POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Elettronica

Tesi di Laurea Magistrale

Design and optimization of a VLSI architecture for Depth Image-Based Rendering

Relatori:
prof. Guido Masera
prof. Maurizio Martina

Candidato:
Riccardo Peloso
matricola: 209264

Ottobre 2016

Acknowledgements

Molti mi hanno supportato (e sopportato) durante il mio cammino scolastico, ma sicuramente chi devo ringraziare di più sono i miei genitori che mi hanno sempre permesso di coltivare la mia passione per l'elettronica ponendo la mia istruzione, nonché la mia soddisfazione, come obiettivo principale in questi anni. Ringrazio Francesca, che con Tucson mi ha permesso di vivere innumerevoli momenti felici, obbligandomi ad uscire di casa per impedirmi di perdere me stesso dietro ai libri, condividendo però sia ore di studio e di svago. La sua pazienza e il suo sostegno sono stati fondamentali per arrivare fin qui. Ringrazio i miei compagni, soprattutto Annachiara e Luca, compagni di lavoro e amici pazienti, sempre disponibili ad ascoltare (e assecondare) le mie strambe idee su possibili nuove invenzioni. Molto importanti sono stati i miei coinquilini Alberto, Alessandro e Francesco che mi hanno permesso di vivere un'ottima esperienza torinese in una casa che non dimenticherò mai. Ringrazio i miei amici di Aosta, soprattutto Matteo, Xavier e Alice, compagni di pomeriggi di studi, caffé e uscite la sera. Poi ovviamente mio fratello Stefano, capace di darmi consigli su come procedere, anche su argomenti che non gli competono direttamente, ma soprattutto un buon amico con cui prendere una birra e passare il tempo, assieme a Jiki e Carolina. Un sentito ringraziamento anche ai professori che mi hanno sostenuto in questa crescita personale, soprattutto i miei relatori che mi hanno spronato a entrare nell'interessante mondo del *video processing*. In particolare il professor Martina con cui ho potuto intrattenere piacevoli dialoghi anche a proposito di progetti personali. Non posso poi non mettere i miei parenti e i genitori di Francesca, per cui ormai sono diventato un terzo figlio. E ovviamente tutti i miei amici, soprattutto Louis, Jessica, Luca (Lux), Roberto, Andrea, Elisa, Aurora, Alberto, David e tutti gli altri che non riuscirei ad elencare qui. Grazie.

Abstract

The consumer video market is now offering many solutions to enjoy three dimensional ultra high definition contents (4K UHD-3D 2160p). The recent video compression standards H.264/AVC and H.265/HEVC are offering a multi-view video plus depth (MVD) option, which can efficiently compress many views of the same scene. For each view, a depth map and two camera calibration matrices are provided, which enable a partial 3D reconstruction of the scene. It is possible to mathematically build a virtual camera able to capture the scene from a different perspective. This is called free viewpoint video synthesis. This work is devoted to the optimization of the Depth Image-Based Rendering (DIBR) algorithm in order to reduce the computational effort to a minimum while maintaining a good overall output picture quality. A software model of the proposed algorithm has been built to confirm the quality expectations and a possible Very Large Scale Integration (VLSI) architecture to cope with the high computational requirements of a real-time, UHD-ready, free viewpoint video synthesis is investigated. The data dependency between the warping and the inpaint stage and the high requirements of the preprocessing stage make a frame level pipeline mandatory, leading to a two frame global latency. A novel occlusion-compatible warping order is presented to reduce the memory requirements and the amount of operations per pixel are greatly reduced with respect to the straightforward DIBR view synthesis. A novel joint crack filling, image fusion and inpaint postprocessing stage based on the hybrid median filter (HMF) is presented to reduce the computational burden while maintaining a good output quality.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, the consumer market is more and more demanding of interactive three-dimensional video contents. Many solutions have flourished to meet this demand. Ranging from 3D cameras to 3D screens, it is now possible to enjoy these technologies with a relatively small budget. At the same time, ultra high definition displays are following the same trend. Capable of showing up to 4096x2160 pixels per view at 30 frames per second (or even more), these televisons need a very high bandwidth to meet the consumer demands. Even if 3D video contents can be successfully compressed exploiting spatial and temporal redundancies between views, it is still impossible to transmit (or even produce) a quantity of viewpoints of the same scene capable of a smooth viewer movement around it. The so called Free Viewpoint Video content wants to meet this request and let the final user move the camera freely, without jumping from a viewpoint to the next one. An efficient way to accomplish this is to use a finite set of calibrated cameras, whose mathematical model is well definite, along with the depth map for each pixel and exploit linear algebra and image processing to synthesize the viewpoint of a virtual camera of the scene. It can lower the compression requirements for a 3D video content, offer a smooth transition between real viewpoints and achieve real-time performances. As each frame requires a big amount of computations, there is the need to define a custom architecture capable of ultra high definition free viewpoint rendering at high framerates in real-time. In this work an architectural study of the issue is pursued, from the analysis of the previous works to a mathematical optimization of the basic algorithm, then

to a simplification of the required steps in order to reach an acceptable result up to the definition of a possible Very Large Scale Integration (VLSI) architecture that could be scaled up if higher performances are needed, with an optimized memory utilization. Eventually, the results obtained by the implementation of the described structure in a software environment using *MathWorks MATLAB* are presented and compared to previous literature. The developed software is reported in Appendix A for future reference.

1.1 Historical background

1.1.1 Still images

Ever since the humankind has begun, the need to frame the environment and the thoughts has been of a great concern. The most ancient pictographs known to this day are the ones from *El Castillo Cave* (Spain). These paintings are more than forty thousand years old but they still don't represent a stylization of reality, which will start about ten thousand years later, in the *Chauvet Cave* in France, where the first animal paintings (in particular some bisons) were born. Then the humanity has proceeded towards more and more sophisticated tools for this task, inventing many types of paint and colors. The canvas changed from the mere stone to easier to transport materials, starting from animal skins and wood then evolving to papyrus, paper and fabric. Also the drawing techniques evolved along the ability to represent more complicated subjects. The shading of the colors went to a higher level adding illumination points and shadows. The goal was still a natural looking picture, but a huge step forward was the first perspective conception by the Italian painter *Brunelleschi* in 1415. He found that the edges that are parallel in the real world, when looked by an observer, converge to a precise point that will be later called *vanishing point*. Thus, in order to frame a geometric structure, a *perspective grid* could be drawn on the canvas as a geometrical guide for an easier and better looking painting. The refinement of this technique continued in parallel to the exploitation of the *camera obscura*, a precursor of the modern still camera, as an aid to trace the outline of a painting starting directly from the image projected through a pinhole camera structure. Later on, in 1826, *Joseph Nicéphore Niépce* and *Louis-Jacques-Mandé Daguerre* had the brilliant idea to use a set of new photosensitive materials in order to capture the image focused by the *camera obscura* on a metal plate, creating the first photography. While still monochromatic, this method quickly became a true innovation with respect to paintings and many new photosensitive materials were developed until, thanks to the works of the physicist *James Clerk Maxwell*, the first color image saw its birth through a combination of red, green and blue color filters. The mass production of color films is dated 1935, the *Kodachrome film* developed by *Kodak*, a special layered film capable of capturing red, green and

blue colors in three stacked emulsion layers. It was now possible to capture with a compact camera a color picture in a very fast and convenient way. In 1957 the first digital imaging sensor was invented by a team lead by *Russell A. Kirsch* but the first charge-coupled sensor has seen its birth only twelve years later, in *AT&T Bell labs*, thanks to *Willard Boyle* and *George E. Smith*. Nowadays, millions of pictures are shot on a daily basis and stored (or, better, shared) mainly on the Internet.

1.1.2 Moving pictures

An even more sophisticated way to capture the reality is based on motion pictures. The earlier attempts to give life to an animation are based on spinning disks which could display many frames per second. The first single lens motion picture camera has been patented in 1888 by *Louis Le Prince* who shooted the earliest short silent movie: *Roundhay Garden Scene*. Three years later, *William Kennedy Dickson* (an employee of *Thomas Alva Edison*) presented his *Kinetograph* that evolved later in the *Lumière* brothers' *Cinématographe*, an apparatus that could take, print and project movies. The movie industry emerged and in 1899 *Edward Raymond Turner* produced the first color movie, followed by the famous *Technicolor* system and the addition of sound to the shot movies. Eventually, as a natural consequence, the digital sensors have been exploited as a medium to record a sequence of motion pictures whose duration was mainly limited by the high storage requirements.

1.1.3 Depth illusion

The next natural evolution of the topic is the 3D recording and displaying. In order to perceive the third dimension, the depth, the human brain needs the eyes to see two different points of view of the same scene. An internal sophisticated neural network processing is able to discern the depth of the objects using the stereopsis and parallax information, the distance between the same point of the scene viewed by the two different perspectives. This means that each eye needs a slightly different image of the scene to correctly perceive the depth. The amount of data needed to trick the brain so that it thinks it is not seeing a single flat image but a full three dimensional scene is thus doubled with respect to the two dimensional case. Earlier attempts to represent a 3D scene using two single images is about contemporary to

the invention of the photography, in 1932 by *sir Charles Wheatstone*, even if the topic were already been studied by *Euclid* and *Leonardo da Vinci*. The problem of the stereoscopic systems is that the equipment required is much more sophisticated than for the 2D content entertainment. Nowadays there are many ways to enjoy 3D contents using very ingenious media. There is even the possibility to superimpose in the same picture two images and use a special pair of glasses with passive optical filters to separate the spectrum in two distinct bands, a low pass filter for an eye and a high pass filter for the other. This is called anaglyph rendering. The color rendering can not be good because of the two definite bands. A refinement of this technology is the *Dolby 3D* technology. In this way a pair of special glasses, with two different dichroic lenses, can filter three different subbands of the light spectrum. Each lens can display a slightly different set of red, green and blue colors that can be added to form the final color in an additive mixing way. Each eye will thus see a different image, leading to a stereoscopic vision. Other ways to show 3D contents on the same frame with glasses employ polarized light (for example the *RealD* technology uses circularly polarized filters, one polarization direction per eye) or even glass-free technologies. A remarkable entertainment product using a glass-free approach is the *Nintendo 3DS*, a portable gaming console equipped with a liquid crystal display that uses a parallax barrier to send a different image to each eye. Nowadays there are more and more prototypes of glass-free displays, for example by using a lenticular array to make the user see the scene from different perspectives just changing the point of view. This is called *Multi Autostereoscopic Display* technology. Probably in the future this will be the dominating way to enjoy 3D contents as entertainment. A major flaw for multiple viewpoint television is that there should be one picture for each point of vision. It would be impossible to transmit (or even store) each view in an uncompressed way so there is the need to compress or render the needed point of view of the scene starting from a reference view. Modern video compression standards, like H.264/AVC or H.265/HEVC, are able to encode or decode a multiview video stream with a limited overhead with respect to the 2D case, exploiting the fact that in stereoscopy the content of the left and right frames is similar as they refer to the same scene. It is also possible to compress multiple points of view in a format called MVD (Multi View plus Depth). In this format each point of view has an additional greyscale channel, the depth,

that indicates how far would be a pixel with respect to the camera if reprojected in the 3D world. This additional information, in conjunction with some other camera calibration parameters, can be used to mathematically reconstruct the scene in a true three dimensional coordinate system, as will be demonstrated in the next section.

1.2 The pinhole camera

The simplest camera setup capable of rendering perspective projection is called pinhole camera. Its name derives from the fact that historically it had been built as a box without objective lenses and a small point that is used as an aperture in order to let the light rays enter the chassis and project to the captured image on the camera film. The resulting picture will be a scaled version, upside-down image of the scene. The human eye, apart from the focusing lens, can be seen as a natural extension of this working principle, with the iris being the aperture. The ideal pinhole camera can be mathematically modeled as a way to map a 3D orthogonal coordinate system to a 2D image plane. This is a linear transformation in an homogeneous coordinate system, which is a way to represent the system using an additional dimension that is useful to transform projective spaces in linear ones. This mathematical property is able to delay the division required by the projection operation to the final stage of the computation and exploit linear algebra rules to simplify the calculations. In particular, the parameters required to describe a pinhole camera setup can be condensed in a matrix form, much more flexible to use. The camera is described by a 3x3 matrix, called *intrinsic matrix* $\mathbf{K}$, a position in space (a column vector $\mathbf{t}$) and a rotation matrix $\mathbf{R}$. The intrinsic matrix presents always the following structure

$$\mathbf{K} = \begin{pmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \tag{1.1}$$

The parameters f_x and f_y are the focal lengths of the camera over the x and y dimensions, the γ parameter is a *skew* factor in case of non square pixels and c_x and c_y the principal point (also called image center) position, that despite its name could not be exactly in the middle of the frame. The rotation matrix $\mathbf{R}$ is a 3x3 orthogonal unitary matrix (i.e. $\mathbf{R}^T = \mathbf{R}^{-1}$ and $\det \mathbf{R} = \pm 1$, property that will be useful later for computations), that represents the camera rotation as a rigid body with respect to the full 3D coordinate system and, finally, the translation vector that represents the distance of the camera with respect to the origin of the 3D space.

The projection operation is then given by the following relation

$$\begin{pmatrix} w \cdot u \\ w \cdot v \\ w \end{pmatrix} = \mathbf{K} \left(\mathbf{R} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} + \mathbf{t} \right) \tag{1.2}$$

that maps the 3D coordinates *(X,Y,Z)* to the homogeneous coordinate system of the 2D image. The w parameter is a scale factor that coincides with the distance between the pixel located at *(u,v)* and the center of the camera. It can be also seen as the distance travelled by a ray of light, originated in the center of the camera, in order to reach the corresponding 3D point.
The interesting part of this linear equation is that it can be reversed in order to reconstruct the 3D coordinate starting from the matrices and the u, v and w coordinates.

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \mathbf{R}^{-1} \left(\mathbf{K}^{-1} \begin{pmatrix} w \cdot u \\ w \cdot v \\ w \end{pmatrix} - \mathbf{t} \right) \tag{1.3}$$

These two equations can be merged in one single transformation that maps the points from a frame to a different location on a different frame. It is the same as reprojecting the 2D coordinates into the 3D world and then projecting the points to a new frame, in this case a virtual view. In this thesis the subscript r will be used to indicate a real viewpoint of the scene, captured by a proper camera, while the subscript v will refer to a virtual camera viewpoint, mathematically constructed starting from this linear equation

$$\begin{pmatrix} u'_v \\ v'_v \\ w_v \end{pmatrix} = \mathbf{K}_v \left(\mathbf{R}_v \cdot \mathbf{R}_r^{-1} \left(\mathbf{K}_r^{-1} \begin{pmatrix} u'_r \\ v'_r \\ w_r \end{pmatrix} - \mathbf{t}_r \right) + \mathbf{t}_v \right) \tag{1.4}$$

The fact that this equation does not need any information about the geometry of the scene looses the computational burden of a full 3D rendering pipeline. It is sufficient to know the matrices and the homogeneous coordinates to compute a full 11 degrees of freedom (3 axis rotation, 3 axis translation and 5 independent intrinsic

parameters of the virtual camera) free viewpoint scene rendering. In other words, a raster image with a depth map and some coefficients is able to provide a sufficient amount of information so that a virtual view of the same scene can be reconstructed. The algorithm that employs this technique is called Depth Image Based Rendering (DIBR). This is the reason behind the depth channel in the MVD video compression.

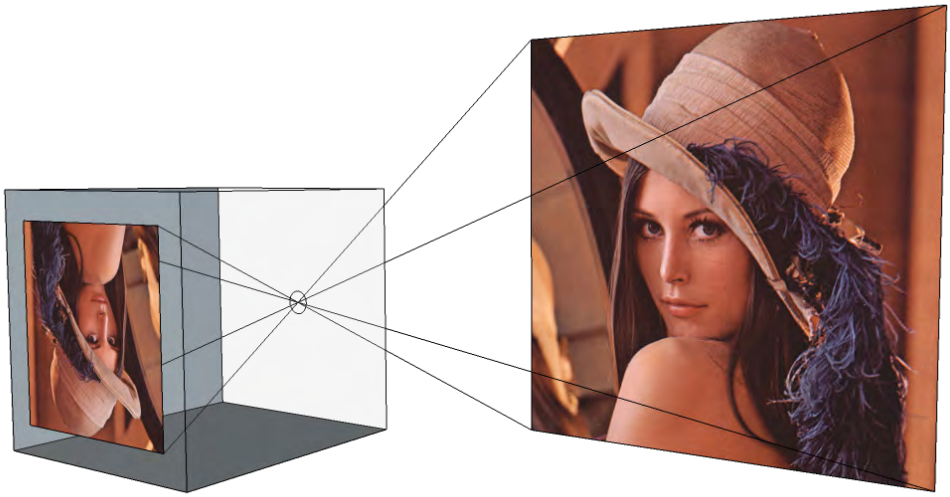


Figure 1.1: Working principle of a pinhole camera

1.3 Previous DIBR works

The reference software for a DIBR implementation is currently developed as a part of *ISO/IEC JTC1/SC29 WG 11 (MPEG) 3D Video* standard by *Nagoya University*, *Thomson Inc.*, *Zhejiang University*, *GIST*, *NTT*, and *TUT/Nokia*. It is called *View Synthesis Reference Software* (VSRS) and it is able to synthesize virtual views for both a simplified case where the cameras are placed in a parallel fashion as a linear array (1D parallel option) or for the general case using the equation (1.4). The 1D parallel setup supposes that between views only horizontal shift is possible so that virtual views can be generated only between the two cameras. The algebra involved is also much simpler but the result is too constrained for a true free viewpoint content generation. The majority of the works in the previous literature about VLSI implementations of this algorithm are focused on this particular case scenario as [1], [2] and [3]. Other custom hardware products use, instead, another algorithm called *Image Domain Warping* (IDW) like [4]. This warping method works only for small horizontal shifts of the camera, it is thus similar to the 1D parallel DIBR approach. Eventually, there are few works about the general DIBR algorithm. The most impressive one is [5] that is able to perform both the DIBR options (obviously the 1D parallel can achieve higher performances) and it is equipped by a High Profile MVD H.264/AVC video decoder. This ASIC is able to output a virtual camera view with a 4096x2160 resolution at a frame rate of 30 frames per second. It exploit some shortcuts to shrink the memory requirements (both size and bandwidth), first of all an occlusion compatible scanning order of the pixels. Other works are mainly focused on software optimizations of the algorithm. The recent trend is to shift the workload from the computer CPU to a GPGPU (General Purpose Graphic Processing Unit) as it is able to process many pixels in parallel thanks to the high amount of computing cores and memory bandwidth. With some optimizations, the real time requirement can be achieved for lower resolution pictures. In the conclusive section of this work, the proposed architecture will be compared to some of these works in order to highlight advantages and disadvantages of the various solutions.

1.3.1 View Synthesis Reference Software

The VSRS analysis is needed to understand further parts of this work. It is composed by many sequential steps. Here only the general camera setup case will be described as it is more interesting for this work. A key role in this software is occupied by OpenCV, Open Source Computer Vision, an impressive C/C++ library optimized for real-time computer vision applications. Apart from that, the first part of the software is devoted to matrix preprocessing. The software internally calculates the camera parameters from correspondences between the two views using the OpenCV function *cvFindHomography*, which uses the singular value decomposition technique to extract the various camera calibration parameters. In this work they are assumed to be embedded in the header of the video stream so that they are already available for later processing. The VSRS creates 256 3x3 homography matrices, one for each possible depth level, an alternative way to perform the equation (1.4). It then proceeds then applying a series of filters to the input depth channels to minimize the related errors. Next it starts the forward warping: the depth maps are warped to the virtual camera point of view using the homography matrices. The resulting depth maps present quantization errors similar to impulsive noise that is removed principally by the median filter. The cleaned virtual depth maps are then re-warped to the input views in order to pick the color channels by interpolation. This step can be avoided by directly warping the input color channels by forward warping but the output quality will suffer. This is the highest trade-off between quality and computation complexity. Once the two backward warped views are available, the two images are fused in a single output frame exploiting the information from both the views by alpha blending. Some regions cannot provide texture information. They can be either zones outside the input frame or occluded zones. An occlusion is generated by a foreground objects casting a sort of shadow on the foreground parts, as they prevent the light rays coming from the background from hitting the camera objective. In this case there are two solutions: if the other warped view can give information about the background objects it should be used to fill the occluded zone, but if in both the views the background remains unfilled there is the need of an inpaint step. In both cases VSRS selectively dilates the remaining zones toward the background direction as there could be border artifacts. This behaviour will

be better described in the next chapter. VSRS uses both the virtual view depth channel and the warped color channels to inpaint the occluded zones. It calculates the local depth gradient and looks for patches in the neighborhood of the border for zones sharing similar textures. When the patch has been selected, that zone of the occlusion is filled with the selected patch an then the process is repeated until all the occluded zones have been inpainted. This is difficult to accomplish in real time, mainly because of the iterative nature of the process that is very difficult to parallelize. In fact, the solution proposed here tries to find a solution for the forward warping resampling artifacts, the image fusion and the inpaint at the same time with a simplified process that does not involve the depth channel nor the calculation of the gradient. Obviously the output quality is inferior than the VSRS approach but it can reach the real-time requirement with a simpler hardware structure.

