IROS2012 Vila Moura, Algarve, Portugal

# PCL :: Segmentation - Planes, Clusters & More

Alexander J B Trevor, Georgia Institute of Technology

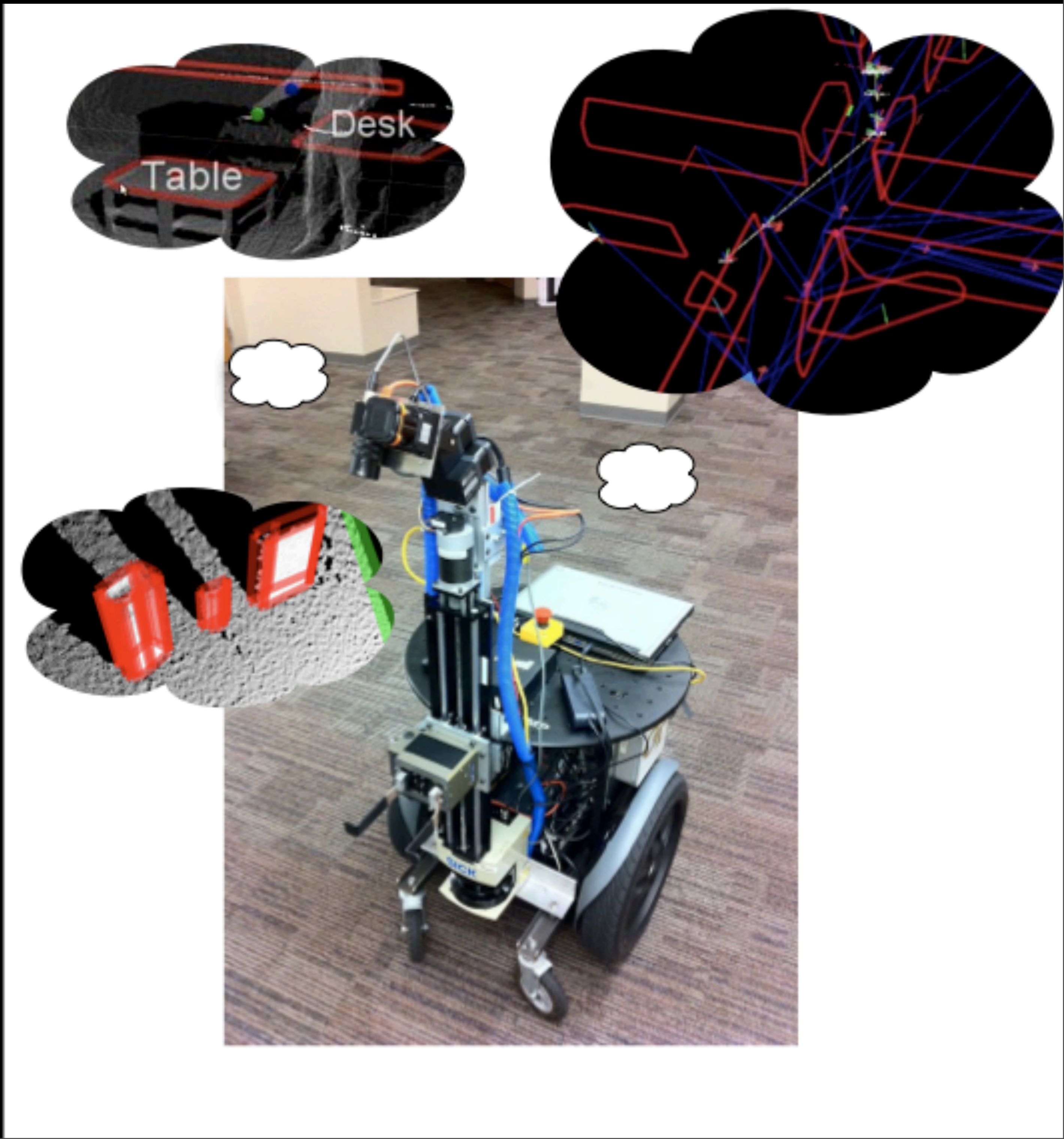
October 13, 2012

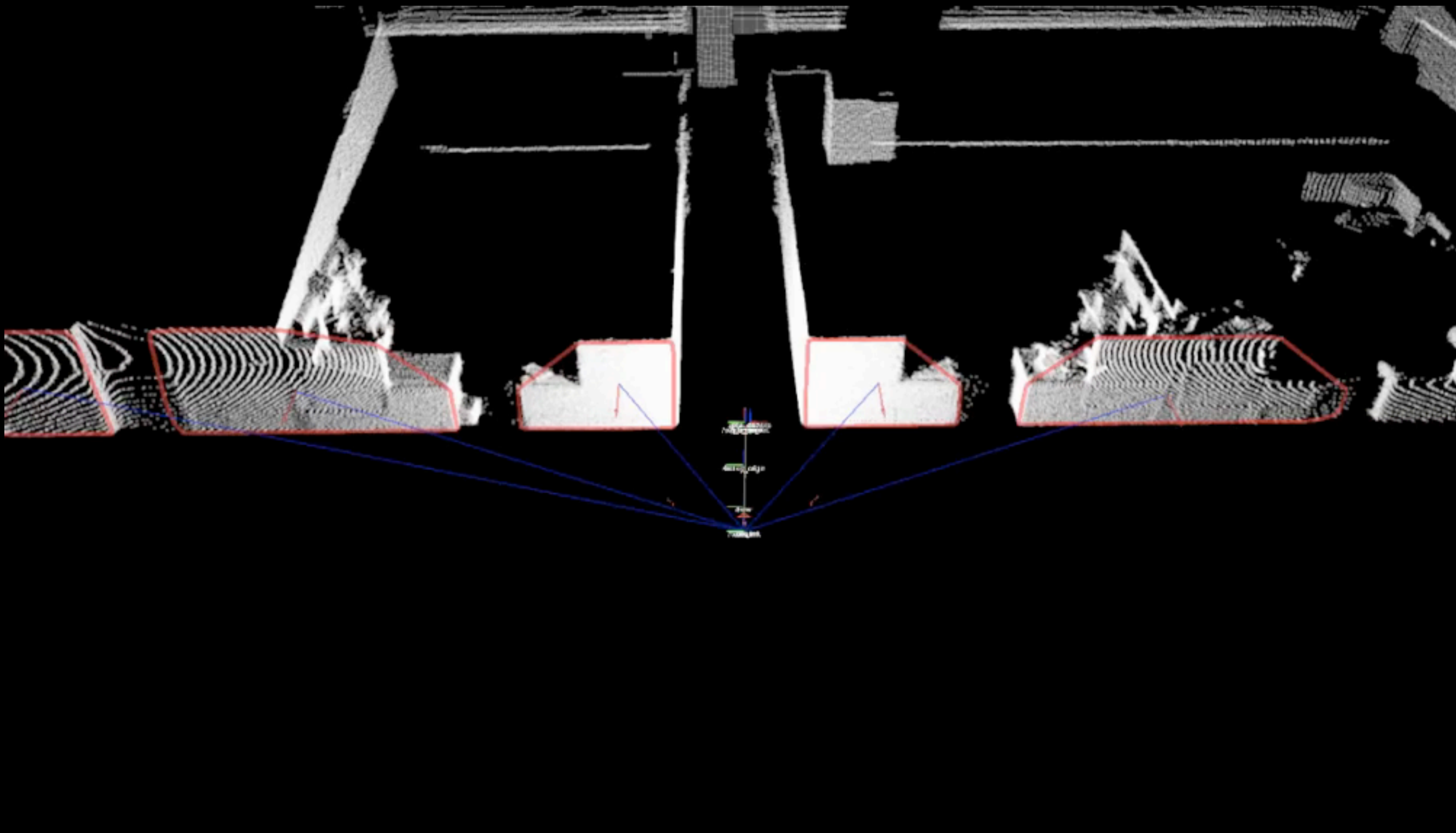
# Motivation

- Segmentation is used for many applications
  - Object Detection & Recognition
  - Mapping
  - Obstacle avoidance
- Make a fast, general approach focused on Kinect data

# My application:
# What should maps look like?

- Let's focus on fetch & carry tasks
- Need to know task-relevant features:
  - Structures: walls, tables, shelves
  - Objects: things we want to manipulate or interact with
  - People: users (not addressed here)

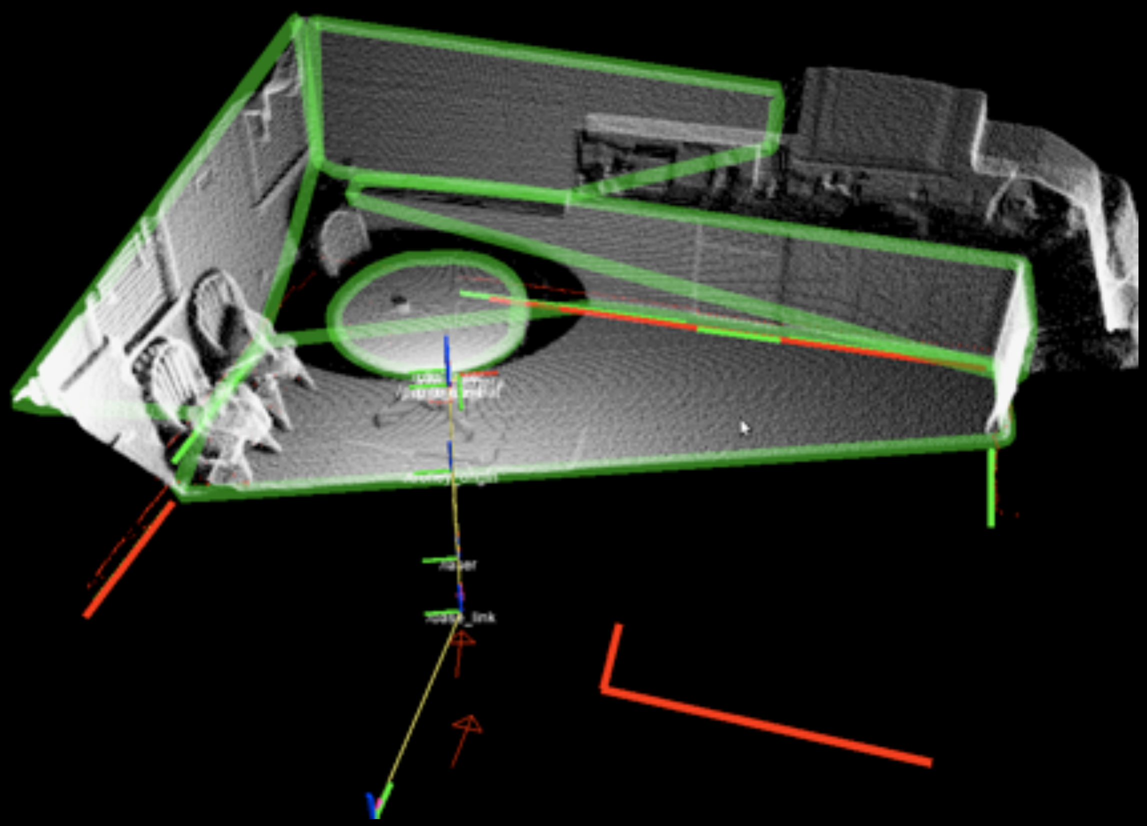




A. J. B. Trevor, J. G. Rogers III, C. Nieto-Granda, H. I. Christensen, "Planar Surface SLAM with 3D and 2D Sensors", ICRA, 2012.

# Limitations of the approach

- RANSAC takes a lot of time, which is okay for scanning lasers, but not for Kinect
- Convex hulls poorly represent the shape of many surfaces
- We can do better if we do smarter processing!



# How can we improve this?

- The previous approach was fast enough for laser data. What if we want to use a Kinect on a robot, or even handheld sensor?
- Convex hulls don't represent shapes well, and alpha shapes can be problematic (choosing alpha)
- So, let's not throw away organized structure, and do smarter processing

# Fast Segmentation of Organized Point Cloud Data

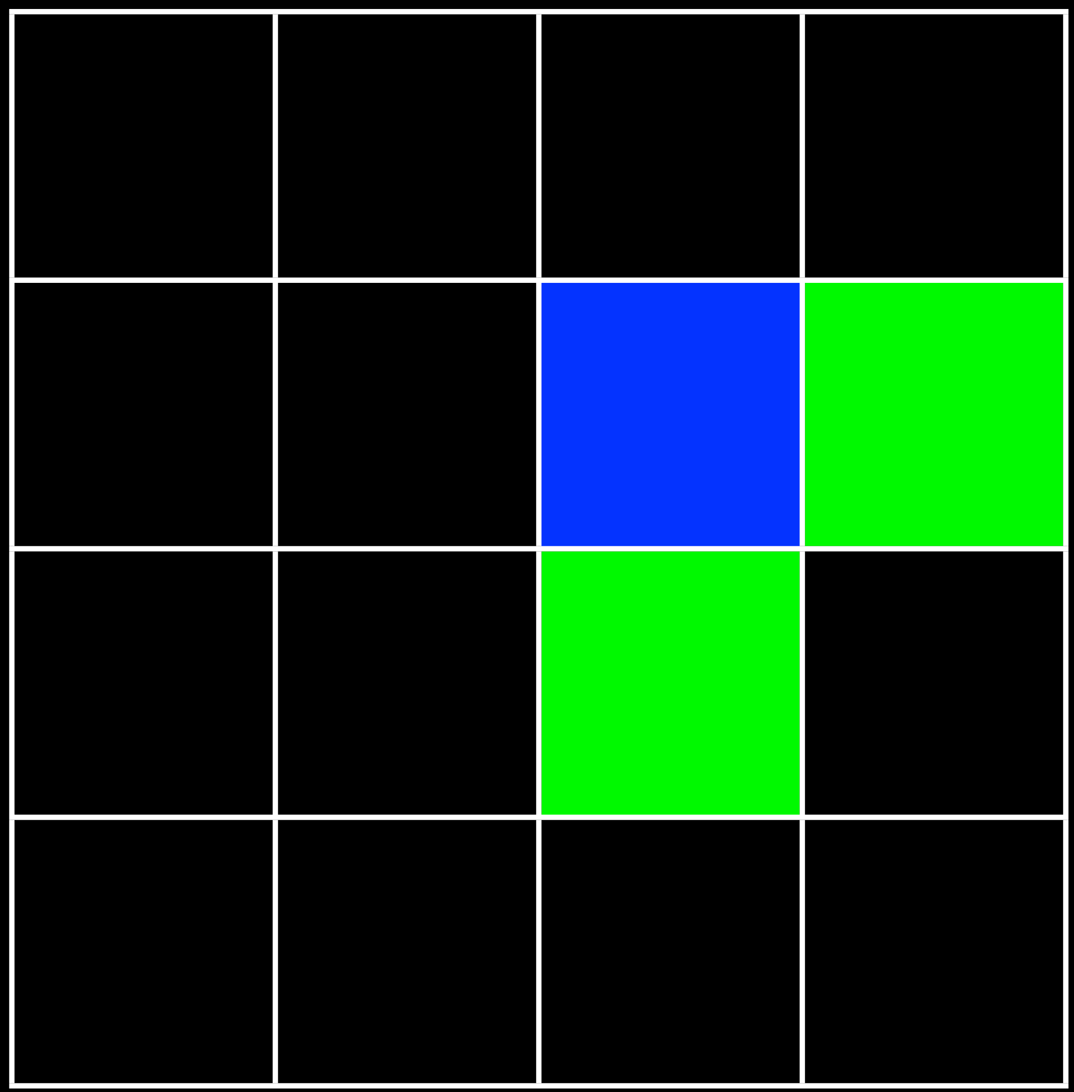
- First step: calculate surface normals
- Use Integral Image Normal Estimation
- Takes about 70 ms per frame (covariance matrix method), or 30-40 ms (simple gradient method)

