

# ELEC4630 Assignment 4

Samuel Eadie - 44353607

May 31, 2018

# 1 Volumetric Modelling - Shape from Silhouette

## 1.1 Introduction

Volumetric reconstruction can fuse numerous camera images from arbitrary positions into a three-dimensional model. A 'shape from silhouette' reconstruction is implemented for 36 camera angles to create a three-dimensional model of a toy dinosaur. In addition, an alternative voting method is implemented and compared.

## 1.2 Method

Two variations of the 'shape from silhouette' method were implemented: a carving method and voting method. These methods are outlined below, as well as operations which are in common for both methods. Following either of these methods, the volumetric model was flipped.

### 1.2.1 Thresholding Silhouettes

The silhouettes, used in both images, was created by thresholding the images in HSV domain, since this accentuated the difference between the blue background and the red/orange dinosaur. The dinosaur silhouette was selected as the largest component post thresholding. This process is explicated below, and the results are shown in Figure 1.

Listing 1: Creating silhouettes through HSV-domain thresholding

```
1 function [silhouette] = createDinoSilhouette(image)
2     % Threshold in HSV space
3     I = rgb2hsv(image);
4     minHSV = 0.792;
5     maxHSV = 0.566;
6     mask = (I(:,:,1) >= minHSV) | (I(:,:,1) <= maxHSV);
7
8     %Select largest component
9     silhouette = zeros(size(mask));
10    stats = regionprops('table', mask, 'Area', 'PixelIdxList');
11    [~, sortingIndex] = sort(stats.Area, 'descend');
12    silhouette(stats.PixelIdxList{sortingIndex(1)}) = 1;
13 end
```

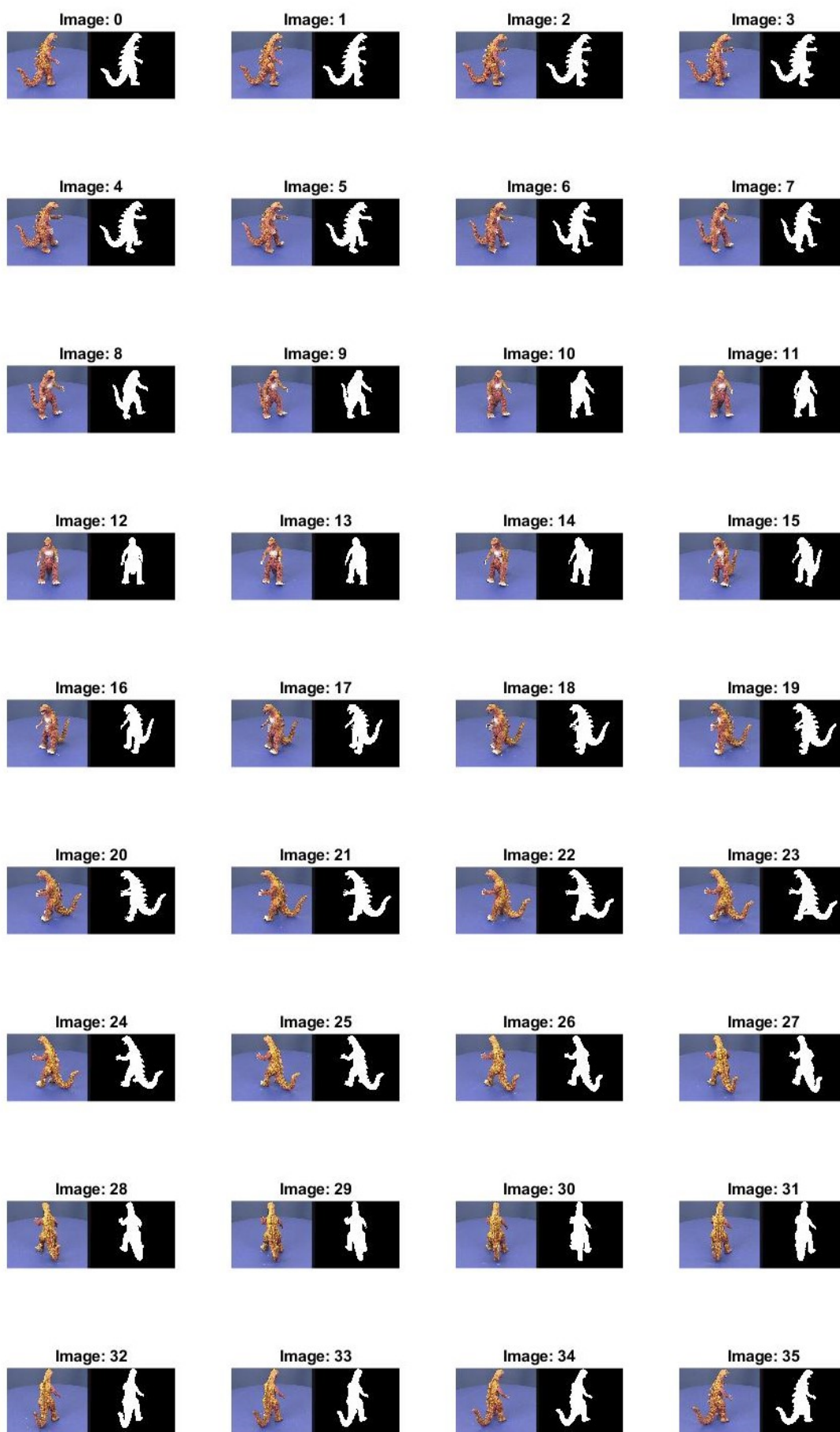


Figure 1: Dinosaur Silhouettes using HSV-domain thresholding

### 1.2.2 Initialisation of Voxel Space

Both methods begin by creating a rectangular voxel space from which the dinosaur is created. This is explicated in the following listing.

Listing 2: Initialisation of Voxel Space

```
1 % Create voxel space
2 RESOLUTION = 2;
3 voxelRanges = [-180 90; ... %X range
4               -80 70; ... %Y range
5               20 460]; %Z range
6 [voxelX, voxelY, voxelZ, voxelValues] = initialiseVoxels(voxelRanges,
7               RESOLUTION);
8
9 function [voxelX, voxelY, voxelZ, voxelValues] = initialiseVoxels(
10     voxelRanges, resolution)
11     x = voxelRanges(1,1):resolution:voxelRanges(1,2);
12     y = voxelRanges(2,1):resolution:voxelRanges(2,2);
13     z = voxelRanges(3,1):resolution:voxelRanges(3,2);
14
15     [X, Y, Z] = meshgrid(x, y, z);
16     voxelX = X(:);
17     voxelY = Y(:);
18     voxelZ = Z(:);
19     voxelValues = ones(numel(X), 1);
20 end
```

### 1.2.3 Projection to Image Space

Both methods project voxels in real-world space onto the image space using the provided projection

matrices. This corresponds to:

$$\begin{bmatrix} x_i \\ y_i \\ w_i \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

This is explicated in the code listing below:

Listing 3: Projection Function

```
1 function [imageX, imageY] = projectOntoImage(projectionMatrix, voxelX,
2     voxelY, voxelZ)
3     z = projectionMatrix(3,1) * voxelX ...
4     + projectionMatrix(3,2) * voxelY...
5     + projectionMatrix(3,3) * voxelZ ...
6     + projectionMatrix(3,4);
7
8     imageX = round((projectionMatrix(1,1) * voxelX ...
9     + projectionMatrix(1,2) * voxelY...
10    + projectionMatrix(1,3) * voxelZ ...
11    + projectionMatrix(1,4)) ./ z);
```

```

12     imageY = round((projectionMatrix(2,1) * voxelX ...
13                 + projectionMatrix(2,2) * voxelY...
14                 + projectionMatrix(2,3) * voxelZ ...
15                 + projectionMatrix(2,4)) ./ z);
16 end

```

### 1.2.4 Carving Method

The carving method projects the voxel space onto each of the cameras in turn, and removes all voxels which are not projected onto the inside of the silhouette of that camera. The resulting voxels, after iteration on each camera, form the volumetric model of the dinosaur. This is explicated in the listing below, and the carving process is outlined in Figure 2.

Listing 4: Carving Method

```

1  %Carve voxel space from each angle
2  for i = 1:36
3      %Get silhouette
4      image = imread(sprintf('dino/dino%02d.jpg', i-1));
5      silhouette = createDinoSilhouette(image);
6
7      %Project voxels onto silhouette
8      [imageX, imageY] = projectOntoImage(projectionMatrices(:,:,i), voxelX
          , voxelY, voxelZ);
9
10     %Crop any projection outside image
11     [imageHeight, imageWidth, ~] = size(image);
12     cropMask = find((imageX >= 1) & (imageX <= imageWidth) & ...
13                 (imageY >= 1) & (imageY <= imageHeight));
14     imageX = imageX(cropMask);
15     imageY = imageY(cropMask);
16
17     % Create silhouette mask
18     imageIndices = sub2ind([imageHeight, imageWidth], round(imageY), round
        (imageX));
19     silhouetteMask = cropMask(silhouette(imageIndices) >= 1);
20
21     % Carve silhouette into voxels
22     voxelX = voxelX(silhouetteMask);
23     voxelY = voxelY(silhouetteMask);
24     voxelZ = voxelZ(silhouetteMask);
25     voxelValues = voxelValues(silhouetteMask);
26
27     % Show Result
28     figure('Position', [100 100 600 300]);
29     plotDino(voxelX, voxelY, voxelZ, voxelValues);
30 end

