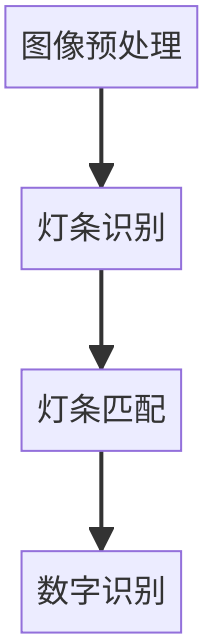
视觉组第10次培训

1. 装甲板识别

装甲板识别是RM视觉最基础的任务，虽然目前各种目标检测的深度学习算法层出不穷，但基于图像处理的方法凭借其高效性，仍然有其优势。

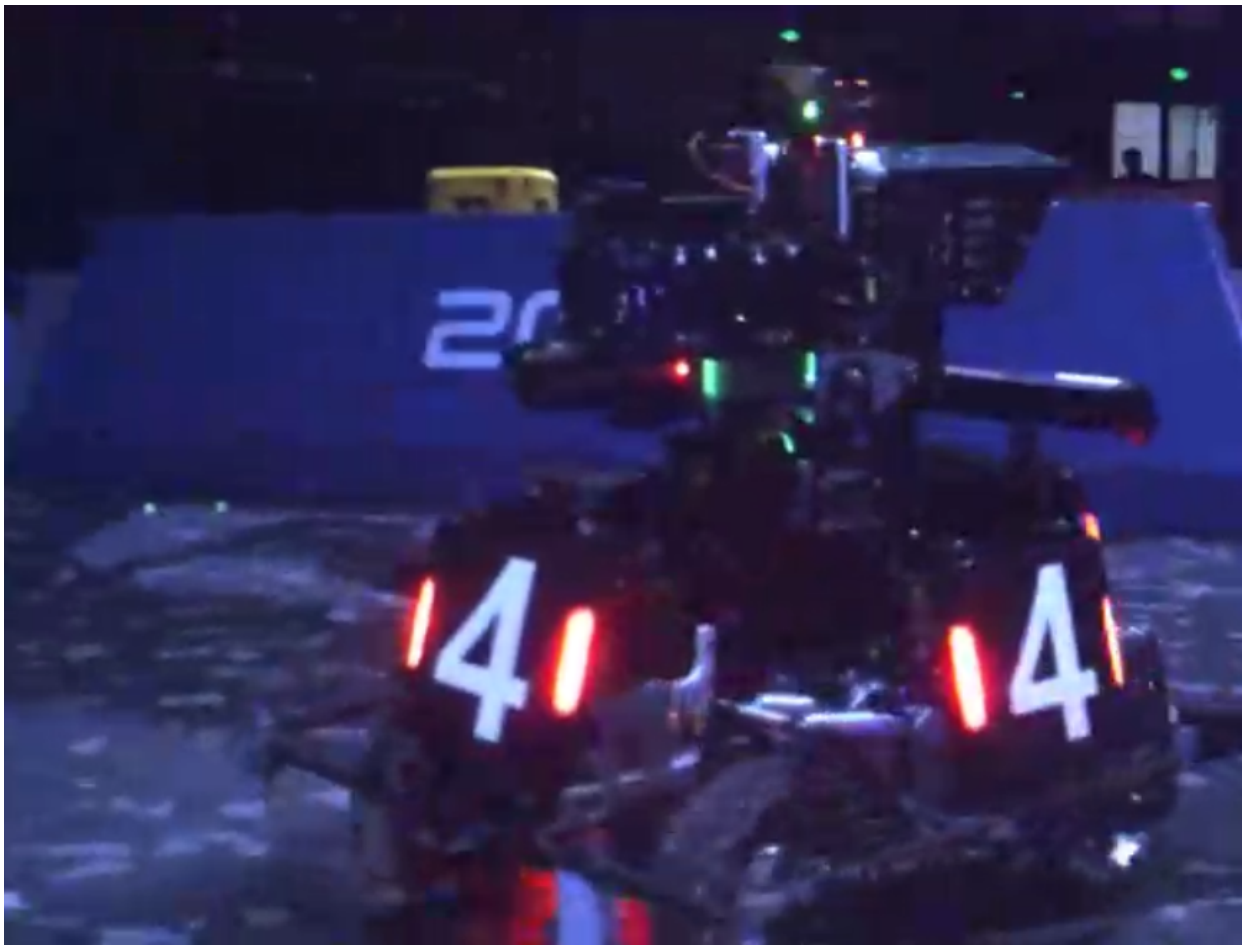
1.1 装甲板识别流程



1.2 图像预处理

图像预处理主要目的是提取图像中有用的部分用于之后的图像分析

如下图，该图为原始采集图像，纹理十分复杂，如果直接在原始图像上进行轮廓检测，将会检测出大量无效的轮廓，造成巨大的计算负担。



如果我们能去除图像中无效的纹理，只保留我们关心的装甲板部分的轮廓进行分析，将减少很多工作量。完成这项工作的函数就是二值化。

如下图，经过二值化后的图像只保留了灯条的轮廓，去除了其他与装甲板识别任务无关的轮廓，大大简化了识别任务的难度。

在RM自瞄，敌人可能是红色或者蓝色，我们可以利用颜色信息进行二值化，这就是通道相减法和HSV法

- 通道相减法 优点：速度快 缺点：对光线敏感
- HSV法 优点：对光线不敏感 缺点：速度慢

```
// 通道相减法
std::vector<cv::Mat> channels;
// 获取BGR三个通道
cv::split(src, channels);
cv::Mat gray, binary;
// 如果要识别红色装甲板，灯条R通道会远大于B通道
// 如果用R通道-B通道，红色区域的像素值应远大于0
if (enemy_color_ == ArmorColor::BLUE) {
    cv::subtract(channels[0], channels[2], gray);
} else {
    cv::subtract(chnnels[2], channels[0], gray);
}
cv::threshold(gray, binary, 100, 255, cv::THRESH_BINARY);
```

```
// HSV法
