CMUcam3: An Open Programmable Embedded Vision Sensor

Anthony Rowe Adam Goode Dhiraj Goel
Illah Nourbakhsh

CMU-RI-TR-07-13

May 2007

Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

In this paper we present CMUcam3, a low-cost, open source, embedded computer vision platform. The CMUcam3 is the third generation of the CMUcam system and is designed to provide a flexible and easy to use open source development environment along with a more powerful hardware platform. The goal of the system is to provide simple vision capabilities to small embedded systems in the form of an intelligent sensor that is supported by an open source community. The hardware platform consists of a color CMOS camera, a frame buffer, a low cost 32-bit ARM7TDMI microcontroller, and an MMC memory card slot. The CMUcam3 also includes 4 servo ports, enabling one to create entire, working robots using the CMUcam3 board as the only requisite robot processor. Custom C code can be developed using an optimized GNU toolchain and executables can be flashed onto the board using a serial port without external downloading hardware. The development platform includes a virtual camera target allowing for rapid application development exclusively on a PC. The software environment comes with numerous open source example applications and libraries including JPEG compression, frame differencing, color tracking, convolutions, histogramming, edge detection, servo control, connected component analysis, FAT file system support, and a face detector.

1 Introduction

The CMUcam3 is an embedded vision sensor designed to be low cost, fully programmable, and appropriate for realtime processing. It features an open source development environment, enabling customization, code sharing, and community.

In the world of embedded sensors, the CMUcam3 occupies a unique niche. In this design exercise we have avoided high-cost components, and therefore do not have many of the luxuries that other systems have: L1 cache, an MMU, DMA, or a large RAM store. Still, the hardware design of the CMUcam3 provides enough processing power to be useful for many simple vision tasks [1], [2], [3], [4], [5] .

An ARM microcontroller provides excellent performance and allows for the execution of a surprisingly broad set of algorithms. A high speed FIFO buffers images from the CIF-resolution color camera. Mass storage is provided by an MMC socket using an implementation of the FAT filesystem so that the file written by the CMUcam3 are immediately readable to an ordinary PC. User interaction occurs via GPIO, servo outputs, two serial UARTs, a button, and three colored LEDs.

We provide a full C99 environment for building firmware, and include libraries such as `libjpeg`, `libpng`, and `zlib`. Additionally, we have developed a library of vision algorithms optimized for embedded processing. Full source is provided for all components under a liberal open source license.

The system described in this paper has been implemented and is fully functional. The system has passed CE testing and is available from multiple international commercial vendors for a cost of approximately US$239. [12]

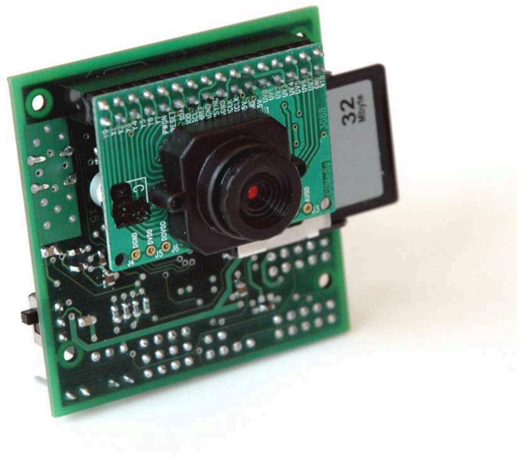

Figure 1: Photograph of the CMUcam3 mated with the CMOS camera board. An MMC memory card used for mass storage can be seen protruding on the right side of the board. The board is 5.5cm $\times$ 5.5cm and approximately 3cm deep depending on the camera module.

1.1 Embedded Vision Challenges

Embedded Vision affords a unique set of functional requirements upon a computational device meant to serve as a visual-sensor. In fact, taken as a general purpose processor, the CMUcam3 is rather underpowered compared to desktop computers or even PDAs. However, if examined as a self-contained vision subsystem, several benefits become clear.

The system excels in IO-constrained environments. The small size and low power of the CMUcam3 enables it to be placed in unique environments, collecting data autonomously for later review. If coupled with a wireless network link (such as 802.15.4 or GPRS), the CMUcam3 can perform sophisticated processing to send data only as needed over a potentially expensive data channel.

Its low cost allows the CMUcam3 to be purchased in greater quantities than other solutions. This makes the CMUcam3 more accessible to a larger community of developers. In several applications, for instance surveillance, reduced cost allows for a meaningful tradeoff between high performance from a single sensor node and the distribution of lower-cost nodes to achieve greater coverage.

The CMUcam3 also has benefits when used as a self-contained part of a greater system. Because of its various standard communications ports (RS-232, SPI, I^2C), adding vision to an existing system becomes straightforward, particularly because the computational overhead is assumed by the separately dedicated CMUcam3 processor rather than imposed upon the main processor and its I/O system.

