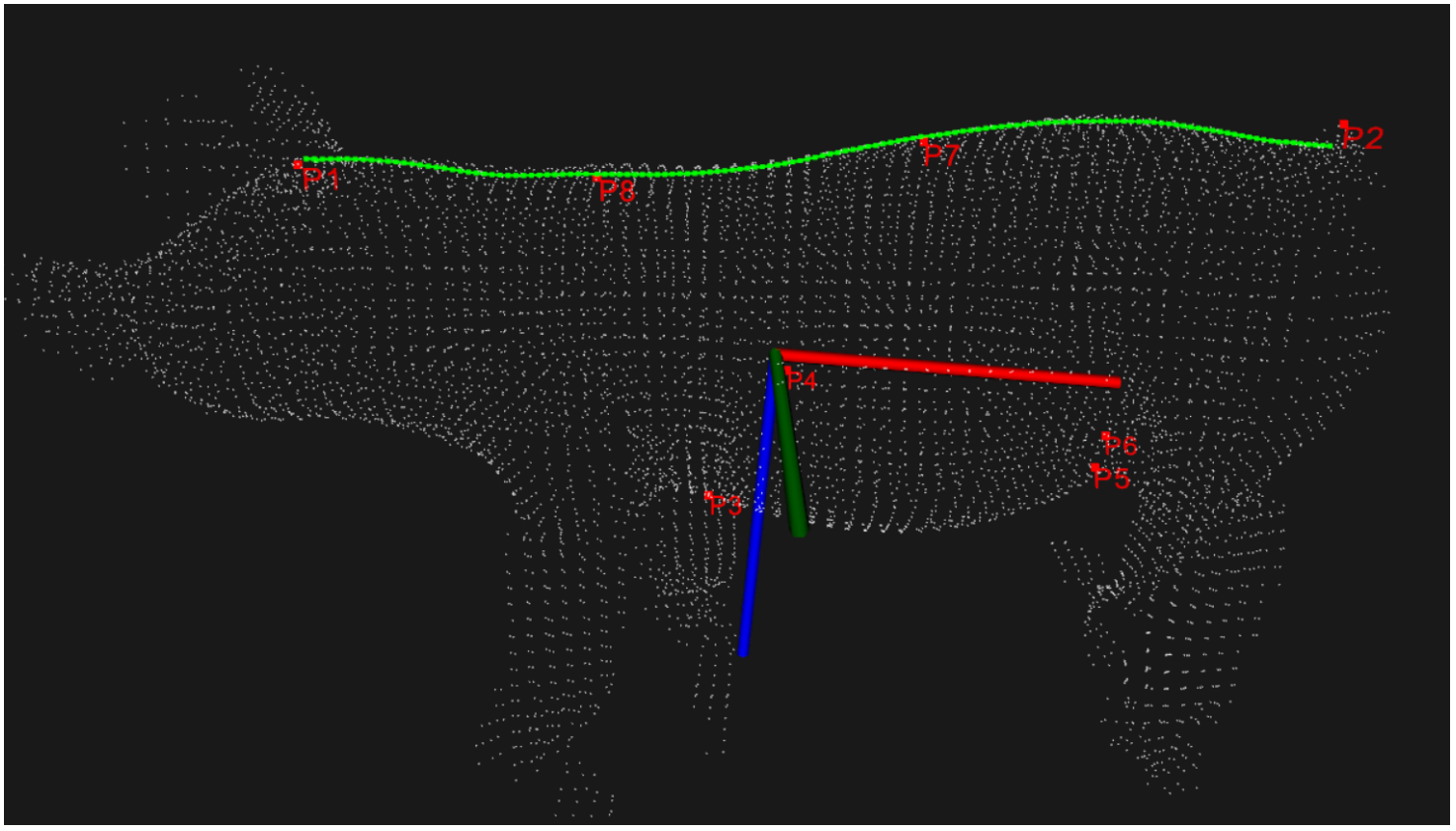
基于 8 个关键点的猪体尺测量方案

关键点选取和定位

如下图所示，首先在猪背部中线左右位置，定位 4 个关键点 P1、P2、P7、P8

- P1：猪的两个耳朵根部的连线的终点，作为猪体长测量的起点；
- P2：猪的臀部和尾巴的连接点，作为猪体长测量的终点；
- P7：腰腹部位置的最高点，用于确定腰围计算的切片点云；
- P8：肩部位置的最高点，用于确定体高；
- P3、P4：前腿根部和躯干连接区域稍靠后一点的位置，用于确定胸围计算的切片；
- P5、P6：后腿根部和躯干连接区域稍靠前一点的位置，用于确定臀围计算的切片；



这里对猪整体点云进行了 PCA 变换，以猪的几何学长轴为 x 轴方向，也就是红色轴（向右为正方向）；第二长轴为 z 轴方向，也就是绿色轴（向下为正方向）；第三长轴为 y 轴方向，也就是蓝色轴（朝向屏幕外为正方向）；

