LONDON'S GLOBAL UNIVERSITY

3D model reconstruction from 2D images: Visual Hull in Python 3

Candidate Number: MVVN2b[1]

Computer Science

Supervisor's name: Dariush Hosseini
Supervisor's name: Tobias Ritschel

Submission date: 4/ 05/ 2021

Abstract

This project aims to solve the reconstruction of a 3-dimensional model in a virtual space from plain images. As input, plain images of an object captured from different angles and orientations by putting the object on a rotating table are used. For the different orientations, the precise coordinates and angles are recorded. Then these images are intended to be used as the dataset to create a 3D model of that object using python programming language. Having no experience of programming in python properly before and even more so of having no experience with 3D programming or Computer Vision principles before, I took this task as a challenge to teach myself all these skills to be able to do this project. I start out with reading papers as to how 3D construction has been done and then worked on a code file to explore and understand these concepts to greater detail.

Contents

Chapter 1

Introduction

Virtual world is fascinating and one of the upcoming latest state-of-the-art technology. Since this world is unreal, any entity can be created and represented here. The only limit to creativity in this world is one's imagination. A much more imaginative world than our real human world can be constructed and experienced virtually. This project focuses on the very core aspect of this. That is translating the real world into a virtual space.

Definition:
"A virtual space is a computer-simulated environment that mimics the real world. There are no topographical or geographical limits in this world."

Thus this project serves as the building block to achieving a fully immersive virtual space. We could transfer all worldly objects such as buildings, houses, and roads to this virtual space. We can then use this virtually created space in many applications such as geographical mapping or remote exploration.
We could also use this space to make video games using real-world structures without manually designing individual entities such as buildings or their design using software such as Unity. Instead, use a camera to take pictures of the object and easily translate a photo-consistent and accurate model in the virtual space.
The first step to solving these is, of course, how to translate anything from real to a virtual world. For this project, the concern is not how accurate the translation of the entity's dimensions is or how colour accurate it is. It is more about how to translate something to a different dimension in the first place. After translating, the next focus is to make the structural appearance of the models as accurate as possible.

1.1 Motivation

The biggest motivation for this project is drawn from video games. The entirety of the gaming world is based in a virtual world. However, its application is not limited to just games but extends to medical imaging, human face detection and object detection.

Figure 1.1: *Iron Man* Hologram Scene: the blue holograms of buildings representing the real world. ©Marvel Studios, Paramount Pictures

However, to mention the single scene that got me into thinking about Computer Vision and its might is this part from Iron Man, where it showcases a hologram of map buildings. That is what got me thinking; what if we could take pictures of all the buildings in the world (thankfully already available through Google Maps) and then make a program to simulate them in the virtual space just like this scene.
In this section, few areas of Computer Vision is highlighted where we try to visualise how these can be improvised further or how they can help achieve the aim of this project.

1.1.1 Augmented Reality (AR)

Augmented reality is probably becoming one of the biggest technology trends of modern time. AR can be characterised as a framework that satisfies three fundamental highlights: a blend of natural and virtual universes, real-time interaction, and precise 3D registration of virtual and genuine objects. The smartphone app Pokemon Go is the most famous example of AR technology. This experience is so well integrated with the physical world that it is considered a fully immersive part of the real world. Augmented reality modifies one's ongoing experience of a real-world environment, while virtual reality fully replaces the user's real-world environment with a simulated one.

Figure 1.2: Pokemon Go ©Nintendo .

How Pokemon Go can be visualised to be improved is if we had accurate models of buildings inside the game that would represent the buildings the player sees in the real world, would be an excellent feature making the game more life-like.

1.1.2 3D Object Recognition

In a photograph or range scan, 3D object recognition entails recognising and determining 3D details, such as the pose, length, or shape, of user-selected 3D objects. In most cases, a vision system is shown an image of the object to be identified in a controlled setting. Then the system locates the previously presented object with an arbitrary input, such as a video stream.
The procedure for identifying a 3D structure is determined by the object's properties. This has been approached in two ways:

1. pattern recognition
2. feature-based geometry.

Pattern recognition approaches use low-level image presence knowledge to locate an object, and feature-based geometric approaches create a model for the object to be identified and align it against the picture. Our study is more motivated towards feature-based geometric approaches.

Feature-based geometric approaches

Feature-based methods are ideal for objects with distinct characteristics. Feature-based object recognisers function by taking a specific set of views of the object to be identified in advance, extracting features from these views, and then matching these features to the scene and applying geometric restrictions throughout the recognition process.

This approach very closely follows the work that we intend to achieve in this project. According to the [Rothganger et al. 2004] which exemplifies this, the method creates a 3D model for the object based on a variety of camera views, including the 3D spatial location and orientation of each element. Since there are many views of the object, each aspect is usually present in several adjacent views. Since the centre points of such matching features align, and observed features are oriented along the primary gradient direction, the points at (1, 0) in the features local coordinate system [Rothganger et al. 2004], as well as the points (0, 1) in the parallelogram's local coordinates, indeed match.
Thus, three-point pair correspondences are known for any pair of matching features in neighbouring views. Given at least two matching features, the Structure From Motion Algorithm was used to estimate the positions of the points.

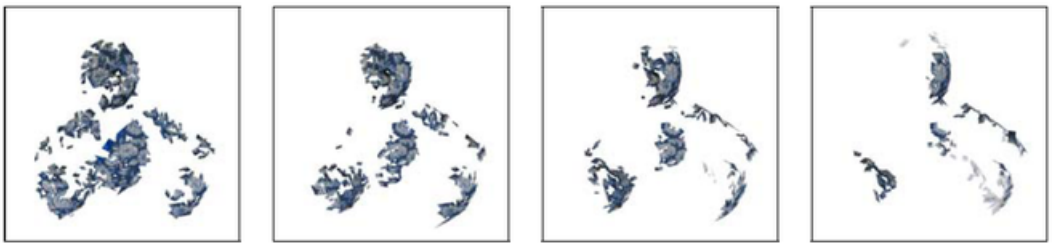


Figure 1.3: Partially projected 3D projections of features created from surrounding views of a teddy bear [Rothganger et al. 2004].

1.1.3 Remote Sensing

Remote sensing involves gathering information about an object or phenomena without physically making contact with the surface being surveyed. The concept of remote sensing primarily refers to gathering data about the Earth in particular and, in a recent case, about Mars.
It is used in various fields, mainly being used for land surveying. It has broad military or economic planning applications, commercial development, and deep-sea exploration. To record the information about the surrounding, a pulse radar emits pulses of energy to scan for objects and locations, during which a sensor absorbs and analyses the reflected and scattered radiation from the target. Examples are The RADAR (RAdio Detection And Ranging) and LIDAR (Light Detection and Ranging) technology. These devices extrapolate the sensor data with respect to a reference point, including distances among the already known points on the ground.

LiDaR - Light Detection and Ranging

This technology is now equipped with smartphones. iPhone 12 has a 3D scanning app that lets users scan objects around them and put them in an AR world using LiDaR technology. The fact

Figure 1.4: Making furniture a model in our phones using LiDaR technology.©Apple

that this comes packaged along with phones motivates us to work on this and take the application usage of this further. With this in hand, a user could scan around anything ranging from objects to buildings around them. For a long time, this has been used in submarines as well, thus opening further deep-sea exploration opportunities.

We could go around our building taking a scan of it and then remodel it in an app. We could also scan the entire interior of a room and put it in Unity or Unreal Engine, where we could do interior designing using custom made objects.

A recent trend, where shopkeepers use similar technology to measure the size of one's feet to recreate a customer wearing a shoe inside the app without the customer ever having to visit a shop.

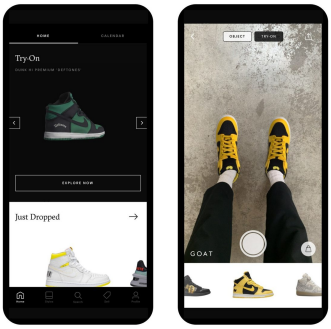


Figure 1.5: Shoe Try on Feature. ©GOAT App - The Wall Street Journal

The motivation section above helped us understand that using sophisticated technology is increasing, and so are its application usages. The equipment is available easily; that is, it is reduced to a smartphone that the general public can carry every day around with them, and sensors are equipped with it. Some 3D detection techniques have been observed, and how they can inspire our work has been explored.

1.2 Project Objectives

Few things that I want to gain out of this project:

1. Introduce how 3D information is captured in the first place
2. Go over the possible methods that can be used to reconstruct any 3d models
3. Understand how visual hulls work
4. Explore the maths behind 3D model projection from 2D images
5. Being new to python, diving into the implementation of visual hulls in a programming language in more depth
6. Experiment and compare results with the different parameters affecting the reconstruction

Chapter 2

Background Study and Literature Review

For a computer vision project, the first steps to trace to is the methodology and the design decisions taken in order to implement it. The first step hence is by investigating the ways in which the 3D shape is acquired. From [Moons:2009] paper, we learn about the different available 3D acquisition categorisations.

2.1 3D Acquisition

For all the 3D capturing methods available, there is a distinction between active and passive methods. To obtain 3D data from active techniques, the light sources are carefully monitored. The light intensity in active lightning is moderated in some way, either perceptually or spatially. For passive methods, light is not regulated or is only controlled in terms of picture clarity, and it mostly uses whatever light is available in the environment, that is, ambient light.
As our interest is in capturing information is in which no external measures are taken, such as adding illuminated lightning, since we want to keep our purpose as simple as using the program using sensors equipped in our handheld devices such as smartphones, we are only interested in passive methods where we only work on the raw data without any manipulation.

For Passive 3D shape Extraction:

1. Singular View-Point Methods
   (a) Shape from texture
   (b) Shape from Occlusion
   (c) Shape from Focus
   (d) Shape from Contour
2. Multiple View-Point Methods
   (a) Passive Stereo
   (b) Structure from Motion
   (c) Shape from Silhouette

2.1.1 Single View-Point

Single View-point methods work by using a "single view" [HAL:112]. For example, the Shape from Focus technique which is also referred to as the Depth from Focus is based on the foundation of the depth of field. 3D acquisition of a scene with high occlusions is solved using this technique.
"Occlusion" is described as "the result of one object/point in a three-dimensional space blocking the view of another object/point", resulting in missing point/series of points or object in one view which can be observed from another view. It uses a stack of 2D images to create a depth map of a picture. This stack is created by adjusting the camera/object distance according to a set of steps, with each step requiring the acquisition of an image in order to search the scene.

2.1.2 Multiple View-Point - Triangulation

As the name suggests, it uses multiple images/view-points. The theory of triangulation is used in many multi-vantage methods to obtain depth knowledge.
Triangulation is a surveying technique that involves measuring the angles created by three survey control points in a triangle. The other lengths in the triangle are determined using trigonometry and the estimated length of just one side.
We will be working with many view-points that is a collection of images that is easily obtainable using smartphones, hence multiple view-point methods are further investigated.

2.1.3 Stereo

Stereo is the term for such a configuration where we have two photographs that were shot at the same time but from different perspectives. Stereo-based 3D reconstruction works on the following principle: given two projections of the same point in the universe into two images, the intersection of the two projection rays is used to determine the point's 3D direction. This method is repeated many times to achieve the 3D shape and orientation of the objects in the scene. Since this uses triangulation, the ray equations and complete knowledge of the cameras: parameters such as their relative positions and orientations are needed for this operation.

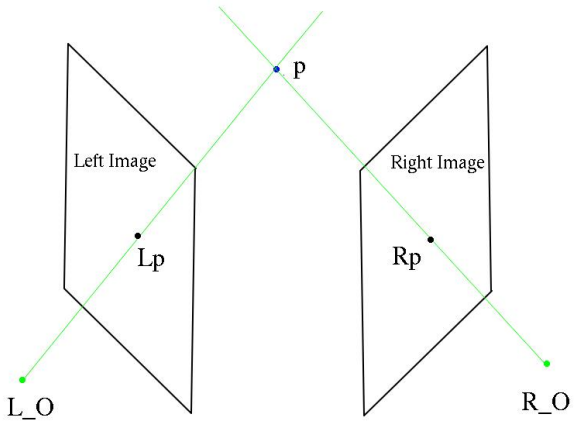


Figure 2.1: Ray intersection
L_O and R_O are the left and right ray projection points respectively. Lp is the point on the left image, and Rp is the point on the right image. The ray intersects at p, which in the virtual space.

2.1.4 Structure from Motion (SfM)

SfM varies from Stereo in the basis that, for passive stereo, the images had to be static, however for SfM, the camera is actually in motion.

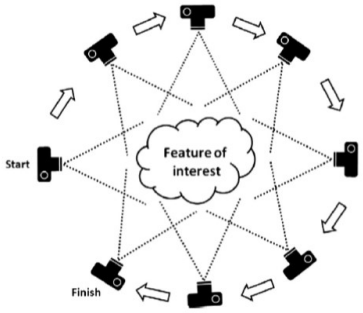


Figure 2.2: Structure from Motion : the camera revolves around the feature of interest capturing different view-ponts of the feature [SFM:16]

2.1.5 Visual Hull - Shape from Silhouette

The Visual Hull is a 3D reconstruction principle based on the Shape from Silhouette (SFS) methodology. The basic concept is to combine silhouettes from several images taken from different angles to create a 3D representation of an object. Separate camera views project both of these silhouettes as a cone, and the intersection of each of these cones is a representation of the actual object's form. In certain ways, the visual hull carves out areas of space where the object is not present.