Finally, having completely open source firmware allows flexibility and reproducibility—anyone can download and compile the code to run on the hardware or alternatively a desktop computer (using the `virtual-cam` module).

1.2 Related Work

There have been numerous embedded image processing systems constructed by the computer vision community in the service of research. In this section we will present a selection of systems that have similar design goals to that of the CMUcam3.

The Cognachrome [10] system is able to track up to 25 objects at speeds as high as 60 Hz. Its drawbacks include cost (more than US$2000), size (four by two by ten inches) and power (more than 5$\times$ that of the CMUcam3) as limitations when creating small form factor nodes and robots.

The Stanford MeshEye [6] was designed for use in low power sensor networks. The design uses two different sets of image sensors, a low resolution pair of sensors is used to wake the device in the presence of motion, while the second VGA CMOS camera performs image processing. The system is primarily focused on sensor networking applications, and less on general purpose image processing.

The UCLA Cyclops [7], also designed around sensor networking applications, uses an 8-bit microprocessor and an FPGA to capture and process images. The main drawbacks are low image resolution (128$\times$128) due to limited RAM and slow processing of images (1 to 2 FPS).

Specialized DSP based systems like the Bluetechnix [9] Blackfin camera boards provide superior image processing capabilities at the cost of power, price and complexity. They also typically require expensive commercial compilers and external development hardware (i.e. JTAG emulators). In contrast, the CMUcam3's development environment is fully open source, freely available and has built-in firmware loading using a serial port.

Various attempts have been made to use general purpose single board computers including the Intel Stargate [8] running Linux in combination with a USB webcam for image processing. Though open source, such systems are quite expensive, large, and demanding of power. Furthermore, USB camera acquired images are typically transmitted to the processor in a compressed format. Compressed data results in lossy and distorted image information as well as the extra CPU overhead required to decompress the data before local processing is possible. The use of slow external serial bus protocols including USB v1.0 limits image bandwidth resulting in low frame rates.

Finally, a number of systems [1], [2], [3] consist of highly optimized software designed to run on standard desktop machines. The CMUcam3 is unique in that it targets applications where the use of a standard desktop machine would be prohibitive because of size, cost or power requirements.

2 CMUcam3

In the following section, we will describe and justify the design decisions leading to the hardware and software architecture of the CMUcam3.

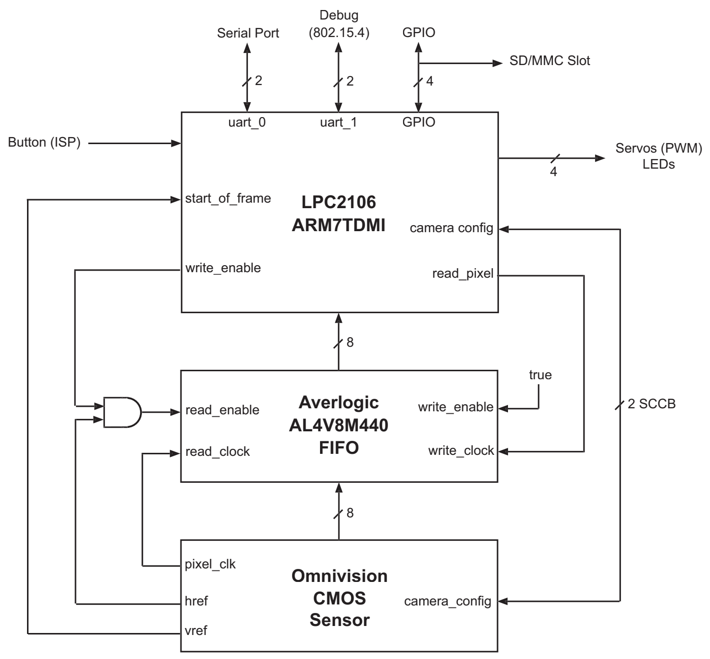

Figure 2: CMUcam3 hardware block diagram consisting of three main components: processor, frame buffer and CMOS camera.

2.1 Hardware Architecture

As shown in Figure 2, the hardware architecture for the CMUcam3 consists of three main components: a CMOS camera chip, a frame buffer, and a microcontroller. The microcontroller configures the CMOS sensor using a two-wire serial protocol. The microcontroller then initiates an image transfer directly from the CMOS camera to the frame buffer. The microcontroller must wait for the start of a new frame to be signaled at which point it configures the system to asynchronously load the image into the frame buffer. Once the CMOS sensor has filled at least 2 blocks of frame buffer memory (128 bytes), the main processor can begin asynchronously clocking data 8 bits at a time out of the image buffer. The end of frame triggers a hardware interrupt at which point the main processor disables the frame buffer's write control line until further frame dumps are needed.

