

A Flexible Architecture for Fisheye Correction in Automotive Rear-View Cameras

Introduction

Fisheye cameras are finding an increasing number of applications in automobile rear-view imaging systems due to their ultra-wide-angle properties and cost-effectiveness. However, while fisheye lenses (1) provide very large wide-angle views (theoretically the entire frontal hemispheric view of 180°) the images produced suffer from severe distortion as a result of the hemispherical scene being projected onto a flat surface.

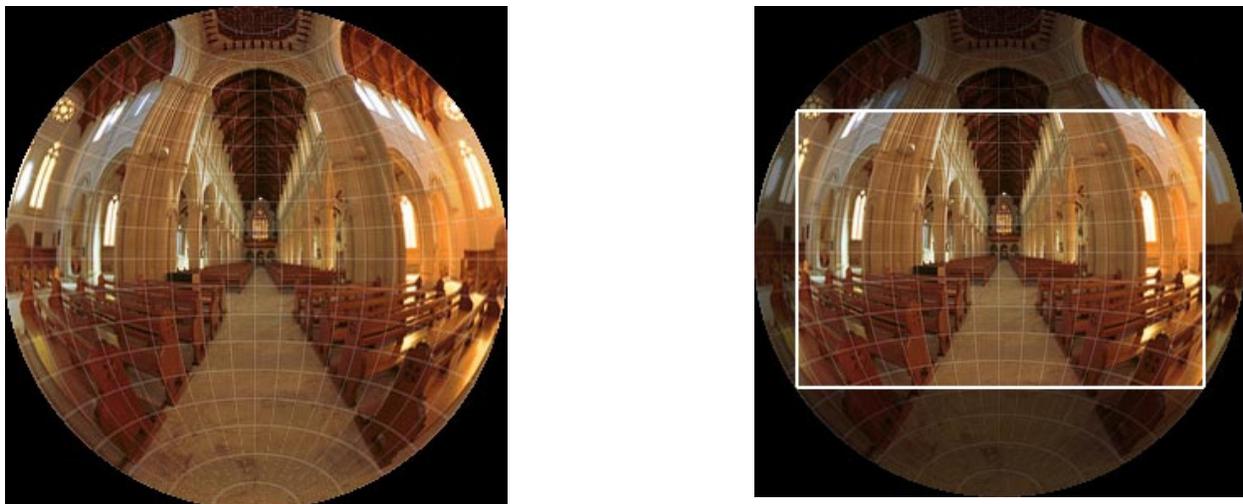
For a viewer of fisheye images, such distortion can be both unusual and confusing. Therefore, it is desirable that the images captured by fisheye cameras be corrected to approximately rectilinear versions before being presented to viewers in applications such as automobile rear-view cameras. For cases where the basic camera parameters are known, correcting fisheye distortion is relatively straightforward mathematically. However, given the intensive computations involved, it cannot be easily implemented on a FPGA. This paper discusses an innovative architecture developed by Altera and Manipal Dot Net (MDN) to perform fisheye correction on a FPGA when basic camera parameters are known.

Fisheye Distortion and Mapping Functions

Fisheye lenses achieve extremely wide fields of view (FOVs) by foregoing the perspective (rectilinear) mapping common to non-fisheye lenses and opting instead for a special mapping (e.g., equisolid angle) that gives images the characteristic convex appearance shown in Figure 1 (left). The radial distortion caused by fisheye mappings is one in which image magnification decreases with distance from the optical axis. Also known as “barrel distortion,” the apparent effect is that of an image that has been mapped around a sphere. As result, fisheye images do not preserve the most important feature of rectilinear images, which is that they map straight lines in the scene onto straight lines in the image. There are two kinds of fisheye images:

- *Circular (hemispherical) fisheye images* are formed when the entire hemispherical view is projected onto a circle within the film frame. In other words, the image circle is inscribed in the film or sensor area. These have a 180° angle of view along the vertical and the horizontal directions as shown in Figure 1(left).
- *Full-frame fisheye images* are formed when the hemispherical image is circumscribed around the film or sensor area as depicted in Figure 1 (right). These have a 180° angle of view along the diagonal, while the horizontal and vertical angles of view are smaller.

