University of Alberta

A GPU-based Framework for Real-time Free Viewpoint Television

by

Kyrylo Shegeda

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Spring 2014
Edmonton, Alberta

Abstract

Thesis addresses two main problems of Free Viewpoint TV: generation of arbitrary viewpoint in real-time and its delivery to end-user. For the first problem a GPU-based algorithm capable of generating free viewpoints from a network of fixed HD video cameras was developed. We used a space-sweep algorithm to estimate depth information. The view generation sub-system uses it to render new synthetic images for an arbitrary viewpoint. We show how the computations can be divided from each other and parallelized using CUDA. The tests for publicly available Microsoft sequence are provided.

For the second problem, we proposed an open-source browser-based Free Viewpoint Video player. The player is implemented using WebGL where rendering is performed solely on a GPU. It is completely asynchronous - data retrieval from the server and rendering do not block the browser. The player fits perfectly into the presented framework and can greatly improve scalability.

Acknowledgements

First of all, I would like to thank my supervisor — Dr. Pierre Boulanger. I've learnt many things while working with you. You gave me an opportunity to work on a number of interesting projects that helped me to get to know all the sides of the research. Thank you for the fun and friendly environment that you have created in the lab. Working with you has always been a pleasure.

Dr. Harlyn Baker, thank you for all the time you've spent on giving the feedback and ideas for the direction of research.

Fabio Rocha and Peter Wood - thank you for being awesome room-mates and friends. Thanks for always being there for me, for all the fun and giggles that we had. Fabio, thanks for all the interview preparations that you did and for pushing me to do the same. Peter, thanks for your delicious pizza times that cheered up everyone in the house. Thank you for all the barbecues that we had on our backyard. I hope that our life paths will not diverge too much and we will always have time to keep in touch and spend some time together. Alejandro Ramirez, Ryan Kiros, Arnamoy Bhattacharyya, Amritpal Saini, Victor Guana, Lengdong Wu, Ailin Zhou thanks for being great friends and peers to hang out with, TA and study.

A separate word goes to a special person in my life, that I'm happy to be with — Yulia Chepurna. Your endless, unconditional love and support have always been around me, I could feel that even when we were thousands of miles apart from each other. Thanks for being patient and believing in me. You have no idea how happy I am that everything turned the way it is right now.

Denis, thanks for being such a wonderful elder brother. You've always known how to cheer me up, thank you for always caring about me and of course thank you for changing my diapers, the fact that I don't remember that part of my childhood doesn't mean that I don't appreciate it.

Mom, Dad thank you for being the best parents one can only dream about. All the time that you've spent on helping me and being around, all the knowledge and

wisdom that you've passed to me. For all the love and warmth that surrounded me all my life. Thank you for all the supporting all of my decisions and endeavours. I hope that when I get a chance to become a parent I will be able to be at least one tenth as good as you guys.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Traditional video technologies are passive and two-dimensional in nature. Viewers can only observe video images from the cameraman viewpoint. Systems capable of generating arbitrary viewpoint have been known for quite some time, but only with recent technology advances in video camera, computer vision, and graphics hardware such systems are now possible to be built. This new type of video delivery system is called Free Viewpoint Video (FVV)[38]. Using FVV users can interactively choose a virtual viewpoint generated from a network of cameras located at a remote site, possibly even creating a novel view that could not be acquired by a physical camera. By using these systems, one can create a certain sense of presence by manipulating the viewpoint. Application of FVV systems ranges from covering sports events and concerts to being used in tele-presence systems or for performing complex remote manipulations using robotics. Another application of FVV systems is the so called Free Viewpoint Television (FTV) where images are coded, transmitted, and decoded at home on a setup box connected to a standard television. In these systems, the user can select arbitrary viewpoints using a simple interface like a game pad. FVV and FTV systems usually consists of three main sub-systems:

1. Video capture and synchronization;
2. Arbitrary view generation;
3. Compression for delivering to end-user.

Before describing the nuts and bolts of a FTV and FVV system lets first review its development from an historical perspective.

1.1 A Brief History of FVV and FTV Systems

In 1995, Katayama *et al.*[22] proposed a system for autosterescopic displays which can show viewpoint dependent images according to viewer's movement. The system assumes that all the cameras are setup along one line. The novel views were calculated using epipolar-plane images. At that point the live images couldn't be processed by the system as the computational powers of computers were low compared to modern machines, but it was one of the early proofs of concept.

Probably the first real prototype of a FVV system, called *Virtualized Reality*, was developed by Takeo Kanade and Peter Rander at Carnegie Mellon University back in 1997 [21]. The system consisted of 51 cameras with a resolution of 512x512 pixels that were installed on a 5 meters geodesic dome (see Figure 1.1). The system was divided into a recording sub-system and an online post-processing sub-system. During the post-processing step, a so-called *scene description* was built using a multi-camera stereo matching technique [31]. The resulting textured triangle meshes could then be viewed at an arbitrary viewpoint using the standard OpenGL graphic pipeline. On January 28, 2001 a version of this system called *EyeVision* was demonstrated during the SuperBowl XXXV. The event was captured by more than 30 cameras located around the stadium. Each camera was installed on a computer controlled robotic pan-tilt unit. The robotic unit was equipped with sensors that were constantly measuring the relative orientation between the cameras and a master camera. The operator was responsible for changing the rotation and zoom of the master camera to focus on a region of interest (e.g. player with a ball, touchdown, fumble etc.). This information was collected and processed by the computer and the appropriate control signals were then sent to the remaining cameras so that all of them were converging towards the same region of interest. The frames captured from the system were synchronized and used to create novel views not available from the master camera.

In 2000, Matusik *et al.*[26] presented an image-based algorithm for computing and shading visual hulls from silhouettes. The algorithm was tested on four calibrated video cameras with a resolution of 256x256 pixels. The processing tasks was distributed among a quad-core server and four client machines. The rendered object was segmented by the client machines, whereas an Image-Based Visual Hull (IBVH) algorithm was executed on the server machine. Authors were able to generate new

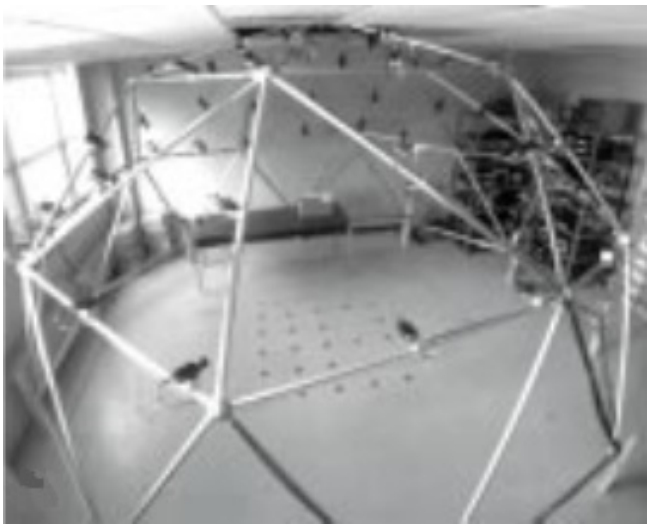

Figure 1.1: Virtualized Reality system

views at 8 frames per second for IBVH consisting of 8000 pixels.

In 2003, Carranza *et al.*[9] introduced a system that uses multi-view synchronized video to digitize actors from arbitrary viewpoints. The video was recorded from eight synchronized cameras with a resolution of 320x240. Using this system, human body motion estimation was performed off-line and a subsequent view generation was performed on a Graphic Processing Unit (GPU) in real-time.

In 2004, Zitnick *et al.*[43] presented a camera system consisting of 8 cameras located along a 1D arc spanning around 30 degrees from one end to another (see Figure 1.2). Each camera had a resolution of 1024x768 pixels. To synchronize the cameras, two concentrators were developed. These concentrators were synchronized using two Fire-Wire cables. The system allowed to capture synchronized videos with a subsequent off-line computation of scene geometry extraction and texturing. For each camera a depth map was calculated using color segmentation-based algorithm. To generate novel views at interactive rates a shader based algorithm was proposed.

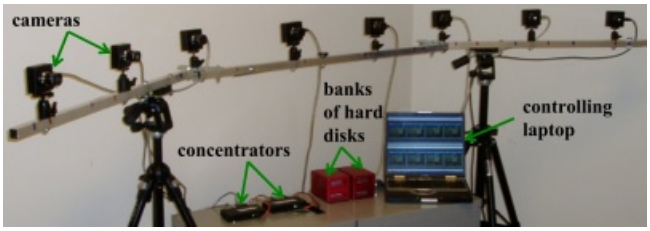

Figure 1.2: Video capture system proposed in [43]

In 2005, Baker *et al.*[7] proposed Coliseum - an immersive videoconferencing system. Using five cameras at user endpoints it was able to produce arbitrary-perspective renderings of the participant and transmit them to others at 15 frames per second rate. The system demonstrated its capabilities to support the conferencing of up to 10 participants. The paper shows an indepth analysis of how each part of the system contributes to the total latency which gives an idea how details on the software level can impact the performance.

Huge contributions to the field of the FTV was made by Tanimoto's laboratory at Nagoya University, Japan. The laboratory is dedicated to the development of all aspects of FTV systems dealing with data capture, camera synchronization, compression, transmission, and free viewpoint rendering. In 1996, T. Fujii *et al.* Tanimoto *et al.* [14] proposed a ray-space method for the representation of multi-camera rays in 3D space. In 2003, Tanimoto's laboratory constructed a FTV system consisting of an array of 16 cameras connected to 16 computers under the command of a single server [28]. In 2006, this setup was further expanded to capture larger camera networks with 100 cameras [15](see Figure 1.3(a)). The ray-space method was widely popularized by Tanimoto's work as a way for representing the scene in FTV and as a way to generate arbitrary views. Moreover in 2007, Manoh et al. [25] developed a 360 degree mirror scan ray-capturing system (see Figure 1.3(b)). Ray-space methods are based on representation of the way the rays are travelling from the scene to the camera sensors. Hence, a dense camera array is needed for such a representation. But since the number of cameras that can be located around the scene is limited by the size of the cameras, algorithms for ray interpolation are needed. Many algorithms for different camera configurations were proposed. They can be roughly divided into two groups: algorithms for linear camera arrangement (the orthogonal ray-space) and algorithms for circular camera arrangement (the spherical ray-space).

In 2010, KDDI Japan developed the first commercial FTV system based of 4K camera technology (http://www.kddi.com). Little is known about how this technology works but in many ways it was truly one of the first realistic demos of a FTV technology.

In Summer 2012 during the London Olympic games, Replay Technologies (http://replay-technologies.com/) demonstrated a system capable of creating free viewpoint replays of athlete performances using a network of 16 4K cameras . This system was ca-

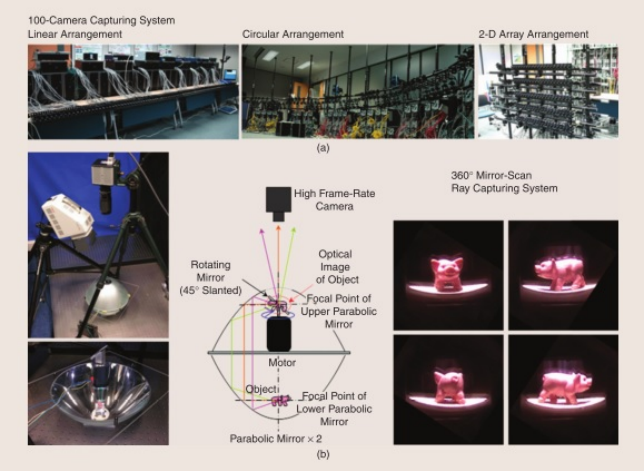

Figure 1.3: (a) 100 camera capture system, (b) ray-capturing system based on parabolic mirrors

pable of generating 360 degree virtual view of an athlete performance based on a minimum of 4 camera viewpoints.

1.2 Motivations

In order to develop a FTV system a series of challenges have to be resolved:

1. The captured video has to be synchronized, so that all of the frames are capturing the same picture at the pixel level;

2. An algorithm for arbitrary view generation from a limited set of pictures has to be developed;

3. The information needed for view generation has to be effectively encoded in order to be sent to the end users. Considering the fact that the amount of information coming from the cameras is enormous this is really a tough challenge to tackle;

4. Ideally the system has to be able to serve a large population of viewers at the same time;

These problems have been studied thoroughly during the last decade. But yet there's no complete systems implemented. Moreover, performance studies of real-time HD FTV systems are actually quite rare and to date there are no real commercial systems that can deliver a real FTV experience.

The goal of this thesis is to address some of those challenges. How to build a HD FTV system that works in real-time starting from capturing the multiple videos at pixel precision to users selecting a view of interest at thousands of different locations.

The thesis is organized as following. In Chapter 2, we review previous works on arbitrary view generation, FTV/FVV system architectures, and on the delivery of FTV services to end-users.

In the Chapter 3, a real-time processing pipeline (Figure 1.4) based on plane-sweep algorithm using GPU is presented. The Chapter also introduces a discussion on the efficient implementation of the algorithm using the CUDA language. The algorithm is compared to other algorithms using standard image sequences from Microsoft. An example of the result generated by the proposed system can be viewed in Figure 1.5

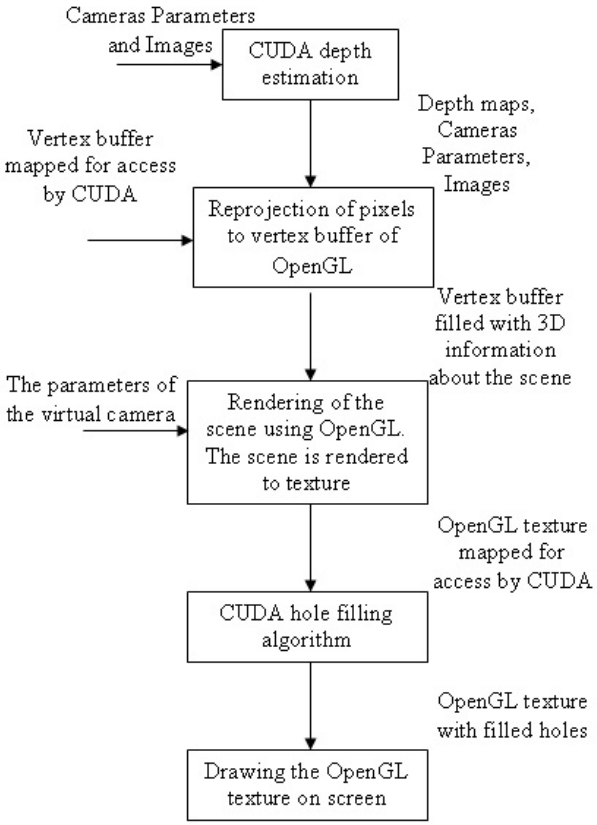

Figure 1.4: The pipeline of the proposed system

Chapter 4 describes a novel WebGL based FVV player that runs natively on standard web browsers and provides an efficient way to generate novel views of a scene using reconstructed geometry and texture. The typical GUI and an arbitrary

Figure 1.5: Virtual image generated from two real cameras

view rendered for a scene covered by 5 Kinect cameras can be seen in the Figure 1.6. It is shown how this player can be incorporated with existing scene reconstruction algorithms to achieve a high scalability for the FVV/FTV applications with many end-users.

