Freescale Semiconductor

Document Number: AN4629

Application Note

Rev. 1, 02/2015

Fast Image Processing with i.MX 6 Series

Computer vision and image processing have been present for at least 50 years in the computer science and artificial intelligence areas. More recently, these topics have surfaced in the embedded world and have brought with them a series of complex problems to be addressed.

This application note is intended as a start-up guide in understanding the basics of image processing techniques and common use cases. This document also describes the i.MX 6 series' capabilities, including GPU, which is used to accelerate image processing.

Contents

1 Introduction

2 Image processing stages

3 Application setup

4 Image processing in the GPU

5 More complex use case: Object detection

6 Implementing the face detector

7 Conclusions and future work

8 References

1 Introduction

Artificial vision, or computer vision, systems include a large series of processes, techniques, tools and mathematical analysis needed in order to emulate what we call vision. Giving computational entities the ability to “see” is a relatively young area (less than 70 years), compared to other more mature science areas such as physics and mathematics.

In general, computer vision can be defined as the capability of a computer entity to input data from its environment (still images or live camera feed) with the purpose of making decisions or processing the input to generate an output. [cast96].

The image processing problem can be decomposed into three levels [rod-sossa2012]:

a) Computational Level: This is related to the task that the computational entity must perform.

b) Algorithmic Level: related to the “how-to”. The concrete series of steps that our system must perform in order to achieve the goal (i.e. a shape or color tracking use case).

c) Hardware Level: The concrete limitations and constraints of the hardware that we will use for our purposes. In our case, this means the Freescale i.MX 6 series 32-bit Multimedia Processor.

In this Application Note, we will be focusing on b) and c). We will take advantage of the Graphics Processing Unit available on the i.MX 6 Series Multimedia Processors, meaning we will share the load of b) between the CPU and the GPU.

1.1 Scope

The goal of this application note is to present the reader with concrete techniques, mathematical background, and code examples that will allow them to perform Image Processing using the Freescale i.MX 6 series reference boards. We will be focusing on:

1. Image acquisition
2. Segmentation techniques
3. Image filter operators
4. Morphologic operations
5. Image binarization
6. Tracking techniques
7. Feature extraction and its uses
8. Face detection techniques

As stated before, some of these algorithms will be implemented in traditional C++ code that will run on the CPU. However, the goal of this application note is to show the reader how to implement several OpenGL ES 2.0 shaders that will run on the GPU and will do the most intense and operation-heavy work in order to accelerate the image processing tasks.

1.2 Prerequisites

This application note assumes that the reader is an experienced i.MX developer and they are familiar with the tools and packages that Freescale provides along with the reference boards. The author also assumes that the reader has intermediate 3D graphics development skills and knowledge, along with OpenGL ES 2.0 programming experience because we will not dig into the details of basic setup.

1.3 Getting started

The recommended setup in order to follow this application note includes the following:

- Freescale i.MX 6 series hardware reference design
- Latest Freescale Linux BSP
- Oneric rootfs distributed with the Linux BSP
- OpenCV 2.0.0

See the Linux BSP user guide and release notes for information on how to build and boot the board with the Ubuntu Oneric rootfs. After that is complete, build the OpenCV 2.0.0 source package in the board (retrieve it from http://sourceforge.net/projects/opencvlibrary/files/).

Be sure to build the kernel with V4L camera support within the Ltib, because this is controlled by videodev support in the Linux Kernel.

Also ensure that you have installed the package libv4l-dev in your host machine.

Lastly, note that use of OpenCV is not mandatory. In this application note, the author decided to use it only for convenience, since it´s very easy to capture feed from the camera and use its IplImage structure for decoding and analysis. However, OpenCV´s advanced math or image processing features are not used because they are all CPU-based. In order to speed up the image processing, avoid using all CPU-based features. Another reason to include OpenCV is that the reader is able to easily implement several of the GPU techniques presented here and compare the results of CPU-based versus GPU-based techniques.

2 Image processing stages

Many paradigms have been stated in the literature over the last years. Here, the author presents one broadly adopted from Gonzalez & Woods text [gonz92] and [jain95]. Below is a combination of both:

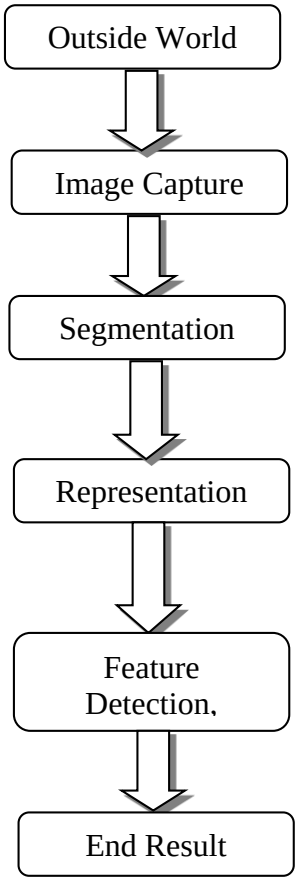

Figure 1: Stages of image processing

Per the figure above, the first step is *Image Capture*. As the name implies, this means obtaining our source image (to be explained in detail in the section on how to obtain it using OpenCV).

Segmentation is a crucial stage in image processing. Segmentation is the division of the source image into subregions that are of interest; this could mean segmentation by color, by size, open regions, closed regions, etc. It is important to note that the sum of all regions equals the source image. The output of segmentation is very important because it will be the input of the next stages. For example, if we want to detect a certain object, we could start by segmenting the source image by the color of the object we are interested in.

Representation is the step in which the segmented image is represented in terms that the system can interpret. This could be a series of enhancements to the segmented image, a binarization of the image (black and white only), and a series of morphological operations (erosion, dilation) used in order to enhance what we need and get rid of the rest. These will be covered in detail in another section of the document.

Feature detection and interpretation is the step when the system needs to know the output of the segmentation. This means that the system has to know if the required object is in the segmented image. For example, if the system searching for a yellow ball, but there is a yellow cube in the segmented

image instead. If it segmented by color, then the cube is still the object of interest, but with feature detection and interpretation the system will know that it is actually a cube, not a ball. Several techniques exist regarding feature extraction, detection and interpretation, and this document will cover several of them; all of these techniques are dependent on the use case.

Lastly, an end result is produced. In the ball example, it is possible to mark a line around the ball perimeter and track its movements along the screen.

In this application note, the author will present different implementation stages and will use OpenGL ES 2.0 shader programs to do the segmentation, representation and feature detection. Also, C++ code will be used to aid the OpenGL ES 2.0 shaders; such as performing morpohological operations like erosion or dilation. And in the end, the system will be able to track objects, apply filters and image operators and detect faces.

3 Application setup

At this point, OpenGL ES 2.0 application should be up and running in the i.MX 6 series board of your choice. For an OpenGL ES 2.0 working sample, download the i.MX 6 series GPU SDK which contains samples and tutorials of simple OpenGL ES 2.0 applications.

Be sure to grab a simple 3D plane example, along with its vertex and fragment shader. These need to be the simplest shaders, but make sure the application is also capable of mapping a 2D texture to the plane.

3.1 OpenCV auxiliary code

The basic idea is to implement the image processing stages that were discussed in the previous section. The stages are as follows:

1. Get the live feed from the USB camera using the V4L support that OpenCV uses. Fill an IplImage structure for each frame to be processed.
2. Create an OpenGL texture from the image data accessible through the IplImage structure and map this texture to a 3D plane in OpenGL ES 2.0.
3. Use the Fragment Shader to perform fast image processing calculations. In this application note we will examine segmentation, binarization, sobel operator, color tracking and face detection.
4. Apply morphologic operations to clean the segmented image (if neccesary).
5. Perform multi-pass renders to chain several image processing shaders to get an end result (i.e. the output of one shader is the input to another one and so on).

First, import the relevant OpenCV headers:

```
#include "opencv/cv.h"
#include "opencv/cxcore.h"
#include "opencv/cvaux.h"
#include "opencv/highui.h"
```

Next, set a texture size. The sized used for this example is 320x240, but this can easily be changed to 640x480.

```
#define TEXTURE_W 320
#define TEXTURE_H 240
```

After this, it is necessary to create an OpenCV capture device to enable the V4L camera to get the live input:

```
CvCapture *capture;
capture = cvCreateCameraCapture (0);
cvSetCaptureProperty (capture, CV_CAP_PROP_FRAME_WIDTH,  TEXTURE_W);
cvSetCaptureProperty (capture, CV_CAP_PROP_FRAME_HEIGHT, TEXTURE_H);
```

When it is complete, close the stream as shown below:

```
cvReleaseCapture(&capture)
```

OpenCV has a very convenient structure used for storing pixel arrays (images) called IplImage:

```
IplImage *bgr_img1;
IplImage *frame1;
bgr_img1 = cvCreateImage (cvSize (TEXTURE_W, TEXTURE_H), 8, 4);
```

OpenCV has another convenient function for capturing a frame from the camera and storing it in a IplImage structure:

```
frame1 = cvQueryFrame(capture);
```

In order to divide the camera capture process from the actual image processing and rendering, the use of a separate thread to exclusively handle the camera capture is recommended:

```
#include <pthread.h>
pthread_t camera_thread1;
pthread_create (&camera_thread1, NULL, UpdateTextureFromCamera1,(void
*)&thread_id);
```

The UpdateTextureFromCamera() function should be something like this:

```
void *UpdateTextureFromCamera (void *ptr)
{
      while(1)
      {
            frame1 = cvQueryFrame(capture);
            cvCvtColor(window->frame, bgr_img1, CV_BGR2BGRA);
      }
       return NULL;
}
```

Finally, the rendering loop should be something like this:

```
while (! window->Kbhit ())
      {
            tt = (double)cvGetTickCount();
            Render ();
            tt = (double)cvGetTickCount() - tt;
            value = tt/(cvGetTickFrequency()*1000.);
            printf( "\ntime = %gms --- %.2lf FPS", value, 1000.0 /
value);
        }
```

3.2 Map the camera live feed to an OpenGL texture

As expected, a Render() function is needed to be called for every frame. The basic details of OpenGL or EGL setup are out of the scope of this application note. For this kind of information, consult the i.MX 6 series OpenGL 2.0 examples and tutorials.

