Lab-06 Report

st122246

July 22, 2022

1 OpenSFM

Improvement

- Q: Do you think the automatically extraced distortion parameters are working well?
  Ans:Automatically extracted distortion parameters are not working well. Radial distortion can still be seen in undistorted images.

- Q: Does esitmated camera parameters seem reasonable?
  Ans: Estimated camera parameters seems not reasonable.

- Q: Would giving your own parameters be better?
  Ans: Giving own parameters improve the result.

Thought experiments

- How could you write a program to automatically extract keyframes using optical flow method?

  The program would do the following: - Optical flow method calculated of each frame and selected frame with extreme optical difference as key frame and others as candidate keyframes - By calculating the mutual information of key frame set, the minimum mutual information entropy is taken as threshold - Frame larger than threshold are put into key frame set

- What would you require of optical flow between keyframe i and keyframe i+1?

  Mutual information of keyframe i and keyframe i+1 is needed.

Figure 1: Berlin Dataset

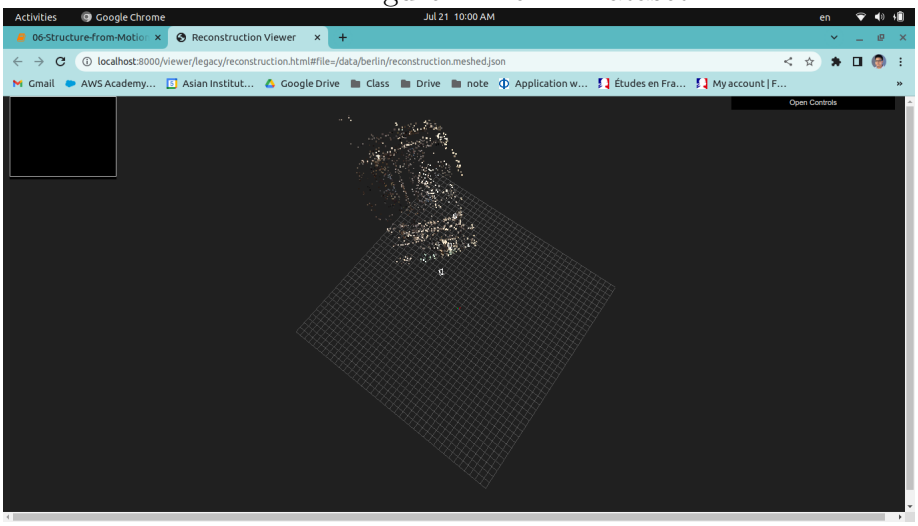


Figure 2: Lab06 frames Dataset

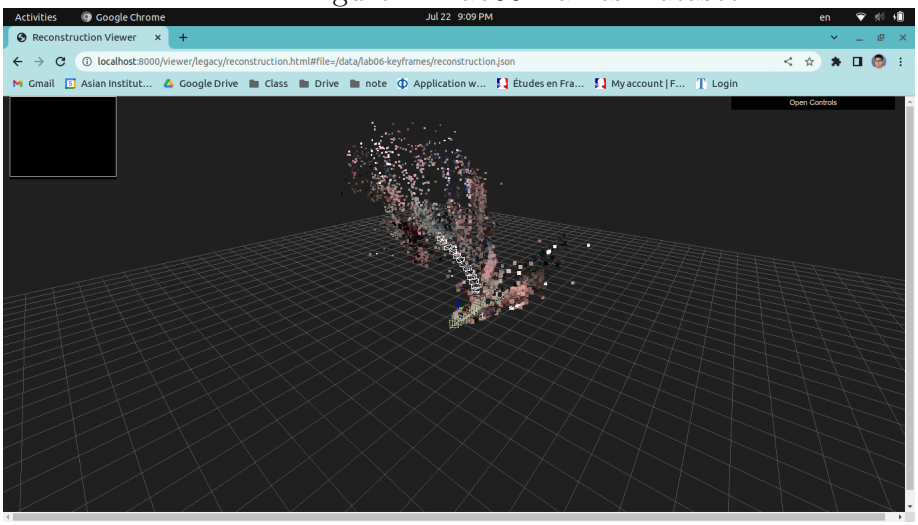


Figure 3: Reconstruction of Lab06 frames Dataset

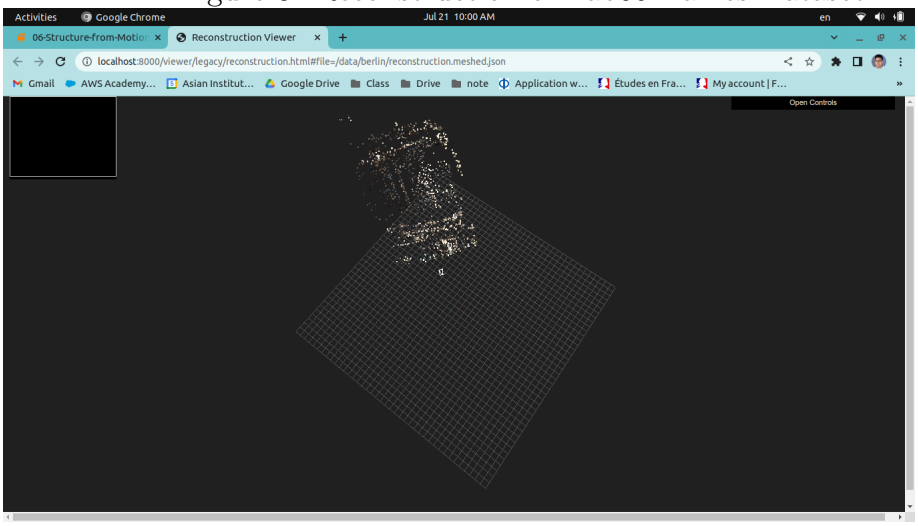


Figure 4: Vizualization of SfM in OpenCV

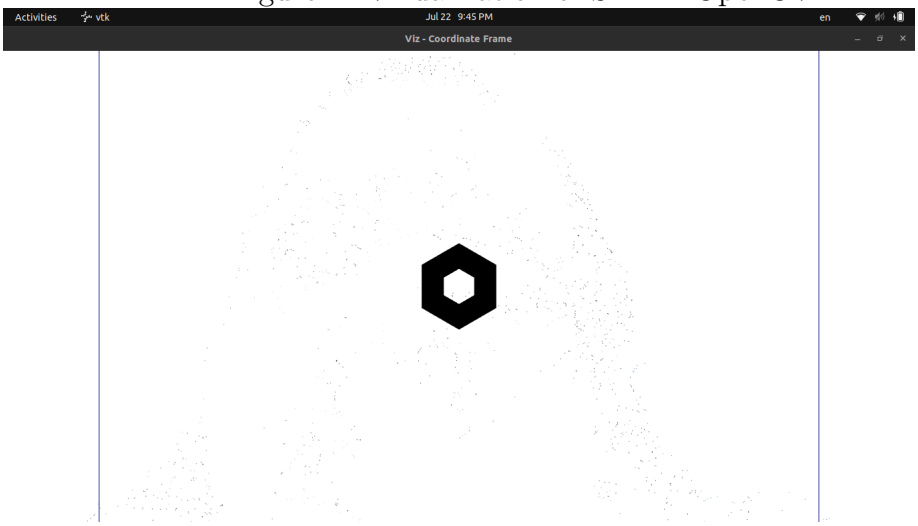
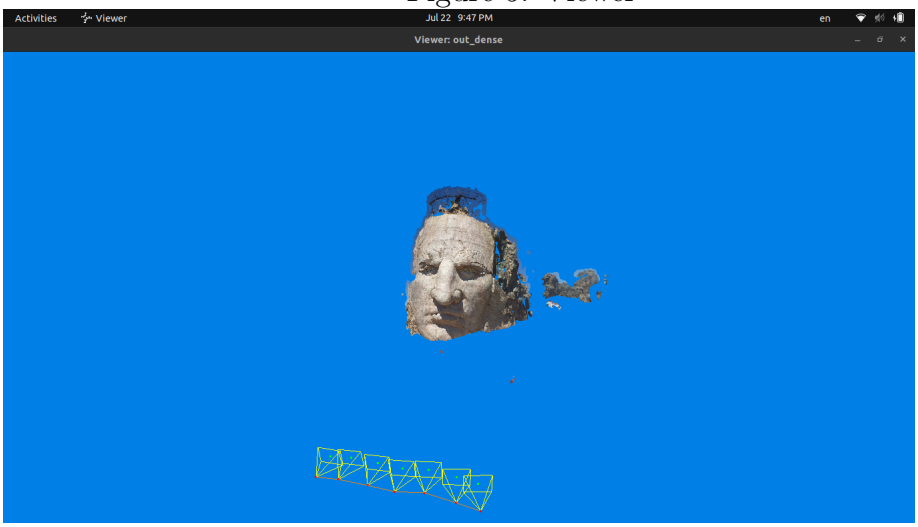


