#ifndef YOLO\_V2\_CLASS\_HPP

#define YOLO\_V2\_CLASS\_HPP

#ifndef LIB\_API

#ifdef LIB\_EXPORTS

#if defined(\_MSC\_VER)

#define LIB\_API \_\_declspec(dllexport)

#else

#define LIB\_API \_\_attribute\_\_((visibility("default")))

#endif

#else

#if defined(\_MSC\_VER)

#define LIB\_API

#else

#define LIB\_API

#endif

#endif

#endif

#define C\_SHARP\_MAX\_OBJECTS 1000

struct bbox\_t {

unsigned int x, y, w, h; // (x,y) - top-left corner, (w, h) - width & height of bounded box

float prob; // confidence - probability that the object was found correctly

unsigned int obj\_id; // class of object - from range [0, classes-1]

unsigned int track\_id; // tracking id for video (0 - untracked, 1 - inf - tracked object)

unsigned int frames\_counter; // counter of frames on which the object was detected

float x\_3d, y\_3d, z\_3d; // center of object (in Meters) if ZED 3D Camera is used

};

struct image\_t {

int h; // height

int w; // width

int c; // number of chanels (3 - for RGB)

float \*data; // pointer to the image data

};

struct bbox\_t\_container {

bbox\_t candidates[C\_SHARP\_MAX\_OBJECTS];

};

#ifdef \_\_cplusplus

#include <memory>

#include <vector>

#include <deque>

#include <algorithm>

#include <chrono>

#include <string>

#include <sstream>

#include <iostream>

#include <cmath>

#ifdef OPENCV

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

#include <opencv2/highgui/highgui\_c.h> // C

#include <opencv2/imgproc/imgproc\_c.h> // C

#endif

extern "C" LIB\_API int init(const char \*configurationFilename, const char \*weightsFilename, int gpu);

extern "C" LIB\_API int detect\_image(const char \*filename, bbox\_t\_container &container);

extern "C" LIB\_API int detect\_mat(const uint8\_t\* data, const size\_t data\_length, bbox\_t\_container &container);

extern "C" LIB\_API int dispose();

extern "C" LIB\_API int get\_device\_count();

extern "C" LIB\_API int get\_device\_name(int gpu, char\* deviceName);

extern "C" LIB\_API bool built\_with\_cuda();

extern "C" LIB\_API bool built\_with\_cudnn();

extern "C" LIB\_API bool built\_with\_opencv();

extern "C" LIB\_API void send\_json\_custom(char const\* send\_buf, int port, int timeout);

class Detector {

std::shared\_ptr<void> detector\_gpu\_ptr;

std::deque<std::vector<bbox\_t>> prev\_bbox\_vec\_deque;

std::string \_cfg\_filename, \_weight\_filename;

public:

const int cur\_gpu\_id;

float nms = .4;

bool wait\_stream;

LIB\_API Detector(std::string cfg\_filename, std::string weight\_filename, int gpu\_id = 0);

LIB\_API ~Detector();

LIB\_API std::vector<bbox\_t> detect(std::string image\_filename, float thresh = 0.2, bool use\_mean = false);

LIB\_API std::vector<bbox\_t> detect(image\_t img, float thresh = 0.2, bool use\_mean = false);

static LIB\_API image\_t load\_image(std::string image\_filename);

static LIB\_API void free\_image(image\_t m);

LIB\_API int get\_net\_width() const;

LIB\_API int get\_net\_height() const;

LIB\_API int get\_net\_color\_depth() const;

LIB\_API std::vector<bbox\_t> tracking\_id(std::vector<bbox\_t> cur\_bbox\_vec, bool const change\_history = true,

int const frames\_story = 5, int const max\_dist = 40);

LIB\_API void \*get\_cuda\_context();

//LIB\_API bool send\_json\_http(std::vector<bbox\_t> cur\_bbox\_vec, std::vector<std::string> obj\_names, int frame\_id,

// std::string filename = std::string(), int timeout = 400000, int port = 8070);

std::vector<bbox\_t> detect\_resized(image\_t img, int init\_w, int init\_h, float thresh = 0.2, bool use\_mean = false)

{

if (img.data == NULL)

throw std::runtime\_error("Image is empty");

auto detection\_boxes = detect(img, thresh, use\_mean);

float wk = (float)init\_w / img.w, hk = (float)init\_h / img.h;

for (auto &i : detection\_boxes) i.x \*= wk, i.w \*= wk, i.y \*= hk, i.h \*= hk;

return detection\_boxes;

}

#ifdef OPENCV

std::vector<bbox\_t> detect(cv::Mat mat, float thresh = 0.2, bool use\_mean = false)

{

if(mat.data == NULL)

throw std::runtime\_error("Image is empty");

auto image\_ptr = mat\_to\_image\_resize(mat);

return detect\_resized(\*image\_ptr, mat.cols, mat.rows, thresh, use\_mean);

}

std::shared\_ptr<image\_t> mat\_to\_image\_resize(cv::Mat mat) const

{

if (mat.data == NULL) return std::shared\_ptr<image\_t>(NULL);

cv::Size network\_size = cv::Size(get\_net\_width(), get\_net\_height());

cv::Mat det\_mat;

if (mat.size() != network\_size)

cv::resize(mat, det\_mat, network\_size);

else

det\_mat = mat; // only reference is copied

return mat\_to\_image(det\_mat);

}

static std::shared\_ptr<image\_t> mat\_to\_image(cv::Mat img\_src)

