Invited Paper

Multithreaded real-time 3D image processing software architecture and implementation

Vikas Ramachandra, Kalin Atanassov, Milivoje Aleksic and Sergio R. Goma
Qualcomm Incorporated.

ABSTRACT

Recently, 3D displays and videos have generated a lot of interest in the consumer electronics industry. To make 3D capture and playback popular and practical, a user friendly playback interface is desirable. Towards this end, we built a real time software 3D video player. The 3D video player displays user captured 3D videos, provides for various 3D specific image processing functions and ensures a pleasant viewing experience. Moreover, the player enables user interactivity by providing digital zoom and pan functionalities. This real time 3D player was implemented on the GPU using CUDA and OpenGL. The player provides user interactive 3D video playback. Stereo images are first read by the player from a fast drive and rectified. Further processing of the images determines the optimal convergence point in the 3D scene to reduce eye strain. The rationale for this convergence point selection takes into account scene depth and display geometry. The first step in this processing chain is identifying keypoints by detecting vertical edges within the left image. Regions surrounding reliable keypoints are then located on the right image through the use of block matching. The difference in the positions between the corresponding regions in the left and right images are then used to calculate disparity. The extrema of the disparity histogram gives the scene disparity range. The left and right images are shifted based upon the calculated range, in order to place the desired region of the 3D scene at convergence. All the above computations are performed on one CPU thread which calls CUDA functions. Image upsampling and shifting is performed in response to user zoom and pan. The player also consists of a CPU display thread, which uses OpenGL rendering (quad buffers). This also gathers user input for digital zoom and pan and sends them to the processing thread.

Keywords: 3D video, GPU, CUDA, rectification, stereo convergence, 3D zooming, OpenGL

1. INTRODUCTION

With the rising popularity of consumer 3D displays, it is expected that user generated 3D images and videos will become a reality soon. Typically, 3D capture and display have happened in a very controlled environment, e.g., in 3D movies or 3D graphics, where perfect control of the scene is possible and thus tight control on the 3D effect can be exercised. Moreover, the display of 3D images happens traditionally on known screen geometry and is static, i.e., it does not allow user interactions. In contrast, the consumer model for 3D usage allows for the display to occur in a more ad-hoc environment, where the scene geometry is not known beforehand nor are there means to influence it. Also, to be commercially viable, 3D technology needs to be interactive and portable, i.e., display-aware. Since the scene and display geometry constraints are unknown a priori, this presents an interesting problem as it requires adding scene and display control at the algorithmic level (post processing). To enable consumers to view their 3D content on their displays, there is a need for a 3D software player which can not only display 3D videos but also perform the various steps of 3D image processing involved in real time. Such a player should be able to accommodate for the 3D scene depth characteristics and various display geometries to ensure a pleasant viewing experience for the user. The player must also be capable of processing user feedback such as zoom and pan. In this paper, we aim to provide such a user friendly and interactive 3D player. Towards this end, a real time 3D player was implemented on the GPU using CUDA[1] and OpenGL. The objective of the player is to provide user interactive 3D video playback.

Qualcomm Inc., 5775 Morehouse Drive, San Diego CA 92121.

Real-Time Image and Video Processing 2011, edited by Nasser Kehtarnavaz, Matthias F. Carlsohn,
Proc. of SPIE-IS&T Electronic Imaging, SPIE Vol. 7871, 78710A · © 2011 SPIE-IS&T
CCC code: 0277-786X/11/$18 · doi: 10.1117/12.872594

2. PLAYER ARCHITECTURE

In this section, the architecture of the 3D player is explained in more detail. The 3D player consists of the following three CPU threads:

- Read thread:
  - This consists of a read function to read a set of stereo images (previously captured using a stereo camera) continuously from a fast drive or a RAM disk. The image frame read speed is synchronized to the display refresh rate (120 Hz in this case) using a programmable sleep function. Once every new stereo frame is read in, this thread releases a semaphore to the processing thread.
- Processing thread: This involves the following steps performed using the GPU, on the images read into memory: Image rectification, convergence point selection and image shifting. The complete 3D processing algorithm is explained in the following section.
- Display thread: This thread consists of the following functions:
  - Display: Uses OpenGL rendering (quad buffers) in conjunction with the NVIDIA Quadro graphics card to drive active shutter glasses based stereo and/or stereo with polarized glasses.
  - User input capture: Keyboard callback functions are used to capture user input in terms of digital zoom and pan. These inputs are sent to the processing thread to recompute the disparity statistics since zoom and pan change the depth characteristics of the 3D scene.