The CMUcam3 has two serial ports (one level shifted), I^{2}C, SPI, four standard hobby servo outputs, three software controlled LEDs, a button and an MMC slot. A typical operating scenario consists of a microcontroller communicating with the CMUcam3 over a serial connection. Alternatively, I^{2}C and SPI can be used, making the CMUcam3 compatible with most embedded systems without relying soley on RS-232. The SPI bus is also used to communicate with FLASH storage connected to the MMC slot. This allows the CMUcam3 to read and write gigabytes of permanent storage.

Unlike previous CMUcam systems, all of these peripherals are now controlled by the processor's hardware and hence do not detract from processing time. The expansion port on the CMUcam3 is compatible with various wireless sensor networking motes including the Telos [15] motes from Berkeley.

The image input to the system is provided by either an Omnivision OV6620 or OV7620 CMOS camera on a chip [14]. As in the CMUcam and CMUcam2, the CMOS camera is mounted on a carrier board which includes a lens and supporting passive components. The camera board is free running and will output a stream of 8-bit RGB or YCbCr color pixels. The OV6620 supports a maximum resolution of 352×288 at 50 frames per second. Camera parameters such as color saturation, brightness, contrast, white balance gains, exposure time and output modes are controlled using the two-wire SCCB protocol. Synchronization signals including a pixel clock (directly connected to the image FIFO) are used to read out data and indicate new frames as well as horizontal rows. The camera also provides a monochrome analog signal.

One major difference between the CMUcam2 and the CMUcam3 is the use of the NXP LPC2106 microcontroller. The LPC2106 is a 32-bit 60 MHz ARM7TDMI with built-in 64 KiB of RAM and 128 KiB of flash memory. The processor is capable of software controlled frequency scaling and has a memory acceleration module (MAM) which provides it with near single cycle fetching of data from FLASH. A built-in boot loader allows downloading of executables over a serial port without external programming hardware. Since the processor uses the ARM instruction set, code can be compiled with the freely available GNU GCC compiler. Built-in downloading hardware and free compiler support makes the LPC2106 an ideal processor for open source development.

The frame buffer on the CMUcam3 is a 50 MHz, 1 MB AL4V8M440 video FIFO manufactured by Averlogic. The video FIFO is important because it allows the camera to operate at full speed and decouples processing on the CPU from the camera's pixel clock. Running the camera at full frame rate yields better automatic gain and exposure performance due to factory default tuning of the CMOS sensor. Even though pixels can not be accessed in a random access fashion, the FIFO does allow for resetting the read pointer which enables multiple pass image processing. One disadvantage of the LPC2106 is that it has relatively slow I/O. Reading a single pixel value can take as long as 14 clock cycles, of those 12 are spent waiting on I/O. Software down sampling, operating on a single image channel, or doing software windowing greatly accelerates image processing since skipping a pixel takes only 8 cycles. Using the FIFO, algorithms can be developed that first process a lower resolution image and can later rewind and revisit regions at higher resolutions if more detail is required. For example frame differencing can be performed on a low resolution gray scale image, while frames of interest containing motion can be saved as high resolution color images. Since processing is decoupled from individual pixel access times, the pixel clock on the camera does not need to be set to the worst case per pixel processing time. This in turn allows for higher frames rates that would not be possible without the frame buffer.

In many embedded applications, such as sensor networks, power consumption is an important factor. To facilitate power savings, we provide three power modes of operation (*active*, *idle* and *power down*) as well as the ability to power down just the camera module. In the active mode of operation when the CPU, camera and FIFO

	Voltage (V)	Current (mA)	Power (mW)
CPU core	1.8	60	108
CPU peripherals	3.3	15	49.5
Frame Buffer	3.3	52	171
Camera	5	25	125
MMC	3.3	4	13.2
Misc	3.3	10	33
Total	na	na	499.7

Table 1: This table shows a breakdown of the power consumption of various components while the camera is fully active.

are all fully operating the system consumes 500 mW of power. Table 1 shows the distribution of power consumption across the various components. When in an *idle* state, where RAM is maintained and the camera is disabled, the system consumes around 300 mW. The transition time between *idle* and *active* is on the order of 30 us. For applications where very low duty cycles are required and startup delays of up to 1 second can be tolerated, we provide an external *power down* pin which gates external power to the board bringing the consumption down to nearly zero (25 uW). In the *power down* state of operation, the processor RAM is not maintained and hence camera parameters must be restored by the firmware at startup.

2.2 Software Architecture

Standard vision systems assume the availability of PC-class hardware. Systems such as OpenCV [17], LTI-Lib [19], and MATLAB [13] require megabytes of memory address space and are written in runtime-heavy languages such as C++ and Java. The CMUcam3 has only 64 KiB of RAM and thus cannot use any of these standard vision libraries.

