```cpp
/*------------------------------------------------------------------------------
Filename:     vision.cpp
Project:      IEEE SoutheastCon Hardware Competition 2019
School:       Auburn University
Organization: Student Projects and Research Committee (SPARC)
Description:  Takes pictures on the Raspberry Pi Camera V2 and processes them
with OpenCV2 via color recognition.

Color Indices = red(0), yellow(1), blue(2), green(3)
------------------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <string>
#include <opencv2/opencv.hpp>
#include <opencv2/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/videoio.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/highgui.hpp>
#include <raspicam/raspicam_cv.h>
#include <ctime>
#include <std_msgs/Float32.h>
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
#include <opencv_node/vision_msg.h>
#include "Vision3D.h"

// Constants
const double PI = 3.14159265;
const int MIN_AREA = 200;
const int MAX_AREA = 30000;
const bool VISION_DEBUG_IMAGE = true;
const int VISION_DEBUG_COLOR_IMAGE = -1; // -1 to disable (0 red,1 yellow, 2 blue, 3 green)
const bool VISION_DEBUG_TEXT = false;
const bool VISION_DEBUG_3D = true;
const double DEBRIS_MIN_W2H = 0.75;
const double DEBRIS_MAX_W2H = 1.5;
const double CORNER_MIN_W2H  = .1;
const double CORNER_MAX_W2H = .4;
const double DEBRIS_MIN_PERCENT_FILLED = 0.70;
const double DISTANCE_MULTIPLIER = 26.95;
int colorChoose = 0;
// Common Namespaces
using namespace cv;
using namespace std;


void colorSelected(const std_msgs::Float32ConstPtr &msg){
                colorChoose = int(msg->data);
        }


enum Colors {
  Red,
  Yellow,
  Blue,
  Green,
  All
};



// Debris Object Namespace
namespace IEEE_VISION
{
struct DebrisObject
{
  Point center;
  int width;
  int height;
  int colorIndex;
  int angle;
  double distance;
  Point2d position;

  enum class ObjectType {
    Debris,
    Corner,
    CenterFace,
    Unknown
  } type;

  DebrisObject(Rect boundingRect, int new_colorIndex, int new_angle, double new_distance, ObjectType typeIn, Point2d positionIn) : colorIndex{new_colorIndex}, angle{new_angle}, type{typeIn}, position{positionIn}
  {
    center.x = boundingRect.x + boundingRect.width / 2;
    center.y = boundingRect.y + boundingRect.height / 2;
    width = boundingRect.width;
    height = boundingRect.height;
    distance = new_distance;
  }
  void printProperties()
  {
    cout << "X=" << center.x << " Y=" << center.y << " Width=" << width << " Height=" << height << " colorIndex=" << colorIndex
    << " angle=" << angle << " distance=" << distance << "\n";
  }
  double getHalfHeight() const {
    if(type == ObjectType::Debris)
      return Vision3D::AvgDebrisHeight / 2;
    else if(type == ObjectType::Corner)
      return Vision3D::CornerHeight / 2;
    else if(type == ObjectType::CenterFace)
      return 0.0;   //unimplemented
    else
      return 0.0;
  }

};

vector<DebrisObject> objectProperties;

struct VisionHandle
{
  raspicam::RaspiCam_Cv Camera;
  Mat image, hsv, threshed, threshedSecondary;

  private:
  vector<vector<Point>> contours;
  vector<Vec4i> hierarchy;
  Mat temp;
  Scalar lowerThreshes[4] = {Scalar(0, 98, 105), Scalar(23, 80, 90), Scalar(89, 56, 100), Scalar(37, 44, 70)};
  Scalar upperThreshes[4] = {Scalar(9, 255, 255), Scalar(35, 255, 255), Scalar(117, 255, 255), Scalar(77, 255, 255)};
  Scalar redSecondaryLower{170, 42, 52};
  Scalar redSecondaryUpper{180, 255, 255};
  Scalar colors[4] = {Scalar(0, 0, 255), Scalar(0, 255, 255), Scalar(255, 0, 0), Scalar(0, 255, 0)};
  String labels[4] = {"Red", "Yellow", "Blue", "Green"};
  Mat kernel = getStructuringElement(MORPH_CROSS, Size(3, 3));
  Size resolution;
  clock_t begin;
  int desiredColor = All;


  public:
  VisionHandle()
  {
    Camera.set(CV_CPU_POPCNT, CV_8UC3);
    Camera.set(CAP_PROP_FRAME_WIDTH, 640);
    Camera.set(CAP_PROP_FRAME_HEIGHT, 480);
    if (!Camera.open())
    {
      throw std::runtime_error("Error opening the camera");
    }
  }
  ~VisionHandle()
  {
    Camera.release();
  }

  // Takes a picture, saves it in image, and converts it to HSV
  void takePicture()
  {
    if (VISION_DEBUG_TEXT){
      begin = clock();
      cout << "getting picture..." << endl;
    }
    Camera.grab();
    Camera.retrieve(image);
    cvtColor(image, hsv, COLOR_BGR2HSV);
    resolution = image.size();
  }

  //Finds objects of all colors; assumes a picture has been taken
  void findObjects()
  {
    objectProperties.clear();
    if (desiredColor == All){
      for (int i = 0; i < 4; i++) {
        findObjectsOfColor(i);
      }
    }
    else{
      findObjectsOfColor(desiredColor);
    }
    if(VISION_DEBUG_TEXT)
      ROS_INFO("%s", "Finished finding objects");
    if(VISION_DEBUG_IMAGE)
      displayImage("output");
  }