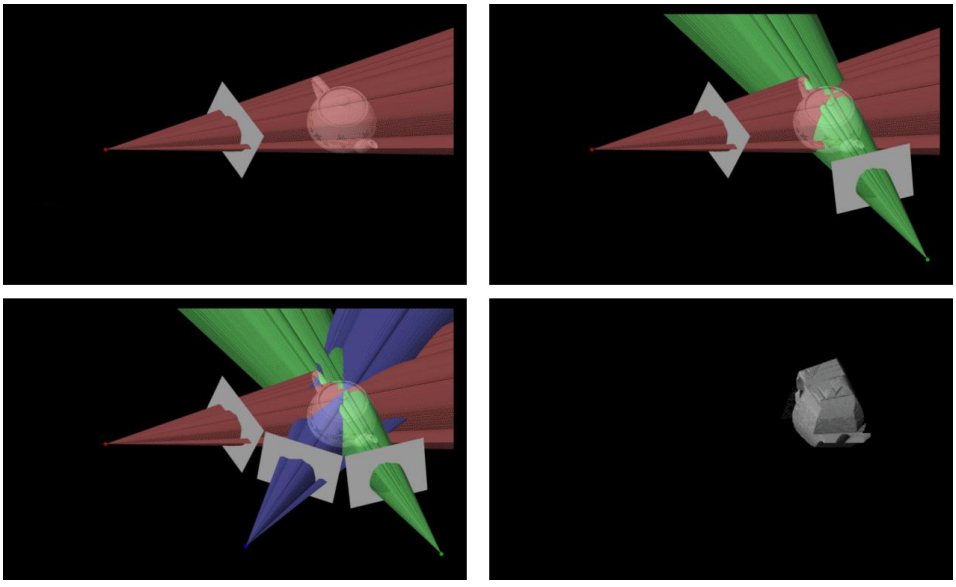


Figure 2.3: Visual Hull. [Moons:2009] Four images showing the visual hull forming sequentially starting from top left image.

The first photo on the top left, shows a single projection of a singular visual cone of the tea-pot. For the next two images, two more view-points are taken and similarly two more visual cones are projected on the tea-pot silhouettes. The intersection of these back projections forms the object's visual hull. As further views are taken, the visual hull achieves its true state, but cavities not seen in the silhouettes are not restored.

2.1.6 Advantages of Visual Hull

1. The silhouette equation is easy to apply when we assume an enclosed environment with regulated/ambient conditions, such as static light and static cameras.

2. The SFS-algorithm is simple to implement, particularly when compared to other shape estimation techniques such as multi-baseline stereo.

3. The object is always contained within a visual hull. The SFS construction provides an upper bound of the real object's shape in comparison to a lower bound, which is a major advantage for obstacle avoidance in robotics or navigation visibility analysis.

Algorithm that object is contained within the volume produced

Assume that a collection of R reference views is used to display an initial 3D object. The silhouette s_r is present in each reference view r, with the interior pixels hidden by the object. For view r, all the rays that start at the image's viewpoint p_r and pass through these interior points on the image plane determine the cone-like volume vh_r . The real object is guaranteed to be included in vh_r. This statement remains true for all r, meaning that the object must be contained within the volume $vh_R = \cap_{r \in R} vh_r$ [Cheung03].

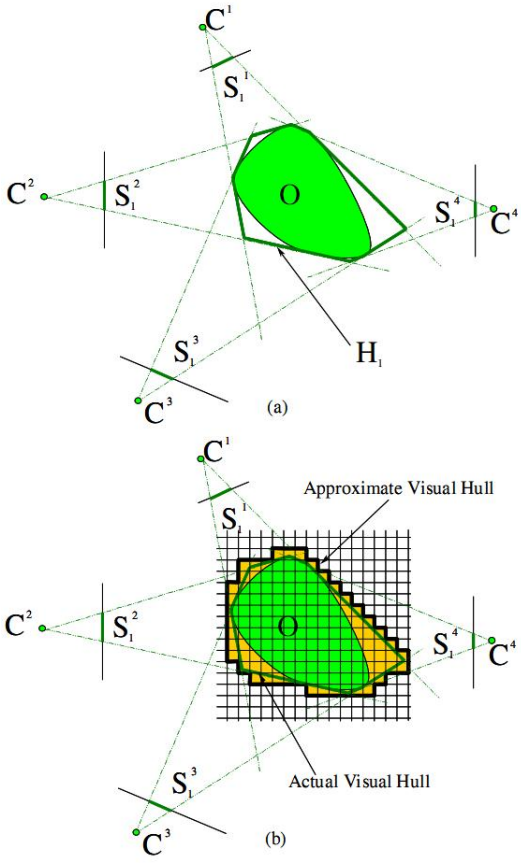


Figure 2.4: Figure taken from Cheung, 2003 [Cheung03] showing volume enclosed by ray projections.

Figure (a) shows four perspectives C^1, C^2, C^3 and C^4, each with a unique viewpoint on the target O and hence separate silhouettes S_1^1,S_1^2,S_1^3, and S_1^4.The projected silhouettes converge to shape the Visual Hull H_1. The difference between an object's Visual Hull and its approximation is shown in Figure (b). We can see that the yellow region is unwanted or yet to carved out space remaining. As claimed earlier, if we take $R \to \infty$, then vh_R converges to a form known as the visual hull vh of the original geometry and contains all possible views. Since concave surface regions can never be separated from silhouette details alone, the visual hull is not guaranteed to be the same as the original object.

2.2 Further look into constructing Visual Hull

2.2.1 Volume Carving

Volume carving is a technique for converting silhouette contours into visual hulls. Each camera takes a picture of a clear part of an object that can be used to make a contour. As projecting onto the image plane, this contour encloses a collection of pixels known as the object's silhouette. Volume carving uses the silhouettes of figures from various angles to ensure consistency. Since we know the object is contained within the intersection of its visual cones, we begin by specifying a working volume inside which we know the object is located.
This volume is subdivided into small units known as voxels, creating a voxel grid. A cross-sectional view of this can be seen in the figure b of Figure 2.4. Each voxel in the voxel grid is projected onto each of the views. A voxel in a vision is discarded if it is not contained by the silhouette. As a result, at the end of the volume carving algorithm, we are left with the voxels enclosed within the visual hull.

2.2.2 Volume Based Approach

Another method to obtain visual hulls is a volume based approach which is very fast and simple, where the virtual space is made up of voxels v. A cuboid of voxels is considered, so there are total of $v = 1..N^3$ voxels.
An Algorithm from [Schneider 2014] is given:

1. Divide the space of interest into $N \times N \times N$ discrete voxels v_n, $n = 1, \ldots, N^3$.
2. Initialize all the N^3 voxels as inside voxels.
3. For $n = 1$ to N^3 {
   For $k = 1$ to K {
   (a) Project ν_n into the k^{th} image plane by the projection function $\prod^k()$;
   (b) If the projected area $\prod^k(\nu_n)$ lies completely outside S_j^k, then classify ν as $outside$ voxel;
   }
   }
4. The Visual Hull H_j is approximated by the union of all the inside voxels.

The final shape is depicted in this version by 3D volume voxels. The structure was then split into two parts, which we called inside and outside depending on whether they were in the visual cones.

Disadvantage of volume based

The resulting image is considerably greater than the actual object shape, it should only be used in applications where an approximation is used.[Cheung:03]

2.2.3 CSG Rendering

Another method by which Visual Hulls are constructed is by the use of Constructive Solid Geometry rendering. Ray tracing can be used to render an object marked by a tree of CSG operations without explicitly computing the resulting solid [Roth82]. This is obtained by looking at each ray individually and computing the interval along the ray occupied by each object, again making use of the principle of intersection, in this case, the intersection of rays in 1D. For each axis, a particular axis is chosen over which the CSG operations will then be carried out over the interval sets . The computation of a 3D ray-solid intersection is needed for this process. Due to the fixed cross section of the cone, 3D ray intersections can be simplified to 2D ray intersections.

Calculation

Each viewing ray's intersection with the visual hull is calculated given a desired view. Since computing a visual hull only requires intersection operations, the CSG calculations can be done in any order. Furthermore, in the sense of the visual hull, every CSG primitive is a projective visual cone.

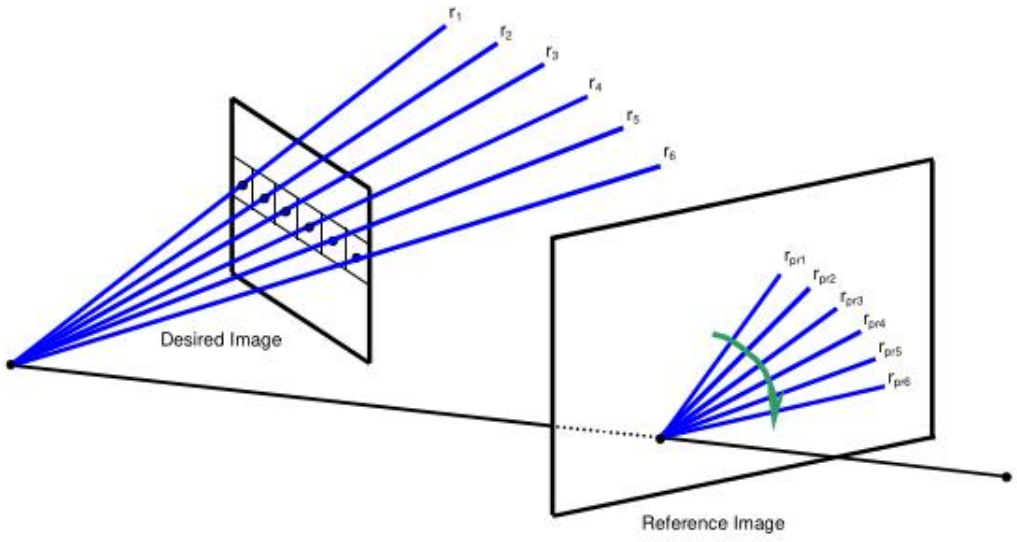


Figure 2.5: Ray projection on desired image and reference image [Chen95], the r's refer to individual rays.

A scanline's pixels in the desired image trace a pencil of line segments in the reference image. An orderly scanline traversal can sweep out these segments such that their slope across the epipole differs linearly.

According to Figure2.5,

1. On a reference image, a 3D viewing ray is projected.
2. The projected ray's intersection with the 2D silhouette is performed. These intersections result in a list of intervals around the ray that are interior to the cone's cross-section.
3. Each interval is then elevated back into 3D using a simple projective mapping and intersected with the results of the ray-cone intersections of other references.

Algorithm to compute by

```
IBVHisect (intervalImage &d, refImList R){
  for each referenceImage r in R
    computeSilhouetteEdges (r)
  for each pixel p in desiredImage d do
    p.intervals =  {0..inf}
  for each referenceImage r in R
    for each scanline s in d
      for each pixel p in s
        ray3D  ry3 = compute3Dray(p,d.camInfo)
        lineSegment2D l2 = project3Dray(ry3,r.camInfo)
        intervals int2D = calcIntervals(l2,r.silEdges)
        intervals int3D = liftIntervals(int2D,r.camInfo,ry3)
        p.intervals = p.intervals ISECT int3D
}
```

Figure 2.6: From [Chen95], computing intersection of rays.

This is the algorithm to compute ray-intersection.

1. n is the number of pixels in a scanline
2. d is the number of pixels in an image
3. k is the number of references
4. l is the average of the intersection of the rays with silhouette

The complexity of the running time for all the intersections for a desired view is $O(lkn^3)$.

2.2.4 Image Based

In the paper from [Chen95], they create 3D perspective viewing effects using real-time image processing. Cylindrical panoramas were used as the photographs. Since each image includes all of the details needed to look around in 360 degrees, the panoramas are orientation independent. A series of these images may be linked together to create a walkthrough chain.

A camera pointed at the object's centre and orbiting in both the longitude and latitude directions in about 10-degree intervals is used to image it. This method generates hundreds of frames, one for each of the possible viewing directions. In dynamic playback, the frames are contained in a 2D sequence that is indexed by two rotational parameters.

2.3 Summary

We have gone through how to acquire the 3D volume. We have looked at the methods in order to do this. We have also looked at some of the advantages regarding few methods as to how it can be beneficial to us, in terms of computational power, and also use case wise.
Then we have further investigated the methods for constructing visual hulls. We have gone through volume carving, volumetric approaches, CSG rendering and image-based options.
Most feasible for me to do, seemed to be the volume carving one. It is fast and efficient, has a much tighter convex hull around the object than volumetric approaches. And compared to CSG renderings and image-based, it is much faster and simpler.
On the other hand, even though image-based visual hulls and CSG renderings do tend to produce much more accurate photo-consistent visual hulls, it is much more complex to implement, regarding that more work needs to be done on the algorithm to generate the ray-intersections and camera parameters.

Chapter 3

Modelling the Data

Here we learn to process the input dataset that we feed into our program to generate the visual hulls. In this chapter, we also see segmentation methods since we have chosen to use the volume carving method and choose a suitable method applicable in our case.

3.1 The raw datasets - Temple

For our input, we take a toy temple as our input model. To make silhouettes, the background has been intentionally kept black in colour. These were taken after rotating the temple toy on a rotating table under ambient light focusing solely on the toy.

Figure 3.1: 16 images of the toy model depicting different view-points, the pictures are rotated, this is the raw format of the data we received. The dataset is produced and provided by Steve Seitz, James Diebel, Daniel Scharstein and Rick Szeliski of Univesity of Washington, Stanford University, Middlebury College and Microsoft Research, respectively.

3.2 Silhouettes - Segmentation Methods

Segmentation is the process of dividing an image into regions that correspond to objects. All pixels in the entire region of the image shares a common property. The intensity is the most basic property that pixels can share.
There are three different categories of segmentation methods are available:

1. Intensity-based segmentation - thresholding
2. Edge-based segmentation
3. Region-based segmentation

Thresholding is basically the distinction between light and dark areas.
Some assumptions are made about thresholding:

1. The different regions being classified have different intensity values
2. Among the individual regions, which represent a particular object or to say a common series of pixels have similar intensity values. Example:

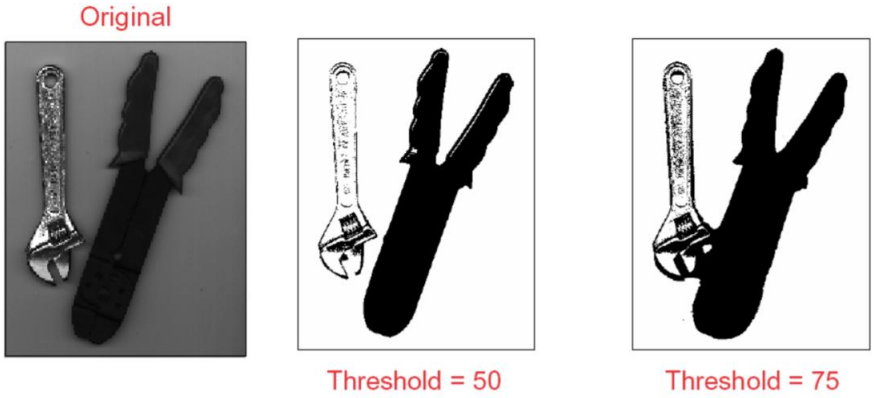


Figure 3.2: Thresholding [UoV2009], with higher threshold values, separation of background and object of interest is made more distinct.

