```c
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "std_image_write.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <linux/videodev2.h>
#include <math.h>
#include <SDL2/SDL.h>
#include <omp.h>
#include <pthread.h>
// gcc -Wall labb2test.c -o labb2test -lSDL2 -fopenmp -lpthread

#define CAM_WIDTH 640
#define CAM_HEIGHT 480
unsigned char *arr[2];
int length;
/*
Window:
you might notice that there is no documentation link to SDL_Window.
This is because the structure is opaque;
your program cannot see what is actually contained in a "SDL_Window."
You will simply manage a pointer to a SDL_Window.

CreateWindow: it takes parameters specifying
the name, size, position, and options for the window, and
returns a pointer to the new Window structure.

Renderer:
Think of SDL_Window as physical pixels,
and SDL_Renderer and a place to store settings/context.
So you create a bunch of resources,
and hang them off of the renderer
and then when its ready, you tell renderer
to put it all together and send the results to the window.

window:the window where rendering is displayed

index:the index of the rendering driver to initialize, or -1 to initialize
the first one supporting the requested flags

flags:0, or one or more SDL_RendererFlags OR'd together

*/

int open_handle()
{
    // open the device handel
    int cameraHandle = open("/dev/video0", O_RDWR, 0);
    // set a supported video format
    struct v4l2_format format;
    memset(&format, 0, sizeof(format));
    format.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    format.fmt.pix.width = CAM_WIDTH;
    format.fmt.pix.height = CAM_HEIGHT;
    format.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV; // MJPG
    format.fmt.pix.field = V4L2_FIELD_ANY;

    if (ioctl(cameraHandle, VIDIOC_S_FMT, &format) < 0)
    {
        printf("Vidioc_s_fmt video format set fail\n");
        return -1;
    }
    return cameraHandle;
}

int request(int cameraHandle)
{

    struct v4l2_requestbuffers req;
    memset(&req, 0, sizeof(req));
    req.count = 2;
    req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    req.memory = V4L2_MEMORY_MMAP;
    if (ioctl(cameraHandle, VIDIOC_REQBUFS, &req) < 0)
    {
        printf("VIDIOC_REQBUFS failed!\n");
        return -1;
    }
    return 0;
}

int set_query(int cameraHandle)
{

    for (int i = 0; i < 2; i++)
    {


        struct v4l2_buffer buf;
        memset(&buf, 0, sizeof(buf));
        buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        buf.memory = V4L2_MEMORY_MMAP;
        buf.index = i;
        if (ioctl(cameraHandle, VIDIOC_QUERYBUF, &buf) < 0)
        {
            printf("VIDIOC_QUERYBUF failed!\n");
            return -1;
        }
        arr[i] = mmap(NULL, buf.length, PROT_READ | PROT_WRITE,
MAP_SHARED, cameraHandle, buf.m.offset);
        length = buf.length;
    }
    return 0;
}

int queue_up(int cameraHandle, int i)
{

    struct v4l2_buffer buf;
    memset(&buf, 0, sizeof(buf));
    buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    buf.memory = V4L2_MEMORY_MMAP;
    buf.index = i;
    if (ioctl(cameraHandle, VIDIOC_QBUF, &buf) < 0)
    {
        printf("VIDEOC_OBUF failed!\n");
        return -1;
    }
    return 0;
}

int start_camera(int cameraHandle)
{

    enum v4l2_buf_type type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    if (ioctl(cameraHandle, VIDIOC_STREAMON, &type) < 0)
    {
        printf("VIDOC_STREAMON failed!\n");
        return -1;
    }
    return 0;
}

typedef struct Pixel
{
    unsigned char R;
    unsigned char G;
    unsigned char B;
    unsigned char A;

} Pixel;

void *YUYVtoRGB(unsigned char y, unsigned char u, unsigned char v, Pixel
*_rgba, int cap)
#define MIN(a, b) (((a) < (b)) ? (a) : (b))
#define MAX(a, b) (((a) < (b)) ? (b) : (a))
{
    int c = y - 16;
    int d = u - 128;
    int e = v - 128;
    if (cap == 1)
    {
        _rgba->A = 255; // Alpha
        _rgba->R = MAX(0, MIN((298 * c + 409 * e + 128) >> 8, 255));
        _rgba->G = MAX(0, MIN((298 * c - 100 * d - 208 * e + 128) >> 8,
255));
        _rgba->B = MAX(0, MIN((298 * c + 516 * d + 128) >> 8, 255));
    }
    else{
        _rgba->A = 255; // Alpha
        _rgba->B = MAX(0, MIN((298 * c + 409 * e + 128) >> 8, 255));
        _rgba->G = MAX(0, MIN((298 * c - 100 * d - 208 * e + 128) >> 8,
255));
        _rgba->R = MAX(0, MIN((298 * c + 516 * d + 128) >> 8, 255));
    }

    return 0;
}

void *Capture(void *args)
{
    Pixel *rgbThread = args;
    stbi_write_png("bild_from_cap.png", CAM_WIDTH, CAM_HEIGHT, 4,
rgbThread, CAM_WIDTH * 4);
    free(args);
    return 0;
}

int ProcessImage(const unsigned char *_yuv, int _size, SDL_Renderer
*g_renderer, SDL_Texture *g_streamTexture, int cap)
{
    void *pixels;
    int pitch;
    SDL_LockTexture(g_streamTexture, NULL, &pixels, &pitch);

#define RGB_SIZE CAM_WIDTH *CAM_HEIGHT
   double t1 = omp_get_wtime();
    Pixel *rgbConversion = (Pixel *)pixels;
#pragma omp parallel num_threads(2)
    {

        int id = omp_get_thread_num();
        int start = id * (_size / 2);
