**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.

???

**“Image-based volume rendering with opacity light fields”**

* 1. **Motivation**

In computer graphics, some of the objects could be rendered by polygons such as buildings, weapons, characters in video game and so on. However, some other requirements are needed to render the inner information for the object or the model is too hard to create as a polygon. Some of them could be rendered by volume rendering with 3D data sets. For example, the CT, MRI and engineering drawing which is used in medical and engineering visualization. Moreover, there might be some transparent objects could also be rendered by volume rendering, such as trees, fuzzy and gas.

With the rapid volume data expansion, the computation became more and more huge quickly. It will cost a lot to render the volume data directly on commodity desktop machine with interaction. Some devices even cannot support the large GPU memory. The lower computing power device would have worse result, such as in mobile devices.

In this case, some image based rendering methods were proposed to overcome the complex computing. Image based rendering has several advantages including speeding up the process and directly showing the result of volume rendering independent of the complexity of 3D datasets.

* 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. To reduce artefacts between cells, blurring was added onto the result images for better performance. 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. By changing the resolution of the same dataset, the cell dimension of each image and the number of components used in PCA, users could choose suitable parameters for this PCA image based rendering on different dataset.

* 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**

In computer graphics, rendering is the process of generating an image from a 2D or 3D scene, which might contain several models. The result of this scene is related to lighting simulation, texture fidelity, shading information and viewpoint direction. However, when the scene is too complex or the subtle light effect in the real world is too difficult to simulate, traditional techniques have some limitation or the cost to simulate is too high. In this case, image-based rendering system was created to improve the realism of the 3D model and used to approximate global illumination effects as well.

The plenoptic function [9] is a function using pencil of rays’ angle of a given point, position for this given point, time dimension and intensity vanes with wavelength, to describe possible environment maps for a given scene from this given point in space. In Plenoptic Modeling [6], McMillan and Bishop discussed finding the disparity of each pixel in stereo pairs of cylindrical images. First, they used a cylindrical projection as the plenoptic sample representation instead of normal six planar projection onto a cube. Then, they reconstructed and resampled image warps that map sample images to arbitrary angle of cylindrical viewpoints by using plenoptic function.

Gortler and Grzeszczuk [7] used a 4D function called Lumigraph. This system is made for sampling the plenoptic function by handheld camera and use these light in Lumigraph. Instead of using 5 parameters to represent position and direction in plenoptic function, they proposed a way to reduce parameters to 4 by using 2D slices of 4D space. Direction is parameterized using two parallel planes (Figure 1). All the point can be represented by 4 coordinates in the Lumigraph. Then, they applied a discrete subdivision onto a 4D space and associate a coefficient and a basis function (reconstruction kernel) with each 4D grid point.

