CVPI
A Computer Vision Library
For Mobile and Embedded Platforms

Devin Homan

Version 0.1

Contents

Chapter 1

Introduction

CVPI is a library for implementing computer vision programs on computers supporting OpenVG. It adds additional image processing capabilities to OpenVG that are necessary for computer vision, as well a as providing an interface to setup the rendering environment.

OpenVG is a hardware accelerated C API for vector and raster 2D graphics [5]. It is widely supported on mobile and on embedded platforms.

1.1 Motivation

Single board computers, such as the Raspberry Pi, have become a popular platform for robotics. These boards come with GPUs, which are currently not being utilized in this area. The GPUs on these computers support OpenVG and OpenGL ES. Current computer vision software, such as OpenCV, rely on OpenCL or CUDA, which are GPU languages that are not supported on current mobile and embedded GPUs. Currently, to do computer vision on mobile and embedded platforms, all computation has to be performed by the on board CPU, or the data has to be transferred to a remote computer for processing. OpenVG offers the necessary functionality to create computer vision software and is available natively on these platforms.

The Raspberry Pi is already being used for computer vision controlled robots and is marketed as a computing platform for teaching programming and computer hardware concepts to secondary students. In one such example, a high-school student built a line tracking robot using OpenCV. The Raspberry Pi, which normally runs at 800MHz, was overclocked to 1GHz, and the tracking rate was between 12 to 14fps [8]. Running OpenVG and other GPU code is non-trivial and under-documented on the Raspberry Pi. The CVPI EGL interface aims to alleviate the complexity as much as possible. CVPI can also allow for the same results, that the student achieved with OpenCV, with greater efficiency by moving computation to the GPU.

CVPI could be utilized beyond the Raspberry Pi. Many GPUs used in smart phones and other mobile devices support OpvenVG; though Android apparently does not have an OpenVG API implementation. There are other competing single boards, such as pcDuino, Odroid, and Banana Pi that could also utilize CVPI.

1.2 Existing Software Architecture

OpenVG interfaces with the kernel using EGL (Native Platform Graphics Interface). EGL, OpenVG, and OpenGL ES are C APIs published by Khronos. EGL is used to set up the rendering context: surface dimensions, what libraries to support (OpenVG or OpenGL), pixel format, where rendering will be done (on screen or in memory) [6]. The Raspberry Pi's Khronos APIs were implemented by the GPU chip manufacturer, Broadcom, and are BSD licensed [11].

OpenGL ES was not used in this project. OpenVG and OpenGL ES cannot communicate directly with each-other; they cannot share contexts [6, p. 7]. They could potentially communicate by reading and writing to and from main memory. OpenVG has so far been sufficient for implementing computer vision related algorithms.

1.3 CVPI Interface and Library

CVPI has two main parts, an EGL setup and take-down interface, and a library of image processing functions relevant to computer vision.

CVPI offers a template interface for setting up and taking down EGL. Setting up EGL correctly is non-trivial, especially on the Raspberry Pi, with steps that have to be done in a certain order and settings that are very particular. The majority of the steps are always the same; however, there are some steps that cannot be accounted for, which are implementation dependent, that the user must provide.

CVPI is meant to add to OpenVG rather than exist on top of it. OpenVG is a C API, and so CVPI was written in C, specifically C99. C also has the benefit of portability between client languages. CVPI can be used by programmers wishing to write their projects in other languages. Most languages have built-in or standard methods for binding to C functions and values.

Chapter 2

CVPI EGL Interface

Setting up and taking down EGL is non-trivial. The file `cvp_egl_config.h` contains the functions `cvpi_egl_settings_create`, `cvpi_egl_settings_check`, `cvpi_egl_instance_setup`, and `cvpi_egl_instance_takedown` to help the programmer in this task.

`cvpi_egl_settings_create` creates a structure of settings information that can be passed to `cvpi_egl_settings_create` or to `cvpi_egl_settings_check`. `cvpi_egl_settings_check` can be used to check for faulty settings in the structure. If a bad setting is found, CVPI_FALSE is returned and a warning is printed to the standard error output (stderr). Every setting has associated `set` and `check` functions.

`cvpi_egl_instance_setup` creates a cvpi_egl_instance structure that can be passed to `cvpi_egl_instance_takedown`. The structure gives the take-down procedure the information needed to undo the setup procedure. The `cvpi_egl_instance_setup` template performs the following steps to setup EGL on the Raspberry Pi:

1. Calls bcm_host_init()
    - Specific to the Raspberry Pi, it is an undocumented procedure required for Broadcom's implementation
2. Calls eglGetDisplay()
3. Calls eglInitialize()
    - Initializes the display returned by eglGetDisplay
4. Calls eglBindAPI()
    - Chooses the rendering API: OpenVG or OpenGL ES

Figure 2.1: Algorithm for setting up EGL. *(cont.)*

5. Calls eglChooseConfig() twice

    - First, to get the number of configurations supporting the client provided settings. Space is allocated, and `eglChooseConfig()` is called again to populate the space with configuration information.

6. An EGL surface is created. If the surface is a pixmap or window, then the user must supply a function to create an EGLNativePixmapType or an EGLNativeWindowType, respectively. These are implementation specific types.

    - If the user specified a pixmap surface
        (a) EGLNativePixmapType is created using a user supplied function.
        (b) `eglCreatePixmapSurface()` uses the EGLNativePixmapType and the `eglChooseConfig()` generated configuration list to create an EGL surface. The function loops through configurations returned by `eglChooseConfig()` until one works.
    - If the user specified a window surface
        (a) EGLNativeWindowType is created using a user supplied function. There are multiple Broadcom implementations for creating window surfaces, such as the set of functions `vc_dispmanx_*`, which are Broadcom specific, and OpenWF, a Khronos API for creating window surfaces [11].
        (b) `eglCreateWindowSurface()` uses the EGLNativeWindowType and the `eglChooseConfig()` generated configuration list to create an EGL surface. The function loops through configurations returned by `eglChooseConfig()` until one works.
    - If the user specified a pbuffer surface
        (a) `eglCreatePbufferSurface()` is called with the `eglChooseConfig()` generated configuration list to create an EGL surface. The function loops through configurations returned by `eglChooseConfig()` until one works.
    - If no surface type was specified, then this step is skipped.

7. Calls `eglCreateContext()` to create a rendering context.

8. Calls `eglMakeCurrent()` if the user specified for this context to be the current context.

    - The current-context `read` and `draw` parameters are set to the same surface. OpenGL ES allows for different surfaces for reading and drawing; however, OpenVG does not. The user can change this after `cvpi_egl_instance_setup` finishes.

Figure 2.1: Algorithm for setting up EGL.

Every step in the process is logged to `cvpi_log_file`, which defaults to `stderr`. Finding

a working EGL setup procedure for a particular project will likely require multiple revisions. Broadcom's EGL implementation is poorly documented and does not give explanations why failures occur when EGL fails to start, which is in part the reason for `cvp_egl_config.h`.

If an EGL error occurs in `cvpi_egl_instance_setup`, then the function will jump to the appropriate place in the take-down procedure. If a system error occurs, then the GPU might be left in a bad state that makes it impossible to create another EGL instance without reboot. The operating system does not appear to entirely clean up GPU related material after the program halts. The take-down procedure is as follows:

1. Call `eglDestroyContext()`.
2. Call `eglDestroySurface()`.
3. If a window or pixmap was created, call a user supplied function to undo what was done by the user supplied EGLNativeWindowType or EGLNativePixmapType returning function.
4. Free memory allocated for `eglChooseConfig()`.
5. Call `eglTerminate()`.
6. Free memory allocated for `cvpi_egl_instance_setup`'s return value.
7. Call `bcm_host_deinit()`, if using the Broadcom implementation.

Figure 2.2: Algorithm for taking down EGL.

Chapter 3

CVPI Image Library

3.1 OpenVG API

OpenVG has functions for manipulating images. OpenVG can read and write using a variety of different formats. Internally, images are represented using four 8-bit channels per pixel [5].

