Omni-directional stereo for 360° 3D virual reality video

Prashanth Chandran
ETH Zurich
www.ethz.ch
chandranp@student.ethz.ch

Sasha Pagani
ETH Zurich
www.ethz.ch
paganis@student.ethz.ch

Julia Giger
ETH Zurich
www.ethz.ch
jgiger@student.ethz.ch

Abstract

This project is an implementation of an Omni directional stereo (ODS) renderer; 'Jump', that was proposed by Anderson et.al in [2]. To render ODS content, we worked with data that was captured from a camera rig consisting of 10 cameras. The ODS rendering pipeline proposed by [2] consists of the stages of camera calibration, optical flow estimation, exposure correction, view interpolation, and composting. Staying mostly faithful to the rendering pipeline of [2], we introduced a few simplifications owning to the limitations of our camera rig and time considerations. The renderer was implemented from scratch in Python and relies on OpenCV only for image I/O and optical flow calculation. Our ODS renderer is extremely generic, easy to use and can be readily used to render ODS content from an arbitrary rig only given its calibration data.

1. Introduction

An ideal virtual reality environment is one where the user can experience the world in 1) stereo, where eye get an image appropriate to where the user is sees, and 2) in 360 degrees, where the user is free to look in any direction. The rendering of Omni directional stereo content is therefore absolutely necessary for visual immersion.

The primary reason why VR environments have to be rendered in stereo is because of the way in which humans perceive depth. The perception of depth in human stereo or binocular vision is a result of the fusion of images captured by the left and right eyes. In order to render stereo images, we need to capture synchronized frames from two cameras set apart by the interpupillary distance (IPD); which denotes the distance between our two eyes. The IPD on average about 6.4 cm. However, since for applications in virtual reality, we not only require images that are rendered in stereo, but for them to also be omni-directional.

One straight forward method of capturing such videos in 360 would be to use two omni-directional cameras that are separated by the human IPD. This method however, suffers from two limitations. The first problem with such an approach is that the two cameras would see each other. The second and more important problem is that the objects which lie on the line passing through the two camera centers will have no disparity (see figure 1). The absence of disparity makes the perception of depth impossible and therefore, with this simple solution, we would not be able to render videos in omni-directional stereo.

Ideally, what we want is to have a stereo image pair for every orientation of the head (see figure 2). This would result in the capture of a huge number of images. Instead, if we could capture only the central ray from each camera and borrow other rays from the central rays of neighbouring camera positions, we could greatly reduce the number of images we need(see figure 3). Figure 4 illustrates the extension of this approach to 360°.

This report is organized as follows. In section 2, we discuss related work and highlight the contributions of [2]. Section 3 describes how the work was split among the members of our team. Section 4 describes our implementation of the ODS stitching pipeline in detail. In section 5, we present the results obtained using our renderer and discuss its advantages and shortcomings. Our contributions are summarized in section 6.

2. Related work

Several methods in the past have been proposed to tackle the problem of rendering omni directional stereo content. We discuss two such methods that are particularly interesting. 'Omnistereo: Panaromic stereo imaging', proposed in [7], described the multiple view point projection involved in the generation of omni-directional stereo images. The authors used a single rotating camera to capture images in 360 degrees. Another approach is 'MegaStereo' [8].

Both methods [7] and [8], capture omni-directional panaromas with a single rotating camera. This means that they rely on the scene being static or that there is little motion. In [2], the authors propose a new camera rig with 16 cameras that allows for large scene motion. They also use optical flow based view interpolation to synthesize images be-

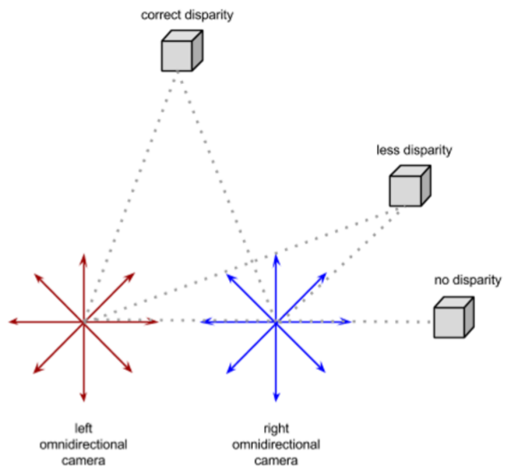

Figure 1. Illustration of two 360° cameras placed next to each other. This figure has been borrowed from [5].