Figure 5: Viewer



2 Impelementing SfM in OpenCV

```
1 // main.cpp
2 #define CERES_FOUND true
3 #define _USE_OPENCV true
4 #define OPENCV_TRAITS_ENABLE_DEPRECATED
5
6 #include <iostream>
7 #include <algorithm>
8 #include <string>
9 #include <numeric>
10
11 #include <opencv2/opencv.hpp>
12 #include <opencv2/sfm.hpp>
13 #include <opencv2/viz.hpp>
14 #include <opencv2/calib3d.hpp>
15 #include <opencv2/core.hpp>
16 #include <opencv2/core/utils/logger.hpp>
17 #include <opencv2/core/utils/filesystem.hpp>
18 #include <opencv2/xfeatures2d.hpp>
19
20 #include <boost/filesystem.hpp>
21 #include <boost/graph/graph_traits.hpp>
22 #include <boost/graph/adjacency_list.hpp>
```

```
23 #include <boost/graph/connected_components.hpp>
24 #include <boost/graph/graphviz.hpp>
25
26 #include <OpenMVS/MVS/Interface.h>
27
28 using namespace cv;
29 using namespace std;
30 namespace fs = boost::filesystem;
31
32
33 class StructureFromMotion {
34
35 public:
36     StructureFromMotion(const string& dir,
37                         const float matchSurvivalRate =
     0.5f,
38                         const bool viz = false,
39                         const string mvs = "",
40                         const string cloud = "",
41                         const bool saveDebug = false)
42             : PAIR_MATCH_SURVIVAL_RATE(
    matchSurvivalRate)
43             , visualize(viz)
44             , saveMVS(mvs)
45             , saveCloud(cloud)
46             , saveDebugVisualizations(saveDebug)
47
48     {
49         findImagesInDiretcory(dir);
50     }
51
52     void runSfM()
53     {
54         extractFeatures();
55         matchFeatures();
56         buildTracks();
57         reconstructFromTracks();
58         if (visualize) {
59             visualize3D();
60         }
61         if (saveMVS != "") {
```

```
62              saveToMVSFile();
63          }
64          if (saveCloud != "") {
65 //             CV_LOG_INFO(TAG, "Save point cloud to: "
    + saveCloud);
66              viz::writeCloud(saveCloud, pointCloud,
    pointCloudColor);
67          }
68      }
69
70 private:
71      void findImagesInDiretcory(const string& dir)
72      {
73 //         CV_LOG_INFO(TAG, "Finding images in " + dir);
74
75          utils::fs::glob(dir, "*.jpg", imagesFilenames);
76          utils::fs::glob(dir, "*.JPG", imagesFilenames);
77          utils::fs::glob(dir, "*.png", imagesFilenames);
78          utils::fs::glob(dir, "*.PNG", imagesFilenames);
79
80          std::sort(imagesFilenames.begin(),
    imagesFilenames.end());
81
82 //         CV_LOG_INFO(TAG, "Found " + std::to_string(
    imagesFilenames.size()) + " images");
83
84 //         CV_LOG_INFO(TAG, "Reading images...");
85          for (const auto& i : imagesFilenames) {
86 //             CV_LOG_INFO(TAG, i);
87              images[i] = imread(i);
88              imageIDs[i] = images.size() - 1;
89          }
90      }
91
92      void extractFeatures()
93      {
94 //         CV_LOG_INFO(TAG, "Extract Features");
95
96          auto detector = AKAZE::create();
97          auto extractor = AKAZE::create();
98
```

```
 99          for (const auto& i : imagesFilenames) {
100              Mat grayscale;
101              cvtColor(images[i], grayscale,
     COLOR_BGR2GRAY);
102              detector->detect(grayscale, keypoints[i]);
103              extractor->compute(grayscale, keypoints[i],
      descriptors[i]);
104
105 //               CV_LOG_INFO(TAG, "Found " + to_string(
     keypoints[i].size()) + " keypoints in " + i);
106
107              if (saveDebugVisualizations) {
108                  Mat out;
109                  drawKeypoints(images[i], keypoints[i],
     out, Scalar(0, 0, 255));
110                  imwrite(fs::basename(fs::path(i)) + "
     _features.jpg", out);
111              }
112          }
113      }
114
115      vector<DMatch> matchWithRatioTest(
116              const DescriptorMatcher& matcher, const Mat
     & desc1, const Mat& desc2)
117      {
118          // Raw match
119          vector<vector<DMatch>> nnMatch;
120          matcher.knnMatch(desc1, desc2, nnMatch, 2);
121
122          // Ratio test filter
123          vector<DMatch> ratioMatched;
124          for (size_t i = 0; i < nnMatch.size(); i++) {
125              DMatch first = nnMatch[i][0];
126              float dist1 = nnMatch[i][0].distance;
127              float dist2 = nnMatch[i][1].distance;
128
129              if (dist1 < MATCH_RATIO_THRESHOLD * dist2)
     {
130                  ratioMatched.push_back(first);
131              }
132          }
```

```
133
134         return ratioMatched;
135     }
136
137     void matchFeatures()
138     {
139 //         CV_LOG_INFO(TAG, "Match Features");
140
141         BFMatcher matcher(NORM_HAMMING);
