Visual Slam implementation for the C++ course

Jere Knuutinen: 994349
Juuso Korhonen: 652377
Olivia Palmu: 1031997

1. Overview

This project's topic is simultaneous localization and mapping (SLAM) by using monocular video input. This is done by simultaneously estimating locations of environment (map) points and camera poses using video frames, and visualizing the resulting map (a set of poses and map points) with a user-interface.

Our SLAM implementation follows the ORB-SLAM paper (Mur-Artal et al., 2015), and implements map initialization, tracking and new point mapping. This is the so called frontend of SLAM. We also implement bundle adjustment to optimize the estimated map points and poses, the backend part of SLAM. Current state of the project is that the system performs few iterations (set to 10) of local tracking and mapping process. To proceed we should add filtering for badly estimated poses and points as they start to cause failure in bundle adjustment.

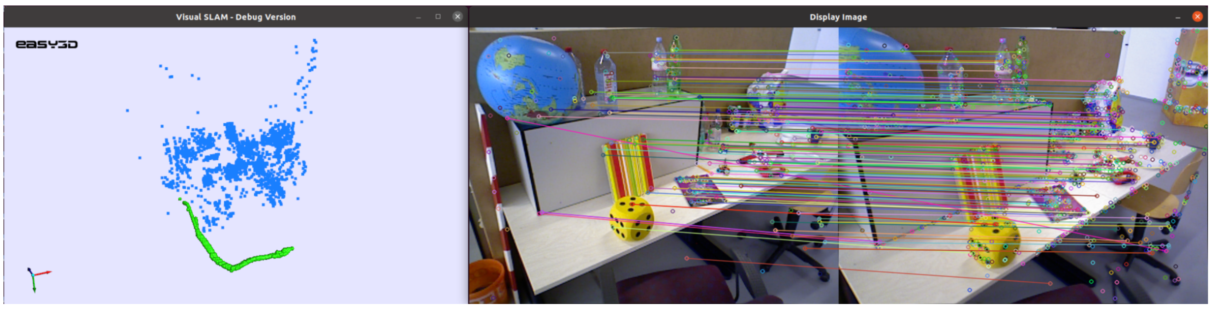


Figure 1. User-interface (on the left) visualizing the estimated camera poses (in green) and map points (in blue). We also (optionally) visualize the matching process between last keyframe (left frame) and current frame.

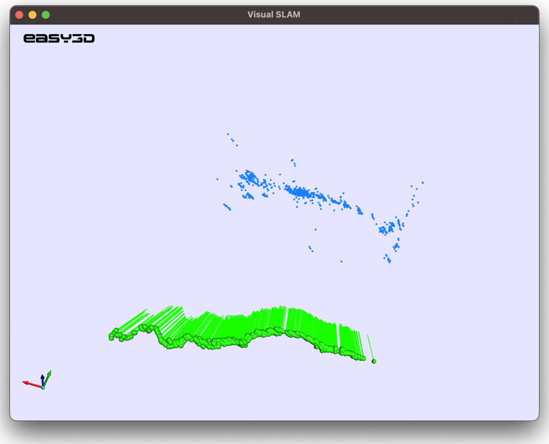


Figure 1.1. Extra interface with vectors to show camera facing directions

2. Software structure

2.1 Overall architecture:

The overall architecture consists out of Map, Point3D, Frame, and PointClouds (+ Screen) classes.

Map:

Map object holds maps (std::map), with pointers (std::shared_ptr) to Frame and Point3D objects as values and corresponding frame ids and point ids as keys. Map also has member functions InitializeMap(), LocalTracking(), LocalMapping(), BundleAdjustement(), MotionOnlyBundleAdjustement(). They implement the main algorithmic workload of this SLAM implementation. In SLAM terminology InitializeMap(), LocalTracking(), and LocalMapping() implement the frontend and BundleAdjustement() implements the backend. This is not to be confused with the user-interface frontend which is implemented with the Screen class.

Frontend:

InitializeMap():

InitializeMap starts reading video frames (iterating over png files) to establish first two keyframes (Frame objects) and a set of map points (Point3D objects). First keyframe is set as the first video frame with identity pose and added to map. To search for the second keyframe, it then starts estimating essential transformation between the first frame and current frame. When estimation succeeds, i.e. over 90% of the matched keypoints fit the transformation, a new keyframe is found. From the estimated transformation a pose is recovered and for the points fitting the transformation an estimation of their 3D location is provided with triangulation. Current Frame object with the restored pose, and Point3D objects created with the estimated 3D locations and correspondences to imagepoints and features in the current frame and first frame, are then added to the map and the function breaks.

