Image-Based Rendering for Volume Visualization by Using Principle Component Analysis

by

Yinghan Xu, B.Sc.

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2017

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Yinghan Xu

August 28, 2017

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Yinghan Xu

August 28, 2017

Acknowledgments

I am heartily thankful to my supervisor Dr. John Dingliana for his guidance and support throughout my masters study which set up a strong understanding of the subject, which led to the completion of this thesis.

I would like to show my gratitude to all my classmates for helping me throughout the master.

I would also like to thank my parents for their support, trust, and encouragement.

Last but not the least, I would like to acknowledge and appreciate my girlfriend Xinyue, for spending her time to proofread this dissertation.

Yinghan Xu

University of Dublin, Trinity College

September 2017

Image-Based Rendering for Volume Visualization by Using Principle Component Analysis

Yinghan Xu, M.Sc.
University of Dublin, Trinity College, 2017

Supervisor: John Dingliana

This dissertation investigates and analyses imrovement to a PCA approach for image-based rendering in C++ based on paper [2]. This technique allows to visualize volume data with lower GPU requirement and lower computing complicity than direct volume rendering. First, eigenspaces were calculated with a volume data set, pre-rendered using a standard ray-caster, from a spherically distributed range of camera positions. The parallel programming with CUDA was used to speed up the reconstruction process. Then, PCA function in OpenCV is used to calculate eigenspaces and reconstruct results. Artefacts in the result of paper [2] were smoothed by MedianBlur with small computing costs. Results quality, memory cost and frame rate were tested and evaluated under different conditions. The result with different resolution of images, cells dimension, number of eigenvectors were given for real-world application.

Contents

List of Figures

Chapter 1

Introduction

After 3D modelling had being introduced in the early 1990s, computer graphics became ubiquitous over the past 20 years. The rise in use of computer graphics has enhanced computer technology and most home computers now can handle at least some of the rendering tasks.

Rendering is one of the major topics under computer graphics and it is broadly used in different industries. The idea behind rendering is basically about generating photorealistic or non-photorealistic images from 2D or 3D models. Most of objects with clear edges like buildings, weapons and characters can be rendered by polygons but for fluid or transparent objects, we need to apply volume rendering techniques instead of normal polygon-based models.

Volume rendering is an alternative approach to show objects. It gives the possibility to display more fuzzy surfaces and allows us to display 2D projection of a 3D discretely sampled data set. Volume rendering is typically popular and used for medical and engineering visulisations. Two advantages of volume rendering are listed below. By varying the color and transparency of each voxel, users can observe specific area of the whole volume data based on requirements, including interior features that would normally be

occluded.

In this project, I will introduce and explore PCA, which is one approach of image-based rendering. PCA can be used as an alternative to image-based volume visulisation. It gives us the opportunity to obtain similar results to volume rendering but with potential benefits in real applications. Further details and related works will be discussed in following chapters.

1.1 Motivation

With the rapid volume data expansion, the computation size becomes massive quickly. Interaction with volume data on home pcs will take up a huge amount of memory of GPU and some devices cannot afford it. Devices with low computing capability such as mobile devices will even provide a worse result. In such circumstances, 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.

In game industry, there are already existing image-based rendering techniques for complex background, trees, fire, grass and so on. It is hard to reconstruct the whole scene by 3D modelling for real world. Although with shaders we can achieve certain level of similarity to real world, it is never going to be the same because of slightly varying of luminance. Instead, Lumigragh, Plenoptic function are used to show the whole scene by image-based rendering. Models are not necessary any more. People can show the whole scene by reconstruct original images. For example, input data can be a set of images shot by rotating camera around a static object under complicated lighting condition. By linearly resampling original images or some other methods, people can reconstruct new images with different angles. Then, users can rotate and observe the whole scene by new

rendered images. The result could be quite real and handy than building this whole scene by 3D model and rendering by complex shaders. Moreover, input data could also be a set of images shot by moving cameras from one site to another site. By using image-based rendering, people could follow through the camera like in a video which is widely used in Virtual Reality applications.

Considering the better performance and lower cost for image-based rendering in some situations, several image-based rendering techniques were proposed to replace volume rendering. Colors and transparency could be adjusted by interpolate different values for input images.

1.2 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 can be used in real world application.

1.3 Contribution

The work in this paper extends on work proposed by Alakkari and Dingliana [2] for image-based volume rendering in PCA. I implemented the PCA function in C++ and speed up the reconstruction process with OpenCL module in OpenCV. To reduce artefacts, blurring was added onto 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 data set, 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 data sets.

1.4 Summary of Chapters

This dissertation is structured as follows:

- Chapter 2 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.
- Chapter 3 presents the design and implementations of the preprocessing, the CPU reconstruction, and the method of speeding up the reconstruction by using parallel programming. Smoothing function are used to blur artefacts for result images.
- Chapter 4 details the result gained from experiment. A discussion relating to results is also presented.
- Chapter 5 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 is dependent on lighting simulation, texture fidelity, shading information and viewpoint direction. With any change in the factors above, the result may be generated differently. However, when the scene is complicated or the subtle light effect in real world is difficult to simulate. Those traditional techniques might involve additional time and work, or moreover, even not workable in such case. In this situation, image-based rendering system was used to improve the realism of 3D model and to approximate global illumination effects.

The plenoptic function [1] 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 [23], McMillan and Bishop discussed the disparity of each pixel in stereo pairs of cylindrical images. At first, they used a cylindrical projection as the plenoptic sample representation instead of normal six planar projection onto a cube. Then, they reconstructed and re-sampled image warps that mapped sample images to arbitrary angle of

cylindrical viewpoints by using plenoptic function.

The work of [15] used a 4D function called Lumigraph. This system is used as a tool for sampling plenoptic function by handheld camera and then use the light in Lumigraph. Instead of using five parameters to represent position and direction in plenoptic function, they proposed a new way to reduce parameters needed to four by using 2D slices of 4D space. Direction is parameterized by using two parallel planes shown in figure 2.1. All the points can be represented by four coordinates in Lumigraph. Then, a discrete subdivision was applied onto a 4D space and associate a coefficient and a basis function (reconstruction kernel) with each 4D grid point.

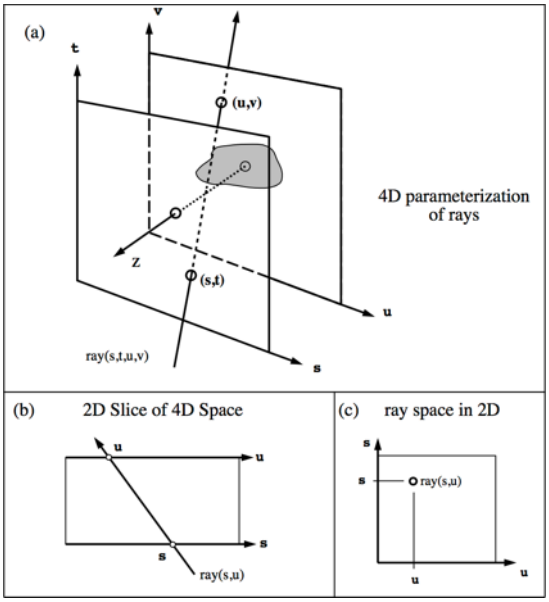


Figure 2.1: Parameterization of the Lumigraph[15].

[19] introduced 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 re-sampled 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, is another approach to do image-based rendering. The object of imposters is to reduce the complexity of rendering a 3D scene. For example, when we are modelling trees in video games, it requires additional work for each single leaf. Without sacrificing resolution, 3D objects could be easily represented as a 2D image rotating with viewers observed direction. There are two categories of imposters [6], static and dynamically-generated. Static imposter is usually created by artist to show 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 section 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 data set of this scene. We intended to reconstruct corresponding image from users interaction, rather than rotating one image with different angle according to camera position, like billboards mentioned in 2.1.

The paper by Alakkari and Dingliana [2] suggested to use Principal Component Analysis(PCA). PCA is a statistical method to compress the dataset and reconstruct images by back projecting process as a method to implement image-based rendering. It uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables. The number of 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 visualized as manifolds in low-dimensional subspaces and provides a useful mechanism for investigating face pose [14]. A reconstruction image could be generated by adding mean image of data set and

several number of highest eigentextures (sometimes called eigenfaces) multiplied with projected scores (figure 2.2).

Figure 2.2: Projection process (Reconstruction) [14].

Although image-based rendering is simple and handy for a stand-alone object without any background for the virtual reality, there are some disadvantages. For example, when we are placing stand-alone objects into real-world background, images-based rendering will not give us cast shadows under real illuminations. This will impair the performance of image-based rendering when its put into an environment. Trees or grasses in the game might not have the shadow, lack of reality. When combined with the real-world scene, image-based rendering could not be mixed together appropriately. Moreover, it is difficult to create the real-world model without losing details, such as tiny change of illuminations and complex structure of model surfaces. To solve this problem, Ko Nishino [28] 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, the texture of the model is shown properly and quickly. In the meantime, cast shadows are calculated in shader according to the 3D model. So, it gives a desirable control of the mixed reality system. The Eigen-Texture method they used is the same as ours. Cell-based PCA are calculated to reduce the number of the eigenvectors and to improve the texture quality. Differences are that they are using triangle as the cell shape and they summed several illuminations by linear function to the result. Their results were merged into the real-world while our results were only used separately.

2.3 Volume rendering

2.3.1 Direct volume rendering

Contrary to surface rendering, volume rendering (figure 2.3) is a technique to render 3D volume data to show the interior information of the object. It is used in medical industry and for some transparent objects, like fog, hair and grass. There are several ways to do 3D volume data visualization. The most basic volume rendering algorithm is ray-casting which uses straight-forward numerical evaluation of the volume rendering integral [10]. 2D texture [32] and 3D texture [39] mapping was used in these places. Without down-sample the result for better qualities, distributed volume rendering [11] were proposed to automatically crop and partition volumes into small volume blocks.

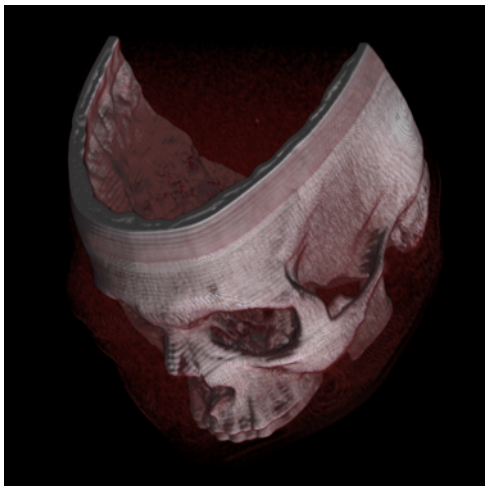


Figure 2.3: A volume rendered cadaver head using view-aligned texture mapping and diffuse reflection[5].

Ray-casting approaches [18] used an opacity threshold to adaptively terminate ray casting. Volume data were ordered as a front-to-back format. With striking the opaque object, opacity of this ray would accumulate. The colour of the ray becomes more and more stabilized. When exceeding a level of threshold, the ray casting would be terminated. The goal for this technique is to reduce image-generation time. The depth complexity of the scene would affect degrees of optimization.

Cell-projection methods [38] used projection of individual volume cells. Processing is cell by cell instead of pixel by pixel. Coherence were taken advantage of for projection pixels onto neighbors. Fourier volume rendering [21] presented a technique that rendering projected images of an scalar array with complexity $O(N^2 logN)$ after a pre-processing step. The 3-D Fast Fourier Transform (FFT) were used to transfer 3-D volume data into the frequency domain. Once the pre-processing operation were finished, "projection images at arbitrary angles can be quickly generated by re-sampling along a plane oriented perpendicular to the viewing direction and by taking an inverse 2-D transform of the resultant array."[21] However, there would be some blurry for result images. Shear-warp [17] technique proposed an object-order rendering algorithm based on the factorization. It can make sure rows of voxels in the volume are aligned with rows of pixels in the intermediate image. Image-order algorithms were operated by ray casting. Computation redundancy was appeared because of traversing all data for one ray. Object-order algorithms were operated by splatting voxels into the image, which cannot implement the ray termination process, leading to higher computation as well. Shear-warp factorization combined these two techniques by "a factorization of the viewing matrix into a 3D shear parallel, a projection to form a distorted middle image, and a 2D warp to produce the final image."[17]

2.3.2 Image-based volume rendering

Because of the large data size, it costs 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.

