Realtime Free Viewpoint Video System Based on a new Panorama Stitching Framework

by

Muhammad Usman Aziz

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science
University of Alberta

Abstract

Free Viewpoint Video and TV is regarded as the future of digital entertainment allowing users to navigate through multiple video streams of an event to select novel viewpoints. This new capability will be able to give to the users the illusion that they are present at the event. In this thesis, we propose a novel framework for real-time video panorama generation. We first present our work for acquiring multiple high-definition video streams using hardware synchronized cameras with GPU accelerated Bayer demosaicing. The multiple streams are then color corrected using camera gain adjustment and Macbeth color checker chart to remove color variations between cameras. Next, we propose an automatic real-time video stitching algorithm using feature matching based approach and automatic camera calibration that is capable of producing spherical panoramas. The algorithm works by splitting the automatic image stitching steps into one-time and recurring computations. We utilize GPUs for accelerating the recurring computations and achieve real-time performance. By effectively caching the repeated transform maps database, we improve the stitching performance by two orders of magnitude compare to CPU implementations. In order to remove seams between the stitching images, we use Vornoi diagram based optimal seam selection algorithm. The stitched video streams are blended together using the first real-time implementation of Laplacian pyramid blending algorithm using multi-GPUs. Finally, we analyze the problems associated with the scalable transmission and rendering of the virtual views using commodity computers and network resources. The multiple stitching steps are pipelined to distribute the panorama stitching computations between multiple processing nodes to produce visually appealing panoramas at 30Hz without any noticeable blurring or stitching artifacts.

Dedication

This thesis is lovingly dedicated to my mother, Tahira Parveen. Her support, encouragement, and constant love have sustained me throughout my life

Acknowledgements

First of all, I would like to thank my supervisors Dr. Pierre Boulanger and Dr. Nilanjan Ray for their support, financial assistance, mentorship and for giving me the opportunity to work on several interesting projects that helped me to learn so much more beyond the scope of my thesis. Thank you for the friendly learning environment that you have created in the labs and for allowing me to pursue my interests.

Especially I would like to thank Dr. Boulanger for sharing invaluable life experiences and for supporting me to setup all the infrastructure required for this work.

I am also grateful to Dr. Dileepan Joseph from the Electrical and Computer Engineering Department for taking the time to read the thesis and serve as an examiner.

I would like to acknowledge the help from George Dramitinos and Constantin Papadas from ISD S.A. whose support has been instrumental in overcoming several issues.

Special thanks to my friends Waqar and Nael for the debates, food parties and recursion trips that made life enjoyable in Edmonton. Thank you Salik for your invitations and lending a drive whenever needed. I would also like to acknowledge Dr. Atif Kamran and Dr. Farooq Wahab for helping me settle in Edmonton and for providing me guidance throughout my degree.

A big thanks to my fellow lab mates in the AMMI lab, Michael, Sheehab, Ian, Ga-Young, Peter and Stephanie for their support and discussions. I would also like to give a special thanks to Andy Hess for the long brain storming sessions over many lunches on several interesting topics. Congratulations on starting a family and I hope you meet your dreams in life.

I would like to express deep gratitude to my sisters Asma and Hadiqa, and brother Annas for their unconditional love and support throughout my life. I would also warmly thank my aunt Raheela and her family for hosting me at their home on several occasions and providing me with love and care of a family throughout my Masters studies.

Finally, I wish to take this opportunity to express my deepest appreciation to my beloved parents Abdul Aziz and Tahira Parveen for always providing me and my siblings with the best possible education and supporting us throughout our lives. My life achievements wouldn't have been possible without your endless love, support, and belief.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Since the invention of television almost a century ago, viewers are restricted to watch a single view of the transmission controlled by the broadcaster. Even though digital electronics has evolved greatly, little has changed in the way most people enjoy television over the last few decades. This is undergoing an innovation with the development of new technologies around free view point video (FVV) and television (FTV). These new and exciting technologies could revolutionize the world of video recording and transmission by giving viewers the ability to select different viewpoints of a digitized scene. Depending on the technologies used to capture and process the video, users can freely navigate between a set of synchronized camera viewpoints allowing them to select the best viewpoint to see an event. This gives end-viewers more an immersive and engaging sense of presence as if they were actually present at the scene.

Video and television transmission quality has improved significantly over the last few years with the adoption of high definition (HD) compressed streams and displays. Hence, any new technology must be able to keep up with the new and upcoming video quality standards. At the same time, the technology should be integrable without major changes to the infrastructure of normal video transmission e.g. it should be compatible with the existing bandwidth provided by commercial networks and should be compatible with existing set-top boxes at the viewers' homes. Another key requirement is that FVV and FTV systems should be able to deal with live coverage of events like sports matches, telepresence systems, concerts, video conferencing, fashion shows, virtual tours etc that may require serving millions of

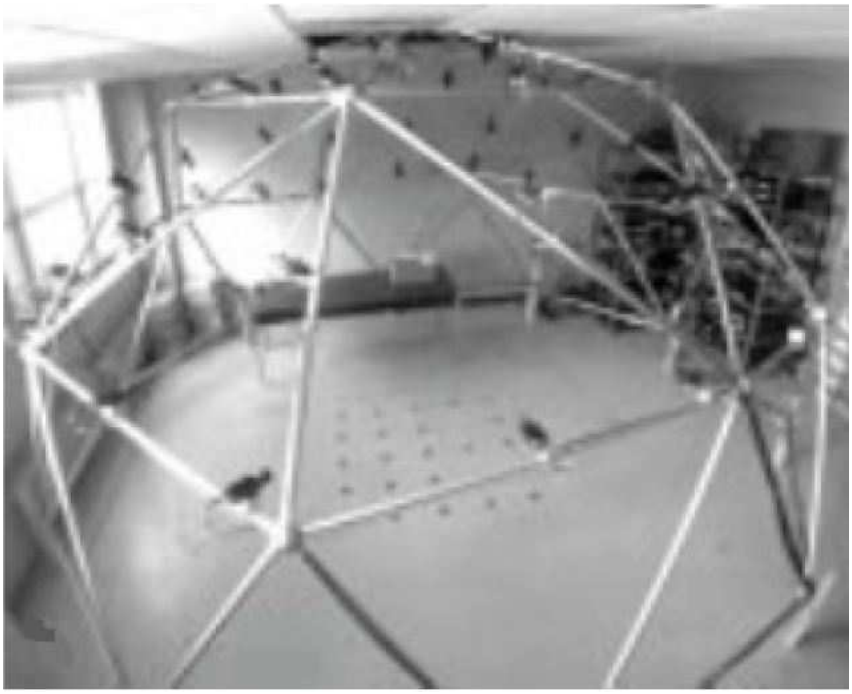


Figure 1.1: Kanade's Virtual Reality Geodesic Dome [38]

viewers. All of these requirements present great scientific and technological challenges, some of which we will attempt to address in this thesis. However before going into the details, we would like to present a brief history of the FVV and FTV systems.

1.1 A Brief History of Free Viewpoint Technologies

One of the first significant efforts to construct a FVV system was made by T. Kanade et al. [38] by developing the *Virtualized Reality* system in 1997. The authors constructed a 5 meters diameter geodesic dome (see Figure 1.1) with 51 monochrome cameras mounted on a frame. Each of these cameras had a resolution of 512 x 512 pixels and were running at 17 Hz. A 3D studio software was created to capture synchronized video of the scene. The synchronized video streams were then processed using a stereo matching algorithm. The result of this long processing was a textured 3D triangular mesh of the scene that can then be projected from any virtual viewpoints using standard computer graphics software.

In the same year, Nayar et al. [65] generated one of the first panoramic videos

using an omnidirectional video camera system. Since omnidirectional camera system has a full 360° field-of-view, it does not need to stitch multiple images together to create a panorama. Hence, a panoramic video can be created with little processing.

In 2000, the company *Flycam* produced an inexpensive 360° full-motion panoramic video camera system [22]. FlyCam system provides real-time 360° panoramic video by stitching together images from multiple video cameras pointed uniformly along the 360°. The panorama is delivered via a http web-server called *FlyServer* that can transmit a live motion panorama to a web browser client. Virtual cameras can be automatically controlled using person tracking with image and sound analysis, resulting in an ”automatic cameraman” for unattended meeting and event capture. Selecting a region of the panorama gives a ”virtual camera” that can be electronically panned, tilted, and zoomed. Multiple, independent virtual cameras can be created providing different users different views of the same scene simultaneously.

By using synchronized video stream from multiple cameras, Zitnick et al. [89] developed an eight cameras system each with a resolution of 1024 x 768 pixels along an arc spanning about 30° at the opposite ends. The cameras were synchronized using two video concentrators connected through the FireWire port. The system calculated depth maps from the camera using a color segmentation based algorithm with refinements. The depth map is then used to construct virtual views.

One of the recent commercial software for FTV [79] using 4K cameras was introduced by KDDI Technologies in 2010 that allows users to watch the soccer game from the perspective of the players (”walk-through system”). The software provides good quality results with a large degree of freedom for virtual views while using advanced multi-view coding (MVC) for transmission compression.

1.2 Motivation

There are many systems that provide high-quality panorama stitching for images and support is available in all major imaging software and mobile imaging devices

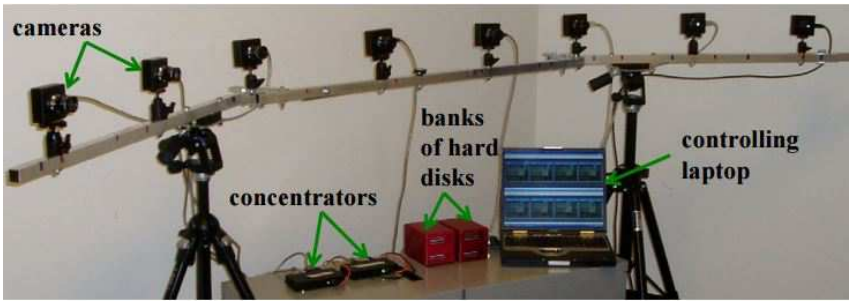


Figure 1.2: An 8 camera system by Zitnick [89]

like mobile phone. However, panoramic video development has been slow, primarily due to the lack of complete systems that can meet the challenges associated with real-time panoramas. A complete real-time panorama video system can be based on the following features:

1. A multiple camera system that can capture high quality video in perfect synchronization;

2. Each camera must be color calibrated or color equalized;

3. A real-time high quality Bayer demosaicing algorithm to guarantee picture quality;

4. Efficient design of a stitching algorithm that is suitable for real-time video processing;

5. A high-quality image blending algorithm that can match the quality of video panoramas to image based panoramas;

6. Usage of parallel processing to handle the large processing load of video streams, depending on the video resolution, especially for HD quality;

7. Scalable compression and rendering system that require minimal network resources when delivering multiple steams to a large number of users. At the same time, the system should provide maximum flexibility to the end user in selecting different view points.

All of these constraints have been researched in the last several years, but most systems only target a few aspects of the real-time video panorama problems. Most frequently, one system cannot be integrated with another system due to differences in design. Furthermore, problems such as blending multiple video streams have received less attention resulting in low quality video panoramas. In this thesis, we attempt to solve some of these challenges by designing and developing a system based on the amazing computing power of GPUs. We also aim at developing a deep understanding of the design and processing aspects of an end-to-end real-time video panorama application in order to develop guidelines for future development.

1.3 Contributions

The prime objective of this thesis was to produce a high quality real-time (30Hz) video panorama system that can be used to generate novel views within a field-of-view of the panorama using a number of video cameras with unknown intrinsic and extrinsic parameters. We also wanted to exploit the benefits of using GPUs by utilizing all the optimization provided by the massively parallel hardware. The main contributions of the thesis are:

1. Adaptation of automatic panorama registration algorithm based on global features for real-time video stitching;

2. GPU accelerated implementation of a high quality Bayer demosaicing filter;

3. Real-time color correction to enhance the picture quality;

4. Efficient mapping of Cartesian coordinates to spherical coordinates to achieve two orders of magnitude speedup while stitching 5 full HD video streams;

5. First real-time implementation of Laplacian pyramid blending algorithm for panorama videos on GPUs by reducing execution time to 30 ms while providing 30 times speedup over an optimized CPU based version;

6. An analysis of scalable transmission and rendering schemes for virtual view generation.

1.4 Experimental Setup

We used two commodity computers to perform benchmarking of the different algorithms developed in this thesis. The first computer is a laptop with mobile processors. This system is a good representation of a mid-performance computer. The compute processors of this machine are labeled as CPU1 and GPU1. Machine2 is a high-end desktop computer and most of the benchmarks are performed on this machine. This machine is a good representation of a current generation system and we use this machine as a baseline performance machine. Correspondingly, the compute processors of this machine are labeled as CPU2 and GPU2. Table 1.1 and Table 1.2 lists the detailed hardware and software specifications of these two machines.

Table 1.1: Experimental Machine1 Specifications

Machine1 Specifications	
CPU1	Intel(R) Core(TM) i7-4710HQ 2.50 GHz
CPU1 Cores / Threads	4 / 8
RAM	8GB DDR3 1600MHz
PCI-Express bus	PCI-Express version 3.0
Operating System	Microsoft (R) Windows (TM) 8.1 Pro
GPU1	NVIDIA (R) GeForce (TM) GTX 860M
CUDA Cores	640 (Maxwell)
Global Memory	2048MB GDDR5
Driver Version	353.06

Table 1.2: Experimental Machine2 Specifications

Machine2 Specifications	
CPU2	Intel(R) Core(TM) i7-4930K 3.40GHz
CPU2 Cores / Threads	6 / 12
RAM	32GB DDR3 1600MHz
PCI-Express bus	PCI-Express version 3.0
Operating System	Microsoft (R) Windows (TM) 8.1 Pro
2 x GPU2	2 x NVIDIA (R) GeForce (TM) Titan Black
CUDA Cores	2880 (Kepler)
Global Memory	6144MB GDDR5
Driver Version	353.06

1.5 Thesis Organization

The thesis is organized as following. Chapter 2 reviews the state-of-the-art of novel view-point generation methods and compares their algorithms and performance aspects. Chapter 3 presents our video capture system along with its configuration and supported features. We describe the high quality Bayer demosaicing algorithm along with its efficient implementation on GPUs using the CUDA language. We also explain the color correction algorithm and show the resulting improvements. Chapter 4 describes the panorama stitching process using a spherical panorama representation. We explain the steps we had to take to construct a computationally low cost stitching algorithm that is suitable for real-time processing while retaining automatic registration. We also describe CUDA implementation for the stitching algorithm along with performance benchmarks. Chapter 5 describes the process of accelerating the optimal seam selection and a high-quality image blending algorithm on the GPU using CUDA. It also reviews the design choices in order to keep the real-time performance of the system by using dual GPUs. The chapter ends with examples of panorama stitching results. Chapter 6 describes the process of rendering virtual views from the video panorama. We also review the different strategies to compress, transmit, and render the virtual views to the end users depending upon various application discussed at the end of the chapter. In Chapter 7, we conclude the thesis and discuss some possible future works and improvements of the system.

Chapter 2

Literature Review

Free view point video (FVV) and TV (FTV) systems have been extensively covered in the literature and can be broadly classified in two main categories. Systems that are based on geometry estimation are called *model-based rendering techniques* and systems using images only are called *image-based rendering techniques*. Most of the scientific literature that we have come across is based on image base rendering techniques since they are considered to produce photo-realistic virtual views and their computations are amiable to massively parallel processors, such as GPU. Moreover, we are interested in systems that can perform in real-time both for the processing and rendering of the virtual views. In this chapter, we will review the most significant works in free view-point video.