In Chapter 5, I conclude by presenting the pros and cons of the proposed algorithms and the future directions of FVV/FTV systems.

1.3 Thesis Contributions

This thesis addresses two main problems: generation of arbitrary view in real-time and the problem of the efficient delivery of free viewpoints to multiple end-users.

For the first challenge, the goal was to develop a system that can generate arbitrary views from a network of cameras with known extrinsic and intrinsic parameters. Since the process of view generation is always computationally expensive but in essence parallel, general purpose GPUs are used to achieve real-time performance. For this aspect of the thesis, two main contributions must be noted:

- Adaptation of existing stereo matching algorithms for view generation that can efficiently exploit the parallelism of a GPU;

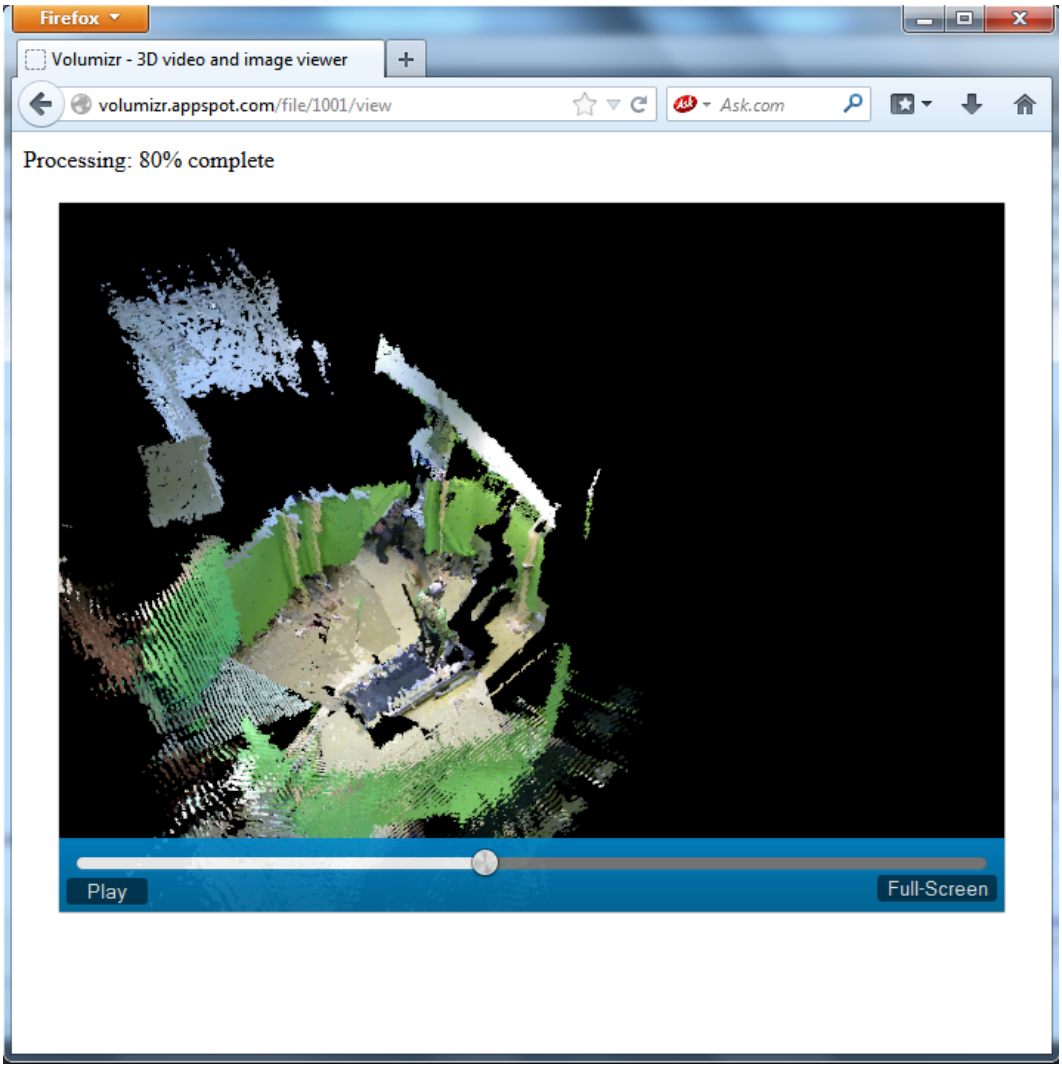

Figure 1.6: The view of the proposed player in the browser

- A deep analysis of the factors impacting the performance of parallel algorithms implemented in CUDA, as well as the importance of implementation details when writing a code on a GPU, especially for real-time applications. This is something that is usually not mentioned in similar works, as technical details are often considered irrelevant when presenting research.

In the modern world everything tends to be oriented towards the web. So for the second part of the thesis, a new FTV delivery system based on WebGL was implemented. This web-based FVV player uses the GPU of the client for rendering novel views and doesn't require any third-party software to be installed. The fact that the rendering is performed on the client side allows the efficient delivery of a service to many users as the server is responsible for just providing pictures and

depth maps to the user. This allows the FVV player to be used in a broad variety of applications: FVV replay of the events, real-time coverage of the events, a web-interface for 3D tele-conferencing, as well as an easy way to turn a Kinect 3D sensor into a cheap 3D webcam.

We have also published three papers in international conferences:

1. "A GPU-based Real-time Algorithm for Virtual Viewpoint Rendering from Multi-video", IEEE International Symposium on Multimedia (ISM2013).

2. "Web browser-based Free Viewpoint Video player", IEEE International Symposium on Multimedia (ISM2013).

3. "A GPU-based Real-time Algorithm for Virtual Viewpoint Rendering from Multi-video" (extended version), 2013 Symposium on GPU Computing and Applications.

Chapter 2

Literature Review

2.1 Arbitrary View Generation

Numerous systems geared towards solving the challenges of arbitrary view generation can be found in the scientific literature. In this chapter, we will review some of them.

2.1.1 Classification of the Methods with Respect to View Generation Approaches

In terms of approaches of solving the problem of free viewpont generation most of the systems can roughly be classified into two main categories: Model-Based Rendering (MBR) and Image-Based Rendering (IBR) techniques.

Model-Based Rendering Approaches

The first category are methods that estimate the geometry of the scene by solving the correspondence problem and then use this information to generate new views using simple computer graphics techniques. Probably the most popular methods in this category are shape-from-silhouette approaches where for each cameras the object that is being modelled is first extracted from its background using segmentation method and then projected into 3D space to reconstruct the 3D shape of the object. Examples of model-based systems include shape-from-silhouette techniques where the scene is represented by a visual hull ([26, 13]), global multi-view stereo reconstruction techniques which use both photo-consistency and some additional information such as silhouette constraints or shape priors [35], surface growing approaches that perform reconstruction from a set of reliable seed points and then a surface growing algorithm to reconstruct a 3D map from the seed-points [16], view-

dependent multi-view stereo reconstruction which obtains a separate reconstruction of a scene for each of the cameras and then merge them together [34]. These types of methods have two main disadvantages: the computational cost of scene reconstruction is huge and the virtual images produced are usually not photo-realistic. In addition, all of the methods that are based on the shape-from-silhouette approaches require a prior foreground extraction which means that only a part of the scene will be rendered in 3D (e.g. background is going to be omitted) and that the quality of the method depends on the quality of this segmentation.

Image-Based Rendering Approaches

The second category is based on techniques where a large number of images is used to keep light ray information. Three closely related methods were presented at approximately the same time: Lumigraph [18], Light Field rendering [23], and ray space method [14]. These methods are based on describing how light rays travel through space. A good example of the quality of images generated by IBR approach is presented in Mori *et al.* [27]. In Mori's paper, the images are generated through warping a pre-computed depth maps for each camera and then post-processing (smoothing, boundary matting, and in-painting) the results to create an image without artifacts. Numerous versions of these methods were proposed and some real-time implementations were developed. Particularly, in [43] where new images are rendered in real-time. One limitation of this implementation was that it assumes that a prior off-line depth-map estimation is performed beforehand. In [41] plane sweeping algorithm together with shader programming techniques are used to create an algorithm that can process in real-time images of a size 320x240. The disadvantage of this method is that it is not able to handle occlusions properly. In [37], a new algorithm is proposed to generate new views using ray-interpolation in parallel using 16 "clients" processors under the command of one server machine. The system is able to generate new views at 16 fps rate for images of 640x480 pixels with a 4 cm baseline. Do *et. al* [12] has proposed to use GPUs to speedup the process of depth image based rendering. Authors have used CUDA to project the points from neighbouring cameras of a virtual view and then used standard graphics API to postprocess the warped images (filling cracks and performing blending to reduce ghosting artifacts) going back to CUDA for final in-painting regions that are not visible by any of the neighbouring cameras. The tested algorithm was able to

achieve real-time performance on Microsoft data set ([43]), but once again the depth map for this algorithm is assumed to be precalculated.

2.1.2 Classification of the Algorithms with Respect to Speed

In terms of processing speed one can divide algorithms into off-line and on-line methods.

Off-line Methods

In off-line methods a part of computation or all of the computations needed for view generation are done on a pre-recorded videos and thus are not applicable for real-time FTV applications. The first off-line method was proposed by Kanade *et. al* [21]. Due to the volume of a data the arbitrary viewpoint was chosen by the camera operator and computed off-line to be presented in a form of replay video delivered as a conventional 2D video to the end-user.

The visual hull based method described in [13] is composed of 3 main steps: first a geometric approximation of the visual hull is computed by retrieving its visible edges, second local orientation and connectivity rules are used to build the surface that defines the 3D object boundaries and third a final post-processing is used to identify the planar contours of the polyhedron that represents the object. The authors of this paper claimed that for four cameras with a resolution of 640x480 pixels their algorithm requires 142 msec to reconstruct a 3D shape.

Approach proposed by Zitnick *et. al* [43] can also be considered an off-line method since the computation of the depth maps for each of the cameras is done before the rendering begins. They use color-based segmentation stereo algorithm in order to generate high-quality photo-consistent correspondences across all cameras. In this method, depth discontinuities are also extracted in order to reduce the artifacts during the rendering process. The rendering of the novel view for the already computed depth maps and texture maps is done in real-time with the help of GPU. Goldlucke *et. al* has proposed a similar approach in [17] but ignoring depth discontinues.

An interesting approach for FVV rendering of a scene was proposed by Starck *et. al* in [36]. In their paper, the proposed algorithm is capable of producing novel view in real-time. But it cannot be considered to be an online method since it assumes that input consists of camera images, camera calibration parameters (which are the

usual input for any of the FVV algorithms) and a pre-computed 3D proxy of the scene geometry which is usually done by scene reconstruction algorithms. Their technique for novel view generation consists of the following steps:

1. Identify a subset of cameras closest to the virtual one;
2. Compute surface visibility in each camera to prevent color sampling in the presence of occlusion;
3. Derive a blend field based on the proximity to the virtual view camera. This step is performed to identify a relative contribution for a camera at each pixel that is not occluded;
4. Generate a virtual view based on the blend field.

All of the steps were implemented on a GPU using GLSL. The approach was tested on the data set consisting of eight cameras with a resolution of 1920x1080 for which the camera calibration and scene geometry were pre-calculated. The rendered virtual views were performed at a resolution of 1024x768 pixels. Using NVIDIA Quadro FX1700, the authors were able to achieve 19 fps. The implementation code together with demo data were released as an open source project.

In [27] the proposed method for view interpolation is similar to the one proposed by Zitnick *et. al* and Goldlucke *et. al* in the sense that the authors were assuming that the depth maps for the real cameras have been pre-computed. Then authors proposed a method for virtual view generation that is capable of handling artifacts created by depth discontinuities. When rendering a virtual viewpoint depth maps of the two nearest cameras are projected to a virtual view, then the projected depth maps are post-filtered by smoothing the images with a bilateral filter. The maps are then back-projected to real cameras and the textures are projected to a virtual plane using the new depth maps. A boundary matting is performed on the resulting image together with in-painting to fill the blank area.

The method proposed by Furukawa and Ponce in [16], despite being able to produce high quality pictures, requires too much time for computation to be tractable. Using the Middlebury data sets as a reference, the approach proposed by these authors is one of the best ones in terms of reconstruction quality when compared to the ground truth, but the processing time ranges between 20 minutes per picture to up to a few hours.

On-line Methods

In on-line methods the reconstruction and view generation is done in real-time. Thus making it possible to be use in real FTV applications. In case of on-line methods, the price for the speed is accuracy. One cannot expect the methods that are tightly restricted by the time limits to generate images of the same quality as methods that can do all the processing without any time limits. Due to the computationally expensive nature of the FVV most algorithms use parallel programming environments. With advances of GPUs and the development of general purpose GPU languages such as CUDA, numerous on-line implementations have exploited the power of vector graphics processors architecture. The speed-up factor after porting an algorithm from CPU to GPU can be really significant. Orman *et. al* [32] specifies that for their "Cinematized Reality" system they were able to achieve a 100x speedup by using custom shaders during the reconstruction and rendering process using a shape-from-silhouette method.

