기말 프로젝트

프로젝트 주요 진행 :

1. 이미지 순서대로 입력시(인덱스 순서대로 이미지가 결합되기 때문에 순서 중요합니다) 해당 이미지로 파노라마 이미지 구현
2. detectandcompute함수 구현

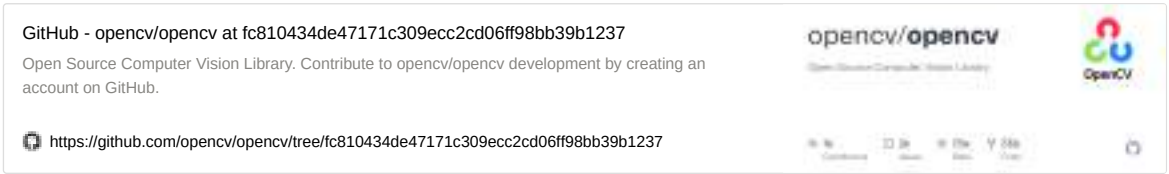
이용한 라이브러리 : `opencv`

구현한 함수의 라이브러리 및 docs

프로젝트 설정 :

해당 설정은 사용자 마다 다양할 수 있습니다. 편한대로 직접 사용하시면 됩니다.

1. opencv를 설치합니다. C 드라이브에 다운로드 해서 위치하는 것을 추천합니다.
2. opencv와 vs 설정을 해줘야합니다.
   a. 일반 C++ 언어 표준은 17로 설정해줘야합니다.
   b. C/C++ 추가 포함 디렉토리 : 해당 라이브러리의 Include 위치를 설정해줘야합니다.
   c. 링커의 추가 종속성에 해당 opencv의 버전에 맞추어서 .lib파일 넣어줘야합니다.
   d. 라이브러리 자체에 opencv_world470.dll와 opencv_world470d.dll 파일을 넣어줘야합니다.
3. 아래 라이브러리 설정을 해줘야합니다.



   1. 링크할 수 있는 파일의 위치에 저장합니다. 똑같이 위의 설정과 마찬가지로 링크 해줍니다.
   2. 해당 라이브러리의 필요한 것들을 cmake을 진행해줍니다. (솔직히 계속 이것 저것 다 넣어서 좀 헷갈립니다.)
   3. module파일과 연결이 필요합니다. module의 필요한 파일을 직접 일일이 연결할 수 있지만 module 전체를 솔류션과 연결하는 것이 가장 간편합니다.

만약 이 과정을 진행해도 안된다면 경로상의 한국어가 있는 지 확인거나 환경변수 설정을 확인 또는 컴퓨터를 다시 시작하는 것을 추천합니다. 그래도 해당 문제가 해결이 되지 않는다면, jsilvercastle@naver.com이나 010-3571-9455로 연락주시면 됩니다.

프로젝트 실행 영상

AI GameProgramming

https://www.youtube.com/watch?v=N4keC5X777I

주요코드 설명

1. **프로젝트의 시작에 있어서 값을 넘기기 위해 Cylinder 형식으로 변환 (참고함)**

```
// Cylinder로 치환. (참고 한 코드임)
ProjectOntoCylinder(secImage, secImageCylindrical, maskX, maskY);
cv::Mat secImageMask = cv::Mat::zeros(secImageCylindrical.rows, secImageCylindrical.cols, secImageCylindrical.type());

for (int i = 0; i < maskX.size(); i++)
{
  cv::Vec3b& value = secImageMask.at<cv::Vec3b>(maskY[i], maskX[i]);
  for (int k = 0; k < secImageCylindrical.channels(); k++)
    value.val[k] = 255;
}
```

2. **DetectAndCompute 함수 구현을 위한 초기 작업**

```
// SIFT를 사용하여 이미지에서 키포인트와 디스크립터를 찾음
Ptr<SIFT> Sift = SIFT::create();
cv::Mat BaseImage_des, SecImage_des;
cv::Mat BaseImage_Gray, SecImage_Gray;

// Gray이미지가 특징점과 descriptor를 찾을 수 있는 유용한 방법임
// -> gray scale로 이미지 변환 진행
cv::cvtColor(baseImage, BaseImage_Gray, cv::COLOR_BGR2GRAY);
cv::cvtColor(secImageCylindrical, SecImage_Gray, cv::COLOR_BGR2GRAY);
```