3.2.1 Intensity-based segmentation

The process of separating one or more regions of significance in an image from regions that do not hold important information (in our case pixels) is known as segmentation. The fundamental premise for intensity-based strategies is that pixels belonging to features of interest have a different set of intensity values than the range of intensity values of background pixels. Background refers to regions that do not hold important information, such as in our case, that would be the black pixels in temple datasets. We also assume that background intensity values are lower than feature intensity values in our case.

If we conclude that the distribution of feature pixels and background pixels is roughly Gaussian, we get a distinctive intensity distribution of two peaks in the histogram. The distribution is bimodal, as we get two modes in total, one for the background and one for the feature.
Because of the additive noise and intensity non-linearities of the features, intensities are usually distributed across the modal values.
The most basic solution to segmentation will be to use an acceptable intensity threshold. All pixels with values greater than the threshold value are classified as feature pixels, and all pixels with values less than the threshold value are classified as background pixels.
This is displayed in the equation below:

$$I_T = \begin{cases} 1 & \text{for } I(x,y) \geq T \\ 0 & \text{for } I(x,y) < T \end{cases} \tag{3.1}$$

This equation is used to create an image. In this image, according to the equation given, $I(x, y)$ are the original image pixels, and $I_T(x, y)$ are the image pixels after thresholding was done. It is known as a binary image since it contains only two values: 1 for feature pixels and 0 for background pixels.
Since each location (x,y) of the original image has a value of 1 in the mask if it is a feature pixel, it can be used as a mask.

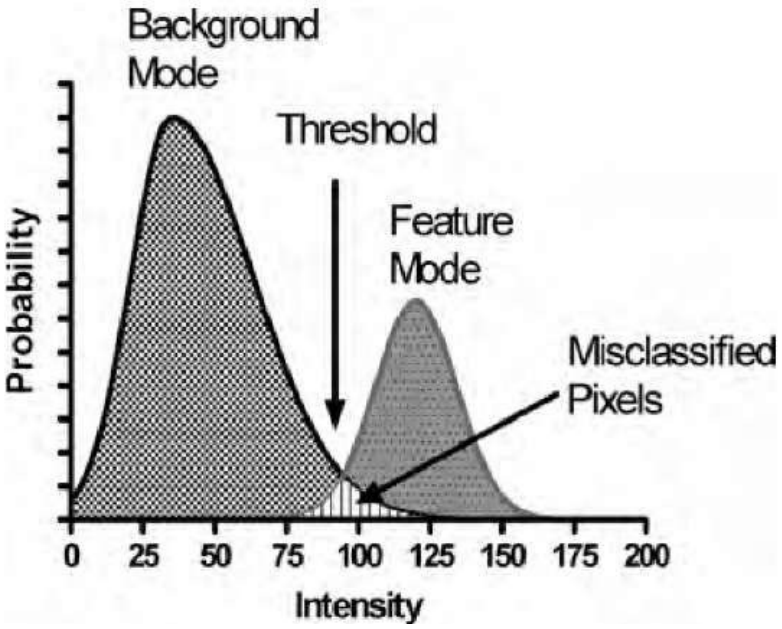


Figure 3.3: A Bimodal Histogram [WWH]

In general, background pixels are darker than function pixels. According to the figure 3.3, the x-axis is based on intensity, with white being 255 and black being 0. However, the intensities varied due to noise and intensity nonlinearities, which is why the bell curves for both the modes are rather spread out. With an appropriate threshold, intensity-based separation may be performed, but certain pixels are misclassified.

For this project, Intensity based segmentation method is chosen for the following reasons:

Advantages of Intensity based segmentation thresholding

1. It is simple to implement
2. If repeated on similar images, it is rendered very fast
3. This method is appropriate for images with controlled lighting

3.2.2 Finding the threshold value

The intensity histogram is carried out on a sample image of the temple dataset and we get the following bimodal distribution.

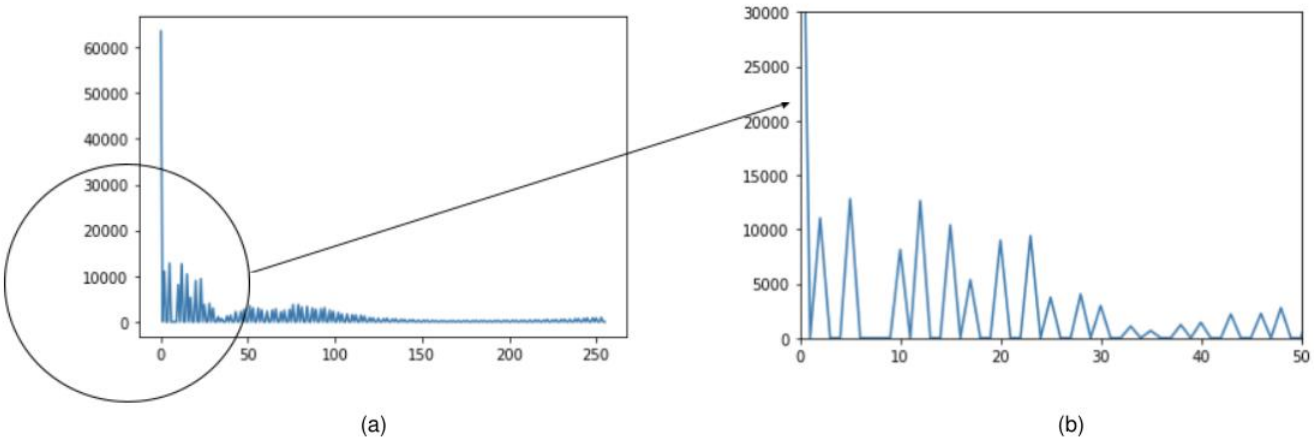


Figure 3.4: Bimodal distribution of a single temple image

In Figure 3.4 (a), a histogram of the different intensities of pixels in a single image of the temple dataset is plotted.
The the x-axis just as in Figure 3.3, refers to the different intensity values, with white being 255 and black being 0. The y-axis corresponds to the availability of the number of pixels at each intensity values of x-axis.
Since out background is black and majority of the pixels in the image is black, high spike of over 60000 is seen at x = 0.
Figure 3.4 (b) is a zoomed in version to figure out the threshold value. It can be roughly estimated that from part(b) that the spike rises somewhat diminishes somewhere after 20-25. For our program, this has to be divided out of 255, to give us a threshold value among 0 to 1. All spike among the 25 is taken to account for background noise such as the presence of a dark grey colour in the background.

The threshold value for all the 16 images of the temple dataset is taken [Appendix B] and a mean is calculated:

Image no (i):	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Threshold val (t):	20	18	1	4	1	15	19	23	22	11	17	23	0	8	14	13

Therefore the mean threshold:

$$mean = \frac{\sum_{i=1}^{16} t^i}{16} = 13.0625 \tag{3.2}$$

Hence, the threshold calculated is:

$$threshold = \frac{13.0625}{255} = 0.0512 \tag{3.3}$$

3.2.3 Silhouettes of the temple dataset

Silhouettes of the initial 16 images obtained by running segmentation with threshold value of 0.0512:

Figure 3.5: 16 silhouettes of the initial dataset formed using the python code [Appendix A].

Chapter 4

Transforming from 3D world space to 2D image space

Figure 4.1: This is taken from ©CoolOpticalIllusions.com
Shows different views of the painting on the road depending on different views, hence such perspective correction is required

Here why we need to transform the 3d world space to our 2d space is discussed. To understand this, we need to look at Viewing transform.

4.1 Viewing transformation

The viewing transformation's aim is to orient the objects in a coordinate system with the center of projection at the origin. This coordinate system is often referred to as camera space. This transformation projects the 3D model to a 2D image plane.
The pipeline for the entire process:

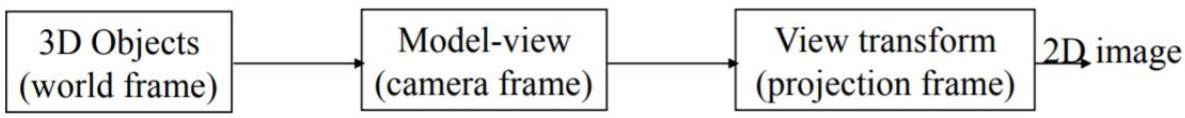


Figure 4.2: The Viewing tranformation pipeline from the Richard Szeliski Computer Vision book

The example below illustrates what we want to achieve after the transformation.

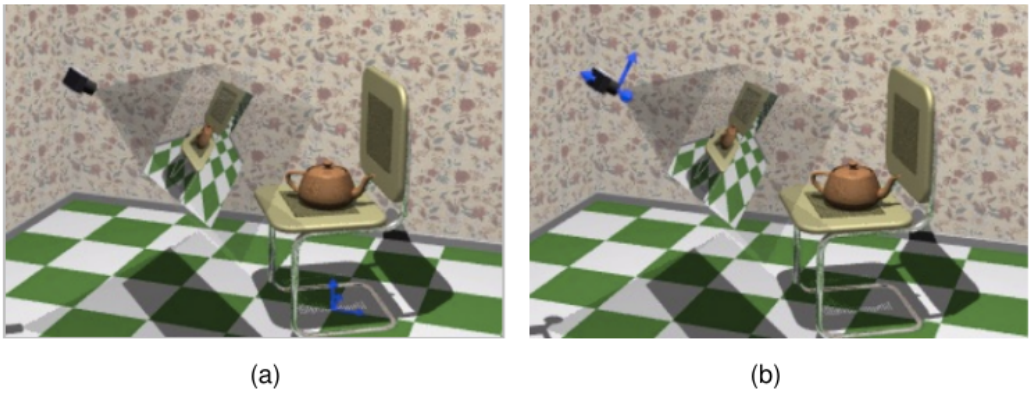


Figure 4.3: After applying viewing transformation [MIT96]

In Figure 4.3 (a), we can see that the blue coordinate axes frame under the chair represents the origin of the world space. After the transformation, in part (b) we see that our coordinate system origin is the point of vision from the camera, that is the camera's origin.
One thing to notice is that, our camera projection is being modelled by this transformation. Camera projection hence is best illustrated by the pin-hole model. This model depicts the nature of projections that are used in computer graphics.

4.1.1 Pinhole Camera Model

Pinhole camera is the simplest type of camera we can find. It is a plain lightproof box with a small hole in the front known as an aperture. This aperture is our pinhole.

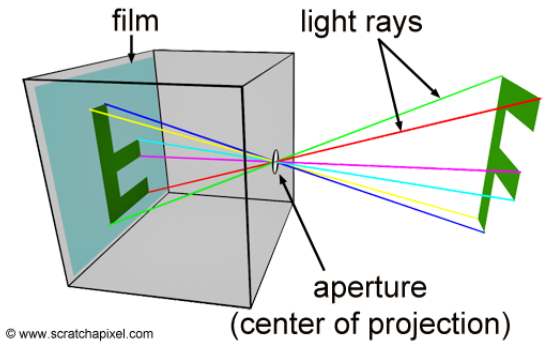


Figure 4.4: pinhole camera ©scratchpixel

Light still travels in a straight line. When light hits an object, it is reflected back into the scene in random ways, but only one of these rays reaches the camera and strikes the film in a single position. The projected light from the target passes through the pinhole, creating an inverted pattern on the imaging surface. The focal length is the distance between the imaging surface and the pinhole. The pinhole is also known as the origin of projection in geometry when all rays approaching the camera converge to it and diverge from it on the other direction.

4.2 Co-ordinate System

From the viewing transform, it is clear that there exists two types of coordinate systems:

1. Camera coordinate system
   (a) Origin: the centre of camera projection
   (b) X-axis: the horizontal direction
   (c) Y-axis: the vertical direction
   (d) Z-axis: the direction in which the camera is pointed
2. World coordinate system
   (a) Origin: It does not depend on any specific entity, it cna be anywhere

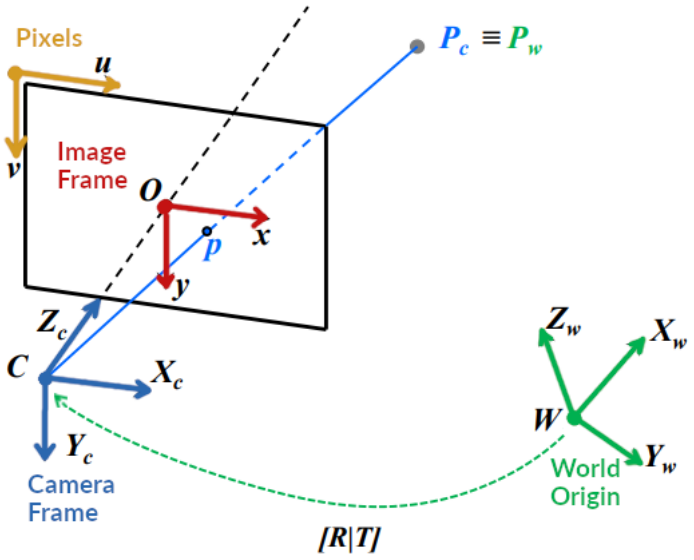


Figure 4.5: Co-ordinate Systems [Co-ord]

Objective

Find pixel coordinates (u,v) of point P_w in the world frame:

1. The world point P_w is converted to camera point P_c through transformation [R,t]
2. P_c is converted to image-plane coordinates (x,y)
3. The co-ordinates (x,y) to pixel coordinates (u,v)

4.2.1 Image frame - Coordinate System

From the [Co-ord] article, we see that for camera models, besides the two co-ordinate systems mentioned, two other systems also exist:

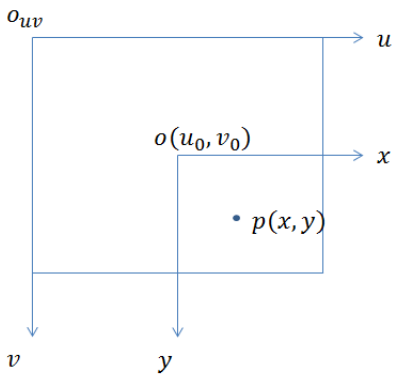


Figure 4.6: Image Frame [Co-ord]

1. **Image pixel**

2. **imaging plane**

The centre of the image plane coordinate system and the pixel coordinate system is the intersection of the camera's optical axis and the imaging plane, which is normally the midpoint of the imaging plane.
The imaging plane frame of reference and the pixel coordinate system differs by the origin and measurement unit, and therefore the two coordinate systems differ by a zoom ratio and a translation of the origin.
According to Figure 4.6, we consider p to have the co-ordinates of (x,y) from Figure 4.6 which are the imaging plane co-ordinates. We also assume that d_x and d_y represent the physical size of each pixel in the image in the imaging plane. This increment or decremental size is basically the zoom ratio.
If the coordinates of the imaging plane's origin in the pixel coordinate scheme are (u_0, v_0), then the following translation theorem applies:

$$\begin{cases} u = & \frac{x}{dx} + u_0 \\ v = & \frac{y}{dy} + v_0 \end{cases} \Longrightarrow \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{dx} & 0 & u_0 \\ 0 & \frac{1}{dy} & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{4.1}$$

4.2.2 Perspective projection

Perspective projection is a method of linear projection that involves projecting three-dimensional structures onto a picture plane. As a consequence, distant objects look smaller than nearby ones.

