Práctica Final Visión

RESUMEN

Defensa práctica final visión artificial
Marvin Pancracio Manso ,GIRS, 2023.

Detección de Pelota en 2D y 3D

En 2D se ha procedido a modificar el filtro de color de la p4 y usando el mismo algoritmo de detección de regiones.

```
cv::Mat balls_cv(const cv::Mat input_image) {

  cv::Mat gray,out_image, pink_img;
  out_image = input_image.clone();
  pink_img = pink_filter_cv(input_image);
  cvtColor(out_image, gray, CV_BGR2GRAY);

  cv::medianBlur(gray, gray, 5);
  std::vector<cv::Vec3f> circles;
  cv::HoughCircles( gray, circles, cv::HOUGH_GRADIENT, 1,
               gray.rows/16, // change this value to detect circles with different distances
                    to each other
               175, 30, 1, 100 // change the last two parameters
          // (min_radius & max_radius) to detect larger circles
  );
  //circles_img = gray;
  for( size_t i = 0; i < circles.size(); i++ )
  {
      cv::Vec3i c = circles[i];
      int radius = c[2];
      cv::Point2f center = cv::Point2f(c[0], c[1]);
      k_center.push_back(center); // usado en el apartado extra.
      k_radius.push_back(radius); // usado en el apartado extra.
      // circle center
      if (pink_img.at<uchar>(center) > 0) {
        point.push_back(center);
        cv::circle( out_image, center, 1, cv::Scalar(0,0,0), 3, cv::LINE_AA);
        cv::circle( out_image, center, radius, cv::Scalar(0,0,255), 3, cv::LINE_AA);
      }
      // circle outline
  }
  return out_image;
}
```

Para la detección en pcl, se ha usado el mismo algoritmo que en la p6 con el mismo filtro de color y métodos auxiliares como: outliers() o draw_square()

```
void get_points(pcl::PointCloud<pcl::PointXYZRGB> cloud_in)
{
    // Create the filtering object
    pcl::PointCloud<pcl::PointXYZRGB> filtered_cloud, cloud_f;
    pcl::PointXYZ position;


    pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients ());
    pcl::PointIndices::Ptr inliers (new pcl::PointIndices ());
    //pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_f (new pcl::PointCloud<pcl::PointXYZ>);
    // Create the segmentation object
    pcl::SACSegmentation<pcl::PointXYZRGB> seg;
    // Optional
    seg.setOptimizeCoefficients (true);
    // Mandatory
    seg.setModelType (pcl::SACMODEL_SPHERE);
    seg.setMethodType (pcl::SAC_RANSAC);
    seg.setMaxIterations (1000);
    seg.setDistanceThreshold (0.01);

    // Create the filtering object
    pcl::ExtractIndices<pcl::PointXYZRGB> extract;

    int i = 0, nr_points = (int) cloud_in.size ();
    // While 30% of the original cloud is still there
    while (cloud_in.size () > 0.01 * nr_points)
    {
      // Segment the largest planar component from the remaining cloud
      seg.setInputCloud (std::make_shared<pcl::PointCloud<pcl::PointXYZRGB>>(cloud_in));
      seg.segment (*inliers, *coefficients);
      if (inliers->indices.size () == 0)
      {
        std::cerr << "Could␣not␣estimate␣a␣planar␣model␣for␣the␣given␣dataset." << std::endl
            ;
        break;
      }

      // Extract the inliers
      extract.setInputCloud (std::make_shared<pcl::PointCloud<pcl::PointXYZRGB>>(cloud_in));
      extract.setIndices (inliers);
      extract.setNegative (false);
      extract.filter (filtered_cloud);

      float sphere_center_x = coefficients->values[0];
      float sphere_center_y = coefficients->values[1];
      float sphere_center_z = coefficients->values[2];

      position.x = sphere_center_x;
```

```
      position.y = sphere_center_y;
      position.z = sphere_center_z;

      //std::cerr << "x: " << sphere_center_x << ",y: "<< sphere_center_y << " ,z: "<<
          sphere_center_z << std::endl;

      square_pos.push_back(position);

      // Create the filtering object
      extract.setNegative (true);
      extract.filter (cloud_f);
      cloud_in.swap (cloud_f);

      i++;
    }
}
```