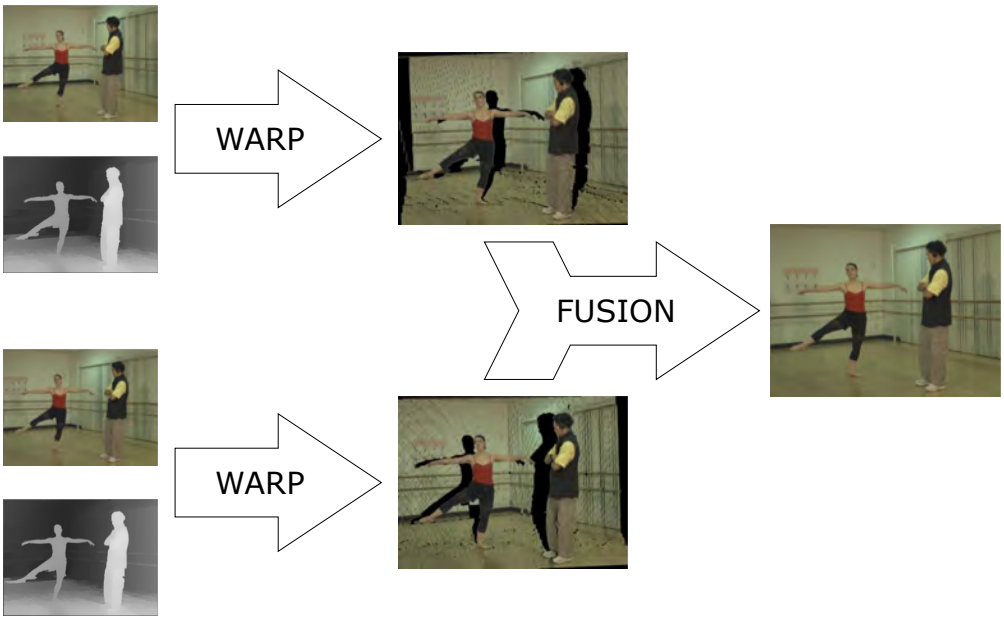


Figure 1.2: Forward view synthesis

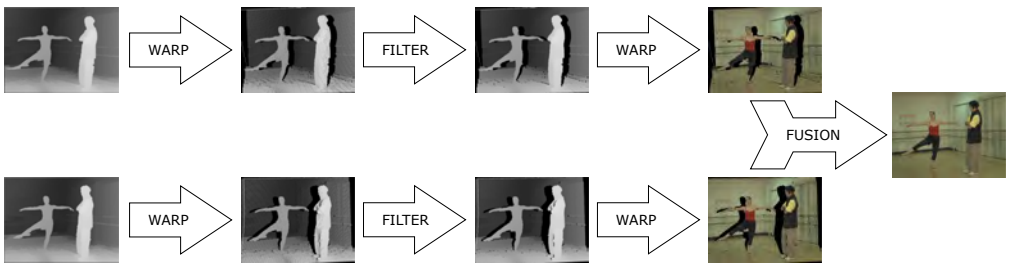


Figure 1.3: Backward view synthesis

Chapter 2

Proposed optimizations

This chapter presents the logical steps behind the development of a custom tailored algorithm to achieve the general mode free viewpoint with the smallest footprint possible. Many optimizations have been found by analyzing at first the most time and memory consuming step, the DIBR warping engine, followed by the remaining parts, the preprocessing and the postprocessing stages.

2.1 Forward warping engine optimization

The basic DIBR algorithm is slow and presents a heavy computational burden. The forward warping only algorithm is lighter to execute with respect to the one comprising the backward warping step but the final image quality suffers. It is mandatory, in order to achieve real-time performances, to exploit some algebraic transformations to reduce the intermediary word width and thus save silicon real estate. In this section these modifications are presented along with the rationale behind them. The naïve implementation of the algorithm requires for each warping stage various matrix-vector multiplication. The result of this warping is represented in an homogeneous coordinates system so a further division is required to find the destination of each pixel in the target frame. In particular, in the forward approach (which will be the ground for the derivation of the proposed architecture), for every input pixel, the resulting location in the arrival frame can be computed starting

from

$$\begin{pmatrix} u_v \\ v_v \\ 1 \end{pmatrix} z_v = \mathbf{A} \begin{pmatrix} u_r \\ v_r \\ 1 \end{pmatrix} z_r + \mathbf{b} \tag{2.1}$$

where

$$\mathbf{A} = \mathbf{K}_v \mathbf{R}_v \mathbf{R}_r^{-1} \mathbf{K}_r^{-1} \tag{2.2}$$

$$\mathbf{b} = \mathbf{K}_v \left(\mathbf{R}_v \mathbf{R}_r^{-1} \mathbf{t}_r - \mathbf{t}_v \right) \tag{2.3}$$

and

$$z = \frac{1}{\frac{d}{255}\left(\frac{1}{Z_{near}} - \frac{1}{Z_{far}}\right) + \frac{1}{Z_{far}}} = \frac{255 \cdot Z_{near} \cdot Z_{far}}{d\left(Z_{far} - Z_{near}\right) + 255 \cdot Z_{near}} \tag{2.4}$$

The matrix **A** and the vector **b** are constant for each couple of input-output frames, u and v are the coordinates of the pixel in the image coordinates system and d is the disparity value for each pixel in an unsigned 8 bit value, where the higher values are closer to the camera and the lower values are the most distant ones. This is usually the most resource intensive part of the whole DIBR algorithm as many operations have to be performed pixel by pixel. For example, a fullHD frame (resolution of 1920x1080 pixels) contains about 2 millions pixels that have to be processed in real-time. The full matrix-vector multiplication and the full length division is unacceptable both for timing and hardware area requirements. The optimization of this part is a crucial step to achieve low output latencies without too much hardware overhead. The first optimization is at the algebraic level, then a re-arrangement of the operators will shorten the required word length and, finally, a pipeline and a resource duplication will lead to the required results.

2.1.1 Algebraic manipulations

The first part that needs to be fixed is the multiplication by z_r. In previous works, like [1], the fact that the disparity term d takes only 256 possible values is used to compute and store in a look-up table all the corresponding 256 values of z. This requires many calculations and a table that in reality are not required to find the values of u_v and v_v. In particular

$$\begin{pmatrix} u_v \\ v_v \\ 1 \end{pmatrix} z_v = \mathbf{A} \begin{pmatrix} u_r \\ v_r \\ 1 \end{pmatrix} z_r + \mathbf{b} = z_r \left[\mathbf{A} \begin{pmatrix} u_r \\ v_r \\ 1 \end{pmatrix} + \mathbf{b} \cdot \frac{1}{z_r} \right] \tag{2.5}$$

that is the same as writing

$$\begin{pmatrix} u_v \\ v_v \\ 1 \end{pmatrix} \frac{z_v}{z_r} = \mathbf{A} \begin{pmatrix} u_r \\ v_r \\ 1 \end{pmatrix} + \mathbf{b} \cdot \frac{1}{z_r} \tag{2.6}$$

The use of homogeneous coordinates helps getting rid of the multiplication by z_r moving it to product of $\mathbf{b}$ and the reciprocal of z_r, which can be calculated using the equation (2.4) as

$$\begin{aligned} \frac{1}{z_r} &= \frac{d\,(Z_{far} - Z_{near}) + 255 \cdot Z_{near}}{255 \cdot Z_{near} \cdot Z_{far}} = \\ &= \frac{1}{255 \cdot Z_{near} \cdot Z_{far}} \cdot [d\,(Z_{far} - Z_{near}) + 255 \cdot Z_{near}] \end{aligned} \tag{2.7}$$

The term

$$\frac{1}{255 \cdot Z_{near} \cdot Z_{far}} \tag{2.8}$$

is a constant, this means that it is possible to exploit another time the homogeneous

coordinates and write

$$\begin{pmatrix} u_v \\ v_v \\ 1 \end{pmatrix} \frac{z_v}{z_r} = \frac{1}{255 \cdot Z_{near} \cdot Z_{far}} \left\{ 255 \cdot Z_{near} \cdot Z_{far} \cdot \mathbf{A} \begin{pmatrix} u_r \\ v_r \\ 1 \end{pmatrix} + \right. \\ \left. + \mathbf{b} \cdot [d\,(Z_{far} - Z_{near}) + 255 \cdot Z_{near}] \right\} \tag{2.9}$$

so that

$$\begin{pmatrix} u_v \\ v_v \\ 1 \end{pmatrix} \frac{z_v}{z_r} \cdot (255 \cdot Z_{near} \cdot Z_{far}) = \left\{ 255 \cdot Z_{near} \cdot Z_{far} \cdot \mathbf{A} \begin{pmatrix} u_r \\ v_r \\ 1 \end{pmatrix} + \right. \\ \left. + \mathbf{b} \cdot [d\,(Z_{far} - Z_{near}) + 255 \cdot Z_{near}] \right\} \tag{2.10}$$

There is still a problem in this form. The product of Z_{near} and Z_{far} can lead to a very big number, resulting in a possible overflow. To limit this behaviour it is possible to collect in the right-hand side of the equation Z_{near} or Z_{far}. Z_{near} is the best candidate as it removes more terms than its counterpart, leading to:

$$\begin{pmatrix} u_v \\ v_v \\ 1 \end{pmatrix} \frac{z_v}{z_r} \cdot (255 \cdot Z_{far}) = \left\{ 255 \cdot Z_{far} \cdot \mathbf{A} \begin{pmatrix} u_r \\ v_r \\ 1 \end{pmatrix} + \right. \\ \left. + \mathbf{b} \cdot \left[d \left(\frac{Z_{far}}{Z_{near}} - 1 \right) + 255 \right] \right\} \tag{2.11}$$

It is now possible to define

$$\alpha = \frac{z_v}{z_r} \cdot (255 \cdot Z_{far}) \tag{2.12}$$

as the scale factor of homogeneous coordinates and avoid expensive divisions. Sadly,

the numerical ranges of $\mathbf{A}$ and $\mathbf{b}$ are unpredictable as they depend on the physical camera setup. This means that it is not worth trying to simplify also Z_{far} or 255 from the equation.

It is instead possible to incorporate the various terms in a more compact form:

$$\alpha \begin{pmatrix} u_v \\ v_v \\ 1 \end{pmatrix} = 255 \cdot \mathbf{C} \begin{pmatrix} u_r \\ v_r \\ 1 \end{pmatrix} + \mathbf{d} \cdot d + 255 \cdot \mathbf{b} \tag{2.13}$$

where

$$\mathbf{C} = Z_{far} \cdot \mathbf{A} \tag{2.14}$$

and

$$\mathbf{d} = \left(\frac{Z_{far}}{Z_{near}} - 1 \right) \cdot \mathbf{b} \tag{2.15}$$

It is important to notice how, apart from the Z normalization, all the divisions are avoided. This means that only where there is the need to normalize the homogeneous coordinates there is the need of a true divider.

Another interesting thing to point out is that the terms $\mathbf{C}$ and $\mathbf{b}$ can be merged because the vector that multiplies $\mathbf{C}$ has *one* as third value. It is thus possible to define a matrix $\mathbf{E}$ as

$$\mathbf{E} = \begin{pmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} & \mathbf{C}_{1,3} + \mathbf{b}_1 \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} & \mathbf{C}_{2,3} + \mathbf{b}_2 \\ \mathbf{C}_{3,1} & \mathbf{C}_{3,2} & \mathbf{C}_{3,3} + \mathbf{b}_3 \end{pmatrix} \tag{2.16}$$

so that the whole equation becomes

$$\alpha \begin{pmatrix} u_v \\ v_v \\ 1 \end{pmatrix} = \mathbf{E} \begin{pmatrix} u_r \\ v_r \\ 1 \end{pmatrix} + \mathbf{d} \cdot d \tag{2.17}$$

The matrix $\mathbf{E}$ and the vector $\mathbf{d}$ don't change through the entire frame.

2.1.2 Incremental computation

In a real case scenario, adjacent pixels share similar u_r and v_r coordinates. It is possible to define, for the current pixel i, $u_r^{(i)} = u_r^{(i-1)} + \Delta u$ and $v_r^{(i)} = v_r^{(i-1)} + \Delta v$ so that

$$\alpha \begin{pmatrix} u_v^{(i)} \\ v_v^{(i)} \\ 1 \end{pmatrix} = \mathbf{E} \begin{pmatrix} u_r^{(i)} \\ v_r^{(i)} \\ 1 \end{pmatrix} + \mathbf{d} \cdot d^{(i)} = \mathbf{E} \begin{pmatrix} u_r^{(i-1)} + \Delta u \\ v_r^{(i-1)} + \Delta v \\ 1 \end{pmatrix} + \mathbf{d} \cdot d^{(i)} \tag{2.18}$$

An important thing to notice is that the values of Δu and Δv are exactly equal to *one* moving between neighborhood pixels. It is thus possible to recycle the value

$$\mathbf{E} \begin{pmatrix} u_r^{(i)} \\ v_r^{(i)} \\ 1 \end{pmatrix} \tag{2.19}$$

from the previous result and add a precomputed constant at each iteration, avoiding the need of new multiplications.
Now that an alternative matrix formulation of the problem has been found, it is possible to normalize the homogeneous coordinate system and finally find the true cartesian coordinates in the image plane

$$u_v = \frac{\alpha u_v}{\alpha} = \frac{\mathbf{E}_{1,1} \cdot u_r + \mathbf{E}_{1,2} \cdot v_r + \mathbf{E}_{1,3} + \mathbf{d}_1 \cdot d}{\mathbf{E}_{3,1} \cdot u_r + \mathbf{E}_{3,2} \cdot v_r + \mathbf{E}_{3,3} + \mathbf{d}_3 \cdot d} \tag{2.20a}$$

$$v_v = \frac{\alpha v_v}{\alpha} = \frac{\mathbf{E}_{2,1} \cdot u_r + \mathbf{E}_{2,2} \cdot v_r + \mathbf{E}_{2,3} + \mathbf{d}_2 \cdot d}{\mathbf{E}_{3,1} \cdot u_r + \mathbf{E}_{3,2} \cdot v_r + \mathbf{E}_{3,3} + \mathbf{d}_3 \cdot d} \tag{2.20b}$$

From now on, only the u_v term will be discussed as the v_v one is very similar to it apart from some matrix indices.

As said before, a large part of the numerator and of the denominator is already computed for previous pixel positions so that the only true products that must be performed are $\mathbf{d}_1 \cdot d$ and $\mathbf{d}_3 \cdot d$. These can be reduced to only one product per coordinate by rewriting the expression as

$$u_v = \frac{\mathbf{d}_1}{\mathbf{d}_3} \cdot \frac{\frac{\mathbf{E}_{1,1}}{\mathbf{d}_1} \cdot u_r + \frac{\mathbf{E}_{1,2}}{\mathbf{d}_1} \cdot v_r + \frac{\mathbf{E}_{1,3}}{\mathbf{d}_1} + d}{\frac{\mathbf{E}_{3,1}}{\mathbf{d}_3} \cdot u_r + \frac{\mathbf{E}_{3,2}}{\mathbf{d}_3} \cdot v_r + \frac{\mathbf{E}_{3,3}}{\mathbf{d}_3} + d} \tag{2.21}$$

It is important to stress out that the upper case parameters are constant for all the frame so operations between them can be performed once for all the frame in a preprocessing stage and their results stored in a Look-Up Table (LUT) or in dedicated registers for fast retrieval during the warping stage.
At this point it is also possible to exploit the partial fraction decomposition and write

$$u_v = \frac{\mathbf{d}_1}{\mathbf{d}_3} \cdot \left(1 + \frac{\left(\frac{\mathbf{E}_{1,1}}{\mathbf{d}_1} - \frac{\mathbf{E}_{3,1}}{\mathbf{d}_3}\right) \cdot u_r + \left(\frac{\mathbf{E}_{1,2}}{\mathbf{d}_1} - \frac{\mathbf{E}_{3,2}}{\mathbf{d}_3}\right) \cdot v_r + \left(\frac{\mathbf{E}_{1,3}}{\mathbf{d}_1} - \frac{\mathbf{E}_{3,3}}{\mathbf{D}_3}\right)}{\frac{\mathbf{E}_{3,1}}{\mathbf{d}_3} \cdot u_r + \frac{\mathbf{E}_{3,2}}{\mathbf{d}_3} \cdot v_r + \frac{\mathbf{E}_{3,3}}{\mathbf{d}_3} + d} \right) \tag{2.22}$$

in order to remove d from the numerator of the fraction This will potentially lead to a multiplierless formulation of the problem like

$$u_v = \frac{\mathbf{D}_1}{\mathbf{d}_3} + \frac{\left[\frac{\mathbf{d}_1}{\mathbf{d}_3}\left(\frac{\mathbf{E}_{1,1}}{\mathbf{d}_1} - \frac{\mathbf{E}_{3,1}}{\mathbf{D}_3}\right)\right] u_r + \left[\frac{\mathbf{d}_1}{\mathbf{d}_3}\left(\frac{\mathbf{E}_{1,2}}{\mathbf{d}_1} - \frac{\mathbf{E}_{3,2}}{\mathbf{d}_3}\right)\right] v_r + \left[\frac{\mathbf{d}_1}{\mathbf{d}_3}\left(\frac{\mathbf{E}_{1,3}}{\mathbf{d}_1} - \frac{\mathbf{E}_{3,3}}{\mathbf{d}_3}\right)\right]}{\frac{\mathbf{E}_{3,1}}{\mathbf{d}_3} u_r + \frac{\mathbf{E}_{3,2}}{\mathbf{d}_3} v_r + \frac{\mathbf{E}_{3,3}}{\mathbf{d}_3} + d} \tag{2.23}$$

There is a numeric problem with this result with respect to the previous one. This time the arrival coordinate is obtained through the sum of two parts. This means that the numerical range of the division is unpredictable as it depends on the first term. While the result of the equation (2.21) has to be in a range defined by the width and the height of the output frame (which is usually the same as the input frame), the result in the equation (2.23) does not present a fixed output range and, usually, needs a signed division to balance the first term of the sum. In other words the hardware divider needed in (2.21) should be no wider than 14 bit (for subpixel warping precision) and, as the result cannot be negative, the sign check is simpler to perform. The drawback is the presence of at least one multiplier. The (2.23) needs more complicated precomputed terms and the divider is wider but there is no need of multiplications. As the divider, in particular a signed one, is a rather expensive piece of hardware, the form in (2.21) should be preferred.

2.1.3 Numerical-wise optimization

There is a third way to optimize the equation (2.21) to have a result similar to the (2.23) but with a smaller first term for the sum. It is based on the fact that a

multiplication by a power of two in binary is performed as a simple bit shift. The equation (2.20a) can be rewritten, in order to focus only on the multiplicative term of d, as

$$u_v = \frac{① + ② \cdot d}{③ + d} \tag{2.24}$$

where

$$① = \frac{\mathbf{E}_{1,1}}{\mathbf{d}_3} \cdot u_r + \frac{\mathbf{E}_{1,2}}{\mathbf{d}_3} \cdot v_r + \frac{\mathbf{E}_{1,3}}{\mathbf{d}_3} \tag{2.25a}$$

$$② = \frac{\mathbf{d}_1}{\mathbf{d}_3} \tag{2.25b}$$

$$③ = \frac{\mathbf{E}_{3,1}}{\mathbf{d}_3} \cdot u_r + \frac{\mathbf{E}_{3,2}}{\mathbf{d}_3} \cdot v_r + \frac{\mathbf{E}_{3,3}}{\mathbf{d}_3} \tag{2.25c}$$

It is possible to write then, exploiting the partial fraction decomposition, the following equation

$$\begin{aligned} u_v &= \frac{① + ② \cdot d + n \cdot d - n \cdot d}{③ + d} = \frac{\big(① + n \cdot d\big) + \big(② - n\big) \cdot d}{③ + d} = \\ &= \big(② - n\big) + \frac{\big(① + n \cdot d\big) - \big(② - n\big) \cdot ③}{③ + d} = \\ &= \big(② - n\big) + \frac{\big[① - \big(② - n\big) \cdot ③\big] + n \cdot d}{③ + d} \end{aligned} \tag{2.26}$$

Also this time, about all terms are precomputed constants plus iteratively accumulated quantities so that the only multiplication to perform each time is the $n \cdot d$ in the numerator. This result is similar to the equation (2.23) but there is more flexibility about the first term of the sum. As said before, the total value of the expression has to be a value in the range $[0.25, \text{WIDTH} - 0.25]$ thus n can be chosen as a small multiplicative constant followed by a bit shift, i.e. $p \cdot 2^s$, in order to reduce the quantity $\big(② - n\big)$ to a small negative value. In this way the fractional term of (2.26) will be performed by an unsigned division with a bit width similar to the one of the division in (2.21) but with a smaller multiplication to perform. In particular the advantages are that each digit of p tightens the integer part of ② by one bit and

$n \cdot d$ is a product between integers, with d usually stored on an 8 bit word. All this reasoning makes sense only if the integer part of ② is not too wide and this depends on the camera real world physical setup. If the quantity ② is shrunk by n in a way that its weight on the final result is less than half pixel, it can be safely removed without a perceivable loss of precision as it would have changed the outcome of the equation only for few cases. The best value for n would then be

$$n = -\text{round}\left(② \cdot 2^2\right) \cdot 2^{-2} \tag{2.27}$$

The powers of two are needed to take into account two more bit before the *round* operation for subpixel precision. In hardware the quantities n and $\left(② - n\right)$ can be computed easily by splitting ② in two trunks of bit whose length is dictated by n. Another possible n is the one that makes the quantity $\left(② - n\right)$ approximately WIDTH/2. This, in conjunction to a signed divider, is able to decrease the range of the division by 1 bit in exchange for the need of one more sum per pixel. This is achievable as

$$n = -\text{round}\left(② \cdot 2^2\right) \cdot 2^{-2} + \text{WIDTH}/2 \tag{2.28}$$

This is conceptually similar to the modification of the K_v matrix translating the image coordinate center from a corner of the frame to the middle of it

$$\mathbf{K}'_v = \begin{pmatrix} a_x & \gamma & c_x - \text{WIDTH}/2 \\ 0 & a_y & c_y - \text{HEIGHT}/2 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.29}$$

Apart from this fact, the easiest way to handle this situation (that will be used in this work) is to let $\left(② - n\right)$ be a quantity less than half pixel and avoid the last sum.

