UNIVERSITY OF HAIFA

אוניברסיטת חיפה جامعة حيفا

RT Systems Lab

Weekly Report number (5)

Contact Us

Julian Ewaied 215543497 Namir Ballan 326165156

8/24/2023

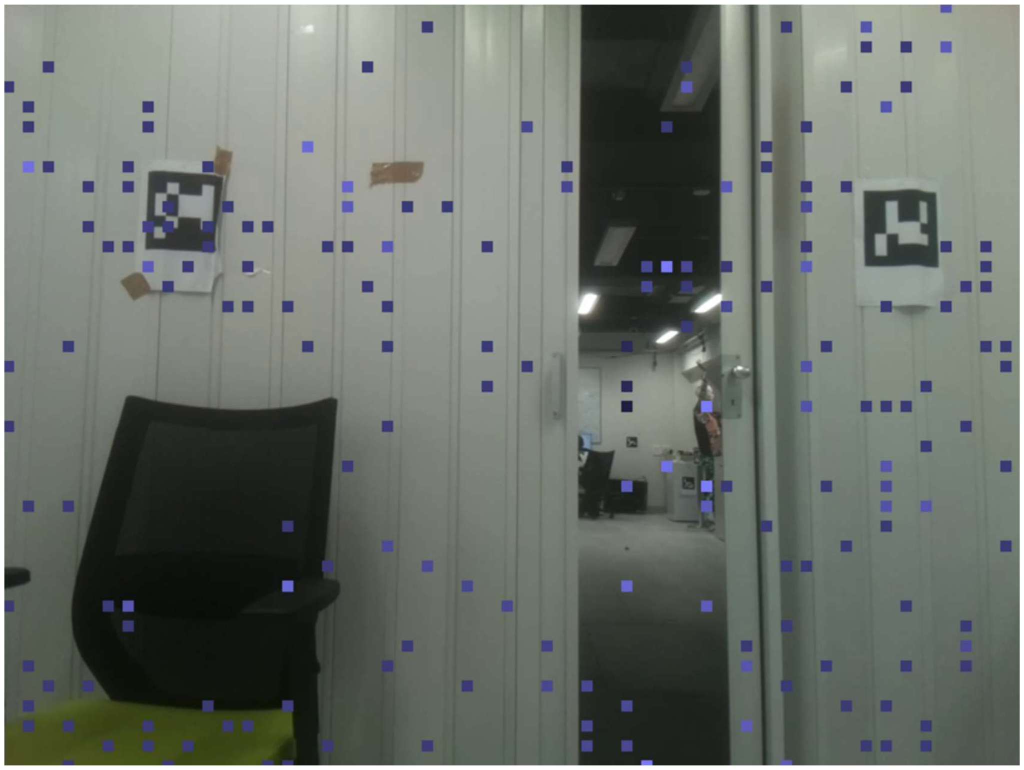
Contents

This Week's Tasks

- ☞ Implementing Average Color Filtering Algorithm
- ☞ Implementing Randomized Color Filtering Algorithm
- ☞ Implementing the Median Compressing Algorithm
- ☞ Getting Data from another room
- ☞ Searching up DNN libraries for C++.

Implementing Average Color Filtering Algorithm

First of all, we attach a picture showing how the floor is problematic. We can see the floor is quite bright, meaning its interoperated as close points.



I implemented the algorithm described in the previous report. For each frame, I took the diagonal of each macro block and calculated its average color. Then, I calculated the sum of squared distances between each macro-block and its neighbors, and applied a threshold:

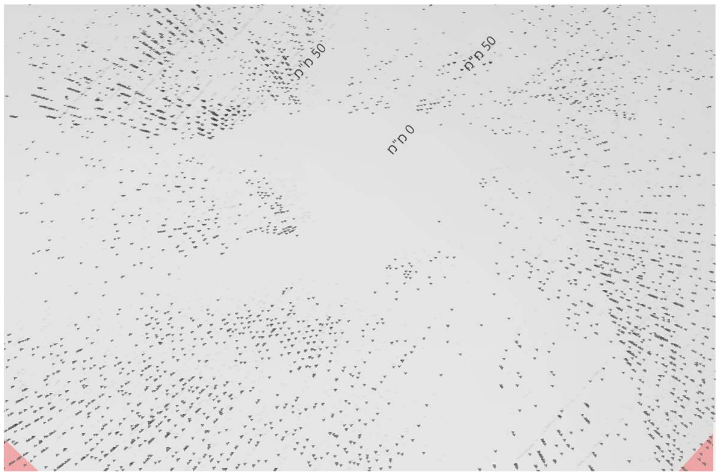
$$filter(ss) = \begin{cases} 1, ss \geq f \\ 0, ss < f \end{cases}$$

Then multiplied motion-vectors by $filter(ss)$, and this way if $filter(ss) = 0$, $dy = 0$, which means our algorithm ignores it. Here is the code for what we described.

```
vector<int8_t> processImage(cv::Mat image)
{
    uint8_t* pixels = static_cast<uint8_t*>(image.data);
    int cn = image.channels();
    vector<int8_t> flags;
    // step 1: average each macro-block
    vector<vector<cv::Vec3d>> colAvg;
    for (int i = 0;i < image.rows-16;i+=16)
    {
        colAvg.push_back(vector<cv::Vec3d>());
        for (int j = 0;j < image.cols-16;j+=16)
        {
            cv::Vec3d avg(0, 0, 0);
            // k moves between diagonals
            for (int k = 0; k < 16;k++)
            {
                if (i + k < image.rows && j + k < image.cols)
                {

                    cv::Vec3d x;
                    x(0) = pixels[i * cn * image.cols + j * cn + 0];
                    x(1) = pixels[i * cn * image.cols + j * cn + 1];
                    x(2) = pixels[i * cn * image.cols + j * cn + 2];
                    avg = avg + x;
                }
            }
            avg = (1.0 / 16) * avg;
            colAvg[i/16].push_back(avg);
        }
    }

    // step 2: calculate the distances between all adjacent macro-blocks
    for (int i = 0;i < colAvg.size();i++)
        for (int j = 0; j < colAvg[0].size();j++)
        {
            long long sum = 0;
            if (i > 0)
            {
                sum += cv::norm(colAvg[i][j] - colAvg[i - 1][j]);
                if (j > 0)
                {
                    sum += std::pow((cv::norm(colAvg[i][j] - colAvg[i-1][j - 1])),2);
                    sum += std::pow(cv::norm(colAvg[i][j] - colAvg[i][j - 1]), 2);
                }
                if (j < colAvg[0].size() - 1)
                    sum += std::pow(cv::norm(colAvg[i][j] - colAvg[i - 1][j + 1]),2);
            }
            if (i < colAvg.size() - 1)
            {
                sum += std::pow(cv::norm(colAvg[i][j] - colAvg[i + 1][j]),2);
                if (j > 0)
                    sum += std::pow(cv::norm(colAvg[i][j] - colAvg[i + 1][j - 1]),2);
                if (j < colAvg[0].size() - 1)
                {
                    sum += std::pow(cv::norm(colAvg[i][j] - colAvg[i + 1][j + 1]),2);
                    sum += std::pow(cv::norm(colAvg[i][j] - colAvg[i][j + 1]),2);
                }
            }
            // step 3: for every macro block with distances sum<f, write 0 (which
will cancel the motion vector)
            if (sum < COLOR_FILTER)
            {
                flags.push_back(0);
            }
            else
                flags.push_back(1);
```

```cpp
        }
    return flags;
}

frames filterMVs(const frames& motionVectors, const vector<int8_t>& filters)
{
    frames mvs;
    for (int i = 0;i < motionVectors.size();i++)
    {
        if (filters[i])
            mvs.push_back(motionVectors[i]);
        else
            mvs.push_back(MotionVector(0, 0));
    }
    return mvs;
}
// writes points to obj file to be projected in 3D Builder with filtering
void writeFilteredOBJ(const vector<string>& mvFiles, const vector<string>&
heightFiles, const vector<string>& vidFiles,const string& outputPath)
{
    std::ofstream out;
    out.open(outputPath);
    if (!out.is_open())
        exit(1);
    vector<Eigen::Vector3d> points;
    Analyzer a(fx, fy, cx, cy);
    for (int i = 0;i < mvFiles.size();i++)