3. **DetectAndCompute 함수로 구분하는 분기점 도달**

```
// 문제의 부분 : detectAndCompute 함수
if (version == 1)
{
  Sift->detectAndCompute(BaseImage_Gray, cv::noArray(), baseImageKeyPoints, BaseImage_des);
  Sift->detectAndCompute(SecImage_Gray, cv::noArray(), secImageKeyPoints, SecImage_des);
}
if (version == 2)
{
    MydetectAndCompute(BaseImage_Gray, cv::noArray(), baseImageKeyPoints, BaseImage_des);
    MydetectAndCompute(SecImage_Gray, cv::noArray(), secImageKeyPoints, SecImage_des);
}
```

4. Flann 의 KnnMatch를 이용해서 matching 진행

```
// 매치를 찾기 해당 방법과 bfs 방법이 있는데 해당 구현에서는 FlannbaseMatcher를 사용함.
cv::FlannBasedMatcher Flann_Matcher;
std::vector<std::vector<cv::DMatch>> InitialMatches;
// opencv function 사용
Flann_Matcher.knnMatch(BaseImage_des, SecImage_des, InitialMatches, 2);
```

5. **특정한 값을 기준으로 값을 필터링 및 처리함**

```
// Ratio Test를 적용하여 좋은 매치를 필터링
const float distanceRatioThreshold = 0.75;
std::vector<cv::DMatch> goodMatches;
for (size_t i = 0; i < InitialMatches.size(); ++i)
{
  // 첫번째 거리가 두번째 거리(distanceRatioThreshold를 곱함) 보다 작은지 확인
  if (InitialMatches[i][0].distance < distanceRatioThreshold * InitialMatches[i][1].distance)
    goodMatches.push_back(InitialMatches[i][0]);
/
```

```
// 잘 매칭된 이미지의 위치에 대해서 base와 sec에 값을 집어 넣음
std::vector<cv::Point2f> baseImagePoints, secImagePoints;
for (const cv::DMatch& match : goodMatches)
{
  baseImagePoints.push_back(baseImageKeyPoints[match.queryIdx].pt);
  secImagePoints.push_back(secImageKeyPoints[match.trainIdx].pt);
}
// 영상 왜곡 보정을 위한 호모그래피 행렬 계산
cv::Mat homographyMatrix = cv::findHomography(secImagePoints, baseImagePoints, cv::RANSAC, 4.0);
```

6. 행렬 단순 계산

```
// 초기 행렬 설정
double initialMatrix[3][4] = { {0, (double)width - 1, (double)width - 1, 0},
                               {0, 0, (double)height - 1, (double)height - 1},
                               {1.0, 1.0, 1.0, 1.0} };
cv::Mat initialMatrixMat = cv::Mat(3, 4, CV_64F, initialMatrix);

// 최종 행렬 계산
cv::Mat finalMatrix = homographyMatrix * initialMatrixMat;

// x, y, c 좌표 추출
cv::Mat x = finalMatrix(cv::Rect(0, 0, finalMatrix.cols, 1));
cv::Mat y = finalMatrix(cv::Rect(0, 1, finalMatrix.cols, 1));
cv::Mat c = finalMatrix(cv::Rect(0, 2, finalMatrix.cols, 1));
```

7. 좌표 보정 단계

```
// x, y 좌표를 c 좌표로 나누어줌
cv::Mat xByC = x.mul(1 / c);
cv::Mat yByC = y.mul(1 / c);

// x, y 좌표의 최소값과 최대값 계산
double minX, maxX, minY, maxY;
cv::minMaxLoc(xByC, &minX, &maxX);
cv::minMaxLoc(yByC, &minY, &maxY);
minX = (int)round(minX); maxX = (int)round(maxX);
minY = (int)round(minY); maxY = (int)round(maxY);
```

8. 좌표 보정 및 변환 단계

```
// 이미지가 음수인 경우 보정
if (minX < 0)
{
    newWidth -= minX;
    correction[0] = abs(minX);
}
if (minY < 0)
{
    newHeight -= minY;
    correction[1] = abs(minY);
}

// 새로운 이미지 크기 보정
newWidth = (newWidth < baseImageShape[1] + correction[0]) ? baseImageShape[1] + correction[0] : newWidth;
newHeight = (newHeight < baseImageShape[0] + correction[1]) ? baseImageShape[0] + correction[1] : newHeight;

// x, y 좌표에 보정값을 더해줌
cv::add(xByC, correction[0], xByC);
cv::add(yByC, correction[1], yByC);

// 초기 좌표와 최종 좌표 설정
cv::Point2f oldInitialPoints[4], newFinalPoints[4];
oldInitialPoints[0] = cv::Point2f(0, 0);
oldInitialPoints[1] = cv::Point2f(width - 1, 0);
oldInitialPoints[2] = cv::Point2f(width - 1, height - 1);
oldInitialPoints[3] = cv::Point2f(0, height - 1);
```

