universiteit hasselt

Improving 360-degree Panoramic Video Stitching

A thesis submitted for the partial fulfilment of the requirements for the degree of Master of Science in Engineering Technology Electronics-ICT

Thomas Dendale

21 June 2016

Abstract

This paper proposes improvements on an existing implementation of 360-degrees panoramic view stitching with static cameras by Huang et al. (2014). Seam stitching and performance are the main focus. Dynamic seam stitching is implemented using last-frame difference (LFD) and dynamic programming to detect moving objects. Brightness correction and blending are also improved. Furthermore, performance is increased to reach a FPS of 33 over the original 15, while maintaining identical quality.

Contents

Chapter 1

Introduction

360-degrees panoramic views are used more and more in a variety of situations. One implementation of such panoramic view consists of multiple cameras arranged in a circle, resulting in 360 degrees with overlapping areas. Each overlapping area needs to create a seam dividing the left and right image. A full-stack implementation using 4 cameras was developed by Kai-Chen Huang for his master thesis at the Institute of Computer science and Information Engineering at the National Chung-Cheng University (Huang et al., 2014) and patented by Jiun-In et al. (2015).

Figure 1.1 shows the final panorama view as used throughout this paper.

Figure 1.1: Final 360-degrees panorama view.

1.1 Goal

The aim of this paper is to try to improve his implementation of the 360-degrees panoramic video system. This paper will make an attempt at summarizing the original implementation

and explain in detail where necessary. The main focus will resolve around improving seam stitching for static camera systems, while maintaining real-time performance. Further improvements to the complete implementation are a secondary focus.

1.2 Paper overview

The remainder of the paper is structured in three sections. Section 2 will contain the description of the original implementation and sketch the flow of the program, including a detailed description of every important step. Section 3 will then present the proposed implementation improvements including experiments and results. Lastly, section 4 will address the reasoning behind decisions and results shown in the earlier sections.

Chapter 2

Original Implementation

The flow of the original system consists of two main threads (not included the QT thread handling the GUI). Both threads are started at the same time, after setup. Both threads run in parallel and should optimally only run once each frame.

2.1 Calibration using SIFT

The program is ran initially to calibrate the cameras using SIFT feature detection. This data is saved in a file and reused in subsequent runs, as calibration is only needed when the physical configuration of the cameras changes.

The position shift of each camera in respect to the first camera is calculated using SIFT feature matching. Information about the position of each camera is used in constructing the lookup tables discussed below.

2.2 Setup

The setup is divided into three parts.

First, after initializing the camera and capturing the first frame, a LUT (the locations LUT) is constructed transforming the fisheye camera coordinates to cylindrical coordinates, as discussed in 2.4.1.

Then, a second LUT (the cyl LUT) is constructed transforming the cylindrical coordinates to individual camera screen coordinates, see 2.4.2. This table will later be used to select the correct interpolated pixel values, as explained in part 2.4.3.

Lastly, a third LUT (the whole LUT) is constructed combining the individual cylLUTs. This table will be used to calculate the final panorama pixels.

A mutex lock is used to control the parallel access of the capture data memory and the screen output memory.

After setup, the two threads are created. The Capture thread (2.3) handles the camera capturing and the Stitch Thread (2.4) will handles the panorama stitching.

2.3 Capture Thread

The capture thread loops and handles the input from the cameras, writing the frame data of each camera into appropriate memory locations. When processing a recorded video, the full video is loaded into memory and only the starting address of the memory is updated each frame. The mutex is locked while writing to memory, followed by a sleep for a period of time. In the case of camera capture, the time to sleep corresponds to the frame rate of the camera. Figure 2.1 shows the full thread flow.

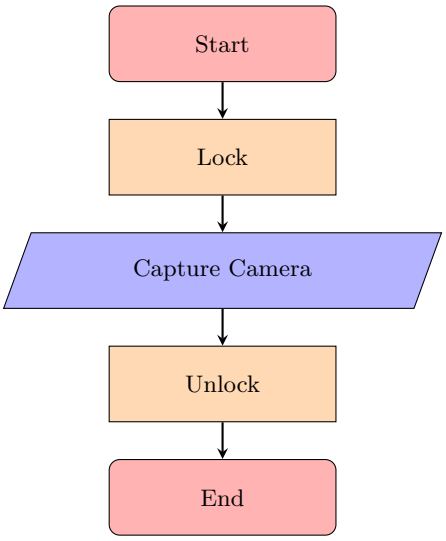


Figure 2.1: Capture thread program flow.

2.4 Stitching Thread

The stitch thread loop consists of the bulk of the processing. Figure 2.2 shows the flowchart of one full cycle.

For each camera, an additional thread is created that applies the fisheye correction, as described in 2.4.1, and interpolation (2.4.3). Conversion from YUV to RGB and back is needed for the fisheye correction step as fisheye correction is implemented in RGB color space. Finally all created threads are joined.

Next, for each overlap an additional thread is created to calculate the dynamic seam, as described in 2.5. Finally all created threads are joined.

Using the interpolated images, the whole LUT and calculated seam data, the final panorama view is stitched. For each output pixel in the panorama, the corresponding camera ID and interpolated image coordinate is calculated, and the final pixel is written.

Finally, additional threads are created for each seam and the resulting seams are blended by linearly interpolating between the left and right image depending on the position in the seam. All created threads are joined.

The result is converted to RGB a final time and drawn to the screen using QT. Note the lock and unlock of the mutex at the start and end of the complete cycle.