Various threads of the 3D player and their interaction is illustrated in Fig. 1.

3. THE 3D PROCESSING ALGORITHM

Some of the important factors which need to be considered for the 3D algorithm design are that:

- Relative positions of the two cameras need to be selected carefully to ensure minimal scene distortion and eye strain,
- Mechanical tolerances related to the camera placement need to be compensated via post processing since it is not practical to expect that cameras will be perfectly aligned,
- Scene dependent convergence control is needed to reduce eye strain considering the fact that there is a "safe " 3D effect area around the screen and the 3D scene geometry may cause the 3D effect to fall out of that area, thus requiring processing.[2]
- Adding zoom/pan functionality is critical to 3D adoption since the end user is already used to these operations when capturing/reviewing 2D content.
- Screen geometry needs to be taken into consideration during processing,
- Every time the image conditions (zoom, pan) or display conditions change, the 3D processing algorithm needs to be rerun.

Based on the above considerations, the 3D processing algorithm, which is a part of the processing thread, involves the following steps:

- Rectification: Rectifies images based on the calibration matrix parameters. The 3x4 calibration matrix used is a standard projective transformation whose parameters are found a priori using a stereo calibration procedure.
- Optimal convergence selection: This step is very important for reducing eye strain. The rationale for this convergence point selection takes into account the scene depth characteristics as well as the display geometry. This involves:

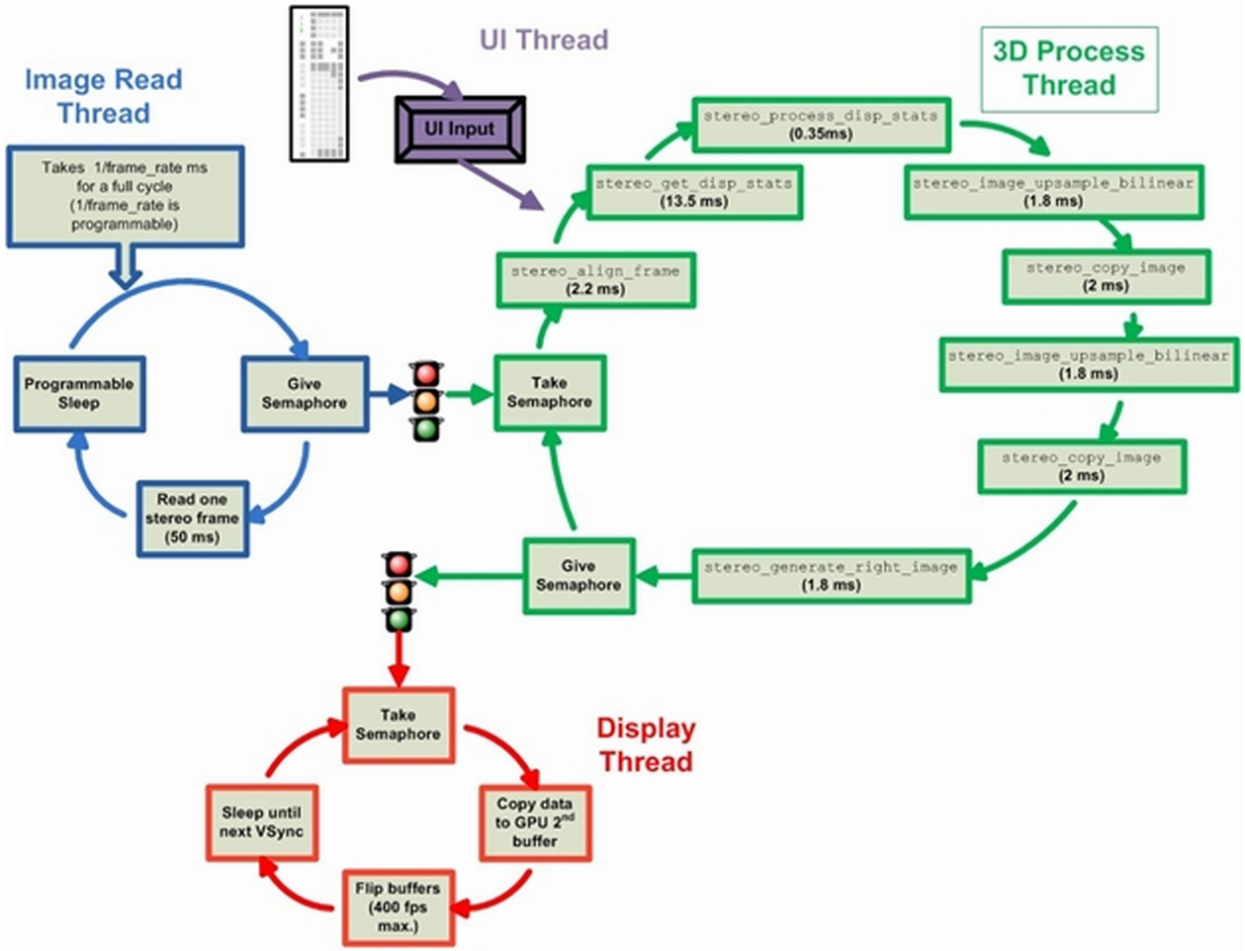


Figure 1. The 3D player thread outline: The player consists of three main threads for image read, processing and display. The individual functions and their execution times are listed.

- Keypoint detection: Vertical edge locations are detected and are labeled as keypoint locations.
- Block matching: For selected keypoints, block matching is performed between local image patches in the left and right camera images. A list of best disparities is computed, one for each keypoint. For speed, local image patches are loaded into the GPU's shared memory so that multiple GPU threads (corresponding to neighboring pixels) can use the same data.
- Disparity histogram processing: From the above disparity list, a histogram is constructed. This histogram is pruned to remove outliers. Extrema of this histogram give the disparity range of the scene.
- Image shifting: The left and right images are shifted accordingly to place regions of the scene at the mean value of the calculated disparity range at convergence. Image upsampling and shifting: This feature caters to the digital zoom and pan inputs from the user.
- Zoom and pan processing (based on user inputs).

All the above functions, (except for the histogram processing, which uses the CPU) are implemented on the GPU. The above mentioned processing steps are explained in more detail below. A functional block diagram of the 3D processing algorithm (in the processing thread) is presented in Fig. 7.

3.1 Functions in the 3D processing algorithm (processing thread)

3.1.1 Rectification on GPU

The camera calibration matrices are used to align the left and right camera images so as to make them parallel.[34] In our case, we perform a projective transform on the left image only. Let $M_l = [R_l, T_l]$ and $M_r = [R_r, T_r]$ be the 3x4 calibration matrices (obtained a priori and stored) for the left and right cameras. Here, the $R_{l,r}$s are the 3x3 rotation matrices and $T_{l,r} = [t_x, t_y, t_z]$s are the 3x1 translation vectors, which together relate the camera coordinates to the real world coordinates. In the mobile camera setup used for this paper, the translations along the axes (other than the desired camera separation) were negligible. This was made possible by the mechanical construction of the stereo rig. Thus, the left image is compensated only for the relative 3D rotation, and this transformation maps a pixel at location $[x_{il}, y_{il}]$ to $[x_{il,warped}, y_{il,warped}]$, and is given by:
$[x_{il,warped}, y_{il,warped}]' = R_r * (R_l)^{-1} * [x_{il}, y_{il}]'$
This rotates the left image to make it parallel to the right image. With negligible relative shifts $[t_{lx} - t_{rx}, t_{lz} - t_{rz}] \approx 0$, the left and right camera images should now differ only by horizontal disparities.