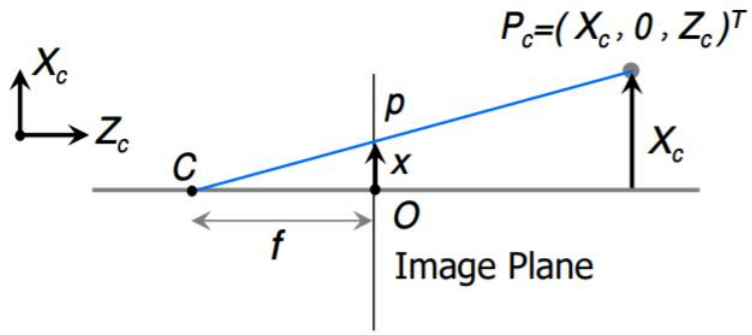


Figure 4.7: Perspective projection

In Figure 4.7, a cross-sectional area along the y-axis is taken from Figure 4.5.
Camera point $P_c = (X_c, 0, Z_c)$ T projects to p=(x, y) onto the image plane. f is the focal length, it is the length from C, the optical centre to O, the pixel plane.
We can also deduce:

$$\frac{x}{f} = \frac{X_c}{Z_c} \Rightarrow x = f\frac{X_c}{Z_c} \tag{4.2}$$

$$\frac{y}{f} = \frac{Y_c}{Z_c} \Rightarrow x = f\frac{Y_c}{Z_c} \tag{4.3}$$

Camera coordinate system

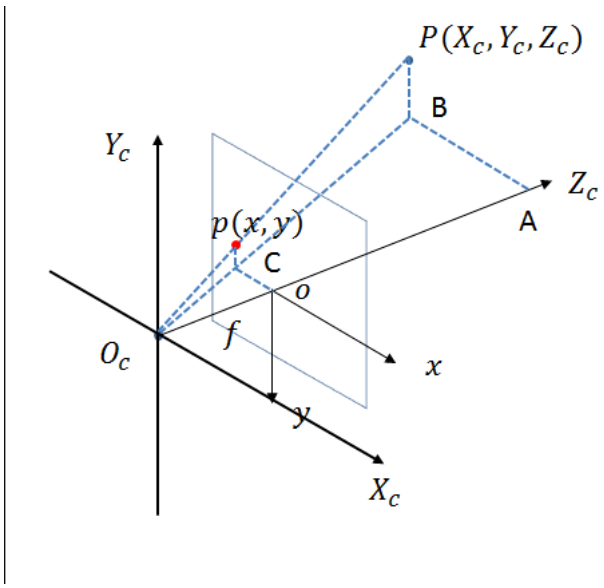


Figure 4.8: Camera Co-ordinate System [Co-ord]

There is a perspective projection between the points on the camera coordinate system and the imaging plane coordinate system have a perspective projection relationship. The coordinate of the

point P in the camera coordinate system corresponding to p is (X_c, Y_c, Z_c) from Figure 4.5 and the figure above, the imaging plane coordinate system and the camera coordinate system have the following translation relationship:

$$\begin{cases} x = & f\frac{X_c}{Z_c} \\ y = & f\frac{Y_c}{Z_c} \end{cases} \Longrightarrow Z_c \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \tag{4.4}$$

World coordinate system

The world coordinate system is the name given to the system where a reference coordinate system in the world to define the camera's and objects' positions are chosen. The rotation matrix R and the translation vector t define the relationship between the camera coordinate system and the world coordinate system.
By considering (X_w, Y_w, Z_w) from Figure 4.5 as the the coordinates of P in the world coordinate system, the conversion relationship between the camera coordinate system and the world coordinate system is given below:

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{4.5}$$

The final conversion

Using all these pieces of transformations, we can use the following equation to convert between the world coordinate system and the pixel coordinate system:

$$Z_c \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{dx} & 0 & u_0 \\ 0 & \frac{1}{dy} & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 & 0 \\ 0 & f_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{4.6}$$

This equation is later used in the implementation of the code in the next chapter.

Chapter 5

Implementation

Here, how visual hulls are created, is explained with its code. The code was initially written by Millin [milin] in Python 2, which I converted to Python 3 and I have extended and worked on it further to use latest algorithms for the 3D plotting, optimised to get slightly faster run-time [Appendix A] and use optimal threshold values in Equation 3.2.

5.0.1 Imports

```
1  import numpy as np
2  import cv2
3  import matplotlib.pyplot as plt
4  from skimage import measure
5  from skimage.filters import threshold_minimum
6  from mpl_toolkits.mplot3d.art3d import Poly3DCollection
7
```

Numpy is used to calculate all the matrix equation and transformations shown in chapter 4.
It is also used to process images used in our dataset.
CV-2 is an open-CV library that we use to calculate threshold values of intensity levels of images.
It is also used to convert images to check for if silhouettes are correctly formed.
Matplot library is used to plot the 3d models of the visual hull created.
The rest of the libraries are image processing utilities.

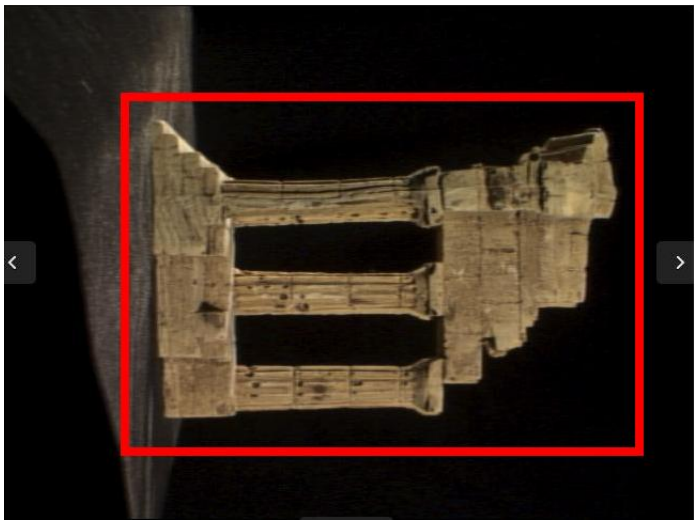
5.0.2 Data class

```
8   class Data:
9       def __init__(self, data_set):
10          self.data_set = data_set
11          self.init_bounding_box()
12          self.load_camera_info()
13
14      def init_bounding_box(self):
15          bounding_boxes = {
16              'templeSR': {
17                  'x': np.array([-0.08, 0.03]),
18                  'y': np.array([0.0, 0.18]),
19                  'z': np.array([-0.02, 0.06])
20              }
21          }
22          self.bounding_box = bounding_boxes[self.data_set]
```

Data class is where all the information for producing models are stored. Here, the images are stored and the silhouettes that are formed are also stored. This class also initialises certain parameters for post-processing.
The init_bounding boxes method initialises the parameters for the plotting of the temple dataset. From lines 17 to 19, it initialises the x,y and z axis for bounding boxes.

Bounding box

It is an imaginary box that acts as a reference point for object detection and generates a collision box for that object.



So as we work on to find any point pixel or silhouettes from the images, the bounding box is the area of region where the the program will look to process information. It will not look for any information outside the box. It will confine its search inside the box, that is the red box in this case in the above picture. The way these x y and z limits of bounding boxes were calculated can be found here: [B-box calc].

5.0.3 Camera Information

```
24  def load_camera_info(self):
25
26      # Dictionary of data_sets with image_paths
27      image_paths = {
28          'templeSR': 'templeSparseRing/',
29          'temple': 'temple/'
30      }
31      string = ""
32
33      # Open file in read mode
34      path = image_paths[self.data_set]
35      fp = open(path+self.data_set+'_par.txt', 'r')
36      num_images = int(fp.readline())
37      self.size = num_images
38
39      P = []            # projection matrices
40      T = []            # camera positions
41      imgs = []         # images
42      arr = []
43      for i in range(num_images):
44          raw_info = fp.readline().split()
45          img_name = raw_info[0]
46          raw_info = list(map(float, raw_info[1:]))
47
48
49          # Projection Matrix of the camera
50          K = np.array(raw_info[0:9]).reshape(3,3)
51
```

```
52          #Rotation Matrix
53          R = np.array(raw_info[9:18]).reshape(3,3)
54
55          #translation matrix
56          t = np.array(raw_info[18:])
57
58          Rt = np.stack((R[:,0], R[:,1], R[:,2], t), axis=1)
59
60          P.append(np.matmul(K, Rt))
61
62          T.append(t)
63
64          # Read Image and store
65          img = plt.imread(path+img_name)
66          imgs.append(img)
67
68          # threshold calculation
69          nimg = cv2.imread(path+img_name,0)
70          arr.append(threshold_minimum(nimg)-50)
71          string += str(threshold_minimum(nimg)-50)+ " & "
72
73      print("arr: ",arr)
74      print("str:",string)
75      print(np.mean(arr))
76
77      self.P = np.array(P)
78      self.T = np.array(T)
79      self.imgs = np.array(imgs)
```

This method mostly loads the camera parameters and synthesises it for further processing. From lines lines 27 to 36 it is just locating the images and appending any information needed. A sample from the par.txt file is given below:

templeSR0001.png 1520.400000 0.000000 302.320000 0.000000 1525.900000 246.870000 0.000000 0.000000 1.000000 0.02187598221295043000 0.98329680886213122000 -0.18068986436368856000 0.99856708067455469000 -0.01266114646423925600 0.05199500709979997700 0.04883878372068499500 -0.18156839221560722000 -0.98216479887691122000 -0.0726637729648 0.0223360353405 0.614604845959

The first 9 numbers make up the parameters for the camera coordinate system described in the previous chapter. That is f_x, f_y, u_0 and v_0, and the rest of the 0s are there as it was a 3 by 3 matrix which is flattened. This is then reshaped back into a 3x3 matrix and assigned to K in line 58.

The 9 numbers after the initial 9 numbers make up the Rotation matrix, since this is flattened, this is again reshaped into 3x3 matrix and assigned into R in line 53.

The rest of the numbers, that is the last three numbers all make up the translation vector assigned to T in line 56.
In line 58, by Rt, the R is stacked upon and appended with t to make up a 3x4 matrix. Example for first image is given below:

```
Rt [[ 0.02187598  0.98329681 -0.18068986 -0.07266377]
 [ 0.99856708 -0.01266115  0.05199501  0.02233604]
 [ 0.04883878 -0.18156839 -0.9821648   0.61460485]]
```

The rest of the code is storing the matrices in arrays for later use.

5.0.4 Segment class

```
98  class ThresholdSegment:
99      i = 0
100     def __init__(self, image, threshold=0.0512):
101         self.image = image
102         self.threshold = threshold
103         ThresholdSegment.i += 1
104
105
106     def run(self):
107         sil = np.zeros((self.image.shape[0], self.image.shape[1]))
108         sil[np.where(self.image > self.threshold)[:2]] = 1
109         return sil
110
111 temple.init_silhouttes(ThresholdSegment)
```

```
81      def sample_image(self):
82          return self.imgs[np.random.randint(self.size)]
83
84      def init_silhouttes(self, Segment):
85          sils = []        # silhouettes
86
87          for img in self.imgs:
88              # create silhouttes using segmentation method
89              sil = Segment(img).run()
90              sils.append(sil)
91
92          self.sils = np.array(sils)
```

Threshold segment class is where the intensity based segmentation thresholding as mentioned in the Silhouettes chapter is implemented here. The threshold of 0.0512 as calculated is chosen here.

5.0.5 Voxel class

Voxel

In 3D space, a voxel represents a value on a regular grid. If we have a cuboid, then if we split the cuboid into the smallest sections, then those smallest sections (boxes) are known as voxels.
A visualization of a voxel is given below.