cv::Mat hsv;
cv::cvtColor(src, hsv, cv::COLOR_BGR2HSV);

cv::Mat binary;
if (enemy_color_ == ArmorColor::BLUE) {
    cv::Scalar lower = {100, 43, 46};
    cv::Scalar upper = {124, 255, 255};
    cv::inRange(hsv, lower, upper, binary);
} else {
    ...
}
```

BUT, 由于一般工业相机的动态范围不够大，导致若要能够清晰分辨装甲板的数字，得到的相机图像中灯条中心就
会过曝，灯条中心的像素点的值往往都是 R=B，这样导致了使用通道相减法时，容易出现轮廓断裂的问题，需要配合**图像形态学操作** cv::morphologyEx()才能取得比较好的效果。而HSV法的色彩空间转换的过程非常耗时，因此这里使用了亮度二值化的方法。

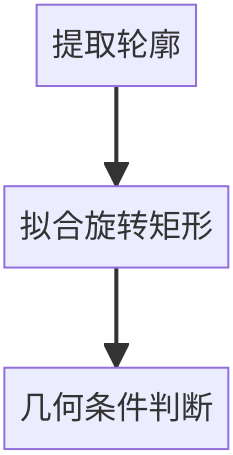
		
原图	通道相减法	亮度二值化

那颜色怎么办？我们将颜色判断放在了后面的逻辑里，单独判断。

```
cv::Mat Detector::preprocessImage(const cv::Mat & rgb_img)
{
  cv::Mat gray_img;
  cv::cvtColor(rgb_img, gray_img, cv::COLOR_RGB2GRAY);
  cv::Mat binary_img;
  cv::threshold(gray_img, binary_img, binary_thres, 255, cv::THRESH_BINARY);
  return binary_img;
}
```

1.3 灯条识别

由上一节可以知道，预处理完后的图像保留了灯条的轮廓，我们可以使用轮廓检测的方法提取出灯条的轮廓，拟合出最小外接矩形，通过长宽比等几何条件判断该轮廓是否是灯条轮廓。