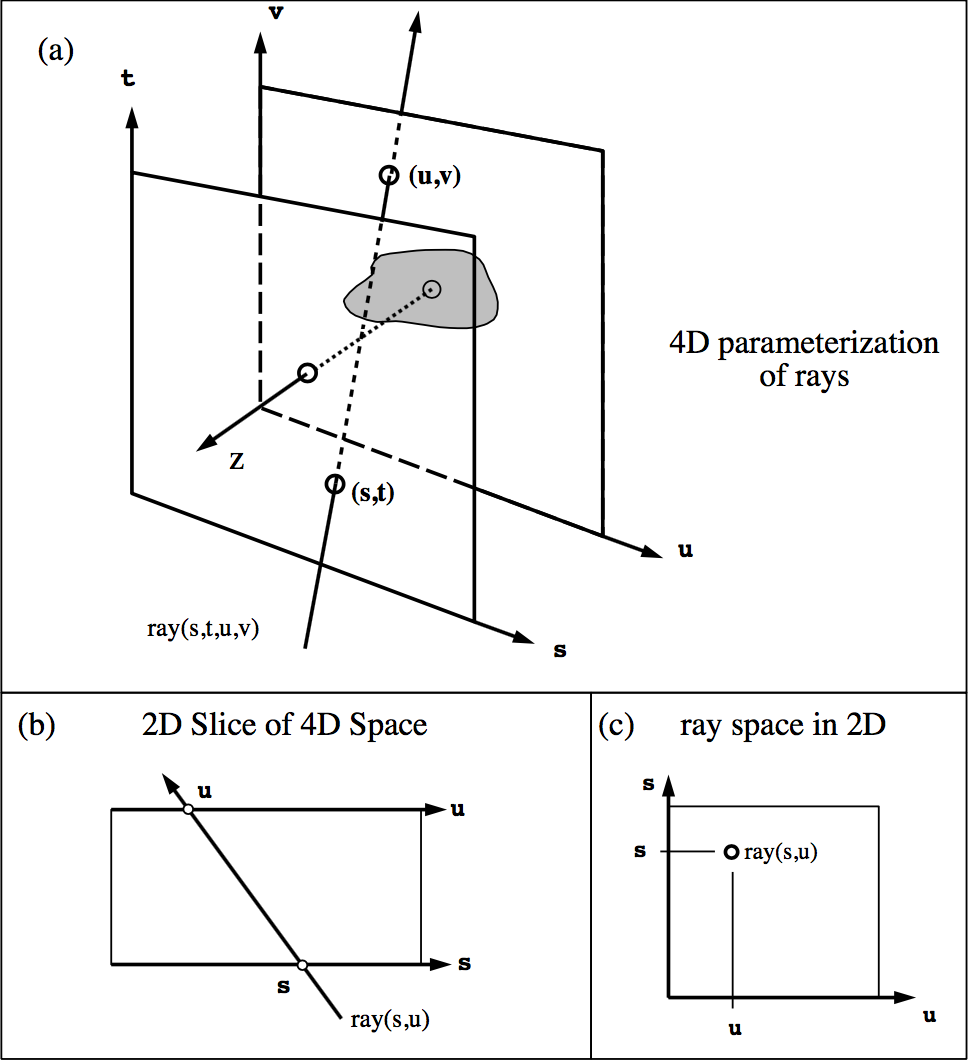


Figure 1 Parameterization of the Lumigraph [7]

Marc and Pat [8] propose a robust way named light field rendering to give more freedom view of image-based rendering. The main idea is to fill in regions of space free of occludes by creating a light field from a set of images corresponding to inserting each 2D slices into the 4D light field representation. Instead of using depth information, they simply combined and resampled available images. A new view could be shown in real time by extracting slices in appropriate direction if a light field is created.

Imposters or billboards are also a way which can be called image based rendering. The motivation for this technic is to reduce the complexity of rendering a 3D scene such as trees, grass, backgrounds in video game. Without losing resolution, 3D objects could be represented as a 2D image rotate with viewer’s observed direction. There’re two categories of imposters [11], static and dynamically-generated. Static imposter is usually created by artist and showed a right-angle like billboards. Dynamically-generated imposter is regenerated at runtime by rendering an image of a 3D object to a texture.

**2.2 Principle component analysis**

As discussed in 2.1, there are several techniques used in image based rendering. For our purpose, we intend to show the result image of a scene by reconstructing from the dataset of this scene. We intended to reconstruct corresponding image from user’s interaction, rather than rotate one image with different angle according to camera position, like billboards mentioned in 2.1.

In this case, Alakkari [1] proposed to use Principal component analysis (PCA), a statistical method, to compress the dataset and reconstruct images by back projecting process as a method to implement image based rendering. Principal component analysis [12] is a statistical procedure that uses an [orthogonal transformation](https://en.wikipedia.org/wiki/Orthogonal_transformation) to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables. The number of principal components is less than or equal to the smaller of the number of original variables or the number of observations.

Dimensionality reduction using PCA enables pose changes to be visualised as manifolds in low-dimensional subspaces and provides a useful mechanism for investigating face pose [13]. A reconstruction image could be generated by adding mean image of dataset and several number of highest eigentextures (sometimes called eigenfaces) multiplied with projected scores (figure 2).



Figure 2 Projection process (Reconstruction) [5]

Ko Nishino [14] proposed the Eigen-Texture method to create a 3D model of an object from a sequence of range images. By aligning the pixel value from images into the 3D model, it can get a nice control of the mixed reality system. Although image based rendering is simple and handy for a stand-alone object without any background for the virtual reality, there’re some disadvantages, such as cast shadows under real illuminations, when combining these object into a real-world background. Our cell-based PCA process is similar with the generating virtual images process in this approach. The difference is that their result is summation of component virtual images sampled under single illuminations.

**2.3 Volume rendering**

Contrary to surface rendering, volume rendering (figure 3) is a technique to render 3D volume data to show the interior information of the object, usually used in medicine and some transparent objects. There are several ways for the 3D volume data visualization. The most basic volume rendering algorithm is ray-casting which uses straight-forward numerical evaluation of the volume rendering integral [15]. 2D texture [16] and 3D texture [17] mapping were used in these places. Without downsample the result for better qualities, distributed volume rendering [21] were proposed to automatically crop and partition volumes into small volume blocks. However, because of the large data size, it still cost a lot to render the volume data directly on standard hardware. In this case, image-based rendering, as mentioned previously, is an alternative to deal with volume data.