Chen [4] presented a way to render surfaces from volume data by integrating the fully volume rendered image (the keyview) and the model of the volume. First, Chen constructed

the surface model of the volume and texture mapping the keyview onto the geometry. Then he used cast-ray to render the rest of new region when changing the view slightly so that he can save several computing processes for visualizing volume data. [25] also proposed a similar way for keyview image-based rendering. The difference is they controlled compounding errors by changing the thickness between slabs. The texture-mapped polygons would transform according to viewers angle by using a scene graph and commodity graphics boards.

Meyer [24] came up with an image-based volume rendering with opacity light fields. They separated the process to three parts, 1.getting ray slices and proxy surface using a standard volume rendering application on any type of volume data, 2.generating new key views for each ray slice viewpoint and 3.interactively rendering as opacity light fields. This approach can take advantage of different features of the hardware like leaving the heavy rendering work to the good graphics cards and combining the result in remote application. A main difference from image-based volume rendering work in [4] is their technique allows to change view to arbitrary angle.

Tikhonova [36] 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, making it possible to run on low-end hardware.

2.4 Parallel programming

2.4.1 CUDA

To satisfy increased demand and expected improvement of 3D graphics, GPU has evolved into a highly parallel 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)[29]. Apart from using parallel programming on graphics, there are more and more applications involved CUDA to speed up calculations[27]. CUDA language is a minimal extension of the C and C++ programming languages. A serial program could be written as simple functions or full programs. These functions are called parallel kernels. Each kernel could be executed as a threads and a set of parallel threads could be executed together. A block is a set of concurrent threads that can shared memory space and cooperated among themselves through barrier synchronization. A grid is a set of blocks that may each be executed independently and thus may execute in parallel. CUDA is supported by 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[30]. Only devices support CUDA can run CUDA kernel for acceleration.

2.4.2 OpenCL

Open Computing Language (OpenCL)[26] is a framework for writing programs that execute across heterogeneous platforms. It defines a C-like language for writing programs. Functions executed on an OpenCL device are called “kernels”. [13] Inside a kernel, multiple processing elements are run on parallel. OpenCL provides methods for accessing to GPUs and it could be used for non-graphical computing (GPGPU). The GPU runs multi-

thread kernel to speed up the computing process. 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.

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[34]. PCA is a statistical technique to find patterns in data of high dimension. For our data sets, all input images are assumed as observed objects with high dimension (Each pixel is seemed as one dimension). After calculating patterns (eigenvectors) from these data, we can use these patterns to reconstruct original data by back projection.

3.1.1 Put the observed object into the matrix

The observed objects should have same dimension. For example, all the training images should have the same dimension. Every image is put into one vector. If the image is $a \times b$ dimension, the vector size is $a \times b$ (The image size is assumed as $d \times d$ in this place). If the training images number is n, the matrix with n rows and $a \times b$ columns is formed as

follow,

$$M = \begin{pmatrix} vec_1 \\ vec_2 \\ vec_3 \\ . \\ . \\ . \\ vec_n \end{pmatrix} \tag{3.1}$$

where vec_i represents the i_{th} images all pixel values (For one channel), $i \in [1, n]$.

3.1.2 Subtract the mean

Adjusted data are variances for each pixel. It is calculated by using original matrix to subtract mean matrix. Subtract mean matrix is obtained by calculating average value of original matrix in column. Adjusted data is calculated as below,

$$DataAdjust = M - vec_{mean} \tag{3.2}$$

where M is the result from equation 3.1, vec_{mean} is the mean vector with $d \times d$ elements.

3.1.3 Calculate the covariance matrix

Covariance is a similar way to standard deviation and variance to perform "how much the dimensions vary from the mean with respect to each other"[34] For example, two vectors X and Y with 10 elements in each vector,

$$Covariance\,(X, Y) = \frac{\sum_{i=1}^{10} \left(X_i - \overline{X}\right)\left(Y_i - \overline{Y}\right)}{9} \tag{3.3}$$

where $\overline{X}$ is the mean of vector X, $\overline{Y}$ is the mean of vector Y.

Covariance matrix is used to measure multi-dimensional vector variance from their mean.

For a matrix with n rows. The covariance matrix is calculated as below,

$$Covariance\_matrix = \frac{1}{n-1} \times DataAdjust \times DataAdjust^T \tag{3.4}$$

where $DataAdjust$ is calculated by equation 3.2, $DataAdjust^T$ is the transpose of the $DataAjust$, n is the number of observed vectors. If the $DataAjust$ is n rows with $d \times d$ columns, the result of covariance matrix should be $d \times d$ rows with $d \times d$ columns.

3.1.4 Calculate the Eigenvectors and Eigenvalues

Eigenvectors can be calculated by solving the equation as below,

$$(A - \lambda I)\, v = 0 \tag{3.5}$$

where v is an eigenvector of the linear transformation A, the scale factor λ is the eigenvalue corresponding to that eigenvector and I is the n by n identity matrix. In this place, the previous result calculated as covariance matrix is the A matrix in equation 3.5. The number of eigenvectors and eigenvalues should be $d \times d$. Each eigenvector has $d \times d$ elements.

3.1.5 Sort Eigenvalues

After getting the eigenvectors and eigenvalues, I sorted those eigenvectors from the highest to lowest by eigenvalues. 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

Although its possible to use all eigenvectors to reconstruct the original observed vector, we can also reconstruct with just a few of the highest eigenvectors but with some data

missing. It points out that the more eigenvectors we use, the higher quality result we will get. For an original observed vector, the projection value (referred to as a score) is calculated by projecting one eigenvector onto the original vector as below,

$$S_k = vec_i \times {eigenvector_k}^T \tag{3.6}$$

where vec_i is i_{th} original vector, ${eigenvector_k}^T$ is the transpose of k_{th} highest eigenvectors, and S_k is the score of this eigenvector projected onto i_{th} original vector, $i \in [1, n]$, $k \in [1, d \times d]$.

3.1.7 Reconstruct 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. This step is the only step at run-time and previous steps are preprocessing. The equation is as follow,

$$vec_{reconstructed} = vec_{mean} + S_1 \times eigenvector_1 + S_2 \times eigenvector_2 + \cdots + S_{num} \times eigenvector_{num} \tag{3.7}$$

where $eigenvector_i$ is the i_{th} highest eigenvector of this original vector, S_i is the corresponding i_{th} score for this highest eigenvector, $vector_{reconstructed}$ is the result of the reconstructed vector, $i \in [1, num]$, num is the number we choose as components, $num \leq d \times d$.

3.1.8 Whole image PCA and cell-based image PCA

As we discussed above, an observed vector has $imageWidth \times imageHeight$ dimensions. Paper [2] proposed a method named cell-based PCA to crop the whole image into same dimension cells. In this case, we just need smaller number of eigenvectors to reconstruct original cells. Then we can put these reconstructed cells together. The result of whole image is nearly the same, while cell-based method could save more memory costs.

3.2 Reconstruction with CPU

To build up an initial prototype of the reconstruction and save the preprocessing time, I used 36 training images at a resolution of 300×300 pixels. These images were obtained by rotating the camera around the head of a textured polygonal model (a solider character obtained from [12] and rendered using basic Phong illumination) horizontally through 360 degrees with 10 degree increments. 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 [2], I used the same cell-dimension 20×20 to create an observed matrix with 36 rows and 400 columns. 20×20 pixels values were put into one row (vector). Each column stored one pixel value. In this case, there were 225 observed matrices 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 RGB image has three channels and the specific locations pixel value was obtained by the corresponding width and height location. For OpenGL, this can be done by SOIL library[20]. Then, the image pixel value can be accessed by a char array. Eigen library [9] was used here for matrix calculation. 255 observed matrix is represented as below,

$$Matrix_1 = \begin{pmatrix} Image1Cell1 \\ Image2Cell1 \\ Image3Cell1 \\ . \\ . \\ . \\ Image36Cell1 \end{pmatrix} Matrix_2 = \begin{pmatrix} Image1Cell2 \\ Image2Cell2 \\ Image3Cell2 \\ . \\ . \\ . \\ Image36Cell2 \end{pmatrix} \cdots Matrix_{255} = \begin{pmatrix} Image1Cell255 \\ Image2Cell255 \\ Image3Cell255 \\ . \\ . \\ . \\ Image36Cell255 \end{pmatrix} \tag{3.8}$$

3.2.2 Calculate eigenvectors

According to 3.2.1, 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,

$$Covariance\_matrix_i = \frac{1}{35} \times DataAdjust_i \times {DataAdjust_i}^T \tag{3.9}$$

where $DataAdjust_i$ is i_{th} adjusted data of $Matrix_i$, ${DataAdjust_i}^T$ is the transpose of $DataAdjust_i$, $Covariance\_matrix_i$ is corresponding covariance matrix of $DataAdjust_i$, $i \in [1, 255]$. In Eigen library, the $EigenSolver$ () function is used to calculate eigenvectors and eigenvalues by offered covariance matrix. Then, the bubble sorting method can be used to sort eigenvalues and the corresponding eigenvectors.

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 image 2D texture. New texture was bound and rendered for each frame in OpenGL.

3.3 Reconstruction with CUDA (GPU)

3.3.1 Parallel programming design for reconstruction

GPU is powerful for multithread computing. There are several ways to complete a complex computing. In my reconstruction process, inputs were eigenvectors and scores read from files. The output was 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 pseudo-code is as below,

Algorithm 1 CPU reconstruction

1: **for all** $i \leq n$ **do**
2: **for all** $j \leq num$ **do**
3: $vec_{reconstructed} = vec_{reconstructed} + S_i \times eigenvector_i$
4: **end for**
5: $vec_{reconstructed} = vec_{reconstructed} + vec_{mean}$
6: $putCellIntoImage\,()$
7: **end for**

n is the number of cells, num is the number of components we chosen, vec_{mean} is the mean vector for each cell, $putCellIntoImage\,()$ is the function that resizes reconstructed cell data from row to corresponding dimension and put the specific cell to corresponding position as 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 pseudo-code is as below,

Algorithm 2 CUDA reconstruction

1: **for all** $i \leq n$ **do**
2: $cudaSpeedUp\,()$
3: $putCellIntoImage\,()$
4: **end for**

Everything was the same as CPU computing except that the reconstruction loop and adding mean process are replaced by the $cudaSpeedUp\,()$ function.

3.3.2 CUDA Implementation details

The data structure must be point array in order to transfer data from host memory to device memory and 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 matrices used by Eigen library to custom host-vector by host-vector variables in CUDA language.

Then, memory in device were allocated by function $cudaMalloc(\&d\_arry, size)$, where $\&d\_arry$ is the double array and $size$ 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 $(dev\_dim, 1)$, where dev_dim is the dimension of each cell images, because it can only handle one pixel value in one thread.