# Connected Components

- Next, do connected component labeling

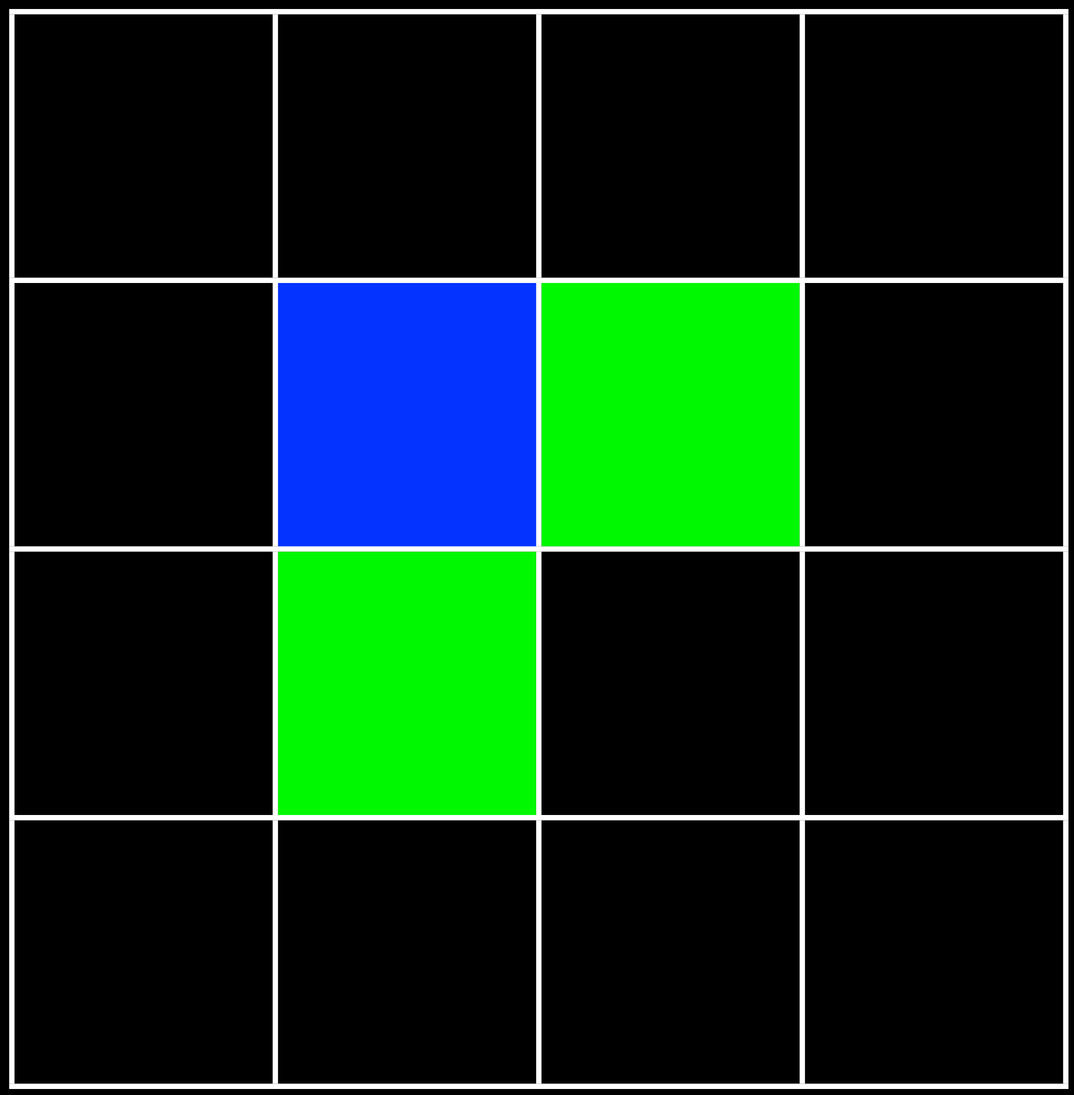
- Implementation allows various different comparisons between neighboring pixels

- Different comparison functions allow different types of segmentation: planes, objects, color blobs, etc



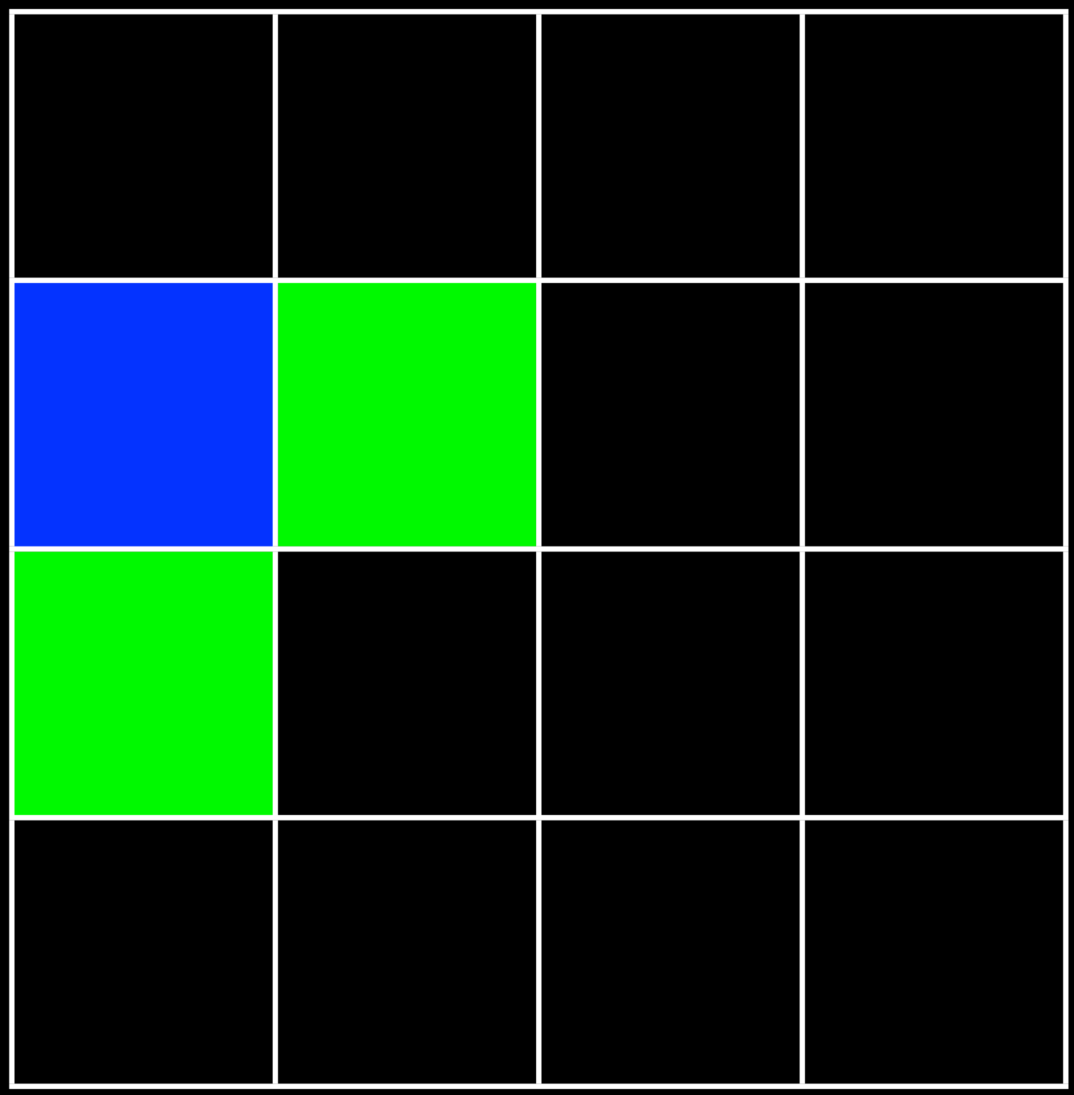
# Connected Components

- Next, do connected component labeling
- Implementation allows various different comparisons between neighboring pixels
- Different comparison functions allow different types of segmentation: planes, objects, color blobs, etc



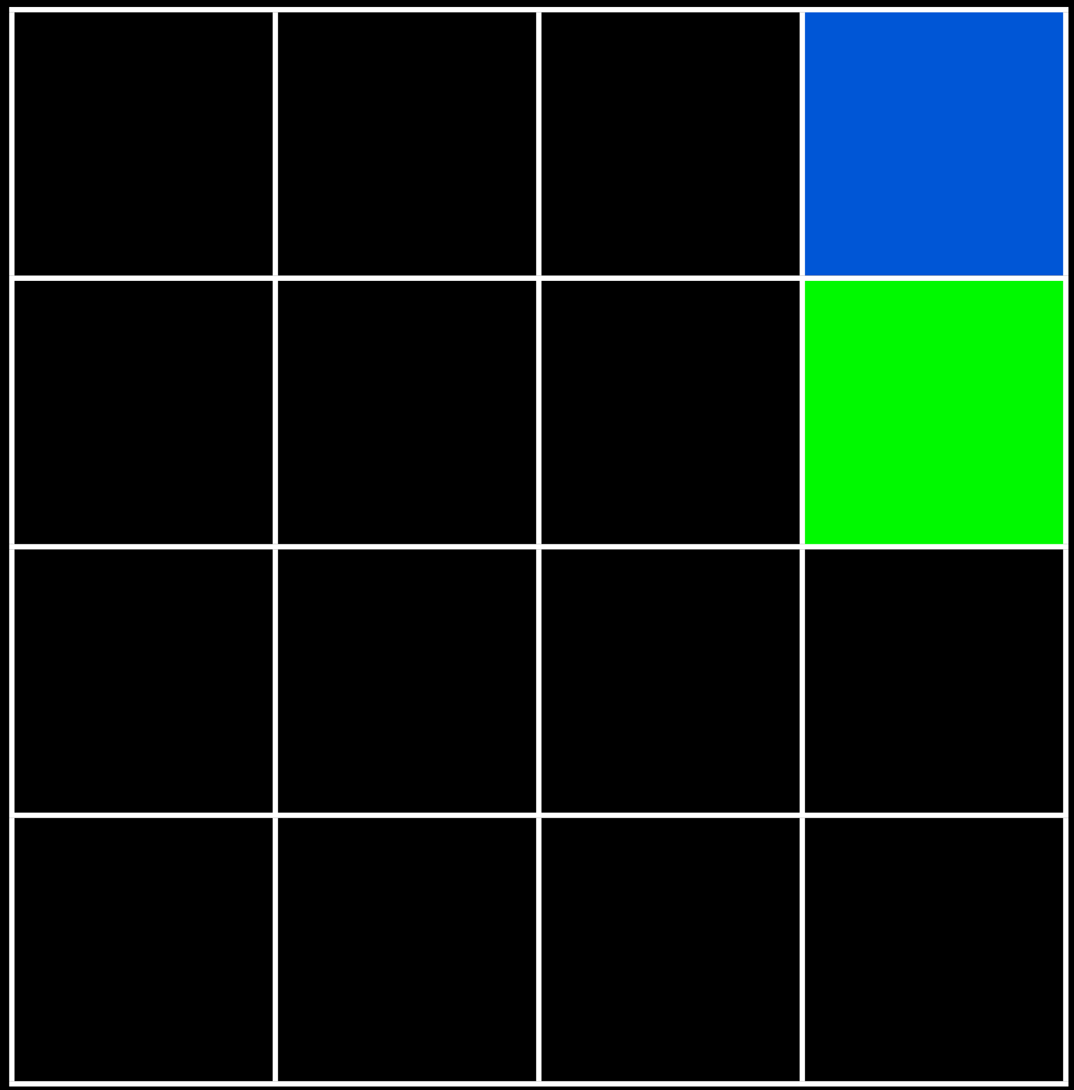
# Connected Components

- Next, do connected component labeling
- Implementation allows various different comparisons between neighboring pixels
- Different comparison functions allow different types of segmentation: planes, objects, color blobs, etc



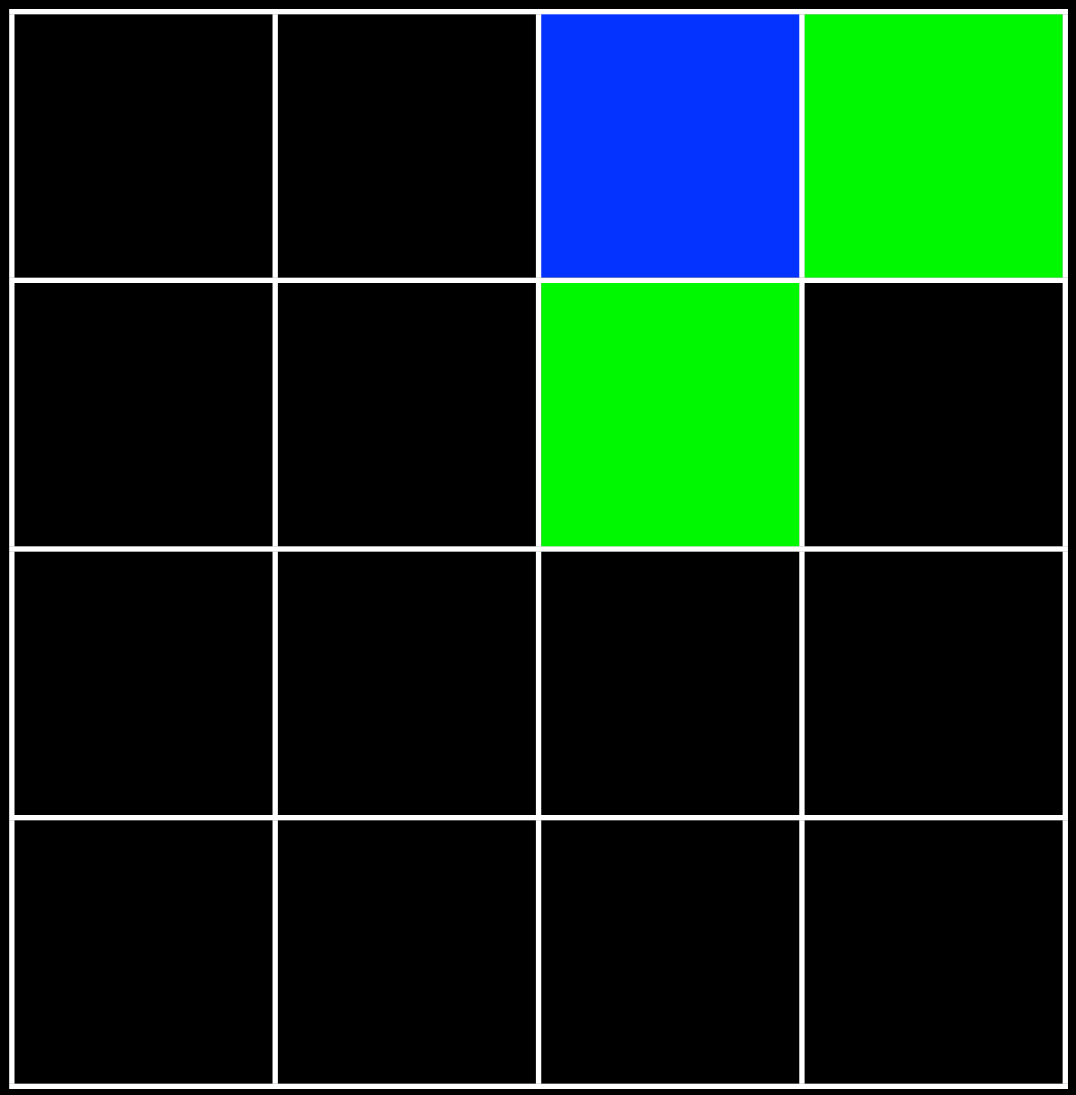
# Connected Components

- Next, do connected component labeling
- Implementation allows various different comparisons between neighboring pixels
- Different comparison functions allow different types of segmentation: planes, objects, color blobs, etc