142
143         for (size_t i = 0; i < imagesFilenames.size() -
      1; ++i) {
144             for (size_t j = i + 1; j < imagesFilenames.
    size(); ++j) {
145                 const string imgi = imagesFilenames[i];
146                 const string imgj = imagesFilenames[j];
147
148                 // Match with ratio test filter
149                 vector<DMatch> match
150                         = matchWithRatioTest(matcher,
    descriptors[imgi], descriptors[imgj]);
151
152                 // Reciprocity test filter
153                 vector<DMatch> matchRcp
154                         = matchWithRatioTest(matcher,
    descriptors[imgj], descriptors[imgi]);
155                 vector<DMatch> merged;
156                 for (const DMatch& dmr : matchRcp) {
157                     bool found = false;
158                     for (const DMatch& dm : match) {
159                         // Only accept match if 1
    matches 2 AND 2 matches 1.
160                         if (dmr.queryIdx == dm.trainIdx
     and dmr.trainIdx == dm.queryIdx) {
161                             merged.push_back(dm);
162                             found = true;
163                             break;
164                         }
165                     }
166                     if (found) {
167                         continue;
```

```
168                     }
169                 }
170
171                 // Fundamental matrix filter
172                 vector<uint8_t> inliersMask(merged.size
    ());
173                 vector<Point2f> imgiPoints, imgjPoints;
174                 for (const auto& m : merged) {
175                     imgiPoints.push_back(keypoints[imgi
    ][m.queryIdx].pt);
176                     imgjPoints.push_back(keypoints[imgj
    ][m.trainIdx].pt);
177                 }
178                 findFundamentalMat(imgiPoints,
    imgjPoints, inliersMask);
179
180                 vector<DMatch> final;
181                 for (size_t m = 0; m < merged.size(); m
    ++) {
182                     if (inliersMask[m]) {
183                         final.push_back(merged[m]);
184                     }
185                 }
186
187                 if ((float) final.size() / (float)match
    .size() < PAIR_MATCH_SURVIVAL_RATE) {
188 //                      CV_LOG_INFO(TAG,
189 //                                  "Final match '" +
    imgi + "'->" + imgj + "' has less than "
190 //                                  + to_string(
    PAIR_MATCH_SURVIVAL_RATE) + " inliers from orignal.
    Skip");
191                     continue;
192                 }
193
194                 matches[make_pair(imgi, imgj)] = final;
195
196 //                  CV_LOG_INFO(TAG,
197 //                              "Matching " + imgi + "
    and " + imgj + ": " + to_string(final.size()) + " /
    "
```

```
198 //                                  + to_string(match.size())
    );
199
200                 if (saveDebugVisualizations) {
201                     Mat out;
202                     vector<DMatch> rawmatch;
203                     matcher.match(descriptors[imgi],
    descriptors[imgj], rawmatch);
204                     vector<pair<string, vector<DMatch
    >&>> showList { { "Raw Match", rawmatch },
205
                    { "Ratio Test Filter", match }, { "
    Reciprocal Filter", merged },
206
                    { "Epipolar Filter", final } };
207                     for (size_t i = 0; i < showList.
    size(); i++) {
208                         drawMatches(images[imgi],
    keypoints[imgi], images[imgj], keypoints[imgj],
209                                     showList[i].second,
     out, CV_RGB(255, 0, 0));
210                         cv::putText(out, showList[i].
    first, Point(10, 50), FONT_HERSHEY_COMPLEX, 2.0,
211                                     CV_RGB(255, 255,
    255), 2);
212                         cv::putText(out, "# Matches: "
    + to_string(showList[i].second.size()),
213                                     Point(10, 100),
    FONT_HERSHEY_COMPLEX, 1.0, CV_RGB(255, 255, 255));
214                         cv::imwrite(fs::basename(fs::
    path(imgi)) + "_" + fs::basename(fs::path(imgj))
215                                     + "_" + to_string(i
    ) + ".jpg",
216                                     out);
217                     }
218                 }
219             }
220         }
221     }
222
223     void buildTracks()
```

```
224     {
225 //         CV_LOG_INFO(TAG, "Build tracks");
226 
227         using namespace boost;
228 
229         struct ImageFeature {
230             string image;
231             size_t featureID;
232         };
233         typedef adjacency_list<listS, vecS, undirectedS
    , ImageFeature> Graph;