Figure 1. Fisheye Images—Circular (left) and Full-Frame (right)

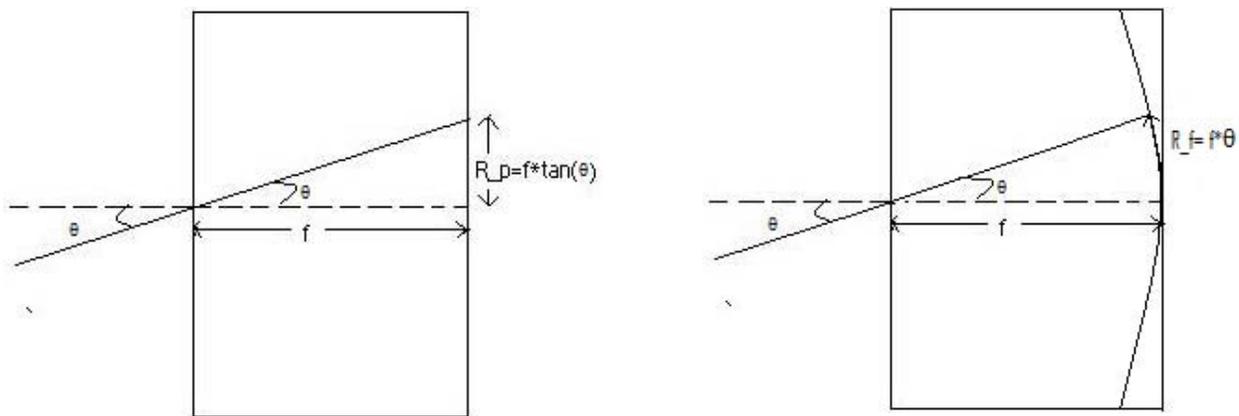


Clearly, a circular fisheye can be made full-frame if it is captured with a smaller sensor or film size. A fisheye lens is characterized by two basic parameters: the focal length and the FOV. In a fisheye camera, the parameters are related for a given sensor or film size. Different fisheye lenses distort images differently and the nature of the distortion is defined by their mapping function.

If θ is the angle between a point in the real world and the optical axis, which goes from the center of the image through the center of the lens, the radial position R of a point on the image is related to θ and to the focal length f of the lens for different mapping functions (2):

- *Perspective projection (normal, non-fisheye lens):* $R = f \tan(\theta)$. This simply works like a pinhole camera and is the basis for the rectilinear distortion-free mapping of normal cameras as shown in Figure 2 (left).
- *Linear scaled (equidistant):* $R = f\theta$, where θ is in radians as shown in Figure 2 (right). This is the simplest mapping function for a fisheye lens and it clearly indicates that for a fisheye lens, the radial position of a point on the film is different from that of a perspective mapping and thus is shifted to a different position.

Figure 2. Perspective Projection (left) and Linear-Scaled Projection (right)



Similarly, other mapping functions for fisheye lens are possible:

- *Equisolid angle:* $R = 2f \sin(\theta/2)$. This popular mapping function is assumed for our analysis.
- *Orthographic:* $R = f \sin(\theta)$.
- *Stereographic (conform):* $R = 2f \tan(\theta/2)$.

Algorithm Description

As described above, the radial position of a point in a fisheye image (R_f) is different from that in a perspective image (R_p). Therefore, the task of correcting a distorted fisheye image is one of finding a relationship between R_p and R_f . This is found by solving for the unknowns in the two equations defining the perspective mapping and the fisheye mapping. Since solving those equations involves the computation of trigonometric functions, this is a difficult task to implement on an FPGA.

Altera and MDN have developed a novel method that simplifies the procedure for correcting a distorted fisheye image. The basic idea is based on the observation that the relationship between R_p and R_f is completely determined by the camera geometry, i.e., the focal length (f) and the FOV. This implies that it can be pre-computed and stored in the form of a look-up table (LUT). The FPGA's task then is to use the LUT to map the pixels of the distorted fisheye input image to that of the corrected output image. Since this involves sub-pixel rendering, the method requires the FPGA perform some form of pixel interpolation. The 9-point interpolation method, a simple and very efficient form of pixel interpolation, produces a tolerable and distortion-free output image.