LocalTracking():
The purpose of LocalTracking method is to perform tracking process for every frame. In addition to that it decides whether newkey frame should be inserted. First new instance of frame object is created with unique frame_id, after that feature matching between this new frame points and lastkeyframe points that have known 3d point correspondence is performed. After that pose estimation is performed using Perspective-n-Point algorithm. When new pose estimate is known the next task is to perform motion only bundle adjustment where only camera poses are optimization and point locations are held fixed. Program exists from this function after trackframecount is larger than 19 and number of inliers has decreased below 80.

LocalMapping():
The purpose of LocalMapping() method is to perform mapping process for every new key frame. In the method, new matches between two keyframes are searched and new 3d points are created using linear triangulation method. After that poses and points are again optimized using bundle adjustment.

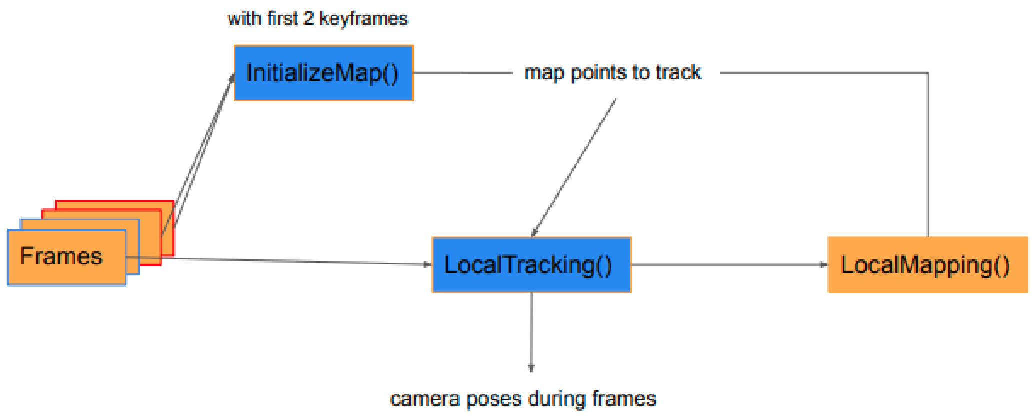


Figure 2. Algorithmic workflow of the frontend of the SLAM.

Backend:

BundleAdjustement():

BundleAdjustement() method builds a graph consisting out of frame poses and points as nodes and edges describing how the points are projected to the frames. The optimization objective is to tune the poses and point locations so that they minimize the reprojection error.

The optimization is done using Levenberg–Marquardt optimization, which is a non-linear least-squares method.

Point3D:

Point object stores estimated 3D location and map (std::map) with correspondences to imagepoints and features in Frame objects that see this map point.

Frame:

Frame object stores information processed from individual video frames:

- ID when inserted to the Map
- the actual RGB image
- keypoints and features extracted with helper class FeatureExtractor
- estimated pose of the camera during frame
- parents of the frame (also Frame objects)
- and boolean flag set to true if it is to be considered as a keyframe

Frame-class also implements static function Match2Frames(), that uses helper class FeatureMatcher to match features of two Frame objects.

Visualization:

Visualization is implemented through PointCloud classes and some freestanding functions in Screen.hpp. The workflow is to initialize a easy3d::Viewer object, register point clouds to be rendered using the viewer, configuring the point clouds and then running the visualization. Any updates sent to the point clouds after will be shown on screen. This file used to hold a Screen class, but it was deprecated due to a bug causing it to be unusable for real-time data updates.

