《计算机视觉》

实验文档

（实验六）

学 院 名 称 ：	数据科学与计算机学院
专业（班级）：	16 软件工程（数字媒体）
学 生 姓 名 ：	黎汛言
学　　号 ：	16340109
时　　间 ：	2018　年　12　月　02　日

实验五 ：Panorama Image Stitching

一. 实验目的

1. 了解SIFT的原理，学会提取SIFT特征点
2. 学会使用kd-tree进行特征匹配
3. 掌握RANSAC算法的原理和实现方法
4. 了解图像拼接和图像融合的方法

二. 实验内容

1. 输入一系列连续图像，提取每个图像的SIFT特征点；
2. 使用kd-tree进行相邻图片的特征匹配；
3. 使用RANSAC算法筛选匹配的特征点；
4. 根据RANSAC拟合出来的模型，进行图像拼接。

三. 实验环境

Windows 10 64位，Visual Studio 2017

四. 实验过程

1. SIFT特征提取（Feature.h和Feature.cpp）

   这部分使用了vlfeat这个计算机视觉的库来提取图像的SIFT特征。我将SIFT特征提取的操作封装在类Feature中，能够提取单张图像的SIFT特征：

```
class Feature {
private:
    CImg<unsigned char> src;        // 原图像
    CImg<float> img;                // 灰度图像
    vector<VlSiftKeypoint> kpts;    // SIFT关键点的集合
    vector<float*> descriptors;     // SIFT关键点描述子的集合

public:
    Feature();
    Feature(CImg<unsigned char> image);

    void toGreyScale();                             // 将图像转为灰度图
    void extractFeatures();                         // 提取图像的SIFT特征

    // getter and setter
    vector<VlSiftKeypoint> getKeypoints() { return kpts; }
    vector<float*> getDescriptors() { return descriptors; }
    CImg<unsigned char> getImage() { return src; }
    void setKeypoints(vector<VlSiftKeypoint> k) { kpts = k; }
    void setDescriptors(vector<float*> d) { descriptors = d; }
};
```

转化灰度图的过程比较简单，如下：

```
void Feature::toGreyScale()
{
    cimg_forXY(src, x, y) {
        int r = src(x, y, 0);
        int g = src(x, y, 1);
        int b = src(x, y, 2);
        int grey = (r * 30 + g * 59 + b * 11 + 50) / 100;
        img(x, y) = grey;
    }
}
```

特征提取的过程如下：

```cpp
void Feature::extractFeatures() {
    toGreyScale();

    // 初始化VlSiftFilt，总共计算4个塔，每个塔的尺度空间有3层
    VlSiftFilt* vl_sift = vl_sift_new(img.width(), img.height(), 4, 3, 0);
    // 设置剔除不稳定关键点的阈值
    vl_sift_set_peak_thresh(vl_sift, 0.04);
    vl_sift_set_edge_thresh(vl_sift, 10);
    // 将灰度图的数据作为SIFT特征提取所需的数据
    vl_sift_pix *data = (vl_sift_pix*)(img.data());

    // 在每一个塔，计算特征点和特征点描述子
    if (vl_sift_process_first_octave(vl_sift, data) != VL_ERR_EOF) {
        while (true) {
            vl_sift_detect(vl_sift);
            // 得到关键点
            VlSiftKeypoint* pKpts = vl_sift->keys;

            for (int i = 0; i < vl_sift->nkeys; i++) {
                double angles[4];

                // 计算特征点的方向，包括主方向和辅方向，最多4个
                int angleCount = vl_sift_calc_keypoint_orientations(vl_sift, angles, pKpts);

                // 对于方向多于一个的特征点，每个方向分别计算特征描述子
                // 并且将特征点复制多个
                for (int i = 0; i < angleCount; i++) {
                    float *des = new float[128];
                    vl_sift_calc_keypoint_descriptor(vl_sift, des, pKpts, angles[0]);
                    descriptors.push_back(des);
                    kpts.push_back(*pKpts);
                }
                pKpts++;
            }
            // 处理下一个塔
            if (vl_sift_process_next_octave(vl_sift) == VL_ERR_EOF) {
                break;
            }
        }
    }
    // 释放资源
    vl_sift_delete(vl_sift);
}
```

