Gaze Correction for One-to-One Teleconferencing

Third Year Project

Vladislav Milenov Boshnakov
Supervisor: Dr. Aphrodite Galata

A project report presented in the final year of undergraduate studies.

MANCHESTER 1824
The University of Manchester
School of Computer Science
University of Manchester
United Kingdom
3 May 2016
Abstract

Teleconferencing via the Internet has now been established as one of the leading methods when it comes to conducting business or contacting relatives, performing interviews and others. The nature of this technology does not allow, currently, for communicating parties to establish stable eye contact, introducing a level of disconnection from the feeling of a "real-world" conversation. This paper focuses on researching and developing software, that would provide the users with the pleasant experience of viewing their interlocutor’s eyes. There are ways to achieve this effect through physical adaptations of the hardware, however those methods have been discarded as impractical and unscalable. The method used deploys image based rendering techniques in computer vision to build a front-facing image from two original slightly skewed images. This paper concludes that even though theoretically possible, and practically completed to a reasonable extent, this software cannot support commercial application’s demands for quick and clear real-time rendered images.
Acknowledgements

I would like to thank my supervisor, Dr. Aphrodite Galata, for her support and guidance during the development of the project.
Secondly I want to thank the lecturers for teaching me the foundations of Computer Science, and the staff of the School for their impeccable timing and support.
Lastly I would like to thank Valentin Baltadzhiev, without whom this paper would have a few too many apostrophes.
# Contents

0.1 Introduction .................................................. 4
0.2 Project Rationale ........................................... 4
   0.2.1 Assumptions and Hypotheses ......................... 4
   0.2.2 Aim .................................................. 5
   0.2.3 Objectives .......................................... 5
0.3 Background .................................................. 6
   0.3.1 Computer Vision ...................................... 6
   0.3.2 Computerized Stereo Vision ......................... 6
0.4 Focused Research ............................................ 7
   0.4.1 Stereo Calibration .................................. 9
   0.4.2 Image Rectification ................................ 11
   0.4.3 Solving The Correspondence Problem .............. 12
   0.4.4 Gaze Correction ................................... 17
0.5 Design and Implementation ................................. 21
   0.5.1 Requirements Gathering ............................. 21
   0.5.2 System Architecture ................................ 22
   0.5.3 Technologies and Development .................... 27
0.6 Outcomes .................................................... 29
   0.6.1 Camera Calibration .................................. 29
   0.6.2 Image Rectification ................................ 30
   0.6.3 Gaze Correction .................................... 31
   0.6.4 Performance and Evaluation ....................... 33
0.7 Conclusions ................................................ 36
# List of Figures

1. Stereo Geometry. This figure shows two identical cameras with focal lengths $f$ and at distance $b$ from each other. The disparity $D$ of a scene point $P$ at depth $Z$ is $D = \frac{fb}{Z}$.

2. A Pinhole Camera. The 2D image of the object is formed by perspective projection: each ray of light passes through the common center of projection (the pinhole) and intersects a plane behind this center (image plane) at a unique position.

3. Mathematical representation of the projection process.

4. Image Rectification. Focus on the epipoles, as the image is rotated and translated through the application of the homography. Notice in the final image the epipoles lie on the same Y-axis.

5. Epipolar Constraints. The projection of $P$ in the left image $p_L$ corresponds to a point on its epipolar line $e_R$ in the right image.

6. Dynamic Programming. (a) The matrix on which DP is based; each node represents a pair of pixels in the left and right scanlines. A matching cost $M(l,r)$ is associated to each node, with the aim of finding a minimum cost path joining the opposite corners of the matrix. (b) A view of higher granularity, showing the set of moves performed at each node by the algorithm. The circles represent nodes on the graph.
Cyclopian View Generation. (a) A matched point \( p \in P \) is projected orthogonally onto the virtual scanline \( v \). The luminance is equal to the average of the corresponding pixels \( l \) and \( r \) in the left and right images respectively. (b) Generating a "halo" artefact. When treating the occluded segments the same as matched segments a lens like effect appears from incorrect estimation of the luminance at the point of projection. (c) Frontoparallel background extension: the halo is removed if we extend the background at constant disparity. An occlusion point \( p \) is assumed to lie on the extension of the background and is projected orthogonally onto \( v \).

System Architecture Flowchart. 23

The GUI. (1) Shows the minimalistic control GUI. (2) An example of a prompt during the calibration process. (3) Loading a calibration from file. (4) Selecting input images for calibration. These are displayed with default values. 24

Unsuccessful Calibration. This lensing is apparent when the calibration object has not been rotated into a variety of positions during the process. 29

A rectified pair of images. 30

Sample of our gaze correction algorithm’s work on a pair of rectified images. 31

Occlusion Constant Variation. First image has a value of 0.0032. The second: 0.01; The third: 0.25. The background is not extended at constant disparity and occluded regions are intentionally left marked in extreme colours to be more visible. The impact of scene illumination and proper constant selection is clearly visible. 32

(1) Agile Task Board. Early picture of the task board, right after the completion of the first development cycle. The Interface API has been completed and is waiting for QA approval. (2) Multithreading Example. As is clearly visible the running software utilises all available computational resources to complete its task. 35
0.1 Introduction

A video conference is simply the exchange and articulation of information, live, over the Internet. It’s history begins with the German post-office in 1936, where the staff had connected televisions in a few of their offices via a coaxial cable[1]. However, this was expensive, and the technology did not gain traction until the 1990s, with the ”CU-SeeMe” system developed in Cornell University, which provided a real-time videoconferencing over the Internet for Macintosh and IBM PC computers[2]. A long way has been traveled since then in hardware advancements, software development and audio video compression research, with built-in front-facing cameras being the standard for laptops now and conversations over the Internet - commonplace. There is still an inconvenience, that when we’re talking to somebody, due to the displacement and angling of the mounted cameras, we cannot make eye contact. Gaze Correction attempts to change the image of the interlocutors from misaligned to centered, front-facing, so that the illusion of eye contact is achieved, albeit both parties not looking directly into their camera lenses.