```

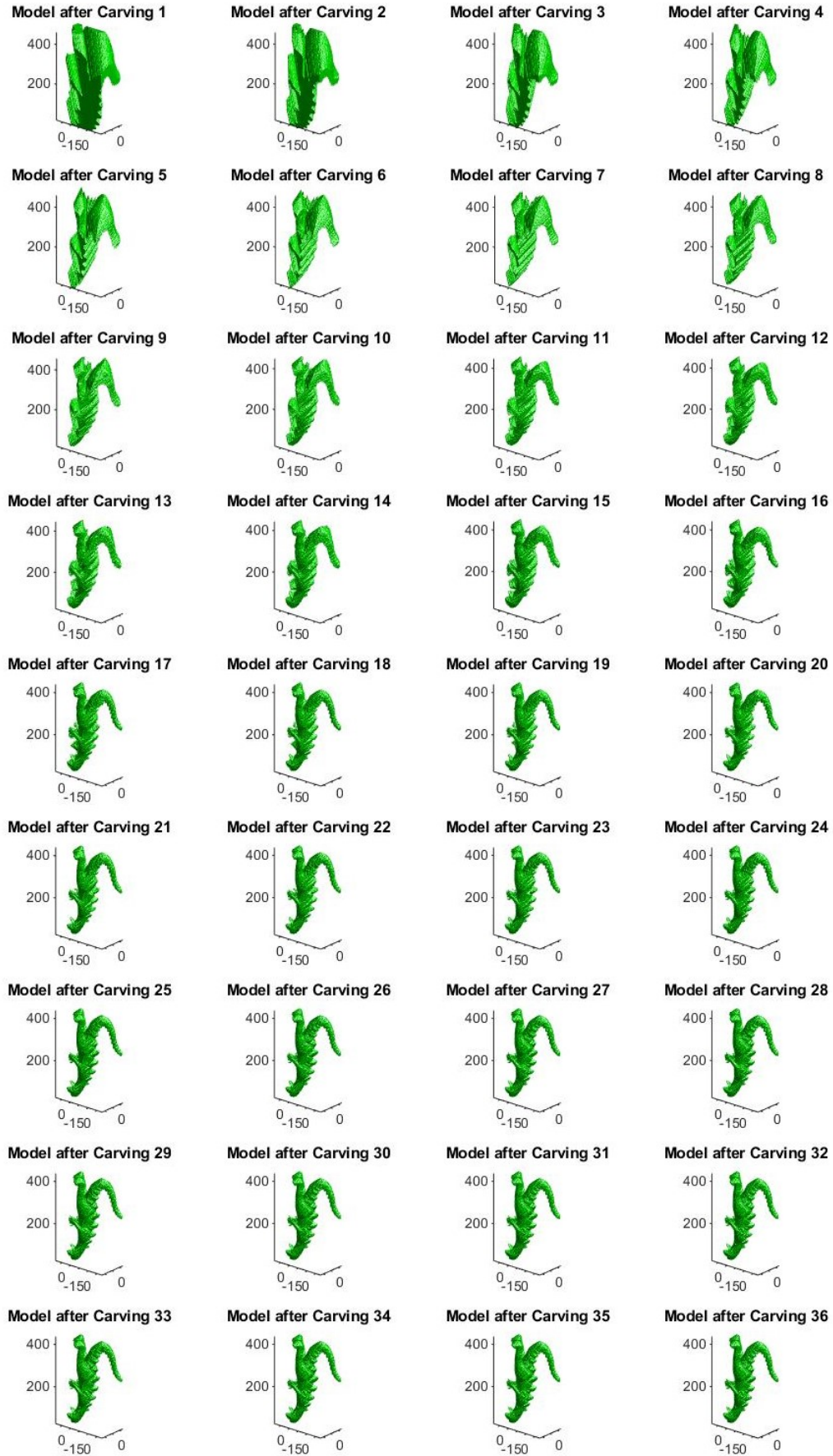


Figure 2: Carving Method after each projection

### 1.2.5 Voting Method

The voting method projects the voxel space onto each of the camera angles in turn, but instead of removing all voxels outside the silhouette, it increments the vote assigned to each voxel that is projected inside the silhouette. After voting from all 36 projections, the voxels can be thresholded based on the number of votes they received. This allows the flexibility to include voxels that were, for instance, valid for all but one of the camera angles. If a threshold of 36 is used, equal to the number of projections, the voting and carving methods result in the same volumetric model. This is less computationally efficient than the carving method since the initial number of voxels must be used in each iteration since voxels are not discarded until after all iterations. This process is explicated in the listing below, and the vote thresholding process is outlined in Figure 3.

Listing 5: Voting Method

```
1  % Vote for voxels from each angle
2  for i = 1:36
3      %Get silhouette
4      image = imread(sprintf('dino/dino%02d.jpg', i-1));
5      silhouette = createDinoSilhouette(image);
6
7      %Project voxels onto image
8      [projectedX, projectedY] = projectOntoImage(projectionMatrices(:,:,i)
          , voxelX, voxelY, voxelZ);
9
10     %Vote
11     [imageHeight, imageWidth, ~] = size(image);
12     for i = 1:length(projectedX)
13         if(projectedY(i) <= imageHeight && projectedY(i) > 0 &&...
14             projectedX(i) <= imageWidth && projectedX(i) > 0 &&...
15             silhouette(projectedY(i), projectedX(i)) >= 1)
16             voxelValues(i) = voxelValues(i) + 1;
17         end
18     end
19 end
20
21 figure();
22 votingThresholds = [0 5 10 15 20 25 30 35 36 40]
23 for i = 1:10
24     % 'Carve' using voting threshold
25     thresholdedVoxelX = voxelX(voxelValues >= votingThresholds(i));
26     thresholdedVoxelY = voxelY(voxelValues >= votingThresholds(i));
27     thresholdedVoxelZ = voxelZ(voxelValues >= votingThresholds(i));
28     thresholdedVoxelValues = voxelValues(voxelValues >= votingThresholds(
        i));
29
30     % Show Result
31     subplot(5,2,i);
32     plotDino(thresholdedVoxelX, thresholdedVoxelY, thresholdedVoxelZ,
        thresholdedVoxelValues);
33     title(sprintf('Voting Threshold = %d', votingThresholds(i)));
34 end
```



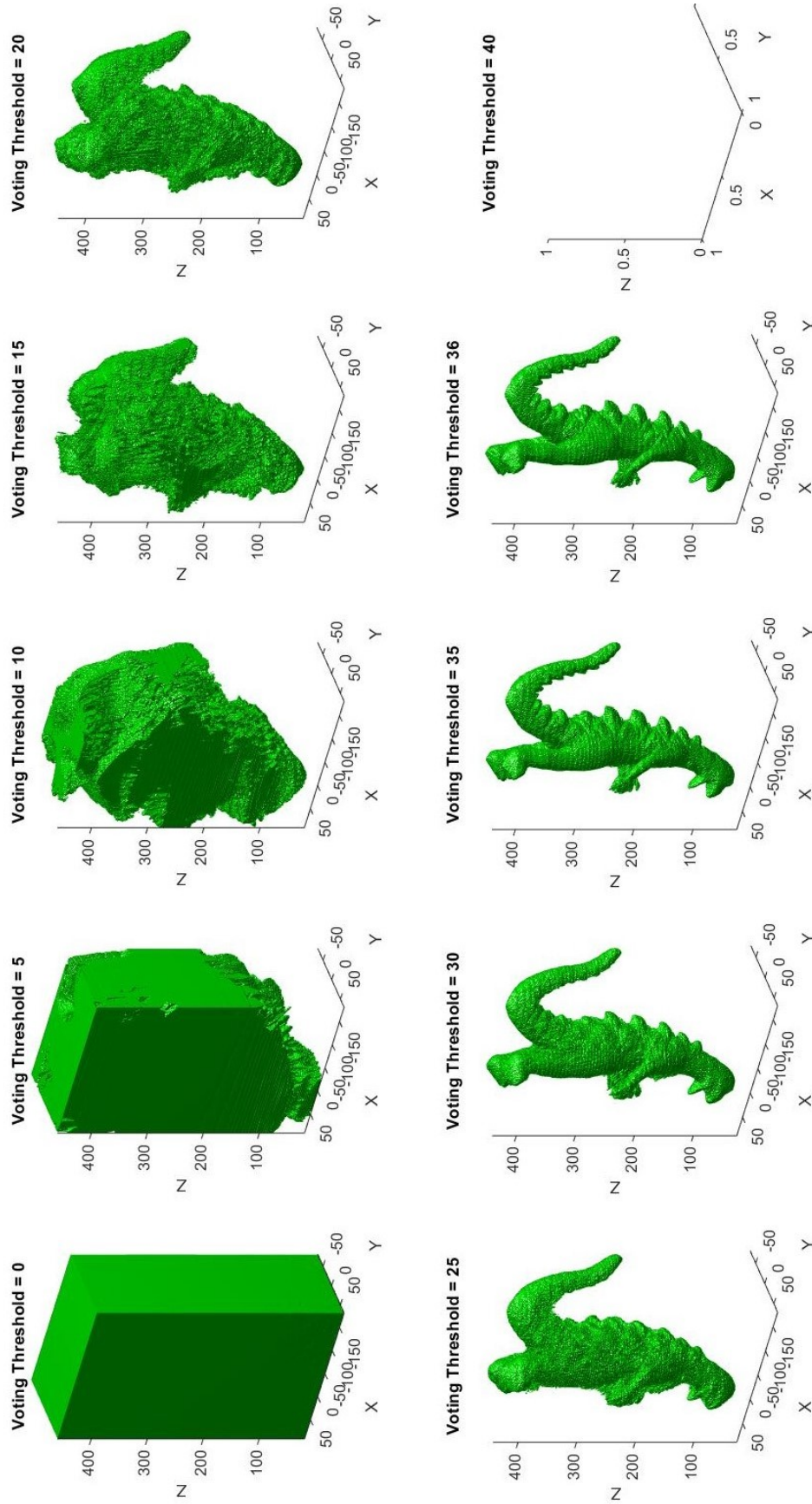


Figure 3: Various thresholds on the number of votes



### 1.3 Results

The equivalent dinosaur from both the voting and carving methods is shown from various angles in Figure 4.