The choice of what functions to implement came from a list of functions which are described on the web site, Image Processing Learning Resources in the "Worksheets" subdirectory [10]. Other functions, such as histogram equalization, come from Computer Vision by Richard Szeliski [14]. For the most part, functions were only implemented if they could initialize the GPU and could not be trivially and directly done with the existing OpenVG functions. Other functions that cannot utilize the GPU might be added to CVPI in the future. The GPU is very proficient at performing mapping operations on data sets but is not very good at doing reduction operations.

3.1.1 CPU/GPU Memory Transfer

OpenVG can read data directly from CPU memory into an image using `vgImageSubData`, and can write directly to CPU memory with `vgGetImageSubData`. OpenVG uses RGBA (red, green, blue, alpha) to represent the four channels [5]. On little-endian systems, CVPI uses VG_sARGB_8888 for the color space. When processing data, returned by `vgGetImageSubData`, with array indexing, the blue channel is the first index, green is the second, red is the third, and alpha is the last. CVPI has not yet been tested on a big-endian system. ARM CPUs support both endiannesses but, so far, there are no available big-endian operating system images for the Raspberry Pi.

CVPI assumes that in memory, the image's origin is in the upper left with data proceeding left to right and then top to bottom, which is the standard for Video4Linux (V4L) formats [12, Sec. 2.1]. OpenVG's image coordinate system starts in the lower left corner with the x-position in the horizontal and the y-position in the vertical [5, Sec. 10.1]. Essentially the y-axis is flipped and translated. CVPI functions are not affected by this difference, unless otherwise noted.

3.1.2 Copying Images

`vgCopyImage` can copy part of an image into another image [5]. This function is utilized extensively by CVPI.

3.1.3 Pixel Vector Matrix Multiplication

`vgColorMatrix` multiplies a 4×4-matrix by each pixel vector and adds a bias vector to the resulting value to create a new image [5]. `vgColorMatrix`'s matrix argument is the transpose of the mathematical representation, plus four bias values.

```
A[20] = {1,  2, 3, 4,
         5,  6, 7, 8,
         9, 10,11,12,
         13,14,15,16,

         17,18,19,20};
```

$$\begin{pmatrix} R' \\ G' \\ B' \\ A' \end{pmatrix} \leftarrow \begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix} + \begin{pmatrix} 17 \\ 18 \\ 19 \\ 20 \end{pmatrix} \rightarrow \begin{pmatrix} R + 5G + 9B + 13A + 17 \\ 2R + 6G + 10B + 14A + 18 \\ 3R + 7G + 11B + 15A + 19 \\ 4R + 8G + 12B + 16A + 20 \end{pmatrix}$$

Figure 3.1: `vgColorMatrix` argument and mathematical representation [5, p. 176].

The square matrix is an array of floats between zero and one. The last four parameters of the input array are also floats between zero and one. OpenVG represents pixel values as decimal numbers between 0 and 1 when multiplying and adding, but the output is then re-scaled to range from 0 to 255.

Pre-Defined Matrices

CVPI has a number of pre-defined inputs.

cvpi_invert_colors

$$\begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix} + \begin{pmatrix} 255 \\ 255 \\ 255 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 255 - R \\ 255 - G \\ 255 - B \\ A \end{pmatrix}$$

cvpi_avuy2ayuv Change the channel ordering returned by the function `cvpi_yuyv2yuva` to canonical form.

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} V \\ U \\ Y \\ A \end{pmatrix} \rightarrow \begin{pmatrix} Y \\ U \\ V \\ A \end{pmatrix}$$

cvpi_pixel_average Average all channel values together and output the average in each channel.

cvpi_pixel_color_average Average the color channel values together and output the average in each color channel.

cvpi_channel_red, cvpi_channel_green, cvpi_channel_blue, cvpi_channel_alpha Replace the values in the other channels with the value in the specified channel.

3.1.4 Convolution

OpenVG has two functions for convolution `vgConvolve` and `vgSeparableConvolve`. OpenCV also has a Gaussian blur function, `vgGaussianBlur` [5].

OpvenVG's convolution formula is:

$$I(x,y) = s\left(\sum_{i=0}^{w-1}\sum_{j=0}^{h-1} k(w-i-1, h-j-1)P(x+i-s_x, y+j-s_y)\right) + b$$

where w and h are the image's width and height, k is the convolution kernel, P is the input image, s_x and s_y are integer values for shifting the input and output, s is a scalar, and b is a bias. Image channel values are treated as floating point values between 0 and 1, so b must be a value between -1 and 1. The image and kernel are indexed from zero starting in the lower left corner [5, pp. 177-180].

When coding in a convolution kernel in C, the mathematical definition of the convolution kernel must be transposed then flipped. The kernel dimensions will also be switched from the mathematical representation.

For example,

$$\begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}^T \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 3 & 2 & 1 \\ 6 & 5 & 4 \\ 9 & 8 & 7 \end{pmatrix}$$

```
VGshort kernel[9] = {3,2,1,
                     6,5,4,
                     9,8,7};
```

Proof. Given a two-dimensional C array representation of an OpenVG kernel, C defines the one-dimensional index of an element in a two-dimensional array to be $y \cdot \text{arrayHeight} + x$. The OpenVG kernel entry (i, j) is located at $i \cdot \text{kernelHeight} + j$ [5, p. 178]. So because arrayHeight = kernelHeight and arrayWidth = kernelWidth, array$[x][y]$ = kernel(i, j), where $x \equiv i$ and $y \equiv j$. Because the array starts in the upper left and the kernel starts in the lower left, this mapping will flip vertically the visual ordering. The matrix array is in

row-major form, so to get the column major form, the matrix must be transposed. To go from the mathematical representation, to the OpenVG input array representation, the steps must be done in reverse order. Thus, the transpose must be done before flipping.

$$\left(\begin{pmatrix}3 & 2 & 1\\ 6 & 5 & 4\\ 9 & 8 & 7\end{pmatrix}\begin{pmatrix}0 & 0 & 1\\ 0 & 1 & 0\\ 1 & 0 & 0\end{pmatrix}\right)^T \rightarrow \begin{pmatrix}1 & 4 & 7\\ 2 & 5 & 8\\ 3 & 6 & 9\end{pmatrix}$$

□

3.1.5 Value Mapping

OpenVG has two functions, `vgLookup` and `vgLookupSingle`, that can be used for function mapping. The mapping functions are represented by arrays of length 256, with values ranging from 0 to 255. `vgLookup` uses a different mapping function for each channel.

$$\begin{pmatrix}R\\ G\\ B\\ A\end{pmatrix} \begin{matrix}\xrightarrow{f_R}\\ \xrightarrow{f_G}\\ \xrightarrow{f_B}\\ \xrightarrow{f_A}\end{matrix} \begin{pmatrix}R'\\ G'\\ B'\\ A'\end{pmatrix}$$

While, `vgLookupSingle` maps a single channel, C, to a value representing all four channels using a single mapping function [5].

$$C \rightarrow \begin{pmatrix}R'\\ G'\\ B'\\ A'\end{pmatrix}$$

CVPI provides a number of function arrays for use with `vgLookup`.

cvpi_identity_array The input values equal the output values.

cvpi_inversion_array Output equals 255 − input.

cvpi_255_array All values are mapped to 255.

cvpi_zeros_array All values are mapped to 0.

cvpi_binary_array 0 is mapped to 0, and non-zero values are mapped to 255.

cvpi_binary_array_inverted 0 is mapped to 255, and non-zero values mapped to 0.

cvpi_sqrt_array_floor $\lfloor\sqrt{x}\rfloor$

cvpi_sqrt_array_ceil $\lceil\sqrt{x}\,\rceil$

cvpi_sqrt_array_round $\lfloor\sqrt{x}+0.5\rfloor$

3.2 Filters

CVPI provides a number of pre-defined feature detection kernels. It also provides a function, `cvpi_image_magnitude`, for calculating the gradient magnitude, that is

$$\sqrt{K_1^2 + K_2^2}$$

where K_1 and K_2 are the convolution results. The function is applied independently to each channel.

