**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 reduce the 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 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.

**3.1.3 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 and with 10 elements in each vector, where is the mean of vector , is the mean of vector .

(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 is calculated by equation (2), is the transpose of the , n is the number of observed vectors. If the is n rows with dd columns, the result of covariance matrix should be dd rows with dd columns.

**3.1.4 Calculate the Eigenvectors and Eigenvalues**

Eigenvectors can be calculated by solving the equation as below

(5)

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 (5). The number of eigenvectors and eigenvalues should be dd. Each eigenvector should have dd elements.

**3.1.5 Sort 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.

**3.1.6 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:

(6)

where is original vector, is the transpose of highest eigenvectors, and is the score of this eigenvector projected onto original vector, , .

**3.1.7 Reconstruction of the original image**

The final step is to reconstruct the original image by using the sum of product of some number of highest eigenvectors and corresponding scores projected on one original image. The equation is as follow,

(7)

where is the highest eigenvector of this original vector, is the corresponding score for this highest eigenvector, is the result of the reconstructed vector, , is the number we choose as components, .

**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.

**3.2.1 Crop to cell and 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. Eigen library [4] was used here for matrix calculation. 255 observed matrix is represented as below,

. . . (8)

**3.2.2 Calculate eigenvectors**

According to the 3.1.2, we can get the adjusted data for each matrix by subtract the mean vector in each row. Then, for the soldier head example I used previously, the covariance matrix is calculated as below:

(9)

where is adjusted data of , is the transpose of , is corresponding covariance matrix of , .

In Eigen library, the function could be used to calculate eigenvectors and eigenvalues by offered covariance matrix. Then, the bubble sorting method was used to sort eigenvalues and the corresponding eigenvectors were sorted as well.

**3.2.3 Reconstruct the original cell image and put cells together to a whole image**

After getting the ordered eigenvectors, we can project them onto every original image we want to reconstruct to obtain the corresponding scores. The result of these eigenvectors and scores were saved in files. Every time, instead of spending long time for preprocessing, we can read these eigenvectors and scores from files to reconstruct any original cell image. The final step is to put these reconstructed cells together back to the whole image. Three channels pixel values were put back to the char array for texture loading in OpenGL.

**3.3 Small training data with Cuda (GPU)**

**3.3.1 Parallel programming design for reconstruction**

GPU is powerful for multithread computing. However, there are several ways to complete a complex computing. For here, the input we got are eigenvectors and scores read from files. The output is a whole image pixel values. The most significant procedure inside was reconstruction for each cell image. And this procedure would be in a loop for number of cells time for reconstructing the whole image. CPU computing scudo-code is as below,

1: for all do

2: for all do

3:

4: end for

5:

6:

7: end for

is the number of cells, is the number of components we chosen, is the mean vector for each cell, is the function that resize reconstructed cell data from row to corresponding dimension and put the specific cell to corresponding position in a whole image.

Considering the same reconstruction process for one cell and the delay for transporting time between host memory and device memory, the best optimization is to use multithread programming in the process of reconstructing one cell. Cuda computing scudo-code is as below,

1: for all do

2:

3:

4:

5: end for

Everything was the same as CPU computing except that the reconstruction loop was replaced by a function.

**3.3.2 Detail of Cuda speeding up**

In this function, ???

**3.4 Large training data with OpenCV**

**3.4.1 Preprocess with OpenCV**

**3.4.2 Speed up reconstruction by OpenCL (GPU)**

**3.5 Experiment for real-world application**

**Chapter 4 Results and Evaluation**

The following chapter details the result gained from the implementation. Some experiment based on image quality, frame rate (reconstruction speed), and memory required are discussed and evaluated.