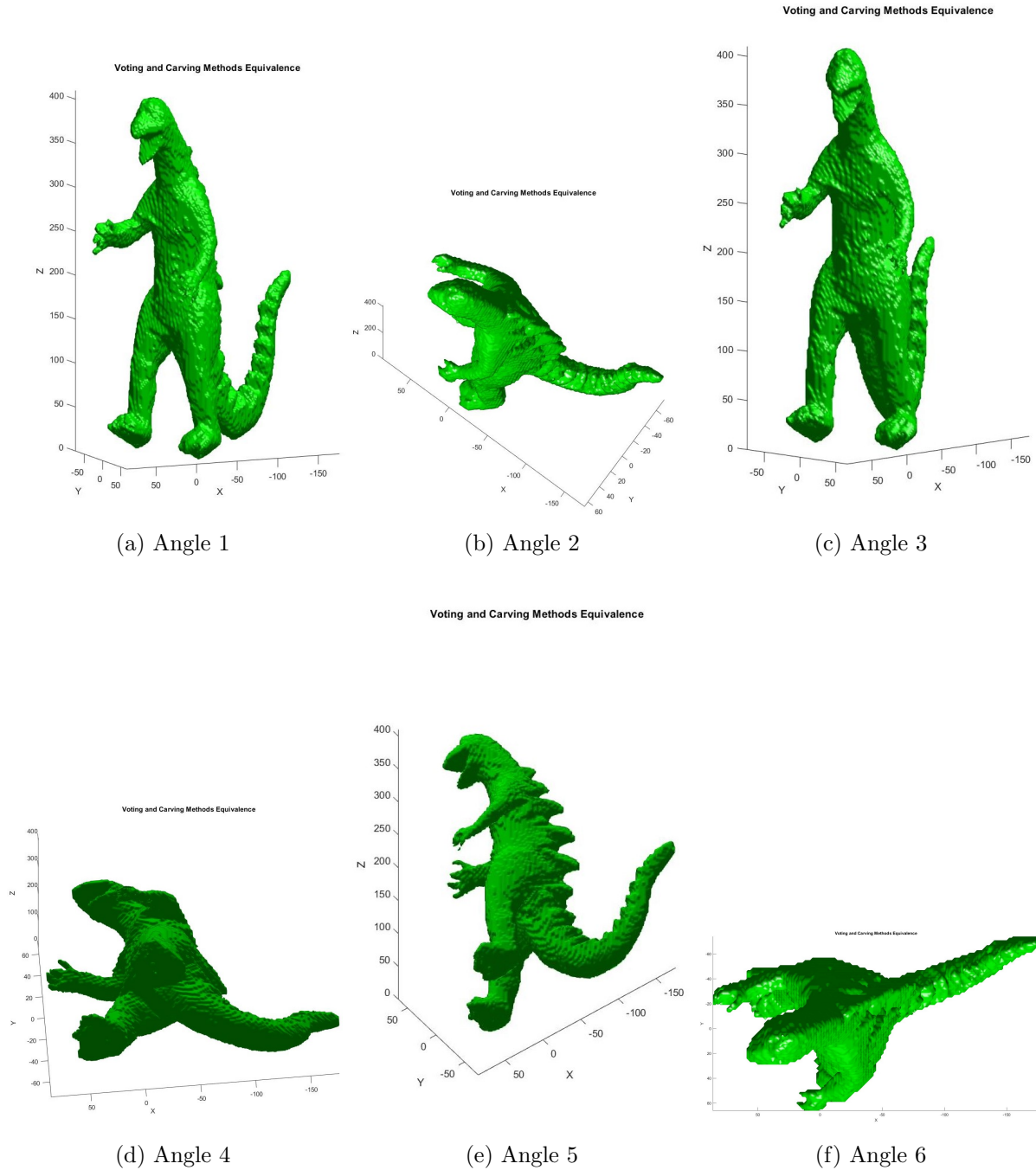


Figure 4: Final Result for Volumetric Model

## 1.4 Discussion

While the method successfully created a volumetric model of the dinosaur, the model did not capture colour information. Two different methods were implemented, albeit unsuccessfully, in an attempt to incorporate this colour information. This is an avenue for future work. Both methods require the location and direction of the cameras. These were determined by decomposing the projection matrices into their intrinsic matrix,  $K$  and their extrinsic matrices,  $R$ ,  $T$ .  $R$  and  $T$  correspond to the rotation and translation matrices respectively and hence the location of the camera is given by  $-R^T T$ . The intrinsic matrix contains the focal length in  $x$  and  $y$ , the skew coefficient and the principal points in  $x$  and  $y$ . The normal is then derived from this and the pixel image center. This is explicated in the listing below, and shown in Figure 5.

Listing 6: Location and Direction of Cameras

```
1 function [location, direction] = getCameraLocation(projectionMatrix)
2     [q,r] = qr(inv(projectionMatrix(1:3,1:3)));
3     invK = r(1:3,1:3);
4     R = inv(q);
5     if det(R) < 0
6         R = -R;
7         invK = -invK;
8     end
9     t = invK*projectionMatrix(:,4);
10
11     location = -q * t;
12
13     imageCenter = [360; 288; 1];
14     X = R' * (invK * imageCenter);
15     direction = X ./ norm(X);
16 end
```

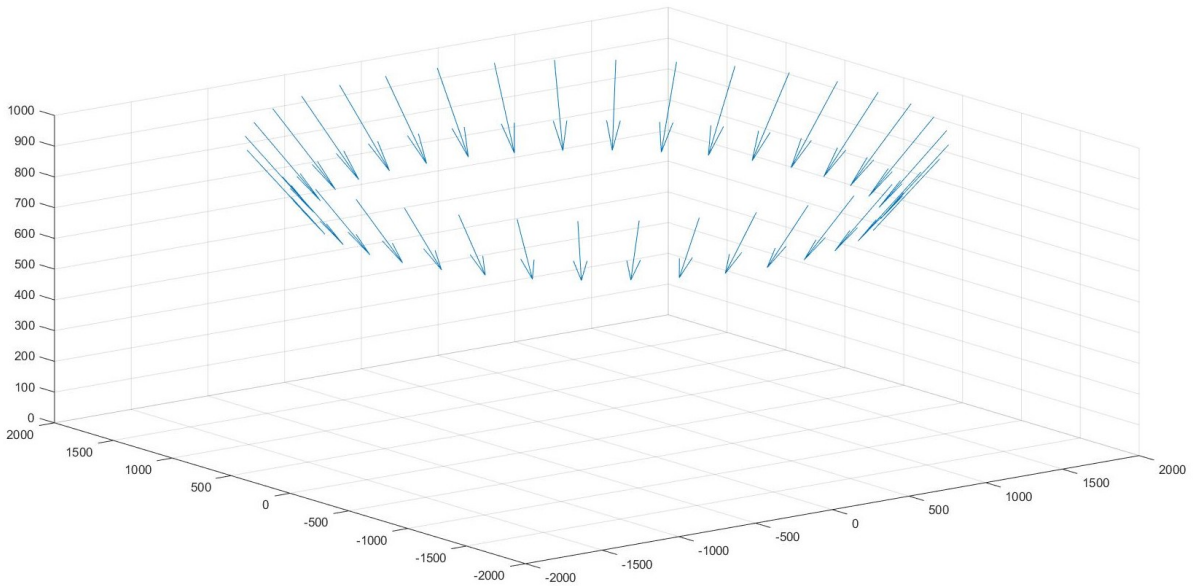


Figure 5: Location and Direction of Cameras

### 1.4.1 Colouring from Minimum Euclidean Distance

This method sought to match each voxel to a pixel in the image of the camera it was closest to. Specifically, it:

1. Calculates the euclidean distance between the voxel and all camera locations
2. Finds the camera location with the minimum euclidean distance to the voxel
3. Projects the voxel onto the image from that minimum-distance camera location
4. Assigns the projected pixel's colour to the voxel

This process is explicated in the below listing, and the unsuccessful result is shown in Figure 6.

Listing 7: Colouring by Minimum Euclidean Distance

```
1 function colourDinoUsingDistance(p, images, silhouettes,
   projectionMatrices, cameraLocations)
2     vertices = get(p, 'Vertices');
3     nc = uint8(zeros(size(vertices, 1), 3));
4
5     % For each voxel...
6     for i = 1:size(vertices, 1)
7         % Calculate euclidean distance between all camera locations
8         [~, closestCameraIndex] = pdist2(cameraLocations, vertices(i, :),
           'squaredeuclidean', 'Smallest', 1);
9         % Project onto closest camera's image
10        [imageX, imageY] = projectOntoImage(projectionMatrices(:, :,
           closestCameraIndex), vertices(i,1), vertices(i,2), vertices(i,
           3));
11        % Assign colour from projected pixel
12        image = images(:, :, :, closestCameraIndex);
13        nc(i, :) = reshape((image(imageY, imageX, :)), 1, 3);
14    end
15
16    set(p, 'FaceVertexCData', nc, 'FaceColor', 'interp');
17    p.EdgeColor = 'none';
18 end
```

