

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ**

**ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

**(ДГТУ)**

Факультет «Информатика и вычислительная техника»

наименование факультета

Кафедра «Математика и информатика»

наименование кафедры

**Отчет**

по «Лабораторная работа №2»

Выполнил студент

группы МСК21

Леонавичус Даниил

Ростов-на-Дону

2024

**Цель работы**

Освоение основных видов отображений и использование геометрических преобразований для решения задач пространственной коррекции изображений.

**Теоретическое обоснование**

**Геометрические преобразования изображений**

Геометрические преобразования изображений подразумевают пространственное изменение местоположения множества пикселей с целочисленными координатами (x, y) в другое множество с координатами (x′, y′), причем интенсивность пикселей сохраняется. В двумерных плоских геометрических преобразованиях, как правило, используется евклидово пространство P2 с ортонормированной декартовой системой координат. В этом случае пикселю изображения соответствует пара декартовых координат, которые интерпретируются в виде двумерного вектора, представленного отрезком из точки (0, 0) до точки (x, y). Двумерные преобразования на плоскости можно представить в виде движения точек, соответствующих множеству пикселей.

Для общей совместимости с дальнейшими преобразованиями будем использовать однородные координаты, обладающие тем свойством, что определяемый ими объект не меняется при умножении всех координат на одно и то же ненулевое число. Из-за этого свойства необходимое количество координат для представления точек всегда на одну больше, чем размерность пространства Pn, в котором эти координаты используются. Например, для представления точки X = (x, y) на плоскости в двумерном пространстве P2 необходимо три координаты X = (x, y, w).

При помощи трех однородных координат можно описать любое линейное преобразование плоскости. Таким образом, геометрические преобразования являются матричными преобразованиями, множества координат пикселей преобразованного и исходного изображений связаны следующим матричным соотношением либо в строчном виде X′ = XT, либо в столбовом виде X′ = TX.

**Конформные преобразования**

Конформные преобразования — это отображения, при которых сохраняется форма бесконечно малых фигур и углы между кривыми в точках их пересечения. Основными линейными конформными преобразованиями являются евклидовы преобразования. К ним относятся сдвиг, отражение, однородное масштабирование и поворот. Конформные преобразования являются подмножеством аффинных преобразований.

**Аффинные преобразования**

Аффинные преобразования включают в себя линейные операции, такие как масштабирование, поворот и скос. В непрерывной геометрии любое аффинное преобразование имеет обратное аффинное преобразование, а произведение прямого и обратного дает единичное преобразование, которое оставляет все точки на месте. Аффинные преобразования являются подмножеством проекционных преобразований.

**Проекционные преобразования**

Проекционные преобразования включают в себя перспективные преобразования, где прямые линии остаются прямыми, но параллельные линии могут пересекаться. Проекционные преобразования могут быть параллельными (изменяется масштаб) или проективными (изменяется геометрия фигуры).

**Нелинейные преобразования**

Нелинейные преобразования включают в себя полиномиальные и синусоидальные преобразования. Полиномиальные преобразования используют полиномы для изменения координат пикселей, а синусоидальные преобразования используют синусоидальные функции для искажения изображения.

**Коррекция дисторсии**

Дисторсия — это оптическое искажение, выражающееся в искривлении прямых линий. Для коррекции дисторсии используются различные методы, такие как подушкообразная и бочкообразная дисторсия. Для вычисления коэффициентов коррекции используются пары соответствующих точек на исходном и искаженном изображениях.

**Профили и проекции изображения**

Профили и проекции изображения используются для анализа интенсивностей пикселей вдоль определенных линий или осей. Профиль вдоль линии — это функция интенсивности изображения, распределенного вдоль данной линии. Проекция на ось — это сумма интенсивностей пикселей изображения в направлении, перпендикулярном данной оси.

**Склейка изображений**

Склейка изображений подразумевает объединение двух изображений с пересекающимися областями в одно изображение. Для этого используются методы коррекции второго изображения для его перевода в систему координат первого и автоматическая склейка изображений.

Эти методы и преобразования позволяют эффективно обрабатывать и анализировать изображения, улучшая их качество и выделяя необходимые объекты.

**Ход выполнения работы**

**1. Простейшие геометрические преобразования**

**Исходное изображение**



**Листинг программной реализаций**

/\*

1. Простейшие геометрические преобразования. Выбрать произвольное изображение.

