Lessons Learned Using a Camera Cluster to Detect and Locate Objects

Daniel Stødle, Phuong Hoai Ha, John Markus Bjørndalen,
Otto J. Anshus

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr,
F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 71-78, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

Lessons Learned Using a Camera Cluster to Detect and Locate Objects

Daniel Stødle, Phuong Hoai Ha, John Markus Bjørndalen, and Otto J. Anshus

Dept. of Computer Science
Faculty of Science
N-9037 University of Tromsø, Norway
E-mail: {*daniels, phuong, jmb, otto*}*@cs.uit.no*

A typical commodity camera rarely supports selecting a region of interest to reduce bandwidth, and depending on the extent of image processing, a single CPU may not be sufficient to process data from the camera. Further, such cameras often lack support for synchronized inter-camera image capture, making it difficult to relate images from different cameras. This paper presents a scalable, dedicated parallel camera system for detecting objects in front of a wall-sized, high-resolution, tiled display. The system determines the positions of detected objects, and uses them to interact with applications. Since a single camera can saturate either the bus or CPU, depending on its characteristics and the image processing complexity, the system supports configuring the number of cameras per computer according to bandwidth and processing needs. To minimize image processing latency, the system focuses only on detecting where objects are, rather than what they are, thus reducing the problem's complexity. To overcome the lack of synchronized cameras, short periods of waiting are used. An experimental study using 16 cameras has shown that the system achieves acceptable latency for applications such as 3D games.

1 Introduction

This paper reports on lessons learned using a cluster of cameras to detect the position of objects in front of a wall-sized, high-resolution, tiled display. The system is used to support multi-user touch-free[a] interaction with applications running on a 220-inch 7x4 tiles, 7168x3072 pixels resolution display wall (Fig. 1). This requires that the system can accurately and with low latency determine the positions of fingers, hands, arms and other objects in front of the wall. To achieve this, a consistent and synchronized set of position data from each camera is needed.

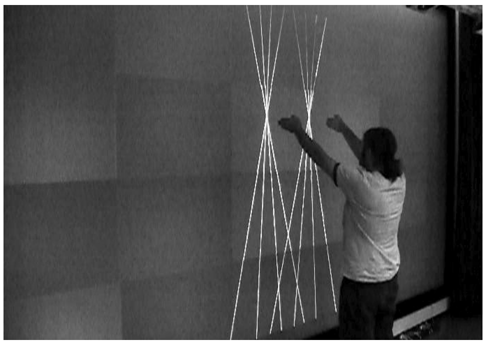
Figure 1. Using the system.

A grayscale camera producing images at a rate of 30 frames per second with a resolution of 640x480 pixels requires a bandwidth of about 8.78 megabytes/second. A FireWire 400 bus can accommodate at most three cameras producing data at this rate; higher-resolution or higher-framerate cameras further decrease this bound. To support more cameras, additional FireWire buses can be used on a single computer. Scalability may now

[a]As the display wall's canvas is not rigid, users must be able to interact with the display wall without actually touching it - thus the term "touch-free."

be limited by the CPU, either due to image processing complexity or deadlines on when results are needed. Finally, most commodity cameras have no support for hardware- or software-based inter-camera synchronization. This limits the accuracy of object positioning, as it reduces the system's ability to relate images captured from different cameras to each other.

This paper presents a parallel system for processing streaming video from several cameras. The system architecture comprises four layers: (i) Camera and image processing, (ii) object-position processing, (iii) event distribution, and (iv) end-application use of position data. The first layer uses 16 cameras connected pairwise to 8 computers. Each computer processes images from two cameras, locating objects and determining their one-dimensional position. When three or more cameras in the first layer see the same object, the second layer can determine the object's 2D position using triangulation. The third layer distributes position data between the other three layers. The fourth layer is comprised of applications using the position data for interaction. An experimental study has shown that the system achieves acceptable latency for common applications like the 3D games Quake 3 Arena and Homeworld (see Section 5 and Ref. 1).

The main contributions of this paper are the lessons learned from building and using the system, including: (i) The flexibility of the system architecture allows configuring available camera and processing resources to accommodate end-applications' needs, (ii) by reducing the complexity of image processing from identifying *what* objects are to identify *where* they are, processing is reduced, and (iii) despite the lack of synchronized cameras, useful results may still be obtained by introducing short periods of waiting.

