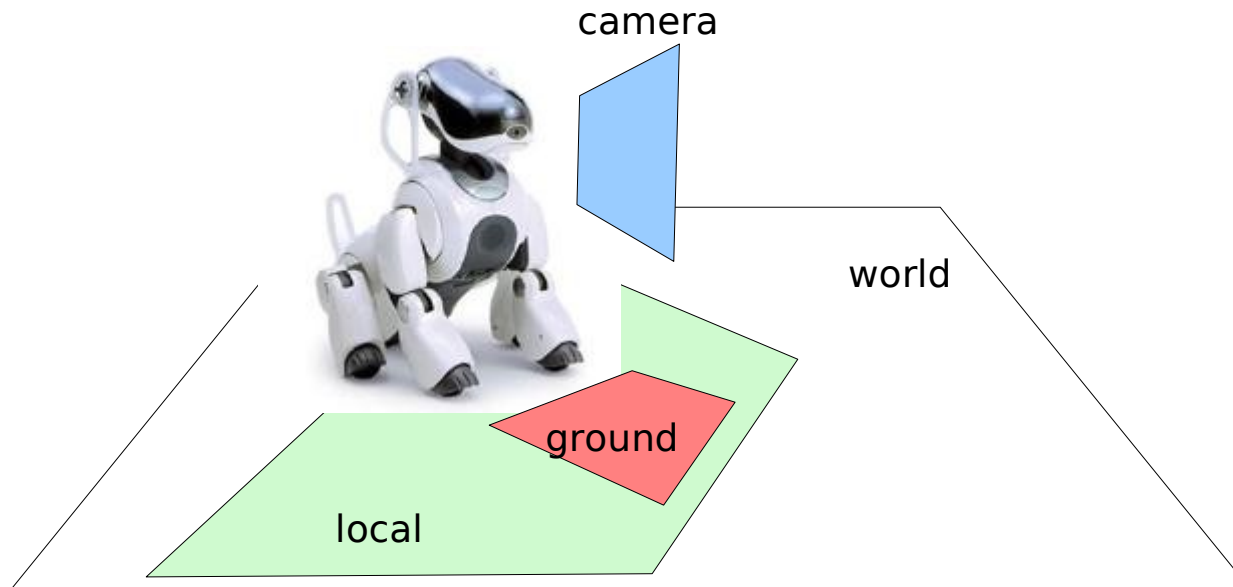


# World Maps and Localization

15-494 Cognitive Robotics  
David S. Touretzky &  
Ethan Tira-Thompson

Carnegie Mellon  
Spring 2008

# Frames of Reference



- Camera frame: what the robot sees.
- `projectToGround()` = kinematics + planar world assumption.
- Local map assembled from camera frames each projected to ground; robot moves head but not body.
- World map assembled from local maps built at different spots in the environment.

# Four Shape Spaces

- camShS = camera space
- groundShS = camera shapes projected to ground plane
- localShS = body-centered (egocentric space);  
constructed by matching and importing shapes  
from groundShS
- **worldShS** = world space (allocentric space);  
constructed by matching and importing shapes  
from localShS
- The robot is explicitly represented in worldShS

# Deriving the Local Map

- 1) MapBuilder extracts shapes from the camera frame
  - Use a request of type `MapBuilderRequest::cameraMap` if you just want camera space shapes.
- 2) MapBuilder does `projectToGround()`
  - Use `MapBuilderRequest::groundMap` if you only want ground shapes from the current camera frame.
- 3) MapBuilder matches ground shapes against local shapes.
  - Request type should be `MapBuilderRequest::localMap`
- 4) MapBuilder moves to the next gaze point and repeats.
  - The world is assumed not to change during this process.

# Deriving the World Map

- The local map covers only what the robot can see from a single viewing position.
- The world map can cover much larger territory.
  - Use `MapBuilderRequest::worldMap`
- The world map persists over a long time period.
  - The world will change. Updates must be possible.
- We update the world map by:
  - Constructing a local map.
  - Aligning it with the world map (by translation and rotation)
  - Importing shapes from the local map.
  - Noting additions and deletions since the last local map match

# Localization

- How do we align the local map with the world map?
- This turns out to be equivalent to determining our position and orientation on the world map.
- Tricky, because:
  - The local map is noisy
  - The environment can be ambiguous (multiple pink landmarks)
- Sensor model: describes the uncertainty in our sensor measurements.
  - Can mix sensor types (vision, IR), info types (bearing, distance)