2.1 Image-based Rendering

Image-based rendering techniques are based on the application of the plenoptic function to create photo realistic representations of virtual views. The general form of a plenoptic function [2] is a 7D function $P(\theta, \phi, \gamma, V_x, V_y, V_z, t)$ that represents all the possible light-rays that goes to a point in space and time. Several methods were developed to simplify [52] the original plenoptic function into a 5D function by removing the time t and wavelength λ terms. Using this approximation, ray-space [23] methods were developed to describe how light-rays are distributed in space using only five parameters: the position (x, y, z) and direction (θ, ϕ). If the surface lie in a plane, i.e. z = 0, one can further reduce the dimensionality of the plenoptic

function to four. An example of this is the Lumigraph [28] which was introduced around the same time. The Lumigraph sampling is a 4D plenoptic function where the volume of interest is limited around the convex hull of a bounded object. Its competitor, the light field [43] method limits the plenoptic function to quantify and optimize the relationship between camera configuration and rendering quality. In both cases, compression of the function is performed by vector quantization which have been used to provide reasonable compression rate at fast decoding speeds such as the one used in the well known Quicktime VR application [15].

Shum et. al. [73] further reduce the dimensionality of the plenoptic function to a 3 dimensional representation by creating concentric mosaics. In a simple camera structure for concentric mosaics, a camera is mounted on a stand taking images at discrete time intervals. The light-rays are recorded as a function of the light beam angle θ and the location of the pixel (x, y). The light ray in 3 dimensions can be represented as $r(\theta, x, y)$. The lower dimensionality of the concentric mosaic simplifies the rendering process. However the virtual view generation is limited inside a planer circle. The radius of this circle is given as $R\sin(\frac{FOV}{2}))$ where R is the center of the camera path and FOV is the field of view of the camera. Instead of moving a single camera, multiple cameras at different rotation angles θ can be used to capture a complete mosaic at the same time, allowing to capture dynamic scenes. For best visual quality, the increments in the rotation angles should be small such that intermediate views between the camera rotations can be interpolated from the neighboring views. This concentric mosaic method suffers from vertical distortion which needs to be corrected using depth information. Zhang [87] proposed an algorithm to compress the data in concentric mosaics using motion compensation and residue coding. They also developed a decoder that deals only with a small field of view that needs to be rendered in real-time.

A simplification can be made in the concentric mosaics by fixing the position of the camera such that it can only move around its center axis from a single point while only the zoom or focus of the camera can be changed. This further reduces the dimensionality of the plenoptic function to 2D. In this scheme image mosaics or image panoramas (single center of projection) and can be described in rectilinear,

cylindrical or spherical representations. A light-ray in spherical panorama can be represented as a 2D plenoptic function $r(\theta, \phi)$. The primary purpose of creating image panoramas is to increase the field-of-view of the scene. These panoramas are constructed by stitching together images taken from multiple cameras or multiple viewpoint from a rotating camera. They allow the user to navigate around a part of the field-of-view by providing pan, tilt, and zoom functionalities during rendering. Initially, some systems were developed to capture panoramic images using hardware [53]. Nayar [56] introduced an omni-directional camera that capture live images in all directions from a single center-of-projection (COP).

If one could develop a camera system that can move from a single COP location to another, one could create mosaics with multiple center of projections (COPs) where each COP represents an individual panorama. Mosaics with multiple COPs are called *manifold mosaics* as they are indexed using multiple manifolds. Peleg et al. [64] creates stereo panoramas using a two manifold mosaics. They use a single rotating slit or a video camera by using vertical image strips to create left and right panoramas for each corresponding eye. Although they utilized a single camera, a set of two cameras with a constant distance between them can be used efficiently to produce stereo video panorama recording. In its general form, the same idea can be extended to create any number of COPs creating interesting applications, such as Zheng et al. [88] which uses continuous panoramic representation for matching and recognizing objects for a robotic traffic control system.

The panoramic video is a 3-dimensional extension of the panorama images created by adding the time dimension [15] [57]. Compared to regular video sequences, panorama videos record a larger field-of-view generated by stitching multiple single camera views at each time frame. Panoramic videos comes with additional computational complexity due to the recording and processing of several video streams, especially for high video resolutions. Initially, panorama video was used to create interactive 'virtual reality' videos of an environment that enables them to navigate static scenes. Flyabout [39] is a system that captures a 360° panoramic video while the camera is moved through a route or space of interest. In this way, several divergent paths can be created. In the same way, this system allows to create video-maps

where the camera is installed on a moving vehicle where a video-map is created as the vehicle moves around. The actual stitching of the video into a video panorama is performed off-line once all the videos are recorder. The users can navigate through the video-maps through a web based *videoMap* player that allows the user to select individual routes and renders the views in real-time while the users can move forward or back, pan, tilt, and zoom. A 4-cameras system capable of generating 360° panorama was marketed as *FlyCam*. In dynamic scenes, single camera systems cannot be used due to the loss of correspondence between multiple camera views during the rotation of a single camera. Hence, multi-camera [39] or omni-directional [56] cameras were developed. Synchronization between multiple cameras is necessary to avoid multiple stitching artifacts that can be handled either in the software or hardware. Fisheye lens [84] based spherical panoramas has been developed and multiple applications are available especially in photography and film.

In recent years, panoramic videos for capturing sport events using cameras installed in arenas has gained popularity. Yow et al. [86] were amongst the first ones to use video panoramas for digitizing sports events. The panorama is created by spatially and temporally superimposing multiple frames. However un-calibrated superimposition of images is bound to produce alignment errors and visual artifacts which needs to be post-processed making it hard to deliver the video stream in real-time. Bagadus system [31] provides post game sports analytic and annotations based on panoramic videos. They use a synchronized and calibrated array of cameras and propose a video capture, panorama stitching, and display system. Li et al. [44] creates 'motion panorama' by combining multiple images centered on an athlete action e.g. in diving. The background from multiple images is used to segment a foreground panorama. This foreground panorama is used for biometric analysis and recognition of the athlete actions.

2.2 Model-based Rendering

Model-based rendering methods first constructs a 3D model of an object or a scene by estimating the geometry of its surface. The geometry is estimated by solving the

correspondence problem between multiple images taken from different cameras. Once a 3D model is obtained, it is simple to create novel views using computer graphics rendering techniques. Most model-based methods found in the literature are based on calculating the 3D shape of an object surface from its silhouette [48]. In silhouette methods, multiple cameras extract the foreground image of an object using image segmentation, and then project these segmented images to the common 3D space to create a volume from the intersecting cones for each camera. The common volume for all of the cameras is called the visual hull [51]. Starck [75] improves method by first creating an initial mesh representation of the model of the scene surface and then refines this mesh using an optimization technique. These meshes are then merged after calculating correspondence between multiple views. Global multi-view stereo reconstruction techniques [76] often start with shape priors and utilizes correspondence and photo consistency information between all the views to construct a 3D model. Surface growing techniques [24] performs 3D reconstruction from selected seed points. In our application, we are interested in generating virtual views of an entire scene rather than a few extracted objects, making most of these techniques unfit for our purpose. As these techniques are based on 3D modeling, the resultant virtual views are not always photo realistic, which is a major requirement for our application. Furthermore, most of these approaches are very compute intensive and real-time implementations are generally not possible.

2.3 Image-based Rendering Performance

A key concern with image-based rendering methods are its real-time performance suitability. Based on design of the algorithm and the resultant execution time to complete the processing, image-based methods can be further classified into on-line and off-line methods.

2.3.1 Off-line Methods

The methods classified as *off-line* either includes some pre-processing steps that are required to be completed before the actual rendering of the virtual views begins,

or due to the nature of the algorithm along with the computational workload and implementation details, they have not been implemented in real-time systems using commodity processors such as CPUs or/and GPUs.

In Kanade's virtualized reality system [38], they use a multiple camera dome to digitize a moving scene inside the dome by computing its 3D structure. A user then selects a virtual viewpoint and prepares a video segment of the scene off-line.

Zitnick et al. [89] compute high quality virtual views using stereo based depth maps. They first perform color segmentation to get a description of regions with similar disparities and estimate an initial disparity map based on these segments. These disparity maps are then refined and a matting process is used to extract depth discontinuities to reduce view synthesis artifacts. Although the video view generation steps are less compute intensive, the algorithm needs to perform disparity map calculations off-line as they are computationally intensive.

Brown and Lowe [11] proposed a system for automatic panorama stitching using SIFT features to establish camera correspondences. They also utilize bundle adjustment to refine the transforms between images and then provide advanced features such as automatic straightening and multi-band blending [12] to create a panorama. In this system, they do not provide any separation of one-time and recurring components, making their system only applicable to static images. They also do not mention any GPU porting to improve the execution speed of their method.

Levin et al. [42] minimize the main source of artifacts in panorama images i.e. visible seam by using gradient cost functions over image derivatives. Essentially, they provide a global smoothing cost function and solve the iterative optimization problem by linear programming. They report an execution time of 2 minutes for an image of 200 x 300 pixels which is far from HD quality and real-time performance. Due to the iterative nature of the algorithm, it will be hard to adapt it for parallel computing using GPUs.

Jia and Tang [36] proposed a panorama stitching algorithm to improve the results of optimal seam selection and blending using sparse deformations. The algorithm solves three subproblems sequentially: i) optimal partition computation by detecting salient 1D features, ii) matching of these 1D features while minimizing

an energy function, and iii) propagation of deformations to the neighboring pixels of the detected 1D features to produce smooth transitions. The authors mention that the benefits of this approach over blending is visible when severe intensity discrepancies exist between the cameras, which is rare for calibrated multi-camera systems. Due to several compute intensive steps, the algorithm is not suitable for video panorama stitching.

Baudisch et al. [7] presents a system for *real-time panoramic photography* and was able to achieve a speed of 4 fps. Their definition of real-time is interactivity in image based panoramas, making their system not suitable for stitching high-resolution panoramic videos at 30 fps or more.

The systems presented in [67] [4] [20] perform real-time panorama stitching on mobile processors. Similar systems also exists for popular mobile operating systems such as iOS and Android. However these system either work for low resolution videos that are only suitable for mobile displays or perform the stitching at less than 30 frames per second while using only a single camera rotating around its axis.

2.3.2 On-line Methods

The on-line methods complete the execution of all processing steps in real-time or near real-time speeds. Some of these methods use a pipelining to distribute the computations into multiple steps that are processed separately by different machine producing a constant time lag. However a constant time lag of a few seconds is acceptable and common, especially for broadcast systems. Most of the real-time systems sacrifice image resolution, support lower number of concurrent camera streams, and avoid using any pre-processing or post-processing steps resulting in lower visual quality of the virtual reconstruction. Most of the techniques also utilizes massively parallel and general purpose GPGPUs using languages such as CUDA, OpenCL and GLSL to accelerate the processing and rendering operations. The speedup achieved is variable depending on the algorithm and implementation details. Since GPUs have minimal control unit features, its implementation often require several optimization steps that can have a large impact on the final speed.

Li et al. [45] combined the computation of photo hull with rendering using resolution of 320 x 240 pixels. They used a distributed setup of 4 PCs to achieve a speed 3 fps using an older generation GPU and should perform in real-time with a modern GPU. However the picture quality of the rendered objects is far from photo-realistic.

Yang et al. [85] proposed a distributed architecture where they utilized a network of machines. They used a 64 camera acquisition system that transfers the data to several computation machines connected in a client-server configuration. They implemented a distributed light-field rendering technique to generate virtual views. The system does not easily scale with the requirement of new virtual views and the complex system architecture hinders mass integration and usage.

Shegeda and Boulanger [71] accelerated the common plane sweep based virtual-view rendering algorithm on a multi-GPU system using CUDA and obtained real-time performance for two streams of full-HD video. However in order obtain the desired performance, they decided to sweep through every 8th depth plane. They use pixel interpolation for hole filling and produced good visual results with small leftover artifacts.

Stensland et al. [77] proposed a system that computes real-time panorama for sports and used soccer as a case study. They use GPUs to accelerate their processing and supports several camera streams. However their system does not support automatic camera configuration reducing the flexibility of the system. They are also using cylindrical panorama representation that supports a limited vertical field-of-view. The cameras acquisition system does not provide hardware based frame synchronization and feather blending [74] is utilized which may fail to provide high quality blended images in some lightening conditions.

Qamira [69] developed *Camargus* as a commercial system that utilizes up to 16 calibrated cameras to produce a large panorama videos of a soccer field. The system uses dedicated and expensive hardware for real-time video stitching and supports ultra-high definition resolutions. However due to the dedicated hardware, the system might be out of reach for most users. Since the system is offered as a commercial product, detailed specifications are not publicly available.

Video-Stitch [83] is a commercial product and offers a software development kit (SDK) to create 360° panoramas and circular fisheye domes. They also offer *Vahana VR* solution for live streaming of the panoramas. The system streams the complete panorama, taxing the limited commercial network resources leaving virtual view generation for the client set-top box. Hence although the system might work for small private networks, it might not be suitable for live internet protocol TV (IPTV) systems.

Ikena ISR [35] is an interesting application on how to stitch LiDAR images using super-resolution algorithms. The system offers features such as pan, tilt, rotate, and zoom. However the system is limited to the number of input streams and resolution of the images. The application specific algorithms are also not suitable for other areas such as FVV.

2.4 Conclusion

The online or real-time image based rendering methods based on 4D or 5D plenoptic functions provide a higher degree of freedom to the users in selecting their viewpoint. However, we have found that most of the implementation found in the literature produce low visual quality virtual views that are not acceptable for normal TV applications. Some systems require very complex and expensive infrastructure to compute even a few virtual views. In one of the recent papers by [71], the authors conclude with this statement *"...(in) on-line systems, 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"*. Especially for television and sports broadcasting applications, the quality of the virtual views is of key importance and the experience of users should not be sacrificed due to system limitations. Systems based on panoramic videos can only fulfill these requirements if the display quality matches HD broadcast quality. We believe that our proposed real-time image base panorama stitching is capable of addressing these challenges. The proposed method can deal with: automatic camera calibration, support for sev-

eral high definition video streams, color correction during video acquisition, and most importantly, advanced real-time image blending to produce smooth and visually pleasing panoramas. In the next chapters, we will describe a system that can provide real-time automatic high-quality spherical panoramas from at least 5 high definition 1080p video streams using a single, or at most dual GPUs.

Chapter 3

Image Acquisition System

The first step in a real-time panorama stitching system is to acquire multiple pixel-synchronized frames produced by cameras located around the scene. In this thesis, we constructed a custom setup to capture synchronized multi-videos using a prototype camera system. The synchronized multi-video frames are pre-processed to improve the image quality. Then these corrected streams are fused in real-time to create a high-quality panorama. The quality of the panorama critically dependents on the quality and performance of the image acquisition and in the algorithm used to perform frame fusion. In this chapter, we will describe the camera setup and the various steps to acquire and pre-process the video frames.

3.1 Herodion Camera System

We use the Herodion camera system [6] to capture pixel-synchronized frames necessary to create a real-time panorama . The cameras are built using the Altachrome A3372E3-4T sensor from AltaSens, Inc. which is able to deliver high quality, uncompressed video at 1080p (1920x1080) or 720p (1280x720) formats and at various user selectable frame rates (25, 30, 50 and 60 frames per second). The cameras also supports two color depths (8 or 12 effective bits per pixel) in Bayer format. The cameras can be connected directly to a grabber board through a standard 3M CAT6 UTP connector that carries captured video data, control signals, as well as power. The grabber board comes with a PCI-X bus to install it in a commodity desktops and servers. In our prototype system, we have attached five Herodion HD cameras

Figure 3.1: Herodion Cameras on Tripod

to a circular base pointing inwards to the center of the circle (Figure 3.1). The cameras are connected to a single machine that can handle the video stream in real-time. Additional cameras can be added for a larger system with 5 cameras per machine over a distributed network connected with a 100 Mb/s connection. The Herodion camera system bundles a hardware-based frame synchronization feature that ensures that each image pixels are digitized and delivered to the system at exactly the same time. This is performed at the camera electronic board and solves one of the most common sources of error such as alignment and ghosting artifacts found in many multi-view video systems.

3.1.1 Herodion API

Herodion provides a C++ API to communicate with the cameras. A single board of the Herodion camera system supports up to 6 camera streams concurrently. Individual camera streams are selected by setting its corresponding activation bit in the *status mask* registry. The API also provides a way to choose image acquisition options such as video size, frame rate, video format, and some special effects. We have shown some of the selected camera options in Table 3.1. Additionally, the

Figure 3.2: Herodion Camera Circuit Board

camera provides more than 50 modifiable registers to set up advanced options. The cameras are mounted on top of a tripod stand and attached with a metallic plate and Figure 3.1 shows an image of the system. The tripod allows rotation around its vertical axis. The cameras can also be disassembled from their metal jackets to reveal the camera's circuit board as shown in Figure 3.2 for a custom assembly or integration to an existing structure.