	Y	X
Sobel	$\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$	$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$
Scharr	$\begin{pmatrix} 3 & 10 & 3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{pmatrix}$	$\begin{pmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{pmatrix}$
Prewitt	$\begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}$
Robert's Cross	$\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$

Figure 3.2: Predefined convolution matrices [10][16].

3.2.1 Magnitude

$$G = \sqrt{G_x^2 + G_y^2}$$

.

The function `cvpi_image_magnitude` calculates the pixel magnitude between two images. For the magnitude to be computed on the GPU, the sum cannot exceed 255. The inputs G_x and G_y are integer values between 0 and 255. Let $G_x = ax$ and $G_y = ay$, the formula becomes

$$G = a\sqrt{x^2 + y^2}$$

, where $x^2 + y^2 \leq 255$, $x^2 \leq 127.5$, and $y^2 \leq 127.5$. Given $G_x^2 = a^2x^2$ and solving for a where $G_x = 255$ and $x = \sqrt{127.5}$, $a = 22.5831795814$.

Three `vgLookup` array functions were calculated with G_x as the index input.

$$x_{\text{ceil}} = \left\lceil \frac{G_x}{a} \right\rceil,\ x_{\text{floor}} = \left\lfloor \frac{G_x}{a} \right\rfloor,\ \text{and } x_{\text{round}} = \left\lfloor 0.5 + \frac{G_x}{a} \right\rfloor$$

`cvpi_image_magnitude` applies the same look-up table to both inputs, G_x and G_y. The user specifies which table to use. The resulting images are added together with `cvpi_image_add`. The sums will vary between 0 and 255. The square root function is applied to the sum, s, using `vgLookup`. Which of the three square root functions to apply depends on the user.

$$\lceil \sqrt{s} \rceil, \lfloor \sqrt{s} \rfloor, \lfloor 0.5 + \sqrt{s} \rfloor$$

So far $\sqrt{x^2 + y^2}$ has been calculated. So to get the magnitude G, the image channels are multiplied by a, using `vgColorMatrix`.

3.3 OpenVG API Wrappers

There are a number of convenience functions in `cvpi_vg_ext.h` that wrap OpenVG functions; they do not offer additional functionality but may be more intuitive and cut down on code size.

The functions `vgConvolve` and `vgSeparableConvolve` scale image values between 0 and 1. It would be more intuitive and convenient if the values ranged from 0 to 255. `vgConvolveNormal` and `vgSeparableConvolveNormal` take the same parameters as `vgConvolve` and `vgSeparableConvolve`, but the bias parameter is divided by 255 before being passed to the wrapped function.

Another problem with `vgConvolve` and `vgSeparableConvolve` is that the pixel being convolved over is in the lower left hand corner of the kernel instead of in the center. The functions `vgConvolveNoShift`, `vgSeparableConvolveNoShift`, `vgConvolveNormalNoShift`, and `vgSeparableConvolveNormalNoShift` move the convolved pixel to the center of the kernel by shifting in the y up by $\lfloor \frac{h}{2} \rfloor$ and shifting in the x right by $\lfloor \frac{w}{2} \rfloor$. If the correction is not done and zeros are given for the shifts, then data points in the resulting image will be shifted towards the image origin.

The function `vgColorMatrix` also scales image values between 0 and 1. `vgColorMatrixNormal` allows the user to specify values between 0 and 255 for the bias.

The function `vgCreateImagePainted` acts like `vgCreateImage` but adds four color parameters, allowing the user to create a new image having a solid color other than the default white.

The function `vgPixelBits` takes a VGImageFormat and returns the number of bits per pixel for that format.

Emacs Calc Extension

In the `extras` directory, there is the file `convolve.el`. It contains Emacs-Lisp Calc code to calculate convolutions, with the convolution kernel origin at the kernel's center pixel. The first argument is the image, the second the kernel, and the third is optional. If the third argument is omitted, then the image will be padded with the same value as the closest edge value. If a number is given, then the input matrix will be padded with the given value. This is useful for investigating how convolutions behave.

```
convolve([[0,0,0,0,0],
          [0,1,1,1,0],
          [0,1,1,1,0],
          [0,1,1,1,0],
          [0,0,0,0,0]],
         [[1,1,1],[1,1,1],[1,1,1]])
```

$$convolve\left(\begin{pmatrix}0&0&0&0&0\\0&1&1&1&0\\0&1&1&1&0\\0&1&1&1&0\\0&0&0&0&0\end{pmatrix},\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}\right)\rightarrow\begin{pmatrix}1&2&3&2&1\\2&4&6&4&2\\3&6&9&6&3\\2&4&6&4&2\\1&2&3&2&1\end{pmatrix}$$

$$convolve\left(\begin{pmatrix}0&0&0&0&0\\0&1&1&1&0\\0&1&1&1&0\\0&1&1&1&0\\0&0&0&0&0\end{pmatrix},\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix},1\right)\rightarrow\begin{pmatrix}6&5&6&5&6\\5&4&6&4&5\\6&6&9&6&6\\5&4&6&4&5\\6&5&6&5&6\end{pmatrix}$$

3.4 YUYV to YUVA

Unlike OpenCV, CVPI does not natively support camera capture; however, this can be achieved using a video capture API, such as Video4Linux (V4L). Most low-cost web cameras output to YUYV, where two horizontal adjacent pixels are represented by 4 bytes. Both pixels share the U and V channels but have different Y channels [12]. The function `cvpi_yuyv2yuva` splits the pixels into separate 4 byte blocks. This function allows the other OpenVG and CVPI functions to manipulate camera input; all other functions require that each pixel have its own separate 4 byte representation. The user must read the raw image data from memory into a VGImage, using CVPI's default pixel format VG_sARGB_8888, and pass it to `cvpi_yuyv2yuva`. The output image will be the same height, but twice the width. The alpha channel will be set to 255.

The input image channel mapping is:

$$\begin{pmatrix}Y2\\U\\Y1\\V\end{pmatrix}\rightarrow\begin{pmatrix}R\\G\\B\\A\end{pmatrix}$$

where, $Y1$ is the yellow channel for the first pixel and $Y2$ is the yellow channel for the second pixel.

Using the function `vgColorMatrixNormal`, the input image is split into two different images with the channels reordered and the alpha channel set to the maximum value, 255.

$$I_1\leftarrow\begin{pmatrix}0&0&0&1\\0&1&0&0\\0&0&1&0\\1&0&0&0\end{pmatrix}\begin{pmatrix}Y2\\U\\Y1\\V\end{pmatrix}+\begin{pmatrix}0\\0\\0\\255\end{pmatrix}=\begin{pmatrix}V\\U\\Y1\\255\end{pmatrix}\rightarrow\begin{pmatrix}R\\G\\B\\A\end{pmatrix}$$

$$I_2\leftarrow\begin{pmatrix}0&0&0&1\\0&1&0&0\\1&0&0&0\\0&0&1&0\end{pmatrix}\begin{pmatrix}Y2\\U\\Y1\\V\end{pmatrix}+\begin{pmatrix}0\\0\\0\\255\end{pmatrix}=\begin{pmatrix}V\\U\\Y2\\255\end{pmatrix}\rightarrow\begin{pmatrix}R\\G\\B\\A\end{pmatrix}$$

The two images, I_1 and I_2, are then combined in vertical, 1-pixel wide strips using `vgCopyImage`. If the image is written to memory, Y will be at array index 0, U at index 1, V at index 2, and A at index 3.

3.5 Color to Black and White

OpenVG can output to several formats including binary black and white, one bit per pixel. `cvpi_image_rgba2bw` converts an image channel to such an image. With `vgLookup`, the `cvpi_binary_array` or `cvpi_binary_array_inverted` is used to convert the desired channel to black and white, and `cvpi_zeros_array` is used to convert the other channels to zero. `vgCopyImage` is used to convert `vgLookup`'s output to a binary image; 255 gets mapped to 1 bits and 0 gets mapped to 0 bits.