Выполнить над ним линейные и нелинейные преобразования (конформные, аффинные и проективные отображения)

\*/

#include <opencv2/opencv.hpp>

#include <iostream>

#include <cmath>

using namespace cv;

using namespace std;

int main() {

    // Загрузка изображения

    string imagePath = "background.png";

    Mat img = imread(imagePath);

    if (img.empty()) {

        cerr << "Ошибка: Не удалось загрузить изображение по пути: " << imagePath << endl;

        return -1;

    }

    // 1. Конформные преобразования

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

    // 1.1 Сдвиг

    Mat T\_shift = (Mat\_<double>(2, 3) <<

        1, 0, 50,

        0, 1, 30);

    Mat img\_shift;

    warpAffine(img, img\_shift, T\_shift, img.size());

    // 1.2 Однородное масштабирование

    Mat T\_scale = (Mat\_<double>(2, 3) <<

        1.5, 0, 0,

        0, 1.5, 0);

    Mat img\_scale;

    warpAffine(img, img\_scale, T\_scale, Size(int(img.cols \* 1.5), int(img.rows \* 1.5)));

    // 1.3 Поворот

    double angle = 45.0;

    Point2f center(img.cols / 2.0, img.rows / 2.0);

    Mat T\_rotate = getRotationMatrix2D(center, angle, 1.0);

    Mat img\_rotate;

    warpAffine(img, img\_rotate, T\_rotate, img.size());

    // 1.4 Отражение

    Mat T\_reflect = (Mat\_<double>(2, 3) <<

        1, 0, 0,

        0, -1, img.rows);

    Mat img\_reflect;

    warpAffine(img, img\_reflect, T\_reflect, img.size());

    // 2. Аффинные преобразования

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

    // 2.1 Скос

    Mat T\_shear = (Mat\_<double>(2, 3) <<

        1, 0.5, 0,

        0.2, 1, 0);

    Mat img\_shear;

    warpAffine(img, img\_shear, T\_shear, Size(img.cols \* 2, img.rows \* 2));

    // 2.2 Неоднородное масштабирование

    Mat T\_nonscale = (Mat\_<double>(2, 3) <<

        2.0, 0, 0,

        0, 0.5, 0);

    Mat img\_nonscale;

    warpAffine(img, img\_nonscale, T\_nonscale, Size(img.cols \* 2, img.rows / 2));

    // 3. Нелинейные преобразования

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

    // 3.1 Проективное преобразование

    Point2f srcPoints[4] = {

        Point2f(0, 0),

        Point2f(img.cols - 1, 0),

        Point2f(img.cols - 1, img.rows - 1),

        Point2f(0, img.rows - 1)

    };

    Point2f dstPoints[4] = {

        Point2f(30, 30),

        Point2f(img.cols - 30, 50),

        Point2f(img.cols - 50, img.rows - 30),

        Point2f(50, img.rows - 50)

    };

    Mat T\_projective = getPerspectiveTransform(srcPoints, dstPoints);

    Mat img\_projective;

    warpPerspective(img, img\_projective, T\_projective, img.size());

    // 3.2 Кусочно-линейное отображение

    Mat img\_piecewise = img.clone();

    Mat rightHalf = img\_piecewise(Rect(img.cols / 2, 0, img.cols / 2, img.rows));

    Mat T\_piecewise = (Mat\_<double>(2, 3) <<

        2.0, 0, 0,

        0, 1.0, 0);

    warpAffine(rightHalf, rightHalf, T\_piecewise, Size(rightHalf.cols \* 2, rightHalf.rows));

    // 3.3 Полиномиальное отображение

    Mat img\_polynomial;

    const double T[2][6] = {

        {0, 1, 0, 0.00001, 0.002, 0.002},

        {0, 0, 1, 0, 0, 0}

    };

    if (img.depth() == CV\_8U) {

        img.convertTo(img\_polynomial, CV\_32F, 1.0 / 255);

    } else {

        img\_polynomial = img.clone();

    }

    vector<Mat> channels;

    split(img\_polynomial, channels);

    for (int k = 0; k < channels.size(); k++) {

        Mat transformedChannel = Mat::zeros(channels[k].size(), channels[k].type());

        for (int x = 0; x < channels[k].cols; x++) {

            for (int y = 0; y < channels[k].rows; y++) {

                int newX = int(round(T[0][0] + x \* T[0][1] + y \* T[0][2] +

                                     x \* x \* T[0][3] + x \* y \* T[0][4] + y \* y \* T[0][5]));

                int newY = int(round(T[1][0] + x \* T[1][1] + y \* T[1][2] +

                                     x \* x \* T[1][3] + x \* y \* T[1][4] + y \* y \* T[1][5]));

                if (newX >= 0 && newX < channels[k].cols && newY >= 0 && newY < channels[k].rows) {

                    transformedChannel.at<float>(newY, newX) = channels[k].at<float>(y, x);

                }

            }

        }

        channels[k] = transformedChannel;

    }

    merge(channels, img\_polynomial);

    img\_polynomial.convertTo(img\_polynomial, CV\_8U, 255);