2 Related Work

There exists much work on multi-camera systems. In Ref. 2, the authors demonstrate how a 100-camera array is used to capture very high-resolution video at 3800x2000 pixels at 30 FPS, or high-speed video with 640x480 pixels at 1560 FPS. Their implementation uses custom circuit boards to communicate with the FireWire cameras and relies on hardware synchronization of cameras, while the system presented in this paper is exclusively based on use of commodity components; cameras without support for synchronization and no use of custom hardware. In Ref. 3, the authors show how displays may be synchronized using an external synchronization source combined with software adjustment of display timings (software genlocking). Their use of a hardware synchronization signal precludes applying their technique to synchronize commodity camera capture.

Other work has used many low-resolution cameras to generate a 3D reconstruction of objects, either for collaborative applications[4] or for creating 3D models. Our system does not attempt to generate high-resolution video or imagery, or reconstruct 3D objects. Instead, the goal is to use a cluster of cameras to determine the 2D position of objects in a plane parallel to the display wall. Previous work reports on different ways of achieving this. In Ref. 5, the author combines internal reflection of infrared light with a camera mounted behind a (rigid) canvas to support multi-touch interaction. Our system differs in that it doesn't require users to actually touch the canvas in order to interact, and in the use of a parallel architecture for capturing and processing images. In Ref. 6, a set of cameras with on-board image processing is mounted in the corners of a large display, and used to detect multiple points of contact. Rather than build custom cameras, our system uses commodity cameras mounted on the floor in front of the approximately 6 meter wide display wall, and performs all processing on a compute cluster.

3 Design

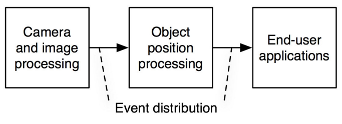

Figure 2. The system architecture.

The system architecture is comprised of four layers, as detailed in the introduction and shown in Fig. 2. The camera and image processing layer captures and processes images from cameras used by the system. To allow for many cameras to be used simultaneously as well as flexibility in image processing complexity, this layer is designed to run in parallel. The layer produces 1D positions and radii for detected objects in each image for each camera. An object's 1D position is defined as the centre of a detected object along the horizontal axis of a captured image (the centner of the finger in Fig. 4), and its radius defined as half the width (in pixels) of the detected object. The object position processing layer combines the position data from each computer in the image processing layer using triangulation, to determine the each object's 2D position.

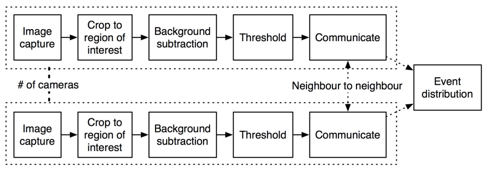

Figure 3. The camera and image processing layer design. The layer can operate in parallel with any number of cameras. Communication happens between each participant and its immediate neighbours.

The camera and image processing layer consists of several steps: (i) Image capture, (ii) cropping, (iii) background subtraction, (iv) thresholding and (v) communication, shown in Fig. 3. When the layer first starts up, it stores the current image from the camera it works with to a buffer. This image will be referred to as the background. As new images are captured, a horizontal region of interest (ROI) is isolated, before the pixel values in the ROI are subtracted from corresponding pixels in the background image. If the absolute difference between a pixel in the current and in the background image is beyond a given threshold, an object is detected at the position of the given pixel in the image. The ROI is determined dynamically when each camera starts capturing images, by identifying the two brightest, horizontal regions in the image[b].

Figure 4 shows an example of how a single image from a single camera is processed. The two horizontal lines (1) indicate two regions of interest in the image. A finger extends from the hand visible in the image, intersecting both ROIs. The background (2) is subtracted from the current image (3), resulting in (4), before the thresholding step is applied, yielding (5). Continuous regions of white indicate where objects have been found in the image.

To account for changes in lighting, the background is updated when too many objects are detected in a single frame from a given camera. An earlier implementation updated the background continuously by merging it with the current image. This did not work well, as users often point at the same location for longer periods of time (on the order of several

[b]The system makes use of a set of "Christmas lights" running along the ceiling, directly above the cameras, in order to create high contrast with intersecting objects.

seconds). The result was “ghost” objects appearing when the user eventually moved his hand.

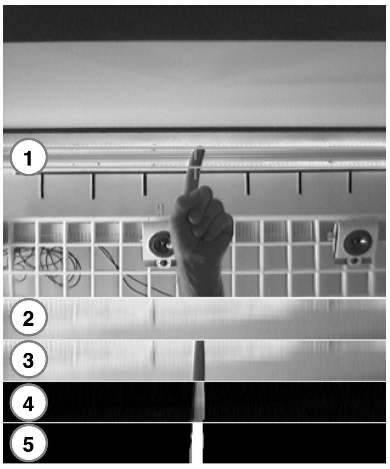

Figure 4. A sample image being processed by the image processing layer. The camera looks directly at the ceiling.