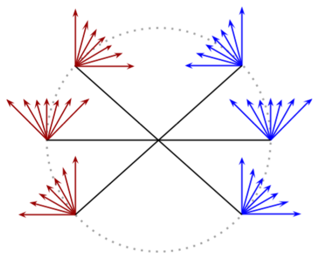

Figure 2. This image depicts the optimal situation, where a full image is captured for each orientation of head and for each eye. The red rays illustrate the left eye and the blue ones the left eye. This figure has been borrowed from [5].

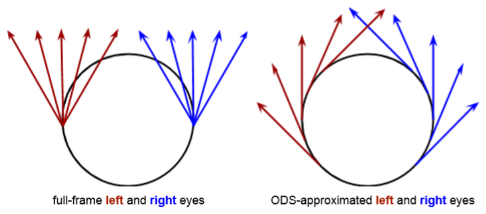
full-frame left and right eyes
ODS-approximated left and right eyes

Figure 3. The image on the left shows represents viewing rays for each eye when the entire image is captured for each viewing direction. The image on the right illustrates the ODS approximation of the left image, where in only the central ray for each camera position is captured. This figure has been borrowed from [5].

tween adjacent cameras in the rig. A key contribution of [2] is that their view interpolation and composting framework, can generate ODS panaromas from only 16 images, while previous methods like [8] required hundreds of images.

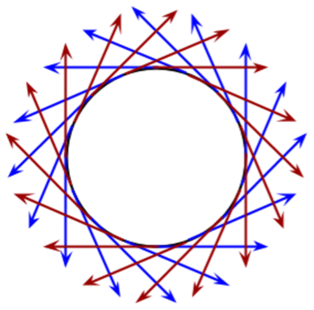

Figure 4. This image shows the viewing ray directions (per eye) when the ODS approximation is done for 360°. The red rays correspond to the left eye and the blue ones to the right eye. This figure has been borrowed from [5].

2.1. Our implementation of *Jump* [2]

In this project, our goal was to implement the ODS stitching pipeline proposed by Anderson et al. [2]. While staying loyal to main steps of their stitching pipeline, we would like to highlight a few differences in our implementation.

The first difference is that we did not have access to the 'Go-Pro Oddessy' rig that was designed and used by the authors of [2] to capture 360 degree content. Instead, we were provided with a dataset that was recorded from a rig of 10 cameras. More details about the deviations of this camera rig from the one used in [2] are provided in section 4.1.

The second is that the authors in [2], proposed a novel optical flow algorithm, that was designed to be optimal for the task of view interpolation in scenes with large motion. Owing to time constraints, we do not implement the optical flow they propose and instead use an off the shelf dense optical algorithm.

Furthermore, we had to simplify the final composting step because we did not have as many images to blend as in [2].

3. Work subdivision

The steps involved in our ODS stitching pipeline is shown in figure 5. Work was divided among our team members as follows.

- **Prashanth Chandran**
  - View interpolation and ODS stitching
  - Experiments with homography based stitching and other visualizations
  - Exposure correction
  - Final report
- **Sasha Pagani**

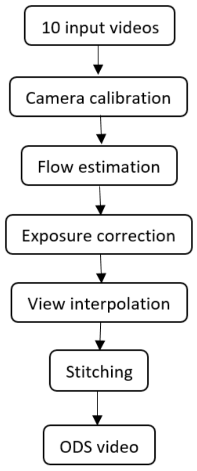

Figure 5. ODS rendering pipeline.

