# 基于meanshift算法的多目标跟踪

因为项目需要，写了一个meanshift算法类对目标进行跟踪，由于需要跟踪多个目标，并记录这些目标的位置及状态（运行和停止），如果停止的时间超过一定阈值，则认为该目标已跟丢，放弃该目标，当所有目标都跟丢，则重新寻找目标进行跟踪。该方法应用于短时干扰较多的场合，例如，在复杂背景的手势识别过程中，手势的动作是基本是持续的，常态化的，间断时间较小，而其它背景因素的干扰虽然很多，但基本是短时的。在手势跟踪过程中，先对所有的运动目标进行跟踪，待一段时间后，原有的干扰噪声会消失，而手势仍然在运行，因此可以有效的从复杂背景中提取并跟踪手势。

## 实现方法

### FrameStatus类

定义一个FrameStatus类，用于记录点的轨迹序列，并根据最新的点位置判定当前被跟踪物体是否处于运行或停止的状态。该类的实现代码如下：

//保存前FRAME\_NUM帧图像的位置，用于判断此时被跟踪物体的状态

#define FRAME\_NUM 20

class FrameStatus

{

public:

FrameStatus();

//添加最新的位置

void addPos(*Point* pt);

//获取当前状态，0-停止，1-运动

int getStatus();

private:

//轨迹序列

*Point* posArr[FRAME\_NUM];

//当前序号

int idx;

//序列是否已满

bool bFull;

};

FrameStatus::FrameStatus()

{

idx = 0;

bFull = false;

}

void FrameStatus::addPos(*Point* pt)

{

if (idx >= FRAME\_NUM)

{

idx = 0;

bFull = true;

}

posArr[idx] = pt;

idx++;

}

int FrameStatus::getStatus() // 0 - stop 1 - run

{

int res = 0;

int endIdx = bFull ? FRAME\_NUM - 1 : idx;

*Point* averPos = *Point*(0, 0);

for (int i = 0; i <= endIdx; i++)

{

averPos.*x* += posArr[i].*x*;

averPos.*y* += posArr[i].*y*;

}

averPos.*x* /= (endIdx + 1);

averPos.*y* /= (endIdx + 1);

if (*abs*(posArr[endIdx].*x* - averPos.*x*) +

*abs*(posArr[endIdx].*y* - averPos.*y*) > 10)

{

res = 1;

}

else

{

res = 0;

}

return res;

}

### ObjectTracker类

定义一个ObjectTracker类，用于执行meanshift算法，只提供一个track函数，给定一幅彩色图像和一个矩形区域，执行跟踪算法，并更新矩形区域，实现过程在此不介绍，该类的实现代码如下：

class ObjectTracker

{

public:

ObjectTracker();

int track(*Mat* imag, *Rect*& pos);

private:

bool bInit;

*Mat* hsv;

*Mat* hist;

*Mat* prehist;

*Mat* mask;

*Mat* hue;

*Mat* backproj;

int vmin;

int vmax;

int smin;

int hsize;

float hranges[2];

const float \*phranges;

FrameStatus fs;

};

ObjectTracker::ObjectTracker():phranges(hranges)

{

vmin = 10;

vmax = 256;

smin = 30;

hsize = 16;

hranges[0] = 0;

hranges[1] = 180;

bInit = false;

}

int ObjectTracker::track(*Mat* im, *Rect*& pos)

{

//转换到HSV空间

*cvtColor*(im, hsv, *CV\_RGB2HSV*);

//生成掩码

*inRange*(hsv, *Scalar*(0, smin, *MIN*(vmin, vmax)), *Scalar*(180, 256, *MAX*(vmin, vmax)), mask);

int ch[] = { 0, 0 };

//提取色度图

hue.*create*(hsv.*size*(), hsv.*depth*());

*mixChannels*(&hsv, 1, &hue, 1, ch, 1);

if (!bInit)

{

//提取感兴趣区域

*Mat* roi(hue, pos), maskRoi(mask, pos);

//计算直方图并归一化

*calcHist*(&roi, 1, 0, maskRoi, hist, 1, &hsize, &phranges);

*normalize*(hist, hist, 0, 255, *CV\_MINMAX*);

bInit = true;

}

//计算直方图投影图像

*calcBackProject*(&hue, 1, 0, hist, backproj, &phranges);

//执行meanShift算法

*meanShift*(backproj, pos, *TermCriteria*(*CV\_TERMCRIT\_EPS* | *CV\_TERMCRIT\_ITER*, 10, 1));

*Point* pt(pos.*x* + 0.5\*pos.*width*, pos.*y* + 0.5\*pos.*height*);

fs.addPos(pt);

return fs.getStatus();

}

### TrackerManager类

该类对多个tracker对象进行管理，包括生成指定数量的traker对象，获取跟踪器组的跟踪位置，获取当前跟踪器的数量，跟踪并更新跟踪器的状态这些函数。对这些函数进行说明如下：

1. void generate(*vector*<*Rect*> \_rects)

给定一个Rect的矩形区域向量，生成向量Size数量的跟踪器，并初始化各跟踪器的初始跟踪位置。

1. *vector*<*Rect*> getCurentPos()