0.2 Project Rationale

This section will overview the reasoning behind the selection of this project, introduce the assumptions and hypothesis, outline the aims of the product and summarize the personal and global objectives to-be-achieved.

0.2.1 Assumptions and Hypotheses

This paper makes the following assumptions about the reader:

- They are familiar with basic computer science theory and practices.
- They are familiar with mathematical notation.
- They are familiar with a programming language, and have some programming experience.

This paper establishes the following status-quo:

- The system to-be-built is to be run client-side mainly, as to minimize bandwidth consumption.
• Each party has the necessary hardware installed.

After this we can now introduce the hypotheses:

1. Gaze Correction is possible using image-based rendering techniques.

2. Gaze Correction can be implemented in a way which produces reasonably readable imagery.

3. Gaze Correction can run in real time.

0.2.2 Aim

Based on the assumptions and hypotheses the identifiable aim of this project is to produce a prototype software, which synthesises a synthetic view from a pair of images. The images will be taken from a pair of cameras located at a certain distance from each other, providing with a setup suitable for Stereo Vision techniques to be applied. The images will have to be pre-processed before any synthesizing algorithms can be run. Further optimisation of the system will attempt real-time performance.

0.2.3 Objectives

Global

1. Produce a system that corrects gaze of communicating parties, ensuring stable eye-contact.

2. Optimize system to run swiftly, meeting the demands of real-time commercial software.

Personal

• Investigate and implement current research-level synthetic view generation algorithms.

• Carefully and thoroughly research stereo vision, when applied in the computer vision domain.

• Learn and use current technologies for computer vision application development.
• Implore strong software development practices, such as testing, reflection and agile methodologies.

0.3 Background

Let’s start with laying the foundations to this project. Researching and explaining the basic physical phenomena, how they transfer to computer systems. There will also be a short discussion on the selected technologies.

0.3.1 Computer Vision

The human brain is remarkable at processing visual information, considering the limited space of it’s visual cortex. This is dictated by the complex control attention mechanism of the human cortex, which allows for objects in sight to be prioritized, and thus remain the focus, whereas others discarded as unnecessary information, or background noise[4]. Computer Vision deals with that classification, prioritization and interpretation of visual data, extracting information about the world and creating a perception(model) of it.

0.3.2 Computerized Stereo Vision

As Computer Vision is such a large field with numerous branches, this paper will only be focusing on those, that take us closer to the goals and achieve the aim of the project.

Enabling stereo vision requires a dual camera setup, mounted on the screen(see figure below). This mimics the binocular vision of most animals, with two camera lenses instead of eyes. Evolutionary stereo vision, solves one of the biggest problems in vision - namely - ”How to determine depth?”. In the context of this paper Stereo Vision is used to mean a problem in the area of Computer Vision, which deals with the process of estimating the depth of scene points from the change in their position between two images. Given a 2D representation of the world, like the one captured from a camera, we cannot build a singular 3D model, but rather infinite ones are possible, if there is no loss of information or noise in the original image. Thus we introduce a second image captured synchronously at a slight offset in space. The cameras need to be calibrated and the images rectified(both of these concepts
are explained in detail in later chapters) for us to be able to correctly apply
the triangulation principles of the stereopsis (binocular vision).

In [1] we see a point \( P \) and its two projected images \( Pl \) and \( Pr \), left and right respectively. \( f \) is the focal length, which is the distance between the centre of projection and the image plane. The baseline (physical distance between the cameras) is denoted by \( b \). The distance between \( P \) and the centre of projection is given as \( Z \) and the lateral offset of the cameras (with respect to the left camera) by \( X \). Taking that the X-coordinates of \( Pl \) and \( Pr \) are \( x_L \) and \( x_R \), we can conclude using similar triangles that:

\[
\frac{x_L}{f} = \frac{X}{Z} \quad \text{and} \quad \frac{x_R}{f} = \frac{X + b}{Z}
\]  

(1)

The change in image location is then derived as:

\[
D = x_R - x_L = \frac{fb}{Z}
\]  

(2)

The change or disparity \( D \) is inversely proportional to the depth. As the focal length and baseline remain constants during the image, evaluating \( D \) for each pair of corresponding points provides a direct encoding of the scene depth.\[5\] This evaluation works only on points in the image \( Pl \) and \( Pr \) we know to be projections of exactly the same point in 3-D space, giving rise to a correspondence problem, the solution for which can be seen in Chapter \[0.4.3\]. After we have identified the corresponding points and triangulated them, we can generate a depth map, which gives the synthesizing algorithm a representation of the topology of the image, effectively separating foreground from background, identifying edges and objects at different depths. Once the depth map has been retrieved, an estimation about the colour intensity of each point on that map is made, and then projected onto a new plane - this completes the novel-view synthesis.