{

cv::Mat img;

if (img\_src.channels() == 4) cv::cvtColor(img\_src, img, cv::COLOR\_RGBA2BGR);

else if (img\_src.channels() == 3) cv::cvtColor(img\_src, img, cv::COLOR\_RGB2BGR);

else if (img\_src.channels() == 1) cv::cvtColor(img\_src, img, cv::COLOR\_GRAY2BGR);

else std::cerr << " Warning: img\_src.channels() is not 1, 3 or 4. It is = " << img\_src.channels() << std::endl;

std::shared\_ptr<image\_t> image\_ptr(new image\_t, [](image\_t \*img) { free\_image(\*img); delete img; });

\*image\_ptr = mat\_to\_image\_custom(img);

return image\_ptr;

}

private:

static image\_t mat\_to\_image\_custom(cv::Mat mat)

{

int w = mat.cols;

int h = mat.rows;

int c = mat.channels();

image\_t im = make\_image\_custom(w, h, c);

unsigned char \*data = (unsigned char \*)mat.data;

int step = mat.step;

for (int y = 0; y < h; ++y) {

for (int k = 0; k < c; ++k) {

for (int x = 0; x < w; ++x) {

im.data[k\*w\*h + y\*w + x] = data[y\*step + x\*c + k] / 255.0f;

}

}

}

return im;

}

static image\_t make\_empty\_image(int w, int h, int c)

{

image\_t out;

out.data = 0;

out.h = h;

out.w = w;

out.c = c;

return out;

}

static image\_t make\_image\_custom(int w, int h, int c)

{

image\_t out = make\_empty\_image(w, h, c);

out.data = (float \*)calloc(h\*w\*c, sizeof(float));

return out;

}

#endif // OPENCV

public:

bool send\_json\_http(std::vector<bbox\_t> cur\_bbox\_vec, std::vector<std::string> obj\_names, int frame\_id,

std::string filename = std::string(), int timeout = 400000, int port = 8070)

{

std::string send\_str;

char \*tmp\_buf = (char \*)calloc(1024, sizeof(char));

if (!filename.empty()) {

sprintf(tmp\_buf, "{\n \"frame\_id\":%d, \n \"filename\":\"%s\", \n \"objects\": [ \n", frame\_id, filename.c\_str());

}

else {

sprintf(tmp\_buf, "{\n \"frame\_id\":%d, \n \"objects\": [ \n", frame\_id);

}

send\_str = tmp\_buf;

free(tmp\_buf);

for (auto & i : cur\_bbox\_vec) {

char \*buf = (char \*)calloc(2048, sizeof(char));

sprintf(buf, " {\"class\_id\":%d, \"name\":\"%s\", \"absolute\_coordinates\":{\"center\_x\":%d, \"center\_y\":%d, \"width\":%d, \"height\":%d}, \"confidence\":%f",

i.obj\_id, obj\_names[i.obj\_id].c\_str(), i.x, i.y, i.w, i.h, i.prob);

//sprintf(buf, " {\"class\_id\":%d, \"name\":\"%s\", \"relative\_coordinates\":{\"center\_x\":%f, \"center\_y\":%f, \"width\":%f, \"height\":%f}, \"confidence\":%f",

// i.obj\_id, obj\_names[i.obj\_id], i.x, i.y, i.w, i.h, i.prob);

send\_str += buf;

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

sprintf(buf, "\n , \"coordinates\_in\_meters\":{\"x\_3d\":%.2f, \"y\_3d\":%.2f, \"z\_3d\":%.2f}",

i.x\_3d, i.y\_3d, i.z\_3d);

send\_str += buf;

}

send\_str += "}\n";

free(buf);

}

//send\_str += "\n ] \n}, \n";

send\_str += "\n ] \n}";

send\_json\_custom(send\_str.c\_str(), port, timeout);

return true;

}

};

// --------------------------------------------------------------------------------

#if defined(TRACK\_OPTFLOW) && defined(OPENCV) && defined(GPU)

#include <opencv2/cudaoptflow.hpp>

#include <opencv2/cudaimgproc.hpp>

#include <opencv2/cudaarithm.hpp>

#include <opencv2/core/cuda.hpp>

class Tracker\_optflow {

public:

const int gpu\_count;

const int gpu\_id;

const int flow\_error;

Tracker\_optflow(int \_gpu\_id = 0, int win\_size = 15, int max\_level = 3, int iterations = 8000, int \_flow\_error = -1) :

gpu\_count(cv::cuda::getCudaEnabledDeviceCount()), gpu\_id(std::min(\_gpu\_id, gpu\_count-1)),

flow\_error((\_flow\_error > 0)? \_flow\_error:(win\_size\*4))

{

int const old\_gpu\_id = cv::cuda::getDevice();

cv::cuda::setDevice(gpu\_id);

stream = cv::cuda::Stream();

sync\_PyrLKOpticalFlow\_gpu = cv::cuda::SparsePyrLKOpticalFlow::create();

sync\_PyrLKOpticalFlow\_gpu->setWinSize(cv::Size(win\_size, win\_size)); // 9, 15, 21, 31

sync\_PyrLKOpticalFlow\_gpu->setMaxLevel(max\_level); // +- 3 pt

sync\_PyrLKOpticalFlow\_gpu->setNumIters(iterations); // 2000, def: 30

cv::cuda::setDevice(old\_gpu\_id);

}

// just to avoid extra allocations

cv::cuda::GpuMat src\_mat\_gpu;

cv::cuda::GpuMat dst\_mat\_gpu, dst\_grey\_gpu;

cv::cuda::GpuMat prev\_pts\_flow\_gpu, cur\_pts\_flow\_gpu;

cv::cuda::GpuMat status\_gpu, err\_gpu;

cv::cuda::GpuMat src\_grey\_gpu; // used in both functions

cv::Ptr<cv::cuda::SparsePyrLKOpticalFlow> sync\_PyrLKOpticalFlow\_gpu;

cv::cuda::Stream stream;

std::vector<bbox\_t> cur\_bbox\_vec;

std::vector<bool> good\_bbox\_vec\_flags;

cv::Mat prev\_pts\_flow\_cpu;

void update\_cur\_bbox\_vec(std::vector<bbox\_t> \_cur\_bbox\_vec)