When all objects in the image have been found, the communication step begins. First, each participant sends the number of objects it has detected on the left- and right-hand side of the image to the neighbours on its left and right. The participant receives data from its neighbours, but to avoid introducing additional latency, the participant will use values that are up to 66 ms old[c]. The received values are used to determine if the participant’s results coincide with those of its neighbours. If the number of objects it has detected for the left or right side of the image is identical to the number of objects a neighbour has detected for the same side, no further processing is done. However, if the participant has detected fewer objects than its neighbour, it will re-perform the image processing sequence with a lowered threshold, in an attempt at discovering objects lost due to noise in the captured image. Similarly, if it detects more objects than a neighbour, the image processing sequence is re-performed with a raised threshold. Once this is done, the final 1D positions and radii are sent to the object position processing layer using the event distribution layer.

The object position layer receives 1D positions for located objects from the image processing layer, and uses the positions to triangulate their positions. In order to do this successfully, at least three 1D positions from three different cameras are required, as shown in Fig. 5; any less, and false positives occur when multiple objects are visible. The triangulation is performed by computing intersections between lines projecting from the cameras and up, at an angle determined by the 1D positions. An object’s position in 2D is successfully identified when two or more points of intersection from different cameras lie sufficiently close to each other. The final 2D position is computed as the average of the X and Y components of the 2D intersection points.

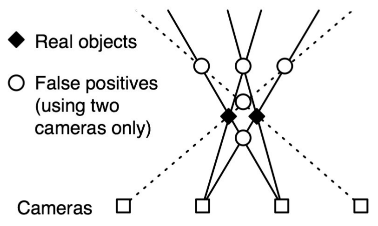

Figure 5. Line segments from each camera and passing through each object are generated. Each line segment is intersected with every other segment. At least three cameras are required to position an object, as using only two cameras results in many false positives.

4 Implementation

The system is comprised of 16 Unibrain Fire-i cameras, connected in pairs to a cluster of 8 Mac minis. In addition, a display cluster of 28 computers, each driving one projector at

[c]This is a tradeoff between system latency, and object detection accuracy. Data that is 66 ms = two frames old may still contain the correct number of (current) objects.

1024x768, is used to provide the graphics capabilities of the display wall, and a MacBook Pro is used to perform object position processing using data from the 8 Mac minis.

The cameras use IEEE-1394 (FireWire) to communicate with the Mac minis, and capture 640x480 grayscale (8-bit) images at 30 frames per second (FPS). They are mounted along the floor and spaced 32 cm apart, as shown in Fig. 6. The cameras do not support external or software-based triggers to synchronize the image capture of multiple cameras, not even when they are on the same FireWire bus. This means that two cameras may capture images spaced in time as far as 33 ms apart (the time between two frames at 30 FPS.).

The Mac minis are interconnected using Gigabit Ethernet. Each Mac mini captures and processes images from the two cameras it is connected to independently of the others, and runs a custom application for performing image capture and processing. This application is written in Objective-C, and uses libdc1394[7] to communicate with the cameras. Each camera is handled by a separate thread within the application, where each thread corresponds to one participant in the camera and image processing layer. Once a frame has been analyzed, the 1D positions and radii of any detected objects are sent to the the object position processing layer.

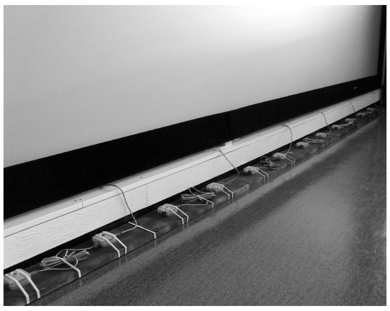
Figure 6. The image shows 12 of the 16 cameras mounted along the floor and looking at the ceiling.

The object position layer runs a loop operating at the same rate as the cameras, and uses the 1D object positions it receives to triangulate the positions of potential objects. To handle the lack of synchronized cameras, the object positioning software waits for up to 33 ms to receive (possibly empty) sets of object positions from all participating cameras. For the case when a camera has not detected an object, it will notify the object positioning layer of this for the first "no-detect" event only.