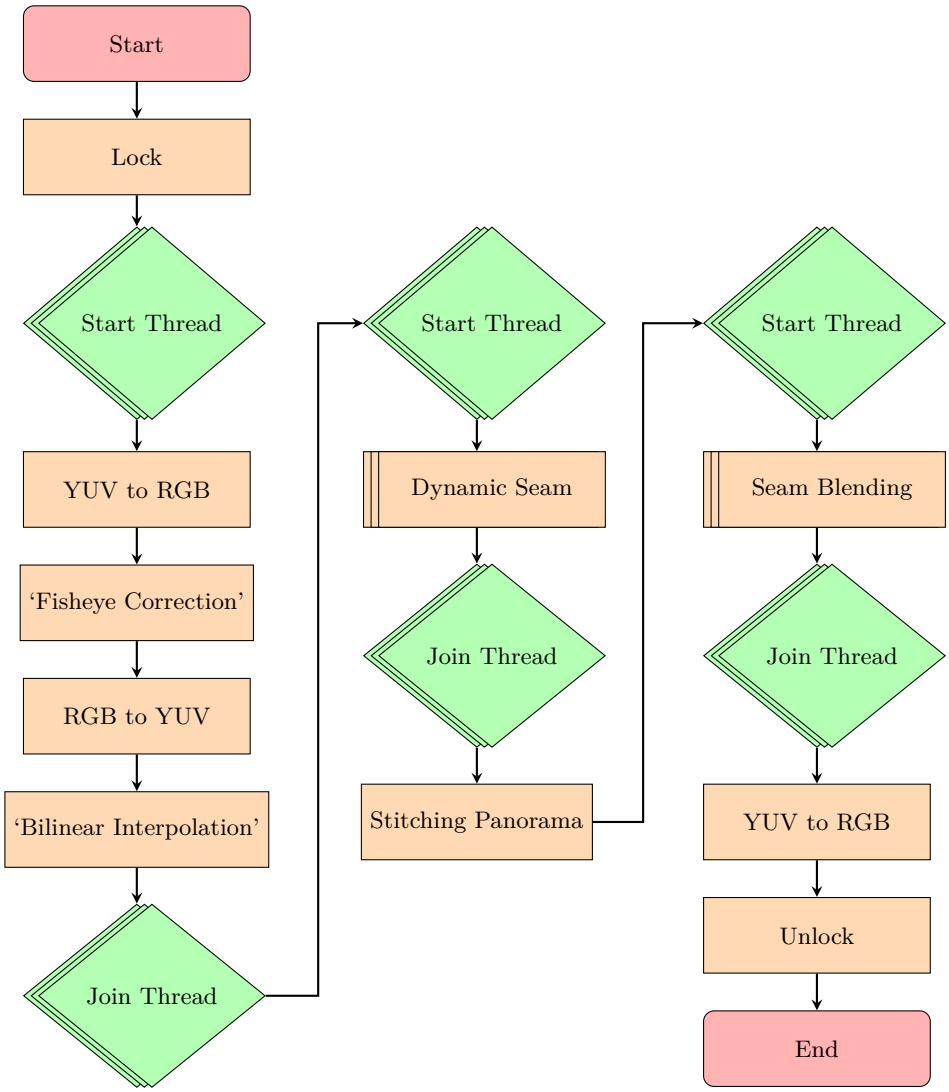


Figure 2.2: Stitch thread program flow.

2.4.1 Fisheye Correction

The first step in converting the camera input data to a usable camera image is mapping the original distortion, caused by the fish-eye, to the proper cylindrical coordinates. This transformation uses a predefined matrix (KK) containing focal length and the optical axis of the lens. Formula 2.1 shows this transformation.

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}_{corrected} = \mathbf{KK} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}_{camera} \tag{2.1}$$

Pixel values are calculated with additional bilinear interpolation, as shown in Figure 2.3.

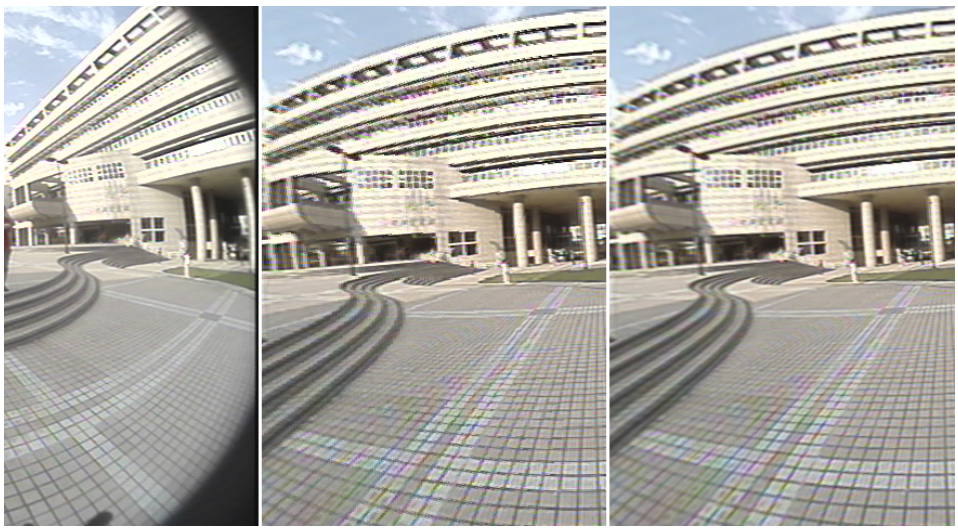


Figure 2.3: Original image, fisheye corrected image and bilinear interpolation

2.4.2 Cylindrical Projection

To align the camera views, only translations are used. When applying only 2D image manipulations (homography transformations) to align views, warp distortion will occur; the panorama view will become stretched near the sides.