{

cur\_bbox\_vec = \_cur\_bbox\_vec;

good\_bbox\_vec\_flags = std::vector<bool>(cur\_bbox\_vec.size(), true);

cv::Mat prev\_pts, cur\_pts\_flow\_cpu;

for (auto &i : cur\_bbox\_vec) {

float x\_center = (i.x + i.w / 2.0F);

float y\_center = (i.y + i.h / 2.0F);

prev\_pts.push\_back(cv::Point2f(x\_center, y\_center));

}

if (prev\_pts.rows == 0)

prev\_pts\_flow\_cpu = cv::Mat();

else

cv::transpose(prev\_pts, prev\_pts\_flow\_cpu);

if (prev\_pts\_flow\_gpu.cols < prev\_pts\_flow\_cpu.cols) {

prev\_pts\_flow\_gpu = cv::cuda::GpuMat(prev\_pts\_flow\_cpu.size(), prev\_pts\_flow\_cpu.type());

cur\_pts\_flow\_gpu = cv::cuda::GpuMat(prev\_pts\_flow\_cpu.size(), prev\_pts\_flow\_cpu.type());

status\_gpu = cv::cuda::GpuMat(prev\_pts\_flow\_cpu.size(), CV\_8UC1);

err\_gpu = cv::cuda::GpuMat(prev\_pts\_flow\_cpu.size(), CV\_32FC1);

}

prev\_pts\_flow\_gpu.upload(cv::Mat(prev\_pts\_flow\_cpu), stream);

}

void update\_tracking\_flow(cv::Mat src\_mat, std::vector<bbox\_t> \_cur\_bbox\_vec)

{

int const old\_gpu\_id = cv::cuda::getDevice();

if (old\_gpu\_id != gpu\_id)

cv::cuda::setDevice(gpu\_id);

if (src\_mat.channels() == 1 || src\_mat.channels() == 3 || src\_mat.channels() == 4) {

if (src\_mat\_gpu.cols == 0) {

src\_mat\_gpu = cv::cuda::GpuMat(src\_mat.size(), src\_mat.type());

src\_grey\_gpu = cv::cuda::GpuMat(src\_mat.size(), CV\_8UC1);

}

if (src\_mat.channels() == 1) {

src\_mat\_gpu.upload(src\_mat, stream);

src\_mat\_gpu.copyTo(src\_grey\_gpu);

}

else if (src\_mat.channels() == 3) {

src\_mat\_gpu.upload(src\_mat, stream);

cv::cuda::cvtColor(src\_mat\_gpu, src\_grey\_gpu, CV\_BGR2GRAY, 1, stream);

}

else if (src\_mat.channels() == 4) {

src\_mat\_gpu.upload(src\_mat, stream);

cv::cuda::cvtColor(src\_mat\_gpu, src\_grey\_gpu, CV\_BGRA2GRAY, 1, stream);

}

else {

std::cerr << " Warning: src\_mat.channels() is not: 1, 3 or 4. It is = " << src\_mat.channels() << " \n";

return;

}

}

update\_cur\_bbox\_vec(\_cur\_bbox\_vec);

if (old\_gpu\_id != gpu\_id)

cv::cuda::setDevice(old\_gpu\_id);

}

std::vector<bbox\_t> tracking\_flow(cv::Mat dst\_mat, bool check\_error = true)

{

if (sync\_PyrLKOpticalFlow\_gpu.empty()) {

std::cout << "sync\_PyrLKOpticalFlow\_gpu isn't initialized \n";

return cur\_bbox\_vec;

}

int const old\_gpu\_id = cv::cuda::getDevice();

if(old\_gpu\_id != gpu\_id)

cv::cuda::setDevice(gpu\_id);

if (dst\_mat\_gpu.cols == 0) {

dst\_mat\_gpu = cv::cuda::GpuMat(dst\_mat.size(), dst\_mat.type());

dst\_grey\_gpu = cv::cuda::GpuMat(dst\_mat.size(), CV\_8UC1);

}

//dst\_grey\_gpu.upload(dst\_mat, stream); // use BGR

dst\_mat\_gpu.upload(dst\_mat, stream);

cv::cuda::cvtColor(dst\_mat\_gpu, dst\_grey\_gpu, CV\_BGR2GRAY, 1, stream);

if (src\_grey\_gpu.rows != dst\_grey\_gpu.rows || src\_grey\_gpu.cols != dst\_grey\_gpu.cols) {

stream.waitForCompletion();

src\_grey\_gpu = dst\_grey\_gpu.clone();

cv::cuda::setDevice(old\_gpu\_id);

return cur\_bbox\_vec;

}

////sync\_PyrLKOpticalFlow\_gpu.sparse(src\_grey\_gpu, dst\_grey\_gpu, prev\_pts\_flow\_gpu, cur\_pts\_flow\_gpu, status\_gpu, &err\_gpu); // OpenCV 2.4.x

sync\_PyrLKOpticalFlow\_gpu->calc(src\_grey\_gpu, dst\_grey\_gpu, prev\_pts\_flow\_gpu, cur\_pts\_flow\_gpu, status\_gpu, err\_gpu, stream); // OpenCV 3.x

cv::Mat cur\_pts\_flow\_cpu;

cur\_pts\_flow\_gpu.download(cur\_pts\_flow\_cpu, stream);

dst\_grey\_gpu.copyTo(src\_grey\_gpu, stream);

cv::Mat err\_cpu, status\_cpu;

err\_gpu.download(err\_cpu, stream);

status\_gpu.download(status\_cpu, stream);

stream.waitForCompletion();