To triangulate the positions of objects, the cameras are placed in a coordinate system where cameras are spaced 1 unit apart (1 unit corresponds to 32 cm). For each camera, line segments starting at the camera's position and passing through the centre of each detected object are generated (Fig. 5). The lines are then intersected with all lines from the two cameras to the current camera's left and right. The resulting intersection points are compared, and points that are sufficiently close result in an object being identified. The identified objects' 2D positions and radii are then sent to end-user applications. It is each end-user application's responsibility to interpret the events to allow user interaction.

5 Evaluation

We have evaluated the system by measuring the latency incurred by the system's different layers. In particular, we measure the latency for the following components: (i) Camera capture, (ii) image processing, (iii) event distribution, and (iv) object position processing.

To measure camera capture latency, one camera was connected to a computer and pointed at the computer's display. A custom application fills the computer's display with black, and then starts capturing images from the camera. At one-second intervals, the display is filled with white, and a timer is started. When the difference between average pixel values from a 20x20 pixel square in the centre of the image in the previous and the current frame exceeds 150 (because the image goes from being black to being white), the timer is stopped, yielding the camera latency. The experiment was conducted on a Mac mini (1.66 GHz Intel Core Duo, 512 MB RAM) running Mac OS X 10.4.9 and a workstation (Intel Pentium 4 3.0 GHz, 2 GB RAM, HyperThreading enabled) running Ubuntu Linux 6.10 to investigate potential differences in latency caused by the operating system or hardware.

The image processing and object position processing latencies were measured by instrumenting the code that performs the two tasks and measure the execution time of 1000 iterations. The event layer's latency was measured using a ping-pong style benchmark, determining the round-trip time for one event sent back and forth. The resulting round-trip time was divided by 2 to find the one-way latency.

	Cam. capture	**Image proc.**	**Event distr.**	**Object pos.**	**Sum**
Samples	923 (852)	1000	1000	1000	-
Average	81 ms (93 ms)	1.16 ms	1.9 ms	31 ms	115 ms
Std. dev.	10 ms (9 ms)	0.11 ms	0.02 ms	10 ms	-
Minimum	58 ms (72 ms)	0.97 ms	1.6 ms	0.008 ms	62.7 ms
Maximum	104 ms (114 ms)	3.3 ms	3.8 ms	139 ms	250.1 ms

Table 1. Results from the latency experiments. Results for camera capture latency in parentheses are from running the experiment on the Linux workstation.

Table 1 shows the results from the experiments. The majority of total system latency of 115 ms is due to the cameras, with about 10 ms separating the measured latency on Mac OS X and Linux. The next biggest contributor to latency is object position processing (object pos.), which incurs an average latency of 31 ms, which is close to the rate at which the cameras deliver data (every 33 ms). Image processing in the system does not incur much latency. Event distribution (only counted once in the table, but generally incurred twice; once for sending events from the image processing layer to the object position layer, and then once more for sending events from the object position layer to end-user applications) incurs a negligible latency.

6 Discussion

The lack of synchronized cameras is the main limiting factor for accuracy in the system. As two cameras can capture images taken as much as 33 ms apart, the accuracy of the triangulation is significantly affected when the object is moving. The effect is further compounded because three cameras are required to accurately position an object. Filtering can reduce the impact of the uncertainty in position, but at the cost of higher latency. The object position processing layer already introduces up to 33 ms of latency to receive updated position data from all cameras. Although latency could be reduced by not waiting for all cameras, this has the effect of reducing the triangulation accuracy and the rate at which object positions are correctly triangulated drops.

Without synchronized cameras, the question of the system's accuracy can be raised. How fast can an object move while still being accurately positioned? Let p and r be the centre of an object O and its radius, respectively. Since the system uses only the horizontal axis to position objects, position and movement of an object are implicitly assumed to be horizontal[d].

We observe that a position x of the object detected by a camera can be considered accurate as long as x lies within $[p-r, p+r]$. Therefore, the position of a moving object can be detected accurately if there exists a common position $x*$ that satisfies the accuracy requirement for three images taken by three adjacent cameras during the interval $t = 33ms$[e]. Let $p' > p$ be the new horizontal position of the object's centre due to the object movement during the interval t. The common position $x*$ must satisfy $x* \leq p + r$ and $x* \geq p' - r$. Such a common position exists if $p' - r \leq p + r$ or $p' - p \leq 2r$. That means the system can detect an object's position accurately if the object does not move longer than $2r$ - its diameter - during the interval t.