**4.1 Reconstruction of Images**

**4.1.1 Small training images results, reconstructed with CPU and GPU**

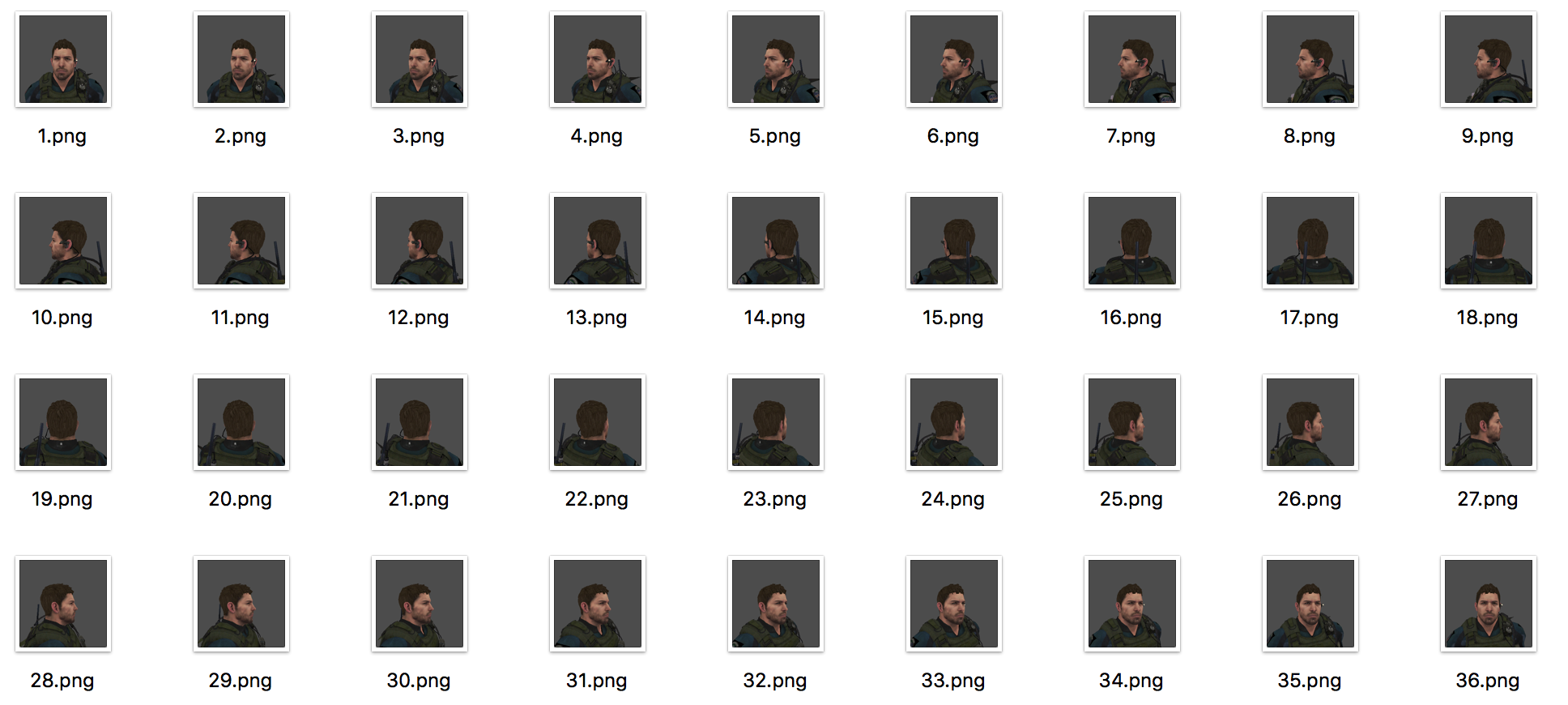


Figure Head training data with 36 images

Figure Left: reconstructed by 15 eigenvectors. Right: reconstructed by 20 eigenvectors.

Figure Left: reconstructed by 30 eigenvectors. Right: original image.

First implementation of PCA with CPU is based on solider head sample. From human eyes, the higher the eigenvectors used, the better the quality of the reconstruction compared to original image.

I didn’t focus on the relationship about result quality, number of components and frame rate in this place. Instead, the reconstruction process was focused at first. Calculation for reconstruction based on CPU was used as normal. The reconstruction time for the whole image was 10 seconds in devices with i7 core CPU, 2.3Hz, 2.4G memory? By using Cuda kernel function, the reconstruction time for the whole image was 2 seconds, 5 times faster than CPU computing.

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]