# SLAM

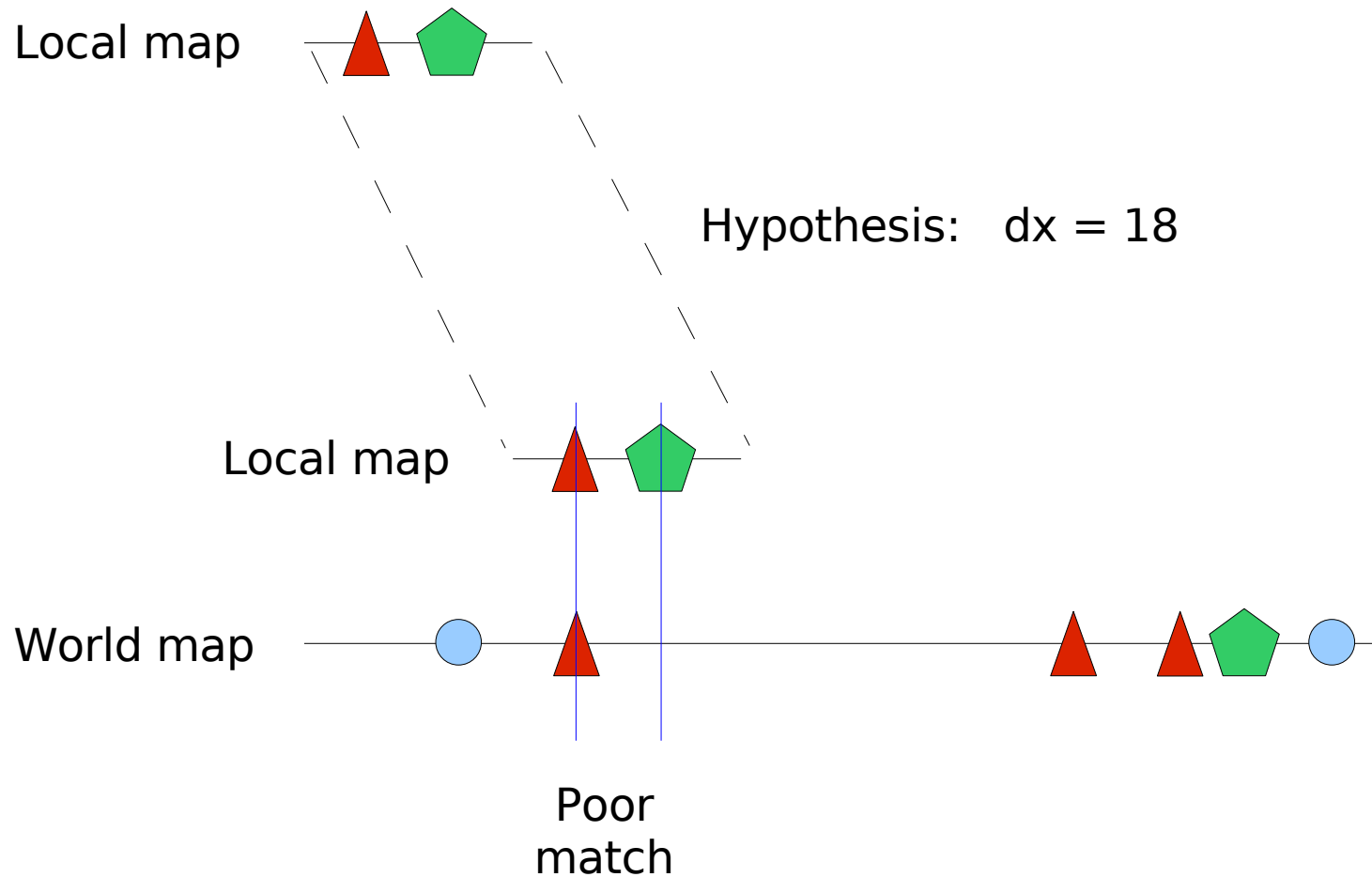
- Simultaneous Localization and Mapping
- When is this necessary?
  - When we don't know the map in advance.
  - When the world is changing (landmarks can appear or disappear, or change location.)
  - When we're moving through the world.
- How do we localize on a map that we are still in the process of building?
- Motion model: estimates (by odometry) our motion through the environment.

# Particle Filtering

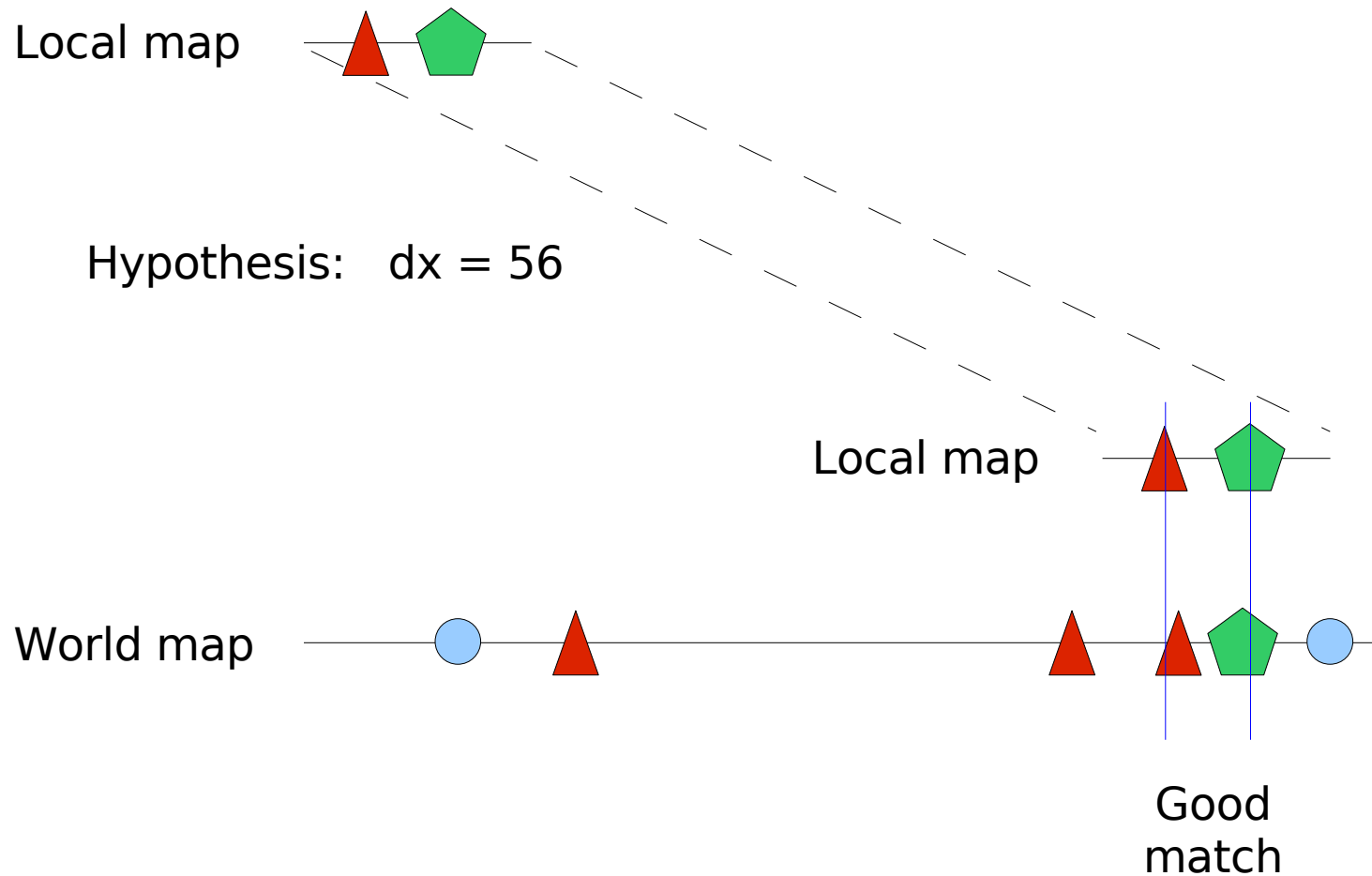
- A technique for searching large, complex spaces.
- What is the hypothesis space we need to search?
  - Robot's position (x,y)
  - Robot's orientation  $\theta$
  - Which world space shapes have disappeared since last update?
  - What new shapes have appeared in local space?
- Each particle encodes a point in the hypothesis space.
- How can we evaluate hypotheses?
  - Use sensor and motion models to update particle weights



