Lab7 - 全景图拼接

学号：3210106034
姓名：王伟杰

> 实验环境：
>
> ```
> $ uname -a
> Linux ****** 5.15.0-101-generic #111~20.04.1-Ubuntu SMP Mon Mar 11 15:44:43 UTC 2024 x86_64 x86_64 x86_64 GNU/Linux
> $ python --version
> Python 3.9.19
> ```

实验内容

- 完成两组图像序列的拼接任务

理论分析

全景图拼接是利用同一场景的多张图像通过重叠部分寻找匹配关系，从而生成整个场景图像的技术。全景图的拼接方法有很多，如按场景和运动的种类可以分为单视点全景拼接和多视点全景拼接。

对于平面场景和只通过相机旋转拍摄的场景来说，可以使用求每两幅图像之间的一个 `Homography` 变换来映射到一张图像的方法，还可以使用恢复相机的旋转的方式得到最终的全景图。当相机固定只有水平方向旋转时，也可以使用柱面或球面坐标映射的方式求得全景图。

柱面坐标转换

柱面坐标转换用于解决由于相机透视变换导致的图像投影不一致问题。在全景图像拼接中，相机在拍摄过程中会产生视角的变化，这导致图像的投影方式也随之变化。为了消除这种变化，我们可以使用柱面坐标投影（也称为极坐标投影）将图像转换为柱面状，使得拼接后的图像具有一致的投影形式。柱面图像的坐标变换为：

$$
\begin{aligned}
\mathrm{x}' &= \mathrm{r}\tan^{-1}\left(\frac{\mathrm{x}}{\mathrm{f}}\right) \\
\mathrm{y}' &= \frac{\mathrm{ry}}{\sqrt{\mathrm{x}^2+\mathrm{f}^2}}
\end{aligned}
$$

其中(x', y')为柱面上的坐标, (x, y)为平面图像坐标，其坐标原点都已移至图像中心，r为柱面半径，f为焦距。

然而为了得到柱面投影图像，我们往往需要将柱面图像上的点逆变换到平面图像上的对应像素点，进行插值，得到完整的柱面图像，逆变换的变换公式为:

$$x = f\tan(\frac{x'}{r})$$

$$y = \frac{y'}{r}\sqrt{x^2 + f^2}$$

特征匹配

对每两幅相邻的柱面图像进行特征提取和匹配，寻找两幅相邻图像的对应关系。

SIFT特征匹配

SIFT特征通过检测图像中的关键点，并计算这些关键点周围区域的局部特征描述子。这些描述子具有尺度不变性和旋转不变性，因此适用于拼接不同视角或尺度的图像。在特征匹配阶段，通过比较不同图像中的特征描述子，可以找到相似的特征点，从而进行图像对齐和拼接。

ORB特征匹配

ORB算法结合了FAST特征检测和BRIEF特征描述，具有快速和鲁棒的特点。ORB同样能够提取图像中的关键点和对应的描述子，用于特征匹配。在全景图像拼接中，ORB可用于寻找相邻图像之间的重叠区域，进而实现准确的图像对齐。

全景图拼接

使用上一步得到的匹配关系，可以求出每两幅柱面图像的一个平移变换，利用平移变换将所有图像拼接到一起。得到一幅全景图。

但是在实际操作过程中，我发现平移变换无法很好的变换图片，让被拼接的图片对齐。这里我采用投影变换来替代平移变换，实现更高的复杂度，使得图片更好的被转换拼接。

实验细节

柱面坐标转换

`cylindricalWarp` 函数实现了将输入图像进行柱面投影变换的操作。该变换旨在校正由相机透视造成的图像失真，将图像投影到柱面上。通过计算柱面投影变换后的坐标，从原始图像中提取像素，并映射到目标图像上。这一步骤为后续图像拼接提供了校正后的图像数据。

```cpp
Mat Panorama6034::cylindricalWarp(const Mat& image, double f) const {
    int width = image.cols;
    int height = image.rows;
    int cx = (width - 1) / 2;
    int cy = (height - 1) / 2;
    int max_x = (int)(r * atan(cx / f));
    int max_y = (int)(r * cy / f);
    Mat result = Mat::zeros(max_y * 2 + 1, max_x * 2 + 1, CV_8UC3);

    for (int i = 0; i < max_x * 2 + 1; ++i) {
        for (int j = 0; j < max_y * 2 + 1; ++j) {
            double x = f * tan((i - max_x) / r);
            double y = (j - max_y) / r * sqrt(x * x + f * f);
            x += cx;
            y += cy;
            if (x ⩾ 0 && x < width && y ⩾ 0 && y < height) {
                result.at<Vec3b>(j, i) = image.at<Vec3b>((int)y, (int)x);
            }
        }
    }
    return result;
}
```

特征提取与匹配

在`findH`函数中，利用SIFT（尺度不变特征变换）算法提取图像的特征点和描述子。首先使用SIFT算法检测关键点，然后计算每个关键点的描述子。随后，利用暴力匹配器（Brute-Force Matcher）对两幅图像的特征描述子进行匹配，并通过筛选出的良好匹配点集来估计透视变换矩阵（Homography Matrix）。

实验过程中我使用了ORB和SIFT两种检测器，发现在提取特征时，ORB检测算法实现的速度较快，但SIFT可以检测到效果更好的特征点，由于我们的任务是静态图片拼接，所以我选择了SIFT作为实现方法。