Here is a simple rendering loop that contains the main concept of what needs to be implemented:

```
void Render (void)

{

      static float x_angle = 0;

      static float y_angle = 0;

      static float z_angle = 0;


      static float x_angle2 = 0;

      static float y_angle2 = 0;

      static float z_angle2 = 0;


      // Clear the colorbuffer and depth-buffer

      glClearColor (0.0f, 0.0f, 0.0f, 0.0f);

      glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);


      x_angle += 0.5;

      y_angle += 0.5;

      z_angle += 1.5;

      plane1->PlaneSetTex(bgr_img1);
```

```
        plane1->PlaneMove(PLANE_Z_AXIS, -2500);
        plane1->PlaneMove(PLANE_Y_AXIS, 200);
        plane1->PlaneMove(PLANE_X_AXIS, -200);
        plane1->PlaneRotate(PLANE_Z_AXIS, z_angle);
        //plane1->PlaneRotate(PLANE_Y_AXIS, y_angle);
        //plane1->PlaneRotate(PLANE_X_AXIS, x_angle);
        plane1->PlaneDraw();

        // Swap Buffers.
        // Brings to the native display the current render surface.
        eglSwapBuffers (window->egldisplay, window->eglsurface);
        assert (eglGetError () == EGL_SUCCESS);
        return;
}
```

In order to bind a texture to it, create a texture handler:

```
GLuint _texture;
```

Next, create the texture and set its parameters

```
void GLCVPlane::PlaneCreateTex(int texture_w, int texture_h, int ch)
{
        // textures
        _texture_w = texture_w;
        _texture_h = texture_h;
```

```
      _texture_data=cvCreateImage(cvSize(_texture_w, _texture_h), 8,
ch);

      _texture=GenTextures();

      glTexParameterf (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);

      glTexParameterf (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);

      return;

}
```

The GenTextures() function generates the OpenGL textures, shown below:

```
void GLCVPlane::PlaneSetTex (IplImage *texture_data)

{

      cvCvtColor (texture_data, _texture_data, CV_BGR2RGB);

      glBindTexture(GL_TEXTURE_2D, _texture);

      glTexImage2D (GL_TEXTURE_2D, 0, GL_RGB, _texture_w, _texture_h,
0, GL_RGB, GL_UNSIGNED_BYTE, _texture_data->imageData);


}
```

The PlaneSetTex() function is used to map the texture to the plane:

```
void GLCVPlane::PlaneSetTex (IplImage *texture_data)

{

      cvCvtColor (texture_data, _texture_data, CV_BGR2RGB);

      glBindTexture(GL_TEXTURE_2D, _texture);

      glTexImage2D (GL_TEXTURE_2D, 0, GL_RGB, _texture_w, _texture_h,
0, GL_RGB, GL_UNSIGNED_BYTE, _texture_data->imageData);


}
```

At this point, the OpenGL texture is ready to be used as a sample in the fragment shader and it is also ready to be mapped to a 3D plane.

The overall workflow is presented in the following diagram:

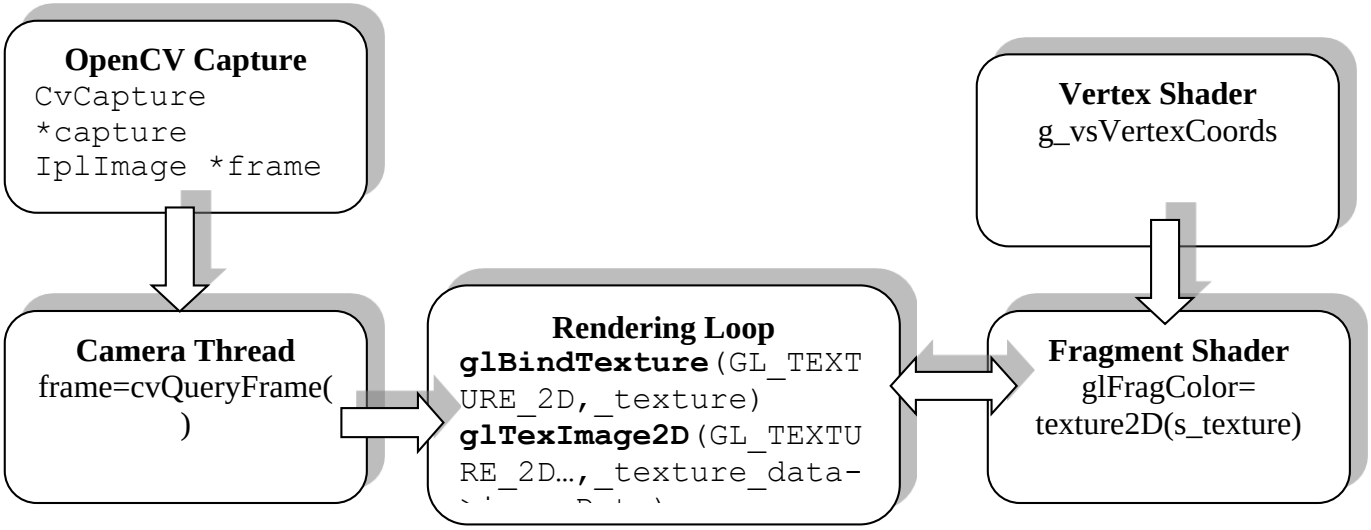

Figure 2: Workflow

The details of how to draw a plane depend strictly on the use case and initialization, and will not be covered here. The details of compiling, linking, and using the shader programs will also not be covered.

4 Image processing in the GPU

As mentioned before, OpenGL ES 2.0 contains a basic application that maps the live camera feed to a 3D textured plane.

4.1 Basic shader setup

Start from a shader base. Below are the most basic shaders that are needed in order to perform image processing.

The vertex shader is a general-purpose programmable method for operating on vertices. Vertex shaders can be used for traditional vertex-based operations such as transforming the position by a matrix, computing the lighting equations to generate a per-vertex color, and generating or transforming texture coordinates.

The vertex shader has the following inputs:

Attributes: Consist of per-vertex data using vertex arrays

Uniforms: Constant data used by the vertex shader

Shader program: The actual source code of the shader. It contains the instructions and operations that will be performed on the vertex.

Consider the following code as a baseline for what comes next. Even though we will not be doing anything special within the vertex shader, we must use it in our image processing application nonetheless.

```
uniform   mat4 g_matModelView;

    uniform   mat4 g_matProj;

    attribute vec4 g_vPosition;

    attribute vec3 g_vColor;

    attribute vec2 g_vTexCoord;

    varying   vec3 g_vVSColor;

    varying   vec2 g_vVSTexCoord;

    void main()

    {

        vec4 vPositionES = g_matModelView * g_vPosition;

        gl_Position  = g_matProj * vPositionES;

        g_vVSColor = g_vColor;

        g_vVSTexCoord = g_vTexCoord;

    }
```

The fragment shader is a general-purpose method for interacting with fragments. The fragment shader program is executed for each fragment in the rasterization stage. It has the following inputs:

Varying variables: Outputs of the vertex shader that are generated by the rasterization unit for each fragment using interpolation.

Uniforms: Constant data used by the fragment shader.

Samplers: A specific type of uniform that represents textures used by the fragment shader.

Shader program: Fragment shader program source code or executable that describes the operations that will be performed on the fragment.

The fragment shader can either discard the fragment or generate a color value referred to as *gl_FragColor*. The color, depth, stencil, and screen coordinate location (x, y) of screen coordinates generated by the rasterization stage become inputs to the per-fragment operations stage of the pipeline.

After the fragment shader, the next stage is per-fragment operations. A fragment produced by rasterization with (x,y) screen coordinates can only modify the pixel at location (x,y) in the framebuffer (see figure below).

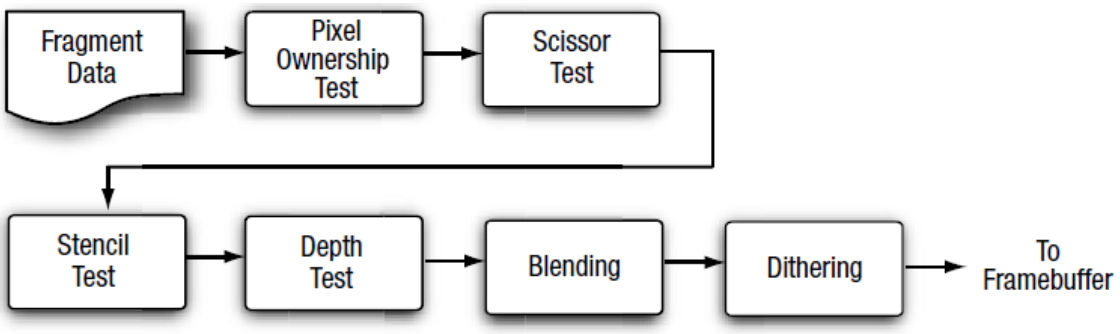

Figure 3: Per-fragment operations

At the end of the per-fragment stage, either the fragment is rejected or a fragment color, depth, or stencil value is written to the framebuffer at location (x,y) of the screen. The fragment color, depth, and stencil values are written depending on whether the appropriate write masks were enabled.