Coloured using Vertex Distances

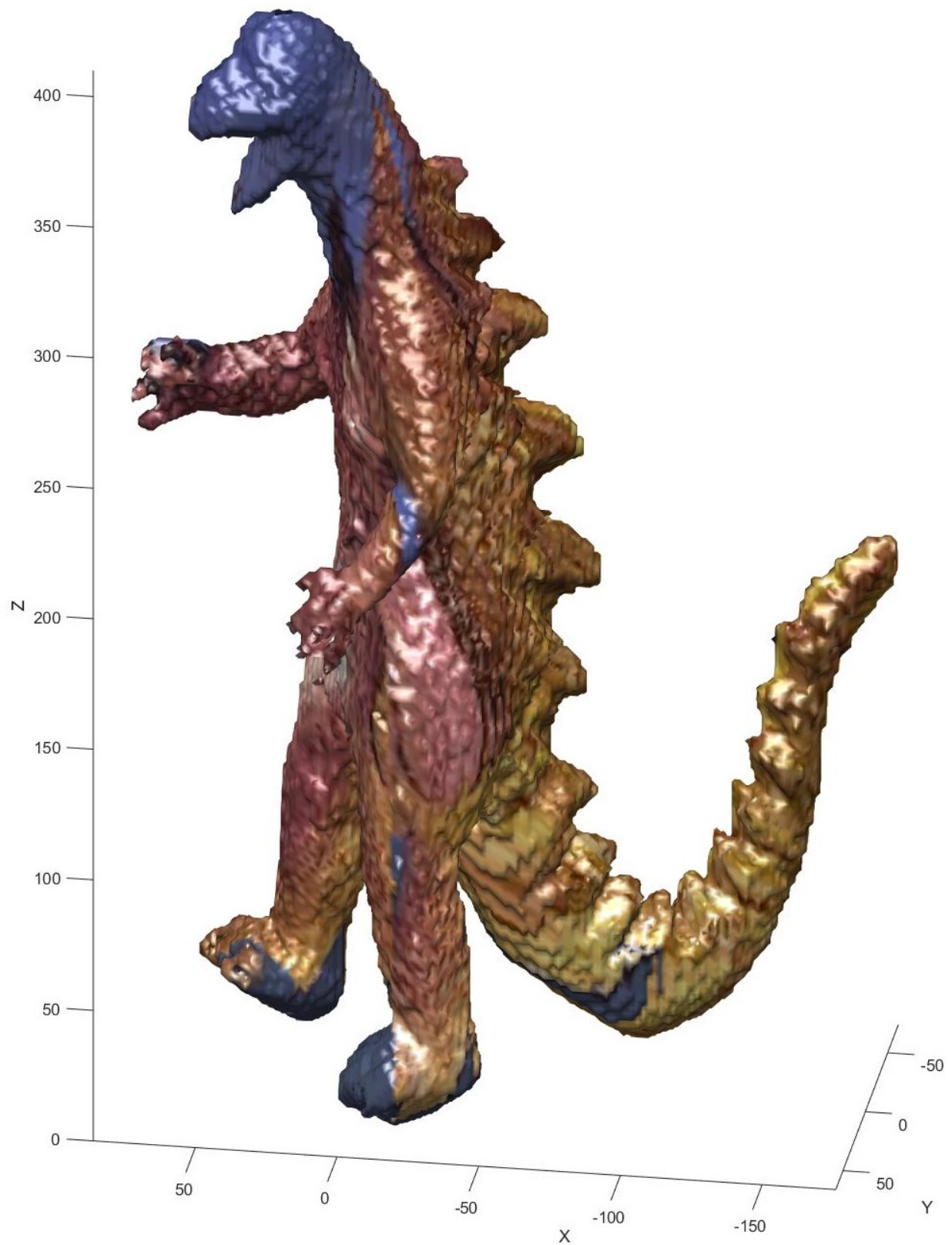


Figure 6: Colouring of Dinosaur using Minimum Euclidean Distance

### 1.4.2 Colouring from Maximum Normal Alignment

This method sought to match each voxel to a pixel in the image of the camera that had a direction closest to the normal of the vertex of the pixel. Specifically, it:

1. Calculates the angle between the vertex normal and the direction of each camera
2. Finds the camera with the minimum dot product
3. Projects the voxel onto the image from that camera location
4. Assigns the projected pixel's colour to the voxel

This process is explicated in the below listing, and the unsuccessful result is shown in Figure 7.

Listing 8: Colouring by Maximum Normal Alignment

```
1 function colourDinoUsingNormals(p, images, silhouettes,  
   projectionMatrices, cameraDirections)  
2  
3     vertices = get(p, 'Vertices');  
4     vertexNormals = get(p, 'VertexNormals');  
5     nc = uint8(zeros(size(vertexNormals, 1), 3));  
6  
7     % For each voxel...  
8     for i = 1:size(vertexNormals, 1)  
9         % Calculated dot products  
10        angles = vertexNormals(i,:)*cameraDirections'./norm(vertexNormals  
   (i,:));  
11        % Find minimum angle  
12        [~, cameraOrder] = sort(angles, 'ascend');  
13        cameraIndex = 1;  
14        % Project onto that camera  
15        [imageX, imageY] = projectOntoImage(projectionMatrices(:,:,  
   cameraOrder(cameraIndex)), vertices(i,1), vertices(i,2),  
   vertices(i,3));  
16        while((imageX < 1) || (imageX > size(images,2)) || (imageY < 1)  
   || (imageY > size(images, 1)))  
17            cameraIndex = cameraIndex + 1;  
18            [imageX, imageY] = projectOntoImage(projectionMatrices(:,:,  
   cameraOrder(cameraIndex)), vertices(i,1), vertices(i,2),  
   vertices(i,3));  
19        end  
20        % Assign colour from projected pixel  
21        image = images(:,:,:,cameraOrder(cameraIndex));  
22        nc(i,:) = reshape(image(imageY, imageX, :), 1, 3);  
23    end  
24  
25    set(p, 'FaceVertexCData', nc, 'FaceColor', 'interp');  
26    p.EdgeColor = 'none';  
27 end
```

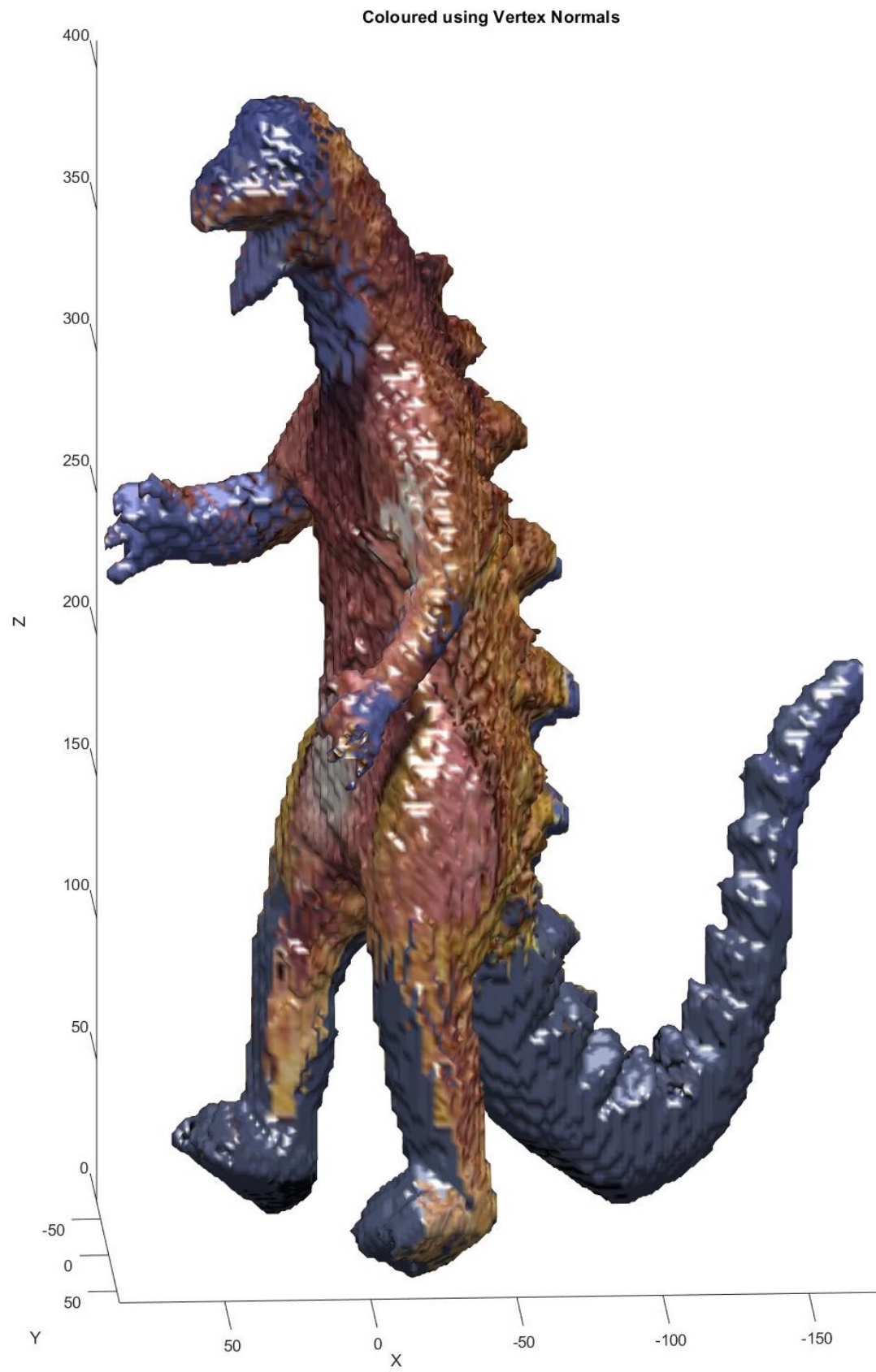


Figure 7: Colouring of Dinosaur using Maximum Normal Alignment



## 2 Eigenface Recognition

### 2.1 Introduction

A face recognition system was developed using template matching based on the eigenface technique.

### 2.2 Method

The face recognition system was developed through computing eigenfaces from test images, projecting new faces on these eigenfaces for recognition, and wrapping this algorithm in a GUI. These are outlined below.

#### 2.2.1 Computing Eigenfaces

Eigenfaces were computed using face images from various people. These eigenfaces are basis vectors which span the face space and hence can be used to characterise the population. The linear combination of these eigenface basis vectors required for each person's face is representative of the face and is later used for recognition. The eigenfaces were computed by:

1. Training face images were vectorised. These training images were preprocessed through centering, scaling and grayscaling.
2. The average face vector,  $\Psi$ , was computed from the image vectors,  $\Gamma$  by:  $\Psi = \frac{1}{M} \sum_{i=1}^M \Gamma_i$
3. The face images were standardised,  $\Phi$ , by:  $\Phi_i = \Gamma_i - \Psi$
4. The covariance matrix,  $C$  was computed, where:  $C = \frac{1}{M} \sum_{n=1}^M \Phi_n \Phi_n^T$ . However, for computational efficiency,  $C$ , was computed through  $C = AA^T$  where  $A = [\Phi_1, \Phi_2, \Phi_3, \dots, \Phi_M]$ . This returns the  $M$  eigenvectors with the highest associated eigenvalues.
5. The eigenvectors of the covariance matrix was computed. Again, for computational efficiency,  $C = AA^T$ , was used. The relationship between the eigenvectors of this smaller covariance matrix,  $v_i$ , and the full covariance matrix,  $\mu_i$  is  $\mu_i = Av_i$ . The eigenvalues are equivalent.
6. The training face images were projected onto these eigenface basis vectors to characterise them through their linear combination weightings.

This process is explicated in the below listing

Listing 9: Computation of Eigenface Basis Vectors

```
1 function self = EigenfaceRecognizer(images)
2     %Vectorise face images
3     self.theFaceVectors = reshape(images, [], size(images, 3));
4
5     %Compute average face
6     self.theMeanFaceVector = mean(self.theFaceVectors, 2);
7
8     %Standardise face vectors
9     faceDiffVectors = double(self.theFaceVectors) - self.
10        theMeanFaceVector;
```

```

11      %Compute covariance matrix as A*transpose(A)
12      C = faceDiffVectors' * faceDiffVectors;
13
14      %Compute eigenvectors of covariance
15      [eVectors, ~] = eig(C);
16      eVectors = faceDiffVectors * eVectors;
17
18      %Normalise eigenvectors
19      self.theEigens = eVectors ./ vecnorm(eVectors, 2, 1);
20
21      %Characterise faces as linear combination of eigenface basis
22      %vectors
23      self.theProjections = zeros(size(self.theEigens, 2));
24      for faceNum = 1:size(images, 3)
25          self.theProjections(faceNum, :) = sum(self.theEigens .*
26              faceDiffVectors(:,faceNum));
27      end
28  end

```

The mean face and computed eigenfaces are shown in Figure 8.