234         typedef graph_traits<Graph>::vertex_descriptor
    Vertex;
235 
236         map<pair<string, int>, Vertex>
    vertexByImageFeature;
237 
238         Graph g;
239 
240         // Add vertices - image features
241         for (const auto& imgi : keypoints) {
242             for (size_t i = 0; i < imgi.second.size();
    i++) {
243                 Vertex v = add_vertex(g);
244                 g[v].image = imgi.first;
245                 g[v].featureID = i;
246                 vertexByImageFeature[make_pair(imgi.
    first, i)] = v;
247             }
248         }
249 
250         // Add edges - feature matches
251         for (const auto& match : matches) {
252             for (const DMatch& dm : match.second) {
253                 Vertex& vI = vertexByImageFeature[
    make_pair(match.first.first, dm.queryIdx)];
254                 Vertex& vJ = vertexByImageFeature[
    make_pair(match.first.second, dm.trainIdx)];
255                 add_edge(vI, vJ, g);
256             }
257         }
```

```
258
259         using Filtered = filtered_graph<Graph, keep_all
        , std::function<bool(Vertex)>>;
260         Filtered gFiltered(g, keep_all {}, [&g](Vertex
        vd) { return degree(vd, g) > 0; });
261
262         // Get connected components
263         std::vector<int> component(num_vertices(
        gFiltered), -1);
264         int num = connected_components(gFiltered, &
        component[0]);
265         map<int, vector<Vertex>> components;
266         for (size_t i = 0; i != component.size(); ++i)
        {
267             if (component[i] >= 0) {
268                 components[component[i]].push_back(i);
269             }
270         }
271         // Filter bad components (with more than 1
        feature from a single image)
272         std::vector<int> vertexInGoodComponent(
        num_vertices(gFiltered), -1);
273         map<int, vector<Vertex>> goodComponents;
274         for (const auto& c : components) {
275             set<string> imagesInComponent;
276             bool isComponentGood = true;
277             for (int j = 0; j < c.second.size(); ++j) {
278                 const string imgId = g[c.second[j]].
        image;
279                 if (imagesInComponent.count(imgId) > 0)
         {
280                     // Image already represented in
        this component
281                     isComponentGood = false;
282                     break;
283                 } else {
284                     imagesInComponent.insert(imgId);
285                 }
286             }
287             if (isComponentGood) {
288                 for (int j = 0; j < c.second.size(); ++
```

```
    j) {
289                     vertexInGoodComponent[c.second[j]]
    = 1;
290                 }
291                 goodComponents[c.first] = c.second;
292             }
293         }
294
295         Filtered gGoodComponents(g, keep_all {},
296                                  [&
    vertexInGoodComponent](Vertex vd) { return
    vertexInGoodComponent[vd] > 0; });
297
298 //          CV_LOG_INFO(TAG, "Total number of components
    found: " + to_string(components.size()));
299 //          CV_LOG_INFO(TAG, "Number of good components:
    " + to_string(goodComponents.size()));
300         const int accum = std::accumulate(
    goodComponents.begin(), goodComponents.end(), 0,
301                                           [](int a,
    pair<const int, vector<Vertex>>& v) { return a + v.
    second.size(); });
302 //          CV_LOG_INFO(TAG,
303 //                      "Average component size: " +
    to_string((float)accum / (float)(goodComponents.size
    ())));
304
305         if (saveDebugVisualizations) {
306             struct my_node_writer {
307                 my_node_writer(Graph& g_, const map<
    string, int>& iid_)
308                         : g(g_)
309                         , iid(iid_) {};
310                 void operator()(std::ostream& out,
    Vertex v)
311                 {
312                     const int imgId = iid[g[v].image];
313                     out << " [label=\"" << imgId
314                         << "\" colorscheme=\"accent8\"
    fillcolor=" << (imgId + 1)
315                         << " style=filled]";
```

```
316                 };
317                 Graph g;
318                 map<string, int> iid;
319             };
320             std::ofstream ofs("
    match_graph_good_components.dot");
321             write_graphviz(ofs, gGoodComponents,
    my_node_writer(g, imageIDs));
322             std::ofstream ofsf("match_graph_filtered.
    dot");
323             write_graphviz(ofsf, gFiltered,
    my_node_writer(g, imageIDs));
324         }
325
326         // Each component is a track
327         const size_t nViews = imagesFilenames.size();