### 0.4 Focused Research

This section will cover the research that has been performed into each of the areas necessary for the project’s success, concepts and findings will be explained, as well as briefly discussed. Conclusions will be drawn with arguments made about a selection of a certain path, if a number were presented.
Figure 1: Stereo Geometry. This figure shows two identical cameras with focal lengths $f$ and at distance $b$ from each other. The disparity $D$ of a scene point $P$ at depth $Z$ is $D = x_R - x_L = fb/Z$. 
0.4.1 Stereo Calibration

What we initially have is a dual camera setup consisting of two cameras mounted on top of the screen at a slight distance from each other. We will call this a stereo pair of cameras. We want to ensure the pair is calibrated, that means that we know the cameras’ relative position and rotation from each other in space, as well as their internal properties e.g. focal length. In our case both of these cameras are the same make and model (Logitech HD Webcam C270), and we have them anchored along the same axis. This however is not enough information for us to proceed. Only through knowing all parameters about the hardware and the surrounding environment, can we attempt projection and scene manipulation.

To explain projections we first need to have a short discussion on camera models. The pinhole camera model is a mathematical approximation, which describes the relationship between a point or object in 3D space, and it’s projection around the camera’s origin point in the 2D projection plane. In the model shown in 2 we look at the rays which leave the object and pass through the centre of projection. The light originating at the scene goes through an assumed point - a pinhole, and projects on a projection plane behind the hole at distance $f$. We call $f$ the focal length. The usefulness of this model is that, it abstracts away the physical properties of a camera, e.g. lenses, detectors. Image distortions introduced by the physical components of the cameras are not accounted for by the pinhole model, but they can be corrected in pre-processing. Mathematically we can describe a projection using homogeneous coordinates. Each point $P = [X, Y, Z]$ (defined in 3-dimensional coordinates) is extended by a dummy coordinate $w$ that maps this point to a line through the origin in a space, whose dimension is one higher than the original space. Take our point $P$; introducing this new dummy coordinate, the representation of $P$ is now given by a the set of vectors $[wX, wY, wZ, w]^T$. Homogeneous coordinates do not do anything, and can be considered redundant, however the dimensionally elevated representation allows for expressing a non-linear transformation linearly, using a transformation matrix.

In 0.4.1 can be seen the method in which we take a 3D point in space and project it onto a 2D plane, like in a pinhole camera. The matrix $P$ is called a projection matrix, it is a $3 \times 4$ matrix containing both the intrinsic and extrinsic camera parameters. The intrinsics are:

- the focal length $f$
Figure 2: A Pinhole Camera. The 2D image of the object is formed by perspective projection: each ray of light passes through the common center of projection (the pinhole) and intersects a plane behind this centre (*image plane*) at a unique position.
Figure 3: Mathematical representation of the projection process.

\[
\begin{bmatrix}
u \\
v \\
w
\end{bmatrix} =
\begin{bmatrix}
P
\end{bmatrix}
\begin{bmatrix}
X \\
Y \\
Z \\
1
\end{bmatrix}
\]

- the aspect ratio
- the position of the origin of the image coordinate system

To discover \( P \) we need to calibrate the camera. Calibrating a stereo camera setup, gives us the projection matrices \( P \) of the first camera and \( P_1 \) of the second camera. In \( P_1 \) the position of the origin of the image coordinate system will be given as an offset from the origin position in \( P \) if the calibration has been successful.

Stereo camera calibration is a complicated field and its inner workings fall outside of the scope of this project. The calibration process is handled by the open-source library for computer vision named OpenCV, detailed information on which can be found in Section 0.5.3. What is important for us is that camera calibration requires a physical object, a chessboard pattern grid, to be presented to the calibration software. Numerous images are then taken by the software of the pattern held at different angles and positions. This is done to retrieve multiple references of the calibration points (the inner crossings of the chessboard). The software then calculates the essential matrices for each camera.

### 0.4.2 Image Rectification

Once the physical properties of the system have been discovered and recorded through the process of calibration we want to use them to make our future work in identifying corresponding points and synthesizing views easier. Image rectification is the process of re-projecting the input images onto a common plane parallel to the baseline, with their rows aligned into a frontal parallel configuration. No knowledge of scene geometry is required in order to re-project an image, since only the image plane changes position, while the center of projection and all projection rays remain stationary. The process of rectification can be described by a *homography* \( H \), which is a 3x3 projection matrix. The homography describes a transformation by combining two
major elements rotation and translation. Again, this action will be handled by OpenCV for the most part, as the inner workings are beyond the scope of the project. What is sufficient to be said is that each pixel or point \((x, y)\) will have its position altered by \(H\) in the rectified image, and an intensity value must be computed from the nearest pixels through interpolation.\(^6\) 0.4.2 shows the main stages of the rectification process with relevant abstraction. Notice how corresponding points lie on the same Y-axis. This will become important later.

### 0.4.3 Solving The Correspondence Problem

We’ve been talking about epipoles and corresponding points, but to what extent these are relevant we will explain in this chapter. Look at points \(p_L\) and \(p_R\) are the projections of scene point \(P\) on the left and right image planes respectively. We know that by definition here, but when we are looking at a pair of images we need to construct this picture, and identify that \(p_L\) and \(p_R\) do indeed correspond. Solving this correspondence problem is the truly difficult task of stereo vision. In our analysis we are implicitly assuming that corresponding points have the same intensities in both images, this is equivalent to assuming that the scene is composed of Lambertian surfaces, which are perfectly matte surfaces, whose brightness depends only on the angle of incident light and not on the angle of observation.\(^5\) This obviously may not be true, not only due to the fact that some surfaces are reflective or transparent, but because light from multiple sources may be hitting one of the cameras in a vastly different way from the other, despite their relative closeness.