The above warping is implemented on the GPU as follows. For each pixel location, the warped pixel location is precomputed and stored. For every input video frame, the recomputed warp locations are looked up and the pixels are shifted to their new coordinates. One GPU thread is used for each pixel during the warping.

3.1.2 Keypoint detection on GPU

Identification of keypoints is typically done with high pass operations at different resolutions. In this case, vertical variations are not considered since only horizontal shift is present between the left and right images after the rectification step. Horizontal variations (edges with some vertical component) are used to find the keypoints in the image. Only one resolution is used here. The algorithm is as follows.

- Subsampling of the left image luma component by 6 - the purpose here is to improve algorithm robustness to noise and decrease demand for processing power. This downsampling is performed on the GPU by local averaging using a thread for each 6x6 block of the high resolution image.
- Horizontal high pass filtering (on the GPU). The filter, which has a 3x4 kernel, has a response given by:
$$\begin{pmatrix} -1 & -1 & 1 & 1 \\ -1 & -1 & 1 & 1 \\ -1 & -1 & 1 & 1 \end{pmatrix}$$
- The response to this filter is calculated at each pixel location. In a given horizontal neighborhood, the local maximum is selected.

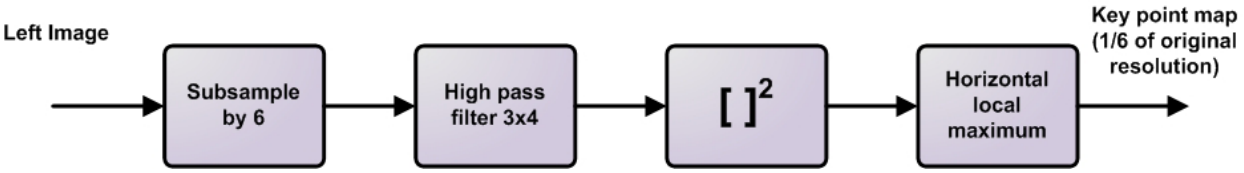


Figure 2. Block diagram for the key point detection algorithm.

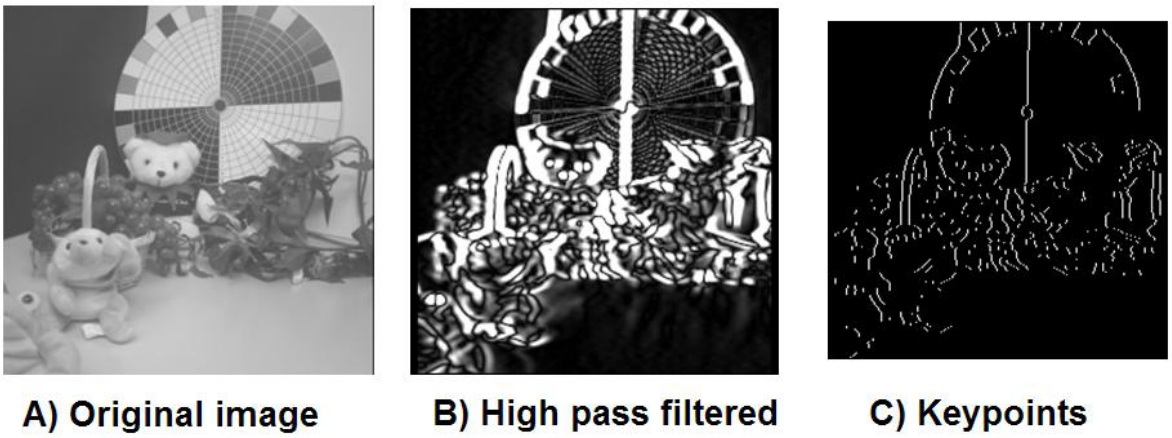


Figure 3. An example of key point detection on a real image.

The keypoint detection steps are illustrated in Fig. 2. For an example image, the detected keypoints are shown in Fig. 3.