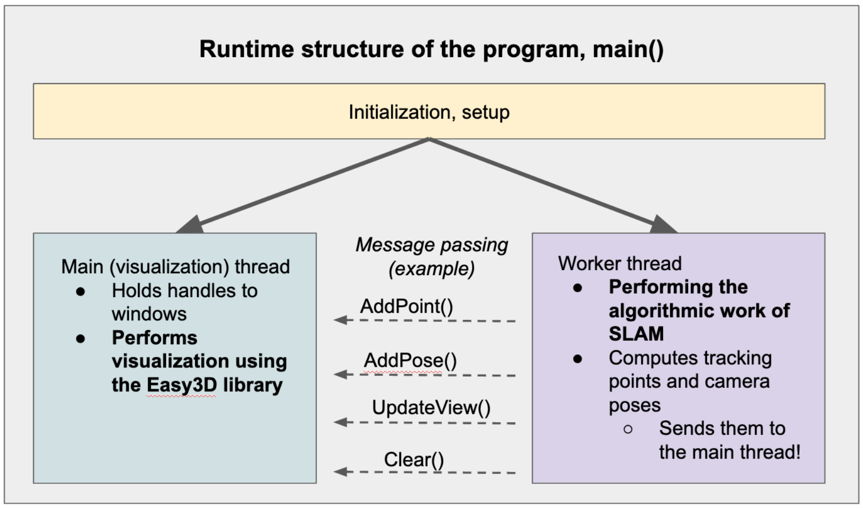

Runtime structure of the program, main()
Initialization, setup
Main (visualization) thread
Holds handles to windows
Performs visualization using the Easy3D library
Message passing (example)
AddPoint()
AddPose()
UpdateView()
Clear()
Worker thread
Performing the algorithmic work of SLAM
Computes tracking points and camera poses
Sends them to the main thread!

Figure 3: The overarching structure of the main() function, what responsibility does the visualization have in it

ConfigureModel()

This function configures the settings of a PointCloud or Graph object (which both derive from easy3d::Model). This is typically done at the start of the problem, before any points are added, but can be done during the middle of execution as well, although this is uncommon. The function has many parameters for customization:

- size: The radius of the points or the width of the line being rendered.
- plain_style: If true, this model will be rendered using untextured pixels. If set to false instead (the default), points will be rendered as spheres and cylinders.
- color: The color this is drawn with. Every point will have the same color; you cannot customize the color of individual points. This is a vec4 in normalized RGBA format, meaning each of its elements are in the range 0.0 to 1.0.

PointClouds

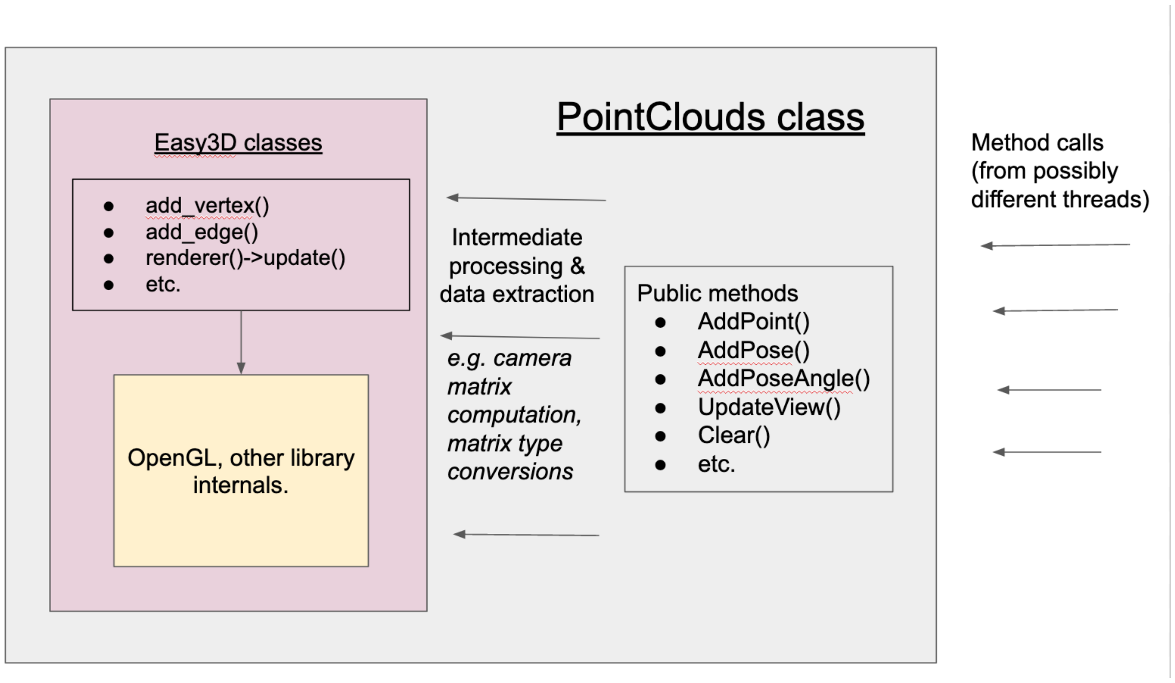

PointClouds class
Easy3D classes
add_vertex()
add_edge()
renderer()->update()
etc.
OpenGL, other library internals.
Intermediate processing & data extraction
e.g. camera matrix computation, matrix type conversions
Public methods
AddPoint()
AddPose()
AddPoseAngle()
UpdateView()
Clear()
etc.
Method calls (from possibly different threads)

Figure 4: The structure of the PointClouds class

The PointClouds class manages the contents of two PointCloud objects, used to keep track of detected points and camera poses. This class can be interfaced to add and remove points from the clouds, as well as to request a screen refresh from the Easy3D viewer.