Summation of scores multiplying eigenvectors could be speeded up by one kernel $arradd1$ and summation with mean vector could be speeded up by another kernel $arradd2$. Kernel $arradd1$ and $arradd2$ are using the following code.

```
__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 pseudo-code of function $cudaSpeedUp()$ is as below,
In kernel $arradd1()$, d_m is the eigenvector of cell images, d_n is result for one cell image, $size$ is the size of eigenvector, $score$ is the projectedValue. In kernel $arradd2()$, d_m is the mean vector of one cell image, d_n is the result of one cell image, $size$ is the size of mean vector.

Algorithm 3 cudaSpeedUp

1: **for all** $i \leq pca\_dim$ **do**
2: $score = savedMatrix\,(i)$
3: $arradd1\,(d\_m, d\_n, size, score)$
4: **end for**
5: $arradd2\,(d\_m, d\_n, size)$

3.4 Reconstruction with OpenCV

Calculating 225 sets of eigenvalues and eigenvectors for a 400 by 400 covariance matrix needs 20 hours, and additionally RGB three channels need to be calculated. Other than 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 provide a 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 reduces pre-processing time by using OpenCL to calculate eigenvalues and eigenvectors. I was focusing on the reconstruction process and series experiments of the reconstructed result, so 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 sample data. This dataset is also used in [2].

In this PCA class, the main function of $PCA\,(data, mean, flag, maxComponents)$ would return a set of matrices, where *data* represented the input data, *mean* represented the mean vector on rows or columns (depends on what flag used) of input data, $flag$ represented the observing type of the input data, and $maxComponents$ represented 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 20×20, the data matrix would be 400 by 900. Even though we can calculate mean vector of theses 900 vectors, we can still leave *mean* parameter to empty($Mat\,()$). PCA function automatically calculates mean vector for us. $flag$ parameter can 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 $flag$ was $DATA\_AS\_ROW$. If there is no value or the value is larger than the size of observed vector in the last parameter $maxComponents$, the number of returned eigenvectors and eigenvalues will be the size of observed vector. Otherwise, this $maxComponents$ number will make sure the returned set only retain specific number of eigenvectors and eigenvalues. Result of this function is a set of matrices. 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. Pseudo-code for cropping to cells and calculating PCA is as below,

Algorithm 4 PCA with OpenCV

```
for all i ≤ image_height / cell_dimension do
    for all j ≤ image width / cell_dimension do
        for all k ≤ image_num do
            loadImage (k)
            cropImage (i, j)
            saveCellToMatrix (i, j)
        end for
    end for
end for
for all i ≤ cell_num do
    calculatePCA (data, Mat, DATA_AS_ROW, num_components)
    savePCAResults ()
end for
savePCAResultsToFile ()
```

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 positions 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 matrices, eigenvalues, eigenvectors and mean, in $FileStorage()$ function, I used 'eigenvalues', 'eigenvectors' and 'mean' tag to save them in one file for future convenience to read the data.

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

Algorithm 5 Calculate and save scores

```
1: for all i ≤ image_num do
2:     for all j ≤ cell_num do
3:         score = pca[j].project(cells[j].row[i])
4:         saveScoreResult()
5:     end for
6: end for
7: saveScoreResultToFile()
```

where *cell_num* is the number of cropped cells, $pca[j]$ is j_{th} cell pca result, *project* () is a method in pca to project eigenvectors into original data vector, $cells[j].row[i]$ is to get the i_{th} row vector in j_{th} 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, tagged with 'scores'.

3.4.2 Reconstruction

The reconstruction process with PCA class in OpenCV is simple and clear. PCA result has already packed eigenvectors and mean vector. Scores vector were also calculated by

projection in section 3.4.1. Pseudo-code for reconstruction is as below,

Algorithm 6 Reconstruction by PCA function (OpenCV)

```
1: for all i ≤ image_height / cell_dimension do
2:     for all j ≤ image width / cell_dimension do
3:         pca = getOnePCAResult (i, j)
```

