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

```
function [silhouette] = createDinoSilhouette(image)
    % Threshold in HSV space
    I = rgb2hsv(image);
    minHSV = 0.792;
    maxHSV = 0.566;
    mask = (I(:,:,1) >= minHSV) | (I(:,:,1) <= maxHSV);

    %Select largest component
    silhouette = zeros(size(mask));
    stats = regionprops('table', mask, 'Area', 'PixelIdxList');
    [~, sortingIndex] = sort(stats.Area, 'descend');
    silhouette(stats.PixelIdxList{sortingIndex(1)}) = 1;
end
```

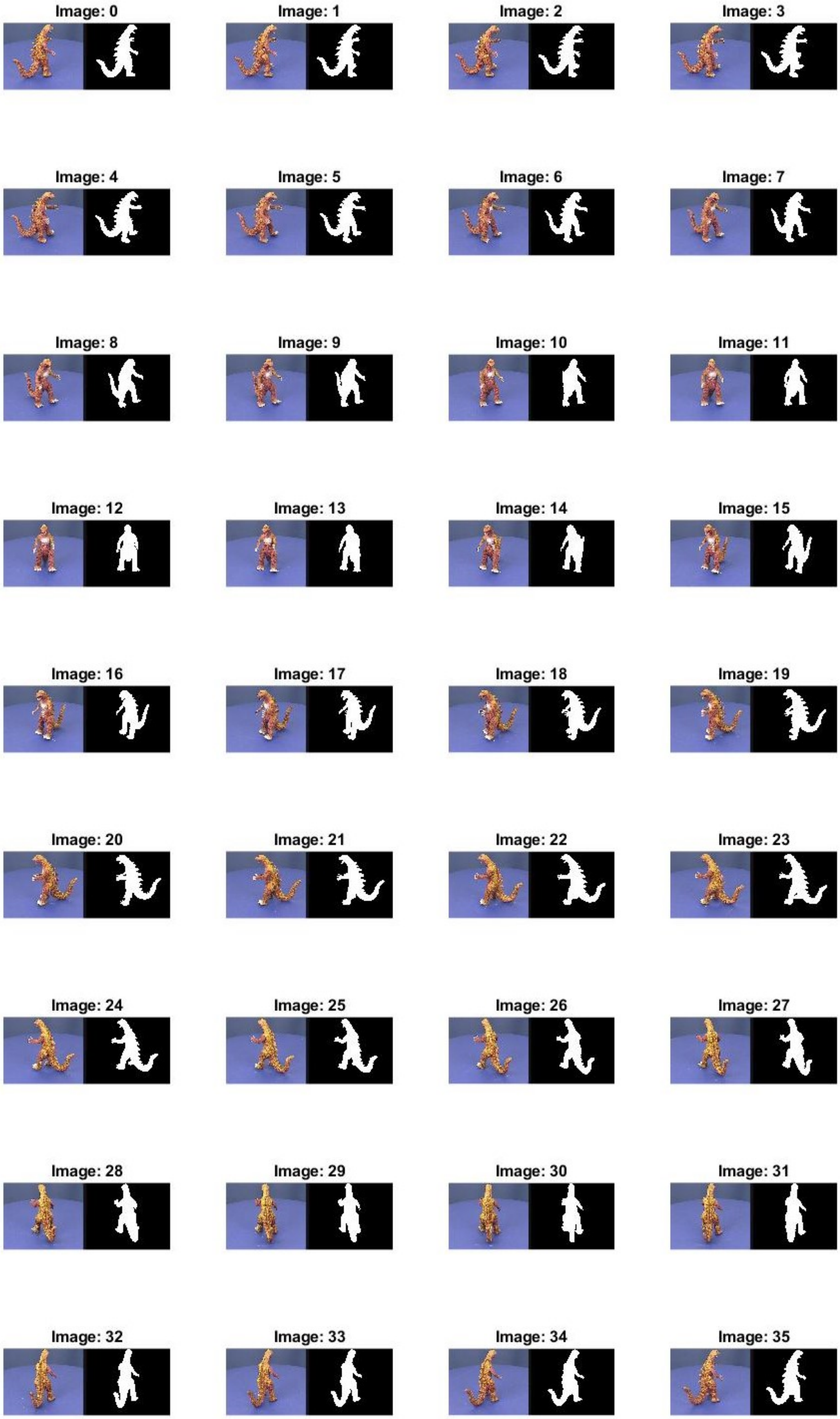


Figure 1: Dinosaur Silhouettes using HSV-domain thresholding

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

```
% Create voxel space
RESOLUTION = 2;
voxelRanges = [-180 90; ... %X range
                -80 70; ... %Y range
                 20 460];   %Z range
[voxelX, voxelY, voxelZ, voxelValues] = initialiseVoxels(voxelRanges,
   RESOLUTION);


function [voxelX, voxelY, voxelZ, voxelValues] = initialiseVoxels(
   voxelRanges, resolution)
    x = voxelRanges(1,1):resolution:voxelRanges(1,2);
    y = voxelRanges(2,1):resolution:voxelRanges(2,2);
    z = voxelRanges(3,1):resolution:voxelRanges(3,2);

    [X, Y, Z] = meshgrid(x, y, z);
    voxelX = X(:);
    voxelY = Y(:);
    voxelZ = Z(:);
    voxelValues = ones(numel(X), 1);
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

```
function [imageX, imageY] = projectOntoImage(projectionMatrix, voxelX,
   voxelY, voxelZ)
    z = projectionMatrix(3,1) * voxelX ...
      + projectionMatrix(3,2) * voxelY...
      + projectionMatrix(3,3) * voxelZ ...
      + projectionMatrix(3,4);

    imageX = round((projectionMatrix(1,1) * voxelX ...
                  + projectionMatrix(1,2) * voxelY...
                  + projectionMatrix(1,3) * voxelZ ...
                  + projectionMatrix(1,4)) ./ z);

```

```
    imageY = round((projectionMatrix(2,1) * voxelX ...
                 + projectionMatrix(2,2) * voxelY...
                 + projectionMatrix(2,3) * voxelZ ...
                 + projectionMatrix(2,4)) ./ z);
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

```
%Carve voxel space from each angle
for i = 1:36
    %Get silhouette
    image = imread(sprintf('dino/dino%02d.jpg', i-1));
    silhouette = createDinoSilhouette(image);

    %Project voxels onto silhouette
    [imageX, imageY] = projectOntoImage(projectionMatrices(:,:,i), voxelX
       , voxelY, voxelZ);

    %Crop any projection outside image
    [imageHeight, imageWidth,~] = size(image);
    cropMask = find((imageX >= 1) & (imageX <= imageWidth) & ...
                (imageY >= 1) & (imageY <= imageHeight));
    imageX = imageX(cropMask);
    imageY = imageY(cropMask);

    % Create silhouette mask
    imageIndices = sub2ind([imageHeight,imageWidth], round(imageY), round
       (imageX));
    silhouetteMask = cropMask(silhouette(imageIndices) >= 1);

    % Carve silhouette into voxels
    voxelX = voxelX(silhouetteMask);
    voxelY = voxelY(silhouetteMask);
    voxelZ = voxelZ(silhouetteMask);
    voxelValues = voxelValues(silhouetteMask);

    % Show Result
    figure('Position', [100 100 600 300]);
    plotDino(voxelX, voxelY, voxelZ, voxelValues);
end
```