这里我取了匹配到的点中距离小于`max(2 * min_dist, 0.5 * max_dist)`的点作为计算`H`用到的点。

```cpp
Mat Panorama6034::findH(const Mat &img1, const Mat &img2) {
    Ptr<SIFT> sift = SIFT::create();
    std::vector<KeyPoint> keypoints1, keypoints2;
    Mat descriptors1, descriptors2;
    sift->detectAndCompute(img1, Mat(), keypoints1, descriptors1);
```

```
    sift->detectAndCompute(img2, Mat(), keypoints2, descriptors2);

    BFMatcher matcher;
    std::vector<DMatch> matches;
    matcher.match(descriptors1, descriptors2, matches);

    std::vector<DMatch> good_matches;
    float min_dist = matches[0].distance, max_dist = matches[0].distance;
    for (const auto & match : matches) {
        min_dist = std::min(min_dist, match.distance);
        max_dist = std::max(max_dist, match.distance);
    }
    for (const auto & match : matches) {
        if (match.distance <= std::max(2 * (double)min_dist, 0.5 * max_dist))
{
            good_matches.push_back(match);
        }
    }
    if (good_matches.size() < 4) {
        return Mat();
    }
    Mat img_matches;
    drawMatches(img1, keypoints1, img2, keypoints2, good_matches,
img_matches);
    imshow("matches", img_matches);
    waitKey(0);

    std::vector<Point2f> points1, points2;
    for (auto & good_match : good_matches) {
        points1.push_back(keypoints1[good_match.queryIdx].pt);
        points2.push_back(keypoints2[good_match.trainIdx].pt);
    }

    Mat H = findHomography(points2, points1, RANSAC);
    return H;
}
```

图像拼接

`mergeImages` 函数实现了将经过柱面投影变换后的图像进行拼接的操作。在该函数中，首先估计两幅图像之间的透视变换矩阵，并利用透视变换将第二幅图像映射到第一幅图像的视角上。然后根据透视变换后的图像位置信息，将两幅图像进行融合，生成拼接后的全景图像。

这里注意在转换的过程中，有几个需要注意的点：

1. 遍历图片进行转换时需要遍历结果图片，而不是被转换的图片，否则容易留下空洞。
2. 图片的大小通过被转换图片的四个角点确定，然后需要对结果图片进行坐标系平移以适应转换后的图片。
3. 在图片序列中，从中间开始向两边拼接可以提升结果的观赏性，不会出现一边大一边小的情况。
4. 之前拼接好的结果不作为被拼接的图片，扭曲的次数过多会导致失真。

```
bool Panorama6034::makePanorama(std::vector<cv::Mat>& img_vec, cv::Mat&
img_out, double f) {
    if (img_vec.empty()) {
        return false;
    }
    std::vector<Mat> warped_images;
    for (const auto& img : img_vec) {
        warped_images.push_back(cylindricalWarp(img, f));
    }
    int mid = (img_vec.size() - 1) / 2;
    img_out = warped_images[mid];
    for (int i = mid - 1; i ⩾ 0; --i) {
        img_out = mergeImages(img_out, warped_images[i]);
        if (img_out.empty()) return false;
    }
    for (int i = mid + 1; i < img_vec.size(); ++i) {
        img_out = mergeImages(img_out, warped_images[i]);
        if (img_out.empty()) return false;
    }

    return true;
}

Mat Panorama6034::mergeImages(Mat &img1, Mat &img2) {
    Mat H = findH(img1, img2);
    if (H.empty()) {
        return Mat();
    }

    std::vector<Point2f> corners(4);
    corners[0] = Point2f(0, 0);
    corners[1] = Point2f((float)img2.cols - 1, 0);
    corners[2] = Point2f((float)img2.cols - 1, (float)img2.rows - 1);
    corners[3] = Point2f(0, (float)img2.rows - 1);
```

```
    perspectiveTransform(corners, corners, H);
    int right = (int)std::max(corners[1].x, corners[2].x);
    right = std::max(right + 1, img1.cols);
    int down = (int)std::max(corners[2].y, corners[3].y);
    down = std::max(down + 1, img1.rows);
    int left = (int)std::min(corners[0].x, corners[3].x);
    left = std::min(left, 0);
    int up = (int)std::min(corners[0].y, corners[1].y);
    up = std::min(up, 0);

    int result_col = right - left;
    int result_row = down - up;

    Mat result = Mat::zeros(result_row, result_col, CV_8UC3);

//    warpPerspective(img1, result, H, Size(result_col, result_row));
    std::vector<Point2f> p(1);
    Mat reverse_H = H.inv();
    for (int i = 0; i < result_row; ++i) {
        for (int j = 0; j < result_col; ++j) {
            if (result.at<Vec3b>(i, j) != Vec3b(0, 0, 0)) {
                continue;
            }
            int j_trans = j + left;
            int i_trans = i + up;
            p[0] = Point2f(j_trans, i_trans);
            perspectiveTransform(p, p, reverse_H);
            int x = (int)p[0].x;
            int y = (int)p[0].y;
            if (x >= 0 && x < img2.cols && y >= 0 && y < img2.rows &&
img2.at<Vec3b>(y, x) != Vec3b(0, 0, 0)) {
                result.at<Vec3b>(i, j) = img2.at<Vec3b>(y, x);
            }
        }
    }
    for (int i = 0; i < img1.rows; ++i) {
        for (int j = 0; j < img1.cols; ++j) {
            Vec3b img1_pixel = img1.at<Vec3b>(i, j);
            Vec3b result_pixel = result.at<Vec3b>(i - up, j - left);
            if (norm(img1_pixel) > norm(result_pixel) ) {
                result.at<Vec3b>(i - up, j - left) = img1_pixel;
            }
        }
```

```
76          }
77          return result;
78      }
```

结果展示

第一组图片：

第二组图片：