- Optical flow estimation with openCV
- Camera calibration
- Experiments with Unity on the Google cardboard
- Mid-term review presentation
- Final report

- **Julia Giger**
  - View interpolation and ODS stitching
  - Experiments with Unity on the Google cardboard
  - Mid-term review presentation
  - Final report

4. Method

As mentioned before, figure 5 depicts our ODS stitching pipeline. The inputs to our renderer are synchronized videos captured from 10 cameras arranged on a rig. As a first step, these cameras are calibrated to scale using standard structure from motion algorithms [4]. For synthesizing novel views between adjacent cameras, dense optical flow is estimated between temporally consistent frames of neighbouring camera pairs. Camera gains that compensate for differences in camera exposure are then calculated in an exposure correction step. Using the previously estimated optical flow, a novel linear view interpolation, as proposed by [2] is carried out and the images are composted to result in an ODS panaroma.

4.1. Camera rig and calibration

The 10 cameras in the rig were calibrated with the help of 'Kalibr' [1]. The calibration data consists of the intrinsics and the relative extrinsics of each camera. For our rendering purposes, we would need the global extrinsics (the extrinsics of each camera with respect to the origin), as opposed to the relative extrinsics (which is w.r.t location and orientation of the previous camera in the rig). Global extrinsics for each camera was calculated from the relative extrinsics using equation 1.

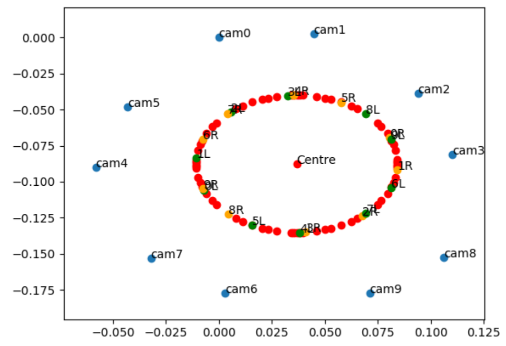

Figure 6. An illustration of the arrangement of cameras on the rig, the viewing circle and the projection of the camera centres onto the viewing circle for the left and right eyes. The points in green on the viewing circle are camera centre projections computed for the left eye and those in orange correspond to the projections for the right eye.

$$GE_i = GE_{i-1} \cdot RE_i^{-1} \quad (1)$$

where GE_i represents global extrinsics (w.r.t to the origin) of camera i while RE_i represents the relative extrinsics (with respect to the previous camera) of camera i. Camera 0 was assumed to be at the origin.

Figure 6 shows how the cameras in our rig are arranged. The circle in red is the viewing circle: i.e. the circle from which the ODS viewing rays emanate. The projections of the camera centres for each eye onto the viewing circle are also shown in figure 6.

Figure 7 displays the image planes captured by each camera in the rig. As one can see, the rig is composed of 5 pairs of stereo cameras. There is a strong overlap in the images captured by a stereo pair and very less overlap between the images captured by two different stereo pairs. For example, there is very good overlap between the images captured by cameras 0 and 1, while there is very little overlap between the images of cameras 1 and 2.

4.2. Flow estimation

When optical flow is computed between temporally consistent frames of a two spatially separated cameras, the horizontal flow approximates disparity. To compute per-pixel optical flow vectors, we use the two frame optical flow estimation algorithm that was proposed by [3]. Figure 8 shows the result of our dense optical flow estimation. In figure

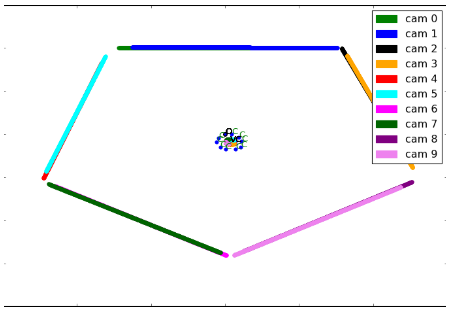

Figure 7. Visualizing the image planes as captured by our camera rig.

