#include <iostream>

#include <iomanip>

#include <string>

#include <vector>

#include <queue>

#include <fstream>

#include <thread>

#include <future>

#include <atomic>

#include <mutex> // std::mutex, std::unique\_lock

#include <cmath>

// It makes sense only for video-Camera (not for video-File)

// To use - uncomment the following line. Optical-flow is supported only by OpenCV 3.x - 4.x

//#define TRACK\_OPTFLOW

//#define GPU

// To use 3D-stereo camera ZED - uncomment the following line. ZED\_SDK should be installed.

//#define ZED\_STEREO

#include "yolo\_v2\_class.hpp" // imported functions from DLL

#ifdef OPENCV

#ifdef ZED\_STEREO

#include <sl/Camera.hpp>

#if ZED\_SDK\_MAJOR\_VERSION == 2

#define ZED\_STEREO\_2\_COMPAT\_MODE

#endif

#undef GPU // avoid conflict with sl::MEM::GPU

#ifdef ZED\_STEREO\_2\_COMPAT\_MODE

#pragma comment(lib, "sl\_core64.lib")

#pragma comment(lib, "sl\_input64.lib")

#endif

#pragma comment(lib, "sl\_zed64.lib")

float getMedian(std::vector<float> &v) {

size\_t n = v.size() / 2;

std::nth\_element(v.begin(), v.begin() + n, v.end());

return v[n];

}

std::vector<bbox\_t> get\_3d\_coordinates(std::vector<bbox\_t> bbox\_vect, cv::Mat xyzrgba)

{

bool valid\_measure;

int i, j;

const unsigned int R\_max\_global = 10;

std::vector<bbox\_t> bbox3d\_vect;

for (auto &cur\_box : bbox\_vect) {

const unsigned int obj\_size = std::min(cur\_box.w, cur\_box.h);

const unsigned int R\_max = std::min(R\_max\_global, obj\_size / 2);

int center\_i = cur\_box.x + cur\_box.w \* 0.5f, center\_j = cur\_box.y + cur\_box.h \* 0.5f;

std::vector<float> x\_vect, y\_vect, z\_vect;

for (int R = 0; R < R\_max; R++) {

for (int y = -R; y <= R; y++) {

for (int x = -R; x <= R; x++) {

i = center\_i + x;

j = center\_j + y;

sl::float4 out(NAN, NAN, NAN, NAN);

if (i >= 0 && i < xyzrgba.cols && j >= 0 && j < xyzrgba.rows) {

cv::Vec4f &elem = xyzrgba.at<cv::Vec4f>(j, i); // x,y,z,w

out.x = elem[0];

out.y = elem[1];

out.z = elem[2];

out.w = elem[3];

}

valid\_measure = std::isfinite(out.z);

if (valid\_measure)

{

x\_vect.push\_back(out.x);

y\_vect.push\_back(out.y);

z\_vect.push\_back(out.z);

}

}

}

}

if (x\_vect.size() \* y\_vect.size() \* z\_vect.size() > 0)

{

cur\_box.x\_3d = getMedian(x\_vect);

cur\_box.y\_3d = getMedian(y\_vect);

cur\_box.z\_3d = getMedian(z\_vect);

}

else {

cur\_box.x\_3d = NAN;

cur\_box.y\_3d = NAN;

cur\_box.z\_3d = NAN;

}

bbox3d\_vect.emplace\_back(cur\_box);

}

return bbox3d\_vect;

}

cv::Mat slMat2cvMat(sl::Mat &input) {

int cv\_type = -1; // Mapping between MAT\_TYPE and CV\_TYPE

if(input.getDataType() ==

#ifdef ZED\_STEREO\_2\_COMPAT\_MODE

sl::MAT\_TYPE\_32F\_C4

#else

sl::MAT\_TYPE::F32\_C4

#endif

) {

cv\_type = CV\_32FC4;

} else cv\_type = CV\_8UC4; // sl::Mat used are either RGBA images or XYZ (4C) point clouds

return cv::Mat(input.getHeight(), input.getWidth(), cv\_type, input.getPtr<sl::uchar1>(

#ifdef ZED\_STEREO\_2\_COMPAT\_MODE

sl::MEM::MEM\_CPU

#else

sl::MEM::CPU

#endif

));