Table 3.1: Selected Options for Herodion Camera System

Camera Options	**Selected Value**
Status Mask	0x0000003F
HER_TIMEOUT	150
Special Effect	HER_EFFECT_DISJOINT
Video Size	HER_SIZE_1080p
Frame Rate	HER_FPS_30
Video Format	HER_FMT_08_08_08

3.2 High Quality Real-time Bayer Demosaicing on GPU

Most digital cameras use a single CCD or CMOS sensor covered with a colored filtered array (CFA) where the sensor captures only one color component per pixel. The colors are usually red, green and blue. The Bayer pattern is the most common CFA pattern where green filter is organized in the quincunx form whereas the blue and red filters are located at alternating pixel locations in the horizontal and vertical directions. The green component is twice denser than the red and blue components to match the sensitivity of the human visual system. Since each component only produces one color per pixel, the remaining colors at each pixel location must be reconstructed by interpolation from the discrete colors, this process is called *demosaicing*. Demosaicing algorithm are used to reconstruct the missing two color components at each pixel location. Figure 3.3 shows an image of the Bayer pattern. The Herodion camera system does not provide a hardware based Bayer demosaicing which is left to be handled by software allowing us to improve the demosaicing quality.

Several Bayer demosaicing algorithms exist [68] starting from basic linear interpolation algorithms such as nearest neighbours, bilinear interpolation, and cubic spline interpolation [40]. These algorithms have ideal computational density for real-time computation, however they suffer from interpolation artifacts, especially near the image edges or at high frequency area. There are numerous non-linear filters that takes gradient and edge information into consideration to improve color interpolation [46]. The hue based interpolation algorithm [16] maintains a hue of

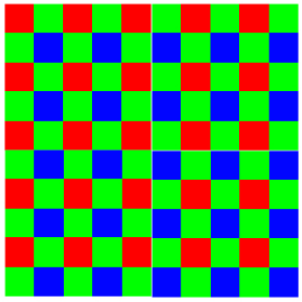


Figure 3.3: Real-time Bayer Pattern Pixel Demosaicing

colors to avoid sharp changes in the hue, except at the image edges. The algorithm maintains a gradual change in the hue and in turn reduces some color fringe artifacts. However the hue based approach operates in logarithmic space and needs to be linearized before. The method also fails to take into account the correlation between the color values that can further deteriorate image quality. There are some two-pass algorithms that perform median-based or gradient-based interpolation. Directional filtering using a posteriori decision [54] is one of the most recent two-pass algorithm and works by first estimating edge direction using the green channel and then utilizes this information to interpolate blue and red components along the edges and not across it, improving interpolation quality around the edges. However most of the two-pass and adaptive algorithms suffer from higher computational cost which makes them difficult to use in real-time implementation. Secondly, because of their non-linearity, these algorithms are not able to efficiently exploit the caching mechanism of parallel processors. In a real-time system, it is imperative to utilize an approach that consumes minimum compute resources. As Bayer demosaicing is the first part of the image acquisition phase, it is not a good idea to spend a considerable amount of computation time on this single step while leaving little resources for other processing steps. At the same time, the algorithm needs to produce accurate interpolation results that minimizes visible artifacts such as color fringes, zipping, and aliasing. These artifacts can be detrimental to later image processing steps and choosing a very basic interpolation algorithm will significantly diminish the quality of the panorama produced by the system. Since we are using the GPUs for all the processing the selected algorithm needs to map well to the parallel GPU architecture.

For our purpose, we selected a high quality interpolation algorithm [50] that provides good quality results with low computational complexity. The algorithm provides a 5.5 dB of PSNR improvement over bilinear interpolation, outperforming most linear interpolation methods and providing comparable results to two-pass algorithms. Furthermore, the algorithm uses 9 coefficients for the green channel and 11 for the R and B channels which is amongst the lowest in terms of computational cost. For our system, the algorithm was parallelized and implemented using CUDA

for real-time GPU processing.

3.2.1 Demosaicing Algorithm

The algorithm takes a gradient-based interpolation approach and is built on the criteria that the image edges have a much stronger luminance component than its chrominance component. As the human visual system is more sensitive to changes in the luminance than the changes in the chrominance [19], utilizing the luminance information at a pixel location can improve the interpolation results. Therefore, for interpolating a missing component value at a pixel, for example green component value at a red pixel, the algorithm takes into account the current value of the red pixel as well. In this case, a red value is calculated by bilinear interpolation of the nearest red values, and if the current red value is different from this interpolated red value, this means that there is a sharp change in luminance at that pixel. This luminance change is calculated for the red pixel and a portion of this value is added to the interpolated green value. Hence, if $\hat{G}_B(i,j)$ is the bilinear interpolated value of green component at a red pixel, the corrected value of green $\hat{G}(i,j)$ at this location is calculated by:

$$\hat{G}(i,j) = \hat{G}_B(i,j) + \alpha\Delta_R(i,j), \tag{3.1}$$

where the gradient $\Delta_R(i,j)$ is the bilinear interpolation of R at this location calculated as:

$$\Delta_R(i,j) = R(i,j) - \frac{1}{4}\sum_{m=-2}^{2}\sum_{n=-2}^{2} R(i+m, j+n). \tag{3.2}$$

By symmetry, similar calculations are applied to interpolate the value of a green pixel at the blue pixel location. Using the same reasoning, the expressions to calculate the value of R at the green and blue pixels respectively are given by:

$$\hat{R}(i,j) = \hat{R}_B(i,j) + \beta\Delta_G(i,j), \tag{3.3}$$

$$\hat{R}(i,j) = \hat{R}_B(i,j) + \gamma\Delta_B(i,j). \tag{3.4}$$

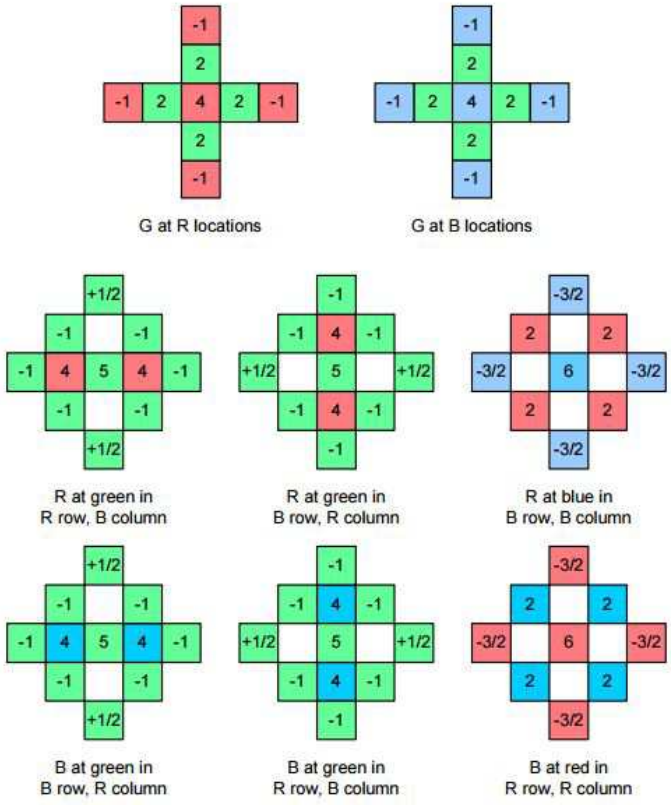


Figure 3.4: Filter coefficients for high-quality Bayer demosaicing found in [50]

The optimal value for the α, β, and γ coefficients are set at $\frac{1}{2}$, $\frac{5}{8}$ and $\frac{3}{4}$ receptively by minimizing the mean root square error between a Kodak image reference [30] and the digitized value. Using this model, the finite impulse response (FIR) filters obtained as illustrated in Figure 3.4 to be used for demosaicing.

3.2.2 GPU Demosaicing Implementation

At the start of processing, the raw Bayer images obtained from the cameras are directly transferred to the GPU memory using a DMA mechanism. The RAW images are single channel frames and therefore their sizes are three times smaller than uncompressed 3-channel color images.

The CUDA implementation of this algorithm works by dividing the calculations into separate threads that run in parallel. In the case of full HD images, this translates into 2 million threads that run concurrently. Since the computation of

(a) Raw Image (b) Color Image

Figure 3.5: High Quality Bayer Demosaicing

each pixel is independent, there is no need for synchronization between individual threads making the execution massively parallel.

Each thread first calculates if the current pixel $\boldsymbol{P}(x, y)$ is a red, green, or blue by solving the following calculations to find out mR, mB, and mG. For a given $\boldsymbol{P}(x, y)$ only one of these values can be equal to 1 representing that the color pixel is either red, blue, or green respectively:

$$mR = \frac{(1 + \cos(\pi * (x + 1))) * (1 + \cos(\pi * (y + 1)))}{4}, \tag{3.5}$$

$$mB = \frac{(1 - \cos(\pi * (x + 1))) * (1 - \cos(\pi * (y + 1)))}{4}, \tag{3.6}$$

$$mG = \frac{1 - \cos(\pi * (x + 1 + y + 1))}{2}. \tag{3.7}$$

Next, we calculate if the current row is a 'red row' or a 'blue row' by using Equations 3.5 through 3.7 for one previous and one following pixel to the current

Table 3.2: Execution Time for Bayer Demosaicing on GPU2

Image Size	**GPU1 (ms)**	**GPU2 (ms)**
960 x 640	0.73	0.35
1920 x 1080	1.77	0.84

pixel in the horizontal direction. Similarly, we find whether the current column is a 'blue column' or a 'red column' by computing pixel colors in the vertical direction of the current pixel. Once we have found these values, we can directly apply the demosaicing filters shown in Figure 3.4. These operations can be replaced with a simple pre-computed map for the filters to provide a small improvement in the execution time. Figure 3.5 shows an example of Bayer demosaicing result produced from an image taken from the Kodak image dataset [41].

The filters read between 9 and 11 adjacent values depending on the current pixel location. Since a single value can be in the adjacency of multiple pixels, this means that a single value is read multiple times. As an optimization, the entire image is bounded to a 2D texture cache that stores the pixel value on first call and provides cached access for subsequent reads, increasing the overall performance. To further speed-up the execution, we use fast math and CUDA optimized single precision trigonometric functions.

We benchmarked the GPU implementation on the *Machine2* of our experimental setup to gauge the execution time. The experiments were repeated for 10 subsequent frames to get an average estimate of the running time. Table 3.2 shows the execution time of our implementation on GPU2 for two different image sizes.

3.3 Macbeth Color Correction

Although the Herodion image sensors are sensitive to relative color differences, they have variations in representing absolute colors. This is a common problem even in expensive camera sensors due to small manufacturing differences. These variations are often not critical for single-camera applications. However in order to stitch multiple images together, small variations can cause artifacts and perceptual variations that cannot be ignored. Specifically for image panoramas that assume inter-camera color consistency, the variations in color can create visible seams. The color variations can also create differences in feature matching, especially for feature representations that take the color and brightness components into account. Some of these variations can be reduced by blending, however that alone might not be able to completely eliminate the color differences. This warrants using some color correction at the image acquisition phase to reduce the color variations.

Figure 3.6: Original Color from 6 camera streams

Figure 3.6 shows the original colors of the 6 separate camera streams. Notice that there is a large difference in the colors of the images. No two streams produce exactly same color and some of the streams have a large variations due to white, yellow, green, and pink tinges. We noticed that the cameras kept these color tinges over time. In our experiments, using these images for feature extraction as will be discussed in Chapter 4 produced variations in the measured features that in turn resulted in misalignments in the stitching results. Furthermore, the stitched panorama was full of overlapping colors producing visible seams that significantly reduced the quality of the panorama. Adding more cameras to the system is bound to further worsen the issue. Hence a robust color correction technique is needed that can

completely resolve these variations.

We use the Macbeth color chart [63] for color calibration of the images from all cameras. The standard Macbeth color chart is a color checker board with rectangular 4 x 6 array of color patches and covers a very large color space gamut which includes spectral components of light and dark skin, foliage, blue sky etc. as well as 6 steps grey scale to adjust for non-uniform lightening. The patches are square in shape and have a size of 25 mm x 25 mm. The color chart is used in a variety of applications and produces ideal color correction in our setup. Next, we explain the procedure for color correction along with its implementation on a GPU.

3.3.1 Color Correction GPU Implementation

We start by capturing a set of images of the Macbeth color chart from each camera. Ideally, the Macbeth pattern should be placed at the center of the field-of-view of the camera and the lighting should be neutral. These images are then converted to color images using the Bayer demosaicing algorithm discussed earlier. The system then detects the position of the color patches in each of these images. First, contours of the histogram equalized images are extracted. Next, bounding boxes are calculated for these contours and the center of the bounding boxes gives us the center of the color patch. Once we have located the centers, we take an average value of the surrounding area around the center to obtain the mean color value for each patch. The same process is repeated for the set of images to get an average color value of the color for each patch. This gives us a 4 x 6 matrix of measured color values represented by the matrix $\boldsymbol{A}$.

The real color values of all the patches are already known as described by the manufacturer defining matrix $\boldsymbol{B}$. Next, we use this matrix $\boldsymbol{B}$ of known color values and find a linear transformation $\boldsymbol{R}$ to approximate $\boldsymbol{A}$. The transformation matrix $\boldsymbol{R}$

Table 3.3: Herodion Camera Gain Changes

Register Name	**Default Value**	**Modified Value**
DP_COURSE_GAIN	0x8144	0x8166
DP_FINE_GAIN_EVEN	0x8144	0x8155
DP_FINE_GAIN_EVEN	0x8144	0x8155

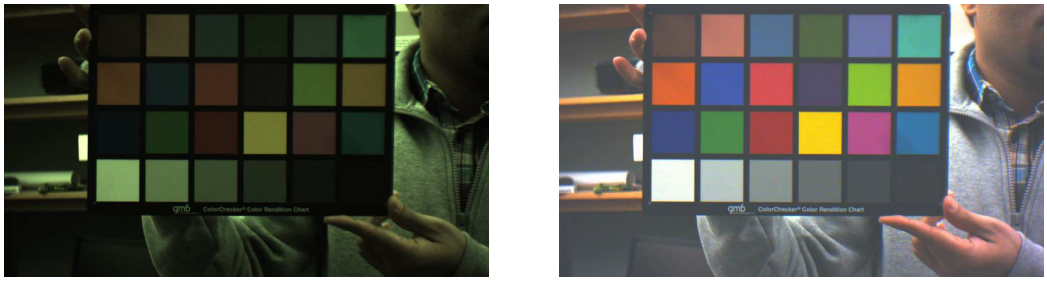
Table 3.4: Execution Time for Macbeth Color Correction

Image Size	GPU2 (ms)
960 x 640	0.25
1920 x 1080	0.63

can be determined using a least square solution using the generalized inverse:

$$\boldsymbol{R} = \left(\left(\boldsymbol{A}^{\boldsymbol{T}} * \boldsymbol{A}\right)^{-1}\right) * \left(\boldsymbol{A}^{\boldsymbol{T}} * \boldsymbol{B}\right))^{\boldsymbol{T}}. \tag{3.8}$$

In our setup, we calculate the transformation matrices once for a given setup of cameras and lightening conditions. These transformation are then saved in a transform database to be used later in the processing and also stored on disk for fast re-initialization. As the images already reside on the GPU memory after Bayer demosaicing, we use CUDA to apply this linear transform to the image pixels. In order to speed-up the application of the calculation, we store these matrices in the GPU's constant cache that gives the best possible read times. For the 5 streams in our system, we have 5 transformation matrices. These transformations are applied to the images right after the demosaicing step is completed. At the end of the transformation, the images are kept on the GPU memory to be processed by the next step in the processing pipeline.