Some of the visual hull approaches were implemented to run in real-time. Those are usually the methods that use volumetric approach. Unlike the methods that use meshes, volumetric representation allows to efficiently parallelize the computations. The first paper using this method was presented in [26]. The approach takes advantage of epipolar geometry and incremental computation to achieve a constant rendering cost per pixel. For the experiments, the authors designed a system consisting of 4 client PCs that were acquiring and segmenting the images from 256x256 images and a quad-core server with 550 MHz core that was responsible for the computations of the virtual view. The visual hull computations were interleaved between the cores to speed-up the generation of a novel view. For the virtual views a frame rate of 8 fps was achieved for an image of 640x480 pixels. Li *et. al* [24] have proposed an algorithm that utilizes the capabilities of the GPU to accelerate visual hull and generate arbitrary views at 80 fps. The reconstruction and rendering of the visual hull representation was incorporated in a single routine so there would be no latency associated to data transfer. While the acquisition and silhouette extraction for the experimental evaluation of the approach was performed using 4 PCs (one for each 320x240 camera), the rendering was performed on one PC with NVIDIA GeForce3 graphics card. One of the most recent papers that presents a novel algorithm for real-time arbitrary view generation with visual hulls was presented by Waizenegger

et. al [39]. The proposed method starts with the extraction of a silhouette using a marching square algorithm. Then, they propose a novel technique that allows to cache the line segments of the polygonal representations of the silhouette borders. This cache is used to minimize the intersection tests with epi-lines originated by pixels of the virtual view. The 3D intervals are computed for each pixel of the virtual view. Finally the depth map of the voxelized volume is generated. The algorithm is implemented on a GPU using CUDA.

Image based rendering techniques are usually easy to parallelize, which together with its capabilities to create more photo-realistic images and render all of the scene made it one of the most popular approaches for real-time FVV/FTV.

One of the ways to achieve a parallel computation is to use a network of computers that conduct computations simultaneously. Particularly, this approach was utilized by Yang *et. al* in [40] where the information from a 64-camera acquisition system was processed by several computers organized in client-server scheme using a distributed light field rendering technique. The client computers are transferring the data related to the novel view only. While this allows lower bandwidth requirements for the data being transferred and create a novel view in real-time, the drawback is that the system is not really scalable when multiple new views need to be rendered. Suzuki *et. al* [37] was able to show interactive frame rate for ray-interpolation method with 16 computers. But their system was only able to render views from a 1D array camera arrangement. Since the GPUs are becoming more and more powerful, it is cheaper and more efficient to use one graphic card for calculations than several computers.

Yang *et. al* [41] has adopted the plane-sweep approach to work on a GPU using shaders allowing real-time rendering for 320x240 pixels cameras. Nozick and Saito [29] proposed an adaptation of a plane-sweep algorithm. The main contribution to a conventional plane-sweep approach is the description of a new scoring method. The scoring goes as follows. For every plane D_i from the furthest one to the nearest one:

1. Each of the points p of the plane D_i is projected on every input image;
2. Variance of the color $score_p$ as well as the average color $color_p$ is calculated;
3. The point is projected onto the virtual camera. If the projected score is better than the current one then the score and color of the corresponding pixels are

updated.

In this implementation they use four of the neighbouring cameras to speed up the computation process. OpenGL shaders are used to speed up the computation process. The implemented algorithm was able to generate novel views at 15 fps rate for 4 web-cams with a 320x240 pixels resolution on duo core 1.6GHz laptop with a NVIDIA GeForce 7400 TC. The number of planes used was equal to 60. The authors made one more step forward and proceeded with a way to extend the algorithm for generation of k virtual views instead of just one simultaneously. During the score projection step each of the scores is projected on all of the virtual images at the same time. This allows the method to be efficiently scaled with the number of users, something that is frequently not considered by many authors.

Since in this research we wanted to focus on the on-line system, we will never be able to achieve the same quality as the offline methods. That led us to making an assumption that some artifacts that distort small details of the picture are acceptable as long as people can understand what is going on in the picture. We also wanted to show that by carefully designing the algorithm and employing some new algorithms on a GPU it is now possible to generate new views not only for standard videos (640 x 480 which is the resolution of the testing videos that was popular in the FVV papers), but also for HD (1900 x 1200) as well.

The reviewed literature has also directed us to take a deeper look at the problem of the system scaling. All recent systems either don't take this factor into account or propose a scheme that is parallelized by employing more computational power (e.g. computers). Such an approach would be a dead end for systems that deliver FTV to millions of people as it would be really expensive and impractical. That is why we propose to use the computational power that is hidden from the client's computer as additional help during the rendering step.

2.1.3 Client-side View Rendering

The generation of a novel view is one of the most important parts of any FVV/FTV system. One of the possible ways is to generate a novel views on the "server" side, i.e. the side where the cameras are located, but this makes it hard or even sometimes impossible to scale with the growth of the number of consumers as it generates a huge computational load on the server. This is the approach taken by most of the papers described in the previous Section 2.1. A different approach is to perform

the view generation partially on the client-side. This can be achieved for example by transferring the calculated scene geometry computed on the server in a form of point clouds, depth maps or silhouettes to a client with subsequently generate new views from the transferred data. Such an approach is better when scaled for a large number of users as it allows to efficiently deliver the service, since the main tasks of the server is to acquire the images from the cameras, calculate the geometry, and ship the point-cloud to all of the connected clients.

With the development of the World Wide Web all of the services tend to be delivered in the form of web application. The reason for this is convenience as consumers do not have to install anything - all they have to do is to access a website. So many of the upcoming FVV/FTV systems will have to be delivered in the form of new web browser which will be responsible for 3D rendering the new views.

Sketchfab[5] and Verold[6] have presented really interesting platforms for visualization of 3D data which support some of the most popular data formats. The former one is built as a social network platform where users can share their 3D models. The second system is a tool for 3D artists where people can collaborate on their models.

A couple of FVV players have also been presented in the research literature. Viewpoint Corporation has developed Viewpoint Media Player that can be used to display interactive 3D models. The biggest disadvantage is that it is shipped as a browser plug-in that has to be installed. Voxelogram (http://voxelogram.com/) has presented Voxelo a desktop based FVV player for FVV file format (a custom file format proposed by Voxelogram). The main advantage of the player is that it is completely open-source, the disadvantage is that it is a desktop application that runs on Windows machines only.

In [8] Chris Budd, Oliver Grau and Peter Schubel have proposed a first WebGL-based FVV player. The authors assumed that the reconstruction of the scene geometry was performed using the shape-from-silhouette method and the scene is represented as a triangular mesh. The data transferred to the client from the server are classified into two categories: static and dynamic. The static content consists of the JavaScript code and static background textures and is delivered through the XMLHttpRequest. While the dynamic content is delivered via WebSockets and consists of 3D mesh data and texture images for each of the original cameras. For a system with 7 HD cameras, the total amount of data for one frame transmitted to

a client is equal to 14.4 MB/s with the size of mesh object equal to around 300 KB. Authors have encoded the transferred data in Base64 which introduced an extra overhead.

The system by Budd *et al.* in [8] has inspired us to develop a WebGL-based FVV player that allows to efficiently solve the scaling problem in FVV/FTV algorithms. We wanted to create a player that is free and accessible for everyone and that could easily be integrated into the view-generation part of the FTV frameworks. With this ideas in mind, we wanted to create a player that works with depth maps obtained from cameras instead of a triangular mesh (which is currently impossible to efficiently obtain in real-time).

Chapter 3

Multi-camera Interpolation Algorithm[1]

3.1 Chapter Overview

This chapter describes an implementation of a real-time arbitrary view generation algorithm. As mentioned earlier in Chapter 1.2, the goal of the thesis was to develop a system that could generate arbitrary views from a network of cameras with known extrinsic and intrinsic parameters. Since the process of view generation is always computationally expensive but in essence parallel, a general purpose GPU implementation was developed to achieve real-time performance. In the next sections, we will describe the proposed algorithm and the peculiarities of its implementation on a GPU. We will then present experimental results using standard image sequences from Microsoft and then conclude on the pros and cons of the proposed algorithms, as well as how the approach scales for multiple users.

3.2 Common Plane Sweeping Algorithm

The proposed algorithm is based on a relatively old idea presented by Collins [11] called the *common plane sweeping algorithm*. Originally the algorithm was proposed to work as a way to match features that were obtained from images from multiple viewpoints. The general idea of the method is to discretize the space in front of the camera using planes parallel to the plane of the camera and then project each of the features on all depth planes calculating the number of features from different cameras which fall within a region of a specific size.

[1] A version of this chapter has been accepted for publication at IEEE International Symposium on Multimedia.

This algorithm can be adapted to work in real-time for estimating depth information from a set of synchronized HD cameras. Let's assume that we have synchronized color images $\mathbf{I}_1$, $\mathbf{I}_2$,..., $\mathbf{I}_N$ obtained from cameras 1,..., N with their corresponding intrinsic calibration projective matrices $\mathbf{A}_1$, $\mathbf{A}_2$,..., $\mathbf{A}_N$ and extrinsic parameters defined by the rotation matrices $\mathbf{R}_1$, $\mathbf{R}_2$,..., $\mathbf{R}_N$ and translation matrices $\mathbf{T}_1$, $\mathbf{T}_2$,..., $\mathbf{T}_N$. The algorithm consists of two independent steps:

1. Depth map estimation from the cameras;
2. Virtual viewpoint rendering $\mathbf{I}_V$ from camera with intrinsic and extrinsic matrices $\mathbf{A}_V$, $\mathbf{R}_V$, $\mathbf{T}_V$.

3.3 Depth Map Estimation Algorithm

Let's assume that one wants to estimate the depth map for camera j. If one takes only the image of neighbouring camera $\mathbf{I}_{j+1}$ and try to estimate the j's depth map D_j there will be conditions where some of the points will be occluded. To resolve this problem, we propose to look at both of the neighbouring cameras $\mathbf{I}_{j-1}$ and $\mathbf{I}_{j+1}$ to estimate the depth map D_j for $\mathbf{I}_j$. Of course it is possible that some of the points in $\mathbf{I}_j$ will be occluded in both $\mathbf{I}_{j-1}$ and $\mathbf{I}_{j+1}$ but one can assume that in most cases this will not happen if the cameras base-line is small enough.

Let's sweep a single plane through space along the Z axis which is perpendicular to the camera sensor plane. As a result, the plane always has equation $Z = z_i, i = 1, .., k$ where k is a specified number of depths levels that one wants to sweep through. The algorithm to compute the depth map from the two cameras is described in Figure 3.1.

The result of this algorithm is D_j - the matrix of depth values for each of the pixels of j-th camera. Note that the first and second steps of the algorithm can be done due to the fact that the intrinsic and extrinsic parameters of all the cameras are known. Using the calibration information, one can easily calculate the camera projection matrix for any camera j using $\mathbf{C}_j = \mathbf{A}_j[\mathbf{R}_j; \mathbf{T}_j]$ equation which links a point in the 3D world coordinates to its counterpart in the sensor 2D coordinates. One can project a point (u_i, v_i) from virtual camera i to its 3D coordinates value with $Z = z_i$ where the x_i and y_i coordinates are determined by a simple line to plane intersection algorithm. Then one can re-project the 3D point back to camera j with 2D coordinates (u_j, v_j) using:

Algorithm

1. Cast the rays from the center of the j-th camera through every pixel of image $\mathbf{I}_j$ and record the intersection with the plane z_i, i.e. the ray that goes through pixel (u_j, v_j) intersects the plane z_i in the point with coordinates (x_i, y_i);
2. Re-project the point (x_i, y_i, z_i) back to the image planes of the neighbouring cameras $\mathbf{I}_{j-1}$ and $\mathbf{I}_{j+1}$, with coordinates (u^i_{j-1}, v^i_{j-1}) as being the coordinates in an image plane of camera $j-1$ and (u^i_{j+1}, v^i_{j+1}) of camera $j+1$ (see Figure 3.2);
3. Quantify the similarity between pixels $\mathbf{I}_j(u_j, v_j)$ and $\mathbf{I}_{j-1}(u^i_{j-1}, v^i_{j-1})$ and $\mathbf{I}_j(u_j, v_j)$ and $\mathbf{I}_{j+1}(u^i_{j+1}, v^i_{j+1})$ using some similarity function $F(\mathbf{I}_t(u_t, v_t), \mathbf{I}_s(u_s, v_s))$. Add the two values obtained from the simlarity function. This value is defined as the in-between camera consistency for the depth level z_i;
4. Repeat steps 1-3 for all the depth levels z_i where $i = 1, ..., k$;
5. Set the depth value of the pixel (u_j, v_j), $D_j(u_j, v_j)$, to be the z_i corresponding the optimal in-between camera consistency.

Figure 3.1: Depth map estimation algorithm

$$\begin{bmatrix} u_j \\ v_j \\ 1 \end{bmatrix} = \mathbf{C}_j * \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix}. \tag{3.1}$$

The depth values obtained using this algorithm are actually in the coordinate system of the scene and not in the coordinate system of the camera j. The motivation for this unusual transformation will be explained in the view synthesis part of the thesis.

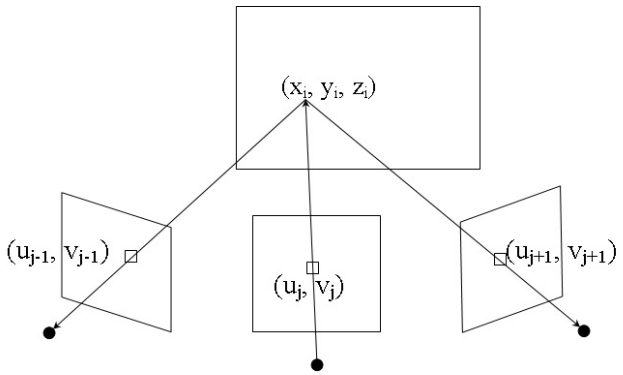

Figure 3.2: Ray re-projection

3.4 Pixel Similarity Function

One of the most simple and popular functions for quantifying pixel similarity is the Sum of Absolute Differences (SAD). Smaller SAD values mean that pixels in neighbouring cameras are similar. Because of its computational simplicity, it was decided to use it for the GPU implementation. Since individual pixels are sensitive to the noise and changes in light when dealing with SADs, block matching techniques are used - rectangular blocks of pixels that are located around the target pixel are compared.