```
4:         score = getOneScoreResult (i, j)
5:         result = pca.backProject (score) .reshape (1, cell_dimension)
6:         copyResultToWholeImage (i, j)
7:     end for
8: end for
9: normalizeResultImage ()
10: mergeBGRChannels ()
11: copyResultImageToTexture ()
```

where $pca = getOnePCAResult\,()$ function is to get one of PCA result from the file, which saves all PCA results, $score = getOneScoreResult\,()$ function is to get one of score vector from the file, which saves all score vectors results, $backProject\,()$ method in PCA is discussed in section 3.1.7, implementing the equation 3.7, $reshape\,()$ 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, $normalizeResultImage\,()$ 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[7] 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 was much less than I reconstructed with Eigen library. After digging into the source code of PCA class in OpenCV, I found out that they used OpenCL parallel programming to speed up matrix arithmetic, discussed in section 2.4.2.

If the device supports OpenCL[35], input data are passed to a *ocl_gemm* () function, the kernel named ocl::Kernel with tag 'gem' are executed. Details inside kernel are not discussed here, because its 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, pseudo-code as below,

Algorithm 7 CPU reconstruction by OpenCV

1: *initResultMatrix* ()
2: **for all** $i \leq image\_num$ **do**
3: $\quad result = result + scores\,[i] * pca.eigenvectors.row\,[i]$
4: **end for**
5: $result = result + pca.mean$

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 3.7, 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 [2]. 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, a smoothing function *medianBlur* () was used. It also helped reduce computing cost. To perform a smoothing operation (aka. Blurring operation), a filter was added to an image. In OpenCV, there are several most used filters, such as blur, GaussianBlur, medianBlur, bilateralBlur[3]. Theories and features are discussed as below and the reason for using medianBlur beyond other algorithms is discussed.

3.5.1 Mean blur

Linear is the most commonly used filter. The 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 this filter is the mean. One advantage for this mean filter is its low computing cost. The kernel is as below,

$$K = \frac{1}{K_{width} \cdot K_{height}} \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & 1 & 1 & \dots & 1 \\ . & . & . & \dots & 1 \\ . & . & . & \dots & 1 \\ 1 & 1 & 1 & \dots & 1 \end{bmatrix}$$

Figure 3.1: Kernel for blur filter[8].

3.5.2 Gaussian Blur

GaussianBlur filter is normally the most useful filter for noise reduction in an image. "It is done by convolving each point in the input array with a Gaussian kernel and then summing them all to produce the output array"[3]. The pixel located in the middle of Gaussian curve has the highest weight. The weight of its neighbours decreases as the spatial distance between them. Artefacts between cells are not equivalent to noise of the result image, so this higher computing cost filter was not used in real-time reconstruction process in this project.

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 (3×3 in a kernel). Artefacts of observed image are preserved. The discussion about better preservation of edges for medianBlur filter is in this paper[3].

3.5.4 Bilateral Blur

BilateralBlur filter is another non-linear filter. It is similar to GaussionBlur filter. It 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. This method was also discarded because of relatively higher computing cost in real-time reconstruction process. As I explained above, both commonly used mean blur filter and non-linear medianBlur filter can 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 are discussed in chapter 4).

Chapter 4

Results and Evaluation

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

4.1 Evaluation methods

The evaluation is based on images reconstructed using the techniques presented in this chapter. These images are lossy compression from the original data set. The technology with results generated can be applied in medical research or analytics and games, when humans are observers. However, images viewed by human beings are usually quantified visual image quality by subjective evaluation, which is unconvincing. In practice, people proposed methods to evaluate compressed, blurred or reproduced image's quality with the original image. The structural similarity (SSIM) and HDR-VDP are some of the most popular methods to develop quantitative measures that can automatically predict perceived image quality.

4.1.1 SSIM

One of the easiest and most broadly used full-reference quality matrices is the mean squared mistake (MSE). It is processed by averaging the squared force contrasts of misshaped and reference picture pixels, along with the related quantity of peak signal-to-noise ratio (PSNR) [37]. The difference between SSIM with former method is that SSIM approaches estimate absolute errors. Moreover, as a perceptual model, SSIM considers images degradation as perceived change in structural information. Some perceptual phenomena are also considered, such as luminance masking and contrast masking terms. Structural information uses strong inter-dependencies as relationships for pixels, which lays closely. The visual effect of structure for one scene could be represented favourably in these dependencies information. In my experiment, a free source code named scikit-image[33], which is a collection of algorithms for image processing, is used to calculate SSIM. The SSIM function provided in this kit needs two images as input parameters, and then a quality metric will be returned.

4.1.2 HDR-VDP-2

[22] proposed an algorithm to calculate the metric based on a new visual model for all luminance conditions, which has been derived from new contrast sensitivity measurements. HDR-VDP-2 is shown to provide much improved performance compared to the original HDR-VDP and VDP metric. The image quality predictions are regarded to be the same or better performance than SSIM method. The advantages [16] for HDR-VDP-2 include separate predictions for visibility and quality. Separate predictions serve very different purpose and are not necessarily very well correlated. It is also extensively tested, calibrated against actual measurements to ensure the highest possible accuracy. The source code for Matlab is provided in [16].

There are five parameters in HDR-VDP-2 function. The first two are tested image and

referenced image. The third one is colour encoding. 'sRGB-display' is used in our data set, standard (LDR) colour images. The fourth one is pixels per degree. This parameter specifies the angular resolution of the image in terms of the number of pixels per one visual degree. The typical value 30 for a standard resolution computer display is used. The last one is options, which we leave it to null. The outputs are a structure. It includes probability of detection per pixel (matrix 0-1), a single valued probability of detection (scalar 0-1), threshold normalized contrast map, maximum threshold normalized contrast and quality correlation. Quality correlation is 100 for the best quality and if quality is reduced, Q value will decrease correspondingly. Q can be negative in case of very large differences. Q value are only used in our experiments.

4.2 Reconstruction