(a) Original (b) Corrected Color

Figure 3.7: Improvement due to color correction

3.3.2 Experimental Results

In our experimental setup, the cameras were located in a lab environment and produced very dark images at the default settings, partially due to the ambient light-

ening and camera gain. In order to correct this, we increased the gain of the cameras by adjusting the *DP_COURSE_GAIN*, *DP_FINE_GAIN_EVEN* and *DP_FINE_GAIN_ODD* registers. The values of the default and modified gain values are shown in Table 3.3. This resulted in a small brightness improvement, however the issue of color variations still existed. Color correction improved both the brightness and color saturation of the images. Most importantly, the color of the images was equalized which proved to be very beneficial in later steps and in producing a visually pleasing panorama. The cameras are expected to produce good results at an adjusted gain in an outdoor environment. However applying color correction should provide improvements in all cases. In terms of performance, the color correction transformations takes a very small fraction of the total execution time on GPU2 as shown in Table 3.4.

Chapter 4

Panorama Generation

In this chapter, we will describe the stitching process involved in the panorama generation. Image stitching is the process of combining multiple images together creating a panorama. Panorama stitching is the process of re-projecting the stream of images from the cameras to the a panorama surface, such as a plane, a cylinder, or a sphere. What is required is an efficient panorama generation algorithm that combines all images without any misalignments, visible seams, color variations, or other visual artifacts. The complete process consists of several steps and in the next sections we will describe these steps in detail.

For image stitching, most of the algorithms can be categorized as either direct-methods or feature based methods [33] [78]. The main difference is how the camera or image transformations are derived. The direct approach generally starts with an initializing step, often requiring user intervention, and solves for pixel-to-pixel matches between images using a pixel matching model and an error metric. Several techniques are developed for direct-matching using hierarchical motion estimation [9], Fourier based alignment [61], incremental refinement [26], and parametric motion based methods [49]. Feature based matching algorithms are mostly used for automatic stitching and provides advanced benefits such as image ordering, gain compensation, and automatic straightening which requires minimal initialization or manual user intervention. Most of the features based algorithms use interest point detectors [70], feature matching [81] followed by geometrical registration. These algorithms are often followed by a global registration step such as bundle adjustment [80] that minimizes the re-projection error between the set of images to further

improve the alignment. However, due to their iterative nature, bundle adjustment, and refinement are often very compute intensive. Due to lower processing requirements after the initialization, most of the real-time video stitching systems utilize direct-methods to calculate correspondence between images produced from different cameras. Although direct-based methods work for fixed camera systems, they are less adaptive making it harder to extend them to mobile and portable applications. Quite simply, everything needs to be re-calibrated once the relative positions of cameras are changed.

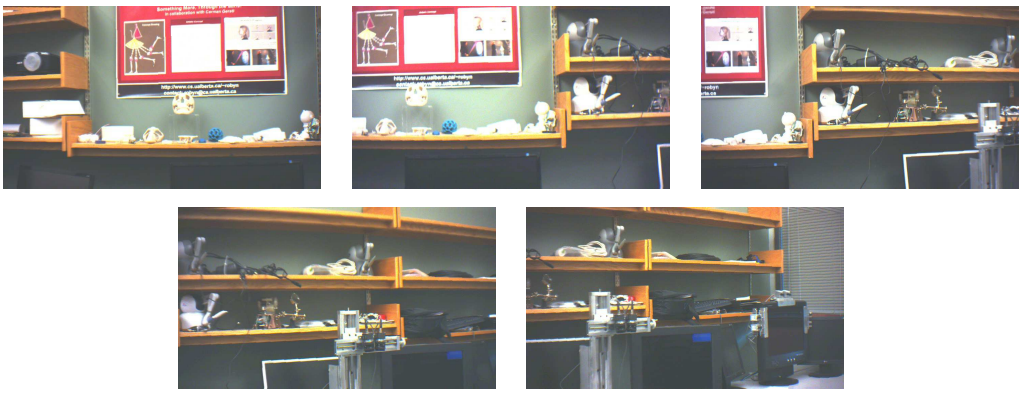


Figure 4.1: Sample images from our cameras system

There are several approaches to perform panorama stitching that one can find in literature. The majority of these algorithms only works for static images as the processing cost is too large for real-time video processing [27]. Some real-time implementations either work for very small image sizes [47] or support only two cameras [1]. We have not come across any published work that use an advanced image blending algorithm on commodity hardware. There is some support for creating spherical panoramas in OpenCV using GPUs. However it has already been reported to perform much slower than the requirement for a real-time video stitching system [32]. Most systems do not combine additional pre-processing and post-processing steps to enhance the quality of the panorama. Hence, we decided to process the compute intensive components of the stitching pipeline based on our requirement of stitching at least 5 full high-definition (FHD) video streams in real-time. We also included advanced pre-processing and post-processing steps, some of which have been been reserved only for image based stitching, to help producing

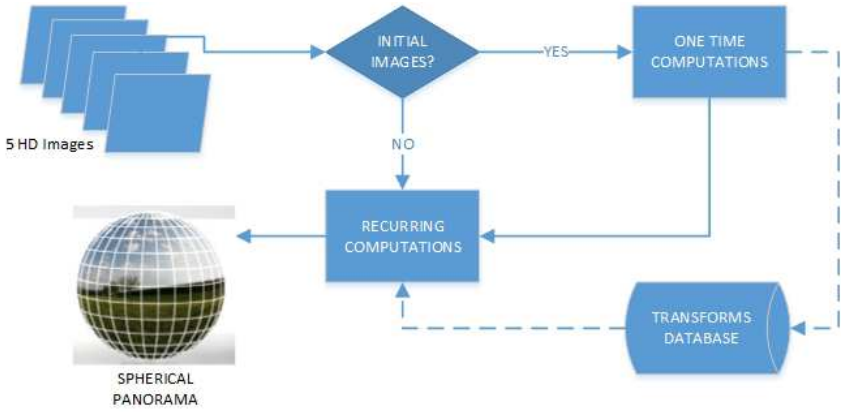


Figure 4.2: One-time and Recurring Components Decision

smooth and pleasant looking panoramas.

For our application, we have used a feature based approach to find the correspondence and transforms between images taken from multiple cameras. This approach has several advantages over the direct or static approaches. Most importantly, the feature based approach allows for automatic image stitching by removing the need for static components and user intervention for calculating the initial transforms between multiple cameras. This makes the system robust to changes to the relative positions of cameras due to vibrations or rearrangements, which is common in mobile and portable applications. Even for fixed systems, the cameras can lose calibration due to vibrations or other natural causes that can happen even when small changes in relative distance or orientation of cameras occur e.g. when cameras are installed on vehicles or an open environment. Continuous feature based approach comes with the additional computational cost of finding, matching, and refining the image features, making them unsuitable for video-stitching systems. To avoid this issue the real-time system can trigger a feature base registration process only at the start of processing or as required by the user. This eliminates the overhead of these computations in normal operation as we effectively merge the automation from feature registration based techniques with the performance benefits of direct methods.

Our panorama stitching pipeline is divided into two main categories; the first category consists of one-time steps that include feature detection and matching, ho-

Table 4.1: Distribution of One-Time and Recurring Computations

One-Time Steps	**Recurring Steps**
Color Correction Transformations	Color Correction Mapping
Feature Detection	Bayer Demosaicing
Feature Matching	Uploading Image to GPU
Homography Estimation	Spherical Coordinate Mapping
Bundle Adjustment	Laplacian Pyramid Blending
Spherical Maps Calculation	Virtual View Projection
Optimal Seam Selection	H.264 Video Encoding

mography or rotation estimation, bundle adjustment, stitching transformation calculations, and optimal seam selection. The second category consists of recurring steps that includes spherical re-projection, image blending, and compositing. From our experimental results, we show that the computational load for the recurring steps is suitable with real-time processing on a GPU. Table 4.1 lists the steps for both categories for the complete system. These are divided in such a way that only the steps that are absolutely necessary to be computed for every new set of frames become part of the recurring category and all the rest are one-time computations. Once calculated, the data obtained from one-time computations is kept in a transform database and used on the remaining frames in the video and is not updated unless the user requests an update of the transformations. While there is no change in camera configuration, the system effectively ”turns off” all the one-time computations. We also store this data to disk in a YML format which is loaded once the system has started and the camera configurations has not changed. This saves considerable amount of time during initialization. Figure 4.1 shows sample images from our video camera system.

4.1 Image Feature Detection and Matching

The image stitching begins with image features or interest point detection and leads up to the calculation of the transformation matrices between cameras. As noted earlier, these computations are only computed once or sparingly during the runtime of the system. Image features detection is the process of locating interest points in an image that uniquely describe different aspects of the image. We decided to use

Speed up Robust Features (SURF) [8] algorithm that meets our needs in terms of performance as well as matching quality. The SURF features also perform 3 times faster than Scale Invariant Feature Transform (SIFT) [11] and produced equally good features for our images in our experimentation. Since the features are calculated either at initialization or the user request, there is no need to further speed-up feature detection and any reasonable implementation should work fine with our system. We have used an open source CPU based implementation [60] for SURF feature detection. The algorithm works by first detecting key-points in each images. These key-points are stored in a vector that is later used for matching. The feature matching is performed using a Random Sampling and Consensus (RANSAC) algorithm [21] that takes a matching error threshold as argument. For our system, we used a value of 0.65 as the distance ratio which have shown to give good quality matching results in our experiments. Image features can produce registration errors partly due to limitations of the features and small variations in images taken from different cameras, producing false feature matches. To compensate for these issues, we first apply color correction to all images as described in Section 2.3 that reduces feature matching errors significantly. Next we perform global alignment through bundle adjustment to further reduce any misalignment in our transformations. From our experimental results, we have learned that these steps remove all noticeable alignment errors to compute camera transforms. A potential issue in transformation matrices is caused by registration error due to lapse in the feature detection or matching depending on the scene. This can result in sudden and completely skewed camera transformations that are too large for bundle adjustment to correct, leading to very bad stitching and alignment errors. This issue can be removed by taking a few consecutive images in a sequence and selecting the median value of calculated transformations.

4.2 Homography Estimation and Bundle Adjustment

The cameras in our system are arranged in such a way that they can be represented as the rotation of a single camera around its optical axis. Hence it is possible to

establish a homography relationship between the images produced by the cameras. A 3 parameters homography relationship between a pair of cameras (a, b) can be represented as :

$$\boldsymbol{H_{ab} = K_a R_a R_b^{-1} K_b^{-1}.} \tag{4.1}$$

Given a point $\boldsymbol{P}_a(u, v, 1)$ in an image from camera $\boldsymbol{C}_a$ and another point $\boldsymbol{P}_b(u, v, 1)$ in an image from camera $\boldsymbol{C}_b$ where $\boldsymbol{R}_a$ and $\boldsymbol{R}_b$ represents the rotation of the images across the canonical axis and $\boldsymbol{K}_a$ and $\boldsymbol{K}_b$ represents the intrinsic matrices of the two cameras, the two points are related by homography as:

$$\boldsymbol{P_b = H_{ab} P_a.} \tag{4.2}$$

Once the matching set of features are obtained in the previous step, they are used for the calculation of a homography relationship using a direct-linear transformation method [33]. Since we do not fully calibrate our cameras, but rather take approximation of camera intrinsic parameters from the captured images in order to keep image registration automatic, this leaves small calibration errors leading to inaccuracies in the calculated transformations. These errors can be significantly reduced by using a bundle adjustment algorithm that minimizes the re-projection error between captured images. We use a ray projection based bundle adjustment algorithm that works by calculating the shortest distance for a ray between the pixel position of the image feature and the center of the respective camera [34]. Since the purpose of our system is not to accelerate the one-time computations, we use an existing [60] implementation of a Homography based estimator and ray based bundle adjustment modules. Once we have calculated the refined homography transformations and camera intrinsic parameters, we are ready for stitching the images.

4.3 Spherical Panoramas

Rectilinear panoramas are most commonly used for displaying panoramic images. They are created by the projection of the images onto the surface of a plane and can be created by solving the homography relationships between the images. However

Table 4.2: Limits of different panorama representation methods

Panorama Representation	**Horizontal FOV Limits**	**Vertical FOV Limits**
Rectilinear Panoramas	120°	120°
Cylindrical Panoramas	360°	120°
Spherical Panoramas	360°	180°

increasing the field-of-view beyond 120° horizontally or vertically creates excessive warping as the pixels are strongly stretched towards the edges of the panorama. This greatly limits the number of cameras that can be used to create large panoramas. The cylindrical panoramas are the next best choice that offer a full 360° horizontal field of view. They work by projecting the images to the surface of a cylinder, where the images can be unrolled later to a planer image for viewing. Although this considerably improve the quality of the panoramas, the vertical field of view is still limited at around 120°, which is the same as rectilinear panoramas. This limits the vertical tilt of the cylindrical panorama. In our implementation we use a spherical panorama [17] representation that allows maximum expansion both horizontally and vertically. There is a little overhead in terms of computation due to calculating the projection for both longitude and latitude components in case of spherical panoramas, however, these computation are well handled in our system through the efficient GPU implementation. Table 4.2 compares the different panoramic representations with their horizontal and vertical field-of-view limits. Note that these are not hard limits because panoramas with larger field-of-views than the one listed are possible, however they will result in excessive warping as discussed earlier.

The spherical panoramas are created by warping the images to the external surface of the sphere. The spherical representation is designed so that the camera is placed at the center of the sphere. The warping or mapping of images can be accomplished by projecting every pixel of the image to the surface of a pre-defined sphere that converts them from Cartesian coordinates to spherical coordinates. For this conversion, we need to know beforehand the intrinsic parameters of the cameras. Since we are using similar cameras in our pipeline, the intrinsic parameters of the cameras are considered to be the same. If there are small variations in the

focal length of the cameras, one can deal with this by computing an average of the focal length for all cameras. In case a large variation is expected in one of the cameras, the median focal length can be taken as well. However in our experiments, taking either produces similar results. The focal length is used define the scale of the spherical panorama. For n cameras each with a focal length f_i, the scale of the panorama is calculated as:

$$scale = \frac{1}{2n} \sum_{i=0}^{n} f_i. \tag{4.3}$$

In order to align the optical axis of each cameras frame to the canonical axis, we first compensate for camera rotation $\boldsymbol{R}$ and intrinsic parameters $\boldsymbol{K}$ of each input image with pixel locations $\boldsymbol{P}(x, y)$ by creating a new image with pixels $\boldsymbol{P'}(x', y')$. This transform is described as:

$$\boldsymbol{P' = RK^{-1}P.} \tag{4.4}$$

In our implementation, we do not need to calculate a separate image, rather these transform can be applied to every pixel before the spherical mapping is performed. Hence, the rotation matrices are used directly rather than solving for the complete homography.

The 3D Cartesian coordinates in which the images lie can be is expressed as (x, y, z). Whereas, the spherical coordinates are represented as (r, θ, ϕ) where (θ, ϕ) are the coordinates of the image pixels on the surface of the sphere and r is the radius of the sphere. Based on Figure 4.3, one can derive the following relationship between spherical and Cartesian coordinates:

$$r = \sqrt{x^2 + y^2 + z^2}, \tag{4.5}$$

$$\theta = \arctan\left(\frac{x}{z}\right), \tag{4.6}$$

$$\phi = \pi - \arccos\left(\frac{y}{r}\right). \tag{4.7}$$

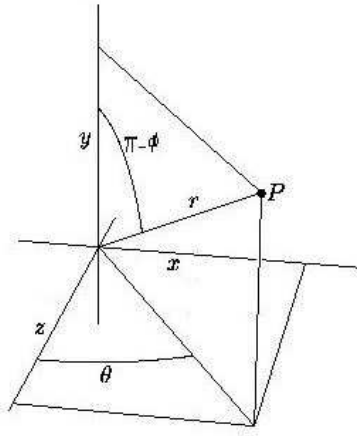


Figure 4.3: Cartesian to Spherical Relation

Hence each pixel of the image can be warped to the spherical coordinates by solving these equations for the pixel coordinate. To stitch multiple images, the same procedure is applied to all the images. As the images in spherical coordinates are distorted, in order to recover the original images, one needs to take an inverse of these operations. The inverse operations after the perspective division are described as:

$$z = \cos\theta \sin\big(\pi - \phi\big), \tag{4.8}$$

$$y = \frac{\cos\big(\pi - \phi\big)}{z}, \tag{4.9}$$

$$x = \frac{\sin\theta\cos\phi}{z}. \tag{4.10}$$

Sometimes it is necessary to perform correction in the projections and remove errors due to left over alignment errors. Because of the bundle adjustment step, we remove any misalignments and notice minimal residual projection errors.