提取到的SIFT特征点示例：

2. 特征匹配（Match.h和Match.cpp）

这部分进行了特征点的匹配和匹配特征点的过滤。输入两个图像，使用kd-tree进行SIFT特征点匹配，找到匹配对。然后使用RANSAC算法，排除Outlier，拟合出最佳的转换模型，得到单应矩阵H。

定义一个struct，来表示匹配点对：

```
struct pointPair {
    VlSiftKeypoint a;
    VlSiftKeypoint b;
    pointPair(VlSiftKeypoint _a, VlSiftKeypoint _b) {
        a = _a;
        b = _b;
    }
};
```

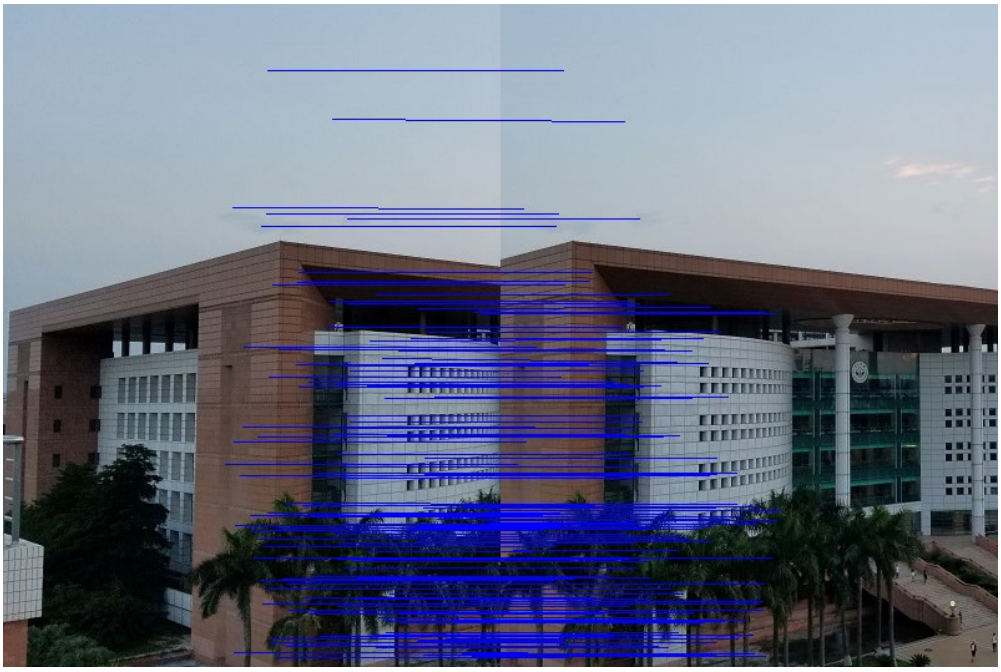
将特征匹配和RANSAC操作封装在类Match中：

```
class Match {
private:
    // 两幅待匹配图像
    CImg<unsigned char> img1;
    CImg<unsigned char> img2;

    // 图像的特征点集合
    vector<VlSiftKeypoint> kpts1;
    vector<VlSiftKeypoint> kpts2;

    // 图像的特征点描述子集合
    vector<float*> descriptors1;
    vector<float*> descriptors2;

    // 匹配点对和过滤后的匹配点对
    vector<pointPair> matchPoints;
    vector<pointPair> realMatchPoints;

    // 拟合出的单应矩阵
    CImg<double> H;

public:
    Match(CImg<unsigned char> img1, vector<VlSiftKeypoint> kpts1, vector<float*> descriptors1,
        CImg<unsigned char> img2, vector<VlSiftKeypoint> kpts2, vector<float*> descriptors2);
    void findMatchPoints(); // 特征点匹配
    void showMatchPoints(); // 显示匹配点
    CImg<float> RANSAC();   // RANSAC过程
};
```