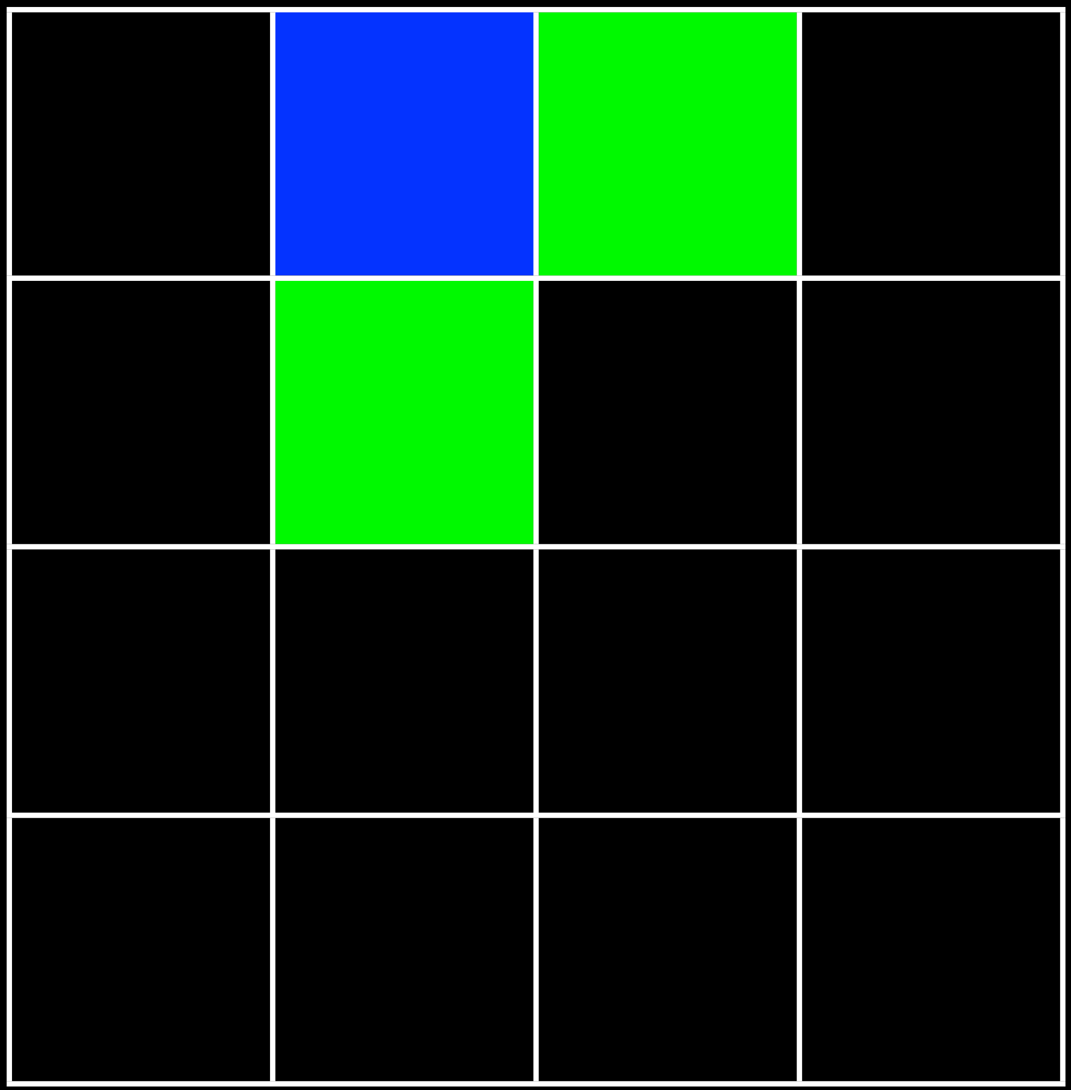
# Connected Components

- Next, do connected component labeling
- Implementation allows various different comparisons between neighboring pixels
- Different comparison functions allow different types of segmentation: planes, objects, color blobs, etc



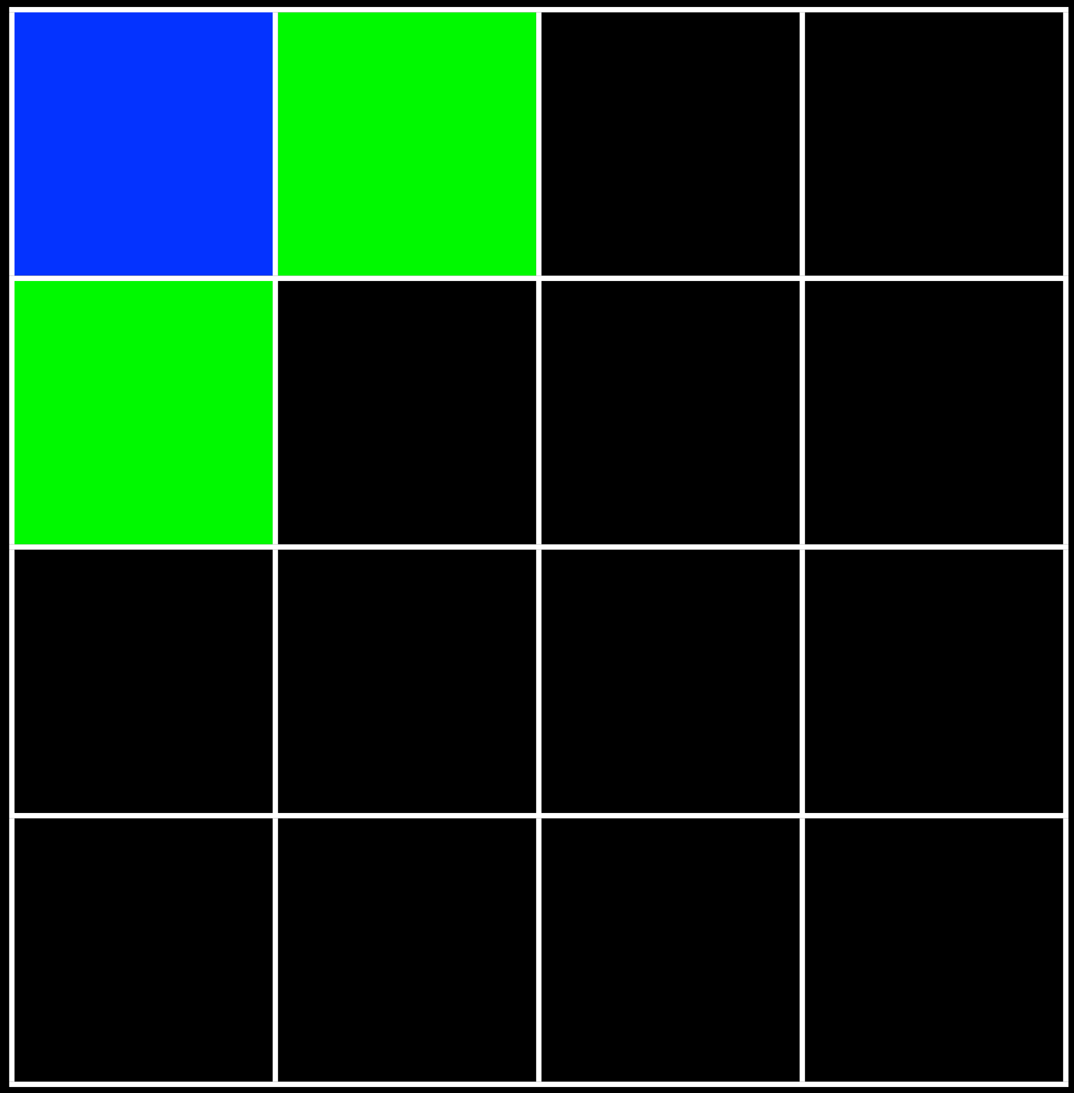
# Connected Components

- Next, do connected component labeling
- Implementation allows various different comparisons between neighboring pixels
- Different comparison functions allow different types of segmentation: planes, objects, color blobs, etc



# Connected Components

- Next, do connected component labeling
- Implementation allows various different comparisons between neighboring pixels
- Different comparison functions allow different types of segmentation: planes, objects, color blobs, etc

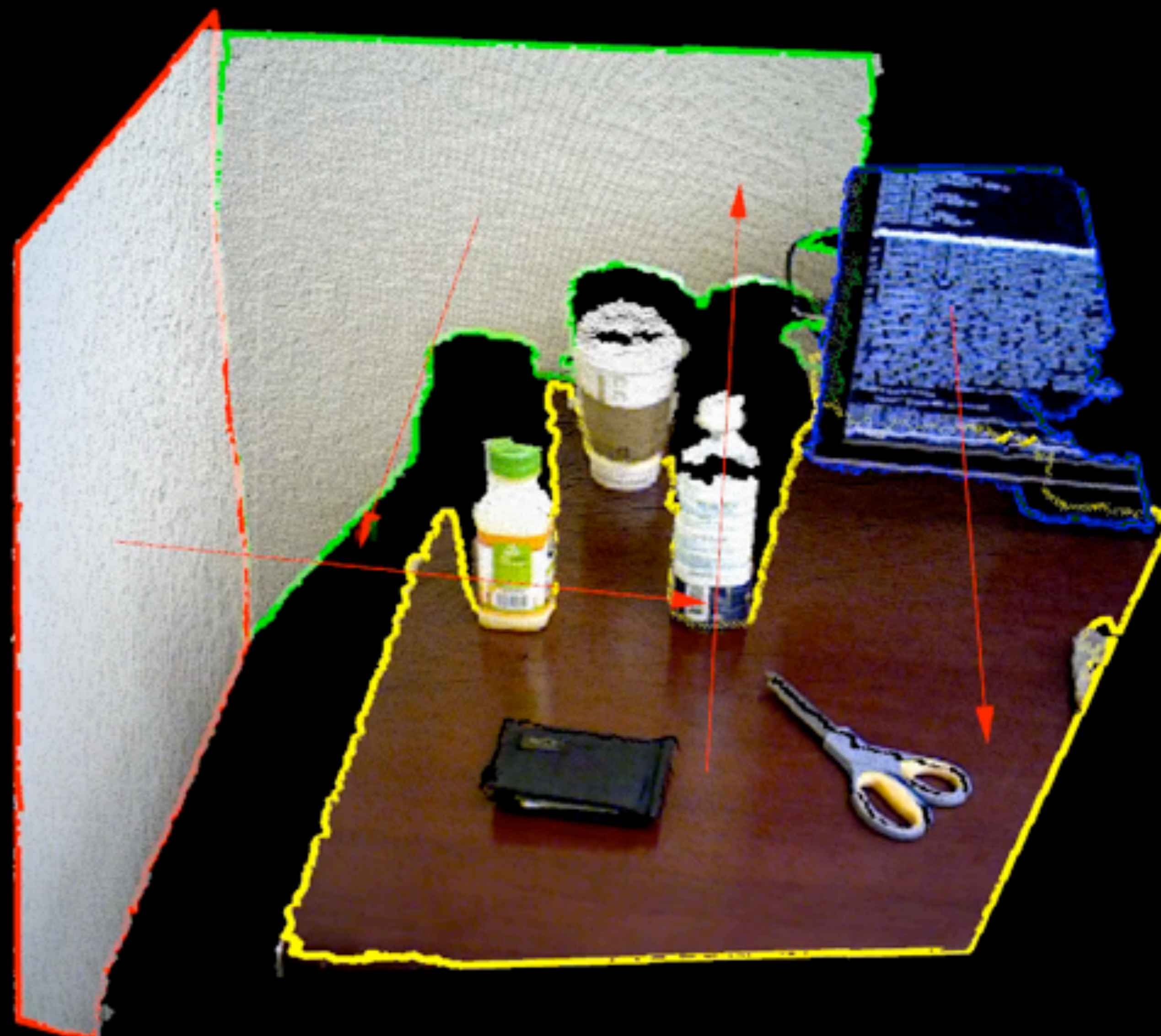


# Multi Plane Segmentation

- Comparison function for plane segmentation:

$$n_1 \cdot n_2 < \theta_{thresh}$$
$$|d_1 - d_2| < d_{thresh}$$

- Fit planes to regions with > min_inliers



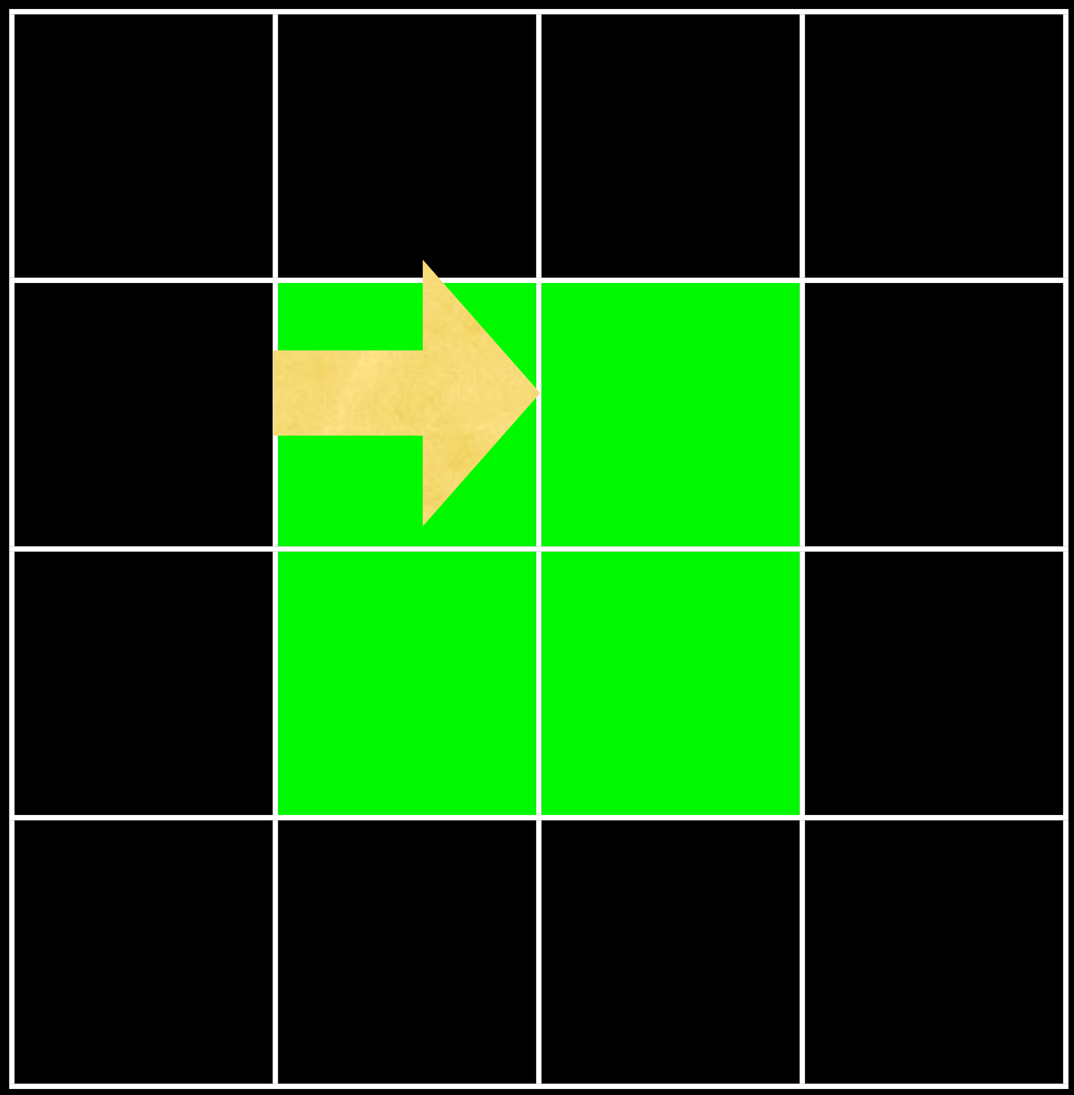
181.4.9 FPS

# Plane Coefficient Comparator

```
/** \brief Compare points at two indices by their plane equations.  True if the angle between the normals is less than the angular threshold,
  * and the difference between the d component of the normals is less than distance threshold, else false
  * \param idx1 The first index for the comparison
  * \param idx2 The second index for the comparison
  */
virtual bool
compare (int idx1, int idx2) const
{
  float threshold = distance_threshold_;
  if (depth_dependent_)
  {
    Eigen::Vector3f vec = input_->points[idx1].getVector3fMap ();

    float z = vec.dot (z_axis_);
    threshold *= z * z;
  }
  return ( (fabs ((*plane_coeff_d_)[idx1] - (*plane_coeff_d_)[idx2]) < threshold)
           && (normals_->points[idx1].getNormalVector3fMap ().dot (normals_->points[idx2].getNormalVector3fMap () ) > angular_threshold_ ) );
}
```