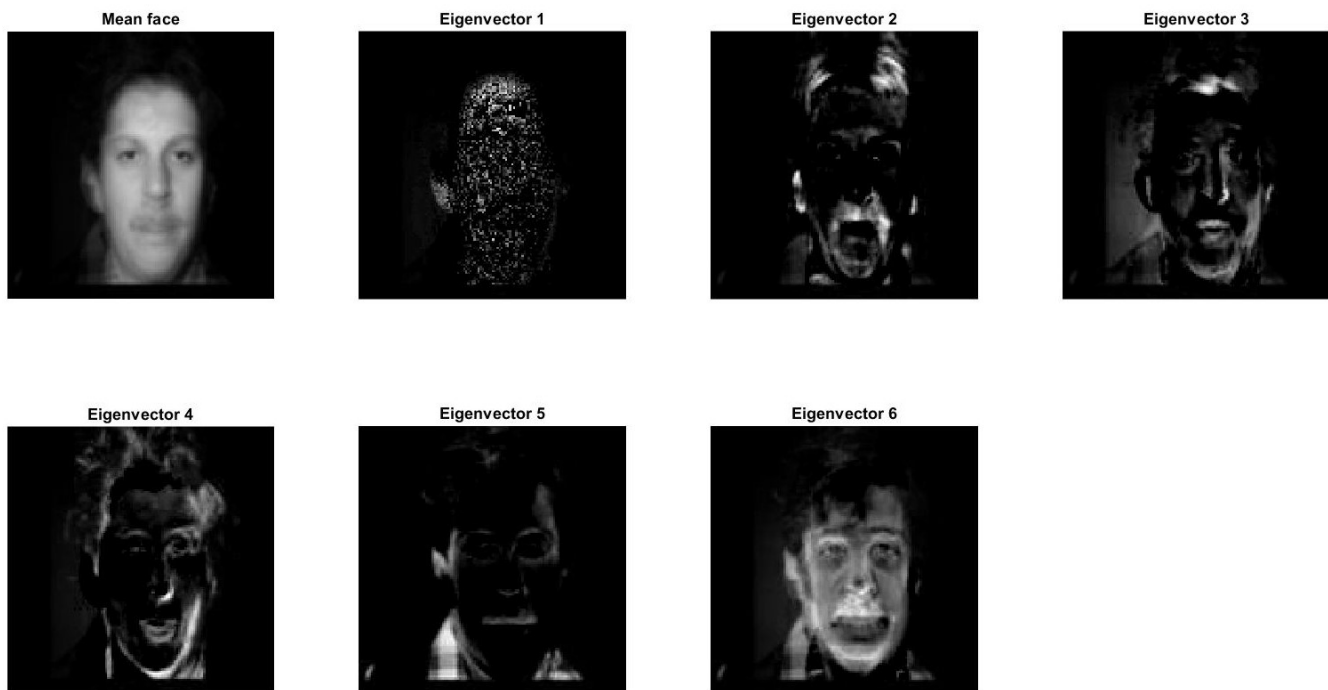


Figure 8: Mean Face and Computed Eigenfaces

This process is analogous to Principal Component Analysis (PCA) with the eigenfaces corresponding to the principal components. The effectiveness of PCA, and hence this eigenface technique, is shown in Figure 9. This shows all the face images projected onto the two most prominent principal components (eigenfaces 1 and 2). From inspection, this successfully clusters the face images into the different faces. The recognition process described below is equivalent to finding the minimum euclidean distance between a face and these clusters.

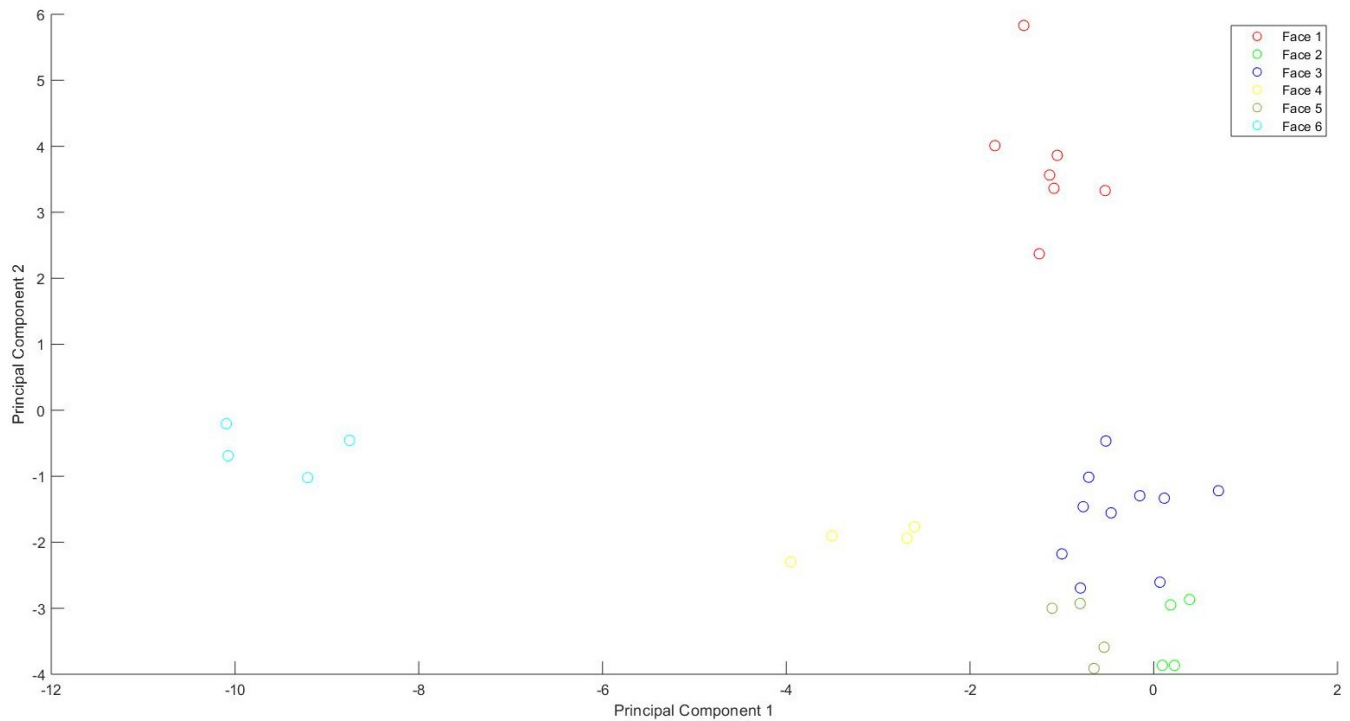


Figure 9: Face Images projected onto two principal components

### 2.2.2 Face Recognition

Face recognition uses template matching on the eigenface template projections. This recognition process is outlined below:

1. The face image is vectorised into  $\Gamma$ . This image was preprocessed through centering, scaling and grayscaleing
2. The face image,  $\Phi$ , was standardised, using the average face previously computed, by:  $\Phi = \Gamma - \Psi$
3. The face image is projected onto the eigenface basis vectors:  $\Omega = \text{proj}_{\mu_i} \Phi$
4. The projection weightings,  $\Omega$ , are compared to the test image weightings. Specifically, the distances between  $\Omega$  and the test image weightings are computed and the minimum distance found. The image is 'recognised' as the test face image corresponding to the minimum distance. A euclidean distance was used.

This process is explicated in the below listing:

Listing 10: Recognition through Eigenface Template Matching

```

1  function classification = recognizeFace(self, face)
2      %Vectorise face image
3      faceVector = face(:);
4
5      %Project normalised face
6      faceDiffVector = double(faceVector) - self.theMeanFaceVector;
7      projection = sum(self.theEigens .* faceDiffVector);

```

```

8
9      %Recognize face as closest projection
10     distances = pdist2(projection, self.theProjections, self.
11                        theDistanceMetric);
12     [~, classification] = min(distances);
end

```

### 2.2.3 GUI

A GUI was developed to allow interaction with this algorithm. It uses the *EigenfaceRecognizer* class outlined above to classify images and displays the corresponding test image match. The user can repeatedly select for new images to recognise through searching their computer's file structure. After recognition, the GUI shades all non-matching faces to leave only the recognized face. The GUI is shown in Figure 10 and explicated in the code listing below.

Listing 11: Facial Recognition GUI

```

1 function GUI
2     %Create figure
3     figHeight = 700;
4     figWidth = 1400;
5     f = figure('Visible','off','Position',[25, 50, figWidth, figHeight]);
6
7     %Load eigenfaces
8     images = [];
9     imagesAxes = [];
10    subplotNums = [1 2 3 7 8 9];
11    for i = 1:6
12        hold on;
13        [image, map] = imread(sprintf('faces/eig/%da.bmp', i));
14        image = rgb2gray(ind2rgb(image, map));
15        images = cat(3, images, image);
16        imagesAxes = [imagesAxes subplot(2,6,subplotNums(i))];
17        imshow(images(:,:,i), [0 max(max(images(:,:,i)))]);
18        title(sprintf('Face %d', i));
19    end
20    hold off;
21
22    %Create eigenface recognizer
23    recognizer = EigenfaceRecognizer(images);
24
25    % Test Image plot
26    testImageAxes = subplot(2, 6, [4 5 6 10 11 12]);
27
28    %Create menubar
29    menubar = uimenu(f, 'Text', 'File');
30    newFileMenu = uimenu(menubar, 'Text', 'New File', 'MenuSelectedFcn',
31                        @newFileCallback);
32    closeMenu = uimenu(menubar, 'Text', 'Exit', 'MenuSelectedFcn',
33                      @closeCallback);

