UNIVERSITY OF HAIFA

אוניברסיטת חיפה جـامعـة حـيفـا

RT Systems Lab

Weekly Report number (1)

Contact Us

Julian Ewaied 215543497   Namir Ballan 326165156

Contents

This Week's Tasks

☞ Understanding the camera matrix.

☞ Installing OpenCV and Eigne3 libraries for C++

☞ Recovering original coordinates

☞ Considering improvements

Tasks Done

Understanding Camera Matrix

The camera is a physical device which takes a 3D scene, and converts it into pixels. To be more precise, it takes all points $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$ and converts them into points $\begin{pmatrix} x_p \\ y_p \end{pmatrix}$, where all the points are of depth 1. Since this is not a bijection, we need more than one picture to recover the z-coordinate. The camera matrix is a matrix that recovers the points with normalized z coordinate $\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$. Let C let the camera matrix, then:

$$C = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

Where (c_x, c_y) is the center of the image in pixels and (f_x, f_y) are the focal length of the image.

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = C^{-1} \begin{pmatrix} x_p \\ y_p \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} x_p \\ y_p \\ 1 \end{pmatrix}$$

With simple calculations one can see that:

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} \dfrac{1}{f_x} & 0 & \dfrac{-c_x}{f_x} \\ 0 & \dfrac{1}{f_y} & \dfrac{-c_y}{f_y} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_p \\ y_p \\ 1 \end{pmatrix} = \begin{pmatrix} \dfrac{x_p - c_x}{f_x} \\ \dfrac{y_p - c_y}{f_y} \\ 1 \end{pmatrix}$$

If d is the depth of the point, then

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = d \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Now let's look at two frames with vertical translation upwards as described in the following figure.
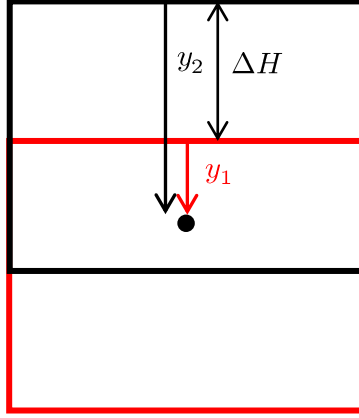
Figure1 : read frame is frame k-1, and black frame is frame k. we look at the y coordinate of the same point in two different frames, and calculate the depth using the distance

We can see that

$$\frac{\Delta H}{y_2} = \frac{y_2 - y_1}{y_2} = \frac{y_2 - y_1}{y_2} = \frac{y_{p_2} - y_{p_1}}{f_y y_2}$$

And therefore, we can conclude that:

$$d = \frac{y_2}{y_2} = \frac{f_y \Delta H}{y_{p_2} - y_{p_1}} = f_y \frac{\Delta H}{\Delta y_p} = f_y \left| \frac{\partial H}{\partial y_p} \right|$$

And hence,

$$d = f_y \left| \frac{\partial H}{\partial y_p} \right|$$

Therefore, we can conclude that:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = dC^{-1} \begin{pmatrix} x_p \\ y_p \\ 1 \end{pmatrix} = f_y \left| \frac{\partial H}{\partial y_p} \right| \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} x_p \\ y_p \\ 1 \end{pmatrix}$$

Installing OpenCV and Eigen3 libraries for C++

All the packages we needed are OpenCV for image files handling, and Eigen3 for doing the calculations as fast as possible. The installation was very complex at the beginning, but then we realized we can simplify it using visual studio. After downloading the code files of each of the libraries, all we need to do is to add the path to the library code to the properties. The following videos explain perfectly how to do it.

| Download OpenCV	Download Eigen	OpenCV tutorial	Eigen tutorial

Note! OpenCV automatic installer can't download to a privileged folder, be careful.

We ran a sample OpenCV code (and also Eigen3 code, but it's less interesting):

```cpp
#include <iostream>
#include <opencv2/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
using namespace cv;
int Run()
{
    cv::Mat img = cv::imread("C:/Users/WIN10PRO/Pictures/fish.png");
    namedWindow("Test", WINDOW_AUTOSIZE);
    imshow("Test", img);
    cv::waitKey(0);
    cv::destroyAllWindows();
    return 0;
}
```