To solve this problem and stitch a 360-degree view, cylindrical transformation was implemented. This transformation projects each camera screen image onto a cylinder. The actual implementation uses the inverse transformation to calculate the projection of screen coordinates to cylindrical coordinates. This is done by evaluating formulas 2.2 for every pixel in the corrected camera image, storing the result in the cyl LUT.

$$
\begin{aligned}
x &= f \cdot tan(\frac{x'_{img}}{f}) \\
y &= \frac{y'_{img} \cdot \sqrt{x^2 + f^2}}{f} \\
LUT(x_{img}, y_{img}) &= (x, y)
\end{aligned}
\tag{2.2}
$$

Where x_{img} is the x position of the pixel in the camera image and $x`_{img} = x_{img} - imagewidth/2$. Identical for y.

However, cylindrical projection is not a one-to-one transformation, some pixels in the final image will not correspond to a pixel after warping. Bilinear interpolation of neighbour pixels is performed to fill in those missing pixels.

2.4.3 Bilinear Interpolation

The original implementation uses a custom implementation of bilinear interpolation, by creating a 16 times bigger image (4*4 for each pixel) containing the original pixels in the corner of every window, as shown in figure 2.4. This method is easily extendable to quarter-pixel interpolation, as discussed in the original paper. The Y part of the bilinear interpolation is performed over a 4 by 4 pixel window.

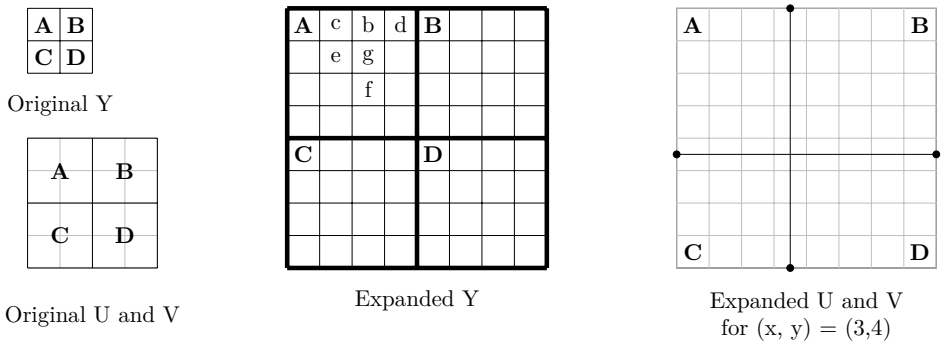


Figure 2.4: Bilinear Interpolation

This is called the Expanded Image. Using formula 2.3, all remaining pixels are calculated

$$
\begin{aligned}
b &= (A + B)/2 \\
c &= (2 \cdot A + B)/3 \\
d &= (A + 2 \cdot B)/3 \\
e &= (A + B + A + C)/4 \\
f &= (A + B + C + D)/4 \\
g &= b + f = (3 \cdot A + 3 \cdot B + C + D)/8
\end{aligned}
\tag{2.3}
$$

The U and V part of the interpolation uses a 8 by 8 pixel window. For every pixel in the window, the modulo of the position is taken to get the local x and y position inside the window.

Using this position, the final value is calculated using formula 2.4.

$$p = \frac{A \cdot (8-x) \cdot (8-y) + B \cdot x \cdot (8-y) + C \cdot (8-x) \cdot y + D \cdot x \cdot y}{64} \tag{2.4}$$

2.5 Original Dynamic Seam

Dynamic seams are important in video panoramas, as moving objects in the foreground are viewed from different angles for adjacent cameras. If the seam is simply a strait line, moving objects passing through this seam will create distorted results, especially after blending.

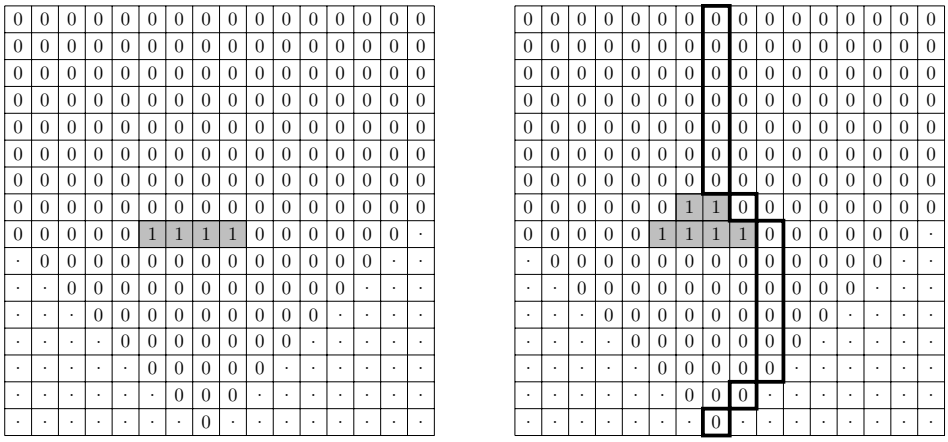
To prevent this distortion, the original implementation attempts to construct a dynamic seam that tries to avoid those objects. This is done by comparing the average luminance of a 3 by 3 pixel window between both sides of the seam. Dynamic programming (DP) is used to calculate the seam with the lowest difference. Finally, the previous frame is recalculated with the new camera images. The current seam total is compared to this previous seam total, and the seam is only updated if the new seam total is lower then a certain threshold below the previous total. This prevents the seam from updating too frequently and feeling unstable.