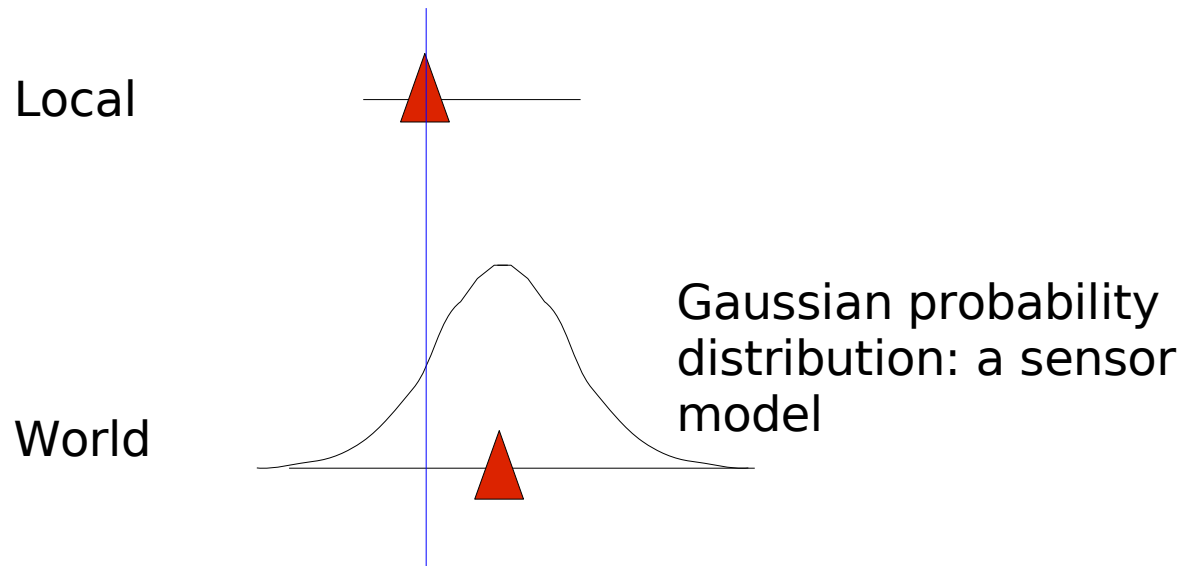
# Ranking a Particle: 1-D Case



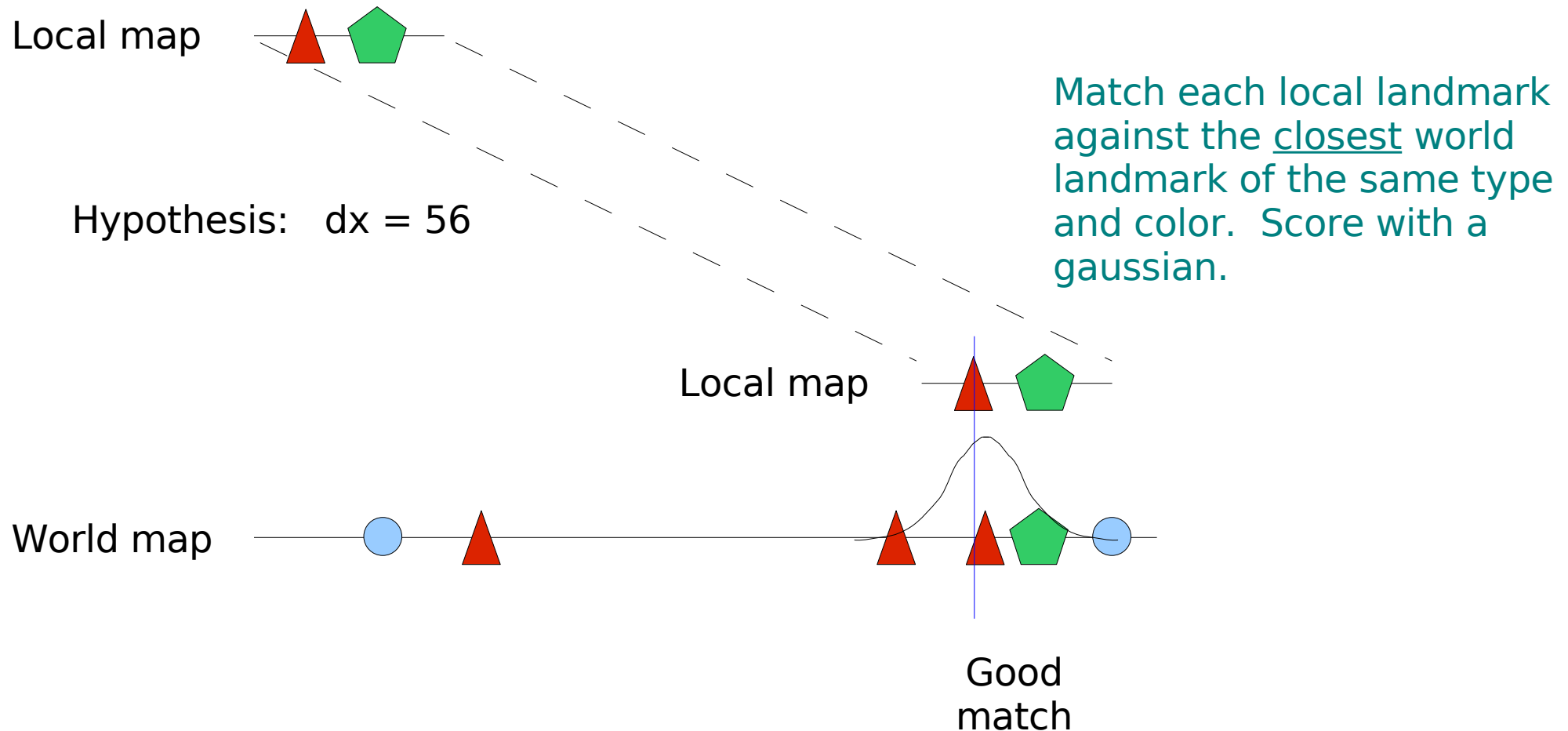
# Ranking a Particle: 1-D Case



# Matching a Landmark



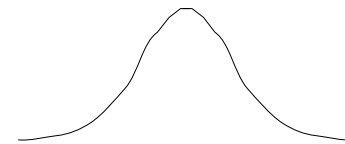
# Pick the Best Candidate



# Matching a Set of Landmarks

- Take the product of the match probabilities of the individual landmarks:

$$G(x, x_0) = \exp\left[\frac{-(x - x_0)^2}{\sigma^2}\right]$$



L.s = coordinate of  
shape s in Local map

$$P(s \in L, t \in W | h) = G(L.s + h, W.t)$$

W.t = coordinate of  
shape t in World map

$$P(s \in L | W, h) = \max_{t \in W} P(s \in L, t \in W | h)$$

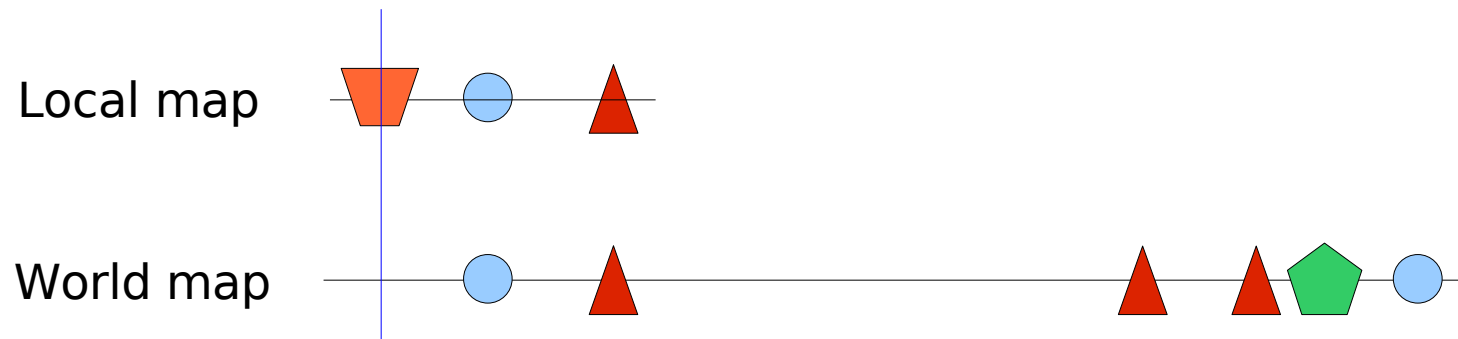
h = location hypothesis

$$P(h) = \prod_{s \in L} P(s | W, h)$$

- Allow penalty terms for addition, deletion.