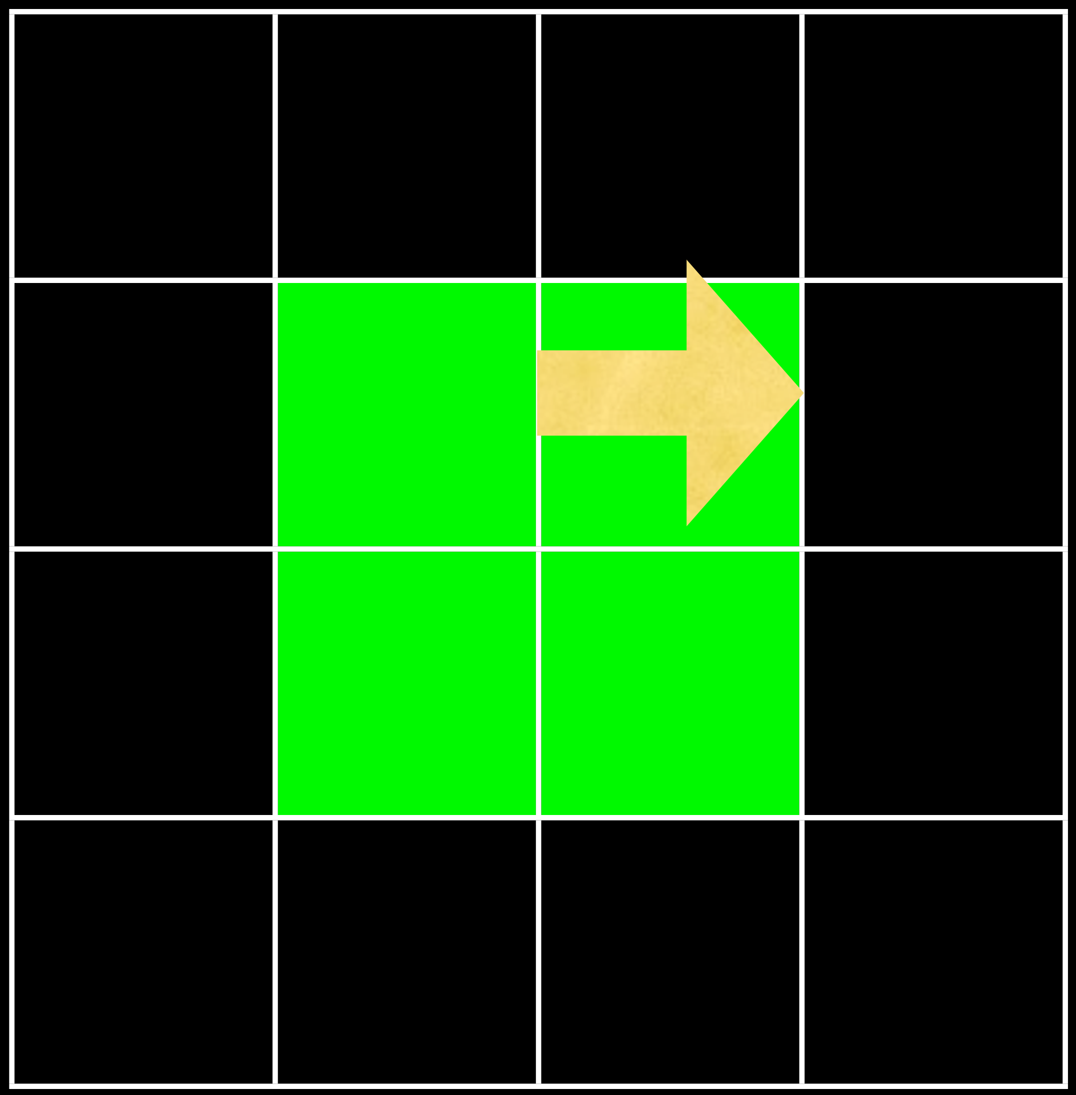
# Boundary Detection

- Organized data makes it easy to find the boundary of segmented regions
- Trace outer contour of a region
- Example convex hull timing (80k pts): 0.012 sec
- Example boundary timing (80k pts): 0.000048 sec



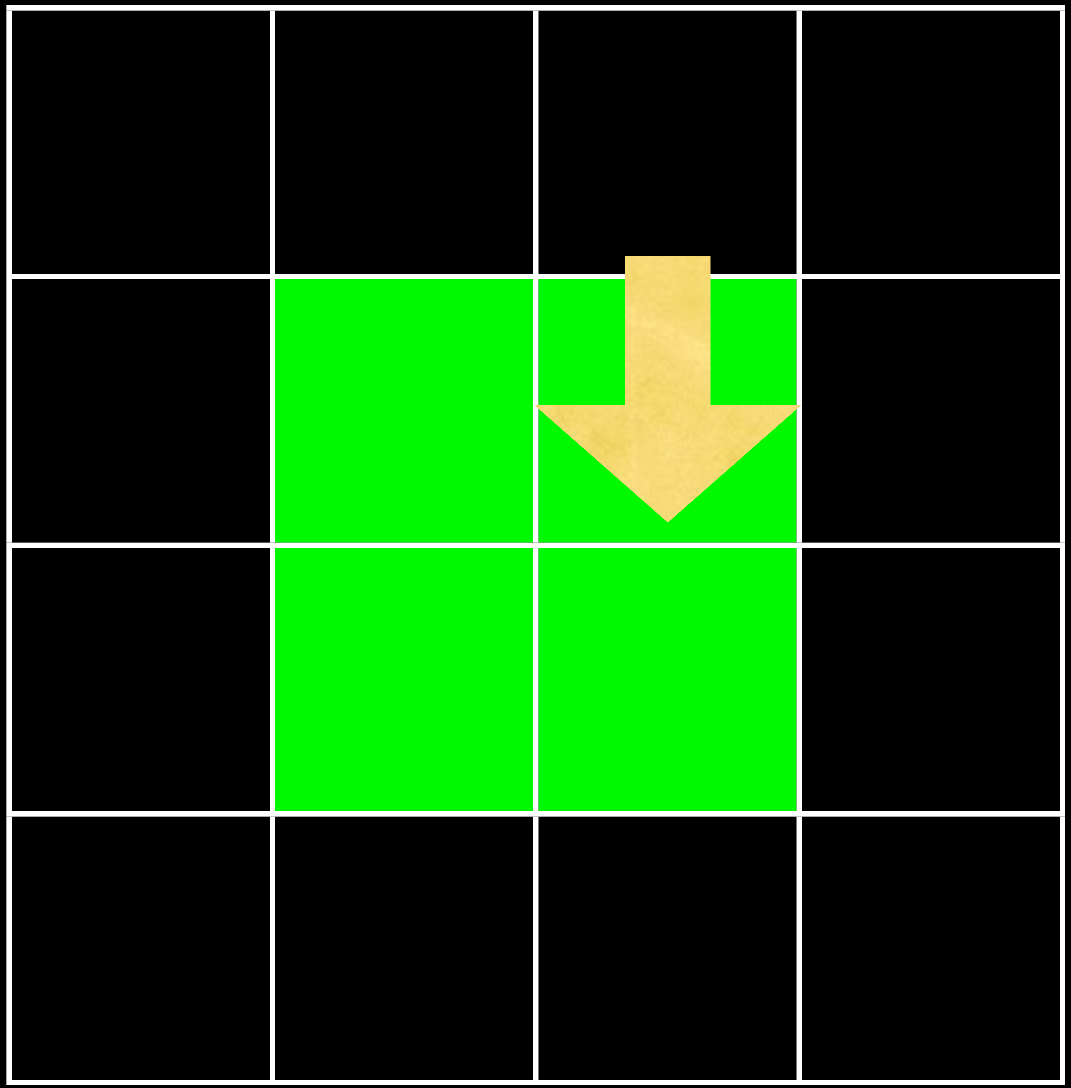
# Boundary Detection

- Organized data makes it easy to find the boundary of segmented regions
- Trace outer contour of a region
- Example convex hull timing (80k pts): 0.012 sec
- Example boundary timing (80k pts): 0.000048 sec



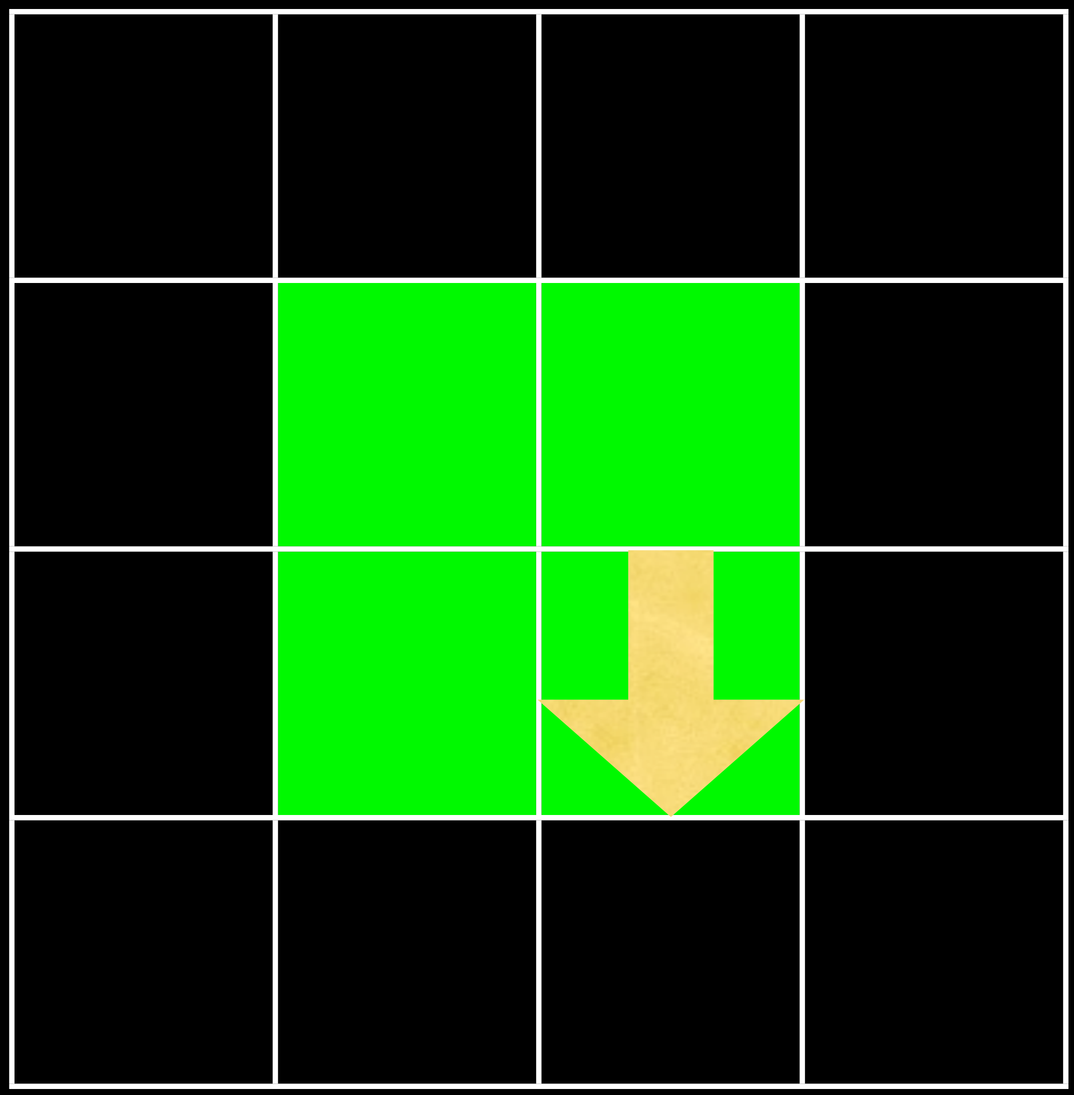
# Boundary Detection

- Organized data makes it easy to find the boundary of segmented regions
- Trace outer contour of a region
- Example convex hull timing (80k pts): 0.012 sec
- Example boundary timing (80k pts): 0.000048 sec



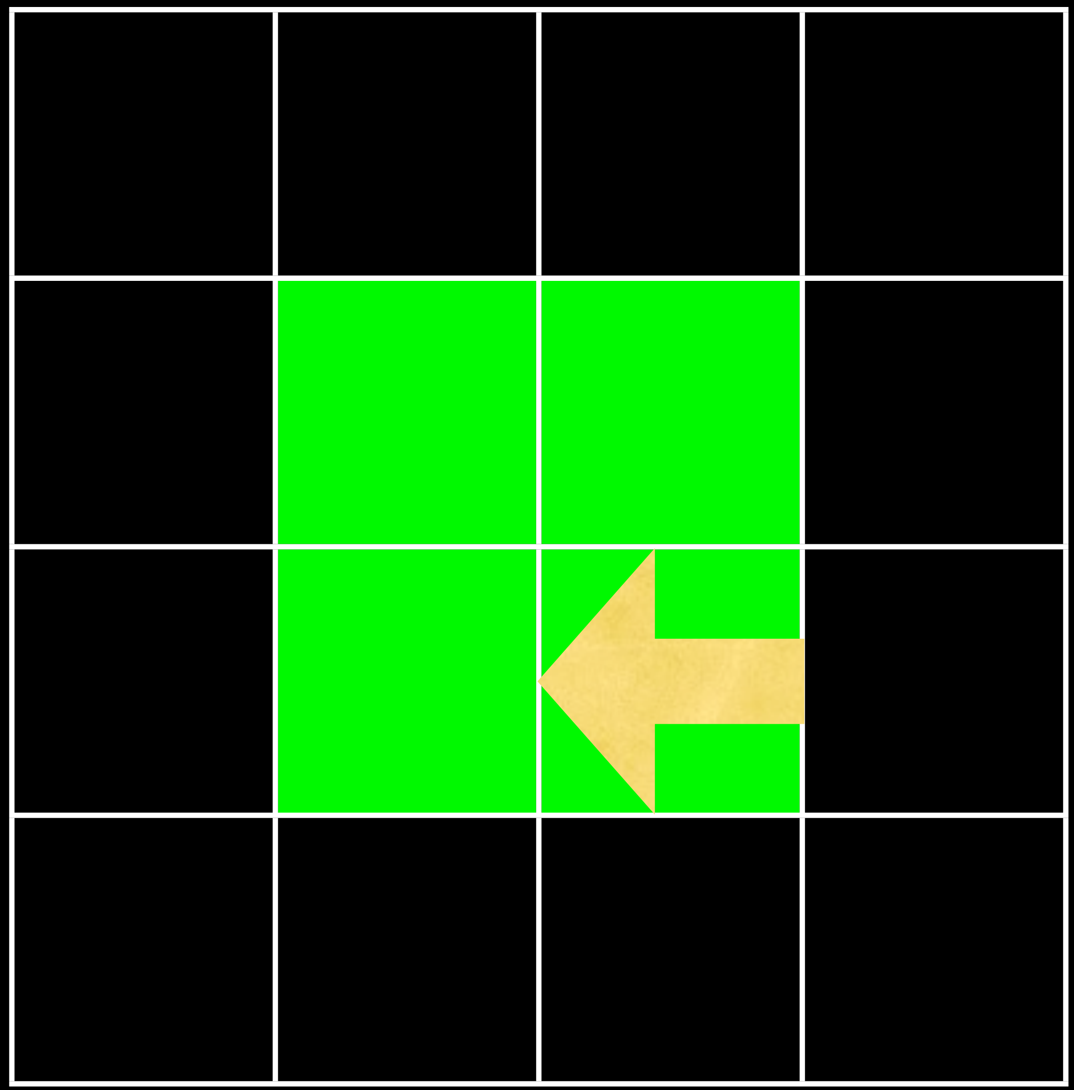
# Boundary Detection

- Organized data makes it easy to find the boundary of segmented regions
- Trace outer contour of a region
- Example convex hull timing (80k pts): 0.012 sec
- Example boundary timing (80k pts): 0.000048 sec