}

cv::Mat zed\_capture\_rgb(sl::Camera &zed) {

sl::Mat left;

zed.retrieveImage(left);

cv::Mat left\_rgb;

cv::cvtColor(slMat2cvMat(left), left\_rgb, CV\_RGBA2RGB);

return left\_rgb;

}

cv::Mat zed\_capture\_3d(sl::Camera &zed) {

sl::Mat cur\_cloud;

zed.retrieveMeasure(cur\_cloud,

#ifdef ZED\_STEREO\_2\_COMPAT\_MODE

sl::MEASURE\_XYZ

#else

sl::MEASURE::XYZ

#endif

);

return slMat2cvMat(cur\_cloud).clone();

}

static sl::Camera zed; // ZED-camera

#else // ZED\_STEREO

std::vector<bbox\_t> get\_3d\_coordinates(std::vector<bbox\_t> bbox\_vect, cv::Mat xyzrgba) {

return bbox\_vect;

}

#endif // ZED\_STEREO

#include <opencv2/opencv.hpp> // C++

#include <opencv2/core/version.hpp>

#ifndef CV\_VERSION\_EPOCH // OpenCV 3.x and 4.x

#include <opencv2/videoio/videoio.hpp>

#define OPENCV\_VERSION CVAUX\_STR(CV\_VERSION\_MAJOR)"" CVAUX\_STR(CV\_VERSION\_MINOR)"" CVAUX\_STR(CV\_VERSION\_REVISION)

#ifndef USE\_CMAKE\_LIBS

#pragma comment(lib, "opencv\_world" OPENCV\_VERSION ".lib")

#ifdef TRACK\_OPTFLOW

/\*

#pragma comment(lib, "opencv\_cudaoptflow" OPENCV\_VERSION ".lib")

#pragma comment(lib, "opencv\_cudaimgproc" OPENCV\_VERSION ".lib")

#pragma comment(lib, "opencv\_core" OPENCV\_VERSION ".lib")

#pragma comment(lib, "opencv\_imgproc" OPENCV\_VERSION ".lib")

#pragma comment(lib, "opencv\_highgui" OPENCV\_VERSION ".lib")

\*/

#endif // TRACK\_OPTFLOW

#endif // USE\_CMAKE\_LIBS

#else // OpenCV 2.x

#define OPENCV\_VERSION CVAUX\_STR(CV\_VERSION\_EPOCH)"" CVAUX\_STR(CV\_VERSION\_MAJOR)"" CVAUX\_STR(CV\_VERSION\_MINOR)

#ifndef USE\_CMAKE\_LIBS

#pragma comment(lib, "opencv\_core" OPENCV\_VERSION ".lib")

#pragma comment(lib, "opencv\_imgproc" OPENCV\_VERSION ".lib")

#pragma comment(lib, "opencv\_highgui" OPENCV\_VERSION ".lib")

#pragma comment(lib, "opencv\_video" OPENCV\_VERSION ".lib")

#endif // USE\_CMAKE\_LIBS

#endif // CV\_VERSION\_EPOCH

void draw\_boxes(cv::Mat mat\_img, std::vector<bbox\_t> result\_vec, std::vector<std::string> obj\_names,

int current\_det\_fps = -1, int current\_cap\_fps = -1)

{

int const colors[6][3] = { { 1,0,1 },{ 0,0,1 },{ 0,1,1 },{ 0,1,0 },{ 1,1,0 },{ 1,0,0 } };

for (auto &i : result\_vec) {

cv::Scalar color = obj\_id\_to\_color(i.obj\_id);

cv::rectangle(mat\_img, cv::Rect(i.x, i.y, i.w, i.h), color, 2);

if (obj\_names.size() > i.obj\_id) {

std::string obj\_name = obj\_names[i.obj\_id];

if (i.track\_id > 0) obj\_name += " - " + std::to\_string(i.track\_id);

cv::Size const text\_size = getTextSize(obj\_name, cv::FONT\_HERSHEY\_COMPLEX\_SMALL, 1.2, 2, 0);

int max\_width = (text\_size.width > i.w + 2) ? text\_size.width : (i.w + 2);

max\_width = std::max(max\_width, (int)i.w + 2);