3.6 Image Addition and Subtraction

The function `cvpi_image_add` allows the user to add and subtract images of same dimensions. The function is described by the formula:

$$C_{i,j} = s \cdot (a \cdot A_{i,j} + b \cdot B_{i,j}) + t$$

where pixel values range between 0 and 255, s is a floating point value, t is a bias passed to `vgConvolveNormal`, and a and b are integers. A, B, and C are images, with i and j being pixel locations. The function can perform subtraction by setting a or b to a negative value.

If the image height is less than or equal to half the maximum height, then the two images are combined into a single image by copying horizontal strips, 1-pixel in height into another image of twice the height, such that the first row is from the first image, the second row is from the second image, and this is repeated for all following rows. The resulting image is then convolved with `vgConvolveNormal` using a 2×1 kernel, k, where $k_{0,1} = a$ and $k_{0,0} = b$.

OpenVG's convolution formula becomes:

$$I(x,y) = s\left(\sum_{j=0}^{1} k(0, 1-j) \cdot P(x, y+j)\right) + t$$

$$I(x,y) = s(k_{0,1} \cdot P(x,y) + k_{0,0} \cdot P(x, y+1)) + t$$

$$I(x,y) = s(a \cdot P(x,y) + b \cdot P(x, y+1)) + t$$

where P is the input image and I is the output image, with pixel coordinate parameters, the origin being in the lower left corner.

`vgCopyImage` is used to copy the even-indexed rows to another image, and the odd indexed rows are thrown out.

For example, if two images A and B, both 1-pixel wide and two pixels high, are to be added, the following steps are performed.

1. Combining the images:

Index	
3	B10
2	A10
1	B00
0	A00

2. Convolution:

Index	
3	B10+Padding
2	A10+B10
1	B00+A10
0	A00+B00

3. Removing the odd numbered rows:

Index	
2	A10+B10
0	A00+B00

Figure 3.3: Example: Adding two 1×2 images, A and B.

For images larger in height than half the max height, the images are split in half, and their upper and lower halves are added separately. If the image height is odd, then the top rows are added separately from the other rows. That is, the image minus the top row is split in half and added like in the even case, and then the top rows of each image are added together.

1. Two images are created, combining image rows:

Upper Index	
3	B30
2	A30
1	B20
0	A20

Lower Index	
3	B10
2	A10
1	B00
0	A00

2. Convolution:

Upper Index	
3	B30+Padding
2	A30+B30
1	B20+A30
0	A20+B20

Lower Index	
3	B10+Padding
2	A10+B10
1	B00+A10
0	A00+B00

3. Removing the odd numbered rows and combining into a single image:

Index	
2	A30+B30
0	A20+B20
2	A10+B10
0	A00+B00

Figure 3.4: Example: Adding two 1×4 images, A and B, where the maximum allowed image height is 4.

1. Three images are created.

Top-row

Index	
1	B40
0	A40

Upper

Index	
3	B30
2	A30
1	B20
0	A20

Lower

Index	
3	B10
2	A10
1	B00
0	A00

2. Convolution:

Top-row

Index	
1	B40+Padding
0	A40+B40

Upper

Index	
3	B30+Padding
2	A30+B30
1	B20+A30
0	A20+B20

Lower

Index	
3	B10+Padding
2	A10+B10
1	B00+A10
0	A00+B00

Figure 3.5: Example: Adding two 1×5 images, A and B, where the maximum allowed image height is 5. *(cont.)*

3. Removing the odd numbered rows and combining into a single image:

Index	
0	A40+B40
2	A30+B30
0	A20+B20
2	A10+B10
0	A00+B00

Figure 3.5: Example: Adding two 1×5 images, A and B, where the maximum allowed image height is 5.

3.7 Channel Addition and Subtraction

There are three functions for adding pixel channels of the same image together; `cvpi_channel_add`, `cvpi_color_channels_add`, and `cvpi_all_channels_add`. These functions use `vgColorMatrixNormal` to perform the adding.

The function, `cvpi_all_channels_add`, allows the user to specify a scalar for each channel, which channels to output to, and separate biases for each output channel.

$$\begin{pmatrix} R' \\ G' \\ B' \\ A' \end{pmatrix} \leftarrow \begin{pmatrix} S_R & S_G & S_B & S_A \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix} + \begin{pmatrix} B_R \\ 0 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} RS_R + GS_G + BS_B + AS_A + B_R \\ G \\ B \\ A \end{pmatrix}$$

$$\begin{pmatrix} R' \\ G' \\ B' \\ A' \end{pmatrix} \leftarrow \begin{pmatrix} S_R & S_G & S_B & S_A \\ 0 & 1 & 0 & 0 \\ S_R & S_G & S_B & S_A \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix} + \begin{pmatrix} B_R \\ 0 \\ B_B \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} RS_R + GS_G + BS_B + AS_A + B_R \\ G \\ RS_R + GS_G + BS_B + AS_A + B_B \\ A \end{pmatrix}$$

The functionality is similar for `cvpi_color_channels_add`.

$$\begin{pmatrix} R' \\ G' \\ B' \\ A' \end{pmatrix} \leftarrow \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ S_R & S_G & S_B & 0 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ B_A \end{pmatrix} \rightarrow \begin{pmatrix} R \\ G \\ B \\ RS_R + GS_G + BS_B + B_A \end{pmatrix}$$

$$\begin{pmatrix} R' \\ G' \\ B' \\ A' \end{pmatrix} \leftarrow \begin{pmatrix} S_R & S_G & S_B & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix} + \begin{pmatrix} B_R \\ 0 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} RS_R + GS_G + BS_B + B_R \\ G \\ B \\ A \end{pmatrix}$$

With the function `cvpi_channel_add`, the user selects which two channels to add and the output channels. The effective adding formula is the same as that used by `cvpi_image_add`, but is performed by `vgColorMatrixNormal`. If the two input channels are the same,

then their scalars are added together. For example, adding channel R with channel G and outputting to B and A, with B's bias = 0 and A's bias = 1.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 5 \\ 10 \\ 20 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} 1 \\ 5 \\ 6 \\ 7 \end{pmatrix}$$

3.8 Combining Images by Channel

The function `cvpi_image_combine_channelwise` can combine two images, where the user selects the channels to include from one image, and then the complement of the selected channels are included from the other image in the output. The function works by using `vgColorMatrix` to produce two new images where excluded channels are zeroed. Then the two resulting images are added using `cvpi_image_add`. For example, given two one pixel images, with the first two channels from the first image and the second two channels in the second image are selected.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 5 \\ 6 \\ 7 \\ 8 \end{pmatrix} \rightarrow \begin{pmatrix} 1 \\ 2 \\ 7 \\ 8 \end{pmatrix}$$

3.9 Thresholding

CVPI contains five thresholding functions; `cvpi_channel_threshold`, `cvpi_image_threshold`, `cvpi_channel_threshold_sector`, `cvpi_image_threshold_sector`, and `cvpi_image_threshold_adaptive_mean`. Thresholding is used to filter out information within a certain value range.

For `cvpi_channel_threshold` and `cvpi_image_threshold`, the user specifies value ranges, whether to keep or remove pixels whose values are within the range while doing the opposite for those outside the range, and what the replacing value will be for removed pixels. `cvpi_image_threshold` adds the additional option to specify whether channels are dependent or independent. That is, if any channel in a pixel falls outside the threshold range, and if the channels are dependent, then the pixel's channels will be removed. If they are independent, then a pixel channel is kept or removed regardless of whether the other channels were within the threshold range.

The `cvpi_channel_threshold` function works by building an array to pass to `vgLookup`. For channels that are not thresholded, their `vgLookup` functions are set to the identity function array.

The functions `cvpi_channel_threshold_sector` and `cvpi_image_threshold_sector` threshold a channel or image in sectors or blocks. The user must pass in a function that is used to compute a statistic, such as the average, about the sector and the statistic is used as the upper bound for masking or not masking, depending on what the user specifies. The lower bound is zero. The image is divided into `vgChildImage` sectors. If the height or width does

not divide evenly, then their are additional sectors around the upper and right edges, whose width or height are the modulo of the image and sector dimensions.