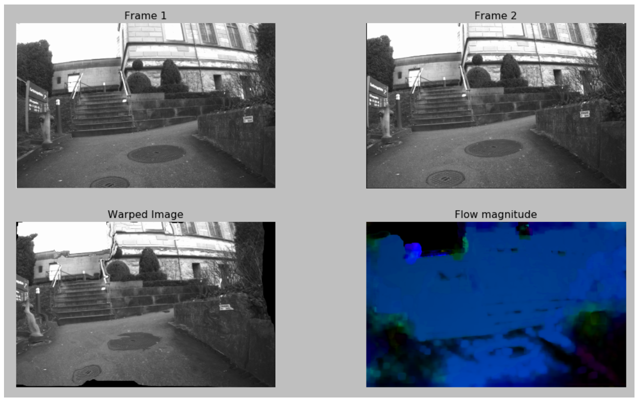

Figure 8. *Top left*: Image captured by camera 0. *Top right*: Image captured by camera 1. *Bottom left*: The image captured by camera 0 has been warped according to the estimated flow. We see that image has rightly been displaced to the left. *Bottom right*: Horizontal flow vectors visualized in HSV color space.

8, optical flow was estimated between two corresponding frames of cameras 0 and 1.

Note that this method of estimating optical flow between two cameras works well only when there is sufficient overlap between the two cameras. When the motion between two neighbouring cameras is large; as is the case between two neighbouring stereo pairs in our rig: for example cameras 1 and 2, the optical flow estimate goes wrong. We discuss these effects in detail in section 5.

4.3. Exposure correction

We implemented the exposure correction algorithm from [2]. First, the region of overlap between every pair of neighbouring images is identified. This is done by detecting correspondences between the two images. The average intensity in the region of overlap of an image with its previous and next image in the camera rig are calculated. Let these intensities be denoted by P_i and N_i respectively. We estimate the gain to be applied to each image g_i, by minimizing

$$\sum_{i=1}^{n}(g_iN_i - g_{i+i}P_{i+i})^2 + \epsilon(1-g_i)^2 \qquad (2)$$

The second term in equation 2 corresponds to a prior which requires gains to be close to 1. Once the gains g_i's have been calculated for each image, the gain for each column is computed as a weighted average of gains computed for neighbouring cameras.

$$g_c = \frac{(\theta_c - \theta_0)}{(\theta_1 - \theta_0)} * g_0 + \frac{(\theta_1 - \theta_c)}{(\theta_1 - \theta_0)} * g_1 \qquad (3)$$

In eq. 3, θ_0 and θ_1 correspond to locations where two cameras on either side of column 'c' map onto the ODS panaroma. θ_c is the location where column 'c' maps onto the ODS panaroma.

4.4. View interpolation

To synthesize views between two neighbouring cameras, we use the linear view interpolation algorithm (eq. 4) as proposed in [2].

$$\theta_p = \frac{(\theta_b - \theta_1) * \theta_0 + (\theta_0 - \theta_a) * \theta_1}{\theta_b - \theta_a + \theta_0 - \theta_1} \qquad (4)$$

where θ_0 and θ_1 are the headings of the two cameras in the ODS stitch. θ_a is the heading of a point in the first camera and θ_b is the heading of the same point; as calculated by optical flow, in the second camera. These angles are illustrated in figure 9.

4.5. Stitching

Once novel views have been synthesized between cameras, they are projected onto the ODS stitch as explained in section 4.4. We use a simple weighted average to blend the images on the final ODS panaroma. Using more sophisticated algorithms for blending will of course yield better results.

5. Results and Discussion

For testing our implementation we used a dataset that was provided by our supervisor.

5.1. Optical flow for dense correspondences

As discussed in section 4.2, estimating optical flow between two cameras with [3] works satisfactorily when the motion between the two cameras is small. In figure 10, we see the result of estimating optical flow between cameras 4 and 5.