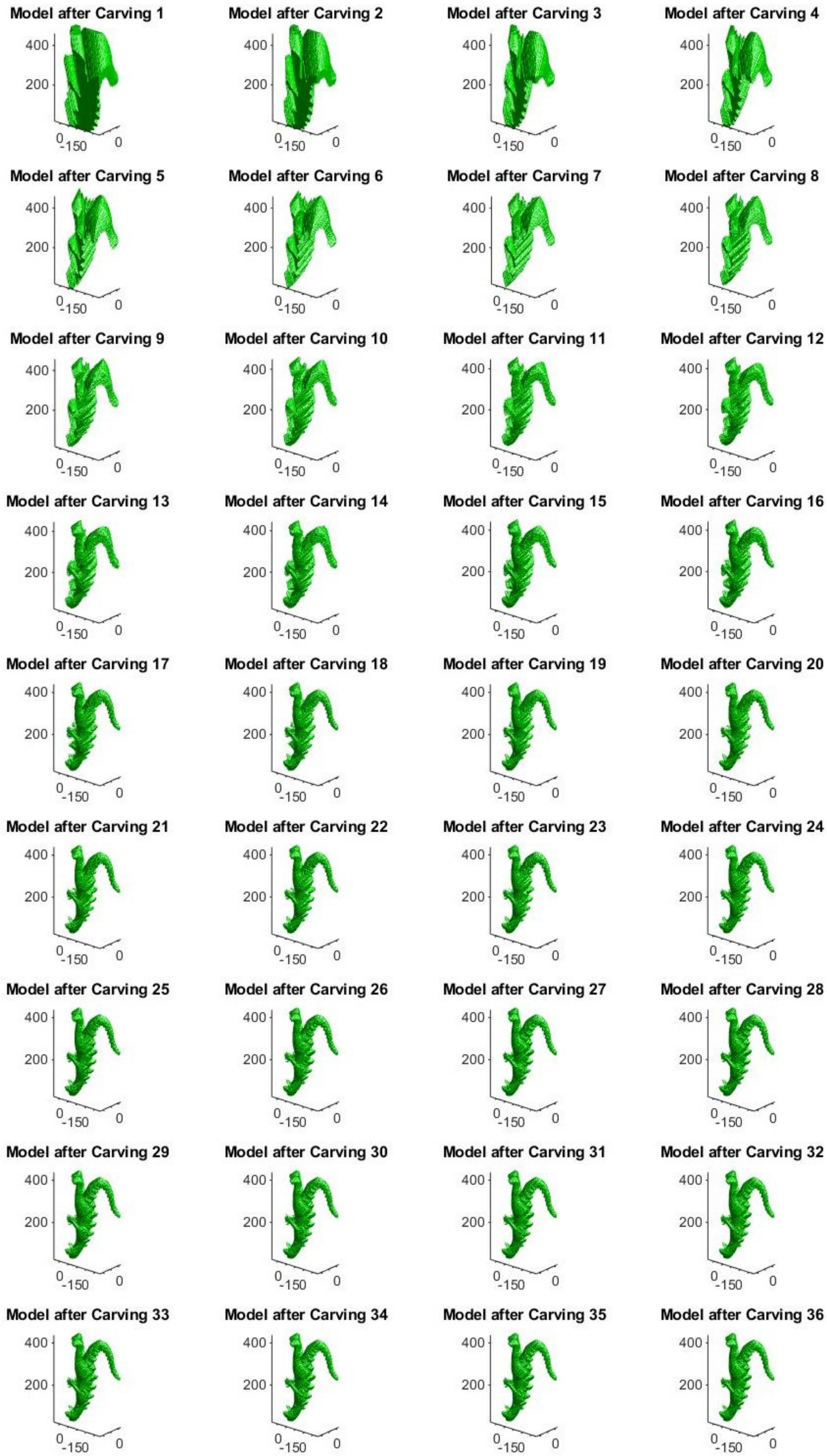


Figure 2: Carving Method after each projection

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

```
% Vote for voxels from each angle
for i = 1:36
    %Get silhouette
    image = imread(sprintf('dino/dino%02d.jpg', i-1));
    silhouette = createDinoSilhouette(image);

    %Project voxels onto image
    [projectedX, projectedY] = projectOntoImage(projectionMatrices(:,:,i), voxelX, voxelY, voxelZ);

    %Vote
    [imageHeight, imageWidth,~] = size(image);
    for i = 1:length(projectedX)
        if(projectedY(i) <= imageHeight && projectedY(i) > 0 &&...
           projectedX(i) <= imageWidth && projectedX(i) > 0 &&...
           silhouette(projectedY(i), projectedX(i)) >= 1)
              voxelValues(i) = voxelValues(i) + 1;
        end
   end
end

figure();
votingThresholds = [0 5 10 15 20 25 30 35 36 40
for i = 1:10
    % 'Carve' using voting threshold
    thresholdedVoxelX = voxelX(voxelValues >= votingThresholds(i));
    thresholdedVoxelY = voxelY(voxelValues >= votingThresholds(i));
    thresholdedVoxelZ = voxelZ(voxelValues >= votingThresholds(i));
    thresholdedVoxelValues = voxelValues(voxelValues >= votingThresholds(i));

    % Show Result
    subplot(5,2,i);
    plotDino(thresholdedVoxelX, thresholdedVoxelY, thresholdedVoxelZ, thresholdedVoxelValues);
    title(sprintf('Voting Threshold = %d', votingThresholds(i)));
end
```

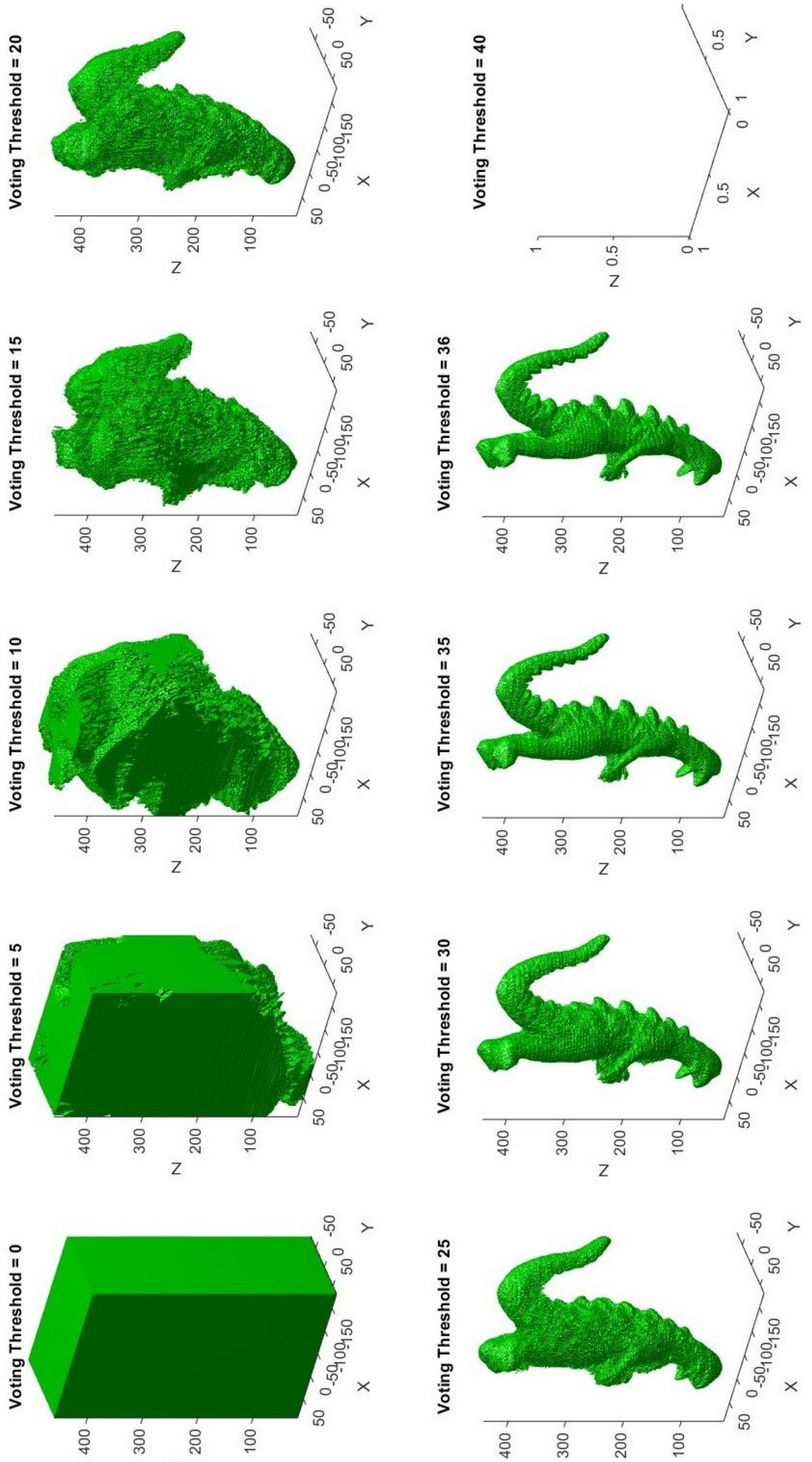
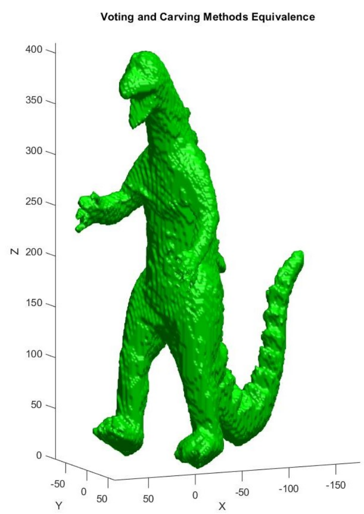


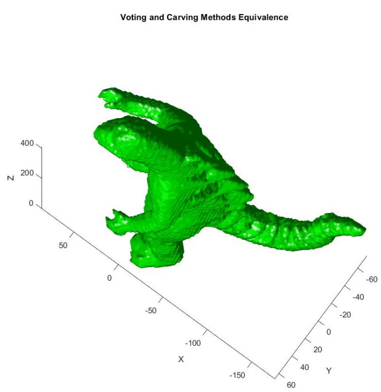
Figure 3: Various thresholds on the number of votes

1.3 Results

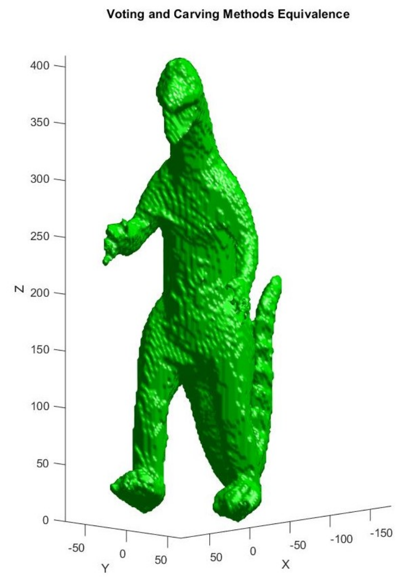
The equivalent dinosaur from both the voting and carving methods is shown from various angles in Figure 4.