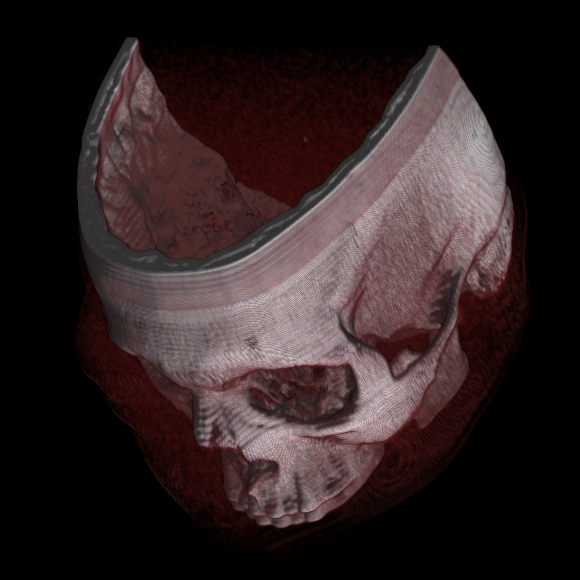


Figure 3 A volume rendered cadaver head using view-aligned texture mapping and diffuse reflection [10]

Chen [18] presented a way to render surfaces from volume data by integrating the fully volume rendered image (the keyview) and the model of the volume. It was made by first constructing the surface model of the volume and texture mapping the keyview onto the geometry. Then use cast-ray to render the rest of new region when changing the view slightly so that they can save several computing processes for visualizing volume data.

Meyer [19] came up with an image-based volume rendering with opacity light fields. They separated the process to three parts, getting ray slices and proxy surface using a standard volume rendering application on any type of volume data, generating new key views for each ray slice viewpoint and interactively rendering as opacity light fields. This approach can take advantage of the different features of the hardware, leave the heavier rendering work to the good graphics cards and combine the result in remote application. A main difference from image-based volume rendering work in [19] is their technique could change view to arbitrary angle.

Tikhonova [20] used a representation of a multi-layered image, or an explorable image to show interior structure of the object, by simulating opacity changes and recolouring of individual features. From the single-view, a small number of rendered images were generated from 3D data set. Their technique could automatically extract multiple layers. By decomposing the data into different layers corresponding to different structures in the data, users could change the opacity and colour of each layer to interact with scene without re-rendering whole volume data. This can save GPU and CPU in devices as well, making it possible to run on low-end hardware.

**2.4 Parallel programming**

2.4.1 CUDA

To satisfy huge and fast increasing requirement of 3D graphics, GPU has evolved into a highly parallel, multithreaded processor. High definition result made by millions of pixels are relied on drawing each vertex and shader model in one thread. CUDA is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU) [22]. Apart from using parallel programming on graphics, there’re more and more applications involved CUDA to speed up calculations [23]. ----“CUDA language is a minimal extension of the C and C++ programming languages. The programmer writes a serial program that calls parallel kernels, which may be simple functions or full programs. A kernel executes in parallel across a  
set of parallel threads. The programmer organizes these threads into a hierarchy of grids of thread blocks. A thread block is a set of concurrent threads that can cooperate among themselves through barrier synchronization and shared access to a memory space private to the block. A grid is a set of threads blocks that may each be executed independently and thus may execute in parallel.”

2.4.2 OpenCL

“Open Computing Language (OpenCL) is a [framework](https://en.wikipedia.org/wiki/Software_framework) for writing programs that execute across [heterogeneous](https://en.wikipedia.org/wiki/Heterogeneous_computing) platforms consisting of [central processing units](https://en.wikipedia.org/wiki/Central_processing_unit) (CPUs), [graphics processing units](https://en.wikipedia.org/wiki/Graphics_processing_unit)(GPUs), [digital signal processors](https://en.wikipedia.org/wiki/Digital_signal_processor) (DSPs), [field-programmable gate arrays](https://en.wikipedia.org/wiki/Field-programmable_gate_array) (FPGAs) and other processors or [hardware accelerators](https://en.wikipedia.org/wiki/Hardware_accelerator).” [24] “In particular OpenCL provides applications with an access to GPUs for non-graphical computing (GPGPU) that in some cases results in significant speed-up. In Computer Vision, many algorithms can run on a GPU much more effectively than on a CPU: e.g. image processing, matrix arithmetic, computational photography, object detection etc.” [25]

**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: end for

Everything was the same as CPU computing except that the reconstruction loop and adding mean process were replaced by the function.