The function `cvpi_image_threshold_adaptive_mean` adaptively thresholds an image by using `vgConvolveNormalNoShift` to find the local mean for each pixel. The user specifies the $N \times N$ kernel size. Each kernel element has the value 1. The scale is $\left(\frac{1}{N}\right)^2$. If the user wants to keep pixels greater than the convolution results, then the resulting image from the convolution gets subtracted from the original image. Else, if the user wants to keep pixels less than the convolution results, the original image gets subtracted from the image resulting from the convolution. The user can also specify an integer bias, which is passed to `cvpi_image_add` as the bias parameter when finding the difference between the images. Image values are saturated between 0 and 255, so negative values become zero and values above 255 become 255. The image resulting from the difference can be used as a mask on the original image; values that are zero are pixels that are to be removed by the threshold. To create the masking image, values in the difference image are mapped to 0 if they are non-zero, and mapped to 255 if they are zero. If the user wants removed pixels to be shown in white, then the mask is added to the original image; else if the user wants removed pixels to be shown in black, then the mask is subtracted from the original image. If the user wants the image channels to be dependent on each-other; that is, only keep those pixels where all of the channels survive thresholding, then the mean image is split into four separate images, one for each color channel. The resulting images are then AND'ed together (see section 3.13) and then turned into the masking image.

3.10 Masking

Masking one image with another can be accomplished easily with `cvpi_image_add`, by setting the masking pixels to 255 and the non-masking pixels to 0, and then subtracting it from or adding it to the image being masked. However, the user might want to keep masking information inside an image channel, such as the alpha channel. The function, `cvpi_image_mask_channel`, adds or subtracts a channel from the other image channels using `vgColorMatrix`. It is assumed that the masking channel is already set to the correct values.

If alpha is the masking channel and the mask is being added, then:

$$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix} \rightarrow \begin{pmatrix} R+A \\ G+A \\ B+A \\ A \end{pmatrix}$$

If red is the masking channel and the mask is being subtracted, then:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix} \rightarrow \begin{pmatrix} R \\ G-R \\ B-R \\ A-R \end{pmatrix}$$

3.11 Statistics

CVPI contains some statistical operations. Statistical operations, being reduction type operations, cannot really utilize the GPU. However, some mapping functions, such as thresholding, require results from statistical operations.

3.11.1 Average

CVPI provides two averaging functions; `cvpi_image_mean_arithmetic` and `cvpi_image_mean_gpu`. `cvpi_image_mean_arithmetic` computes the arithmetic mean for each channel. This operation is performed entirely on the CPU. The function is not heavily optimized and could be improved on ARM CPUs supporting NEON SIMD. `cvpi_image_mean_gpu` uses convolution in a way similar to `cvpi_image_add` by averaging vertically adjacent pixels until the resulting image is one pixel high. The image is reduced in height by 1 for each iteration. A one-pixel image is produced by averaging horizontally adjacent pixels in the one-pixel high image. The resulting value is not a true statistic but is close to the average. The exact average is not always desired in computer vision and something like it may be good enough. The function has a parameter to limit the number of iterations, returning an image with dimensions: (input width) $\times$ ((height – number of iterations) or (1, if height – number of iterations ≤ 0)).

3.11.2 Channel to Data

The function `vgGetImageSubData`, returns the data for an entire image [5]. `cvpi_channel2data` returns the data for just a single channel. The function works by reordering the channels so that the desired data is in the alpha channel using `vgColorMatrix`. The image is then copied to an image of type VG_A_8. This will discard the non-alpha channels, and then the alpha image is written to memory.

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 0 \\ 0 \\ R \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 0 \\ 0 \\ G \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 0 \\ 0 \\ B \end{pmatrix}$$

3.11.3 Max and Min

CVPI has three functions for determining a channel's minimum and maximum values; `cvpi_channel_max`, `cvpi_channel_min`, and `cvpi_channel_max_min`. All three functions pass the input image and channel specifier to `cvpi_channel2data`, and then use sequential searches.

3.11.4 Histogram and Cumulative Distribution

`cvpi_channel_histogram`, `cvpi_color_channels_histogram`, and `cvpi_image_histogram` create arrays in memory where the index represents an 8-bit color value, and the index entry represent the number of times that value occurs in an image. `cvpi_channel_histogram`'s data array can represent one of four channels: red, green, blue, or alpha. `cvpi_color_channels_histogram`'s data is an array of 768 elements where the first 256 elements represent red values, then next 256 elements represent green values, and the last 256 elements represent blue values. `cvpi_image_histogram` adds another 256 entries, after the blue entries, for the alpha channel values. This way channel data is kept contiguous and the data format is consistent between these functions.

The functions `cvpi_channel_cumulative_distribution`, `cvpi_color_channels_cumulative_distribution`, and `cvpi_image_cumulative_distribution`, take the respective outputs of `cvpi_channel_histogram`, `cvpi_color_channels_histogram`, and `cvpi_image_histogram`, and return the same type of data structures, except that it is a cumulative distribution. This can be used for histogram equalization. Each entry is computed using the formula:

$$c(I) = \frac{1}{N} \sum_{i=0}^{I} h(i)$$

where $h(i)$ is the value computed by the corresponding histogram function [14, p. 95].

These functions are used by `cvpi_channel_histogram_equalization`, `cvpi_color_channels_histogram_equalization`, and `cvpi_image_histogram_equalization` to create histogram equalized images. First, the histogram $h(i)$, and then the cumulative distribution $c(I)$ are computed. The output of the cumulative distribution function is converted to a VGubyte array. If the user wants the array values scaled from 0 to 255, then the $c(I)$ array is re-scaled using the formula

$$h(v) = \text{round}\left(\frac{cdf(v) - cdf_{min}}{(M \cdot N) - cdf_{min}} \cdot 255\right)$$

where M and N are the image dimensions, and cdf_{min} is the smallest positive value in the cumulative distribution [15]. The $c(I)$ array is then passed to `vgLookup`. For channels not being operated on, the identity array is given.

3.12 Histogram Equalization

CVPI has three histogram equalization functions; `cvpi_channel_histogram_equalization`, `cvpi_color_channels_histogram_equalization`, and `cvpi_image_histogram_equalization`;

and their sector counterparts; `cvpi_channel_histogram_equalization_sector`, `cvpi_color_channels_histogram_equalization_sector`, and `cvpi_image_histogram_equalization_sector`. The first three functions use the same histogram and cumulative distribution statistic for the entire image. The latter three functions partition the image, similar to `cvpi_channel_threshold_sector` and `cvpi_image_threshold_sector`, and apply their image-wide counterparts to the sub-images. Adaptive histogram equalization is not implemented.

3.13 Logic Operations

CVPI can perform all binary logical operations: AND, NAND, OR, NOR, XOR, XNOR, complement (\), inverse complement. The complement operation is the same as $A\&!B$.

A	B	AND	OR	$A\backslash B$	XOR	$!(A\backslash B)$	NOR	XNOR	NAND
0	0	0	0	0	0	1	1	1	1
0	1	0	1	0	1	1	0	0	1
1	0	0	1	1	1	0	0	0	1
1	1	1	1	0	0	1	0	1	0

For two inputs, there are 16 possible operations The operations $B\backslash A$ and $!(B\backslash A)$ can be achieved by reversing the inputs. Operations that result in A, B, **1**, or **0** are tautological. The NOT operator is the same as applying the inversion array with `vgLookup`.

1. Allow the user to specify what values true and false outputs will be mapped to; TC and FC. The user can specify, for the input images, whether 0 maps to true and !0 maps to false, or the opposite.
2. Set all of A's pixels to 1's (TRUE) and 0's (FALSE).
3. Set all of B's pixels to 2's (TRUE) and 0's (FALSE).
4. Add the images.
5. Where TC = TRUE and FC = FALSE, use `vgLookup` to:

   AND (intersection) set all 3's to TC and non-3's to FC.

   NAND set all 3's to FC and non-3's to TC.

   OR (union) set all 0's to FC and non-0's to TC.

   NOR set all 0's to TC and non-0's to FC.