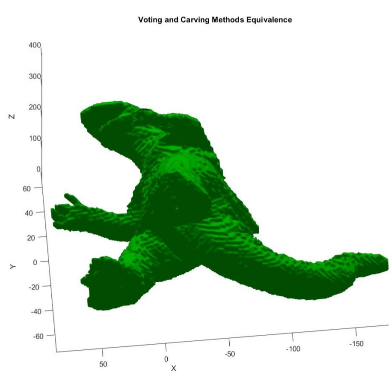
(a) Angle 1



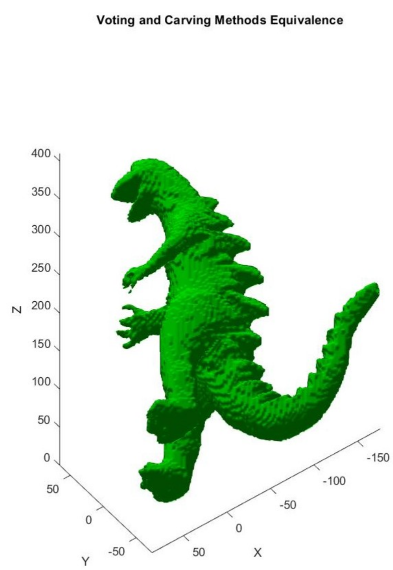
(b) Angle 2



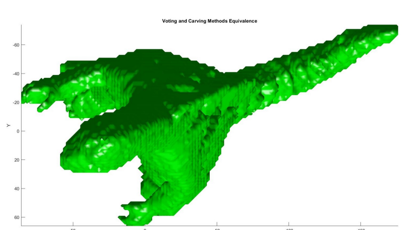
(c) Angle 3



(d) Angle 4



(e) Angle 5



(f) Angle 6

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

```
function [location, direction] = getCameraLocation(projectionMatrix)
    [q,r] = qr(inv(projectionMatrix(1:3,1:3)));
    invK = r(1:3,1:3);
    R = inv(q);
    if det(R) < 0
        R = -R;
        invK = -invK;
    end
    t = invK*projectionMatrix(:,4);

    location = -q * t;

    imageCenter = [360; 288; 1];
    X = R' * (invK * imageCenter);
    direction = X ./ norm(X);
end
```

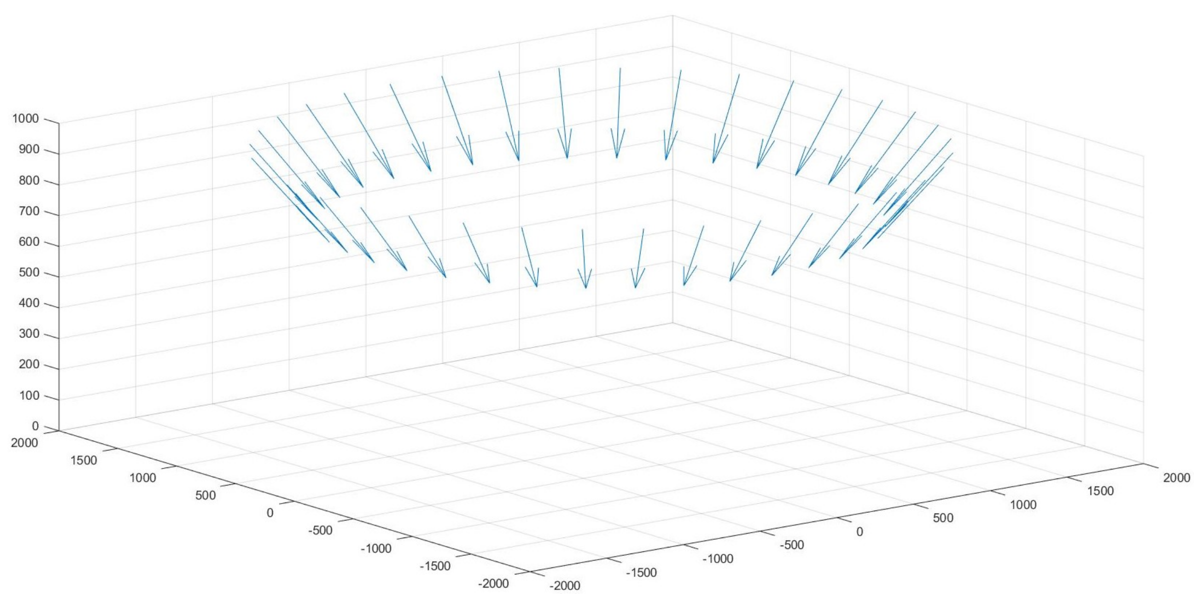


Figure 5: Location and Direction of Cameras

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

```
function colourDinoUsingDistance(p, images, silhouettes,
    projectionMatrices, cameraLocations)
    vertices = get(p, 'Vertices');
    nc = uint8(zeros(size(vertices, 1), 3));

    % For each voxel...
    for i = 1:size(vertices, 1)
        % Calculate euclidean distance between all camera locations
        [~, closestCameraIndex] = pdist2(cameraLocations, vertices(i, :),
            'squaredeuclidean', 'Smallest', 1);
        % Project onto closest camera's image
        [imageX, imageY] = projectOntoImage(projectionMatrices(:,:,
            closestCameraIndex), vertices(i,1), vertices(i,2), vertices(i
            ,3));
        % Assign colour from projected pixel
        image = images(:,:,:,closestCameraIndex);
        nc(i,:) = reshape((image(imageY, imageX, :)), 1, 3);
    end

    set(p, 'FaceVertexCData', nc, 'FaceColor', 'interp');
    p.EdgeColor = 'none';
end
```