2.5.1 Dynamic Programming

The implementation of the total minimal path using dynamic programming is completed in three steps.

First the start and end position are sought. These points are the intersections between the outline of the left and right image.

Next, a cost table is constructed containing the accumulated cost of each pixel from top to bottom, only keeping the minimum of three possible values: the sum of the current pixel and one of the three pixels directly above. This method constrains the seam to only take a path that is straight down or diagonally between two pixels. A second table is filled with the appropriate direction of the minimum path: 1 for left, 2 for middle and 3 for right, as shown in figure 2.5.



Last-frame difference of a single moving object.

Accumulated minima and corresponding seam path

Figure 2.5: Dynamic programming example showing bottom section with a small moving object near the seam area.

Finally, the seam is constructed by looping over the direction table from start to end, storing the x position as array value for each y position.

2.6 Seam Blending

Finally, using the seam data, each seam is blended using linear interpolation between the left and right seam. As information from both cameras is needed, only the overlapping area of the camera can be interpolated.

In some situations ghosting appears: the moving object might be seen in two different positions at the same time, caused by interpolating images from cameras in different angles. This is the most notable when the angle between the cameras is too big or and the interpolation area is too large.

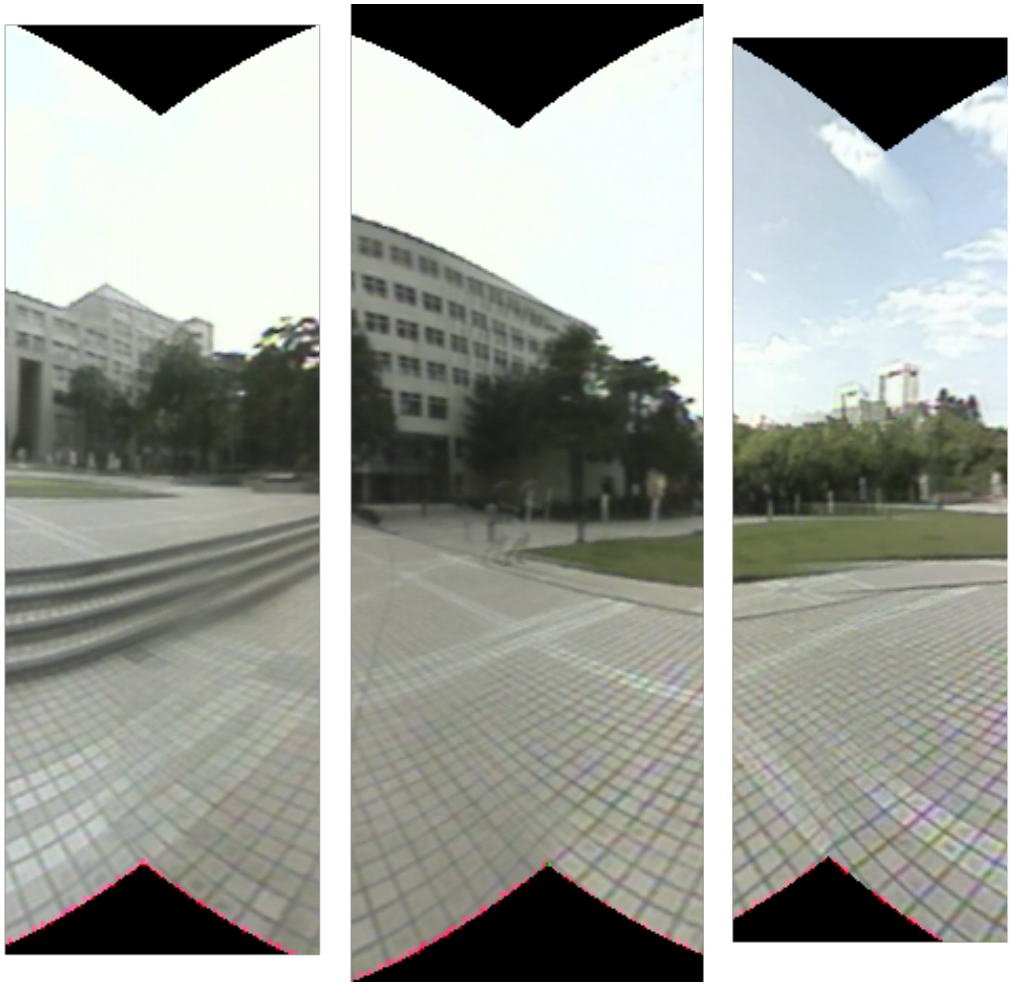


Figure 2.6: Final seam image. Note ghosting appearing, most notably in the middle image.

Chapter 3

Proposed Implementation

The next parts of the paper will talk about the proposed changes to the original paper to achieve better quality or performance. Part 3.2 will review related works and the final dynamic seam algorithm is proposed. Next, multiple performance improvements to the original implementation will be listed in part 3.3.

3.1 general refactoring

Before any implementation changes were made, a refactoring of the original data structure was performed. The intention of the refactor was readability and data encapsulation.