   XOR (symmetric difference) set all 1's and 2's to TC and the rest to FC.

   XNOR set all 1's and 2's to FC and the rest to TC.

   COMPLEMENT (relative) set all 1's to TC and the rest to FC.

   INV. COMPLEMENT set all 1's to FC and the rest to TC.

Figure 3.6: Algorithm for performing binary logical operations on images.

3.14 Morphology

CVPI provides five morphology functions; `cvpi_image_dilate`, `cvpi_image_erode`, `cvpi_image_hit_miss`, `cvpi_image_thin`, and `cvpi_image_thicken`. These functions treat non-zero values as true and zero values as false, or the opposite if the user specifies it; so input images are essentially treated as binary images. Like the logic functions, the output is an image consisting of two-value channels, one color being mapped to zero valued elements and another color being mapped to non-zero valued elements.

Assuming that the user chooses non-zero values as true, `cvpi_image_dilate` causes non-zero regions to expand. `cvpi_image_erode` causes non-zero regions to retract; i.e. causes zero regions to expand. `cvpi_image_hit_miss` detects corners. `cvpi_image_thin` and `cvpi_image_thicken` are similar to `cvpi_image_erode` and `cvpi_image_dilate` but rely on `cvpi_image_hit_miss`.

For each of these functions, given an image of zeros and non-zeros, if non-zero values are true, then non-zero values are mapped to 1 and zero is mapped to 0, else the opposite will be done. A 3×3 binary convolution kernel is then mapped over the output [10, Morphology].

3.14.1 Dilation and Erosion

The convolution kernel for dilation and erosion is

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

The input image is converted to "binary", and the kernel is applied [10, Morphology]. True values are mapped to the true color, and false values are mapped to the false color. `cvpi_image_erode` uses `cvpi_image_dilate` with the false color and true color switched; and with non-zero equal to false and zero equal to true if the user specified non-zero equal to true, else the other way around if the user specified zero equal to true. Erosion is essentially dilation with the binary input values flipped.

3.14.2 Thinning, Thickening, and the Hit-and-Miss Transform

The hit-and-miss transform uses four different convolution kernels:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

The input image, after being converted to "binary", is convolved with each of these kernels, returning four different images, and then the outputs are or-ed together [10, Morphology].

For thinning and thickening, a hit-and-miss transform is performed on the input image. In the thinning function, the logical complement function is taken between the original image and the image returned by the hit-and-miss transform. In the thickening function, the logical OR function is taken between the original image and the image returned by the hit-and-miss transform [10, Morphology].

3.15 Data Points

Once the GPU has removed extraneous information from an image, the remaining data likely needs to be processed by the CPU. Often it is the remaining pixel coordinates and not the pixel values that are desired. With the function `cvpi_image_coordinate_table`, CVPI can return a coordinate table for the CPU to operate on. The user can specify whether the origin is in the upper or lower left corner. `cvpi_image_coordinate_table` outputs the input image to memory and uses the CPU to create a table of non-zero pixels. The user can specify which channels to check for non-zero data, unspecified channels are ignored. The function assumes that the image format being used has its origin in the upper left corner, so to switch the origin to the lower left corner, it subtracts the y coordinate from the image height. The function assumes that every pixel could be non-zero, so sufficient space is allocated on the heap to have a coordinate entry for every image pixel. The coordinate data is represented by a union of a 32 bit unsigned integer with a two element array of 16 bit unsigned integers, with the first element representing the x coordinate and the second, the y coordinate. `cvpi_image_coordinate_table` returns a structure consisting of a pointer to an array of coordinates and the length the array. If not all of the allocated memory is used then the memory allocation is reduced to fit the coordinate table data.

3.16 YUVA to RGBA

Currently, there are no image formats that support YUVA, so the image must be converted for human viewing. RGBA is commonly supported. This conversion must be done on the CPU. The function `cvpi_avuy2argb` converts an image in CPU memory to an RGBA image in memory. The function `cvpi_image2argb` is a simple wrapper around `cvpi_avuy2argb` that takes a VGImage, writes it to memory, and calls `cvpi_avuy2argb` on it.

```
C = Y - 16
D = U - 128
E = V - 128

R = clip((298 * C           + 409 * E + 128) >> 8)
G = clip((298 * C - 100 * D - 208 * E + 128) >> 8)
B = clip((298 * C + 516 * D           + 128) >> 8)
```

Figure 3.7: Algorithm for converting YUV to RGB [17].

Chapter 4

Logging

OpenVG has error logging. Errors can be returned using `vgGetError()`. However, it is not possible to create new VG errors, so CVPI uses its own error logging facilities.

CVPI provides logging functions. The user can compile CVPI to write logs synchronously with `fprintf`, asynchronously using POSIX threads, or not at all. The log writing location can be changed with `cvpi_log_file_set` and returned using `cvpi_log_file_get`. The default location is `stderr`. If the user wants to log to a file, then `cvpi_log_file_set` must be used at the beginning of the program and `cvpi_log_file_unset` at the end of the program. CVPI provides seven logging functions; `cvpi_log`, `cvpi_log_1`, `cvpi_log_2`, `cvpi_log_3`, `cvpi_log_4`, `cvpi_log_5`, and `cvpi_log_6`, though `cvpi_log` is not intended for direct use. The six numbered logging functions have corresponding formatting functions; `cvpi_log_format_1`, `cvpi_log_format_2`, `cvpi_log_format_3`, `cvpi_log_format_4`, `cvpi_log_format_5`, and `cvpi_log_format_6`. This is used to standardize log formatting and allow different log strings to be passed to a POSIX thread when performing asynchronous logging. POSIX threads can only take a single argument that is normally a pointer. The logging thread is passed a pointer to a structure containing the log format type, which tells the thread what other structure parameters are to be printed. The structure contains an `fprintf` formatting string, plus the arguments to the string. The log type tells the thread what structure elements to pass to `fprintf`. The logging thread is detached after being created.

Chapter 5

Image Headers

CVPI has functions for writing image headers for Portable Bitmap (PBM), Portable Graymap (PGM), and Bitmap (BMP). The PBM and PGM functions, `cvpi_pbm_header_write` and `cvpi_pgm_header_write`, are wrappers around `fprintf`, which write to the specified file. The BMP functions, `cvpi_bmp_header_alloc`, `cvpi_bmp_header_write`, and `cvpi_bmp_header_alloc_write`, create a bitmap version 4 header structure in heap memory and write it to the specified file.

All bitmap header elements are type `uint32_t`, `int32_t`, `uint16_t`, or `int8_t`, and it is 122 bytes long [9]. CVPI has support for multiple pixel encodings; `RGB_565`, `BGR_565`, `XRGB_8888`, `XBGR_8888`, `ARGB_8888`, `BGRA_8888`, and `RGBA_8888`, where the letters specify the channels and the numbers specify the number of bits per channel. CVPI's `cvpi_avuy2argb` function requires `ARGB_8888`. `CVPI_BMP_DEFAULT` macro specifies the correct header type. Again, this has only been tested on a little endian system and might not produce correct results on a big endian system.

Chapter 6

Video4Linux

CVPI has a set of functions for starting, stopping, and getting data from a camera. These functions can be found in `cvpi_camera_setup.h` and `cvpi_camera_setup.c` in the `tests` directory. `cvpi_camera_setup.c` is based off of the file `capture.c.xml`, which can be found in the Video4Linux system documentation. `cvpi_camera_setup.h` has four functions; `cvpi_camera_create`, `cvpi_camera_start`, `cvpi_camera_read_frame`, and `cvpi_camera_takedown`. Camera capture itself is outside the scope of CVPI, but it is necessary for using CVPI in many computer vision applications.

`cvpi_camera_create` requires the camera's width, height, and format, as well as the number of image buffers to allocate in which to dump raw image data. The function returns a structure on the heap that gets passed to `cvpi_camera_start`, which starts the camera, turning on the camera light if it has one. To act on camera input, the function `cvpi_camera_read_frame`, runs a given function on the captured image. The function `cvpi_camera_takedown`, stops the camera and frees the camera information. Use the command line function `v4l2-ctl --all` to get information about what cameras are on the system, and `v4l2-ctl --list-formats-ext` to get information about supported image formats and dimensions.