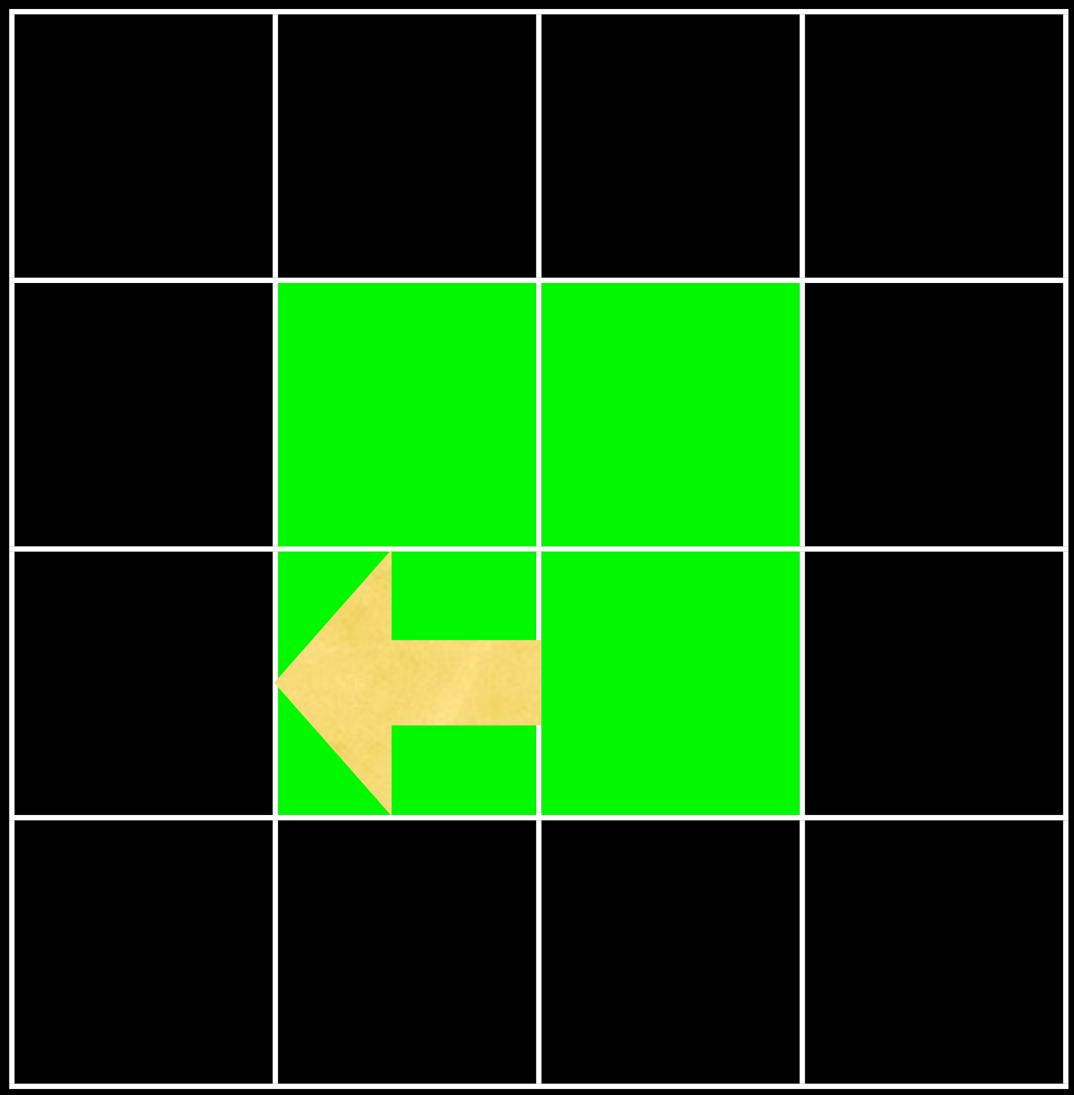
# Boundary Detection

- Organized data makes it easy to find the boundary of segmented regions
- Trace outer contour of a region
- Example convex hull timing (80k pts): 0.012 sec
- Example boundary timing (80k pts): 0.000048 sec



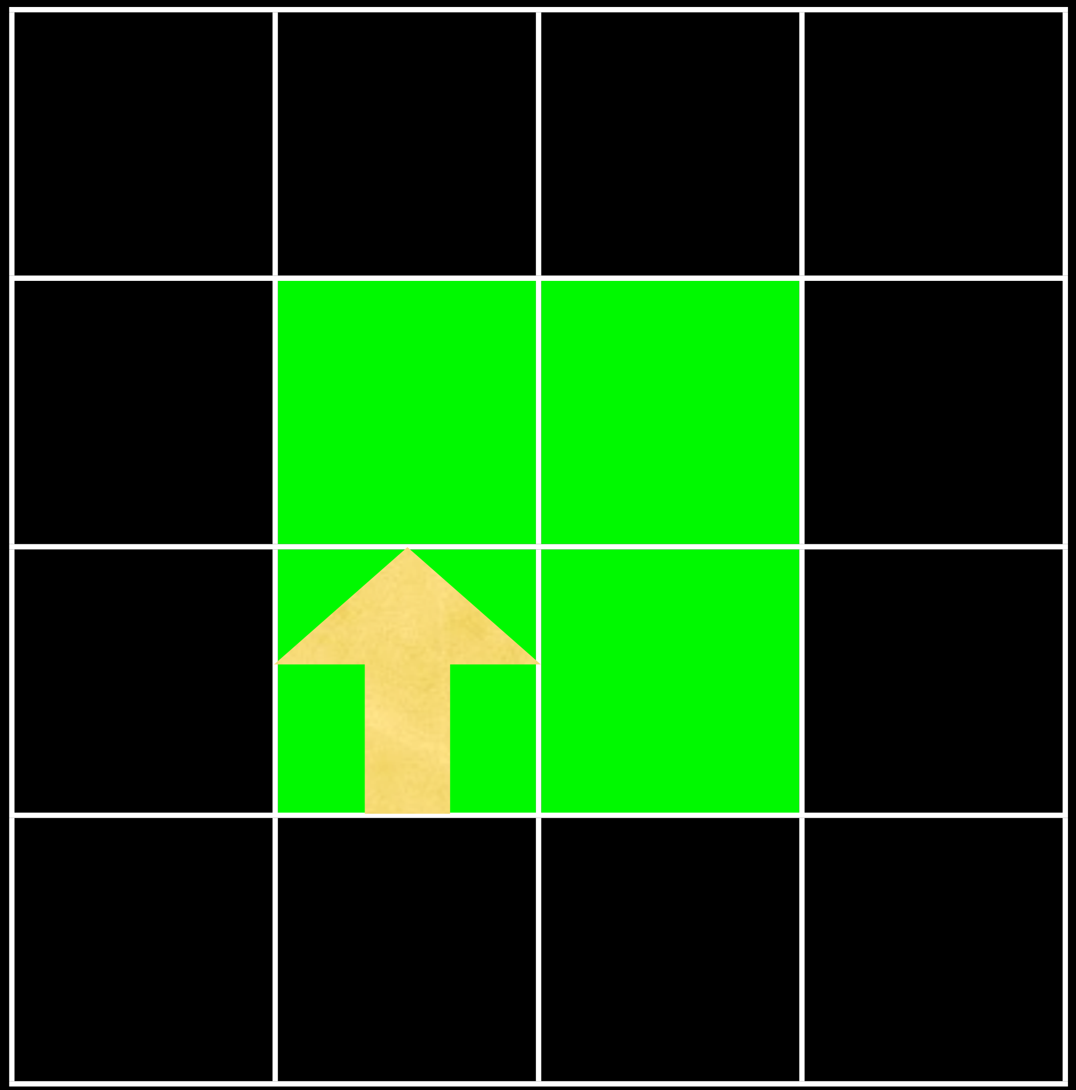
# Boundary Detection

- Organized data makes it easy to find the boundary of segmented regions
- Trace outer contour of a region
- Example convex hull timing (80k pts): 0.012 sec
- Example boundary timing (80k pts): 0.000048 sec



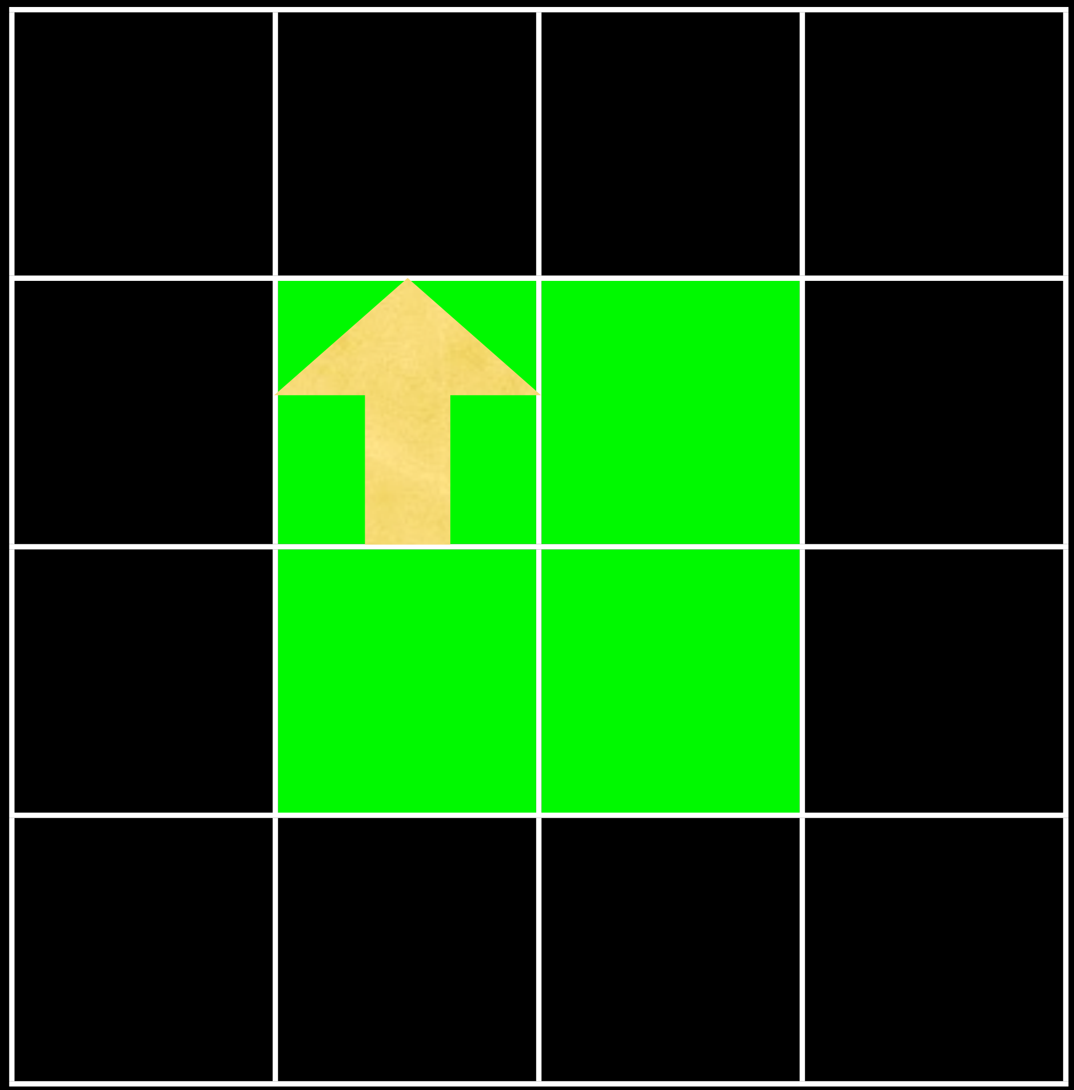
# Boundary Detection

- Organized data makes it easy to find the boundary of segmented regions
- Trace outer contour of a region
- Example convex hull timing (80k pts): 0.012 sec
- Example boundary timing (80k pts): 0.000048 sec



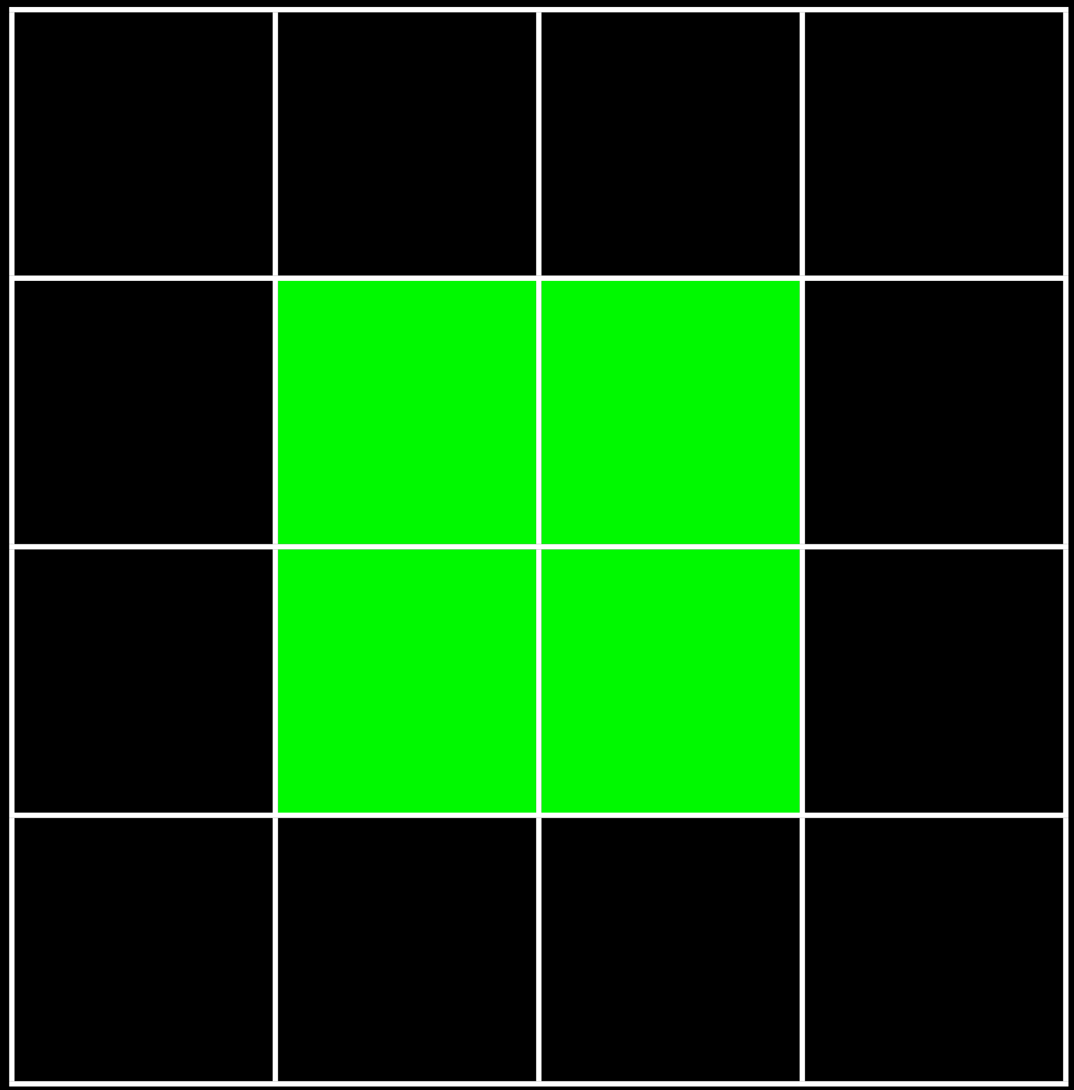
# Boundary Detection

- Organized data makes it easy to find the boundary of segmented regions
- Trace outer contour of a region
- Example convex hull timing (80k pts): 0.012 sec
- Example boundary timing (80k pts): 0.000048 sec