However, when we attempt to compute optical flow between images where there is very little overlap: for example the images captured by camera 1 and camera 2 where the

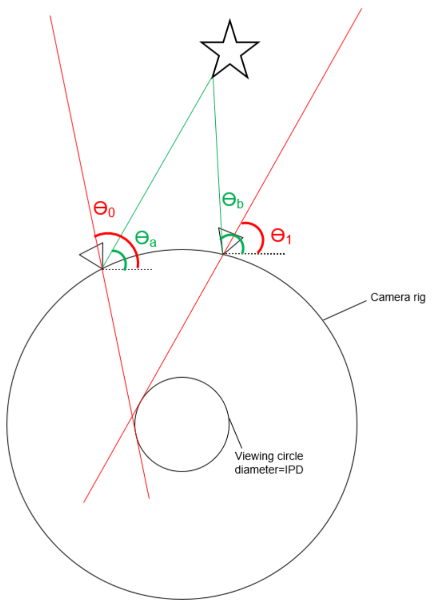

Figure 9. An illustration of the different angles used in the view interpolation formula. The red lines are the central rays of two cameras between which a novel view is being synthesized. The green lines are rays corresponding to the same point as seen from two different cameras.

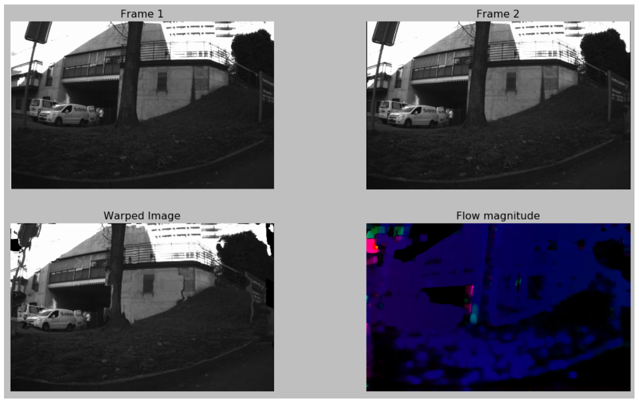

Figure 10. *Top left*: Image captured by camera 4. *Top right*: Image captured by camera 5. *Bottom left*: The image captured by camera 4 has been warped according to the estimated flow. We see that image has rightly been displaced to the left. *Bottom right*: Horizontal flow vectors visualized in HSV color space.

two images look quite different from each other; the optical flow estimate is incorrect. This is as shown in the figure 11.

For obtaining correspondences between such wide baseline images, we tried to use standard feature matching techniques such as SIFT [6]. We found feature matching with SIFT to be much better at finding correspondences in scenes with large motion than the flow based technique used in [2]. Figure

Unfortunately, the overlap between certain cameras in

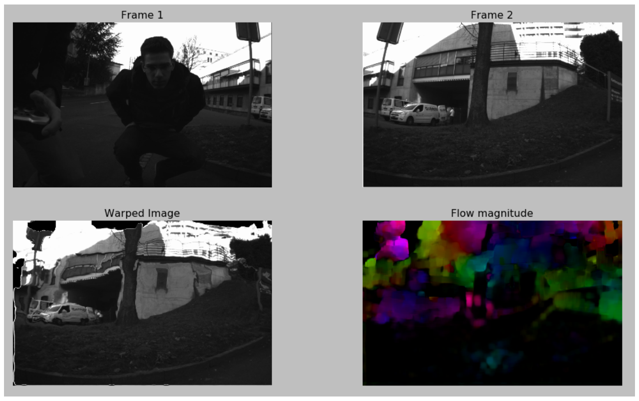

Figure 11. *Top left*: Image captured by camera 7. *Top right*: Image captured by camera 4. *Bottom left*: The image captured by camera 4 has been warped according to the estimated flow. *The flow vectors are incorrect in this case as is seen from the result of the warp. Ideally, image on the right (from camera 4) has to be displaced to the extreme right in the warped image to correspond with the capture from camera 7. Bottom right*: Horizontal flow vectors visualized in HSV color space.

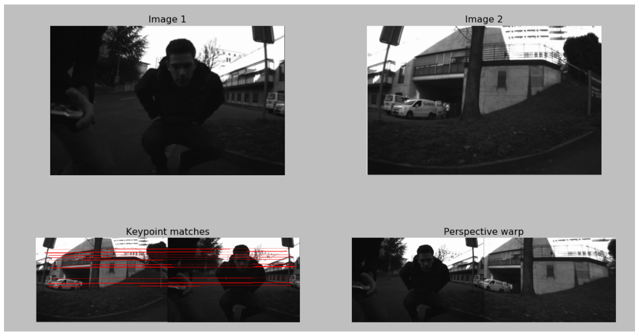

Figure 12. Homography based stitching of two images with SIFT. Top left: Image captured from camera 7. Top right: Image captured from camera 4. Bottom left: Key point matches. Bottom right: The image from camera 7 has been perspective warped onto the image plane of camera 4. In this case, we see that because the key point matches are accurate, we get a good stitch.

our rig is so small that even wide baseline feature matching techniques don't work all the time. This is illustrated in figure 13.