kd-tree特征点匹配过程如下，同样使用了vlfeat这个库：

```
void Match::findMatchPoints()
{
    // 初始化kd森林
    VlKDForest* forest = vl_kdforest_new(VL_TYPE_FLOAT, 128, 1, VlDistanceL1);
    // 获取img1全部特征点描述子的数据
    float *data = new float[128 * descriptors1.size()];
    for (int i = 0; i < descriptors1.size(); i++) {
        for (int j = 0; j < 128; j++) {
            data[i * 128 + j] = descriptors1[i][j];
        }
    }
    // 建立kd森林
    vl_kdforest_build(forest, descriptors1.size(), data);

    VlKDForestSearcher* searcher = vl_kdforest_new_searcher(forest);
    VlKDForestNeighbor neighbours[2];

    // 遍历img2的特征点描述子集合，寻找img1的最近邻特征点
    for (int i = 0; i < descriptors2.size(); i++) {
        float *tempData = new float[128];
        for (int j = 0; j < 128; j++) {
            tempData[j] = descriptors2[i][j];
        }
        vl_kdforestsearcher_query(searcher, neighbours, 2, tempData);
        float ratio = neighbours[0].distance / neighbours[1].distance;
        if (ratio < 0.5) {
            VlSiftKeypoint left = kpts1[neighbours[0].index];
            VlSiftKeypoint right = kpts2[i];
            matchPoints.push_back(pointPair(left, right));
        }
        delete[] tempData;
        tempData = NULL;
    }
    // 释放资源
    vl_kdforestsearcher_delete(searcher);
    vl_kdforest_delete(forest);
    delete[] data;
    data = NULL;
}
```

匹配示例：



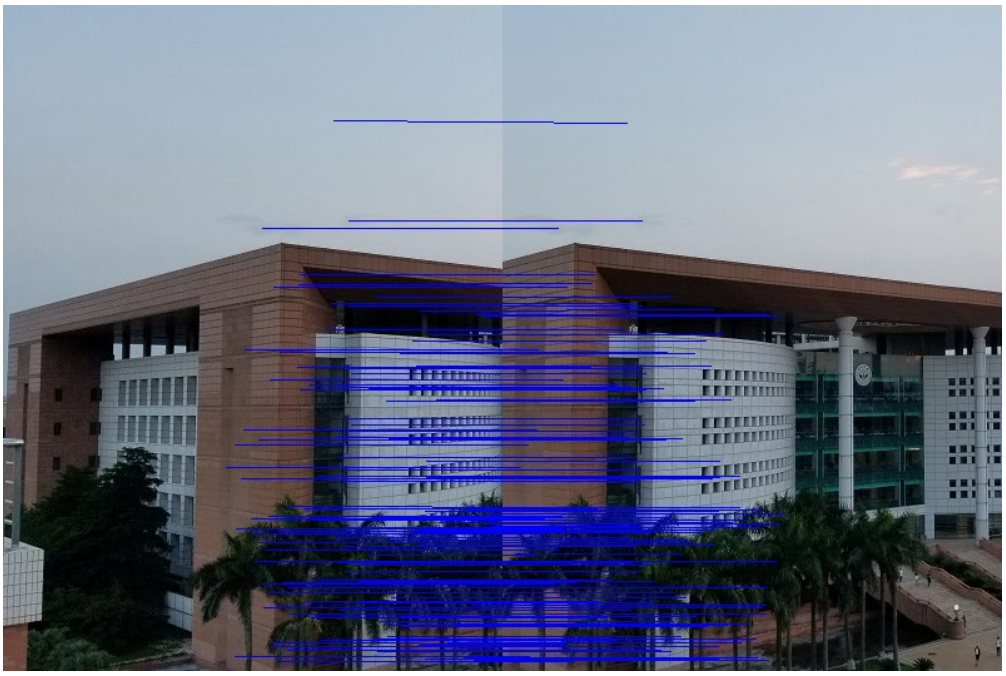
RANSAC算法的实现如下，实现方式见注释：

```
CImg<float> Match::RANSAC()
{
    int interation = numberOfIterations(0.99, 0.5, 4);  // 计算循环次数
    vector<int> maxInliersIndexes;                      // Inlier最多的模型所包含的Inlier的下标

    for (int i = 0; i < interation; i++) {
        vector<pointPair> randomPairs;
        set<int> indexes;
        vector<int> inliersIndexes;
        // 随机取4对匹配点
        for (int j = 0; j < 4; j++) {
            int index = random(0, matchPoints.size() - 1);
            while (indexes.find(index) != indexes.end()) {
                index = random(0, matchPoints.size() - 1);
            }
            indexes.insert(index);
            randomPairs.push_back(matchPoints[index]);
        }
        // 从4对匹配点中计算单应矩阵H
        CImg<float> Homography = calculateHomography(randomPairs);
        // 对每一对匹配点，使用H计算匹配点的位置，并计算与其与实际匹配点的距离，统计Inlier的数量
        for (int j = 0; j < matchPoints.size(); j++) {
            float x1 = matchPoints[j].a.x; float y1 = matchPoints[j].a.y;
            float x_1 = matchPoints[j].b.x; float y_1 = matchPoints[j].b.y;
            CImg<float> xy1(1, 3, 1, 1, 1);
            xy1(0) = x1;
            xy1(1) = y1;
            CImg<float> target = Homography * xy1;
            target /= target(2);
            float target_x = target(0);
            float target_y = target(1);
            // 计算距离，统计Inlier
            float distance = sqrt(pow(target_x - x_1, 2) + pow(target_y - y_1, 2));
            if (distance < 1) {
                inliersIndexes.push_back(j);
            }
        }
        // 找出Inlier最多的模型
        if (inliersIndexes.size() > maxInliersIndexes.size()) {
            maxInliersIndexes = inliersIndexes;
            H = Homography;
        }
    }
    for (int i = 0; i < maxInliersIndexes.size(); i++) {
        realMatchPoints.push_back(matchPoints[maxInliersIndexes[i]]);
    }
    return H;
}
```