To solve this problem, we designed and implemented the `cc3` vision system as the main software for CMUcam3. We also implement several components on top of `cc3` as described in this section.

2.2.1 The cc3 Software Vision System

The `cc3` system is a C API for performing vision and control, optimized for the small environment of the CMUcam3.

Features:

- Abstraction layer for interfacing with future hardware systems
- Modern C99 style with consistently named types and functions
- Support of a limited number of image formats for simplicity
- Documentation provided via Doxygen [18]
- Versioned API for future extensibility
- `virtual-cam` module for PC-based testing and debugging (see below)

`cc3` is a part of the CMUcam3 distribution, and is openly available at the CMUcam website [12]. Below is an example of the `cc3` based source code showing you how to track a color:

```
int main(void)
{
  cc3_image_t img;
  cc3_color_track_pkt t_pkt;

  // init filesystem driver
  cc3_filesystem_init ();

  // configure uarts
  cc3_uart_init (0, CC3_UART_RATE_115200, CC3_UART_MODE_8N1, CC3_UART_BINMODE_TEXT);

  cc3_camera_init ();

  cc3_camera_set_colorspace(
     CC3_COLORSPACE_RGB);
  cc3_camera_set_resolution (
     CC3_CAMERA_RESOLUTION_LOW);
  cc3_camera_set_auto_white_balance (true);
  cc3_camera_set_auto_exposure (true);

  printf( "Enter color bounds to track: " );
  scanf( "%d %d %d %d %d %d\n", &t_pkt.lower_bound.chan[CC3_RED_CHAN],
   &t_pkt.lower_bound.chan[CC3_GREEN_CHAN], &t_pkt.lower_bound.chan[CC3_BLUE_CHAN],
   &t_pkt.upper_bound.chan[CC3_RED_CHAN], &t_pkt.upper_bound.chan[CC3_GREEN_CHAN],
   &t_pkt.upper_bound.chan[CC3_BLUE_CHAN] );

  img.channels = 3;
  img.width = cc3_g_pixbuf_frame.width;
  img.height = 1;
  img.pix = cc3_malloc_rows (1);

 while(1) {
   cc3_pixbuf_load ();
   cc3_track_color_scanline_start (t_pkt);

   while (cc3_pixbuf_read_rows(img.pix, 1)){
     cc3_track_color_scanline (&img, t_pkt);
   }
   cc3_track_color_scanline_finish (t_pkt);

   printf( "Color blob found at %d, %d\n", t_pkt.centroid_x, t_pkt.centroid_y );
 }
}
```

The next example shows how a developer can access raw pixels. The following code section returns the location of the brightest red pixel found in the image:

```
uint8_t max_red, max_red_y, max_red_x;
cc3_pixel_t my_pix;

max_red=0;
cc3_pixbuf_load ();
while(cc3_pixbuf_read_rows(img.pix, 1)){
  // read a row into the image
  // picture memory from the camera
  for(uint16_t x = 0; x < img.width; x++) {
    // get a pixel from the img row memory
    cc3_get_pixel (&img, x, 0, &my_pix);
    if(my_pix.chan[CC3_CHAN_RED] > max_red){
      max_red = my_pix.chan[CC3_CHAN_RED];
      max_red_x = x;
      max_red_y = y;
    }
  }
  y++;
}
printf( "Brightest Red Pixel: %d, %d\n",
  max_red_x, max_red_y );
```

2.2.2 virtual-cam

The `virtual-cam` module is part of the `cc3` system as mentioned above. It provides a simulated environment for testing library and project code on any standard PC by compiling with the system's native GCC compiler. This allows for full use of the PC's debugging tools to diagnose problems in user code. Oftentimes, a difficult to understand behavior observed on the CMUcam3 will easily manifest itself as a bad pointer dereference or other easily found bug when run on a standard PC with memory protection.

While not all of CMUcam3's functionality is implemented in `virtual-cam` (missing features include the hardware-specific components of servo control and GPIO), enough functionality is provided to enable off-line diagnostic testing.

2.2.3 CMUcam2 Emulation

The CMUcam2 [20] provides a simple human readable ASCII communication protocol allowing for interactive control of the camera from a serial terminal program or a micro-controller. The CMUcam2 is capable of many functions including in-built color tracking, frame differencing, histogramming as well as binary image transfers. The CMUcam2 comes with a graphical user interface running on a PC that allows users to experiment with various functions. The CMUcam3 emulates most of the CMUcam2's