4.2.1 Small training images results

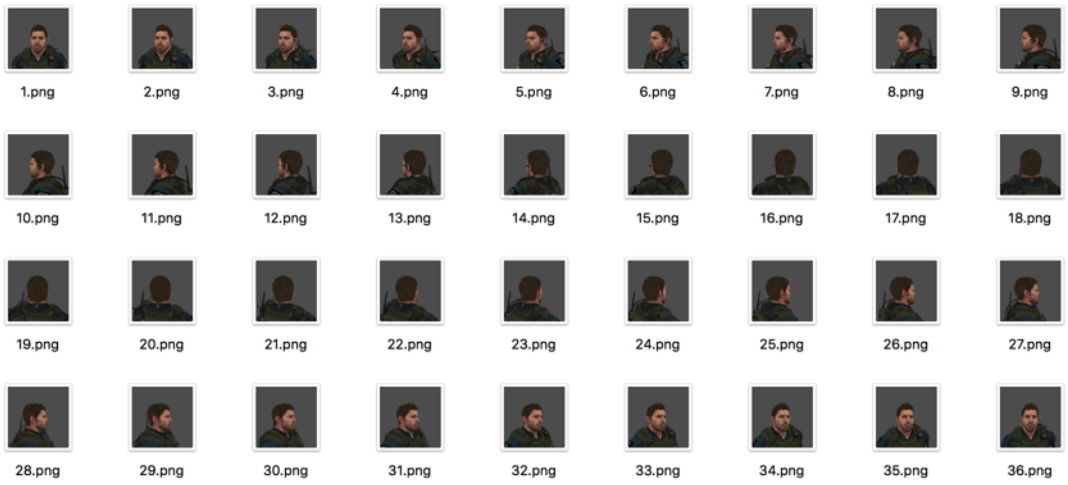


Figure 4.1: Head training data with 36 images.

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.

Figure 4.2: Left: reconstructed by 15 eigenvectors. Right: reconstructed by 20 eigenvectors.

Figure 4.3: Left: reconstructed by 30 eigenvectors. Right: original image.

4.2.2 Reconstruction speed comparison between CPU and GPU (CUDA)

I do not focus on the relationship about result quality, number of components and frame rate in this section. Instead, the reconstruction process is focused on at first. Calculation for reconstruction based on CPU is 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 shown as below,

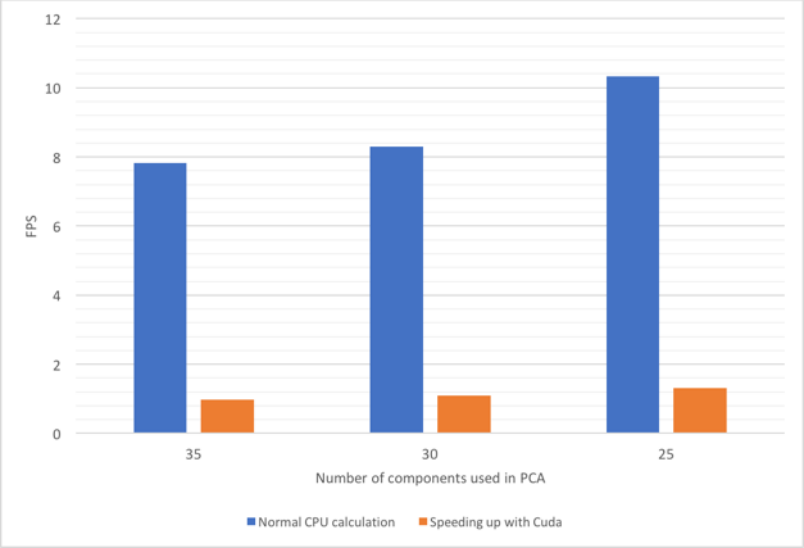


Figure 4.4: Reconstruction speed comparison between normal CPU and CUDA.

From figure 4.4, we can see that construction time in CPU will increase with increasing number of components used, while this process time in CUDA (GPU) only increase a little. Furthermore, the average construction time of GPU is around 6 times faster than CPU.

4.2.3 Large training images results

The dataset using here is the same one used in the previous paper by Alakkari and Dingliana [2], comprising 900 training images pre-rendered by standard volume rendering

from uniformly-spaced viewing angles (3.6 spacing for the azimuthal angle and 20 spacing for the elevation angle). In this section, downsampled dataset with resolution 720×720 are used for these result images. Other resolution data set should exhibit the similar result, while the only difference is the performance varying with different cells size and different number of components used (analyzed in section 4.4.2). More detailed results of PCA will not be shown here, since details have already been discussed in referenced paper[2]. For a more convenient visualization comparison, only one of the 900 views is shown in Figure 4.5.

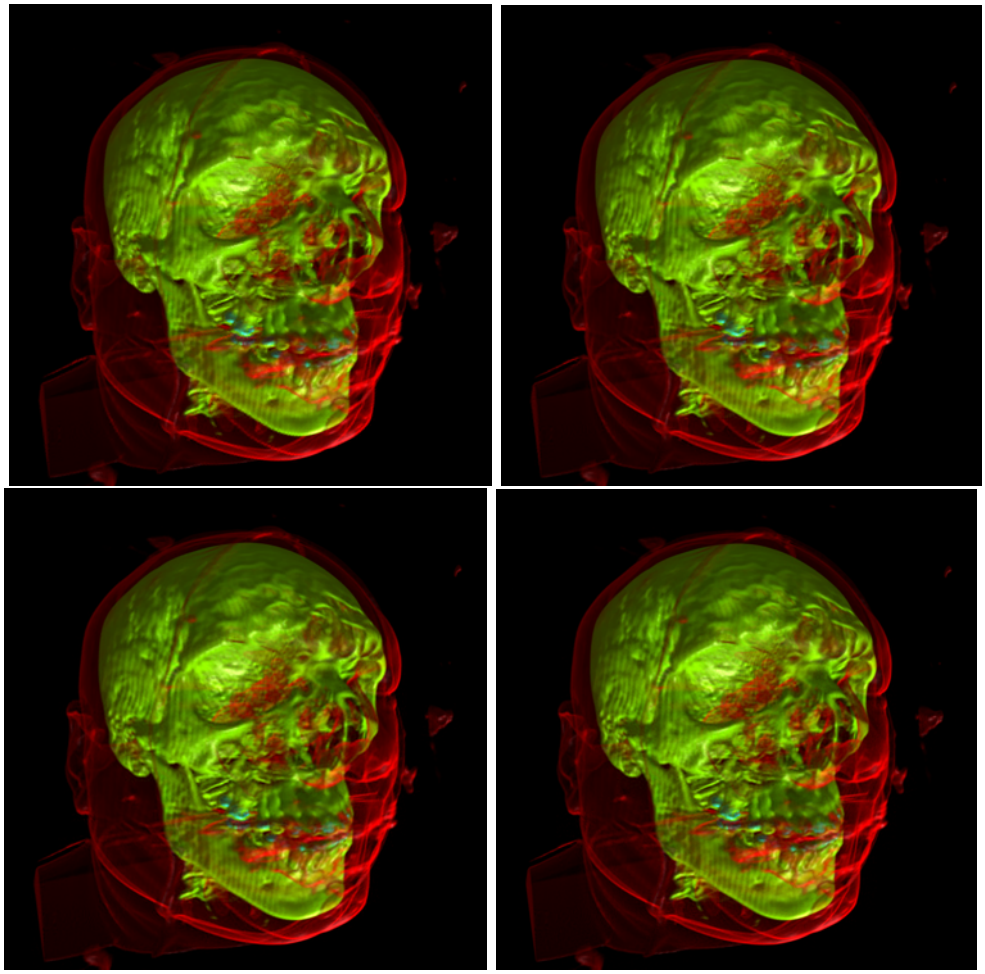


Figure 4.5: The SSIM value for these four images are all around 0.967. The top left image used 4 eigenvectors with 5*5 cell dimensions. The top right image used 13 eigenvectors with 10*10 cell dimensions. The bottom left image used 40 eigenvectors with 20*20 cell dimensions. The bottom right image used 90 eigenvectors with 30*30 cell dimensions.

Figure 4.5 shows one image from the data set reconstructed using different number of eigenvectors and cell-dimensions. We cannot see any obvious visual differences because the SSIM is quite high at this image resolution. Different scenes require different dimensions of results. If we zoom in to the extend shown as below, artefacts between cells are noticeable, which affect the performance of results. Images in figure 4.6 show zoomed in area of figure 4.5.

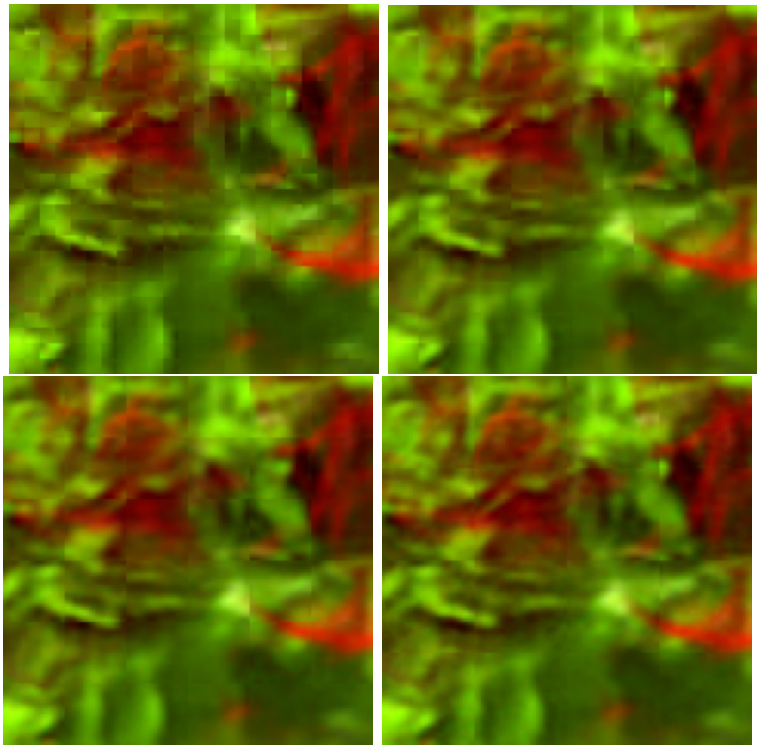


Figure 4.6: These four zoomed in images are the same part of area (120*120 pixels) corresponding to figure 8 four images.

From figure 4.6, we can see that the top left image (5*5 cell dimension) has the largest number of artefacts between cells, while the bottom right image (30*30 cell dimension) has the lowest number of artefacts between cells. Even though the result quality for both images are the same, the second one uses more number of components, which means higher memory requirement in reconstruction process. Details of relationship on result quality and memory cost are discussed in section 4.3.

4.2.4 Reconstruction speed comparison between CPU and GPU (OpenCL)

The next test was conducted on a machine with macOS Sierra and with 2 GHz Intel Core i5 processor, 8.0G memory and Intel Iris Graphics 540.

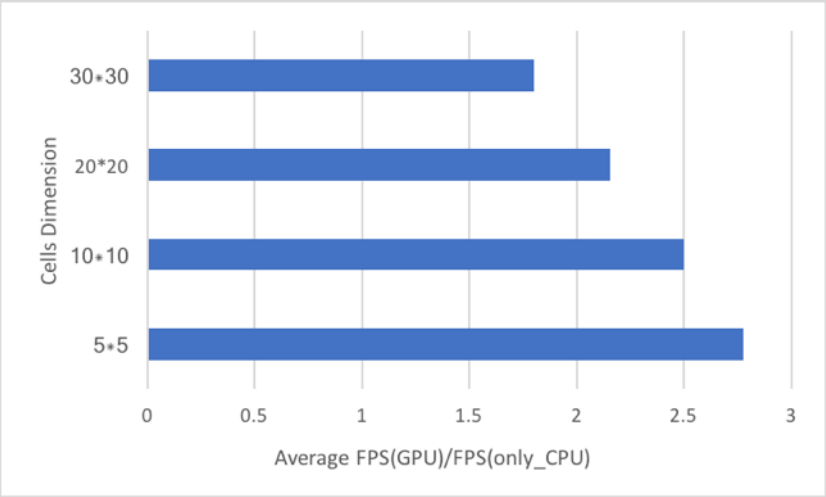


Figure 4.7: FPS performance comparison between GPU and only CPU with different cells dimension.

According to figure 4.7, reconstruction by GPU will speed up around 1.8 times to 2.8 times than by only using CPU. The fluctuation between the speeding up ratio is caused by different cells dimension. In 4.2.2, we compared CUDA (GPU) with CPU, which is 6 times faster for former method. However, after compared using CPU and CUDA by 720×720 resolutions images, I found that FPS is much lower when using CUDA (tested in machine with Intel(R) Xeon(R) CPU E3-1240 v3, 3.40GHz, 16.0G memory and NVIDIA Quadro K2000 graphic card). One possible reason is that the previous CPU reconstructions by Eigen library matrix are not efficient as CPU reconstructions by OpenCV library matrix.

In PCA class, OpenCV uses OpenCL to speed up the matrix arithmetic. When computing by GPU, there will be some delay for copying data from device memory to host

memory and from host memory back to device memory. If this transferring process are too long, the delay would influence the performance of GPU speeding up process significantly. As discussed in section 3.4.3, each cell reconstruction process is inside a loop. Loop times are decided by the number of cells, which means the more cells we use, the longer loop times will be. The specific experiment for this is discussed in section 4.4.

4.3 Smooth artefacts of edges

4.3.1 Blur and Median Blur Comparison

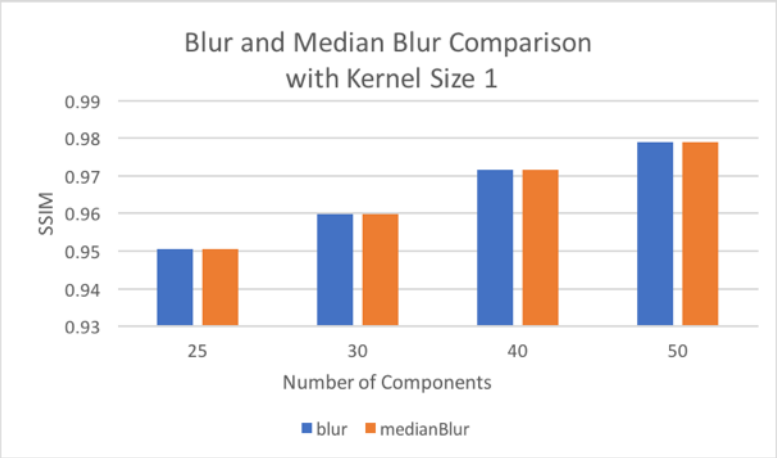


Figure 4.8: Blur and Median Blur Comparison with Kernel Size 1.

All the experiment in this section is based on 720×720 images with 20×20 cell dimension reconstructed images. According to figure 4.8, 4.9 and 4.10, we can see that even though SSIM value does not have much differences between blur algorithm and median blur algorithm when the kernel size is 1, median blur algorithm demonstrates higher quality results when the kernel size increasing. As I discussed before, this might because the median blur algorithm would smooth noises except edges of the image, which could save more details for result images and just smooth artefacts. So, for my implementation, there was an option for smoothing result images in real time. In the later section, I only use Median Blur for smoothing.

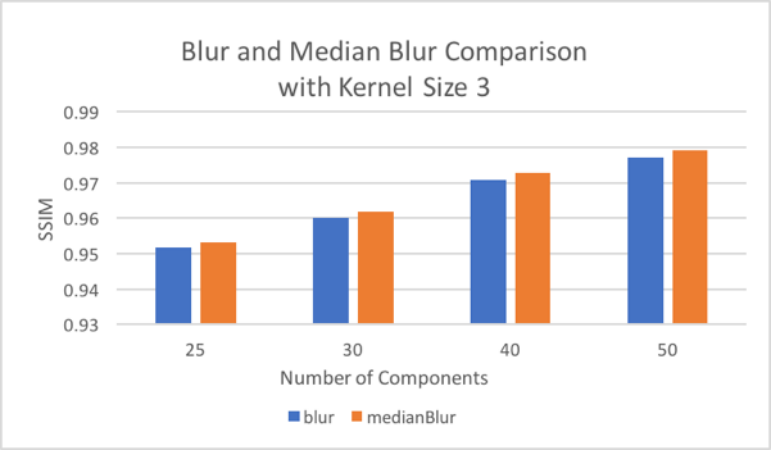


Figure 4.9: Blur and Median Blur Comparison with Kernel Size 3.

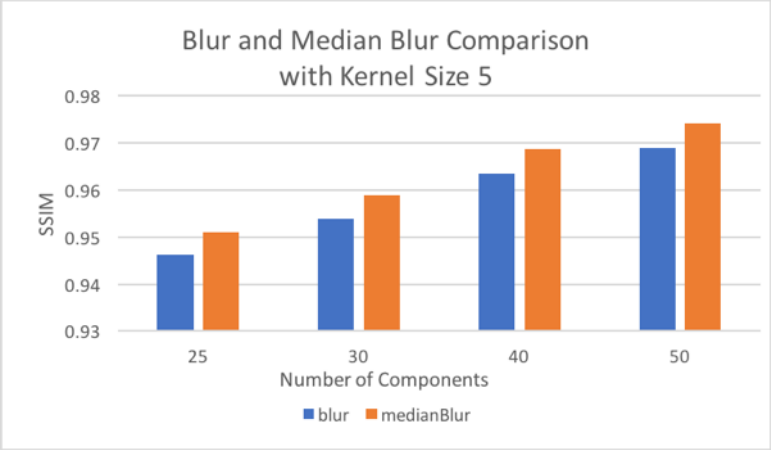


Figure 4.10: Blur and Median Blur Comparison with Kernel Size 5.

4.3.2 Kernel size chosen for Median Blur

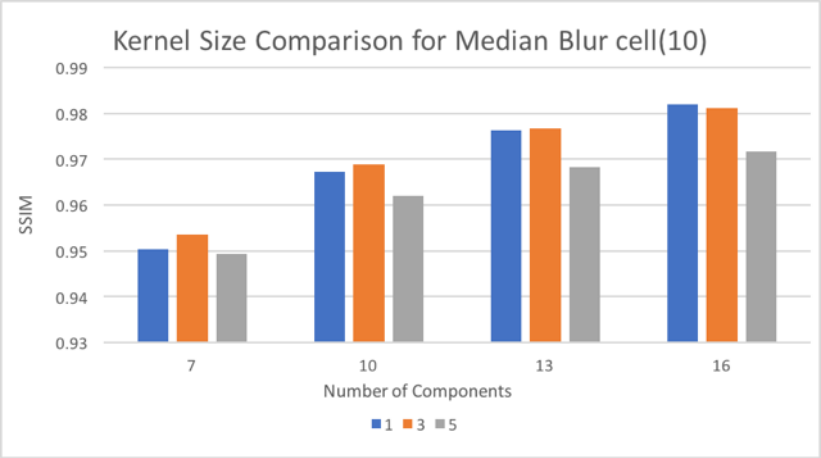


Figure 4.11: Cells dimension are 10*10. Image quality with the different size of kernel in different number of components.

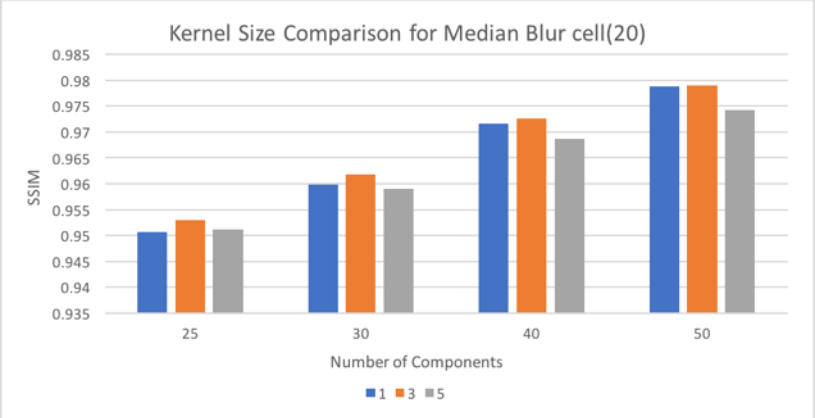


Figure 4.12: Cells dimension are 20*20. Image quality with the different size of kernel in different number of components.

According to results from figure 4.11, 4.12 and 4.13, image quality will not change much in different size of cells. It is because artefacts between cells edges are always 2 pixel lines. Inside each chart, quality differences will decrease when the result image is more similar to the original one. Kernel size 3 always has superiority than kernel size 1 and kernel size 5 except when the result image and original image are very alike. Kernel sizes bigger than 7 are not considered because kernel size 3 shows the peak result for

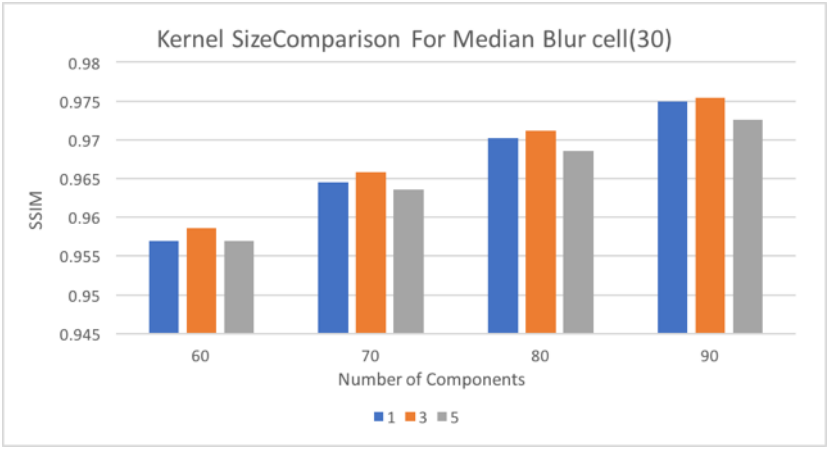


Figure 4.13: Cells dimension are 30*30. Image quality with the different size of kernel in different number of components.

