point clouds.

3DBenchy

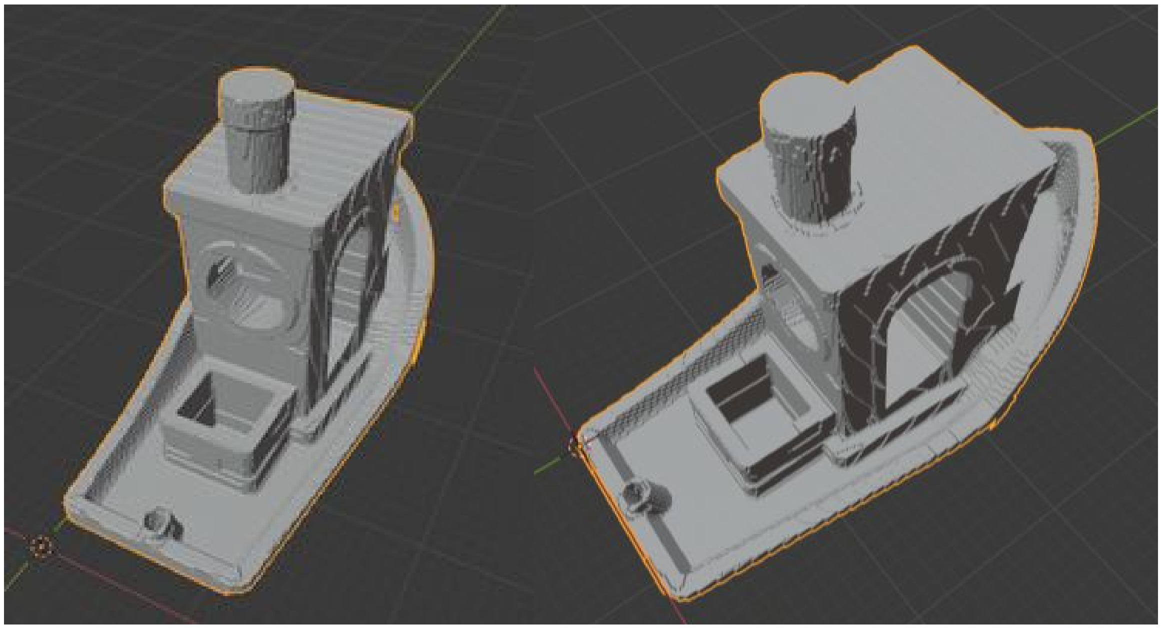


Fig 2.3 & Fig 2.4

The boat reconstructed using marching cubes before we automated mesh cleanup

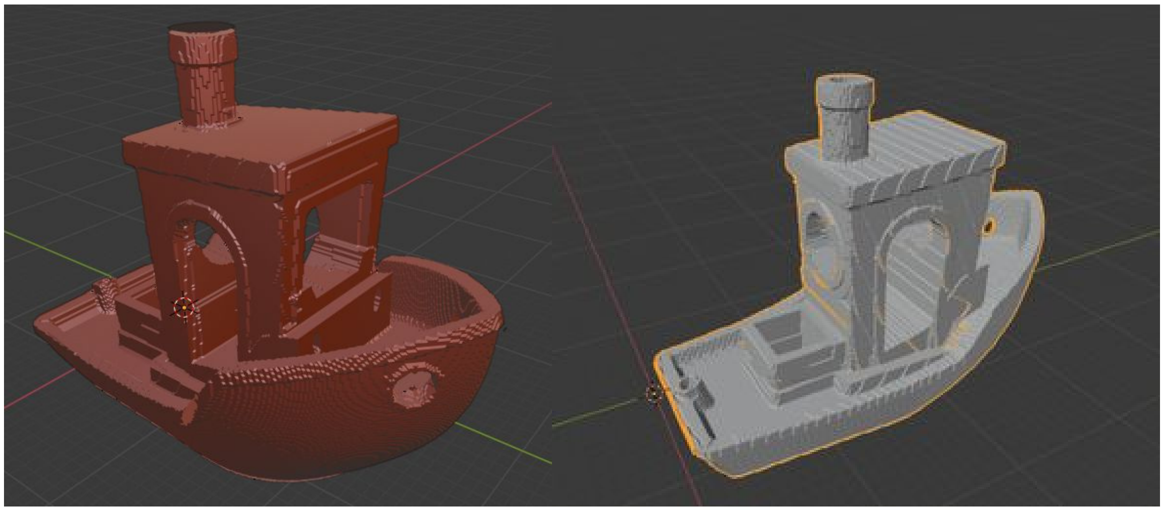


Fig 2.5 & Fig 2.6

Left image is the boat after automated cleanup while the right image is the model from Fig 2.3 & 2.4 but with manual cleanup