其中，使用CImg对象来表示矩阵。矩阵的求解过程使用了CImg的solve方法：

```
CImg<float> calculateHomography(vector<pointPair> randomPairs) {
    int size = randomPairs.size();
    CImg<float> x(1, 8, 1, 1, 0); // 单应矩阵的参数向量
    CImg<float> b(1, 2 * size, 1, 1, 0);
    CImg<float> A(8, 2 * size, 1, 1, 0);

    for (int i = 0; i < randomPairs.size(); i++) {
        float x1 = randomPairs[i].a.x; float y1 = randomPairs[i].a.y;
        float x_1 = randomPairs[i].b.x; float y_1 = randomPairs[i].b.y;
        A(0, i * 2) = x1; A(1, i * 2) = y1; A(2, i * 2) = 1; A(6, i * 2) = -(x1 * x_1); A(7, i * 2) = -(y1 * x_1);
        A(3, i * 2 + 1) = x1; A(4, i * 2 + 1) = y1; A(5, i * 2 + 1) = 1; A(6, i * 2 + 1) = -(x1 * y_1); A(7, i * 2 + 1) = -(y1 * y_1);
        b(i * 2) = x_1; b(i * 2 + 1) = y_1;
        //cout << "(" << x1 << ", " << y1 << "), (" << x_1 << ", " << y_1 << ")" << endl;
    }
    // 对于Ax = b, 求解x
    x = b.solve(A);
    CImg<float> H(3, 3, 1, 1, 0);
    cimg_forY(x, y) {
        H(y) = x(0, y);
    }
    H(2, 2) = 1;
    return H;
}
```

使用RANSAC后得到的匹配点示例：



3. 图像拼接（Stitch.h和Stitch.cpp）

这部分实现了图像的拼接。输入两张图像，先计算拼接后的图像大小，然后分别计算前向映射和反向映射的变换矩阵，利用矩阵变换，实现图像拼接。

将拼接操作封装在类Stitch中：

```
class Stitch {
private:
    CImg<unsigned char> dst;        // 原图像
    CImg<unsigned char> src;        // 待拼入的图像
    CImg<float> homographyForward;  // 前向映射矩阵
    CImg<float> homographyBackward; // 后向映射矩阵
    CImg<unsigned char> stitched;   // 拼接后的图像
    vector<VlSiftKeypoint> kpts;    // 待拼入图像的特征点集合
    vector<float*> descriptors;     // 待拼入图像的特征点描述子集合
public:
    Stitch(CImg<unsigned char> dst, CImg<unsigned char> src,
        CImg<float> hF, CImg<float> hB, vector<VlSiftKeypoint> kpts, vector<float*> descriptors);
    int calculateWidthAndHeight();  // 根据两幅图像的信息，计算拼接图像的大小
    Feature stitch();               // 图像拼接
};
```