quality all kernel size in general. So, comparisons are among kernel size 1, 3 and 5. In general, kernel size 3 should be used in most conditions except the result image quality is very high, which doesn't need too much smoothing for artefacts. At the same time, if cells dimension is very small, in case 10×10 cells in 720×720 image, kernel size should be 1 when quality value is higher than 0.97. In my application, kernel size and smoothed function are users input which can be changed in real time.

4.3.3 Comparison between smoothed images and unsmoothed images

In figure 4.14, under same condition, smoothed images show higher SSIM value comparing to unsmoothed images. The difference occurred when different cells dimension and number of components were used. Differences between smoothed images' SSIM value and unsmoothed images' SSIM value was larger. The smaller cells dimension led to larger differences. In total, if cells dimension is smaller, smoothed images would have a better result. Possibly small cells dimension has large number of cells, which means more artefacts will be smoothed.

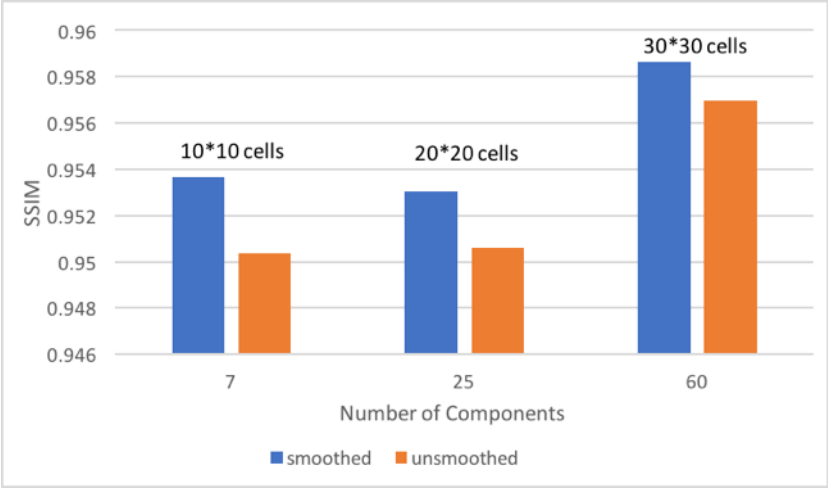


Figure 4.14: The dataset used here is 720*720 head images. The chart compares qualities (SSIM) between smoothed image and unsmoothed image in different cells dimension. Reconstruction with 10*10, 20*20 and 30*30 cells are shown here. The kernel size 3 used in smoothed image.

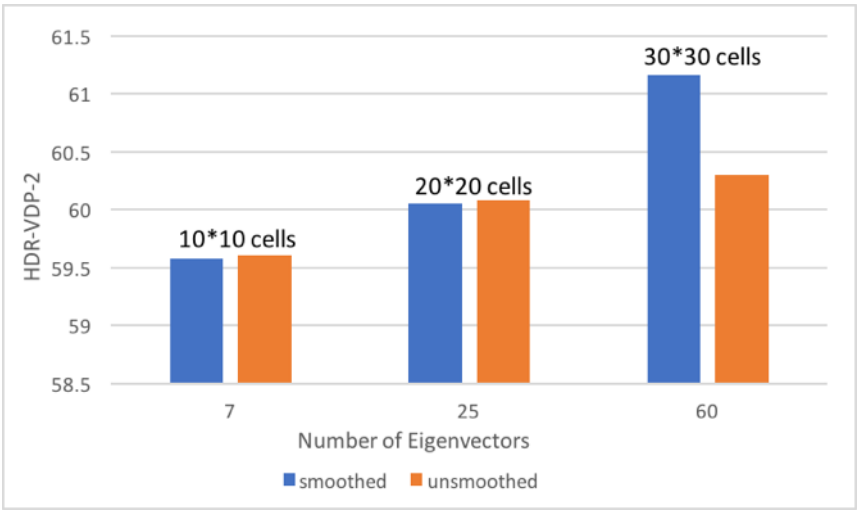
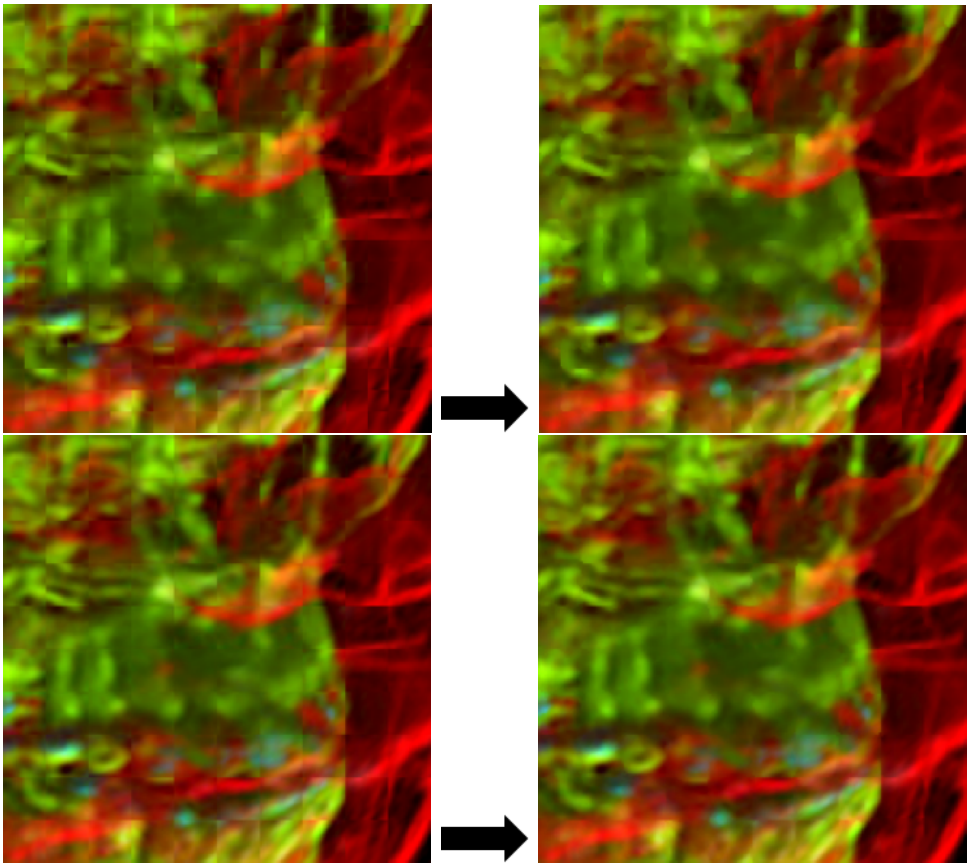


Figure 4.15: The dataset used here is 720*720 head images. The chart compares qualities (HDR-VDP-2) between smoothed image and unsmoothed image in different cells dimension. Reconstruction with 10*10, 20*20 and 30*30 cells are shown here. The kernel size 3 used in smoothed image.

The same experiment data set and the same setting of parameters were used for HDR-VDP-2 shown in Figure 4.15. The result shows that when cells dimension is relatively small, 10×10 and 20×20 in this place, image qualities are nearly the same for smoothed images and unsmoothed images. When cells dimension gets larger, for example 30×30, image qualities are much improved for smoothed images. Even though SSIM and HDR-VDP-2 provided different results, rough qualities for smoothed images showed a better performance than unsmoothed images. In Figure 4.16, 200×200 pixels zoomed in area comparison are shown. For human eyes, smoothed images shows a better visual effect than unsmoothed and the 10×10 cells dimension shows the largest variation. However, the pixels per degree used in HDR-VDP-2 was 30, which were suitable for un-zoomed in images. Whole image comparisons are shown in Figure A.1.



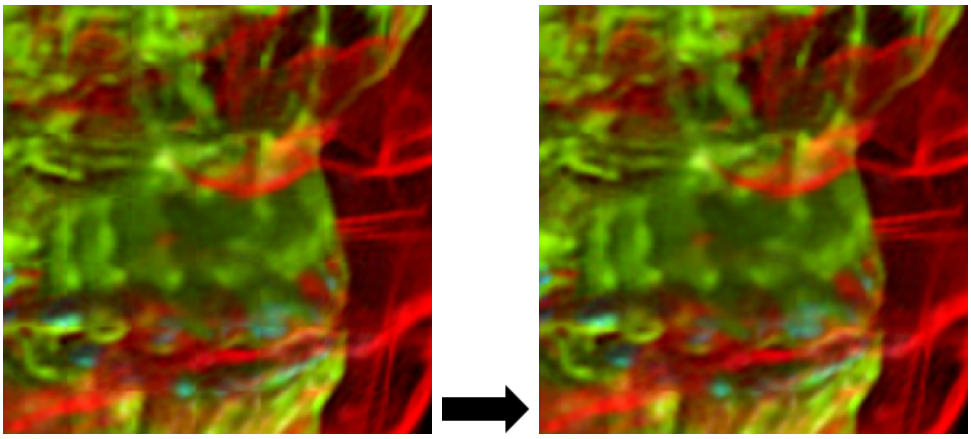


Figure 4.16: Zoomed in (200*200) comparison between unsmoothed images and smoothed images. The left column is unsmoothed images. The right column is smoothed images, whose kernel size is 3. The first row's images are constructed by 7 eigenvectors for 10*10 cells dimension. The second row's images are constructed by 25 eigenvectors for 20*20 cells dimension. The third row's images are constructed by 60 eigenvectors for 30*30 cells dimension.

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

As we discussed in Chapter 2, image-based rendering is an approach for showing result with lower computing cost and memory cost. The PCA method is used to reduce the memory cost in image-based rendering. But in practice, because of the process for reconstruction, there might be some time delay, which could influence the quality of result images and the frame rate. In real-world applications, users can use the result in medical industry or video games. During an operation, doctors need higher image quality for patients' CT, while in such case, the frame rate is not the major concern. However, when people need to visualize the transparent objects in mobile devices, people will be more concerned about the real-time animations so image quality becomes less important than the frame rate. There are four variables we are able to change, number of images, resolution of images, cells dimension and number of eigenvectors used in reconstruction.

4.4.1 Same resolution data set

The data set I have used in my experiment are 900 head images with resolution 720×720, while four different kinds of cells dimension are tested. In this section, I focused on the relationship between image quality and memory cost, image quality and frame rate. The main purpose of stated experiments is to find the suitable cells dimension and the number of eigenvectors should be used in different kinds of requirements.

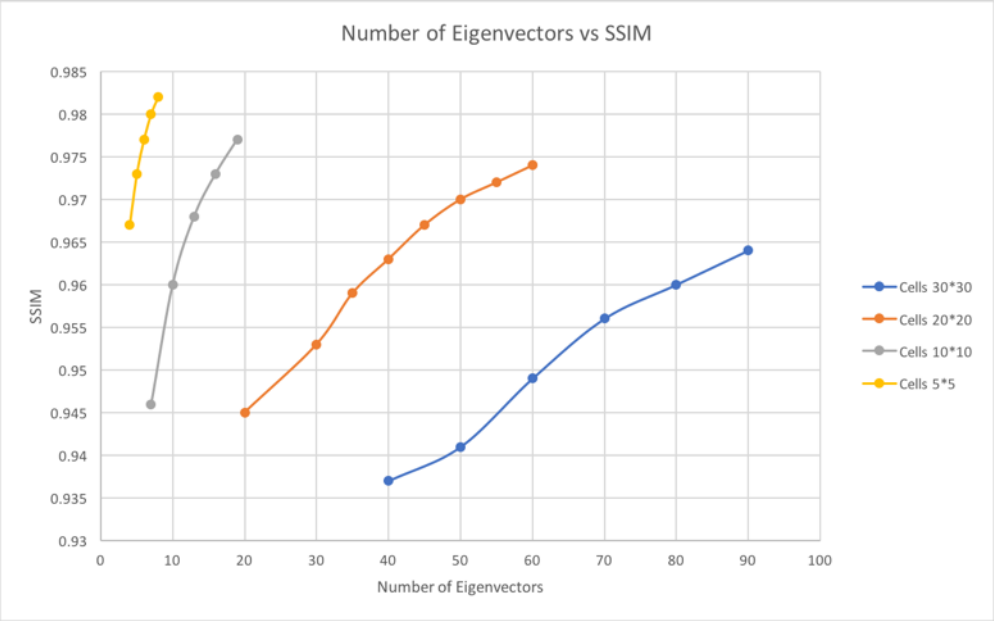


Figure 4.17: The relationship between number of eigenvectors used and SSIM value.

According to figure 4.17, we can conclude that the larger cells dimension will result in a lower efficiency for eigenvectors. The reason is, for a smaller cells dimension, the differences inside one cell can be represented by less eigenvectors and it is a lot easier to reconstruct image with less eigenvectors. This is one of the contribution in paper [2]. Alakkari compared the whole image reconstruction with 20×20 cell-based reconstruction. The latter method saved much more memory cost to get the same image quality. In my experiments, different cells dimensions are tested from small to big, where the smallest cells dimension is 5×5. For 5×5 cells dimension, with only 4 eigenvectors we could get a reasonable result, which do save enormous memory for the device. It indi-

cates that when we are determining cells dimension, maybe we should make it as small as we can. In the case of when observed unit is 1×1 resolution, we can possibly use only one highest eigenvector with the corresponding projected scores to reconstruct the whole original image without losing many details. However, the frame rate is not shown in this chart. The chart below would show a disadvantage of smaller cells dimension that a longer reconstruction time might be required because of more loop times when using small cells dimension.

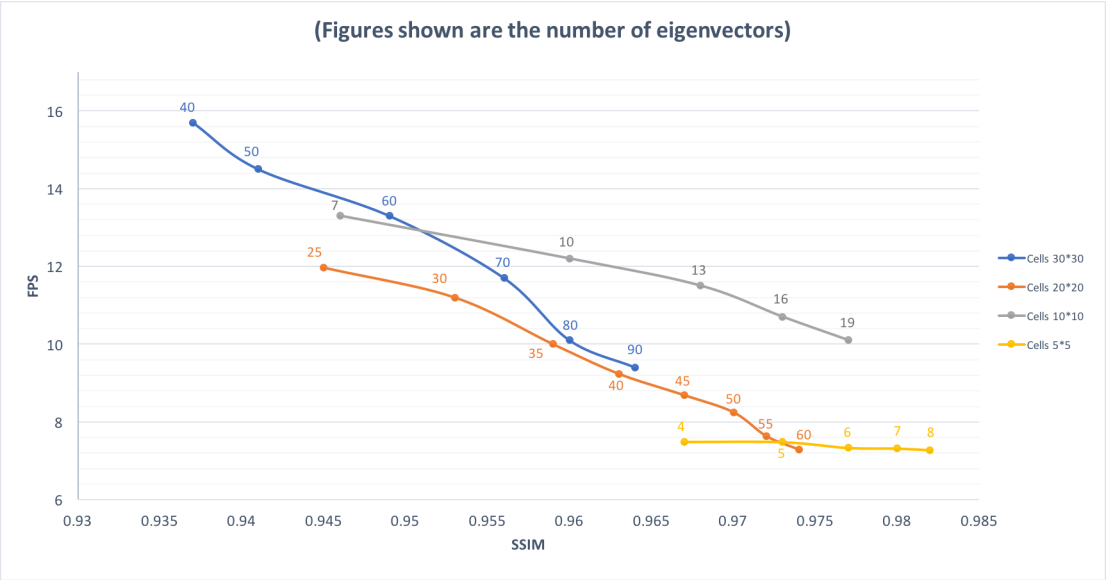


Figure 4.18: The relationship between SSIM value and frame per second (720*720 resolutions). The corresponding used eigenvectors are labeled upon lines.

The main observed variables in figure 4.18 are frame per second (FPS) and SSIM. All lines' slopes are negative, which means the higher image quality we want, the lower FPS we will get. Apparently, it is because when the number of eigenvectors are larger, the result would be more similar to the original image. But meanwhile, it will cost more time to reconstruct.

From the slopes, we can see that if the slope is more steep, cells dimension is larger. When cells dimension is very small, such as 5*5 in the chart, even though the image

quality would raise with increase in the number of eigenvectors, there will be only a tiny change in FPS. If we don't care about the memory cost, when the SSIM value is lower than around 0.95, with the same SSIM value, larger cells dimension could lead to higher FPS, when the SSIM value is higher than around 0.95, with the same SSIM value, smaller cells dimension, such as 10×10 could lead to higher FPS.

As we discussed in figure 4.14, using smaller cells dimensions with smaller number of eigenvectors needed, could get around the same image quality compared to using larger cells dimensions. In figure 4.17, FPS could be over 12 when cells dimension is larger than 10×10 whilst number of eigenvectors is relative low. However, in 5×5 cells dimension, even though the number of eigenvectors is very low, FPS is only around 8. This is because each cell reconstruction is inside the whole loop and loop time is decided by number of cells.