Other difficulties may present themselves with partially occluded objects (such objects are visible by only one of the cameras). This is a problem that view generation deals with, and is quite influenced by, thus we will discuss it in 0.4.4.

Ignoring occlusions for now, we will introduced the matching approach selected by us in this paper. We will consider large image regions around the pixels, which should contain enough information to yield a definitive match. This approach is called area-based stereo matching\(^8\), and has an advantage that it produces a dense disparity map. Since project’s aim is to produce a synthesised view, which requires a good disparity map, this matching approach suits our needs.

Let’s now consider where can we look for these potential matches. Assume
Figure 4: Image Rectification. Focus on the epipoles, as the image is rotated and translated through the application of the homography. Notice in the final image the epipoles lie on the same Y-axis.
Figure 5: Epipolar Constraints. The projection of $P$ in the left image $p_L$ corresponds to a point on its epipolar line $e_R$ in the right image.
we have two cameras represented by their respective projection centres $C_L$ and $C_R$ and their image planes $L$ and $R$. If we observe a point in the left image, say point $p_L$, where do we search the right image for it’s matching point $p_R$? It so happens that we don’t have to search the entire image, but can restrict ourselves to only one line - the epipolar line $e_R$ corresponding to $p_L$. This dramatically reduces search space by a power of 2 - from 2D to 1D.

To see why the corresponding point $p_R$ must lie on a line, observe that any scene point $P$ projecting to $p_L$ has to lie on the projection ray defined by $p_L$, i.e., the line through $C_L$ and $p_L$. If we take into account that this ray should be visible by the right camera, then $p_R$ must lie on the projection of that ray onto $R$. We call this the epipolar line. In essence the epipolar line is the collection of all possible locations of projections of point $P$ in the right camera, if we know that the projection of $P$ in the left is $p_L$. This is illustrated in. The relationship between points in one image and their corresponding epipolar lines in the other is called epipolar geometry. This can easily be derived if we know the specifics about the positions of $C_L$ and $C_R$, the image planes and the point $P$. Remember that we already extrapolated this data through the process of calibration.

Correspondence is confirmed through knowing the epipolar geometry of the points, which can be represented as a fundamental matrix (a 3x3 matrix). This matrix relates a point $p$ in an image to it’s corresponding epipolar line $e$ like this:

$$Fp = e.$$  

(3)

A point $p = [x y w]^T$ in homogeneous coordinates is the point $(u, v) = (u/w, v/w)$. The line $e = [abc]^T$ is defined with the equation $au + bv + c = 0$. Because we have rectified our images a very simplistic form of epipolar geometry emerges. The rectification process has made our image planes coincide and the X-axes have become parallel to the baseline. This means that our corresponding epipolar lines are horizontal and have the same Y-coordinates. We will call these corresponding scanlines. The stereo matching problem becomes much easier in this simplified geometry, as we only look along identical y-coordinates for matching points.
Figure 6: Dynamic Programming. (a) The matrix on which DP is based; each node represents a pair of pixels in the left and right scanlines. A matching cost $M(l, r)$ is associated to each node, with the aim of finding a minimum cost path($P$) joining the opposite corners of the matrix. (b) A view of higher granularity, showing the set of moves performed at each node by the algorithm. The circles represent nodes on the graph.
0.4.4 Gaze Correction

In this section we will discuss what we understand to be gaze correction, and the key elements that necessary for us to correct the gaze. Observations have concluded, that simply synthesizing a frontoparallel image of a face with the eyes looking straight ahead creates the illusion of constant eye contact, i.e, the subject in the picture seems to be "following you". The observers always see an object depicted in a frontoparallel pose (eg a portrait en face or the frontal view of a torso) as facing them squarely, whatever the angle of view. This is clearly born out by the empirically determined pictorial reliefs and by the judgments of frontoparallel points along horizontal lines\[7\]. Taking this information from a study conducted in the University of Utrecht we determine that a corrected gaze will simply be one, in which your image is projected squarely in a front-facing fashion, as if looking directly into the lens. This takes our problem of correcting the gaze down to three steps:

1. Calculate the depth of each point in the image.

2. Evaluate colour intensities for newly synthesized pixels.

3. Project the image onto a frontoparallel surface behind the screen.

Computing The Depth Map

The research in this section and the methodology taken is largely based on the findings of A.Criminisi and J.Shotton et.al in their 2003 Efficient Dense-Stereo and Novel-view Synthesis for Gaze Correction paper\[3\]. There are two main algorithm approaches to computing the disparity map. In the local methods the emphasis falls onto aggregating a computed matching cost at each step. To extrapolate disparity then, we implore an inverse of the "greedy" ideology where at each step we take the path with a minimal matching cost assigned to it. A limitation of this approach (and many other correspondence algorithms) is that uniqueness of matches is only enforced for one image (the reference image), while points in the other image might get matched to multiple points\[8\].

In the global method we look into finding a disparity function that minimizes global energy. Once this is defined a variety of algorithms may be used to find local minimums. This approach is shown to be NP-hard\[9\], so we will not be using in in this project.