Computation of the LUT

With the input frame captured by the fisheye camera denoted as the source image and the corrected output as the target image, the task of correcting the source fisheye distorted image can be defined as follows: *For every pixel location in the target image, compute its corresponding pixel location in the source image(3).*

Let x_p and y_p be the x and y coordinates, respectively, of a target pixel in the output perspective (rectilinear) image, and similarly, let x_f and y_f be those of a source pixel in the input fisheye image. Then assuming an equisolid angle mapping for the fisheye image, the following equations hold true:

$$R_f = 2f \sin(\theta/2)$$

$$R_p = f \tan(\theta)$$

$$x_p/y_p = x_f/y_f$$

where R_p is the radial position of a pixel on the perspective image from the center and R_f is the radial position of a pixel on the fisheye image from the center. Assuming the center of the image corresponds to the center of the fisheye lens and eliminating θ between the above three equations gives:

$$x_f = \frac{2x_p \sin(\tan^{-1}(\lambda/2))}{\lambda}$$

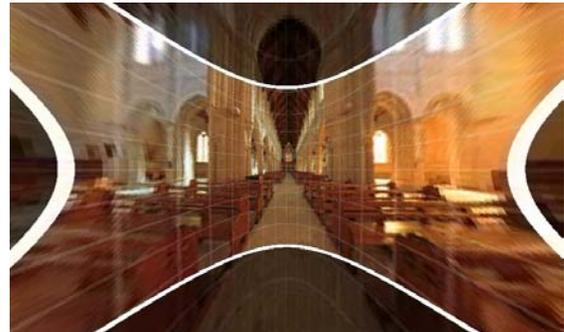
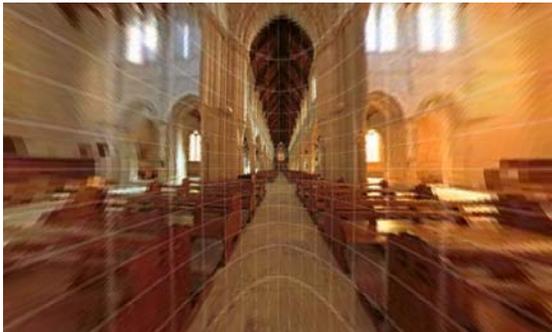
$$y_f = \frac{2y_p \sin(\tan^{-1}(\lambda/2))}{\lambda}$$

where $\lambda = \left(2\sqrt{x_p^2 + y_p^2}\right)/f$ and f is the focal length in pixels, which can be calculated as $f = \frac{\text{image_width}}{4 \sin(\text{FOV}_{\text{horz}}/2)}$.

Thus for every target pixel, the corresponding pixel location in the fisheye image can be computed and stored in the form of an LUT. All parameters needed for LUT generation are known beforehand, such as the imager characteristics (including imager FOV and size of the input frame) and display system characteristics (including the display size). Therefore for a given imager and display system, the LUT is computed only once and off-line.

In **Figure 3** left and right, the results of correcting the fisheye images of **Figure 1** left and right, respectively, are shown. Note that to get a corrected fisheye image with a 180° FOV along some direction requires that the size of the corrected image be infinite. Since this is impossible, the FOV of the target is restricted to less than 180°. In the case of a full-frame fisheye image, where the FOV of the source image along horizontal and vertical directions is less than 180°, if the FOV of the target image is greater than that of the source fisheye, then points outside the FOV of the source image are rendered black. This produces the characteristic “bow tie” effect in which the corners are stretched out as depicted in **Figure 3** (right). This can be avoided by ensuring that the FOV of the target is always sufficiently smaller than that of the source fisheye.