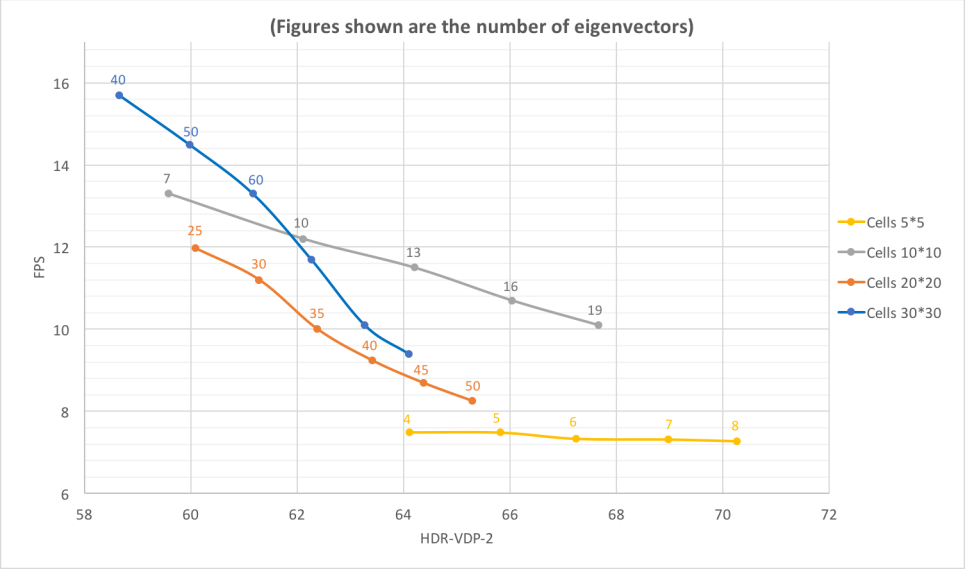


Figure 4.19: The relationship between HDR-VDP-2 quality value and frame per second (720*720 resolutions). The corresponding used eigenvectors are labeled upon lines.

Figure 4.19 shows another evaluation method to quality the result images. Compared to figure 4.18, the general relationship between image quality and FPS is similar. However, because of more convincing performance for HDR-VDP-2 metric, we could reference more

quality threshold from figure 4.19. In general, some people proposed that quality value larger than 60 is acceptable for human eyes. So, we should use the corresponding number of eigenvectors when HDR-VDP-2 value is larger than 60, to make sure images performance.

To conclude, in general for resolution 720×720 data set, if users want to reduce memory cost, cells dimension should be lower than 5×5, and give over FPS. If users are more willing to reach a high FPS, for approximately real-time interaction with these images, cells dimension should be around 10×10, which can give a high FPS up to 13 and at meantime memory cost could also be saved (10 eigenvectors in devices). The disadvantage is relatively lower image quality. But on the other hand, image quality may be affected.

4.4.2 Different resolution data set

The data set used in section 4.4.1 is all based on 720×720 resolutions. In this section, the down-sampled data set with 540×540 resolutions and enlarge-sampled data set with 1080×1080 resolutions were used in experiments to test the performance of our method.

In figure 4.20, the relationship between SSIM value and FPS is nearly the same as figure 4.18. More number eigenvectors led to higher images quality, corresponding with lower FPS. The slope for 55 cells dimension is close to zero. 30×30 cells dimension has the highest FPS value. Using around 10 number of eigenvectors in 10×10 cells dimension is still the best choice to take frame rate, images quality and memory cost into account. The difference for the smaller data sets is that the whole value of FPS is relatively higher than previous larger data set. Even though the FPS of 55 cells dimension didn't change too much when the number of eigenvectors decreased, around 13 FPS is a better result than around 7 FPS in 720×720 resolutions data set. For 10×10 cells dimension, the value

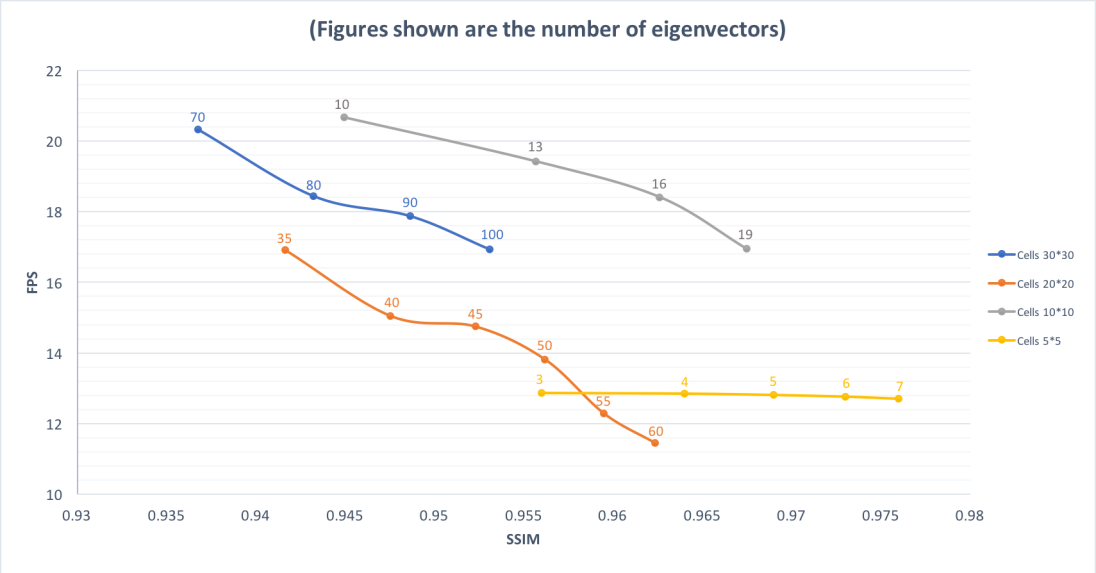


Figure 4.20: The relationship between SSIM value and frame per second (540*540 resolutions). The corresponding used eigenvectors are labeled upon lines.

of FPS could even reach 21, which is enough for an excellent performance in real-time application with good images quality.

900 images of the head data set were used in three different resolutions in Figure 4.21. The first resolution was 1080×1080 pixels. The second one was three quarters size width and height of the first one, 720×720 pixels. The third one was half width and height of the first one. To keep the other variables constant, all three resolutions data were cropped with 20×20 cells dimension.

If the image dimension is larger, there will be more number of cells. Each cell will then contain less information. So, for the same HDR-VDP-2 quality value, the larger resolutions data set needs smaller number of eigenvectors. In 1080×1080 resolution data set, around 18 eigenvectors could get an acceptable result quality, while 720×720 resolution data set needs around 25 eigenvectors and 540×540 resolution data set needs around 35 eigenvectors. The corresponding memory cost is around 16MB, 10MB and 8MB, which is much smaller than original volume data set. Considering the reconstruction speed, 1080×1080 resolution data set could only reach around 6 FPS for acceptable quality,

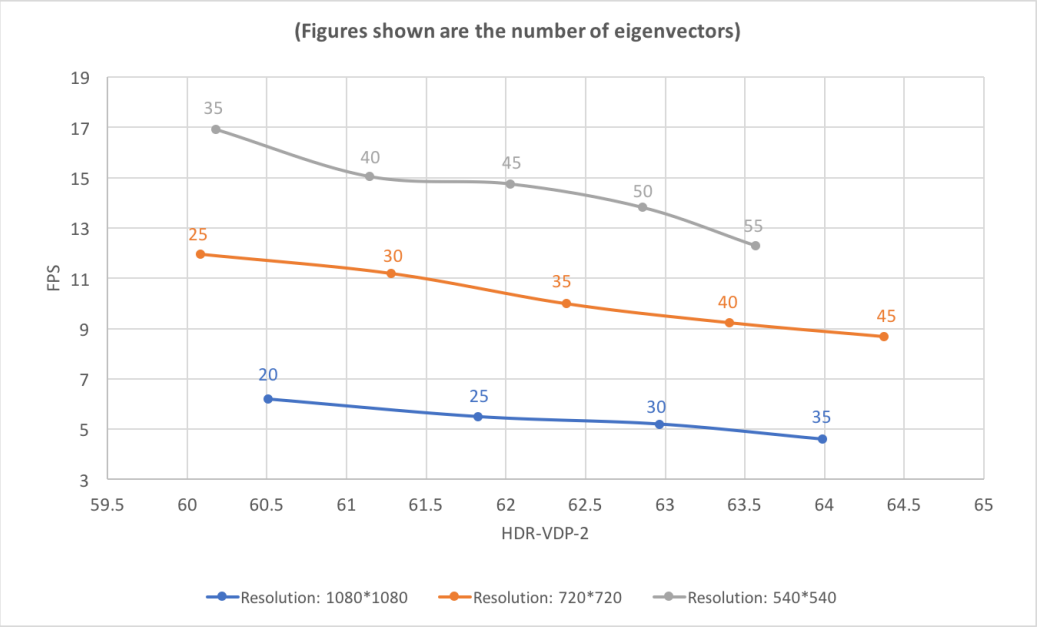


Figure 4.21: Three different kinds of resolution data set with the same content. 20*20 cells dimension are used.

while 540×540 resolution data set could reach around 12 FPS and 540×540 resolution data set could reach around 17 FPS. If users want to use high resolution data set, the delay for interaction with the scene may be extended. Otherwise, they have to reduce the resolution to achieve a real-time interaction.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This dissertation implements a PCA image-based rendering in C++, mainly for volume rendered dataset. The reconstruction process is speeded up by two kinds of GPU kernel. One is CUDA and the other one is OpenCL function in OpenCV. Several performance experiments were tested for real-world applications.

For volume rendering, instead of using direct ray casting technique, there are several techniques which can be used as alternatives to achieve similar result with much lower computing costs. The referenced paper[2] presented "Volume Visualization Using Principal Component Analysis" without exhaustive analysis for its performance and usability. The original motivation behind my project was to re-implement this technique in C++, which the technique could be more easily used in cross platform devices and in real world application. Even though the pre-processing took a lot of time for each kinds of cells dimensions, we only need to calculate PCA data and projected scores data once and save them for later use. The pre-processing was also a part of the whole work but it was not the main problem discussed in this project.

The reconstruction processing is like images processing, handling hundreds of values with the same function for each one. Improving performance of reconstruction process was implemented and tested by multithread programming–CUDA. The small dataset for solider head screen shots were used here. Matrices computing with Eigen library was used for reconstruction by CPU. Matrices computing with two kernels was used for reconstruction by GPU. Reconstruction with GPU is six times faster than reconstruction with CPU.

Considering relatively lower pre-processing time for Eigen library, I used PCA class provided in OpenCV, which is 3 times faster for calculating eigenvectors and eigenvalues than former library. Then, the reconstruction time was also reduced by using the back-project function in PCA class, which were tested by calculating with only CPU. The acceleration times is around double on average.

Artefacts between cells, presented in referenced paper, were optimized in this dissertation. A number of smoothing algorithms used in image processing were discussed and evaluated. The median blur algorithm was chosen to be the most suitable for blurring artefacts. Result images looked much better from human eyes and the image quality were improved. The trade-off of this algorithm was that it gave a lower FPS, reduced by around 0.6.

Finally, the relationship among memory costs, image qualities, and frame rate were tested for different resolution data. Users can adjust cells dimension to satisfy different requirements for different devices. Lower memory costs might lead to lower frame rate. Higher frame rate might lead to relatively lower image quality. If users want to get a result with high frame rate, high image quality and low memory cost, a lower resolution dataset around 540×540 should be considered.

5.2 Future work