Suzanne

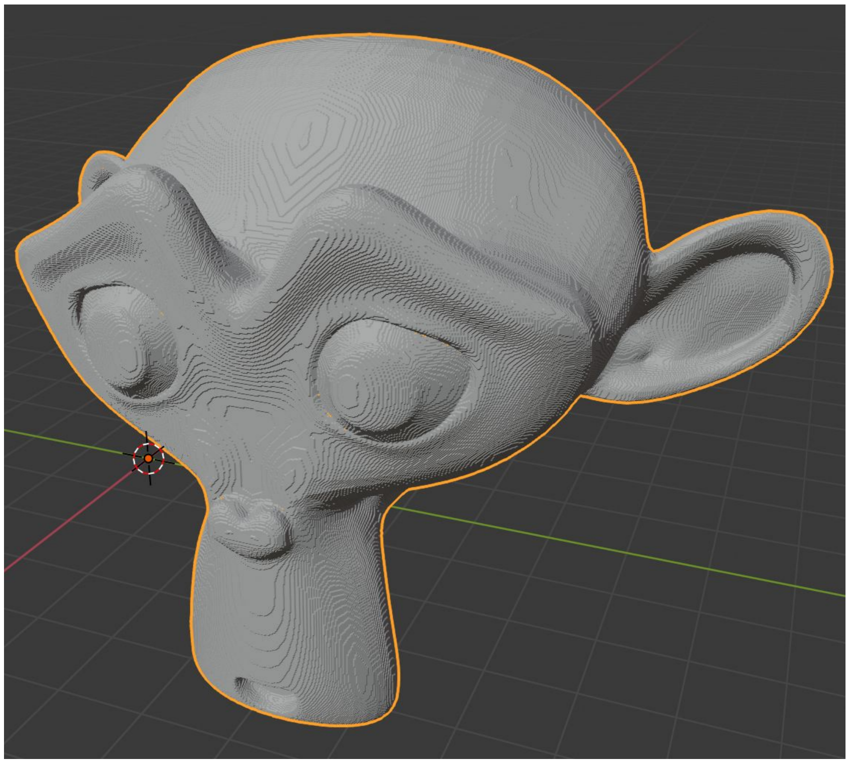


Fig 2.7

Front view of Suzanne after reconstruction from the 6 depth maps using marching cubes

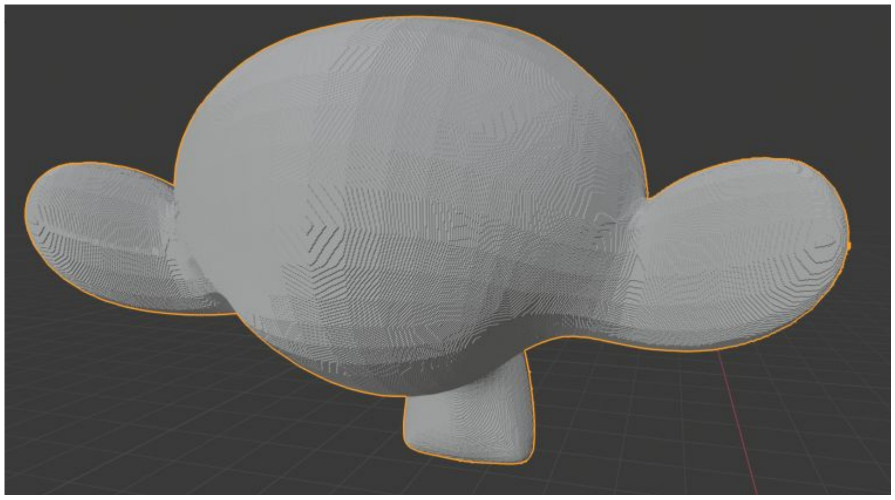


Fig 2.8

Back view of Suzanne after reconstruction from the 6 depth maps using marching cubes