//max\_width = std::max(max\_width, 283);

std::string coords\_3d;

if (!std::isnan(i.z\_3d)) {

std::stringstream ss;

ss << std::fixed << std::setprecision(2) << "x:" << i.x\_3d << "m y:" << i.y\_3d << "m z:" << i.z\_3d << "m ";

coords\_3d = ss.str();

cv::Size const text\_size\_3d = getTextSize(ss.str(), cv::FONT\_HERSHEY\_COMPLEX\_SMALL, 0.8, 1, 0);

int const max\_width\_3d = (text\_size\_3d.width > i.w + 2) ? text\_size\_3d.width : (i.w + 2);

if (max\_width\_3d > max\_width) max\_width = max\_width\_3d;

}

cv::rectangle(mat\_img, cv::Point2f(std::max((int)i.x - 1, 0), std::max((int)i.y - 35, 0)),

cv::Point2f(std::min((int)i.x + max\_width, mat\_img.cols - 1), std::min((int)i.y, mat\_img.rows - 1)),

color, CV\_FILLED, 8, 0);

putText(mat\_img, obj\_name, cv::Point2f(i.x, i.y - 16), cv::FONT\_HERSHEY\_COMPLEX\_SMALL, 1.2, cv::Scalar(0, 0, 0), 2);

if(!coords\_3d.empty()) putText(mat\_img, coords\_3d, cv::Point2f(i.x, i.y-1), cv::FONT\_HERSHEY\_COMPLEX\_SMALL, 0.8, cv::Scalar(0, 0, 0), 1);

}

}

if (current\_det\_fps >= 0 && current\_cap\_fps >= 0) {

std::string fps\_str = "FPS detection: " + std::to\_string(current\_det\_fps) + " FPS capture: " + std::to\_string(current\_cap\_fps);

putText(mat\_img, fps\_str, cv::Point2f(10, 20), cv::FONT\_HERSHEY\_COMPLEX\_SMALL, 1.2, cv::Scalar(50, 255, 0), 2);

}

}

#endif // OPENCV

void show\_console\_result(std::vector<bbox\_t> const result\_vec, std::vector<std::string> const obj\_names, int frame\_id = -1) {

if (frame\_id >= 0) std::cout << " Frame: " << frame\_id << std::endl;

for (auto &i : result\_vec) {

if (obj\_names.size() > i.obj\_id) std::cout << obj\_names[i.obj\_id] << " - ";

std::cout << "obj\_id = " << i.obj\_id << ", x = " << i.x << ", y = " << i.y

<< ", w = " << i.w << ", h = " << i.h

<< std::setprecision(3) << ", prob = " << i.prob << std::endl;

}

}

std::vector<std::string> objects\_names\_from\_file(std::string const filename) {

std::ifstream file(filename);

std::vector<std::string> file\_lines;

if (!file.is\_open()) return file\_lines;

for(std::string line; getline(file, line);) file\_lines.push\_back(line);

std::cout << "object names loaded \n";

return file\_lines;

}

template<typename T>

class send\_one\_replaceable\_object\_t {

const bool sync;

std::atomic<T \*> a\_ptr;

public:

void send(T const& \_obj) {

T \*new\_ptr = new T;

\*new\_ptr = \_obj;

if (sync) {

while (a\_ptr.load()) std::this\_thread::sleep\_for(std::chrono::milliseconds(3));

}

std::unique\_ptr<T> old\_ptr(a\_ptr.exchange(new\_ptr));

}

T receive() {

std::unique\_ptr<T> ptr;

do {

while(!a\_ptr.load()) std::this\_thread::sleep\_for(std::chrono::milliseconds(3));

ptr.reset(a\_ptr.exchange(NULL));

} while (!ptr);

T obj = \*ptr;

return obj;

}

bool is\_object\_present() {

return (a\_ptr.load() != NULL);

}

send\_one\_replaceable\_object\_t(bool \_sync) : sync(\_sync), a\_ptr(NULL)

{}

};

int main(int argc, char \*argv[])

{

std::string names\_file = "data/coco.names";

std::string cfg\_file = "cfg/yolov3.cfg";

std::string weights\_file = "yolov3.weights";

std::string filename;

if (argc > 4) { //voc.names yolo-voc.cfg yolo-voc.weights test.mp4

names\_file = argv[1];

cfg\_file = argv[2];

weights\_file = argv[3];

filename = argv[4];