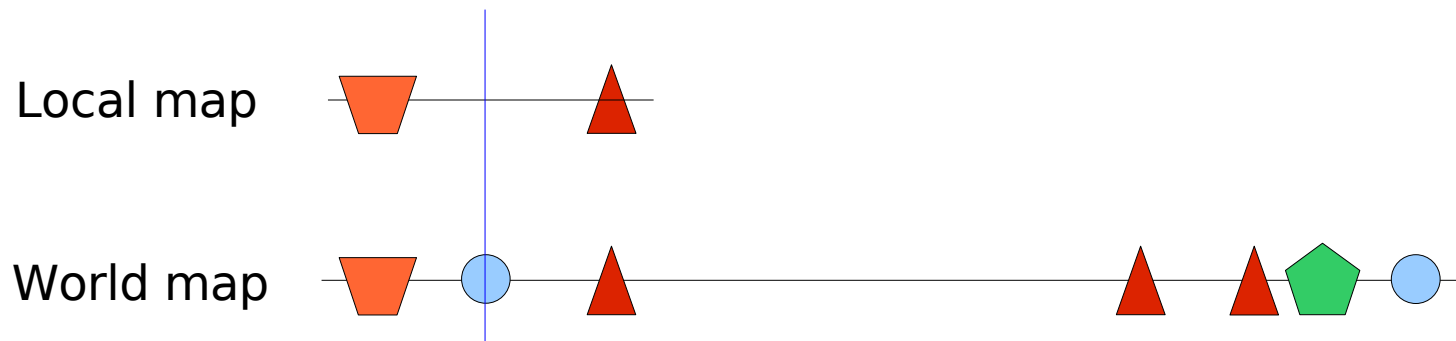
# Addition Penalty

- A shape in the local map that isn't in the world map must be accounted for as an addition.
- Assess a penalty on  $P(h)$  for each addition, but remove that shape from the product term for  $P(h)$  so the product doesn't go to zero.



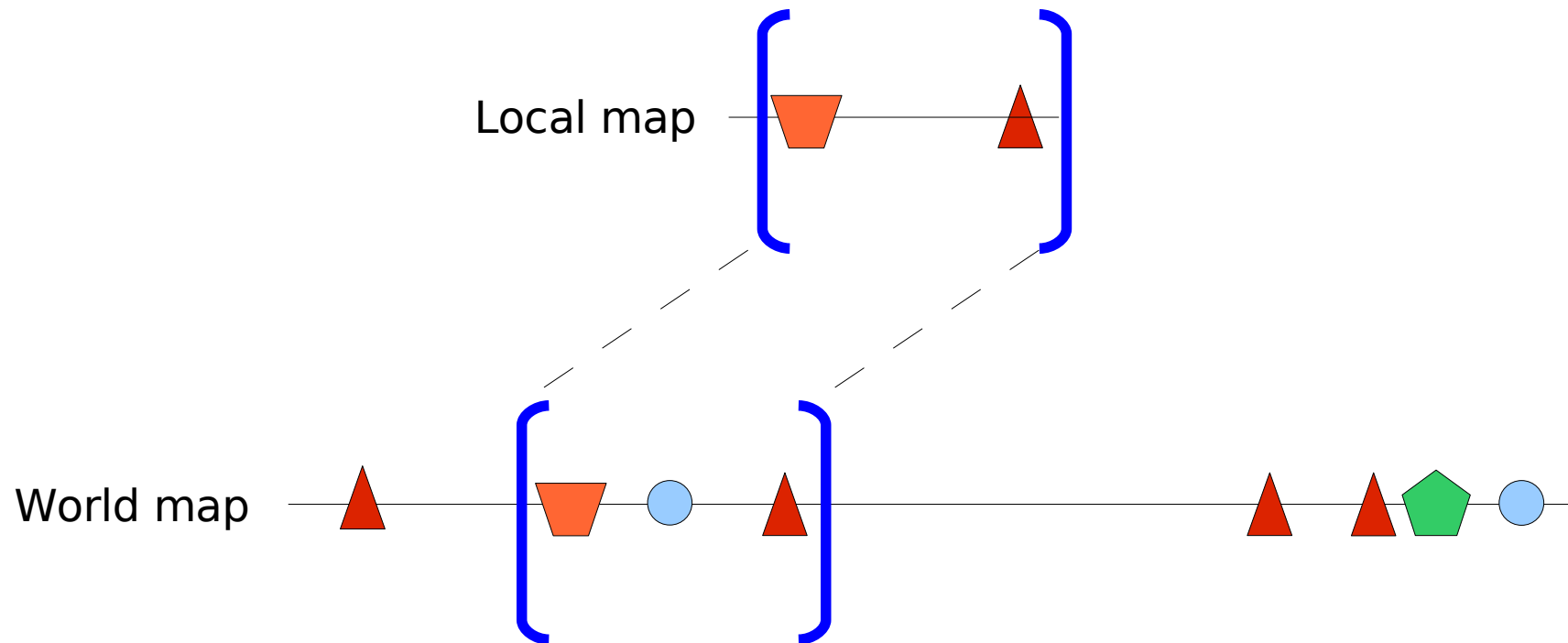
# Deletion Penalty

- A shape in the world map that should be visible in the local map but isn't must be accounted for as a deletion.
- Assess a penalty on  $P(h)$  for each deletion, but remove that shape from the product term for  $P(h)$  so the product doesn't go to zero.



# What Shapes Should be Visible?

- Take bounding box of shapes in local space.
- All shapes within that box should be visible in world space.





# When Objects Move

- If an object moves only a little bit, it will still match, and the position will be updated.
- If an object moves by a larger amount, we'll get:
  - An object deletion at the old location
  - An object addition at the new location
- Could watch for this and combine both changes into a single “move” penalty.
- If  $h$  is a poor hypothesis, then every object will appear to have “moved”.

# Importance Sampling

- For each particle  $h$ , calculate the probability  $P(h)$
- Create a new generation of particles by resampling from the previous population:
  - Particles with high probability should be more likely to be sampled, and will therefore multiply.
  - Particles with low probability likely won't be sampled, and will therefore probably die out.
- The new particle's parameters are “jiggled” a little bit. This is how we search the space.
- Repeat this resampling process for several generations.

# Jiggling a Particle

- Perturb the translation term  $(x, y)$
- Perturb the orientation term  $\theta$
- Flip the state of an “addition” bit: one bit for each local shape
  - A value of 1 means this is a new addition to the world.
- Flip the state of a “deletion” bit: one bit for each world shape.
  - A value of 1 means this world shape has been deleted.