Chapter 7

Building

The project currently only builds using SCons. The current SCons build is no more flexible than the original Makefile build; however, the SCons build can be further improved and made more portable. SCons has the full power of Python built in. SCons can also support non-Unix-like systems such as Windows Mobile, something which GNU Autotools cannot do easily [13]. Building with SCons is nearly as easy as building with Make. Unfortunately, SCons has its own shortcomings, not shared by Autotools or CMake [4]. Autotools, CMake, and make cannot be used for the same project; however, SCons can be used with those build systems without interference, so it is likely that CVPI will continue to have two build systems, one SCons and the other Autotools or CMake.

7.1 Compiler Options

CVPI has two levels of logging and three kinds of logging. CVPI must be recompiled to switch between these. The C pre-processor variable `CVPI_CAREFUL` if equal to 1 causes CVPI to check the return values of all OpenVG commands with `vgGetError`. Without it, only commands that necessarily have to allocate GPU memory are checked, though every OpenVG command could potentially cause a memory error [5, p. 29]. If `CVPI_LOGGING` is set to 2, then logging is done asynchronously; if it is set to 1, then logging is done synchronously; else no logging is performed. CVPI could be built in different directories with different logging settings and then the client code linked depending on the user's needs. Little or no logging makes the system harder to debug but reduces or removes the overhead incurred by logging.

CVPI contains some inline assembly with alternative C code. To use the assembly code, set `CVPI_ASSEMBLY` to 1.

There is also code specific to Broadcom's EGL API that is necessary for EGL to run on the Raspberry Pi. To enable the code, set the pre-compiler variable `HAVE_BCM_HOST` to 1.

Chapter 8

Using CVPI

To use EGL, OpenVG, and CVPI, headers must be included in the correct order. Simply including `cvpi.h` should prevent such errors. The EGL and OpenVG headers will be pulled in with it.

There are a few parameters common to many of OpenVG's image functions that for CVPI to work correctly, must be set to certain values. The parameters are `outputLinear`, `outputPremultiplied`, and `allowedQuality`. The parameter `outputLinear` must always be `OUTPUT_LINEAR`. The parameter `outputPremultiplied` must always be `VG_FALSE`. The parameter `allowedQuality` must always be `VG_IMAGE_QUALITY_NONANTIALIASED`. When creating an image for use by a CVPI function, use `CVPI_COLOR_SPACE`.

For creating and destroying a new EGL instance, CVPI provides a simple method of doing so. In the following code in Figure 8.1, `cvpi_egl_surface_pixmap_native_creator` and `cvpi_egl_surface_pixmap_native_destroyer` are names of user provided functions, see Figure 8.2. These functions are system dependent. The code creates an EGL instance able to run CVPI on the Raspberry Pi.

```
cvpi_egl_settings settings = cvpi_egl_settings_create();

settings->surface_pixmap_create_function
         = cvpi_egl_surface_pixmap_native_creator;
settings->surface_pixmap_destroy_function
         = cvpi_egl_surface_pixmap_native_destroyer;
settings->renderable_api = cvpi_egl_renderable_api_openvg;
settings->current_surface_type = cvpi_egl_surface_type_pixmap;
/* additional settings may be required ... */
cvpi_egl_instance instance = cvpi_egl_instance_setup(settings);

/* OpenVG/CVPI code */

cvpi_egl_instance_takedown(instance);
free(settings);
```

Figure 8.1: Creating and taking down an EGL instance using CVPI's interface.

```
/* the surface_pixmap_create_function function pointer type */
typedef EGLNativePixmapType (*pixmap_function_pointer)(cvpi_egl_instance);

/* function for surface_pixmap_create_function function pointer */
EGLNativePixmapType
cvpi_egl_surface_pixmap_native_creator(cvpi_egl_instance egl_instance)
{
  cvpi_egl_settings egl_settings_p = egl_instance->egl_settings;

  EGLint* pixmap_id = malloc(sizeof(*pixmap_id) * 5);
  if(pixmap_id == NULL) {
    return NULL;
  }

  pixmap_id[0] = 0;
  pixmap_id[1] = 0;
  pixmap_id[2] = egl_settings_p->width;
  pixmap_id[3] = egl_settings_p->height;
  pixmap_id[4] = egl_settings_p->pixel_format
               | egl_settings_p->pixel_format_brcm;

  EGLint stride = cvpi_egl_bytes_per_pixel(egl_settings_p->pixel_format)
                                          * egl_settings_p->width;
  eglCreateGlobalImageBRCM(egl_settings_p->width, egl_settings_p->height,
                           pixmap_id[4], NULL, stride, pixmap_id);
  if(!(pixmap_id[0]) && !(pixmap_id[1])) {
    if(pixmap_id != NULL) {
      free(pixmap_id);
    }
    return NULL;
  }
  egl_instance->native_data = pixmap_id;
  return pixmap_id;
}

/* function for surface_pixmap_destroy_function function pointer */
EGLBoolean
cvpi_egl_surface_pixmap_native_destroyer(cvpi_egl_instance egl_instance)
{
  if(egl_instance->native_data != NULL) {
    EGLBoolean retval = EGL_TRUE;
    if(!eglDestroyGlobalImageBRCM(egl_instance->native_data)) {
      fprintf(stderr, "eglDestroyGlobalImageBRCM returned EGL_FALSE.\n");
      retval = EGL_FALSE;
    }
    free(egl_instance->native_data);
    egl_instance->native_data = NULL;
    return retval;
  } else {
    return EGL_FALSE;
  }
}
```

Figure 8.2: Raspberry Pi surface pixmap functions.

Chapter 9

Sample Program

The directory `tests` contains a sample motion detection program called `cvpi_sample_motion_detection`. There are two wrapper shell scripts, `cvpi_sample_motion_detection_data.sh` and `cvpi_sample_motion_detection.sh`. The former program gathers frames per second performance data at different resolutions. The latter program is for user interaction, allowing the user to select the resolution and number of frames to capture.

`cvpi_sample_motion_detection` uses Video4Linux to capture image data from a web camera. The image data returned from the camera is in YUYV format. CVPI converts the image from YUYV to YUVA. The function `cvpi_image_add` is used to subtract the current captured image from the previously captured image, and also subtract the previously captured image from the current captured image. The two resulting images are OR'ed together using `cvpi_image_logical_or` with non-zero channel values set to 0 and zero channel values set to 255. The resulting image is subtracted from the captured image, so that values that did not change between frames are zeroed and new values are kept. This image is then output to a file. The file output is done asynchronously using a separate thread.

Data was gathered using `cvpi_sample_motion_detection_data.sh`, but the data was somewhat scattered. The program was run in an ssh terminal at run-level 3, without the X.Org server running, so there was little else running. Time samples were taken both inside and outside the program's image capture and processing loop, with the lesser of the two samples taken as the frame rate. The average frame rate for all resolutions tested was 17.9 fps with a standard deviation 5.9, and a maximum of frame rate of 30.3 fps and a minimum of 12.1 fps. There was no real correlation between the resolution and the frame rate. The result could be due to the lack of a real time clock, the lack of a real time operating system, or value overflow. Still, I think that the numbers are not entirely inaccurate, and that you could expect process at around 16 to 18fps on average in simple vision applications.

Chapter 10

Bindings

CVPI is written entirely in C so it should not be too difficult to create bindings in other languages. Currently there are bindings being written for Chicken Scheme and C++ as part of CVPI. There are already Java and Python bindings for OpenVG [11]. Chicken Scheme's compiler translates scheme to C code that then gets passed to a C compiler [2]. Making bindings in Chicken Scheme for C functions and variables is quite easy. Scheme has the advantage of a more powerful macro language and can be faster to write.

While C++ can use OpenVG and CVPI directly, but doing so does not allow for use of C++ exception handling and memory management, so class interfaces to OpenVG and CVPI were created.