9. 호모그래피 계산 및 적용

```
// 호모그래피 행렬 계산
for (int i = 0; i < 4; i++)
  newFinalPoints[i] = cv::Point2f(xByC.at<double>(0, i), yByC.at<double>(0, i));
homographyMatrix = cv::getPerspectiveTransform(oldInitialPoints, newFinalPoints);

// 새로운 프레임 크기 설정
```

```
newFrameSize[0] = newHeight; newFrameSize[1] = newWidth;

// 보정된 이미지에 호모그래피 변환 적용
cv::Mat secImageTransformed, secImageTransformedMask;
cv::warpPerspective(secImageCylindrical, secImageTransformed, homographyMatrix, cv::Size(newFrameSize[1], newFrameSize[0]));
cv::warpPerspective(secImageMask, secImageTransformedMask, homographyMatrix, cv::Size(newFrameSize[1], newFrameSize[0]));
```

10. **이미지 합성**

```
// 기준 이미지와 같은 크기의 빈 이미지 생성
cv::Mat baseImageTransformed = cv::Mat::zeros(newFrameSize[0], newFrameSize[1], baseImage.type());

// 보정된 이미지를 복사하여 기준 이미지에 붙여넣기
baseImage.copyTo(baseImageTransformed(cv::Rect(correction[0], correction[1], baseImage.cols, baseImage.rows)));

// 이미지 합성
cv::Mat stitchedImage = baseImageTransformed.clone();
cv::addWeighted(stitchedImage, 0.8, secImageTransformed, 0.8, 0.0, stitchedImage);
// 완성된 이미지 반환
return stitchedImage;
```