A last possibility is to exploit a look-up table (LUT) to perform the multiplication. A LUT is a table that keeps the precomputed results of repetitive operations. It can implement a multiplication by a constant as needed in this algorithm. It can also perform only a part of the full operation leaving the rest of it to the arithmetic unit. In particular, in this implementation, the operation that should be avoided is in the form of

$$\begin{aligned} OUT &= constant \cdot input\_depth \\ &= n \cdot d \end{aligned} \tag{2.30}$$

This is a linear equation so it is possible to store in the LUT only a subset of the results and compute the others by a simple linear interpolation and by partial products addition. It is also possible to add a bias to the equation and reduce the size of the table by half

$$\begin{aligned} OUT &= n \cdot d \\ &= n \cdot (d + bias - bias) \\ &= n \cdot (d - bias) + constant \end{aligned} \tag{2.31}$$

If the bias is chosen as 128 (half of the range of d), there will be a symmetry between the positive and the negative part of the LUT. In this way, a simple check of the most significant bit of d will cut the LUT dimension by half and the constant term will be added to the other constant part of the numerator in the preprocessing step. This strategy can also be used for the full multiplier approach in order to save some logic gates but it is particularly well suited for the LUT based implementation.

2.1.4 Summary of the forward warping operations

To sum up, the forward warping stage consists of a preprocessing stage where the required constants are computed and stored in a temporary register file. It is followed by a real-time, pixel-level, arithmetic core custom engineered to satisfy the stringent timing required. While the complete set of instructions needed for the first stage is reported in appendix B in order to not pollute this section with a cumbersome list of actions, the final formulæ to perform the incremental forward warping are then summarized here. At first it is better to simplify the constants in order to rewrite them in the shortest form possible.

$$② = \frac{\mathbf{d}_i}{\mathbf{d}_3} = \frac{\mathbf{b}_i \cdot \left(\frac{Z_{far}}{Z_{near}} - 1\right)}{\mathbf{b}_3 \cdot \left(\frac{Z_{far}}{Z_{near}} - 1\right)} = \frac{\mathbf{b}_i}{\mathbf{b}_3} \tag{2.32a}$$

$$\begin{aligned}\frac{\mathbf{E}_{i,3}}{\mathbf{d}_3} &= \frac{\mathbf{C}_{i,3}}{\mathbf{d}_3} + \frac{\mathbf{E}_i}{\mathbf{d}_3} \\ &= \frac{255 \cdot Z_{far} \cdot \mathbf{A}_{i,3}}{\mathbf{b}_3 \cdot \left(\frac{Z_{far}}{Z_{near}} - 1\right)} + 255\frac{\mathbf{b}_i}{\mathbf{b}_3} \\ &= 255\left(\frac{Z_{far} \cdot Z_{near}}{Z_{far} - Z_{near}} \cdot \frac{\mathbf{A}_{i,3}}{\mathbf{b}_3} + \frac{\mathbf{b}_i}{\mathbf{b}_3}\right)\end{aligned} \tag{2.32b}$$

$$\frac{\mathbf{E}_{i,j\neq 3}}{\mathbf{d}_3} = \frac{\mathbf{C}_{i,j\neq 3}}{\mathbf{d}_3} = 255\frac{Z_{far} \cdot Z_{near}}{Z_{far} - Z_{near}} \cdot \frac{\mathbf{A}_{i,j\neq 3}}{\mathbf{b}_3} \tag{2.32c}$$

The next thing to do is to simplify is the numerator of the equation (2.26).

$$\begin{aligned}\Big[① - \Big(② - n\Big) \cdot ③\Big] &= \left[\frac{\mathbf{E}_{1,1}}{\mathbf{d}_3} \cdot u_r + \frac{\mathbf{E}_{1,2}}{\mathbf{d}_3} \cdot v_r + \frac{\mathbf{E}_{1,3}}{\mathbf{d}_3} - \left(\frac{\mathbf{d}_1}{\mathbf{d}_3} - n\right)\right. \\ &\qquad \left. \cdot \left(\frac{\mathbf{E}_{3,1}}{\mathbf{d}_3} \cdot u_r + \frac{\mathbf{E}_{3,2}}{\mathbf{d}_3} \cdot v_r + \frac{\mathbf{E}_{3,3}}{\mathbf{d}_3}\right)\right] \\ &= \left(\frac{\mathbf{E}_{1,1}}{\mathbf{d}_3} + n \cdot \frac{\mathbf{E}_{3,1}}{\mathbf{d}_3} - \frac{\mathbf{d}_1}{\mathbf{d}_3} \cdot \frac{\mathbf{E}_{3,1}}{\mathbf{d}_3}\right) \cdot u_r + \left(\frac{\mathbf{E}_{1,2}}{\mathbf{d}_3} + n \cdot \frac{\mathbf{E}_{3,2}}{\mathbf{d}_3} - \frac{\mathbf{d}_1}{\mathbf{d}_3} \cdot \frac{\mathbf{E}_{3,2}}{\mathbf{d}_3}\right) \\ &\quad \cdot v_r + \left(\frac{\mathbf{E}_{1,3}}{\mathbf{d}_3} + n \cdot \frac{\mathbf{E}_{3,3}}{\mathbf{d}_3} - \frac{\mathbf{d}_1}{\mathbf{d}_3} \cdot \frac{\mathbf{E}_{3,3}}{\mathbf{d}_3}\right) \\ &= \text{step\_U} \cdot u_r + \text{step\_V} \cdot v_r + \text{const}\end{aligned} \tag{2.33}$$

In the end the final form for an incremental computation is

$$u_v^{(i)} = \left(\frac{\mathbf{b}_1}{\mathbf{b}_3} \pm n_u\right) + \frac{\text{NUM\_U}^{(i-1)} + \text{step\_U}_u \cdot \Delta u_r + \text{step\_V}_u \cdot \Delta v_r + \mp n_u \cdot d}{\text{DEN}^{(i-1)} + \text{step\_U}_D \cdot \Delta u_r + \text{step\_V}_D \cdot \Delta v_r + d} \tag{2.34a}$$

$$v_v^{(i)} = \left(\frac{\mathbf{b}_2}{\mathbf{b}_3} \pm m_v\right) + \frac{\text{NUM\_V}^{(i-1)} + \text{step\_U}_v \cdot \Delta u_r + \text{step\_V}_v \cdot \Delta v_r + \mp m_v \cdot d}{\text{DEN}^{(i-1)} + \text{step\_U}_D \cdot \Delta u_r + \text{step\_V}_D \cdot \Delta v_r + d} \tag{2.34b}$$

where $\text{NUM\_U}^{(i-1)}$, $\text{NUM\_V}^{(i-1)}$ and $\text{DEN}^{(i-1)}$ are the accumulation variables that store the partial computation of the numerators and the denominator.

Other optimizations require assumptions over the values of the coefficients but, as written before, they depend on the real world camera setup. These parameters could be checked at runtime but the overhead of the operation is not worth it. It is a wiser choice to design a custom arithmetic unit able to give a result despite possibly large parameter variations, as described before.

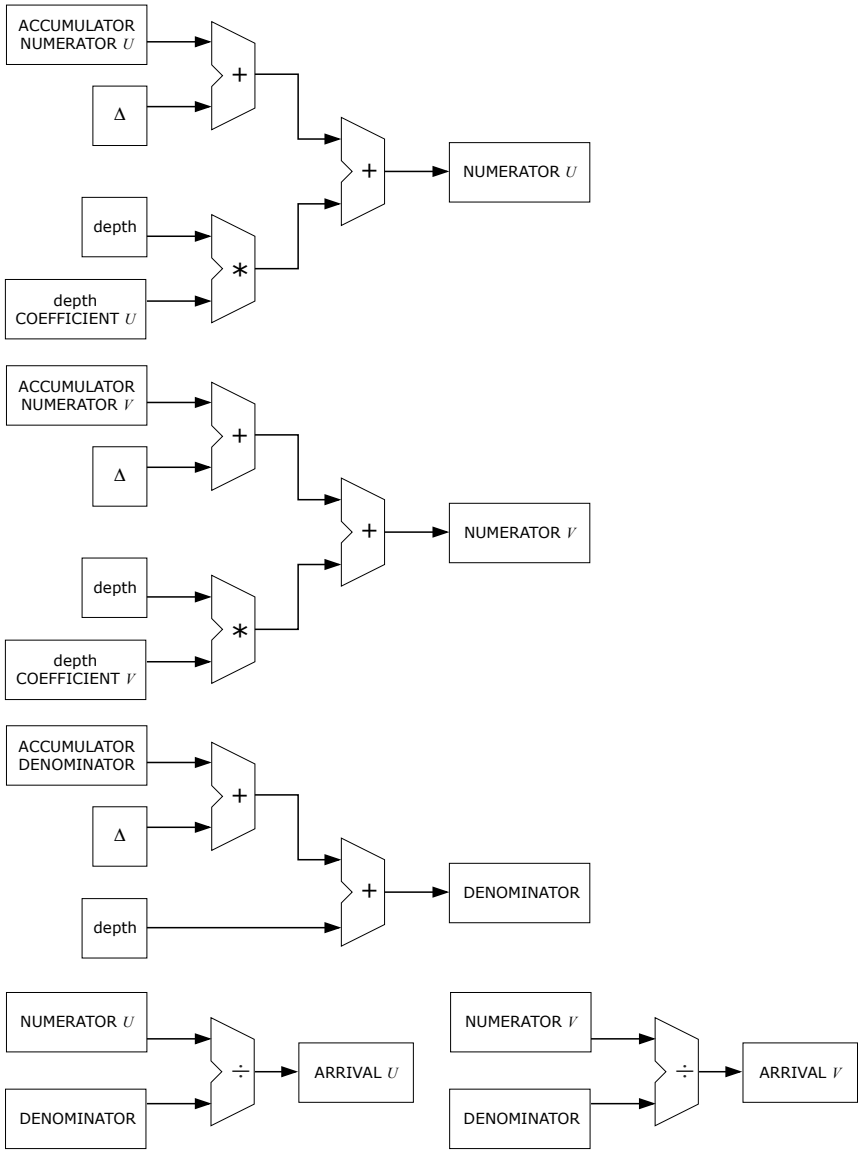


Figure 2.1: Block scheme of the warping engine

2.2 Depth channel preprocessing and tile subdivision

While computing the parameters for the DIBR, the incoming pixel stream should be preprocessed and stored in an external memory for later utilization. A tile memory organization is a good choice for an occlusion compatible pixel processing and to exploit the spatial locality principle between adjacent pixels (after the warping, many neighbour pixels will found their place near each other. Only a small amount of outlier points will be projected to a different location).

2.2.1 Depth preprocessing

In this thesis a high quality input depth channel is considered. This means that noise and blocking artifacts due to compression should be already filtered out. The remaining source of output degradation due to the depth image is its misalignment with respect to the color picture. This introduces a halo in the background that can be seen in figure 2.2a. The culprit of this effect is an object in the scene in foreground with respect to a surrounding background. The depth estimated for the scene is almost always misaligned by one or two pixels so the boundaries in the depth channel have to be expanded by this small quantity. In order to leave unaffected the tiny surface depth variations, a simple, hardware friendly, depth filtering is here proposed. It consists on a selective dilation of object boundaries. At first a morphological grayscale dilation is applied and then the pixels that have changed by more than a threshold will be updated and the other ones are left untouched. The grayscale dilation is performed, in this simple case, substituting each pixel by the maximum value in a neighborhood defined by a binary structuring element that acts like a mask for the maximum value search. The structuring element used here is cross shaped to better approximate a circle, as in table 2.1.

When the maximum value in the neighborhood is found, it is checked to see if it is more than a threshold bigger than the current pixel value. This is similar to control if the gradient between adjacent pixels is greater than a fixed threshold (in order to apply the preprocessing only to foreground objects

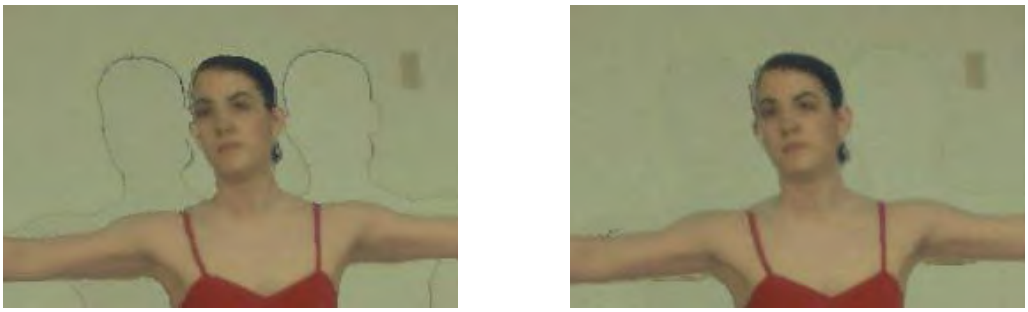
0	1	0
1	1	1
0	1	0

Table 2.1: Dilation mask

boundaries) and then expand the object silhouette in that direction but with less algebra involved. The quality of the result with a threshold of 8 is shown in figure 2.2.

2.2.2 Tile organization

As the pixels are usually received in a raster scan line pattern, they should be reorganized in tile blocks of constant size to fill an external memory in order to fully exploit the burst mode of dynamic random access memories (DRAM). In fact the memory occupation of an uncompressed single RGB 4:4:4 plus depth frame takes 32 bit per pixel. this means that even for a relatively small frame of 1024x768 pixels there is the need of 1024x768x32 bit of memory space, roughly 25Mbit or 3MByte. Even if there are static RAM modules capable of hosting a similar frame, they are expensive and each frame would require an additional piece of hardware. On the contrary, DRAMs are cheap and full of space but they are more difficult to handle effectively. It is thus mandatory to pack the data in a way capable to take the most out of the spatial locality principle. The simplest way is to make a constant size grid usually called tiling approach. An NxN group of pixels forms a tile that is stored in memory sequentially so that a single row memory access through burst cycles could load or write the entire tile as if it were a single macropixel. A burst memory access is a way to access sequentially N adjacent memory locations with reduced latency and, if this is done on a already selected row (also called page) of memory, this data exchange can be very fast. In fact, the video processing pipeline needs both fast



(a) Withouth selective depth dilation (b) With selective depth dilation

Figure 2.2: Effect of morphological dilation on boundary misalignment

random memory accesses and very high data throughputs. For example a 24bpp, 1920x1080 pixel video at 60Hz refresh needs, just for its streaming, an impressive bitrate of about 3 gigabit per second. This is usually done by splitting the data over several parallel wires and stream the data at some hundreds of MHz. This is why the graphic memories are made up from a high quantity of DRAM modules (some Graphic Processing Units, GPUs, present even 16 memory ICs) with a bus width up to 512 bit. It is thus important to optimize the design for external memory utilisation: even a simple single data rate DRAM has theoretically a bandwidth capable of running the DIBR algorithm for high definition video sources (see [6]) but this is valid only if the burst mode on the same row is fully exploited. As the forward warping maps the pixels to scattered locations, the best approach to take is to work with nearby pixels that, if there are no strong gradients of depth, should map to a restricted area of memory. A tiled memory approach (along with an internal memory cache) is a perfect candidate to solve these issues. There is still a problem with this approach: in order to rearrange the scan line in tiles there would be the need of $N - 1$ line buffers. A line buffer is a "First In, First Out" (FIFO) type of memory whose length is equal to the pixel width of the frame. This means that for a 1920x1080 frame at 32bpp (24 bit for the color channels and 8 bit for the depth) there is the need for 1920x32b $\approx$ 61kb of internal memory per line. For 4k resolution, 3840x2160 pixels, the line buffer length doubles. This is not easily doable with an internal memory so there is the need of a workaround to handle a N lines buffer on an external memory. An external SRAM could do the job but it would be better to use the same DRAM module(s) used for the frame buffer. This is possible by writing and reading in burst mode only a subset of the line buffer. As this is a deterministic process it is possible to prefetch the data from memory to not slow down operations. Sometimes it is interesting to take into account more memory usage and use the external border of the tile as a padding region: if it is known a priori that there will be a section that requires a filter applied over a (2N+1)x(2N+1) window, it is possible to pad the tile with N pixels each side to have a more coherent cache locality. As this is already a memory hungry algorithm this won't be applied.

2.3 Occlusion compatible warping order

In equation (2.1) it is possible to notice that if z_r is set to zero there is a special point that not depends on the input frame coordinates. This point is, in three dimensions, the center of the reference camera, the point where the distance from the pinhole of the camera is zero, i.e. the pinhole itself. If the real camera pinhole coordinates are projected by (2.1) to the virtual camera frame, the resulting point is technically called the *epipole*. It can be seen as the point of departure of the real camera rays so this particular point is of great importance for an occlusion compatible warping order. The objects closer to the epipole will always occlude the other objects placed behind them. This means that if this point is taken as a reference for the warping order there is no need to build a *Z-buffer*, a list containing for each warped pixel the arrival u, v and z coordinates, nor a virtual camera depth map, thus saving a good amount of memory. Using the Z-buffer it is possible to warp each pixel and then find which one is the nearest to the camera to eventually display it over the others on the output picture but this takes a lot of memory (with increased bandwidth), a reordering stage and a final rendering stage. This approach usually do not respect the spatial locality of warped pixel so memory misses could potentially ruin the smoothness of execution. If, instead, the epipole is used as reference, the input camera is correctly calibrated, and the input depth maps present a good quality, it is sure that the pixels that are closer to it will occlude other pixels that would have been warped to the same location. There are then two methods to perform correctly a Z-buffer free warping. The first one is the used in [5] and consists on following the lines that depart from the epipole, called epipolar lines, for warping. This is difficult to perform correctly as there is the need of quantizing the slope of the lines, especially near the epipole there are various issues as it is the center of a bundle of straight lines. In this region there is a mix of all the possible slopes and so it is not possible to correctly find a working order to execute the warping. It also needs both an ordering and a reordering buffer to work which can be difficult to handle.

The second method is proposed here to solve these issues. Instead of following the epipolar lines it is possible to warp the scene as if it were made of concentric rings whose center is located at the epipole. As circular shapes are difficult to handle for raster frames, the same concept works also for concentric squares. It is possible

to act in two different ways. The first is to start from the epipole and discard all the pixels that warp to an already filled location; the second method consists on the opposite approach, that means to start from the most external ring and blindly overwrite the old pixel value with the new one. This could need a slightly higher bandwidth for the external memory but if there is a sufficiently large memory cache the spatial locality principle behind the DIBR algorithm should be able to reduce this side effect, avoiding the conditional nature of the diverging approach which would instead require a way to signal if the pixel is already been written or if in that position there is still a blank value. The converging approach can also be effectively mixed with a tile based memory handling. The pixel containing the epipole can be set as the reference to pick the next tile to load from memory and then a fine-grained pixel processing would take act inside the tile to ensure the occlusion compatible warping. This is a type of painter's algorithm: points far away are overwritten by nearer objects without a Z-buffer.

The coordinates of the epipole are simply

$$u_{v,epi} = \frac{\mathbf{b}_1}{\mathbf{b}_3} \tag{2.35a}$$

$$v_{v,epi} = \frac{\mathbf{b}_2}{\mathbf{b}_3} \tag{2.35b}$$

that are the same as ② in equation (2.26). The proposed way to move around this point is to firstly calculate the distance between it and the four borders of the frame. The most distant border(s) will dictate what is the current square radius. At this point, if there is more than one border at that distance, there is the need to warp all their pixel tiles. The warping order can be set at will as long as all the pixels are processed in the correct order. In the proposed algorithm a clockwise implementation is given. When all the tiles belonging to the maximum radius have been warped, the corresponding borders are shrunk by one and the process is repeated until all the tiles have been warped. In order to exploit the incremental computation of the warping parameters it is important to store (and update each time) their values for the tiles at the corners of the remaining area to be warped. The occlusion handling must be performed at both tile and pixel levels, calculating the warping order also for the area inside the tile.

The "next tile address generation" can be performed while the warping engine is doing its job, successfully prefetching from memory the next tile that has to be put in queue.