The change consists of rewriting most of the global state to local variables and using function arguments instead. Next, most of the coupled data, variables that are always needed together, were encapsulated into structures. Finally, most of the multi-dimensional array access (e.g. the y, u and v in a YUV image was a 3-layered array) was converted to pointers as structure members. Most notably, the fisheye correction was rewritten to use a single array containing 8 elements instead of 8 separate arrays containing one. (In this case object oriented design of the data is an actual improvement on the performance.)

3.2 Dynamic seam

3.2.1 Related work: Feature detection and Deformation

One method of improving seam stitching is presented by Jia and Tang (2008); Li et al. (2015). These methods perform feature detection and matching along the seam, followed by deformation, possibly followed by blending, to improve quality. This deformation will stretch and shrink the original image between feature points to create perfect transition from left to right.

Firstly, this method creates the seam by minimizing the frame difference between left and right. However, this does not optimize for distortion of moving objects. On the contrary, the quality of the seam even improves in cases where the seam crosses objects.

Secondly, when used in video, feature matching will need to be perfect at all times or the seam will become very unstable. Feature matching quality comes at a big performance cost. As such, this method is not usable for video.

As a result, feature matching and distortion cannot be considered a solution when calculating the seam of video panorama.

3.2.2 Last-frame Difference

When applying a constraint of static (non-moving) cameras, time information can be used to improve the seam. Moving objects can be easily recognised when using last-frame difference (LFD). Last-frame difference, as the name suggests, subtracts the current frame with the previous frame. This results in a clearly visible moving objects.

Background subtraction (Gandhamal and Talbar, 2015) can also be seen as a possible solutions to detect foreground objects. However, after implementing and testing simple LFD, simple moving-object detection appeared to produce very satisfactory results.

Last-frame difference is constructed by subtracting the current frame with the previous frame. Moving objects will create a clear outline. To reduce noise, very low values of difference are ignored and large differences are increased by using an exponential function.

The final result is shown in figure 3.1. Formula 3.1 were used, where LFD_{left} and LFD_{right} are the last-frame differences from left and right image.

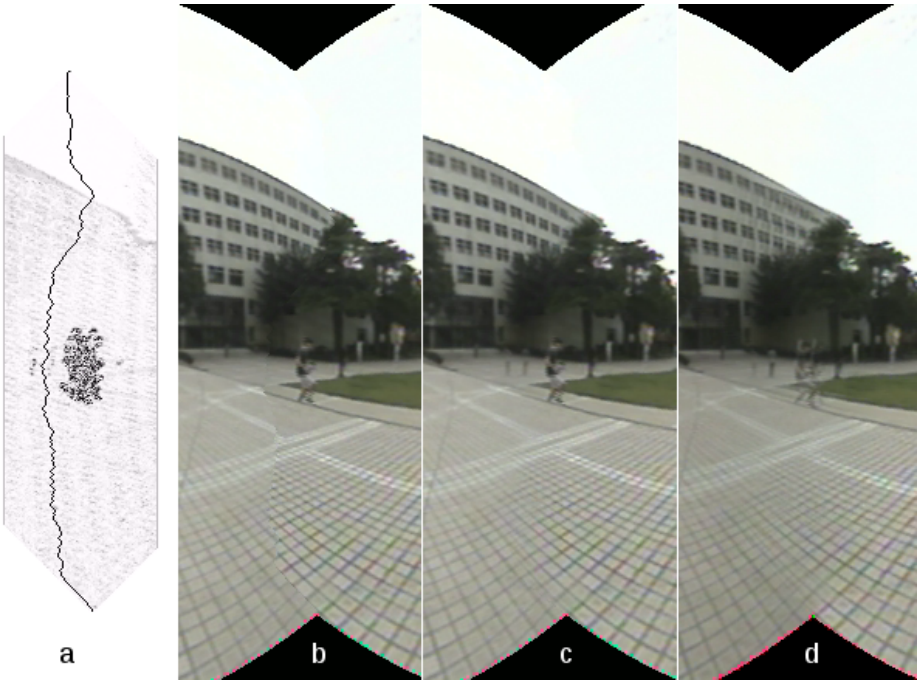


Figure 3.1: Last-frame difference (LFD) and seam (a), corresponding stitched image (b), final blended image (c) and original implementation for comparison (d). Darker pixels are more likely to correspond to moving objects.

$$result(x, y) = LFD_{left}(x, y)^2 + LFD_{right}(x.y)^2 \tag{3.1}$$

Note the periodic noise from the camera, clearly visible in the LFD image in figure 3.1 (a). Noise has a small impact on the seam result, as the exact path of the seam in the background is not of any real significance, as long as the moving object in the foreground has a much higher LFD value.

3.2.2.1 Dynamic Programming

The implementation of the dynamic programming algorithm is very similar to the original. But instead of constructing a table containing the most optimal direction between pixels, only the accumulated cost table is created.

Then, while constructing the seam data, the minimum is actively calculated instead of stored in a separate table.

Also, as discussed in 2.5.1, the original implementation uses a small averaging window to reduce the noise. This, however, is not used in the proposed implementation, as noise is much less of a problem when using last-frame difference.

Figure 3.2 shows the accumulated differences of the example scene.

Figure 3.2: Accumulated last-frame difference as used by DP.