3.4.1 Block Matching Technique

The standard block-matching SAD algorithm is quite simple: for each pixels that have to be compared a block is placed exactly in rectangular shape and the sum of absolute difference of the corresponding pixels in the block are calculated. This approach works great on rectified images, but because cameras in most of the setups are not rectified (they might be far from being parallel to each other) it is better to use a modification of the block matching algorithm called projective block matching.

3.4.2 Projective Block Matching

In projective block matching, one uses a rectangular $(2M + 1) \times (2L + 1)$ pixels grid around the pixel (u_j, v_j) to compute SAD. Most pixels in this grid should have the depth values close to each other. This is why in this scheme, the whole grid is projected to a given depth z_i and then re-projected back to neighbouring cameras $j - 1$ and $j + 1$ (Figure 3.3). Then the similarity function between a neighbouring $\mathbf{I}_{j-1}$ and reference $\mathbf{I}_j$ images is defined as $F(\mathbf{I}_{j-1}(\alpha_j, \beta_j), \mathbf{I}_j(u_j, v_j))$ where (α_j, β_j) is the re-projected 2D coordinate of the grid centered at (u_j, v_j) and is equal to:

$$SAD = \sum_{k=-M}^{M} \sum_{p=-L}^{L} |\mathbf{I}_{j-1}(\alpha_{j+k}, \beta_{j+p}) - \mathbf{I}_j(u_{j+k}, v_{j+p})| \tag{3.2}$$

The advantage of projective block matching over conventional orthographic projection technique is due to the fact that objects shapes in different cameras are different depending on the angle that a camera makes with the scene. The method has two main disadvantages:

1. The closer the angle between the cameras to 90 degrees the worse the results are. This can be explained by the fact that in such cases the rectangular block

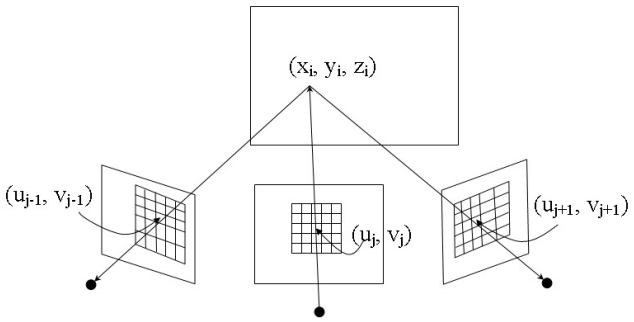

Figure 3.3: Projective block matching

of pixels will be strongly distorted by perspective transform;

2. For large SAD blocks the pixels closer to the edge of the block have a higher probability of having different depth than the central pixel.

3.5 Virtual Viewpoint Rendering

The next step is to generate an arbitrary viewpoint specified by the user. We propose to generate the new virtual viewpoint $\mathbf{I}_m$ from the two neighbouring real cameras $\mathbf{I}_j$ and $\mathbf{I}_{j-1}$ in three steps:

1. Project the pixels from neighbouring cameras into 3D spaces using the depth map information;

2. Re-project the 3D points onto the image plane of the virtual camera using one of the available 3D rendering APIs, e.g. DirectX/OpenGL;

3. Post-process the rendered image by filling void pixels created by occlusions, sampling, and low textured regions using an interpolation technique.

The first step consists of creating 3D vertices that can be used by a rendering API. The number of vertices equals to the number of pixels in each of the cameras multiplied by the number of neighbouring cameras used for the interpolation (in our case is equal to two). In this scheme there is a 3D vertex associated with each pixel of the real camera. The coordinates of the vertices are calculated based on the computed depth map, with color being the color of a projected pixel. One could also project the pixels from all available cameras which would allow us to represent the scene more accurately as it will reduce occlusions and increase image quality. But the amount of computations would be prohibitive for a real-time implementation with current GPU. For this reason, we decided to use only two neighbouring cameras.

Note that since one needs to recreate a scene from 3D vertices in order to be compatible with graphic cards, all depth maps must be in the same global coordinate system. The rendering API for the viewpoint rendering is based on OpenGL. At this step, the data obtained from the previous step is transformed using OpenGL virtual rotation, translation, and projection matrices. As a result, we get a rendered picture of the 3D world, as if it was viewed from a camera located at a virtual viewpoint. Because of sampling and occlusions problems the image will have unrendered pixels that will deteriorate the image quality. In order to solve this problem a post-processing step is necessary where all unrendered pixels are filled using a linear interpolation technique.

The final step is done by interpolating each void pixel from the nearest valid pixels that have color information in the four directions: left, right, top and bottom (see Figure 3.4). Directions that do not have any valid pixel color information (i.e. the direction goes to the picture boundary) are just ignored. So if a pixel that needs to be interpolated is $\mathbf{I}_m(u_h, v_h)$ and the four nearest pixels are $\mathbf{I}_m(u_l, v_h)$, $\mathbf{I}_m(u_r, v_h)$, $\mathbf{I}_m(u_h, v_t)$ and $\mathbf{I}_m(u_h, v_b)$ in each direction, the void pixel will be filled using following formula:

$$\begin{aligned}\mathbf{I}_m(u_h, v_h) = \frac{v_b - v_t}{(u_r - u_l) + (v_b - v_t)}\Big(\frac{(u_r - u_h)\mathbf{I}_m(u_l, v_h)}{u_r - u_l} + \\ \frac{(u_h - u_l)\mathbf{I}_m(u_r, v_h)}{u_r - u_l}\Big) + \\ \frac{u_r - u_l}{(u_r - u_l) + (v_b - v_t)}\Big(\frac{(v_b - v_h)\mathbf{I}_m(u_h, v_t)}{v_b - v_t} + \\ \frac{(v_h - v_t)\mathbf{I}_m(u_h, v_b)}{v_b - v_t}\Big).\end{aligned} \tag{3.3}$$

3.6 GPU Accelerated Algorithm and its Implementation

Unlike CPU that works at a really high frequency to achieve high speed, GPUs have a parallel architecture composed of numerous streaming multiprocessors that work at a lower frequency allowing for lower power consumption and faster speed if the algorithm can be made parallel. In February 2007, NVIDIA introduced CUDA which made it possible to write general-purpose algorithms that run on GPUs in an efficient manner. From the programmers viewpoint CUDA is like an extension of the C language. The code that is written in CUDA is running in threads that are

(a)

(b)

Figure 3.4: (a) Generated view with void pixels (one of the pixels to be interpolated is in yellow and the directions of search for pixels to be used in bi-linear interpolation is in red) and (b) interpolated void pixel using the proposed algorithm

grouped together in blocks of code. Threads in a same block share a fast memory region called shared memory. For the full CUDA specification please refer to [30].

The algorithm presented in the chapter can be easily made parallel using CUDA. Thanks to CUDA graphics inter-operability one can do both depth estimation and viewpoint rendering without the need to process data on the CPU which reduces significantly the processing speed as CPU to GPU transfer are known to be very slow. Since, getting a maximum performance on a GPU depends on a lot of small things we will present some important implementation details.

Stam in [33] presented an efficient way of computing disparity maps of rectified images using CUDA. We modified his algorithm to deal with our context where rectification is not necessary. The design specifications for our algorithm implementation are:

1. Avoid obscenely redundant computation — many computations performed for one pixel can also be used by neighbours;

2. Keep global memory coalesced;

3. Minimize global memory reads/writes;

4. Exploit texture hardware — textures in CUDA have the interpolation implemented on a hardware level;

5. Create enough threads and thread blocks to keep the streaming processors busy.

During the step of SAD calculations it is hard to keep the global memory access coalesced since after the re-projection of a 3D point to left and right cameras the access pattern to the memory storing information about the values of the pixels of the respective cameras is completely random. Hopefully texturing from CUDA Arrays helps us to circumvent the requirement of coalesced memory access, as well as providing boundary clamping and hardware interpolation. This is really important considering the fact that after the re-projection step the coordinates of the pixels in left/right image aren't going to be integers. This is why the images obtained from the cameras are stored in CUDA texture memory.

In the first part of the algorithm, each thread is dedicated to process part of a column of pixels of the camera for which the depth map is being estimated. Unlike most of the algorithms for depth estimation on a GPU, each thread computes SAD of column of pixels with a block size of $(2M + 1)$ instead of individual pixel. It accumulates the SAD function between pixels in the reference image and its corresponding re-projected pixels in the neighbouring images. The SADs function for each column are stored in shared memory arrays. To illustrate thread cooperation for a 5×5 kernel, a 16 thread block size is shown in Figure 3.5. Thus we have one shared memory array that contains the sum of SAD functions between the reference and the left cameras and the reference and the right cameras. After the calculations for a column are done, each thread sums up neighbouring column values within the block width $(2L + 1)$. Once the first row of pixels has been processed by a thread block, a rolling window scheme is applied to speed up the calculations. Instead of repeating the SAD calculation and summation for each of the columns, the SAD function for the pixels in the first row are subtracted from the corresponding one

in the accumulation arrays and then the SAD value of the pixels in a new row are added. This reduces the amount of calculations that has to be done and more importantly reduces the need to read and write into memory. The whole process is repeated for every possible z_i. The trade-off between parallelization and sequential update of the column SAD functions depends on the number of stream processors that GPU has and can only be determined empirically.

Because the SAD blocks on the far left side and far right side of the block of threads require extra $\lfloor \frac{(2L+1)}{2} \rfloor$ SAD calculations, some extra threads will have to be used. In Figure 3.5 these are threads 0-3 on the far right side.

Lets analyze the number of operations that this approach saves: assuming that the height of the block is $2M+1$, to calculate the whole column of pixels each thread needs to make $2M+1$ reads from texture memory (it's actually $3*(2M+1)$ reads since we need to make an extra read for left and right cameras as well) and the same number of writes to shared memory. At the same time, we need to re-project each pixel first to global coordinate system and then back to left camera and right camera, which makes it to at least $3*(2M+1)$ (in reality it's going to be a bit bigger due to the fact that re-projection consists of several arithmetic operations). Using the trick described above each thread for each new row of pixels will only need to make 3 reads from texture memory, to subtract the SAD of top row of pixels and to add the new one) and one write to the shared memory. The number of operations that will be done at this step will be 3 as well. That gives a significant reduction in number of operations performed by GPU especially for block-matching with big block size.

One thing regarding the use of shared memory has to be mentioned: since the order of execution of the threads in a block of code is random, some of the threads may be far behind in number of performed instructions than others. That means that for the threads in the same block, before performing SAD accumulation step we have to synchronize all the threads in order to get correct results. Hopefully, CUDA provides an easy barrier synchronization mechanism using **___synchthreads()** command. This allows to wait until the shared data is filled for the current row of blocks of pixels.

We can further improve performance by using a technique that shows how even smallest changes to the code on a GPU can lead to a completely different behaviour.

- The local variables defined inside the GPU code are stored in registers, which

is the fastest memory type on GPUs. But developers are not able to allocate the arrays of arbitrary size in CUDA registers, they have to use global memory instead. Interestingly, if the size of the array is known in advance then the allocation can be performed in a register memory. We decided to utilize this fact and store the height of the blocks of pixels that has to be processed by one thread using a preprocessor definition (**#define** directive in C/C++). This allows us to allocate an array responsible for handling the SAD of the whole column in a register memory (due to the fact that the size of the array is known during the compilation time). And as a result a higher memory access speed and reduced number of operations. We do not need to recalculate the SAD for a top row of pixels now as we can look it up in a corresponding row of an array. The test results have shown that this approach allowed us to increase the speed of the depth map calculation by two folds by simply adding the use of only one extra register per thread. This also shows how important the implementation details are when dealing with general purpose GPU programming.

- Another interesting capability of the CUDA compiler is that it is able to unroll the loops of a known size. This is why we decided to store the number of depth levels that were sweeping through in a preprocessor definition. That might seem to be impractical, since if one wants to use a different number of depth levels then one would have to recompile the software, but if you consider the application domain of real-time FTV (concerts, sports events) then these settings are known ahead of time and they should not change during the event. We believe that this trick is a reasonable trade-off between having flexibility in the software and having a higher speed of computations.

- Cameras' projection matrices are stored in constant memory which provides cache for faster access.

- In order to further increase efficiency, we used CUDA intrinsic functions for multiplication and division. All calculations are performed in single precision.

For the image viewpoint interpolation, the processing is performed both using CUDA and OpenGL. The projection and post-processing steps are done using CUDA, whereas re-projection to a virtual plane is done using OpenGL. CUDA allows

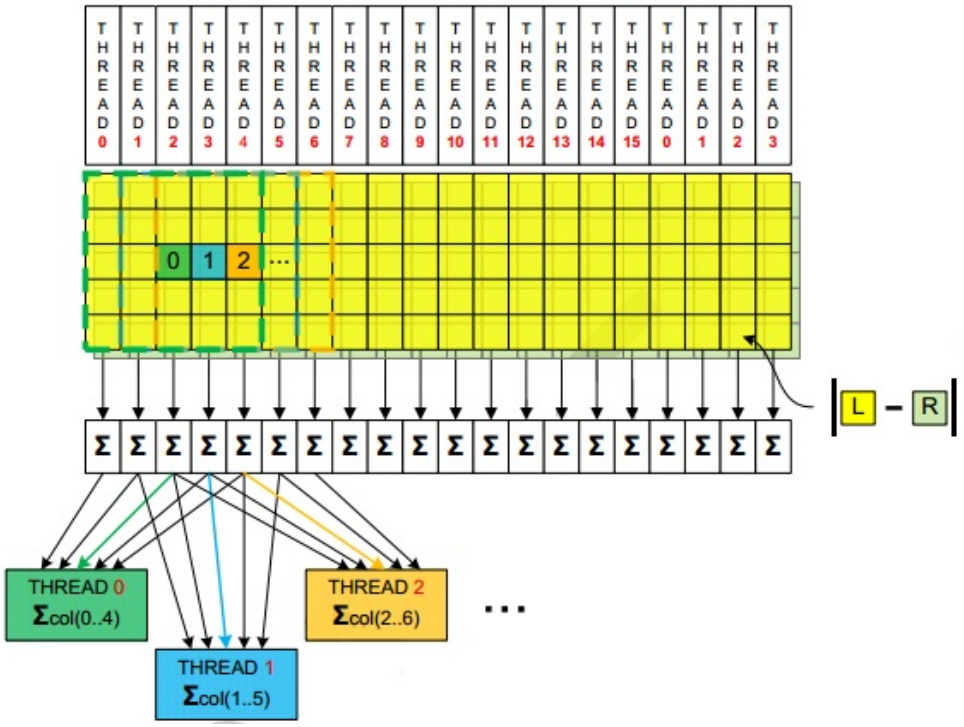

Figure 3.5: Thread cooperation when calculating SAD by using shared memory([33])

to use graphics inter-operability which means that some of the OpenGL resources can be mapped into the address space of CUDA and *vice versa*. Particularly, vertex buffers and render buffer objects can be accessed from CUDA.