**3.3.2 Detail of CUDA speeding up**

Because of transporting data from host memory to device memory, the data structure needed to be point array. Moreover, the kernel function in CUDA cannot pass matrix data structure. So, I transferred all vectors used by Eigen library to custom double array and all matrixes used by Eigen library to custom host-vector by host-vector variables in CUDA language.

Then, memory in device were allocated by function , where is the double array and is the size of this array. Next, calculated data were copied from host to device. Grids in each thread were allocated to (1, 1), and number of grids in each block were allocated to (, 1), where is the dimension of each cell images, because it just need to handle one pixel value in one thread.

Summation of scores multiplying eigenvectors could be speeded up by one kernel and summation with mean vector could be speeded up by another kernel .

\_\_global\_\_ void arradd1(double\* d\_m, double\* d\_n, double size, double projectionValue) {

int myid = threadIdx.x;

d\_n[myid] = d\_n[myid] + projectionValue \* d\_m[myid];

}

\_\_global\_\_ void arradd2(double\* d\_m, double\* d\_n, double size) {

int myid = threadIdx.x;

d\_n[myid] += d\_m[myid];

}

The scudo-code of function is as below,

1: for all do

2:

3:

4: end for

5:

In kernel arradd1, is the eigenvector of cell images, is result for one cell image, is the size of eigenvector, projectionValue is the score. In kernel arradd2, is the mean vector of one cell image, is the result of one cell image, is the size of mean vector.

**3.3.3 Supported devices**

CUDA supports Windows, Linux and Mac OS. It is a standard feature in all NVIDIA GeForce, Quadro, and Tesla GPUs as well as NVIDIA GRID solutions. A full list can be found on the CUDA GPUs Page [28]. Only devices support CUDA can run CUDA kernel for acceleration.

**3.4 Large training data with OpenCV**

Calculating 225 sets of eigenvalues and eigenvectors for a 400 by 400 covariance matrix needs 20 hours, and there were RGB three channels need to be calculated. Except Eigen library, there are several ways to implement pre-processing. I found that OpenCV, which is a most famous Open Source for images processing, also have function to calculate eigenvalues and eigenvectors from a covariance.

**3.4.1 Calculate PCA in OpenCV**

PCA class in OpenCV not only contains the eigensolver function in Eigen library, but it also packs the back-project function and the final projecting function together. It reduced pre-processing time by using OpenCL to calculate eigenvalues and eigenvectors. I was focusing on the reconstruction process and series experiments of the reconstruction result, so the OpenCV with OpenCL preprocessing detail would not be discussed here. To simulate the real-world application of volume rendering data, I used 900 head images with 1080 by 1080 resolution as test data sample. This dataset is also used in [1].

In this PCA class, the main function of would return a set of matrixes, where represents the input data, represents the mean vector on rows or columns (depends on what flag used) of input data, represents the observing type of the input data, represents the number of components users want to save in PCA.

Based on cells size, all 900 images were cropped into the same cell dimension. Each data represented one cell information in the same position of these 900 images. Each cell was resized into one vector and put into one row of the data matrix, so the data column size would be 900 in this case. For example, if the cell size is 2020, the data matrix would be 400 by 900. Even though we can calculate mean vector of theses 900 vectors, we can still leave parameter to empty (Mat()). PCA function will automatically calculate mean vector for us. Flag parameter could be DATA\_AS\_ROW or DATA\_AS\_COL. DATA\_AS\_ROW indicates that the input samples are stored as matrix rows. DATA\_AS\_COL indicates that the input samples are stored as matrix columns. For here, I put each cell into row, so the option for Flag should be DATA\_AS\_ROW. If there is no value or the value is larger than the size of observed vector in the last parameter , the number of returned eigenvectors and eigenvalues would be the size of observed vector. Otherwise, this number would make sure the returned set only retain specific number of eigenvectors and eigenvalues.

Result of this function is a set of matrixes. It includes eigenvalues, eigenvectors and mean vector of the input data. Eigenvalues and eigenvectors have already been sorted from highest to lowest for these returned set.