PCA 变换实现代码：这一段是为了将猪的几何学形态方向长、高、宽和三维坐标系下的坐标轴对齐，方便进行体高的测量；主要使用 PCL 中的 PCA 类，

```
Eigen::Affine3f pca_transform(PointCloudT::Ptr cloud_in, PointCloudT::Ptr& cloud_out, const std::vector<PointT>& keypoints_for_orientation)
{
	pcl::PCA<PointT> pca;	// 初始化PCA类
	pca.setInputCloud(cloud_in);	// 设置输入点云

	Eigen::Vector4f centroid = pca.getMean();	// 获取输入点云的均值，作为坐标原点
	Eigen::Matrix3f eigenvectors = pca.getEigenVectors();	// 获取特征向量，也就是点云的3个主方向

	// --- 手动调整坐标轴方向 ---
	// 假设:一般来说，最大特征值对应的特征向量是长度方向、其次是高度、然后是宽度
	// - 第一个主成分 (col 0) 是长度方向 -> new X
	// - 第二个主成分 (col 1) 是高度方向 -> new Z // 初始假设
	// - 第三个主成分 (col 2) 是宽度方向 -> new Y

	Eigen::Vector3f new_x = eigenvectors.col(0);
	Eigen::Vector3f new_z = eigenvectors.col(1); // 暂时假设高度是第二个主成分

	// a. 使用P1->P2向量纠正X轴（长度）的方向
	if (keypoints_for_orientation.size() >= 2) {
		const PointT& p1 = keypoints_for_orientation[0];	// 获取P1点坐标
		const PointT& p2 = keypoints_for_orientation[1];	// 获取P2点坐标
		Eigen::Vector3f p1_p2_vec(p2.x - p1.x, p2.y - p1.y, p2.z - p1.z);
```

```
            p1_p2_vec.normalize();  // 获取P1->P2方向的单位向量

            // 如果点积为负，说明PCA得到的X轴方向与P1->P2方向相反，需要翻转
            if (new_x.dot(p1_p2_vec) < 0) {
                  new_x = -new_x;
                  std::cout << "PCA X-axis was flipped based on keypoints." << std::endl;
            }
      }

      // b. 使用原始Z轴(0,0,1)纠正新Z轴（高度）的方向，使其朝上（这里犯的错误是AI默认原始Z轴是向上的）
      // 假设原始扫描时，Z轴代表“上”（这里注意，之所以之前调整z轴向上没有生效，就是因为原始z轴就是向下的，所以变换方向的if条件应该写成它们的向量乘积>0，也就是方向相同，才需要反向）
      if (new_z.dot(Eigen::Vector3f::UnitZ()) < 0) {
            new_z = -new_z;
            std::cout << "PCA Z-axis was flipped to point upwards (attempted)." << std::endl;
      }

      // c. 使用叉乘计算Y轴，确保是右手坐标系
      Eigen::Vector3f new_y = new_z.cross(new_x);
      new_y.normalize();

      // 重新校正Z轴，确保三轴严格正交（注意：这步会覆盖步骤b的校正！）
      new_z = new_x.cross(new_y);
      new_z.normalize();
      std::cout << "PCA Z-axis recalculated via cross product." << std::endl;


      // 构建修正后的旋转矩阵，新的XYZ轴的方向向量就是PCA变换的旋转矩阵
      Eigen::Matrix3f rotation_corrected;
```

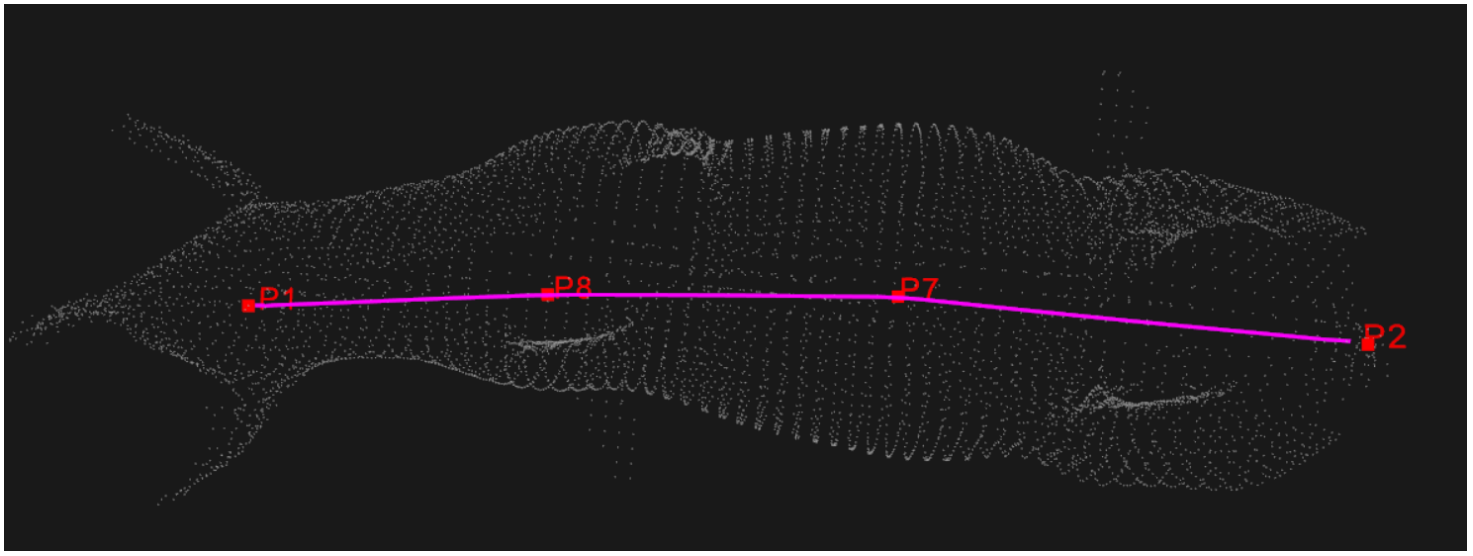
```
        rotation_corrected.col(0) = new_x;
        rotation_corrected.col(1) = new_y;
        rotation_corrected.col(2) = new_z;


        // 构建最终的变换矩阵
        Eigen::Affine3f transform = Eigen::Affine3f::Identity();

        // 1. 设置旋转部分：我们希望将猪的坐标系(new_x, new_y, new_z)旋转到世界坐标系(X,Y,Z)，所以用转置
        transform.rotate(rotation_corrected.transpose());

        // 2. 设置平移部分：必须使用旋转后的质心来计算平移量
        transform.translation() = -transform.rotation() * centroid.head<3>();

        // 应用变换
        pcl::transformPointCloud(*cloud_in, *cloud_out, transform);

        return transform;   // 返回4*4的变换矩阵
    }
```

体长测量

体尺测量可视化如下图所示，测量方案为连接关键点 P1P8、P8P7、P7P2 线段，然后在 P1P8、P8P7、P7P2 上每隔固定长度比如 10mm 采样一个点作为引导点，确保体尺测量的这条线的方向大致沿着 P1-> P8-> P7-> P2 这条路径，然后在每个引导点的邻域内搜索选择 **猪背部 z 值最小的点** 作为采样点，这样确保采样点都在猪背部贴着背部点云前进，更符合测量经验；