    {

        // import all data for file
        auto motionVectors = a.importMV(mvFiles[i]);
        vector<cv::Mat> images = extractFrames(vidFiles[i]);
        CSVFile height_file(heightFiles[i], NUM_FRM);
        height_file.openFile();
        auto heights = height_file.readColumn();
        auto centers = a.getCenters();
        CSVFile sads_file(mvFiles[i], NUM_FRM);
        sads_file.openFile();
        auto sads = sads_file.getSAD();
        // cont_heights is a backup for calculating depths
        vector<double> cont_heights;
        for (auto h : heights)
        {
            cont_heights.push_back(h);
        }
        vector<Eigen::Vector3d> tmp;
        // continuize the heights function.
        a.continuize(cont_heights);
        // for each frame in the video
        for (int k = 1;k < motionVectors.size();k++)
        {
            // get the mapped points then add the heights, and add it to the cloud
            vector<int8_t> filter_images = processImage(images[k]);
            motionVectors[k] = filterMVs(motionVectors[k], filter_images);
            vector<Eigen::Vector3d> tmp = a.mapPoints(centers, motionVectors[k],
cont_heights[k] - cont_heights[k - 1], sads[k]);
            for (auto v : tmp)
                v(1) += heights[k];
            Analyzer::rotatePoints(tmp, 60 * i);
            for (int j = 0;j < tmp.size();j++)
                if (NORM(tmp[j](0), tmp[j](2)) < FILTER_RADIUS)
                    points.push_back(tmp[j]);
        }
        std::cout << "Processing Angle : " << 60 * i << std::endl;

    }
    // write to obj file
    int line = 1;
    vector<string> faces;
    for (auto point : points)
    {
        string s = "f ";
        for (int k = 0;k < 3;k++) {
            int A = 1.5;
            out << "v " << point(0) + ((k % 3) / 2) * A << " " << point(2) + (((k + 
1) % 3) / 2) * A << " " << point(1) / 10 + A * ((k + 2) % 3) / 2 << std::endl;
            s = s + std::to_string(line++) + " ";
        }
        faces.push_back(s + "\n");
    }
    for (auto f : faces)
    {
        out << f;
    }

}
```

Here are the results.



We can see that the room maintained its shape, but has lost lots of points, which makes it hard for us to see the shape of the room, and even harder for a computer to recognize. However, we can see that at the exit it is almost a clear area, which defines a tradeoff between how clear the room borders are and how clean the exit is.

Randomized Color Average Algorithm

We could already see that the tradeoff we have is very extreme that we don't need to use the color average algorithm. Our explanation to this extreme tradeoff is that macro-blocks of the same color represent together an object, which is of some size. By applying our algorithm we are only taking the "borders" of these objects, which makes it shallower and harder to recognize with a top-down view. Therefore, we will be giving up this algorithm.

Median Compressing Algorithm

We heard from Bar that another group ran this algorithm on another dataset and got some very good results. We believe this algorithm would also work on our dataset, and that it is useless to reimplement it again.

Getting Data and Searching DNN Libraries

Unfortunately, we will not be able to run on another room and use DL algorithms to optimize the filtering, but we can always return to this project and improve it all the time.

Next Week's Tasks

- ☞ Summarizing the project.
- ☞ Recording a demo.
- ☞ Commenting and finalizing the code.