    // 3.4 Синусоидальное искажение

    Mat u = Mat::zeros(img.rows, img.cols, CV\_32F);

    Mat v = Mat::zeros(img.rows, img.cols, CV\_32F);

    for (int x = 0; x < img.cols; x++) {

        for (int y = 0; y < img.rows; y++) {

            u.at<float>(y, x) = float(x + 20 \* sin(2 \* M\_PI \* y / 90));

            v.at<float>(y, x) = float(y);

        }

    }

    Mat img\_sinusoid;

    remap(img, img\_sinusoid, u, v, INTER\_LINEAR);

    // Отображение всех результатов

    namedWindow("Оригинальное изображение", WINDOW\_NORMAL);

    imshow("Оригинальное изображение", img);

    namedWindow("Сдвиг", WINDOW\_NORMAL);

    imshow("Сдвиг", img\_shift);

    namedWindow("Однородное масштабирование", WINDOW\_NORMAL);

    imshow("Однородное масштабирование", img\_scale);

    namedWindow("Поворот", WINDOW\_NORMAL);

    imshow("Поворот", img\_rotate);

    namedWindow("Отражение (Ox)", WINDOW\_NORMAL);

    imshow("Отражение (Ox)", img\_reflect);

    namedWindow("Скос", WINDOW\_NORMAL);

    imshow("Скос", img\_shear);

    namedWindow("Неоднородное масштабирование", WINDOW\_NORMAL);

    imshow("Неоднородное масштабирование", img\_nonscale);

    namedWindow("Проективное преобразование", WINDOW\_NORMAL);

    imshow("Проективное преобразование", img\_projective);

    namedWindow("Кусочно-линейное отображение", WINDOW\_NORMAL);

    imshow("Кусочно-линейное отображение", img\_piecewise);

    namedWindow("Полиномиальное отображение", WINDOW\_NORMAL);

    imshow("Полиномиальное отображение", img\_polynomial);

    namedWindow("Синусоидальное искажение", WINDOW\_NORMAL);

    imshow("Синусоидальное искажение", img\_sinusoid);

    // Ожидание нажатия клавиши

    cout << "Нажмите 'Q' для выхода." << endl;

    while (true) {

        int key = waitKey(0);

        if (key == 'q' || key == 'Q') {

            break;

        }

    }

    return 0;

}

**Комментарии**

Конформные преобразования: Эти преобразования сохраняют углы между линиями, что позволяет изменять размер и положение изображения без искажения формы объектов.

* Сдвиг: Изображение смещено на 50 пикселей вправо и на 30 пикселей вниз.
* Однородное масштабирование: Изображение увеличено в 1.5 раза.
* Поворот: Изображение повернуто на 45 градусов вокруг центра.
* Отражение: Изображение отражено относительно горизонтальной оси.

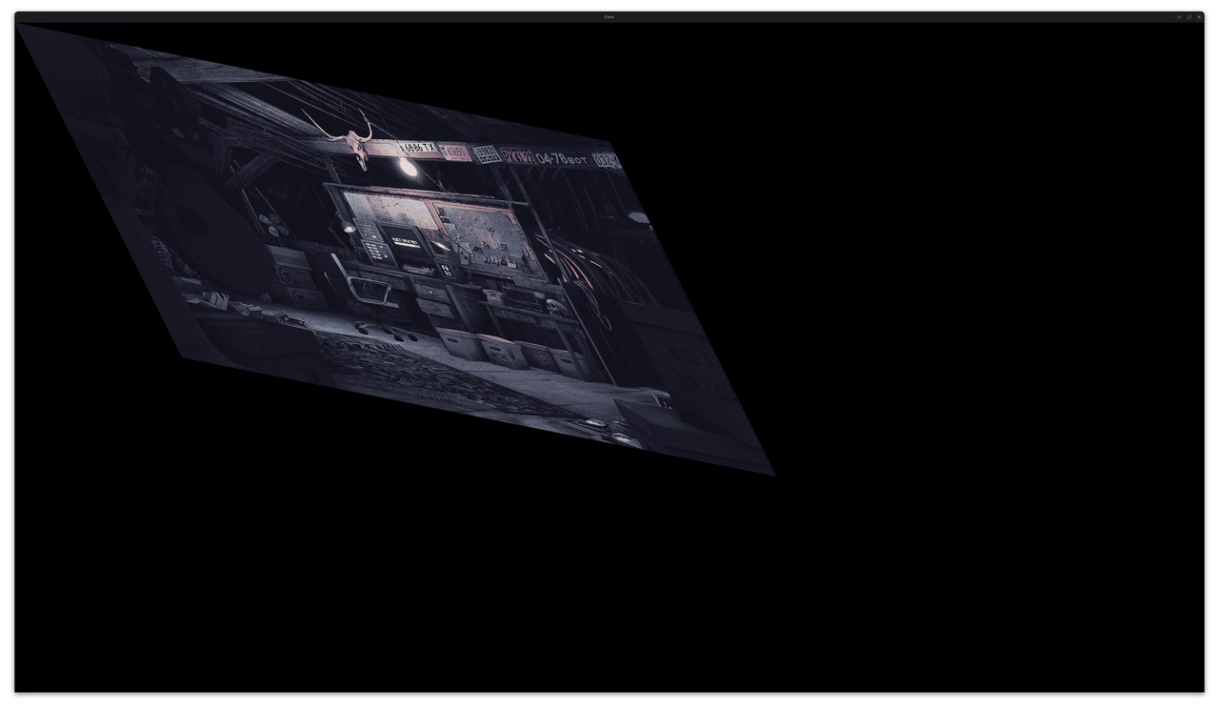
Аффинные преобразования: Эти преобразования включают в себя линейные операции, такие как масштабирование, поворот и скос.