得到所有猪背部表面的采样点之后，使用滑动窗口平均法将他们平滑的连接起来，就得到了体长测量线，也就是下图紫色的这条线，然后计算所有采样点的距离之和，就是体长；

同时，这段体长测量线，也作为衡量猪整体躯干姿态的骨架线，用于确定后续体宽、胸围、腰围、臀围计算时的切片方向。



代码实现：这一段 `compute_surface_back_skeleton` 主要是实现了根据背部关键点P1、P8、P7、P2，在猪背部点云上确定一条经过这些关键点且贴合猪背部的能够反映猪躯干姿态的骨架。

```
    /**
     * @brief （思路1-规则A 实现版）计算贴合背部表面的骨架点云
     * @details 沿着P1->P8->P7->P2路径，通过在路径点邻域内搜索Z值最小的点，
     * 限制尾部区域Z值，并平滑连接。
     * @param transformed_cloud PCA变换后的完整猪点云
     * @param all_keypoints 包含P1,P8,P7,P2的变换后关键点向量
     * @param step_mm 沿着路径每隔多少毫米采样一个引导点
     * @param search_radius_mm 在引导点周围多大半径内搜索邻居点
     * @param smoothing_window_size 滑动平均窗口大小（奇数）
     * @return 平滑且贴合背部的骨架点云
     */
PointCloudT::Ptr compute_surface_back_skeleton(
        const PointCloudT::Ptr& transformed_cloud,          // PCA变换后的完整猪点云
        const std::vector<PointT>& all_keypoints,           // 包含P1,P8,P7,P2的变换后关键点向量
        float step_mm,                      // 沿着路径每隔多少毫米采样一个引导点
```

```
16        float search_radius_mm,       // 在引导点周围多大半径内搜索邻居点
17        int smoothing_window_size) // 滑动平均窗口大小 (奇数)
18    {
19        PointCloudT::Ptr skeleton_cloud(new PointCloudT);
20        if (transformed_cloud->empty() || all_keypoints.size() < 8) {
21              PCL_ERROR("点云为空或关键点不足，无法计算表面骨架。\n");
22              return skeleton_cloud;
23        }
24
25        // --- 0. 准备工作 ---
26        const PointT& p1 = all_keypoints[0];    // 从变换后的关键点坐标中读取确定骨架需要的P1、P8、P7、P2
27        const PointT& p8 = all_keypoints[7];
28        const PointT& p7 = all_keypoints[6];
29        const PointT& p2 = all_keypoints[1];
30
31        std::vector<PointT> path_segments_starts = { p1, p8, p7 };  // 确定三条线段的起点
32        std::vector<PointT> path_segments_ends = { p8, p7, p2 };    // 确定三条线段的终点
33        std::vector<PointT> raw_surface_points; // 存储找到的原始骨架点
34
35        // 创建 KdTree 用于快速邻域搜索
36        pcl::KdTreeFLANN<PointT>::Ptr kdtree(new pcl::KdTreeFLANN<PointT>);
37        kdtree->setInputCloud(transformed_cloud);   // 在输入的PCA变换后的猪点云上创建KdTree加速邻域搜索
38
39         // --- 1. 分段提取表面 Z 值最小的邻近点 --
40         for (size_t seg = 0; seg < path_segments_starts.size(); ++seg) {  // 分为P1->P8, P8->P7, P7->P2三段
41              const PointT& start_pt = path_segments_starts[seg]; // 当前分割段的路径起点
42              const PointT& end_pt = path_segments_ends[seg];     // 当前分割段的路径终点
```

```
43
44              Eigen::Vector3f start_vec = start_pt.getVector3fMap();
45              Eigen::Vector3f end_vec = end_pt.getVector3fMap();
46              Eigen::Vector3f segment_dir = (end_vec - start_vec);     // 当前分割段线段的方向向量
47              float segment_length = segment_dir.norm();       // 当前分割段线段的长度
48              if (segment_length < 1e-3) continue; // 跳过长度为零的段
49              segment_dir.normalize();     // 当前分割段的单位方向向量
50
51              int num_steps = static_cast<int>(segment_length / step_mm); // 每隔10mm采样一个点，总采样点为“分割段长度/步长”
52              if (num_steps < 1) num_steps = 1; // 至少走一步
53
54              // --- 处理起点 ---
55              if (seg == 0) {
56                  // 对于起点P1，也搜索其邻域并取Z最小值，使其从一开始就贴合表面
57                  std::vector<int> pointIdxRadiusSearch;   // 存储邻域点索引
58                  std::vector<float> pointRadiusSquaredDistance;   // 存储邻域点距离平方
59                  PointT start_surface_pt = start_pt; // 默认值
60                  start_surface_pt.z = std::numeric_limits<float>::max();
61                  bool found_start_neighbor = false;
62
63                 // 以起点为中心进行半径搜索
64                  if (kdtree->radiusSearch(start_pt, search_radius_mm, pointIdxRadiusSearch, pointRadiusSquaredDistance) > 0) {
65                      for (int idx : pointIdxRadiusSearch) {  // 遍历搜索点的所有邻域点，找到邻域中z值最小的点
66                          if (transformed_cloud->points[idx].z < start_surface_pt.z) {
```

```
67                                  start_surface_pt = transformed_cloud->points[idx];
68                                  found_start_neighbor = true;
69                              }
70                          }
71                      }
72                      // 如果在P1邻域找到点，使用P1邻域内Z最小的点；否则使用原始P1
73                      raw_surface_points.push_back(found_start_neighbor ? start_surface_pt : start_pt);
74                  }
75
76                  // --- 沿路径采样引导点并搜索，沿着分割段线段的方向逐步前进，进行邻域点搜索，找寻贴合背部表面的点 ---
77                  for (int i = 1; i <= num_steps; ++i) {
78                      float current_dist = i * step_mm;          // 当前距离起点的距离
79                      PointT current_center; // 当前引导点
80                      Eigen::Vector3f current_center_vec;
81                      // 判断是否为最后一步或接近终点
82                      bool is_last_step = (i == num_steps);
83                      if (current_dist >= segment_length - step_mm / 2.0) {    // 如果当前点距离终点不到半个步长，则直接使用终点