# So What's In A Particle?

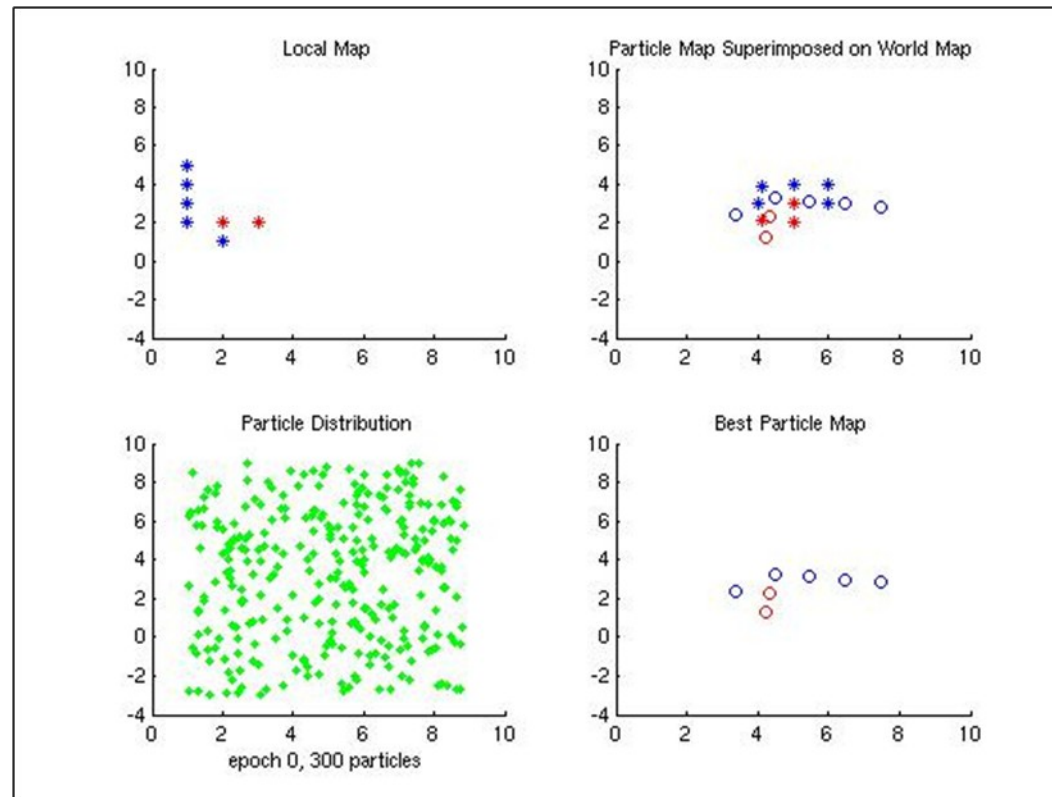
```
float dx, dy;  
  
AngTwoPi orientation;  
  
vector<bool> additions(numLocalShapes, false);  
vector<bool> deletions(numWorldShapes, false);
```

Parameters to adjust:

- Number of particles (2000)
- Number of generations (15)
- Amount of noise to add to dx, dy,  $\theta$
- Probability of flipping an add or delete bit