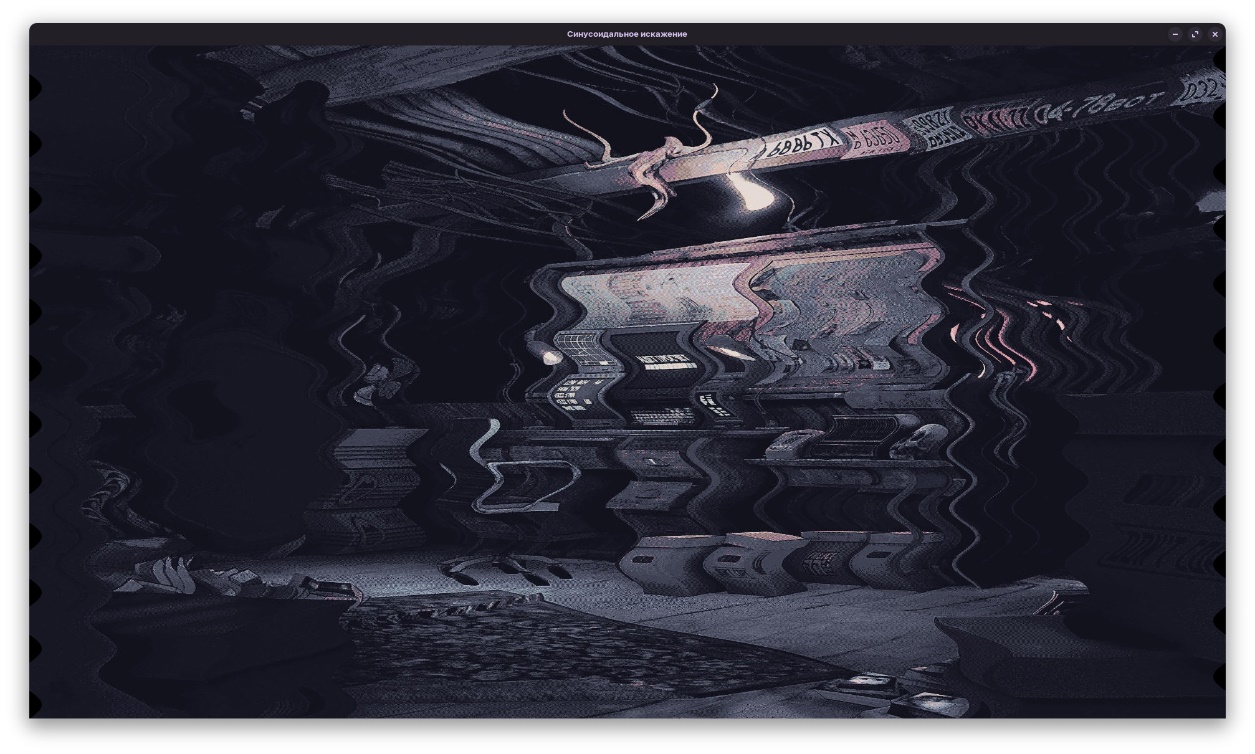
* Скос: Изображение искажено по горизонтали и вертикали.
* Неоднородное масштабирование: Изображение увеличено в 2 раза по горизонтали и уменьшено в 2 раза по вертикали.