328         tracks.resize(nViews);
329         for (int i = 0; i < nViews; i++) {
330             tracks[i].create(2, goodComponents.size(),
    CV_64FC1);
331             tracks[i].setTo(-1.0);
332         }
333         int i = 0;
334         for (auto c = goodComponents.begin(); c !=
    goodComponents.end(); ++c, ++i) {
335             for (const int v : c->second) {
336                 const int imageID = imageIDs[g[v].image
    ];
337                 const size_t featureID = g[v].featureID
    ;
338                 const Point2f p = keypoints[g[v].image
    ][featureID].pt;
339                 tracks[imageID].at<double>(0, i) = p.x;
340                 tracks[imageID].at<double>(1, i) = p.y;
341             }
342         }
343
344         if (saveDebugVisualizations) {
345             vector<Scalar> colors
346                     = { CV_RGB(240, 248, 255), CV_RGB
    (250, 235, 215), CV_RGB(0, 255, 255),
```

```
347                         CV_RGB(127, 255, 212), CV_RGB
    (240, 255, 255), CV_RGB(245, 245, 220),
348                         CV_RGB(255, 228, 196), CV_RGB
    (255, 235, 205), CV_RGB(0, 0, 255),
349                         CV_RGB(138, 43, 226), CV_RGB
    (165, 42, 42), CV_RGB(222, 184, 135) };
350
351             vector<Mat> imagesM;
352             for (const auto m : images)
353                 imagesM.push_back(m.second);
354             Mat out;
355             hconcat(vector<Mat>(imagesM.begin(),
    imagesM.begin() + 4), out);
356             RNG& rng = cv::theRNG();
357             const Size imgS = imagesM[0].size();
358             for (int tId = 0; tId < 20; tId++) {
359                 const int trackId = rng(tracks[0].cols)
    ; // Randomize a track ID
360
361                 // Show track over images
362                 for (int i = 0; i < 3; i++) {
363                     Point2f a = Point2f(tracks[i].col(
    trackId));
364                     Point2f b = Point2f(tracks[i + 1].
    col(trackId));
365
366                     if (a.x < 0 or a.y < 0 or b.x < 0
    or b.y < 0) {
367                         continue;
368                     }
369
370                     const Scalar c = colors[tId %
    colors.size()];
371                     a.x += imgS.width * i;
372                     b.x += imgS.width * (i + 1);
373                     circle(out, a, 7, c, FILLED);
374                     circle(out, b, 7, c, FILLED);
375                     line(out, a, b, c, 3);
376                 }
377                 cv::imwrite("tracks.jpg", out);
378
```

```
379                 // Show track patches
380                 const int patchSize = 20;
381                 const Point2f patch(patchSize,
    patchSize);
382                 for (int i = 0; i < tracks.size(); i++)
     {
383                     Point2f a = Point2f(tracks[i].col(
    trackId));
384                     if (a.x < patchSize or a.y <
    patchSize or a.x > imgS.width - patchSize
385                         or a.y > imgS.height -
    patchSize) {
386                         continue;
387                     }
388
389                     cv::imwrite("track_" + to_string(
    trackId) + "_" + to_string(i) + ".png",
390                                 imagesM[i](Rect(a -
    patch, a + patch)));
391                 }
392             }
393         }
394     }
395
396     bool reconstructFromTracks()
397     {
398 //         CV_LOG_INFO(TAG, "Reconstruct from " +
    to_string(tracks[0].cols) + " tracks");
399         const Size imgS = images.begin()->second.size()
    ;
400         const float f = std::max(imgS.width, imgS.
    height);
401         Mat K
402                 = Mat(Matx33f { f, 0.0, imgS.width /
    2.0f, 0.0, f, imgS.height / 2.0f, 0.0, 0.0, 1.0 });
403         cv::sfm::reconstruct(tracks, Rs, Ts, K,
    points3d, true);
404
405         K.copyTo(K_);
406
407 //         CV_LOG_INFO(TAG, "Reconstruction: ");
```

```
408 //         CV_LOG_INFO(TAG, "Estimated 3D points: " +
    to_string(points3d.size()));
409 //         CV_LOG_INFO(TAG, "Estimated cameras: " +
    to_string(Rs.size()));
410 //         CV_LOG_INFO(TAG, "Refined intrinsics: ");
411 //         CV_LOG_INFO(TAG, K_);
412
413         if (Rs.size() != imagesFilenames.size()) {
414 //             CV_LOG_ERROR(TAG,
415 //                          "Unable to reconstruct all
    camera views (" + to_string(imagesFilenames.size())
416 //                          + ")");
417             return false;
418         }
419
420         if (tracks[0].cols != points3d.size()) {
421 //             CV_LOG_WARNING(
422 //                     TAG, "Unable to reconstruct all
    tracks (" + to_string(tracks[0].cols) + ")");
423         }
424
425         // Create the point cloud