This is why the fragment shader is needed to do GPU-based image processing; the most simple fragment shader code needed to start is as follows:

```
#ifdef GL_FRAGMENT_PRECISION_HIGH
   precision highp float;
#else
precision mediump float;
#endif
uniform sampler2D s_texture;
varying   vec3      g_vVSColor;
varying   vec2 g_vVSTexCoord;
void main()
{
    gl_FragColor = texture2D(s_texture,g_vVSTexCoord);
}
```

At this point, it is very important to have the correct output from the application. You should be able to see the live camera input feed as a texture in the 3D plane.

Figure 4: Live camera input

4.2 Binary Image

One of the oldest and most frequently used techniques for image segmentation is the binary filter operation. The function f(x,y) represents a pixel in the screen:

$$
\begin{aligned}
&\text{if } f(x,y) < threshold \\
&\qquad f(x,y) = 0 \\
&\text{if } f(x,y) > threshold \\
&\qquad f(x,y) = 1
\end{aligned}
$$

This means that we will have only two values, 0 or 1, as an output, giving a black and white only output.

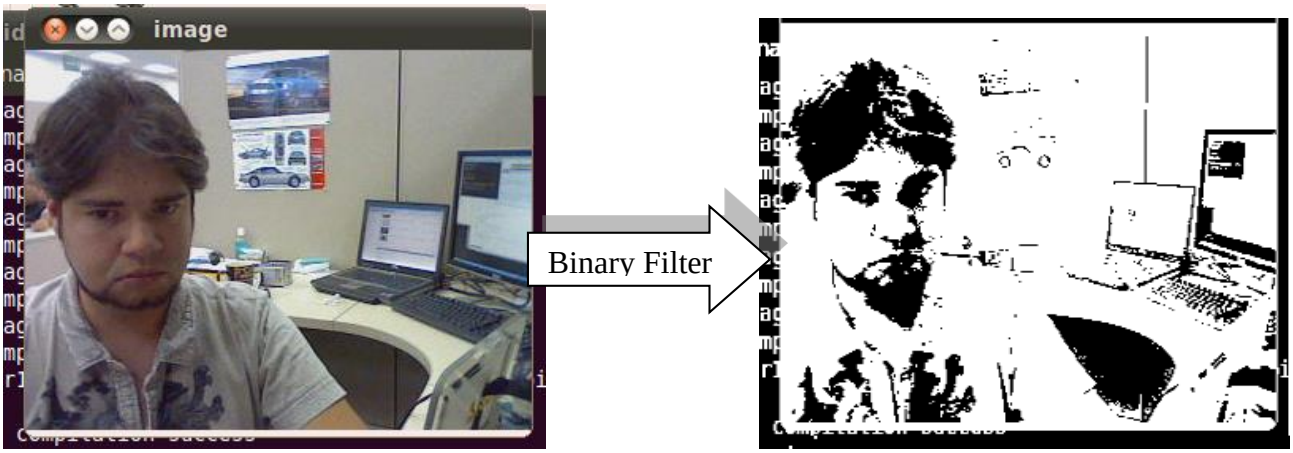

Figure 5: Output from binary filter

This can be calculated by a luminance value of the pixel using the so-called log-average luminance of an image [huss2010]. The log-average luminance is calculated by finding the geometric mean of the luminance values of all pixels. In a grayscale image, the luminance value is the pixel value. In a color image, the luminance value is found by a weighted sum:

$$Luminance = 0.27*red + 0.67*green + 0.06*blue$$

This can be observed in the following shader code:

```
const char* g_strRGBtoBlackWhiteShader =
    #ifdef GL_FRAGMENT_PRECISION_HIGH
       precision highp float;
    #else
       precision mediump float;
    #endif
      varying   vec2 g_vVSTexCoord;
      uniform sampler2D s_texture;
      uniform float threshold;


      void main() {
        vec3 current_Color = texture2D(s_texture,g_vVSTexCoord).xyz;
        float luminance = dot (vec3(0.114,0.587,0.0.299),current_Color);
```

```
        if(luminance>threshold)
            gl_FragColor = vec4(1.0);
          else
            gl_FragColor = vec4(0.0);
       }
```

The reader may have noticed the switched coefficients. This is because the camera live feed comes in BGR format, not the typical RGB, therefore the coefficients must the switched accordingly. Failure to do so may lead to unwanted results. This applies in all cases, except if the developer has not switched the values already.

The last step is to actually define a threshold. There are several ways to obtain a valid threshold: One is to test different thresholds until a satisfactory result is reached.

On the other side there are formal methods. Below is presented a very useful thresholding technique called the Otsu thresholding which uses the image histogram to calculate the optimum thresholding value [otsu1979].

It is important to note, however, that this thresholding algorithm must run on the CPU. Because a histogram is needed and the shader operates at a pixel value only, the histogram and Otsu thresholding have to be calculated on the CPU side. As stated previously, the overall goal is to share the load between the CPU and the GPU. That being said, the histogram and threshold value should be calculated in normal C code. The good news is that they are both very cheap in terms of CPU cycles with linear complexity.

This means that the threshold must be calculated *before* the image reaches the shader code, since the value is passed via a ‘uniform’ to the shader code.

As mentioned before, the source image data resides in the IplImage structure, meaning operations should be performed on that structure and its array of pixel data.

It is out of the scope of this application note to explain the thresholding techniques. For now, a working value will be provided. If the reader would like to dig into more detail they can consult “A Survey of Thresholding Techniques” [sahoo1988], and [otsu1979].

A good threshold starting value would be between 0.2f and 0.3f.

4.3 Border detection by Sobel Operator

Sobel is a very common filter, since it is used as a foundation for many complex Image Processing processes, particularly in edge detection algorithms. The Sobel operator is based on convolutions. The convolution is made of a particular mask, often called a kernel (on common terms, usually a 3x3 matrix).

The Sobel operator calculates the gradient of the image at each pixel. This determines how a pixel differs from the pixels surrounding it, meaning how it increases or decreases (darker to brighter values).

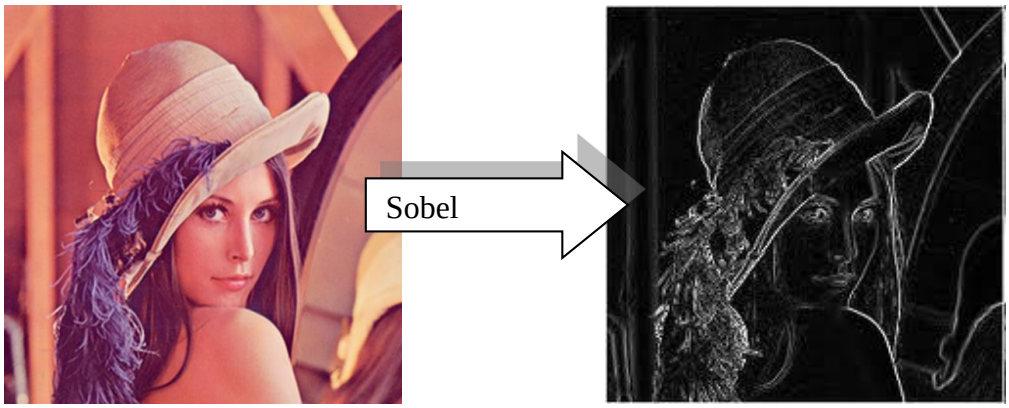

Figure 6: Image gradient using Sobel operator

When working with images, a vector gradient indicates the maximum variation of *f* in pixel *(x,y)*. These variations come from the differences in horizontal and vertical changes. Therefore, two masks are needed: one for vertical change, and the other for horizontal change.The two masks are presented as follows:

-1	0	1
-2	0	2
-1	0	1

-1	-2	-1
0	0	0
1	2	1

Since these masks do not use the central value (both are 0), they are less sensitive to isolated noise and more sensitive to linear noise. These masks need to be convoluted around every pixel of the source image, and the output value is a sum of the masks. Next, the shader and its parts are presented in detail.