11. **DetectAndCompute 함수 (해당 함수 내에 구현되어 있는 몇개의 함수는 다른 라이브러리나 블로그의 코드를 가져왔기에 해당 설명은 생략하겠습니다)**

```
void MydetectAndCompute(InputArray inputImage, InputArray inputMask, std::vector<KeyPoint>& keyPoints, OutputArray outputDescriptors)
{
  int firstOctave = 0, actualNOctaves = 0;
  Mat image = inputImage.getMat(), mask = inputMask.getMat();

  // Keypoint 필터링
  // 영상 경계 내에 있는 유효한 키포인트만을 값으로 보존함.
  std::vector<KeyPoint> filteredKeypoints;
  for (size_t i = 0; i < keyPoints.size(); i++)
  {
    KeyPoint& kp = keyPoints[i];
    if (kp.pt.x >= 0 && kp.pt.x < image.cols && kp.pt.y >= 0 && kp.pt.y < image.rows)
    {
      filteredKeypoints.push_back(kp);
    }
  }

  // 키포인트 스케일에 따라 정렬
  std::sort(filteredKeypoints.begin(), filteredKeypoints.end(), [](const KeyPoint& a, const KeyPoint& b) {
    return a.size < b.size;
    });

  int maxOctave = INT_MIN;
  for (size_t i = 0; i < filteredKeypoints.size(); i++)
  {
    int octave = filteredKeypoints[i].octave & 255;
    int layer = (filteredKeypoints[i].octave >> 8) & 255;

    // octave 값을 0~127 또는 -128~-1 사이로 변환
    octave = octave < 128 ? octave : (-128 | octave);

    float scale;
    if (octave >= 0)
    {
      // octave가 0 이상인 경우 scale을 계산
      scale = 1.f / (1 << octave);
    }
    else
    {
      // octave가 음수인 경우 scale을 계산
      scale = static_cast<float>(1 << -octave);
    }

    // 최소 octave와 최대 octave 갱신
    firstOctave = std::min(firstOctave, octave);
    maxOctave = std::max(maxOctave, octave);
  }


  // 최소 octave 값을 0과 비교하여 갱신합니다.
  firstOctave = std::min(firstOctave, 0);

  // actualNOctaves는 최대 octave와 최소 octave를 기반으로 계산됩니다.
  // 최대 octave와 최소 octave 사이의 개수를 계산합니다.
  actualNOctaves = maxOctave - firstOctave + 1;
```

```
  // 가우시안 피라미드를 저장할 벡터를 생성합니다.
  std::vector<Mat> gaussianPyramid;

  // 이미지를 기반으로 초기 이미지를 생성합니다.
  // firstOctave 값이 0보다 작은 경우, 이미지의 크기에 따라 옥타브 개수를 조정합니다.
  // 초기 이미지는 (float)1.6 크기로 생성되며, true로 설정되어 이미지를 배경으로 사용합니다.
  Mat base = createInitialImage(image, firstOctave < 0, (float)1.6, true);

  int numOctaves;
  if (actualNOctaves > 0)
  {
    // actualNOctaves 값이 0보다 큰 경우, 해당 값을 그대로 사용합니다.
    numOctaves = actualNOctaves;
  }
  else
  {
    // actualNOctaves 값이 0인 경우, 이미지의 크기에 따라 옥타브 개수를 계산합니다.
    // 로그를 이용하여 최적의 옥타브 개수를 결정합니다.
    numOctaves = cvRound(std::log(static_cast<double>(std::min(base.cols, base.rows))) / std::log(2.) - 2) - firstOctave;
  }

  // 가우시안 피라미드를 생성합니다. (해당 함수가 있어서 참고해서 대신 사용함)
  buildGaussianPyramid(base, gaussianPyramid, numOctaves);


  if (outputDescriptors.needed())
  {
    // 디스크립터의 크기를 지정합니다.
    int descriptorSize = 128;
    Mat descriptors;

    // calcDescriptors 멀티스레딩
    // 멀티스레드로 calcDescriptors 함수를 호출하여 디스크립터를 계산합니다.
    // 각 스레드는 100개의 키포인트를 처리합니다.
    std::vector<std::thread> threads;
    for (int i = 0; i < filteredKeypoints.size(); i += 100)
    {
      int endIndex = std::min(i + 100, static_cast<int>(filteredKeypoints.size()));

      // 스레드를 생성하여 calcDescriptors 함수를 실행합니다.
      threads.emplace_back([&gaussianPyramid, &filteredKeypoints, &descriptors, descriptorSize, firstOctave, i, endIndex]() {
        calcDescriptors(gaussianPyramid, std::vector<KeyPoint>(filteredKeypoints.begin() + i, filteredKeypoints.begin() + endIndex), d
        });
    }

    // 생성한 모든 스레드가 종료될 때까지 기다립니다.
    for (auto& thread : threads)
      thread.join();

    // outputDescriptors 행렬을 생성합니다.
    // 크기는 필터링된 키포인트의 개수로 설정되고, 디스크립터 크기와 타입은 고정됩니다.
    outputDescriptors.create(static_cast<int>(filteredKeypoints.size()), descriptorSize, 5);

    // outputDescriptors 행렬에 계산된 디스크립터를 복사합니다.
    Mat output = outputDescriptors.getMat();
    descriptors.copyTo(output);
  }

}
```

소스 이미지






이미지 결합 결과

이미지 : 원본 함수 사용

카페 이미지

데스크탑 이미지

이미지 : 구현 함수 사용

카페 이미지

데스크탑 이미지

확인할 수 있다 싶이 특징점이 많은 cafe의 이미지 결합의 경우 충분한 특징점이 있기 때문에 내가 만든 함수로 구현을 진행해도 크게 차이점이 없었지만, 특징점이 상대적으로 적고 맞추기가 힘든 desktop 이미지의 경우에는 파노라마 이미지가 잘 출력이 안되는 것을 확인 할 수 있다. 완벽하게 구현을 하고 싶었지만, 이미지의 특징점의 위치의 특수성이나 개수에 따라서 이미지 결합의 결과가 달라져 아쉽긴 했다.

결론 :

완벽하게 sift의 해당함수를 구현하는 것은 불가능이지만 해당함수를 공부하면서 많을 것을 배울 수 있었다. 이번 프젝을 구현하기 위해 사용하지 않았어도 공부한 내용들도 많은 도움이 되었다. corner detection부터 DoG, Gaussian 피라미드, 디스크립터 벡터 등등 많은 것들의 의미와 활용을 해보와 너무 좋았다.

아쉬운 점은 구현이 복잡해지고 어려운 이미지 간의 결합에서는 좋은 결과를 내지 못한 다는 점이 아쉬웠고, 기회가 된다면 더 업그래이드 해서 제출해보고 싶다는 생각이 들었다.