This project’s selected approach is the dynamic programming which can find
the global minimum of independent scanlines in polynomial time [8]. A matrix of pairwise matching costs between corresponding scanlines is constructed. Then the minimum cost path is selected, this minimum cost path is the density map for this pair of scanlines. There are a few problems that arise working with this method:

- Selecting a good matching cost function.
- Inter-scanline consistency.

We will try to address both when constructing our algorithm.

3-Move Dynamic Programming

The entire workings of the algorithm happen in two passes forward and backward. In the forward pass a matrix of size $n \times m$ is constructed with $n$ and $m$ being the sizes of the scanlines in pixels. Considering we have two images from identical cameras we are enforcing consistency by making both images of same size, resulting in $n = m$. This matrix is initialised with values of $+\infty$. Then at each step along the scanline the algorithm aggregates and stores a cumulative matching cost $C$ by the following recurrence:

$$C(l, r) = \min \left\{ \begin{array}{l}
C(l-1, r) + \text{OccConstant} \\
C(l-1, r-1) + M(l, r) \\
C(l, r-1) + \text{OccConstant}
\end{array} \right. \quad (4)$$

Where $C(l, r)$ denotes the cumulative cost of the path from (0,0) to the point (l,r). The three moves permitted are a horizontal and vertical occluded moves and a diagonal matched move (see 0.4.4). 45-degree matches represent a surfaces at constant disparity, occluded moves represent occlusions or surfaces that are not frontoparallel. The occlusion constant mentioned is manually selected. Tests have been performed to maximise the accuracy of this constant, this is shown in Chapter 0.6.4. At each iteration the cost is normalized, the minimum selected and aggregated; a table of backwards links is stored for the backwards phase.

$M(l, r)$ of a pair of pixels using the following windowed Normalised Sum of Square Differences:

$$M(l, r) = \frac{M'(l, r)}{2} \quad (5)$$
with $M'(l, r)$ defined as:

$$M'(l, r) = \frac{\sum_{\Omega}[(I_{pl}^l - \bar{I}_{pl}) - (I_{pr}^r - \bar{I}_{pr})]^2}{\sum_{\Omega}(I_{pl}^l - \bar{I}_{pl})^2 + \sum_{\Omega}(I_{pr}^r - \bar{I}_{pr})^2}$$

(6)

where $I$ denotes intensity. $p_l$ and $p_r$ are pixel positions in the left and right images, respectively; $\Omega$ is an $n \times m$ template of the original image, centered around the pixel we’re looking at; $\delta$ is a 2D variable displacement vector. The bar indicates the mean operator for the intensity values in the template. This matching cost has been chosen as to deal with the inter-scanline consistency problem, which produces streaky artefacts if ignored. The template application enables us to propagate cost across multiple scanlines (especially if the template window is chosen to be taller) without smoothing the disparity map.\(^\text{3}\)

The backwards phase follows the saved links from $(l = \text{End}, r = \text{End})$ to the origin $(l = 0, r = 0)$ and defines the minimum-cost path $P$ as a sequence of visited nodes. Evaluating $P$ results in the discovery of the dense correspondences between pixels in the pair of scanlines. The collection of all $P$s for all scanline pairs in the images gives us the disparity map.

**Cyclopian View Generation**

To synthesize our cyclopian novel-view we will follow the algorithm proposed in the paper by A.Criminisi\(^\text{3}\). It consists of the following steps:

For each pair of scanlines given their matching path $P$:

- For each point $p \in P$
  1. Take the colours $I^l(l)$ and $I^r(r)$ of the corresponding pixels $l$ and $r$ of the left and right scanlines respectively:
  2. compute the average value $\bar{I} = \frac{1}{2}(I^l(l) + I^r(r))$;
  3. project the new pixel orthogonally to the virtual image scanline, into the virtual image point $v$ i.e. $I^v(v) = \bar{I}$.

You will notice that this algorithm applies to matched pixels only and occluded areas must be treated in a slightly more precise manner. If this were applied to all pixels indiscriminately, a halo artefact will appear around occluded regions. To combat this we have made a frontoparallel background assumption about the structure of the world\(^\text{5}\). This means that we can fill
Figure 7: Cyclopian View Generation. (a) A matched point \( p \in P \) is projected orthogonally onto the virtual scanline \( v \). The luminance is equal to the average of the corresponding pixels \( l \) and \( r \) in the left and right images respectively. (b) Generating a "halo" artefact. When treating the occluded segments the same as matched segments a lens like effect appears from incorrect estimation of the luminance at the point of projection. (c) Frontoparallel background extension: the halo is removed if we extend the background at constant disparity. An occlusion point \( p \) is assumed to lie on the extension of the background and is projected orthogonally onto \( v \).
these occluded matches by extending the background at a constant disparity; namely for a right occlusion the intensity of the virtual image will become $I^v(v) = I^l(l)$ as the background information will be held in the left image. For a left occlusion the vise-versa is true (see Figure [0.4.4]).

This concludes our research and exploration into the background and methods used for this project. The next section will cover design decisions and implementation techniques in detail.

0.5 Design and Implementation

Chapter [0.4] introduced us to the principles and research behind Novel-View Synthesis, now we are going to discuss the approach and methodologies applied in the design and implementation of the Gaze Correction Software.

0.5.1 Requirements Gathering

To be able to build a stable working system an identification and listing of it’s workings needs to be produced. This will then be used to develop the software in modules, abstracting away from hardware in each step. Because this system is a prototype of recent computer vision theories, all requirements have been based on the research done.