The multithread programming method used here is provided by OpenCV. Even though CUDA also speeded up the process, it didn't work on large datasets. Considering features for the construction process, some other custom multithread programming technique might work better for large datasets. Moreover, datasets used here are just one image in each angle. Users cannot change colors and transparencies for each surface of the volume data to look inside, which is the main advantage for volume data visualization. Tikhonovas [36] multi-layer images technique might be combined into our PCA image-based technique to show inner information of the volume data. Instead of reconstructing one image in one angle, we could reconstruct multiple images as multiple layers for the preparation of [36], saving more memory costs and computing processes.

In my final application, users could rotate the whole head model vertically and horizontally by keyboard. However, result scene is limited by the input data set, which means users cannot see other views outside the input data set. In Alakkari and Dingliana's paper [2], they acquired unique views by applying more number of views after calculating eigenspacese. To render new views, we can combine Light Field Rendering technique with our reconstructed images. By combing and resampling the reconstructed images, we could obtain new views. Users can get more completed scene from arbitrary camera position without spending extra resources. Currently, all results qualities are tested by SSIM metric and HDR-VDP-2 metric. Although they can represent human vision to qualify images, user surveys for the result can make the experiment more convincing. In addition, advantages of low memory cost and small bandwidth requirement make potential client-server visualization of volume data possible.

Appendix A

Abbreviations

Short Term	**Expanded Term**
PCA	Principle Component Analysis
CT	Computed Tomography
MRI	Magnetic Resonance Imaging
GPU	Graphics Processing Unit
CPU	Central Processing Unit
OpenCV	Open Source Computer Vision Library
OpenCL	Open Computing Language
FPS	Frame Per Second
SSIM	Structural Similarity

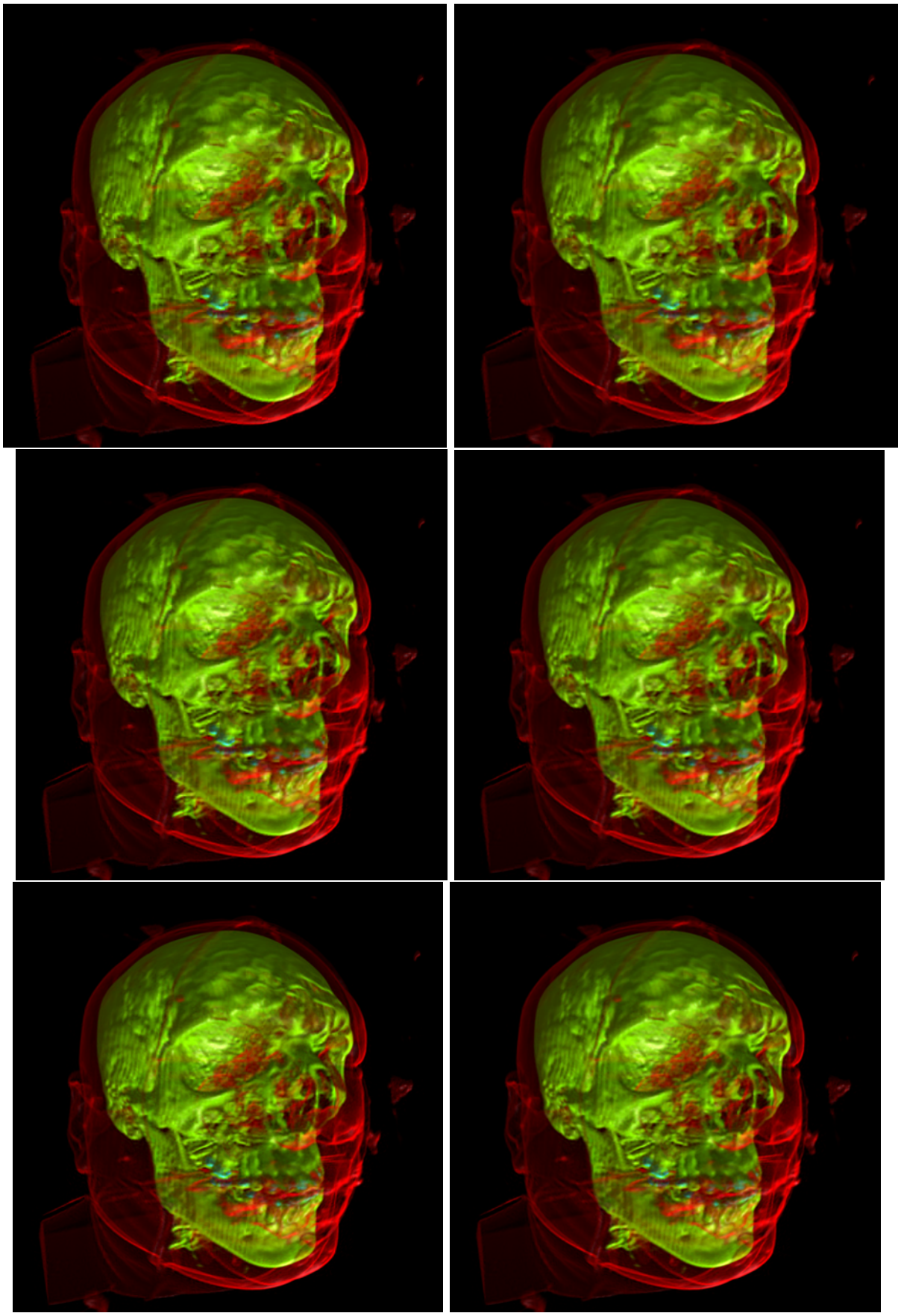


Figure A.1: These three pair of images are one reconstructed result of the dataset. The dataset used here is 720*720 head images. The first row uses 10*10 cells dimension. The second row uses 20*20 cells dimension. The third row uses 30*30 cells dimension. The left column is unsmoothed images, the right column is smoothed with kernel size 3.

Bibliography

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

[2] Alakkari, S. and 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.

[3] Arias-Castro, E. and Donoho, D.L., 2009. Does median filtering truly preserve edges better than linear filtering?. The Annals of Statistics, pp.1172-1206.

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

[5] Commons.wikimedia.org. (n.d.). File:CTSkullImage.png - Wikimedia Commons. URL https://commons.wikimedia.org/wiki/File:CTSkullImage.png

[6] Davis, A. (2006). Dynamic 2D Imposters: A Simple, Efficient DirectX 9 Implementation. Gamasutra.com. URL http://www.gamasutra.com/view/feature/130911/dynamic_2d_imposters_a_simple_.php.

[7] Docs.opencv.org. (n.d.). OpenCV: cv::PCA Class Reference. URL http://docs.opencv.org/trunk/d3/d8d/classcv_1_1PCA.html.

[8] Docs.opencv.org. (n.d.). Smoothing Images OpenCV 2.4.13.3 documentation. URL http://docs.opencv.org/2.4/doc/tutorials/imgproc/gausian_

median_blur_bilateral_filter/gausian_median_blur_bilateral_filter.html.

[9] Eigen.tuxfamily.org. (n.d.). Eigen. URL http://eigen.tuxfamily.org/index.php?title=Main_Page.

[10] Engel, K., Hadwiger, M., Kniss, J., Rezk-Salama, C. and Weiskopf, D., 2006. Real-time volume graphics. CRC Press.

[11] Frank, S. and 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.

[12] Free3d.com. (2012). Special forces Soldier - 3d model - .3ds, .obj, .sldprt. URL https://free3d.com/3d-model/chris-15987.html.

[13] Gaster, B., Howes, L., Kaeli, D.R., Mistry, P. and Schaa, D., 2012. Heterogeneous Computing with OpenCL: Revised OpenCL 1. Newnes.

[14] Gong, S., McKenna, S. and Collins, J.J., 1996, October. An investigation into face pose distributions. In Automatic Face and Gesture Recognition, 1996., Proceedings of the Second International Conference on (pp. 265-270). IEEE.

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

[16] Hdrvdp.sourceforge.net. (n.d.). hdrvdp. URL http://hdrvdp.sourceforge.net/wiki/

[17] Lacroute, P. and Levoy, M., 1994, July. Fast volume rendering using a shear-warp factorization of the viewing transformation. In Proceedings of the 21st annual conference on Computer graphics and interactive techniques (pp. 451-458). ACM.

[18] Levoy, M., 1990. Efficient ray tracing of volume data. ACM Transactions on Graphics (TOG), 9(3), pp.245-261.

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

[20] Lonesock.net. (n.d.). lonesock.net: SOIL. URL http://www.lonesock.net/soil.html

[21] Malzbender, T., 1993. Fourier volume rendering. ACM Transactions on Graphics (TOG), 12(3), pp.233-250.

[22] Mantiuk, R., Kim, K.J., Rempel, A.G. and Heidrich, W., 2011, August. HDR-VDP-2: A calibrated visual metric for visibility and quality predictions in all luminance conditions. In ACM Transactions on Graphics (TOG) (Vol. 30, No. 4, p. 40). ACM.

[23] McMillan, L. and Bishop, G., 1995, September. Plenoptic modeling: An image-based rendering system. In Proceedings of the 22nd annual conference on Computer graphics and interactive techniques (pp. 39-46). ACM.

[24] Meyer, M., Pfister, H., Hansen, C., Johnson, C., Meyer, M., Pfister, H., Hansen, C. and Johnson, C., 2005. Image-based volume rendering with opacity light fields. no. UUSCI-2005-002. Tech Report.

[25] Mueller, K., Shareef, N., Huang, J. and Crawfis, R., 1999, October. Ibr-assisted volume rendering. In Proceedings of IEEE Visualization (Vol. 99, pp. 5-8).

[26] Munshi, A., 2009, August. The opencl specification. In Hot Chips 21 Symposium (HCS), 2009 IEEE (pp. 1-314). IEEE.

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

[28] Nishino, K., Sato, Y. and Ikeuchi, K., 1999. Eigen-texture method: Appearance compression based on 3D model. In Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on. (Vol. 1, pp. 618-624). IEEE.

[29] Nvidia.com. (n.d.). Parallel Programming and Computing Platform — CUDA — NVIDIA — NVIDIA. URL http://www.nvidia.com/object/cuda_home_new.html

[30] NVIDIA Developer. (n.d.). CUDA GPUs. URL https://developer.nvidia.com/CUDA-gpus.

[31] Opencv.org. (n.d.). OpenCL - OpenCV library. URL http://opencv.org/platforms/opencl.html.

[32] Rezk-Salama, C., Engel, K., Bauer, M., Greiner, G. and 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.

[33] Scikit-image.org. (n.d.). scikit-image: Image processing in Python scikit-image. URL http://scikit-image.org.

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

[35] Software.intel.com. (n.d.). OpenCL Drivers and Runtimes for Intel Architecture — Intel Software. URL https://software.intel.com/en-us/articles/opencl-drivers.

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

[37] Wang, Z., Bovik, A.C., Sheikh, H.R. and Simoncelli, E.P., 2004. Image quality assessment: from error visibility to structural similarity. IEEE transactions on image processing, 13(4), pp.600-612.

[38] Wilhelms, J. and Van Gelder, A., 1991. A coherent projection approach for direct volume rendering (Vol. 25, No. 4, pp. 275-284). ACM.

[39] Wilson, O., Van Gelder, A. and Wilhelms, J., 1994. Direct Volume Rendering Via 3D Terxtures. Computer Research Laboratory [University of California, Santa Cruz].