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

if (err\_cpu.cols == cur\_bbox\_vec.size() && status\_cpu.cols == cur\_bbox\_vec.size())

{

for (size\_t i = 0; i < cur\_bbox\_vec.size(); ++i)

{

cv::Point2f cur\_key\_pt = cur\_pts\_flow\_cpu.at<cv::Point2f>(0, i);

cv::Point2f prev\_key\_pt = prev\_pts\_flow\_cpu.at<cv::Point2f>(0, i);

float moved\_x = cur\_key\_pt.x - prev\_key\_pt.x;

float moved\_y = cur\_key\_pt.y - prev\_key\_pt.y;

if (abs(moved\_x) < 100 && abs(moved\_y) < 100 && good\_bbox\_vec\_flags[i])

if (err\_cpu.at<float>(0, i) < flow\_error && status\_cpu.at<unsigned char>(0, i) != 0 &&

((float)cur\_bbox\_vec[i].x + moved\_x) > 0 && ((float)cur\_bbox\_vec[i].y + moved\_y) > 0)

{

cur\_bbox\_vec[i].x += moved\_x + 0.5;

cur\_bbox\_vec[i].y += moved\_y + 0.5;

result\_bbox\_vec.push\_back(cur\_bbox\_vec[i]);

}

else good\_bbox\_vec\_flags[i] = false;

else good\_bbox\_vec\_flags[i] = false;

//if(!check\_error && !good\_bbox\_vec\_flags[i]) result\_bbox\_vec.push\_back(cur\_bbox\_vec[i]);

}

}

cur\_pts\_flow\_gpu.swap(prev\_pts\_flow\_gpu);

cur\_pts\_flow\_cpu.copyTo(prev\_pts\_flow\_cpu);

if (old\_gpu\_id != gpu\_id)

cv::cuda::setDevice(old\_gpu\_id);

return result\_bbox\_vec;

}

};

#elif defined(TRACK\_OPTFLOW) && defined(OPENCV)

//#include <opencv2/optflow.hpp>

#include <opencv2/video/tracking.hpp>

class Tracker\_optflow {

public:

const int flow\_error;

Tracker\_optflow(int win\_size = 15, int max\_level = 3, int iterations = 8000, int \_flow\_error = -1) :

flow\_error((\_flow\_error > 0)? \_flow\_error:(win\_size\*4))

{

sync\_PyrLKOpticalFlow = cv::SparsePyrLKOpticalFlow::create();

sync\_PyrLKOpticalFlow->setWinSize(cv::Size(win\_size, win\_size)); // 9, 15, 21, 31

sync\_PyrLKOpticalFlow->setMaxLevel(max\_level); // +- 3 pt

}

// just to avoid extra allocations

cv::Mat dst\_grey;

cv::Mat prev\_pts\_flow, cur\_pts\_flow;

cv::Mat status, err;

cv::Mat src\_grey; // used in both functions

cv::Ptr<cv::SparsePyrLKOpticalFlow> sync\_PyrLKOpticalFlow;

std::vector<bbox\_t> cur\_bbox\_vec;

std::vector<bool> good\_bbox\_vec\_flags;

void update\_cur\_bbox\_vec(std::vector<bbox\_t> \_cur\_bbox\_vec)

{

cur\_bbox\_vec = \_cur\_bbox\_vec;

good\_bbox\_vec\_flags = std::vector<bool>(cur\_bbox\_vec.size(), true);

cv::Mat prev\_pts, cur\_pts\_flow;

for (auto &i : cur\_bbox\_vec) {

float x\_center = (i.x + i.w / 2.0F);

float y\_center = (i.y + i.h / 2.0F);

prev\_pts.push\_back(cv::Point2f(x\_center, y\_center));

}

if (prev\_pts.rows == 0)

prev\_pts\_flow = cv::Mat();

else

cv::transpose(prev\_pts, prev\_pts\_flow);

}

void update\_tracking\_flow(cv::Mat new\_src\_mat, std::vector<bbox\_t> \_cur\_bbox\_vec)

{

if (new\_src\_mat.channels() == 1) {

src\_grey = new\_src\_mat.clone();

}

else if (new\_src\_mat.channels() == 3) {

cv::cvtColor(new\_src\_mat, src\_grey, CV\_BGR2GRAY, 1);

}

else if (new\_src\_mat.channels() == 4) {

cv::cvtColor(new\_src\_mat, src\_grey, CV\_BGRA2GRAY, 1);

}

else {

std::cerr << " Warning: new\_src\_mat.channels() is not: 1, 3 or 4. It is = " << new\_src\_mat.channels() << " \n";

return;

}

update\_cur\_bbox\_vec(\_cur\_bbox\_vec);

}

std::vector<bbox\_t> tracking\_flow(cv::Mat new\_dst\_mat, bool check\_error = true)

{

if (sync\_PyrLKOpticalFlow.empty()) {

std::cout << "sync\_PyrLKOpticalFlow isn't initialized \n";

return cur\_bbox\_vec;

}

cv::cvtColor(new\_dst\_mat, dst\_grey, CV\_BGR2GRAY, 1);

if (src\_grey.rows != dst\_grey.rows || src\_grey.cols != dst\_grey.cols) {

src\_grey = dst\_grey.clone();

//std::cerr << " Warning: src\_grey.rows != dst\_grey.rows || src\_grey.cols != dst\_grey.cols \n";

return cur\_bbox\_vec;

