Introduction to Visual Computing

Assignment #13

Tracking - Bonus

May 18, 2020

Description

We want now to fasten our app by actually tracking the corners of the board.

The algorithm that is used here is a Kalman Filter. We will implement it this week.

Objectives

Based on the results of corner detection from last week, using the Kalman Filter algorithm to track the corners in a webcam stream.

Specific Challenges

Understanding a typical, real-world, image processing algorithm.

Preliminary steps

If needed, finish last week assignment: it is a pre-requisite for today.

Part I

Principle of Tracking

Several reasons explain why detection algorithms are often coupled with tracking techniques:

1- Tracking is faster than detection: Tracking is based on prior knowledge obtained at previous frames. Tracking algorithms use previous estimation of pose, speed and direction of the tracked object. The basic principle of tracking is to use this information to predict the location of the object in the next frame, whereas detection starts from scratch at each frame. Efficient systems usually run object detection every nth frame, and rely on a tracking algorithm in the n-1 frames in between.

Why don't we simply detect the object in the first frame and track it in all subsequent frames? It is true that tracking benefits from the extra information it has, but you can also lose track of an object when it goes behind an obstacle for an extended period of time, or if it moves so fast that the tracking algorithm cannot catch up. It is also common for tracking algorithms to accumulate errors, so that the bounding box tracking the object slowly drifts away from the object it is tracking. To fix these problems with tracking algorithms, a detection algorithm is run every so often.

2- Tracking can help when detection fails: If you are running a face detector on a video and the person's face gets occluded by an object, the face detector will most likely fail. A good tracking algorithm, on the other hand, can handle some level of occlusion.

3- Tracking preserves identity: The output of object detection is an array of rectangles that contain detected objects. However, there is no identity attached to the object. For example, in your system, the detector that detects corners will output positions corresponding to the 4 most probable corners it has detected in a frame. In the next frame, it will output another array of points. In the first frame, a particular corner might be represented by the point at location 10 in the array, while in the second frame the same corner could be at location 17. While using detection on a frame we have no idea which corners and lines corresponds to which object. On the other hand, tracking provides a way to literally connect the dots!

Kalman Filter

The Kalman Filter works in two major steps: the **prediction** and the **correction**. In the prediction step, the position of the tracked object is estimated on the basis of an a-priori position estimate. In the correction step, given the actual position of the tracked object, the model updates the prediction (a-posteriori) and updates the estimated error of the Kalman Filter.

$$\begin{tikzcd} \text{Time Update (‘Predict’)} \arrow[r, bend left] & \text{Measurement Update (‘Correct’)} \arrow[l, bend left] \end{tikzcd}$$