Figure 3. Fisheye-Corrected Images—Circular (left) and Full-Frame (right)

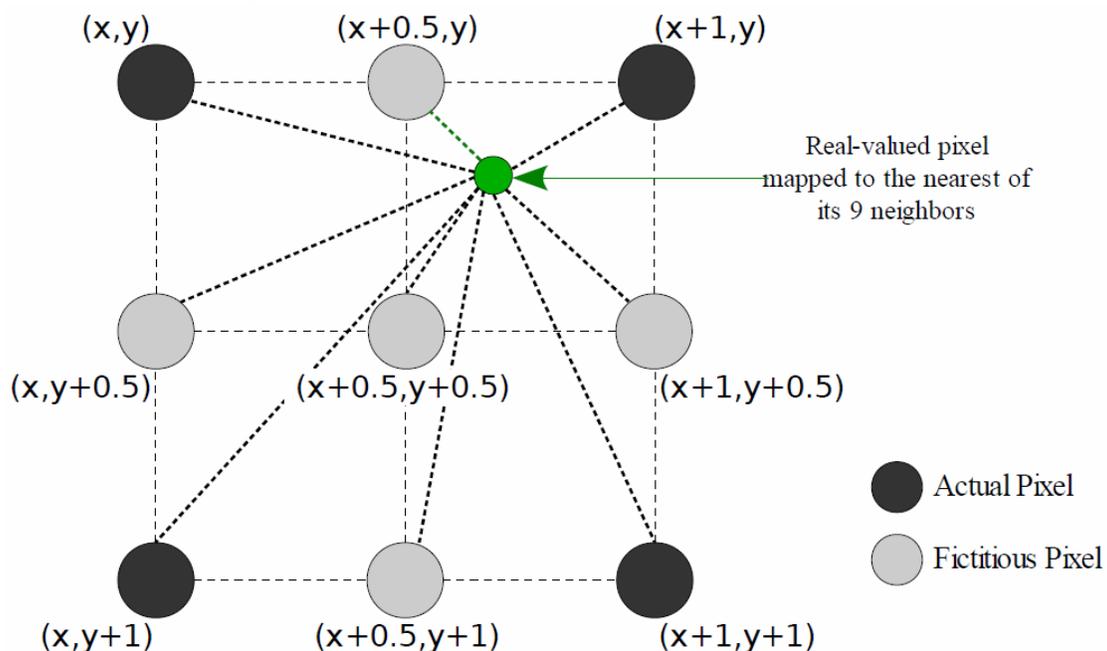


9-Point Interpolation

As detailed in the previous section, the LUT can provide a source pixel location for every target pixel location. However, since the source pixel location can be a real number, using it to compute the actual pixel values of the target image requires some form of pixel interpolation. In addition, an LUT with real values (needing floating-point representations) will become unwieldy for an FPGA. However, the 9-point interpolation scheme addresses these issues by providing a simple and efficient method for pixel interpolation that also avoids the storage of real numbers in the LUT.

Alternate methods for pixel interpolation are less successful. The “nearest neighbor” interpolation is a simple and fast method for computation, but is also somewhat coarse and can lead to visible image artifacts. More complex interpolation techniques such as bilinear interpolation involve floating-point operations that the FPGA is not suited to handle efficiently. The 9-point interpolation scheme—a middle path between the nearest neighbor and bilinear interpolation schemes—involves mapping the LUT’s real-valued pixel to the nearest of its nine neighbors, as shown in Figure 4.

Figure 4. The 9-Point Interpolation Scheme



In this method, the intensity values of all fictitious pixels are calculated by taking the average of the intensities of its adjacent actual pixels. For example, the color intensity of the pixel $(x, y+0.5)$ is the average of the color intensities of the actual pixels (x, y) and $(x, y+1)$. The intensity value of $(x+0.5, y+0.5)$ is the average of the intensities of the actual pixels (x, y) , $(x+1, y)$, $(x, y+1)$ and $(x+1, y+1)$.

The main advantage this technique possesses over the two formerly discussed methods is simplified computation without significant sacrifice in the quality of the corrected image. This is because the only computation involved is taking the averages of either two or four quantities. Division by two can be realized by “right-shifting” the sum of the numbers by one bit, while division by four can be done by right-shifting the sum by two bits. This sort of computation is very simple for an FPGA.