It is necessary to get the UV coordinates of the sample image from the vertex shader:

```
const char* plane_sobel_filter_shader_src =

#ifdef GL_FRAGMENT_PRECISION_HIGH

precision highp float;

#else

precision mediump float;

#endif

varying   vec2 g_vVSTexCoord;
```

```
uniform sampler2D s_texture;
```

Next is the kernel, or the mask, mentioned below. Note that these are the horizontal and vertical masks:

```
mat3 kernel1 = mat3 (-1.0, -2.0, -1.0,
                      0.0, 0.0, 0.0,
                     1.0, 2.0, 1.0);
mat3 kernel2 = mat3 (-2.0, 0, 2,
                      0.0, 0.0, 0.0,
                       -1.0, 0.0, 1.0);
```

A new function within the shader is needed to convert to grayscale, since only grayscale information is needed for the Sobel operator. To convert to grayscale, it only needs to take the average of the 3 colors.

```
float toGrayscale(vec3 source) {
 float average = (source.x+source.y+source.z)/3.0;
 return average;
  }
```

The following shader code shows the crucial part: performing the convolutions. It is important to note that according to the OpenGL ES 2.0 spec, no recursion or dynamic indexing is supported, so the matrix operation cannot be automated. It is necessary to define vectors and multiply them in order to get the results needed.

```
float doConvolution(mat3 kernel) {
 float sum = 0.0;
 float current_pixelColor =
           toGrayscale(texture2D(s_texture,g_vVSTexCoord).xyz);
 float xOffset = float(1)/1024.0;
 float yOffset = float(1)/768.0;
 float new_pixel00 =
```

```
          toGrayscale(texture2D(s_texture, vec2(g_vVSTexCoord.x-
                          xOffset,g_vVSTexCoord.y-yOffset)).xyz);
 float new_pixel01 =
toGrayscale(texture2D(s_texture,
              vec2(g_vVSTexCoord.x,g_vVSTexCoord.y-yOffset)).xyz);

float new_pixel02 = toGrayscale(texture2D(s_texture,
       vec2(g_vVSTexCoord.x+xOffset,g_vVSTexCoord.y-
                                              yOffset)).xyz);

 vec3 pixelRow0 = vec3(new_pixel00,new_pixel01,new_pixel02);
 float new_pixel10 = toGrayscale(texture2D(s_texture,
                   vec2(g_vVSTexCoord.x-xOffset,g_vVSTexCoord.y)).xyz);

 float new_pixel11 = toGrayscale(texture2D(s_texture,
                       vec2(g_vVSTexCoord.x,g_vVSTexCoord.y)).xyz);

 float new_pixel12 = toGrayscale(texture2D(s_texture,
                 vec2(g_vVSTexCoord.x+xOffset,g_vVSTexCoord.y)).xyz);
 vec3 pixelRow1 = vec3(new_pixel10,new_pixel11,new_pixel12);

 float new_pixel20 = toGrayscale(texture2D(s_texture,
           vec2(g_vVSTexCoord.x-xOffset,g_vVSTexCoord.y+yOffset)).xyz);

float new_pixel21 = toGrayscale(texture2D(s_texture,
                    vec2(g_vVSTexCoord.x,g_vVSTexCoord.y+yOffset)).xyz);

float new_pixel22 = toGrayscale(texture2D(s_texture,
            vec2(g_vVSTexCoord.x+xOffset,g_vVSTexCoord.y+yOffset)).xyz);
```

```
vec3 pixelRow2 = vec3(new_pixel20,new_pixel21,new_pixel22);

vec3 mult1 = (kernel[0]*pixelRow0);

vec3 mult2 = (kernel[1]*pixelRow1);

vec3 mult3 = (kernel[2]*pixelRow2);

sum= mult1.x+mult1.y+mult1.z+mult2.x+mult2.y+mult2.z+mult3.x+

          mult3.y+mult3.z;\n"

   return sum;
```

If the reader observes the last part of the function, it can be seen that all the multiplication values are added to a sum. In this sum, the pixel variations regarding its neighbors emerge.

In the last part of this long shader all these functions will be used. It is important to notice that the convolution needs to be applied horizontally and vertically:

```
void main() {

  float horizontalSum = 0.0;

  float verticalSum = 0.0;

  float averageSum = 0.0;

  horizontalSum = doConvolution(kernel1);

  verticalSum = doConvolution(kernel2);

    if( (verticalSum > 0.2)|| (horizontalSum >0.2)||

                  (verticalSum < -0.2)|| (horizontalSum <-0.2))


            averageSum = 0.0;

    else

            averageSum = 1.0;

   gl_FragColor = vec4(averageSum,averageSum,averageSum,1.0);

      }
```

Again values are compared against a threshold (0.2 in this case). This threshold will come by empirical methods rather than formal methods because simple variables like camera type and ambient room illumination can make a noticeable impact on the obtained results.

4.4 Color and object tracking

The techniques presented up until this point receive an input image (from the camera) and apply a pixel based filter or operation in a shader, but what about a more complex use case? Let us assume that we

want to track an object, a ball, for example. First the ball needs to be identified, then its position determined, and lastly it must be tracked along the screen. The techniques and methods that follow will focus on this topic.

The overall process is as follows:

1. Obtain the source image.
2. Perform segmentation at shader level.
3. Render the segmented image to an offscreen buffer.
4. Read the offscreen segmented image and calculate its centroid.
5. Switch to another shader program that takes the centroid as input (via uniform).
6. Render a blue dot over the centroid of the original source image, or mark the contour of the ball.

For this use case, different shader programs will be needed: a shader program that contains a vertex shader and a segmentation (fragment) shader, and a second program that includes a vertex shader and the tracking (fragment) shader.

In the init code, there should be something like the following:

```
g_hShaderProgram = glCreateProgram();
glAttachShader( g_hShaderProgram, hVertexShader );
glAttachShader( g_hShaderProgram, hSegmentShader );
g_hVideoShaderProgram = glCreateProgram();
glAttachShader( g_hShaderProgram, hVertexShader );
glAttachShader( g_hShaderProgram, hTrackingShader );
```

`g_hShaderProgram` Will contain the vertex shader and the segmentation shader,
`g_hVideoShaderProgram` Will contain the same basic vertex shader and the tracking shader.

The fragment shader `hSegmentShader` will perform color segmentation (as its name suggests). This shader takes an input color vec3 as a uniform, and that color will be the segmentation criteria. Color image segmentation is very useful to isolate areas of interest in the whole scene. This means that only pixels that have a linear close that value to the input color will be allowed to “pass”, while all the others will be black ones.

There are a variety of techniques to achieve color segmentation, such as stochastic based approaches [belongie98] and morphological watershed based region growing [rod-sossa2012]. Other techniques are presented in [deng-manjunath].

A very popular method for sementation is the HSV Color Space. It is represented by a hexacone, where the central vertical axis represents the Intensity. This is more intuitive for the user. This model was developed to try to ease the perception according to how the human eye sees color.

The hue defines the color itself. It is defined as an angle in the range [2, 2PI], or 0 to 360 degrees, beginning and ending with red and going through green, blue, and all the intermediary colors.

Saturation indicates the degree to which the hue differs from neutral gray. Its value can be from 0, meaning no color saturation, to 1, which is the maximum saturation of a given hue at a given illumination.

Intensity indicates the illumination level, also called value (HSV). It varies from 0 meaning pure black (no light), to 1, full illumination (white color).

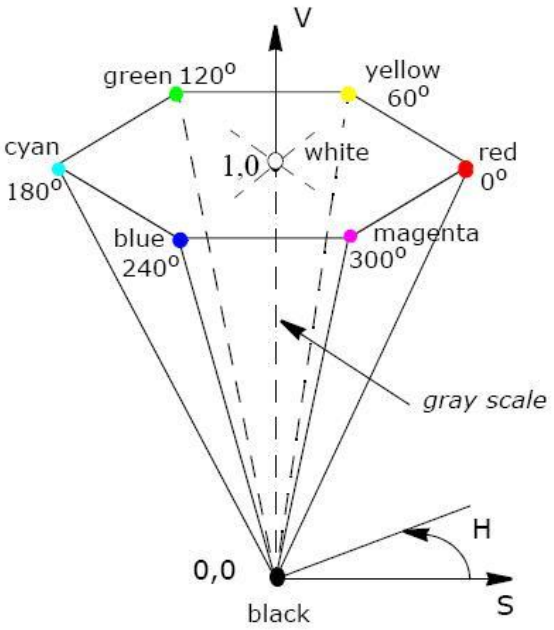

Figure 7: HSV hexacone

The HSV hexacone defines the subset of the HSV space with RGB values. The ***value*** V is the vertical axis, and the vertex V=0 corresponds to black color. Similarly, a color solid or 3D-representation, of the HLS model is a double hexcone with *lightness* as the axis, and the vertex of the second hexacone corresponding to white [sural-qian].

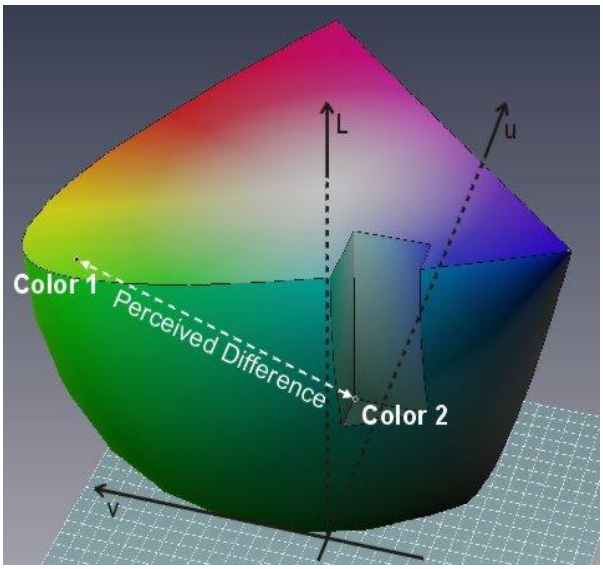

Figure 8: 3D HLS model

For this use case, the author suggests the use of a simple method since the idea is to track homogeneous colors (objects, skin color), not complex textured objects. That being said, the suggested method is an adaptation of the mean-shift color technique presented by D.Comaniciu and P.Meer in [comaniciu-meer]. The idea is to treat the image as a set of RGB vectors with a homogeneity criterion of color similarity. Images are stored in RGB (or BGR in this case, since that is the input from the camera), but to ensure the isotropy of the feature space, a uniform color space with the perceived color differences measured by Euclidean distances should be used. The 'L', 'u' v space is chosen, whose coordinates are related to the RGB values by nonlinear transformations. The chromatic information is carried by 'u' and 'v' while the lightness coordinate 'L' can be the relative brightness.