426         pointCloud.clear();
427         for (const auto& p : points3d)
428             pointCloud.emplace_back(Vec3f(p));
429
430         // Get the point colors
431         pointCloudColor.resize(pointCloud.size(), Vec3b
    (0, 255, 0));
432         vector<Point2f> point2d(1);
433         for (int i = 0; i < (int)pointCloud.size(); i
    ++) {
434             for (int j = 0; j < imagesFilenames.size();
     ++j) {
435                 Mat point3d = Mat(pointCloud[i]).
    reshape(1, 1);
436                 cv::projectPoints(point3d, Rs[j], Ts[j
    ], K_, Mat(), point2d);
437                 if (point2d[0].x < 0 or point2d[0].x >=
     imgS.width or point2d[0].y < 0
438                     or point2d[0].y >= imgS.height) {
```

```
439                     continue;
440                 }
441                 pointCloudColor[i] = images[
    imagesFilenames[j]].at<Vec3b>(point2d[0]);
442                 break;
443             }
444         }
445
446         return true;
447     }
448
449     void visualize3D()
450     {
451 //        CV_LOG_INFO(TAG, "Visualize reconstruction");
452
453         if (saveDebugVisualizations) {
454             // 3d point reprojections
455             Mat points2d;
456             Mat points3dM(points3d.size(), 1, CV_32FC3)
    ;
457             for (int i = 0; i < points3d.size(); i++) {
458                 points3dM.at<Vec3f>(i) = Vec3f(points3d
    [i]);
459             }
460             for (int j = 0; j < imagesFilenames.size();
     j++) {
461                 cv::projectPoints(points3dM, Rs[j], Ts[
    j], K_, noArray(), points2d);
462
463                 Mat out;
464                 images[imagesFilenames[j]].copyTo(out);
465                 for (int i = 0; i < points2d.rows; i++)
     {
466                     circle(out, points2d.at<Point2f>(i)
    , 3, CV_RGB(255, 0, 0), FILLED);
467                 }
468                 cv::imwrite("reprojection_" + to_string
    (j) + ".jpg", out);
469             }
470         }
471
```

```
472         // Create 3D windows
473         viz::Viz3d window("Coordinate Frame");
474         window.setWindowSize(Size(500, 500));
475         window.setWindowPosition(Point(150, 150));
476         window.setBackgroundColor(viz::Color::white());
477 
478         // Recovering cameras
479         vector<Affine3d> path;
480         for (size_t i = 0; i < Rs.size(); ++i)
481             path.push_back(Affine3d(Rs[i], Ts[i]));
482 
483         // Add the pointcloud
484         viz::WCloud cloud_widget(pointCloud,
    pointCloudColor);
485         window.showWidget("point_cloud", cloud_widget);
486         // Add cameras
487         window.showWidget("cameras_frames_and_lines",
488                           viz::WTrajectory(path, viz::
    WTrajectory::BOTH, 0.1, viz::Color::black()));
489         window.showWidget(
490                 "cameras_frustums", viz::
    WTrajectoryFrustums(path, K_, 0.1, viz::Color::navy
    ()));
491         window.setViewerPose(path[0]);
492 
493         /// Wait for key 'q' to close the window
494 //        CV_LOG_INFO(TAG, "Press 'q' to close ...")
495 
496         window.spin();
497     }
498 
499     void saveToMVSFile()
500     {
501 //        CV_LOG_INFO(TAG, "Save reconstruction to MVS
    file: " + saveMVS)
502 
503         MVS::Interface interface;
504         MVS::Interface::Platform p;
505 
506         // Add camera
507         MVS::Interface::Platform::Camera c;
```

```
508         const Size imgS = images[imagesFilenames[0]].
    size();
509         c.K = Matx33d(K_);
510         c.R = Matx33d::eye();
511         c.C = Point3d(0, 0, 0);
512         c.name = "Camera1";
513         c.width = imgS.width;
514         c.height = imgS.height;
515         p.cameras.push_back(c);
516 
517         // Add views
518         p.poses.resize(Rs.size());
519         for (size_t i = 0; i < Rs.size(); ++i) {
520             Mat t = -Rs[i].t() * Ts[i];
521             p.poses[i].C.x = t.at<double>(0);
522             p.poses[i].C.y = t.at<double>(1);
523             p.poses[i].C.z = t.at<double>(2);
524             Mat r;
525             Rs[i].copyTo(r);
526             Mat(r).convertTo(p.poses[i].R, CV_64FC1);
527 
528             // Add corresponding image
529             MVS::Interface::Image image;
530             image.cameraID = 0;
531             image.poseID = i;
532             image.name = imagesFilenames[i];
533             image.platformID = 0;
534             interface.images.push_back(image);
535         }
536         p.name = "Platform1";
537         interface.platforms.push_back(p);
538 
539         // Add point cloud
540         for (size_t k = 0; k < points3d.size(); ++k) {
541             MVS::Interface::Color c;