4.4 Implementation Details

The algorithm steps described from Equation 4.5 through Equation 4.7 are applied to every pixel of each video frame in order to convert the input into a stitched

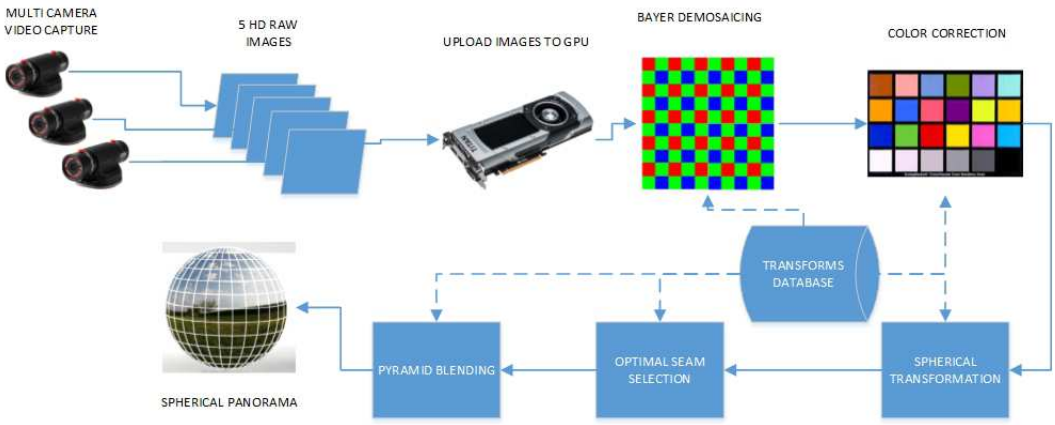


Figure 4.4: Pipeline of the Recurring Steps

panorama. For a stream of 5 color images with a resolution of 1920 x 1080 pixels, the operations are repeated over 31 million times, translating to 933 million operations per second for a video stream running at 30 frames per second (FPS). Especially with several trigonometric operations for each pixel, the total calculations translate into very large number of compute intensive operations, making it prohibitive to be computed in a real-time system. Subsequently, we made some changes to support this algorithm for real-time video.

Secondly, when the pixels are converted to spherical coordinates, it is possible that some values in the resultant panorama remain unfilled as corresponding values are not represented in the input image. This can produce 'holes' that degrades the quality of the panorama. In order to resolve this issue, we first find the bounds of the spherical image calculated by projecting each pixel of the compensated image to the spherical coordinates according to the transforms described in Equation 4.5 to Equation 4.7. Once we have the spherical image bounds, we calculate a backwards projection of each (θ, ϕ) pair to the original. These backwards projections are calculated by the inverse operations shown from Equation 4.8 through Equation 4.10.

4.4.1 Recurring step

In the video panorama stitching, as long as the relative camera positions remain constant, the transformation values from the Cartesian coordinates to the spherical

coordinates remains the same, and it is possible to make a transform map or conversion table for these transformations. This forms the basis of the recurring step in our pipeline where we make one map for each image as we calculate the backwards projection from spherical coordinates to Cartesian at the start of the execution and save these maps in the transform database for the subsequent images. These maps can be updated through user request or at regular interval. The conversion table acts as a look-up table and each pixel from the video frames can be directly converted to spherical coordinates without the need to solve trigonometric equations over and over again. The same operations are applied to the valid masks of the images and collectively these tables save considerable processing time. We first created a CPU based implementation for the conversion maps to gauge performance and verify functionality. Since the same conversions values are valid for all three channels, we used vector types to read three values from memory in a single step and then apply the same conversion to all of them. In our experiments, the single threaded CPU performed at around 5 FPS which renders the system useless for our requirement. However if a CPU is the absolute requirement, a multi-threaded version in a distributed setup without advanced blending might be feasible. To speed-up the processing of the recurring steps, we implemented a complete pipeline to run on GPUs using CUDA. Below we list some of the optimizations that we apply to the GPU processing:

- **Copy-Execution Overlap:** A large percentage of the processing time is consumed in transferring data to the GPU memory and back to the host memory after the processing has completed. The data transfer times for typical image sizes are presented in Table 4.3. To minimize the effect of transfer time, we use copy and execution overlap feature in CUDA. We create separate streams for data upload and processing and execute these streams concurrently for all the images. Consequently, when the data from one stream is being copied to the GPU, another stream is being processed by the GPU simultaneously.

- **Texture Memory Transformation Maps:** For a given configuration of cameras, the transformations applied to every pixel remains the same and can be reused

Table 4.3: Image Transfer times for common 3-channel image sizes

Image Size	Transfer Time (ms)
960 x 640	0.74
1280 x 720	0.92
1920 x 1080	2.13
4196 x 2160	8.05

in the recurring component. Hence we save the transform map to a file, as we do with the camera parameters and homography information. This table is loaded once at the start of the program and remains in memory throughout the run of the program. We bind these maps to the texture memory of the GPU in order to cache the access of these maps. We observe around 20% performance increase due to texture caching in our application.

- **GPU Occupancy and Memory Access:** In order to maximize the occupancy of the available GPU resources, we eliminated branching from the code. We also created as many GPU threads as the number of pixels in the image i.e. 2 million threads for each image. Furthermore, we used 2D grids and blocks as well as 2D surfaces to store the images. This helps us to get maximum GPU utilization and occupancy. Furthermore, the global memory accesses were fully coalesced to provide maximum memory bandwidth. As with the CPU based implementation, we used vector types that alone resulted in 40% performance increase in our experiments.

4.4.2 Experimental Results

In order to benchmark our GPU implementation and to compare it against CPU version, we used sample images from our camera stream at two different resolutions of 960 x 640 pixels and 1920 x 1080 pixels to estimate the performance trends due to image sizes. The images have variable overlap ranging between 10% and 30%. A decent amount of overlap is necessary for the images to establish feature correspondence. We also repeated the experiments after changing the relative positions of the cameras to gauge the effectiveness of automatic stitching and obtained the

same results. The computed execution times are average for 10 subsequent runs on the same set of images for both CPU and GPU.

Table 4.4: Panorama Warping Benchmarks

Image Resolution	**CPU1 Runtime (ms)**	**CPU2 Runtime (ms)**	**GPU1 Runtime (ms)**	**GPU2 Runtime (ms)**
960 x 640	54.41	51.7185	0.922	0.315
1920 x 1080	233.58	223.39	2.57	0.965

Figure 4.5: Relative Speedup of Panorama Warping

According to these benchmarks, the GPU completes the processing for 5 frames under a millisecond. The CPU version is computing the exact same mapping functionality but is hundreds of times slower. Without including the execution time for the other related components, the stitching module for 5 HD images is running at around 1000 frames per seconds (FPS). Even for the mobile GPU, the processing completes with real-time constraints that shows that the system can perform well even on commodity hardware. In actual usage, the panorama stitching is a only one part of the complete pipeline, so the actual frame rate will be based on the collective execution times of all the components. To be fair, the CPU implementation can benefit from multi-threaded version that can increase its performance by 3 to 5 times. Similarly, the GPU processing can benefit from dual GPUs available in our test system. However due to adequate performance and in the interest of keeping the

design simpler, we chose to compute the spherical mapping operation on a single GPU. Even after adding the image acquisition time, the GPU2 completes processing in less than 15 ms which is still tens of times faster than the CPU based version. It also leaves ample time to complete the processing of other components on the same machine. In our system, we spend the extra time at reading the images, performing the Bayer demosaicing followed by color correction i.e. in practice, most the processing except blending can be computed on the same machine. Once the panorama is constructed, it is transferred to the next module for the post-processing steps of optimal seam selection and image blending. These steps are discussed in detail in the next chapter.

Chapter 5

Seam Selection and Blending

The panorama obtained at the end of the last chapter is a stitched representation of the original image, but suffer from several artifacts that needs to be corrected before the panorama is ready to be delivered. Existing panorama systems utilizes a variable number of post processing steps to improve their quality. Improved seam selection followed by image blending are most frequently used in image based or video based systems. The difference between image and video based systems is that most of the video stitching systems rely on simpler techniques to reduce computational cost, especially for blending the images. In this chapter, we explain the post-processing steps we used in detail and list the visual and compute performance improvements due to our work.

5.1 Seam Selection

Overlapping regions between multiple images can produce visible seams during the compositing of warped images to the final panorama that can cause artifacts such as ghosting and cut out areas. It is therefore imperative to find and select the seam lines so that only one image contribute to each pixel in the composited panorama. Additionally, the chance of selecting the best seam line becomes greater if the seam line lies near matching image regions, thereby minimizing the variance across the selected seam. Figure 5.1 (b) shows the improvements due to seam selection against the original image in Figure 5.1 (a) with ghosting and blurring artifacts. In both the cases, the image was later blended together using the same algorithm. Notice how

the blue marker shows ghosting and blending artifacts due to wrong placement of seams that has been corrected by appropriate seam selection.

Several ways are proposed for improving seam selection in the literature and some of those methods are translated for GPU processing [1]. One approach is based on label assignment that minimizing the sum of two objective functions [3]. The first objective function is called *image objective* that selects the best contenders for the final composite image by using an automated maximum and minimum likelihood criteria. The second objective function *seam objective* penalizes the pixel locations that are not suitable for seam placement. The aim is to minimize the sum of these two objective functions, represented as *Markov Random Field Energy* using one of the several ways proposed by the authors. Although the algorithm culminates in very good seam placement, its complexity and computational requirement makes it unsuitable for a real-time system. Another algorithm proposes to find overlapping images and then locate the regions where there is a difference in the images [82]. These regions are referred to as regions of difference (ROD). A pair of overlapping pixels is termed as an edge, whereas individual pixels are referred to as a vertices. In order to remove all overlap, vertices are removed until no edges remain between the images. The vertices are removed using a vertex cover algorithm. The images are eventually blended using feathering [74] to get the stitched image. This approach removes artifacts due to moving objects between the two images. However we do not have this issue in our system as our camera system is pixel synchronized and captures images exactly at the same time. Hence a simpler and less compute intensive approach is more suitable for our case.

In our system, we use Vornoi diagrams [5] to select the seams lines in overlapping region of the images and the approach produces good results at very high speeds. The bounds of each image mask is compared with other image masks to select pairs of images that have overlapping regions. The size of the overlapping region can be found by taking a difference of the mask vertices under consideration. Once an overlapping region is calculated, the next step is to take the distance transform [10] of the pixels in that region with respect to centers of respective images. The distance for overlapping regions in the masks are compared and the smaller

(a) Ghosting artifacts (b) Ghosting artifacts removed

Figure 5.1: Improvement of artifacts due to bad seam selection

(a) Incorrect seams (b) Improved seams

Figure 5.2: Removal of hard edges by improved seam selection

of the two values for each overlapping pixel is selected while the other is set to zero. This way only one pixel contributes to the previously overlapping region and the resultant distribution of pixels around image centers is a Vornoi diagram. The Vornoi diagrams based seam selection gives greater weightage to the central area of an image instead of the edges, thereby reducing the effects of radial distortion and at the same time ensures less variation in the scene due to dynamic seam generation. Figure 5.2 (a) shows an image without seam selection. Notice the hard edges and corners of the overlapping frames. Figure 5.2 (b) shows the same image after appropriate seam line has been selected.

In our system, Vornoi diagram based seam selection method coupled with the an advanced blending algorithm, as will be explained in the next section, helps remove ghosting and other artifacts created by visible seams.

Figure 5.3: Improved Seam selection execution time comparison

5.1.1 Implementation and Experimental Results

The key aspect for the Vornoi diagram based seam selection algorithm is the calculation of the distance transform of the overlapping images. Through our experiments, we learned that the computation is not compute intensive and has little impact on the overall execution time of the system. Furthermore, seam placement is computed only for the masks and does not add to the processing time of the recurring steps. We used a CPU based distance transform implementation and copy the masks for small overlapping regions back to the CPU. In our experiments, the size of the overlapping area is around 20 times smaller than the sizes of the images, hence there is minimal overhead of moving these masks back and forth to the GPU. Given small overlapping regions, this operation is computed with real-time constraints. To obtain maximum performance, we create as many CPU threads as the number of overlapping images, using POSIX threads [66] to significantly improve the performance. In our experiments for a set of 5 input images, there were a total of 8 overlapping regions and correspondingly, we create 8 separate CPU threads that work in parallel. This number of overlapping images and threads can change depending on the relative position of the cameras. Figure 5.3 shows the total execution time to find seams in the overlapping regions of 5 images. We repeated the experiments for two different image sizes and benchmarked the module on *Machine2*.

5.2 Image Blending and Compositing

The final step in the panorama generation process is to blend the images together to form a smooth, single image representation. Blending the images together remove artifacts due to camera exposure differences, vignetting and small misalignment errors, making it an essential part of the stitching pipeline. Performing image blending also helps the system to achieve good results even without the pre-processing step of color calibration. Image blending is one of the steps that is often overlooked in real-time systems, partially due to high computational requirements of better performing algorithms. In real-time systems, blending is often performed by average or median filtering [27] or most commonly by feathering [74]. We propose a high-quality real-time image blending algorithm by efficient implementation of multi-resolution splines or Laplacian pyramid blending [13] on modern GPUs. The algorithm is sufficient to remove any visible color variations, small misalignments and other minor artifacts in our setup, producing a smooth and pleasant looking panorama without any visible degradation in the sharpness of the images.

5.2.1 Laplacian Pyramid Blending Algorithm

We start by explaining the weighted average based approach for image blending, often referred to as feathering, and build up from it to explain the Laplacian blending algorithm. Feather blending works by creating a weighted filter for each image using a distance map, where the weights are set to maximum at the center of the filter and decrease monotonically towards the edges. The images are then multiplied with their corresponding weight filters and added together to create a mosaic. If L represents the size of the blending region and $\boldsymbol{L}(x, y)$ is a pixel in that region, a pixel $\boldsymbol{A}(x, y)$ of image A can be blended with a pixel $\boldsymbol{B}(x, y)$ of image B to produce the corresponding pixel value $\boldsymbol{R}(x, y)$ in the resultant image using the following expression:

$$\boldsymbol{R}(x, y) = \big(\frac{\boldsymbol{L}(x, y)}{L}\big) \times \boldsymbol{A}(x, y) + \big(1 - \frac{\boldsymbol{L}(x, y)}{L}\big) \times \boldsymbol{B}(x, y) \tag{5.1}$$

However the feathering approach for the entire image leaves some visible arti-

Figure 5.4: Feather blending

facts, primarily because it treats the entire image as a single frequency band. This can produce a small step in the overlapping region along with blurring. Depending on the length of the overlap, feathering can also lead to visible edges and in some cases double exposure and ghosting artifacts. The results of feathering in our system can be seen in Figure 5.4, where the algorithm leaves much to be improved in order to produce a smooth and natural looking mosaic. In our experiments, feathering produces wide seam lines that are visible throughout the image. The width of this line is equivalent to the size of the blending region L divided between the two images and increasing this region diminishing the line at the cost of blurring the image further and producing double exposures. Secondly, there is minimal blending of color and the variations of color across the seam are very noticeable. Some of these issues abate when tested with a better seam selection algorithm, but cannot be completely eliminated. These artifacts significantly affect the viewing experience when repeated throughout the video stream and led us to consider a better blending approach to obtain a smoother panorama.