To calculate this within a fragment shader, a distance between the current color and the reference color (input) can be compared against a threshold (set empirically) and if the distance is larger than the threshold the pixel will be black.

With the above explanation, the fragment shader code below should be straightforward.

```
const char* g_strTrackingShader =
 #ifdef GL_FRAGMENT_PRECISION_HIGH
   precision highp float;
 #else
   precision mediump float;
endif

 uniform sampler2D s_texture;
 varying   vec4      g_vVSColor;
 uniform vec3 g_vInputColor;
 varying   vec2 g_vVSTexCoord;
 vec4 inputColor = vec4(g_vInputColor,1.0);
 float threshold = 0.10;
 vec3 normalizeColor(vec3 color){
```

```
    return color/ max(dot(color,vec3(1.0/3.0)),0.3);
 }
 vec4 BgrToRgb(vec4 source){
   return vec4(source.b,source.g,source.r,1.0);
 }
 vec4 maskPixel(vec4 pixelColor, vec4 maskColor) {
    float d;
 vec4 calculatedColor;
 //distance between current color and reference color
   d =
distance(normalizeColor(pixelColor.rgb),normalizeColor(maskColor.rgb));
   //if color difference is larger than threshold return black
    calculatedColor = (d > threshold) ? vec4(0.0) : vec4(1.0);
  return calculatedColor;
  }
  vec4 coordinateMask(vec4 maskColor,vec2 coordinate) {
  return maskColor * vec4(coordinate, vec2(1.0));
}
void main()
 {
  vec4 pixelColor = BgrToRgb(texture2D(s_texture,g_vVSTexCoord));
  vec4  maskedColor = maskPixel(pixelColor,inputColor);
  gl_FragColor = maskedColor;

}
```

The obtained results should be something like this:

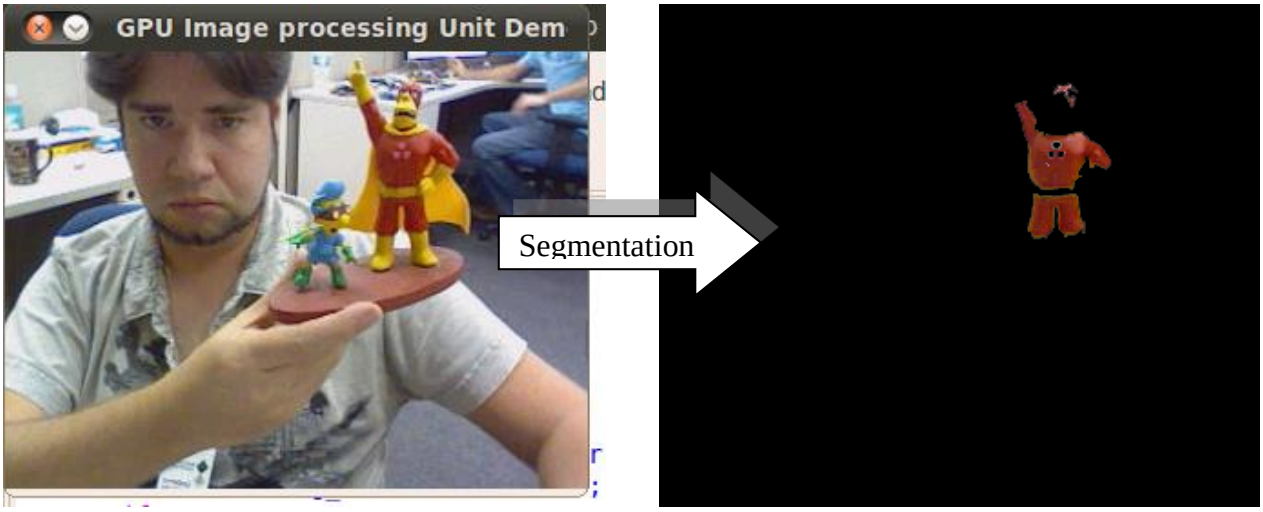

Figure 9: Fragment shader

At this point, only the relevant pixel data is rendered (red-ish pixels), but for this section, as explained before, it is necessary to render to an offscreen buffer, perform some calculations (get the centroid), and draw a circle (overlay) with a second shader. This is a longer process, but it is still quite straightforward.

First the offscreen buffer must be created. See the following code snippets:

```
FrameBufferObject *g_pOffscreenFBO = NULL;
//later in init code

CreateFBO(320,240,GL_RGB,GL_UNSIGNED_SHORT_5_6_5,&g_pOffscreenFBO));
```

Functions to create and destroy the offscreen frame buffer objects:

```
bool CreateFBO(GLuint nWidth, GLuint nHeight, GLuint nFormat, GLuint nType,
FrameBufferObject** ppFBO)
{
  (*ppFBO) = new FrameBufferObject;
  (*ppFBO)->m_nWidth = nWidth;
   (*ppFBO)->m_nHeight = nHeight;
  //create an offscreen texture
  glGenTextures(1, &(*ppFBO)->m_hTexture);
  glBindTexture(GL_TEXTURE_2D,(*ppFBO)->m_hTexture);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP_TO_EDGE);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP_TO_EDGE);
  glTexImage2D(GL_TEXTURE_2D,0,nFormat, nWidth, nHeight, 0, nFormat,
nType,NULL);

//create the depth buffer
  glGenTextures(1, &(*ppFBO)->m_hRenderBuffer);
  glBindRenderbuffer(GL_RENDERBUFFER, (*ppFBO)->m_hRenderBuffer);
  glRenderbufferStorage(GL_RENDERBUFFER,GL_DEPTH_COMPONENT16,nWidth,nHeight);

//create the color buffer
  glGenFramebuffers(1,&(*ppFBO)->m_hFrameBuffer);
  glBindFramebuffer(GL_FRAMEBUFFER,(*ppFBO)->m_hFrameBuffer);
  glFramebufferTexture2D(GL_FRAMEBUFFER,GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
(*ppFBO)->m_hTexture,0);

glFramebufferRenderbuffer(GL_FRAMEBUFFER,GL_DEPTH_ATTACHMENT,GL_RENDERBUFFER,
(*ppFBO)->m_hRenderBuffer);

  if(GL_FRAMEBUFFER_COMPLETE !=glCheckFramebufferStatus(GL_FRAMEBUFFER))
      return FALSE
  glBindFramebuffer(GL_FRAMEBUFFER,0);
  glBindRenderbuffer(GL_RENDERBUFFER,0);

  return TRUE;

}
```

Now the function to destroy the FBO, which is much simpler:

```
void DestroyFBO(FrameBufferObject* pFBO){
 glDeleteTextures(1, &pFBO->m_hTexture);
 glDeleteFramebuffers(1, &pFBO->m_hFrameBuffer);
 glDeleteRenderbuffers(1, &pFBO->m_hRenderBuffer);
 delete pFBO;

}
```

EndFBO() binds to GL_FRAMEBUFFER the frame buffer 0, and creates a viewport with it:

```
void EndFBO(FrameBufferObject* pFBO)
{
  glBindFramebuffer(GL_FRAMEBUFFER,0);
  glViewport(0,0,g_nWindowWidth,g_nWindowHeight);

}
```

Now, presenting the render loop, this is the first part of the render function. Some aspects are important to note here:

1. BeginFBO() is the function that prepares the previously created FBO to render write to it.
2. glClear() clears the recently prepared FBO
3. glUseProgram() calls g_hShaderProgram that includes our segmentation shader (more details to come)
4. plane1->PlaneDraw() draw your textured plane to the screen
5. glReadPixels(): at this point, the end result of the previous operations performed by the fragment shader (segmentation) has been drawn to the FBO, the result is passed to rawPositionPixels, which is defined as:

```
rawPositionPixels =
(GLubyte*)malloc(WIDTH*HEIGHT*4*sizeof(GLubyte));
```

6. calculateCenter(), this function takes the recently read buffer and calculates the segmented image centroid, and stores it
7. Call EndFBO() to return to framebuffer 0 as main rendering target
8. Use glClear() to clear the color buffer

9. glUseProgram() with the second shader in this case g_hVideoShaderProgram
10. Set your texture handler with the input image again. At this point, the data stored in frame1 can be used as before.
11. Create the texture handle angle again and bind the recently created texture to it.
12. Set the uniform locations, pass the tracking points obtained from the calculateCenter() function.

```
//render into the primary backbuffer with a blue background
BeginFBO(g_pOffscreenFBO);
// Clear the colorbuffer and depth-buffer
glClearColor( 0.0f, 0.0f, 0.0f, 0.0f );
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
glActiveTexture(GL_TEXTURE0);
// Set the shader program
    glUseProgram( g_hShaderProgram );
…
plane1->PlaneSetTex(bgr_img1);
glReadPixels(0,0,g_nWindowWidth,g_nWindowHeight,GL_RGBA,
             GL_UNSIGNED_BYTE,rawPositionPixels);
calculateCenter(rawPositionPixels, g_TrackingPoints);

EndFBO(g_pOffscreenFBO);
// Clear the colorbuffer and depth-buffer
glClearColor( 0.0f, 0.0f, 1.0f, 1.0f );
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
glUseProgram(g_hVideoShaderProgram); //video shader program
plane1->PlaneSetTex(bgr_img1);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, *texture);
glUniformMatrix4fv( g_hModelViewMatrixLoc, 1, 0, matModelView );
glUniformMatrix4fv( g_hProjMatrixLoc,      1, 0, matProj );
/* … set your uniform locations here */
glUniform2f(g_trackingLoc,g_TrackingPoints[0],g_TrackingPoints[1]);
glDrawArrays(GL_TRIANGLE_STRIP, 8, 4); //draw everything!!
```

Note that this is an abstract of the required code for example purposes only. It may change according to the developer´s needs, so it should not by any means be copy/pasted.