The incentive for creating bindings as part of the project was due to the size of the test code written in C and the amount of time spent debugging it. Lisp was my natural choice. Lisp's macro system allows code to be concise; common patterns that cannot be turned into functions can be rolled into macros. Chicken Scheme has both Scheme and Lisp-like macros systems. Traditional lisp macros can be written with minor changes using `ir-macro-transformer` and `er-macro-transformer`, the former which implicitly renames variables to prevent variable capture [2, ./man/4/Macros].

Chapter 11

Tests

Currently, there is a single testing program that gets compiled by SCons in the `tests` directory called `cvpi_tests`. The main file, `cvpi_tests.c`, runs a list of test functions. A test returns a CVPI_BOOL value, and the test function name and its return value are printed. Some of the tests check that the output is correct but a number currently only check that the function runs.

I plan to move further testing to Chicken Scheme, and only use C for testing that a function runs. Chicken Scheme's macro system should make for smaller test code and it should be easier to do file IO in Chicken than in C. The Chicken code will certainly introduce new bugs, so it cannot completely replace testing in C.

Chapter 12

Coding Conventions

12.1 Naming

All global CVPI function and variable names start with cvpi. This is meant to serve the same role as a name-space. Following the name-space is the the object type, and then the action. Actions are verbs followed by adverbs. Object types are nouns followed by adjectives. While this scheme does produce odd names from an English speaking perspective, but it creates a logical classification system where the type is primary and action is secondary. Words are separated by underscores. Common CVPI object type names are `channel`, `color_channels`, and `image`. Channel functions only act on one or two channels. `color_channel` functions act on the color channels, and image functions act on all four channels.

Conversion functions are named such that a '2' separates the two types.

OpenVG extensions follow OpenVG's naming convention of 'vg' followed by the rest of the name in camel case.

Constant macros are in all caps. General purpose function macros are written in lower case. Boolean values and tests are in upper case. Enumerations are in lower case. All structure and unions are type-defined or have type-defined pointers to them.

12.2 Code Structure

The overall CVPI project structure follows the GNU coding standards required by GNU Autotools [3].

All functions that allocate data on the heap or in GPU memory have a function local jump point for freeing memory. Every function has its own definition of the macro `TAKEDOWN`, and every function has the variable `BADSTATE`. This is used to implement a kind of function level deconstructor. At the beginning of the function, all heap pointers are initialized to NULL, return values are initialized, and all OpenVG "objects" are set to VG_INVALID_HANDLE. After every OpenVG function, `vgGetError` is called; and after every heap memory allocation, the return value to checked for NULL. If memory is freed outside of TAKEDOWN, the variable is set to NULL or VG_INVALID_HANDLE. If an error value is returned, BADSTATE is set to true and the function jumps into its TAKEDOWN code. If a memory variable is non-NULL, it is freed, and if BADSTATE is true, then the return value is set to

the function's error return value, which is VG_INVALID_HANDLE for functions that return VGImage's, NULL for functions that return heap pointers, or else it is function specific. The pre-compiler condition `CVPI_CAREFUL` allows the user to turn off error checking of OpenVG functions that do not necessarily involve memory allocation. According to the OpenVG specification, every OpenVG function can throw a memory error; however, memory errors are likely to only come from `vgCreateImage` and `vgDestroyImage`, and checking for errors after every OpenVG call can needlessly slow down program execution.

All included headers are surrounded by `ifndef` include guards. All header files, if defined more than once, will print a compiler message.

12.3 Optimization

The code is written with the target platform being the ARMv6 CPU. Where possible, loops count down to zero. When comparing to zero on the ARM CPU, a processor flag is set. Overall, this should result in fewer instructions per cycle [7, p. 180].

The GNU compiler has intrinsic C support for ARM's NEON SIMD extensions; however, ARMv6 does not have NEON instructions but it does have other parallel instructions that GCC does not have intrinsic C support for. ARMv6 chips and higher have instructions for parallel computations on two 16-bit and four 8-bit values in normal registers. If these instructions are intrinsically supported, they would be in the C header `arm_acle.h` [1, p. 14]. GCC does not have this header and the Clang compiler currently has an incomplete version. Some inline assembly instructions were used where GCC was unlikely to optimize correctly, specifically in the `cvpi_avuy2argb` function. Single ARM instructions are capable of multiple actions. Many instructions have a conditional flag that will cause the instruction to be executed conditionally, and one of the inputs passes through a barrel shifter before being passed to the operator [7, p. 37].

The function `cvpi_image_coordinate_table` returns a coordinate table consisting of pairs of unioned `uint16_t`, and OpenVG writes images to memory as a packed block with four 8-bit values per pixel. This allows the user to take advantage of ARMv6's parallel instructions. CVPI defines the union type, `cvpi_pixel`, with an array accessor and channel index aliases `cvpi_pixel_red`, `cvpi_pixel_green`, `cvpi_pixel_blue`, `cvpi_pixel_v`, `cvpi_pixel_u`, `cvpi_pixel_y`, and `cvpi_pixel_alpha`. This allows the user to treat an image in memory as if it were a two dimensional array, with the pixels being one dimension and the channels being the other dimension.

Chapter 13

License

The code is licensed under the LGPLv3 (GNU Lesser General Public License version 3); with the exception of `convolve.el`, which is licensed under the GPLv3 (GNU General Public License version 3), and code in the `m4` directory.

Graphics and data used in testing are licensed under the CC0 (Creative Commons Zero) 1.0 Universal or any later version published by the Creative Commons. All other graphics are licensed under the Creative Commons Attribution 4.0 International License or any later version published by the Creative Commons.

All documents are licensed under the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation.

Bibliography

[1] ARM. *ARM C Language Extensions*, 1.1 edition, November 2013. infocenter.arm.com/help/topic/com.arm.doc.ihi0053a/IHI0053A_acle.pdf.

[2] Chicken wiki, January 2015. wiki.call-cc.org/.

[3] Free Software Foundation, Inc. Gnu coding standards: The release process, December 2014. www.gnu.org/prep/standards/standards.html#Managing-Releases.

[4] Gentoo Foundation, Inc. Scons, January 2015. wiki.gentoo.org/wiki/SCons.

[5] Khronos Group Inc. *OpenVG Specification: Version 1.1*, December 2008. www.khronos.org/registry/vg/specs/openvg-1.1.pdf.

[6] Khronos Group Inc. *Khronos Native Platform Graphics Interface: EGL Version 1.4*, December 2013. www.khronos.org/registry/egl/specs/eglspec.1.4.pdf.

[7] J. A. Langbridge. *Professional Embedded ARM Development*. John Wiley & Sons, Indianapolis, IN, 2014.

[8] H. Lynn. An image-processing robot for robocup junior, August 2014. www.raspberrypi.org/an-image-processing-robot-for-robocup-junior/.

[9] Microsoft. BITMAPV4HEADER structure, 2015. msdn.microsoft.com/en-us/library/windows/desktop/dd183380%28v=vs.85%29.aspx.

[10] Robert, Perkins, Walker, and Wolfart. Hypermedia image processing reference, 2000. homepages.inf.ed.ac.uk/rbf/HIPR2/wksheets.htm.

[11] Raspberry pi videocore apis, December 2013. elinux.org/Raspberry_Pi_VideoCore_APIs.

[12] Schimek, Dirks, and Verkuil. Video for linux two api specification, 2006. v4l.videotechnology.com/dwg/v4l2.html.

[13] Svartalf. Autotools vs. cmake vs. scon, May 2014. stackoverflow.com/questions/4071880/autotools-vs-cmake-vs-scons#answer-18291580.

[14] R. Szeliski. *Computer Vision: Algorithms and Applications*. Springer, London, 2011.

[15] Wikipedia. Histogram equalization — Wikipedia, the free encyclopedia, 2015. [Online; accessed 01-April-2015].

[16] Wikipedia. Sobel operator — Wikipedia, the free encyclopedia, 2015. [Online; accessed 01-April-2015].

[17] Converting between yuv and rgb, April 2010. `msdn.microsoft.com/en-us/library/aa917087.aspx`.