Functional Requirements

The functional requirements are a list of actions which the system should perform when in use. They are based on user interaction, hardware and software compatibility and performance. We have the following requirements:

- The user must be able to see the camera feeds.
- The cameras must be working at the same time, and display data accordingly.
- The software must be able to record and store images and video.
- The software must be able to work with stored images and video.
- The software must be able to work with live images and video.
The software must prompt the user accordingly when actions are to be taken.

- Camera Calibration modules are required.
- Image Rectification modules are required.
- Visualization of input imagery is required.
- Visualization of output imagery is required.
- Simple operational GUI is required.

**Non-Functional Requirements**

The non-function requirements describe what the system should "be".

- The user should be able to use any two cameras.
- The system should provide adequate controls for the crucial stages of its operation (e.g., how many images to be used in calibration, image display size).
- The functions should be clearly separable for maintainability and flexibility.
- Design should be minimalistic, with the highest available abstraction at the user level.

### 0.5.2 System Architecture

To ensure a steady and successful developmental process a modular paper design was produced, describing the basic blocks, their functionality and the flow of the system. A *top-down* approach was selected when designing the blocks of the system, adding functionality and increasing granularity as necessary. The flowchart in Figure 0.5.2 shows the control flow of the system, with the basic separable modules; the arrows pointing down denote control interactions, while the arrows pointing up denote dependency relations. The details of each module, and the design decisions are outlined and discussed below.
Figure 8: System Architecture Flowchart.
At the highest level the system had to interface the user directly, this implied a development of control GUI, that would highlight the most important features of the system, abstracting away all functionality and information, which was deemed unnecessary. Simple layout and design was selected with buttons that allow access to the three pillar functions. Behind each button input boxes prompt the user to enter vital parameters, which define the way the functions will be fulfilled. Figure 0.5.2 shows how this was put together. The three main branches of the GUI are:

1. Calibrate and Save - This requires the user to enter the size of their chessboard pattern as a tuple, the size of the individual chess square in cm, the number of images to be used in the calibration process and the name of the folder, where this calibration will be stored. After all these parameters are entered the system initiates the calibration.

2. Calibrate and Capture Image - To enable the user to load a calibration of their choice, and to provide them with the flexibility of recalibrating at will, the recording of the images for calibration is set behind this section, separated from the actual calibration process. Here the user is prompted to enter a folder where the calibration was stored, and
the system proceeds to capture and store images for the actual gaze correction.

3. Run Algorithm - This prompts the user to enter the names of the left and right images that will be used in the gaze correction and executes a multithreaded version of the 3-move DP algorithm described in Section 0.4.4.

The GUI prompts come with default values and a short manual is available with the software, ensuring the user has a smooth experience.

Calibration Module

The calibration module is based on a free open-source software wrapper for OpenCV named Stereo Vision developed by Daniel Lee. It consists of two classes:

- StereoCalibration - This class provides an object which stores all calibration parameters for a calibrated stereo pair. It can be initialised through the StereoCalibrator class, or through a stored calibration in memory. This class’ main functionality is to load/store calibration objects and apply rectification to selected images, using it’s instance variables as parameters.

- StereoCalibrator - This class is to be treated as an interface, which acts as a wrapper to OpenCV functions, abstracting away the numerous input and output parameters required when operating with this library. When StereoCalibrator is extended, access to OpenCV’s chessboard corner detection and calibration functions is gained. This Interface operates on and stores values to a StereoCalibration object.

Results from all the functions described in this module were displayed using the hardware interface API discussed at the end of this section.

Algorithm Environment

The algorithm is prototyped using a native to python scripting approach. The algorithm is dependent only on having two input images to work on, and to it’s workings all pre-processing or interfacing is completely irrelevant, which is why it was selected to be abstracted into it’s own module. This
made testing and evaluation easier and more straightforward by minimising overhead and reducing code complexity.

The environment defines these separate functions:

1. Evaluate Matching Cost - Here is encapsulated the process of calculating the matching cost $\mathbf{M}(l, r)$ of two pixels using the SSD described in Equation 5. During the working of the DP algorithm this operation is executed the most, which ratifies it’s own function.

2. Cumulative Cost - This function implements the forward pass of the DP algorithm and produces a disparity map, using the recursive relation described in Equation 4.

3. Backwards Pass - This section’s functionality was meant to be encapsulated into one function, however, to increase the performance of the algorithm, a multithreaded implementation was required at the later stages of development, splitting this part into 2 sub-parts: (1) Finding $\mathbf{P}$ by following the backlinks; (2) Projecting each point of $\mathbf{P}$ onto the virtual scanline. The result of this separation was higher code readability coupled with the flexibility of evaluating multiple matching paths and projecting them onto the same image simultaneously.

4. Multiprocessing Manager - Python has a *global interpreter lock* (GIL), which does not allow native threads to execute bytecode simultaneously. This is because Python’s memory management is not thread safe. In attempt to reach this project’s goals of real-time performance full machine resource use had to be explored. The multiprocessing manager allowed for a workaround GIL. More in Chapter 0.5.3

**Hardware Interface API**

The hardware interface API is loosely based on another part of the StereoVision wrapper written by Daniel Lee. It provides an object encapsulation of the OpenCV camera access modules by initialising the two cameras together in a singular instance. It natively adds the ability to read an image from the cameras and to display it in screens side-by-side. For the purpose of this project, this API was mended to include a video reader and writer, image writer and video stream display capabilities. The wrapper was also trimmed down removing unnecessary functions to reduce the bloating of this code.
0.5.3 Technologies and Development

The platform was developed on the Windows 10 OS, which gave us access to easily installed tools and libraries necessary for the project. The main technologies used are:

OpenCV

OpenCV is an open source computer vision platform released under a BSD license, which makes it free for commercial and academic use. Written on C/C++ with supporting interfaces for Java and Python under the Windows, Linux, Mac OS, iOS and Android\[11\]. It’s use on this project was to facilitate the low level hardware detection and interconnection, and to provide calibration and image rectification facilities.