Proyección de líneas y cubos de distancia

Para la proyección de lineas en OpenCv bastó con reutilizar el código de la p5:

```
cv::Mat project_lines(cv::Mat input_image){

  cv::Mat img_clone = input_image.clone();
  std::vector<cv::Point3f> point_3D;
  cv::Mat K, R, T;
  K = cv::Mat(3,3,CV_64F,K_.val);

  for (int i = 0; i < distance_trackbar; i++) {

    point_3D.push_back(cv::Point3f(i+1,-1.4,0));
    point_3D.push_back(cv::Point3f(i+1,1.4,0));

    R = (cv::Mat_<double>(3,3) << 0,1,0,0,0,1,1,0,0);
    T = (cv::Mat_<double>(3,1) << t.transform.translation.x,t.transform.translation.y,t.
        transform.translation.z);

    projectPoints(point_3D,R,T,K,cv::noArray(),point_line);
  }

  for (int i = 1; i < distance_trackbar; i++) {
    cv::circle(img_clone,point_line[i*2], 1, cv::Scalar(0,i*42,255-i*42), 5, cv::LINE_AA);
    cv::circle(img_clone,point_line[i*2+1], 1, cv::Scalar(0,i*42,255-i*42), 5, cv::LINE_AA);
    cv::line(img_clone,point_line[i*2], point_line[i*2+1], cv::Scalar(0,i*42,255-i*42),2,cv
        ::LINE_AA);
    std::stringstream text;
    text << i+1;
    cv::putText(img_clone,text.str(),point_line[i*2+1] + cv::Point2f(20,5),cv::
        FONT_HERSHEY_SIMPLEX, 0.5, cv::Scalar(0,i*42,255-i*42));
```

```
  }
  return img_clone;
}
```

Por otro lado, la proyección de cubos reutiliza código de la p6, con alguna modificación:

```
pcl::PointCloud<pcl::PointXYZRGB> calculate_cube_pos(pcl::PointCloud<pcl::PointXYZRGB>
    cloud_in, int distance)
{
  cv::Mat rot_tras;
  int r, g, b;
  rot_tras = (cv::Mat_<double>(4, 4) <<
    0.0, 1.0, 0.0, t.transform.translation.x,
    0.0, 0.0, 1.0, t.transform.translation.y,
    1.0, 0.0, 0.0, t.transform.translation.z,
    0.0, 0.0, 0.0, 1.0);

  if(distance > 2) {

    for(int i = 0; i < distance - 2; i++){
    // guardo los puntos de cada distancia
    cv::Mat point1 = (cv::Mat_<double>(4,1) << i+3, 1, 0, 1.0);
    cv::Mat point2 = (cv::Mat_<double>(4,1) << i+3, -1, 0, 1.0);
    // realizo la roatcion y translacin
    cv::Mat t_point1 = rot_tras * point1;
    cv::Mat t_point2 = rot_tras * point2;
    // obtengo los valores normalizados
    double x1 = t_point1.at<double>(0,0) / t_point1.at<double>(3,0);
    double y1 = t_point1.at<double>(1,0) / t_point1.at<double>(3,0);
    double z1 = t_point1.at<double>(2,0) / t_point1.at<double>(3,0);

    double x2 = t_point2.at<double>(0,0) / t_point2.at<double>(3,0);
    double y2 = t_point2.at<double>(1,0) / t_point2.at<double>(3,0);
    double z2 = t_point2.at<double>(2,0) / t_point2.at<double>(3,0);
    // calculo el color
    r = 255 - i * 42, g = i * 42, b = 0;
    // pinto el cubo
    cloud_in = draw_square_dregaded(cloud_in, x1, y1, z1, r, g ,b);
    cloud_in = draw_square_dregaded(cloud_in, x2, y2, z2, r, g, b);
    }
  }
  return cloud_in;
}
```