For instance, assume that the object diameter is 1 cm (e.g. the size of the index-finger). In this case, the object's position can be accurately determined if the object moves at a speed less than $\frac{1cm}{33ms} = 0.3m/s$. Higher framerates can increase this bound, since the maximum delay between two cameras capturing an image will decrease. Doubling the framerate makes the maximum delay go down from 33 ms to 16 ms, and also reduce the object position processing latency. Other limiting factors are the number of cameras detecting the same object, the resolution of the cameras, the speed of the objects, the camera shutter speed, and the accuracy of the image processing layer.

The total system latency of 115 ms is sufficiently low to support playing two games (Quake 3 Arena and Homeworld), as we show in Ref. 1. In that paper, the camera latency was measured to be 102 ms, 21 ms more than reported in this paper. We speculate that the difference is due to a newer OS release in between the first set of results and the results presented in this paper[f]. The results from the Linux workstation show that the operating system or hardware architecture has an impact on the latency from the time at which a camera captures an image, until that image can be processed.

7 Conclusion and lessons learned

We have presented a scalable, dedicated parallel system using a camera cluster to detect and locate objects in front of a display wall. The bottlenecks in such a system can range from the bandwidth required by multiple cameras attached to a single bus and CPU requirements to process images, to deadlines on when results from image processing must be available. Due to our system's parallel architecture, the system can scale both in terms of processing and number of cameras. We currently use two cameras per computer, but with either more cameras, higher-resolution cameras or cameras with higher framerates, the system can be scaled by adding more computers.

Processing images can be CPU-intensive. To avoid image processing incurring too much latency, we reduce the complexity of it by focusing on only detecting that an object is present in an image, rather than determining exactly what the object is. This means that the image processing done by our system can be done quickly, resulting in very low image processing latencies (about 1 ms).

[d]Vertical movement translates to slower shifts in the detected, horizontal position of objects.
[e]The maximum delay between two images taken by two different cameras.
[f]The initial results were gathered on Mac OS X 10.4.8, while the new results are from 10.4.9.

Another challenge in systems using cameras to detect and position objects is relating images from different cameras to each other. High-end cameras can resolve this issue by providing support for either software- or hardware-based synchronization. The commodity cameras used by our system supports neither. Our system resolves this by waiting for data from all cameras currently detecting objects, resulting in up to 33 ms of added latency. This still does not solve the problem of cameras capturing images at different points in time - however, it is better than not detecting objects at all because data from related cameras is processed in alternating rounds.

We have used the system for interacting with different applications on the display wall. This includes controlling the two games Quake 3 Arena and Homeworld[1], and control a custom whiteboard-style application with functionality for creating, resizing and moving simple geometric objects as well as drawing free-hand paths. The system works well for tasks that do not require higher levels of accuracy than our system can deliver, despite the intrinsic lack of synchronization between the cameras.

Acknowledgements

The authors thank Ken-Arne Jensen and Tor-Magne Stien Hagen. This work is supported by the Norwegian Research Council, projects 159936/V30, SHARE - A Distributed Shared Virtual Desktop for Simple, Scalable and Robust Resource Sharing across Computer, Storage and Display Devices, and 155550/420 - Display Wall with Compute Cluster.

References

1. D. Stødle, T.-M. S. Hagen, J. M. Bjørndalen and O. J. Anshus, *Gesture-based, touch-free multi-user gaming on wall-sized, high-resolution tiled displays*, in: *Proc. 4th Intl. Symposium on Pervasive Gaming Applications, PerGames 2007*, pp. 75–83, (2007).
2. B. Wilburn, N. Joshi, V. Vaish, E.-V. Talvala, E. Antunez, A. Barth, A. Adams, M. Horowitz and M. Levoy, *High performance imaging using large camera arrays*, in: *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pp. 765–776, (ACM Press, NY, 2005).
3. D. Cotting, M. Waschbüsch, M. Duller and M. Gross, *WinSGL: synchronizing displays in parallel graphics using cost-effective software genlocking*, Parallel Comput., **33**, 420–437, (2007).
4. J. Mulligan, V. Isler and K. Daniilidis, *Trinocular stereo: A real-time algorithm and its evaluation*, Int. J. Comput. Vision, **47**, 51–61, (2002).
5. J. Y. Han, *Low-cost multi-touch sensing through frustrated total internal reflection*, in: *UIST '05: Proc. 18th Annual ACM symposium on User Interface Software and Technology*, pp. 115–118, (ACM Press, NY, 2005).
6. G. D. Morrison, *A camera-based input device for large interactive displays*, IEEE Computer Graphics and Applications, **25**, 52–57, (2005).
7. D. Douxchamps et. al., *libdc1394, an open source library for handling firewire DC cameras*. http://damien.douxchamps.net/ieee1394/libdc1394/.