Using this scheme, the projection step is done by starting a single thread for each pixel that have to be projected into 3D with the following information: 3D coordinates (can easily be derived from the depth map), color of the pixel, and alpha value. The values are stored on the OpenGL 3D vertex buffer packed as 3 floats for coordinates and one integer representing color and alpha in base-256 system (since each of the color components is represented by the number between 0 and 255). As mentioned previously, each pixel has a separate 3D vertex. This allows us to avoid collisions when we are writing information to the depth buffer, making the projection step fast and simple. To generate the view in the virtual plane, we simply feed the view and projection matrices to OpenGL (see **Section 3.6.1** for more details on how to do it) and render the virtual image into a frame buffer. This frame buffer is accessed once again by the CUDA thread during the post-processing step. If the pixel alpha channel is not 0, then it does not need to be interpolated and the thread is killed, otherwise we perform the steps described in the previous

section to write a new value into the frame buffer. The sequential search for the non-zero values is not the fastest operation, but since most hole sizes are small it can be performed fast. Following hole filling, the resulting image is displayed to the user.

Note that due to the fact that calculations of the depth map and view generation are independent across pixels, the approach is easily scalable by adding more GPUs. When scaling, each CPU thread is responsible for working with one GPU. Unfortunately, it doesn't scale linearly due to the fact that often GPUs reside on the same bus and data transfer in parallel is currently impossible.

The overview of the whole pipeline is shown in Figure 1.4. Note that all the computations are performed completely without the CPU being involved. This allows the main processor to be employed for other tasks. Another noticeable advantage of such a framework is that the depth estimation parts and rendering can easily be changed as they don't depend on each other, all is needed are the resulting depth maps.

Developing using CUDA is sometimes somewhat tricky, due to the fact that all of the functions that are meant to be executed on a GPU have to be in the same file in CUDA 4. But this can quickly lead to error and makes it hard to maintain code. Hopefully, this problem can easily be resolved with a correct use of **#include** directives and correct makefile. The include graph for our implementation of the system is provided in the Figure 3.6. The graph shows the logical separation of our framework into separate files (e.g. a separate file for depth estimation, 3D reprojection and hole filling). To avoid compile errors these files are excluded from build. Instead we include them into one file that is responsible for scheduling of the tasks (file **Scheduler.cu** in the graph). We also put all of the data structures and functions that are used by the GPU functions, such as textures, variables residing in a constant memory, helper functions that are used on a GPU, in a separate file called **DeviceCommon.cuh**. All the data structures that are needed to pass the information between the CPU memory and GPU are stored in **DeviceHostCommon.cuh**. The file **MultiGPU.h** simply have all of the functions and parameters that help to run the code on multiple GPUs using CPU threads. Such an approach provides an effective way to separate the logic in different files, while allowing users that are using IDEs capable of tracking includes to prompt developers of available functions and variables.

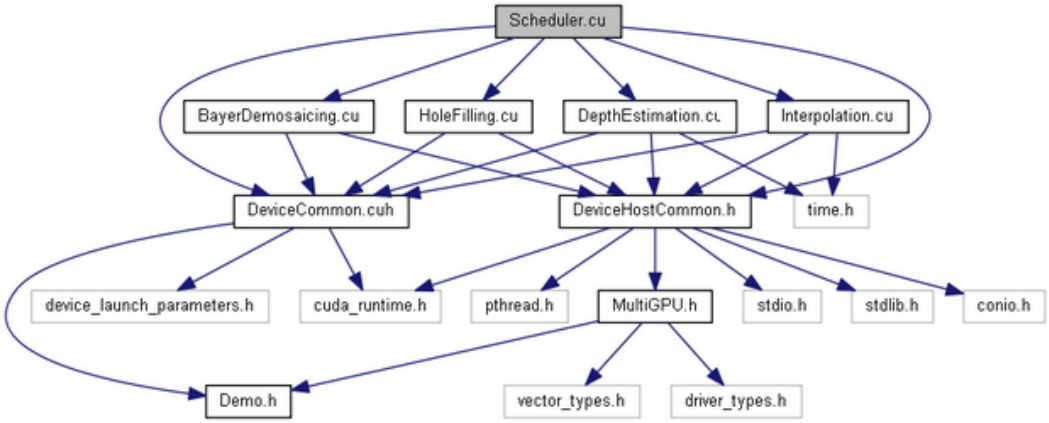

Figure 3.6: The include graph of our implementation

3.6.1 OpenGL Model-View and Projection Matrix Construction for Virtual Camera

To transform the 3D coordinates of objects to 2D coordinates on the screen OpenGL uses two types of matrices: Model-View matrix and Projection matrix.

Since in all the resources of OpenGL the construction of these matrices is done through the OpenGL functions **gluLookAt**, **glTranslatef**, **glRotatef**, **glScalef** for the Model-View matrix and **gluPerspective** for Projection matrix, which did not provide a way to setup all the needed parameters of the matrices (e.g. coordinates of the principal point or the skew coefficient between the axis of the view port), we decided to give more details on how to build these matrices from intrinsic and extrinsic parameters of the virtual camera.

The Model-View matrix, which is the combination of model and view matrices, transforms the coordinates from object space to eye space. Internally, it is represented as a 4 by 4 matrix (Figure 3.7). Keeping in mind that in OpenGL the camera always faces $-Z$ eye coordinates, this matrix (**MV**) can easily be built from rotation (**R**) and translation(**T**) matrices of a virtual camera using Eq. 3.4.

$$MV = \begin{pmatrix} R[0][0] & R[0][1] & -R[0][2] & T[0] \\ R[1][0] & R[1][1] & -R[1][2] & T[1] \\ -R[2][0] & -R[2][1] & R[2][2] & -T[2] \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.4}$$

Projection matrix defines the viewing frustum as well as the projection of 3D scene onto the screen. It's also a 4x4 matrix. We can construct if from near and far Z clipping planes, intrinsic parameters of the virtual camera (**A**), as well as

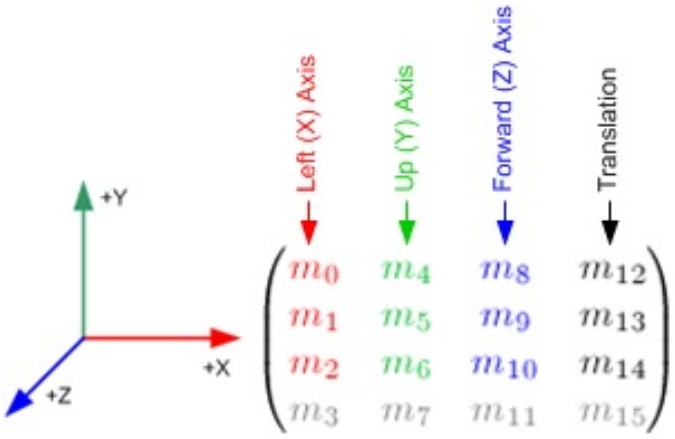

Figure 3.7: The Model-View matrix structure

height(**h**) and width(**w**) of the viewport based on Eq. 3.5.

$$P = \begin{pmatrix} \frac{2*A[0][0]}{w} & \frac{2*A[0][1]}{w} & -(\frac{2*A[0][2]}{w} - 1) & 0 \\ 0 & \frac{2*A[1][1]}{h} & -(\frac{2*A[1][2]}{h} - 1) & 0 \\ 0 & 0 & \frac{far+near}{near-far} & \frac{2*far*near}{near-far} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{3.5}$$

These matrices can successfully be passed to OpenGL using **glLoadMatrixd** function. The only thing one have to remember is that in OpenGL all the matrices are column-major order, so the previous matrices have to be transposed before the transfer.

3.7 Experimental Results

To test our framework, we decided to use the sequence "Breakdancing" published by Zitnick [43] from Microsoft research. The sequence is 100 frames long with a camera resolution of 1024x768 pixels to simulate HD format. In the original paper the depth map was pre-calculated off-line: for each camera the depth map spans 256 depth levels. In order to speed up the calculation process, we decided to sweep through every eighth depth plane. Thus we sacrifice some of the quality to get the algorithm working in real-time. The same approach was used by OpenCV in their calculations on a GPU of disparity for rectified views.

To get the full cycle of image rendering in real-time, we used two Quadro FX 5800 GPUs. Each of them was dedicated to compute depth maps allowing us to achieve real-time speed for two cameras. Following this calculation, all depth information

Table 3.1: The output of the CUDA profiler on Quadro FX 5800

Grid Size	[8 15 1]
Block Size	[128 1 1]
Register Ratio	1(16384/16384)[32 reg per thread]
Shared Memory Ratio	0.25 (4096/16384)[648 bytes per Block]
Active Blocks per SM	4(Max Active Blocks per SM:8)
Active threads per SM	512(Max Active threads per SM:1024)
Potential Occupancy	0.5(16/32)
Achieved Occupancy	0.5(on 30 SMs)
Occupancy limiting factor	Registers

was flushed to one of the cards where the viewpoint interpolation calculations are performed.

Quadro FX 5800 has 30 streaming multiprocessors (SM). That's why the number of SADs calculated by one thread was chosen to be 52, so that the total number of blocks of code that are spawned on a GPU is dividable by 30. The maximum number of active blocks of code per SM for this particular video card is equal to 8, whereas the maximum number of threads is equal to 1024. Based on that the number of threads in one block of code was chosen to be 128. When we ran a profiler it has shown that the maximum occupancy achieved was 0.5, with number of registers being a bottleneck (see Table 3.1). The problem is that the maximum number of registers across all the active threads on SM can not be higher than 16384. So to get a higher occupancy, we need to lower the number of registers in use. To get a 1.0 occupancy factor the number of registers would have to be no more than 16 per thread. But a number of experiments have shown, that lowering the number of registers by either limiting it during the compile-time or by physically removing some register variables from the code can give the maximum occupancy factor at the expense of a lower the performance. It can be explained by the fact that using registers reduces the number of reads to global memory, which is really slow, despite the fact that it limits the number of threads that are being executed concurrently on a multiprocessor.

The time breakdown spent to generate a virtual viewpoint is shown in Table 3.2. As mentioned previously the processing speed does not scale linearly with additional GPUs. In our case by using two Quadro FX 5800 we were only able to achieve a 1.3 speedup compared to one GPU. This is due to the data transfer on the PCIe

Table 3.2: GPU time breakdown to generate a virtual viewpoint

Framework step	Elapsed time(ms)
Depth estimation	16
Projection to 3D	1.15
OpenGL rendering	1.10
Hole Filling	2.4
Total	20.65

2.0 (16 GBps) which is much slower than the internal bus of a GPU (150 GBps). Newer version of the system will allow us to accelerate the transfer rate by a factor two as it will have a PCIe3 bus with a transfer rate of 32 GBps. In addition new GPU card like the Keppler has 3072 CUDA cores compared to the Quadro FX 5800 with 240 CUDA cores. Another thing is that the number of registers per thread in a new architecture is higher, so we expect it to have a higher occupancy factor. Nevertheless the overall frame rate we achieved was 29.7 frames per second which is very promising.

An example of generated viewpoint is shown in Figure 3.8(c) without pixel filling and in Figure 3.8(d) with the void pixels interpolation. The image was generated from two images shown in Figure 3.8(a) and Figure 3.8(b) for camera 3 and 5. We also compared the reconstruction of a virtual camera 4 with the real one by generating a virtual view from cameras 3 and 5 . To do that, we first calculated the weighted luminance channel L of both images using Equation 3.6.

$$L(u,v) = 0.299 * R(u,v) + 0.587 * G(u,v) + 0.114 * B(u,v) \tag{3.6}$$

Then the Peak Signal-to-Noise Ratio (PSNR) value is computed between the real camera L and virtual camera $\hat{L}$ using:

$$PSNR = 10 \log_{10} \frac{255^2}{\frac{1}{W*H} \sum_{u=0}^{H-1} \sum_{v=0}^{W-1} (L(u,v) - \hat{L}(u,v))^2} \tag{3.7}$$

where W and H are the image dimension in pixels.

To determine the effect of using a higher resolution depth map, we also tested our view generation algorithm with depth maps provided by Zitnick. The calculated PSNRs are presented in Table 3.3. Results show that a gain of 4 db in PSNR can be achieved by employing better depth resolution.

Table 3.3: PSNR averaged over 100 frames

Depth map	PSNR
Our algorithm	29
Zitnick	33

3.7.1 Pitfalls along the way

The system described in Section 3.6 was built after a series of trial-and-errors experiments. And we think that it makes sense to dedicate a separate chapter to the problems that we have encountered along the way.

SAD is one of the simplest metrics to quantify pixel inconsistency. But because of its simplicity we were able to use a trick that allowed to reduce the number of operations when calculating the pixel inconsistency of the neighbours. Of course, it leads to artefacts - the incorrect calculation of the depth map for parts of the low-texture background and inability to adequately quantify inconsistency when there is a significant difference in illumination between cameras. That's why we wanted to try a histogram-based approach, where we could compare blocks of pixels based on their histograms. Unfortunately, this approach was not feasible for a real-time application for two reasons - storing histogram in a shared memory minimized the number of threads that could be spawned (due to a 16 KB per SM limitation on the size of shared memory for devices of compute capability 1.3 and lower) and the need to serialize the writes of the calculated histogram from different threads makes GPU utilization useless.

The decision to sample every 8th depth level was made due to the fact that the deterioration level is not significant compared to sweeping through each of the levels. This can be explained by the fact that problems with bad depth sampling in this case occur for the low-texture background regions, for which the depth map is hard to estimate because of the SAD usage. So as a result when the plane sweep is done with a step of one the resulting depth map is less smooth (more depth levels introduce more ambiguities to the estimation process) than the depth map with the step of 8. As a result the sampling rate of 8 is not only the way to speed-up the depth estimation, but also to make the depth map smoother without using post-processing.