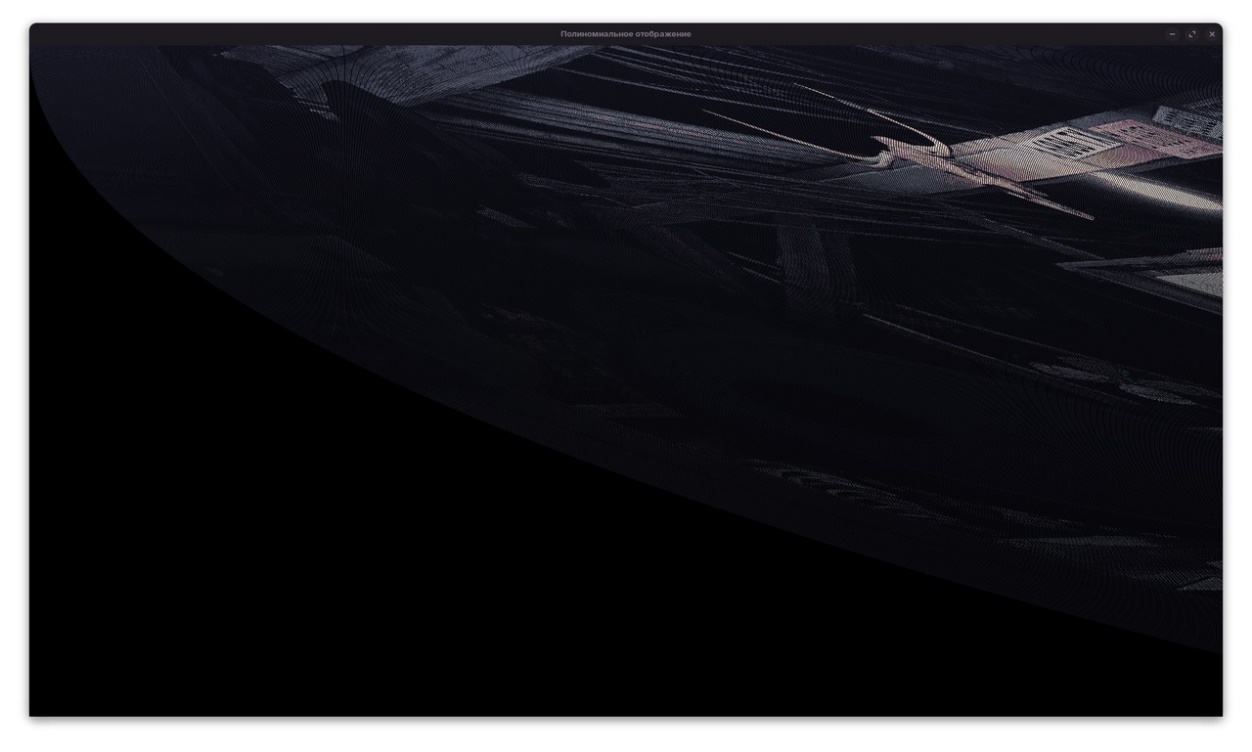
Нелинейные преобразования: Эти преобразования включают более сложные искажения, такие как проективные и полиномиальные отображения.

* Проективное преобразование: Изображение искажено с помощью перспективного преобразования.
* Кусочно-линейное отображение: Правая половина изображения увеличена в 2 раза.
* Полиномиальное отображение: Изображение искажено с помощью полиномиального преобразования.
* Синусоидальное искажение: Изображение искажено с помощью синусоидальной функции.

**Результирующие изображения**





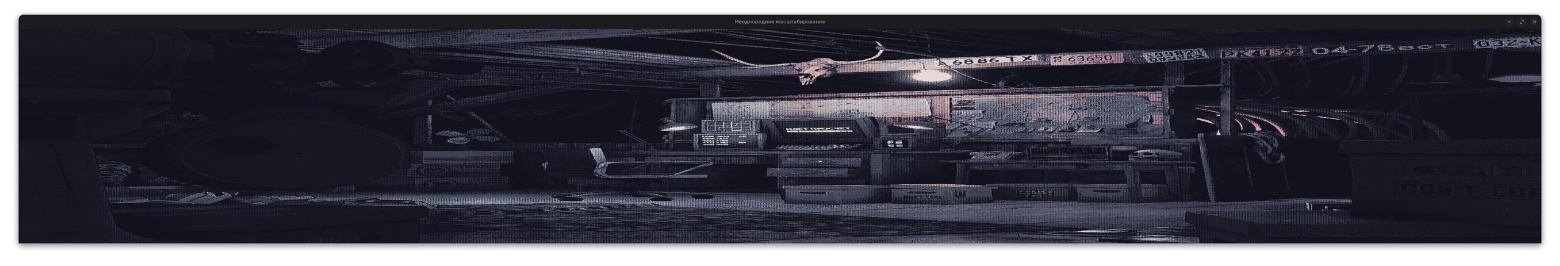












**2. Коррекция дисторсии**

**Исходное изображение**



**Листинг программной реализаций**

/\*

2. Коррекция дисторсии.

Выбрать произвольное изображение либо с подушкообразной, либо с бочкообразной дисторсией.

Выполнить коррекцию изображения.