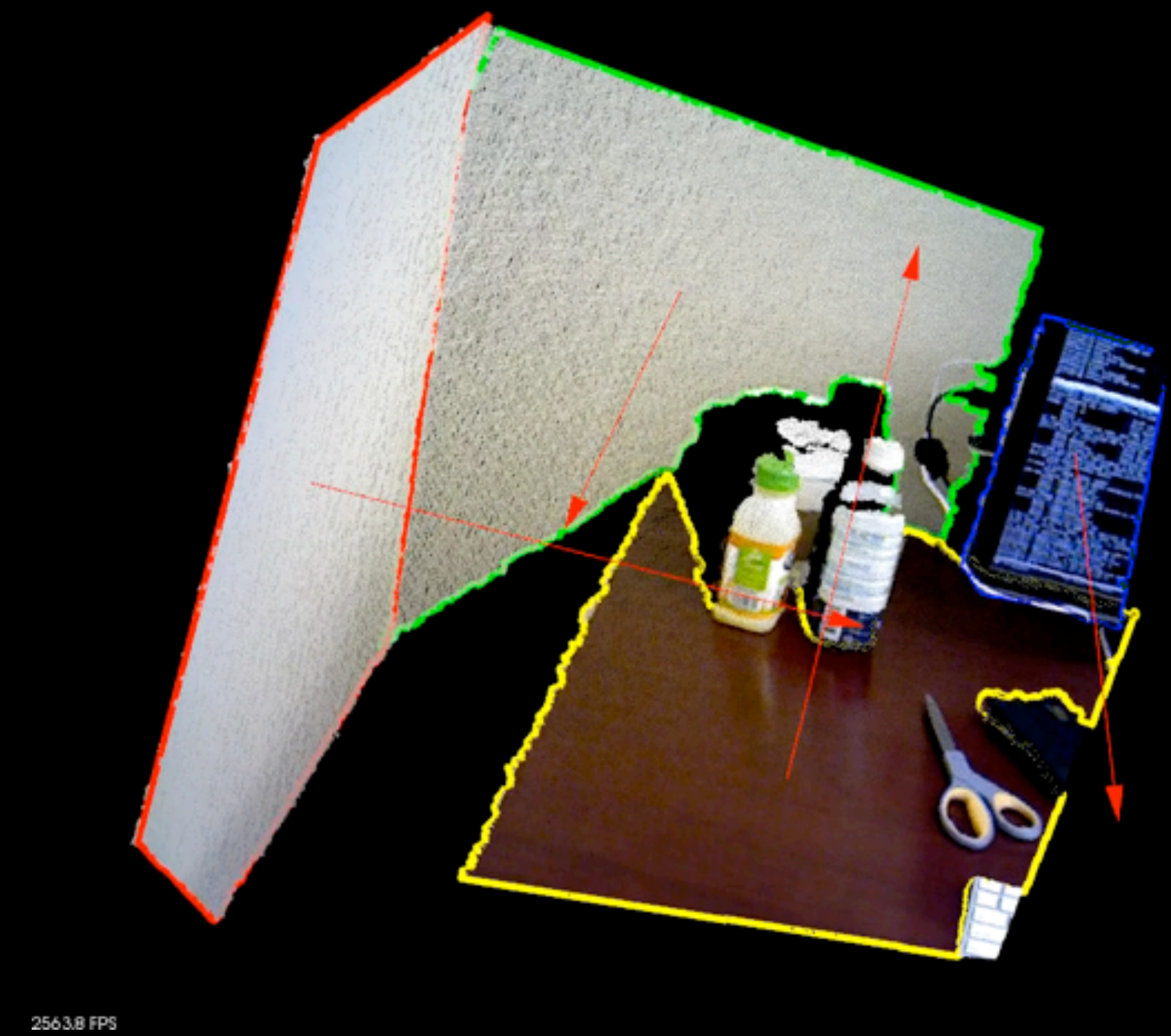
# Boundary Detection

- Organized data makes it easy to find the boundary of segmented regions
- Trace outer contour of a region
- Example convex hull timing (80k pts): 0.012 sec
- Example boundary timing (80k pts): 0.000048 sec



# Refinement of planar regions

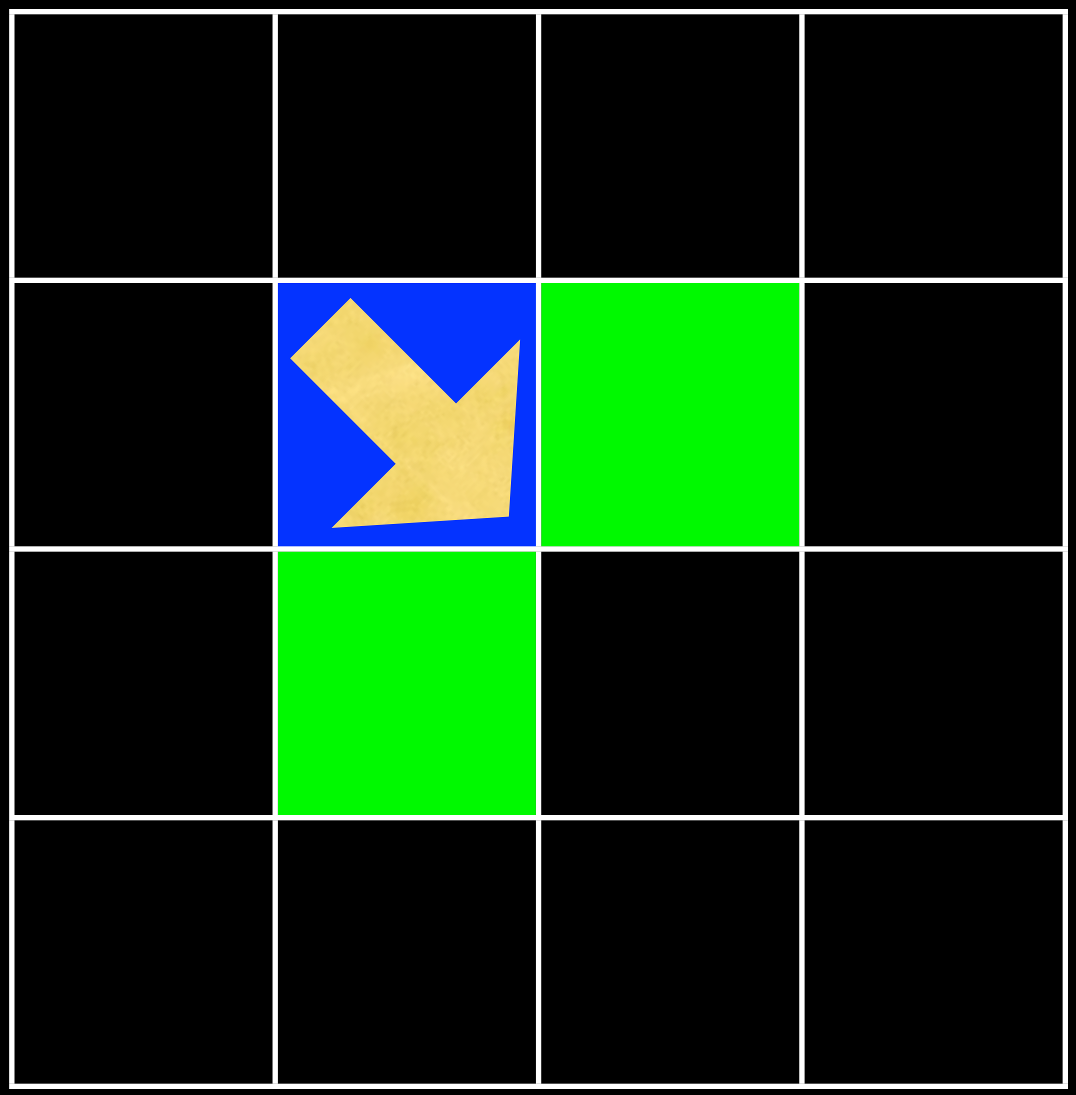
- Normals near surface edges can be noisy
- We can address this issue with an additional two passes
- Instead of comparing neighboring pixels values, we compare to the planar model of neighboring pixels



# Planar Refinement

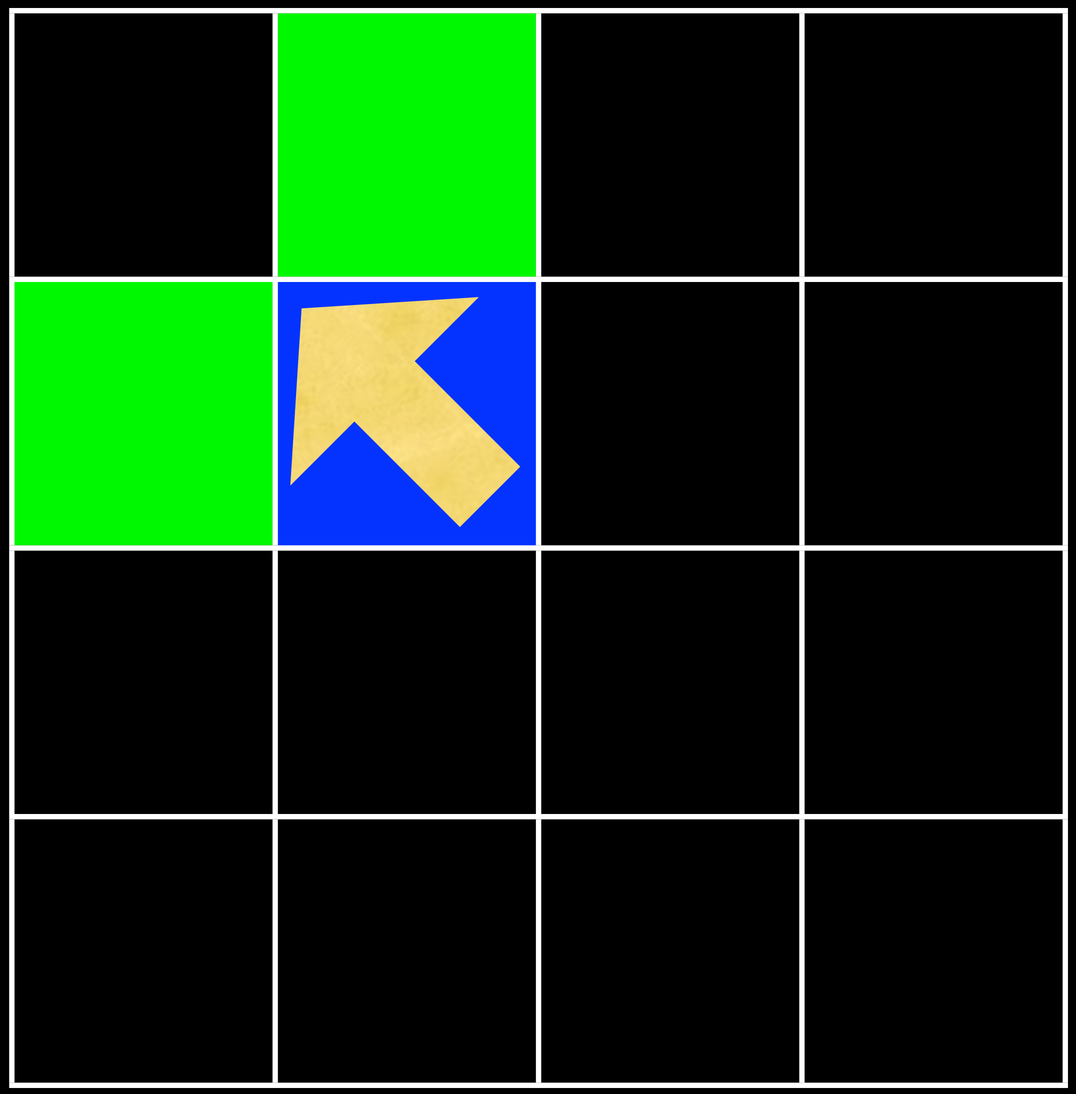
- Given a pixel that is an inlier to a planar region, we can extend the region to adjacent points using point to plane distance

- Requires two additional passes



# Planar Refinement

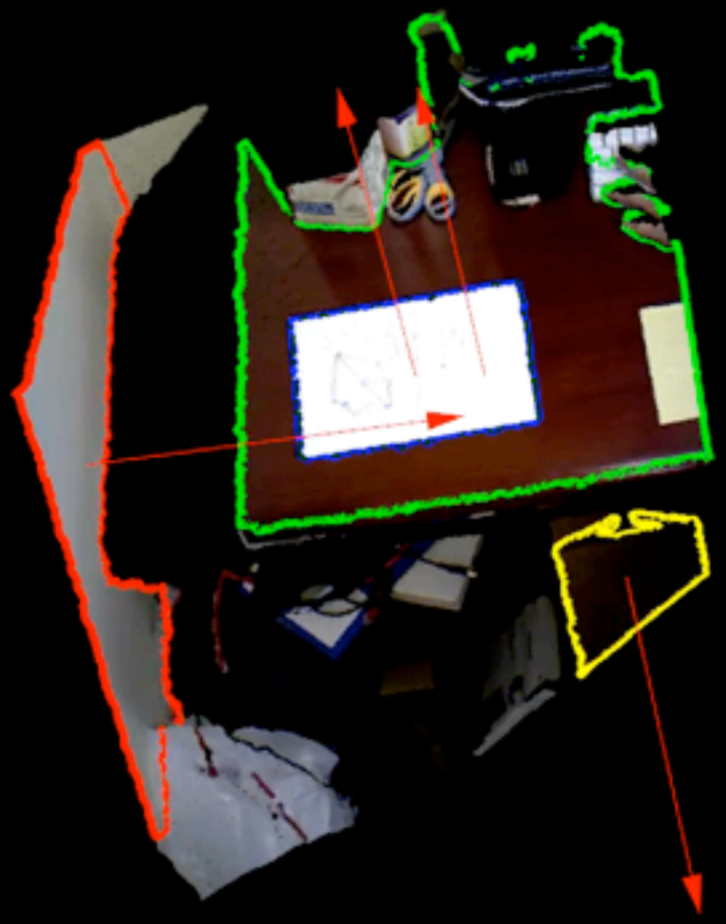
- Given a pixel that is an inlier to a planar region, we can extend the region to adjacent points using point to plane distance
- Requires two additional passes



```cpp
/** \brief Compare two neighboring points
  * \param[in] idx1 The index of the first point.
  * \param[in] idx2 The index of the second point.
  */
virtual bool
compare (int idx1, int idx2) const
{
  int current_label = labels_->points[idx1].label;
  int next_label = labels_->points[idx2].label;

  if (!((*refine_labels_)[current_label] && !(*refine_labels_)[next_label]))
    return (false);

  const pcl::ModelCoefficients& model_coeff = (*models_)[(*label_to_model_)[current_label]];

  PointT pt = input_->points[idx2];
  double ptp_dist = fabs (model_coeff.values[0] * pt.x +
                          model_coeff.values[1] * pt.y +
                          model_coeff.values[2] * pt.z +
                          model_coeff.values[3]);

  // depth dependent
  float threshold = distance_threshold_;
  if (depth_dependent_)
  {
    Eigen::Vector3f vec = input_->points[idx1].getVector3fMap ();

    float z = vec.dot (z_axis_);
    threshold *= z * z;
  }

  return (ptp_dist < threshold);
}
```