The size of the SAD window was chosen based on the experiments - too small

leads to a depth map that looks wavy, too big results in big parts of background on the object boundary being estimated as a foreground. Based on that the size of SAD window was chosen to be the one that gives a better visual quality - 31 by 31 pixels in this particular case.

The artefacts on the boundary of the objects can be avoided by dynamically choosing SAD window shape. One of the possible approaches is to use choose the size and shape of the block using color segmentation techniques - choose the neighbourhood of pixels as a window based on the color threshold. While that could improve the accuracy, we didn't go with this approach because it couldn't be employed by the SAD recalculation technique - subtracting the top row and adding a bottom one might yield incorrect results due to the differences in window shapes.

3.7.2 End-user Delivery

As it was stated in Chapter 1.2 one of the main problems is that it is often hard to deliver a service to end-user, because the view generation method is either not scalable or too expensive to be scaled. So how can the proposed framework deal when the novel views need to be delivered to numerous clients simultaneously?

It might seem that the method described in the thesis is impossible to scale due to the fact that one computer will never be able to generate an arbitrary number of views. While this is true, there is a way to overcome this problem. Instead of making the "server" responsible for all the processing - depth map estimation, view generation, and streaming *all* generated views to clients, one can delegate the step of view generation to the "client". This is possible since the view generation process is performed using the OpenGL API which is generally supported by all of the PC graphics card and computationally is not expensive. Any modern computer's graphics card should be able to render a point cloud generated by depth maps which are the closest to a virtual camera. Thus if each of the clients is responsible for its own view generation, it will release the server responsible for only depth estimation and data streaming. In the next Chapter, we will describe in details the OpenGL client.

(a)

(b)

Figure 3.8: (a) Camera 3 image, (b) Camera 5 image

(c)

(d)

Figure 3.8: (c) Virtual image generated from cameras 3 and 5 without pixel filling, and (d) Virtual image generated from cameras 3 and 5 with pixel filling

Chapter 4

Web-based FVV video-player[1]

This chapter focuses on the problem of how to efficiently deliver multi-view video service to an end-user. Since most applications are using web apps instead of installing desktop applications our goal is to develop a way to build FVV video-player that could be used in both FTV-applications (e.g. creating a novel view of the scene of the real-time events) and FVV (i.e. creating a novel view for prerecorded events) without having to install plug-ins. We wanted the player to have following characteristics:

- Works out of a box in a browser without the need to install any plug-ins;
- Allows the user to arbitrarily change the viewpoint in real-time;
- Allows the user to instantly choose arbitrary frame to be displayed (e.g. if the user chooses 20th frame then he doesn't need to wait for all the previous 20 frames to be downloaded from server);
- Allows the user to pause the video. While the video is paused the player should continue the download process for the subsequent frames;
- The information about the same frame should not be downloaded from server more than once, i.e. the information about already downloaded frames has to be cached;
- The player must be asynchronous, i.e. while the new frames are being downloaded from the server the GUI of the browser as well as the player should not be blocked;

[1] A version of this chapter has been accepted for publication at IEEE International Symposium on Multimedia.

- The player has to be open-sourced, so that other people could improve and extend it.

The main idea is to create a prototype of the YouTube-like video player for FVV. The rest of the chapter is organized as follows: first, we make an overview of technologies that made it possible to create such a player and describe the architecture of the proposed browser-based player as well as details of its implementations. Then we provide experimental results on Huawei/3DLife challenge dataset. We conclude by describing the advantages and disadvantages of the proposed player and how it compares to existing solutions.

4.1 Proposed Browser-based WebGL FVV Player

As described in the Chapter 3, it would be impossible to efficiently deliver FTV or FVV service to a large number of users if all of the computations were performed on the server. To solve this problem, it is better to send the pre-computed depth maps to the user and perform rendering on the end-user machine. This could be done using OpenGL, but that would require that the user would need to install a desktop application. Until recently, it was impossible to perform rendering on a client GPU using a web browser, but with the recent introduction of WebGL API in 2011 this is no longer true. Which means that it is now possible to build a web app that would stream the depth maps to client's computer and perform the rendering on the end-user GPU.

4.1.1 Premises of Browser-based WebGL Player

Recent advances in Web technologies made it possible to create a browser-based FVV player that doesn't require user to install any plug-ins.

The fifth revision of the HTML markup language has introduced new elements. Amongst them is a **<canvas>** element that allows to dynamically render 2D shapes using JavaScript.

In March 2011 Khronos Group has released WebGL - OpenGL ES 2.0 based JavaScript API for rendering of a 3D and 2D graphics within a browser without the use of any plug-ins. WebGL is using HTML5 **<canvas>** element as a target for displaying the results. For the purposes of 3D rendering WebGL supports OpenGL ES Shading Language, Version 1.00, thus allowing it to be performed on the clients

GPU. WebGL has been enabled on all platforms that have a capable graphics card with updated drivers starting with Mozilla, Firefox 4.0, and Google Chrome 9. Starting with Safari 6.0 and Opera 11 these browsers also have native support for WebGL. Experimental version of WebGL is also implemented for IE 11. For the earlier versions of IE the support for WebGL can be added by installing third-party plug-in.

In January of 2011 jQuery Team released jQuery 1.5 - a cross-browser library that helps to make it easier to right client-side JavaScript code. Among other things it has also introduced deferred objects. Deferred objects can be thought of as "a way to represent asynchronous operations which can take a long time to complete" [19]. So instead of blocking a browser from subsequent code execution until the results of the operation are retrieved, a deferred object is returned immediately that will eventually change its state, once the operation is finished. Then callbacks can be attached to this object that will be called once the deferred object will change its state. This is especially useful when communicating with server, since the server's latency at a given time is unknown.

All these frameworks and technologies make it possible to implement a new FVV player that have all of the features described in the beginning of this Chapter.

4.1.2 Proposed FVV Data Format

Since we wanted to focus on creating a FVV player with a YouTube-like functionality, we decided to keep the data format for the first prototype simple but at the same time efficient. In most cases, the FVV video player is provided as a pair of depth and texture maps for each of the cameras either calculated using the depth estimation algorithms or by using color-depth cameras like the Kinect. Since loosing the precision of the depth map due to compression can lead to significant artifacts in the image, especially on the borders of the image, the depth maps information should be compressed using lossless techniques. The color textures on the other hand can be compressed using lossy techniques without a significant loss of quality in 3D reconstruction. An excellent perceptual study of the influence of depth and texture compression can be found in [10]. With regards to this issue we have decided to compress the depth map using PNG compression and compress the color map using JPEG compression. This allows us to easily and efficiently store the data without having to develop our own data compression algorithms. In addition this

scheme allowed us to quickly uncompress the data in Javascript for the rendering.

We propose the data to be stored in a binary format in the following way:

1. The first twelve floats (i.e. first 48 bytes) are dedicated to a header of the file, which stores:
   (a) The number of columns of the cameras;
   (b) The number of lines of the cameras;
   (c) The number of the cameras;
   (d) The near Z clipping plane;
   (e) The far Z clipping plane;
   (f) The number of frames in a video;
   (g) The next 6 numbers define the bounding box of the scene across all the frames and cameras: minimum value of the X, Y, Z coordinates and maximum value of the X, Y, Z coordinates in the point cloud;
2. The parameters of the depth and texture cameras: intrinsic and distortion parameters of the RGB cameras, intrinsic and distortion parameters of the depth camera, rotation and translation matrices from depth camera to RGB, the depth and rotation parameters from the depth camera to global coordinate system;
3. For each frame in a sequence a pair of unsigned 32-bit integers that define the size of the compressed depth map using PNG and compressed texture using JPEG.
4. Subsequent bytes are the depth map stacked in vertical direction and compressed using PNG and stacked textures compressed using JPEG.

This header format allows to have a random access to a separate frame since after the header of the file is processed the start and end byte of the compressed depth maps and textures for each of the frames is known. One could compress the depth map and texture for each of the cameras in a frame separately, but for reasons specified later in the thesis we decided to store them as a stack.

4.1.3 General Overview of the Proposed Player

When the page of the particular FVV video link is requested the player starts the communication with the server. All of the communication is done via standard GET request of the HTTP protocol with the information about the start byte and end byte of the BLOB chunk. These requests can easily be sent to server via *XMLHttpRequest* object. The attributes and methods of this object allow us to set the response type, which in our case is *"blob"* as well as whether the request should return the control to the subsequent Javascript code immediately (i.e. asynchronous request) or if it should wait for the response first (i.e. synchronous request). Since we are interested in non-blocking requests all of the requests to server were performed in an asynchronous manner. When requesting the data, we assumed that the server handler would accept GET request with two parameters - "start" and "end". These parameters allow us to identify the start byte and end byte of the data to be returned. For example, for a web app residing at http://volumizr.appspot.com, the request of the header of the video with id equal to 1 looks like: GET `http://volumizr.appspot.com/file/1/download?start=0\&end=47`.

The player starts with requesting the header of the BLOB (i.e the first 48 bytes). At this step it creates an array of needed size to store the parameters of the cameras as well as start bytes of each of the frames and their length (this is needed due to the fact that the size of one compressed frame is different from another). Once the header is processed the player requests the bytes of the BLOB that correspond to parameters of the cameras and the sizes of the compressed frames. At the same time, the player populates some of the data that is needed for rendering across all the frames. This data is going to be used by WebGL shaders. Once this step is done, the player queues requests of all the frames. This is possible due to the fact that all the needed information about the size of each frame was already processed. When a compressed frame is downloaded it is being decompressed and stored in a cache. For JPEG decompression, we simply created a BLOB from byte array corresponding to texture of the frame with MIME type *image/jpeg* and used *FileReader* object that reads the actual pixel data. For the PNG decompression, we used a third-party library [4]. This library was faster at decompressing the depth map than the one used to decompress the texture.

At the same time as frame data is being requested from server, the player starts

a rendering loop. The request queue is asynchronous, so once it is created, the code won't wait until all of the requests are finished, instead it will pass the control to the subsequent code. If there is a frame ready for rendering, the player allows a person to use controls to choose a viewpoint of the scene. The overview of the proposed player setup and interactions between different parts of the system are presented in Figure 4.1.

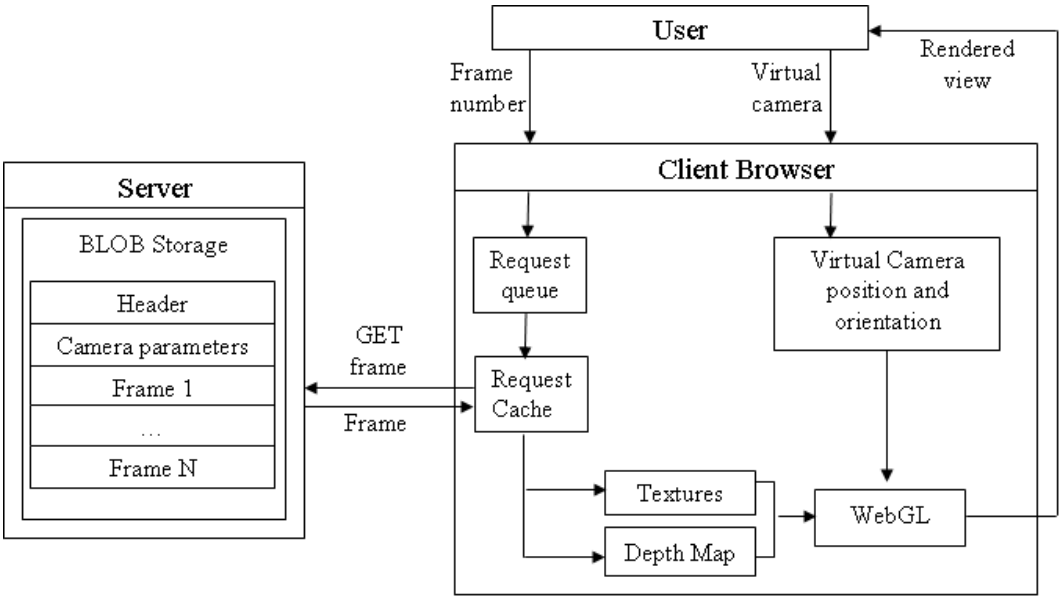

Figure 4.1: The overview of our FVV player setup

If user interacts with the time bar and chooses a different viewpoint, the initial request queue is terminated and a new queue is created (it queries all the frames from the one chosen by user and up to the end of the BLOB).

The queue requests are chained which allows the player to download the data even when the player is paused. Thus it allows a user to pause a video and wait for all the data to be downloaded.

The formation of the queue and requests are done asynchronously which means that the user is able to interact with the player's GUI at any time. In addition, as it was stated in the beginning of this Chapter, we want all the already obtained frames to be cached, so that the player never requests the same frame twice from the server. Both of these features are tricky to achieve, but with the use of jQuery deferred objects this is possible to be implement.

4.1.4 Implementation Details

Rendering a Novel View

The rendering of a novel view is done through the WebGL API. When we want to render a new frame, we pass the uncompressed depth maps and textures to WebGL. Using two shaders - vertex and fragment, WebGL create an image that need to be displayed on the HTML5 canvas. To do so we need to pass several information to the shaders:

1. The vertex buffer that contains the information about the (u, v) coordinates of the pixel in a camera that has to be rendered (passed as **attribute** variable). We have to do it since in GLSL 1.0 (the version of the GLSL language that is currently supported in WebGL) there is no way to figure out the index of the vertex that is being processed by vertex shader;

2. The vertex buffer with index of the camera in the scene that is being processed (passed as **attribute** variable);

3. The parameters of the cameras needed for re-projection (passed as **uniform** variables). Here we have to note that there is no need to create a separate variable for each of the cameras to store parameters as GLSL supports the array of **uniform** variables as long as its size is known at shader compile time. So we have simply chosen it to be big enough to store the information about all of the cameras;

4. View and projection matrices which define the position of the virtual camera in global coordinate space and the way the 3D points are reprojected to virtual camera plane (passed as **uniform** variables);

5. The bounding box coordinates (passed as 6 **uniform** variables);

6. Two textures that correspond to depth map and color map of the cameras (passed as **sampler2D** variables).

First, vertex shader re-projects every pixel of the cameras in the scene in the 3D global coordinate system. To do that, we sample the depth textures for the correct values of the depth map of the point-cloud using the information provided by vertex buffers and using the parameters of the camera. WebGL limits the number of possible textures that can be used by a shader to be equal to 4. So having an

array of textures is impractical. This is the reason why we made the decision to stack-up the depth map and color images together as a colored point-cloud. The next step (still done in vertex shader) is to shift the 3D vertices in such a way that the point-cloud bounding box is centered at $(0, 0, 0)$ in the global coordinate system. This makes it possible for the user to correctly rotate the virtual camera around the center of the scene. Then the points are projected onto the screen using Equation 4.1:

$$gl\_Position = M_{Projection} * M_{ModelView} * \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ 1.0 \end{pmatrix} \tag{4.1}$$

where x_{obj}, y_{obj}, z_{obj} are the coordinates of the centered vertices, $M_{Projection}$ and $M_{ModelView}$ are the projection and model-view matrices respectively, and $gl\_Position$ is the final position of the vertices on the screen. At the end, the vertex shader passes the position on the texture corresponding to each of the 3D points to the fragment shader, which in turn uses it to sample the color from the color texture and store it in the $gl\_FragColor$ variable - the variable responsible for the final colour of the fragment. After the rendering is done, the picture for a novel view is displayed on the canvas element attached to a player.