The PointClouds() constructor takes two PointCloud pointers + a Graph pointer as arguments, for the detected points and the camera poses.

This class has many methods:

- AddPoint(x, y, z, use_poses): Adds a point to the chosen point cloud, using xyz coordinates.
- AddPointMat(mat, use_poses): Adds a point to the specified point cloud, using a cv::Mat object to store the coordinates.
- AddPose(x, y, z): A shorthand for AddPoint(x, y, z, true)
- AddPoseAngle(x0, y0, z0, x1, y1, z1): Adds a short vector on screen representing a camera facing direction.
- AddPoseAnglesMat(angles): Implements the iterated AddPoseAngle() method, taking inputs as 3x3 rotation matrices in cv::Mat form.
- UpdateView(): Requests the visualization runtime to update the points on screen. Unless this is called, the point clouds on screen will not change.
- AddPointsMatUpdate(points, use_poses): A convenience function taking a vector of cv::Mat objects that passes them all to the specified point cloud, and then calls UpdateView().
- AddPoseAnglesMatUpdate(angles): Same as AddPoseAnglesMat, but updates the view.
- Clear(use_poses): Clears the specified point cloud's contents.
- ClearAll(): Clears all point cloud contents, as well as the pose vectors.

Additional classes and functions:

helper_functions.hpp: these are helper functions for repetitive procedures like slicing of matrices and conversions to different matrix types for cv::Mat and Eigen, the former being used in the frontend and the latter being used in the backend when optimizing with g2o. Important estimation functions EstimateEssential() which estimates the essential transformation between first 2 keyframes, and triangulate(), which gives initial estimation for 3D locations of new points, are located here as well.

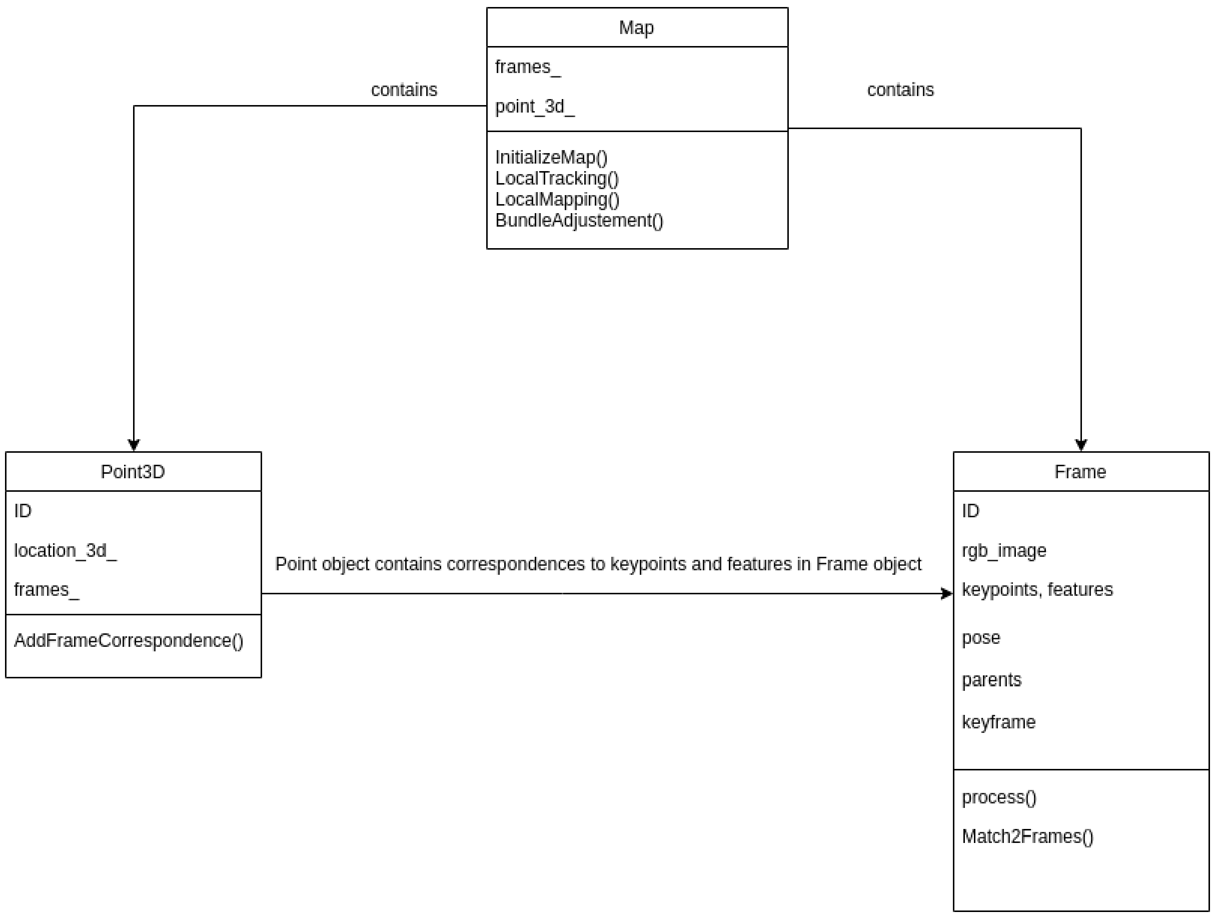
FeatureMatcher: a class to initialize and use OpenCV's implementation of bruteforce-matching (cv::BFMatcher) with K-nearest-neighbours. Provides matching between two Frame objects.