Python

Python is a scripting language with Object-Oriented support, that coupled with it’s straightforward syntax and popularity makes it an ideal candidate for fast prototyping. Python is developed under an OSI-approved open source license, making it freely usable and distributable, even for commercial use.\[12\] It has many useful libraries which are also freely available, most notably used in this project is the library for mathematical data structures and operations named numpy. It provides accelerated support for matrices and matrix operations, which was crucial to this project’s completion.

Python Multiprocessing

Python’s memory management is not thread safe, because of that developers of the language have introduced a system called *global interpreter lock (GIL)*. GIL blocks multiple native threads from executing bytecode simultaneously. The Python Multiprocessing library implores a Queue and a Queue Manager, which allow for a multiple Python processes to be executed in parallel using the OS’ thread management, effectively sidestepping GIL. Because the algorithm’s approach allows for independent calculation and projection of depth paths, a division of the entire image into memory-safe blocks(the individual scanlines) has been utilised in attempt to fulfil the real-time performance goal. The only requirement to ensure a complete image, without memory loss is to simply join all threads at the end of their execution.
Eclipse and pyDev

Eclipse provides an IDE for nearly every language, but is mostly used for Java and PHP. Integrated with pyDev, a package specifically designed for Eclipse it becomes a powerful tool for Python development.

Miniconda

Conda is a cross-platform package manager and environment manager program that installs, runs, and updates packages and their dependencies, so you can easily set up and switch between environments on your local computer. Miniconda is a lightweight version of Conda, giving us more freedom and less trouble in one package. This was used to install the necessary python packages and keep them up-to-date inside pyDev, and also to easily integrate OpenCV with Python.

StereoVision

Stereo Vision is a wrapper package for OpenCV written in python, that provides access to some of the useful OpenCV interfaces which allow for computer vision to be used with a stereo camera setup. It was written by Daniel Lee. This software was made compatible with an old version of OpenCV and required quite a significant amount of debugging to become operable, it did however provide useful functions e.g. folder interaction, store/load of calibration matrices.
Figure 10: Unsuccessful Calibration. This lensing is apparent when the calibration object has not been rotated into a variety of positions during the process.

0.6 Outcomes

After thorough consideration and application of the research presented in Chapter 0.4 and the development of the gaze correction software platform the next phase of this project is to present the results and evaluations. Many tests were carried out, but a vast majority were not easily machine-quantifiable, and a human peer was required to review and make judgements. We will show the most valuable outcomes and findings, and the most relevant tests in this section.

0.6.1 Camera Calibration

The aim of the camera calibration process was to extrapolate the intrinsic and extrinsic characteristics of the stereo setup. To achieve maximum accuracy a proper object needs to be selected as a physical anchor, the object needs to be flat, with no curves, or in other words - of constant depth, as to not confuse the software by introducing a perceived change in distances between POIs due to a "hill". Two chessboard patterns were chosen for this project with the same chess square sizes, but differentiating in dimensions (6 x 9) and (6 x 6) respectively. There was no observable difference in the calibration quality, so in the later stages the larger one was discarded. Calibration was performed with a differentiating number of input images (between 7 and 30) with variable results. Twenty images seemed to give most satisfiable outcomes over a large sample size, the more variation of the chessboard positioning that is presented in these, the better the software will
calibrate - presenting the object in the same position produces strong lensing as seen in Figure 0.6.1.
The distance we placed the chessboard pattern away from the cameras plays a vital role in the performance of the calibration, because this process uses distance differences of just a few centimeters while calculating the parameters. If the calibration images are placed too far away from the cameras then denoting that a point of interest has changed it’s position and evaluating that change becomes harder. This effect can attributed to the pixel precision of our hardware, and it’s limitations when it comes to resolution.

Finally to be noted is the background and illumination. Relatively darker backgrounds with more varying depth and colour in the scenery produced better results, whereas setting with flatter objects and lack of colour reduced performance. This is attributed to the workings of the calibration software, as it looks for ”edges”, places where intensity and disparity drastically changes to make points of interest and track in subsequent imagery. The lack of these reduces precision naturally.

Interesting observation was made when a spherical object with peculiar specularity was apparent in the background. This seemed to cause a rather profound change in the undistortion maps, which in result warped the image the object was appearing in when rectification was applied.

0.6.2 Image Rectification

Passing the parameters calculated during the calibration process to the rectification software for image remapping, results in a pair of frontoparallel image planes. The images projected onto these planes have been translated and rotated to align their corresponding scanlines(Visible in Figure 0.6.2).
This process is completely reliant on the calibration’s success and even slight errors in detection or illumination oddities create large impacts on this process.