这里的几何条件可以用的有很多，例如

- 长度是否合理
- 长宽比是否合理
- 角度是否合理

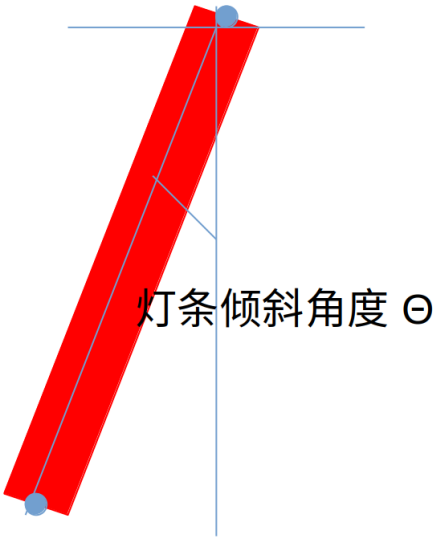
使用findContours()寻找轮廓，提取完轮廓后遍历轮廓，对每个轮廓使用minAreaRect()拟合出其最小面积外接矩形，并构造Light类。Light类继承自cv::RotatedRect类，在构造函数Light::Light()中计算灯条的倾斜角度tilt_angle、长度length和宽度width信息

```
std::vector<Light> Detector::findLights(const cv::Mat & rgb_img, const cv::Mat &
binary_img) {
    // 提取轮廓
    using std::vector;
    vector<vector<cv::Point>> contours;
    vector<cv::Vec4i> hierarchy;
    cv::findContours(binary_img, contours, hierarchy, cv::RETR_EXTERNAL,
cv::CHAIN_APPROX_NONE);
    std::vector<Light> lights;
    // 遍历轮廓
    for (const auto & contour : contours) {
      cv::RotatedRect r_rect = cv::minAreaRect(contour);
      auto light = Light(r_rect);
      // 判断几何条件
      if (isLight(light)) {
        int sum_r = 0, sum_b = 0;
        for (const auto & p : contour) {
          sum_r += rgb_img.at<cv::Vec3b>(p.y, p.x)[0];
          sum_b += rgb_img.at<cv::Vec3b>(p.y, p.x)[2];
        }
      light.color = sum_r > sum_b ? ArmorColor::RED : ArmorColor::BLUE;
      lights.push_back(light);
      }
    }
    return lights;
}
```

```
struct Light : public cv::RotatedRect {
  explicit Light(cv::RotatedRect box) : cv::RotatedRect(box) {
    cv::Point2f p[4];
    box.points(p);
    std::sort(p, p + 4, [](const cv::Point2f & a, const cv::Point2f & b) { return
a.y < b.y; });
    top = (p[0] + p[1]) / 2;
    bottom = (p[2] + p[3]) / 2;

    length = cv::norm(top - bottom);
    width = cv::norm(p[0] - p[1]);

    tilt_angle = std::atan2(std::abs(top.x - bottom.x), std::abs(top.y - bottom.y));
    tilt_angle = tilt_angle / CV_PI * 180;
  }

  cv::Point2f top;
  cv::Point2f bottom;
  double length;
  double width;
  double angle;
};
```

```
bool Detector::isLight(const Light & light)
{
  // The ratio of light (short side / long side)
  float ratio = light.width / light.length;
  bool ratio_ok = min_ratio < ratio && ratio < max_ratio;

  bool angle_ok = light.tilt_angle < max_angle;

  bool is_light = ratio_ok && angle_ok;

  return is_light;
}
```

`light.tilt_angle`示意图



1.4 灯条匹配

一块装甲板有两个灯条，要知道一个装甲板在图像中的坐标，需要知道这块装甲板的两个灯条具体是哪两个，所以在识别完灯条后，需要对灯条进行两两匹配。匹配后仍然通过几何条件判断这两个灯条是否能组成一个装甲板，几何条件可以选的有