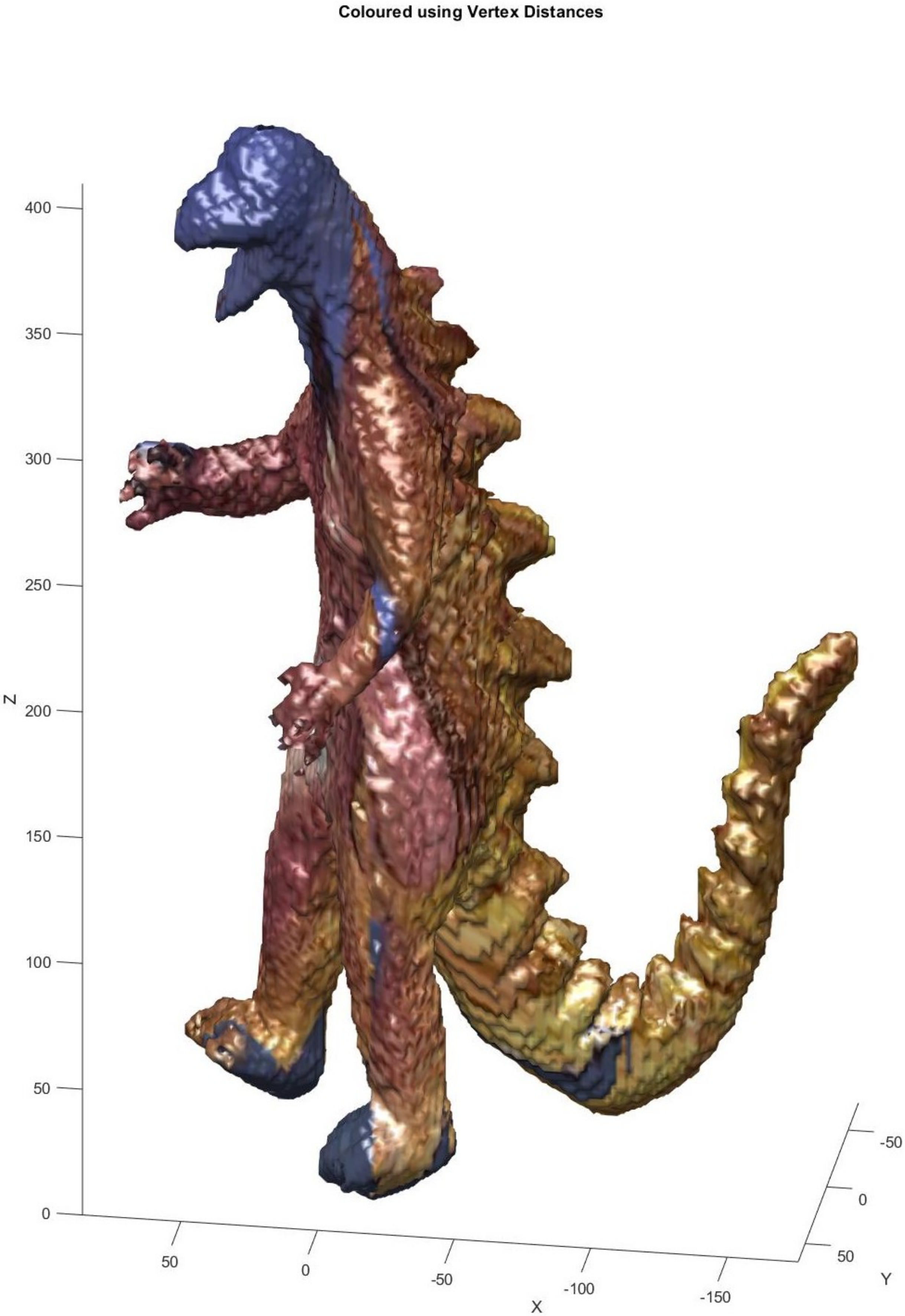


Figure 6: Colouring of Dinosaur using Minimum Euclidean Distance

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

```
function colourDinoUsingNormals(p, images, silhouettes,
   projectionMatrices, cameraDirections)

    vertices = get(p, 'Vertices');
    vertexNormals = get(p, 'VertexNormals');
    nc = uint8(zeros(size(vertexNormals, 1), 3));

    % For each voxel...
    for i = 1:size(vertexNormals, 1)
        % Calculated dot products
        angles = vertexNormals(i,:)*cameraDirections'./norm(vertexNormals
           (i,:));
        % Find minimum angle
        [~, cameraOrder] = sort(angles, 'ascend');
        cameraIndex = 1;
        % Project onto that camera
        [imageX, imageY] = projectOntoImage(projectionMatrices(:,:,
           cameraOrder(cameraIndex)), vertices(i,1), vertices(i,2),
           vertices(i,3));
        while((imageX < 1) || (imageX > size(images,2)) || (imageY < 1)
           || (imageY > size(images, 1)))
            cameraIndex = cameraIndex + 1;
            [imageX, imageY] = projectOntoImage(projectionMatrices(:,:,
               cameraOrder(cameraIndex)), vertices(i,1), vertices(i,2),
               vertices(i,3));
        end
        % Assign colour from projected pixel
        image = images(:,:,:,cameraOrder(cameraIndex));
        nc(i,:) = reshape((image(imageY, imageX, :)), 1, 3);
    end

    set(p, 'FaceVertexCData', nc, 'FaceColor', 'interp');
    p.EdgeColor = 'none';
end
```

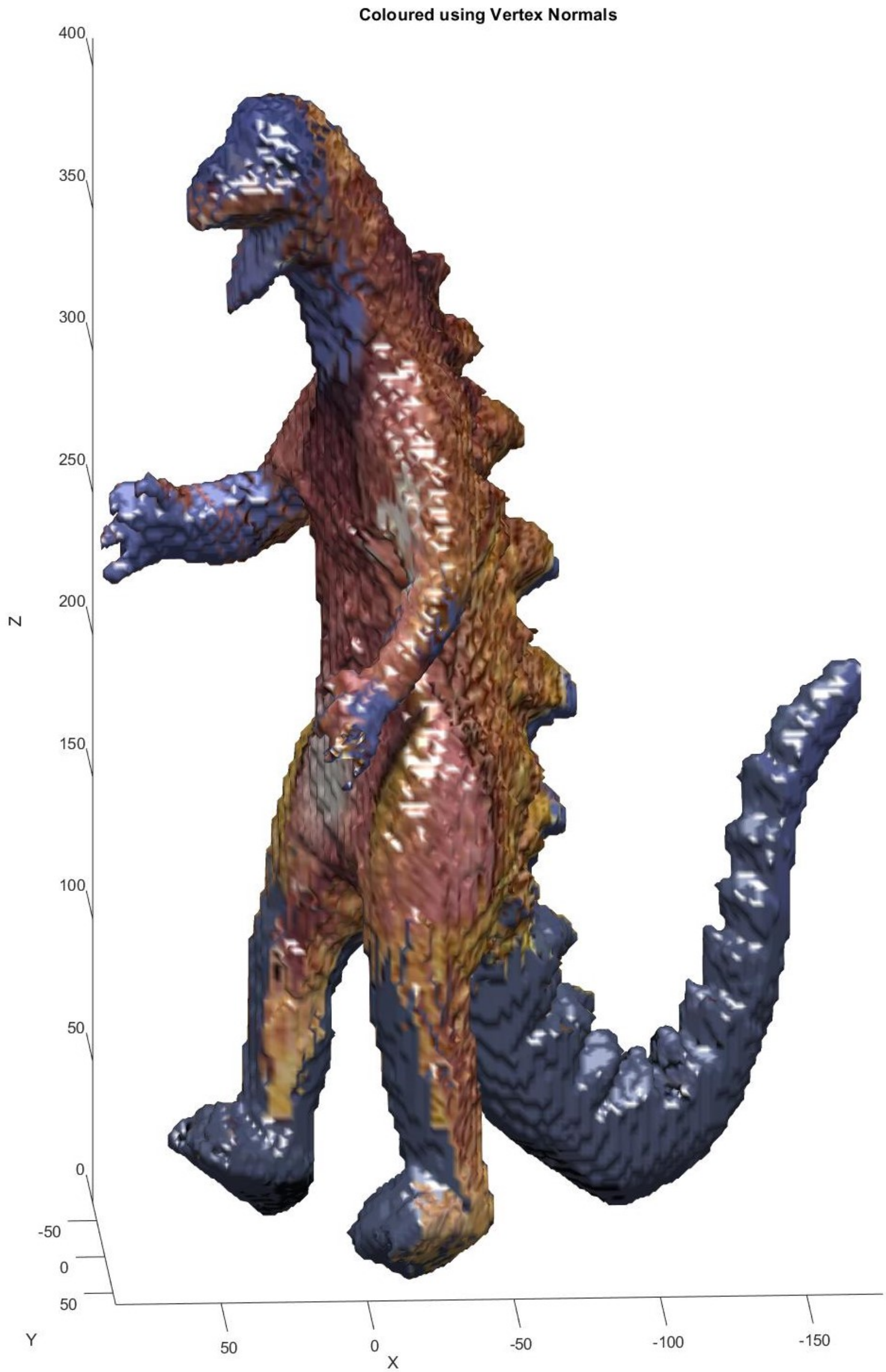


Figure 7: Colouring of Dinosaur using Maximum Normal Alignment

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, Ψ, was computed from the image vectors, Γ by: $\Psi = \frac{1}{M}\sum_{i=1}^{M}\Gamma_i$
3. The face images were standardised, Φ, 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 $A = [\Phi_1, \Phi_2, \Phi_3, ..., \Phi_M]$. 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, μ_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

```
        function self = EigenfaceRecognizer(images)
            %Vectorise face images
            self.theFaceVectors = reshape(images, [], size(images, 3));

            %Compute average face
            self.theMeanFaceVector = mean(self.theFaceVectors, 2);

            %Standardise face vectors
            faceDiffVectors = double(self.theFaceVectors) - self.
               theMeanFaceVector;

```

```
            %Compute covariance matrix as A*transpose(A)
            C = faceDiffVectors' * faceDiffVectors;

            %Compute eigenvectors of covariance
            [eVectors, ~] = eig(C);
            eVectors = faceDiffVectors * eVectors;

            %Normalise eigenvectors
            self.theEigens = eVectors ./ vecnorm(eVectors, 2, 1);

            %Characterise faces as linear combination of eigenface basis
            %vectors
            self.theProjections = zeros(size(self.theEigens, 2));
            for faceNum = 1:size(images, 3)
                self.theProjections(faceNum, :) = sum(self.theEigens .*
                    faceDiffVectors(:,faceNum));
            end
        end
```