### 0.6.3 Gaze Correction

The DP algorithm implemented generates disparity vectors for each pair of corresponding scanlines. The collection of all these vectors is the disparity map. A visualisation of this has not been made available due to it’s irrelevance, as it only persists during the first phase of the dynamic programming algorithm. Using the windowed template SSD to find the matching costs reduces inter-scanline inconsistencies, thus reducing greatly the streaky artefacting that appears in a straightforward DP. A multitude of tests were performed to estimate the *Occlusion Constant* in the moves, and it turns out, that in our implementation a value of **0.01** to **0.03** depending on scene illumination was the most appropriate (see [0.6.3](#)).

Gaze correction in this project was achieved by incremental construction of an image from two source images, by orthogonal projection onto a frontoparallel plane, resulting in a front-facing image with a corrected gaze. This approach allowed us to build the image scanline-by-scanline as the depth maps were calculated, presenting us with an opportunity to optimize the entire process by giving us an atomic action completely independent from all others.
Figure 13: Occlusion Constant Variation. First image has a value of 0.0032. The second: 0.01; The third: 0.25. The background is not extended at constant disparity and occluded regions are intentionally left marked in extreme colours to be more visible. The impact of scene illumination and proper constant selection is clearly visible.
0.6.4 Performance and Evaluation

After the complete introduction to the background research and the development process, it has come to evaluate the project’s performance, and compile a compelling document proposing solutions to unsolved problems.

Speed

Firstly one of the global goals for this project needs to be addressed - real-time performance. The prototype described in this paper had an original runtime of 8 minutes and 32 seconds on a pair of images of size (240 x 175) and runtime of 26 minutes 16 seconds on size (640 x 480). This clearly was not satisfactory. A number of techniques were implored to optimize performance in attempt to achieve the goal. 2D arrays were replaced by low-level optimized 2D matrices, using the native to Python library numPy, this allowed for some reduction in lookup and write operations, by using matrix multiplication. A complicated rewrite of the DP algorithm was necessary to utilise multithreading. The result of both these approaches reduced computation time almost 8-fold to 67s synthesis using images of size (240 x 175), and to 5 minutes 47 seconds on size (640 x 480). This performance is still not satisfactory, but great progress has been made(observe Figure 14), however due to time constraints these further optimisation techniques were not explored:

- Rewrite the entire software on a mid-level language like C, which gives direct access to hardware and better memory management, which is expected to yield large gains in computation time.

- Matching Cost evaluation function is currently run on a template window which is iterated over. Changes to this practice will include an accumulator window, which will ”drop” a single vertical line of values at the end, and insert a new one at the front as the algorithm moves through the scanline. Remember that this window is centered on the pixel which is being currently looked at.

- Replace the Matching Cost template window with a matrix, and adapt to run the optimization described above, plus make all computations using matrix manipulation techniques.
If these optimizations are made near real-time performance should be attained. However Criminisi reports that the runtime of the algorithm is 0.31 seconds per pair, yielding only a 3.23 frames per second[3]. This clearly shows that the project cannot reach true real-time computation for a video stream without any hardware acceleration.

**Developmental Practices**

In the beginning of the project in a waterfall fashion a list of requirements were produced, and a developmental plan was designed. However that was quickly scrapped for a more agile approach, implementing user stories and short development cycles, which quickly started to give results(see Fig. [14]). Due to the complexity of research and erroneous documentation a strong constraint was placed on implementation time, that coupled with the lack of experience resulted in the disregard of one of the most crucial elements of agile - refactoring and reflection. This led to many bugs in later versions of the code resulting in plentiful irrelevant testing erroneous results and requiring time to be debugged. On the other hand many transferable skills in time management, working under pressure and debugging foreign and native code were gained.

**Testing**

Rigorous testing was performed to determine the values of the occlusion constant, numerous tests were also conducted to ensure the proper functionality of the GUI, calibration and rectification modules. Unit tests were carried out on each module, as python supports a native system to perform these with great success and minimum effort. The vast majority of the testing results had to be reviewed by hand, so a testing cycle of 2 days running different versions of the algorithms for matching cost aggregation, depth map generation and novel-view synthesis were rounded off by a day of evaluating the results by hand. This produced no useful logs, but there are thousands of test images available for observation. On further development more emphasis will be put on testing, and on designing testing software which produces logs with quantifiable results that can be plotted and observed as more readable data.
Figure 14: (1) Agile Task Board. Early picture of the task board, right after the completion of the first development cycle. The Interface API has been completed and is waiting for QA approval. (2) Multithreading Example. As is clearly visible the running software utilises all available computational resources to complete it’s task.
0.7 Conclusions

In conclusion the solution presented in this project successfully satisfied the main aim of producing a gaze correction software. The problem of not achieving eye contact during video conferencing has been solved by generating a novel-view with front-facing interlocutor, as if they were looking directly at the camera lens.

Overall the results from this project are considered to be good, although the computational time greatly exceeds that expected of any reasonable commercial application. This prototype solution written in a high-level language met very significant troubles meeting the real-time rendering goal, even though successful optimizations were carried out to enable complete hardware availability consumption. It can be concluded that the main aim of the project has been achieved and personal goals successfully reached.
Bibliography

[1] Associated Press, "German Postoffice To Use Television-Telephone For Its Communication System", The Evening Independent, St. Petersburg, Fl, September 1, 1934


[10] Python Software Foundation Global Interpreter Lock, Last Accessed 03/05/2016


[12] Python Software Foundation Python, Last Accessed 01/05/2016

[13] Continuum Analytics, Miniconda, Copyright 2014