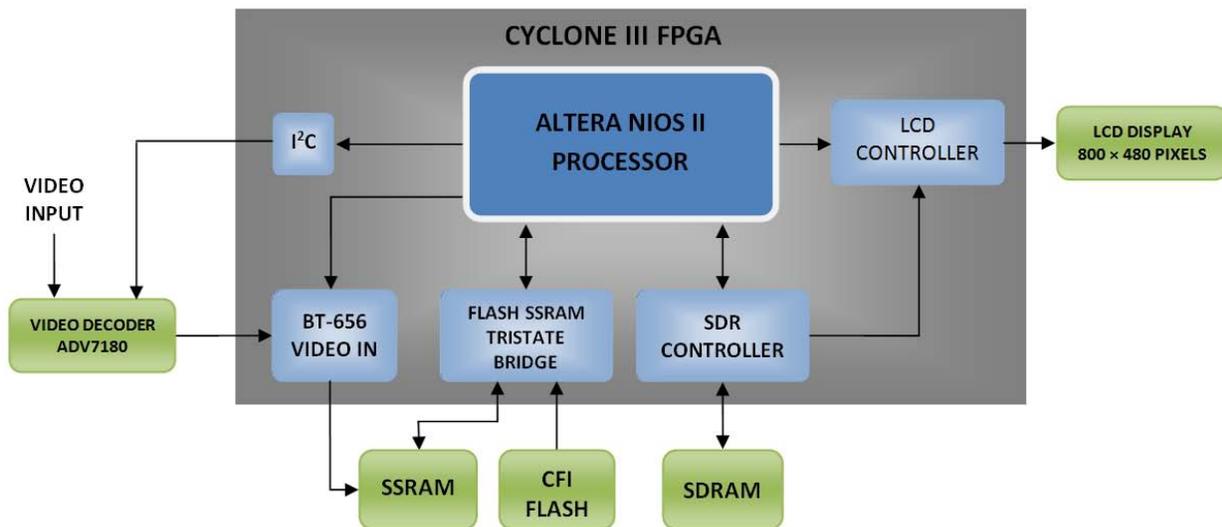
Note that with the 9-point interpolation scheme, there is no need for the LUT to store any real-valued pixel locations. It can directly store the location of either the actual or the fictitious pixel to which the real-valued pixel is mapped. This can be easily achieved with fixed-point representations.

Design Implementation

This section discusses the implementation of the fisheye correction using devices from the Altera® Cyclone® FPGA series and the Nios® II soft-core embedded processors. The Nios II architecture is a RISC soft-core architecture, which is implemented entirely in the programmable logic and memory blocks of Altera FPGAs, and is capable of handling a wide range of embedded computing applications, from DSP to system control. The soft-core nature of the Nios II processor lets the system designer specify and generate a custom Nios II core, tailored for his specific application requirements. Altera's Nios II Embedded Evaluation Kit (NEEK) is used as the development platform.

As shown in Figure 5, the hardware architecture is based on the following: Nios II soft-core embedded processor, a BT-656 video input module, an I²C configuration module, a DDR-SDRAM controller, and a LCD controller.

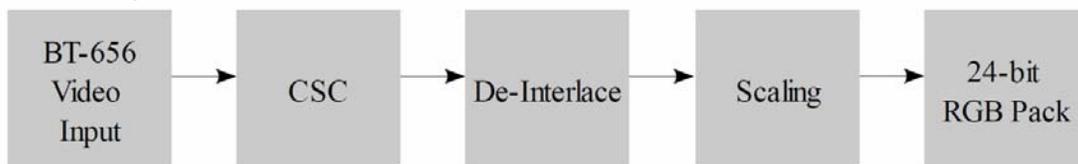
Figure 5. Internal Architecture



BT-656 Video Input Module

The BT-656 video input module, which is designed to be compatible with the ITU-R BT.656 digital video standard, is responsible for color space conversion (CSC), clipping, de-interlacing, scaling, and a 24-bit RGB pack. Each operation is performed sequentially (as shown in Figure 6) and is parameterized by registers controlled by the Nios II processor. Optionally, the video inputs may be clipped and scaled (up or down) depending on the desired output format. The registers allow the system to be customized for various display resolutions and input video formats such as NTSC, PAL, or SECAM. Video data from the module is transferred via a direct memory access (DMA) channel to an external SDRAM frame buffer.