Request Queue and Frame Cache

The jQuery deferred objects are basically a way to specify a chain of callbacks. Each time a programmer chains a new callback, a new deferred object is created. At any given time the object can be in either 3 states: resolved, rejected, or pending. Whenever the object is in resolved state - the callbacks chained for the resolved state are called, whenever the object is in rejected state, the callbacks chained for rejected state are called. More details on deferred objects can be found at [2].

In order to create a queue of asynchronous requests to server, we simply need to chain the download requests to the deferred object. Each successful frame request will turn the deferred to the state resolved, thus firing the next download request. Whenever the user changes the frame using the time bar the current request is aborted and the state of deferred object is changed to rejected, thus stopping the player from issuing all of the queued requests to server. One might think that the better approach would be to simply start the requests for all the frames at the same time, using one or several *XMLHttpRequest* objects. But this would only slowdown the download process and make the user experience worse, since in this case, the

user would have to wait more for one frame to be downloaded and thus would delay the rendering of the scene from an arbitrary viewpoint. Another thing that has to be noted is that browsers quite often limit the number of simultaneously opened *XMLHttpRequest* connections to the server. These two factors are making it impossible to perform several requests from one client at the same time.

While our approach allows to easily create a chain of asynchronous download requests to the server that will not block the browser, it does not provide a way to check whether a frame has previously been downloaded or not.

Hopefully the caching problem can also be resolved with a use of deferred objects as well. We have adopted the idea described in the official jQuery documentation in [3] to implement our caching mechanism. The idea is to use jQuery deferred objects to build a cache factory that is completely abstract of the actual task that is going to be performed. In this scheme a hash mechanism that stores deferred objects that wrap around request to the server for each of the frames is used. Then one can create a function that constructs a closure around the array and first checks whether an element that corresponds to the requested frame is **undefined**, if it is, then it stores the deferred object wrapper and returns it it to the caller. As a result, if the frame has already been requested the deferred object in resolved state will be returned and the queue of queries will simply continue, if it has not yet requested then the deferred object in pending state will be returned and we will be able to continue the chaining. A snippet of the code that implements this idea is described in Listing 4.1. Every call to the function **jQuery.createCache** will return a new cache with a corresponding cache retrieval function. Since we had only one task that had to be cached - downloading frames from server, we needed to call this function only once and reuse the same retrieval function whenever we wanted to download the frame. The key of the cache retrieval function was chosen to be the number of frame that is being requested. One thing that has to be noted here is that in our cache implementation if the jQuery object associated with a specific key is failing (e.g. the frame download process was interrupted) then the cache-repository for that key is set to be **undefined**, so that the frame could be downloaded again later.

Listing 4.1: jQuery Cache factory [3]

```
jQuery.createCache = function( requestFunction ) {
    var cache = {};
    return function( key, callback ) {
        if ( !cache[ key ] ) {
```

```
            cache[ key ] = jQuery.Deferred(function( defer ) {
                requestFunction( defer, key );
            }).promise();
        }
        return cache[ key ].then( function(){if (callbackDone) callbackDone()
    };
}
```

4.2 Experimental Results

For demo purposes, we built a small server using Python and Google App Engine SDK. All of the FVV files are stored in Blobstore provided in SDK. It is build as a RESTful API so that all communication is done via standard http requests. The handlers for data retrieval requests accept two parameters that correspond to the start and end byte positions and return a partial data from Blobstore.

For our display data, we decided to use the data provided for the Huawei/3DLife 2013 ACM Multimedia Challenge ([1]). It is a perfect fit for the purposes of our demo as it was obtained using 5 Kinect cameras located around the scene. The cameras were calibrated using [20] with the needed parameters provided by the challenge organizers. The ACM Multimedia challenge data was preprocessed to comply with the proposed file format and uploaded to the storage on the server. The player was tested in Google Chrome and Mozilla Firefox - the browsers that occupy at least 50% of the market. Examples of the GUI of the player can be seen in Figure 4.2 and Figure 4.3 where the samples are shown for video that has not been fully downloaded yet and the one where all the FVV data is already transferred. The list of available videos to view and the links to them can be found at `http://volumizr.appspot.com`.

The controls of the view position are quite natural: the users should simply click on a player and move the mouse around to change the position and orientation of the virtual view and use the mouse wheel to move it forwards or backwards.

The player works completely asynchronously, so the user always has the control over the GUI with rendering of an arbitrary view being performed in real-time. Since the data that is being transferred is compressed, each frame for 5 cameras takes only up to 1 MB (compared to 23.5 MB if the data is transferred in the form of point cloud). For the 8 MBit/s connection that means that every second the client can download 1 frame, but with the abilities to choose an arbitrary play time and start

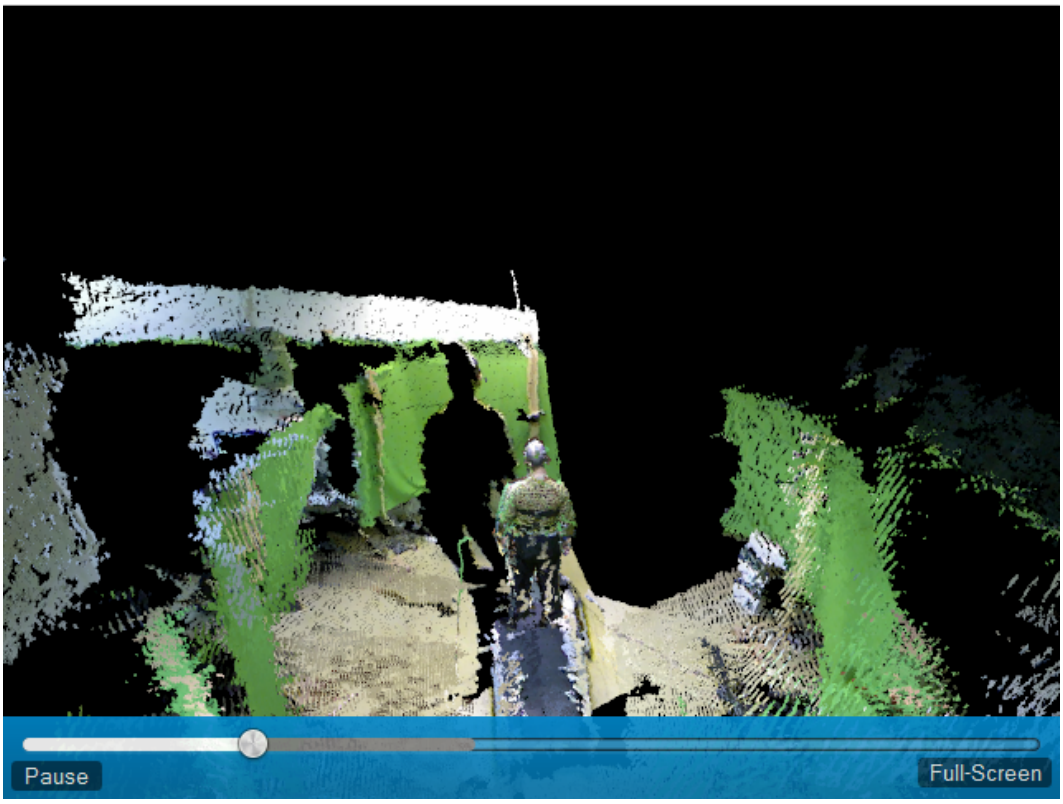

Figure 4.2: GUI of the player when frame downloading is still in progress. Note that the user is able to interact even when only a part of the frames were downloaded

playing the video when even part of the data is loaded that shouldn't cause too many usability problems to the end-user. Still this is currently a bottleneck of our player as once the data is loaded the rendering process is really fast. On a laptop with AMD Radeon HD 6520G in Chrome a frame rate of 40 fps is achieved.

When compared to work presented in [8] the per frame transferred data in our player is substantially lower (1Mbyte vs. 14.4 Mbytes) since we store scene information as depth maps which can easily be compressed and use Blobs for data transferring instead of Base64 encoding. When used in FVV and FTV applications the server serves better as it renders the full scene, not just the foreground with a background being a static picture as it was done in [8]. Although in some applications, this feature might be considered a disadvantage, e.g. virtual reality systems. An approach where the depth maps are used for rendering is way better when it comes to FTV systems, since extracting a high-quality triangular mesh is a very expensive task and thus cannot be used in such a systems. But on the other hand in cases when the video was prerecorded and preprocessed creating a high quality

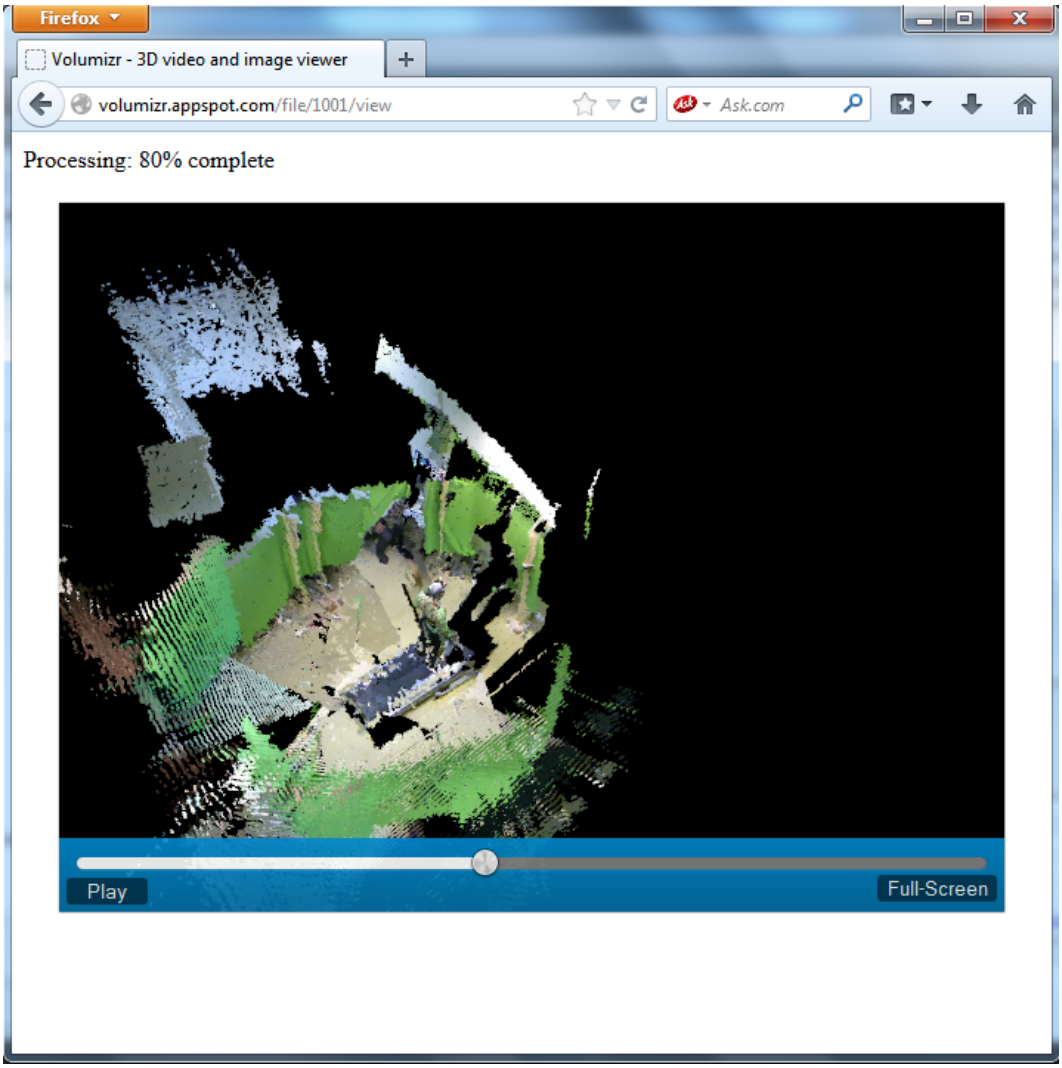

Figure 4.3: Example of the GUI of the proposed FVV player with a picture rendered when all the frame data is already streamed from server.

mesh that represents a scene could yield a really high quality picture. But since we wanted the player to be as versatile as possible and since the major scene reconstruction techniques are calculating the depth map representation, we decided to stick to this rendering mode. As a bonus, this also allows us to use a set of Kinect cameras as a free viewpoint webcam.

The system presented in this Chapter fits perfectly within the framework proposed in Chapter 3. It can easily substitute the part of the framework that uses OpenGL to render a novel view, thus helping the server performance by distributing the computations that linearly grow with the number of service consumers across all of the clients (i.e. each client is responsible for his own rendering). This allows

us to move one step closer to having an efficient implementation of real-time FTV.

Chapter 5

Conclusion

5.1 Conclusion

The future of the television lies in FTV and FVV. Selection of an arbitrary virtual view of the scene by the user gives an ultimate immersive experience to the action. The only problems is that arbitrary view generation is a computationally intensive task. But with the progress of the computers technology this is now possible thanks to the GPU. Considering GPU parallel architecture, the computing power hidden is tremendous. While the first programs that wanted to use this power had to create work-around using GLSL, with the development of general purpose GPU languages such as CUDA, efficient ways of utilizing GPUs for parallel computations have emerged. This work leverages the capabilities of the GPU to adapt existing FVV algorithms to work in real-time with a maximum efficiency.