        int end = (_size / 2) + start;
        int rgbIndex = id * _size / 4;

        for (int i = start; i < end; i += 4)
        {

            unsigned char y1 = _yuv[i + 0];
            unsigned char u = _yuv[i + 1];
            unsigned char y2 = _yuv[i + 2];
            unsigned char v = _yuv[i + 3];
            YUYVtoRGB(y1, u, v, &rgbConversion[rgbIndex++],cap);
            YUYVtoRGB(y2, u, v, &rgbConversion[rgbIndex++],cap);
        }
    }
    double time_spent = (omp_get_wtime() - t1) * 1000;
    printf("Time tooked: %0.2f\r",time_spent);
    // double t0 = omp_get_wtime();
    // int rgbIndex = 0;
    // for (int i = 0; i < _size; i += 4)

    //     {
    //         unsigned char y1 = _yuv[i + 0];

    //         unsigned char u = _yuv[i + 1];

    //         unsigned char y2 = _yuv[i + 2];

    //         unsigned char v = _yuv[i + 3];

    //         YUYVtoRGB(y1, u, v, &rgbConversion[rgbIndex++],cap);

    //         YUYVtoRGB(y2, u, v, &rgbConversion[rgbIndex++],cap);
    //        }

    // double milliseconds = (omp_get_wtime() - t0) * 1000;

    // printf("Time spent: %5.2fms\r", milliseconds);
    for (int y = 0; y < CAM_HEIGHT; y += 1)
    {
        for (int x = 0; x < CAM_WIDTH; x += 1)
        {
            int rgb_size = (CAM_WIDTH * (y)) + (x);
            if (rgbConversion[rgb_size].R == 255 &&
rgbConversion[rgb_size].B == 255 && rgbConversion[rgb_size].G == 255)
            {
                if (y < 300)
                {
                    int l = (CAM_WIDTH * (y)) + (128);
                    int f = (CAM_WIDTH * (y)) + (512);
                    if (rgb_size < l)
                    {
                        printf("Turn Left\r");
                    }
                    else if (rgb_size > l && rgb_size < f)
                    {
                        printf("Move Forword\r");
                    }
                    else
                        printf("Turn Right\r");
                }
                else
                    printf("Move Back\r");
            }
        }
    }

    SDL_UnlockTexture(g_streamTexture);
    SDL_RenderCopy(g_renderer, g_streamTexture, NULL, NULL);
    SDL_RenderPresent(g_renderer);
    if (cap == 1)
    {
        Pixel *rgbmalloc = malloc(RGB_SIZE * 4);

        memcpy(rgbmalloc, rgbConversion, RGB_SIZE * 4);
        pthread_t id;

        pthread_create(&id, NULL, Capture, rgbmalloc);
        pthread_detach(id);

    }

    return 0;
}

int dequeue(int cameraHandle, int i, SDL_Renderer *g_renderer, SDL_Texture
*g_streamTexture, int cap)
{

    struct v4l2_buffer buf;
    memset(&buf, 0, sizeof(buf));
    buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    buf.memory = V4L2_MEMORY_MMAP;

    if (ioctl(cameraHandle, VIDIOC_DQBUF, &buf) < 0)
    {
        return errno;
    }

    ProcessImage(arr[i], buf.bytesused, g_renderer, g_streamTexture, cap);
    return 0;
}

int turn_off(int camera, SDL_Window *g_window,SDL_Renderer *g_renderer)
{
    munmap(arr, length);
    close(camera);
    pthread_mutex_destroy(&lock);
    SDL_DestroyRenderer(g_renderer);
    SDL_DestroyWindow(g_window);
    SDL_Quit();
    return 0;
}




int main()

{
    int camera = open_handle();
    SDL_Window *g_window;
    SDL_Renderer *g_renderer;
    SDL_Texture *g_streamTexture;
    g_window = SDL_CreateWindow("SDL Window", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED, CAM_WIDTH, CAM_HEIGHT, SDL_WINDOW_OPENGL |
SDL_WINDOW_SHOWN);
    g_renderer = SDL_CreateRenderer(g_window, -1,
SDL_RENDERER_ACCELERATED);
    if (SDL_Init(SDL_INIT_VIDEO) < 0) {
        printf("SDL_Init: %s\n", SDL_GetError());
        turn_off(camera, g_window, g_renderer);
        return -1;
    }

    if (!g_window){
        printf("Window failed\n");
        turn_off(camera, g_window, g_renderer);
        return -1;
        }

    if (!g_renderer)
    {
        printf("Renderer failed\n");
        turn_off(camera, g_window, g_renderer);
        return -1;
    }
    g_streamTexture = SDL_CreateTexture(g_renderer,
SDL_PIXELFORMAT_ARGB8888, SDL_TEXTUREACCESS_STREAMING, CAM_WIDTH,
CAM_HEIGHT);
    if (!g_streamTexture)
    {
        printf("Texture failed\n");
        turn_off(camera, g_window, g_renderer);
        return -1;
    }


    request(camera);
    set_query(camera);
    start_camera(camera);

    while (camera > 0)
    {

        for (int i = 0; i < 2; i = +1)
        {
            int capture = 0;
            SDL_Event event;
            SDL_PollEvent(&event);
            if (event.type == SDL_KEYDOWN)
            {
                if (event.key.keysym.sym == SDLK_ESCAPE)
                {
                    turn_off(camera, g_window, g_renderer);
                    camera = 0;
                    return 0;
                }
                if (event.key.keysym.sym == SDLK_c)
                {
                    capture = 1;
                }
            }

            queue_up(camera, i);
            dequeue(camera, i, g_renderer, g_streamTexture, capture);

        }
    }
    return 0;
}
```