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

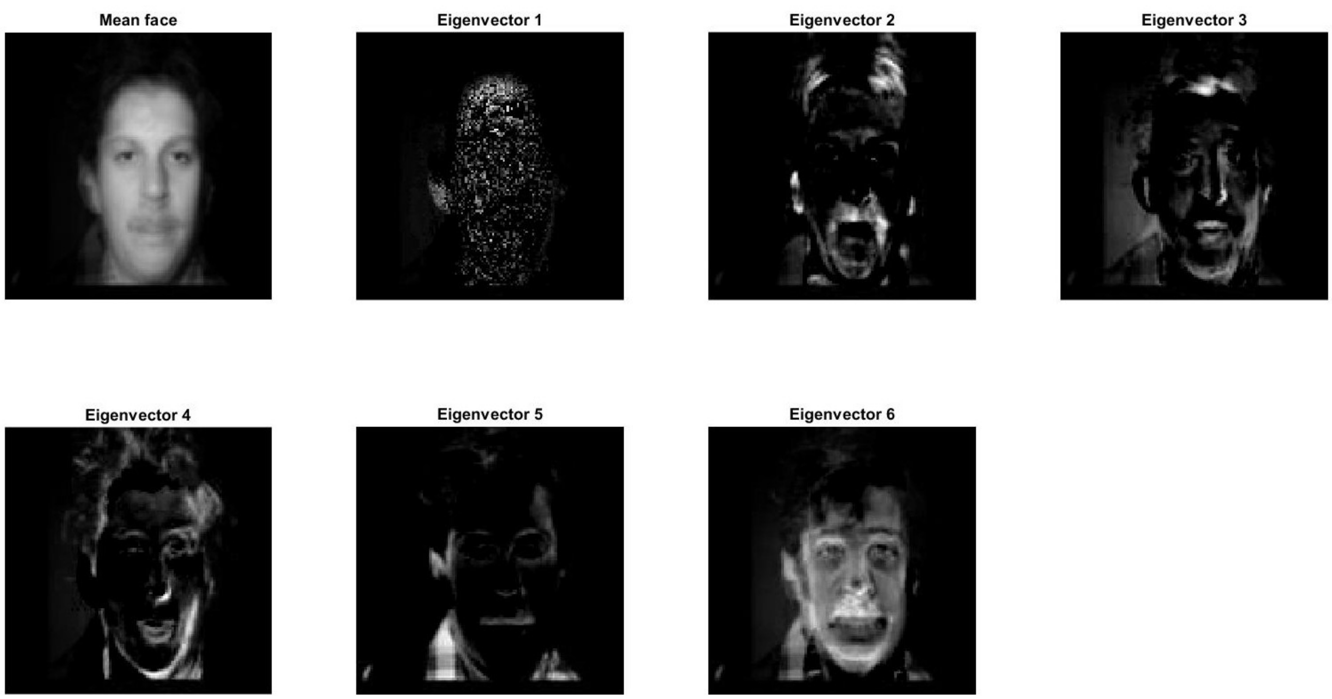


Figure 8: Mean Face and Computed Eigenfaces

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.

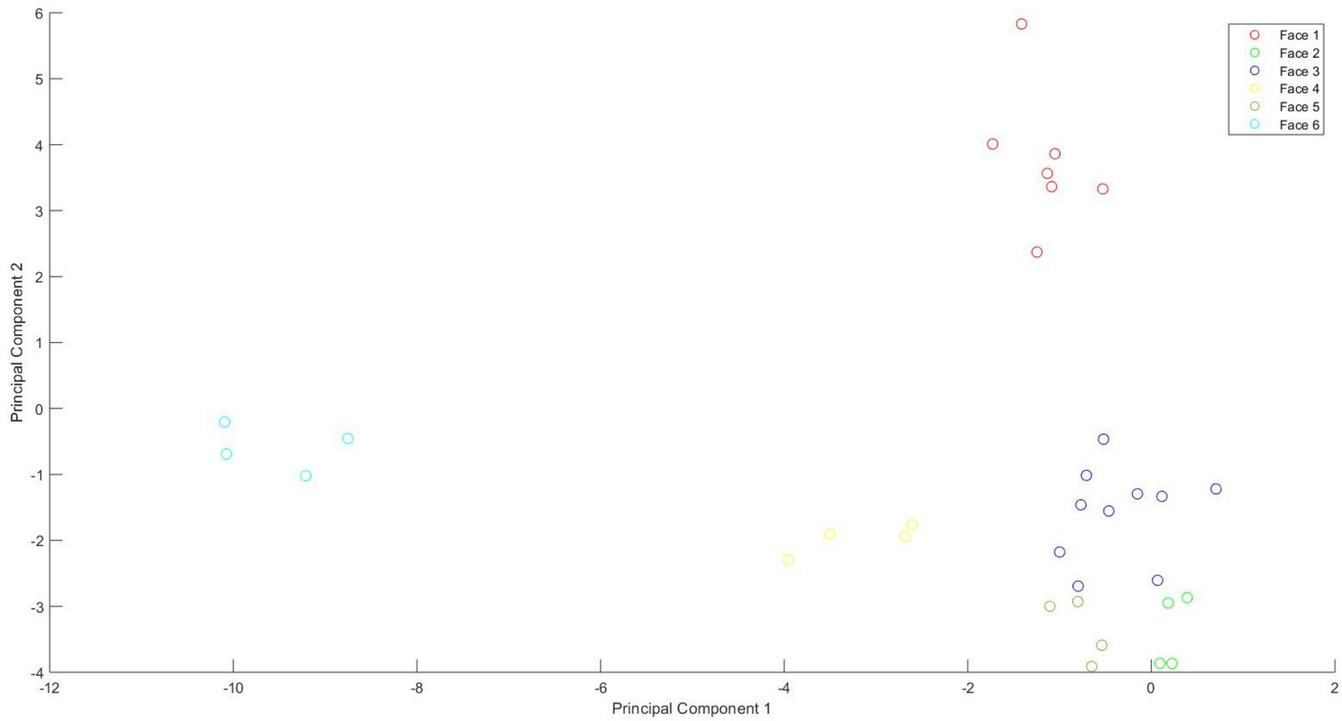


Figure 9: Face Images projected onto two principal components

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 Γ. This image was preprocessed through centering, scaling and grayscaling

2. The face image, Φ, was standardised, using the average face previously computed, by: $\Phi = \Gamma - \Psi$

3. The face image is projected onto the eigenface basis vectors: $\Omega = proj_{\mu_i}\Phi$

4. The projection weightings, Ω, are compared to the test image weightings. Specifically, the distances between Ω 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

```
        function classification = recognizeFace(self, face)
            %Vectorise face image
            faceVector = face(:);

            %Project normalised face
            faceDiffVector = double(faceVector) - self.theMeanFaceVector;
            projection = sum(self.theEigens .* faceDiffVector);
```

```
        %Recognize face as closest projection
        distances  = pdist2(projection, self.theProjections, self.
           theDistanceMetric);
        [~, 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

```
function GUI
    %Create figure
    figHeight = 700;
    figWidth = 1400;
    f = figure('Visible','off','Position',[25, 50, figWidth, figHeight]);

    %Load eigenfaces
    images = [];
    imagesAxes = [];
    subplotNums = [1 2 3 7 8 9];
    for i = 1:6
        hold on;
        [image, map] = imread(sprintf('faces/eig/%da.bmp', i));
        image = rgb2gray(ind2rgb(image, map));
        images = cat(3, images, image);
        imagesAxes = [imagesAxes subplot(2,6,subplotNums(i))];
        imshow(images(:,:,i), [0 max(max(images(:,:,i)))]);
        title(sprintf('Face %d', i));
    end
    hold off;

    %Create eigenface recognizer
    recognizer = EigenfaceRecognizer(images);

    % Test Image plot
    testImageAxes = subplot(2, 6, [4 5 6 10 11 12]);

    %Create menubar
    menubar = uimenu(f, 'Text', 'File');
    newFileMenu = uimenu(menubar, 'Text', 'New File', 'MenuSelectedFcn',
       @newFileCallback);
    closeMenu = uimenu(menubar, 'Text', 'Exit', 'MenuSelectedFcn',
       @closeCallback);

```

```
    function updateImageClasses(classification)
        for i = 1:6
            axes(imagesAxes(i))
            if i ~= classification
                imshow(0.25 * images(:,:,i), [0 max(max(images(:,:,i)))]);
                title(sprintf('Face %d', i));
            else
                imshow(images(:,:,i), [0 max(max(images(:,:,i)))]);
                title(sprintf('Face %d', i));
            end
            hold on;
        end
        hold off;
    end

    function newFileCallback(~, ~)
        [file, path] = uigetfile("*.bmp");
        if ~isequal(file, 0)
            [testImage, map] = imread(fullfile(path, file));
            testImage = rgb2gray(ind2rgb(testImage, map));

            axes(testImageAxes)
            imshow(testImage, [0 max(testImage(:))]);
            title(sprintf('Test Image: %s', file));

            classification = recognizer.recognizeFace(testImage);
            updateImageClasses(classification);
        end
    end

    function closeCallback(~, ~)
        close all
    end

    newFileCallback(0, 0);