84                          current_center = end_pt; // 直接使用段终点作为引导点
85                          current_center_vec = end_vec;
86                          is_last_step = true; // 标记为最后一步
87                          i = num_steps; // 确保循环在此步后结束
88                      }
89                      else {   // 如果距离终点还比较远，超过半个步长，则继续前进一个步长，也就是10mm
90                          current_center_vec = start_vec + current_dist * segment_dir;
91                          current_center.x = current_center_vec.x();
92                          current_center.y = current_center_vec.y();
```

```
93                      current_center.z = current_center_vec.z(); // Z坐标可以暂时用引导点的
94                  }
95
96
97                  // --- 核心修改：在引导点邻域内搜索Z最小点 ---
98                  std::vector<int> pointIdxRadiusSearch;
99                  std::vector<float> pointRadiusSquaredDistance;
100                 PointT min_z_neighbor_pt = current_center; // 初始化
101                 min_z_neighbor_pt.z = std::numeric_limits<float>::max(); // 初始化为最大Z值
102                 bool found_neighbor = false;
103
104                 // 执行半径搜索
105                 if (kdtree->radiusSearch(current_center, search_radius_mm, pointIdxRadiusSearch, pointRadiusSquaredDistance) > 0) {
106                     for (int idx : pointIdxRadiusSearch) {
107                         // 在邻居中寻找Z最小的点（假设Z越小越接近背部）
108                         if (transformed_cloud->points[idx].z < min_z_neighbor_pt.z) {
109                             min_z_neighbor_pt = transformed_cloud->points[idx];
110                             found_neighbor = true;
111                         }
112                     }
113                 }
114                 else {
115                     // 如果半径内没有点，可以尝试找最近的1个点作为备选
116                     std::vector<int> pointIdxNKNSearch(1);
117                     std::vector<float> pointNKNSquaredDistance(1);
118                     if (kdtree->nearestKSearch(current_center, 1, pointIdxNKNSearch, pointNKNSquaredDistance) > 0) {
119                         min_z_neighbor_pt = transformed_cloud->points[pointIdxNKNSearch[0]];
```

```
120                             found_neighbor = true;
121                             PCL_WARN("Radius search found no points near
    guide point, using 1-NN instead.");
122                         }
123                         else {
124                             PCL_WARN("Could not find any neighbors near
    guide point (%f, %f, %f)", current_center.x, current_center.y,
    current_center.z);
125                         }
126                     }
127                     // --- 修改结束 ---
128 
129                     // 如果成功找到邻域内的Z最小点，则添加到原始骨架点列表
130                     if (found_neighbor) {
131                         // // 可选策略1：直接使用找到的Z最小邻居点
132                         //
    raw_surface_points.push_back(min_z_neighbor_pt);
133 
134                         // 可选策略2：使用引导点的X,Y和找到的最小Z（可能使X,Y
    更平滑）
135                         PointT surface_pt = current_center;
136                         surface_pt.z = min_z_neighbor_pt.z;
137                         raw_surface_points.push_back(surface_pt);
138 
139                     }
140                     else if (is_last_step) {
141                         // 如果是最后一步且没找到邻居（不太可能），强制添加原
    始段终点
142                         raw_surface_points.push_back(end_pt);
143                     }
144 
145                     // (这段确保终点添加的逻辑可以简化或移除，因为最后一步会处
    理end_pt)
```

```
                // if (is_last_step && (raw_surface_points.empty() ||
(end_pt.getVector3fMap() -
raw_surface_points.back().getVector3fMap()).norm() > 1e-3))
                // {
                //         // 检查是否已添加过近似终点
                // }
            }
        }

        // --- 2. 去除可能的重复点 (不变) ---
        if (raw_surface_points.size() < 2) {
            PCL_ERROR("提取到的原始骨架点不足2个。\n");
            return skeleton_cloud; // 返回空点云
        }
        std::vector<PointT> unique_raw_points;
        unique_raw_points.push_back(raw_surface_points[0]);
        for (size_t i = 1; i < raw_surface_points.size(); ++i) {
            // 增加去重阈值，避免过于密集的点
            if ((raw_surface_points[i].getVector3fMap() -
unique_raw_points.back().getVector3fMap()).norm() > step_mm * 0.1) { // 例
如，距离大于步长的10%才添加
                unique_raw_points.push_back(raw_surface_points[i]);
            }
        }
        std::cout << "Raw points collected: " << raw_surface_points.size()
<< ", Unique points: " << unique_raw_points.size() << std::endl;


        // --- 3. (修正版) 限制尾部区域 Z 值 ---
        if (unique_raw_points.size() >= 5) { // 需要足够点才有意义
            float tail_influence_dist_sq = 80.0f * 80.0f; // 尾部影响区域
80mm (平方)
            int reference_idx = -1;
            Eigen::Vector3f p2_vec = p2.getVector3fMap();
```