3.1.3 Keypoint matching on GPU

Once the keypoints have been detected in the left image, the best match for each keypoint is found on the right image as follows. This gives us the best disparity vector for each keypoint. For every keypoint,

- Select a neighborhood around the keypoint from the left luma image.
- Compute the best correlation coefficient with the right image neighborhood (within a search range). This involves choosing the best image patch (in terms of maximum cross correlation) from the candidate image patches in the right image neighborhood. Also store the corresponding disparity vector.
- If the correlation coefficient is above a certain threshold, keep the corresponding disparity vector. Otherwise, flag the disparity vector as invalid. This would correspond to regions with occlusions and depth discontinuities.

Normalized cross correlation is used to account for different average luminance values in the left and right images. Keypoint matching is the most computationally intensive step in the processing chain, hence this needs optimization on the GPU. The GPU implementation for this function involves using shared memory between threads (each thread computes a candidate correlation value and a candidate disparity vector for a given location). This reduces the latency due to global memory fetch calls[5].[6] For the calculation of the best disparity vector at a given pixel location (for a block size of $N * N$ and search range of k), $N * N$ pixels are loaded from the left image and $N * (N + k)$ pixels are loaded from the right image into shared memory. A grid size of Image_Width/N and Image_Height/N is used. Each grid has a cluster of k threads. Among the k threads for a block, the same shared memory chunk is reused. This is illustrated in Fig. 4. Also, Fig. 5 provides an example image with the disparity vectors drawn on it (Blue and green indicate opposite directions, and the lengths indicate the disparity values). Note that the above procedure finds disparity vectors at all pixel locations in the image. Since we need disparities at only the keypoint locations, only those values are actually retained for further use.

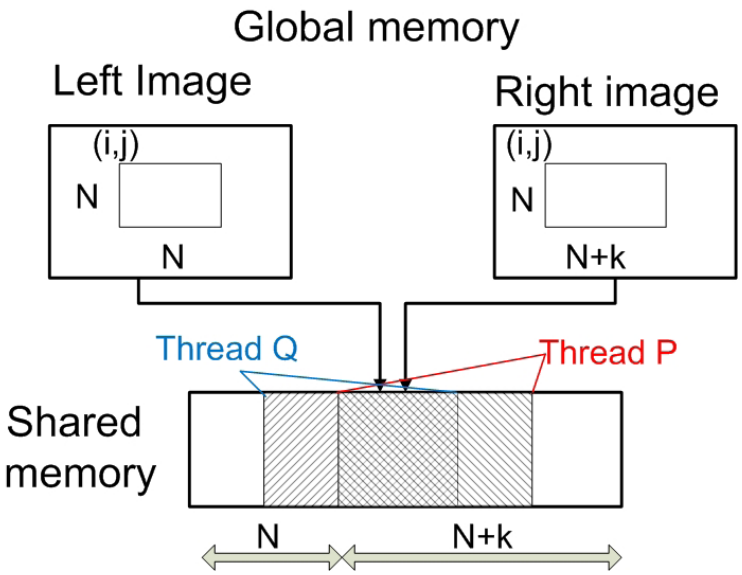


Figure 4. GPU keypoint matching using shared memory. For the same pixel locations, k different threads compute the cross correlation values for k possible disparity vectors. Those k threads need access to different parts of a common shared memory. The disparity vector corresponding to the best correlation is finally selected.



Figure 5. Example image with disparity vectors drawn on it. Blue and green indicate opposite directions, and the lengths indicate the disparity values. Note that the convergence point was estimated near the train tracks, since for the given display, about 1/4 of the image was desired to be 'popping out' of the screen.