```

```

33     function updateImageClasses(classification)
34         for i = 1:6
35             axes(imagesAxes(i))
36             if i ~= classification
37                 imshow(0.25 * images(:,:,i), [0 max(max(images(:,:,i)))]);
38                 ;
39                 title(sprintf('Face %d', i));
40             else
41                 imshow(images(:,:,i), [0 max(max(images(:,:,i)))]);
42                 title(sprintf('Face %d', i));
43             end
44             hold on;
45         end
46         hold off;
47     end
48     function newFileCallback(~, ~)
49         [file, path] = uigetfile("*.bmp");
50         if ~isequal(file, 0)
51             [testImage, map] = imread(fullfile(path, file));
52             testImage = rgb2gray(ind2rgb(testImage, map));
53
54             axes(testImageAxes)
55             imshow(testImage, [0 max(testImage(:))]);
56             title(sprintf('Test Image: %s', file));
57
58             classification = recognizer.recognizeFace(testImage);
59             updateImageClasses(classification);
60         end
61     end
62
63     function closeCallback(~, ~)
64         close all
65     end
66
67     newFileCallback(0, 0);
68
69     % Make the UI visible.
70     f.Visible = 'on';
71 end

```

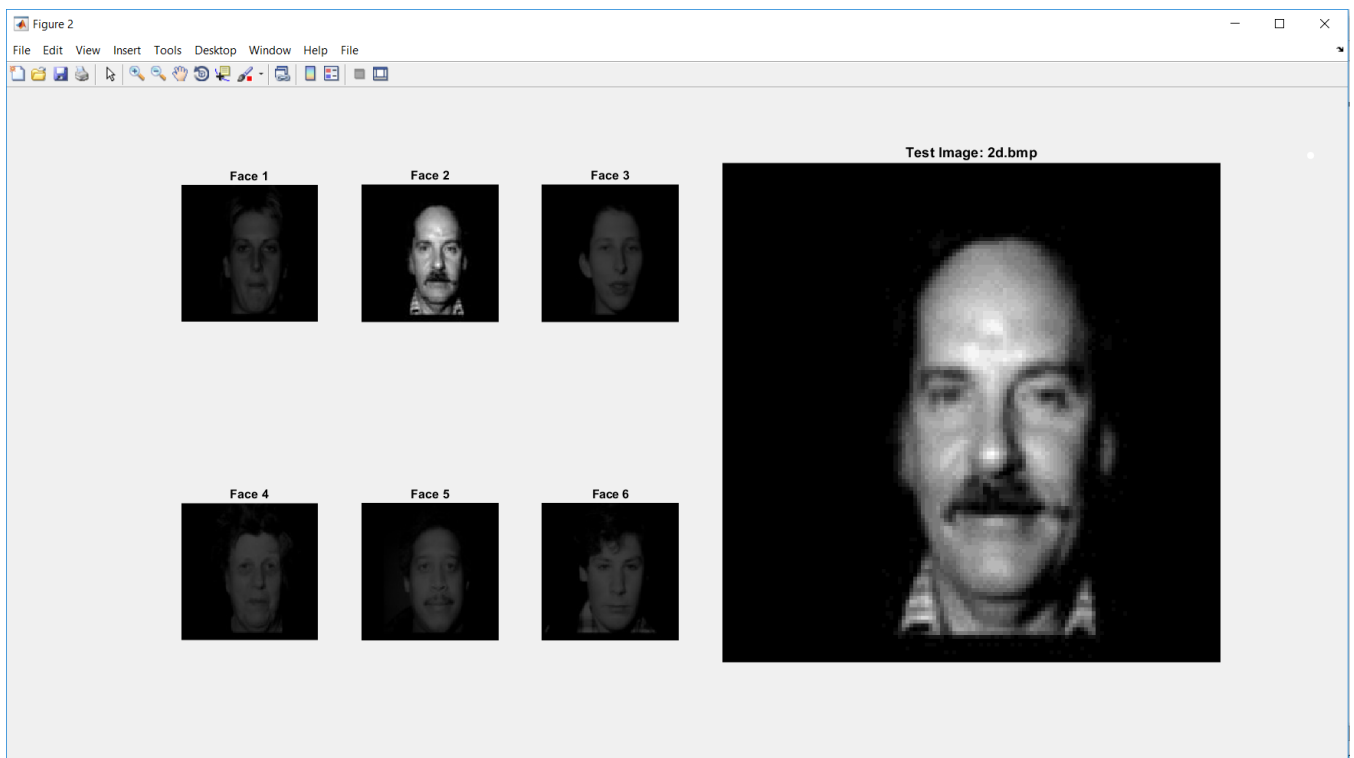


Figure 10: Facial Recognition GUI

## 2.3 Results

The face recognition system correctly identified 32 of the 33 images, corresponding to a 96.97% success rate. The incorrectly identified image is shown in Figure 11.

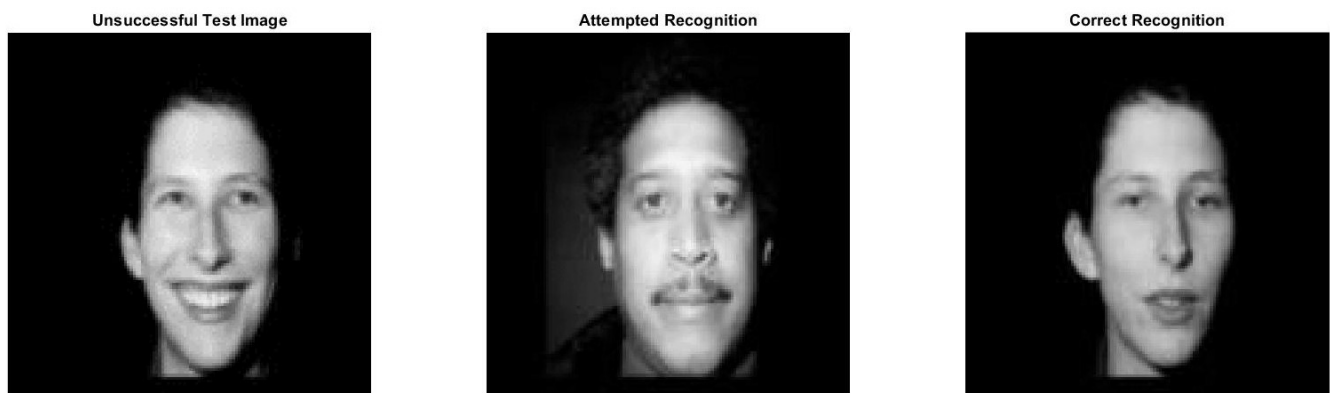


Figure 11: The Incorrectly Recognised Face



# Appendix A: Additional Code Listings

## Volumetric Modelling

Listing 12: Full Volumetric Modelling Process

```
1  % Load projection matrices
2  projectionMatrices = loadProjectionMatrices();
3
4  %%%Carving Method
5  % Create voxel space
6  RESOLUTION = 2;
7  voxelRanges = [-180 90; ... %X range
8                 -80 70; ... %Y range
9                 20 460];    %Z range
10 [voxelX, voxelY, voxelZ, voxelValues] = initialiseVoxels(voxelRanges,
    RESOLUTION);
11
12 %Carve voxel space from each angle
13 for i = 1:36
14     %Get silhouette
15     image = imread(sprintf('dino/dino%02d.jpg', i-1));
16     silhouette = createDinoSilhouette(image);
17
18     %Project voxels onto silhouette
19     [imageX, imageY] = projectOntoImage(projectionMatrices(:, :, i), voxelX
    , voxelY, voxelZ);
20
21     %Crop any projection outside image
22     [imageHeight, imageWidth, ~] = size(image);
23     cropMask = find((imageX >= 1) & (imageX <= imageWidth) & ...
24                 (imageY >= 1) & (imageY <= imageHeight));
25     imageX = imageX(cropMask);
26     imageY = imageY(cropMask);
27
28     % Create silhouette mask
29     imageIndices = sub2ind([imageHeight, imageWidth], round(imageY), round
    (imageX));
30     silhouetteMask = cropMask(silhouette(imageIndices) >= 1);
31
32     % Carve silhouette into voxels
33     voxelX = voxelX(silhouetteMask);
34     voxelY = voxelY(silhouetteMask);
35     voxelZ = voxelZ(silhouetteMask);
36     voxelValues = voxelValues(silhouetteMask);
37
38     % Show Result
39     figure('Position', [100 100 600 300]);
40     plotDino(voxelX, voxelY, voxelZ, voxelValues);
41 end
42
```

```

43 %%%Voting Method
44 % Create voxel space
45 RESOLUTION = 2;
46 voxelRanges = [-180 90; ... %X range
47                -80 70; ... %Y range
48                20 460]; %Z range
49 [voxelX, voxelY, voxelZ, voxelValues] = initialiseVoxels(voxelRanges,
    RESOLUTION);
50 voxelColours = uint8(zeros(length(voxelValues), 3));
51
52 % Vote for voxels from each angle
53 for i = 1:36
54     %Get silhouette
55     image = imread(sprintf('dino/dino%02d.jpg', i-1));
56     silhouette = createDinoSilhouette(image);
57
58     %Project voxels onto image
59     [projectedX, projectedY] = projectOntoImage(projectionMatrices(:,:,i)
        , voxelX, voxelY, voxelZ);
60
61     %Vote
62     [imageHeight, imageWidth, ~] = size(image);
63     for i = 1:length(projectedX)
64         if(projectedY(i) <= imageHeight && projectedY(i) > 0 &&...
65            projectedX(i) <= imageWidth && projectedX(i) > 0 &&...
66            silhouette(projectedY(i), projectedX(i)) >= 1)
67             voxelValues(i) = voxelValues(i) + 1;
68         end
69     end
70 end
71
72 figure();
73 votingThresholds = [0 5 10 15 20 25 30 35 36]; %40
74 for i = 1:9
75     % 'Carve' using voting threshold
76     thresholdedVoxelX = voxelX(voxelValues >= votingThresholds(i));
77     thresholdedVoxelY = voxelY(voxelValues >= votingThresholds(i));
78     thresholdedVoxelZ = voxelZ(voxelValues >= votingThresholds(i));
79     thresholdedVoxelValues = voxelValues(voxelValues >= votingThresholds(
        i));
80
81     % Show Result
82     subplot(5,2,i);
83     plotDino(thresholdedVoxelX, thresholdedVoxelY, thresholdedVoxelZ,
        thresholdedVoxelValues);
84     title(sprintf('Voting Threshold = %d', votingThresholds(i)));
85 end
86
87 %Plot carving method equivalent
88 thresholdedVoxelX = voxelX .* (voxelValues >= 36);