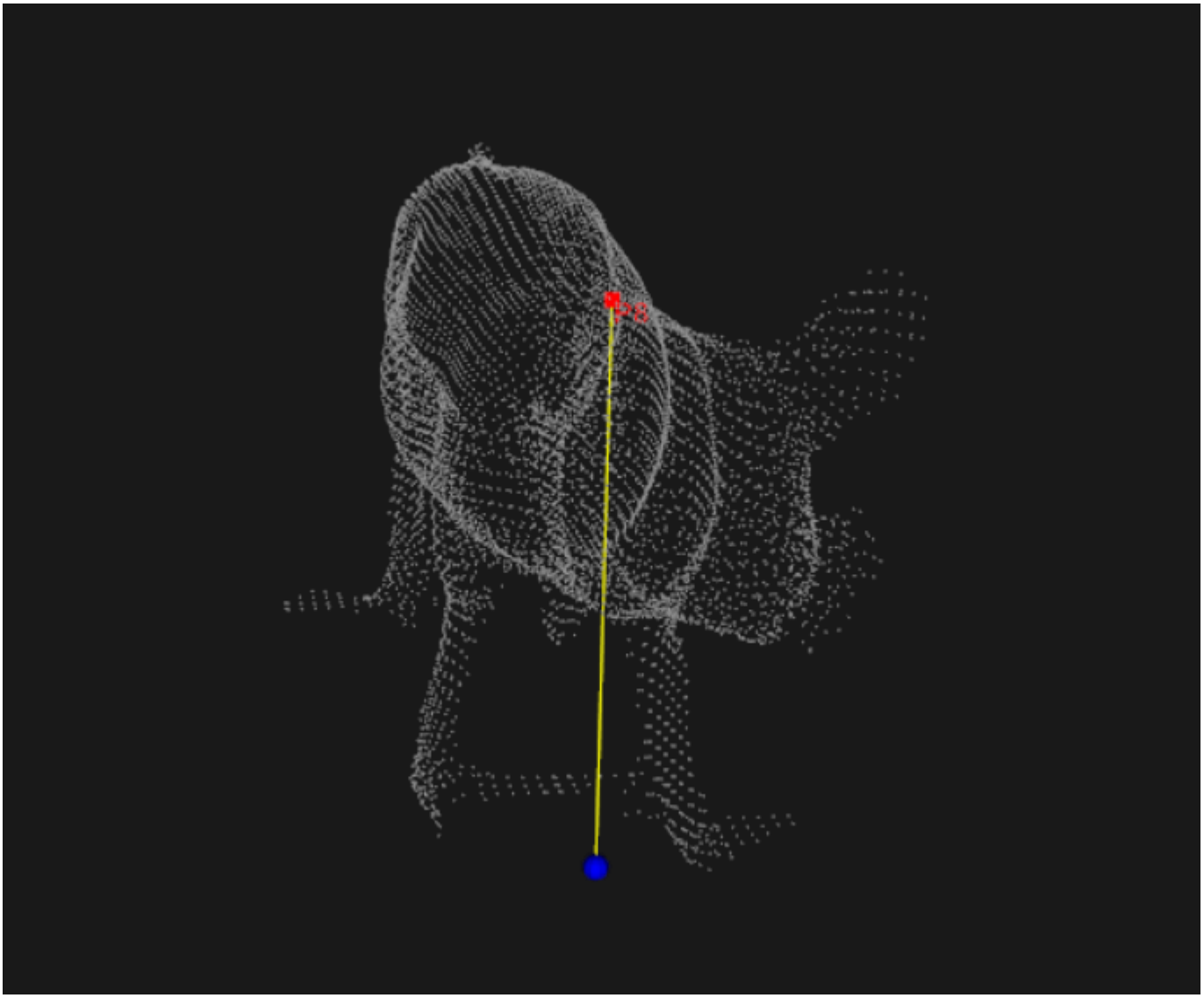
```
174
175             // 找到影响区域外的最后一个点的索引 (不变)
176             for (int i = unique_raw_points.size() - 1; i >= 0; --i) {
177                 if ((unique_raw_points[i].getVector3fMap() -
    p2_vec).squaredNorm() > tail_influence_dist_sq) {
178                     reference_idx = i;
179                     break;
180                 }
181             }
182
183             if (reference_idx != -1 && reference_idx <
    unique_raw_points.size() - 1) {
184                 float reference_z = unique_raw_points[reference_idx].z;
185                 // 允许的最大Z值偏差 (尾部点Z值比背部点Z值小多少以内是可接
    受的)
186                 float max_allowed_z_deviation = 25.0f;
187
188                 std::cout << "Applying tail Z capping starting from
    index " << (reference_idx + 1) << " with reference Z: " << reference_z <<
    std::endl;
189
190                 // 遍历影响区域内的点
191                 for (size_t i = reference_idx + 1; i <
    unique_raw_points.size(); ++i) {
192                     // *** 核心修正：检查Z值是否过小 (翘起) ***
193                     // 如果当前点的Z值比参考Z值 小 超过了允许的偏差 (即点
    过于靠上/翘起)
194                     if (unique_raw_points[i].z < reference_z -
    max_allowed_z_deviation) {
195                         std::cout << "  - Capping point " << i << "
    from Z=" << unique_raw_points[i].z << " to Z=" << reference_z <<
    std::endl;
196                         // 将其Z值强制设为参考Z值 (将其“拉低”到背部水
    平)
```

```
197                                 unique_raw_points[i].z = reference_z;
198                             }
199                             // *** 修正结束 ***
200                         }
201                     }
202                     else {
203                         std::cout << "Tail influence zone covers all points or no reference found, skipping Z capping." << std::endl;
204                     }
205                 }
206 
207                 // --- 4. 平滑处理 (不变) ---
208                 if (unique_raw_points.size() < 3) {
209                     PCL_WARN("去重和尾部限制后点数不足3个，无法进行平滑。将返回当前点。\n");
210                     skeleton_cloud->points.assign(unique_raw_points.begin(), unique_raw_points.end());
211                     skeleton_cloud->width = skeleton_cloud->size();
212                     skeleton_cloud->height = 1;
213                     skeleton_cloud->is_dense = true;
214                     return skeleton_cloud;
215                 }
216                 std::vector<float> x_coords, y_raw, z_raw;
217                 for (const auto& pt : unique_raw_points) {
218                     x_coords.push_back(pt.x);
219                     y_raw.push_back(pt.y);
220                     z_raw.push_back(pt.z);
221                 }
222                 // 确保 smooth_data_moving_average 定义在此函数之前
223                 std::vector<float> y_smooth = smooth_data_moving_average(y_raw, smoothing_window_size);
224                 std::vector<float> z_smooth = smooth_data_moving_average(z_raw, smoothing_window_size);
225 
```

```
226         // --- 5. 构建最终骨架 (不变) ---
227         for (size_t i = 0; i < x_coords.size(); ++i) {
228               PointT skel_pt;
229               skel_pt.x = x_coords[i];
230               skel_pt.y = y_smooth[i];
231               skel_pt.z = z_smooth[i];
232               skeleton_cloud->push_back(skel_pt);
233         }
234
235         // --- 6. 确保骨架顺序与P1->P2方向一致 (不变) ---
236         if (p1.x < p2.x) {
237               std::sort(skeleton_cloud->points.begin(), skeleton_cloud-
      >points.end(), [](const PointT& a, const PointT& b) { return a.x < b.x;
      });
238         }
239         else {
240               std::sort(skeleton_cloud->points.begin(), skeleton_cloud-
      >points.end(), [](const PointT& a, const PointT& b) { return a.x > b.x;
      });
241         }
242
243         skeleton_cloud->width = skeleton_cloud->size();
244         skeleton_cloud->height = 1;
245         skeleton_cloud->is_dense = true;
246         return skeleton_cloud;
247   }
```