Stanford Dragon

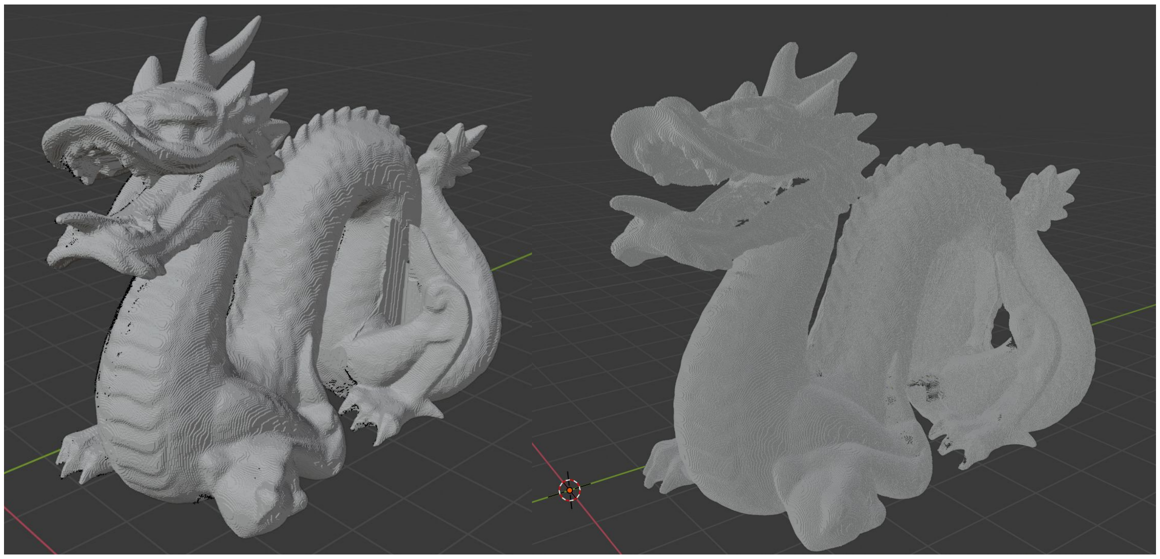


Fig 2.9

Left, reconstructed using marching cubes with automated cleanup. Right, reconstructed using voxels with automated cleanup.

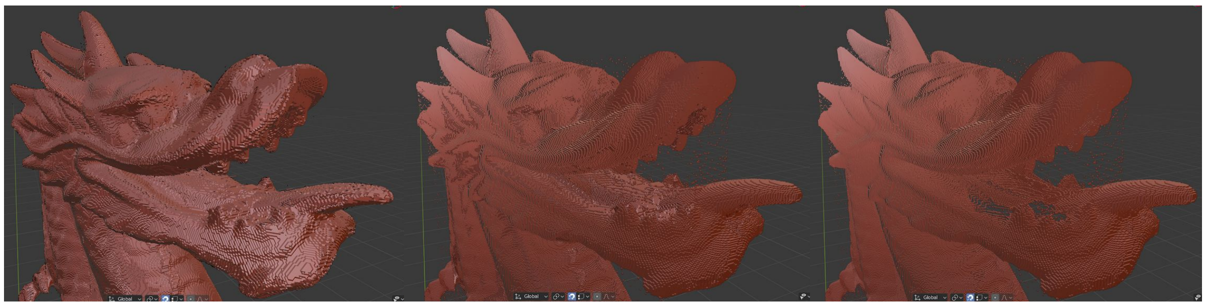


Fig 3.0

Dragon face views. Left is marching cubes only. Right is voxels only. Middle is both techniques overlapping each other.