With the flow changes proposed above, the first shader (segmentation) will be rendered to an offscreen buffer, then the centroid of the segmented image will be calculated and stored in `g_TrackingPoints` which is float[2] array. Next, the second shader program `g_hVideoShaderProgram` will be used and the tracking points will be passed as a uniform. Lastly, a blue circle will be drawn over the centroid with the following shader code:

```
const char* g_strFragmentShader =
    #ifdef GL_FRAGMENT_PRECISION_HIGH
       precision highp float;
```

```
    #else
       precision mediump float;
    #endif

    uniform vec2 g_vCenter;
    uniform sampler2D s_texture;
    varying   vec4      g_vVSColor;
    varying   vec2 g_vVSTexCoord;
    vec4 pixelColor;

    vec4 BgrToRgb(vec4 source){
      return vec4(source.b,source.g,source.r,1.0);
      }
    void main()
    {
     pixelColor = BgrToRgb(texture2D(s_texture,vec2(-g_vVSTexCoord.x,-
g_vVSTexCoord.y)));

      if( distance(g_vCenter,gl_FragCoord.xy)<4.0)
          gl_FragColor = vec4(0.0,0.0,1.0,1.0);
       else
           gl_FragColor  = pixelColor;
    }
```

It can be easily observed that the blue dot is drawn by calculating a certain distance from the uniform `g_vCenter`. This controls the size of the circle.

The end result should be a blue dot following the center of the object that is being tracked.

Figure 10: Tracking object

The observed results are very satisfactory; tracking is performed fast and well. One problem can be that objects with similar color could be found in the image. These objects will generate parasite regions and the centroid calculator will have an erroneous output. This is a new problem that the reader needs to be aware of and will be treated in the next section, which is a more complex use case.

5 More Complex Use Case: Object Detection

Feature extraction and object detection are complex uses cases and this represents a real-life application in embedded devices. The goal of this section is to advise the reader on several techniques and methods, but at the same time suggest an approach that is well suited to the i.MX 6 series microprocessors.

Object detection has been a recent field of interest because of its different applications in biomedics, security systems and human-machine interaction. The author will focus this section of the Application Note on the problem of face detection. Typical software solutions for this are available everywhere. OpenCV provides an easy-to-use face detection algorithm, however, the drawback is speed: near real-time detection is essential. Even though the OpenCV uses a highly optimized Artificial Neural Network [opencv], it does not meet the real-time requirements for an embedded application since it relies only on the CPU to do all the processing.

5.1 Face detection algorithms

As mentioned above, there are many techniques on this topic that have been developed over the last 35 years. Below is a brief description of the most relevant techniques, along with a corresponding set of advantages and disadvantages.

5.1.1 Eigenfaces

The Eigenface face detection method is one of the earliest methods for detecting and recognizing faces. Turk and Pentland [turk-penland] implemented this method and the overall process is described as follows:

- Begin with a large image set of faces.
- Subtract the mean of all faces from every face.
- Calculate the covariance matrix for all of the faces.
- Calculate the eigenvalues and corresponding eigenvectors of the covariance matrix. The largest eigenvectors will be chosen as a basis for face representation. These eigenvectors are called “eigenfaces”.

When trying to determine whether a detection window is a face, the image in the detection window is transformed to the eigenface basis. Since the eigen basis is incomplete, image information will be lost in the transformation process. The difference, or error, between the detection window and its eigenface representation will then be thresholded to classify the detection window. If the difference is small, then there is a large probability that the detection window represents a face.

While this method has been used successfully, one disadvantage is that it is computationally heavy to express every detection window using the eigenface basis. This means that the eigenface method is not suitable for real-time use on an embedded system.

5.1.2 Skin color detection

In normal lighting conditions, all human skin tones seem to have certain properties in color space that can be used for skin tone detection. Having said this, human faces can be found by detecting which pixels in an image correspond to human skin, and further examining and labeling skin regions could lead a program to a conclusion that the given region is a human face.

This research has been deeply studied by Störring [storring], and a face detector implementation using skin color detection is described by Singh, Chauhan, Vatsa and Singh in [singh]. They show that when all different skin tones are plotted in different color spaces, such as RGB, YUV and HSL, they span certain volumes of the color spaces. These volumes can be enclosed by surrounding planes. Color values can then be tested to see if they lie within this skin tone volume.

The main advantage of this method is that it is well suited for GPU implementation: the pixel shader can efficiently examine if a pixel color lies within a skin tone volume, since this type of per-pixel evaluation is what GPUs are designed for. However, there are a few disadvantages with skin detection methods:

- For accuracy, normalized color images are required

- Each image has to be normalized before use

Segmented of skin regions are could have noise or undesired information

5.1.3 Haar-based feature detection

Haar-based feature detection methods are based on comparing the sum of intensities in adjacent regions inside a detection window. This window is called the template or the mask. Usually, several features should be tested in order to determine if what is inside the window is a face or not. Features make use of the fact that objects often have some general properties. For example, for faces it is the darker region around the eyes compared to the forehead, or the fact that the two eye regions are similar for every face. Several features are tested within a detection window to determine whether the window contains a face or not.

There are many types of basic shapes for features, as the next figure shows:

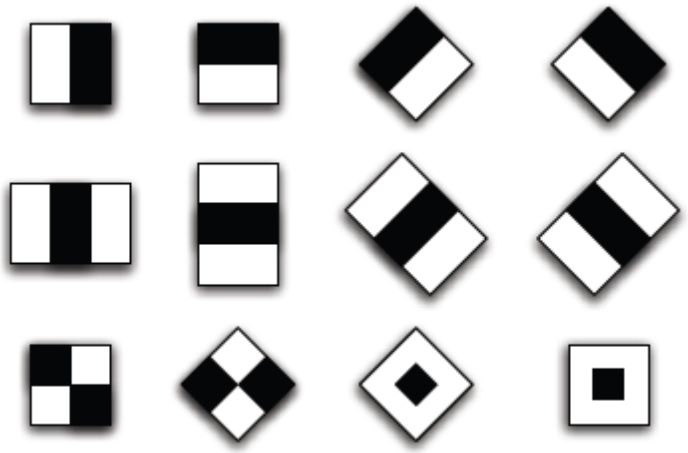

Figure 11: Shapes for features

By varying the position and size of these features within the detection window, an overcomplete set of different features is constructed. The creation of the detector then consists of finding the best combination of features to separate faces from non-faces. This advantage of this method is that the detection process consists mainly of additions and threshold comparisons. This makes the detection fast, perfect for our embedded use case, the main problem is that the accuracy of the face detector is highly dependent on the database used for training. The main disadvantage is the need to calculate sums of intensities for each feature evaluation. This will require lots of lookups in the detection window, depending on the area covered by the feature. Regarding this subject Viola and Jones [viola-jones2001] provide the integral image as a solution to this problem. The use of the integral image allows a Haar based detector to be implemented on GPU, since it is possible to calculate the sum of a region using only a few lookups.

5.2 The detection process

The author of this application note suggests a combination of Skin Tone detection and Haar-like features. However, it will not be a strictly Haar-based technique because the best solution is always the one that fits the developer and use case restrictions. The reader should feel free to experiment with another combination of techniques that may provide better results than the ones suggested here.

5.3 Segment by skin color

The first step is to apply the segmentation technique that was previously explained, but using human skin color. To overcome the problem of different skin colors (white, brown, dark), first the application needs to be "trained" by the user. This can be easily done by taking a sample color of the person that is going to be detected, either from a still picture or from the camera feed (recommended). Again, this is where OpenCV proves to be useful, since we could open an OpenCV window without any processing and write some code to get the desired skin color from a click on that window.

See the code below:

```
void on_mouse( int event, int x, int y, int flags, void* param )
{
    if( !frame )
        return;
    switch( event )
    {
    case CV_EVENT_LBUTTONDOWN:
      CvScalar color = cvGetAt(frame,y,x);
      printf("openCV click at %i,%i  color: %f %f %f
\n",x,y,color.val[0],color.val[1],color.val[2]);
      break;
     }

}
```

This color.val array represents the RGB value of the pixel. This value can be easily passed to the segmentation shader as a vec3 uniform. Following the technique presented in the previous section, a skin-tone segmentation shader can be easily implemented.

In the following image, you can see the result of the segmentation shader and binarization when performed at the same time. Here the developer can choose between two paths:

- Work with grayscale images
- Work with binarized images

If the grayscale path is chosen, then the Integral Image method proposed by Viola-Jones in [viola-jones2001] should be used. If the binary image path is selected, then the integral image is not necessary. Whether to implement it or not depends on the developer.