Scudo-code for cropping to cells and calculating PCA is as below,

for all do

for all do

for all do

end for

end for

end for

for all do

DATA\_AS\_ROW, num\_components)

end for

, where image\_height and image\_width are the height and width of a sample image, cell\_dimension is the width or height of a cell which is normally a square, image\_num is the number of images for dataset, loadImage() function is to load image from disk which used a imread() function in OpenCV, cropImage() function is to crop the loaded image to specific position’s cell, saveCellToMatirx() function is to resize the cropped cell into a vector which use resize() function in OpenCV and save them into a matrix. savePCAResultsToFile() function is to save PCA result to a file by using FileStorage() function in OpenCV. Because of the result consisted of three matrix, eigenvalues, eigenvectors and mean, in this FileStorage() function, I use ‘eigenvalues’, ‘eigenvectors’ and ‘mean’ tag to save them in one file, convenience for reading the data.

After saving PCA results, to optimize the reconstruction process, projecting eigenvectors to every original data is needed in advanced (in pre-processing) to save reconstructing time. Scudo-code for calculating and saving scores is as below,

for all do

for all do

score = pca[j].project(cells[j].row[i])

saveScoreResult()

end for

end for

saveScoreResultToFile()

, where cell\_num is the number of cropped cells, pca[j] is cell pca result, project() is a method in pca to project eigenvectors into original data vector, cells[j].row[i] is to get the row vector in cell matrix (one of original cell data), saveScoreResult() function is to save score vector into a matrix, saveScoreResultToFile() is to save result into a file, tag with ‘scores’.

**3.4.2 Reconstruction**

The reconstruction process with PCA class in OpenCV is direct. pca result has already packed eigenvectors and mean vector. Scores vector were also calculated by projection in section 3.4.1. Scudo-code for reconstruction is as below,

for all do

for all do

pca = getOnePCAResult(i, j)

score = getOneScoreResult(i, j)

result = pca.backProject(score).reshape(1, cell\_dimension)

copyResultToWholeImage(i, j)

end for

end for

normalizeResultImage()

mergeBGRChannels()

copyResultImageToTexture()

, where getOnePCAResult() function is to get one of pca result from the file, which saves all pca results, getOneScoreResult() function is to get one of score vector from the file, which saves all score vectors results, backProject() method in pca is the process discussed in section 3.1.7, implementing the equation (7), reshape() method here is to reshape the result vector back to cell dimension data, copyResultToWholeImage() function is to copy result cell image into a whole image with specific position, normalizeResultImage() function is to make sure result pixel is between 0 and 255, mergeBGRChannels() function is to merge three channels into one BGR image, copyResultImageToTexture() function is to copy this BGR image into unsigned char\* data type for texture.

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

PCA class reference [26] does not contain details about how backProject (reconstruction) method is calculated. However, after implementing whole pre-processing and reconstruction with OpenCV, I found that the reconstruction time is much smaller than I reconstructed with Eigen library. After digging into the source code of PCA class in OpenCV, I found out that they are using OpenCL parallel programming to speed up matrix arithmetic, discussed in section 2.4.2.

If the device support OpenCL [27], input data are passed to a ocl\_gemm() function, the kernel named ocl::Kernel with tag “gemm” are executed. Details inside kernel are not discussed here, because it’s the same process for calculating vectors in CUDA. To test the performance of CUDA, a reconstruction process with only-CPU calculation is made. The pre-processed pca data and scores were input data, scudo-code as below,

initResultMatrix()

for all do

result += scores[i] \* pca.eigenvectors.row[i]

end for

result += pca.mean

, where initResultMatrix() function is to initialize result matrix by creating the empty CV\_32F data type matrix, calculations inside loop are repeating the pca reconstruction process of equation (7), final step is to add mean vector to result vector. The performance comparison between OpenCL and only-CPU calculation is discussed in Chapter 4.

**3.5 Smooth artefacts between cells**

Cell-based PCA method for image-based rendering has a problem mentioned in [1]. Even though the number of all eigenvectors for cell-based images is less than the number used for whole images, artefacts appearing in the cell boundary regions will affect the quality of reconstruction results. To solve this problem with lower computing cost, a smoothing function medianBlur was used here.

To perform a smoothing (also called blurring) operation, a filter is added to an image. In OpenCV, there are several most used filters, such as blur, GaussianBlur, medianBlur, bilateralBlur [30]. Theories and features are discussed as below and the reason for using medianBlur is discussed as well.

3.5.1 Mean blur

Linear is the most common type filter, which output pixel is determined by a weighted sum of input pixels. For example, the simplest one is the mean of its kernel neighbors. Every pixel in this kernel is contributed equal weights for the whole filter. The default parameter for is filter is the mean. One advantage for this mean filter is its low computing cost. The kernel is as below,

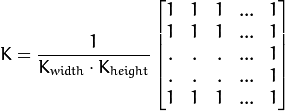


Figure 4 Kernel for blur filter [29]

3.5.2 Gaussian Blur

GaussianBlur filter is normally the most useful filter for noise in an image. “It’s done by convolving each point in the input array with a Gaussian kernel and then summing them all to produce the output array” [30]. The pixel located in the middle of Gaussian curve would have the highest weight. The weight of its neighbours decreases as the spatial distance between them. Artefacts between cells were not noise of the result image, so this relatively higher computing cost filter was not used in real-time reconstruction process.