Utah teapot

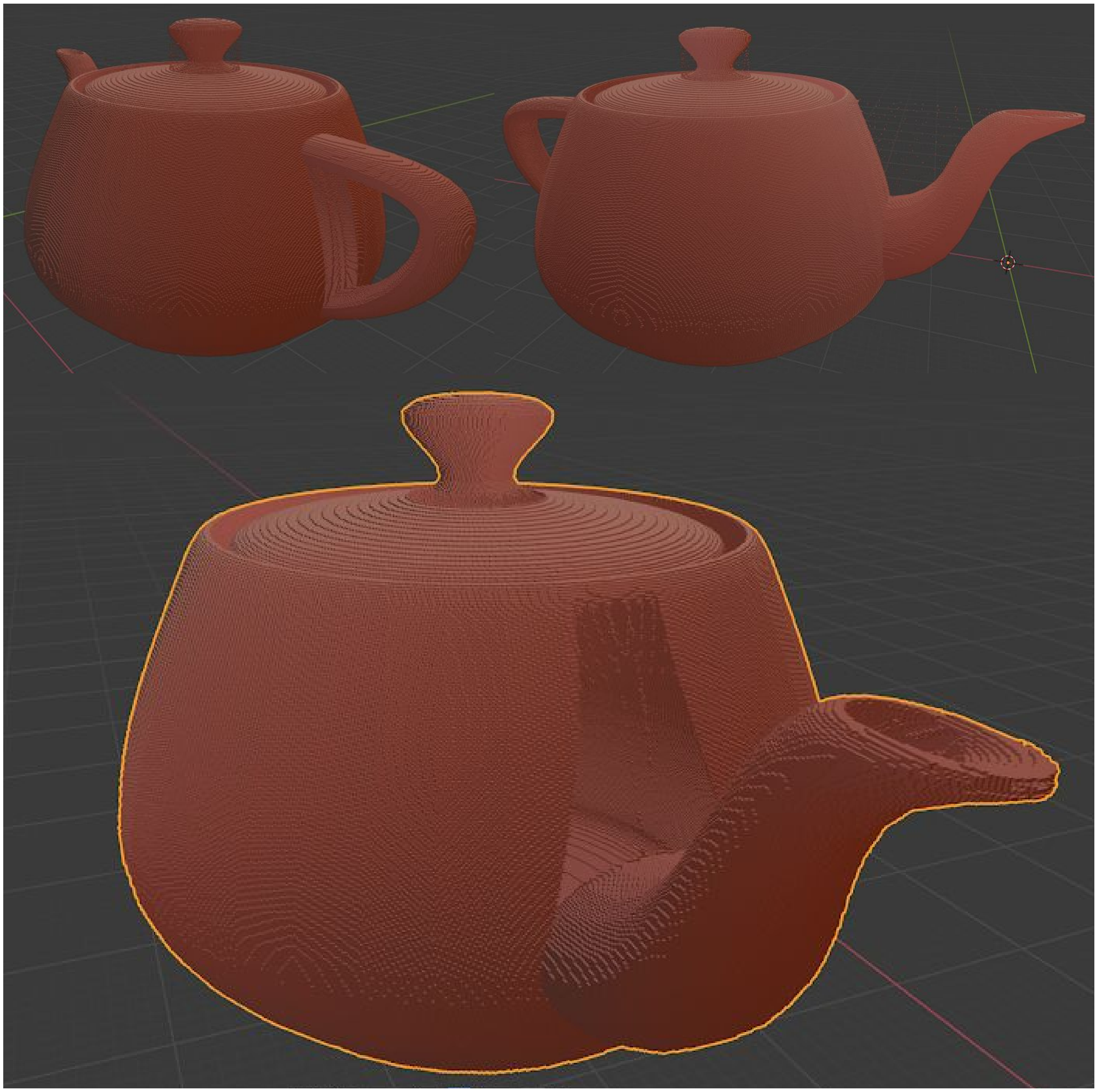


Fig 3.2

Teapot reconstructed using voxels. Top are without automated cleanup, bottom is with automated cleanup. One limitation of our system is that the voxels don't get placed in occulted areas too well. This issue does not affect our marching cubes nearly as much as those work based on values that denote inside/outside (-1,0,1) the mesh rather than on a single value (0) that denotes the mesh's surface. Thus our marching cubes can assume the inside of the surface unless otherwise stated while our voxels are only placed on the surface.