Since all the segmentation theory and implementation has already been discussed, this section will skip it. Note that binarization can be performed at the same stage as segmentation, avoiding the use of another shader. See the following shader code:

```
"vec4 toBlackAndWhite(vec4 pixel) {
       vec3 current_Color = pixel.rgb;
       float luminance =
       dot (vec3(0.299,0.587,0.114),current_Color);
       if(luminance>.2)
           return vec4(1.0);
       else
            return vec4(0.0);

       }
```

This function will return a vec4 and can be easily called after segmentation, providing the following output:

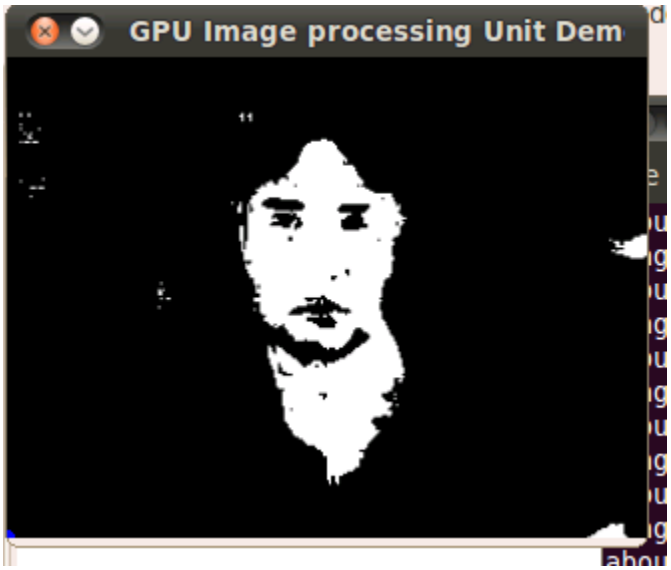

Figure 12: Segmentation output

In the previous image it can easily be seen that there are parasite regions in the segmentation output. This is nearly unavoidable at all times, but there are many options to successfully overcome this problem, commonly known as Mathematical Morphology.

The techniques derived from the mathematical morphology have been widely used in image analysis for many years. It consists of two main operations: erosion and dilation. They can be defined as expanding and shrinking an image, and by applying a feature or “mask” over each pixel. That given pixel will be expanded or shrunk, depending on the structuring element that is being applied. These techniques are deeply studied by Gonzalez and Woods in [gonz-woods2002].

Common structuring elements are:

1	1	1
1	0	1
1	1	1

0	1	0
1	0	1
0	1	0

1	1
1	1

Figure 13: Common structuring elements

The main idea of the erosion operation is better explained by the following image, using the first structuring element shown above:

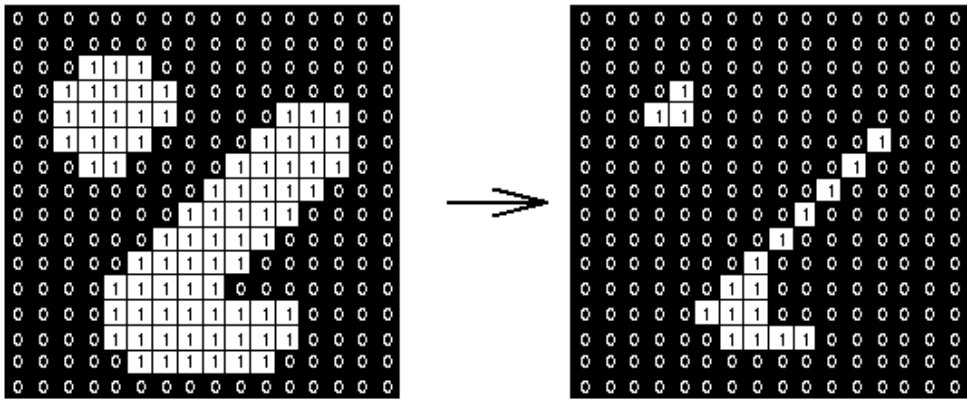

Figure 14: Erosion operation

For this use case, performing a simple erosion operation within the shader (after segmentation is done) is possible to clean up most of the image.

See the shader code below:

```
vec4 doErosion(vec4 pixel) {
    float sum = 0.0;
     float step =2.0;
     int fill=0;
     if (pixel.r!=0.0 && pixel.g!=0.0 && pixel.b!=0.0){
      float new_pixel00 = toGrayscale(segmentColor(BgrToRgb(
                          texture2D(s_texture, vec2(g_vVSTexCoord.x-
xOffset*step,g_vVSTexCoord.y-yOffset*step)))));
     if(new_pixel00 == 0.0) fill++;
      float new_pixel01 = toGrayscale(segmentColor(BgrToRgb(
                          texture2D(s_texture,
vec2(g_vVSTexCoord.x,g_vVSTexCoord.y-yOffset*step)))));
     if(new_pixel01 == 0.0) fill++;
      float new_pixel02 = toGrayscale(segmentColor(BgrToRgb(
                          texture2D(s_texture,
vec2(g_vVSTexCoord.x+xOffset*step,g_vVSTexCoord.y-yOffset*step)))));
     if(new_pixel02 == 0.0) fill++;
     float new_pixel10 = toGrayscale(segmentColor(BgrToRgb(
```

```
                        texture2D(s_texture, vec2(g_vVSTexCoord.x-
xOffset*step,g_vVSTexCoord.y))));
     if(new_pixel10 == 0.0) fill++;
     float new_pixel12 = toGrayscale(segmentColor(BgrToRgb(
                        texture2D(s_texture,
vec2(g_vVSTexCoord.x+xOffset*step,g_vVSTexCoord.y))));
     if(new_pixel12 == 0.0) fill++;
     float new_pixel20 = toGrayscale(segmentColor(BgrToRgb(
     texture2D(s_texture, vec2(g_vVSTexCoord.x-
xOffset*step,g_vVSTexCoord.y+yOffset*step))));
     if(new_pixel20 == 0.0) fill++;
     float new_pixel21 = toGrayscale(segmentColor(BgrToRgb(
     texture2D(s_texture,
vec2(g_vVSTexCoord.x,g_vVSTexCoord.y+yOffset*step))));
     if(new_pixel21 == 0.0) fill++;
     float new_pixel22 = toGrayscale(segmentColor(BgrToRgb(
                         texture2D(s_texture,
vec2(g_vVSTexCoord.x+xOffset*step,g_vVSTexCoord.y+yOffset*step))));
    if(new_pixel22 == 0.0) fill++;

    if(fill>=6)
         pixel= vec4(0.0,0.0,0.0,1.0);
     return pixel;

    }
```

It can be observed that this is a little variation of the erosion technique, because the structuring element may be applied to a distance equal or greater than 1 pixel, in order to provide better results, and lastly if the sum of the matches is greater or equal to 6 (set empirically), then the pixel that we are checking should go away, meaning that the pixel is highly likely to be in an isolated region of the image.

After performing erosion, the output should be similar to the image shown below:

Figure 15: Image output after erosion operation

All but the largest parasite regions have been removed. These remain because the structuring element fits inside the big regions of the image. However, this should not present a large problem because the next step in the process is able to discard those parasite regions as possible faces.

5.4 Selecting the features

This step is part of a “training”: it is necessary to give the program a knowledge base and instruct it to recognize the features we are looking for. The features to be recognized are the eyes and mouth, but this could be easily extended to the nose and/or ears. Ideally, an average of several images should be used to construct the feature, meaning that a wider range of characteristics from the human face are considered.

This example will be limited to the eyes feature. They will be split into right and left eyes.

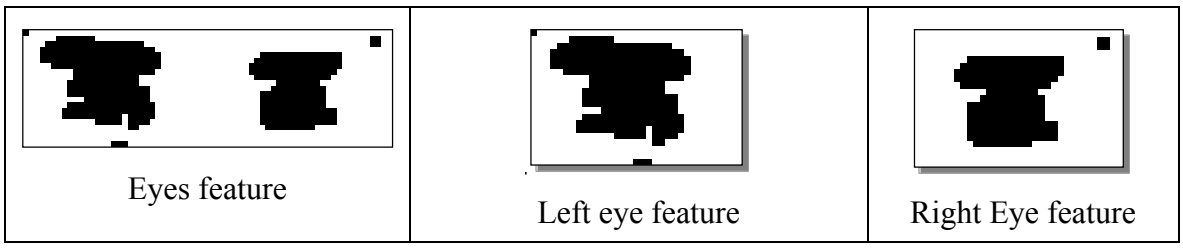

Figure 16: Eyes feature

The eyes feature is a 68x21 pixel window, while the left and right eye feature are 64x21 each. It is very important to determine the window size, as it will be necessary for the following calculations.

Next, it is necessary to obtain some information from our features (both eyes, as well as each eye separately). A huge study on this subject has been made over the last 45 years or so. Ming-Kuei Hu states that it is possible to calculate 7 image moments that are invariant to translation, similitude and orthogonal transformations in [hu1962]. His work was later completed by Flusser and Suk, introducing a general theorem and four new invariants. Even though it is out of the scope of this application note to do a complete study of image invariants, the reader is encouraged to read such material for use in their face detection algorithm. In any case, the output of getting an invariant or another feature extraction procedure is a float/double–like number. The author suggests that the reader calculate Hu´s first invariant (the code is not shown here for reasons mentioned above). An easier way to get some information out of the features is to try to determine the relation between black pixels and white pixels within the feature window. It is very simple to achieve this: it is only necessary to count all the black pixels and divide them by the window size (68x21). By doing so, we are approaching the concept of the Haar feature explained previously. This should be done for all the features we want to examine, in our example we will have 3 float values (or doubles if Hu moments were used).

5.4.1 The weak classifier

As mentioned before, this method uses an approach like the Haar feature classifiers, but the Haar classifiers need a large amount of training data in order to get desirable thresholds/results. Since we are using a simpler use case, we will try to avoid the training step.

Weak classifiers are constructed using one feature. It classifies a detection window as positive (face) or negative (non-face) depending on the “likeness” of the information inside that window and the feature we are testing against. The result from one weak classifier does not give enough information, but several combined into a strong classifier can determine whether a detection window contains a face or not.

One weak classifier would be the whole eyes feature. If the test window does not have enough white pixels (for example if it lies in the background) then it would be discarded immediately, thereby avoiding any unnecessary calculations. Other weak classifiers are the left and right eyes respectively.