  void debugInvalidObj(Mat imageIn, Rect bounds) {
    if(VISION_DEBUG_IMAGE) {
      rectangle(image, bounds.tl(), bounds.br(), Scalar(100, 100, 100), 4);
    }
  }

  void drawRotatedRect(Mat imageIn, RotatedRect rRect, Scalar color) {
    Point2f vertices[4];
    rRect.points(vertices);
    for (int i = 0; i < 4; i++)
      line(imageIn, vertices[i], vertices[(i+1)%4], color, 1);
  }

  // Populates vector array of object's properties; previously "GetObjectProperties"
  void findObjectsOfColor(int index)
  {
    //objectProperties.clear(); // needs removed when using findObjects()
    double area, angle, w2h, percentFilled, distance;
    // Generate contours
    contours.clear();
    hierarchy.clear();
    clock_t begin = clock();
    inRange(hsv, lowerThreshes[index], upperThreshes[index], threshed);
    if(index == Red) {
      inRange(hsv, redSecondaryLower, redSecondaryUpper, threshedSecondary);
      threshed |= threshedSecondary;
    }
    dilate(threshed, threshed, kernel);
    findContours(threshed, contours, hierarchy, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE, Point(0, 0));
    if (VISION_DEBUG_TEXT)
      cout << double(clock() - begin) / CLOCKS_PER_SEC << endl;
    if (index == VISION_DEBUG_COLOR_IMAGE) // ---- Show window of select color ----
    {
      namedWindow(labels[index], WINDOW_NORMAL); // Create a window for display.
      imshow(labels[index], threshed);       // Show our image inside it.
      waitKey(1);
    }
    // ---- Loop through each contour ----
    for (int i = 0; i < contours.size(); i++)
    {
      area = contourArea(contours[i]);
      if (area > MIN_AREA && area < MAX_AREA)
      {
        Rect boundRect = boundingRect(contours[i]);
        float r;
        Point2f cent;
        w2h = (double)boundRect.width / boundRect.height;          // Find width to height ratio, 1.0 is square
        RotatedRect rotatedBounds = minAreaRect(contours[i]);
        percentFilled = area / rotatedBounds.size.area(); // amount of rectangle consumed by contour
        // Determine shape
        DebrisObject::ObjectType objectType = DebrisObject::ObjectType::Unknown;
        Point2d position(0, 0);

        //vector<vector<Point>> smoothedContour(1);
        //approxPolyDP(contours[i], smoothedContour[0], .02 * arcLength(contours[i], true), true);

        if (percentFilled < DEBRIS_MIN_PERCENT_FILLED) {
          if (VISION_DEBUG_IMAGE)
            rectangle(image, boundRect.tl(), boundRect.br(), Scalar(200, 200, 200), 4, 8, 0);
          continue;
        }
        else if (w2h < DEBRIS_MAX_W2H && w2h > DEBRIS_MIN_W2H && boundRect.y != 0) {     //Checking the boundRect prevents detection of a corner that is mostly cut off to where it is square
          objectType = DebrisObject::ObjectType::Debris;
          position = Vision3D::getPosIfHeight((boundRect.br() + boundRect.tl()) / 2, Vision3D::AvgDebrisHeight / 2);    //It is assumed that the center of the boundRect goes through the centroid of the object
          if(position.x <= 0.0) {
            debugInvalidObj(image, boundRect);
            continue;
          }
          if (VISION_DEBUG_IMAGE) {
            rectangle(image, boundRect.tl(), boundRect.br(), colors[index], 4, 8, 0);
            //drawContours(image, contours, i, colors[index]);
          }
        }
        else {
          RotatedRect rotated = minAreaRect(contours[i]);
          double betterw2h = rotated.size.width / rotated.size.height;
          if(w2h > 1) {
            if(betterw2h <= 1)
              betterw2h = 1 / betterw2h;
          }
          else {
            if(betterw2h > 1)
              betterw2h = 1 / betterw2h;
          }
          //putText(image, format("%f", betterw2h), boundRect.br(), FONT_HERSHEY_COMPLEX_SMALL, .8, Scalar(20, 20, 20));

          if(betterw2h < CORNER_MAX_W2H && betterw2h > CORNER_MIN_W2H) {
            double squareEdge = boundRect.height * .3;
            if(boundRect.x  - squareEdge >= 0.0 && boundRect.x + boundRect.width + squareEdge < image.size().width) {   //ensure that the tested areas are inside the image
              double offset = boundRect.height * .4;
              Mat mask(image.size(), CV_8UC1, Scalar::all(0));
              Rect ROI(boundRect.x - squareEdge, boundRect.y + boundRect.height - squareEdge - offset, squareEdge, squareEdge);
              //rectangle(image, ROI.tl(), ROI.br(), Scalar(0, 0, 0), 1);
              mask(ROI).setTo(Scalar::all(255));
              ROI.x = boundRect.x + boundRect.width;
              //rectangle(image, ROI.tl(), ROI.br(), Scalar(0, 0, 0), 1);
              mask(ROI).setTo(Scalar::all(255));
              Scalar meanColor = mean(hsv, mask);       //The purpose of this is to see if the area on both sides of a potential corner is white.