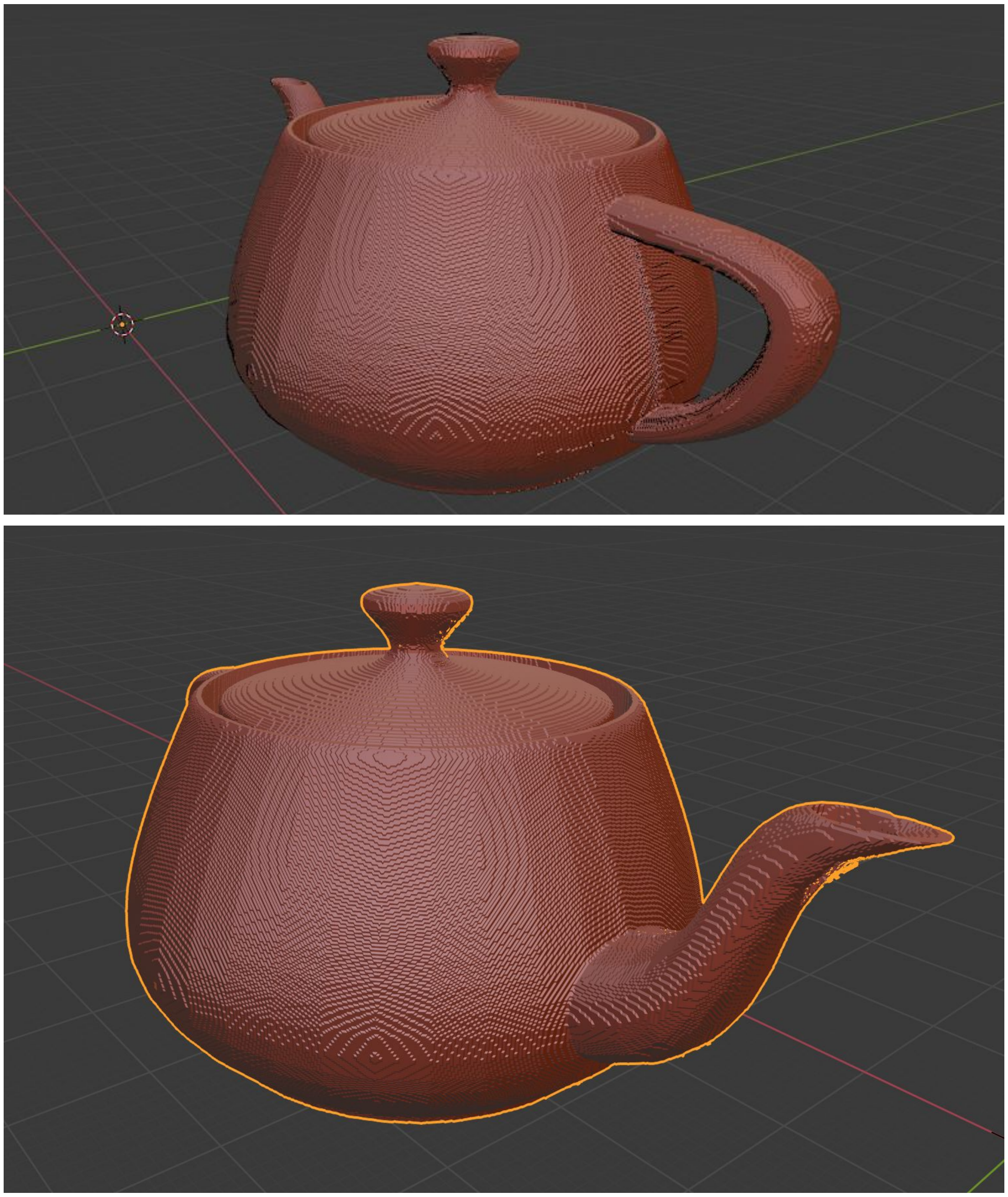


Fig 3.4

Teapot reconstructed using marching cubes. As evident there are no unintended holes in our mesh.