    % Make the UI visible.
    f.Visible = 'on';
end
```

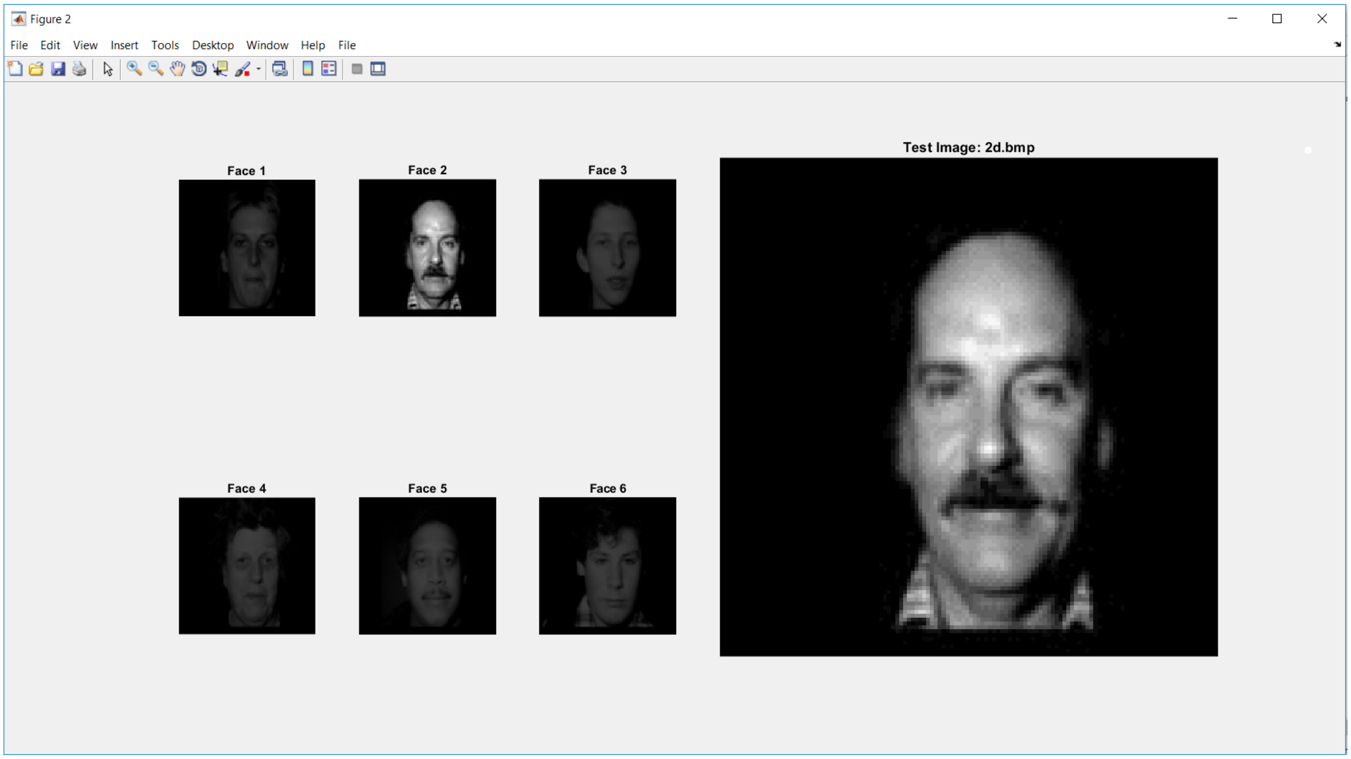


Figure 10: Facial Recognition GUI

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

```
% Load projection matrices
projectionMatrices = loadProjectionMatrices();

%%%%Carving Method
% Create voxel space
RESOLUTION = 2;
voxelRanges = [-180 90; ... %X range
                -80 70; ... %Y range
                 20 460];   %Z range
[voxelX, voxelY, voxelZ, voxelValues] = initialiseVoxels(voxelRanges,
    RESOLUTION);

%Carve voxel space from each angle
for i = 1:36
    %Get silhouette
    image = imread(sprintf('dino/dino%02d.jpg', i-1));
    silhouette = createDinoSilhouette(image);

    %Project voxels onto silhouette
    [imageX, imageY] = projectOntoImage(projectionMatrices(:,:,i), voxelX
        , voxelY, voxelZ);

    %Crop any projection outside image
    [imageHeight, imageWidth,~] = size(image);
    cropMask = find((imageX >= 1) & (imageX <= imageWidth) & ...
                    (imageY >= 1) & (imageY <= imageHeight));
    imageX = imageX(cropMask);
    imageY = imageY(cropMask);

    % Create silhouette mask
    imageIndices = sub2ind([imageHeight,imageWidth], round(imageY), round
        (imageX));
    silhouetteMask = cropMask(silhouette(imageIndices) >= 1);

    % Carve silhouette into voxels
    voxelX = voxelX(silhouetteMask);
    voxelY = voxelY(silhouetteMask);
    voxelZ = voxelZ(silhouetteMask);
    voxelValues = voxelValues(silhouetteMask);

    % Show Result
    figure('Position', [100 100 600 300]);
    plotDino(voxelX, voxelY, voxelZ, voxelValues);
end
```

```
%%%%Voting Method
% Create voxel space
RESOLUTION = 2;
voxelRanges = [-180 90; ... %X range
               -80 70; ... %Y range
                20 460];    %Z range
[voxelX, voxelY, voxelZ, voxelValues] = initialiseVoxels(voxelRanges,
    RESOLUTION);
voxelColours = uint8(zeros(length(voxelValues), 3));

% Vote for voxels from each angle
for i = 1:36
    %Get silhouette
    image = imread(sprintf('dino/dino%02d.jpg', i-1));
    silhouette = createDinoSilhouette(image);

    %Project voxels onto image
    [projectedX, projectedY] = projectOntoImage(projectionMatrices(:,:,i)
        , voxelX, voxelY, voxelZ);

    %Vote
    [imageHeight, imageWidth,~] = size(image);
    for i = 1:length(projectedX)
        if(projectedY(i) <= imageHeight && projectedY(i) > 0 &&...
           projectedX(i) <= imageWidth && projectedX(i) > 0 &&...
           silhouette(projectedY(i), projectedX(i)) >= 1)
              voxelValues(i) = voxelValues(i) + 1;
        end
   end
end

figure();
votingThresholds = [0 5 10 15 20 25 30 35 36]; %40
for i = 1:9
    % 'Carve' using voting threshold
    thresholdedVoxelX = voxelX(voxelValues >= votingThresholds(i));
    thresholdedVoxelY = voxelY(voxelValues >= votingThresholds(i));
    thresholdedVoxelZ = voxelZ(voxelValues >= votingThresholds(i));
    thresholdedVoxelValues = voxelValues(voxelValues >= votingThresholds(
        i));

    % Show Result
    subplot(5,2,i);
    plotDino(thresholdedVoxelX, thresholdedVoxelY, thresholdedVoxelZ,
        thresholdedVoxelValues);
    title(sprintf('Voting Threshold = %d', votingThresholds(i)));
end

%Plot carving method equivalent
thresholdedVoxelX = voxelX .* (voxelValues >= 36);
```

```
thresholdedVoxelY = voxelY .* (voxelValues >= 36);
thresholdedVoxelZ = voxelZ .* (voxelValues >= 36);
thresholdedVoxelColours = voxelColours .* uint8(repmat(voxelValues >= 36,
    1, 3));
thresholdedVoxelValues = voxelValues .* (voxelValues >= 36);

%Correct upside-downness
thresholdedVoxelZ = max(thresholdedVoxelZ(:)) - thresholdedVoxelZ;

% Show Result
figure();
p = plotDino(thresholdedVoxelX, thresholdedVoxelY, thresholdedVoxelZ,
   thresholdedVoxelValues);
% plotColourDino(thresholdedVoxelX, thresholdedVoxelY, thresholdedVoxelZ,
    thresholdedVoxelValues, thresholdedVoxelColours, RESOLUTION);
title('Voting and Carving Methods Equivalence');

% Colour in dino - well, try to
images = [];
silhouettes = [];
cameraLocations = zeros(size(projectionMatrices,3), 3);
cameraDirections = zeros(size(projectionMatrices,3), 3);
for i = 1:36
    images = cat(4, images, imread(sprintf('dino/dino%02d.jpg', i-1)));
    silhouettes = cat(3, silhouettes, createDinoSilhouette(images(:,:,:,i
       )));
    [cameraLocations(i,:), cameraDirections(i,:)] = getCameraLocation(
       projectionMatrices(:,:,i));
end