3.2.3 brightness correction

Depending on the amount of incident light or individual camera settings, some cameras will have darker images compared to adjacent cameras. To improve seam quality in those cases, brightness correction is applied. Only those cameras with luminance above the combined average of all cameras are adjusted. The adjustment consists of a simple factor scaling and clamping of the final pixels. Figure 3.3 shows the final result.

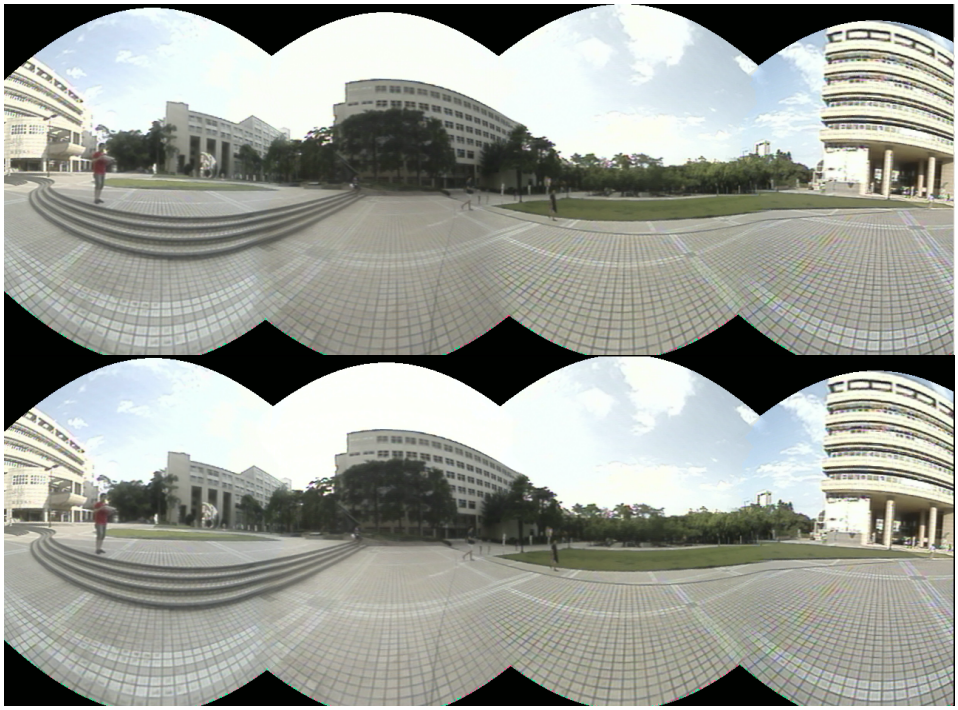


Figure 3.3: Brightness correction comparison. Top: original, bottom: corrected. Note the difference in brightness of the second camera.

3.2.4 Seam Blending

To improve performance and quality of seam blending, not every pixel in the overlapping area is interpolated. A region of 10 to 15 pixels on both sides of the seam is used as interpolation area. This reduces computation considerably.

3.3 Improving performance

The following section will discuss changes to improve performance. Part 3.4.1 summarizes the timed results.

3.3.1 Mutex

A small change was made to the thread locking. An additional mutex was created. One mutex locks when copying camera input to memory and one mutex locks when copying the final stitched panorama to the screen output memory.

3.3.2 Bilinear Interpolation

As discussed in 2.4.3 the original implementation creates an extra YUV image 16 times bigger than the original. When applying interpolation, the appropriate pixel in the enlarged 4 by 4 window is taken. This means 15 out of the 16 interpolation calculations are redundant.

The proposed implementation calculates the coordinates of the used pixel relative to the interpolation window first, using the cyl LUT and the modulo operation, followed by the actual interpolation, only evaluating one of the formulas in 2.3. The resulting interpolated YUV is the same size as the original.

Figure 3.4 shows the distribution of the modulo of this transformation. Figure 3.5 shows the steps to generate the interpolated Y image. A' is the bi-linearly interpolated value, using only one of the formulas in (2.3).

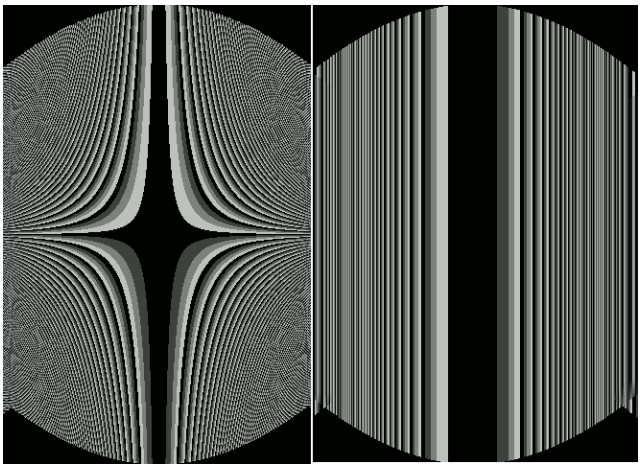


Figure 3.4: Modulo distribution after cylindrical transformation. Left is y, right is x.