获取各跟踪器当前位置区域。

1. int getTrackerNum()

获取当前跟踪器的数量。

1. void update(*Mat* im)

跟踪并更新跟踪器的状态。对输入图像，各跟踪器跟踪当前位置执行一次跟踪，如果发现跟踪器在某个位已停止一定时间，则删除该跟踪器。

实现代码如下：

//处于等待帧的数量，如果一个跟踪器处于停止状态的帧数大于WAITFRAME则放弃跟踪，销毁该跟踪器

#define WAITFRAME 40

class TrackerManager

{

public:

//给定一个区域向量，对每个初始区域创建一个跟踪器

void generate(*vector*<*Rect*> \_rects);

//获取跟踪器的跟踪位置

*vector*<*Rect*> getCurentPos();

//获取跟踪器的数量

int getTrackerNum();

//跟踪并更新跟踪器的状态

void update(*Mat* im);

private:

//清除跟踪器

void clearTracker();

//跟踪器指针向量

*vector*<ObjectTracker\*> trakers;

//保存跟踪器处于等待状态的时间

*vector*<int> waitTimes;

//跟踪位置向量

*vector*<*Rect*> rects;

};

void TrackerManager::generate(*vector*<*Rect*> \_rects)

{

clearTracker();

int num = \_rects.*size*();

for (int i = 0; i < num;i++)

{

ObjectTracker\* traker = new ObjectTracker;

trakers.*push\_back*(traker);

waitTimes.*push\_back*(0);

rects.*push\_back*(\_rects[i]);

}

}

void TrackerManager::clearTracker()

{

for (int i = 0; i < trakers.*size*();i++)

{

delete trakers[i];

trakers[i] = *NULL*;

}

trakers.*clear*();

waitTimes.*clear*();

rects.*clear*();

}

void TrackerManager::update(*Mat* im)

{

for (int i = 0; i < trakers.*size*(); i++)

{

int s = trakers[i]->track(im,rects[i]);

if (!s)

{

waitTimes[i]++;

}

else

{

waitTimes[i] = 0;

}

}

*vector*<ObjectTracker\*>::*iterator* trackerIt = trakers.*begin*();

*vector*<*Rect*>::*iterator* rectIt = rects.*begin*();

*vector*<int>::*iterator* timeIt = waitTimes.*begin*();

while (timeIt!=waitTimes.*end*())

{

if (\*timeIt > WAITFRAME)

{

delete \*trackerIt;

\*trackerIt = *NULL*;

trackerIt = trakers.*erase*(trackerIt);

rectIt = rects.*erase*(rectIt);

timeIt = waitTimes.*erase*(timeIt);

}

else

{

trackerIt++;

rectIt++;

timeIt++;

}

}

}

*vector*<*Rect*> TrackerManager::getCurentPos()

{

return rects;

}

int TrackerManager::getTrackerNum()

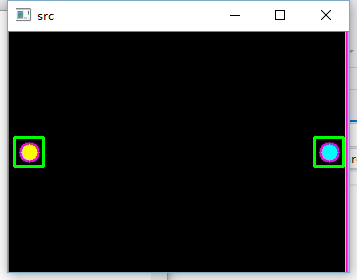
{

return trakers.*size*();

}

## 实例

实例使用的视频共500帧，其中有两个小球，前100帧，两个球停止，100帧以后，两个球向对方位置运动，右边的小球是左边小球运运的1.2倍，当右边小球运动到左边小球的初始位置后，该球运动停止，左边小球运动到右边小球的初始位置后，该球运动也停止，此时为400帧，后100帧，两球静止。



程序开始，绿框绘制出小球的位置，用户按下t键，将两球的初始位置设置为初始跟踪位置，并生成相应的跟踪器，交用红框表示跟踪器的位置。跟踪器持续对小球进行跟踪，当其中一个小球停止一段时间后，放弃对该球的跟踪，对应的跟踪器被删除，当另一个小球停止后一段时间后，该球对应的跟踪器也被删除。

int *\_tmain*(int argc, *\_TCHAR*\* argv[])

{

*VideoCapture* cap("ball.avi");

*Mat* im;

TrackerManager trackManager;

ObjectTracker tracker;

bool btrack = false;

cap >> im;

while (im.*data*)

{

*resize*(im, im, *Size*(340, 240));

if (!btrack)

{

*rectangle*(im, *Rect*(5, 105, 30, 30), *Scalar*(0, 255, 0), 2);

*rectangle*(im, *Rect*(305, 105, 30, 30), *Scalar*(0, 255, 0), 2);

}

else

{

trackManager.update(im);

*vector*<*Rect*> trackPos = trackManager.getCurentPos();

for (int i = 0;i<trackManager.getTrackerNum();i++)

{

*rectangle*(im, trackPos[i], *Scalar*(0, 0, 255), 2);

}

}

*imshow*("src", im);

char key = *waitKey*(10);

{

if (key == 't')

{

btrack = !btrack;

if (btrack)

{

*vector*<*Rect*> rects;

rects.*push\_back*(*Rect*(5, 105, 30, 30));

rects.*push\_back*(*Rect*(305, 105, 30, 30));

trackManager.generate(rects);

}

}

}

cap >> im;

}

return 0;

}