\*/

#include <opencv2/opencv.hpp>

#include <iostream>

#include <vector>

using namespace cv;

using namespace std;

/\*\*

 \* Функция для применения бочкообразной дисторсии к изображению.

 \*

 \* @param img - Исходное изображение.

 \*/

void applyBarrelDistortion(Mat &img) {

    // Создаем сетку координат для изображения

    Mat xi, yi; // Матрицы координат X и Y

    vector<float> t\_x, t\_y;

    // Генерируем координаты столбцов и строк

    for (int i = 0; i < img.cols; i++) {

        t\_x.push\_back(float(i));

    }

    for (int i = 0; i < img.rows; i++) {

        t\_y.push\_back(float(i));

    }

    // Повторяем массивы для создания матриц координат

    repeat(Mat(t\_x).reshape(1, 1), img.rows, 1, xi);  // Горизонтальная сетка (xi)

    repeat(Mat(t\_y).reshape(1, 1).t(), 1, img.cols, yi);  // Вертикальная сетка (yi)

    // Смещение и нормализация координат

    double xmid = xi.cols / 2.0;  // Центр изображения по X

    double ymid = xi.rows / 2.0;  // Центр изображения по Y

    xi -= xmid; // Смещаем центр сетки к (0, 0)

    xi /= xmid; // Нормализуем координаты по ширине

    yi -= ymid; // Смещаем центр сетки к (0, 0)

    yi /= ymid; // Нормализуем координаты по высоте

    // Преобразуем в полярные координаты (радиус и угол)

    Mat r, theta;

    cartToPolar(xi, yi, r, theta);

    // Параметры для бочкообразной дисторсии

    double F3 = 0.1, F5 = 0.12; // Коэффициенты 3-го и 5-го порядка

    Mat r3, r5;

    pow(r, 3, r3); // Вычисляем r^3

    pow(r, 5, r5); // Вычисляем r^5

    // Применяем бочкообразное искажение: r = r + F3 \* r^3 + F5 \* r^5

    r += r3 \* F3;

    r += r5 \* F5;

    // Преобразуем обратно в декартовы координаты

    Mat u, v;

    polarToCart(r, theta, u, v);

    // Восстанавливаем координаты и возвращаем их в исходный диапазон

    u \*= xmid;

    u += xmid;

    v \*= ymid;

    v += ymid;

    // Применяем ремаппинг с учетом новых координат

    Mat I\_barrel;

    remap(img, I\_barrel, u, v, INTER\_LINEAR);

    // Отображаем результат

    namedWindow("Бочкообразная дисторсия", WINDOW\_NORMAL);

    imshow("Бочкообразная дисторсия", I\_barrel);

}

/\*\*

 \* Функция для применения подушкообразной дисторсии к изображению.

 \*

 \* @param img - Исходное изображение.

 \*/

void applyPillowDistortion(Mat &img) {

    // Создаем сетку координат для изображения

    Mat xi, yi; // Матрицы координат X и Y

    vector<float> t\_x, t\_y;

    // Генерируем координаты столбцов и строк

    for (int i = 0; i < img.cols; i++) {

        t\_x.push\_back(float(i));

    }

    for (int i = 0; i < img.rows; i++) {

        t\_y.push\_back(float(i));

    }

    // Повторяем массивы для создания матриц координат

    repeat(Mat(t\_x).reshape(1, 1), img.rows, 1, xi);  // Горизонтальная сетка (xi)

    repeat(Mat(t\_y).reshape(1, 1).t(), 1, img.cols, yi);  // Вертикальная сетка (yi)

    // Смещение и нормализация координат

    double xmid = xi.cols / 2.0;  // Центр изображения по X

    double ymid = xi.rows / 2.0;  // Центр изображения по Y

    xi -= xmid; // Смещаем центр сетки к (0, 0)

    xi /= xmid; // Нормализуем координаты по ширине

    yi -= ymid; // Смещаем центр сетки к (0, 0)

    yi /= ymid; // Нормализуем координаты по высоте

    // Преобразуем в полярные координаты (радиус и угол)

    Mat r, theta;

    cartToPolar(xi, yi, r, theta);

    // Параметр для подушкообразной дисторсии

    double F3 = -0.003; // Коэффициент 3-го порядка

    r += F3 \* r.mul(r); // Применяем подушкообразное искажение: r' = r + F3 \* r^2

    // Преобразуем обратно в декартовы координаты

    Mat u, v;

    polarToCart(r, theta, u, v);

    // Восстанавливаем координаты и возвращаем их в исходный диапазон

    u \*= xmid;

    u += xmid;

    v \*= ymid;

    v += ymid;

    // Применяем ремаппинг с учетом новых координат

    Mat I\_distorted;

    remap(img, I\_distorted, u, v, INTER\_LINEAR);

    // Отображаем результат

    namedWindow("Подушкообразная дисторсия", WINDOW\_NORMAL);

    imshow("Подушкообразная дисторсия", I\_distorted);

}

/\*\*

 \* Основная функция программы.