Here is the result:
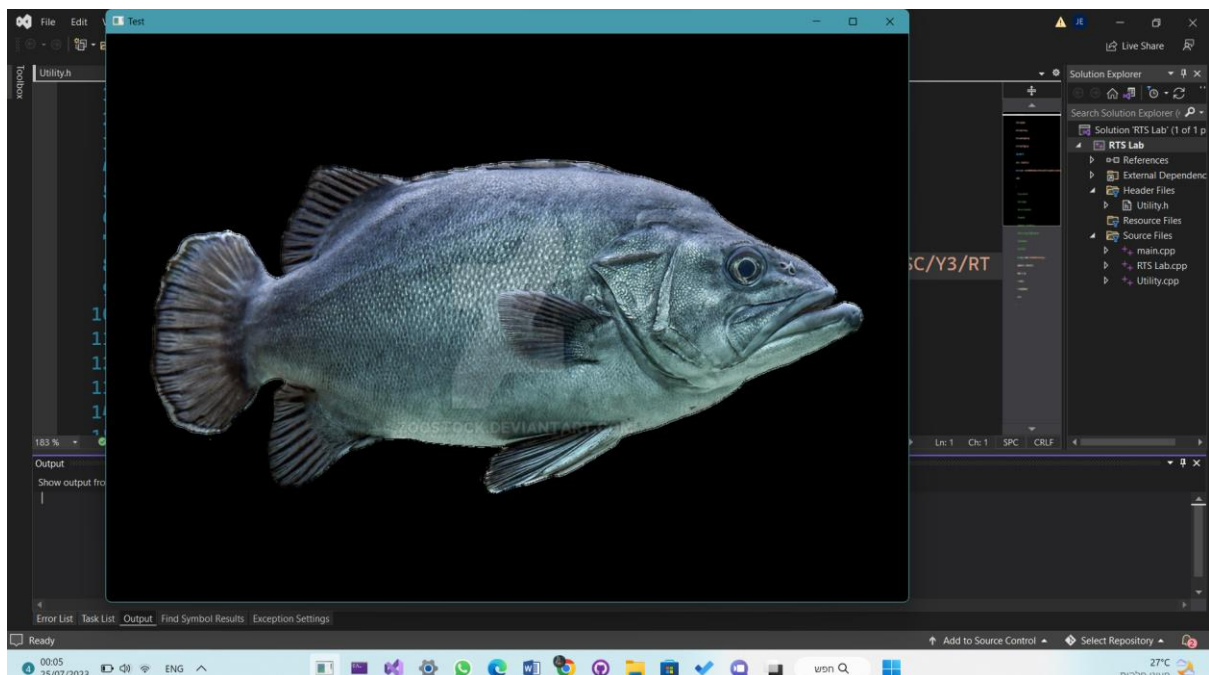
We also fixed the CSV class to make it return a matrix of Eigen::Vector3i objects instead of tuples for later use, and tested it for a simple csv file with the following driver code.
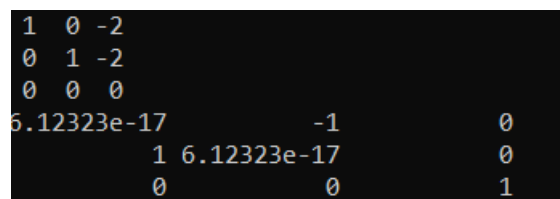
```cpp
#include <iostream>
#include <Eigen/Dense>
#include "../include/Utility.h"
const char* path = "C:/Users/WIN10PRO/Desktop//test.csv";
int Run()
{
    CSVFile file(path);
    file.openFile();
    vector<matrix> v = file.readFile(0, 1);
    std::cout << v[0][0][1] << std::endl;
    file.setPath("None");
    file.closeFile();
}
```

Recovering Original Coordinates

We are currently implementing the calculation $\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = C^{-1} \begin{pmatrix} x_p \\ y_p \\ 1 \end{pmatrix}$. For that we built a class called Analyzer, which takes as an input the camera information, processes the data. The first step was to build the camera matrix inverse, which we did in two steps: first building, then retrieving the matrix. This way, we won't calculate the inverse camera matrix each time we need it, but rather only the first time. Notice that there is no need for us to change the camera parameters in the running stage, so we simply kept them private with no setting methods. We ran a sample example and got the following inverse camera matrix:

We also had a static function that given a rotation axis and a rotation angle in degrees, it returns the rotation matrix. Here's a sample code that we used to do a sanity-check.

```cpp
int Run()
{
    CSVFile file(path);
    Analyzer a(1, 1, 2, 2);
    a.buildCameraMatrix();
    cout << a.getCameraMatrix() << endl;
    cout << Analyzer::getRotationMatrix(90, Eigen::Vector3d(0, 0, 1));
}
```

```
1  0 -2
0  1 -2
0  0  0
6.12323e-17          -1            0
          1 6.12323e-17            0
          0            0            1
```

Next, we will need to get a vector of 2D integer points, and convert them all into the normalized 3D form, which will be done with the function project3D.

We built a function mapPoints, that takes in a vector of 3D points, and applies the calculation:

```cpp
vector<Eigen::Vector3d> Analyzer::mapNormalizedPoints(const
vector<Eigen::Vector2d>& points)
{
      vector<Vector3d> mapped(points.size());
      if (!matrixBuilt) buildCameraMatrix();
      for (int i=0;i<points.size();i++)
            mapped[i] = CameraMatrix * points[i].homogeneous();
      return mapped;
}
vector<Eigen::Vector3d>  Analyzer::mapPoints(const vector<Eigen::Vector2d>&
centers, const vector<Eigen::Vector2d>& mv, double dH)
{
      auto normalized = this->mapNormalizedPoints(centers);
      vector<Vector3d> points;
      // now calculate d = fy * Delta(H)/Delta(y)
      // assuming constant partial derivative of H.
      for (int i =0;i<normalized.size();i++)
      {
            if(mv[i](1) != 0)
                  points.push_back(normalized[i] * fy * dH / mv[i](1));
      }
      return points;
}
```

Or in short, the calculation:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \frac{f_y \Delta H}{\Delta y} C^{-1} \begin{pmatrix} x_p \\ y_p \\ 1 \end{pmatrix}$$

Then, this data is sent to the main function which takes it all and sends it to the class PointDisplayer, which draws a top-down view of the room, and plots it to the screen.

We ran the code on close.csv, only one angle, and here are the results.

Considering Improvements

One can see that the results aren't promising at all. We looked at the data and its processing, and realized the following:

a) The fps rate is very high, hence most of the motion vectors are 0, thus useless.

b) The height sensor is very weak. It could only give us multiples of 10 cm, which means that we don't get the actual height difference (lack of accuracy).

c) The drone is very slow, we need it to move more and get more information.

These are our first 3 issues that we need to deal with. For the first one, we will be trying lower fps rate. For the second one, since this is all the drone has got to give, we will need to think of an imaginary function that approximates the heights from the sensor, but gives non-uniform height differences. And finally, for the third one, we will be giving the drone more height to go to, and use only the part where its moving fast enough.

Next Week's Tasks

☞ Running on a full room (360 degrees)

☞ Testing lower fps rates.

☞ Testing different height approximation functions.

☞ Testing different maximal heights

☞ Moving to Real Time setup.