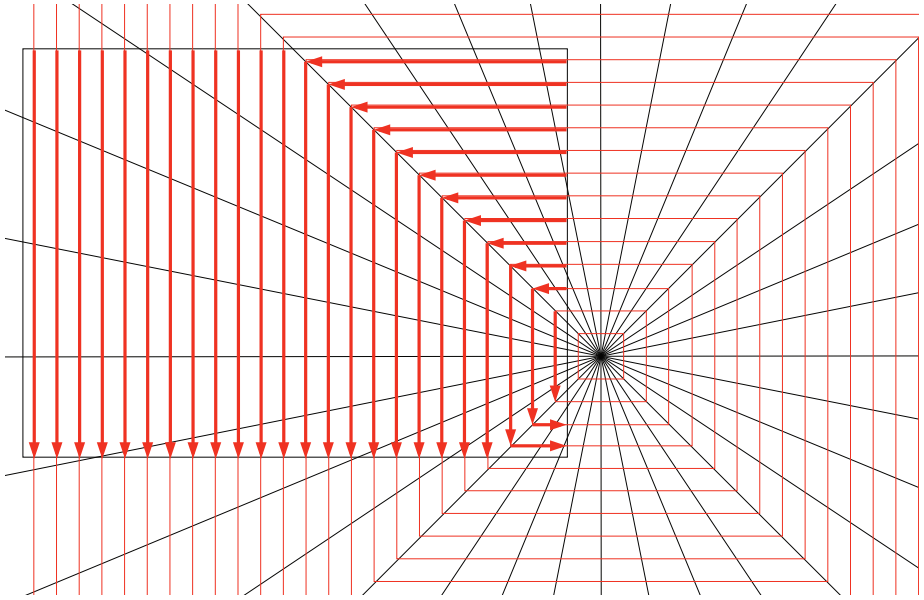


Figure 2.3: Example of occlusion compatible scanning order

2.4 Joint image fusion and crack filling

The forward warping only DIBR algorithm presents many artifacts called *cracks*. These are random missing pixel values mainly due to the resampling operation that takes place during the warping stage. The input pixel grid is different from the output one so a resampling is needed. In order to speed up the operations and reduce the hardware required to the minimum, in this thesis there is no handling of this issue at warping level but a correction is performed during a stage called "joint image fusion, crack filling and inpaint". This stage wants to fuse the two warped views retaining the texture information from both the sources. Because after the warping stage all the depth information is lost, there is the need to operate only on the color side of the warped views. As the cracks are only one or two pixels wide and their content could be a blank value (a pixel where no input point has been warped to) or a value usually much different from the surrounding ones (where a background pixel fills a void between foreground ones), they resemble the *salt-and-pepper noise*. This type of noise is formed by a random distribution of white or black pixels (impulse noise) over the picture area. It can be reduced by a nonlinear filter called *Median Filter* that is particularly robust against outlier values if applied where the texture should not change with too high frequency values.
An interesting variant of the median filter is the *Hybrid Median Filter*. It is less expensive to perform than a full median filter and its directionality nature permits a better high frequency behaviour with respect to the plain median filter. This filter has been customized for this task in order to retain the information from both the views and to still give a satisfactory answer in presence of noise, even if the information is coming from a single view only (as in case of an occlusion). It also takes in account that the unfilled pixels present a black value (in case of RGB color space, or an illegal value for other color spaces) to simplify the filter task.

2.5 Inpaint

Even after the image fusion, it is possible that the occlusion shadows have left some blank regions where there are no information from the input views. In these parts of the frame there is the need to inpaint the missing textures. This is performed according to the idea behind the method described in [7] with some modifications to perform a better job. The idea of directly reusing the previous inpainted pixels to make a single pass convolution inpaint is retained but the kernel is modified in order to use also the information from the pixels that will follow. In particular, as the blank pixels are flagged by a black value (if in the RGB color space, or by an illegal value in other color spaces), it is possible to discern the known good pixels from the blank ones. The idea of reusing the previous calculated pixels to inpaint the next one in a single iteration is retained and merged with the approach described in the previous section. The hybrid median filter is still used but this time the previously processed pixels can be used directly to find the output value, that will be used for the next pixel calculation. In this way there is no need to perform divisions and a single filter pass over a sliding window will be able to perform all the post processing steps at once, greatly reducing the hardware required by different solutions. The filter tries to diffuse the texture from the surrounding pixels in the current one to perform the inpaint. This is a coarse method that does not try to fill the missing regions by probabilistic methods as in other approaches so the overall quality will be inferior to competing solutions but the hardware required is much smaller and

(a) Left warped view (b) Right warped view

Figure 2.4: Warped views present many crack artifacts

simpler, better suited for real-time implementations.

(a) Reference view

(b) Synthesized view

Figure 2.5: Comparison between the reference and the synthesized frame 95 for the sequence Ballet (camera 4)

(a) Reference view

(b) Synthesized view

Figure 2.6: Comparison between the reference and the synthesized frame 95 for the sequence Breakdancers (camera 4)

Chapter 3

Proposed architecture

Now that the general behaviour of the algorithm has been explained, an architectural study of a possible implementation will be next discussed. To summarize, the steps to perform are:

1. Preprocessing
   - Computation of parameters
   - Depth filtering
   - Tile subdivision
2. Forward warping
   - Warping engine
   - Tile and pixel address generator
   - Control unit
   - Cache management
3. Postprocessing
   - Crack filling
   - Image fusion
   - Inpaint
   - Tile to scanline raster

This is already a logical subdivision with respect to data dependencies. The numbered steps must be performed in a strict sequential order for free viewpoint navigation but the dashed ones can be parallelized (or merged, in the case of postprocessing) to improve the throughput via hardware acceleration. In this section a possible

hardware architecture dedicated to the DIBR is explored and described.

3.1 Overview

The definite separation into sequential tasks is well suited to a *frame-level pipeline* architecture. This means that while a frame is being processed by a stage, the other parts of the IC can still go on with other frames like in an assembly line. In exchange for an increase of the total latency, each stage has more time to compute its part, relaxing the timing specifications. The major drawback of this approach is the need to store a temporary frame buffer in each stage, increasing both memory capacity and bandwidth required. It is thus important to optimize the external memory utilization. The tile approach is well suited because, in conjunction with the help of a memory address translation table, it is possible to signal when a tile will no longer be used in order to free memory for new tiles. The frame-level pipeline timing is dictated by the total frame refresh rate: a higher frequency will leave less time to compute the various parts of the assembly line. Currently the worst case scenario for the refresh frequency is 120 frames per second, but as for many other video parameters this quantity is still increasing with the new generation hardware equipment.

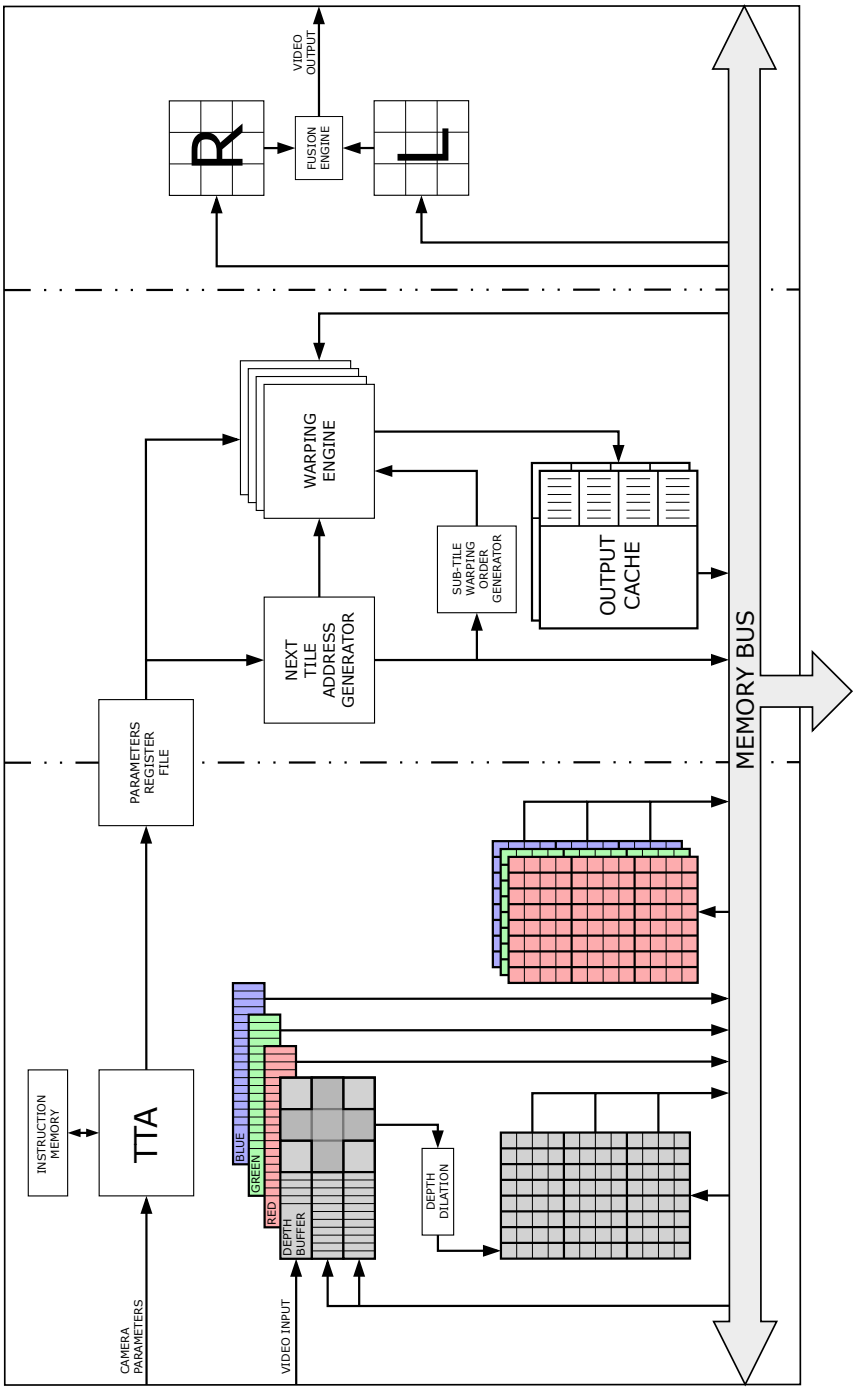


Figure 3.1: Overview of the proposed architecture: the dashed lines separate the three frame-level pipeline stages

3.2 Preprocessing

This part is composed by a general purpose processor unit to compute the warping parameters along with a depth preprocessing and a memory management unit.

3.2.1 General purpose programmable processing unit

The computation of parameters is handled by a programmable processor, in this case a *Transport Triggered Architecture* (TTA). A TTA is a particular architecture in which a controller unit only moves the data between hardware units. It is possible to exploit concurrent operations between free units and add custom modules to perform hardware acceleration. It is called "transport triggered" because there are no control signals and the computation is started by a block only when the data on a particular input port, the trigger port, is changed. For example it is thus possible to load an adder unit with one addend in the non triggering input and then the other one in the triggering port to start the computation of the sum of the two values and pick up the result from the output port after the latency required by the block. It is also possible to bypass the register utilization routing directly the result of an operation into the next function unit, which is also a good energy saving approach. The major drawbacks of this architecture is the difficulty to handle interrupts, but here there is no need of them, and the fact that the binaries compiled for a developed architecture won't work even for a slightly different processor. In this case the architecture is fixed so this is not a problem.
The architecture that will be proposed here is based on the excellent open source work of the Tampere University, the *TTA-based Co-Design Environment* (TCE, [8]). The "Co-Design" term indicates that both software and hardware can be engineered in the same environment. It uses LLVM (a modern, very powerful compiler) to compile the input code for the custom designed processor. An important feature for this project is the software floating point support. As written before, parameters related to the real world could present a high dynamic range. There are few exceptions, for example the rotation matrices that must be by definition orthogonal matrices, but it would be better to use a double precision floating point format for this step and eventually convert them to a fixed point format to feed the warping stage. The C source code for this is presented in appendix B.

The instruction stream has to be fetched from a non volatile read only memory. The input parameters can be passed to this unit as a stream of data using a dedicated I/O functional unit but the computed results should be stored in registers to be used in the warping stage. It is possible to interleave two register files to respect the pipeline memory requirements or stream the computed parameters to the next stage and leave to it the sorting to the specific registers. The register file approach has a slight overhead with respect to the stream solution but it is simpler to manage and the total memory requirement for this is negligible in both cases.
As there is time to perform all the computations, there is the possibility to use a simple *Arithmetic-Logic Unit* (ALU) composed by a 32 bit adder/subtractor plus some bitwise operations, along with a barrel shifter capable of working with signed numbers.
As soon as all the coefficients are computed, a "data_valid" signal should be set to signal to the next stage that there is the possibility to go on with the frame level pipeline. In figure 3.2 is presented a possible TTA architecture for the required implementation. The upper boxes are the functional units, the register file and the control unit. The units exchange data through the buses (thick black lines) using the connections indicated by a black dot. The arrow between the functional units and the buses indicate the data direction and the boxes with a cross are the triggering ports, if new data is enrouted to that port the functional unit should start the computation using the the data presented on the ports.

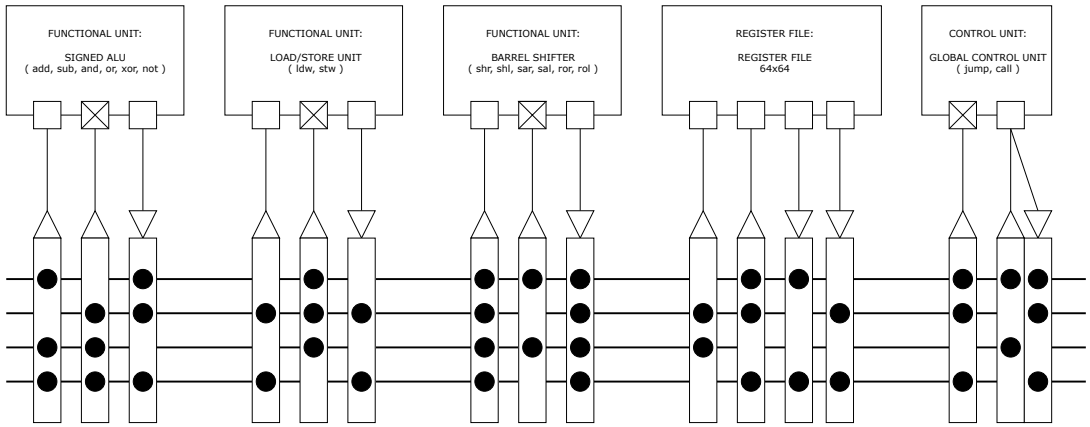


Figure 3.2: Example of a TTA architecture capable of computing the DIBR parameters with a small area footprint

3.2.2 Depth preprocessing

The proposed depth channel preprocessing needs essentially the search of the maximum value in a predefined neighborhood following the 2.1 table structure. This problem can be solved in various ways. As it is not important to sort the values, this can be done with a minimum of four comparisons as in figure 3.3. In this configuration the first two comparisons can be performed in parallel to reduce latencies. The next thing to do in this section is to subtract to the winner the central pixel value and check if the resultant value is greater than the threshold. If the threshold is a power of two value the last check can be performed by a simple OR gate. Usually a value of threshold $T = 8$ is enough to discriminate between sharp object boundaries and slowly changing regions.
The depth value is usually an unsigned integer over 8 bit so the comparators and the registers are lightweight pieces of hardware. This work is based on a two views input setup so, admitting that both frames are streamed in parallel, there is the need of two of these real time processing parts in parallel, one for each view. The total structure is shown in figure 3.3. The last comparator stage is between the central pixel (IN5) and the maximum value of the neighbor pixels. The last multiplexer is driven by the bitwise OR of the result to apply the threshold.

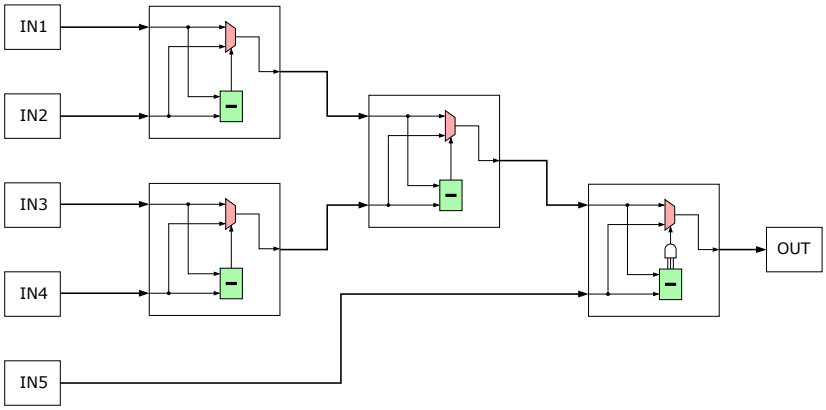


Figure 3.3: Selection network to find the maximum value for 5 entries with threshold in the last stage

3.2.3 Memory management and first stage cache

The input video stream is supposed to be based on a two views plus depth setup. Because of the high bandwidth required, each pixel should be received in a parallel RGB+D (or equivalent color spaces) in parallel. The depth is usually stored as 8 bit per pixel but the width of the color channels can be dependent on various factors. In this work either a 24 bit RGB color space or a 26 bit YCoCg-R ([9]), or equivalent, color spaces is assumed. The latter is a color space based on a simple reversible transform from RGB but with a higher decorrelation between channels, improving the possible compression ratio. It is based on the YUV color space thus it provides a luma channel Y and two chroma channels, Co and Cg. This means that the the human eye will pay more attention to the Y channel variations with respect to the other two. As in this work no on-the-fly frame buffer compression is implemented, it is assumed that at each clock cycle an entire pixel is received through a parallel input interface. This lowers the interface frequency but requires many parallel wires. This choice has been made because it is easier to work with a parallel interface, in particular with respect to the timing, and because it is possible to add a deserializer structure in a second revision of the work, for example to connect to an HDMI, a DisplayPort or even a MIPI interface, all of these being standard serial

interfaces dedicated to video streaming. The parallel interface implemented here is based on the standard VGA configuration with *HSYNC*, *VSYNC*, *Pixel_Data* and *Pixel_Clock* to not depart too much from the classical video interface. The input signal information (pixel width, pixel height, dimension of the various porches, HSYNC and VSYNC pulse width) are assumed to be packed with the camera setup parameters and handled by the processor.
As written in the previous chapter, in order to exploit cache coherency the input stream should be organized in tiles. A good tile dimension is 16x16 pixels but a straightforward implementation of tile reordering would require at least a buffer of 15 lines for each frame. Assuming a RGB+D encoding over 32 bit per pixel, a scene with two views with a line width of 1920 pixels would roughly require 1.8Mbit of internal memory just for the reordering cache (2x15x1920x32b). This is too much of a burden for the internal memory so there is the need to use a temporary external storage for this task. The proposed method consists on:

1. Collect M contiguous pixels from a input line in the internal memory
2. Store this block of pixels to an external DRAM by burst writes over a single page of memory
3. Repeat steps 1 and 2 for the entire line
4. Repeat until N lines are collected in the external memory
5. Fetch N vertically aligned blocks of length M
6. Divide the resulting block in tiles of size NxN
7. Burst write on a single row the tiles into the external memory
8. Repeat steps 5∼7 until all the N lines are processed
9. Repeat steps 1∼8 until the entire frame has been processed

This method requires a fixed amount of internal memory irrespective of the total frame dimensions. To make an example, using a RAM with a burst length of 256x 32b words (full page burst) a total of 256 pixels per burst are written (assuming a 32b RGB+D packing). The DRAM row precharge is a time and energy consuming process, this method limits it as much as possible. It is possible to filter the depth channel in this stage to avoid the full line buffer (as before the drawback is the higher bandwidth required to send and fetch data to the external memory).

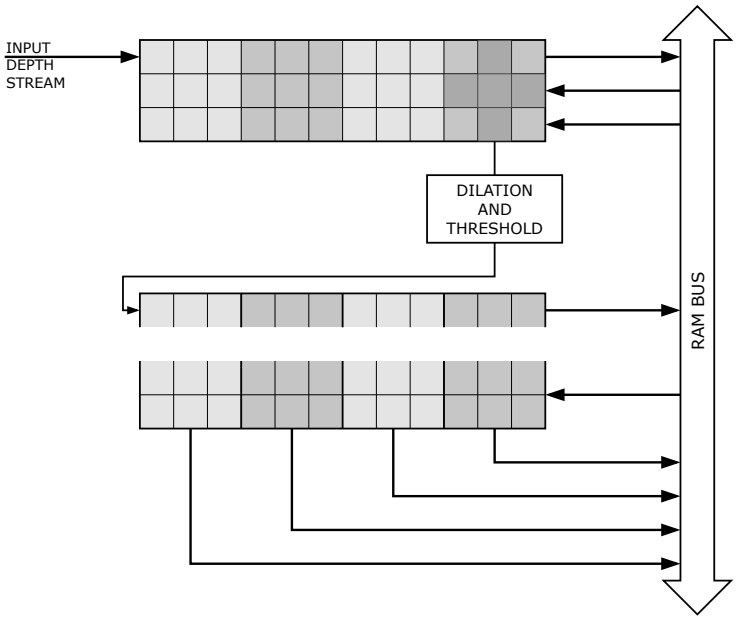


Figure 3.4: Scanline to tile memory reordering with depth preprocessing

3.3 Forward warping

This section is the most difficult part to handle as the highest workload is done here and it should be accomplished in real-time. The occlusion compatible address generation has to work in parallel with the warping engine to ensure the best utilization of hardware and memory bandwidth. The address generator has to find what is the next tile to serve to the warping engine and generate the internal warping order. The warping engine can compute the resulting locations for the entire tile of pixels and a memory management unit can write them to the output cache memory in the right order, following a precise pattern. This will be described later with more details.