Proyección de 2D a 3D

Para la proyección de los puntos de OpenCv en cubos negros en PointCloud, se ha usado los centros calculados en balls_cv, y la imagen de profundidad para calcular z. Los centros se almacenaban en un vector de

cv::Point2f. El siguiente método se usa para almacenar en un vector global pcl::PointXYZ los puntos centrales de estos futuros cubos negros:

```
void project_points( ) {
// metodo para proyectar los puntos de la imagen de profundidad en cubos negros en pcl.
  cv::Mat img_depth = depth_img.clone();
  pcl::PointXYZ position;

  std::vector<cv::Point3f> point3D;

  for (int i = 0; i < img_depth.rows; i++) // eliminamos los valores infinitos.
  {
    for (int j = 0; j < img_depth.cols; j++)
    {
      if(isnan(img_depth.at<float>(i, j)) || isinf(img_depth.at<float>(i, j)))
      {
        img_depth.at<float>(i, j) = 0.0;
      }
    }
  }

  int size = point.size();
  for (int i = 0; i < size; i++)
  {
    float d = img_depth.at<float>(point[i].y,point[i].x);
    float x = point[i].x; float cx = img_depth.rows/2;
    float y = point[i].y; float cy = img_depth.cols/2;

    black_square_pos.push_back(pcl::PointXYZ((x - cx)*d/522,(y - cy)*d/522,d)); //
        almacenamos en el vector los puntos centrales de los cubos
    position.x = (x - cx)*d/522;
    position.y = (y - cy)*d/522;
    position.z = d;

  }
}
```

Posteriormente estos puntos pcl se usaran en el método dedidaco a proyectar estos cubos:

```
pcl::PointCloud<pcl::PointXYZRGB> draw_square_black(pcl::PointCloud<pcl::PointXYZRGB>
    cloud_in, float x_, float y_, float z_)
{ // metodo para dibujar cuadrados en una nube de puntos cloud_in.

  float size = 0.1;
  float step = 0.01;
  std::vector<pcl::PointXYZRGB> vertices;
  for (float i = -size/2; i <= size/2; i += step) {
      for (float j = -size/2; j <= size/2; j += step) {
          for (float k = -size/2; k <= size/2; k += step) {
              pcl::PointXYZRGB vertex(x_+i, y_+j, z_+k, 0, 0, 0);
```

```
            cloud_in.push_back(vertex);
        }
    }
  }
  return cloud_in;
}
```

Detección de persona.