}

if (prev\_pts\_flow.cols < 1) {

return cur\_bbox\_vec;

}

////sync\_PyrLKOpticalFlow\_gpu.sparse(src\_grey\_gpu, dst\_grey\_gpu, prev\_pts\_flow\_gpu, cur\_pts\_flow\_gpu, status\_gpu, &err\_gpu); // OpenCV 2.4.x

sync\_PyrLKOpticalFlow->calc(src\_grey, dst\_grey, prev\_pts\_flow, cur\_pts\_flow, status, err); // OpenCV 3.x

dst\_grey.copyTo(src\_grey);

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

if (err.rows == cur\_bbox\_vec.size() && status.rows == cur\_bbox\_vec.size())

{

for (size\_t i = 0; i < cur\_bbox\_vec.size(); ++i)

{

cv::Point2f cur\_key\_pt = cur\_pts\_flow.at<cv::Point2f>(0, i);

cv::Point2f prev\_key\_pt = prev\_pts\_flow.at<cv::Point2f>(0, i);

float moved\_x = cur\_key\_pt.x - prev\_key\_pt.x;

float moved\_y = cur\_key\_pt.y - prev\_key\_pt.y;

if (abs(moved\_x) < 100 && abs(moved\_y) < 100 && good\_bbox\_vec\_flags[i])

if (err.at<float>(0, i) < flow\_error && status.at<unsigned char>(0, i) != 0 &&

((float)cur\_bbox\_vec[i].x + moved\_x) > 0 && ((float)cur\_bbox\_vec[i].y + moved\_y) > 0)

{

cur\_bbox\_vec[i].x += moved\_x + 0.5;

cur\_bbox\_vec[i].y += moved\_y + 0.5;

result\_bbox\_vec.push\_back(cur\_bbox\_vec[i]);

}

else good\_bbox\_vec\_flags[i] = false;

else good\_bbox\_vec\_flags[i] = false;

//if(!check\_error && !good\_bbox\_vec\_flags[i]) result\_bbox\_vec.push\_back(cur\_bbox\_vec[i]);

}

}

prev\_pts\_flow = cur\_pts\_flow.clone();

return result\_bbox\_vec;

}

};

#else

class Tracker\_optflow {};

#endif // defined(TRACK\_OPTFLOW) && defined(OPENCV)

#ifdef OPENCV

static cv::Scalar obj\_id\_to\_color(int obj\_id) {

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

int const offset = obj\_id \* 123457 % 6;

int const color\_scale = 150 + (obj\_id \* 123457) % 100;

cv::Scalar color(colors[offset][0], colors[offset][1], colors[offset][2]);

color \*= color\_scale;

return color;

}

class preview\_boxes\_t {

enum { frames\_history = 30 }; // how long to keep the history saved

struct preview\_box\_track\_t {

unsigned int track\_id, obj\_id, last\_showed\_frames\_ago;

bool current\_detection;

bbox\_t bbox;

cv::Mat mat\_obj, mat\_resized\_obj;

preview\_box\_track\_t() : track\_id(0), obj\_id(0), last\_showed\_frames\_ago(frames\_history), current\_detection(false) {}

};

std::vector<preview\_box\_track\_t> preview\_box\_track\_id;

size\_t const preview\_box\_size, bottom\_offset;

bool const one\_off\_detections;

public:

preview\_boxes\_t(size\_t \_preview\_box\_size = 100, size\_t \_bottom\_offset = 100, bool \_one\_off\_detections = false) :

preview\_box\_size(\_preview\_box\_size), bottom\_offset(\_bottom\_offset), one\_off\_detections(\_one\_off\_detections)

{}

void set(cv::Mat src\_mat, std::vector<bbox\_t> result\_vec)

{

size\_t const count\_preview\_boxes = src\_mat.cols / preview\_box\_size;

if (preview\_box\_track\_id.size() != count\_preview\_boxes) preview\_box\_track\_id.resize(count\_preview\_boxes);

// increment frames history

for (auto &i : preview\_box\_track\_id)

i.last\_showed\_frames\_ago = std::min((unsigned)frames\_history, i.last\_showed\_frames\_ago + 1);

// occupy empty boxes

for (auto &k : result\_vec) {

bool found = false;

// find the same (track\_id)

for (auto &i : preview\_box\_track\_id) {

if (i.track\_id == k.track\_id) {

if (!one\_off\_detections) i.last\_showed\_frames\_ago = 0; // for tracked objects

found = true;

break;

}

}

if (!found) {

// find empty box

for (auto &i : preview\_box\_track\_id) {

if (i.last\_showed\_frames\_ago == frames\_history) {

if (!one\_off\_detections && k.frames\_counter == 0) break; // don't show if obj isn't tracked yet

i.track\_id = k.track\_id;

i.obj\_id = k.obj\_id;

i.bbox = k;

i.last\_showed\_frames\_ago = 0;

break;

}

}

}

}

// draw preview box (from old or current frame)

for (size\_t i = 0; i < preview\_box\_track\_id.size(); ++i)

{

// get object image

cv::Mat dst = preview\_box\_track\_id[i].mat\_resized\_obj;

preview\_box\_track\_id[i].current\_detection = false;

for (auto &k : result\_vec) {

if (preview\_box\_track\_id[i].track\_id == k.track\_id) {

if (one\_off\_detections && preview\_box\_track\_id[i].last\_showed\_frames\_ago > 0) {

preview\_box\_track\_id[i].last\_showed\_frames\_ago = frames\_history; break;