拼接过程的实现：

```
Feature Stitch::stitch()
{
    // 计算拼接后图像的宽和高
    int yOffset = calculateWidthAndHeight();
    // 将原图移动到拼接后位置
    cimg_forXYC(dst, x, y, c) {
        if (ceil(y+yOffset) < stitched.height())
            stitched(x, y + yOffset, c) = dst(x, y, c);
    }
    // 对待拼接图像的特征点进行前向映射，移动特征点
    for (int i = 0; i < kpts.size(); i++) {
        CImg<float> xy1(1, 3, 1, 1, 1);
        xy1(0) = kpts[i].x;
        xy1(1) = kpts[i].y;
        CImg<float> target = homographyForward * xy1;
        target /= target(2);
        kpts[i].x = target(0);
        kpts[i].y = target(1) + yOffset;
        kpts[i].ix = (int)kpts[i].x;
        kpts[i].iy = (int)kpts[i].y;
    }
    // 从拼接后图像中反向映射，得到待拼接图像对应的像素值，完成拼接
    // 这样做可以确保拼接图像不出现空洞
    cimg_forXYC(stitched, x, y, c) {
        CImg<float> xy1(1, 3, 1, 1, 1);
        xy1(0) = x;
        xy1(1) = y - yOffset;
        CImg<float> target = homographyForward * xy1;
        target = homographyBackward * xy1;
        target /= target(2);
        float target_x = target(0);
        float target_y = target(1);
        if (target_x >= 0 && target_x < src.width() && target_y >= 0 && target_y < src.height()) {
            stitched(x, y, c) = src((int)target_x, (int)target_y, c);
        }
    }
    stitched.display();
    // 获取拼接后图像的特征点和特征点描述子
    Feature f(stitched);
    f.setDescriptors(descriptors);
    f.setKeypoints(kpts);
    return f;
}
```