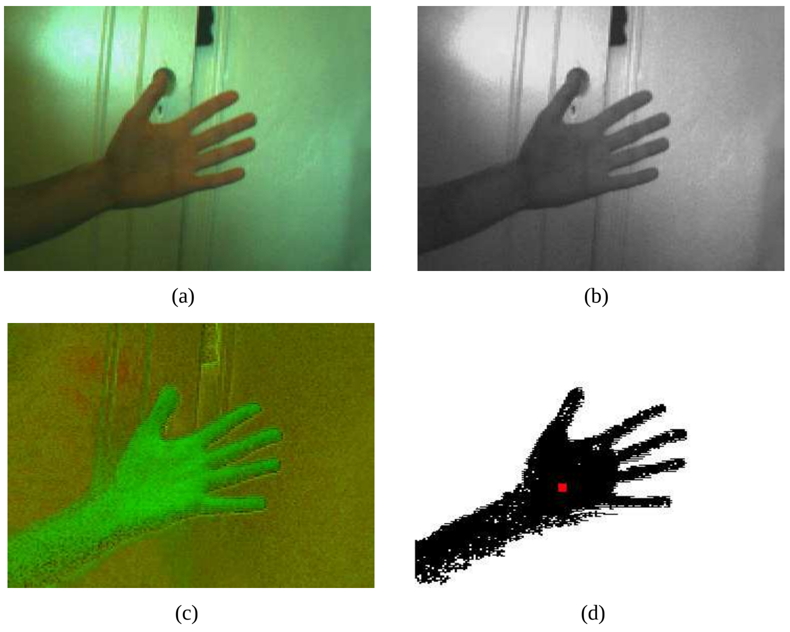
(a) (b) (c) (d)

Figure 3: The following images show the advantage of color tracking in the HSV color space. Figure (a) shows an RGB image, (b) shows the intensity (V) component of the HSV image, (c) shows the Hue and Saturation components of the image without intensity (d) shows the segmented hand with the center of mass in the middle.

functions making it a drop-in replacement for the CMUcam2. The CMUcam2 emulation extends upon the original CMUcam2 with superior noise filtering, HSV color tracking and JPEG compressed image transfers.

2.2.4 Color Tracking

The original CMUcam tracks color blobs using a simple RGB threshold color model. Though computationally lightweight, it does not adapt well to changing light conditions and can only track a single color at one time. The CMUcam3 improves tracking performance by providing the option to use the Hue Saturation Value (HSV) color space, provisions for connected component blob filtering and the ability to track multiple colors. Figure 3 shows how the HSV color space can remove lighting effects simplifying color segmentation. Since the system is open source, it is simple for end users to further improve color tracking by building more complex color models.

2.2.5 Frame Differencing

As an example program to illustrate frame differencing, we provide a simple security camera application. The camera continuously compares the previous image and the current image. If an images changes by more than a preset threshold, the image is saved as a JPEG on the MMC card.

2.2.6 Convolutions

We provide a general convolution library that allows custom kernels to be convolved across an image. This can be used for various filters that perform tasks like edge detection or blurring.

2.2.7 Compression

New to the CMUcam3 is the ability to compress images with both `libjpeg` and `libpng`. Using different destination managers, one can redirect the output of `libjpeg` to the MMC, serial output, or any other communication bus. Depending on the quality of the image, `libjpeg` can produce images as small as 4 KiB.

2.2.8 Face Detection

The CMUcam3 incorporates the ability to detect faces in plain-background environments. The face detector technique is based on the feature-based approach, proposed by Viola and Jones, in which a cascade of classifiers are trained for Haar-like rectangular features selected by AdaBoost [16].

The integral image is a key data structure used in Viola-Jones. Unfortunately, it consumes significant memory. Even a low resolution integral image of 176×144 requires about 76 KiB of memory, far exceeding available memory.

Along with memory constraints, the processor lacks floating point hardware. As a result, two unique customizations were applied to the face detection implementation for CMUcam3:

- Only a part of the whole image is loaded in main memory at any time. As a consequence, the maximum resolution of a detected face is limited to 60×60 pixels.
- All the classifier thresholds and corresponding compared values are computed using fixed point arithmetic, via a binary scaling method.

A few other optimizations were made to improve performance:

- When scanning sub-windows, neighboring sub-windows are illumination normalized with iteratively computed standard deviation (std), instead of being computing independently. This can provide a speed up of approximately $3\times$.
- Sub-windows that are are too homogeneous (std $<$14) or too dark or bright (mean $<$30 or mean $>$200) are discarded immediately, short-circuiting unnecessary computation in regions unlikely to yield positive detection hits.

With these changes, CMUcam3 face detection operates on-board at 1 Hz.