}

bbox\_t b = k;

cv::Rect r(b.x, b.y, b.w, b.h);

cv::Rect img\_rect(cv::Point2i(0, 0), src\_mat.size());

cv::Rect rect\_roi = r & img\_rect;

if (rect\_roi.width > 1 || rect\_roi.height > 1) {

cv::Mat roi = src\_mat(rect\_roi);

cv::resize(roi, dst, cv::Size(preview\_box\_size, preview\_box\_size), cv::INTER\_NEAREST);

preview\_box\_track\_id[i].mat\_obj = roi.clone();

preview\_box\_track\_id[i].mat\_resized\_obj = dst.clone();

preview\_box\_track\_id[i].current\_detection = true;

preview\_box\_track\_id[i].bbox = k;

}

break;

}

}

}

}

void draw(cv::Mat draw\_mat, bool show\_small\_boxes = false)

{

// draw preview box (from old or current frame)

for (size\_t i = 0; i < preview\_box\_track\_id.size(); ++i)

{

auto &prev\_box = preview\_box\_track\_id[i];

// draw object image

cv::Mat dst = prev\_box.mat\_resized\_obj;

if (prev\_box.last\_showed\_frames\_ago < frames\_history &&

dst.size() == cv::Size(preview\_box\_size, preview\_box\_size))

{

cv::Rect dst\_rect\_roi(cv::Point2i(i \* preview\_box\_size, draw\_mat.rows - bottom\_offset), dst.size());

cv::Mat dst\_roi = draw\_mat(dst\_rect\_roi);

dst.copyTo(dst\_roi);

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

int thickness = (prev\_box.current\_detection) ? 5 : 1;

cv::rectangle(draw\_mat, dst\_rect\_roi, color, thickness);

unsigned int const track\_id = prev\_box.track\_id;

std::string track\_id\_str = (track\_id > 0) ? std::to\_string(track\_id) : "";

putText(draw\_mat, track\_id\_str, dst\_rect\_roi.tl() - cv::Point2i(-4, 5), cv::FONT\_HERSHEY\_COMPLEX\_SMALL, 0.9, cv::Scalar(0, 0, 0), 2);

std::string size\_str = std::to\_string(prev\_box.bbox.w) + "x" + std::to\_string(prev\_box.bbox.h);

putText(draw\_mat, size\_str, dst\_rect\_roi.tl() + cv::Point2i(0, 12), cv::FONT\_HERSHEY\_COMPLEX\_SMALL, 0.8, cv::Scalar(0, 0, 0), 1);

if (!one\_off\_detections && prev\_box.current\_detection) {

cv::line(draw\_mat, dst\_rect\_roi.tl() + cv::Point2i(preview\_box\_size, 0),

cv::Point2i(prev\_box.bbox.x, prev\_box.bbox.y + prev\_box.bbox.h),

color);

}

if (one\_off\_detections && show\_small\_boxes) {

cv::Rect src\_rect\_roi(cv::Point2i(prev\_box.bbox.x, prev\_box.bbox.y),

cv::Size(prev\_box.bbox.w, prev\_box.bbox.h));

unsigned int const color\_history = (255 \* prev\_box.last\_showed\_frames\_ago) / frames\_history;

color = cv::Scalar(255 - 3 \* color\_history, 255 - 2 \* color\_history, 255 - 1 \* color\_history);

if (prev\_box.mat\_obj.size() == src\_rect\_roi.size()) {

prev\_box.mat\_obj.copyTo(draw\_mat(src\_rect\_roi));

}

cv::rectangle(draw\_mat, src\_rect\_roi, color, thickness);

putText(draw\_mat, track\_id\_str, src\_rect\_roi.tl() - cv::Point2i(0, 10), cv::FONT\_HERSHEY\_COMPLEX\_SMALL, 0.8, cv::Scalar(0, 0, 0), 1);

}

}

}

}

};

class track\_kalman\_t

{

int track\_id\_counter;

std::chrono::steady\_clock::time\_point global\_last\_time;

float dT;

public:

int max\_objects; // max objects for tracking

int min\_frames; // min frames to consider an object as detected

const float max\_dist; // max distance (in px) to track with the same ID

cv::Size img\_size; // max value of x,y,w,h

struct tst\_t {

int track\_id;

int state\_id;

std::chrono::steady\_clock::time\_point last\_time;

int detection\_count;

tst\_t() : track\_id(-1), state\_id(-1) {}

};

std::vector<tst\_t> track\_id\_state\_id\_time;

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

struct one\_kalman\_t;

std::vector<one\_kalman\_t> kalman\_vec;

struct one\_kalman\_t

{

cv::KalmanFilter kf;

cv::Mat state;

cv::Mat meas;

int measSize, stateSize, contrSize;

void set\_delta\_time(float dT) {

kf.transitionMatrix.at<float>(2) = dT;

kf.transitionMatrix.at<float>(9) = dT;

}

void set(bbox\_t box)

{

initialize\_kalman();

kf.errorCovPre.at<float>(0) = 1; // px

kf.errorCovPre.at<float>(7) = 1; // px

kf.errorCovPre.at<float>(14) = 1;

kf.errorCovPre.at<float>(21) = 1;

kf.errorCovPre.at<float>(28) = 1; // px

kf.errorCovPre.at<float>(35) = 1; // px

state.at<float>(0) = box.x;

state.at<float>(1) = box.y;

state.at<float>(2) = 0;

state.at<float>(3) = 0;

state.at<float>(4) = box.w;

state.at<float>(5) = box.h;

// <<<< Initialization

kf.statePost = state;

}

// Kalman.correct() calculates: statePost = statePre + gain \* (z(k)-measurementMatrix\*statePre);