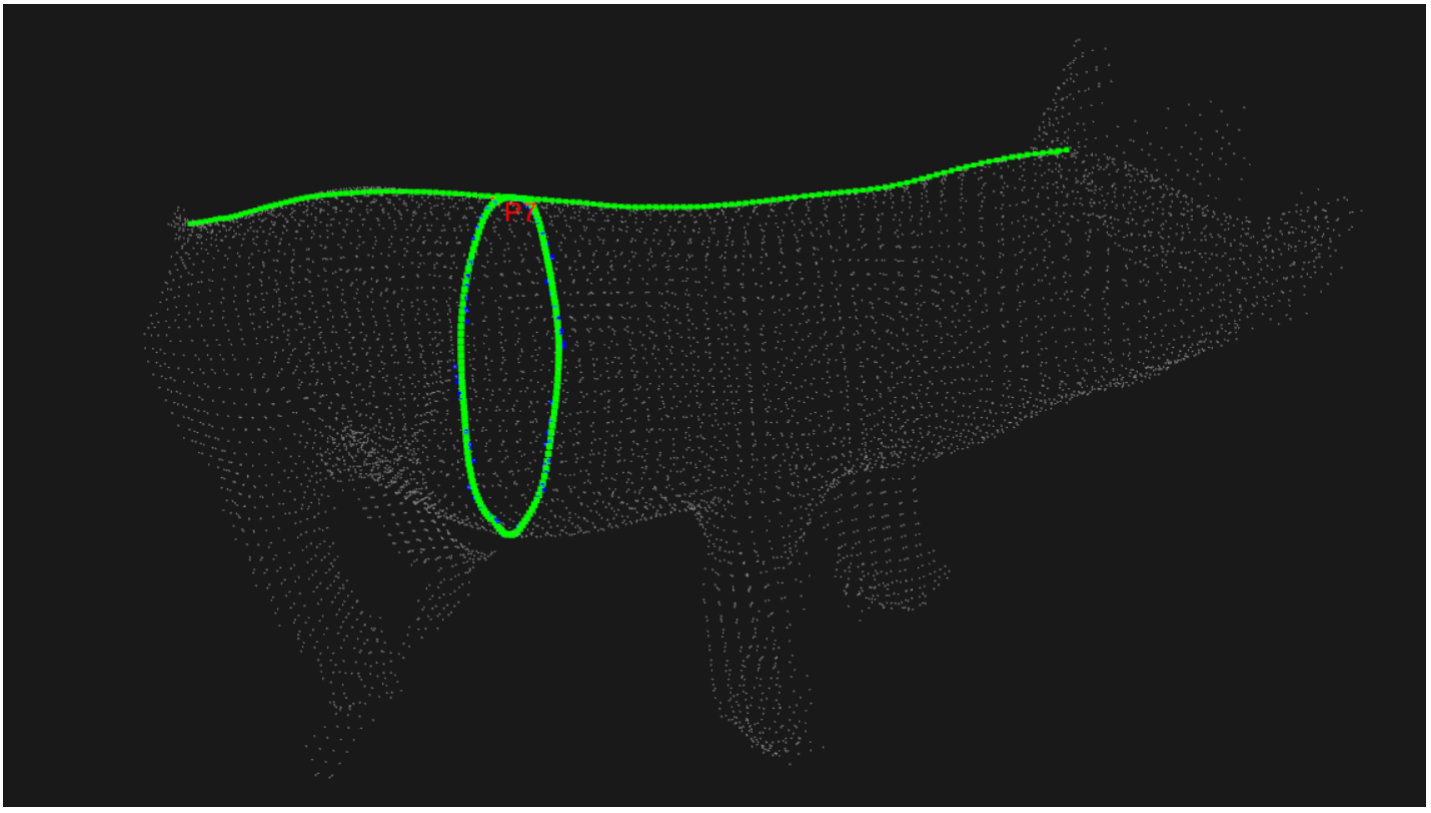
体高测量

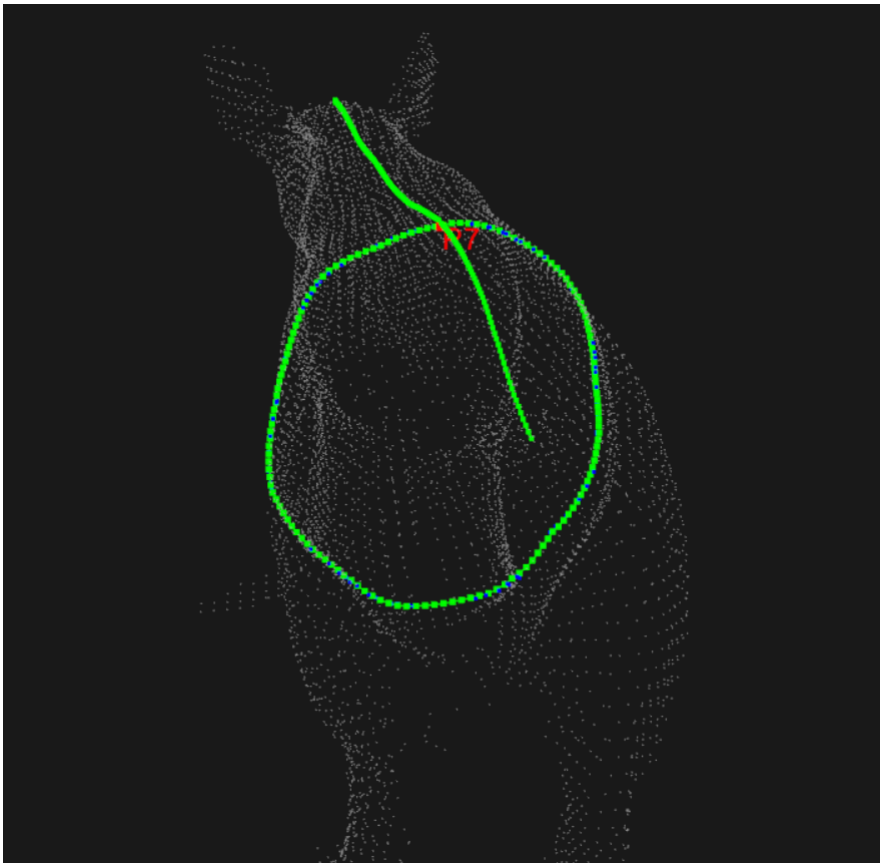
如下图所示，体高测量主要依赖 P8 关键字，我们以 P8 关键点的 z 坐标和猪点云中 z 坐标差值的最大值，作为猪的体高；