```

```cpp
       //ROS_INFO("Average: %f, %f, %f", meanColor.val[0], meanColor.val[1], meanColor.val[2]);

       if(meanColor.val[1] <= 35.0 && meanColor.val[2] >= 145) {
        objectType = DebrisObject::ObjectType::Corner;
        Point2f points[4];
        rotated.points(points);
        sort(std::begin(points), std::end(points), [] (const Point2f& point1, const Point2f& point2) { return point1.y > point2.y; });
        //circle(image, (points[0] + points[1]) / 2.0, 3, colors[Red]);
        position = Vision3D::getPosIfHeight((points[0] + points[1]) / 2.0, 0.0);    //Approximate location of the center of the corner's bottom square. OpenCV rounds when converting from Point2f to Point2i
        if (VISION_DEBUG_IMAGE)
         drawRotatedRect(image, rotated, Scalar(0, 0, 0));
       }
       else {
        //drawContours(image, contours, i, Scalar(0, 0, 0));
        debugInvalidObj(image, boundRect);
       }
      }
      else {
       debugInvalidObj(image, boundRect);
      }
     }
     else { // wrong size ratio
      //drawContours(image, contours, i, Scalar(0, 0, 0));
      debugInvalidObj(image, boundRect);
     }
    }

    if(objectType != DebrisObject::ObjectType::Unknown) {
     angle = atan((double)(boundRect.x - image.cols / 2) / (double)(image.rows - boundRect.y)) * 180 / PI; // Find angle to center of object from centerline
     distance = (1/(double)boundRect.width) * DISTANCE_MULTIPLIER;
     objectProperties.push_back(DebrisObject(boundRect, index, angle, distance, objectType, position));
     if(VISION_DEBUG_3D) {
      stringstream text;
      text << objectProperties.back().position.x << ", " << objectProperties.back().position.y;
      putText(image, text.str().c_str(), objectProperties.back().center, FONT_HERSHEY_COMPLEX_SMALL, .8, Scalar(255, 255, 255));
     }
    }
   }
  }
 }

 // displays image if enabled
 void displayImage(string label)
 {
  if (VISION_DEBUG_IMAGE)
   {
    ROS_INFO("%s", "Displaying/Saving Picture...");
    imwrite("test.jpg",image);
    imshow(label, image); // Show our image inside it.
    waitKey(1);     // Wait for a keystroke in the window
   }
 }
```

```


  int processVision(int argc, char **argv)
  {
    ros::init(argc, argv, "vision_talker"); // initialize ROS
    ros::NodeHandle n;
    ros::Subscriber startColorSub = n.subscribe<std_msgs::Float32>("start_color", 1, colorSelected);
    ros::Publisher pub = n.advertise<opencv_node::vision_msg>("vision_info", 1000); // start publishing chatter
    ros::Rate loop_rate(10);
    while (ros::ok())
    {
      takePicture();
      findObjects();

      opencv_node::vision_msg msg;
      opencv_node::object data;

      for(std::size_t i=0; i<objectProperties.size(); ++i){
      if (objectProperties[i].type == DebrisObject::ObjectType::Debris ){
        data.x_position = objectProperties[i].position.x;
        data.y_position = objectProperties[i].position.y;
        data.color_index = objectProperties[i].colorIndex;
        data.object_type = (int)objectProperties[i].type;
        msg.objects.push_back(data);
      }
      }

      ROS_INFO("%s", "Sending object properties2");

      pub.publish(msg); // Sends messages

      ros::spinOnce();

      loop_rate.sleep();
    }

    return 0;
  }
};

};
int main(int argc, char **argv)
{

  IEEE_VISION::VisionHandle vis; // initialize vision
  vis.processVision(argc,argv);
  return 0;
}

```