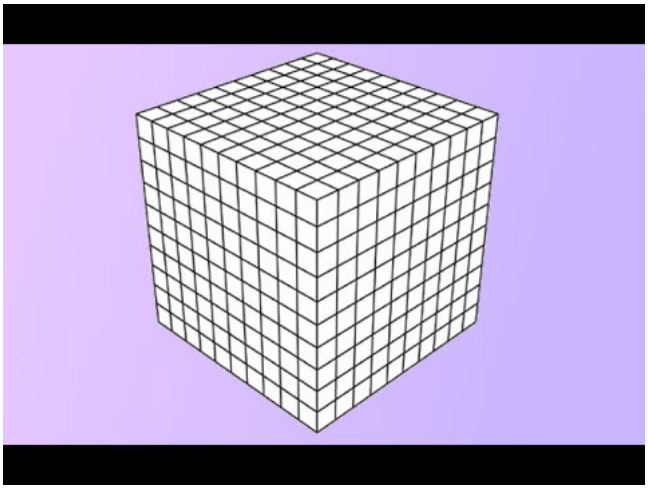


Figure 5.1: Voxel from Unity3D

Each if those boxes are the voxels being mentioned. The entire box is what the initial model of the visual hull would look like.
We try to project the silhouette information into these voxels, those initialised are given a value of 1 and those not are given 0. 1 is given to those voxels which lie within the silhouette's conical volume projection mentioned in the background sections, and then with more silhouettes' projections, out of bounds voxels are carved out.

```
113  class Voxel:
114      def __init__(self, args):
115          self.x = args[0]
116          self.y = args[1]
117          self.z = args[2]
118          self.val = args[3]
119
120      # Prints details about the voxel
121      def log(self):
122          print ("\tx: ", self.x)
123          print ("\ty: ", self.y)
124          print ("\tz: ", self.z)
125          print ("\tval: ", self.val)
126
127      # Returns homogeneous coordinates of the voxel
128      def homo(self):
129          return np.array([self.x, self.y, self.z, 1]).reshape(4, 1)
130
```

From lines 115 to 118, the x,y and z coordinates of the voxels in the 3D space are determined and stored. The width, height and length of the voxels are determined in the main run section of the code. That is, we can change the size of the voxels, we can make it larger, or we can make it smaller to make the model more smooth and photo-consistent.

5.0.6 Visual Hull class

This is where all the processing is done.

```
131  class VisualHull:
132      def __init__(self,
133                   data,
134                   voxel_size=np.array([0.001, 0.001, 0.001])):
135
136          self.data = data
137          self.voxel_size = voxel_size
138
139          self.sils = self.data.sils
140          self.P = self.data.P
141
142          self.img_height = self.data.imgs.shape[1]
143          self.img_width = self.data.imgs.shape[2]
```

```
145      def initialize_voxels(self, xlim, ylim, zlim):
146
147          # Determine shape of visual hull grid
148          hull_shape = np.zeros(3)
149          # print("xlim: ",xlim,"      |voxel size:  ",self.voxel_size)
150          hull_shape[0] = abs(xlim[1] - xlim[0])/self.voxel_size[0]
151          hull_shape[1] = abs(ylim[1] - ylim[0])/self.voxel_size[1]
152          hull_shape[2] = abs(zlim[1] - zlim[0])/self.voxel_size[2]
153          self.hull_shape = hull_shape + np.ones(3)
154
155          # Compute total voxels using grid shape
156          self.total_voxels = np.prod(self.hull_shape)
157          # print(self.total_voxels)
158
159          # List storing (x, y, z, val) of each voxel
160          self.voxels2D_list = np.ones((int(self.total_voxels), 4), float)
161
162          # Compute increment and decrement step direction
163          sx = xlim[0]
164          ex = xlim[1]
165          sy = ylim[0]
166          ey = ylim[1]
167          sz = zlim[0]
168          ez = zlim[1]
169
170
171          x_linspace = np.linspace(sx, ex, int(self.hull_shape[0]))
172          y_linspace = np.linspace(sy, ey, int(self.hull_shape[1]))
173          z_linspace = np.linspace(sz, ez, int(self.hull_shape[2]))
```

In lines 150 to 153, the hull shape is determined. The dimensions of the model is assigned here. For each of the axis, the limits of each axis depending on the bounding box is taken, these two extreme ends are then subtracted from each other to get the length and then this is divided by the voxel width so that we can get the number of possible amount of voxels that can be fitted along each axis inside the initial visual hull cuboid.

In line 156, these divisions in each axis is multiplied to give the total number of voxels.
The number of voxels stand out to be 63568 according to line 156. To note that this is obtained after adding an additional 1 to the hull shape in line 153 so as to include any decimals that were formed after dividing it by the voxel width.

In line 160, for the number of voxels being worked on, an array is made, where for each voxel, 4 values are allocated, the first three being x, y and z coordinates values and the 4th value is to contain a 1 or 0, that is whether it is initialised or not.

In lines 171 to 173, we are making a flattened linear space of the x,y and z axis.
To see what is happening, lets consider the line 171.

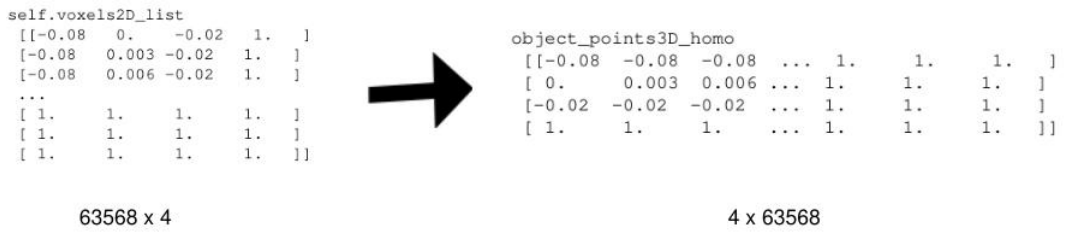
We take the extreme ends as our limits, and our hull shape of that particular axis.
Then the np.linspace converts the information into an array of the size of the hull's shape, with its starting value being the left limit value and the last value being the furthest limit value, in this case: ex, and all the values of the array are equally spaced.
These are made in order to localize and initiate the x,y and z values of the voxels.

```
175         # Initialize voxel list
176         l = 0;
177         for z in z_linspace:
178             for x in x_linspace:
179                 for y in y_linspace:
180                     self.voxels2D_list[l, 0:3] = np.array([x, y, z])
181                     l = l+1
182         print("l is: ",l)
```

This is exactly what is beign done in lines 177 to 181.
By traversing a 3 loop, all the points of the 3D space coordinates are covered and assigned to the voxel2D_list. This is assigned in a linear fashion as all the voxels are stored in a 1D array.

```
184     def create_visual_hull(self):
185
186         # Homogeneous 3D world coordinates
187         object_points3D_homo = np.transpose(self.voxels2D_list)
188
189         # Iterate through each camera position
190         for i in range(self.P.shape[0]):
191             print (i+1,end= " ")
192
193             # Homogeneous image coordinate for camera/image i
194             points2D_homo = np.matmul(self.P[i], object_points3D_homo)
195
196             # Image coordinates
197             points2D = np.divide(points2D_homo[0:2], points2D_homo[2]);
```

In line 187, the matrix are transposed to get homogeneous coordinates. These allow common operations such as translation, rotation and other matrix operations. That is the initial voxel2D_list is a 63568 by 4 matrix, which after transposing becomes 4 by 63568 matrix, now acceptable for matrix multiplication, which is what we desire according to our transforming from real-world to pixel coordinate section.



From line 190, we enter a loop, where we iterate through every picture, or to say we go through each camera position, and we multiply the respective projection matrix of that image with the homogeneous coordinates to get 2D homogeneous coordinates.

In line 197, we divide the coordinates with the z value.
This is a perspective correction, so that by dividing it by z, we scale it so that the extreme ends are not zoomed in. This is done to scale. So that the point near the origin is not too close and the point away from the origin is not too far away. Basically this normalises the coordinates.

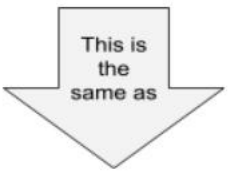
```
199             # Remove points outside image boundary
200             ind_x_lt_0 = np.where(points2D[0] < 0)
201             ind_y_lt_0 = np.where(points2D[1] < 0)
202             ind_x_gt_w = np.where(points2D[0] >= self.img_width)
203             ind_y_gt_h = np.where(points2D[1] >= self.img_height)
204             points2D[:, ind_x_lt_0] = 0
205             points2D[:, ind_y_lt_0] = 0
206             points2D[:, ind_x_gt_w] = 0
207             points2D[:, ind_y_gt_h] = 0
208
209             # Make voxel value 0 for those not being projected
210             ind = np.ravel_multi_index(
211                 (points2D[1,:].astype(int), points2D[0,:].astype(int)),
212                 dims=(self.img_height, self.img_width), order='C')
213             cur_sil = self.data.sils[i]
214             img_val = cur_sil.ravel()[ind]
215             zero_ind = np.where(img_val == 0)
216
217             # Update list of voxels
218             self.voxels2D_list[zero_ind, 3] = 0
219             print(len(self.voxels2D_list))
```

From lines 200 to 207, we remove points that are extra, that is outside of our image boundary. Now what it means is that, since we have created extra voxels due to accommodating for the decimal values in the earlier step, while trying to do that we have created more voxels than needed. These are removed by assigning them 0 as they wont be initiated.

From lines 210 to 215, a lot is happening. It is probably the most important bit of this entire project.

How np.ravel_multi_index works

```
>>> import numpy
>>> numpy.ravel_multi_index([[3,6,6],[4,5,1]], (7,6))
array([22, 41, 37])
```




```
>>> print(numpy.arange(7*6).reshape(7,6))
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]
 [24 25 26 27 28 29]
 [30 31 32 33 34 35]
 [36 37 38 39 40 41]]
>>> numpy.arange(7*6).reshape(7,6)[[3,6,6],[4,5,1]]
array([22, 41, 37])
```

In line 210, we get the 2d matrix of the image, that is by the row and column of the image and depending on the points2D array rows and column we choose the indices, that is it masks out the points in the image 2d matrix where there is a silhouette and where there is not, and linearly those indices are chosen. It is done in such a fashion so that for further operation, the indices can be chosen starting from (0,0) such as (12,2) and not something as (0.712, -0.812) which starts from (0.0,-0.08), which could be in negative and make it impossible to retrieve any index.

In lines 213 to 214, we flatten the silhouette matrix and select the indices based on the earlier configuration.
This allows us in line 215 to mask out the background by assigning them as 0.
For that particular camera projection, in line 218, the entire model for 3d points which is now flattened, is updated.
To note that since this is a loop, iterated on every picture,so every time new silhouette points are discovered, and new voxels are made to be 0 at every run.
This is a classic example of carving out of volume, that is, we start with a deformed shape and slowly carve it to shape it into our desired model with new points of information.

```
221     def create_grids(self):
222         # Initialize Grids - one with all voxel info, one just with voxel values
223         self.voxel3D = np.zeros([int(self.hull_shape[1]), int(self.hull_shape[0]), int(self.hull_shape[2])], dtype=float)
224 
225         # Assign values in 3D grid
226         l=0
227         for zi in range(self.voxel3D.shape[2]):
228             for xi in range(self.voxel3D.shape[1]):
229                 for yi in range(self.voxel3D.shape[0]):
230 
231                     self.voxel3D[yi, xi, zi] = self.voxels2D_list[l, 3]
232                     l = l+1
```

In create grid method, we just make a loop from lines 227 to 232 to assign all this flattened voxel coordinate information to a 3d matrix array.
This is initiated in line 223 where we make the 3d matrix based on the hull's shape.

```
234     def plot_marching_cube(self):
235         verts, faces = measure.marching_cubes(self.voxel3D.T,level=0)
236 
237         fig = plt.figure()
238         # print("faces",len(faces),"verts",len(verts))
239         ax = fig.gca(projection='3d')
240 
241         # Fancy indexing: `verts[faces]` to generate a collection of triangles
242         mesh = Poly3DCollection(verts[faces])
243         # mesh.set_facecolor('yellow')
244         mesh.set_edgecolor('k')
245         ax.add_collection3d(mesh)
246 
247         ax.set_xlabel("x-axis")
248         ax.set_ylabel("y-axis")
249         ax.set_zlabel("z-axis")
250 
251         ax.set_xlim(0, self.voxel3D.T.shape[0]) # a = 6 (times two for 2nd ellipsoid)
252         ax.set_ylim(0, self.voxel3D.T.shape[1]) # b = 10
253         ax.set_zlim(0, self.voxel3D.T.shape[2]) # c = 16
254 
255         # plt.tight_layout()
256         plt.show()
```

From lines 234 marching cube algorithm is run to construct the 3d model. This is the part that takes in all the information and gives us a visual output for evaluation.

Marching Cube Algorithm

Marching cubes is a straightforward algorithm for generating a triangle mesh from a function f(x, y, z) = k, where k may be any arbitrary value. Iterating ("marching") over a uniform grid of cubes superimposed over an area of the function is how it operates. If the cube's eight vertices are all positive, or all negative, the cube is entirely above or entirely below the surface, and no triangles are emitted.
All the cubes are iterated over and for every iteration, a mesh triangle is appended to the list and the final mesh is the union of all of these triangles. The smaller the cubes (voxels in our case), the smaller the mesh triangles, and thus the closer our approximation would be to the target function.

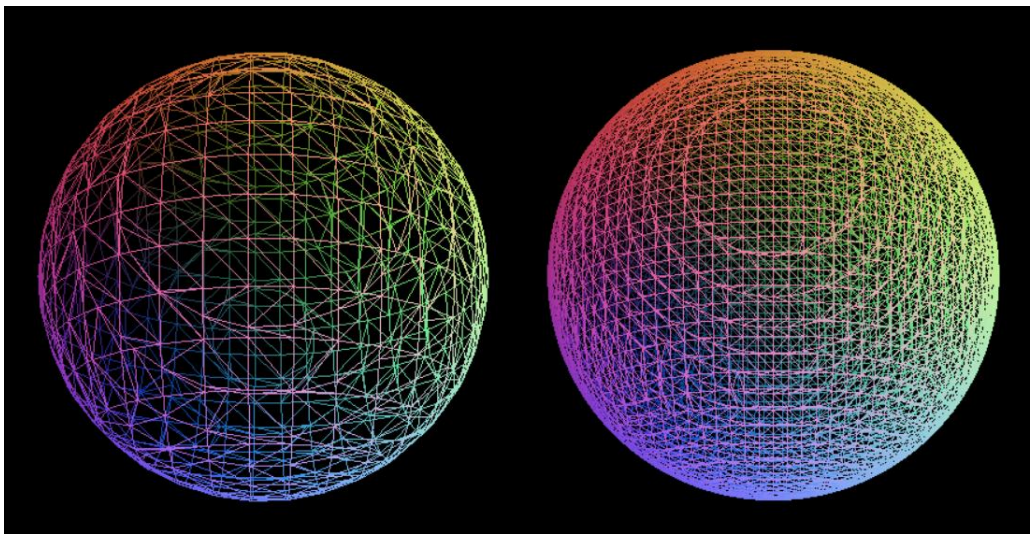


Figure 5.2: Marching cube algorithm run on the function $x^2 + y^2 + z^2 - 1 = 0$. The difference is because of the voxel size, in the right sphere, the cubes used to construct were smaller [March:Algo]

```
258     def run(self):
259         print ("Initializing Voxels ->",end = " ")
260         self.initialize_voxels(
261             self.data.bounding_box['x'],
262             self.data.bounding_box['y'],
263             self.data.bounding_box['z'])
264         print ("Forming Visual Hull ->",end = " ")
265         self.create_visual_hull()
266         print ("-> Creating 3D grids ->",end = " ")
267         self.create_grids()
268         print ("Plotting marching cubes ... ",end = " ")
269         self.plot_marching_cube()
```

```
272 big_voxels = [0.003, 0.003, 0.003]
273 med_voxels = [0.0023, 0.0023, 0.0023]
274 small_voxels = [0.0015, 0.0015, 0.0015]
275 
276 temple_hull_hr = VisualHull(temple, np.array(med_voxels))
277 temple_hull_hr.run()
```

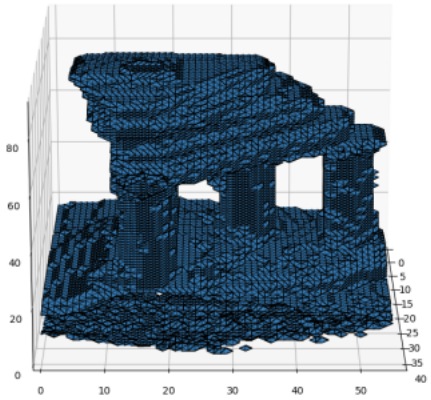
The code from lines 258 to 268 is like the main section (as in C code) of this python code, where everything is initialised and run.
In lines 272 to 274, the voxel sizes are allocated, I chose to use the medium size, to make my plots as smooth as possible at the same time, making the plot clear and visible, since smaller voxels are not clear enough in plots, and big voxels are not accurate enough.