}

else if (argc > 1) filename = argv[1];

float const thresh = (argc > 5) ? std::stof(argv[5]) : 0.2;

Detector detector(cfg\_file, weights\_file);

auto obj\_names = objects\_names\_from\_file(names\_file);

std::string out\_videofile = "result.avi";

bool const save\_output\_videofile = false; // true - for history

bool const send\_network = false; // true - for remote detection

bool const use\_kalman\_filter = false; // true - for stationary camera

bool detection\_sync = true; // true - for video-file

#ifdef TRACK\_OPTFLOW // for slow GPU

detection\_sync = false;

Tracker\_optflow tracker\_flow;

//detector.wait\_stream = true;

#endif // TRACK\_OPTFLOW

while (true)

{

std::cout << "input image or video filename: ";

if(filename.size() == 0) std::cin >> filename;

if (filename.size() == 0) break;

try {

#ifdef OPENCV

preview\_boxes\_t large\_preview(100, 150, false), small\_preview(50, 50, true);

bool show\_small\_boxes = false;

std::string const file\_ext = filename.substr(filename.find\_last\_of(".") + 1);

std::string const protocol = filename.substr(0, 7);

if (file\_ext == "avi" || file\_ext == "mp4" || file\_ext == "mjpg" || file\_ext == "mov" || // video file

protocol == "rtmp://" || protocol == "rtsp://" || protocol == "http://" || protocol == "https:/" || // video network stream

filename == "zed\_camera" || file\_ext == "svo" || filename == "web\_camera") // ZED stereo camera

{

if (protocol == "rtsp://" || protocol == "http://" || protocol == "https:/" || filename == "zed\_camera" || filename == "web\_camera")

detection\_sync = false;

cv::Mat cur\_frame;

std::atomic<int> fps\_cap\_counter(0), fps\_det\_counter(0);

std::atomic<int> current\_fps\_cap(0), current\_fps\_det(0);

std::atomic<bool> exit\_flag(false);

std::chrono::steady\_clock::time\_point steady\_start, steady\_end;

int video\_fps = 25;

bool use\_zed\_camera = false;

track\_kalman\_t track\_kalman;

#ifdef ZED\_STEREO

sl::InitParameters init\_params;

init\_params.depth\_minimum\_distance = 0.5;

#ifdef ZED\_STEREO\_2\_COMPAT\_MODE

init\_params.depth\_mode = sl::DEPTH\_MODE\_ULTRA;

init\_params.camera\_resolution = sl::RESOLUTION\_HD720;// sl::RESOLUTION\_HD1080, sl::RESOLUTION\_HD720

init\_params.coordinate\_units = sl::UNIT\_METER;

init\_params.camera\_buffer\_count\_linux = 2;

if (file\_ext == "svo") init\_params.svo\_input\_filename.set(filename.c\_str());

#else

init\_params.depth\_mode = sl::DEPTH\_MODE::ULTRA;

init\_params.camera\_resolution = sl::RESOLUTION::HD720;// sl::RESOLUTION::HD1080, sl::RESOLUTION::HD720

init\_params.coordinate\_units = sl::UNIT::METER;

if (file\_ext == "svo") init\_params.input.setFromSVOFile(filename.c\_str());

#endif

//init\_params.sdk\_cuda\_ctx = (CUcontext)detector.get\_cuda\_context();

init\_params.sdk\_gpu\_id = detector.cur\_gpu\_id;

if (filename == "zed\_camera" || file\_ext == "svo") {

std::cout << "ZED 3D Camera " << zed.open(init\_params) << std::endl;

if (!zed.isOpened()) {

std::cout << " Error: ZED Camera should be connected to USB 3.0. And ZED\_SDK should be installed. \n";

getchar();

return 0;

}

cur\_frame = zed\_capture\_rgb(zed);

use\_zed\_camera = true;

}

#endif // ZED\_STEREO

cv::VideoCapture cap;

if (filename == "web\_camera") {

cap.open(0);

cap >> cur\_frame;

} else if (!use\_zed\_camera) {

cap.open(filename);

cap >> cur\_frame;