3.3.1 Warping engine

Fixed point migration

As a floating point design can be cumbersome to handle, it would be better to migrate all the algorithm to a fixed point notation. The starting point is that all the pixels warped outside the output frame are useless. This means that the division, the most expensive operation to perform, has a fixed output range. If the numerator of the division is outside the range there is an overflow condition and the pixel won't be warped at all. In other words, if the ratio between the numerator and the denominator is too big, or if the result is negative, there is no need to perform the division as the result will surely be useless. The biggest problem is that both numerator and denominator have a non fixed range. There is the need to normalize the numerator each time the denominator changes scale.
All this said, it is now possible to start developing the divider unit and then reverse-engineer the algorithm to build the best framework for it. In order to be able to take care of a 4k (3840x2160 pixel) resolution up to a subpixel precision, the result of the division should be in a fixed point format **Q12.2**, which means that it should have 12 bit of integer part (4096 pixels) plus 2 bit of fractional part (quarter pixel) for rounding purposes. The basic rule to have a n bit result from a fixed point division is that the format of the numerator has to present n bit of precision more than the denominator [10]. It is then possible to track the denominator scale and

apply the same scale factor to the numerator to avoid loss of precision, i.e. if the divisor has to be shifted by m places for normalization, also the dividends have to follow the same rule to maintain the same bit ratio. If the numerator overflows, the result of the division is discarded and the pixel is flagged as non valid. If the signs of the operands are different the result is negative, outside of the valid output range. Also in this case the result must be discarded. Following these premises, the dividend should be in a **Q(12+N).2** format and the divisor in a **QN.0** format as the total amount of decimal places in the result is the difference of decimal places in the numerator with respect to the denominator. In order to find the value of **N** it is important to consider that the result should be accurate up to 1 LSB. This means that in a long division the shift-and-subtract operation has to be performed 12+2 times to have 12.2 bit of result. The divisor should then be on a **Q14.0** format and the dividend on a **Q26.2**. This operation should be performed as fast as possible so there is the need of a pipelined divider. As the number of bits is not too big, a non-restoring array divider will be used for this purpose like the one in figure 3.5 with some pipeline stages inserted to increase the throughput and split the critical path to meet the real-time requirement.

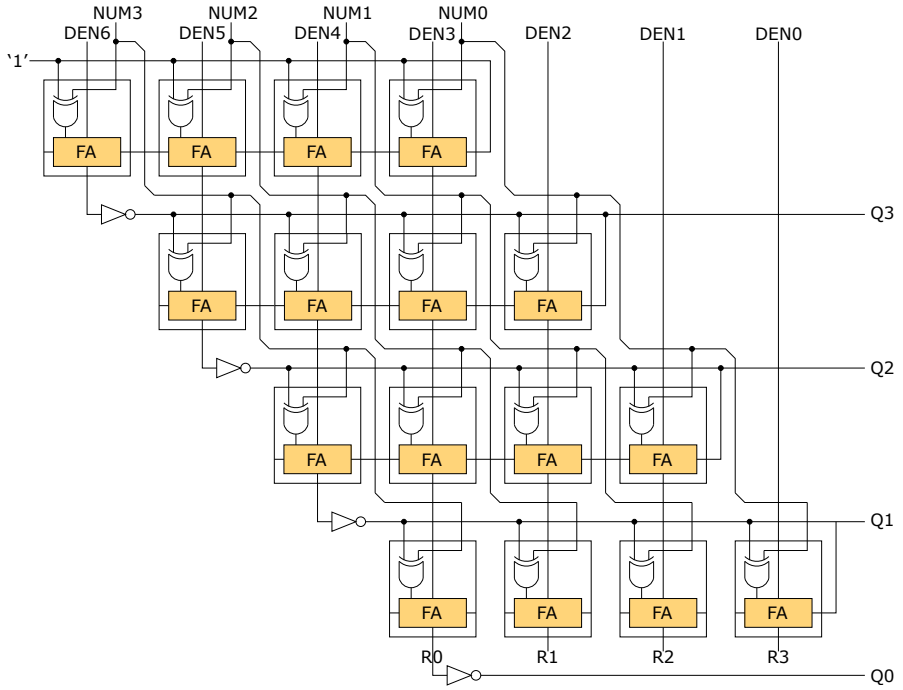


Figure 3.5: Example of a 4 by 7 non-restoring array divider

As written before, the dividend should be normalized following the divisor behaviour. Each time the divisor shifts, the dividend has to follow it. An interesting thing to notice is that this division defines the width of the operands so this is a starting point to build the previous arithmetic stages. In equations (2.34) it is possible to see that both the numerator and the denominator are computed as the sum of three terms: an accumulation variable, which is updated at runtime, an increment step, which is defined by the direction with respect to the previous pixel and the d term, the one that needs a true multiplier for the two numerators. Analyzing further the addends, there are two parts that are multiplied by the input pixel coordinates but this operation is bypassed exploiting an incremental computation. This means that one of the operands of the multiplication is a value ranging from 0 to FRAME_WIDTH or FRAME_HEIGHT. This is an unsigned integer of maximum 12 bit for a frame width of 4096 (the maximum value considered for this architecture). As this range is always positive it wouldn't change the sign of the result so there is the possibility to find the absolute maximum value in advance. The worst case is when positive terms are summed to negative terms of similar absolute value.

In this case there could be a loss of significance (catastrophic cancellation) due to the finite precision arithmetic. This means that there is the need to guarantee that the algorithm still works even in the worst case. This would mean a large accumulation register to take into account its minimum variation. For each pixel

$$\mathrm{ACC} \leftarrow \mathrm{ACC} + \Delta \tag{3.1}$$

A way to avoid accumulation errors is to reset the accumulator to a known precise value once in a while. For example one pixel for each tile could have its value stored in a lookup table or computed with a big multiplier and used as a basis for the other pixel calculations. This reduces the complexity of the accumulation register and the errors due to it. There is the need to ensure that the 28 MSB for the numerator variable and the 14 MSB for the denominator are correctly computed with the smallest error possible. To reduce the size of the LUT it is also possible to store the exact value with greater precision and reduce the table entries to, for example, one each 4 or 8 tiles and compute the surrounding tile values on the fly. The last way, which is the one that will be used here in conjunction to the occlusion compatible addressing, is to take care of only 4 initial values per frame, one for each corner tile, and compute the next tile value starting from these. This requires a slightly larger global accumulation register to accommodate the incremental computation but at the expense of greater precision and simplicity. This will be discussed later.
So, assuming that each corner tile has its starting value, the best way to avoid numeric errors is to accumulate the increments in separate variables as it is certain that all the steps are of equal magnitude but there is no assumption over different values. This means that increments in u and v directions have to be tracked independently and summed in the final stage with the d value and the starting value. As u and v increments are not correlated, they can be handled by separate accumulators, by two small 4 bit multipliers or 16 entry LUTs. The multiplier is the preferred way because it can permit a random access to the two variables and avoid complicated access patterns inside the tile to ensure both the occlusion compatible scanning and an accumulator friendly scanning. To sum up, each tile would need:

– three starting values (one for the u coordinate, one for the v and one for the shared denominator)

- six increment values (as before, a Δu and a Δv value for each part), the same for all the tiles
- two depth coefficients
- six 4 bit multipliers, one for each increment variable.
- two 8 bit multipliers for the d terms at the numerators
- nine adders/subtractors to combine the elements of the numerator (three for each numerator and three for the common denominator)
- one leading zeroes counter (to find by how many positions the numerators and the denominator should be shifted)
- three shifters to align the numerators to the denominator format
- two dividers

Apart from the first two parts, all the other resources have to be replicated for each pixel-level parallelism added. The *starting values* should present at least 28 bit of significand for the numerators and 14 bit for the denominator. They are calculated once per tile by the tile address generator with a higher precision while the previous tile is still being processed. Because of the increased computation time available, a serial arithmetic unit can be employed to shrink the required hardware. Some guard bits could be used to improve the precision of this coefficient to avoid quantization errors. If at least 40 bit are used for the numerator coefficients and 26 for the denominator the errors will be reduced to the minimum.
The depth coefficients have a limited range and are surely representable as a fixed point with two decimal places (quarter pixel precision), i.e. Qx.2. Usually this coefficient is a value under 2^{12} so, in order to take into account also very different camera setups, it will be set as a **Q18.2** number. As d is a integer quantity over 8 bit, the result of this multiplication will be a number on a **Q26.2** format. An important consideration should be done here, mainly to avoid floating point computation of sums. The hardware overhead of a floating point unit is very big, at least a barrel shifter per sum plus some other control parts. It is possible to trade this overhead for greater precision using wider fixed point formats. The depth part should be able to modify the outcome of the division (if this wasn't the case, the depth channel would be useless). This means that the sum of the other parts of the numerator should be of the same magnitude of the result of this multiplication. If smaller, this would be the leading term; if greater, that sum cannot be bigger enough to overcome

the result of the multiplication. This means that the absolute value of this term dictates a sort of "lower bound" for the fixed point implementation for valid DIBR usages. Obviously the denominator can be 28 bit less precise than the numerators as, in the worst case scenario, the numerators would overflow. A statistical study of the MATLAB implementation of the algorithm over various datasets has led to a data format of **Q30.16** for the numerators and **Q16.14** for the denominator, taking into account also some guard bits to reduce the risk of overflow of the denominator for more elaborate camera setups. The diagram of figure 3.6 shows the pixel engine inner working. In particular it is possible to check on the fly if the output pixel is valid, the requirements are that the arrival coordinate should be positive (i.e. the operands of the division must have the same sign), internal with respect to the frame boundaries and the numerator shifters must not overflow (or else it would mean that the arrival coordinate will be wrong and out of bounds).

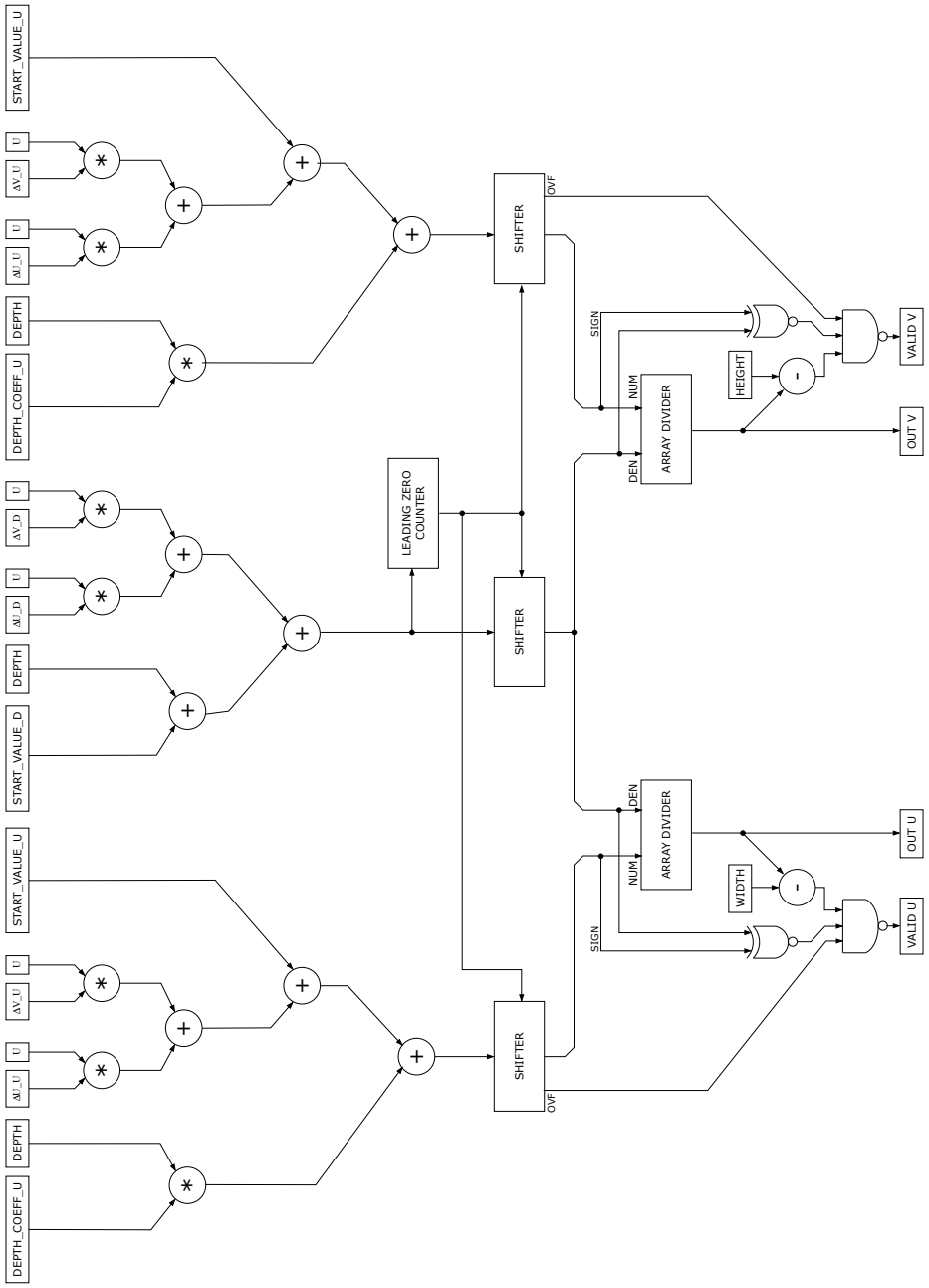


Figure 3.6: Data flow diagram for the warping engine

3.3.2 Tile and pixel address generator

This section analyze the behaviour of the next address generator. It work in a hierarchical fashion: at first the next tile is selected and loaded in memory, then the internal occlusion compatible pattern is defined. The easiest way to accomplish this task is:

1. store the corner tiles of the frame as
   - top-left tile as START_UP
   - top-right tile as START_RIGHT
   - bottom-right tile as START_DOWN
   - bottom-left tile as START_LEFT
2. sort the distances of the borders of the frame from the epipole and flag the largest one(s). Take this value as the starting radius.
3. in clockwise order, starting from the up direction, process the tiles of the directions previously flagged, starting from the START tile and update its value so that it is moved one tile towards the epipole direction. When the start tile of the next direction going clockwise is reached, the tile value is updated like before and the process moves to the following flagged direction.
4. when all the flagged directions have been served, reduce the radius by one tile, flag the new one(s) that are equal to the radius and repeat the process.
5. end when all the frame has been processed, the simplest way to track this is using a 16 bit tile counter (there are other ways to do this but are more complicated to implement)

This algorithm makes sure that all the tiles belonging to the current radius (i.e. distance from the epipole) are processed a single time. When a new direction is processed, the previous one should have moved its starting value to avoid overlapping processing. This pattern converges to the epipole tile or to the border the nearest to it. The condition to start or stop the processing in one direction is simply given by an equity condition that can be accomplished by a XOR operator between two values. The tile coordinates are two unsigned 8 bit values but the coordinates of the tile containing the epipole can be (and usually are) outside the frame. The tile based computation reduces the width of the distance and of the radius variables of 4 bit with respect to a pixel level approach. Each time a START tile is processed,

the value of the accumulator variable is updated for the new tile. The following tiles in the same direction have their accumulation variables directly computed from the previous tile.
The pixel address generator works in the exact same way but using the full coordinate of the epipole to compute the exact access pattern. This could be an overkill in many real world situations but it is wiser to dedicate a general algorithm to accomplish this task and avoid special case issues. Like before, the four corners of the tile are listed and the directions corresponding to the current radius are flagged. The problem here is that there could be more than one pixel processed simultaneously for pixel level parallelism. This means that there is the need to find what pixels should be processed next at the same time. This could be done by a loop unrolling but this time it is important to take more care of the corner values update process to avoid the possibility that a pixel could be processed twice or unprocessed at all. A way is to precompute the pattern and feed this order to a dispatcher but this would still require a way to compute the order serially and output two values at time. This is difficult and resource intensive, so there is the need to prevent the corner special cases in advance at algorithmic level. Here a *divide and conquer* approach is used. The current 16x16 tile is subdivided in sub-tiles of 2x2 that are ordered as before from the ones belonging to the highest radius to the ones nearer to the epipole. Due to the concentric square nature of this approach, there are surely two radii inside the sub-tile. The pixels on the most external one will be flagged as highest priority and the other(s) will be processed later. There are only two possibilities, three or two high priority pixels and the other one(s) low priority. The high priority ones have to be served before the other(s). The only caveat is that the low priority one(s) should overwrite the other one(s) processed at the same time if they warp to the same place. This is easy to check with a simple equality. This method works for up to 4 warping engines in parallel per view but it could be expanded to bigger sub-tiles if needed (obviously at the expense of some hardware overhead). As the warping engine is already rather big for itself, it is safe to assume that the final implementation would present no more than 4 of them per view.

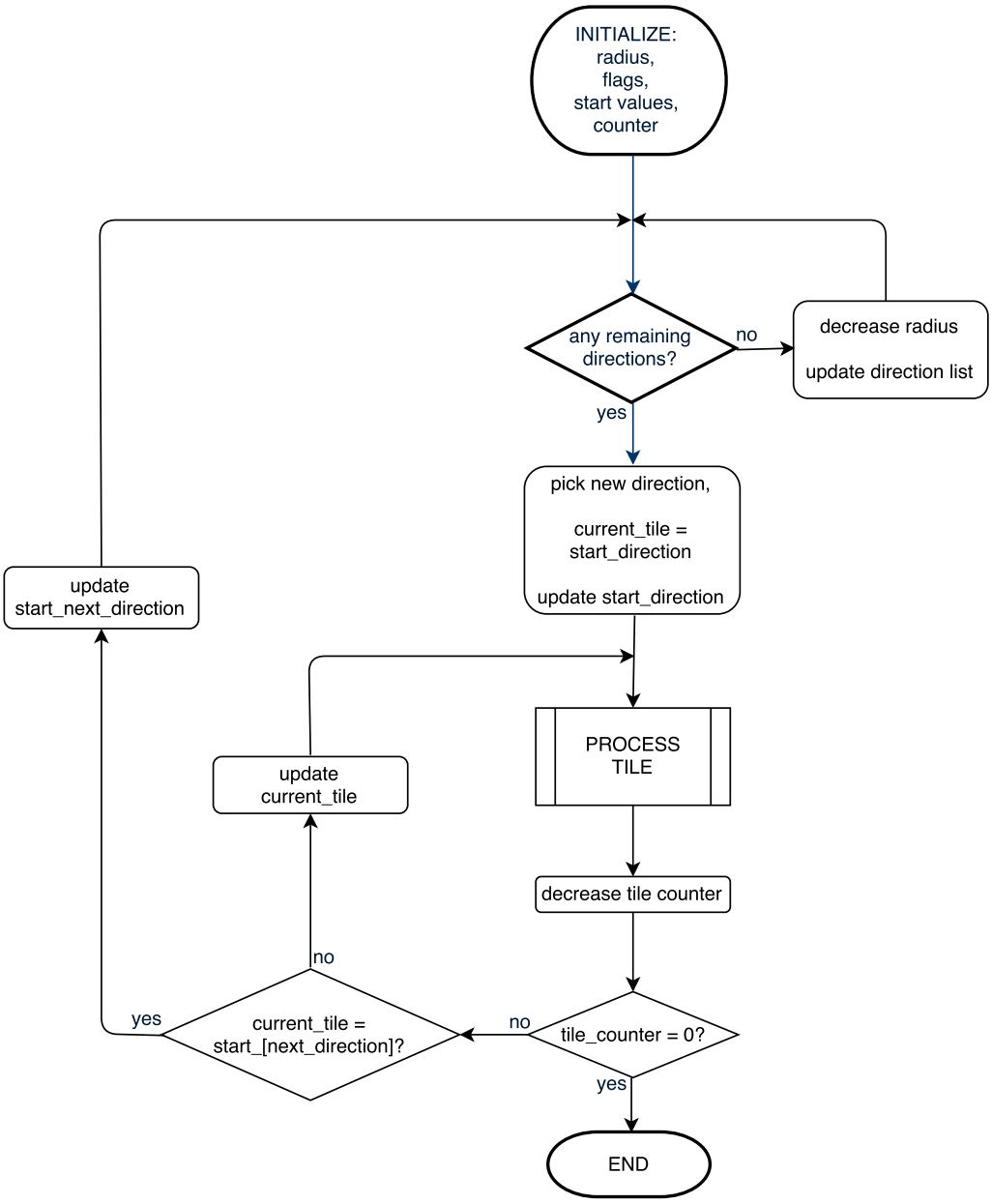


Figure 3.7: Flow chart for tile level execution

3.3.3 Control unit

All the previous parts should be monitored by a custom control unit to follow the algorithm. This is accomplished by a *finite state machine* (FSM) that should check the status flags to decide what to do next. In this case there is the possibility to

exploit the fact that the entire tile processing takes a relatively a long time (at least 64 clock cycles for 4 pixel engines in parallel). The next tile address can be found during this available time and the tile itself can be loaded into an internal cache memory so that there is no need to wait for it to be available when needed. Also because there is so much time, the same circuit can be used to find the next tile also for the other view with a single datapath and two sets of context registers. Because there is the need for speed, some shortcuts have to be taken, bypassing the control unit itself. For example, like described before, in the warping engine the numerator overflow condition after the shift can be directly used to signal if the calculated arrival is a valid pixel location or not. The same is valid for the division by zero condition. This simplifies the global control unit that will only feed the warping engine without taking care of the status conditions until the end of the computations.