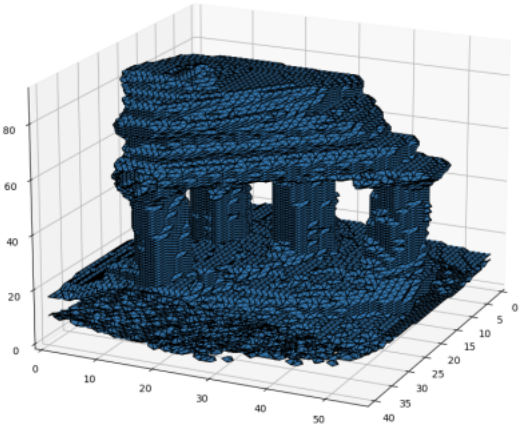
Chapter 6

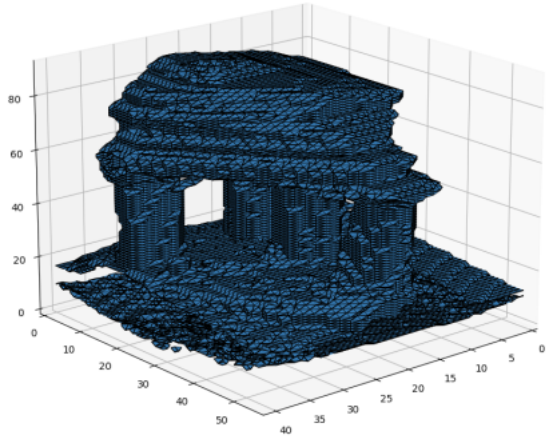
Conclusions

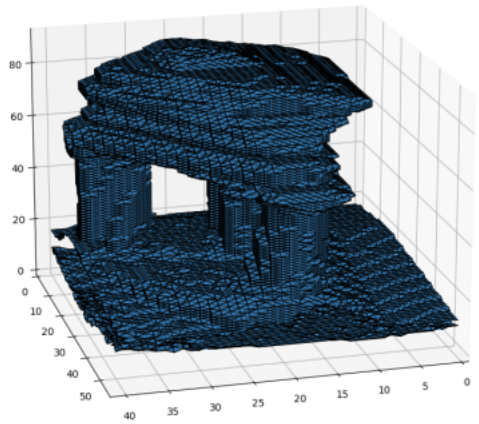
6.1 Achievements

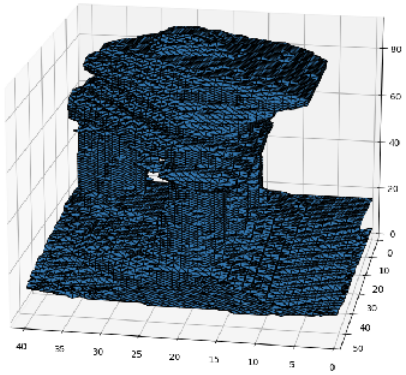
Here I depict the results of the code and everything that I have discussed in this project and worked on about. In the following 16 images, I have tried to show the 3d model from different elevations and horizontal angles, basically by rotating the model I have tried to show different viewpoints.

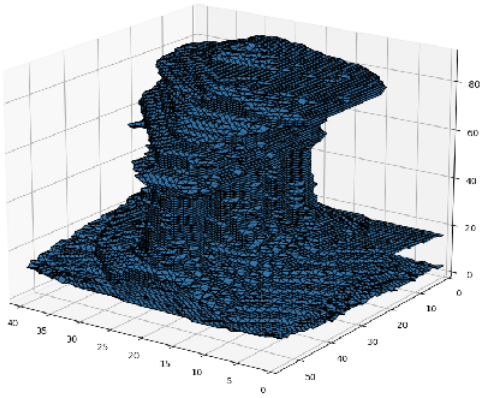


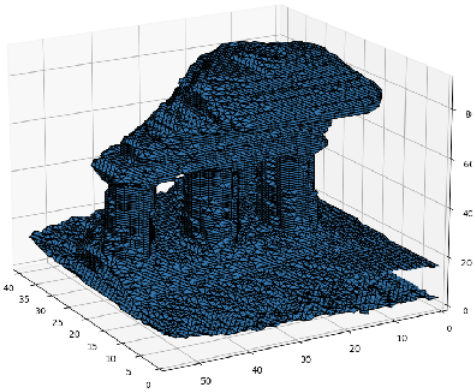


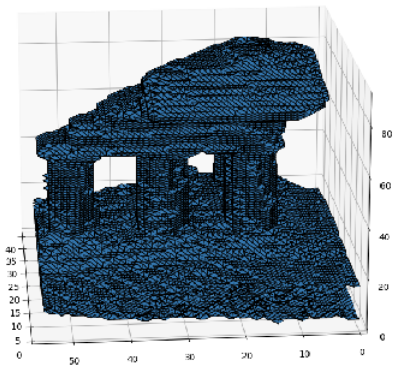


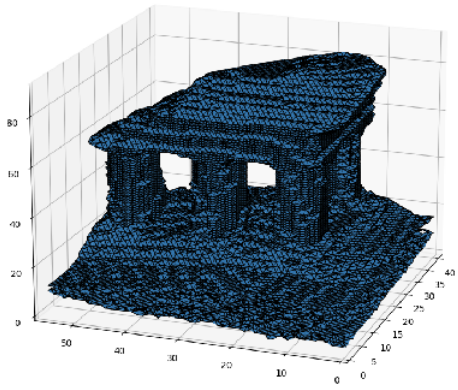


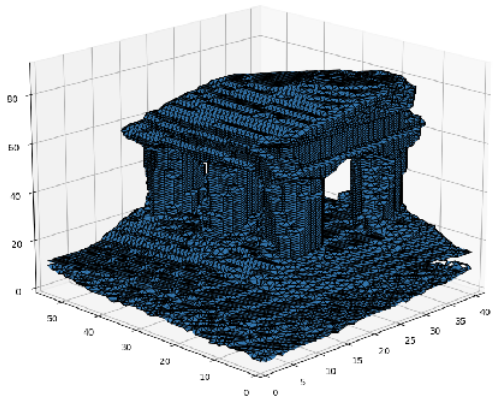


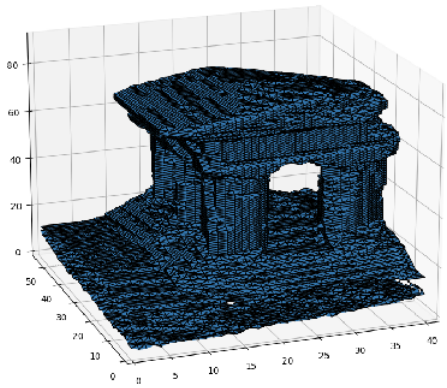


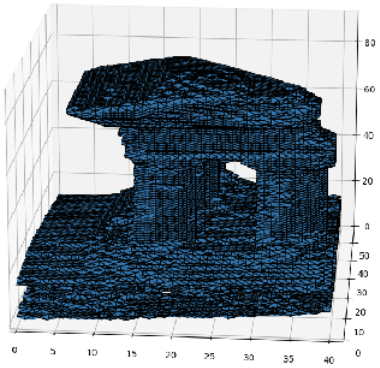


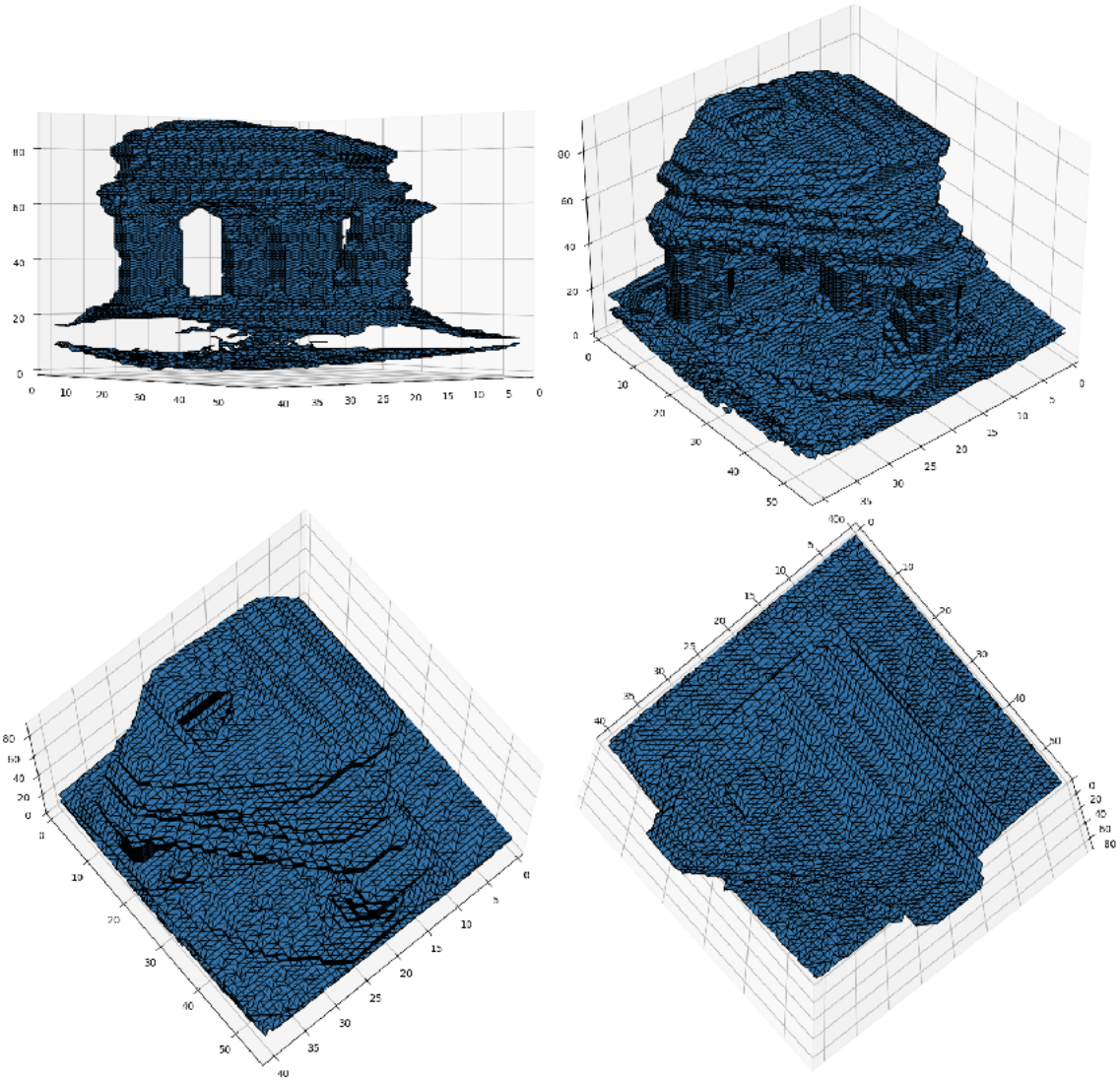












In the last 4 images, I took elevations at 0 degree, and incremented the elevations by 35 degrees, to show the roof of the temple with slight horizontal angle variations.

6.2 Evaluation

Here I try to compare my results with different parameters and show its effect.

6.2.1 Voxel Cube Sizes

As discussed in the marching cube algorithm part, different cube sizes would render different images. This is what I tried experimenting with and got different results:

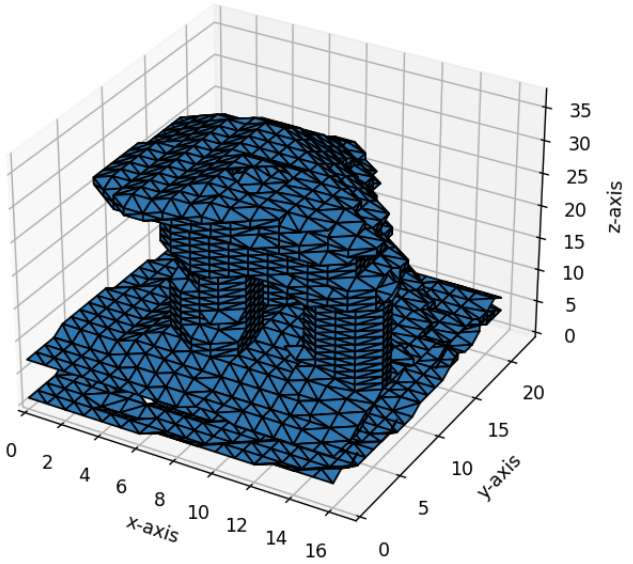


Figure 6.1: Here voxel cubes of size 0.005 is used

Here, since cubes of 0.005^3 were used, the voxels being bigger, resulted in a much deformed shape of the reconstruction of the model as expected. A lot of information and picture details are blended into one another.