542             MVS::Interface::Vertex v;
543             v.X = Vec3f(points3d[k]);
544 
545             // Reproject to see if in image bounds and
    get the RGB color
546             Mat point3d;
```

```
547             Mat(points3d[k].t()).convertTo(point3d,
    CV_32FC1);
548             for (uint32_t j = 0; j < tracks.size(); ++j
    ) {
549                 vector<Point2f> points2d(1);
550                 cv::projectPoints(point3d, Rs[j], Ts[j
    ], K_, Mat(), points2d);
551                 if (points2d[0].x < 0 or points2d[0].x
    > imgS.width or points2d[0].y < 0
552                     or points2d[0].y > imgS.height) {
553                     continue;
554                 } else {
555                     c.c = images[imagesFilenames[j]].at
    <Vec3b>(points2d[0]);
556                     v.views.push_back({ j, 1.0 });
557                 }
558             }
559
560             interface.verticesColor.push_back(c);
561             interface.vertices.push_back(v);
562         }
563
564         MVS::ARCHIVE::SerializeSave(interface, saveMVS)
    ;
565     }
566
567     vector<String> imagesFilenames;
568     map<string, int> imageIDs;
569     map<string, Mat> images;
570     map<string, vector<KeyPoint>> keypoints;
571     map<string, Mat> descriptors;
572     map<pair<string, string>, vector<DMatch>> matches;
573     vector<Mat> Rs, Ts;
574     vector<Mat> points3d;
575     vector<Mat> tracks;
576     vector<Vec3f> pointCloud;
577     vector<Vec3b> pointCloudColor;
578     Matx33f K_;
579
580     const float MATCH_RATIO_THRESHOLD = 0.8f; //
    Nearest neighbor matching ratio
```

```
581     const float
582             PAIR_MATCH_SURVIVAL_RATE; // Ratio of
        surviving matches for a successful stereo match
583     const bool visualize; // Show 3D visualization of
        the sprase cloud?
584     const string saveMVS; // Save the reconstruction in
         MVS format for OpenMVS?
585     const string saveCloud; // Save the reconstruction
        to a point cloud file?
586     const bool saveDebugVisualizations; // Save debug
        visualizations from the reconstruction process
587
588     const string TAG = "StructureFromMotion";
589 };
590
591 int main(int argc, char** argv)
592 {
593     utils::logging::setLogLevel(utils::logging::
        LOG_LEVEL_DEBUG);
594
595     cv::CommandLineParser parser(argc, argv,
596                                  "{help h ? |       |
        help message}"
597                                  "{@dir     | .     |
        directory with image files for reconstruction }"
598                                  "{mrate    | 0.5   |
        Survival rate of matches to consider image pair
        success }"
599                                  "{viz      | false |
        Visualize the sparse point cloud reconstruction? }"
600                                  "{debug    | false |
        Save debug visualizations to files? }"
601                                  "{mvs      |       |
        Save reconstruction to an .mvs file. Provide
        filename }"
602                                  "{cloud    |       |
        Save reconstruction to a point cloud file (PLY, XYZ
        and OBJ). Provide "
603                                  "filename}");
604
605     if (parser.has("help")) {
```

```
606             parser.printMessage();
607             return 0;
608         }
609
610         StructureFromMotion sfm(parser.get<string>("@dir"),
            parser.get<float>("mrate"),
611                                 parser.get<bool>("viz"),
        parser.get<string>("mvs"), parser.get<string>("cloud
        "),
612                                 parser.get<bool>("debug"));
613         sfm.runSfM();
614
615         return 0;
616 }
617
618
```

```
 1 // CMakeList.txt
 2 cmake_minimum_required(VERSION 3.22)
 3 project(sfm)
 4
 5 set(CMAKE_CXX_STANDARD 14)
 6 set(CMAKE_CXX_STANDARD_REQUIRED ON)
 7 set(CMAKE_BUILD_TYPE Release)
 8
 9 add_executable(sfm main.cpp)
10
11 find_package(Ceres QUIET)
12
13 set(OpenCV_DIR "" CACHE PATH "/mnt/ntfs/Data/code/CV/
       source/opencv/build")
14 find_package(OpenCV 4.5.5 REQUIRED COMPONENTS core
       calib3d features2d sfm viz)
15
16 find_package(Eigen3 REQUIRED)
17
18 set(OpenMVS_DIR "" CACHE PATH "/mnt/ntfs/Data/code/CV/
       source/openMVS/build")
19 find_package(OpenMVS REQUIRED)
20
21 find_package(Boost REQUIRED COMPONENTS filesystem graph
```

```
    )
22
23 include_directories(${EIGEN3_INCLUDE_DIR} ${
    OpenMVS_INCLUDE_DIRS} ${Boost_INCLUDE_DIR})
24
25 message(STATUS ${OpenCV_LIBRARIES} ${OpenMVS_LIBRARIES
    })
26
27 target_link_libraries(sfm
28             ${OpenCV_LIBRARIES}
29             ${Boost_LIBRARIES}
30 #             ${OpenMVS_LIBRARIES}
31             )
32
33
```