腰围测量

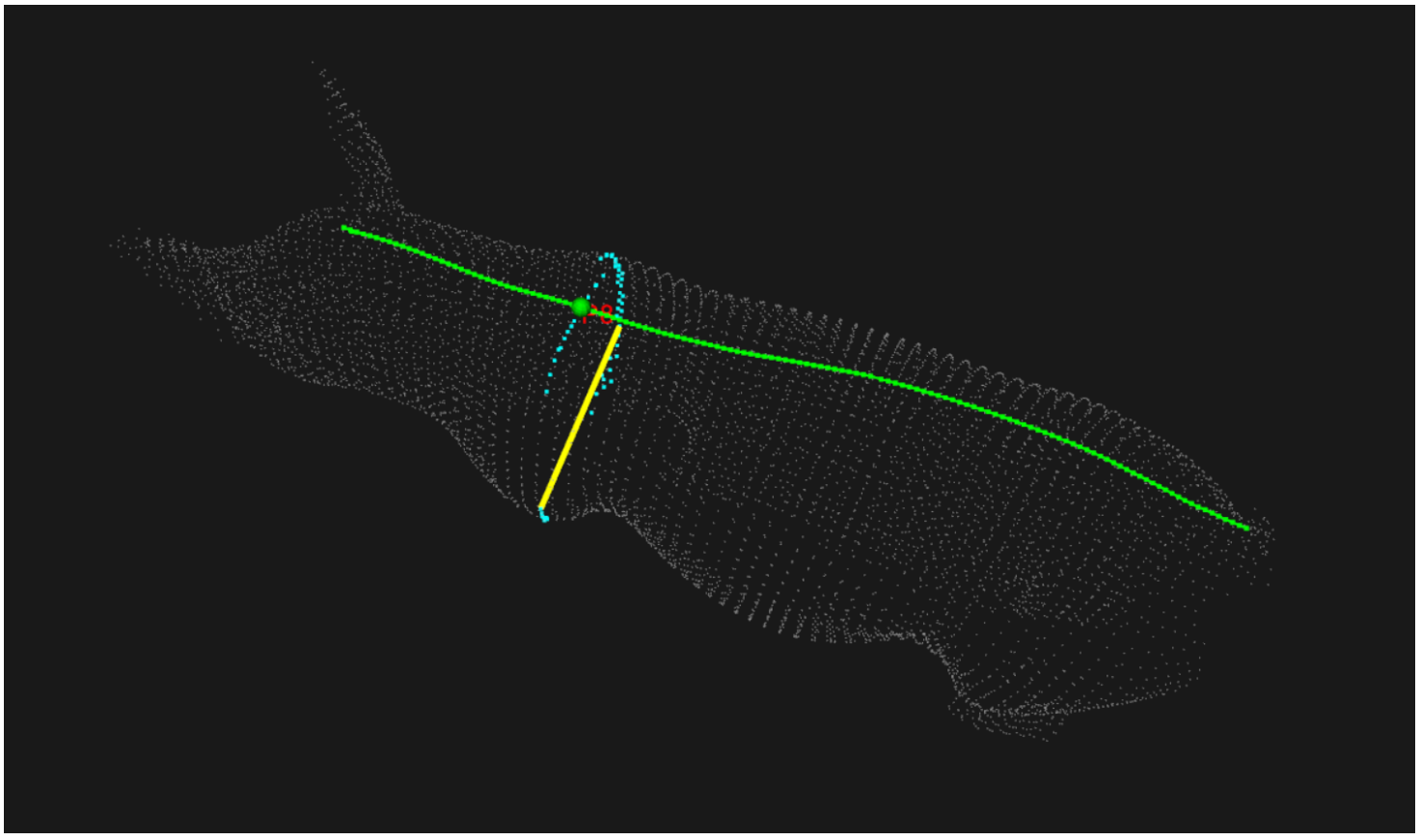
腰围测量主要依赖于关键点 P7，这里采取的测量方案是过点 P7，作一个垂直于 P7 附近的骨架线的切片，然后将这个切片内的点映射到 (r, θ) 的极坐标系下，然后间隔固定角度采样（使用 Catmull-Rom 样条插值处理缺失值和加强平滑性），计算所有采样点连接的距离之和，就是腰围；