3.5.3 Median Blur

MedianBlur filter is a non-linear filter. Unlike mean blur, it replaces each pixel with the median of its neighbours (33 in a kernel). Moreover, the artefacts of observed image is preserved.

3.5.4 Bilateral Blur

BilateralBlur filter is also a non-linear filter like GaussionBlur filter. It also considers the neighbouring pixels with weights assigned to each of them. One feature for this filter is it can smooth noises of an image without smoothing its edges. However, this method was also discarded because of relatively higher computing cost in real-time reconstruction process.

As analyzing in above, commonly used mean blur filter and non-linear medianBlur filter could both be used in smoothing artefacts of images. In my experiments, non-linear medianBlur filter was used for a better quality of results (Results and evaluation is discussed in chapter 4).

**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 cost are discussed and evaluated.

**4.1 PCA results**

**4.2 Reconstruction**

**4.2.1 Small training images results**

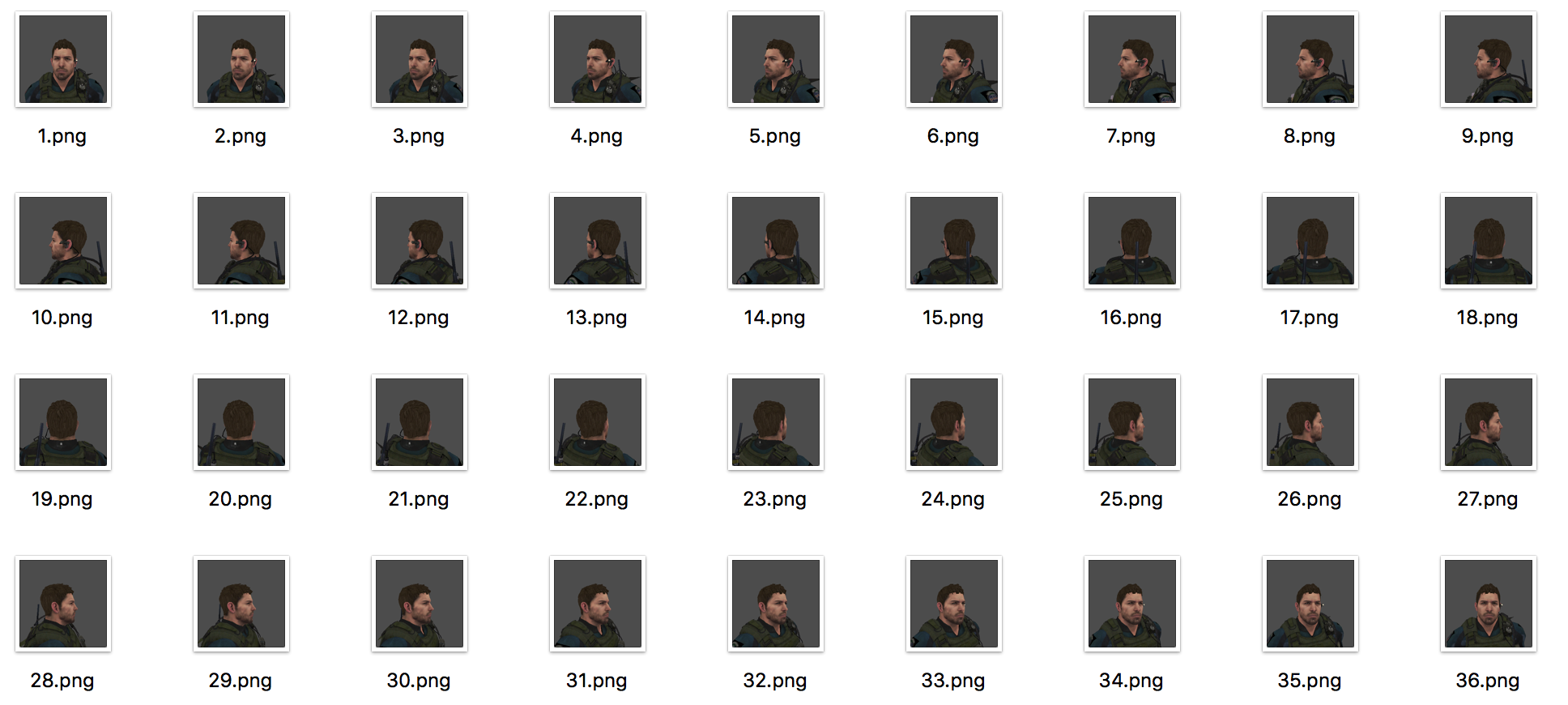


Figure 5 Head training data with 36 images

Figure 6 Left: reconstructed by 15 eigenvectors. Right: reconstructed by 20 eigenvectors.