# Using Color Information

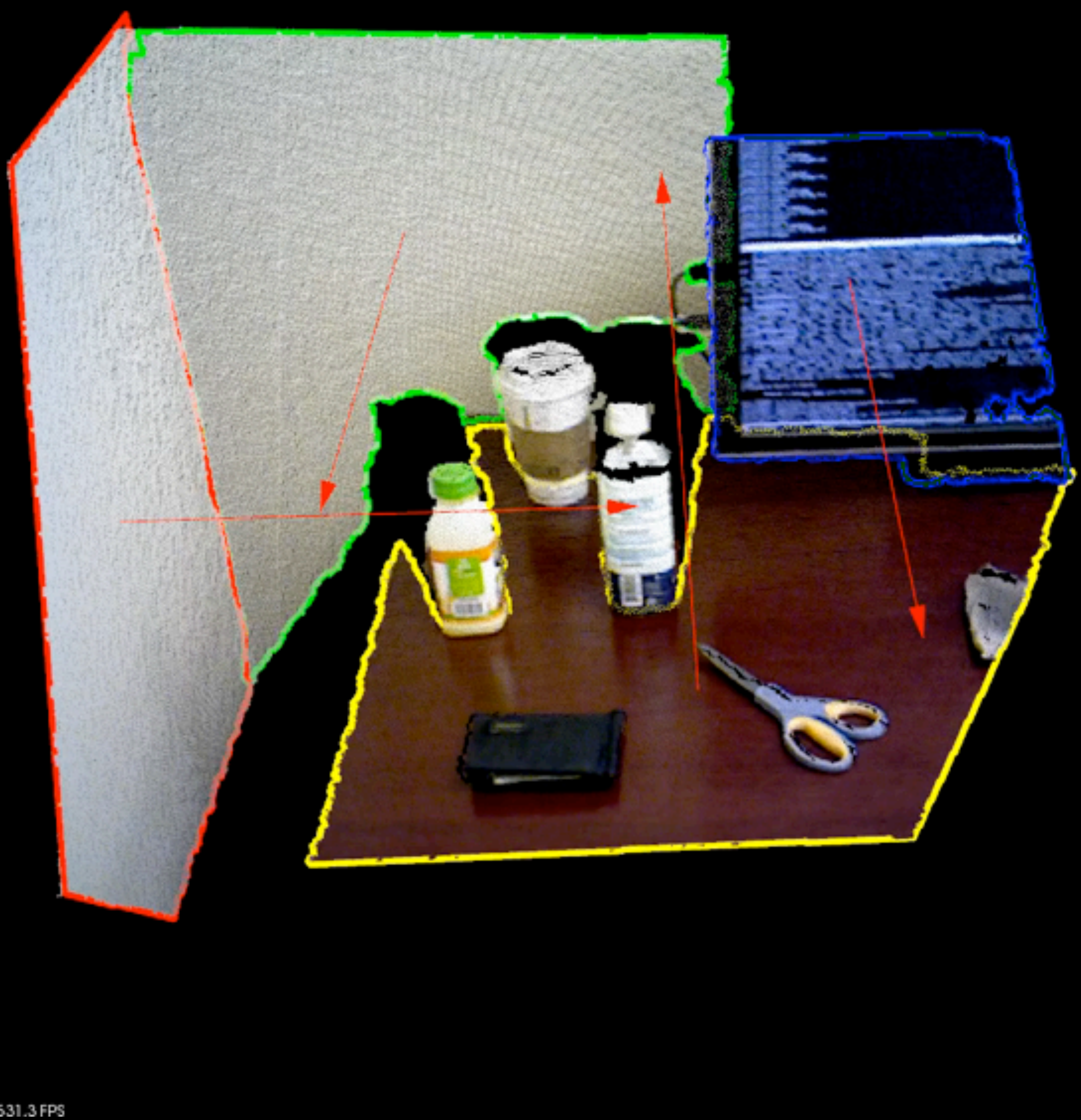
- We can easily segment on color and geometric information at little additional cost
- We can use the same comparison as we did for planes, plus an additional constraint that neighboring pixels have similar color



1984.1 FPS

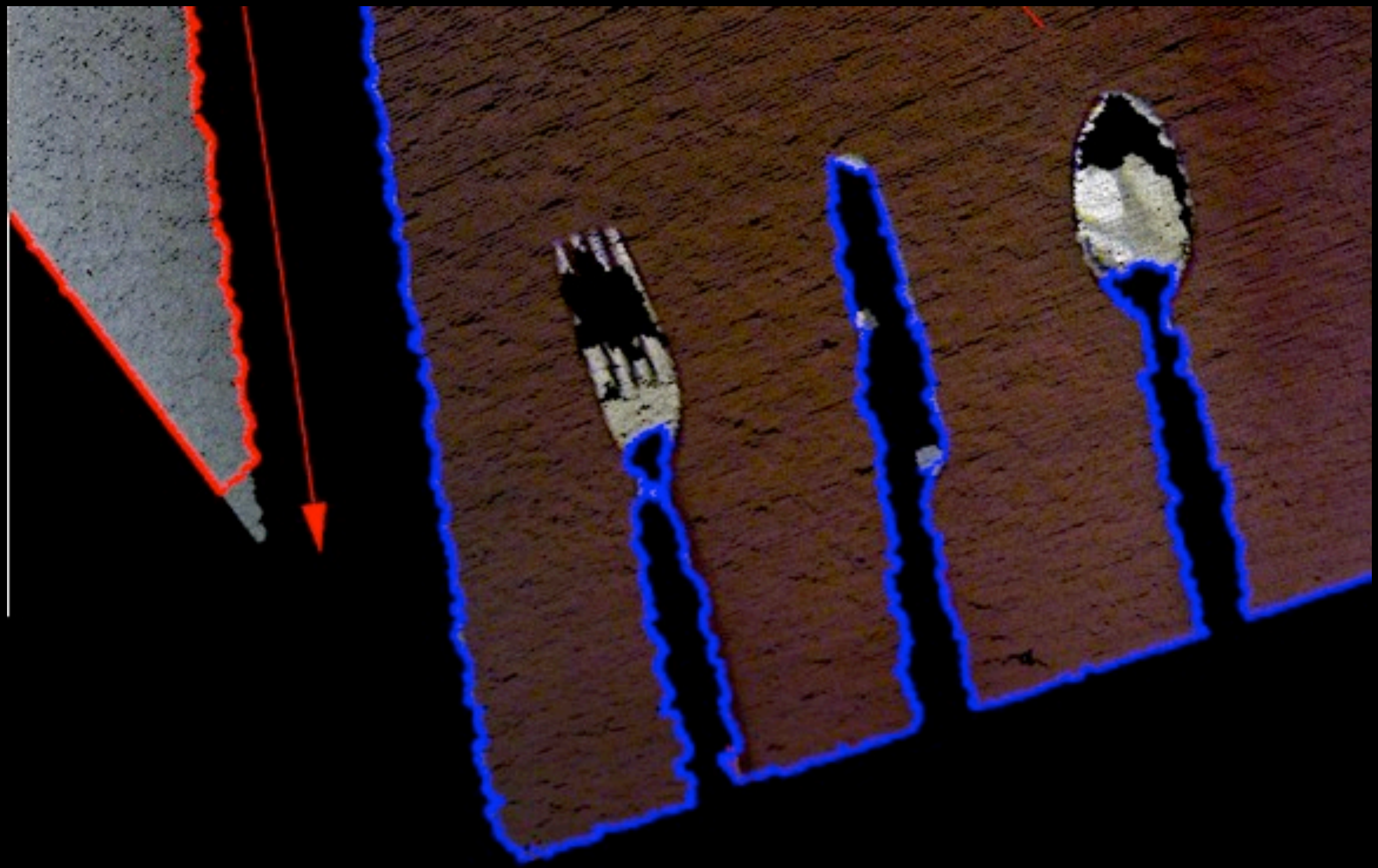
# Euclidean Cluster Extraction

- Use planar regions extracted in the first pass as a mask
- Simply compare euclidean distance between neighboring points



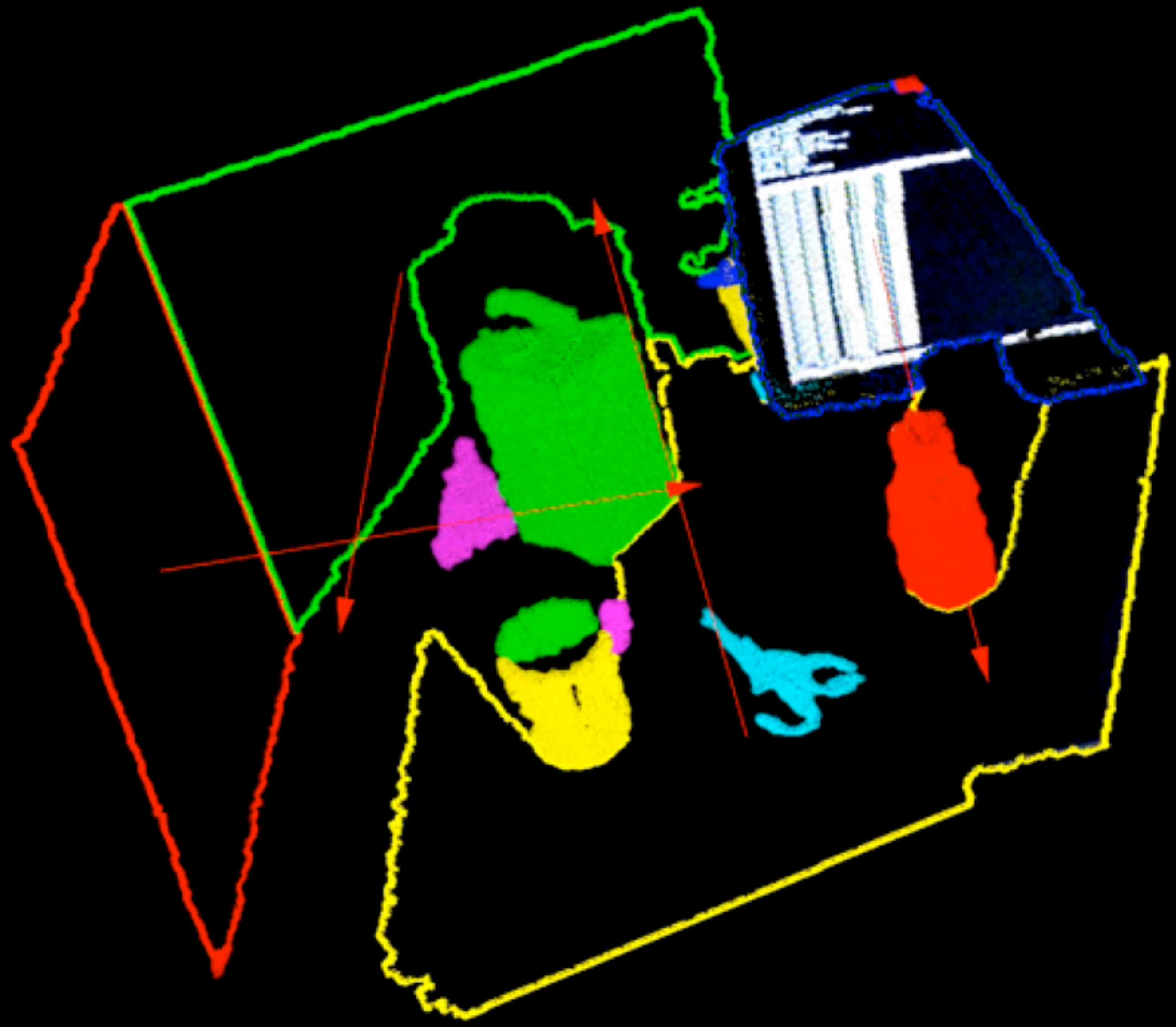
# Using Boundaries and Holes

- Since we have organized data, we can also notice missing / NAN data
- This can be useful for detecting small shiny objects which may not give good depth returns



# Robust to Lighting Conditions

- Comparison functions that don't use color data work fine in the dark!



