```cpp
#include <iostream>
#include <opencv2/opencv.hpp>
#include <vector>
#include <string>
#include <sstream>
#include <limits>
#include <algorithm>


using namespace cv;
using namespace std;


float center_u_pos = -776.8803716;
float center_v_pos = 533.057190;

int center_s = 512;
int center_t = 256;

cv::Mat lf_img_arr(17, 17, CV_32FC2);
std::vector<cv::Mat> UV_arr_ST_images; // a vector of the images in sequential order; accessing requires finding which index based on -> row index * (lf_img_arr.cols) + col index


// row and col of UV_arr_ST_images; T, S row and column of that image
cv::Vec3b get_pixel_4d(int row, int col, int t, int s) {
    cv::Mat current_image = UV_arr_ST_images[row * lf_img_arr.cols + col];  // accesses current UV image at [row][col]
    return current_image.at<cv::Vec3b>(t, s);    // returns the pixel located at (s, t) -> (x, y) in the UV array of ST images defined by row, col input

}


float calc_distance(float x1, float y1, float x2, float y2) {
    return sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
}

// create micro images with aperture
cv::Mat micro_image(int s, int t, float center_x, float center_y, int aperture_radius) {
    cv::Mat current_img = cv::Mat::zeros(17, 17, CV_8UC3);   // initialize the current micro image to all zeros, so if is within aperture radius will get filled with current pixel
    float current_distance = 0;

    for (int i = 0; i < lf_img_arr.rows; i++) {    // this iterates through the 17x17 images
        for (int j = 0; j < lf_img_arr.cols; j++) {
            current_distance = calc_distance(lf_img_arr.at<Vec2f>(i, j)[0], lf_img_arr.at<Vec2f>(i, j)[1], center_x, center_y);

            if (current_distance <= aperture_radius) {  // checks if the distance of the UV image is within the radius from the given center point
                // get st value at pixel location in iteration and assign it to the 17x17 current_img
                current_img.at<Vec3b>(i, j) = get_pixel_4d(i, j, t, s);
            }
        }
    }
    return current_img;
}




int main(int argc, char* argv[]) {


    //*************************** LOAD IMAGES --- START --- ************************************************************************


    // parse all images
    std::cout << "Loading light field ..." << std::endl;
    std::vector<cv::String> lf_imgs;  // vector that stores the filepaths for all the light field images
    cv::glob("C:/Users/ryamh/OneDrive/Documents/VOW/CGRA 352/Assignments/CGRA_352_A1/CGRA_352_A1/light_ray_images", lf_imgs);  // ingests the filepaths of each light field image

    for (cv::String cv_str : lf_imgs) {   // iterates through all the filepath strings in the vector lf_imgs
        // get the filepath
        std::string filepath(cv_str);   // assigns the current iteration string to the string variable filepath
        size_t pos = filepath.find_last_of("/\\");   // stores the last instance of \\ (for windows) to identify where the end of the filepath is
        if (pos != std::string::npos) {              // checks to verify that at least one instance of \\ separator was found
            // replace "_" with " "
            std::string filename = filepath.substr(pos + 1);  // extracts the filename from the filepath and stores it in string filepath variable
            pos = 0;
            while ((pos = filename.find("_", pos)) != std::string::npos) {  // replaces all '_' characters with ' ' character  1_2_3.png -> 1 2 3.png
                filename.replace(pos, 1, " ");
                pos++;
            }
            // parse for values
            std::istringstream ss(filename);  // creates a string stream variable, ss; makes it easy to assign variables to tokens using >> operator
            std::string name;
            int row, col;
            float v, u;
            ss >> name >> row >> col >> u >> v;    // assigns tokens to variables in sequential order; name, row, col, v, u
            if (ss.good()) {
                cv::Mat current_image = cv::imread(filepath);
                UV_arr_ST_images.push_back(current_image);

                lf_img_arr.at<cv::Vec2f>(row, col)[0] = u;    // stores v as first element
                lf_img_arr.at<cv::Vec2f>(row, col)[1] = v;    // stores u as second element
                continue;                                     // skips the rest of the code in the for loop, so doesn't abort
            }
        }

        // throw error otherwise (if (pos == npos))
        std::cerr << "Filepath error with : " << filepath << std::endl;
        std::cerr << "Expected in the form : [prefix]/[name]_[row]_[col]_[v]_[u][suffix]";
        abort();        // forcefully terminates the program
    }

    std::cout << "Finished loading light field" << std::endl;


    //*************************** LOAD IMAGES --- END --- ************************************************************************

    // test get_pixel_4d() function ------------- CORE RESULTS ---------------------------------------
    cv::Vec3b test_pixel = get_pixel_4d(7, 10, 384, 768);
    std::cout << "Blue: " << static_cast<int>(test_pixel[0]) << std::endl;
    std::cout << "Green: " << static_cast<int>(test_pixel[1]) << std::endl;
    std::cout << "Red: " << static_cast<int>(test_pixel[2]) << std::endl;

    //-----------------------------------------------------------------------------------------------


    //*************************** APERTURE --- START --- ************************************************************************

    // vector store all micro images with
    std::vector<cv::Mat> all_micro_images_arr_r_75;
    std::vector<cv::Mat> all_micro_images_arr_r_40;

    for (int t = 285; t < 305; t++) {
        for (int s = 770; s < 870; s++) {
            all_micro_images_arr_r_75.push_back(micro_image(s, t, center_u_pos, center_v_pos, 75));
            all_micro_images_arr_r_40.push_back(micro_image(s, t, center_u_pos, center_v_pos, 40));
        }
    }

    cout << "finished loading all micro images into both vectors" << endl;