}

#ifdef CV\_VERSION\_EPOCH // OpenCV 2.x

video\_fps = cap.get(CV\_CAP\_PROP\_FPS);

#else

video\_fps = cap.get(cv::CAP\_PROP\_FPS);

#endif

cv::Size const frame\_size = cur\_frame.size();

//cv::Size const frame\_size(cap.get(CV\_CAP\_PROP\_FRAME\_WIDTH), cap.get(CV\_CAP\_PROP\_FRAME\_HEIGHT));

std::cout << "\n Video size: " << frame\_size << std::endl;

cv::VideoWriter output\_video;

if (save\_output\_videofile)

#ifdef CV\_VERSION\_EPOCH // OpenCV 2.x

output\_video.open(out\_videofile, CV\_FOURCC('D', 'I', 'V', 'X'), std::max(35, video\_fps), frame\_size, true);

#else

output\_video.open(out\_videofile, cv::VideoWriter::fourcc('D', 'I', 'V', 'X'), std::max(35, video\_fps), frame\_size, true);

#endif

struct detection\_data\_t {

cv::Mat cap\_frame;

std::shared\_ptr<image\_t> det\_image;

std::vector<bbox\_t> result\_vec;

cv::Mat draw\_frame;

bool new\_detection;

uint64\_t frame\_id;

bool exit\_flag;

cv::Mat zed\_cloud;

std::queue<cv::Mat> track\_optflow\_queue;

detection\_data\_t() : exit\_flag(false), new\_detection(false) {}

};

const bool sync = detection\_sync; // sync data exchange

send\_one\_replaceable\_object\_t<detection\_data\_t> cap2prepare(sync), cap2draw(sync),

prepare2detect(sync), detect2draw(sync), draw2show(sync), draw2write(sync), draw2net(sync);

std::thread t\_cap, t\_prepare, t\_detect, t\_post, t\_draw, t\_write, t\_network;

// capture new video-frame

if (t\_cap.joinable()) t\_cap.join();

t\_cap = std::thread([&]()

{

uint64\_t frame\_id = 0;

detection\_data\_t detection\_data;

do {

detection\_data = detection\_data\_t();

#ifdef ZED\_STEREO

if (use\_zed\_camera) {

while (zed.grab() !=

#ifdef ZED\_STEREO\_2\_COMPAT\_MODE

sl::SUCCESS

#else

sl::ERROR\_CODE::SUCCESS

#endif

) std::this\_thread::sleep\_for(std::chrono::milliseconds(2));

detection\_data.cap\_frame = zed\_capture\_rgb(zed);

detection\_data.zed\_cloud = zed\_capture\_3d(zed);

}

else

#endif // ZED\_STEREO

{

cap >> detection\_data.cap\_frame;

}

fps\_cap\_counter++;

detection\_data.frame\_id = frame\_id++;

if (detection\_data.cap\_frame.empty() || exit\_flag) {

std::cout << " exit\_flag: detection\_data.cap\_frame.size = " << detection\_data.cap\_frame.size() << std::endl;

detection\_data.exit\_flag = true;

detection\_data.cap\_frame = cv::Mat(frame\_size, CV\_8UC3);

}

if (!detection\_sync) {

cap2draw.send(detection\_data); // skip detection

}

cap2prepare.send(detection\_data);

} while (!detection\_data.exit\_flag);

std::cout << " t\_cap exit \n";

});

// pre-processing video frame (resize, convertion)

t\_prepare = std::thread([&]()

{

std::shared\_ptr<image\_t> det\_image;

detection\_data\_t detection\_data;

do {

detection\_data = cap2prepare.receive();

det\_image = detector.mat\_to\_image\_resize(detection\_data.cap\_frame);

detection\_data.det\_image = det\_image;

prepare2detect.send(detection\_data); // detection

} while (!detection\_data.exit\_flag);

std::cout << " t\_prepare exit \n";

});

// detection by Yolo

if (t\_detect.joinable()) t\_detect.join();

t\_detect = std::thread([&]()