FeatureExtractor: a class to initialize and use OpenCV's implementation of ORB feature detection and description (cv::ORB). Provides keypoints and features for Frame objects.

2.1.2 Implementation details and use of (advanced) features

Map stores pointers to Frame and Point3D objects. We used std::shared_ptr for easy and efficient memory management for these. Using normal pointers would have required a careful use of destructors as not all created pointers to frames get added to map. For example in InitializeMap() we create multiple Frame pointers, but they do not necessarily get added to the map since essential transform estimation can fail. This could have easily resulted in memory leaks if the destructors would not get called for example in case of runtime-error.

2.2 Class relationships diagram:



2.3 Interfaces to external libraries:

We use the following external libraries: OpenCV 4.0, Eigen 3, g2o, Easy3D. OpenCV is used inherently in the SLAM frontend. Eigen3 and g2o are used inherently in the SLAM backend. Easy3D was used in visualization and Screen is created as interface for the use of it.

3. Instructions for building and using the software

3.1 How to compile:

Requirements:

- OpenCV https://opencv.org
- Eigen3 http://eigen.tuxfamily.org
- g2o https://github.com/RainerKuemmerle/g2o
- Easy3D https://github.com/LiangliangNan/Easy3D
- cmake (minimum required VERSION 3.11.0)
- C++14 compiler

After installing the requirements, We recommend a so-called out of source build which can be achieved by the following command sequence.

- `cd <path-to-cmakelist>`
- `mkdir build`
- `cd build`
- `cmake ../`
- `make`

This creates a run file named “run_slam”, which you can run using ./run_slam while in the build directory.

3.2 Basic user guide:

When you run the runfile (./run_slam) the program should start creating a map based on the video input. The default and tested video input is in the data/rgbd_dataset_freiburg3_long_office_household/rgb/ folder in the form of png file for each video frame. The program outputs to the terminal information of the SLAM phase it is currently in (initialization, tracking, mapping, bundleadjustement). Once the map initialization is ready, the program should open a viewer visible to the user, where the estimated poses and points are then iteratively visualized. The viewer opens upsidedown currently and should be turned using mouse. To close the program, close the viewer.