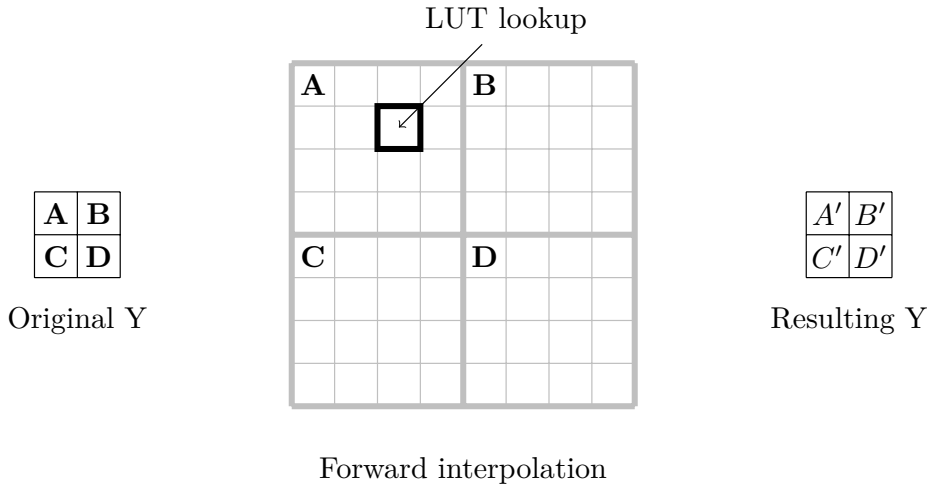


Figure 3.5: Forward Bilinear Interpolation for Y.

3.4 Final Result Difference

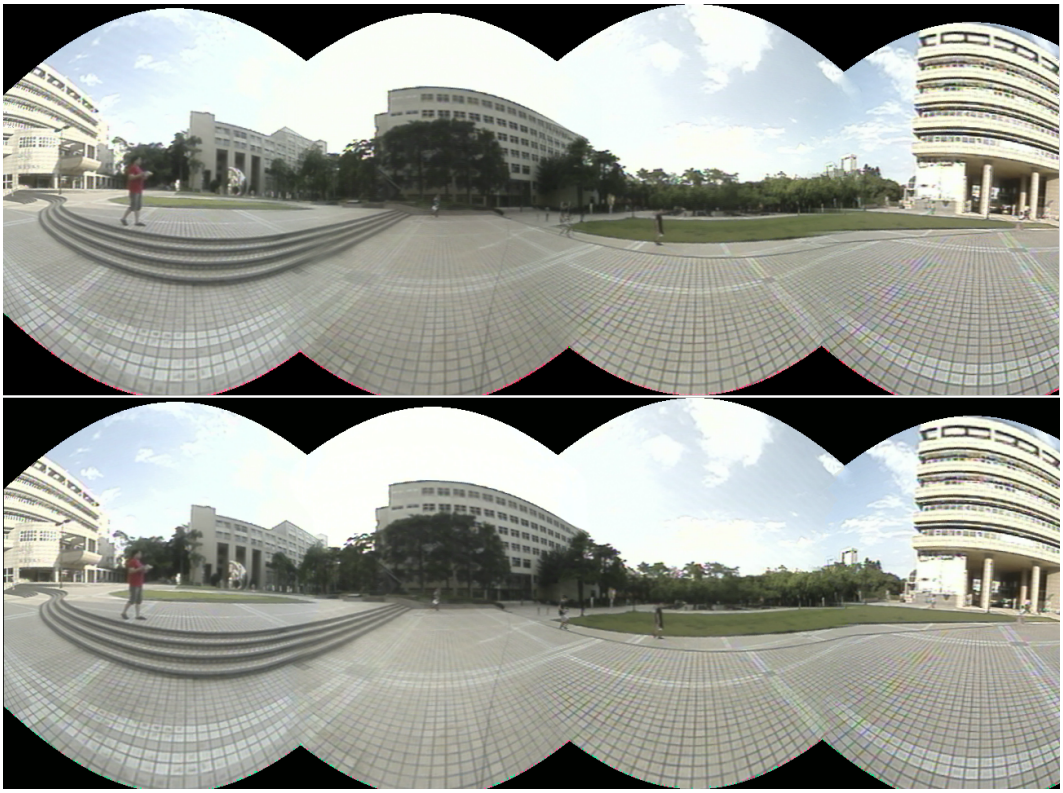


Figure 3.6: Difference between the original (top) and proposed implementation (bottom). Notable differences include dynamic seam and brightness adjustment

3.4.1 Profiling overview table

Table 3.1: Profiling summary.

Name	Original Time (ms)	Proposed Time (ms)	Speed Gain
STITCHTHREAD	153.8778253	71.9367981	110%
Fisheye correction	27.3245352	16.9983821	60%
Interpolating single image	67.2973417	22.9243441	190%
Single dynamic seam	8.1870215	2.7335702	200%
Blending single seam	8.3263934	1.2557025	560%
FPS	15.5	33.1	110%

The full profiling table can be found in Appendix A.

Chapter 4

Discussion

This section will try to elaborate on some of the decisions and will try to clarify the results.

4.1 Data restructure

The performance improvement of the fisheye correction is caused by the restructuring of the data. Instead of using 8 arrays containing separate parts of the needed information, a single struct is stored in a single array containing the 8 elements. This method uses memory much more efficiently.

4.2 Dynamic seam: Last-frame Difference

A very important detail about the usability of LFD is the use of static cameras. If the cameras move slightly, this method, in the current state, will not produce favourable results. However, when applying such constraint, the implementation can be kept simple and fast, with decent quality. Moving objects are very easily detected as long as they have small texture differences and are not too big. Objects bigger than the overlapping area will be harder to detect as they produce less LFD.

Also, as discussed before, by using dynamic programming, noise does not heavily affect the result. Moving objects will always get the highest priority in DP as long as the noise is smaller than the LFD of the object.