Finding dense correspondences between neighbouring images is vital to being able to synthesize novel views between cameras. The interpolation in equation 4, relies heavily on the fact that we have reliable correspondences for every pixel in an image. Given the problems discussed herein; using optical flow to synthesize views between two stereo camera pairs such as cam0-cam1 and cam2-cam3 etc isn't possible.

Using optical flow to find dense correspondences between cameras is not a problem for the authors in [2] because of the fact that there is sufficient overlap between images captured by two neighbouring cameras on their camera

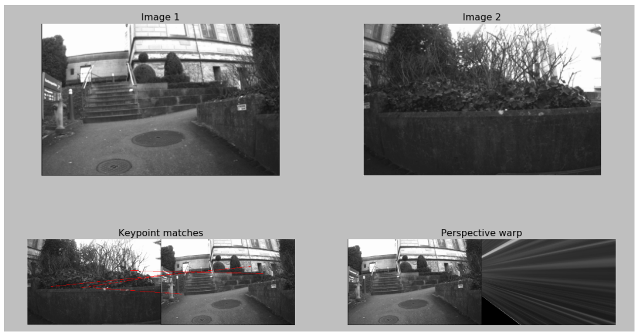

Figure 13. SIFT correspondences between images captured by camera 1 and camera 2. Even SIFT feature matching fails in this case and we get a very distorted homography stitch.

rig. Also, all cameras in their rig are oriented radially outwards as opposed to the stereopairs in our rig. Furthermore, they also use a more sophisticated algorithm for computing dense flow which is capable of handling large motion.

5.2. Content rendered by our ODS renderer

We refer the reader to figure 7 to note that the cameras in our rig are numbered as follows: starting clockwise (0, 1, 2, 3, 8, 9, 6, 7, 4, 5). Following the problems mentioned in section 5.1, we were able to interpolate views only between cameras (0-1, 2-3, 8-9 etc) and not between adjacent stereo cameras pairs i.e. between cameras 1-2, 3-8 etc. As a result, black stripes in the ODS panaroma are to be expected.

Figures 14 and 15 show two variations of the 360 degree ODS stitch produced by our renderer for the left eye. In figure 14, we use every pixel's horizontal flow vector in determining it's position in the ODS panaroma according to eq. 4. In figure 15, we only consider the average horizontal flow of every column in the image to determine the column's final position in the ODS stitch. This results in significantly faster rendering as one would expect.

A careful reader must have observed that the second image stripe from left in figures 15 and 14 seem abnormal. This is because one of the cameras in the rig viz. camera 3, was not functioning as expected. Therefore the images captured by it seem abnormal. These are not artefacts produced during rendering.

ODS content for viewing in a VR device such as the Google cardboard is generally rendered as two images stacked on top of each other. With the given dataset and our rendering pipeline, we were able to generate the result shown in figure 16.

5.3. Implementation details

- Our ODS renderer was written from scratch in Python. All of the multiview projection geometry was implemented. Our renderer relies on OpenCV only for Image I/O operations and for computing the optical flow.

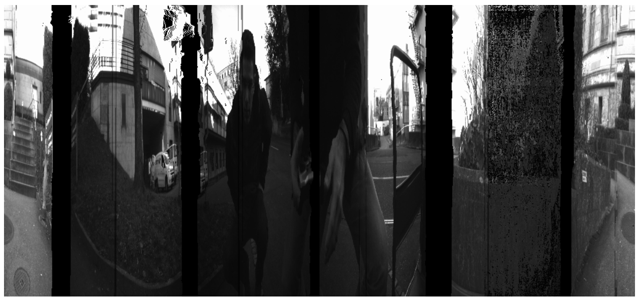

Figure 14. ODS stitch result for the left eye with pixel wise flow for view interpolation.