Part II

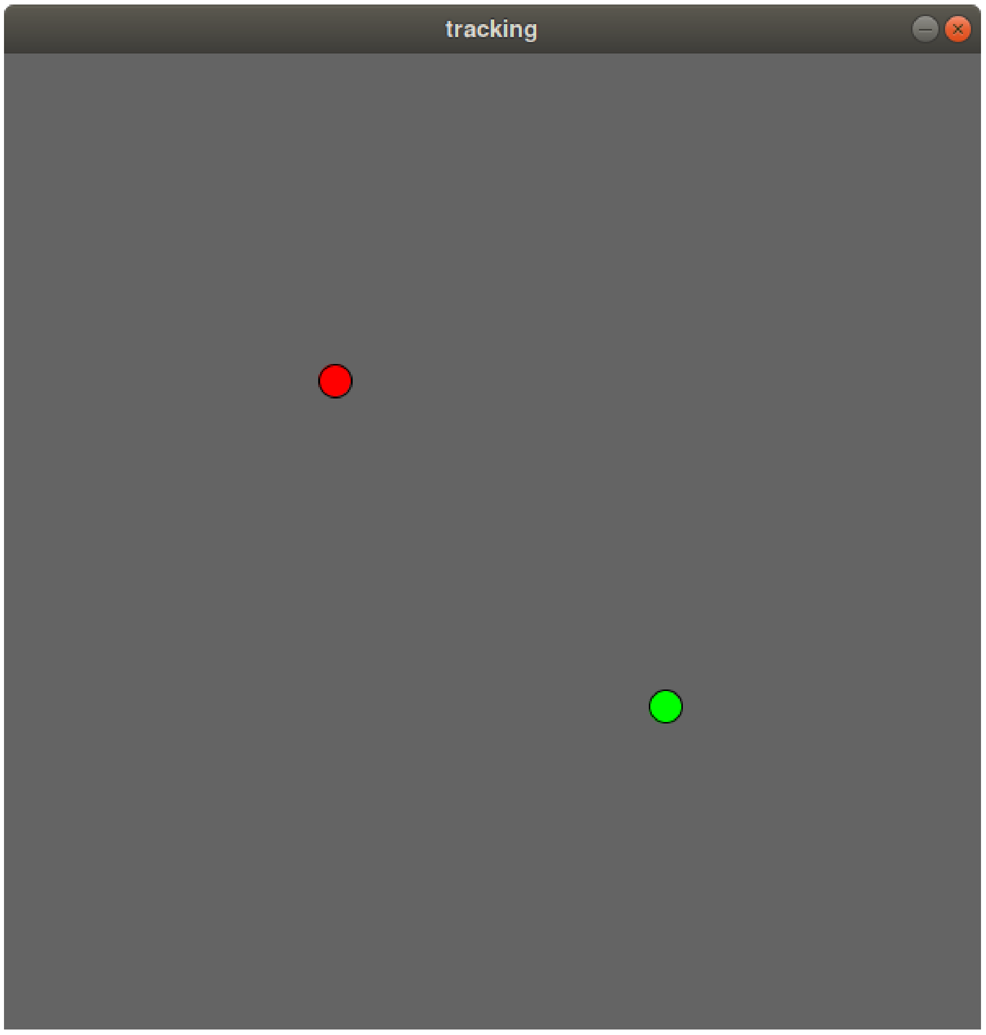
A Basic Kalman Filter

The aim of today's assignment is to efficiently use the Kalman Filter (KF) to track the corners of the board.

In order for you to understand the parameters used in classical KF, we propose to use a KF to track the mouse. In a small separate sketch, you will test various values of the KF parameters and understand their implication on the accuracy of the tracking.

Step 1 – Create a simple sketch

Create a simple sketch drawing a red `circle` for the actual mouse position (use `mouseX` and `mouseY`) and a green `circle` that will display the estimated position using the tracking.



Step 2 – Kalman 2D

Write the `KalmanFilter2D` class that adapts the given `KalmanFilter1D` class (which allows to track the x-position of a 1D object) to be used for `PVector`.

```
class KalmanFilter1D {
    float q = 1;        // process variance
    float r = 2.0;      // estimate of measurement variance, change to see effect
    float xhat = 0.0;   // a posteriori estimate of x
    float xhatminus;    // a priori estimate of x
    float p = 1.0;      // a posteriori error estimate
    float pminus;       // a priori error estimate
    float kG = 0.0;     // kalman gain

    KalmanFilter1D() {};
    KalmanFilter1D(float q, float r) {
        q(q);
        r(r);
    }

    void q(float q) {
        this.q = q;
    }

    void r(float r) {
        this.r = r;
    }

    float xhat() {
        return this.xhat;
    }

    void predict() {
        xhatminus = xhat;
        pminus = p + q;
    }

    float correct(float x) {
        kG = pminus / (pminus + r);
        xhat = xhatminus + kG * (x - xhatminus);
        p = (1 - kG) * pminus;
        return xhat;
    }

    float predict_and_correct(float x) {
        predict();
        return correct(x);
    }
}
```

Step 3 – Test the parameters Q and R

Test the effect of Q and R on tracking and computation.

Part III

Tracking Corners

Now integrate the `KalmanFilter2D` class into your game. In order to track each corner, you will need to instantiate the KF 4 times.

Use the correction method to correct positions of each of the corner, and the prediction to estimate the corner position.

Try not calling the detection methods (all the filters implemented in previous weeks) at every frame and use the `frameCount` variable to determine over how many time it is safe to predict.