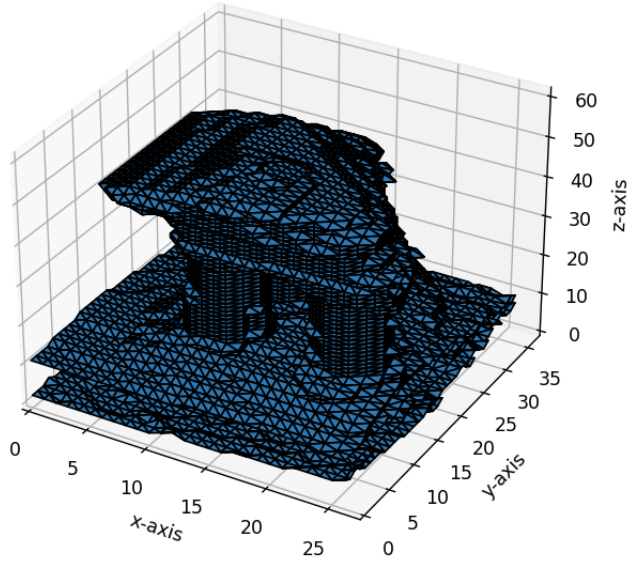


Figure 6.2: Here voxel cubes of size 0.003 is used
With cubes of 0.003^3, we achieve a much more consistent form, here atleast the stairs of the temple model is visible.

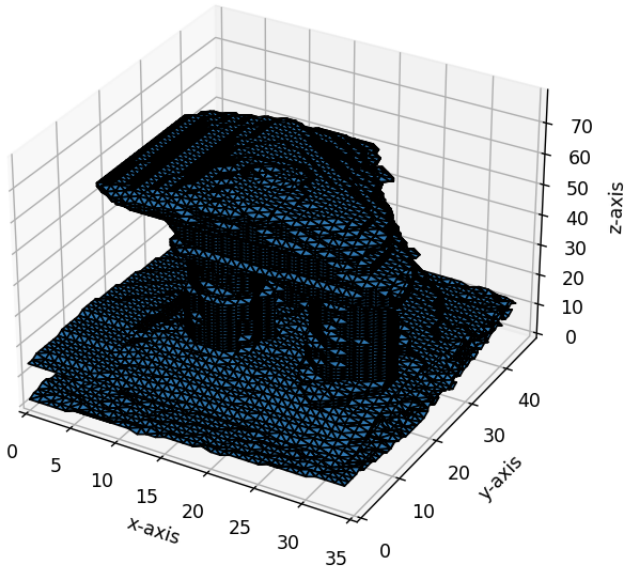


Figure 6.3: Here voxel cubes of size 0.0021 is used

This is the size of cubes, that I have used in my achievement part. Most of the details is seen in this version. The stairs are much clearer, the ridge is visible, the roof plating can also be seen. Even the base, shaft and the capital of the pillar [pillar] is visible. This cube size offers the best visibility in terms of border.
One thing to stress here, is that the border used for the triangle meshes is black in colour and is significant in thickness. This means that the smaller the triangle meshes that is smaller cube sizes available for marching cube algorithm (that is voxel sizes), the triangles would rather look like black pixels due to smaller triangles.

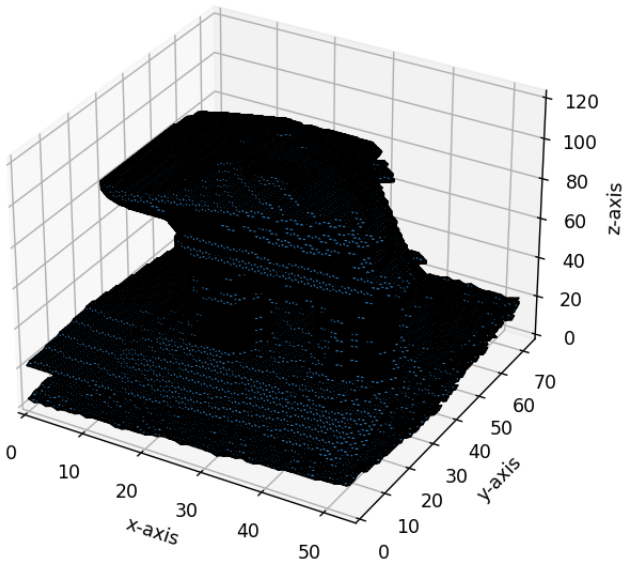


Figure 6.4: Here voxel cubes of size 0.0013 is used

This one has the most detail ingrained into this. However, as the triangle meshes are so small, since cubes are of 0.0013^3, all we can see are the borders enclosed together, forming black dots.
If the borders could be removed and, shadow and lighting effect was added, this would look much clearer and precise. This would be closest to achieve photo consistency.

6.2.2 Threshold Values

I also test my results with different threshold values, to see how it differs the 3d model output.

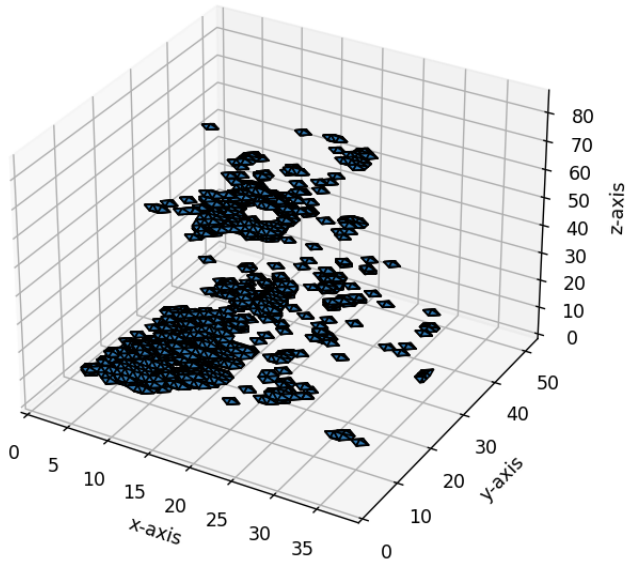


Figure 6.5: Here threshold value of 0.5 is used

Threshold value of 0.5 means a value of 127 out of 255 in the colour range. This means much of the colour information from the images is masked out in the silhouettes formed, resulting in a scattered plot of voxels that highlights on the bright parts of the image.

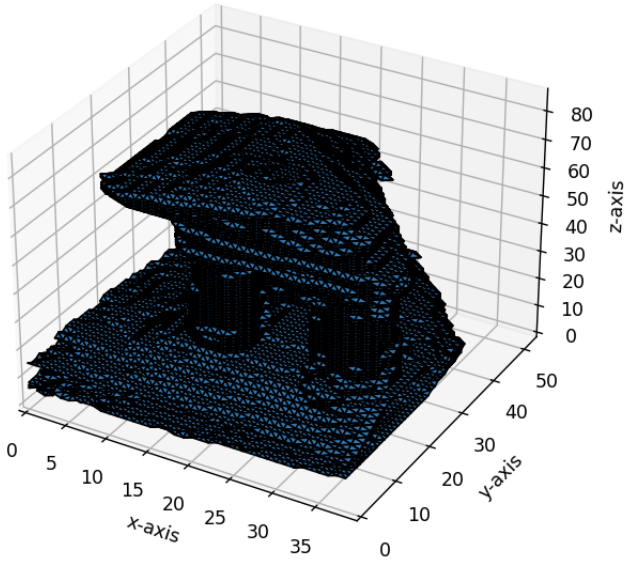


Figure 6.6: Here threshold value of 0.1 is used

A threshold value of 0.1 means, a value of 25 out of 255 in the colour range. This is very close to what we used, and 25 masks much of the grey and black regions. Hence, the picture is accurate since most of the colour regions pertaining to the pixels of the toy temple model is available to be plotted.

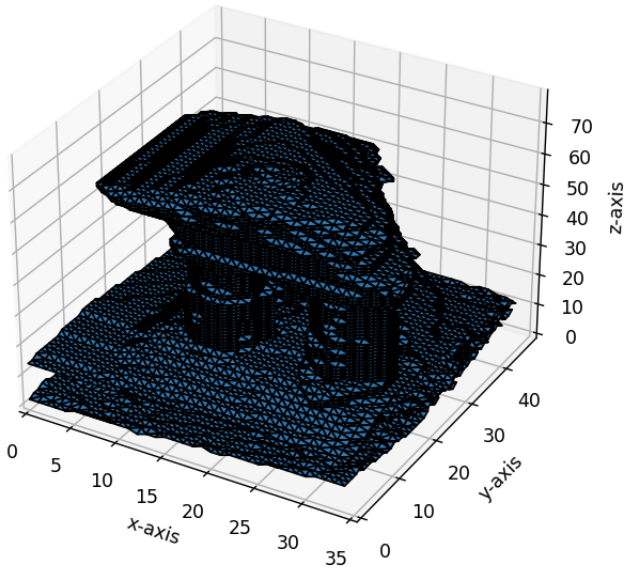


Figure 6.7: Here threshold value of 0.0512 is used

Threshold value of 0.0512 means, a value of 13 out of 255 in the colour range. This is the optimal value that I have found, using threshold minimums of all the images and computing their mean.

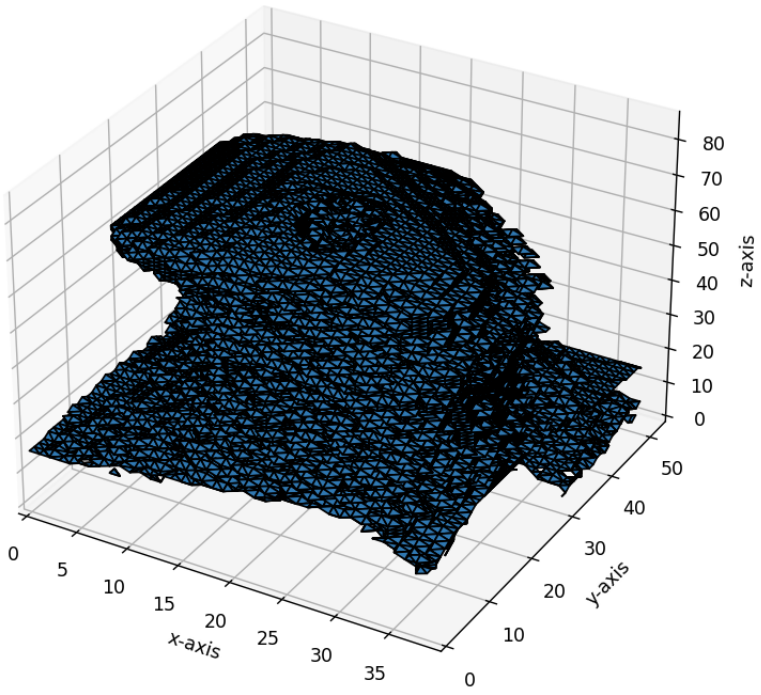


Figure 6.8: Here threshold value of 0.003 is used

Threshold value of 0.003 means, a value of 1 out of 255 in the colour range. This only hides out black but leaves the rest. By hiding out black, I mean it masks out only the RGB value of(0,0,0), that is it masks only true black. So any noise is also considered to be part of the toy temple model, such as RGB(1,1,1). This is why we see unwanted voxels of the background blending with the targeted feature model.

6.3 What has been achieved so far

Starting from knowing almost nothing about python and then being able to write, extend on an existing python codebase and understand it completely has been an accomplishment for me. A wide scope of pre-requisites for Computer vision has been covered and researched for this project. This project can serve as a base point for anyone starting out Computer Vision or 3D model reconstruction. This project contains 3D acquisition methods, reconstruction taxonomy and the math behind all these tasks.
The results achieved are mostly qualitative, rather than quantitative. This is because, at the current stage of this project, no optimisation techniques has been implemented and hence, evaluation of our results relies heavily on the appearance of the models being produced rather than using some equation to verify its accuracy.
For novelty, I would consider the region of colour space to be masked to get the accurate threshold values method that I have tested on. By bundling together a series of spikes of lower colour values together (since our background is black) and using the next point where these initial spike series ends, the mean of which gave out the best model reconstruction results as displayed in the previous section.

6.4 Future Work

This is just the beginning. Visual hull is the first building block for 3d model reconstruction.
If given more opportunity to work on this project,
After this I would like to:

1. Make the model more accurate by running the code on large datasets of say 300 images of the temple model toy covering every view-points possible which would require heavy load graphics card equipped computers to render the models.

2. Add optimisation such as graph cutting algorithms [Sinha:2021], to make the voxels refined and make the model more accurate.

3. Add colour to it, that is voxel colouring [Seitz:1997] to make the model photo-consistent.

4. Add deep learning algorithm to further optimise the model, to denoise the structure and do in-painting [Dmitry:2017]. Infact, with this paper in mind, this project has been done in python in order to use the latest deep learning algorithms in the future.

Bibliography

[Rothganger et al. 2004] Rothganger, F. & Lazebnik, S. & Schmid, C. & Ponce, J.. (2004) Segmenting, modeling, and matching video clips containing multiple moving objects. 914-921. 10.1109/CVPR.2004.1315263.

[Schneider 2014] David C. Schneider Shape from Silhouette Computer Vision, 2014 ISBN : 978-0-387-30771-8

[Cheung03] Kong Man German Cheung Visual Hull Construction, Alignment and Refinement for Human Kinematic Modeling, Motion Tracking and Rendering. PhD thesis, Carnegie Mellon University, 2003.

[Roth82] Roth, S. D. "Ray Casting for Modeling Solids." Computer Graphics and Image Processing, 18 (February 1982), 109-144.