由于代码较多，部分细节没有在此列出。具体实现详见源代码。

五. 测试过程及结果

测试主函数：

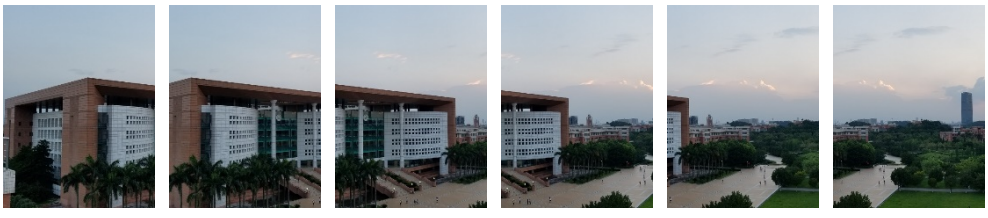
读取图像并提取特征点（暂时没有实现自动读取路径下的全部图片）：

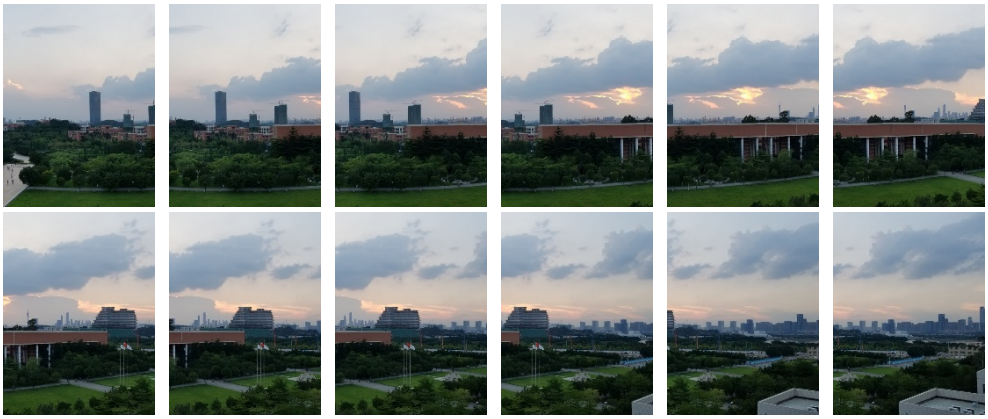
```
// 读取图像
vector<Feature> src_imgs(18);
src_imgs[0] = Feature(CImg<unsigned char>("test/1.jpg"));
src_imgs[1] = Feature(CImg<unsigned char>("test/2.jpg"));
src_imgs[2] = Feature(CImg<unsigned char>("test/3.jpg"));
src_imgs[3] = Feature(CImg<unsigned char>("test/4.jpg"));
src_imgs[4] = Feature(CImg<unsigned char>("test/5.jpg"));
src_imgs[5] = Feature(CImg<unsigned char>("test/6.jpg"));
src_imgs[6] = Feature(CImg<unsigned char>("test/7.jpg"));
src_imgs[7] = Feature(CImg<unsigned char>("test/8.jpg"));
src_imgs[8] = Feature(CImg<unsigned char>("test/9.jpg"));
src_imgs[9] = Feature(CImg<unsigned char>("test/10.jpg"));
src_imgs[10] = Feature(CImg<unsigned char>("test/11.jpg"));
src_imgs[11] = Feature(CImg<unsigned char>("test/12.jpg"));
src_imgs[12] = Feature(CImg<unsigned char>("test/13.jpg"));
src_imgs[13] = Feature(CImg<unsigned char>("test/14.jpg"));
src_imgs[14] = Feature(CImg<unsigned char>("test/15.jpg"));
src_imgs[15] = Feature(CImg<unsigned char>("test/16.jpg"));
src_imgs[16] = Feature(CImg<unsigned char>("test/17.jpg"));
src_imgs[17] = Feature(CImg<unsigned char>("test/18.jpg"));

// 提取特征点
for (int i = 0; i < src_imgs.size(); i++) {
    src_imgs[i].extractFeatures();
}
Feature stitched = src_imgs[0];
```

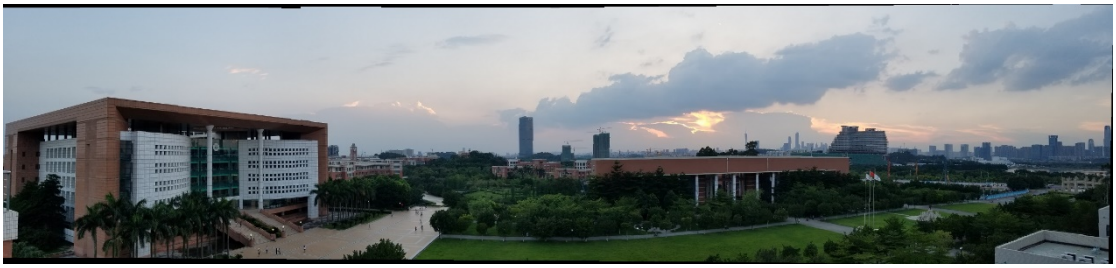
调用函数进行图像拼接：

```
// 逐张图像拼接
for (int i = 0; i < src_imgs.size() - 1; i++) {
    Match matchForward(src_imgs[i + 1].getImage(), src_imgs[i + 1].getKeypoints(), src_imgs[i + 1].getDescriptors(),
        stitched.getImage(), stitched.getKeypoints(), stitched.getDescriptors());
    Match matchBackward(stitched.getImage(), stitched.getKeypoints(), stitched.getDescriptors(),
        src_imgs[i + 1].getImage(), src_imgs[i + 1].getKeypoints(), src_imgs[i + 1].getDescriptors());
    // 特征点匹配
    matchForward.findMatchPoints();
    matchBackward.findMatchPoints();
    // 进行RANSAC过程，得到前向和后向映射的变换矩阵
    CImg<float> homographyForward = matchForward.RANSAC();
    CImg<float> homographyBackward = matchBackward.RANSAC();

    Stitch stitch(stitched.getImage(), src_imgs[i + 1].getImage(), homographyForward, homographyBackward,
        src_imgs[i + 1].getKeypoints(), src_imgs[i + 1].getDescriptors());
    // 释放前一个拼接图像的特征点描述子占用的资源
    vector<float*> descriptors = stitched.getDescriptors();
    for (int i = 0; i < descriptors.size(); i++) {
        delete[] descriptors[i];
        descriptors[i] = NULL;
    }
    // 图像拼接
    stitched = stitch.stitch();
}
// 输出结果
stitched.getImage().save("result.bmp");
return 0;
```