An attractive solution to overcome the limitations of feather blending was proposed by Burt and Adelson [13] where the authors identify that there is no single value of the blending region L that can diffuse the colors around the seam line and avoid blurring and double exposure artifacts at the same time. They propose the Laplacian blending algorithm to solve this problem by decomposing the images

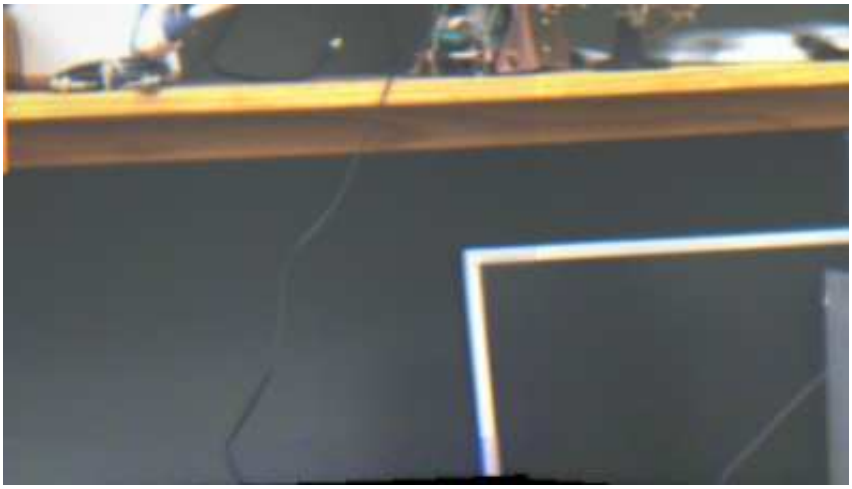


Figure 5.5: Laplacian Pyramid Blending

into several frequency bands and then blending each band separately. Next, we explain the process of blending multiple bands of the image separately.

The first step is to construct a Laplacian pyramid [12] of the images where each level of the pyramid represents one octave of the bandwidth. We begin with calculating a Gaussian Pyramid where each image is first convolved with a small weighing filter creating a low-pass image. For the weighing filter, a generating kernel of size 5x5 is used and in our case we use a 5 tap filter represented as: $\frac{1}{16}[1\,4\,6\,4\,1]$. Hence each image at level $\boldsymbol{A}_i$ is reduced to an image at level $\boldsymbol{A}_{i+1}$ where each value in $\boldsymbol{A}_{i+1}$ is a weighted average of a 5 x 5 window in level $\boldsymbol{A}_i$. For a weight filter $\boldsymbol{W}$, this reduce operation can be represented as:

$$\boldsymbol{A}_{i+1}(x,y) = \sum_{j=-2}^{2}\sum_{i=-2}^{2} \boldsymbol{W}(j,k)\boldsymbol{A}_i(2x+j, 2y+k) \tag{5.2}$$

To obtain a Laplacian pyramid, we subtract from each level of the Gaussian pyramid the next lowest level in the pyramid, except for the last level that does not have a lower level. The next step is to perform feathering of each band in the overlapping images which is similar to the feather blending of the images. However the feathering is performed for each band of the image rather than the entire image. At first, the mask of valid pixels for each image is used to create a Gaussian pyramid of weights by down sampling the masks while the number of levels is kept the same as the number of levels in the Laplacian pyramid of the images. These masks are

then multiplied with corresponding levels of the Laplacian pyramids. In case of a 3 channel image, the same masks are used for each channel. Once all the images are multiplied with the weights, a normalization is performed with the sum of the weights.

The last step of this algorithm is to collapse the resultant pyramid by subsequent expansion followed by addition of all the layers in this normalized Laplacian pyramid leading to the final panorama. For a Laplacian pyramid $\boldsymbol{R}$ obtained after the feathering of all the images on the compositing surface, where l is the level of pyramid, we can represent the expansion operation to create final panorama as:

$$\boldsymbol{R}_l(x,y) = 4 \sum_{j=-2}^{2} \sum_{k=-2}^{2} \boldsymbol{R}_{l-1}\left(\frac{x+j}{2}, \frac{y+k}{2}\right) \tag{5.3}$$

As with the previous steps, each channel of each image is treated separately. The outcome of this step is a significantly improved mosaic without any noticeably color differences and seams. The algorithm is very effective in removing noise for most environmental conditions. The result of Laplacian blending on a panorama from our pipeline is shown in 5.5. Due to high computation requirements, the Laplacian blending algorithm has so far only been used for blending images and applying it for real-time video has been deemed impractical. Shete and Bose [72] implemented Laplacian blending and report a total execution time for 3.38 seconds for a pair of 2K image running on Nvidia Quadro FX 4600 GPU which is far from real-time speeds. OpenCV provides part of the functionality for Laplacian blending on CUDA, but their execution time is also much higher to be suitable for real-time systems. In the next section, we describe implementation details and steps for efficient Laplacian blending on GPUs that performs several times faster as compared to a CPU based implementation. Figure 5.6 shows the complete panorama in our system after performing Laplacian pyramid blending.

5.3 GPU Implementation

We programmed the Laplacian blending using C/C++ and CUDA. The first design decision for this implementation was to eliminate any needless data transfers be-

tween the host and the GPU. The data transfers can be very expensive especially for large images, such as in our case. Furthermore, the input images already reside on the GPU memory as passed from the previous module, therefore avoiding the need to transfer data to the GPU. We also do not need to transfer the final panorama back to the host since we can encode or display the images directly from the GPU memory. This saves a considerable amount of time needed for the real-time application. Table 4.3 shows the data transfer times for transferring single 8-bit 3-channel image of common sizes from host to the GPU without using pinned memory over PCI Express 3.0 x16 bus.

Secondly, we use on-chip data caches on the GPU sparingly to store data that is used frequently and increase the performance by not accessing global memory for every read operation. Each image is binded to a texture before an operation, improving the reading speed of all the images by a large margin. As long as the camera transformations remain the same, we also keep our weighting filters in the GPU memory to be used for subsequent frames, so they are generated once and used throughout the run of the program. One of most frequent operations is the generation of fine to coarse and coarse to fine pyramids, and we optimize these operations by storing the repeated values to be used in the shared memory. The shared memory is used to cache all the pixel values that are accessed by all threads in a block, which gives fast access to those values when used by neighboring pixels. The operations are also highly parallel and a separate thread is created for each pixel of the resultant image. The resultant pyramid generation operation performs several times faster than the corresponding CPU based method as shown in Figure 5.8.

Another frequent set of operations is the multiplication of weights with every channel in all levels of pyramids. In our system with five full HD 3-channel input images, we use a total of 5 bands resulting in 6 level pyramids. This aggregate to 90 image multiplications, while the image sizes range from full HD to much smaller images. Even a small processing time for each operation can aggregate to a large execution time. As the same weight map is applied to each channel of the image, we save some time here by reading the weight map once for an entire 3-channel image. Secondly, we use vector types to read larger chunks of images that alone resulted

in almost 3 times performance increment. We also made certain that all memory accesses are coalesced to utilize maximum available memory bandwidth. These optimizations result in a significant acceleration of multiplications on the GPUs.

5.3.1 Multi-GPU Implementation

We developed a multi-GPU variant for Laplacian blending to utilize maximum available resources by exploiting data level parallelism. The process of creating Laplacian pyramids and then multiplying the pyramids with their respective weight matrices is independent for every image. We have created an API to detect the number of GPUs available in the system and automatically distribute equal number of images to each GPU for processing. In the *Machine2* from our experimental setup, we used two GPUs to process a total of 5 images. Hence one GPU gets 3 images and the other processes the remaining 2 images. We expect to increase the system capacity to stitching between 3 to 5 additional images with the addition of every new GPU to the system. Once the GPUs have generated a feathered Laplacian pyramid per image, all of the pyramids are gathered to the default GPU using peer-to-peer data transfer facility in CUDA [18]. These images are then normalized and collapsed to a single panorama by a single GPU. This panorama is now ready for display, encoding or transfer to the next module (Figure 5.6). The complete processing, including the normalization and pyramid collapse steps can be extended to a distributed array of servers, each with a single or multiple GPUs. This will create a truly distributed architecture for the processing in order to support a large number of cameras. We have not added the distributed processing support yet, but this can be a good future extension to the system.

5.3.2 Experimental Results

In this section, we show the outcome of applying Laplacian blending to generate the composite Panorama. We also list our performance benchmarks and compare them with OpenCV based CPU side implementation to show the speed up achieved by using the GPU based Laplacian blending. The execution time of various components of our Laplacian blending module is shown in Figure 5.5. Notice that both

Figure 5.6: Laplacian Pyramid Blended Panorama

the single and dual GPU versions perform several times faster than the CPU version with the total speed up between 23X and 30X, depending on image resolution.

Table 5.2 and Table 5.3 shows the execution time for various components of Pyramid blending for two different images resolutions. Notice that the Pyramid Normalization and Pyramid Collapse are only valid for multiple images. Figure 5.7 gives a graphical description of the percentage time spent on the execution of each component.

CPU and GPU Execution Time Comparison

Next we compare the execution time of all steps in Laplacian blending between CPU and GPU for full HD images. The largest speed up is provided for the Laplacian feathering operation when compared to the GPU implementation. The complete performance comparison for various devices is given in Table 5.4 and Table 5.5. The dual GPU variants also provides a decent speed up as compared to the single GPU version. Figure 5.8 shows a graphical representation of the speedup achieved.

Table 5.1: Total execution time for blending 5 full HD images

Device	**Total Time (ms)**
CPU2	881
Single GPU2	38.16
Dual GPU2	30.19

Table 5.2: Total execution time (ms) using single GPU2 for single image

Laplacian Blending Steps (Single Image)			
Image Resolution	**Gaussian Pyr**	**Gaussian to Laplacian Pyr**	**Laplacian Feathering**
960 x 640	0.72	1.83	0.66
1920 x 1080	1.44	3.09	1.06

Table 5.3: Total execution time (ms) using single GPU2 for five images

Laplacian Blending Steps (5 Images)					
Image Resolution	**Gaussian Pyr**	**Gaussian to Laplacian Pyr**	**Laplacian Feathering**	**Pyr Norm**	**Pyr Collapse**
960 x 640	3.6	9.8	3.31	0.35	2.88
1920 x 1080	7.2	15.45	5.20	1.13	5.18

Table 5.4: Laplacian Blending Steps (Single Image) 1920 x 1080

Laplacian Blending Comparison (Single Image)			
Device	**Gaussian Pyr**	**Gaussian to Laplacian Pyr**	**Laplacian Feathering**
CPU2	18.47	42.38	67.33
GPU2	1.44	3.09	1.06

Table 5.5: Laplacian Blending Steps (5 Image) 1920 x 1080

	Laplacian Blending Comparison (5 Images)				
Image Resolution	**Gaussian Pyr**	**Gaussian to Laplacian Pyr**	**Laplacian Feathering**	**Pyr Norm**	**Pyr Collapse**
CPU2	93.35	211.85	345.49	15	98
Single GPU2	7.2	15.45	5.20	1.13	5.18
Dual GPU2	5.4	11.58	3.9	1.13	5.18

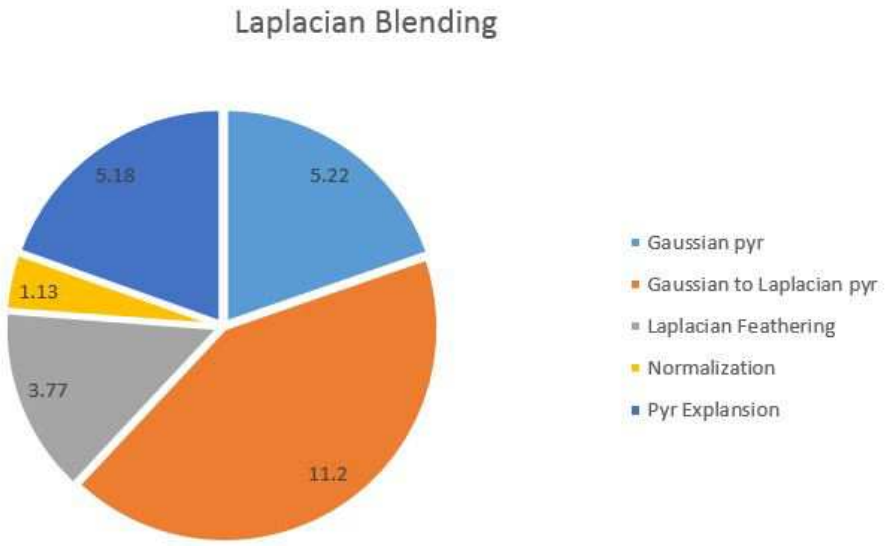


Figure 5.7: Execution time distribution in Laplacian blending

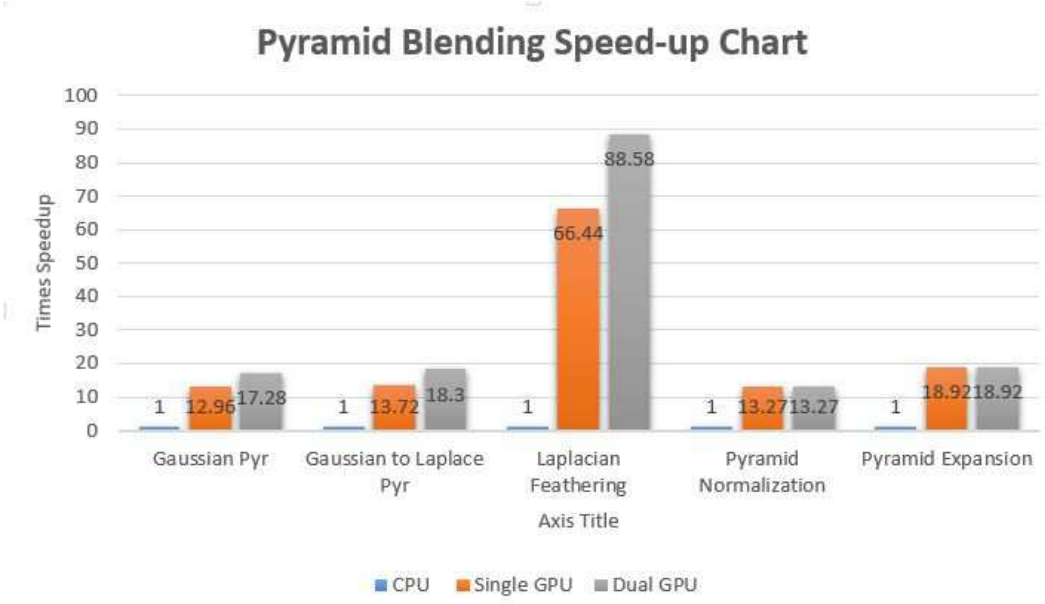


Figure 5.8: GPU Speedup over CPU for various components in Pyramid Blending

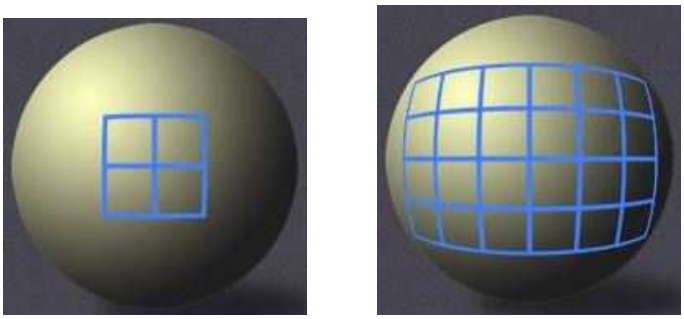
Chapter 6

Virtual View Generation

In the previous chapter, we described the process of creating a stitched panorama that forms the basis for virtual view generation. The final step in the real-time free viewpoint system is the selection of virtual views from this panorama. These views are called *virtual* or *novel* because they may or may not come from a single camera view. In general, the virtual views will be formed by overlapping regions of two or more camera views. These regions of interest are selected by the end users, so it is important to describe the selection and rendering mechanisms. The number of users can be large, hence scalability and transmission are important constraints. In this chapter, we will discuss the theory and process of generating the virtual views from stitched spherical panoramas. Furthermore, we will provide a review, discussion, and guidelines for improving video encoding, transmission, and rendering mechanisms that are compatible with existing TV transmission systems such as setup boxes using available network bandwidth.

6.1 Algorithm

As discussed in Chapter 4, in our system, the stitched images are warped using a spherical coordinate system. In general, spherical mapping distorts images and the level of distortion depends on the horizontal and vertical field-of-views i.e. the distortion is minimal for a small field-of-view, but increase as the field-of-view becomes wider as shown in Figure 6.1. To remove the distortion due to warping, the selected virtual view needs to be mapped to a planer surface from the spherical



(a) Small field-of-view, minimal warping

(b) Larger field-of-view, visible warping

Figure 6.1: Wapring distortion and Field-of-View in panorama reconstruction

panorama. This can be performed by using a perspective projection of the selected spherical view to a planar surface. The generation of the virtual view from a spherical panorama is generally not a very compute intensive operation and can be performed in real-time on modern CPU or GPU systems. Several players are available that use panorama texture capable of providing pan, tilt, and zoom functionalities in real-time [62] [15].