P7


P7

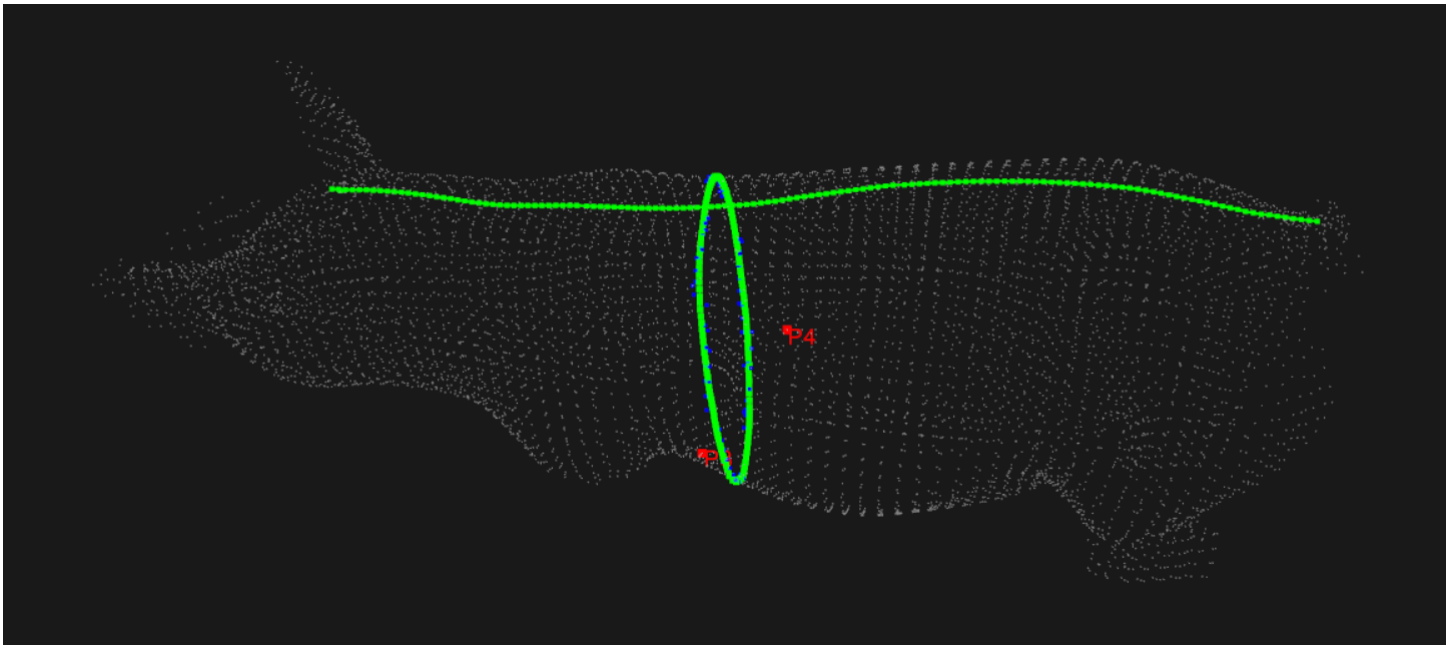
体宽测量

体宽测量也主要依赖于关键点 P8，方法为过 P8 作一个垂直于骨架线的切片，然后找到点 P8 在骨架线上的切线方向，然后将这个切线方向和向下的 z 轴方向作叉乘，就得到体宽的方向 V_width，然后遍历切片中的点，计算切片上的点和 P8 的连线在体宽方向上的投影，这个投影分为两部分，如果切片上的点和 P8 连线和 V_width 方向相同（夹角小于 90°），则投影为正，如果切片上的点和 P8 连线和 V_width 方向相反（夹角大于 90°），则投影为负，所以投影值最大和最小的两个投影是两个相反方向的沿体宽方向长度最大的投影，他们的绝对值之和就是体宽长度；



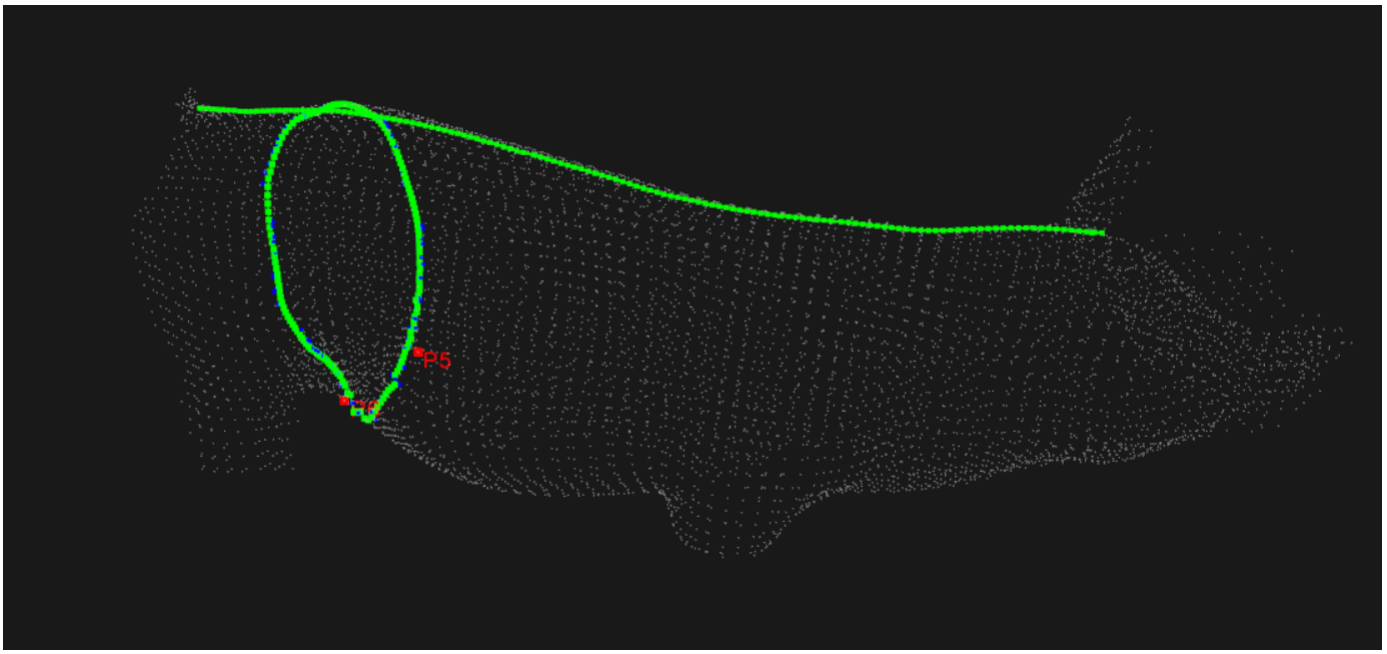
胸围测量

方法和腰围差不多，过 P3、P4 的中点，作一个垂直于骨架线的切片，将切片内的点映射到 (r, θ) 的极坐标系下，然后间隔固定角度采样测量点，计算测量点连接距离之和，就是胸围；



臀围测量

方法和腰围差不多，过 P5、P6 的中点，作一个垂直于骨架线的切片，将切片内的点映射到 (r, θ) 的极坐标系下，然后间隔固定角度采样测量点，计算测量点连接距离之和，就是臀围；



基于6个关键点的猪体尺测量方案

关键点选取和定位