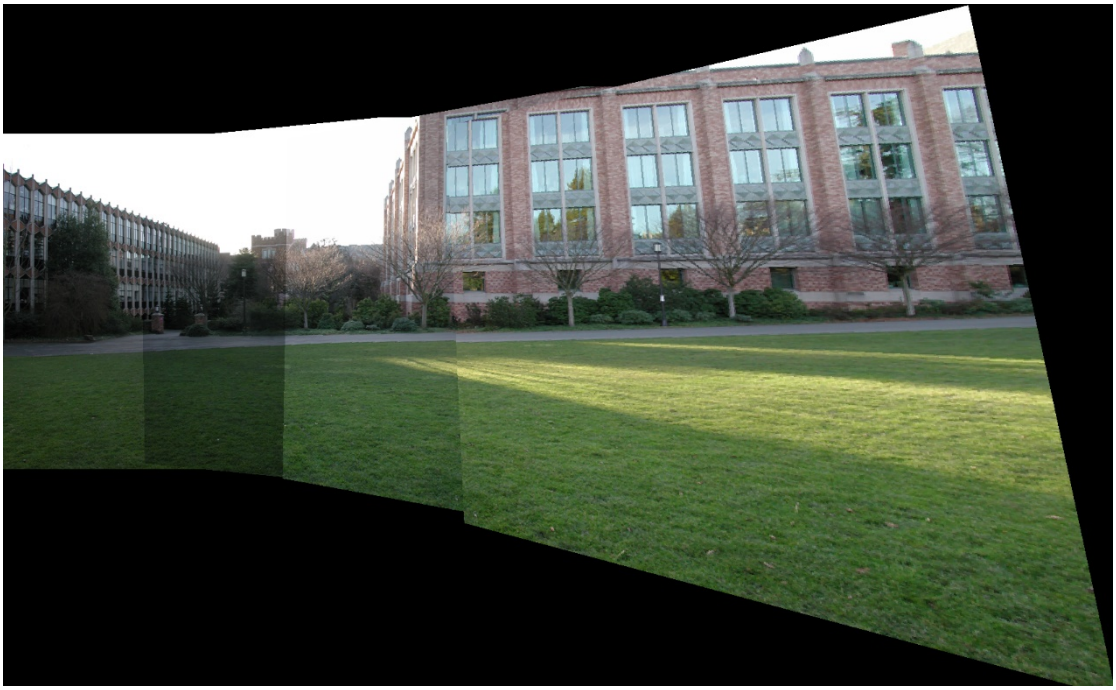
测试结果1：







测试结果2：



测试结果3：

六. 结果分析

在上面的实验结果中，结果1的图片是经过预处理的（从全景照片中截出），而结果2、3的图片是手持相机拍出来的，没有经过任何预处理。在这三个结果中，图像特征点都有很好的匹配，拼接的边缘基本没有错位的现象，说明RANSAC拟合出来的变换矩阵比较正确。

可以看出，经过预处理的照片，拼接起来会在一个平面上，没有出现透视现象。而直接环绕拍摄的照片，拼接起来会有透视现象。原因是这次实验没有进行图像的Bundle Adjustment操作。由于计算特征点匹配矩阵时，采用了具有8个自由度的单应矩阵，因此拼接时，为了最大程度匹配特征点，图像会出现透视变换。例如下图老师课件中的例子。这个是需要改进的地方之一。不过，透视的拼接会让人感觉更加真实，能够应用在虚拟现实中。

未进行Bundle Adjustment

进行Bundle Adjustment后

另外，由于难度过大，我仍未实现图像拼接的泊松融合，导致结果2中的拼接边缘会出现亮度的突变。这也是需要改进的地方。