4.3 Mutex

By using separate mutexes for specific memory locations, brightness correction, seam calculation, seam stitching, frame switching and blending can be performed in parallel to the capture thread. When creating threads, it is important to keep separate threads from locking each other.

4.4 Interpolation

The biggest speed-up is the result of forward-calculating the bilinear interpolation after cylindrical transformation: only the needed pixel value is calculated. This is possible because the interpolated image is only used once, and no other pixel is needed. The biggest slowdown of interpolation is still cache misses; random access of pixels from memory causes the cpu to stall. Further improvements can possibly be made by implementing the interpolation on GPU or even dedicated hardware.

Comparing the fisheye interpolation (in combination with the RGB conversions) with the custom bilinear interpolation shows a matching speed with the proposed method: 31ms against 23 ms. The original method of custom interpolation however is much slower. This, most likely, is a result of attempting to optimise for quarter-pixel instead of bilinear. Adapting the proposed interpolation method to quarter-pixel should be a matter of increasing the available formulas at 2.3. This, however, will result in an even larger cache miss rate and will slow down the program much more.

4.5 Seam Blending

Occasional ghosting of objects can be reduced by applying blur on the LFD result. This will increase the size of the object difference, so the seam will follow a path further away from the object.

4.6 Slow down of YUV to RGB conversion

No code was changed in the YUV to RGB conversion, but as seen in appendix A, a 1 ms slowdown appeared. On the other hand, a speed-up can be seen for the RGB to YUV conversion. The reason behind this slowdown and speed-up is still unknown, but neglectable.

Chapter 5

Future work

The quality of the dynamic seam depends on the (lack of) movement of the camera. Possible solution for future study can include camera stabilization techniques like the works of Kao et al. (2006); Li et al. (2014), object detection or stereo vision.

The implementation performs interpolation twice for each frame. After performance improvements the custom interpolation appears to be faster than the fisheye interpolation and YUV to RGB to YUV conversions. Future improvement can include implementing the proposed bilinear interpolation for fisheye interpolation or combining both transformations into one and only performing interpolation once.

Further improving performance will be key to maintain real-time stitching, especially when using this method for more than 4 cameras.

Appendix A

Full Profiling Summary

Name	Original Time (ms)	Proposed Time (ms)	Speed Gain
YUV to RGB conversion	6.2911176	7.1819992	-10%
Fisheye correction & interpolation	27.3245352	16.9983821	60%
RGB to YUV conversion	9.3440064	7.2022147	30%
Custom Interpolation	67.2973417	22.9243441	190%
Fisheye & Custom Interpolation	110.504393	54.4928374	100%
Single Dynamic Seam	8.1870215	2.7335702	200%
All Dynamic Seams	8.85695	1.9507258	350%
Panorama stitching	10.5132186	8.1432525	30%
Single Seam Blending	8.3263934	1.2557025	560%
All Seam Blending	9.0487657	1.5555886	480%
YUV to RGB conversion, full image	6.3614559	3.8030368	70%
Stitch Thread	153.8778253	71.9367981	110%
-	-	-	-
FPS	15.5	33.1	110%

Bibliography

Kai Chen Huang, Po Yu Chien, Cheng An Chien, Hsiu Cheng Chang, and Jiun In Guo. A 360-degree panoramic video system design. *Technical Papers of 2014 International Symposium on VLSI Design, Automation and Test, VLSI-DAT 2014*, pages 11–14, 2014. doi: 10.1109/VLSI-DAT.2014.6834863.

GUO Jiun-In, CHANG Hsiu-Cheng, CHIEN Cheng-An, and HUANG Kai-Chen. Optimal dynamic seam adjustment system and method for image stitching. *http://hdl.handle.net/11536/128672*, 2015. URL https://ir.nctu.edu.tw/handle/11536/128672.

Jiaya Jia and Chi Keung Tang. Image stitching using structure deformation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2008. ISSN 01628828. doi: 10.1109/TPAMI.2007.70729.

Jing Li, Wei Xu, Jianguo Zhang, Maojun Zhang, Zhengming Wang, and Xuelong Li. Efficient Video Stitching Based on Fast Structure Deformation. *IEEE Transactions on Cybernetics*, 2015. ISSN 21682267. doi: 10.1109/TCYB.2014.2381774.

Akash Gandhamal and Sanjay Talbar. Evaluation of background subtraction algorithms for object extraction. *2015 International Conference on Pervasive Computing: Advance Communication Technology and Application for Society, ICPC 2015*, 00(c), 2015. doi: 10.1109/PERVASIVE.2015.7087065.

Wen-Chung Kao Wen-Chung Kao, Shou-Hung Chen Shou-Hung Chen, and Pei-Yung Hsiao Pei-Yung Hsiao. Real-Time Image Stabilization for Digital Video Cameras. *APCCAS 2006 - 2006 IEEE Asia Pacific Conference on Circuits and Systems*, 00:1651–1654, 2006. doi: 10.1109/APCCAS.2006.342082.

Lengyi Li, Xiaohong Ma, and Zheng Zhao. Real-Time Video Stabilization Based On Fast Block Matching And Improved Kalman Filter. *2014 Fifth International Conference on Intelligent Control and Information Processing (ICICIP)*, pages 394 – 397, 2014.

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Improving 360-degrees Panoramic Video Stitching

Richting: **master in de industriële wetenschappen: elektronica-ICT**
Jaar: **2016**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

Dendale, Thomas

Datum: **27/06/2016**