3.3.4 Cache management

The input and the output formats of the warping engine are not equal. Thanks to the occlusion compatible warping, the depth channel is dropped and also the tile system need not to be the same of the input format. In fact, as the arrival locations could be sparse, a smaller tile system would be more adequate for the task. The engine requires to retain in memory two tiles for each view, one as the current processed tile and the other as next tile, prefetched as soon as its address is known in order to use the external memory at its best. This means 16x16x32b per tile, two tiles per input view, two views for a total of 32kb of memory. The cache for the output should obviously be as big as possible but this is difficult to achieve. As said before, a single line buffer of FullHD (1920 pixels) takes about 61kb of memory for uncompressed RGB+D. This means that the same amount of memory could host 40 8x8 RGB tiles. This is a good starting value to build the output cache. As there is also the need to store the output tile address, the required memory per tile is 8x8x24b+9b+9b = 1554b because a 4096x4096 frame (the largest valid value used for this design) can be subdivided in 512x512 tiles of this dimension. Presuming 16 output tiles per view, the required memory is approximately 50kb, still less than a single 1920 pixel wide line buffer. This covers an area of 32 square pixels per

view, four times the input tile area. If more memory cache is available, obviously the memory misses will decrease but in expense for higher internal area. For this task the cache should be a set associative one, as described later. As the warping engine outputs two or more pixels per view per clock cycle, the cache must be able to follow this throughput with the minimum cache miss ratio. As this is not easy to handle, a way to synchronize the two activities should be used. The proposed way is to add a FIFO memory as a queue system to hold the pixels whose corresponding tile is not in cache so that the other pixels can still be written in memory while the missing tile is fetched from the main memory, replacing the least recently used tile (LRU cache approach). All these methods should be able to ensure a smooth utilization of all the resources, but there is still the need to be able to slow down the warping engine production if also the FIFO queue begins to become full. For this reason the warping engine should be able to be placed in a stall condition until the FIFO has new free slots. This is a classical example of producer-consumer problem. As there is more than one pixel produced at each iteration, there could be the case that there are more than one pixel with a miss condition. Therefore also this FIFO is a resource that should be parallelized according to the pixel level parallelism. It is possible to add a second level of cache, if needed, to store a higher amount of tiles for faster execution times. The most interesting part of this approach is that this is a write only work, no data has to be read by the warping engine. This simplifies the memory handling and a 16 bit mask can be used to select which is the pixel of the selected output tile that should be overwritten, leaving the other ones untouched. As before, the read and write operations on the external DRAM can still be performed by a burst, saving the row precharge time (and energy) that would have been mandatory for a random single pixel write. Each output tile should come along with a flag bit to decide if the tile has already been processed in a previous warping or if it is a newcomer one. In the second case, the flag should be set and the incoming tile initialized to blank pixel values.
To summarize, if the required tiles are present in the cache (as one cache block), the pixels outputted by the warping engine should be written in the respective places.

Parallel computation handling

Another problem to overcome is that each clock cycle outputs more than one pixel in parallel. This can be solved with a multi-port SRAM approach having one write port for each pixel to write. This is usually avoided as the multi-port SRAM cells are more complex to produce and handle. A proposed way to use a conventional 6T SRAM cell (six transistor for each bit of memory) cache is to subdivide the cache in various sub-cache tables and exploit a simple statistical analysis of the warping process for neighbor pixels. Usually their arrival location is not so different, one or two pixels of difference in L1-norm distance. The proposed way here is to subdivide the 8x8 tiles in four 4x4 tiles. Each tile can compose a row of cache and four equal sized tables should be used, one for each 4x4 tile. In this way, the entire tile can be accessed at the same time if needed. It is also possible to access up to four different 4x4 zones at the same time but they cannot be corresponding parts of two different tiles (e.g. the upper left 4x4 sub-tile of two different 8x8 tiles). This can be handled using four small sized FIFOs, one queue for each table. If in a clock cycle one of the tables is not used by the newcomer pixels or the sub-tile that has to be addressed coincides with the one required by the next element in queue, the corresponding queue(s) can advance to the following element. This mechanism is able to empty the queues in few clock cycles processing many elements at once. This is why the depth of the required FIFO memories should not be more than four pixels per queue. If a pixel in queue has to warp to an unavailable tile, it is put in the queue described in the previous section.

3.4 Postprocessing

As soon as the previous stage signal that it has ended its computation, this stage has to start the final part of the algorithm: the image fusion, the crack filling and, finally, the occluded zone inpaint. Because after this stage the video signal has to be outputted to a video data sink component, the tile based frame has to be reconverted to a scanline raster order. As for the input interface, the output signal can be handled by the traditional video interface composed by HSYNC, VSYNC, Pixel_Data and Pixel_Clock.

3.4.1 Joint image fusion, crack filling and inpaint

As described in the previous chapter, the method proposed in this thesis makes use of a modified Hybrid Median Filter. The classical HMF is a filter that uses a three step sorting operation on a moving window of odd dimension, in this case a 3x3 window. It performs the median filter on the five elements in an horizontal cross shape, then on a diagonal cross shape and, finally, looks for the last median value between these two values and the central pixel. The regions of interest are indicated in grey in Table 3.1. The idea here is to take advantage of all the available information to find the output value. As described in the previous chapter, the inpaint process can be handled directly inside this process. It surely won't match the quality of other more sophisticated methods but it is very straightforward to apply and in many real cases is able to deliver an acceptable result. The key point of this process is that the previous results are directly used for the inpaint task. The only uncertain piece of information is coming from the pixel that will be processed after the current one. In this case there could be black pixels to discard. The basic rule to apply for those elements is:

- if both the views presents a black pixel in that location, leave a black value
- if one of the two pixels is black and the other is valid take the valid one as the pixel value
- if both the pixels present valid values take the average of the two

Now that the number of black pixels should have been reduced to the minimum, it is possible to apply the HMF filter but a crucial step is to not consider the remaining black pixels to compute the medians. In particular the color values have to be sorted

from the smallest to the greatest one, the blank pixels should not be considered (they will simply be the in the lowest value positions) and the median will be taken from the remaining one(s), leaving a black value if all the pixels are black. This removes both the black pixel quantization errors and the other errors related to the quantization, in particular the case when a foreground object warps over an already warped background but leaves some quantization cracks. The maximum value of elements that need a median filter with this approach is five so there is only a small subset of ways to perform the median once the elements have been sorted and the black pixels have been counted. In particular, because the previous pixels should already have been processed, only the current and the next ones in the frame can be black elements. This means that this approach limits the possible cases to six, easily serviceable with a multiplexed system. If the number of remaining valid pixels is odd the median is just the central value of the array, if it is an even value the median is the average of the two central pixels of the list. If all the pixels are black the result will just be a black pixel but this is an extreme case that can be expected only for the upper left corner of the picture. As before, this process have to be pipelined to reach the real-time video processing requirement. If a queue is full the warping pipeline should be stalled in order to let the queue flush its content. The overhead of this solution is large and it would be better to avoid the stall condition. If possible, a true multi-port memory approach would be much more easy and efficient solution to this issue.

(a) Output of fusion and inpaint

(b1)

(b2)

(b) Left and right warped frames

Figure 3.8: Fusion of virtual left and virtual right frames

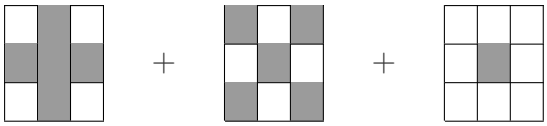


Table 3.1: Hybrid Median Filter: the grey tab

Chapter 4

Conclusion

This chapter is devoted to the exposition of the results obtained, along with a comparison with the previous works in literature. In the last section a list of possible design improvements is presented for future work, outlining the pros and the cons of the various modifications.

4.1 Results

The qualitatively results presented here are based on the MATLAB implementation of the algorithm attached in the appendix A but slightly modified to take into account the reduced numeric precision of fixed point operations. The results with both approaches are very similar, usually within few tenths of dB of Peak Signal to Noise Ratio (PSNR). The two quality metrics used here are the PSNR ([11]) and the SSIM (Structural Similarity) ([12]). The first is an index of the similarity between two images, one noise free (reference image) and one corrupted by noise (compressed or rendered image), valued over the entire image. It is based on the Mean Square Error (MSE) between the reference and the reproduced image. The SSIM is, instead, a perception-based model: it takes into account the structure of the objects represented in the frame, better following the human quality metrics of the scene. It is more significant then PSNR as, in this work, a small amount of numeric error could lead to a shift of a fraction of pixel of the entire reconstructed frame (i.e. cameras not calibrated correctly, resulting in registration errors). The SSIM

index takes this into account, giving a result more similar to the subjective quality assessment. The PSNR result is represented by a positive number, the higher the better. Typical values of PSNR in view synthesis are between 20dB and 40dB. The SSIM result is a number between 0 and 1, the higher the better also in this case. Sometimes the Y-PSNR is used instead of the PSNR. It is calculated only on the luma channel Y in a YCbCr color space, while the original metric takes into account the full RGB color space. In figure 4.1 the values of PSNR and SSIM obtained for the reconstruction of the camera 4 starting from the cameras 3 and 5 of the sequence Ballet [13] over the whole 100 frame sequence are plot in order to demonstrate the constant quality level of the proposed solution. The mean PSNR is about 29.2dB and, more important, the SSIM is about 0.95. This second indicator of the quality of the reconstructed frame is, as reported before, more important as high values of SSIM corresponds to high perceived quality. For example, the work in [14] reports a very high PSNR for the reconstructed scene but the SSIM value is the same as this work. The mean Y-PSNR obtained for this setup is 30.4, less than the solution proposed by [1], but the hardware optimizations presented here are able to save some silicon area and produce a much higher video throughput. In addition, this work is able to handle cases not considered in the previous work: here the epipole can be everywhere, in the previous work it could be only on the outside left or right of the image plane, thus limiting the free viewpoint experience. This work also makes use of a programmable serial processing unit to deal with the matrices, shrinking the required hardware (here the floating point computations can be done by software routines). It is also claimed that the largest divider used is based on the IEEE 754 standard for floating point, in this work the biggest divider is a 28 by 14 bit one in a pipelined array structure plus a leading zero counter and three shifter system (that can be built effectively in VLSI by three-state buffers). The previous work claims for the divider structure about 2.9 thousands equivalent gates just for a sequential implementation, here the combinational array divider needs about 2.6 thousands equivalent NAND logic gates. In return, the throughput is much higher thanks to the described optimizations and parallelization of resources, making the output frame rate independent of the scene complexity even for a ultra high definition video stream. The proposed DIBR equations can be viewed as a natural extension of the method proposed in [15]. Using the LUT based approach

with the proposed method, the dimension of the tables will shrink by 256 times each and a new table of 256x3 entries will be added to take into account the depth parameters. This simple method is a very interesting extension of this previous work. The tables for a 1024x768 pixels were made of 786432 entries the first, 589824 the second plus a third one of 768, for a total of over a million floating point entries. With the proposed method they become just 3072 the first one, 2304 the second one and a third one of 768 entries is added, which sum up to 6144 entries, a 256 times smaller total memory. This should also make the numeric error of less importance as the quantities to be rounded are smaller in the proposed approach. This is a good way to speed up the calculations for GPGPU processing. Sadly, this still does not remove the division operation, the true resource hungry part of the algorithm.

4.2 Limitations of the current work and future possibilities

The current work is based on various assumptions. The first, most important one, consists in the necessity of correctly calibrated cameras. This is usually true but especially the rotation matrices in a 1D parallel camera setup can give a wrong result, in particular in the computation of the position of the epipole. The small numeric value of some elements of the matrix could oscillate around zero and this can lead to wrong epipole values, especially the sign can be a problem. The second assumption used in this work is that the only misbehaviour of the depth channel is related to the misalignment with respect to the color image. This is usually not true as there are always blocking artifacts and inconsistencies between the true depth of the scene and the quantized one. The latter case is due to the depth reconstruction algorithm used and so it largely depends on the input stream quality. Many works employs various depth prefiltering stages with complex functions as the bilateral filter to clean the input depth channels but this is difficult to implement it for real-time processing. In a future work it would be interesting to filter the depth according to the color channels through the Image Guided Filter ([16]) as it is able to smooth the depth channel following the color channel structure with a relatively small computation burden. Then, a big problem is the resampling stage: the output frame pixel grid is

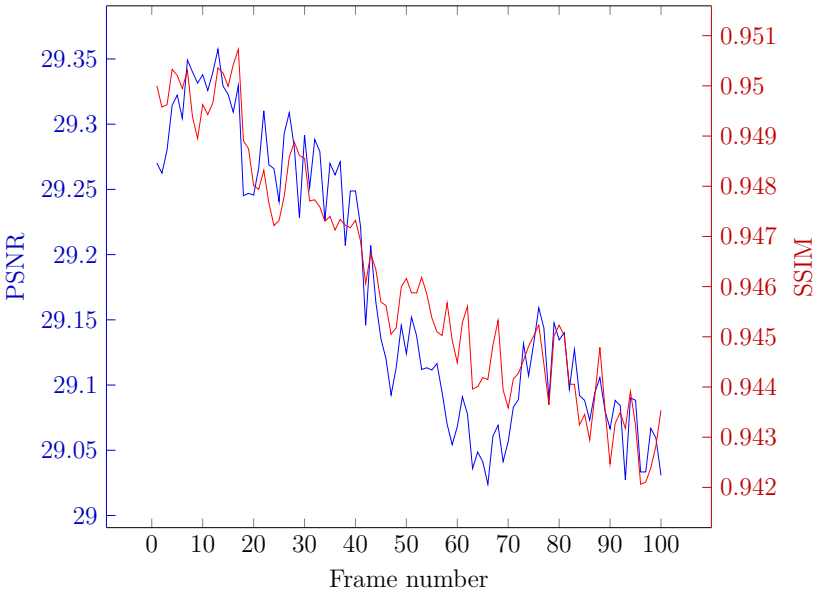


Figure 4.1: PSNR and SSIM values for camera 4 reconstruction starting from camera 3 and camera 5 for the sequence Ballet

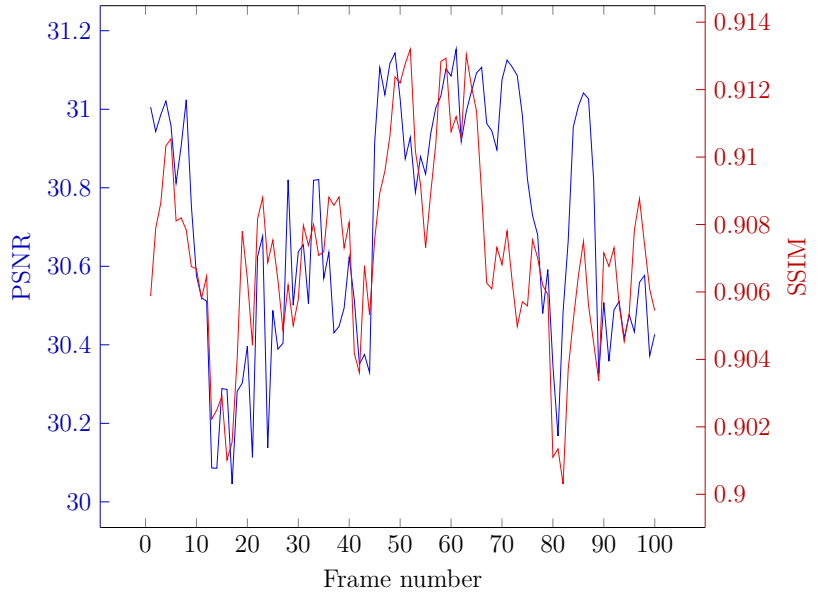


Figure 4.2: PSNR and SSIM values for camera 4 reconstruction starting from camera 3 and camera 5 for the sequence Breakdancers

not coherent with the input sampling grid. This is not easy to solve using a limited computation effort. There are various ways to better resampling the pixels. The first one consists on using a subpixel approach: the input image lattice is oversampled in order to have more input pixel to warp. This is straightforward and easy but an oversampling of factor N requires $2^N - 1$ more calculations per input pixel in the warping stage, along with a much higher memory bandwidth, both internal and external. The second way consists in the backward DIBR approach. At first only the depth channel of both the views is rendered by forward warping then the obtained depths are jointly filtered and used as a basis to pick the color information for each pixel through the reverse warp. This works very well because the depth channels do not contain high frequency textures and so smoothing filters for the resampling operation won't degrade the overall quality. In addition, the backward warping can interpolate between input color pixels to achieve a very good precision. This requires one more processing step with correlated silicon real estate, both for memory and processing. Another way is the *elliptical weighted average* (EWA). It is based on the forward warping only but the output frame size is difficult to handle as it is not fixed like in the simple forward approach ([17]). It embeds an anisotropic filter in order to avoid the aliasing due to the resampling operation. It still needs some costly operations like squaring and divisions so it is best suited as a GPU workload. It is also possible to see the pixel as composed by triangles and render the scene as if it were a true 3D scene, using a complete Z buffer and the GPU rendering pipeline. The last way found, which was also explored (and abandoned for a more traditional implementation) during the evolution of this work is a ray cast approach. For each pixel of the output camera a ray is casted through the scene and the first hit surface sets the color of the pixel, maybe employing also an interpolation to pick the best color value in the neighborhood. This approach directly find the output pixel value without performing the warp so it will be interesting to develop in future works. The DIBR formulæ found in this thesis could be interestingly approached to a backward warping stage as the depth, the reciprocal of the distance between the object and the camera, has been extracted and split from the terms that depend on the input coordinates. The forward-only DIBR used here can work also with a single image reference but the final quality depends on the availability of color information, in particular in occluded zones. The developed postprocessing filter is able to attenuate

the resampling artifacts and inpaint the blank zones but the quality suffers. The filter is well suited for a real-time implementation even if there are better ways to reconstruct the high frequency components of the missing texture regions, at the expense of a much greater processing power and memory usage. The proposed filter is good enough to output a relatively high quality frame but the lack of warped depth channels is a major obstacle for higher quality inpaint. It would be interesting to add a better inpaint engine in future works, maybe with a gradient based detection of features for a multiscale inpaint, like in [18]. It would be also interesting to add in the preprocessing stage a way to create a virtual camera from parameters like rotation, position and focal length without resorting to an external virtual camera generator, maybe using quaternions and the SLERP interpolation method to move smoothly the virtual camera. Finally, the last impasse for this type of algorithms is the memory utilization, both for storage and bandwidth. It would be interesting to add a frame buffer compression engine to save them both, in exchange for a higher computational burden. As the resampling is an intrinsically lossy process, the color channels can be compressed with high compression schemes but the depth channel should be as precise as possible as it directly affects the geometry of the output scene. A lossless compression is required here, or at least a very light lossy scheme, to ensure the best results.

Bibliography

[1] Y. R. Horng, Y. C. Tseng, and T. S. Chang. «VLSI Architecture for Real-Time HD1080p View Synthesis Engine». In: *IEEE Transactions on Circuits and Systems for Video Technology* 21.9 (Sept. 2011), pp. 1329–1340. ISSN: 1051-8215. DOI: 10.1109/TCSVT.2011.2148410.

[2] F. J. Chang, Y. C. Tseng, and T. S. Chang. «A 94fps view synthesis engine for HD1080p video». In: *Visual Communications and Image Processing (VCIP), 2011 IEEE*. Nov. 2011, pp. 1–4. DOI: 10.1109/VCIP.2011.6116009.

[3] J. Wang and L. A. Rønningen. «Real time believable stereo and virtual view synthesis engine for autostereoscopic display». In: *2012 International Conference on 3D Imaging (IC3D)*. Dec. 2012, pp. 1–6. DOI: 10.1109/IC3D.2012.6615128.

[4] M. Schaffner et al. «MADmax: A 1080p stereo-to-multiview rendering ASIC in 65 nm CMOS based on image domain warping». In: *ESSCIRC (ESSCIRC), 2013 Proceedings of the*. Sept. 2013, pp. 61–64. DOI: 10.1109/ESSCIRC.2013.6649072.

[5] P. K. Tsung et al. «A 216fps 4096x2160p 3DTV set-top box SoC for free-viewpoint 3DTV applications». In: *2011 IEEE International Solid-State Circuits Conference*. Feb. 2011, pp. 124–126. DOI: 10.1109/ISSCC.2011.5746247.

[6] Wikipedia. *List of device bit rates — Wikipedia, The Free Encyclopedia*. [Online; accessed 14-August-2016]. 2016. URL: https://en.wikipedia.org/w/index.php?title=List_of_device_bit_rates&oldid=734364934.

[7] Mohiy M. Hadhoud, Kamel A. Moustafa, and Sameh Z. Shenoda. «Digital images inpainting using modified convolution based method». In: vol. 7340. 2009, 73400S. DOI: 10.1117/12.816704. URL: http://dx.doi.org/10.1117/12.816704.

[8] Tampere University. *TTA-Based Co-Design Environment*. 2016. URL: http://tce.cs.tut.fi/index.html.

[9] Gary Sullivan Rico Malvar. *YCoCg-R: A Color Space with RGB Reversibility and Low Dynamic Range*. Tech. rep. July 2003. URL: https://www.microsoft.com/en-us/research/publication/ycocg-r-a-color-space-with-rgb-reversibility-and-low-dynamic-range/.