# Particle Filter Simulation: 2000 Particles

Zero Iterations



World Map

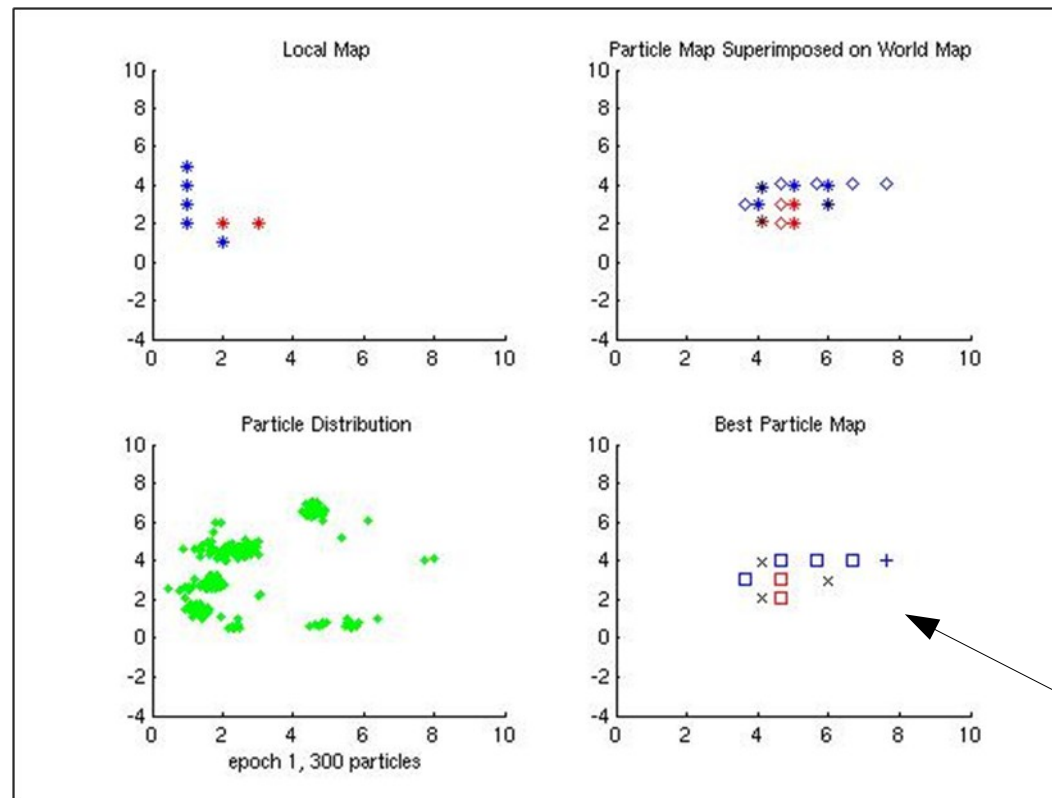


Rotated Local Map



# Particle Filter Simulation

## One Iteration



World Map



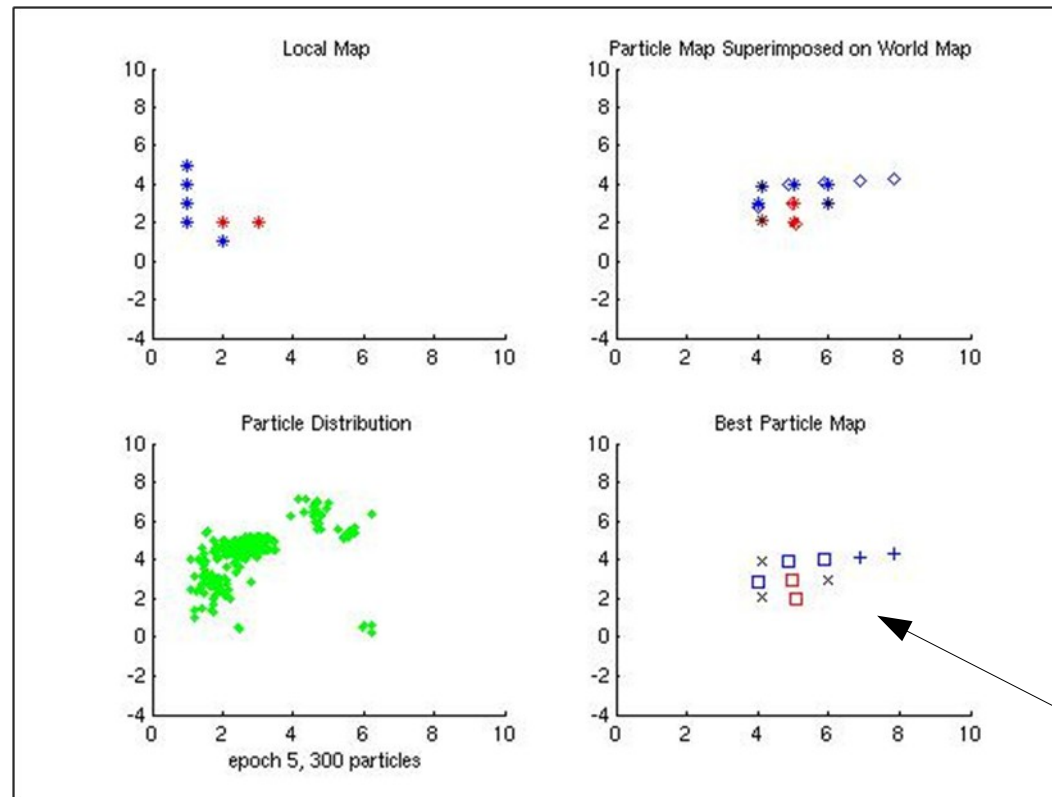
Rotated Local Map



+ means addition  
x means deletion  
□ means match