 \*/

int main() {

    // Загрузка изображения

    Mat img = imread("background.png");

    if (img.empty()) {

        cerr << "Ошибка загрузки изображения!" << endl;

        return -1;

    }

    // Отображаем исходное изображение

    namedWindow("Оригинальное изображение", WINDOW\_NORMAL);

    imshow("Оригинальное изображение", img);

    // Применяем бочкообразную дисторсию

    applyBarrelDistortion(img);

    // Применяем подушкообразную дисторсию

    applyPillowDistortion(img);

    cout << "Нажмите 'Q' для выхода." << endl;

    while (true) {

        int key = waitKey(0);

        if (key == 'q' || key == 'Q') {

            break;

        }

    }

    return 0;

}

**Комментарии**

* Бочкообразная дисторсия: Изображение искажено таким образом, что центральная часть выглядит выпуклой, а края загибаются внутрь. Это достигается путем изменения радиуса в полярных координатах с использованием коэффициентов 3-го и 5-го порядка.
* Подушкообразная дисторсия: Изображение искажено таким образом, что центральная часть выглядит вогнутой, а края загибаются наружу. Это достигается путем изменения радиуса в полярных координатах с использованием коэффициента 3-го порядка.

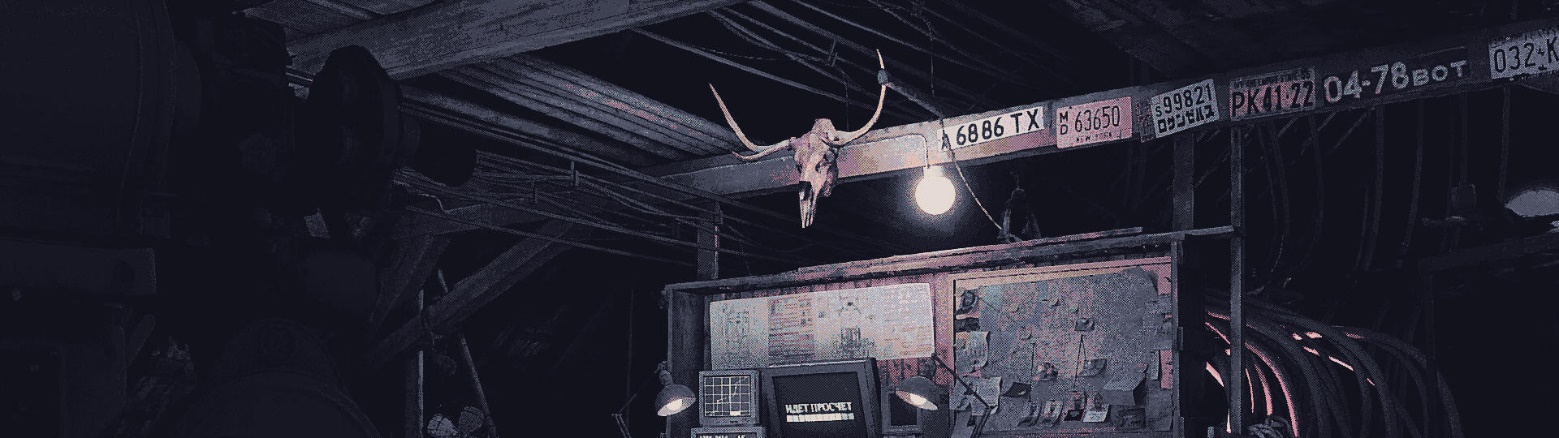
**Результирующие изображения**





**3. «Склейка» изображений**

**Исходное изображение**





**Листинг программной реализаций**

/\*

3. «Склейка» изображений. Выбрать два изображения (снимки с фотокамеры, фрагменты сканированного изображения и пр.), на которых имеется область пересечения.

Выполнить коррекцию второго изображения для его перевода в систему координат первого; затем выполнить автоматическую «склейку» из двух изображений в одно.