{

std::shared\_ptr<image\_t> det\_image;

detection\_data\_t detection\_data;

do {

detection\_data = prepare2detect.receive();

det\_image = detection\_data.det\_image;

std::vector<bbox\_t> result\_vec;

if(det\_image)

result\_vec = detector.detect\_resized(\*det\_image, frame\_size.width, frame\_size.height, thresh, true); // true

fps\_det\_counter++;

//std::this\_thread::sleep\_for(std::chrono::milliseconds(150));

detection\_data.new\_detection = true;

detection\_data.result\_vec = result\_vec;

detect2draw.send(detection\_data);

} while (!detection\_data.exit\_flag);

std::cout << " t\_detect exit \n";

});

// draw rectangles (and track objects)

t\_draw = std::thread([&]()

{

std::queue<cv::Mat> track\_optflow\_queue;

detection\_data\_t detection\_data;

do {

// for Video-file

if (detection\_sync) {

detection\_data = detect2draw.receive();

}

// for Video-camera

else

{

// get new Detection result if present

if (detect2draw.is\_object\_present()) {

cv::Mat old\_cap\_frame = detection\_data.cap\_frame; // use old captured frame

detection\_data = detect2draw.receive();

if (!old\_cap\_frame.empty()) detection\_data.cap\_frame = old\_cap\_frame;

}

// get new Captured frame

else {

std::vector<bbox\_t> old\_result\_vec = detection\_data.result\_vec; // use old detections

detection\_data = cap2draw.receive();

detection\_data.result\_vec = old\_result\_vec;

}

}

cv::Mat cap\_frame = detection\_data.cap\_frame;

cv::Mat draw\_frame = detection\_data.cap\_frame.clone();

std::vector<bbox\_t> result\_vec = detection\_data.result\_vec;

#ifdef TRACK\_OPTFLOW

if (detection\_data.new\_detection) {

tracker\_flow.update\_tracking\_flow(detection\_data.cap\_frame, detection\_data.result\_vec);

while (track\_optflow\_queue.size() > 0) {

draw\_frame = track\_optflow\_queue.back();

result\_vec = tracker\_flow.tracking\_flow(track\_optflow\_queue.front(), false);

track\_optflow\_queue.pop();

}

}

else {

track\_optflow\_queue.push(cap\_frame);

result\_vec = tracker\_flow.tracking\_flow(cap\_frame, false);

}

detection\_data.new\_detection = true; // to correct kalman filter

#endif //TRACK\_OPTFLOW

// track ID by using kalman filter

if (use\_kalman\_filter) {

if (detection\_data.new\_detection) {

result\_vec = track\_kalman.correct(result\_vec);

}

else {

result\_vec = track\_kalman.predict();

}

}

// track ID by using custom function

else {

int frame\_story = std::max(5, current\_fps\_cap.load());

result\_vec = detector.tracking\_id(result\_vec, true, frame\_story, 40);

}

if (use\_zed\_camera && !detection\_data.zed\_cloud.empty()) {

result\_vec = get\_3d\_coordinates(result\_vec, detection\_data.zed\_cloud);

}

//small\_preview.set(draw\_frame, result\_vec);

//large\_preview.set(draw\_frame, result\_vec);

draw\_boxes(draw\_frame, result\_vec, obj\_names, current\_fps\_det, current\_fps\_cap);

//show\_console\_result(result\_vec, obj\_names, detection\_data.frame\_id);

//large\_preview.draw(draw\_frame);

//small\_preview.draw(draw\_frame, true);

detection\_data.result\_vec = result\_vec;

detection\_data.draw\_frame = draw\_frame;

draw2show.send(detection\_data);

if (send\_network) draw2net.send(detection\_data);

if (output\_video.isOpened()) draw2write.send(detection\_data);

} while (!detection\_data.exit\_flag);

std::cout << " t\_draw exit \n";

});

// write frame to videofile

t\_write = std::thread([&]()

{

if (output\_video.isOpened()) {

detection\_data\_t detection\_data;

cv::Mat output\_frame;

do {

detection\_data = draw2write.receive();

if(detection\_data.draw\_frame.channels() == 4) cv::cvtColor(detection\_data.draw\_frame, output\_frame, CV\_RGBA2RGB);

else output\_frame = detection\_data.draw\_frame;

output\_video << output\_frame;

} while (!detection\_data.exit\_flag);

output\_video.release();

}

std::cout << " t\_write exit \n";