% Show Result
figure();
p = plotDino(thresholdedVoxelX, thresholdedVoxelY, thresholdedVoxelZ,
   thresholdedVoxelValues);
colourDinoUsingDistance(p, images, silhouettes, projectionMatrices,
   cameraLocations);
title('Coloured using Vertex Distances');

figure();
p = plotDino(thresholdedVoxelX, thresholdedVoxelY, thresholdedVoxelZ,
   thresholdedVoxelValues);
colourDinoUsingNormals(p, images, silhouettes, projectionMatrices,
   cameraDirections);
title('Coloured using Vertex Normals');

%%%%General Functions
function [voxelX, voxelY, voxelZ, voxelValues] = initialiseVoxels(
   voxelRanges, resolution)
    x = voxelRanges(1,1):resolution:voxelRanges(1,2);
    y = voxelRanges(2,1):resolution:voxelRanges(2,2);
```

```
    z = voxelRanges(3,1):resolution:voxelRanges(3,2);

    [X, Y, Z] = meshgrid(x, y, z);
    voxelX = X(:);
    voxelY = Y(:);
    voxelZ = Z(:);
    voxelValues = ones(numel(X), 1);
end

function [imageX, imageY] = projectOntoImage(projectionMatrix, voxelX,
   voxelY, voxelZ)
    z = projectionMatrix(3,1) * voxelX ...
      + projectionMatrix(3,2) * voxelY...
      + projectionMatrix(3,3) * voxelZ ...
      + projectionMatrix(3,4);

    imageX = round((projectionMatrix(1,1) * voxelX ...
                 + projectionMatrix(1,2) * voxelY...
                 + projectionMatrix(1,3) * voxelZ ...
                 + projectionMatrix(1,4)) ./ z);

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

function [ptch] = plotDino(voxelX, voxelY, voxelZ, voxelValues)
    % First grid the data
    ux = unique(voxelX);
    uy = unique(voxelY);
    uz = unique(voxelZ);

    % Convert to a grid
    [X,Y,Z] = meshgrid(ux, uy, uz);

    % Create an empty voxel grid, then fill only those elements in voxels
    V = zeros(size(X));
    N = numel(voxelX);
    for ii=1:N
        ix = (ux == voxelX(ii));
        iy = (uy == voxelY(ii));
        iz = (uz == voxelZ(ii));
        V(iy,ix,iz) = voxelValues(ii);
    end

    % Now draw it
    ptch = patch(isosurface(X, Y, Z, V, 0.5));
    isonormals(X, Y, Z, V, ptch)
    set(ptch, 'FaceColor', 'g', 'EdgeColor', 'none');
```

```

    set(gca,'DataAspectRatio',[1 1 1]);
    xlabel('X');
    ylabel('Y');
    zlabel('Z');
    view(-140,22)
    lighting('gouraud')
    camlight('left')
    axis('tight')
end

function colourDinoUsingDistance(p, images, silhouettes,
   projectionMatrices, cameraLocations)
    vertices = get(p, 'Vertices');
    nc = uint8(zeros(size(vertices, 1), 3));

    for i = 1:size(vertices, 1)
        [~, closestCameraIndex] = pdist2(cameraLocations, vertices(i, :),
            'squaredeuclidean', 'Smallest', 1);
        [imageX, imageY] = projectOntoImage(projectionMatrices(:,:,
           closestCameraIndex), vertices(i,1), vertices(i,2), vertices(i
           ,3));
        image = images(:,:,:,closestCameraIndex);
        nc(i,:) = reshape((image(imageY, imageX, :)), 1, 3);
    end

    set(p, 'FaceVertexCData', nc, 'FaceColor', 'interp');
    p.EdgeColor = 'none';
end

function colourDinoUsingNormals(p, images, silhouettes,
   projectionMatrices, cameraDirections)

    vertices = get(p, 'Vertices');
    vertexNormals = get(p, 'VertexNormals');
    nc = uint8(zeros(size(vertexNormals, 1), 3));

    for i = 1:size(vertexNormals, 1)
        angles = vertexNormals(i,:)*cameraDirections'./norm(vertexNormals
           (i,:));
        [~, cameraOrder] = sort(angles, 'ascend');
        cameraIndex = 1;
        [imageX, imageY] = projectOntoImage(projectionMatrices(:,:,
           cameraOrder(cameraIndex)), vertices(i,1), vertices(i,2),
           vertices(i,3));
        while((imageX < 1) || (imageX > size(images,2)) || (imageY < 1)
           || (imageY > size(images, 1)))
            cameraIndex = cameraIndex + 1;
            [imageX, imageY] = projectOntoImage(projectionMatrices(:,:,
               cameraOrder(cameraIndex)), vertices(i,1), vertices(i,2),
```

```
                vertices(i,3));
        end
        yeet = images(:,:,:,cameraOrder(cameraIndex));
        nc(i,:) = reshape((yeet(imageY, imageX, :)), 1, 3);
    end

    set(p, 'FaceVertexCData', nc, 'FaceColor', 'interp');
    p.EdgeColor = 'none';
end

function [location, direction] = getCameraLocation(projectionMatrix)
    [q,r] = qr(inv(projectionMatrix(1:3,1:3)));
    invK = r(1:3,1:3);
    R = inv(q);

    if det( R ) < 0
        R = -R;
        invK = -invK;
    end
    t = invK*projectionMatrix(:,4);

    location = -q * t;

    imageCenter = [360; 288; 1];
    X = R' * (invK * imageCenter);
    direction = X ./ norm(X);

end
```

Listing 13: Plots the Volumetric Model of a Dinosaur

```
function [ptch] = plotDino(voxelX, voxelY, voxelZ, voxelValues)
    % First grid the data
    ux = unique(voxelX);
    uy = unique(voxelY);
    uz = unique(voxelZ);

    % Convert to a grid
    [X,Y,Z] = meshgrid(ux, uy, uz);

    % Create an empty voxel grid, then fill only those elements in voxels
    V = zeros(size(X));
    N = numel(voxelX);
    for ii=1:N
        ix = (ux == voxelX(ii));
        iy = (uy == voxelY(ii));
        iz = (uz == voxelZ(ii));
        V(iy,ix,iz) = voxelValues(ii);
    end

    % Now draw it
```

```
    ptch = patch(isosurface(X, Y, Z, V, 0.5));
    isonormals(X, Y, Z, V, ptch)
    set(ptch, 'FaceColor', 'g', 'EdgeColor', 'none' );

    set(gca,'DataAspectRatio',[1 1 1]);
    xlabel('X');
    ylabel('Y');
    zlabel('Z');
    view(-140,22)
    lighting('gouraud')
    camlight('left')
    axis('tight')
end
```

Listing 14: Loads Projection Matrices

```
function [projectionMatrices] = loadProjectionMatrices()
    P0 = [1.134783 1.069317 0.046803 347.735102;
          -0.447199 0.330630 1.035582 -3.804233;
          0.000382 -0.000339 0.000072 1.000000];

    P1 = [1.303228 0.856019 0.046803 347.735102;
          -0.382992 0.403262 1.035582 -3.804233;
          0.000318 -0.000400 0.000072 1.000000];

    P2 = [1.432075 0.616711 0.046803 347.735102;
          -0.307148 0.463642 1.035582 -3.804233;
          0.000243 -0.000449 0.000072 1.000000];

    P3 = [1.517410 0.358664 0.046803 347.735102;
          -0.221971 0.509934 1.035582 -3.804233;
          0.000162 -0.000484 0.000072 1.000000];

    P4 = [1.556638 0.089720 0.046803 347.735102;
          -0.130050 0.540731 1.035582 -3.804233;
          0.000075 -0.000505 0.000072 1.000000];

    P5 = [1.548569 -0.181950 0.046803 347.735102;
          -0.034177 0.555099 1.035582 -3.804233;
          -0.000014 -0.000511 0.000072 1.000000];

    P6 = [1.493447 -0.448092 0.046803 347.735102;
          0.062734 0.552601 1.035582 -3.804233;
          -0.000102 -0.000500 0.000072 1.000000];

    P7 = [1.392948 -0.700619 0.046803 347.735102;
          0.157739 0.533312 1.035582 -3.804233
          -0.000187 -0.000475 0.000072 1.000000];

    P8 = [1.250125 -0.931858 0.046803 347.735102;
          0.247952 0.497819 1.035582 -3.804233;
```

```
        -0.000267 -0.000435 0.000072 1.000000];

    P9 = [1.069317 -1.134783 0.046803 347.735102;
        0.330630 0.447199 1.035582 -3.804233;
        -0.000339 -0.000382 0.000072 1.000000];

    P10 = [0.856019 -1.303228 0.046803 347.735102;
         0.403262 0.382992 1.035582 -3.804233;
         -0.000400 -0.000318 0.000072 1.000000];

    P11 = [0.616711 -1.432075 0.046803 347.735102;
         0.463642 0.307148 1.035582 -3.804233;
         -0.000449 -0.000243 0.000072 1.000000];