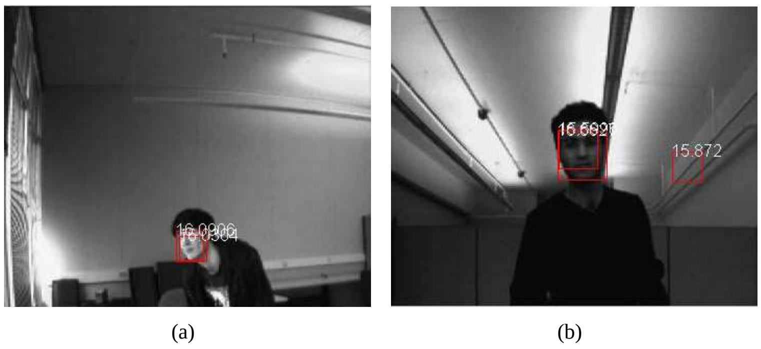

Figure 4: Sample output from a modified Viola-Jones face detector. Faces are denoted with boxes. Image (b) shows how texture in the background can occasionally be detected as a false positive.

2.2.9 Polly

The Polly [3] algorithm provides visual navigation information based on color. This navigation was used on the Polly robot to give tours of the MIT AI laboratory in the early 90's. The algorithm originally consisted of three steps: blurring the image, edge detection and generating a free space map starting from the bottom of the image upward towards any edges. Our implementation applies a 3x3 blur followed by a simple edge detector. We then filter out small edges using our connected component module. As can be seen in Figure 5 the algorithm returns a histogram of the free space in front of the robot. Polly is able to run on-board CMUcam3 at 4 fps, operating on a 176x144 image.

2.2.10 SpoonBot

SpoonBot is a small mobile robot consisting of a CMUcam3, two continuous rotation hobby servos mounted to wheels, a four AA battery pack and a micro-servo connected to a plastic spoon. The two hobby servos allow SpoonBot to drive forward, backward and rotate left and right. The rear mounted micro-servo pushes the spoon up and down acting as a tilt degree of freedom. SpoonBot can use the Polly algorithm described above to drive around a table top or it can follow colored objects. All control and navigation is run locally on the CMUcam3, since the board can compute and command servo control signals directly, without the need for conventional robot control hardware.

3 Performance

In this section we discuss execution time and memory consumption for various CMUcam3 software components. Depending on the image resolution and complexity of the algorithm, these values can vary significantly. The goal of this section is to provide some intuition for the various types of image processing that are possible using the

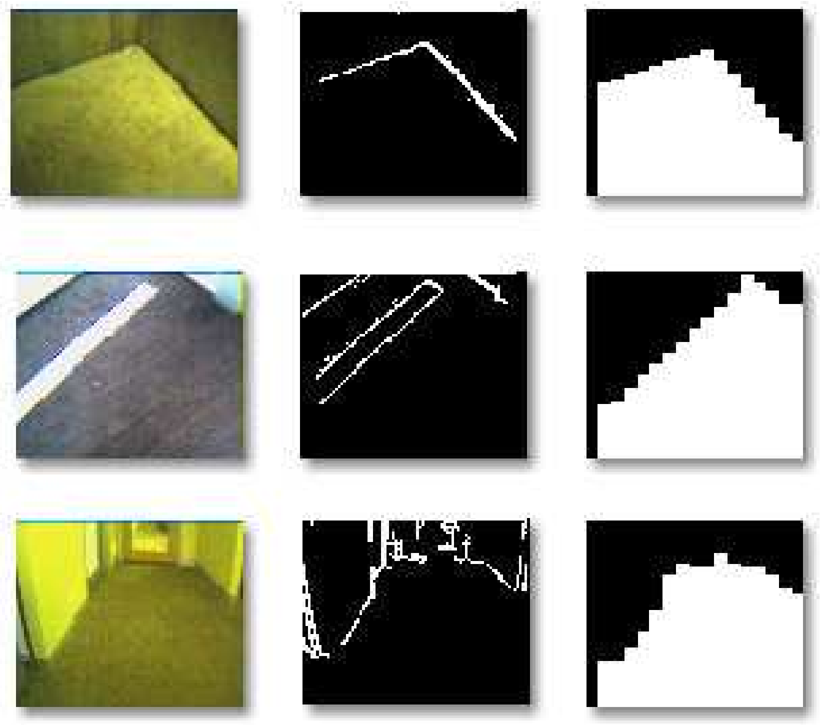
Figure 5: Sample output of the Polly algorithm. The first column shows the original image. The second column shows the image after a blur filter, edge detection and small connected component filter. The final column shows the histogram representing free area in front of the camera.

	CIF RGB	CIF Mono	QCIF RGB	QCIF Mono
Load Frame	2 ms	2 ms	2 ms	2 ms
Copy Memory	210 ms	128 ms	52 ms	32 ms
Pack Pixels	150 ms	160 ms	38 ms	39 ms
Total FPS	2.76	3.45	10.87	13.70

Table 2: This table shows a breakdown of the time required when loading a CIF and QCIF image in color as well as grayscale.

CMUcam3 and to understand where computational and I/O bottlenecks are typically found.