```

```

89 thresholdedVoxelY = voxelY .* (voxelValues >= 36);
90 thresholdedVoxelZ = voxelZ .* (voxelValues >= 36);
91 thresholdedVoxelColours = voxelColours .* uint8(repmat(voxelValues >= 36,
    1, 3));
92 thresholdedVoxelValues = voxelValues .* (voxelValues >= 36);
93
94 %Correct upside-downness
95 thresholdedVoxelZ = max(thresholdedVoxelZ(:)) - thresholdedVoxelZ;
96
97 % Show Result
98 figure();
99 p = plotDino(thresholdedVoxelX, thresholdedVoxelY, thresholdedVoxelZ,
    thresholdedVoxelValues);
100 % plotColourDino(thresholdedVoxelX, thresholdedVoxelY, thresholdedVoxelZ,
    thresholdedVoxelValues, thresholdedVoxelColours, RESOLUTION);
101 title('Voting and Carving Methods Equivalence');
102
103 % Colour in dino - well, try to
104 images = [];
105 silhouettes = [];
106 cameraLocations = zeros(size(projectionMatrices,3), 3);
107 cameraDirections = zeros(size(projectionMatrices,3), 3);
108 for i = 1:36
109     images = cat(4, images, imread(sprintf('dino/dino%02d.jpg', i-1)));
110     silhouettes = cat(3, silhouettes, createDinoSilhouette(images(:,:,i),
        i));
111     [cameraLocations(i,:), cameraDirections(i,:)] = getCameraLocation(
        projectionMatrices(:,:,i));
112 end
113
114 % Show Result
115 figure();
116 p = plotDino(thresholdedVoxelX, thresholdedVoxelY, thresholdedVoxelZ,
    thresholdedVoxelValues);
117 colourDinoUsingDistance(p, images, silhouettes, projectionMatrices,
    cameraLocations);
118 title('Coloured using Vertex Distances');
119
120 figure();
121 p = plotDino(thresholdedVoxelX, thresholdedVoxelY, thresholdedVoxelZ,
    thresholdedVoxelValues);
122 colourDinoUsingNormals(p, images, silhouettes, projectionMatrices,
    cameraDirections);
123 title('Coloured using Vertex Normals');
124
125 %%%General Functions
126 function [voxelX, voxelY, voxelZ, voxelValues] = initialiseVoxels(
    voxelRanges, resolution)
127     x = voxelRanges(1,1):resolution:voxelRanges(1,2);
128     y = voxelRanges(2,1):resolution:voxelRanges(2,2);

```

```

129     z = voxelRanges(3,1):resolution:voxelRanges(3,2);
130
131     [X, Y, Z] = meshgrid(x, y, z);
132     voxelX = X(:);
133     voxelY = Y(:);
134     voxelZ = Z(:);
135     voxelValues = ones(numel(X), 1);
136 end
137
138 function [imageX, imageY] = projectOntoImage(projectionMatrix, voxelX,
    voxelY, voxelZ)
139     z = projectionMatrix(3,1) * voxelX ...
140         + projectionMatrix(3,2) * voxelY...
141         + projectionMatrix(3,3) * voxelZ ...
142         + projectionMatrix(3,4);
143
144     imageX = round((projectionMatrix(1,1) * voxelX ...
145                     + projectionMatrix(1,2) * voxelY...
146                     + projectionMatrix(1,3) * voxelZ ...
147                     + projectionMatrix(1,4)) ./ z);
148
149     imageY = round((projectionMatrix(2,1) * voxelX ...
150                     + projectionMatrix(2,2) * voxelY...
151                     + projectionMatrix(2,3) * voxelZ ...
152                     + projectionMatrix(2,4)) ./ z);
153 end
154
155 function [ptch] = plotDino(voxelX, voxelY, voxelZ, voxelValues)
156     % First grid the data
157     ux = unique(voxelX);
158     uy = unique(voxelY);
159     uz = unique(voxelZ);
160
161     % Convert to a grid
162     [X,Y,Z] = meshgrid(ux, uy, uz);
163
164     % Create an empty voxel grid, then fill only those elements in voxels
165     V = zeros(size(X));
166     N = numel(voxelX);
167     for ii=1:N
168         ix = (ux == voxelX(ii));
169         iy = (uy == voxelY(ii));
170         iz = (uz == voxelZ(ii));
171         V(iy,ix,iz) = voxelValues(ii);
172     end
173
174     % Now draw it
175     ptch = patch(isosurface(X, Y, Z, V, 0.5));
176     isonormals(X, Y, Z, V, ptch)
177     set(ptch, 'FaceColor', 'g', 'EdgeColor', 'none' );

```

```

178
179     set(gca, 'DataAspectRatio', [1 1 1]);
180     xlabel('X');
181     ylabel('Y');
182     zlabel('Z');
183     view(-140, 22)
184     lighting('gouraud')
185     camlight('left')
186     axis('tight')
187 end
188
189 function colourDinoUsingDistance(p, images, silhouettes,
    projectionMatrices, cameraLocations)
190     vertices = get(p, 'Vertices');
191     nc = uint8(zeros(size(vertices, 1), 3));
192
193     for i = 1:size(vertices, 1)
194         [~, closestCameraIndex] = pdist2(cameraLocations, vertices(i, :),
            'squaredeuclidean', 'Smallest', 1);
195         [imageX, imageY] = projectOntoImage(projectionMatrices(:, :,
            closestCameraIndex), vertices(i, 1), vertices(i, 2), vertices(i
            , 3));
196         image = images(:, :, :, closestCameraIndex);
197         nc(i, :) = reshape((image(imageY, imageX, :)), 1, 3);
198     end
199
200     set(p, 'FaceVertexCData', nc, 'FaceColor', 'interp');
201     p.EdgeColor = 'none';
202 end
203
204 function colourDinoUsingNormals(p, images, silhouettes,
    projectionMatrices, cameraDirections)
205
206     vertices = get(p, 'Vertices');
207     vertexNormals = get(p, 'VertexNormals');
208     nc = uint8(zeros(size(vertexNormals, 1), 3));
209
210     for i = 1:size(vertexNormals, 1)
211         angles = vertexNormals(i, :) * cameraDirections' ./ norm(vertexNormals
            (i, :));
212         [~, cameraOrder] = sort(angles, 'ascend');
213         cameraIndex = 1;
214         [imageX, imageY] = projectOntoImage(projectionMatrices(:, :,
            cameraOrder(cameraIndex)), vertices(i, 1), vertices(i, 2),
            vertices(i, 3));
215         while ((imageX < 1) || (imageX > size(images, 2)) || (imageY < 1)
            || (imageY > size(images, 1)))
216             cameraIndex = cameraIndex + 1;
217             [imageX, imageY] = projectOntoImage(projectionMatrices(:, :,
            cameraOrder(cameraIndex)), vertices(i, 1), vertices(i, 2),

```

```

        vertices(i,3));
218     end
219     yeet = images(:,:,:,cameraOrder(cameraIndex));
220     nc(i,:) = reshape((yeet(imageY, imageX, :)), 1, 3);
221 end
222
223 set(p, 'FaceVertexCData', nc, 'FaceColor', 'interp');
224 p.EdgeColor = 'none';
225 end
226
227 function [location, direction] = getCameraLocation(projectionMatrix)
228     [q,r] = qr(inv(projectionMatrix(1:3,1:3)));
229     invK = r(1:3,1:3);
230     R = inv(q);
231
232     if det( R ) < 0
233         R = -R;
234         invK = -invK;
235     end
236     t = invK*projectionMatrix(:,4);
237
238     location = -q * t;
239
240     imageCenter = [360; 288; 1];
241     X = R' * (invK * imageCenter);
242     direction = X ./ norm(X);
243
244 end

```

Listing 13: Plots the Volumetric Model of a Dinosaur

```

1 function [ptch] = plotDino(voxelX, voxelY, voxelZ, voxelValues)
2     % First grid the data
3     ux = unique(voxelX);
4     uy = unique(voxelY);
5     uz = unique(voxelZ);
6
7     % Convert to a grid
8     [X,Y,Z] = meshgrid(ux, uy, uz);
9
10    % Create an empty voxel grid, then fill only those elements in voxels
11    V = zeros(size(X));
12    N = numel(voxelX);
13    for ii=1:N
14        ix = (ux == voxelX(ii));
15        iy = (uy == voxelY(ii));
16        iz = (uz == voxelZ(ii));
17        V(iy,ix,iz) = voxelValues(ii);
18    end
19
20    % Now draw it

```



```

21 ptch = patch(isosurface(X, Y, Z, V, 0.5));
22 isonormals(X, Y, Z, V, ptch)
23 set(ptch, 'FaceColor', 'g', 'EdgeColor', 'none' );
24
25 set(gca, 'DataAspectRatio', [1 1 1]);
26 xlabel('X');
27 ylabel('Y');
28 zlabel('Z');
29 view(-140,22)
30 lighting('gouraud')
31 camlight('left')
32 axis('tight')
33 end

```

Listing 14: Loads Projection Matrices

```

1 function [projectionMatrices] = loadProjectionMatrices()
2     P0 = [1.134783 1.069317 0.046803 347.735102;
3           -0.447199 0.330630 1.035582 -3.804233;
4           0.000382 -0.000339 0.000072 1.000000];
5
6     P1 = [1.303228 0.856019 0.046803 347.735102;
7           -0.382992 0.403262 1.035582 -3.804233;
8           0.000318 -0.000400 0.000072 1.000000];
9
10    P2 = [1.432075 0.616711 0.046803 347.735102;
11          -0.307148 0.463642 1.035582 -3.804233;
12          0.000243 -0.000449 0.000072 1.000000];
13
14    P3 = [1.517410 0.358664 0.046803 347.735102;
15          -0.221971 0.509934 1.035582 -3.804233;
16          0.000162 -0.000484 0.000072 1.000000];
17
18    P4 = [1.556638 0.089720 0.046803 347.735102;
19          -0.130050 0.540731 1.035582 -3.804233;
20          0.000075 -0.000505 0.000072 1.000000];
21
22    P5 = [1.548569 -0.181950 0.046803 347.735102;
23          -0.034177 0.555099 1.035582 -3.804233;
24          -0.000014 -0.000511 0.000072 1.000000];
25
26    P6 = [1.493447 -0.448092 0.046803 347.735102;
27          0.062734 0.552601 1.035582 -3.804233;
28          -0.000102 -0.000500 0.000072 1.000000];
29
30    P7 = [1.392948 -0.700619 0.046803 347.735102;
31          0.157739 0.533312 1.035582 -3.804233
32          -0.000187 -0.000475 0.000072 1.000000];
33
34    P8 = [1.250125 -0.931858 0.046803 347.735102;
35          0.247952 0.497819 1.035582 -3.804233;