- 两个灯条是否平行
- 两个灯条大小长度是否近似
- 两个灯条组成的装甲板的长宽比是否合理

```
std::vector<Armor> Detector::matchLights(const std::vector<Light> & lights) {
  // 两两组合所有灯条
  for (auto light_1 = lights.begin(); light_1 != lights.end(); light_1++) {
    for (auto light_2 = light_1 + 1; light_2 != lights.end(); light_2++) {
      // 只匹配颜色相符的灯条
      if (light_1->color != enemy_color || light_2->color != enemy_color) continue;
      // 对匹配上的灯条通过几何条件判断是否为装甲板
      auto type = isArmor(*light_1, *light_2);
      if (type != ArmorType::INVALID) {
        auto armor = Armor(*light_1, *light_2);
        armor.type = type;
        armors.emplace_back(armor);
      }
    }
  }

  return armors;
}
```

```
struct Armor {
    Armor(const Light & left, const Light & right) : left_light(left),
right_light(right) {}
    ArmorType type;
    int number;
};
```

1.5 识别器

为了提高代码的可维护性，我们使用面向对象的思想对装甲板识别器进行抽象和封装

```
enum class ArmorType {INVALID, LARGE, SMALL};
enum class ArmorColor {RED, BLUE};
```

```
struct DetectorParams {
  ArmorColor color;
  int binary_thresh;
  float light_min_ratio_thresh;
  float light_max_ratio_thresh;
  float light_max_angle_thresh;
};

class Detector {
public:
  explicit Detector(const DetectorParams & params);
  // 识别装甲板
  std::vector<Armor> detect(cv::Mat & rgb_img);
  // 绘制识别结果、各种变量，用于调试
  void drawResults(cv::Mat & rgb_img);
private:
  cv::Mat preprocessImage(const cv::Mat & input);
  std::vector<Light> findLights(const cv::Mat & rgb_img, const cv::Mat &
binary_img);
  std::vector<Armor> matchLights(const std::vector<Light> & lights);

  DetectorParams params_;
  // 上一次识别到的灯条
  std::vector<Light> lights_;
  // 上一次识别到的装甲板
  std::vector<Armor> armors_;
  // 数字识别器，还未讲到
  // std::shared_ptr<NumberClassifier> classifier_;
  // 各种用于调试的变量
  ...
};

// detect函数
std::vector<Armor> Detector::detect(cv::Mat & input) {
  binary_img = preprocessImage(input);
  lights_ = findLights(input, binary_img);
```

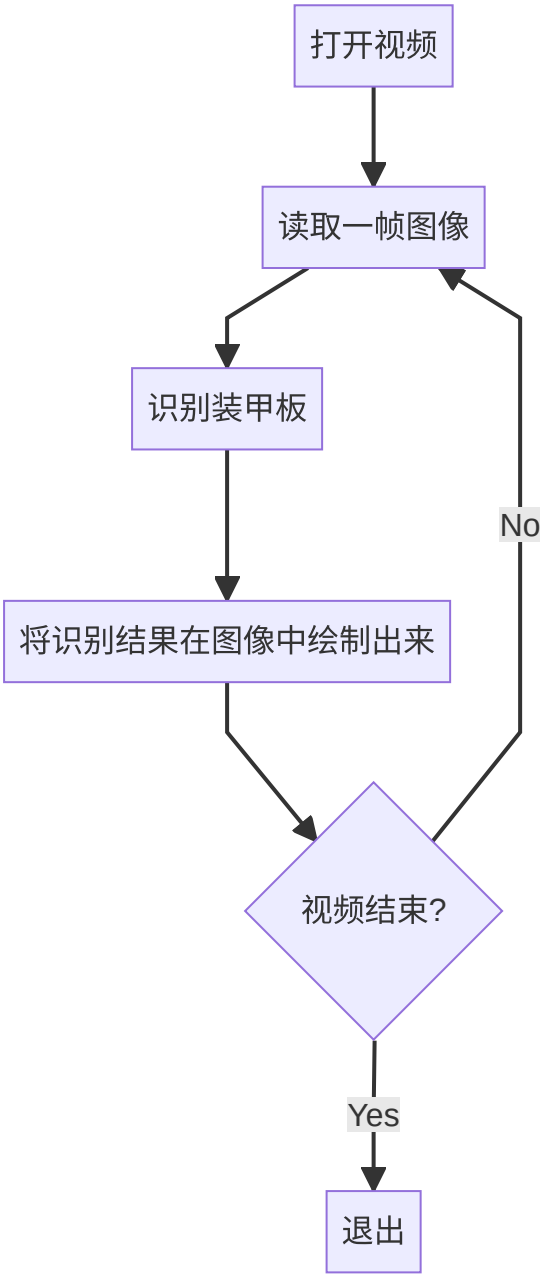
```
  armors_ = matchLights(lights_);

  if (!armors_.empty()) {
    // classifier_->extractNumbers(input, armors_);
    // classifier_->classify(armors_);
  }
  return armors_;
}
```