[10] «Appendix B, Mathematical Computations on Fixed-Point Processors». In: *Digital Media Processing*. Ed. by Hazarathaiah Malepati. Boston: Newnes, 2010, pp. 745–758. ISBN: 978-1-85617-678-1. DOI: http://dx.doi.org/10.1016/B978-1-85617-678-1.00026-0. URL: http://booksite.elsevier.com/9781856176781/on04-app-b-01-18-9781856176781.pdf.

[11] Q. Huynh-Thu and M. Ghanbari. «Scope of validity of PSNR in image/video quality assessment». In: *Electronics Letters* 44.13 (June 2008), pp. 800–801. ISSN: 0013-5194. DOI: 10.1049/el:20080522.

[12] Zhou Wang et al. «Image quality assessment: from error visibility to structural similarity». In: *IEEE Transactions on Image Processing* 13.4 (Apr. 2004), pp. 600–612. ISSN: 1057-7149. DOI: 10.1109/TIP.2003.819861.

[13] Microsoft Research. *MSR 3D Video Dataset*. URL: https://www.microsoft.com/en-us/download/details.aspx?id=52358.

[14] Jiangbin Zheng, Danyang Zhao, and JinChang Ren. «Effective Removal of Artifacts from Views Synthesized using Depth Image Based Rendering». In: ().

[15] Krishna Rao Vijayanagar et al. «Efficient view synthesis for multi-view video plus depth». In: *2013 IEEE International Conference on Image Processing*. IEEE. 2013, pp. 2197–2201.

[16] K. He, J. Sun, and X. Tang. «Guided Image Filtering». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.6 (June 2013), pp. 1397–1409. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2012.213.

[17] Pavlos Mavridis and Georgios Papaioannou. «High Quality Elliptical Texture Filtering on GPU». In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '11. San Francisco, California: ACM, 2011, pp. 23–30. ISBN: 978-1-4503-0565-5.

[18] O. Sims and J. Irvine. «An FPGA Implementation of Pattern-Selective Pyramidal Image Fusion». In: *2006 International Conference on Field Programmable Logic and Applications*. Aug. 2006, pp. 1–4. DOI: 10.1109/FPL.2006.311296.

[19] Julian D. Laderman. «A noncommutative algorithm for multiplying 3×3 matrices using 23 multiplications». In: *Bull. Amer. Math. Soc.* 82.1 (Jan. 1976), pp. 126–128. URL: http://projecteuclid.org/euclid.bams/1183537626.

[20] Nicolas Courtois, Gregory V. Bard, and Daniel Hulme. «A New General-Purpose Method to Multiply 3x3 Matrices Using Only 23 Multiplications». In: *CoRR* abs/1108.2830 (2011). URL: http://arxiv.org/abs/1108.2830.

Appendix A

Matlab test algorithm

This appendix is devoted to the MATLAB code used to validate the correct behaviour of the various parts of the algorithm. Some parts have been simplified, like the tile based addressing. The occlusion compatible scanning order is here implemented in a per pixel basis, without a full tile implementation. In the conclusive chapter of this work the qualitative results of this algorithm are compared to the previous literature and the output virtual frames are presented for subjective image quality inspection. The image processing toolbox has been very effective to load, store and show intermediate images for the algorithm tweak. Many previous implementations have been built in order to test each part of the code, in particular the data width for the warping engine has been optimized trying different view synthesis datasets to measure real world scenarios of physical camera setups to evince the formats to utilize for the various fixed point blocks. This final piece of code has been condensed to avoid external functions that could have made its examination more difficult for the reader. The only custom function here is `load_camera_parameters()` that extract the camera parameters from an input text file. This specific piece of code is custom tailored for the *Ballet* sequence ([13]) as all the cameras share the same values of Z_{far} and Z_{near} (it is a 1D parallel camera setup).

```
1 clc;
2 close all;
```

```
clear;

color_l = imread('color_l.jpg');
color_r = imread('color_r.jpg');
color_v = imread('color_v.jpg');
depth_l = double(rgb2gray(imread('depth_l.png')));
depth_r = double(rgb2gray(imread('depth_r.png')));

[height,width,colors] = size(color_l);

depth_v_r = zeros([height width],'double');
depth_v_l = zeros([height width],'double');

color_v_r = zeros([height width colors],'uint8');
color_v_l = zeros([height width colors],'uint8');
fused_color = zeros([height width colors],'uint8');

%%%% "Ballet" sequence parameters %%%%

Z_far_r = 130.0;
Z_near_r = 42.0;

Z_far_l = 130.0;
Z_near_l = 42.0;

[K_r,R_r,T_r] = load_camera_parameters('camR.txt');
[K_l,R_l,T_l] = load_camera_parameters('camL.txt');
[K_v,R_v,T_v] = load_camera_parameters('camV.txt');

flip = 1; % 1 if the image origin is in the lower left corner, 0 if
    in the upper left

%%%% Input depth selective dilation %%%%
```

```
35
36 NHOOD = [[0 1 0]',[1 1 1]',[0 1 0]'];
37
38 structel = strel('arbitrary',NHOOD);
39 depth_l_tmp = imdilate(depth_l,structel);
40 depth_r_tmp = imdilate(depth_r,structel);
41
42 for j = 1:height
43     for i = 1:width
44         if abs(int16(depth_l_tmp(j,i))   int16(depth_l(j,i))) > 8
45             depth_l(j,i) = depth_l_tmp(j,i);
46         end
47         if abs(int16(depth_r_tmp(j,i))   int16(depth_r(j,i))) > 8
48             depth_r(j,i) = depth_r_tmp(j,i);
49         end
50     end
51 end
52
53 %%%% Flip the intrinsic matrices if needed %%%%
54
55 if flip == 1
56     K_r = [[1 0 0]', [0  1 0]', [0 height 1]']*K_r;
57     K_l = [[1 0 0]', [0  1 0]', [0 height 1]']*K_l;
58     K_v = [[1 0 0]', [0  1 0]', [0 height 1]']*K_v;
59 end
60
61 %%%% Compute forward warping parameters %%%%
62
63 A_r = K_v * R_v * R_r' * inv(K_r);
64 B_r = K_v * ( T_v   R_v * R_r' * T_r );
65 C_r = Z_far_r * A_r;
66 D_r = (Z_far_r/Z_near_r   1) * B_r;
67 E_r = 255.0*(C_r + [0,0,B_r(1);0,0,B_r(2);0,0,B_r(3)]);
```

```
68 epipole_r = [round(B_r(1)/B_r(3)),round(B_r(2)/B_r(3))];
69
70 A_l = K_v * R_v * R_l' * inv(K_l);
71 B_l = K_v * ( T_v   R_v * R_l' * T_l );
72 C_l = Z_far_l * A_l;
73 D_l = (Z_far_l/Z_near_l   1) * B_l;
74 E_l = 255.0*(C_l + [0,0,B_l(1);0,0,B_l(2);0,0,B_l(3)]);
75 epipole_l = [round(B_l(1)/B_l(3)),round(B_l(2)/B_l(3))];
76
77 %%%% Initialize occlusion compatible scanning %%%%
78
79 dist_up_r = abs(1 epipole_r(2));
80 dist_right_r = abs(width epipole_r(1));
81 dist_down_r = abs(height epipole_r(2));
82 dist_left_r = abs(1 epipole_r(1));
83
84 radius_r = max([dist_up_r,dist_right_r,dist_down_r,dist_left_r]);
85
86 flag_up_r = 0;
87 flag_right_r = 0;
88 flag_down_r = 0;
89 flag_left_r = 0;
90
91 start_up_r = [1,1];
92 start_right_r = [1,width];
93 start_down_r = [height,width];
94 start_left_r = [height,1];
95
96 pixel_count = width*height 1;
97 while (pixel_count ~= 0)
98
99     if radius_r == dist_up_r
100         flag_up_r = 1;
```

```
    end
    if radius_r == dist_right_r
        flag_right_r = 1;
    end
    if radius_r == dist_down_r
        flag_down_r = 1;
    end
    if radius_r == dist_left_r
        flag_left_r = 1;
    end

    if ((flag_up_r == 1) && (pixel_count ~= 0))
        current_pixel = start_up_r;
        while ((current_pixel(2) ~= start_right_r(2)+1) && (
pixel_count ~= 0))

            %PROCESS PIXEL
            arrival = E_r*[current_pixel(2);current_pixel(1);1] +
depth_r(current_pixel(1),current_pixel(2))*D_r;
            arrival_u_r = round(arrival(1)/arrival(3));
            arrival_v_r = round(arrival(2)/arrival(3));
            if ((arrival_u_r >= 1) && (arrival_u_r <= width) && (
arrival_v_r >= 1) && (arrival_v_r <= height))
                color_v_r(arrival_v_r,arrival_u_r,:) = color_r(
current_pixel(1),current_pixel(2),:);
            end

            pixel_count = pixel_count   1;
            current_pixel(2) = current_pixel(2)+1;
        end
        % update first and last pixels
        start_up_r(1) = start_up_r(1)+1;
        start_right_r(1) = start_right_r(1)+1;
```

```
130     end
131
132     if ((flag_right_r == 1) && (pixel_count ~= 0))
133         current_pixel = start_right_r;
134         while ((current_pixel(1) ~= start_down_r(1)+1) && (
        pixel_count ~= 0))
135
136             %PROCESS PIXEL
137             arrival = E_r*[current_pixel(2);current_pixel(1);1] +
        depth_r(current_pixel(1),current_pixel(2))*D_r;
138             arrival_u_r = round(arrival(1)/arrival(3));
139             arrival_v_r = round(arrival(2)/arrival(3));
140             if ((arrival_u_r >= 1) && (arrival_u_r <= width) && (
        arrival_v_r >= 1) && (arrival_v_r <= height))
141                 color_v_r(arrival_v_r,arrival_u_r,:) = color_r(
        current_pixel(1),current_pixel(2),:);
142             end
143
144             pixel_count = pixel_count   1;
145             current_pixel(1) = current_pixel(1)+1;
146         end
147         % update first and last pixels
148         start_right_r(2) = start_right_r(2) 1;
149         start_down_r(2) = start_down_r(2) 1;
150     end
151
152     if ((flag_down_r == 1) && (pixel_count ~= 0))
153         current_pixel = start_down_r;
154         while ((current_pixel(2) ~= start_left_r(2) 1) && (
        pixel_count ~= 0))
155
156             %PROCESS PIXEL
```

```
157         arrival = E_r*[current_pixel(2);current_pixel(1);1] +
    depth_r(current_pixel(1),current_pixel(2))*D_r;
158         arrival_u_r = round(arrival(1)/arrival(3));
159         arrival_v_r = round(arrival(2)/arrival(3));
160         if ((arrival_u_r >= 1) && (arrival_u_r <= width) && (
    arrival_v_r >= 1) && (arrival_v_r <= height))
161             color_v_r(arrival_v_r,arrival_u_r,:) = color_r(
    current_pixel(1),current_pixel(2),:);
162         end
163
164         pixel_count = pixel_count   1;
165         current_pixel(2) = current_pixel(2) 1;
166     end
167     % update first and last pixels
168     start_down_r(1) = start_down_r(1) 1;
169     start_left_r(1) = start_left_r(1) 1;
170  end
171
172  if ((flag_left_r == 1) && (pixel_count ~= 0))
173     current_pixel = start_left_r;
174     while ((current_pixel(1) ~= start_up_r(1) 1) && (pixel_count
    ~= 0))
175
176         %PROCESS PIXEL
177         arrival = E_r*[current_pixel(2);current_pixel(1);1] + D_r
    *depth_r(current_pixel(1),current_pixel(2));
178         arrival_u_r = round(arrival(1)/arrival(3));
179         arrival_v_r = round(arrival(2)/arrival(3));
180         if ((arrival_u_r >= 1) && (arrival_u_r <= width) && (
    arrival_v_r >= 1) && (arrival_v_r <= height))
181             color_v_r(arrival_v_r,arrival_u_r,:) = color_r(
    current_pixel(1),current_pixel(2),:);
182         end
```

```
183
184             pixel_count = pixel_count   1;
185             current_pixel(1) = current_pixel(1) 1;
186         end
187         % update first and last pixels
188         start_left_r(2) = start_left_r(2)+1;
189         start_up_r(2) = start_up_r(2)+1;
190     end
191     radius_r = radius_r   1;
192 end
193
194 dist_up_l = abs(1 epipole_l(2));
195 dist_right_l = abs(width epipole_l(1));
196 dist_down_l = abs(height epipole_l(2));
197 dist_left_l = abs(1 epipole_l(1));
198
199 radius_l = max([dist_up_l,dist_right_l,dist_down_l,dist_left_l]);
200
201 flag_up_l = 0;
202 flag_right_l = 0;
203 flag_down_l = 0;
204 flag_left_l = 0;
205
206 start_up_l = [1,1];
207 start_right_l = [1,width];
208 start_down_l = [height,width];
209 start_left_l = [height,1];
210
211 pixel_count = width*height 1;
212 while (pixel_count ~= 0)
213
214     if radius_l == dist_up_l
215         flag_up_l = 1;
```

```
216     end
217     if radius_l == dist_right_l
218         flag_right_l = 1;
219     end
220     if radius_l == dist_down_l
221         flag_down_l = 1;
222     end
223     if radius_l == dist_left_l
224         flag_left_l = 1;
225     end
226 
227     if ((flag_up_l == 1) && (pixel_count ~= 0))
228         current_pixel = start_up_l;
229         while ((current_pixel(2) ~= start_right_l(2)+1) && (
    pixel_count ~= 0))
230 
231             %PROCESS PIXEL
232             arrival = E_l*[current_pixel(2);current_pixel(1);1] +
    depth_l(current_pixel(1),current_pixel(2))*D_l;
233             arrival_u_l = round(arrival(1)/arrival(3));
234             arrival_v_l = round(arrival(2)/arrival(3));
235             if ((arrival_u_l >= 1) && (arrival_u_l <= width) && (
    arrival_v_l >= 1) && (arrival_v_l <= height))
236                 color_v_l(arrival_v_l,arrival_u_l,:) = color_l(
    current_pixel(1),current_pixel(2),:);
237             end
238 
239             pixel_count = pixel_count   1;
240             current_pixel(2) = current_pixel(2)+1;
241         end
242         % update first and last pixels
243         start_up_l(1) = start_up_l(1)+1;
244         start_right_l(1) = start_right_l(1)+1;
```

```
245     end
246
247     if ((flag_right_l == 1) && (pixel_count ~= 0))
248         current_pixel = start_right_l;
249         while ((current_pixel(1) ~= start_down_l(1)+1) && (
    pixel_count ~= 0))
250
251             %PROCESS PIXEL
252             arrival = E_l*[current_pixel(2);current_pixel(1);1] +
    depth_l(current_pixel(1),current_pixel(2))*D_l;
253             arrival_u_l = round(arrival(1)/arrival(3));
254             arrival_v_l = round(arrival(2)/arrival(3));
255             if ((arrival_u_l >= 1) && (arrival_u_l <= width) && (
    arrival_v_l >= 1) && (arrival_v_l <= height))
256                 color_v_l(arrival_v_l,arrival_u_l,:) = color_l(
    current_pixel(1),current_pixel(2),:);
257             end
258
259             pixel_count = pixel_count   1;
260             current_pixel(1) = current_pixel(1)+1;
261         end
262         % update first and last pixels
263         start_right_l(2) = start_right_l(2) 1;
264         start_down_l(2) = start_down_l(2) 1;
265     end
266
267     if ((flag_down_l == 1) && (pixel_count ~= 0))
268         current_pixel = start_down_l;
269         while ((current_pixel(2) ~= start_left_l(2) 1) && (
    pixel_count ~= 0))
270
271             %PROCESS PIXEL
```

```
272         arrival = E_l*[current_pixel(2);current_pixel(1);1] +
    depth_l(current_pixel(1),current_pixel(2))*D_l;
273         arrival_u_l = round(arrival(1)/arrival(3));
274         arrival_v_l = round(arrival(2)/arrival(3));
275         if ((arrival_u_l >= 1) && (arrival_u_l <= width) && (
    arrival_v_l >= 1) && (arrival_v_l <= height))
276             color_v_l(arrival_v_l,arrival_u_l,:) = color_l(
    current_pixel(1),current_pixel(2),:);
277         end
278
279         pixel_count = pixel_count   1;
280         current_pixel(2) = current_pixel(2) 1;
281     end
282     % update first and last pixels
283     start_down_l(1) = start_down_l(1) 1;
284     start_left_l(1) = start_left_l(1) 1;
285  end
286
287  if ((flag_left_l == 1) && (pixel_count ~= 0))
288     current_pixel = start_left_l;
289     while ((current_pixel(1) ~= start_up_l(1) 1) && (pixel_count
    ~= 0))
290
291         %PROCESS PIXEL
292         arrival = E_l*[current_pixel(2);current_pixel(1);1] +
    depth_l(current_pixel(1),current_pixel(2))*D_l;
293         arrival_u_l = round(arrival(1)/arrival(3));
294         arrival_v_l = round(arrival(2)/arrival(3));
295         if ((arrival_u_l >= 1) && (arrival_u_l <= width) && (
    arrival_v_l >= 1) && (arrival_v_l <= height))
296             color_v_l(arrival_v_l,arrival_u_l,:) = color_l(
    current_pixel(1),current_pixel(2),:);
297         end
```

```
298
299             pixel_count = pixel_count   1;
300             current_pixel(1) = current_pixel(1) 1;
301         end
302         % update first and last pixels
303         start_left_l(2) = start_left_l(2)+1;
304         start_up_l(2) = start_up_l(2)+1;
305     end
306
307     radius_l = radius_l 1;
308 end
309
310 figure(1)
311 imshowpair(color_v_r,color_v_l,'montage')
312
313 %%%% Joint image fusion, inpaint and crack filling
314 for j=1:1:height
315    for i=1:1:width
316         if (sum(color_v_r(j,i,:)) ~= 0) && (sum(color_v_l(j,i,:)) ~=
    0)
317             fused_color(j,i,:) = uint8((uint16(color_v_r(j,i,:))+
    uint16(color_v_l(j,i,:)))/2);
318         end
319         if (sum(color_v_r(j,i,:)) == 0) && (sum(color_v_l(j,i,:)) ~=
    0)
320             fused_color(j,i,:) = color_v_l(j,i,:);
321         end
322         if (sum(color_v_r(j,i,:)) ~= 0) && (sum(color_v_l(j,i,:)) ==
    0)
323             fused_color(j,i,:) = color_v_r(j,i,:);
324         end
325         if (sum(color_v_r(j,i,:)) == 0) && (sum(color_v_l(j,i,:)) ==
    0)
```

```
            fused_color(j,i,:) = [0 0 0];
        end
    end
end

figure(2)
imshow(fused_color)

fused_color = padarray(fused_color,[1 1 0],'symmetric');

for c=1:1:colors
    for j=2:1:height
        for i=2:1:width
            submatrix = fused_color((j 1):(j+1),(i 1):(i+1),c);
            diagcross = [submatrix(1,1) submatrix(1,3) submatrix(2,2)
    submatrix(3,1) submatrix(3,3)];
            horcross = [submatrix(1,2) submatrix(2,1) submatrix(2,2)
    submatrix(2,3) submatrix(3,2)];
            diagcross_median = median(diagcross(find(diagcross)));
            horcross_median = median(horcross(find(horcross)));
            hybrid_median = [diagcross_median horcross_median
    submatrix(2,2)];
            fused_color(j,i,c) = median(hybrid_median(find(
    hybrid_median)));
        end
    end
end
fused_color = fused_color(2:height+1,2:width+1,:);

figure(3)
imshow(fused_color)
```

Appendix B

Warping parameters

This appendix is devoted to the derivation of the warping constants needed to perform the warping part of the algorithm.

The basic equation for the DIBR algorithm is the following one:

$$\begin{pmatrix} u_v \\ v_v \\ 1 \end{pmatrix} z_v = \mathbf{A} \begin{pmatrix} u_r \\ v_r \\ 1 \end{pmatrix} z_r + \mathbf{b} \tag{B.1}$$

where

$$\mathbf{A} = \mathbf{K}_v \mathbf{R}_v \mathbf{R}_r^{-1} \mathbf{K}_r^{-1} \tag{B.2}$$

$$\mathbf{b} = \mathbf{K}_v \left(\mathbf{R}_v \mathbf{R}_r^{-1} \mathbf{t}_r - \mathbf{t}_v \right) \tag{B.3}$$

The matrices $\mathbf{A}$ and $\mathbf{b}$ are important as they will be used for the next algebraic manipulations.