The thesis addresses two main problems of real-time FVV: generation of arbitrary view from a set of discrete cameras located around the scene in real-time and the problem of efficient scaling of the system for delivering free viewpoint videos to end-users.

In Chapter 3, we proposed a novel GPU based framework capable of generating arbitrary viewpoints from a network of HD video cameras in real-time. The algorithm for depth estimation and view generation are independent from each other allowing them to be computed on different GPUs. The algorithm doesn't need view rectification and is capable of generating depth map from any virtual viewpoint if needed. It can also easily be scaled by adding more GPUs or more powerful GPU like the Kepler from NVIDIA. Since all computationally intensive operations are performed on a graphics processor the CPU is free to do other tasks. The approach is directed towards utilization when covering events that need real-time computation

such as football game or concert. The proposed algorithm and its implementation has several major advantages over the similar recently proposed methods, i.e. [29, 32, 36]: since the method is not based on shape-from-silhouette it can render both foreground and background of a scene, it does not need any pre-computed 3D proxy, and its rendering process can easily be performed on client-side, leaving the server to compute the depth maps from real cameras. An efficient technique delivering FTV to the client has also been proposed. Chapter 3 also discusses the peculiarities of the implementations of algorithms using CUDA and the factors that impact the execution speed which is really important for real-time applications.

In Chapter 4, we proposed an open-source browser-based Free Viewpoint Video player that works completely out of a box without the necessity to install third-party plug-in. The player is implemented using the WebGL API where rendering is performed solely on a GPU. The main focus was on the development of a Flash player-like functionality with real-time arbitrary view rendering.

The player was written in a way that the JavaScript code for downloading frames from server or rendering process is never blocking the browser. Currently the player has following features: user is free to control the viewpoint, as well as the frame to be rendered, user can pause the video, go to a full-screen mode, all of the frames are cached, so the same frame will never be downloaded from the server more than once, the player downloads all of the frames from the server even if the video is paused. Due to the fact that JavaScript is always exposed to the browser, the player is open-source by default. The services for FVV delivery over the internet that use the proposed player can easily be scaled as the player itself communicates with a server using HTTP protocol only and all the rendering is done on the client machine.

A small demo of player's capabilities was created using the server built with Python and Google App Engine SDK. The player was tested on a data obtained from 5 calibrated synchronized Kinect cameras. That makes it possible to use a framework as a basis to turn the Kinect camera into a cheap 3D web-camera. The player perfectly fits the scaling part of the framework described in Chapter 3 as it uses the GPU of the end-user to generate the views from depth and color textures and thus can be used in FTV applications.

5.2 Future Work

Future work will include the improvement of GPU code by using the features of the newly introduced Kepler architecture and CUDA 5, as well as the DMA access between the GPUs. We are also planning to test other similarity functions than SAD which do not assumed that the cameras have exactly the same gain, that the surface is Lambertian, and that the illumination is uniform. To address this illumination invariance problem, we are planning to implement in the GPU the relative gradients function proposed in [42]. We are also in the process of integrating into our video processing computer a new camera technology from Herodion Inc. This revolutionary technology is capable of streaming 12 pixel synchronized HD cameras on a single computer. Our main challenge will be the data transfer of this large video steam to the GPU memory using the DMA access mechanism.

As for the the research presented in Chapter 4, since the work primarily focused on creating a prototype of an efficient FVV player, with keeping the data format simple, we plan to investigate the use of more sophisticated data compression schemes such as H.264/AVC that would allow us to reduce the bit-transfer rate allowing the whole system to work in real-time and deliver the FTV coverage of events over the Internet.

Bibliography

[1] http://mmv.eecs.qmul.ac.uk/mmgc2013/download.php.

[2] http://api.jquery.com/category/deferred-object/, .

[3] http://learn.jquery.com/code-organization/deferreds/examples/, .

[4] https://github.com/devongovett/png.js.

[5] http://sketchfab.com.

[6] http://verold.com.

[7] H. Harlyn Baker, Nina Bhatti, Donald Tanguay, Irwin Sobel, Dan Gelb, Michael E. Goss, W. Bruce Culbertson, and Thomas Malzbender. Understanding performance in coliseum, an immersive videoconferencing system. *ACM Trans. Multimedia Comput. Commun. Appl.*, 1(2):190–210, May 2005. ISSN 1551-6857. doi: 10.1145/1062253.1062258. URL http://doi.acm.org/10.1145/1062253.1062258.

[8] Chris Budd, Oliver Grau, and Peter Schübel. Web delivery of free-viewpoint video of sport events. Technical report, Nov. 2012.

[9] Joel Carranza, Christian Theobalt, Marcus A. Magnor, and Hans-Peter Seidel. Free-viewpoint video of human actors. *ACM Trans. Graph.*, 22(3):569–577, July 2003. ISSN 0730-0301. doi: 10.1145/882262.882309. URL http://doi.acm.org/10.1145/882262.882309.

[10] I. Cheng, Rui Shen, Xing-Dong Yang, and P. Boulanger. Perceptual analysis of level-of-detail: The jnd approach. In *Multimedia, 2006. ISM'06. Eighth IEEE International Symposium on*, pages 533–540, 2006. doi: 10.1109/ISM.2006.124.

[11] R.T. Collins. A space-sweep approach to true multi-image matching. In *Computer Vision and Pattern Recognition, 1996. Proceedings CVPR '96, 1996 IEEE Computer Society Conference on*, pages 358 –363, jun 1996. doi: 10.1109/CVPR.1996.517097.

[12] L. Do, G. Bravo, S. Zinger, and P.H.N. de With. Gpu-accelerated real-time free-viewpoint dibr for 3dtv. *Consumer Electronics, IEEE Transactions on*, 58 (2):633–640, 2012. ISSN 0098-3063. doi: 10.1109/TCE.2012.6227470.

[13] Jean-Sébastien Franco and Edmond Boyer. Exact polyhedral visual hulls. In *British Machine Vision Conference (BMVC'03)*, volume 1, pages 329–338, Norwich, Royaume-Uni, 2003. URL http://hal.inria.fr/inria-00349075.

[14] Toshiaki Fujii, Tadahiko Kimoto, and Masayuki Tanimoto. Ray-space coding for 3d visual communication. In *Picture Coding Symposium (PCS), 1996*, volume 2, pages 447–451, Mar. 1996.

[15] Toshiaki Fujii, Kensaku Mori, Kazuya Takeda, Kenji Mase, Masayuki Tanimoto, and Yasuhito Suenaga. Multipoint measuring system for video and sound - 100-camera and microphone system. In *ICME*, pages 437–440, 2006.

[16] Y. Furukawa and J. Ponce. Accurate, dense, and robust multiview stereopsis. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(8):1362 –1376, aug. 2010. ISSN 0162-8828. doi: 10.1109/TPAMI.2009.161.

[17] Bastian Goldlücke, Marcus Magnor, and Bennett Wilburn. Hardware-accelerated dynamic light field rendering. In *Proceedings Vision, Modeling and Visualization VMV 2002*, pages 455–462, 2002.

[18] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pages 43–54, New York, NY, USA, 1996. ACM. ISBN 0-89791-746-4. doi: 10.1145/237170.237200. URL `http://doi.acm.org/10.1145/237170.237200`.

[19] Andree Hanson, Addy Osmani, Julian Aubourg, Marcus Amalthea Magnuson, and John Paul. jQuery Deferreds. `http://learn.jquery.com/code-organization/deferreds/`, May 2013.

[20] Daniel Herrera C., Juho Kannala, and Janne Heikkila. Joint depth and color camera calibration with distortion correction. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 34(10):2058–2064, 2012. ISSN 0162-8828. doi: 10.1109/TPAMI.2012.125.

[21] T. Kanade, P. Rander, and P.J. Narayanan. Virtualized reality: constructing virtual worlds from real scenes. *MultiMedia, IEEE*, 4(1):34–47, 1997. ISSN 1070-986X. doi: 10.1109/93.580394.

[22] Akihiro Katayama, Koichiro Tanaka, Takahiro Oshino, and Hideyuki Tamura. Viewpoint-dependent stereoscopic display using interpolation of multiviewpoint images, 1995. URL `http://dx.doi.org/10.1117/12.205854`.

[23] Marc Levoy and Pat Hanrahan. Light field rendering. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pages 31–42, New York, NY, USA, 1996. ACM. ISBN 0-89791-746-4. doi: 10.1145/237170.237199. URL `http://doi.acm.org/10.1145/237170.237199`.

[24] M. Li, M. Magnor, and H.-P. Seidel. Hardware-accelerated rendering of photo hulls. *Computer Graphics Forum (Eurographics'04)*, 23(3):635–642, sep 2004.

[25] Keisuke Manoh, Tomohiro Yendo, Toshiaki Fujii, and Masayuki Tanimoto. Ray-space acquisition system of all-around convergent views using a rotation mirror. pages 67780C–67780C–8, 2007. doi: 10.1117/12.740689. URL `+http://dx.doi.org/10.1117/12.740689`.

[26] Wojciech Matusik, Chris Buehler, Ramesh Raskar, Steven J. Gortler, and Leonard McMillan. Image-based visual hulls. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 369–374, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. ISBN 1-58113-208-5. doi: 10.1145/344779.344951. URL `http://dx.doi.org/10.1145/344779.344951`.

[27] Yuji Mori, Norishige Fukushima, Tomohiro Yendo, Toshiaki Fujii, and Masayuki Tanimoto. View generation with 3d warping using depth information for ftv. *Image Commun.*, 24(1-2):65–72, January 2009. ISSN 0923-5965. doi: 10.1016/j.image.2008.10.013. URL `http://dx.doi.org/10.1016/j.image.2008.10.013`.

[28] Purim Na Bangchang, Toshiaki Fujii, and Masayuki Tanimoto. Experimental system of free viewpoint television. pages 554–563, 2003. doi: 10.1117/12.474141. URL +http://dx.doi.org/10.1117/12.474141.

[29] Vincent Nozick and Hideo Saito. On-line free-viewpoint video: From single to multiple view rendering. *International Journal of Automation and Computing*, 5(3):257–267, 2008. ISSN 1476-8186. doi: 10.1007/s11633-008-0257-y. URL http://dx.doi.org/10.1007/s11633-008-0257-y.

[30] NVIDIA Corporation. NVIDIA CUDA C programming guide, oct 2012. Version 5.0.

[31] M. Okutomi and T. Kanade. A multiple-baseline stereo. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(4):353–363, April 1993. ISSN 0162-8828. doi: 10.1109/34.206955. URL http://dx.doi.org/10.1109/34.206955.

[32] Neal Orman, Hansung Kim, Ryuuki Sakamoto, Tomoji Toriyama, Kiyoshi Kogure, and Robert Lindeman. Gpu-based optimization of a free-viewpoint video system. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, I3D '08, pages 15:1–15:1, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-983-8. doi: 10.1145/1342250.1357027. URL http://doi.acm.org/10.1145/1342250.1357027.

[33] Joe Stam. Stereo imaging with cuda, jan 2008.

[34] J. Starck and A. Hilton. Virtual view synthesis of people from multiple view video sequences. *Graph. Models*, 67(6):600–620, November 2005. ISSN 1524-0703. doi: 10.1016/j.gmod.2005.01.008. URL http://dx.doi.org/10.1016/j.gmod.2005.01.008.

[35] Jonathan Starck and Adrian Hilton. Surface capture for performance-based animation. *Computer Graphics and Applications, IEEE*, 27(3):21 –31, may-june 2007. ISSN 0272-1716. doi: 10.1109/MCG.2007.68.

[36] Jonathan Starck, Joe Kilner, and Adrian Hilton. A free-viewpoint video renderer. *Journal of Graphics, GPU, and Game Tools*, 14(3):57–72, 2009.

[37] K. Suzuki, N. Fukushima, T. Yendo, M.P. Tehrani, T. Fujii, and M. Tanimoto. Parallel processing method for realtime ftv. In *Picture Coding Symposium (PCS), 2010*, pages 330 –333, dec. 2010. doi: 10.1109/PCS.2010.5702500.

[38] Masayuki Tanimoto. Overview of free viewpoint television. *Signal Processing: Image Communication*, 21(6):454 – 461, 2006. ISSN 0923-5965. doi: 10.1016/j.image.2006.03.009. URL http://www.sciencedirect.com/science/article/pii/S0923596506000166. Special issue on multi-view image processing and its application in image-based rendering.

[39] Wolfgang Waizenegger, Ingo Feldmann, Peter Eisert, and Peter Kauff. Parallel high resolution real-time visual hull on gpu. In *Proceedings of the 16th IEEE international conference on Image processing*, ICIP'09, pages 4245–4248, Piscataway, NJ, USA, 2009. IEEE Press. ISBN 978-1-4244-5653-6. URL http://dl.acm.org/citation.cfm?id=1819298.1819908.

[40] Jason C. Yang, Matthew Everett, Chris Buehler, and Leonard McMillan. A real-time distributed light field camera. In *Proceedings of the 13th Eurographics workshop on Rendering*, EGRW '02, pages 77–86, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association. ISBN 1-58113-534-3. URL http://dl.acm.org/citation.cfm?id=581896.581907.

[41] Ruigang Yang, Greg Welch, and Gary Bishop. Real-time consensus-based scene reconstruction using commodity graphics hardware. In *Pacific Conference on Computer Graphics and Applications*, pages 225–235, 2002.

[42] Xiaozhou Zhou and Pierre Boulanger. Radiometric invariant stereo matching based on relative gradients. In *Image Processing (ICIP), 2012 19th IEEE International Conference on*, pages 2989–2992, October 2012. doi: 10.1109/ICIP.2012.6467528.

[43] C. Lawrence Zitnick, Sing Bing Kang, Matthew Uyttendaele, Simon Winder, and Richard Szeliski. High-quality video view interpolation using a layered representation. *ACM Trans. Graph.*, 23(3):600–608, August 2004. ISSN 0730-0301. doi: 10.1145/1015706.1015766. URL `http://doi.acm.org/10.1145/1015706.1015766`.