    // create 1700x1700 image of all micro images radius = 75
    cv::Mat big_matrix_r_75(1700, 1700, CV_8UC3, cv::Scalar(0, 0, 0));
    cv::Mat big_matrix_r_40(1700, 1700, CV_8UC3, cv::Scalar(0, 0, 0));

    int micro_image_index = 0;
    float col_index = 0;
    float row_index = 0;

    cout << "col_index: " << col_index << endl;
    cout << "row_index: " << row_index << endl;


    cout << "All micro images size: " << all_micro_images_arr_r_75.size() << endl;


    for (int i = 0; i < 10000; i++) {
        if (col_index == 1700) {
            col_index = 0;
            row_index += 17;
        }
        all_micro_images_arr_r_75[i].copyTo(big_matrix_r_75(cv::Rect(col_index, row_index, 17, 17)));
        all_micro_images_arr_r_40[i].copyTo(big_matrix_r_40(cv::Rect(col_index, row_index, 17, 17)));

        col_index += 17;
    }


    cv::imshow("big_matrix_r_75", big_matrix_r_75);
    cv::waitKey(0);

    cv::imshow("big_matrxi_r_40", big_matrix_r_40);
    cv::waitKey(0);


    //*************************** APERTURE --- END --- ************************************************************************

    //***************************** FOCUS --- START --- ************************************************************************

    int aperture_radius = 35;  // can alter this value for varying aperture size
    float alpha = 1.05;        // can alter this value for varying virtual focus distance (plane)

    // results image
    cv::Mat focus_result_image_1(512, 1024, CV_8UC3, cv::Scalar(0, 0, 0));

    int norm_s_prime = 0;
    int norm_t_prime = 0;

    float norm_u_pos = 0.0;
    float norm_v_pos = 0.0;

    int light_ray_sum_b = 0;
    int light_ray_sum_g = 0;
    int light_ray_sum_r = 0;

    float valid_ray_count = 1.0; // initialized to 1 to prevent division by zero error

    // iterate through ever s' t' value
    for (int t_prime = -256; t_prime < 256; t_prime++) {
        for (int s_prime = -512; s_prime < 512; s_prime++) {
            // normalize s' t' position
            int norm_t_prime = t_prime;
            int norm_s_prime = s_prime;


            //  iterate through all UV images (17x17)
            for (int v_row = 0; v_row < 17; v_row++) {
                for (int u_col = 0; u_col < 17; u_col++) {
                    // get uv position of current v_row, u_col uv image
                    float v_pos = lf_img_arr.at<cv::Vec2f>(v_row, u_col)[0];
                    float u_pos = lf_img_arr.at<cv::Vec2f>(v_row, u_col)[1];
                    // normalize uv position for current v_row, u_col uv image
                    norm_u_pos = u_pos - center_u_pos;
                    norm_v_pos = v_pos - center_v_pos;

                    // if current_norm_uv is within aperture radius
                    if (calc_distance(norm_v_pos, norm_u_pos, 0.0, 0.0) <= aperture_radius) {
                        // then calculate L(u,v,s,t) to get pixel's (b,g,r) value (Vec3b) based on current s' t' value and the given alpha value
                        //  this will find the proper s, t value within the current uv image being analyzed
                        float s = norm_v_pos + (alpha * norm_s_prime - norm_v_pos) / alpha;
                        float t = norm_u_pos + (alpha * norm_t_prime - norm_u_pos) / alpha;

                        // store the pixel at s, t in the current u, v image in the iteration
                        cv::Mat current_image = UV_arr_ST_images[v_row * lf_img_arr.cols + u_col];

                        // offset the normalization so can locate s and t on current image
                        float remap_s = s + center_s;
                        float remap_t = t + center_t;

                        // this is to enable the code to run
                        // the s and t values are growing outside the range of 1024x512 -> (columns x rows)
                        if (remap_s > 1023) {
                            remap_s = 1023;
                        }
                        else if (remap_s < 0) {
                            remap_s = 0;
                        }

                        if (remap_t > 511) {
                            remap_t = 511;
                        }
                        else if (remap_t < 0) {
                            remap_t = 0;
                        }

                        cv::Vec3b current_light_ray = current_image.at<cv::Vec3b>(remap_t, remap_s);

                        // add found light ray value to culomative sum variable -> light_ray_sum
                        float temp_blue = current_light_ray[0];
                        float temp_green = current_light_ray[1];
                        float temp_red = current_light_ray[2];

                        light_ray_sum_b += temp_blue;
                        light_ray_sum_g += temp_green;
                        light_ray_sum_r += temp_red;

                        // increment valid_ray_count by 1
                        valid_ray_count++;
                    }
                }
            }


            // find average color value for b, g, r
            float result_blue = light_ray_sum_b / valid_ray_count;
            float result_green = light_ray_sum_g / valid_ray_count;
            float result_red = light_ray_sum_r / valid_ray_count;

            // update result_image at current s' t' iteration position with found result_pixel
            cv::Vec3b result_pixel(result_blue, result_green, result_red);
            focus_result_image_1.at<cv::Vec3b>(t_prime + center_t, s_prime + center_s) = result_pixel;

            // zero out all values so can iterate again
            light_ray_sum_b = 0;
            light_ray_sum_g = 0;
            light_ray_sum_r = 0;

            valid_ray_count = 1;   // set to 1 intead of 0 to account for division by zero error

        }
    }

    cv::imshow("completion/challenge result", focus_result_image_1);
    cv::waitKey();


    //***************************** FOCUS --- END --- ************************************************************************



    return 0;
}
```