4. Testing

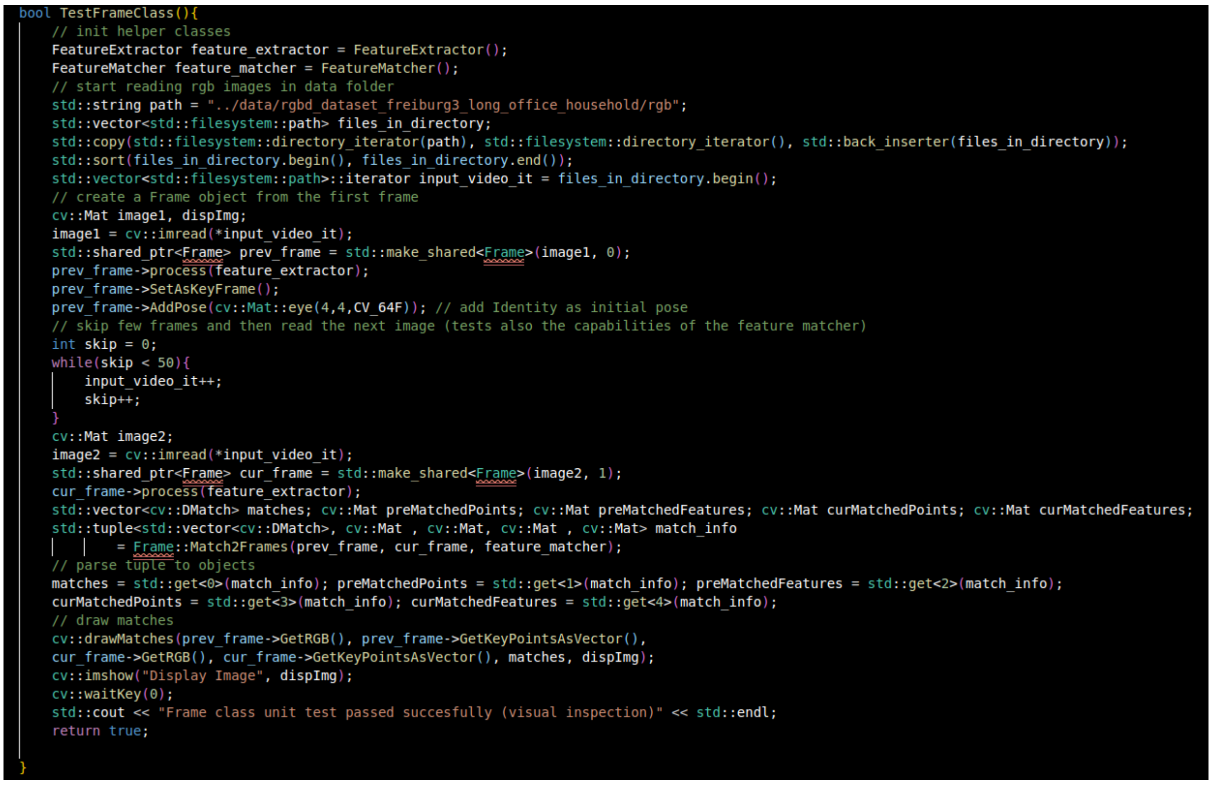
The testing was split into three parts: Unit tests, manual testing (“does the full program behave as expected”), and tooling such as valgrind & static analyzers. Unit tests were used for the Frame and Point classes since they had clearly defined behavior that could be checked in code. End-to-end tests were used in the Map class and the visualization, and they were done manually because their “correct” behavior was complex. Finally, we used static analysis with linters and valgrind throughout the whole project to make sure none of our changes caused a hidden bug in the programs even if they succeeded in the other testing.

Valgrind turned out to be very helpful, as we were able to catch many subtle memory leaks in the code. For example in the PointClouds class it’s required for the user to call the destructor of the “renderer” attribute in easy3d::Graph objects manually, which caused a memory leak that we were able to catch during development thanks to the proper tooling.

Frame class:

Frame class implementation was tested continuously with TestFrameClass() function which tests the main functionalities of the class (and its helpers FeatureExtractor and FeatureMatcher):