```

```

36         -0.000267 -0.000435 0.000072 1.000000];
37
38     P9 = [1.069317 -1.134783 0.046803 347.735102;
39           0.330630 0.447199 1.035582 -3.804233;
40           -0.000339 -0.000382 0.000072 1.000000];
41
42     P10 = [0.856019 -1.303228 0.046803 347.735102;
43            0.403262 0.382992 1.035582 -3.804233;
44            -0.000400 -0.000318 0.000072 1.000000];
45
46     P11 = [0.616711 -1.432075 0.046803 347.735102;
47            0.463642 0.307148 1.035582 -3.804233;
48            -0.000449 -0.000243 0.000072 1.000000];
49
50     P12 = [0.358664 -1.517410 0.046803 347.735102;
51            0.509934 0.221971 1.035582 -3.804233;
52            -0.000484 -0.000162 0.000072 1.000000];
53
54     P13 = [0.089720 -1.556638 0.046803 347.735102;
55            0.540731 0.130050 1.035582 -3.804233;
56            -0.000505 -0.000075 0.000072 1.000000];
57
58     P14 = [-0.181950 -1.548569 0.046803 347.735102;
59            0.555099 0.034177 1.035582 -3.804233;
60            -0.000511 0.000014 0.000072 1.000000];
61
62     P15 = [-0.448092 -1.493447 0.046803 347.735102;
63            0.552601 -0.062734 1.035582 -3.804233;
64            -0.000500 0.000102 0.000072 1.000000];
65
66     P16 = [-0.700619 -1.392948 0.046803 347.735102;
67            0.533312 -0.157739 1.035582 -3.804233;
68            -0.000475 0.000187 0.000072 1.000000];
69
70     P17 = [-0.931858 -1.250125 0.046803 347.735102;
71            0.497819 -0.247952 1.035582 -3.804233;
72            -0.000435 0.000267 0.000072 1.000000];
73
74     P18 = [-1.134783 -1.069317 0.046803 347.735102;
75            0.447199 -0.330630 1.035582 -3.804233;
76            -0.000382 0.000339 0.000072 1.000000];
77
78     P19 = [-1.303228 -0.856019 0.046803 347.735102;
79            0.382992 -0.403262 1.035582 -3.804233;
80            -0.000318 0.000400 0.000072 1.000000];
81
82     P20 = [-1.432075 -0.616711 0.046803 347.735102;
83            0.307148 -0.463642 1.035582 -3.804233;
84            -0.000243 0.000449 0.000072 1.000000];
85

```

```

86 P21 = [-1.517410 -0.358664 0.046803 347.735102;
87 0.221971 -0.509934 1.035582 -3.804233;
88 -0.000162 0.000484 0.000072 1.000000];
89
90 P22 = [-1.556638 -0.089720 0.046803 347.735102;
91 0.130050 -0.540731 1.035582 -3.804233;
92 -0.000075 0.000505 0.000072 1.000000];
93
94 P23 = [-1.548569 0.181950 0.046803 347.735102;
95 0.034177 -0.555099 1.035582 -3.804233;
96 0.000014 0.000511 0.000072 1.000000];
97
98 P24 = [-1.493447 0.448092 0.046803 347.735102;
99 -0.062734 -0.552601 1.035582 -3.804233;
100 0.000102 0.000500 0.000072 1.000000];
101
102 P25 = [-1.392948 0.700619 0.046803 347.735102;
103 -0.157739 -0.533312 1.035582 -3.804233;
104 0.000187 0.000475 0.000072 1.000000];
105
106 P26 = [-1.250125 0.931858 0.046803 347.735102;
107 -0.247952 -0.497819 1.035582 -3.804233;
108 0.000267 0.000435 0.000072 1.000000];
109
110 P27 = [-1.069317 1.134783 0.046803 347.735102;
111 -0.330630 -0.447199 1.035582 -3.804233;
112 0.000339 0.000382 0.000072 1.000000];
113
114 P28 = [-0.856019 1.303228 0.046803 347.735102;
115 -0.403262 -0.382992 1.035582 -3.804233;
116 0.000400 0.000318 0.000072 1.000000];
117
118 P29 = [-0.616711 1.432075 0.046803 347.735102;
119 -0.463642 -0.307148 1.035582 -3.804233;
120 0.000449 0.000243 0.000072 1.000000];
121
122 P30 = [-0.358664 1.517410 0.046803 347.735102;
123 -0.509934 -0.221971 1.035582 -3.804233;
124 0.000484 0.000162 0.000072 1.000000];
125
126 P31 = [-0.089720 1.556638 0.046803 347.735102;
127 -0.540731 -0.130050 1.035582 -3.804233;
128 0.000505 0.000075 0.000072 1.000000];
129
130 P32 = [0.181950 1.548569 0.046803 347.735102;
131 -0.555099 -0.034177 1.035582 -3.804233;
132 0.000511 -0.000014 0.000072 1.000000];
133
134 P33 = [0.448092 1.493447 0.046803 347.735102;
135 -0.552601 0.062734 1.035582 -3.804233;

```

```

136         0.000500 -0.000102 0.000072 1.000000];
137
138     P34 = [0.700619 1.392948 0.046803 347.735102;
139           -0.533312 0.157739 1.035582 -3.804233;
140           0.000475 -0.000187 0.000072 1.000000];
141
142     P35 = [0.931858 1.250125 0.046803 347.735102;
143           -0.497819 0.247952 1.035582 -3.804233;
144           0.000435 -0.000267 0.000072 1.000000];
145
146     projectionMatrices = cat(3, P0, P1, P2, P3, P4, P5, ...
147                               P6, P7, P8, P9, P10, P11, ...
148                               P12, P13, P14, P15, P16, P17, ...
149                               P18, P19, P20, P21, P22, P23, ...
150                               P24, P25, P26, P27, P28, P29, ...
151                               P30, P31, P32, P33, P34, P35);
152
153 end

```

Listing 15: Plots the Camera Locations

```

1  % Load projection matrices
2  projectionMatrices = loadProjectionMatrices();
3
4  % Get camera locations and directions
5  cameraLocations = zeros(size(projectionMatrices,3), 3);
6  cameraDirections = zeros(size(projectionMatrices,3), 3);
7  for i = 1:36
8      [cameraLocations(i,:), cameraDirections(i,:)] = getCameraLocation(
9          projectionMatrices(:,:,i));
10     cameraDirections(i,:) = cameraDirections(i,:) ./ norm(
11         cameraDirections(i,:));
12 end
13
14 quiver3(cameraLocations(:,1), cameraLocations(:,2), -1*cameraLocations
15         (:,3), ...
16         cameraDirections(:,1), cameraDirections(:,2), -1*cameraDirections
17         (:,3), ...
18         0.5);
19
20 axis([-2000 2000 -2000 2000 0 1000])

```

## Eigenface Recognition

Listing 16: EigenfaceRecognizer Class

```

1  classdef EigenfaceRecognizer
2      properties
3          theFaceVectors;
4          theMeanFaceVector;

```

```

5         theEigens;
6         theProjections;
7         theDistanceMetric = 'euclidean';
8     end
9
10    methods
11        function self = EigenfaceRecognizer(images)
12            %Vectorise face images
13            self.theFaceVectors = reshape(images, [], size(images, 3));
14
15            %Compute average face
16            self.theMeanFaceVector = mean(self.theFaceVectors, 2);
17
18            %Standardise face vectors
19            faceDiffVectors = double(self.theFaceVectors) - self.
                theMeanFaceVector;
20
21            %Compute covariance matrix as A*transpose(A)
22            C = faceDiffVectors' * faceDiffVectors;
23
24            %Compute eigenvectors of covariance
25            [eVectors, ~] = eig(C);
26            eVectors = faceDiffVectors * eVectors;
27
28            %Normalise eigenvectors
29            self.theEigens = eVectors ./ vecnorm(eVectors, 2, 1);
30
31            %Characterise faces as linear combination of eigenface basis
32            %vectors
33            self.theProjections = zeros(size(self.theEigens, 2));
34            for faceNum = 1:size(images, 3)
35                self.theProjections(faceNum, :) = sum(self.theEigens .*
                    faceDiffVectors(:, faceNum));
36            end
37        end
38
39        function classification = recognizeFace(self, face)
40            %Vectorise face image
41            faceVector = face(:);
42
43            %Project normalised face
44            faceDiffVector = double(faceVector) - self.theMeanFaceVector;
45            projection = sum(self.theEigens .* faceDiffVector);
46
47            %Recognize face as closest projection
48            distances = pdist2(projection, self.theProjections, self.
                theDistanceMetric);
49            [~, classification] = min(distances);
50        end
51    end

```

52 end

Listing 17: Plots Eigenface PCA Projections

```
1  %Load eigenfaces
2  eigenImages = [];
3
4  for i = 1:6
5      [image, map] = imread(sprintf('faces/eig/%da.bmp', i));
6      image = rgb2gray(ind2rgb(image, map));
7      eigenImages = cat(3, eigenImages, image);
8  end
9
10 %Create eigenface recognizer
11 recognizer = EigenfaceRecognizer(eigenImages);
12
13 %Project images onto two axes using PCA
14 basisVectors = recognizer.theEigens(:,1:2);
15
16 figure();
17 hold on;
18 colours = ['r', 'g', 'b', 'y', 'o', 'c'];
19 projections = [];
20 for i = 1:6
21     groupProjection = [];
22     files = dir(sprintf('faces/%d/*.bmp', i));
23     for j = 1:length(files)
24         [image, map] = imread(sprintf('faces/%d/%s', i, files(j).name));
25         image = rgb2gray(ind2rgb(image, map));
26         faceVector = image(:);
27         faceVector = faceVector - recognizer.theMeanFaceVector;
28         groupProjection = [groupProjection basisVectors' * faceVector];
29     end
30     scatter(groupProjection(1,:), groupProjection(2,:), 52, colours(i), '
        DisplayName', sprintf('Face %d', i));
31 end
32
33 legend;
34 xlabel('Principal Component 1');
35 ylabel('Principal Component 2');
36 hold off;
```