Figure 6. Video Input Module



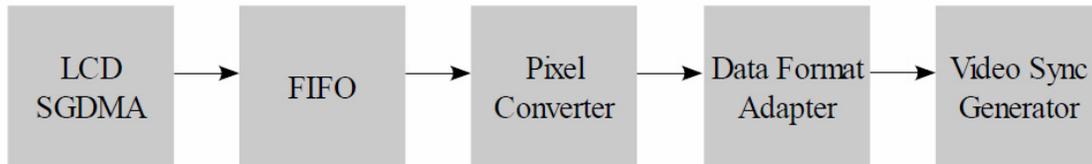
Video Input Configuration Module

The I²C configuration module is used to configure the video decoder, Analog Devices IC ADV7180, in the desired format (4), (5). The ADV7180 provides its output in an 8-bit ITU-R BT.656 YCrCb 4:2:2 digital video standard.

LCD Controller

The LCD controller uses the scatter-gather algorithm to achieve faster data transfer without processor overhead becoming a limiting factor. As shown in Figure 7, the LCD SGDMA takes in data from the external SDRAM frame buffer and sends it to the FIFO that is used to implement a dual port RAM for adjusting data rates suitable for the LCD peripheral. The pixel converter and data format adapter modify the data according to LCD display system specifications. The video sync generator takes in the formatted data, generates the appropriate clock signals, and sends the data for display.

Figure 7. LCD Controller



Memory Requirement and Management

There is continuous data input from the camera and this requires quick real-time processing for optimum throughput. Effective memory management becomes critical for storing and retrieving image and additional processing data required for fisheye correction. The architecture uses the available memory resources on the NEEK for the input frame buffer, output frame buffer, and for storing and reading the static LUT. All these memory functions are performed using control signals sent from the CPU to the SDR controller. The SDR controller connects to the SDRAM chip and handles all SDRAM protocol requirements.

Input frame buffer

This is implemented on the fast SS RAM. The input from the video decoder IC comes in a 16-bit interlaced RGB565 format, which requires 2 bytes per pixel. The resolution of the input camera used is 656×492 pixels. Hence, the amount of memory to be reserved for a single input frame is:

$$\text{width of input image} \times \text{height of input image} \times \text{bytes per pixel} = 656 \times 492 \times 2 \text{ bytes} = 630.375 \text{ Kbytes}$$

Output frame buffer

Implemented on the SDRAM, the buffer size depends on the resolution of the display. The NEEK display screen has a display resolution of 800×480 pixels, and the output format is also RGB565, so the amount of memory to be reserved for an output frame is:

$$\text{width of output image} \times \text{height of output image} \times \text{bytes per pixel} = 800 \times 480 \times 2 \text{ bytes} = 750 \text{ Kbytes}$$

Storing and reading the static LUT

The LUT must be generated externally from the known camera and display parameters each time an image is corrected after power up. Hence, it must be present on the NEEK. Since the RAM cannot be used for permanent storage of the static LUT, the non-volatile CFI flash memory must be used. Since the flash memory has a high access time, the LUT is copied onto the RAM block in order to achieve the desired computational speed.

The LUT contains the pixel coordinates (obtained using 9-point interpolation) of the source image for every pixel of the target image. Accordingly, it can contain either integer values or decimal values up to 0.5 denominations, depending on whether the mapped source pixel is an actual or a fictitious pixel, respectively (as discussed in “9-Point Interpolation”). In other words, the LUT values either can be x or $(x+0.5)$ (where x is an integer), or y or $(y+0.5)$ (where y is an integer). However, two requirements must be met while storing the LUT:

- Avoid storing and computing with floating point values while representing the LUT
- Since the input image is obtained in an interlaced format consisting of even and odd fields, to maximize speed, process the image as is, i.e., without having to de-interlace the input.

To meet the first requirement, note that the integral part of the x and y coordinates of the 9-point interpolated pixel will always correspond to an actual source image pixel. The second requirement is met by computing a proper offset while generating the LUT and during run time, so that any mapped (actual) source pixel can be located directly on the interlaced input image. Therefore, each LUT entry can be represented using a 32-bit data type as follows:

- The first 29 most significant bits (MSBs) encode the integral parts of the x and y coordinates of the 9-point interpolated pixel on the interlaced source image.
- The third least significant bit (LSB) denotes whether the pixel position lies in the even field (bit value “0”) or the odd field (bit value “1”).
- The second LSB denotes whether the x coordinate is an integer value (bit value “0”) or floating value (bit value “1”).
- The LSB denotes whether the y coordinate is integer value (bit value “0”) or floating value (bit value “1”).