In order to select a virtual view, one needs to specify the intrinsic and extrinsic parameters of a virtual camera using a *pinhole camera model* [55]. The intrinsic parameter $\boldsymbol{K}$ of the camera is created by selecting a value of the focal length f. This value can be set as the average focal length of the input cameras f_{avg}. The level of zoom in the virtual view can be adjusted by increasing or decreasing the value of f. The rotation $\boldsymbol{R}$ of the camera can be used to set the pan and tilt of the virtual view. If λ is the homogeneous scaling factor and $\boldsymbol{C}$ is the center of the camera, then a 2D point $\boldsymbol{p} = (x, y, 1)^T$ on a planer surface can be described as:

$$\lambda \boldsymbol{p} = [\boldsymbol{K}|0_3] \begin{bmatrix} \boldsymbol{R} & -\boldsymbol{RC} \\ 0_3^T & 1 \end{bmatrix} \boldsymbol{P}. \tag{6.1}$$

The 2D point $\boldsymbol{p}$ can be back projected to the 3D space to get a set of 3D points in Cartesian coordinates and this operation can be defined by tracing a ray from the center of the camera through point $\boldsymbol{p}$ reaching the point $\boldsymbol{P}(X, Y, Z)$. This simple projection can be computed using the following equation:

$$\boldsymbol{P} = \boldsymbol{C} + \lambda \boldsymbol{R}^{-1}\boldsymbol{K}^{-1}\boldsymbol{p}, \tag{6.2}$$

where the center of the camera $\boldsymbol{C}$ is set to zero as the sphere is centered at the origin and λ is the homogeneous scaling factor calculated as:

$$\lambda = \frac{Z}{k}, \tag{6.3}$$

where the term k is taken after calculating the expression:

$$(r, j, k)^T = \boldsymbol{R}^{-1}\boldsymbol{K}^{-1}\boldsymbol{p}. \tag{6.4}$$

Once we have calculated the 3D point $\boldsymbol{P}$, the corresponding point on the spherical panorama can be calculated by taking a projection of this 3D point on the panorama surface using the Equation 4.6 and Equation 4.7. When the process is repeated for all the pixel locations for the planar image, a virtual view is generated.

Alternatively, the more common approach is to bind the panorama image to a spherical 3D texture and the intersection of this sphere with the ray in Equation 6.2 gives the spherical coordinates of point $\boldsymbol{p}$ in the spherical panorama.

These computations can be performed easily while the panorama image resides on the GPU. This will eliminate the execution time for transmitting the images back to the host memory. When performed on the GPU using CUDA, the execution time is similar to the initial projection of the images to the panorama as described in Chapter 4. However the total execution time will scale depending on the number of virtual views being rendered at the same time depending on the application and number of users.

6.2 Virtual View Transmission

Most of the systems that we have come across transmit the complete panorama or depth maps to the end user over Internet using protocols such as HTTP. There are several key issues that hinders the adaptability to use this practice with existing network infrastructure. First, the system requires a large network bandwidth than needed for a full HD video transmission. This is because the size of a complete

panorama (or depth map) can be several times larger than the region of interest (ROI) that is being rendered by an end-user display. For example, in our setup, the complete panorama has a planar resolution of 4439 x 1189 which is 2.5 times larger than a full HD stream. In a real FTV application with even larger number of cameras, the difference will be even bigger. Hence, the significant network resources are being wasted by transmitting the entire panorama. Depending on the available bandwidth, transmitting the complete panorama can also cause jitters and lags in the video playback, especially over a wide area network. Secondly, the large image sizes makes some systems incompatible with some efficient video encoders. Most of the video encoders works best for some pre-defined video resolutions, most commonly at 1080p, 720p and 480p by performing hardware encoding. The arbitrary resolution videos and comparatively much larger panorama is bound to consume larger encoding resources and at the same time cause encoding inefficiencies. Another key aspect to consider is the computational power of existing user-end set-top boxes. These set-top boxes generally have limited capabilities and a dedicated GPU might not be available. Most of the times, these set-top boxes can only decode a single stream of video in H.264, AVC or HEVC video formats. Hence using arbitrary sized videos, using a web browser based player or performing compute intensive reprojection computations on these devices are out of question for most of the set-top boxes.

6.2.1 Static Views

A simple solution is to create several virtual views on the server side and provide the user with a remote control capability to select one of the views. Although a truly virtual view selection system would allow selecting all possible views or regions within the panorama interactively. However, this might be infeasible due to network bandwidth and computational limitations. To balance the computational requirements with the freedom of choosing view-points, a more practical solution would be to fix a certain number of views that the user can choose from. The user could then select the views either by selecting individual channels where each channel gives a different viewpoint, by using a joy-stick connected through the set-top

Figure 6.2: Multiple Static Views from Panorama

box or even by a smart-phone application. This is based on the assumption that a few carefully selected virtual views might fulfill the requirements of the users. A usability study might be needed to establish the frequency at which a user might want to switch particular viewpoints. The number of available views can be scaled depending on the limited resources of the server. This architecture is readily compatible with existing television transmission and broadcast systems as each virtual view will act as a separate channel and utilize the existing broadcast capabilities. There will be a small lag in selecting the viewpoint, but it can act as a good starting point towards providing multiple views to a large audience.

6.2.2 Dynamic Views

The interactive switching of the virtual views by the end user throughput the panorama viewing space while utilizing efficient encoding and transmission schemes is an interesting area of research. Several architectures exists that might be suitable for different applications.

Neg et. al. proposed [58] an advanced delivery sharing scheme (ADSS) by constructing a video on demand service for panorama video. They divide the entire panorama into multiple tiles of fixed sizes whereas each tile is encoded into a separate stream. The decoder detects and reads multiple tiles associated with a view, multiplexes and decodes them and binds them to a texture buffer for render-

ing and display. They support common geometry models for panoramas such as rectilinear, cylindrical, and spherical. Coupled with an advanced delivery sharing protocol (ADSP) and a distributed setup consisting of local and wide area networks, they reported that they were able to achieve interactive performance standards for generating virtual views. However they utilize a local area network in addition to a wide area network that might not be available for some applications such as TV broadcast systems.

Ravindra et. al. [29] tested five different tile assignment methods using different transmission delays along with a greedy heuristics method and were able to provide a scalable solution to a simulated 50 users experiment at a multicast rate of 5.5 Mbps. However traditional video and tv transmission systems need to support millions of concurrent users and the scalability aspects for that large number of users is not certain.

Gaddam et. al. [25] proposes a tiling based virtual view selection system that utilizes changing constant rate factor (CFQ) where the server encodes the tiles at several resolutions. The client retrieves all tiles, which is equivalent to transmitting the complete panorama. However, lower resolution versions of tiles which do not form the virtual view are transmitted using a feedback system to provide improved network bandwidth utilization.

6.2.3 GPU Side Video Encoding

Once the virtual views are generated, the next step is to transmit them to be rendered at the end user display screen. The virtual images obtained are large in size and transmitting them in uncompressed format will significantly strain the network resources. For example, transmitting an uncompressed color images with the resolution of 1920 x 1080 pixels at 30 fps will require a bandwidth of 1.42 Gbps which is very high even for a local area network. Hence, some video compression is necessary to decrease the bandwidth requirements and at the same time make the stream compatible with traditional decoders.

In order to conserve compute resources, the most efficient way would be to perform a H.264 encoding on the GPU using the NVENC library [59]. The library

Figure 6.3: Oclus Rift and Tobi Eye Tracker

supports multiple video streams per GPU depending on the GPU capabilities and the encoding operation can be scaled to multiple GPU devices per machine.

6.2.4 Storage and Analytics

The system can optionally provide storage facilities for the panorama or the virtual views generated, either in the form of images or as an encoded video, on to a mass storage medium such as disk drives. The stored data can be used for later viewing selected portion of the panorama video. Since the number of views can be large, a scalable and high throughput system based on Redundant Array of Scalable Drives (RAID) controller might be utilized. A good overview for scalable video on demand with storage services is provided by Chan [14]. The recorder data can also be used to generate video analytics that may not be possible to process in real-time. The analytics can be very valuable for some sports where existing computer vision algorithms can be used to generate field heat-map, player performance statistics, performance and result predictions etc. Some of the capabilities such as virtual cameraman can be provided in real-time for tracking and focusing on the ball or a selected player.

6.3 Applications

6.3.1 Wearable Gears

Human beings typically have a horizontal field of view of 180° and a vertical field of view of 135°. An interesting set of applications for real-time panorama video and

Figure 6.4: TV Set Top Box

virtual view generation can be developed with wearable gears such as Oculus Rift coupled with an eye tracking technology. The large panoramas can greatly enhance the visible field of view and thereby enhance the visual experience. The position of the head or eyes can act as a remote control for selecting the viewpoint within a much larger spherical panorama. Similar technologies are used for generating panoramic cockpit display systems for aircrafts such as the one in the Lockheed F-35 aircraft [37]. Real-time panorama based headgear can become common for moving vehicles including cars as they deliver a safer experience by providing drivers complete field of view of their environment. Consequently, these devices would require customized virtual view generation to provide an interactive experience.

6.3.2 TV Set Top Boxes

The set top boxes or units are widely used devices for viewing television and video. In modern devices, the video stream is usually received over a wide area network and most devices support internet connectivity. These devices generally have limited hardware capabilities to maintain low cost. All devices support video decoding usually at 2160p, 1080p and 720p high definition resolution formats. Since custom hardware for selecting and watching multiple views can take years for mass adaptability, a system developed with the limited capabilities of the set top boxes can gain popularity in little time. Static view based systems might be readily supported with

these set top boxes as multiple view channels can transfer multiple view data. Some newer devices support Android and iOS operating systems (OS) and providing an application for these OS will allow easy integration. In larger displays supporting 4K or even higher resolution, it might be possible to tile multiple views at the same time, each with a 1080p or 720p resolution, to provide multiple concurrent views to the user.

Chapter 7

Conclusion

7.1 Summary

The video and television industries are expected to undergo a major transformation as the technologies around FVV and FTV mature. The two existing roadblocks in the adoption of FVV and FTV are real-time processing constraints and the need of comparable image quality between current HD broadcasting and FTV systems. In this thesis, we have presented a system to produce real-time panoramic videos using spherical panorama representation without sacrificing the visual quality of the panorama. Most of the development proposed in this thesis was made possible by the efficient porting of panoramic video algorithms onto GPU's massively parallel processing capabilities.

In Chapter 3, we presented our camera system built from 5 Herodion cameras. The Herodion camera provides hardware based synchronization that help to avoid severe temporal misalignment artifacts for dynamic scenes. The cameras are arranged around a common axis center. Each camera is configured to meet our requirements of capturing 1920 x 1080 pixel resolution video stream at 30 Hz. The five captured streams are in Bayer format which consists of a 1D array of green, blue and red colors. The first processing step is to convert these steams to RGB format in real-time using a Bayer demosaicing filter. In the thesis, we describe our implementation on GPU of those filters and show how good quality color interpolation can be performed in real-time using various optimization steps and data caching mechanisms. This efficient implementation resulted in an execution time

of under one millisecond for Bayer interpolation. In this chapter, we also show that color variations in a camera system is a common problem that need to be resolved before fusing the steams into a live panorama. We show that by a combination of camera gain changes and a color correction algorithm based on a Macbeth color chart one can remove the color variation between multiple video streams. We also show that by using on-chip GPU cache, color calibration can be performed with very little computational overhead on the overall panorama stitching process.

In Chapter 4, we describe the panorama stitching process that we have proposed and developed. We show that the spherical panorama representation provide the best possible horizontal and vertical field-of-views compared to cylindrical and rectilinear representation. We also demonstrate that feature based stitching approach along with its various implementation details is robust to camera changes and can be implemented efficiently by separating the transformation parameter estimation from the application of this transformation at each pixel. This is due to the fact that pixel transformation is eminently parallel and can be separated from the more sequential transformation parameters estimation. Next, we presented a GPU based Cartesian to spherical projection with stitching algorithm and discuss how specific optimizations on GPU such as: copy-execution overlap, texture memory based transformation maps, GPU occupancy, and memory coalescing can be used to guarantee real-time performance. Finally, we present the performance improvements by comparing the execution times of the algorithms on CPU and GPU for images with 640p and 1080p HD resolutions. Our GPU based stitching algorithm provide two orders of magnitude performance increase in processing speed compare to CPU algorithm, effectively expanding its application to commodity GPUs, such as the ones found in modern laptop computers.

Chapter 5 is divided into two parts. In the first part, we explain the optimal seam selection algorithm where we use Vornoi diagrams to select the optimal seams in overlapping image areas. Although there are more complex algorithms available, the advantage of using Vornoi diagrams based seam fusion is that they can be computed very rapidly using a multi-threaded CPU based implementation. Furthermore, when coupled with an advanced image blending technique, the approach

provides seamless panoramas fusion. The masks computed for a single set of images can be used for subsequent video frames and ensures less scene variation. In the second part of this chapter, we explain the proposed real-time Laplacian pyramid blending algorithm. We first present the blending problem and discuss the CPU based algorithm in detail. Next, we discuss the various processing steps to perform this Laplacian blending algorithm and compare it to simpler approaches such as Feather blending. We then explain some of the optimizations for our real-time GPU implementation such as eliminating memory transfers, using shared and texture on-chip GPU caches, kernel design, vector types to improve memory throughput and memory coalescing. In order to achieve even better performance, we also implemented a multi-GPU version of the algorithm where the processing load is divided between two identical GPUs. The multi-GPU implementation can scale to any number of GPUs depending on the number of the concurrent video streams. Finally, we provide experimental results measuring the performance of our implementation by benchmarking the difference between the CPU and GPU algorithms implementation for two video streams at 640p and 1080p high definition resolutions. We show that the various components provide between 13 times to 88 times speed-up compare to an efficient CPU implementation while the total execution time for the complete Laplacian Pyramid blending algorithm is reduced to 30 milliseconds. The resulting panorama does not suffer from any visible artifacts or excessive blurring and matches or exceed the visual quality of broadcast HD transmission.

Chapter 6 describes the process of creating virtual views from the constructed spherical panoramas. As the stitched panorama is distorted, the selected virtual view regions needs to be re-projected onto a planer surface for rendering. We describe two commonly used re-projection algorithms in detail. As the generation of the single virtual view is not computationally intensive, we list several available players that can provide real-time rendering of the virtual views. We also discuss the transmission, compression, and rendering issues associated with the scalable broadcast of virtual views. The simple static views strategy makes the system readily adaptable with the existing broadcast transmission systems and current end-user set top boxes. We also provide a literature review of the existing techniques for

dynamic view generation and transmission which still remains an active and open research topic. In this chapter, we also provide some guidelines for using an efficient video encoder implementation that can provide GPU based video encoding. The system can optionally provide mass storage and video analytics based on advanced computer vision algorithms to provide additional features to the system. Finally, we discuss two common applications for real-time virtual view rendering and discuss how these application effect the system design.

7.2 Future Work

The panorama video system was designed to be modular, making it very easy to extend the system in the future. Most of the future work can be geared towards the scalability of the system in a broadcasting situation. One of the first additions can be to add vertical cameras to extend the virtual field of view to take full advantage of the spherical panorama representation. We believe that by using the proposed algorithms, it is computationally possible to create a complete 360° x 180° real-time panoramic video system by providing additional cameras and computation infrastructure. This would indeed require several mounted cameras both horizontally and vertically.

A limitation of the current cameras was that they had to be connected to the host computer using a PCI-X based concentrator, making it incompatible with modern systems. Hence, we are working with the manufacturers to obtain PCI-Express based concentrators that will provide a large bandwidth jump and expand compatibility. Another possible addition could be to use wireless technologies, such as Wi-Fi for the camera system to make the system truly portable.