Figure 7 Left: reconstructed by 30 eigenvectors. Right: original image.

First implementation of PCA with CPU is based on solider head sample, discussed in section 3.3. From human eyes, the higher the eigenvectors used, the better the quality of the reconstruction compared to original image.

**4.2.2 Reconstruction speed comparison between CPU and GPU (CUDA)**

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. On a standard device with Intel(R) Xeon(R) CPU E3-1240 v3, 3.40GHz, 16.0G memory and NVIDIA Quadro K2000 graphic card, reconstruction speed in normal CPU calculation and CUDA calculation are showed in chart 1 as below,

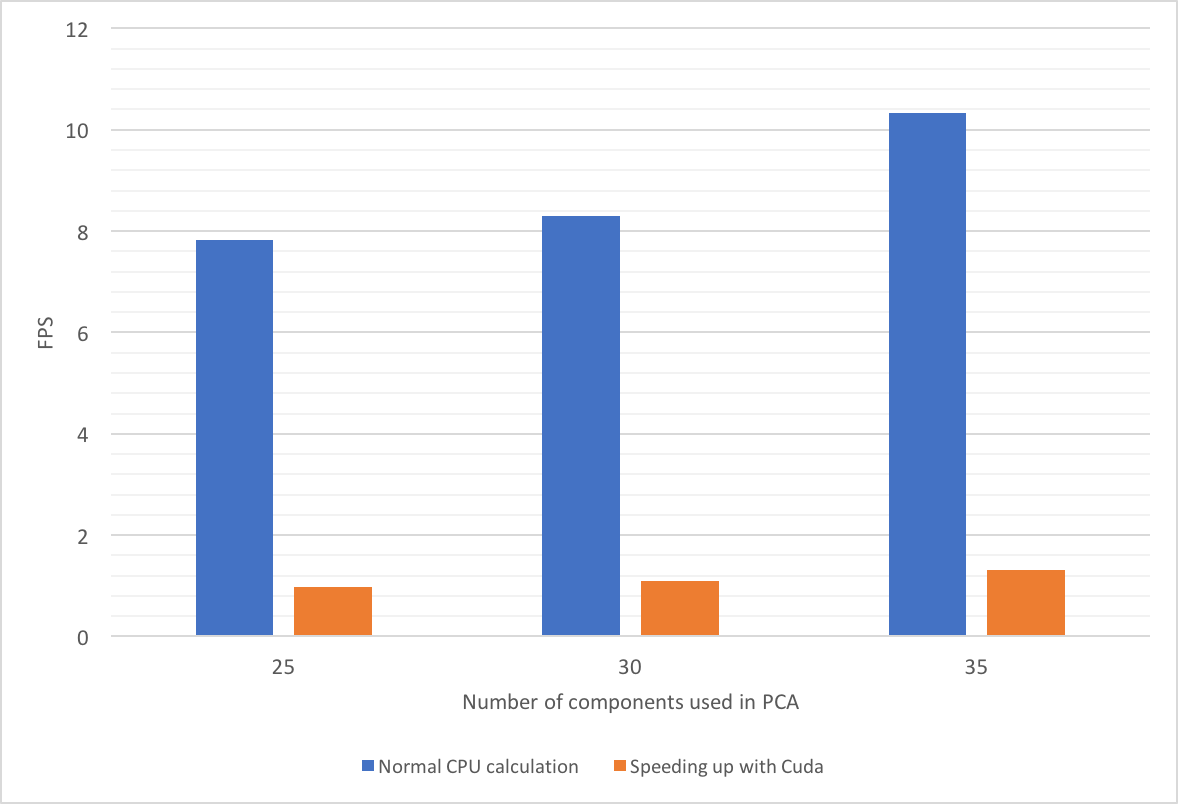


Chart 1 Reconstruction speed comparison between normal CPU and CUDA

From chart 1, we can see that construction time in CPU will increase with the increasing number of components used, while this process time in CUDA (GPU) will not change too much. At the same time, the average construction of GPU is around 6 times faster than CPU.