Stanford Bunny

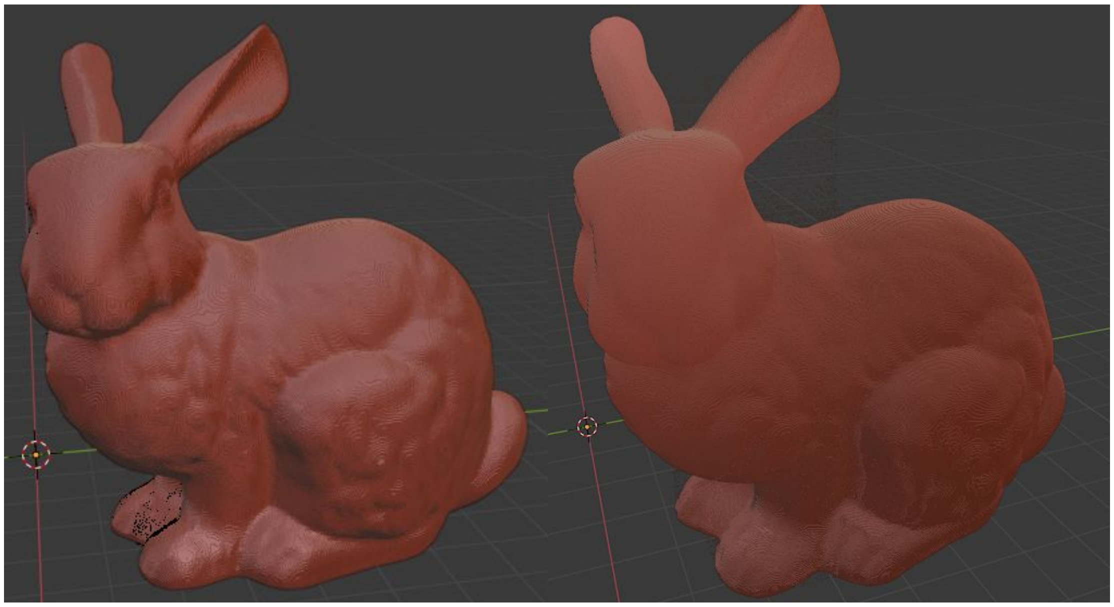


Fig 3.5

Left image uses marching cubes. Right image uses voxels.