// corrected state (x(k)): x(k)=x'(k)+K(k)\*(z(k)-H\*x'(k))

void correct(bbox\_t box) {

meas.at<float>(0) = box.x;

meas.at<float>(1) = box.y;

meas.at<float>(2) = box.w;

meas.at<float>(3) = box.h;

kf.correct(meas);

bbox\_t new\_box = predict();

if (new\_box.w == 0 || new\_box.h == 0) {

set(box);

//std::cerr << " force set(): track\_id = " << box.track\_id <<

// ", x = " << box.x << ", y = " << box.y << ", w = " << box.w << ", h = " << box.h << std::endl;

}

}

// Kalman.predict() calculates: statePre = TransitionMatrix \* statePost;

// predicted state (x'(k)): x(k)=A\*x(k-1)+B\*u(k)

bbox\_t predict() {

bbox\_t box;

state = kf.predict();

box.x = state.at<float>(0);

box.y = state.at<float>(1);

box.w = state.at<float>(4);

box.h = state.at<float>(5);

return box;

}

void initialize\_kalman()

{

kf = cv::KalmanFilter(stateSize, measSize, contrSize, CV\_32F);

// Transition State Matrix A

// Note: set dT at each processing step!

// [ 1 0 dT 0 0 0 ]

// [ 0 1 0 dT 0 0 ]

// [ 0 0 1 0 0 0 ]

// [ 0 0 0 1 0 0 ]

// [ 0 0 0 0 1 0 ]

// [ 0 0 0 0 0 1 ]

cv::setIdentity(kf.transitionMatrix);

// Measure Matrix H

// [ 1 0 0 0 0 0 ]

// [ 0 1 0 0 0 0 ]

// [ 0 0 0 0 1 0 ]

// [ 0 0 0 0 0 1 ]

kf.measurementMatrix = cv::Mat::zeros(measSize, stateSize, CV\_32F);

kf.measurementMatrix.at<float>(0) = 1.0f;

kf.measurementMatrix.at<float>(7) = 1.0f;

kf.measurementMatrix.at<float>(16) = 1.0f;

kf.measurementMatrix.at<float>(23) = 1.0f;

// Process Noise Covariance Matrix Q - result smoother with lower values (1e-2)

// [ Ex 0 0 0 0 0 ]

// [ 0 Ey 0 0 0 0 ]

// [ 0 0 Ev\_x 0 0 0 ]

// [ 0 0 0 Ev\_y 0 0 ]

// [ 0 0 0 0 Ew 0 ]

// [ 0 0 0 0 0 Eh ]

//cv::setIdentity(kf.processNoiseCov, cv::Scalar(1e-3));

kf.processNoiseCov.at<float>(0) = 1e-2;

kf.processNoiseCov.at<float>(7) = 1e-2;

kf.processNoiseCov.at<float>(14) = 1e-2;// 5.0f;

kf.processNoiseCov.at<float>(21) = 1e-2;// 5.0f;

kf.processNoiseCov.at<float>(28) = 5e-3;

kf.processNoiseCov.at<float>(35) = 5e-3;

// Measures Noise Covariance Matrix R - result smoother with higher values (1e-1)

cv::setIdentity(kf.measurementNoiseCov, cv::Scalar(1e-1));

//cv::setIdentity(kf.errorCovPost, cv::Scalar::all(1e-2));

// <<<< Kalman Filter

set\_delta\_time(0);

}

one\_kalman\_t(int \_stateSize = 6, int \_measSize = 4, int \_contrSize = 0) :

kf(\_stateSize, \_measSize, \_contrSize, CV\_32F), measSize(\_measSize), stateSize(\_stateSize), contrSize(\_contrSize)

{

state = cv::Mat(stateSize, 1, CV\_32F); // [x,y,v\_x,v\_y,w,h]

meas = cv::Mat(measSize, 1, CV\_32F); // [z\_x,z\_y,z\_w,z\_h]

//cv::Mat procNoise(stateSize, 1, type)

// [E\_x,E\_y,E\_v\_x,E\_v\_y,E\_w,E\_h]

initialize\_kalman();

}

};

// ------------------------------------------

track\_kalman\_t(int \_max\_objects = 1000, int \_min\_frames = 3, float \_max\_dist = 40, cv::Size \_img\_size = cv::Size(10000, 10000)) :

max\_objects(\_max\_objects), min\_frames(\_min\_frames), max\_dist(\_max\_dist), img\_size(\_img\_size),

track\_id\_counter(0)

{

kalman\_vec.resize(max\_objects);

track\_id\_state\_id\_time.resize(max\_objects);

result\_vec\_pred.resize(max\_objects);

}

float calc\_dt() {

dT = std::chrono::duration<double>(std::chrono::steady\_clock::now() - global\_last\_time).count();

return dT;

}

static float get\_distance(float src\_x, float src\_y, float dst\_x, float dst\_y) {

return sqrtf((src\_x - dst\_x)\*(src\_x - dst\_x) + (src\_y - dst\_y)\*(src\_y - dst\_y));

}

void clear\_old\_states() {

// clear old bboxes

for (size\_t state\_id = 0; state\_id < track\_id\_state\_id\_time.size(); ++state\_id)

{

float time\_sec = std::chrono::duration<double>(std::chrono::steady\_clock::now() - track\_id\_state\_id\_time[state\_id].last\_time).count();

float time\_wait = 0.5; // 0.5 second

if (track\_id\_state\_id\_time[state\_id].track\_id > -1)

{

if ((result\_vec\_pred[state\_id].x > img\_size.width) ||

(result\_vec\_pred[state\_id].y > img\_size.height))

{

track\_id\_state\_id\_time[state\_id].track\_id = -1;

}

if (time\_sec >= time\_wait || track\_id\_state\_id\_time[state\_id].detection\_count < 0) {