3.1.4 Histogram processing and convergence control logic on CPU

Since this operation is not memory or processor intensive for our application, this is performed on the CPU. The 3D control logic is a function with very small processing overhead that performs the following operations:

- Build a histogram of disparity vectors - this is the sufficient statistic for determining the 3D control path.
- Trim the histogram's tails - this is done to ensure that any outliers that can likely be mismatches are removed from the histogram and will not influence final statistics.
- Estimate scene range as difference between min and max values of the trimmed histogram.
- * Compute acceptable disparity range using screen geometry parameters.
- ** Set the convergence points between min and max based on the screen geometry parameters.
- The maximum remaining disparity determines if the scene is fusible or not.
- If a scene is fusible, signal the post processing module to adjust the right image only by performing a horizontal shift corresponding to placing the desired point at convergence.
- If the scene is not fusible, there are several options (based on what is available at the post processing module).
  - Switch to 2D stream (make left and right frames the same)- currently being used in the module.
  - 2D to 3D conversion.
  - Invasive disparity control.

* Computation of the acceptable disparity range using screen geometry parameters is done as follows:

- Let the allowable disparity be $\alpha = 0.5deg - 2deg$ (specified during configuration time).
- Let θ be the display viewing angle.
- Let D be the scene depth , i.e., the difference between maximum ($D2$) and minimum ($D1$) disparity vectors in pixels.
- Let W be image width in pixels.
- Let x be distance from screen.
- Angular disparity (deg) that will be seen on the screen is: $(D/W) * \theta$.
- Test if disparity is fusible, i.e. $\alpha > (D/W) * \theta$.

** The convergence point between min and max based on the screen geometry parameters is set as follows:

$$C = D1 + D * M(x) \tag{1}$$

where D1 is the minimum disparity in the trimmed histogram and M is monotonically increasing function between 0 and 1. This rationale is illustrated in Fig. 6. The idea behind this is that for screens which are close to the viewer, a smaller depth range of the scene needs to 'pop out' for comfortable viewing and vice versa.

3.1.5 Image shifting on GPU

Once the optimal shift has been determined using the above control logic, the right image is shifted on the GPU horizontally by the desired amount. This is similar to the image warping/ rectification function on he GPU mentioned above.

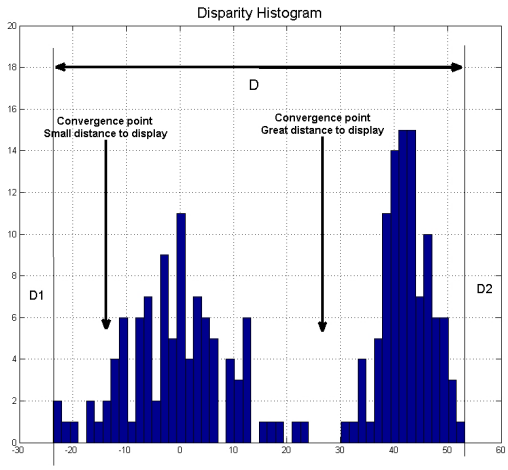


Figure 6. Convergence control: Set the disparity range D based on the viewing angle, and the convergence point based on the distance to the screen.

3.1.6 Zoom and pan processing on GPU

The player provides for user interactivity during 3D video playback. The user can pick regions to zoom into or pan digitally when the video is playing or paused. These user inputs captured in the display thread are passed to the processing thread. The processing algorithm handles zoom/ pan by considering disparity vectors of only those keypoints within the zoom/ pan window (instead of the full input image) to compute the histogram and optimal shift. Also, the fusible range is adjusted continuously as the user zooms into the video, since the disparities become larger in that case. The rest of the processing chain remains unchanged.

4. A NOTE ON THE CODE PERFORMANCE

Overall, the processing for each frame takes about 17ms on average, thus providing a throughput of about 60 fps stereo. The GPU implementation was able to achieve 12x speed up compared to the CPU only version of the program. The most computationally intensive function was key point matching, and a 20x speed up was achieved in this function by moving from a naive implementation to the function which uses shared memory between different GPU threads.

REFERENCES

[1] [*NVIDIA CUDA*]. http://www.nvidia.com/object/cuda_home_new.html.

[2] Hoffman, D., Girshick, A. R., Akeley, K., and Banks, M., "Vergence accommodation conflicts hinder visual performance and cause visual fatigue," in [*Journal of Vision*], (2008).