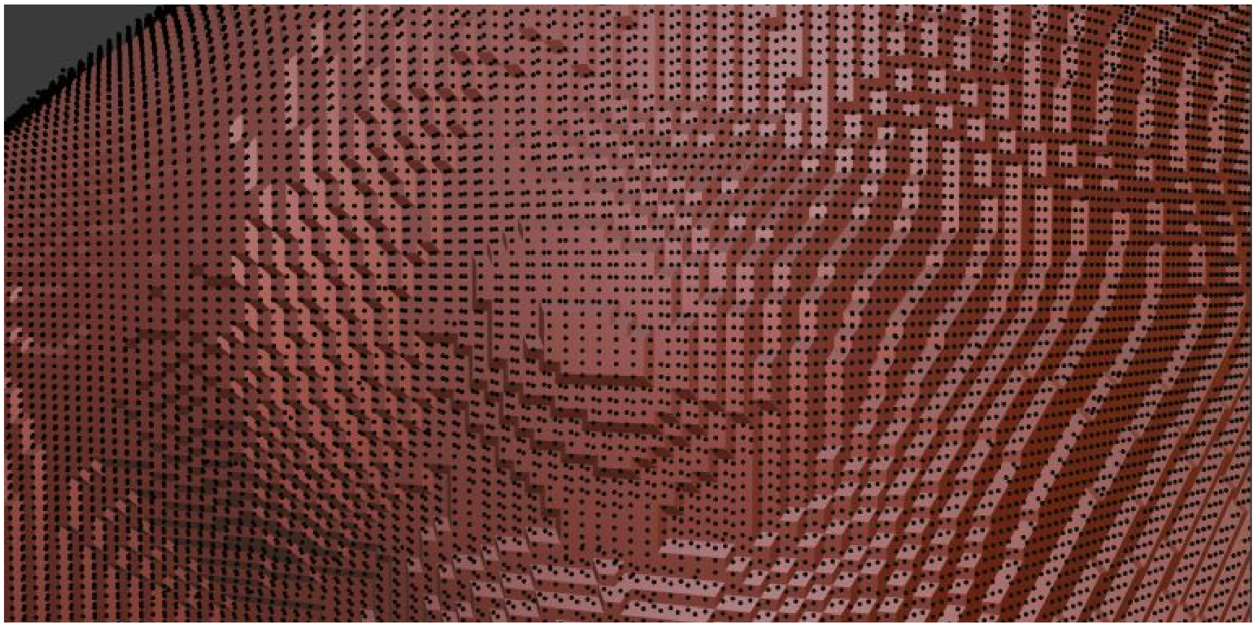


Fig 3.6

Rabbit, Marching Cubes Overlaid with Point Cloud

Conclusion

This project allowed us to reach most of the goals that we set during the initial formation of this group. Though we had some stumbles working with the Blender environment, we were able to produce a viable program that produced the results we wanted. For future improvements of this project, we would like to see it further improved by implementing some of the functions below:

1) Investigate extra geometry added by marching cube algorithm.
2) Package the program in a user-friendly way that doesn't require administrator privileges.
    a) Test with external users to make sure it works.
3) Allow user-friendly depth-map editing within Blender editing.
4) NURBS fitting, or something of similar quality, for generating geometry.
5) Handling images at different scales from one another (a minor usability improvement).
6) Machine learning and depth map generation.
7) In tandem with (6), learning how to "convert" perspective images to those that are orthographic, which would be required for a useful depth map.
8) Semi-automated/sketch tool for starting from "plain" images.

Resources and References

L. Olsen and F. F. Samavati, **"Image-assisted modeling from sketches"**, in *Proceedings of Graphics Interface 2010*, GI '10, (Toronto, Ont., Canada, Canada), pp. 225-232, Canadian Information Processing Society, 2010.

Meshroom: AliceVision, "Meshroom: A 3D reconstruction software"., 2018. https://github.com/alicevision/meshroom

ZBrush's ShadowBox tool: http://docs.pixologic.com/user-guide/3d-modeling/modeling-basics/creating-meshes/shadowbox/

Blender API documentation: https://docs.blender.org/api/current/index.htm

NIST/SEMATECH e-Handbook of Statistical Methods, http://www.itl.nist.gov/div898/handbook/, accessed on December 1, 2019.

https://www.itl.nist.gov/div898/handbook/eda/section3/eda35h.htm

Use of this method was inspired by a StackOverflow answer by Benjamin Bannier, in response to the question "Is there a numpy builtin to reject outliers from a list" asked by aaren.

# Link (Dec. 1, 2019): https://stackoverflow.com/a/16562028

Ag2gaeh. (2017). Generating the surface of a tricylinder: At first two cylinders (red, blue) are cut. The so generated bicylinder is cut by the third (green) cylinder. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Steinmetz_solid#/media/File:Steinmetz-ccc.svg [Accessed 7 Oct. 2019].

Marzullo, K. (1984). ***MAINTAINING THE TIME IN A DISTRIBUTED SYSTEM: AN EXAMPLE OF A LOOSELY-COUPLED DISTRIBUTED SERVICE (SYNCHRONIZATION, FAULT-TOLERANCE, DEBUGGING)*****, ProQuest Dissertations and Theses.**

Complex models retrieved from: https://en.wikipedia.org/wiki/List_of_common_3D_test_models