Descriptor

2317038

허지원

개요

Descriptor는 OpenCV의 SIFT 알고리즘을 활용하여 두 이미지 간의 특징점을 추출하고, 이 특징점들의 Descriptor를 이용하여 이미지 간 매칭을 수행하는 프로젝트이다.

SIFT(Scale Invariant Feature Transform)

: 이미지의 특징점을 검출하고 기술하는 알고리즘으로, 이미지의 크기, 회전, 조명 변화에 대해 강인하게 동작하는 특징이 있다. 이러한 장점 덕분에 동일한 객체의 특징점을 안정적으로 검출할 수 있다.

SIFT 과정

1. Scale Space Extrema Detection

: LoG(Laplacian of Gaussian)을 근사화 한 DoG(Difference of Gaussian)을 사용하여 Blob 검출 방식을 구현한다.

$$D(x, y, \sigma) = \big(G(x, y, k\sigma) - G(x, y, \sigma)\big) * I(x, y)$$

2. Key Point Localization

: 이미지의 특징점을 정확하게 찾고 낮은 대비를 가진 불안정한 특징점을 제거한다.

1) 테일러 급수 전개를 이용하여 서브 픽셀의 정확도 위치를 추정한다

→ 특징점의 위치와 크기를 보정하여 낮은 대비와 에지의 특징점을 제거한다.

$$D(X) \cong D + \frac{\partial D^T}{\partial X} X + \frac{1}{2} X^T \frac{\partial^2 D}{\partial X^2} X$$

$$\hat{X} = -\left(\frac{\partial^2 D}{\partial X^2}\right)^{-1} \frac{\partial D}{\partial X}$$

- $D = D(x, y, \sigma)$: Difference of Gaussian
- $\hat{X}$: 극값을 통해 얻은 서브 픽셀의 정확도 위치

2) Hassian 행렬을 이용하여 에지의 특징점을 제거한다.

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}, \qquad r = \frac{Trace(H)^2}{|H|} = \frac{(\lambda_1 + \lambda_2)^2}{\lambda_1 \lambda_2} = \frac{\left(1 + \frac{\lambda_2}{\lambda_1}\right)^2}{\frac{\lambda_2}{\lambda_1}}$$

- $r > specific\ threshold$ 인 경우 에지로 판단하고 제거한다.

3. Orientation Assignment

: 한 픽셀에 대한 각도를 구하기 위해서 16X16 윈도우에서 그래디언트 방향 히스토그램을 생성한다.

$$L(x,y) = G(x,y,\sigma) * I(x,y)$$

$$m(x,y) = \sqrt{\big(L(x+1,y) - L(x-1,y)\big)^2 + \big(L(x,y+1) - L(x,y-1)\big)^2}$$

$$\theta(x,y) = \tan^{-1}\big((L(x,y+1) - L(x,y-1))/(L(x+1,y) - L(x-1,y))\big)$$

- 가우시안 가중치를 적용한 히스토그램에서 주된 방향을 선택한다.

4. Descriptor Consturction

: 검출된 특징점 주변의 지역적 패턴을 수치화한다.

1) 4X4 서브 블록에서 8방향 그래디언트 히스토그램 계산
2) (16개의 서브 블록)X(8방향) = 128차원의 벡터
3) 정규화 후 높은 그래디언트의 영향을 제거하기 위해서 클리핑을 진행

코드에 대한 설명

SIFT 구현 프로그램

1) 초기 설정 및 라이브러리

```cpp
#include <opencv2/opencv.hpp>
// #include <opencv2/nonfree/nonfree.hpp>
#include <opencv2/features2d.hpp>

#include <iostream>
#include <vector>
#include <cmath>

#define RATIO_THR 0.4

using namespace std;
using namespace cv;
```