One addition to the system could be to compensate for radial lens distortion at the image acquisition phase to further improve the panorama quality. This can be done with a true bundled adjustment calibration programs such as the Agisoft PhotoScan software http://www.agisoft.com/.

Although the Vornoi diagram optimal seam selection algorithm works well, it might be worthwhile to experiment with advanced optimal seam selection algo-

rithms with the current system and report any improvements.

A possible extension of this thesis is to develop a scalable and distributed virtual view generation, encoding and transmission system using the existing or novel algorithms. Some of these algorithms are discussed in Chapter 6. Developing this system would be more practical with access to an existing infrastructure of TV or video broadcast system to verify integration and functionality. Therefore, we have established contacts with TELUS, which is one of the largest IPTV system providers in Canada. Some applications that we are considering include live sports coverage, panoramic operating rooms views, and live teleconferencing systems.

Bibliography

[1] M. Adam, C. Jung, S. Roth, and G. Brunnett. Real-time stereo-image stitching using gpu-based belief propagation. In *VMV*, pages 215–224, 2009.

[2] E. H. Adelson and J. R. Bergen. *The plenoptic function and the elements of early vision*. Vision and Modeling Group, Media Laboratory, Massachusetts Institute of Technology, 1991.

[3] A. Agarwala, M. Dontcheva, M. Agrawala, S. Drucker, A. Colburn, B. Curless, D. Salesin, and M. Cohen. Interactive digital photomontage. *ACM Transactions on Graphics (TOG)*, 23(3):294–302, 2004.

[4] C. Arth, M. Klopschitz, G. Reitmayr, and D. Schmalstieg. Real-time self-localization from panoramic images on mobile devices. In *Mixed and Augmented Reality (ISMAR), 2011 10th IEEE International Symposium on*, pages 37–46. IEEE, 2011.

[5] F. Aurenhammer and H. Edelsbrunner. An optimal algorithm for constructing the weighted voronoi diagram in the plane. *Pattern Recognition*, 1984.

[6] H. H. Baker, D. Tanguay, and C. Papadas. Multi-viewpoint uncompressed capture and mosaicking with a high-bandwidth pc camera array. In *Proc. Workshop on Omnidirectional Vision (OMNIVIS 2005)*, 2005.

[7] P. Baudisch, D. Tan, D. Steedly, E. Rudolph, M. Uyttendaele, C. Pal, and R. Szeliski. An exploration of user interface designs for real-time panoramic. *Australasian Journal of Information Systems*, 13(2), 2006.

[8] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. In *Computer vision–ECCV 2006*, pages 404–417. Springer, 2006.

[9] J. R. Bergen, P. Anandan, K. J. Hanna, and R. Hingorani. Hierarchical model-based motion estimation. In *Computer VisionECCV'92*, pages 237–252. Springer, 1992.

[10] G. Borgefors. Distance transformations in digital images. *Computer vision, graphics, and image processing*, 34(3):344–371, 1986.

[11] M. Brown and D. G. Lowe. Automatic panoramic image stitching using invariant features. *International journal of computer vision*, 74(1):59–73, 2007.

[12] P. J. Burt and E. H. Adelson. The laplacian pyramid as a compact image code. *Communications, IEEE Transactions on*, 31(4):532–540, 1983.

[13] P. J. Burt and E. H. Adelson. A multiresolution spline with application to image mosaics. *ACM Transactions on Graphics (TOG)*, 2(4):217–236, 1983.

[14] S.-H. G. Chan and F. A. Tobagi. *Scalable services for video-on-demand*. Stanford University, 1999.

[15] S. E. Chen. Quicktime vr: An image-based approach to virtual environment navigation. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 29–38. ACM, 1995.

[16] D. R. Cok. Reconstruction of ccd images using template matching. In *IS&Ts 47th Annual Conference/ICPS*, pages 380–385, 1994.

[17] J. E. Coleshill and A. Ferworn. Panoramic spherical videothe space ball. In *Computational Science and Its ApplicationsICCSA 2003*, pages 51–58. Springer, 2003.

[18] Cuda guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`. Accessed: 2015-06-27.

[19] R. C. Dorf. *Circuits, signals, and speech and image processing*. CRC Press, 2006.

[20] M. A. El-Saban, M. Refaat, A. Kaheel, and A. Abdul-Hamid. Stitching videos streamed by mobile phones in real-time. In *Proceedings of the 17th ACM international conference on Multimedia*, pages 1009–1010. ACM, 2009.

[21] M. A. Fischler and R. C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of ACM*, 24(6):381–395, June 1981.

[22] J. Foote and D. Kimber. Flycam: Practical panoramic video and automatic camera control. In *Multimedia and Expo, 2000. ICME 2000. 2000 IEEE International Conference on*, volume 3, pages 1419–1422. IEEE, 2000.

[23] T. Fujii. Ray space coding for 3d visual communication. In *Picture Coding Symposium'96*, volume 2, pages 447–451, 1996.

[24] Y. Furukawa and J. Ponce. Accurate, dense, and robust multiview stereopsis. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(8):1362–1376, 2010.

[25] V. Gaddam and H. N. et. al. Tiling of panorama video for interactive virtual cameras: Overheads and potential bandwidth requirement reduction. *21st International Packet Video Workshop*, 2015.

[26] G. H. Golub and C. F. Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[27] R. C. Gonzalez and R. E. Woods. Digital image processing 3rd edition, 2007.

[28] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. The lumigraph. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 43–54. ACM, 1996.

[29] R. Guntur and W. T. Ooi. On tile assignment for region-of-interest video streaming in a wireless lan. In *Proceedings of the 22nd international workshop on Network and Operating System Support for Digital Audio and Video*, pages 59–64. ACM, 2012.

[30] B. K. Gunturk, Y. Altunbasak, and R. M. Mersereau. Color plane interpolation using alternating projections. *Image Processing, IEEE Transactions on*, 11(9):997–1013, 2002.

[31] P. Halvorsen, S. Sægrov, A. Mortensen, D. K. Kristensen, A. Eichhorn, M. Stenhaug, S. Dahl, H. K. Stensland, V. R. Gaddam, C. Griwodz, et al. Bagadus: an integrated system for arena sports analytics: a soccer case study. In *Proceedings of the 4th ACM Multimedia Systems Conference*, pages 48–59. ACM, 2013.

[32] P. Halvorsen, S. Sægrov, A. Mortensen, D. K. Kristensen, A. Eichhorn, M. Stenhaug, S. Dahl, H. K. Stensland, V. R. Gaddam, C. Griwodz, et al. Bagadus: an integrated system for arena sports analytics: a soccer case study. In *Proceedings of the 4th ACM Multimedia Systems Conference*, pages 48–59. ACM, 2013.

[33] R. Hartley and A. Zisserman. *Multiple view geometry in computer vision*. Cambridge university press, 2003.

[34] M. Hess-Flores, D. Knoblauch, M. A. Duchaineau, K. I. Joy, and F. Kuester. Ray divergence-based bundle adjustment conditioning for multi-view stereo. In *Advances in Image and Video Technology*, pages 153–164. Springer, 2012.

[35] Motiondsp ikene isr real-time mosaicing. `http://www.motiondsp.com/products/ikena-isr/real-time-mosaicing`. Accessed: 2015-06-10.

[36] J. Jia and C.-K. Tang. Image stitching using structure deformation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(4):617–631, 2008.

[37] M. H. Kalmanash. Panoramic projection avionics displays. In *AeroSense 2003*, pages 289–298. International Society for Optics and Photonics, 2003.

[38] T. Kanade, P. Rander, and P. Narayanan. Virtualized reality: Constructing virtual worlds from real scenes. *IEEE multimedia*, pages 34–47, 1997.

[39] D. Kimber, J. Foote, and S. Lertsithichai. Flyabout: spatially indexed panoramic video. In *Proceedings of the ninth ACM international conference on Multimedia*, pages 339–347. ACM, 2001.

[40] R. Kimmel. Demosaicing: image reconstruction from color ccd samples. *Image Processing, IEEE Transactions on*, 8(9):1221–1228, 1999.

[41] Kodak image suite. `http://r0k.us/graphics/kodak/`. Accessed: 2015-06-27.

[42] A. Levin, A. Zomet, S. Peleg, and Y. Weiss. Seamless image stitching in the gradient domain. In *Computer Vision-ECCV 2004*, pages 377–389. Springer, 2004.

[43] M. Levoy and P. Hanrahan. Light field rendering. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 31–42. ACM, 1996.

[44] H. Li, S. Lin, Y. Zhang, and K. Tao. Automatic video-based analysis of athlete action. In *Image Analysis and Processing, 2007. ICIAP 2007. 14th International Conference on*, pages 205–210. IEEE, 2007.

[45] M. Li, M. Magnor, and H.-P. Seidel. Hardware-accelerated rendering of photo hulls. In *Computer Graphics Forum*, volume 23, pages 635–642. Wiley Online Library, 2004.

[46] X. Li, B. Gunturk, and L. Zhang. Image demosaicing: A systematic survey. In *Electronic Imaging 2008*, pages 68221J–68221J. International Society for Optics and Photonics, 2008.

[47] W.-S. Liao, T.-J. Hsieh, W.-Y. Liang, Y.-L. Chang, C.-H. Chang, and W.-Y. Chen. Real-time spherical panorama image stitching using opencl. In *2011 International Conference on Computer Graphics and Virtual Reality*, pages 113–119, 2011.

[48] S. Loncaric. A survey of shape analysis techniques. *Pattern recognition*, 31(8):983–1001, 1998.

[49] B. D. Lucas, T. Kanade, et al. An iterative image registration technique with an application to stereo vision. In *IJCAI*, volume 81, pages 674–679, 1981.

[50] H. S. Malvar, L.-w. He, and R. Cutler. High-quality linear interpolation for demosaicing of bayer-patterned color images. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings.(ICASSP'04). IEEE International Conference on*, volume 3, pages iii–485. IEEE, 2004.

[51] W. Matusik, C. Buehler, R. Raskar, S. J. Gortler, and L. McMillan. Image-based visual hulls. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 369–374. ACM Press/Addison-Wesley Publishing Co., 2000.

[52] L. McMillan and G. Bishop. Plenoptic modeling: An image-based rendering system. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 39–46. ACM, 1995.

[53] J. Meehan. *Panoramic Photograph.* Watson-Guptill., 1990.

[54] D. Menon, S. Andriani, and G. Calvagno. Demosaicing with directional filtering and a posteriori decision. *Image Processing, IEEE Transactions on*, 16(1):132–141, 2007.

[55] Y. Morvan. *Acquisition, compression and rendering of depth and texture for multi-view video*. PhD thesis, Technische Universiteit Eindhoven, 2009.

[56] S. K. Nayar. Catadioptric omnidirectional camera. In *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*, pages 482–488. IEEE, 1997.

[57] U. Neumann, T. Pintaric, and A. Rizzo. Immersive panoramic video. In *Proceedings of the eighth ACM international conference on Multimedia*, pages 493–494. ACM, 2000.

[58] K.-T. Ng, S.-C. Chan, and H.-Y. Shum. Data compression and transmission aspects of panoramic videos. *Circuits and Systems for Video Technology, IEEE Transactions on*, 15(1):82–95, 2005.

[59] Nvidia video codec sdk. https://developer.nvidia.com/nvidia-video-codec-sdk. Accessed: 2015-06-10.

[60] Opencv: Open computer vision library. http://opencv.org. Accessed: 2015-05-13.

[61] A. V. Oppenheim, R. W. Schafer, J. R. Buck, et al. *Discrete-time signal processing*, volume 2. Prentice-hall Englewood Cliffs, 1989.

[62] Panorama viewers. http://wiki.panotools.org/Panorama_Viewers. Accessed: 2015-07-16.

[63] D. Pascale. Rgb coordinates of the macbeth colorchecker. *The BabelColor Company*, pages 1–16, 2006.

[64] S. Peleg and M. Ben-Ezra. Stereo panorama with a single camera. In *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on.*, volume 1. IEEE, 1999.

[65] V. Peri and S. K. Nayar. Generation of perspective and panoramic video from omnidirectional video. In *Proc. DARPA Image Understanding Workshop*, volume 1, pages 243–245. Citeseer, 1997.

[66] Posix threads. http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html. Accessed: 2015-06-27.

[67] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov. Real-time computer vision with opencv. *Communications of the ACM*, 55(6):61–69, 2012.

[68] R. Ramanath, W. E. Snyder, G. L. Bilbro, and W. A. Sander. Demosaicking methods for bayer color arrays. *Journal of Electronic imaging*, 11(3):306–315, 2002.

[69] Qimara realtime virtual camera technology. http://www.qamira.com. Accessed: 2015-06-12.

[70] C. Schmid, R. Mohr, and C. Bauckhage. Evaluation of interest point detectors. *International Journal of computer vision*, 37(2):151–172, 2000.

[71] K. Shegeda and P. Boulanger. A gpu-based real-time algorithm for virtual viewpoint rendering from multi-video. In *GPU Computing and Applications*, pages 167–185. Springer, 2015.

[72] P. P. Shete, P. Venkat, D. M. Sarode, M. Laghate, S. Bose, and R. Mundada. Object oriented framework for cuda based image processing. In *Communication, Information & Computing Technology (ICCICT), 2012 International Conference on*, pages 1–6. IEEE, 2012.

[73] H.-Y. Shum and L.-W. He. Rendering with concentric mosaics. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 299–306. ACM Press/Addison-Wesley Publishing Co., 1999.

[74] H.-Y. Shum and R. Szeliski. Construction of panoramic image mosaics with global and local alignment. In *Panoramic vision*, pages 227–268. Springer, 2001.

[75] J. Starck and A. Hilton. Virtual view synthesis of people from multiple view video sequences. *Graphical Models*, 67(6):600–620, 2005.

[76] J. Starck and A. Hilton. Surface capture for performance-based animation. *Computer Graphics and Applications, IEEE*, 27(3):21–31, 2007.

[77] H. K. Stensland, V. R. Gaddam, M. Tennøe, E. Helgedagsrud, M. Næss, H. K. Alstad, A. Mortensen, R. Langseth, S. Ljødal, Ø. Landsverk, et al. Bagadus: An integrated real-time system for soccer analytics. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 10(1s):14, 2014.

[78] R. Szeliski. *Image Alignment and Stitching: A Tutorial*.

[79] M. Tanimoto, M. P. Tehrani, T. Fujii, and T. Yendo. Ftv for 3-d spatial communication. *Proceedings of the IEEE*, 100(4):905–917, 2012.

[80] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon. Bundle adjustmenta modern synthesis. In *Vision algorithms: theory and practice*, pages 298–372. Springer, 2000.

[81] T. Tuytelaars and K. Mikolajczyk. Local invariant feature detectors: a survey. *Foundations and Trends® in Computer Graphics and Vision*, 3(3):177–280, 2008.

[82] M. Uyttendaele, A. Eden, and R. Skeliski. Eliminating ghosting and exposure artifacts in image mosaics. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 2, pages II–509. IEEE, 2001.

[83] Videostitch. `www.video-stitch.com`. Accessed: 2015-06-12.

[84] Y. Xiong and K. Turkowski. Creating image-based vr using a self-calibrating fisheye lens. In *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*, pages 237–243. IEEE, 1997.

[85] J. C. Yang, M. Everett, C. Buehler, and L. McMillan. A real-time distributed light field camera. *Rendering Techniques*, 2002:77–86, 2002.

[86] D. Yow, B.-L. Yeo, M. Yeung, and B. Liu. Analysis and presentation of soccer highlights from digital video. In *proc. ACCV*, volume 95, pages 499–503. Citeseer, 1995.

[87] C. Zhang and J. Li. Compression and rendering of concentric mosaics with reference block codec (rbc). In *Visual Communications and Image Processing 2000*, pages 43–54. International Society for Optics and Photonics, 2000.

[88] J. Y. Zheng and S. Tsuji. Panoramic representation of scenes for route understanding. In *Pattern Recognition, 1990. Proceedings., 10th International Conference on*, volume 1, pages 161–167. IEEE, 1990.

[89] C. L. Zitnick, S. B. Kang, M. Uyttendaele, S. Winder, and R. Szeliski. High-quality video view interpolation using a layered representation. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 600–608. ACM, 2004.