//std::cerr << " remove track\_id = " << track\_id\_state\_id\_time[state\_id].track\_id << ", state\_id = " << state\_id << std::endl;

track\_id\_state\_id\_time[state\_id].track\_id = -1; // remove bbox

}

}

}

}

tst\_t get\_state\_id(bbox\_t find\_box, std::vector<bool> &busy\_vec)

{

tst\_t tst;

tst.state\_id = -1;

float min\_dist = std::numeric\_limits<float>::max();

for (size\_t i = 0; i < max\_objects; ++i)

{

if (track\_id\_state\_id\_time[i].track\_id > -1 && result\_vec\_pred[i].obj\_id == find\_box.obj\_id && busy\_vec[i] == false)

{

bbox\_t pred\_box = result\_vec\_pred[i];

float dist = get\_distance(pred\_box.x, pred\_box.y, find\_box.x, find\_box.y);

float movement\_dist = std::max(max\_dist, static\_cast<float>(std::max(pred\_box.w, pred\_box.h)) );

if ((dist < movement\_dist) && (dist < min\_dist)) {

min\_dist = dist;

tst.state\_id = i;

}

}

}

if (tst.state\_id > -1) {

track\_id\_state\_id\_time[tst.state\_id].last\_time = std::chrono::steady\_clock::now();

track\_id\_state\_id\_time[tst.state\_id].detection\_count = std::max(track\_id\_state\_id\_time[tst.state\_id].detection\_count + 2, 10);

tst = track\_id\_state\_id\_time[tst.state\_id];

busy\_vec[tst.state\_id] = true;

}

else {

//std::cerr << " Didn't find: obj\_id = " << find\_box.obj\_id << ", x = " << find\_box.x << ", y = " << find\_box.y <<

// ", track\_id\_counter = " << track\_id\_counter << std::endl;

}

return tst;

}

tst\_t new\_state\_id(std::vector<bool> &busy\_vec)

{

tst\_t tst;

// find empty cell to add new track\_id

auto it = std::find\_if(track\_id\_state\_id\_time.begin(), track\_id\_state\_id\_time.end(), [&](tst\_t &v) { return v.track\_id == -1; });

if (it != track\_id\_state\_id\_time.end()) {

it->state\_id = it - track\_id\_state\_id\_time.begin();

//it->track\_id = track\_id\_counter++;

it->track\_id = 0;

it->last\_time = std::chrono::steady\_clock::now();

it->detection\_count = 1;

tst = \*it;

busy\_vec[it->state\_id] = true;

}

return tst;

}

std::vector<tst\_t> find\_state\_ids(std::vector<bbox\_t> result\_vec)

{

std::vector<tst\_t> tst\_vec(result\_vec.size());

std::vector<bool> busy\_vec(max\_objects, false);

for (size\_t i = 0; i < result\_vec.size(); ++i)

{

tst\_t tst = get\_state\_id(result\_vec[i], busy\_vec);

int state\_id = tst.state\_id;

int track\_id = tst.track\_id;

// if new state\_id

if (state\_id < 0) {

tst = new\_state\_id(busy\_vec);

state\_id = tst.state\_id;

track\_id = tst.track\_id;

if (state\_id > -1) {

kalman\_vec[state\_id].set(result\_vec[i]);

//std::cerr << " post: ";

}

}

//std::cerr << " track\_id = " << track\_id << ", state\_id = " << state\_id <<

// ", x = " << result\_vec[i].x << ", det\_count = " << tst.detection\_count << std::endl;

if (state\_id > -1) {

tst\_vec[i] = tst;

result\_vec\_pred[state\_id] = result\_vec[i];

result\_vec\_pred[state\_id].track\_id = track\_id;

}

}

return tst\_vec;

}

std::vector<bbox\_t> predict()

{

clear\_old\_states();

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

for (size\_t i = 0; i < max\_objects; ++i)

{

tst\_t tst = track\_id\_state\_id\_time[i];

if (tst.track\_id > -1) {

bbox\_t box = kalman\_vec[i].predict();

result\_vec\_pred[i].x = box.x;

result\_vec\_pred[i].y = box.y;

result\_vec\_pred[i].w = box.w;

result\_vec\_pred[i].h = box.h;

if (tst.detection\_count >= min\_frames)

{

if (track\_id\_state\_id\_time[i].track\_id == 0) {

track\_id\_state\_id\_time[i].track\_id = ++track\_id\_counter;

result\_vec\_pred[i].track\_id = track\_id\_counter;

}

result\_vec.push\_back(result\_vec\_pred[i]);

}

}

}

//std::cerr << " result\_vec.size() = " << result\_vec.size() << std::endl;

//global\_last\_time = std::chrono::steady\_clock::now();

return result\_vec;

}

std::vector<bbox\_t> correct(std::vector<bbox\_t> result\_vec)

{

calc\_dt();

clear\_old\_states();

for (size\_t i = 0; i < max\_objects; ++i)

track\_id\_state\_id\_time[i].detection\_count--;

std::vector<tst\_t> tst\_vec = find\_state\_ids(result\_vec);

for (size\_t i = 0; i < tst\_vec.size(); ++i) {

tst\_t tst = tst\_vec[i];

int state\_id = tst.state\_id;

if (state\_id > -1)

{

kalman\_vec[state\_id].set\_delta\_time(dT);

kalman\_vec[state\_id].correct(result\_vec\_pred[state\_id]);

}

}

result\_vec = predict();

global\_last\_time = std::chrono::steady\_clock::now();

return result\_vec;

}

};

// ----------------------------------------------

#endif // OPENCV

#endif // \_\_cplusplus

#endif // YOLO\_V2\_CLASS\_HPP