Table 2 shows the breakdown of time consumed by the three major steps involved in loading a frame into the processor's memory. The *Load Frame* column refers to the time required once a new frame arrives and before data can start to be retrieved from the frame buffer. This does not directly correlate to the `cc3_pixbuf_load()` function because this function incorporates leftover time from when the last frame finished. The `Copy Memory` column refers to the time required to move data from the frame buffer into the processor. This directly correlates to the speed of the

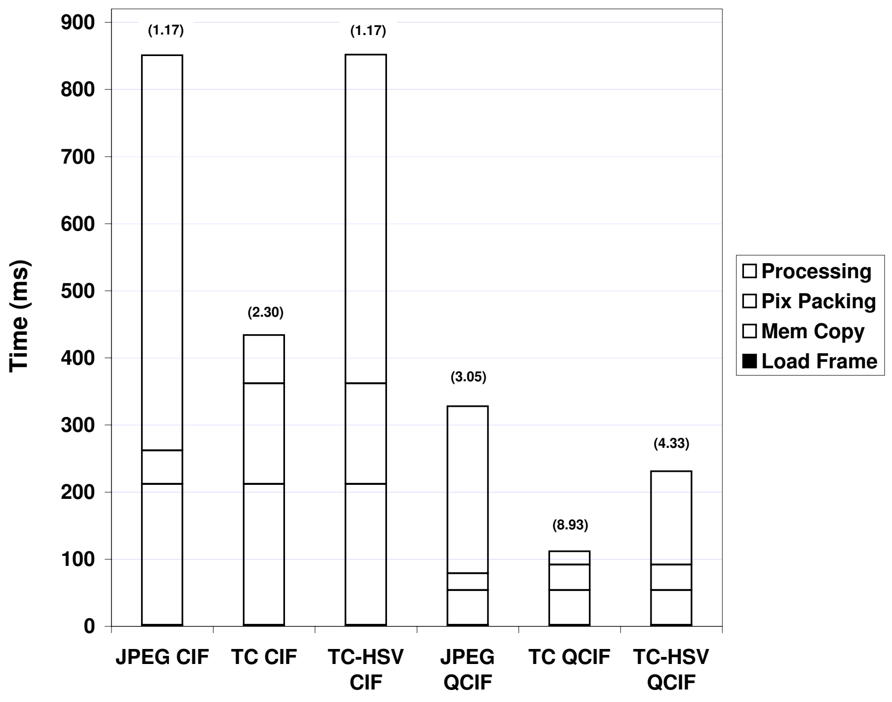

Figure 6: This figure compares the execution times of loading a frame, copying the image from the frame buffer to the processor, unpacking the pixels and processing the new frame. JPEG, Track Color (TC) and Track Color in the HSV color space (TC-HSV) are shown at two different resolutions. The numbers in parenthesis represent the frame rate of the operation.

`cc3_pixbuf_read_rows()` function. Operating at a lower resolution obviously decreases the execution time because fewer pixels are fetched. Operating on a single channel instead of three channels provides only a 1.625× increase in speed. This increase is due to no longer having to read all of the color pixels, however, since the CMOS camera does not have a monochrome output mode, color information must still be clocked out of the FIFO. The final *Pack Pixel* column shows the time required to convert the GRGB pattern from the camera in memory into the local RGB pixel structure. This corresponds to the `cc3_get_pixel()` function call. It is possible to greatly reduce the pixel construction time by designing algorithms that operate on the raw memory from the camera. This becomes a trade-off between simple portable code and execution speed. We provide examples of both methodologies for those interested in highly optimized implementations.

Figure 6 shows the relative time consumption of the previously mentioned frame loading operations along with processing times for three different algorithms: JPEG, Track Color and Track Color HSV. The JPEG algorithm in this example compresses a color image in memory and does not write the output to a storage device. The Track Color (TC) and Track Color HSV (TC-HSV) algorithms are profiled directly from the CMUcam2 emulation code. Each algorithm finds the bounding box, centroid and density of a particular color specified. For this test we show the worst-case performance by tracking all active pixels. The Track Color HSV benchmark is identical to Track Color except that it performs a software based conversion from the RGB to HSV color space for each pixel. The general trend found in these plots is that very simple algorithms such as tracking color are mostly I/O limited. For example Track Color spends only 17% of the time on processing. A more complex algorithm, JPEG, spends 62% of its time on processing. JPEG also shows an example of where optimized pixel accesses can drastically reduce the pixel packing time. However as can be seen in the JPEG operating on a QCIF image, as resolution decreases these optimizations become less relevant.