});

// send detection to the network

t\_network = std::thread([&]()

{

if (send\_network) {

detection\_data\_t detection\_data;

do {

detection\_data = draw2net.receive();

detector.send\_json\_http(detection\_data.result\_vec, obj\_names, detection\_data.frame\_id, filename);

} while (!detection\_data.exit\_flag);

}

std::cout << " t\_network exit \n";

});

// show detection

detection\_data\_t detection\_data;

do {

steady\_end = std::chrono::steady\_clock::now();

float time\_sec = std::chrono::duration<double>(steady\_end - steady\_start).count();

if (time\_sec >= 1) {

current\_fps\_det = fps\_det\_counter.load() / time\_sec;

current\_fps\_cap = fps\_cap\_counter.load() / time\_sec;

steady\_start = steady\_end;

fps\_det\_counter = 0;

fps\_cap\_counter = 0;

}

detection\_data = draw2show.receive();

cv::Mat draw\_frame = detection\_data.draw\_frame;

//if (extrapolate\_flag) {

// cv::putText(draw\_frame, "extrapolate", cv::Point2f(10, 40), cv::FONT\_HERSHEY\_COMPLEX\_SMALL, 1.0, cv::Scalar(50, 50, 0), 2);

//}

cv::imshow("window name", draw\_frame);

int key = cv::waitKey(3); // 3 or 16ms

if (key == 'f') show\_small\_boxes = !show\_small\_boxes;

if (key == 'p') while (true) if (cv::waitKey(100) == 'p') break;

//if (key == 'e') extrapolate\_flag = !extrapolate\_flag;

if (key == 27) { exit\_flag = true;}

//std::cout << " current\_fps\_det = " << current\_fps\_det << ", current\_fps\_cap = " << current\_fps\_cap << std::endl;

} while (!detection\_data.exit\_flag);

std::cout << " show detection exit \n";

cv::destroyWindow("window name");

// wait for all threads

if (t\_cap.joinable()) t\_cap.join();

if (t\_prepare.joinable()) t\_prepare.join();

if (t\_detect.joinable()) t\_detect.join();

if (t\_post.joinable()) t\_post.join();

if (t\_draw.joinable()) t\_draw.join();

if (t\_write.joinable()) t\_write.join();

if (t\_network.joinable()) t\_network.join();

break;

}

else if (file\_ext == "txt") { // list of image files

std::ifstream file(filename);

if (!file.is\_open()) std::cout << "File not found! \n";

else

for (std::string line; file >> line;) {

std::cout << line << std::endl;

cv::Mat mat\_img = cv::imread(line);

std::vector<bbox\_t> result\_vec = detector.detect(mat\_img);

show\_console\_result(result\_vec, obj\_names);

//draw\_boxes(mat\_img, result\_vec, obj\_names);

//cv::imwrite("res\_" + line, mat\_img);

}

}

else { // image file

// to achive high performance for multiple images do these 2 lines in another thread

cv::Mat mat\_img = cv::imread(filename);

auto det\_image = detector.mat\_to\_image\_resize(mat\_img);

auto start = std::chrono::steady\_clock::now();

std::vector<bbox\_t> result\_vec = detector.detect\_resized(\*det\_image, mat\_img.size().width, mat\_img.size().height);

auto end = std::chrono::steady\_clock::now();

std::chrono::duration<double> spent = end - start;

std::cout << " Time: " << spent.count() << " sec \n";

//result\_vec = detector.tracking\_id(result\_vec); // comment it - if track\_id is not required

draw\_boxes(mat\_img, result\_vec, obj\_names);

cv::imshow("window name", mat\_img);

show\_console\_result(result\_vec, obj\_names);

cv::waitKey(0);

}

#else // OPENCV

//std::vector<bbox\_t> result\_vec = detector.detect(filename);

auto img = detector.load\_image(filename);

std::vector<bbox\_t> result\_vec = detector.detect(img);

detector.free\_image(img);

show\_console\_result(result\_vec, obj\_names);

#endif // OPENCV

}

catch (std::exception &e) { std::cerr << "exception: " << e.what() << "\n"; getchar(); }

catch (...) { std::cerr << "unknown exception \n"; getchar(); }

filename.clear();

}

return 0;

}