[3] Trucco, E. and Verri, A., "A compact algorithm for rectification of stereo pairs," in [*Machine Vision and Applications*], (2000).

[4] Woods, A., Docherty, T., and Koch, R., "Image distortions in stereoscopic video systems," in [*Proceedings of the SPIE Volume 1915, Stereoscopic Displays and Applications IV.*], (1993).

[5] Chen, W. and Hang, H., "H.264/avc motion estimation implmentation on compute unified device architecture (cuda)," in [*IEEE International Conference on Multimedia and Expo*], (2008).

[6] [*CUDA stereo*]. http://www.cs.unc.edu/ gallup/cuda-stereo/.

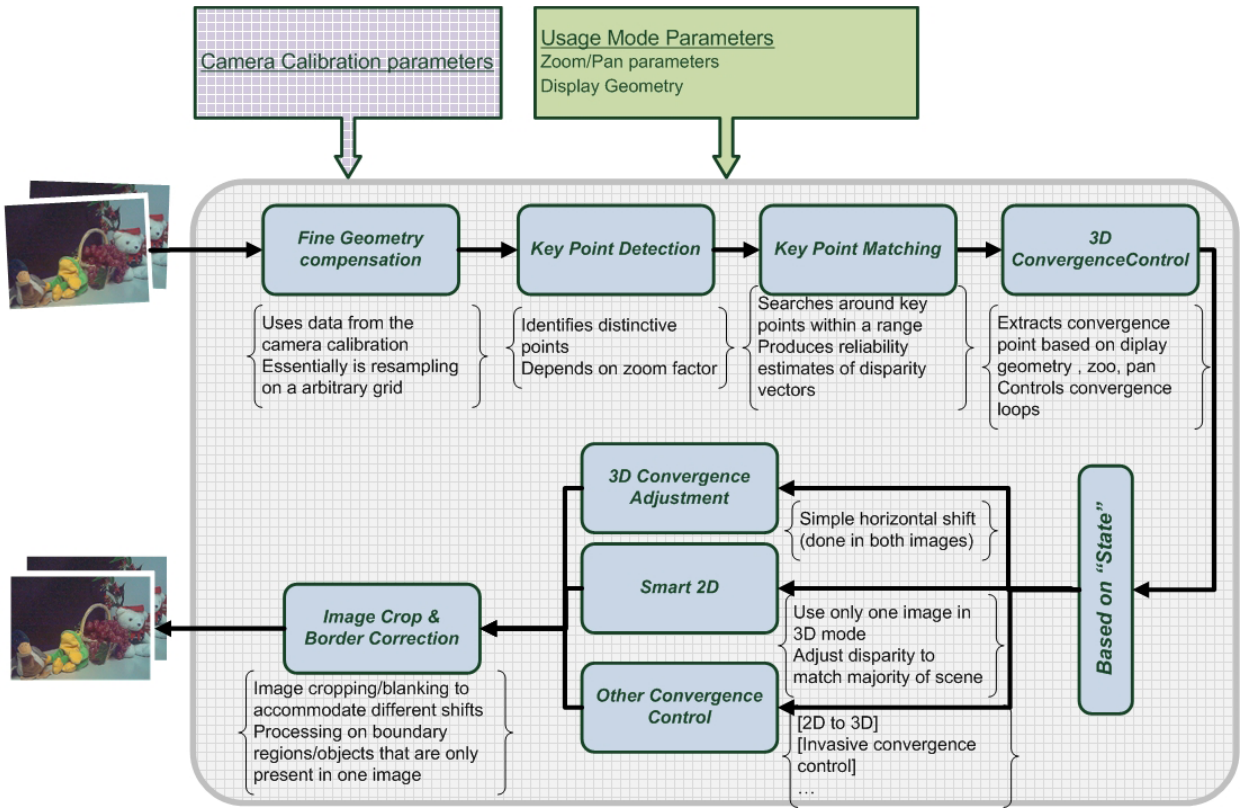


Figure 7. Functional block diagram of the 3D processing algorithm (in the processing thread).