**4.2.3 Large training images results**

**4.2.4 Reconstruction speed comparison between CPU and GPU (OpenCL)**

**4.3 Image quality, frame rate (reconstruction speed) and memory cost**

**4.3.1 Mean blur and median blur**

**4.3.2 Same resolution dataset**

**4.3.3 Different resolution dataset**

**Chapter 5 Conclusions and Future Work**

Acknowledgement:

The VisMale Head dataset is courtesy of the Visible Human Project at the U.S. National Library of Medicine.

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] MCMILLAN, L., ANDBISHOP, G. Plenoptic modeling: An image-based rendering system. In Computer Graphics, Annual Conference Series, 1995, pp. 39–46.

[7] Gortler, S. J., Grzeszczuk, R., Szeliski, R., & Cohen, M. F. (1996, August). The lumigraph. In Proceedings of the 23rd annual conference on Computer graphics and interactive techniques (pp. 43-54). ACM.

[8] Levoy, M., & Hanrahan, P. (1996, August). Light field rendering. In Proceedings of the 23rd annual conference on Computer graphics and interactive techniques (pp. 31-42). ACM.

[9] Adelson, E. H., & Bergen, J. R. (1991). The plenoptic function and the elements of early vision.

[10] Volume rendering: <https://en.wikipedia.org/wiki/Volume_rendering>

[11] Dynamic 2D Imposters: <http://www.gamasutra.com/view/feature/130911/dynamic_2d_imposters_a_simple_.php>

[12] Principle components analysis: <https://en.wikipedia.org/wiki/Principal_component_analysis>

[13] GONG S., MCKENNA S., COLLINS J. J.: An investigation into face pose distributions. In *Automatic Face and Gesture Recognition, 1996., Proceedings of the Second International Conference on* (1996), IEEE, pp. 265–270

[14] NISHINO K., SATO Y., IKEUCHI K.: Eigen-texture method: Appearance compression based on 3D model. In *Computer Vision and Pat- tern Recognition, 1999. IEEE Computer Society Conference on.* (1999), vol. 1, IEEE.

[15] Engel, K., Hadwiger, M., Kniss, J., Rezk-Salama, C., & Weiskopf, D. (2006). *Real-time volume graphics*. CRC Press.

[16] Rezk-Salama, C., Engel, K., Bauer, M., Greiner, G., & Ertl, T. (2000, August). Interactive volume on standard PC graphics hardware using multi-textures and multi-stage rasterization. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (pp. 109-118). ACM.

[17] Wilson, O., Van Gelder, A., & Wilhelms, J. (1994). *Direct Volume Rendering Via 3D Terxtures*. Computer Research Laboratory [University of California, Santa Cruz].

[18] Chen, B., Kaufman, A., & Tang, Q. (2001). Image-based rendering of surfaces from volume data. In *Volume Graphics 2001* (pp. 279-295). Springer, Vienna.

[19] Meyer, M., Pfister, H., Hansen, C., Johnson, C., Meyer, M., Pfister, H., ... & Johnson, C. (2005). Image-based volume rendering with opacity light fields. *no. UUSCI-2005-002. Tech Report*.

[20] Tikhonova, A., Correa, C. D., & Ma, K. L. (2010, March). Explorable images for visualizing volume data. In *PacificVis* (pp. 177-184).

[21] Frank, S., & Kaufman, A. (2005, December). Distributed volume rendering on a visualization cluster. In *Computer Aided Design and Computer Graphics, 2005. Ninth International Conference on* (pp. 6-pp). IEEE.

[22] CUDA home page: [http://www.nvidia.com/object/CUDA\_home\_new.html](http://www.nvidia.com/object/cuda_home_new.html)

[23] Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). Scalable parallel programming with CUDA. *Queue*, *6*(2), 40-53.

[24] OpenCL: <https://en.wikipedia.org/wiki/OpenCL>

[25] OpenCL in OpenCV: <http://opencv.org/platforms/opencl.html>

[26] PAC in OpenCV: <http://docs.opencv.org/trunk/d3/d8d/classcv_1_1PCA.html>

[27] Supported device list for Intel Architecture: <https://software.intel.com/en-us/articles/opencl-drivers>

[28] CUDA GPUs page: [https://developer.nvidia.com/CUDA-gpus](https://developer.nvidia.com/cuda-gpus)

[29] Smoothing images in OpenCV: <http://docs.opencv.org/2.4/doc/tutorials/imgproc/gausian_median_blur_bilateral_filter/gausian_median_blur_bilateral_filter.html>

[30]