A. 헤더 파일: OpenCV와 SIFT 알고리즘을 위한 헤더 파일을 포함
B. 매크로 정의: 최근접 이웃 거리 비율 필터링의 기준이 되는 RATIO_THR 정의
C. 네임스페이스

2) 함수 선언

```cpp
double euclidDistance(const Mat& vec1, const Mat& vec2);
int nearestNeighbor(const Mat& vec, const vector<KeyPoint>& keypoints, const Mat& descriptors);
void findPairs(vector<KeyPoint>& keypoints1, Mat& descriptors1,
    vector<KeyPoint>& keypoints2, Mat& descriptors2,
    vector<Point2f>& srcPoints, vector<Point2f>& dstPoints, bool crossCheck, bool ratio_threshold);
```

A. euclidDistance: 두 특징 벡터 간 유클리드 거리를 계산하기 위한 함수 선언
B. nearestNeighbor: 입력된 벡터와 가장 유사한 최근접 이웃 특징점을 탐색하기 위한 함수 선언
C. findPairs: 양방향 매칭 및 필터링을 수행하기 위한 함수 선언

3) main 함수

```
int main() {

    Mat input1 = imread("input1.jpg", IMREAD_COLOR);
    Mat input2 = imread("input2.jpg", IMREAD_COLOR);
    Mat input1_gray, input2_gray;

    if (!input1.data || !input2.data)
    {
        std::cout << "Could not open" << std::endl;
        return -1;
    }

    resize(input1, input1, Size(input1.cols / 2, input1.rows / 2));
    resize(input2, input2, Size(input2.cols / 2, input2.rows / 2));

    cvtColor(input1, input1_gray, COLOR_RGB2GRAY);
    cvtColor(input2, input2_gray, COLOR_RGB2GRAY);

    Ptr<SIFT> sift = SIFT::create(

        0,      // nFeatures
        4,      // nOctaveLayers
        0.04,   // contrastThreshold
        10,     // edgeThreshold
        1.6     // sigma
    );
```

A. 이미지 로드 및 변환: input(입력 이미지) 변수에 이미지 주소를 받아 원본 이미지들을 로드한 후 cvtColor()를 통해 grayscale 이미지로 변환
B. 에러 핸들링
C. 이미지 축소: 원본 이미지들을 가로, 세로 크기의 절반으로 축소
D. SIFT 검출기 생성
   i. 파라미터
   - nFeatures: 검출할 최대 특징점 수 (0 = 제한 없음)
   - nOctaveLayers: 이미지 피라미트의 계층 수
   - contrastThreshold: 낮은 대비의 특징점 필터링 임계 값
   - edgeThreshold: 에지 응답 임계 값
   - sigma: 초기 이미지의 스무딩 정도를 나타내는 표준편차

```
// Create a image for displaying mathing keypoints
Size size = input2.size();
Size sz = Size(size.width + input1_gray.size().width, max(size.height, input1_gray.size().height));
Mat matchingImage = Mat::zeros(sz, CV_8UC3);

input1.copyTo(matchingImage(Rect(size.width, 0, input1_gray.size().width, input1_gray.size().height)));
input2.copyTo(matchingImage(Rect(0, 0, size.width, size.height)));

// Compute keypoints and descriptor from the source image in advance
vector<KeyPoint> keypoints1;
Mat descriptors1;

sift->detectAndCompute(input1_gray, noArray(), keypoints1, descriptors1);

printf("input1 : %d keypoints are found.\n", (int)keypoints1.size());

vector<KeyPoint> keypoints2;
Mat descriptors2;
```