As previously mentioned, the LPC2106 has 64 KiB of internal RAM and 128 KiB of ROM. By default, 9 KiB of RAM is reserved for stack space and 9 KiB of RAM is used by the core software libraries (including `libc` buffers). A 176×144 (QCIF) gray-scale image requires 25 KiB of RAM, while a 100×100 RGB image requires 30 KiB of memory. All processing on larger sized images must be performed on a section by section basis, or using a sliding window scan-line approach. For example, JPEG requires only eight full rows (8 KiB) of the image in addition to the storage required for the compressed image (less than 12 KiB). The code space consumed by most CMUcam3 applications is quite small. The full CMUcam2 emulation with JPEG compression and the FAT file system requires 96 KiB of ROM. A simple program that loads images and links in the standard library functions requires 52 KiB of ROM. The FAT filesystem and MMC driver require an additional 12 KiB of ROM.

4 Conclusions and Future Works

The goal of this work was to design and publicly release a low cost, open source, embedded color computer vision platform. The system can provide simple vision capabilities to small embedded systems in the form of an intelligent sensor that is supported by an open source community. Custom C code can be developed using an optimized GNU toolchain and flashed onto the board using the serial port without external downloading hardware. The development platform includes a virtual camera target and numerous open source example applications and libraries.

The main drawback of the CMUcam3 hardware platform is the lack of RAM and computation speed required for many complex computer vision algorithms. We currently have a prototype system using a 600 MHz Blackfin media processor from Analog Devices. Ideally, we would like to provide a software environment for this new platform that is compatible with our existing environment to help reduce the learning curve typically associated with high-end DSP systems. Eventually, applications can be prototyped on a PC using our virtual-cam with various hardware deployment options to support that particular application's needs. Staying true to the spirit of the CMUcam project, we are also developing a simpler and cheaper hardware platform using a lower cost ARM7 processor without the frame buffer. This device will be compatible with the current software environment except that it will be restricted to pure scan-line style processing.

Acknowledgements

The authors would like to thank Charles Rosenberg for his continuing contributions to this project.

References

[1] J. Bruce, T. Balch, and M. Veloso, “Fast and Inexpensive Color Segmentation for Interactive Robots”, *The Proceedings of IROS*, 2000.

[2] G.D. Hager and K. Toyama, “The XVision System: A general purpose substrate for real-time vision applications,” *Computer Vision and Image Understanding*, vol. 69, no. 1, pp. 23-27, January 1998.

[3] I. Horswill, “Polly: A vision-based artificial agent”, *The Proceedings of the Eleventh Nataional Conference on Artificial Intelligence*, 1993.

[4] R. Sargent, B. Bailey, C. Witty and A. Wright, “Dynamic Object Capture Using Fast Vision Tracking”, *AI Magazine* vol 18, no.1 1997.

[5] I. Ulrich and I. Nourbakhsh, “Appearance-Based Obstacle Detection with Monocular Color Vision”, *AAAI Conference* pp. 866-871, 2000.

[6] S. Hengstler and H. Aghajan, “A Smart Camera Mote Architecture for Distributed Intelligent Surveillance”, *ACM SenSys Workshop on Distributed Smart Cameras*, Oct. 2006.

[7] M. Rahimi, R. Baer, O. Iroezi, J. Garcia, J. Warrior, D. Estrin, M. Srivastava, “Cyclops: In Situ Image Sensing and Interpretation in Wireless Sensor Networks”, *ACM SenSys*, Nov. 2005.

[8] ”Intel Stargate Platform”, `http://www.xbow.com/Products`, Viewed on March 27, 2007

[9] ”Bluetechnix Blackfin DSP”, `http://www.tinyboards.com`, Viewed on March 27, 2007

[10] R. Sargent and A. Wright ”The Cognachrome Color Vision System”, `http://www.newtonlabs.com/cognachrome`, Viewed on March 27, 2007

[11] K. Konolige, “The SRI Small Vision System Website”, `http://www.ai.sri.com/˜konolige/svs/`, Viewed on March 23, 2006.

[12] “CMUcam Website”, `http://www.cmucam.org`, Viewed on March 25, 2007

[13] “MATLAB”, `http://www.mathworks.com/products/matlab/`, Viewed on March 25, 2007

[14] “Omnivision”, `http://www.ovt.com`, Viewed on March 25, 2007

[15] J. Polastre, R. Szewczyk and D. Culler, ”Telos: Enabling Ultra-Low Power Wireless Research,” *Spots*, 2005.

[16] P. Viola and M. Jones, ”Robust Real-Time Face Detection,” *Computer Vision*, vol. 2, pp. 747-752, 2001.

[17] “Open Source Computer Vision Library”, `http://www.intel.com/technology/computing/opencv/`, Viewed on March 25, 2007.

[18] “Doxygen”, `http://www.doxygen.org/`, Viewed on March 27, 2007.

[19] “LTI-Lib”, `http://ltilib.sourceforge.net/`, Viewed on March 25, 2007.

[20] A. Rowe, C. Rosenberg, I. Nourbakhsh, “A Second Generation Low Cost Embedded Color Vision System”, *Embedded Computer Vision Workshop, CVPR* 2005.