Para este apartado, se ha usado el código proporcionado por el profesor (Josemi). Aplicando ciertas modificaciones para hacerlo más afin a esta práctica. Dividiendo en 3 métodos el mismo:

```
void person_detected(cv::Mat input_img) {
 //  mtodo  principal para deteccin de persona.
  std::vector<cv::String> classes;
  cv::Mat frame, blob;

  frame = input_img;
  cv::dnn::blobFromImage(
      frame, blob, 1 / 255.0, cv::Size(inpWidth, inpHeight), cv::Scalar(
        0, 0,
        0), true, false);

  //Sets the input to the network
  net.setInput(blob);

  // Runs the forward pass to get output of the output layers
  std::vector<cv::Mat> outs;
  net.forward(outs, getOutputsNames(net));

  // Remove the bounding boxes with low confidence
  postprocess(outs);
}
```

```
std::vector<cv::String> getOutputsNames(const cv::dnn::Net & net)
{ // Conseguimos los nombres de los objetos detectados.
  static std::vector<cv::String> names;
  if (names.empty()) {
    //Get the indices of the output layers, i.e. the layers with unconnected outputs
    std::vector<int> outLayers = net.getUnconnectedOutLayers();

    //get the names of all the layers in the network
    std::vector<cv::String> layersNames = net.getLayerNames();

    // Get the names of the output layers in names
    names.resize(outLayers.size());
    for (size_t i = 0; i < outLayers.size(); ++i) {
      names[i] = layersNames[outLayers[i] - 1];
    }
```

```
  }
  return names;
}
```

```
void postprocess(const std::vector<cv::Mat> & outs)
{
  // Procesammos los nombres detectados, y si corresponde con person, hemos detactado lo que
       queriamos.
  for (size_t i = 0; i < outs.size(); ++i) {
    // Scan through all the bounding boxes output from the network and keep only the
    // ones with high confidence scores. Assign the box's class label as the class
    // with the highest score for the box.
    float * data = (float *)outs[i].data;
    for (int j = 0; j < outs[i].rows; ++j, data += outs[i].cols) {
      cv::Mat scores = outs[i].row(j).colRange(5, outs[i].cols);
      cv::Point classIdPoint;
      double confidence;
      // Get the value and location of the maximum score
      cv::minMaxLoc(scores, 0, &confidence, 0, &classIdPoint);
      if (confidence > confThreshold) {
        if (classIdPoint.x == 0) // person es 0.
        {
          person = 1;
        }
      }
    }
  }
}
```

Extra

Se ha querido implementar K-means 3D, pero por el momemnto se ha obtenido en 2D. Para ello se ha usado el código propocionado por el profesor (Josemi), aplicando de nuevo ciertas modificaciones:

```
void k_means(cv::Mat input_img) {
  cv::RNG rng(k_center.size());
  for (;; ) {
    int k, clusterCount = rng.uniform(2, k_center.size() + 1);
    int i, sampleCount = rng.uniform(1, 1001);
    cv::Mat points(sampleCount, 1, CV_32FC2), labels;

    clusterCount = MIN(clusterCount, sampleCount);
    std::vector<cv::Point2f> centers;

    /* generate random sample from multigaussian distribution */
    for (k = 0; k < clusterCount; k++) {
      cv::Point center;
      center.x = k_center[k].x;
      center.y = k_center[k].y;
```

```
      cv::Mat pointChunk = points.rowRange(
        k * sampleCount / clusterCount,
        k == clusterCount - 1 ? sampleCount :
        (k + 1) * sampleCount / clusterCount);
      rng.fill(
        pointChunk, cv::RNG::RNG::NORMAL, cv::Scalar(center.x, center.y),
        cv::Scalar(input_img.cols * 0.05, input_img.rows * 0.05));
    }

    cv::randShuffle(points, 1, &rng);

    double compactness = cv::kmeans(
      points, clusterCount, labels,
      cv::TermCriteria(cv::TermCriteria::EPS + cv::TermCriteria::COUNT, 10, 1.0),
      3, cv::KMEANS_PP_CENTERS, centers);
    input_img = cv::Scalar::all(0);

    for (i = 0; i < sampleCount; i++) {
      int clusterIdx = labels.at<int>(i);
      cv::Point ipt = points.at<cv::Point2f>(i);
      circle(input_img, ipt, 2, colorTab[clusterIdx], cv::FILLED, cv::LINE_AA);
    }
    for (i = 0; i < (int)k_center.size(); ++i) {
      cv::Point2f c = k_center[i];
      circle(input_img, c, k_radius[i], colorTab[i], 1, cv::LINE_AA);
    }

    std::cout << "Compactness:␣" << compactness << std::endl;
    imshow("clusters", input_img);
    char key = (char)cv::waitKey();
    if (key == 27 || key == 'q' || key == 'Q') { // 'ESC'
      break;
    }
  }
}
```

Authors

Marvin Pancracio Manso 3ºGIRS. 2023