A. 매칭 결과 표시를 위해 두 이미지를 나란히 배치한 이미지 생성
   i. 너비 = input2 너비 + input1 너비
   ii. 높이 = 두 이미지 중 더 큰 높이
B. 이미지 병합: 왼쪽 상단에 input2, 오른쪽 상단에 input1을 배치
C. 첫 번째 이미지(오른쪽)의 특징점 추출
   i. detectAndCompute: keypoint 검출과 descriptor 계산
      - 파라미터
        - input1_gray: grayscale의 입력 이미지
        - noArray(): 마스크를 사용하지 않음을 나타내는 빈 행렬
        - keypoints1: 검출된 특징점 저장 벡터
        - descriptors1: 128차원의 SIFT descriptor 행렬 (각 행이 하나의 특징점에 대응)

```
// Detect keypoints

sift->detectAndCompute(input2_gray, noArray(), keypoints2, descriptors2);
printf("input2 : %zd keypoints are found.\n", keypoints2.size());

for (int i = 0; i < keypoints1.size(); i++) {
    KeyPoint kp = keypoints1[i];
    kp.pt.x += size.width;
    circle(matchingImage, kp.pt, cvRound(kp.size*0.25), Scalar(255, 255, 0), 1, 8, 0);
}

for (int i = 0; i < keypoints2.size(); i++) {
    KeyPoint kp = keypoints2[i];
    circle(matchingImage, kp.pt, cvRound(kp.size*0.25), Scalar(255, 255, 0), 1, 8, 0);
}
```

A. 두 번째 이미지(왼쪽) 특징점 추출
B. for문을 활용하여 첫 번째 이미지의 특징점 시각화
   i. 좌표 변환: input1 특징점을 오른쪽 영역에 표시하기 위해 X 좌표에 input2 너비 추가
   ii. 특징점을 시각화 하는 원 생성
C. for문을 활용하여 두 번째 이미지의 특징점 시각화: 특징점을 시각화 하는 원 생성

```
    // Find nearest neighbor pairs
    vector<Point2f> srcPoints;
    vector<Point2f> dstPoints;
    bool crossCheck = true;
    bool ratio_threshold = true;
    findPairs(keypoints2, descriptors2, keypoints1, descriptors1, srcPoints, dstPoints, crossCheck, ratio_threshold);
    printf("%zd keypoints are matched.\n", srcPoints.size());

    // Draw line between nearest neighbor pairs
    for (int i = 0; i < (int)srcPoints.size(); ++i) {
        Point2f pt1 = srcPoints[i];
        Point2f pt2 = dstPoints[i];
        Point2f from = pt1;
        Point2f to = Point(size.width + pt2.x, pt2.y);
        line(matchingImage, from, to, Scalar(0, 0, 255));
    }

    // Display mathing image
    namedWindow("Matching");
    imshow("Matching", matchingImage);


    waitKey(0);
    return 0;
}
```

A. findPairs 함수 호출을 통한 매칭 쌍 검출: keypoints2(소스)→keypoints1(타겟) 방향으로 매칭을 수행하여 결과를 저장
B. 매칭 결과 시각화: 오른쪽 이미지의 점 x 좌표에 왼쪽 이미지의 원본 너비를 더하여 오른쪽 영역에 배치 → 두 특징점을 빨간 선으로 연결
C. 결과 이미지 표시

4) euclidDistance 함수

```
/**
* Calculate euclid distance
*/
double euclidDistance(const Mat& vec1, const Mat& vec2) {
    double sum = 0.0;
    int dim = vec1.cols;
    for (int i = 0; i < dim; i++) {
        sum += (vec1.at<float>(0, i) - vec2.at<float>(0, i)) * (vec1.at<float>(0, i) - vec2.at<float>(0, i));
    }

    return sqrt(sum);
}
```

A. 함수 파라미터
   i. vec1, vec2: 각 객체들의 한 행에서의 여러 열을 가진 벡터
B. 벡터의 열의 수(dim) 계산: vec1.cols를 통해 벡터의 차원 확인
C. for문을 활용하여 각 열의 수마다의 차이의 제곱을 누적
D. 제곱근을 취하여 최종 거리를 반환