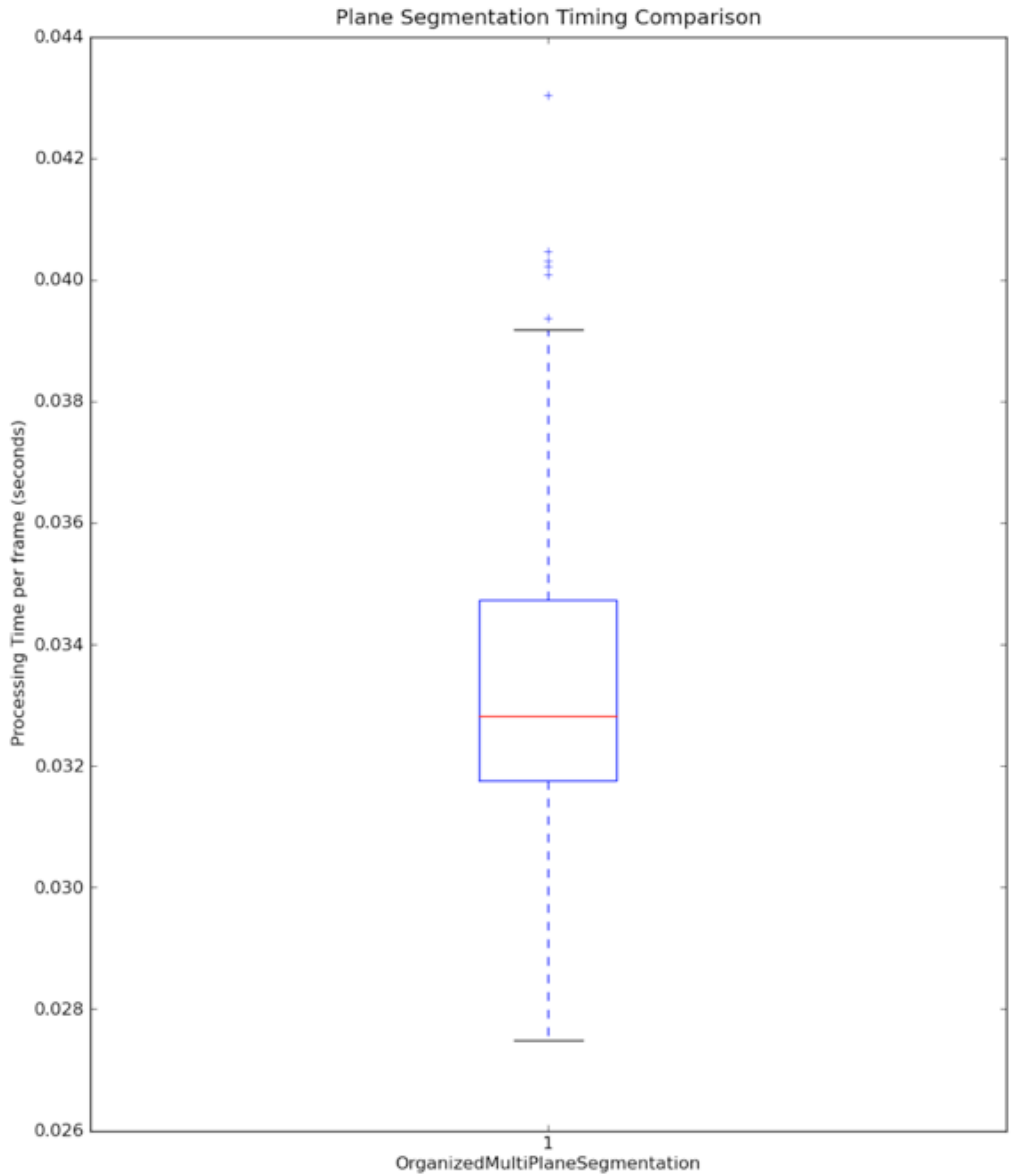
961.6 FPS

# Timing Improvements

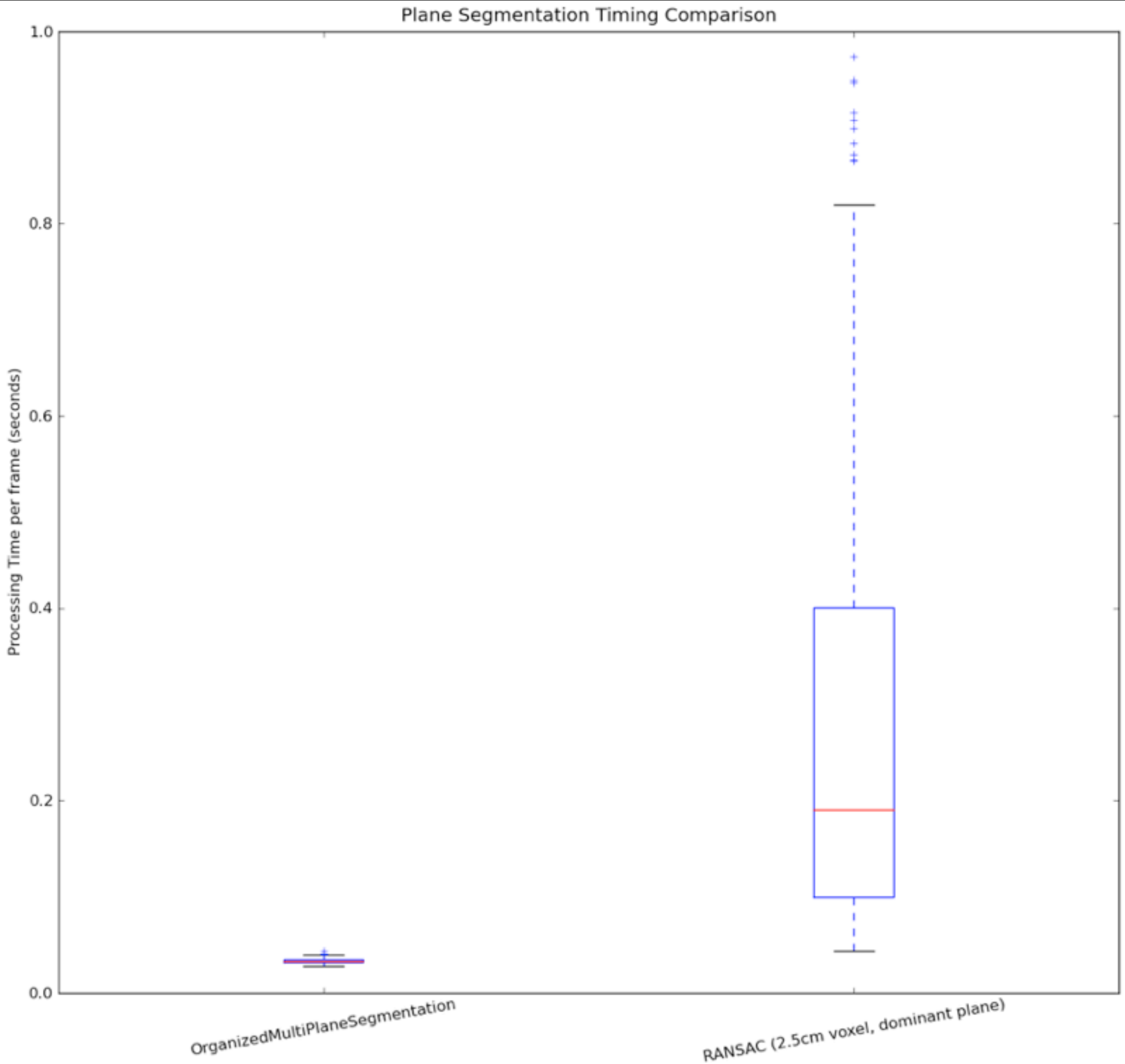
- Fair comparison is not so easy due to the large number of parameters.
- For this comparison, I chose to use Iterative RANSAC as used in my previous mapping work, 1000 iterations
  - Clusters RANSAC planes to separate coplanar disjoint regions (e.g. multiple tables)
  - Computes convex hull, since the MPS gives a boundary
- Data set is kinect data from the "yesterday's sushi" area, 508 frames
- Timings are reported for plane extraction + hull / boundary computation only, so this is on top of normal estimation

# Timing Improvements

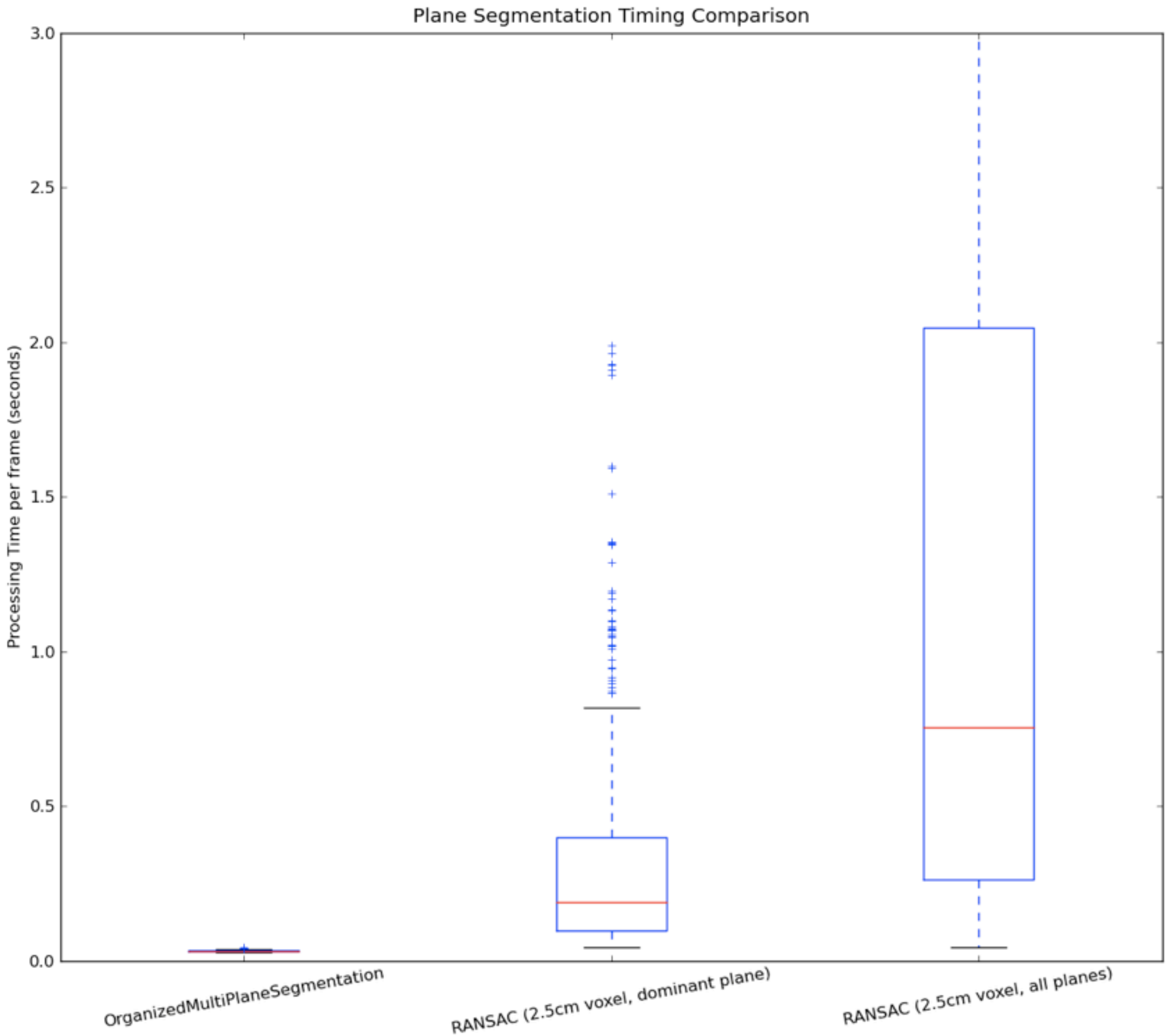
- MultiPlaneSegmentation, PlaneCoefficientComparator, SegmentAndRefine:
  - mean = 0.0332 seconds, stddev = 0.0023
- RANSAC (1000 iterations, dominant plane, 2.5cm voxelized cloud):
  - mean = 0.32 seconds, stddev = 0.34
- RANSAC (1000 iterations, all planes in the scene, 2.5cm voxelized cloud):
  - mean = 1.52 seconds, stddev = 1.79
- RANSAC (1000 iterations, all planes in the scene, 1cm voxelized cloud):
  - mean = 6.18 seconds, stddev = 6.62



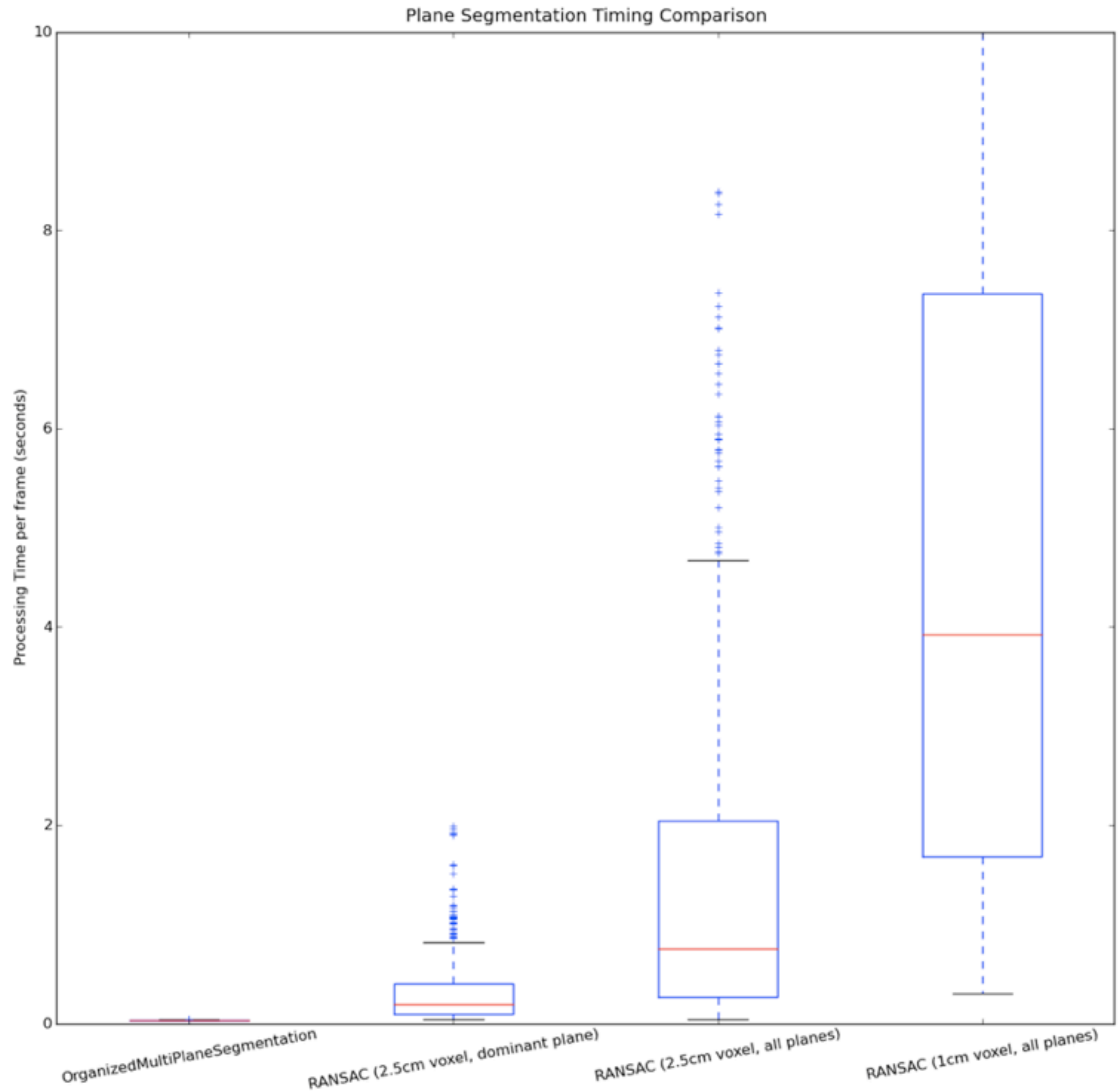
Plane Segmentation Timing Comparison
0.044
0.042
0.040
0.038
0.036
0.034
0.032
0.030
0.028
0.026
Processing Time per frame (seconds)
1
OrganizedMultiPlaneSegmentation



Plane Segmentation Timing Comparison
Processing Time per frame (seconds)
0.0
0.2
0.4
0.6
0.8
1.0
OrganizedMultiPlaneSegmentation
RANSAC (2.5cm voxel, dominant plane)



Plane Segmentation Timing Comparison
Processing Time per frame (seconds)
3.0
2.5
2.0
1.5
1.0
0.5
0.0
OrganizedMultiPlaneSegmentation
RANSAC (2.5cm voxel, dominant plane)
RANSAC (2.5cm voxel, all planes)



Plane Segmentation Timing Comparison
Processing Time per frame (seconds)
0
2
4
6
8
10
OrganizedMultiPlaneSegmentation
RANSAC (2.5cm voxel, dominant plane)
RANSAC (2.5cm voxel, all planes)
RANSAC (1cm voxel, all planes)

# Code Example

```
// Segment planes
pcl::OrganizedMultiPlaneSegmentation<PointT, pcl::Normal, pcl::Label> mps;
mps.setMinInliers (10000);
mps.setAngularThreshold (0.017453 * 2.0); // 2 degrees
mps.setDistanceThreshold (0.02); // 2cm
mps.setInputNormals (normal_cloud);
mps.setInputCloud (cloud);
std::vector<pcl::PlanarRegion<PointT> > regions;
mps.segmentAndRefine (regions);

for (size_t i = 0; i < regions.size (); i++)
{
  Eigen::Vector3f centroid = regions[i].getCentroid ();
  Eigen::Vector4f model = regions[i].getCoefficients ();
  pcl::PointCloud<PointT> boundary_cloud;
  boundary_cloud.points = regions[i].getContour ();
  printf ("Centroid: (%f, %f, %f)\n  Coefficients: (%f, %f, %f, %f)\n Inliers: %d\n",
          centroid[0], centroid[1], centroid[2],
          model[0], model[1], model[2], model[3],
          boundary_cloud.points.size ());
}
```

- See PCL's Organized Segmentation Demo for more: pcl_organized_segmentation_demo

# Polygon Merge

- Given two 3D polygons (not necessarily exactly co-planar) and plane equations, compute the new polygon and merged plane equation
- Project to 2D
- Use boost::geometry's fast 2D polygon boolean operations
- Coming soon to PCL

# Mapping Planar Surfaces from multiple views: Planar Polygon Fusion

- Considering multiple frames is more interesting than single views
- Given a set of frames and sensor locations (from odom, AMCL, etc) we can fuse planar surfaces across views

576.7 FPS