第一次实战训练

本次实战训练所有第一批预备队员一定要参加，第二批预备队员选择性参加（在最终入队考核中会参考）

实战内容

实现一个视频流装甲板识别pipeline，流程图如下



代码要求

为了改善预备队员的代码风格，本次实战训练要求必须实现以下两个类，使用`yaml-cpp`动态读取参数，**每个类必须放在不同文件中**，将算法相关的代码放在rm命名空间下，每个函数尽量不超过120行。

- AlgorithmPipeline
- Detector

要求main函数只能有如下代码

```
int main() {
  std::string config_path = "/your/config/path.yaml"
  auto pipeline = std::make_shared<rm::AlgorithPipeline>(config_path);
  return pipeline.run();
}
```

一个AlgorithmPipeline类的简单定义（仅供参考）：

```
namespace rm {
  class AlgorithmPipeline {
    public:
      AlgorithmPipeline(const std::string & config_path);
      // return 0:正常结束 1:异常结束（参数不对，视频打不开等）
      int run();
    private:
      std::shared_ptr<Detector> detector_;
      cv::VideoCapture video_capture_;
      ...
  };
}
```

YAML-CPP简单介绍

yaml文件是一种简单的key-value格式文件，常用来作为参数文件读取，例如：

```
# config.yaml
debug: true
video_path: "/home/zcf/video/ood_red.mp4"
size: [1280, 720]
binary_thresh: 100
light:
    min_ratio: 1.0
    max_ratio: 5.0
    min_area: 20.0
    max_angle: 40.0
armor:
    ...
```

yaml-cpp是一个用于读取yaml文件的cpp库，使用方法如下：

```cpp
#include <yaml-cpp/yaml.h>
int main() {
  // 创建一个YAML::Node
  YAML::Node config = YAML::LoadFile("config.yaml");
  // 读取不同类型的参数
  bool debug = config["debug"].as<bool>();
  int binary_thresh = config["binary_thresh"].as<int>();
  std::string path = config["video_path"].as<std::string>();
  std::vector<int> size = config["size"].as<std::vector<int>>();
  // 多级参数
  float light_min_ratio = config["light"]["min_ratio"].as<float>();
}
```

2. OpenCV-Python

介绍

OpenCV-Python是OpenCV库的Python接口，实际上是对C++的OpenCV的包装，底层调用的任然是C++库函数。在Python版本中，没有Mat这个类，使用`numpy`的array类型表示矩阵

安装与导入

```
pip install opencv-python
```

```python
import numpy as np
import cv2
```

读取图片

可见，Python的OpenCV与C++的OpenCV使用起来几乎一模一样

```python
src = cv2.imread("./Lena.png", cv2.IMREAD_COLOR);
cv2.imshow("src", src)
cv2.waitKey(0)
```

numpy.ndarray

Python的OpenCV使用numpy.ndarray来代替原来的cv::Mat，array支持切片操作，所以可能用起来比C++还要顺手，更多的用法可以看numpy的文档

```python
# 创建一张黑色图像
src = np.zeros((300, 300, 3), dtype=np.uint8)
cv2.imshow("src", src)
cv2.waitKey(0)
```

```python
src = cv2.imread("./Lena.png", cv2.IMREAD_COLOR)
# 获取图像大小
print(src.shape) # (512, 512, 3)
H, W, C = src.shape

# 访问图像元素
print(src[100,100]) # [78 68 178]
print(src[100,100,0]) # 78

# 切片索引
red_channel = src[:,:,2]
blue_channel = src[:,:,0]
top_half = src[0:int(H/2), :, :]
cv2.imshow("red", red_channel)
cv2.imshow("top_half", top_half)

# 获取平均值
print(src.mean()) # 128.22837575276694
```