5) nearestNeighbor 함수

```cpp
/**
* Find the index of nearest neighbor point from keypoints.
*/
int nearestNeighbor(const Mat& vec, const vector<KeyPoint>& keypoints, const Mat& descriptors) {
    int neighbor = -1;
    double minDist = 1e6;

    for (int i = 0; i < descriptors.rows; i++) {
        Mat v = descriptors.row(i);     // each row of descriptor

        //
        //  Fill the code
        double Dist = euclidDistance(vec, v);
        if( Dist < minDist) {
            minDist = Dist;
            neighbor = i;
        }
        //

    }

    return neighbor;
}
```

A. 함수 파라미터
   i. vec: 비교 대상이 되는 특징 descriptor
   ii. keypoints: 후보 특징점의 목록
   iii. descriptors: keypoints의 descriptor 집합
B. 변수 초기화
   i. neighbor 인덱스 초기화
   ii. minDist를 큰 값으로 초기화
C. 최소 거리 계산: for문을 활용하여 모든 descriptor을 순회하여 거리를 계산한 후 최소 거리를 갱신할 때마다 인덱스(neighbor) 저장

6) findPairs 함수

```cpp
/**
* Find pairs of points with the smallest distace between them
*/
void findPairs(vector<KeyPoint>& keypoints1, Mat& descriptors1,
    vector<KeyPoint>& keypoints2, Mat& descriptors2,
    vector<Point2f>& srcPoints, vector<Point2f>& dstPoints, bool crossCheck, bool ratio_threshold) {
    for (int i = 0; i < descriptors1.rows; i++) {
        KeyPoint pt1 = keypoints1[i];
        Mat desc1 = descriptors1.row(i);

        int nn = nearestNeighbor(desc1, keypoints2, descriptors2); // gk에서 k 값
```

A. 함수 파라미터
   - keypoints1, descriptors1: 소스 이미지의 특징점 데이터
   - keypoints2, descriptors2: 타겟 이미지의 특징점 데이터
   - srcPoints, dstPoints: 소스/타겟 이미지의 매칭점 좌표
   - crossCheck: 양방향 검증 방식의 활용 여부
   - ratio_threshold: 최적과 차선의 매칭 거리 비율 비교 방식의 활용 여부
B. 최근접 이웃 탐색: for문을 활용하여 소스 이미지의 각 특징점에 대한 타겟 이미지에서의 최근접 이웃(nn) 탐색

```cpp
// Refine matching points using ratio_based thresholding
if (ratio_threshold) {
    //
    //  Fill the code
    vector<double> Dists;
    for (int j = 0; j < descriptors2.rows; j++) {
        Mat desc2 = descriptors2.row(j);
        double Dist = euclidDistance(desc1, desc2);
        Dists.push_back(Dist);
    }

    sort(Dists.begin(), Dists.end());
    double dist2 = Dists[1];

    double dist1 = euclidDistance(desc1, descriptors2.row(nn));
    if ((dist1 / dist2) > RATIO_THR) {
        continue;
    }
    //
}
```

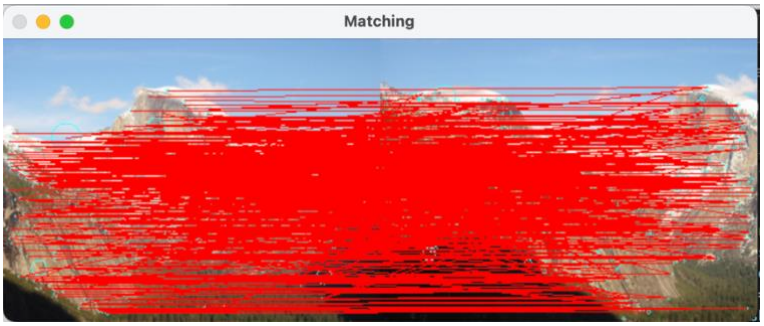
C. ratio 테스트 방식을 활용할 경우
   i. for문을 활용하여 소스 이미지의 특징점과 타겟 이미지의 모든 특징점들 사이의 거리를 계산하여 Dists에 담은 후 정렬
   ii. 최적의 거리(dist1)와 차선의 거리(dist2) 비율을 검사
   → RATIO_THR 초과 시 매칭을 무시

```cpp
    // Refine matching points using cross-checking
    if (crossCheck) {
        //
        //  Fill the code
        Mat desc2 = descriptors2.row(nn);
        int nn2 = nearestNeighbor(desc2, keypoints1, descriptors1);

        if( nn2 != i) {
            continue;
        }
        //
    }
    KeyPoint pt2 = keypoints2[nn];
    srcPoints.push_back(pt1.pt);
    dstPoints.push_back(pt2.pt);
}
}
```