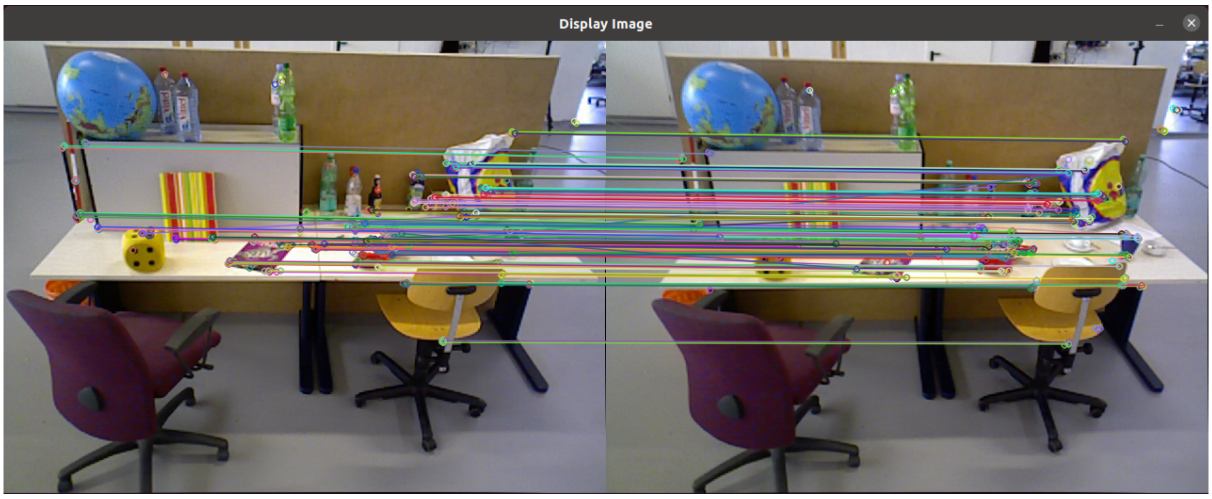
- create 2 Frame objects (std::shared_ptr<Frame> in this case as they should be added to map as such)
- extract keypoints and features in the frames by calling process()
- match keypoints and features by calling static function Match2Frames()
- visualize matches with the drawMatches() function of OpenCV



```cpp
bool TestFrameClass(){
    // init helper classes
    FeatureExtractor feature_extractor = FeatureExtractor();
    FeatureMatcher feature_matcher = FeatureMatcher();
    // start reading rgb images in data folder
    std::string path = "../data/rgbd_dataset_freiburg3_long_office_household/rgb";
    std::vector<std::filesystem::path> files_in_directory;
    std::copy(std::filesystem::directory_iterator(path), std::filesystem::directory_iterator(), std::back_inserter(files_in_directory));
    std::sort(files_in_directory.begin(), files_in_directory.end());
    std::vector<std::filesystem::path>::iterator input_video_it = files_in_directory.begin();
    // create a Frame object from the first frame
    cv::Mat image1, dispImg;
    image1 = cv::imread(*input_video_it);
    std::shared_ptr<Frame> prev_frame = std::make_shared<Frame>(image1, 0);
    prev_frame->process(feature_extractor);
    prev_frame->SetAsKeyFrame();
    prev_frame->AddPose(cv::Mat::eye(4,4,CV_64F)); // add Identity as initial pose
    // skip few frames and then read the next image (tests also the capabilities of the feature matcher)
    int skip = 0;
    while(skip < 50){
        input_video_it++;
        skip++;
    }
    cv::Mat image2;
    image2 = cv::imread(*input_video_it);
    std::shared_ptr<Frame> cur_frame = std::make_shared<Frame>(image2, 1);
    cur_frame->process(feature_extractor);
    std::vector<cv::DMatch> matches; cv::Mat preMatchedPoints; cv::Mat preMatchedFeatures; cv::Mat curMatchedPoints; cv::Mat curMatchedFeatures;
    std::tuple<std::vector<cv::DMatch>, cv::Mat , cv::Mat, cv::Mat , cv::Mat> match_info
            = Frame::Match2Frames(prev_frame, cur_frame, feature_matcher);
    // parse tuple to objects
    matches = std::get<0>(match_info); preMatchedPoints = std::get<1>(match_info); preMatchedFeatures = std::get<2>(match_info);
    curMatchedPoints = std::get<3>(match_info); curMatchedFeatures = std::get<4>(match_info);
    // draw matches
    cv::drawMatches(prev_frame->GetRGB(), prev_frame->GetKeyPointsAsVector(),
    cur_frame->GetRGB(), cur_frame->GetKeyPointsAsVector(), matches, dispImg);
    cv::imshow("Display Image", dispImg);
    cv::waitKey(0);
    std::cout << "Frame class unit test passed succesfully (visual inspection)" << std::endl;
    return true;

}
```

The resulting visual output:



Point class

In point class unit testing is implemented as follows.

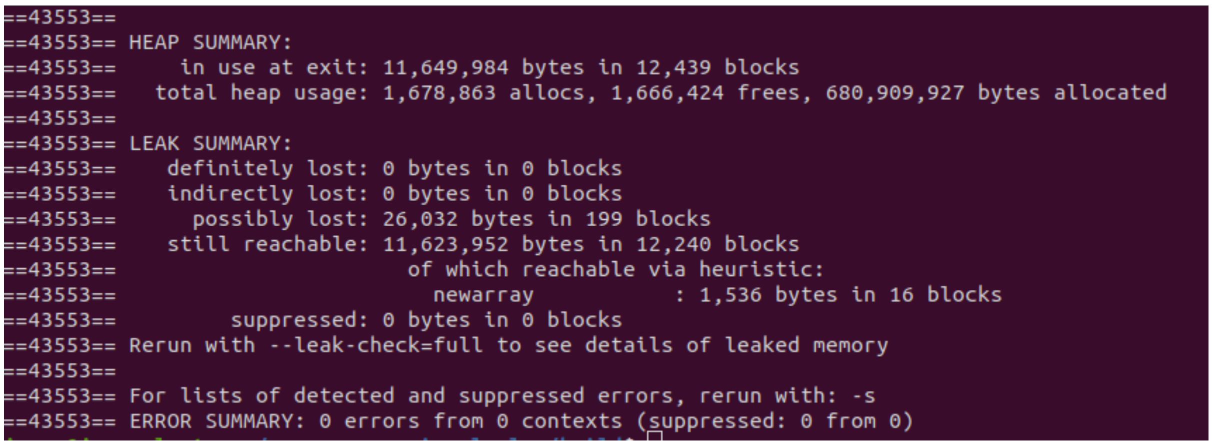
1. Create pointer to point object with point id and 3D location
2. Check that point object stores the point id correctly
3. check that point object stores the 3D location correctly