[HAL:112] Bastien Billiot, Frédéric Cointault, Ludovic Journaux, Jean-Claude Simon, Pierre Gouton. 3D image acquisition system based on shape from focus technique. Sensors, MDPI, 2013, 13 (4), pp.5040-5053. ff10.3390/s130405040ff. ffhal-01190112f

[Moons:2009] Theo Moons, Luc Van Gool, and Maarten Vergauwen 3D Reconstruction from Multiple Images Part 1: Principles DOI: 10.1561/0600000007

[SFM:16] Structure From Motion `https://gsp.humboldt.edu/OLM/Courses/GSP_216_Online/lesson8-2/SfM.html`

[Chen95] Chen, S. E. "Quicktime VR – An Image-Based Approach to Virtual Environment Navigation." SIGGRAPH 95

[UoV2009] University of Victoria `https://www.ece.uvic.ca/~aalbu/computer%20vision%202009/Lecture%209.%20Segmentation-Thresholding.pdf`

[WWH] What When How `http://what-when-how.com/biomedical-image-analysis/intensity-based-segmentation-thresholding-biomedical-image-analysis/`

[MIT96] MIT `https://groups.csail.mit.edu/graphics/classes/6.837/F98/Lecture12/viewing.html`

[Co-ord] Nabil MADALI Structure from Motion Stereo vision, Triangulation, Feature Correspondence, Visual SLAM

[Surrey] Surrey `http://info.ee.surrey.ac.uk/Teaching/Courses/eem.cgi/lectures_pdf/lecture3.pdf`

[B-box calc] Bounding Box `https://d2l.ai/chapter_computer-vision/bounding-box.html`

[pillar] Pillar Structure https://www.britannica.com/technology/shaft-architecture

[March:Algo] Stanford Graphics https://graphics.stanford.edu/~mdfisher/MarchingCubes.html#:~:text=Marching%20cubes%20is%20a%20simple,a%20region%20of%20the%20function.

[milin] Milin Visual Hull https://github.com/millingab/multiview-stereo/blob/master/Project.ipynb

[Sinha:2021] Sinha, Sudipta & Pollefeys, Marc & Mcmillan, Advisor & Ponce, Jean & Reader, Henry & Fuchs, Guido & Gerig, & Reader,. (2021). Silhouettes for Calibration and Reconstruction from Multiple Views.

[Seitz:1997] Seitz & Dyer, "Photorealistic Photorealistic Scene Reconstruction by Scene Reconstruction by Voxel Coloring", Coloring", Proc. Computer Vision and Pattern Recognition (CVPR), . Computer Vision and Pattern Recognition (CVPR), 1997, pp. 1067-1073. . 1067-1073.

[Dmitry:2017] Dmitry Ulyanov, Andrea Vedaldi, Victor Lempitsky Deep Image Prior arXiv:1711.10925

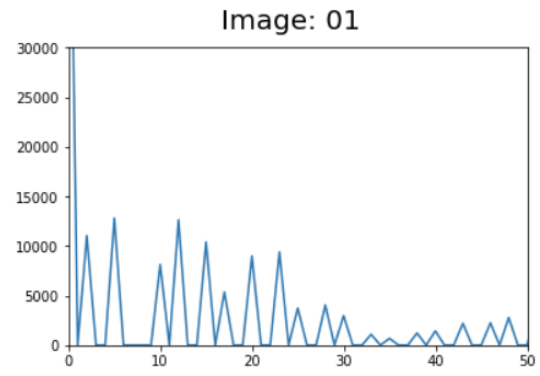
Appendix A

Code listing

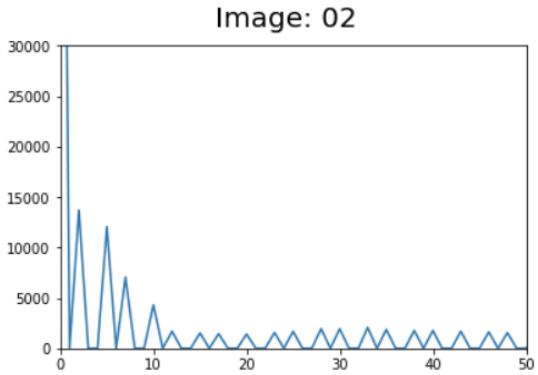
Running Code Github Link:
`https://github.com/songohan98/Visual-Hull-Code-in-Python-3.git`

Appendix B

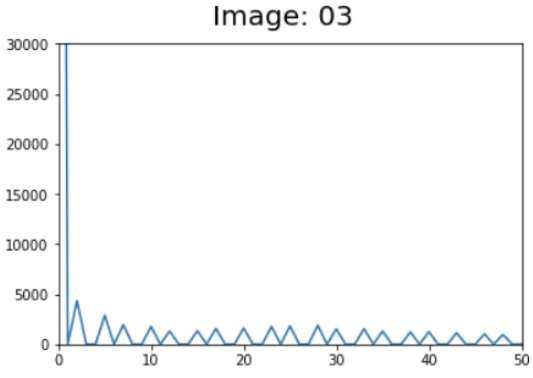
Threshold values from 16 images' colour intensity graph



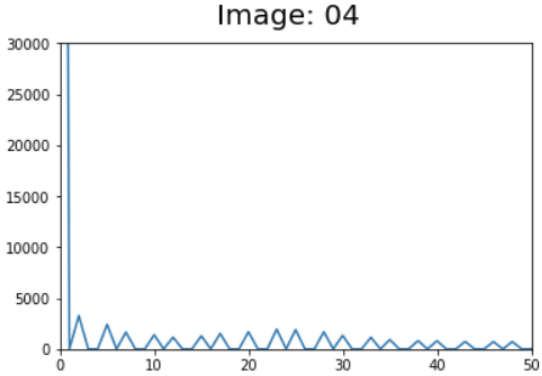
(a) fig 1



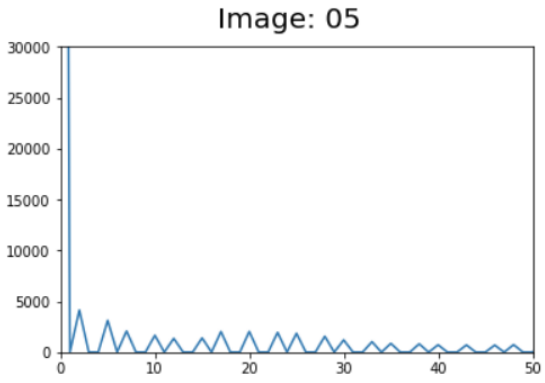
(b) fig 2



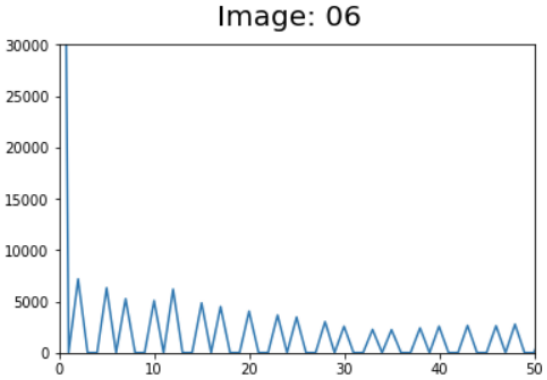
(c) fig 3



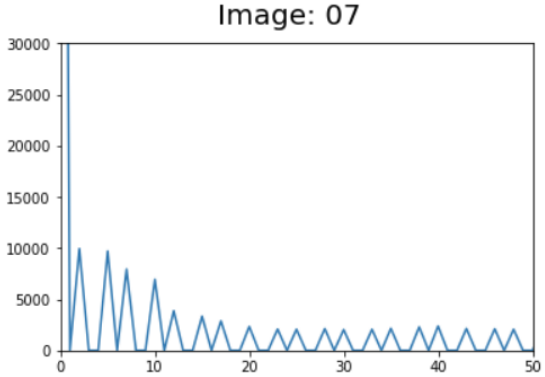
(d) fig 4



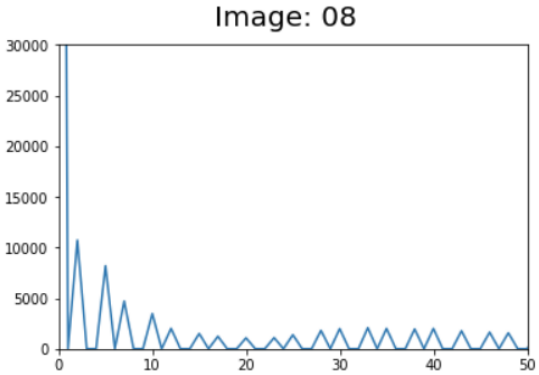
(a) fig 5



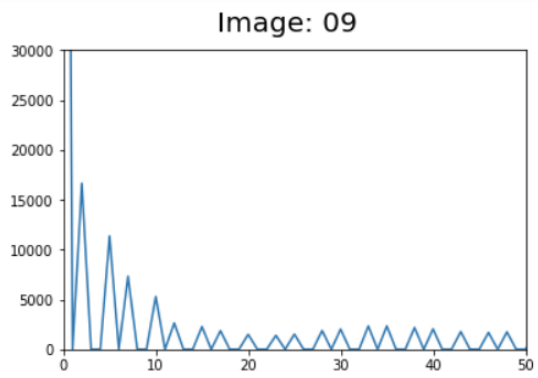
(b) fig 6



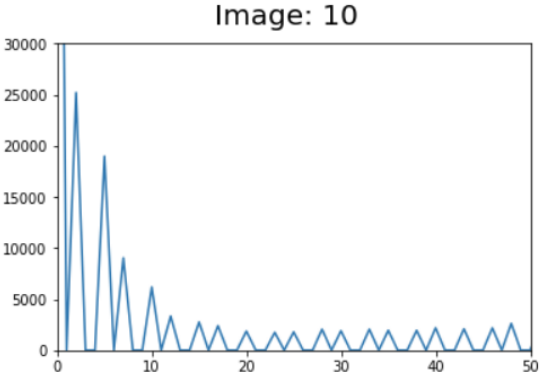
(c) fig 7



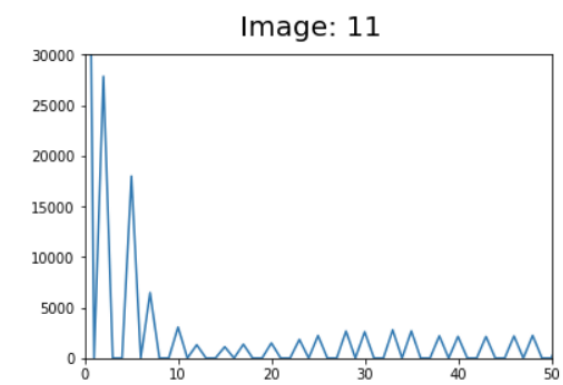
(d) fig 8



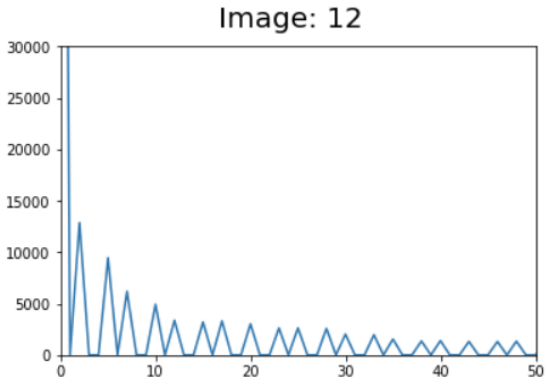
(e) fig 9



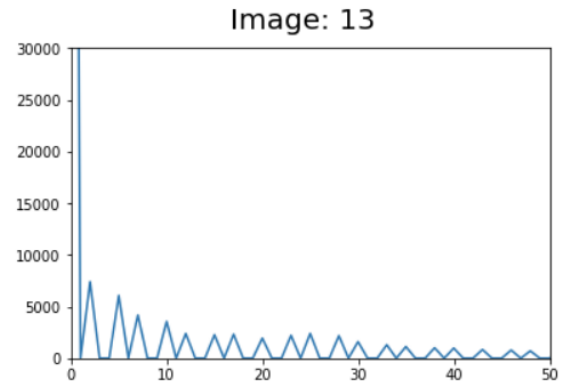
(f) fig 10



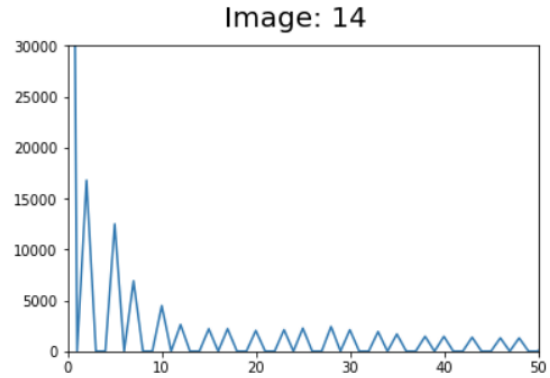
(a) fig 11



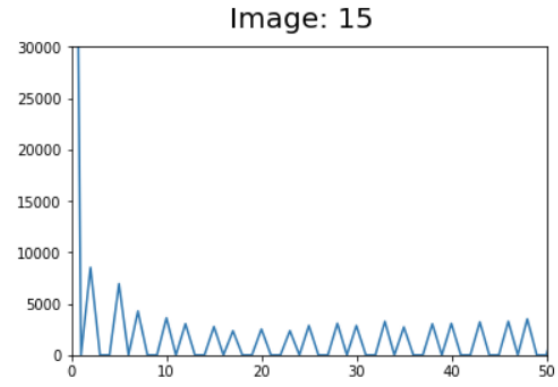
(b) fig 12



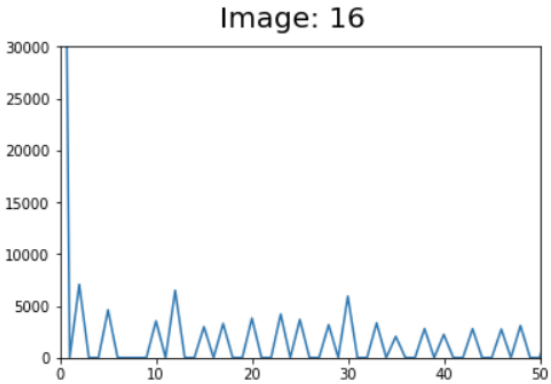
(c) fig 13



(d) fig 14



(e) fig 15



(f) fig 16

Figure B.1: Manually the points of threshold values where identified and used in the segmentation part