\*/

#include <opencv2/opencv.hpp>

#include <iostream>

using namespace cv;

using namespace std;

int main() {

    // Считывание изображений

    Mat topPart = imread("1.jpg", IMREAD\_COLOR); // Верхняя часть изображения

    Mat botPart = imread("2.jpg", IMREAD\_COLOR); // Нижняя часть изображения

    // Проверка на успешную загрузку изображений

    if (topPart.empty() || botPart.empty()) {

        cerr << "Ошибка загрузки изображений!" << endl;

        return -1;

    }

    // Определяем размер шаблона для поиска точки пересечения

    int templ\_size = 10; // Высота шаблона (в пикселях)

    if (topPart.rows < templ\_size) {

        cerr << "Верхняя часть изображения слишком мала для указанного размера шаблона!" << endl;

        return -1;

    }

    // Извлекаем нижнюю часть верхнего изображения в качестве шаблона

    Mat templ = topPart(Rect(0, topPart.rows - templ\_size, topPart.cols, templ\_size));

    // Поиск шаблона в нижнем изображении

    Mat res;

    matchTemplate(botPart, templ, res, TM\_CCOEFF); // TM\_CCOEFF используется для вычисления корреляции

    // Определение позиции максимального значения корреляции (точки пересечения)

    double min\_val, max\_val;

    Point min\_loc, max\_loc;

    minMaxLoc(res, &min\_val, &max\_val, &min\_loc, &max\_loc); // Получаем координаты точки максимума

    // Создаем итоговое изображение с учетом найденной точки склейки

    int overlap\_y = max\_loc.y + templ\_size; // Y-координата начала области пересечения

    int result\_height = topPart.rows + botPart.rows - overlap\_y; // Общая высота результата

    Mat result\_img = Mat::zeros(result\_height, topPart.cols, topPart.type()); // Итоговое изображение

    // Копируем верхнюю часть изображения в итоговое изображение

    topPart.copyTo(result\_img(Rect(0, 0, topPart.cols, topPart.rows)));

    // Копируем нижнюю часть изображения, начиная с точки пересечения

    botPart(Rect(0, overlap\_y, botPart.cols, botPart.rows - overlap\_y))

        .copyTo(result\_img(Rect(0, topPart.rows, botPart.cols, botPart.rows - overlap\_y)));

    // Отображаем результат

    namedWindow("Результат склейки", WINDOW\_NORMAL);

    imshow("Результат склейки", result\_img);

    // Сохранение результата

    imwrite("stitched\_result.jpg", result\_img); // Сохраняем итоговое изображение в файл

    cout << "Нажмите 'Q' для выхода." << endl;

    while (true) {

        int key = waitKey(0);

        if (key == 'q' || key == 'Q') {

            break;

        }

    }

    return 0;

}

**Комментарии**

* Склейка изображений: Два изображения успешно склеены в одно, с учетом области пересечения. Результат сохранен в файл stitched\_result.jpg. Программа использует шаблонное сопоставление для нахождения точки пересечения и объединяет изображения в одно, удаляя пересекающуюся область.

**Результирующие изображения**



**Выводы о проделанной работе**

В ходе выполнения работы были изучены и реализованы различные методы геометрических преобразований изображений, коррекции дисторсии и склейки изображений с использованием библиотеки OpenCV. Простейшие геометрические преобразования позволили изменить размер, положение и форму изображения. Коррекция дисторсии позволила исправить искажения изображения. Склейка изображений продемонстрировала возможность автоматического объединения двух изображений в одно, что полезно для создания панорамных изображений и других задач обработки изображений.

**Вопросы к защите лабораторной работе**

1. **Каким образом можно выполнить поворот изображения, не используя матрицу поворота?**

Поворот изображения можно выполнить без использования матрицы поворота, применяя метод ремаппинга (remapping). Этот метод позволяет определить новые координаты каждого пикселя в повернутом изображении и затем интерполировать значения пикселей из исходного изображения.

1. **Какое минимальное количество соответствующих пар точек необходимо задать на исходном и искаженном изображениях, если порядок преобразования 𝑛 = 4?**

Для преобразования порядка n=4 (проективное преобразование) необходимо минимум 4 пары соответствующих точек на исходном и искаженном изображениях. Проективное преобразование описывается матрицей размером 3x3, которая имеет 8 степеней свободы (9 элементов минус одна степень свободы для масштабирования). Чтобы однозначно определить эту матрицу, необходимо 4 пары соответствующих точек.

1. **После геометрического преобразования изображения могут появиться пиксели с неопределенными значениями интенсивности. С чем это связано и как решается данная проблема?**

После геометрического преобразования изображения могут появиться пиксели с неопределенными значениями интенсивности из-за того, что новые координаты пикселей могут не совпадать с координатами пикселей в исходном изображении. Это связано с тем, что преобразование может привести к появлению "дыр" в изображении, где нет соответствующих пикселей из исходного изображения.

Решение данной проблемы:

* **Интерполяция**: Использование методов интерполяции, таких как билинейная или бикубическая интерполяция, для заполнения неопределенных пикселей на основе значений соседних пикселей.
* **Заполнение фона**: Заполнение неопределенных пикселей заданным фоновым цветом или значением.
* **Экстраполяция**: Использование методов экстраполяции для оценки значений пикселей за пределами исходного изображения