    P12 = [0.358664 -1.517410 0.046803 347.735102;
         0.509934 0.221971 1.035582 -3.804233;
         -0.000484 -0.000162 0.000072 1.000000];

    P13 = [0.089720 -1.556638 0.046803 347.735102;
         0.540731 0.130050 1.035582 -3.804233;
         -0.000505 -0.000075 0.000072 1.000000];

    P14 = [-0.181950 -1.548569 0.046803 347.735102;
         0.555099 0.034177 1.035582 -3.804233;
         -0.000511 0.000014 0.000072 1.000000];

    P15 = [-0.448092 -1.493447 0.046803 347.735102;
         0.552601 -0.062734 1.035582 -3.804233;
         -0.000500 0.000102 0.000072 1.000000];

    P16 = [-0.700619 -1.392948 0.046803 347.735102;
         0.533312 -0.157739 1.035582 -3.804233;
         -0.000475 0.000187 0.000072 1.000000];

    P17 = [-0.931858 -1.250125 0.046803 347.735102;
         0.497819 -0.247952 1.035582 -3.804233;
         -0.000435 0.000267 0.000072 1.000000];

    P18 = [-1.134783 -1.069317 0.046803 347.735102;
         0.447199 -0.330630 1.035582 -3.804233;
         -0.000382 0.000339 0.000072 1.000000];

    P19 = [-1.303228 -0.856019 0.046803 347.735102;
         0.382992 -0.403262 1.035582 -3.804233;
         -0.000318 0.000400 0.000072 1.000000];

    P20 = [-1.432075 -0.616711 0.046803 347.735102;
         0.307148 -0.463642 1.035582 -3.804233;
         -0.000243 0.000449 0.000072 1.000000];

```

```
    P21 = [-1.517410 -0.358664 0.046803 347.735102;
          0.221971 -0.509934 1.035582 -3.804233;
          -0.000162 0.000484 0.000072 1.000000];

    P22 = [-1.556638 -0.089720 0.046803 347.735102;
          0.130050 -0.540731 1.035582 -3.804233;
          -0.000075 0.000505 0.000072 1.000000];

    P23 = [-1.548569 0.181950 0.046803 347.735102;
          0.034177 -0.555099 1.035582 -3.804233;
          0.000014 0.000511 0.000072 1.000000];

    P24 = [-1.493447 0.448092 0.046803 347.735102;
          -0.062734 -0.552601 1.035582 -3.804233;
          0.000102 0.000500 0.000072 1.000000];

    P25 = [-1.392948 0.700619 0.046803 347.735102;
          -0.157739 -0.533312 1.035582 -3.804233;
          0.000187 0.000475 0.000072 1.000000];

    P26 = [-1.250125 0.931858 0.046803 347.735102;
          -0.247952 -0.497819 1.035582 -3.804233;
          0.000267 0.000435 0.000072 1.000000];

    P27 = [-1.069317 1.134783 0.046803 347.735102;
          -0.330630 -0.447199 1.035582 -3.804233;
          0.000339 0.000382 0.000072 1.000000];

    P28 = [-0.856019 1.303228 0.046803 347.735102;
          -0.403262 -0.382992 1.035582 -3.804233;
          0.000400 0.000318 0.000072 1.000000];

    P29 = [-0.616711 1.432075 0.046803 347.735102;
          -0.463642 -0.307148 1.035582 -3.804233;
          0.000449 0.000243 0.000072 1.000000];

    P30 = [-0.358664 1.517410 0.046803 347.735102;
          -0.509934 -0.221971 1.035582 -3.804233;
          0.000484 0.000162 0.000072 1.000000];

    P31 = [-0.089720 1.556638 0.046803 347.735102;
          -0.540731 -0.130050 1.035582 -3.804233;
          0.000505 0.000075 0.000072 1.000000];

    P32 = [0.181950 1.548569 0.046803 347.735102;
          -0.555099 -0.034177 1.035582 -3.804233;
          0.000511 -0.000014 0.000072 1.000000];

    P33 = [0.448092 1.493447 0.046803 347.735102;
          -0.552601 0.062734 1.035582 -3.804233;
```

```
          0.000500 -0.000102 0.000072 1.000000];

    P34 = [0.700619 1.392948 0.046803 347.735102;
          -0.533312 0.157739 1.035582 -3.804233;
          0.000475 -0.000187 0.000072 1.000000];

    P35 = [0.931858 1.250125 0.046803 347.735102;
          -0.497819 0.247952 1.035582 -3.804233;
          0.000435 -0.000267 0.000072 1.000000];

    projectionMatrices = cat(3, P0,  P1,  P2,  P3,  P4,  P5,  ...
                                P6,  P7,  P8,  P9,  P10, P11, ...
                                P12, P13, P14, P15, P16, P17, ...
                                P18, P19, P20, P21, P22, P23, ...
                                P24, P25, P26, P27, P28, P29, ...
                                P30, P31, P32, P33, P34, P35);

end
```

Listing 15: Plots the Camera Locations

```
% Load projection matrices
projectionMatrices = loadProjectionMatrices();

% Get camera locations and directions
cameraLocations = zeros(size(projectionMatrices,3), 3);
cameraDirections = zeros(size(projectionMatrices,3), 3);
for i = 1:36
    [cameraLocations(i,:), cameraDirections(i,:)] = getCameraLocation(
       projectionMatrices(:,:,i));
    cameraDirections(i,:) = cameraDirections(i,:) ./ norm(
       cameraDirections(i,:));
end

quiver3(cameraLocations(:,1), cameraLocations(:,2), -1*cameraLocations
   (:,3), ...
        cameraDirections(:,1), cameraDirections(:,2), -1*cameraDirections
           (:,3), ...
        0.5);

axis([-2000 2000 -2000 2000 0 1000])
```

Eigenface Recognition

Listing 16: EigenfaceRecognizer Class

```
classdef EigenfaceRecognizer
    properties
        theFaceVectors;
        theMeanFaceVector;
```

```
        theEigens;
        theProjections;
        theDistanceMetric = 'euclidean';
    end

    methods
        function self = EigenfaceRecognizer(images)
            %Vectorise face images
            self.theFaceVectors = reshape(images, [], size(images, 3));

            %Compute average face
            self.theMeanFaceVector = mean(self.theFaceVectors, 2);

            %Standardise face vectors
            faceDiffVectors = double(self.theFaceVectors) - self.theMeanFaceVector;

            %Compute covariance matrix as A*transpose(A)
            C = faceDiffVectors' * faceDiffVectors;

            %Compute eigenvectors of covariance
            [eVectors, ~] = eig(C);
            eVectors = faceDiffVectors * eVectors;

            %Normalise eigenvectors
            self.theEigens = eVectors ./ vecnorm(eVectors, 2, 1);

            %Characterise faces as linear combination of eigenface basis
            %vectors
            self.theProjections = zeros(size(self.theEigens, 2));
            for faceNum = 1:size(images, 3)
                self.theProjections(faceNum, :) = sum(self.theEigens .* faceDiffVectors(:,faceNum));
            end
        end

        function classification = recognizeFace(self, face)
            %Vectorise face image
            faceVector = face(:);

            %Project normalised face
            faceDiffVector = double(faceVector) - self.theMeanFaceVector;
            projection = sum(self.theEigens .* faceDiffVector);

            %Recognize face as closest projection
            distances  = pdist2(projection, self.theProjections, self.theDistanceMetric);
            [~, classification] = min(distances);
        end
    end
```

```
end
```

Listing 17: Plots Eigenface PCA Projections

```
%Load eigenfaces
eigenImages = [];

for i = 1:6
    [image, map] = imread(sprintf('faces/eig/%da.bmp', i));
    image = rgb2gray(ind2rgb(image, map));
    eigenImages = cat(3, eigenImages, image);
end

%Create eigenface recognizer
recognizer = EigenfaceRecognizer(eigenImages);

%Project images onto two axes using PCA
basisVectors = recognizer.theEigens(:,1:2);

figure();
hold on;
colours = ['r', 'g', 'b', 'y', 'o', 'c'];
projections = [];
for i = 1:6
    groupProjection = [];
    files = dir(sprintf('faces/%d/*.bmp', i));
    for j = 1:length(files)
        [image, map] = imread(sprintf('faces/%d/%s', i, files(j).name));
        image = rgb2gray(ind2rgb(image, map));
        faceVector = image(:);
        faceVector = faceVector - recognizer.theMeanFaceVector;
        groupProjection = [groupProjection basisVectors' * faceVector];
    end
    scatter(groupProjection(1,:), groupProjection(2,:), 52, colours(i), '
        DisplayName', sprintf('Face %d', i));
end

legend;
xlabel('Principal Component 1');
ylabel('Principal Component 2');
hold off;
```