5.4.2 The strong classifier

The strong classifier is a combination of several weak classifiers. Each of the weak classifiers is given weights depending on their detection accuracy. When classifying a detection window with a strong classifier, all of the weak classifiers are evaluated. At the end, the strong classifier determines if the detecting window has a face or not. If the false positive is too high, the strong classifier then needs more weak classifiers.

5.4.3 The cascade

Instead of evaluating one large strong classifier on a detection window, Viola and Jones suggest in [viola-jones2001] the concept of a cascade. A cascade is only a concept of strong classifiers that all have a high detection rate, and a fairly low false positive ratio. If a strong classifier classifies a detection window as a non-face area, then the window is immediately discarded. The whole cascade (series of strong classifiers) is evaluated when a face is found (or something very similar to a face). Note that OpenCV implements a Haar cascade face detection. The reader might prefer to compile the example and test its results.

6 Implementing the face detector

Up to this point, we have discussed how to obtain information from a series of features that describe a face: from a pure mathematical perspective (Hu´s invariants) or from a more intuitive perspective point of view (black/white ratios, Haar features). In either case, the actual face detection implementation can be done in normal C code that will run in the CPU, or in shader code that will run on the GPU. Either of those is acceptable, since a great deal of work has been done on the GPU side to lower the CPU load.

If the developer chooses the GPU for the detector, then the features can be evaluated in shader code. It is important to note that for a real use case, the detection window will need to vary depending on the subject's distance from the camera; the results from the Hu´s invariants or the black/white ratio will vary depending on that distance. A study on Multi-Scale Template matching can be found in [tang-tao2007] where the template is represented as a linear combination of a small number of binary box features using the integral image. However, passing the integral image information to the shader comes with a problem: The OpenGL ES 2.0 specification does not provide a high precision texture format suitable for representing the integral image.

This problem can be solved by doing the following: upload the 32bit integral image to the GPU by configuring a texture to contain RGBA values, therefore allowing them to be manipulated in the shader code. Since we are trying to keep it simple, we will not use an integral image for now. This means the shader functions to evaluate the classifiers will need also a variable window size.

To implement what is left of the algorithm, the 2-shader programs approach is needed. The first program will perform segmentation and binarization, and the second shader program will perform the weak and hard classification.

One more thing is needed: we need our segment-binary shader output to provide the input of the face-detection shader. For this, the reader should reference the previous section, where the output of the first shader program is written to an offscreen buffer, and is then read as an array of pixels. We will use this code base, but with a modification: the array of pixels will become another texture that will be passed as a sampler to the second shader program.

The reader should note that creating textures is expensive, so multiple shader program chaining is not encouraged. Two shader chained programs should perform well enough, but it is important to avoid unnecessary multiple shader chaining.

In the following C++ code, after writing to the offscreen buffer, the content is read with glReadPixels() and stored in an array. A texture is created with contents of that array.

```
…
glDrawArrays(GL_TRIANGLE_STRIP, 8, 4);
glReadPixels(0,0,g_nWindowWidth,g_nWindowHeight,
GL_RGB,GL_UNSIGNED_BYTE,rgbPixelBuffer);
EndFBO(g_pOffscreenFBO);
//clear background,
// Clear the colorbuffer and depth-buffer
glClearColor( 0.0f, 0.0f, 1.0f, 1.0f );
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
glUseProgram(g_hShaderProgram); //the other shader
glTexImage2D(GL_TEXTURE_2D,0,GL_RGB,WIDTH,HEIGHT,
                    0,GL_RGB,GL_UNSIGNED_BYTE,rgbPixelBuffer);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, *texture);
glUniformMatrix4fv( g_hModelViewMatrixLoc, 1, 0, matModelView );
glUniformMatrix4fv( g_hProjMatrixLoc,      1, 0, matProj );
```

Next, we need to evaluate the weak classifier in shader code (if the user decided to do so). The overall process is as follows:

1. Adjust the window-size input to fit the current detection window size, set by the uniform size
2. Do lookups and compare against the weighted black/white average
3. Compare the value against a given threshold (again, set empirically). The polarity value can indicate if the result value should be greater or smaller than the threshold
4. Return result of comparison

Note that only the prototype function is shown below, since it will be up to the reader to select the features and implement the classifier code.

```
Uniform highp float size;
 float evalWeak1( xpos, ypos, width,
 height, polarity, threshold)
 {
```

```
 // Evaluate classifier in current detection window.
 // Return 1.0 if detection window classified as
 // a face, otherwise 0.0.

 }
```

The next step is to implement the hard classifier. As previously mentioned a strong classifier evaluates several weak classifiers and changes the detection window if necessary. The weights should be the result of a training or the Haar classifiers weight that OpenCV uses.

```
Bool strong1()
 {
 float r1 = 1.47299*evalWeak1( 8.0, 5.0, 5.0,
 19.0, 1.0, -611.0);
 float r2 = 1.27765*evalWeak2(20.0, 0.0, 4.0,
 23.0,-1.0, 137.0);
 float r3 = 1.30542*evalWeak1( 8.0, 3.0, 10.0,
 11.0, 1.0,-1718.0);
 float thresh = 2.02803;
 float res = r1 + r2 + r3;
 return res>thresh;

 }
```

The only thing left to implement is the cascade. The cascade evaluates several strong classifiers, then sets the pixel color to red if a face is found. This means that when a window detects a face, the center will be red. Note that this may cause a bulk of red pixels in the center of the face, but that is what we intend to do in the first place.

Cascade code:

```
void main
 {

 if (!strong1()){
 discard;
 }
 if (!strong2()){
 discard;
 }
 .
 .
 .
 if (!strongN()){
 discard;
 }
 gl_FragColor = vec4(1.0,0.0,0.0,1.0);

 }
```

The results should be something similar to the image shown below. If false positives appear, adjustments should be made to the feature, the weights, or both.

Figure 17: Image after cascade

7 Conclusions and future work

Within this Application Note, every effort has been made to introduce the reader to the basics of image processing theory and algorithms. The main purpose for this application note is to present an alternative way of doing image processing-related computations by using the GPU. Another option, of course, is to use OpenCL, which is available on the higher end of the i.MX 6 series family of processors, namely i.MX 6Quad and i.MX 6Dual.

Interested readers may check the references in the last section, which is a very good compendium of Image Processing papers and books that the author found useful. Lastly, the reader is encouraged to implement these techniques and algorithms in OpenCV or their own implementation, and to compare performance against the GPU techniques presented here.

8 References

[belongie98] Belongie, S. et. al., “Color- and texture-based image segmentation using EM and its application to content-based image retrieval”, *Proc. of ICCV*, p. 675-82. 1998.

[cast96] Castleman, K. R., *Digital Image Processing*. Prentice Hall, 1995

[comaniciu-meer] Comaniciu, D., Meer, P., *Robust Analysis of Feature Spaces: Color Image Segmentation*. Dept of Electrical and Computer Engineering Rugters Univ, Piscataway, NJ.

[gonz-woods2002] Gonzalez, R. C., Woods, R. E., *Digital Image Processing*, second edition. Prentice Hall, 2002.

[hu1962] Ming Kuei Hu, “Visual Pattern Recognition by Moment Invariants”, *IRE Transactions on Information Theory*. 1962.

[huss2010] Hussein, J.A., “Spatial Domain Watermarking Scheme for Colored Images Based on Log-average Luminance”, *Journal of Computing,* vol2 issue 1, January 2010.

[opencv] http://opencv.org/

[opengl] http://www.khronos.org/opengles/

[opengl-ref] OpenGL Reference Pages http://www.khronos.org/opengles/sdk/docs/man/

[opengl-guide]OpenGL ES 2.0 Programming Guide, http://opengles-book.com/

[openGL-spec] OpenGL SL ES Specification
http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf

[otzu1979] Otzu. N., “A thresholding Selection Model from Gray-Level Histograms”, *IEEE Transaction on System, Man, And Cybernetic*, Vol. SMC-9, N0. 1, Jan 1979.

[rod-sossa2102] Rodriguez, R., Sossa, H., *Procesamiento Digital de Imágenes*. Alfa-Omega, 2012.

[sahoo1988] Sahoo, P. K., Soltani,S. and Wong, A. K. C., “A Survey of Thresholding Techniques”, *Computer Vision, Graphics, and Image Processing*, 41-233-260, 1988.

[singh2003] Singh, Sanjay Kr., Chauhan, D. S., Vatsa, Mayank and Singh, Richa, *A robust skin color based face detection algorithm*. 2003.

[storring] Störring, Moritz, *Computer vision and human skin colour.*

[sural-qian] Sural, S., Gang Qian, Pramanik, Sakti, *Segmentation and Histogram Generation using the HSV Color Space for Image Retreval*. Dept of Computer Science and Engineering, MI, USA.

[taing-tao2007] Tang, F., Taio, H., *Fast Multi-Scale Template Matching Using Binary Features.* Department of Computer Engineering, University of California, Santa Cruz, 2007.

[turk-pentland91] Turk, M. and Pentland, A., *Face recognition using eigenfaces*. 1991.

[viola-jones2001] Viola, Paul and Jones, Michael J., *Rapid object detection using a boosted cascade of simple features*. 2001.

Table1. Revision history

Revision	Substantial changes
0	Initial release
1	Updated steps in Section 4.4, added revision history table

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document.

Freescale reserves the right to make changes without further notice to any products herein. Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer’s technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: AN4629
Rev. 1
02/2015