This scheme allows us to store the LUT much more efficiently than if they were stored as floating-point values in the CFI flash memory. The size of the LUT depends on the output frame size. Since the LUT needs 4 bytes (32 bits) of storage for each output pixel, so the amount of memory to be reserved for storing the LUT is:

$$\text{width of output image} \times \text{height of output image} \times \text{bytes per LUT data} = 800 \times 480 \times 4 \text{ bytes} = 1500 \text{ Kbytes}$$

Generating the Output Frame

The LUT is indexed by the pixels of the output frame and its values are the coordinates of the pixels of the input frame the intensity of which must be assigned to or used to compute the corresponding pixel intensities of the output frame. It can be determined whether the LUT entry corresponds to actual or fictitious pixels by checking its LSB and second LSB. A high bit value represents an actual pixel and a low bit value represents a fictitious pixel. Then the RGB565 intensities are assigned to the pixels of the output frame using 9-point interpolation, which involves either direct assignment (in the case of an actual pixel) or appropriate averaging (in the case of a fictitious pixel) to determine the output pixel intensities. The following pseudo code illustrates this method.

```

DETERMINE memory size of LUT by width of display × height of display × 4 bytes
  ASSIGN memory for static LUT on SDRAM
  OPEN flash memory device
  COPY the static LUT from flash memory to SDRAM
  CLOSE the flash memory device
COMPUTING THE OUPUT FRAME
  FOR each row of the output image
    FOR each column of the output image
      OBTAIN the value of pixel position (right-shifting the LUT by 3 bits) from the static
      LUT
      IF pixel location is INVALID THEN
        ASSIGN output pixel as BLACK
      ELSE
        IF pixel location represents REAL pixel THEN
          ASSIGN the output pixel with the corresponding input pixel intensity value
        ELSE IF pixel location represents FICTITIOUS pixel THEN
          COMPUTE the average intensity of the nearest actual input pixels depending on
          whether field is even or odd
          ASSIGN the value to the output pixel
        ENDIF
      ENDIF
    ENDFOR
  ENDFOR
  ENDFOR
DISPLAYING THE FRAME
  ASSIGN the screen buffer with the computed output frame

```

Conclusions

Using FPGAs and soft-core embedded processor technology, Altera and MDN have developed a novel architecture for fisheye correction in wide-angle cameras. This architecture is flexible, scalable, and makes efficient use of the FPGA's resources. Because the architecture's Nios II processor is versatile and powerful enough to take on additional embedded processor functions, this technology is ideally suited for use in applications where wide-angle cameras are used, such as automotive rear-view cameras and others.

References

1. "Fisheye lens," Wikipedia:
http://en.wikipedia.org/wiki/Fisheye_lens
2. *Transformations and Projections in Computer Graphics*, David Salomon, Springer, 2006.
3. *Nios II Processor Reference Handbook*:
www.altera.com/literature/lit-nio2.jsp
4. *Nios II Embedded Evaluation Kit, Cyclone® III Edition, User Guide*:
www.altera.com/literature/ug/niosii_eval_user_guide.pdf
5. *Nios II Flash Programmer User Guide*:
www.altera.com/literature/ug/ug_nios2_flash_programmer.pdf

Further Information

- *Implementing a Flexible CPLD-Only Digital Dashboard for Automobiles*:
www.altera.com/literature/wp/wp-01072-implementing-flexible-cpld-only-digital-dashboard-automobiles.pdf
- *Creating Low-Cost Intelligent Display Modules With an FPGA and Embedded Processor*:
www.altera.com/literature/wp/wp-01074-creating-low-cost-intelligent-display-modules-with-fpga.pdf
- *Applying Graphics to FPGA-Based Solutions*:
www.altera.com/literature/wp/wp-01075-applying-graphics-to-fpga-based-solutions.pdf
- *Using LEDs as Light-Level Sensors and Emitters*:
www.altera.com/literature/wp/wp-01076-leds-as-light-level-sensors-and-emitters.pdf



MANIPAL DOT NET
www.manipal.net



101 Innovation Drive
San Jose, CA 95134
www.altera.com

Copyright © 2008 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.