B.1 Matrix A

The matrix $\mathbf{A}$ is composed by intrinsic and rotation matrices. One important thing to notice is that a rotation matrix is by definition orthogonal, this means that the condition number should be exactly one (apart from small numeric errors due to

a finite representation of the rotation) so errors are not magnified when the two rotation matrices are multiplied by each other. Another very interesting property of orthogonal matrices is that $\mathbf{Q}^T\mathbf{Q} = \mathbf{Q}\mathbf{Q}^T = 1$, in other words $\mathbf{Q}^{-1} = \mathbf{Q}^T$. The inverse of an orthogonal matrix its equal to its transpose.

$$\mathbf{A} = \mathbf{K}_v\mathbf{R}_v\mathbf{R}_r^{-1}\mathbf{K}_r^{-1} = \mathbf{K}_v\mathbf{R}_v\mathbf{R}_r^{T}\mathbf{K}_r^{-1} = \mathbf{K}_v\mathbf{R}_{TOT}\mathbf{K}_r^{-1} \tag{B.4}$$

$$\mathbf{R}_{TOT} = \mathbf{R}_v\mathbf{R}_r^{T} \tag{B.5}$$

The inverse of a full 3x3 matrix is expensive to perform so this is an important shortcut to the final result. The naïve way of performing this needs 27 multiplications and 18 sums. A better approach is able to cut the multiplication number down to 23 but the sums increase up to 98 (Laderman solution [19]) or 107 (Courtois, Bard and Hulme solution [20]). The intrinsic matrix is upper triangular with a 1 in the lower right corner. The inverse of a matrix structured like this is again an upper triangular matrix with a 1 in the lower right corner

$$\mathbf{K}_r^{-1} = \begin{pmatrix} a_x & \gamma & c_x \\ 0 & a_y & c_y \\ 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} \frac{1}{a_x} & -\frac{\gamma}{a_x \cdot a_y} & \frac{\gamma \cdot c_y - a_y \cdot c_x}{a_x \cdot a_y} \\ 0 & \frac{1}{a_y} & -\frac{c_y}{a_y} \\ 0 & 0 & 1 \end{pmatrix} \tag{B.6}$$

This is an important result because it shows a simple, exact way to compute analytically the matrix $\mathbf{A}$.

$$\mathbf{A} = \mathbf{K}_v\mathbf{R}_{TOT}\mathbf{K}_r^{-1} =$$
$$= \begin{pmatrix} a_{v,x} & \gamma_v & c_{v,x} \\ 0 & a_{v,y} & c_{v,y} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{R}_{1,1} & \mathbf{R}_{1,2} & \mathbf{R}_{1,3} \\ \mathbf{R}_{2,1} & \mathbf{R}_{2,2} & \mathbf{R}_{2,3} \\ \mathbf{R}_{3,1} & \mathbf{R}_{3,2} & \mathbf{R}_{3,3} \end{pmatrix} \begin{pmatrix} \frac{1}{a_{r,x}} & -\frac{\gamma_r}{a_{r,x} \cdot a_{r,y}} & \frac{\gamma_r \cdot c_{r,y} - a_{r,y} \cdot c_{r,x}}{a_{r,x} \cdot a_{r,y}} \\ 0 & \frac{1}{a_{r,y}} & -\frac{c_{r,y}}{a_{r,y}} \\ 0 & 0 & 1 \end{pmatrix} \tag{B.7}$$

B.2 Vector b

The vector $\mathbf{b}$ needs, again, the matrix $\mathbf{R}_{TOT}$ to perform

$$\mathbf{b} = \mathbf{K}_v\left(\mathbf{R}_v\mathbf{R}_r^{-1}\mathbf{t}_r - \mathbf{t}_v\right) = \mathbf{K}_v\left(\mathbf{R}_{TOT}\mathbf{t}_r - \mathbf{t}_v\right) \tag{B.8}$$

This time there are no division in the result so this vector of parameters is faster to perform. The first step is the rotation of the vector $\mathbf{t}_r$ by the matrix $\mathbf{R}_{TOT}$, then the vector $\mathbf{t}_v$ is subtracted from it and, eventually, the result is again multiplied by the upper triangular matrix $\mathbf{K}_v$.

B.3 Final equation

The last step to perform is the inclusion of the Z parameters into the previous results in order to finally reach the true fast DIBR parameters, as in equation (2.17). The matrix A has to be multiplied by a term Z_{far} to exploit the homogeneous coordinate properties.

B.4 Summary of operations

All these operations have to be performed once per frame, in particular each time the physical camera setup changes. As these calculations are too expensive to be performed in real time, a sequential preprocessing stage is needed before the true forward warping. A frame level pipeline stage will be inserted here, as described in the previous chapters, in order to give the required time to the preprocessing core to perform all the required operations. In order to save area, the floating point operations should be performed in a software emulation environment.

```
1  #include "tceops.h"
2
3  int main()
4  {
5    double tmp_R_matrix[3][3] = {{0}};
6    double tmp_K_inv_matrix[3][3] = {{0}};
7    double tmp_KR_matrix[3][3] = {{0}};
8
9    double K_r[3][3] = {{0}};
10   double R_r[3][3] = {{0}};
11   double A_r[3][3] = {{0}};
```

```
12  double E_r[3][3] = {{0}};
13  double K_l[3][3] = {{0}};
14  double R_l[3][3] = {{0}};
15  double A_l[3][3] = {{0}};
16  double E_l[3][3] = {{0}};
17
18  double T_r[3] = {0};
19  double B_r[3] = {0};
20  double D_r[3] = {0};
21  double T_l[3] = {0};
22  double B_l[3] = {0};
23  double D_l[3] = {0};
24
25  char i = 0, j = 0, k = 0;
26  char new_params_valid = 0;
27  double tmp = 0, det = 0;
28  double Z_near_r = 0, Z_far_r = 0, Z_near_l = 0, Z_far_l = 0;
29  int height_r = 0, width_r = 0, height_l = 0, width_l = 0, height_v
       = 0, width_v = 0;
30  char flip_r = 0, flip_l = 0, flip_v = 0;
31  double epi_r_u_float = 0, epi_r_v_float = 0;
32  double epi_l_u_float = 0, epi_l_v_float = 0;
33  long epi_r_u = 0, epi_r_v = 0;
34  long epi_l_u = 0, epi_l_v = 0;
35
36
37
38  while(1)
39  {
40    READ_EXT_SIGNAL(new_params_valid); // read external valid signal
41
42    if (new_params_valid == 1) // if there are new parameters, load
         and process them
```

```
43    {
44      LOAD_PARAMS(K_r[3][3],R_r[3][3],T_r[3],Z_near_r,Z_far_r,flip_r,
            height_r,width_r); // right view parameters, double float
            format
45      LOAD_PARAMS(K_l[3][3],R_l[3][3],T_l[3],Z_near_l,Z_far_l,flip_l,
            height_v,width_v); // left view parameters, double float
            format
46      LOAD_PARAMS(K_v[3][3],R_v[3][3],T_v[3],flip_v,height_v,width_v)
            ; // virtual view parameters, double float format
47
48      if (flip_r == 1)
49      {
50        K_r[2][2] = -K_r[2][2];
51        K_r[2][3] = height_r - K_r[2][3];
52      }
53
54      if (flip_l == 1)
55      {
56        K_l[2][2] = -K_l[2][2];
57        K_l[2][3] = height_l - K_l[2][3];
58      }
59
60      if (flip_v == 1)
61      {
62        K_v[2][2] = -K_v[2][2];
63        K_v[2][3] = height_v - K_v[2][3];
64      }
65
66      ///// right view /////
67
68      // multiply the rotation matrices (R_TOT)
69      for (i = 0; i < 3; i++)
70      {
```

```
71          for (j = 0; j < 3; j++)
72          {
73            tmp = 0;
74            for (k = 0; k < 3; k++)
75            {
76              tmp = tmp + R_v[i][k] * R_r[j][k]; // includes R_r
                    transposition
77            }
78            tmp_R_matrix[i][j] = tmp;
79          }
80        }
81
82        // invert K_r using definition
83        tmp_K_inv_matrix[1][1] = 1 / K_r[1][1];
84        tmp_K_inv_matrix[2][2] = 1 / K_r[2][2];
85        tmp_K_inv_matrix[2][3] = - K_r[2][3] * tmp_K_inv_matrix[2][2];
86        tmp_K_inv_matrix[1][2] = - K_r[1][2] * tmp_K_inv_matrix[1][1] *
              tmp_K_inv_matrix[2][2];
87        tmp_K_inv_matrix[1][3] = - tmp_K_inv_matrix[1][2] * K_r[2][3]
             - K_r[1][3] * tmp_K_inv_matrix[1][1];
88        tmp_K_inv_matrix[2][1] = 0;
89        tmp_K_inv_matrix[3][1] = 0;
90        tmp_K_inv_matrix[3][2] = 0;
91        tmp_K_inv_matrix[3][3] = 1;
92
93        // compute (K_v * R_TOT)
94        tmp_KR_matrix[1][1] = K_v[1][1] * tmp_R_matrix[1][1] + K_v
             [1][2] * tmp_R_matrix[2][1] + K_v[1][3] * tmp_R_matrix
             [3][1];
95        tmp_KR_matrix[1][2] = K_v[1][1] * tmp_R_matrix[1][2] + K_v
             [1][2] * tmp_R_matrix[2][2] + K_v[1][3] * tmp_R_matrix
             [3][2];
```

```
96      tmp_KR_matrix[1][3] = K_v[1][1] * tmp_R_matrix[1][3] + K_v
           [1][2] * tmp_R_matrix[2][3] + K_v[1][3] * tmp_R_matrix
           [3][3];
97      tmp_KR_matrix[2][1] = K_v[2][2] * tmp_R_matrix[2][1] + K_v
           [2][3] * tmp_R_matrix[3][1];
98      tmp_KR_matrix[2][2] = K_v[2][2] * tmp_R_matrix[2][2] + K_v
           [2][3] * tmp_R_matrix[3][2];
99      tmp_KR_matrix[2][3] = K_v[2][2] * tmp_R_matrix[2][3] + K_v
           [2][3] * tmp_R_matrix[3][3];
100     tmp_KR_matrix[3][1] = tmp_R_matrix[3][1];
101     tmp_KR_matrix[3][2] = tmp_R_matrix[3][2];
102     tmp_KR_matrix[3][3] = tmp_R_matrix[3][3];
103
104     // compute A_r = (K_v * R_TOT) * inv(K_r)
105     A_r[1][1] = tmp_KR_matrix[1][1] * tmp_K_inv_matrix[1][1];
106     A_r[1][2] = tmp_KR_matrix[1][1] * tmp_K_inv_matrix[1][2] +
           tmp_KR_matrix[1][2] * tmp_K_inv_matrix[2][2];
107     A_r[1][3] = tmp_KR_matrix[1][1] * tmp_K_inv_matrix[1][3] +
           tmp_KR_matrix[1][2] * tmp_K_inv_matrix[2][3] + tmp_KR_matrix
           [1][3];
108     A_r[2][1] = tmp_KR_matrix[2][1] * tmp_K_inv_matrix[1][1];
109     A_r[2][2] = tmp_KR_matrix[2][1] * tmp_K_inv_matrix[1][2] +
           tmp_KR_matrix[1][2] * tmp_K_inv_matrix[2][2];
110     A_r[2][3] = tmp_KR_matrix[2][1] * tmp_K_inv_matrix[1][3] +
           tmp_KR_matrix[1][2] * tmp_K_inv_matrix[2][3] + tmp_KR_matrix
           [2][3];
111     A_r[3][1] = tmp_KR_matrix[3][1] * tmp_K_inv_matrix[1][1];
112     A_r[3][2] = tmp_KR_matrix[3][1] * tmp_K_inv_matrix[1][2] +
           tmp_KR_matrix[1][2] * tmp_K_inv_matrix[2][2];
113     A_r[3][3] = tmp_KR_matrix[3][1] * tmp_K_inv_matrix[1][3] +
           tmp_KR_matrix[1][2] * tmp_K_inv_matrix[2][3] + tmp_KR_matrix
           [3][3];
114
```

```
115         // compute B_r = (K_v * R_TOT) * T_r - K_v * T_v
116         B_r[1] = tmp_KR_matrix[1][1] * T_r[1] + tmp_KR_matrix[1][2] *
                T_r[2] + tmp_KR_matrix[1][3] * T_r[3] - K_v[1][1] * T_v[1]
                - K_v[1][2] * T_v[2] - K_v[1][3] * T_v[3];
117         B_r[2] = tmp_KR_matrix[2][1] * T_r[1] + tmp_KR_matrix[2][2] *
                T_r[2] + tmp_KR_matrix[2][3] * T_r[3] - K_v[2][1] * T_v[1]
                - K_v[2][2] * T_v[2] - K_v[2][3] * T_v[3];
118         B_r[3] = tmp_KR_matrix[3][1] * T_r[1] + tmp_KR_matrix[3][2] *
                T_r[2] + tmp_KR_matrix[3][3] * T_r[3] - K_v[3][1] * T_v[1]
                - K_v[3][2] * T_v[2] - K_v[3][3] * T_v[3];
119
120         // compute E_r
121         E_r[1][1] = 255 * (Z_far_r * A_r[1][1]);
122         E_r[1][2] = 255 * (Z_far_r * A_r[1][2]);
123         E_r[1][3] = 255 * (Z_far_r * A_r[1][3] + B_r[1]);
124         E_r[2][1] = 255 * (Z_far_r * A_r[2][1]);
125         E_r[2][2] = 255 * (Z_far_r * A_r[2][2]);
126         E_r[2][3] = 255 * (Z_far_r * A_r[2][3] + B_r[2]);
127         E_r[3][1] = 255 * (Z_far_r * A_r[2][1]);
128         E_r[3][2] = 255 * (Z_far_r * A_r[2][2]);
129         E_r[3][3] = 255 * (Z_far_r * A_r[2][3] + B_r[3]);
130
131         tmp = Z_far_r/Z_near_r - 1;
132
133         // compute D_r
134         D_r[1] = tmp * B_r[1];
135         D_r[2] = tmp * B_r[2];
136         D_r[3] = tmp * B_r[3];
137
138         // compute epipole for right view with rounding
139         epi_r_u_float = B_r[1]/B_r[3];
140         epi_r_v_float = B_r[2]/B_r[3];
141         if (epi_r_u_float >= 0)
```

```
142       {
143         epi_r_u = (long)(epi_r_u_float + 0.5);
144       }
145       else
146       {
147         epi_r_u = (long)(epi_r_u_float - 0.5);
148       }
149       if (epi_r_v_float >= 0)
150       {
151         epi_r_v = (long)(epi_r_v_float + 0.5);
152       }
153       else
154       {
155         epi_r_v = (long)(epi_r_v_float - 0.5);
156       }
157
158       ///// left view /////
159
160       // multiply the rotation matrices (R_TOT)
161       for (i = 0; i < 3; i++)
162       {
163         for (j = 0; j < 3; j++)
164         {
165           tmp = 0;
166           for (k = 0; k < 3; k++)
167           {
168             tmp = tmp + R_v[i][k] * R_l[j][k]; // includes R_l
                      transposition
169           }
170           tmp_R_matrix[i][j] = tmp;
171         }
172       }
173
```

```
174        // invert K_l using definition
175        tmp_K_inv_matrix[1][1] = 1 / K_l[1][1];
176        tmp_K_inv_matrix[2][2] = 1 / K_l[2][2];
177        tmp_K_inv_matrix[2][3] = - K_l[2][3] * tmp_K_inv_matrix[2][2];
178        tmp_K_inv_matrix[1][2] = - K_l[1][2] * tmp_K_inv_matrix[1][1] *
               tmp_K_inv_matrix[2][2];
179        tmp_K_inv_matrix[1][3] = - tmp_K_inv_matrix[1][2] * K_l[2][3]
              - K_l[1][3] * tmp_K_inv_matrix[1][1];
180        tmp_K_inv_matrix[2][1] = 0;
181        tmp_K_inv_matrix[3][1] = 0;
182        tmp_K_inv_matrix[3][2] = 0;
183        tmp_K_inv_matrix[3][3] = 1;
184
185        // compute (K_v * R_TOT)
186        tmp_KR_matrix[1][1] = K_v[1][1] * tmp_R_matrix[1][1] + K_v
              [1][2] * tmp_R_matrix[2][1] + K_v[1][3] * tmp_R_matrix
              [3][1];
187        tmp_KR_matrix[1][2] = K_v[1][1] * tmp_R_matrix[1][2] + K_v
              [1][2] * tmp_R_matrix[2][2] + K_v[1][3] * tmp_R_matrix
              [3][2];
188        tmp_KR_matrix[1][3] = K_v[1][1] * tmp_R_matrix[1][3] + K_v
              [1][2] * tmp_R_matrix[2][3] + K_v[1][3] * tmp_R_matrix
              [3][3];
189        tmp_KR_matrix[2][1] = K_v[2][2] * tmp_R_matrix[2][1] + K_v
              [2][3] * tmp_R_matrix[3][1];
190        tmp_KR_matrix[2][2] = K_v[2][2] * tmp_R_matrix[2][2] + K_v
              [2][3] * tmp_R_matrix[3][2];
191        tmp_KR_matrix[2][3] = K_v[2][2] * tmp_R_matrix[2][3] + K_v
              [2][3] * tmp_R_matrix[3][3];
192        tmp_KR_matrix[3][1] = tmp_R_matrix[3][1];
193        tmp_KR_matrix[3][2] = tmp_R_matrix[3][2];
194        tmp_KR_matrix[3][3] = tmp_R_matrix[3][3];
195
```

```
196     // compute A_l = (K_v * R_TOT) * inv(K_l)
197     A_l[1][1] = tmp_KR_matrix[1][1] * tmp_K_inv_matrix[1][1];
198     A_l[1][2] = tmp_KR_matrix[1][1] * tmp_K_inv_matrix[1][2] +
            tmp_KR_matrix[1][2] * tmp_K_inv_matrix[2][2];
199     A_l[1][3] = tmp_KR_matrix[1][1] * tmp_K_inv_matrix[1][3] +
            tmp_KR_matrix[1][2] * tmp_K_inv_matrix[2][3] + tmp_KR_matrix
            [1][3];
200     A_l[2][1] = tmp_KR_matrix[2][1] * tmp_K_inv_matrix[1][1];
201     A_l[2][2] = tmp_KR_matrix[2][1] * tmp_K_inv_matrix[1][2] +
            tmp_KR_matrix[1][2] * tmp_K_inv_matrix[2][2];
202     A_l[2][3] = tmp_KR_matrix[2][1] * tmp_K_inv_matrix[1][3] +
            tmp_KR_matrix[1][2] * tmp_K_inv_matrix[2][3] + tmp_KR_matrix
            [2][3];
203     A_l[3][1] = tmp_KR_matrix[3][1] * tmp_K_inv_matrix[1][1];
204     A_l[3][2] = tmp_KR_matrix[3][1] * tmp_K_inv_matrix[1][2] +
            tmp_KR_matrix[1][2] * tmp_K_inv_matrix[2][2];
205     A_l[3][3] = tmp_KR_matrix[3][1] * tmp_K_inv_matrix[1][3] +
            tmp_KR_matrix[1][2] * tmp_K_inv_matrix[2][3] + tmp_KR_matrix
            [3][3];
206
207     // compute B_l = (K_v * R_TOT) * T_r - K_v * T_v
208     B_l[1] = tmp_KR_matrix[1][1] * T_l[1] + tmp_KR_matrix[1][2] *
            T_l[2] + tmp_KR_matrix[1][3] * T_l[3] - K_v[1][1] * T_v[1]
            - K_v[1][2] * T_v[2] - K_v[1][3] * T_v[3];
209     B_l[2] = tmp_KR_matrix[2][1] * T_l[1] + tmp_KR_matrix[2][2] *
            T_l[2] + tmp_KR_matrix[2][3] * T_l[3] - K_v[2][1] * T_v[1]
            - K_v[2][2] * T_v[2] - K_v[2][3] * T_v[3];
210     B_l[3] = tmp_KR_matrix[3][1] * T_l[1] + tmp_KR_matrix[3][2] *
            T_l[2] + tmp_KR_matrix[3][3] * T_l[3] - K_v[3][1] * T_v[1]
            - K_v[3][2] * T_v[2] - K_v[3][3] * T_v[3];
211
212     // compute E_l
213     E_l[1][1] = 255 * (Z_far_l * A_l[1][1]);
```

```
214       E_l[1][2] = 255 * (Z_far_l * A_l[1][2]);
215       E_l[1][3] = 255 * (Z_far_l * A_l[1][3] + B_l[1]);
216       E_l[2][1] = 255 * (Z_far_l * A_l[2][1]);
217       E_l[2][2] = 255 * (Z_far_l * A_l[2][2]);
218       E_l[2][3] = 255 * (Z_far_l * A_l[2][3] + B_l[2]);
219       E_l[3][1] = 255 * (Z_far_l * A_l[2][1]);
220       E_l[3][2] = 255 * (Z_far_l * A_l[2][2]);
221       E_l[3][3] = 255 * (Z_far_l * A_l[2][3] + B_l[3]);
222
223       tmp = Z_far_l/Z_near_l - 1;
224
225       // compute D_l
226       D_l[1] = tmp * B_l[1];
227       D_l[2] = tmp * B_l[2];
228       D_l[3] = tmp * B_l[3];
229
230       // compute epipole for right view with rounding
231       epi_l_u_float = B_l[1]/B_l[3];
232       epi_l_v_float = B_l[2]/B_l[3];
233       if (epi_l_u_float >= 0)
234       {
235         epi_l_u = (long)(epi_l_u_float + 0.5);
236       }
237       else
238       {
239         epi_l_u = (long)(epi_l_u_float - 0.5);
240       }
241       if (epi_l_v_float >= 0)
242       {
243         epi_l_v = (long)(epi_l_v_float + 0.5);
244       }
245       else
246       {
```

```
247         epi_l_v = (long)(epi_l_v_float - 0.5);
248       }
249     }
250
251     WRITE_EXT_SIGNAL(new_params_valid); //  signal to the next stage
            that the new parameters are ready to be used
252   }
253   return 0;
254 }
```