**Chapter 1 Introduction**

This dissertation proposes the use of image based rendering with PCA to reconstruct the original image from the training data. This method is intended to save the rendering time and requirement of devices for complex rendering, such as volume rendering.

* 1. **Motivation**

As it still cost a lot to render the volume data directly on standard hardware in real time, image based rendering could be an approach to speed up the process and directly show the result of volume rendering.

* 1. **Methodology**

The goal is to use PCA to reconstruct original images and use multithread programming to speed up the process of construction, so that we can test if the image based rendering for volume rendering could be used in real world application.

* 1. **Contribution**

Based on the original paper [1], I re-implement the PCA function in C++ and speed up the reconstruction process with OpenCL module in OpenCV. According to the implementation, experiments were created to test the quality of reconstruction images, the memory required for reconstruction, the frame rate for real-time application and the relationship between these three variables.

* 1. **Summary of Chapters**

This dissertation is structured as follows:

* Chapter2 gives an overview of previous works done related to imaged based rendering, principle component analysis, volume rendering, and parallel programming. A detailed overview is given to main works related to this project, IBR (image based rendering) with PCA (principle component analysis) on volume rendering data.
* Chapter3 presents the design and implementations of the preprocessing, the CPU reconstruction, and the method of speeding up the reconstruction by using parallel programming.
* Chapter4 details the result gained from experiment. A discussion relating to the results is also presented.
* Chapter5 summarizes the project and provide a discussion of future work.

**Chapter 2 Background**

**2.1 Image based rendering**

**2.2 Principle component analysis**

**2.3 Volume rendering**

**2.4 Parallel programming**

**Chapter 3 Design and implementation**

**3.1 Basic PCA implementation**

In this section, I’ll explain the detail of the PCA implementation which is based on a PCA tutorial [3].

**3.1.1 Put the observed object into the matrix**

The observed object should have same dimension. For example, the observation we handle here are images, the training images should have the same dimension. Every image is put into one vector. If the image is dd dimension, the vector size is dd. If the training images number is n, the matrix with n rows and dd columns is formed as follow,

(1)

where vec\_ represents the image’s all pixel values (For one channel).

**3.1.2 Subtract the mean**

To get the variance for each pixel, calculate average value in column, get the mean matrix, and use the original matrix to subtract this mean matrix. The adjusted data is calculated as below,

(2)

where is the result from equation (1), is the mean vector with dd elements.

**3.1.2 Calculate the covariance matrix**

Covariance is a similar way like standard deviation and variance to perform “how much the dimensions vary from the mean with respect to each other” [3]. It is always measured between two-dimensional data. For example, two vectors X and Y with 10 elements in each vector, where is the mean of vector X, is the mean of vector Y.

(3)

Covariance matrix is to measure multi-dimensional vector variance from their mean. For a matrix with n rows. The covariance matrix is calculated as below,

(4)

where DataAdjust is calculated by equation (2), DataAdjust\_Transpose is the transpose of the DataAdjust. If the DataAdjust is n rows with dd columns, the result of covariance matrix should be dd rows with dd columns.

**3.1.3 Calculate the Eigenvectors and Eigenvalues**

Eigenvectors can be calculated by solving the equation as below

(6)

where  is an eigenvector of the linear transformation , the scale factor  is the eigen-value corresponding to that eigenvector and is the n by n identity matrix. [5]

In this place, the previous result which is calculated as covariance matrix is the matrix in equation (6). The number of eigenvectors and eigenvalues should be dd. Each eigenvector should have dd elements.

**3.1.4 Sort the Eigenvalues**

After getting the eigenvectors and eigenvalues, the sorting process with highest to lowest order should be calculated by eigenvalues and sorting the corresponding eigenvectors. The highest eigenvector represents the highest variance for each dimension compared to mean vector. The second eigenvector represents the second highest variance for each dimension compared to mean vector, and so on.

**3.1.5 Choose the number of components and calculate scores**

While using all dd eigenvectors could reconstruct the same original observed vector, choosing some of the highest eigenvectors could also reconstruct the original vector with subtle data missing which means the more eigenvectors we used, the higher quality for the reconstructed vector. For an original observed vector, the projection value (referred to as a score) is calculated by project one eigenvector onto the original vector as below:

where is original vector, is one of the eigenvector, and is the score of this eigenvector projected onto this original vector.

**3.1.6 Reconstruction of the original image**

In

**3.2 Small training data with CPU, image based rendering**

To build up a prototype of the reconstruction and save the preprocessing time, I used 36 training images, at a resolution of 300\*300 pixels. These images were saved screenshots by rotate the camera around a soldier [2] model’s head horizontally for 360 degrees (10 spacing for the azimuthal angle). At first, I directly used cell-based method to crop the whole image to cells instead of using the whole image as the observation. Like the previous experiment [1], I used the same cell-dimension 2020 to create an observed matrix with 36 rows and 400 columns. 2020 pixels values were put into one row (vector) which means each column represented one pixel value. In this case, there were 225 observed matrixes to save the whole images information for each channel.

**Put the image pixel value into the matrix**

The raw rbg image has three channels and the specific location’s pixel value is obtained by the corresponding width and height location. For OpenGL, this can be made by using SOIL library. Then, the image pixel value can be accessed by a char array.

. and the used on training images to reconstruct the original image. Eigen library [4] was used here for matrix calculation.

For the soldier head example I used previously, the covariance matrix is calculated as below:

(5)

. . . (1)

Reference:

[1] Alakkari, S., & Dingliana, J. (2016, September). Volume visualization using principal component analysis. In Proceedings of the Eurographics Workshop on Visual Computing for Biology and Medicine (pp. 53-57). Eurographics Association.

[2] Soldier model link: <https://free3d.com/3d-model/chris-15987.html>

[3] Smith, L. I. (2002). A tutorial on principal components analysis. *Cornell University, USA*, *51*(52), 65.

[4] Eigen library: <http://eigen.tuxfamily.org/index.php?title=Main_Page>

[5] Eigenvector and eigenvalue: <https://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors>

[6]