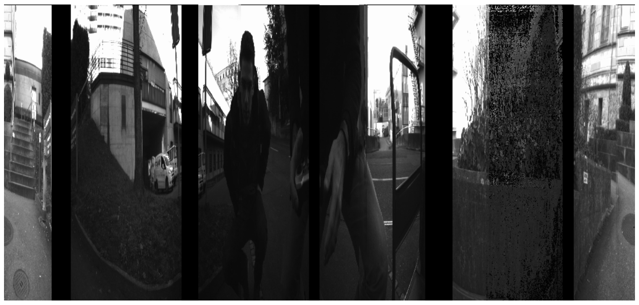

Figure 15. ODS stitch result for the left eye with column wise flow for view interpolation.

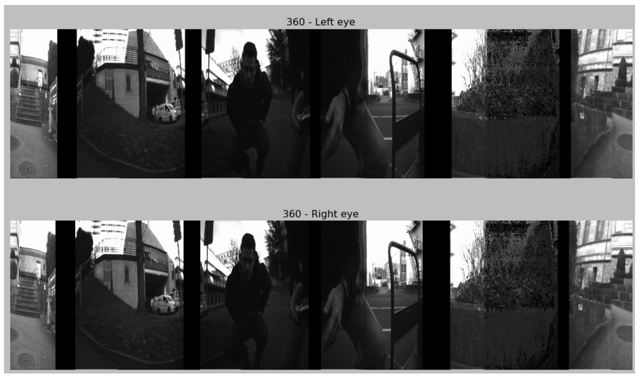

Figure 16. Rendered ODS images for the left and right eye.

- Given a .yaml camera calibration file, our renderer can be extended to any arbitrary camera rig without any modification. This highlights the generic nature of our implementation.

- Our implementation takes approximately 6 seconds to render a 2K panaroma when column wise flow for interpolation is used. It takes about 2 minutes to render a 2K panaroma when per-pixel flow for interpolation is used.

- Though exposure correction has been implemented and tested (refer file *ExposureCorrect.py* in the handout), we weren't able to use it in the final ODS rendering results because of black stripes in the rendered results.

6. Conclusion

In this project, we implemented an ODS renderer based on [2] that can render 360 stereo video. The results obtained by Anderson et al. [2] are much better than ours but this was to be expected for several reasons. Firstly, the authors in [2] designed and used a perfectly radial Odyssey GoPro rig with 16 cameras. Results in this project, on the other hand are rendered from only different 10 cameras with a different rig geometry. We also use an off the shelf optical flow algorithm instead of the much more powerful optical flow algorithm proposed in [2]. Our composting methods are much simpler than those used in [2] too. With improvements to the optical flow and compositing algorithms, and with a camera rig similar to that used in [2], we are confident that our renderer will produce better results.

References

[1] Kalibr. `https://github.com/ethz-asl/kalibr/wiki`. Accessed: 2017-06-25.

[2] R. Anderson, D. Gallup, J. T. Barron, J. Kontkanen, N. Snavely, C. H. Esteban, S. Agarwal, and S. M. Seitz. Jump: Virtual reality video. 2016.

[3] G. Farnebäck. Two-frame motion estimation based on polynomial expansion. In *Proceedings of the 13th Scandinavian Conference on Image Analysis*, SCIA'03, pages 363–370, Berlin, Heidelberg, 2003. Springer-Verlag.

[4] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, New York, NY, USA, 2 edition, 2003.

[5] G. Inc. Rendering omni-directional stereo content, 2016.

[6] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, Nov. 2004.

[7] S. Peleg, M. Ben-Ezra, and Y. Pritch. Omnistereo: Panoramic stereo imaging. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(3):279–290, Mar. 2001.

[8] C. Richardt, Y. Pritch, H. Zimmer, and A. Sorkine-Hornung. Megastereo: Constructing high-resolution stereo panoramas. In *Proceedings of the International Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1256–1263, June 2013.