Map class + Visualization:

To test the whole program we need to basically run the whole program, since its functions implement the ui and main algorithmic load and it's hard to split up into parts. So for us calling main() ended up being the test function for the Map class. This is a an end-to-end test but it wasn't automated meaning we determined whether the tests were successful based on the visual output (also not crashing or slowing down). This also tests the interplay between Point3D and Frame classes. For the future, we could have been able to take copies of our testing main() functions and put them into new files so we could test different kinds of functionality in the program.

4.1 Valgrind memory check

Here is the result when running valgrind ./run_slam --leak-check=full in the build directory without the screen implementation.



```
==43553==
==43553== HEAP SUMMARY:
==43553==     in use at exit: 11,649,984 bytes in 12,439 blocks
==43553==   total heap usage: 1,678,863 allocs, 1,666,424 frees, 680,909,927 bytes allocated
==43553==
==43553== LEAK SUMMARY:
==43553==    definitely lost: 0 bytes in 0 blocks
==43553==    indirectly lost: 0 bytes in 0 blocks
==43553==      possibly lost: 26,032 bytes in 199 blocks
==43553==    still reachable: 11,623,952 bytes in 12,240 blocks
==43553==                       of which reachable via heuristic:
==43553==                         newarray           : 1,536 bytes in 16 blocks
==43553==         suppressed: 0 bytes in 0 blocks
==43553== Rerun with --leak-check=full to see details of leaked memory
==43553==
==43553== For lists of detected and suppressed errors, rerun with: -s
==43553== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5. Work log

The main division of work was that Juuso and Jere implement the SLAM, and Olivia implements the visualization of the map produced by the SLAM.

Responsibilities of individual group members:

Juuso:

- Frame class (whole implementation)
- Map (in collaboration with Jere):
    - Main contribution to following methods:
        - initializeMap()
        - localTracking()
- From helper functions:
    - EstimateEssential()
    - Triangulate()

Week	What was being done	Hours
43	Thinking project subject and eventually deciding to implement visual SLAM. Looking for project members.	5
44	Setting up programming environment and planning out the project	10
45	Starting out the Frame class	10
46	Implementing Frame class	10

47	Implementing Map class: - map initialization, essential transformation - tracking, solvepnp	15
48	debugging BundleAdjustement and adding of points and frames to map	25
49	debugging, commenting, writing report, getting screen to work with the map	20

Jere:

Week	What was being done	Hours
43	Thinking project subject and eventually deciding to implement visual SLAM. Looking for project members.	4
44	Starting to setup programming environment and stating to creating more specific plan for the project. Also getting familiar with Eigen and opencv syntax.	10
45	Making project plan. Stating to implementing Point and Map classes	10
46	Continuing map class by adding basic features. Also adding more features	10
47	Starting to work with g2o, trying to make g2o to compiile, and starting to implementing bundle adjustment. Initialization of	15

	map.	
48	Continuing bundle adjustment. Starting to implement local mapping. Implementing triangulate method. SolvePnPRansac. Starting to write final report	25
49	Commenting and cleaning the code. Continuing the report.	20

- Point3D class (whole implementation)
- Map (in collaboration with Juuso):
  - Main contribution to following methods:
    - localMapping()
    - BundleAdjustement()
    - initializeMap()
- helper functions

Olivia:

Week	What was being done	Hours
44	Figuring out how to set up the programming environment and start researching my tasks (camera matrices, 3d rendering)	5
45	Establishing plans for how to structure the visualization using a component model and inheritance, search for 3d point cloud visualization libraries	10
46	Add a skeleton for the Screen and ScreenComponents classes, try to set up QT and PCL on my machine. Our fourth team member left the project due to external circumstances around this	15

	time, so we decided to go slightly simpler and use the Easy3D library instead for simplicity.	
47	Implement the logic of the Screen class whilst debugging an ongoing bug where concurrent threads would not be able to update the visualizer	20
48	Refactor the Screen class into the PointClouds class and some freestanding functions to fix the aforementioned bug, partially rewrite and restructure the file, polish the API.	25
49	Writing more documentation and continuing the report, also implement camera pose vectors (the green lines that point in the camera direction) now that previous issues have been resolved	15