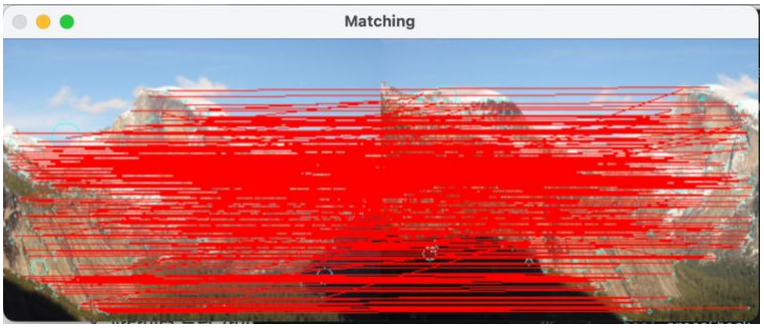
D. cross-check 방식을 활용할 경우
   i. 타겟 이미지의 각 특징점에 대한 소스 이미지에서의 최근접 이웃(nn2) 탐색
   ii. 양방향 매칭(nn2 == i)이 일치하지 않으면 무시
E. 유효한 매칭을 srcPoints, dstPoints에 각각 저장

7) 실행 결과
A. cross-check와 ratio_threshold 방식을 모두 비활성화한 경우
   → input1 : 702 keypoints are found.
   input2 : 618 keypoints are found.
   618 keypoints are matched.



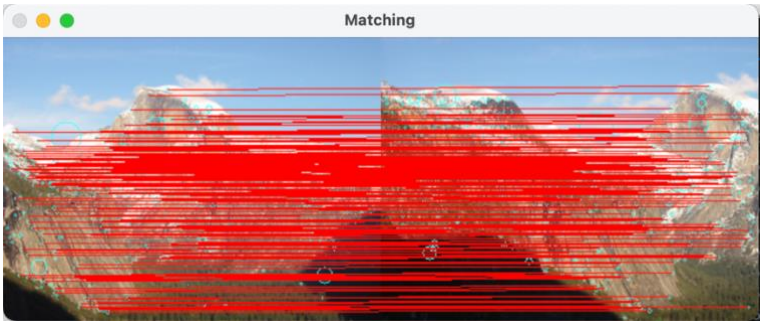
: 타겟 이미지의 모든 특징점들이 매칭되었다.

B. cross-check 방식만 활성화한 경우
→ input1 : 702 keypoints are found.
input2 : 618 keypoints are found.
340 keypoints are matched.



: 양방향 검증을 통해 Case A와 비교하여 270개의 특징점을 제거하였다.

C. cross-check와 ratio_threshold 방식을 모두 활성화한 경우
→ input1 : 702 keypoints are found.
input2 : 618 keypoints are found.
209 keypoints are matched.



: 모호한 매칭 제거와 양방향 검증을 통해 Case A와 비교하여 409개의 특징점을 제거하였다.

- Case1 vs Case2: Cross-Check 방식으로 매칭에 대한 필터링을 진행하였을 때 일방향으로만 매칭이 된 특징점들을 제거함으로써 정확도를 향상시켰다.
- Case1 vs Case3: Ratio-Threshold 방식을 추가하여 필터링을 진행하였을

때 최적/차선 거리의 비율에 따라 모호한 매칭을 제거함으로써 정확도를 크게 향상시켰다.