# Particle Filter Simulation

Five Iterations



World Map



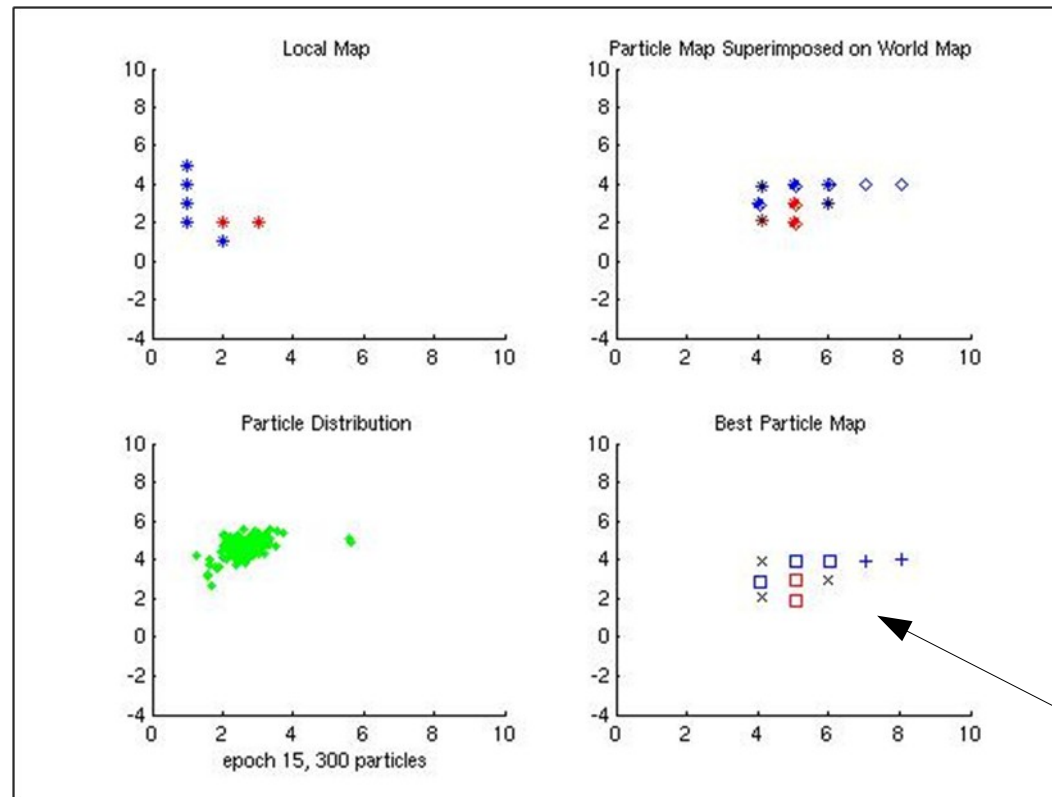
Rotated Local Map



+ means addition  
x means deletion  
means match

# Particle Filter Simulation

Fifteen Iterations



World Map



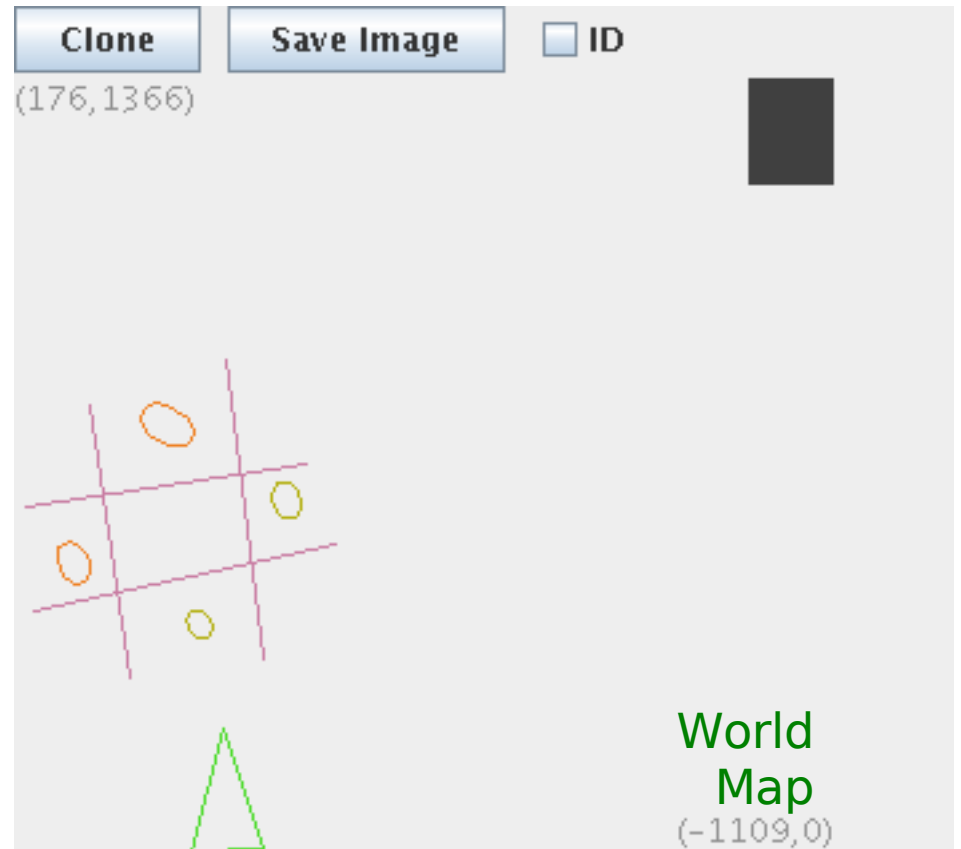
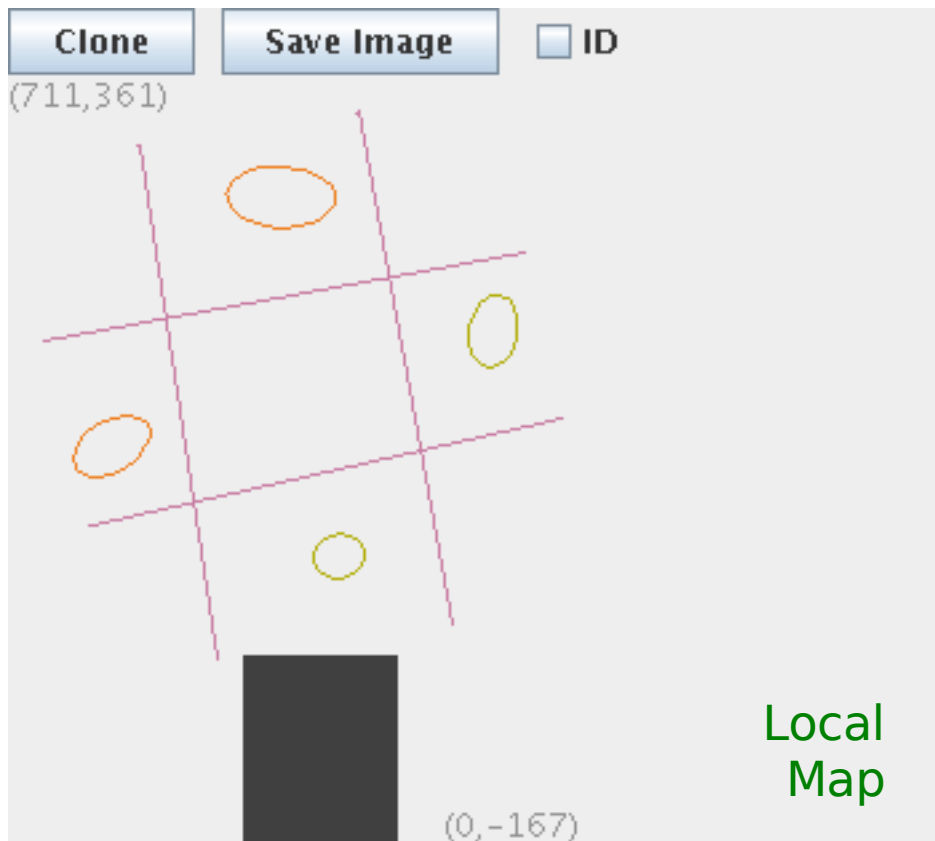
Rotated Local Map



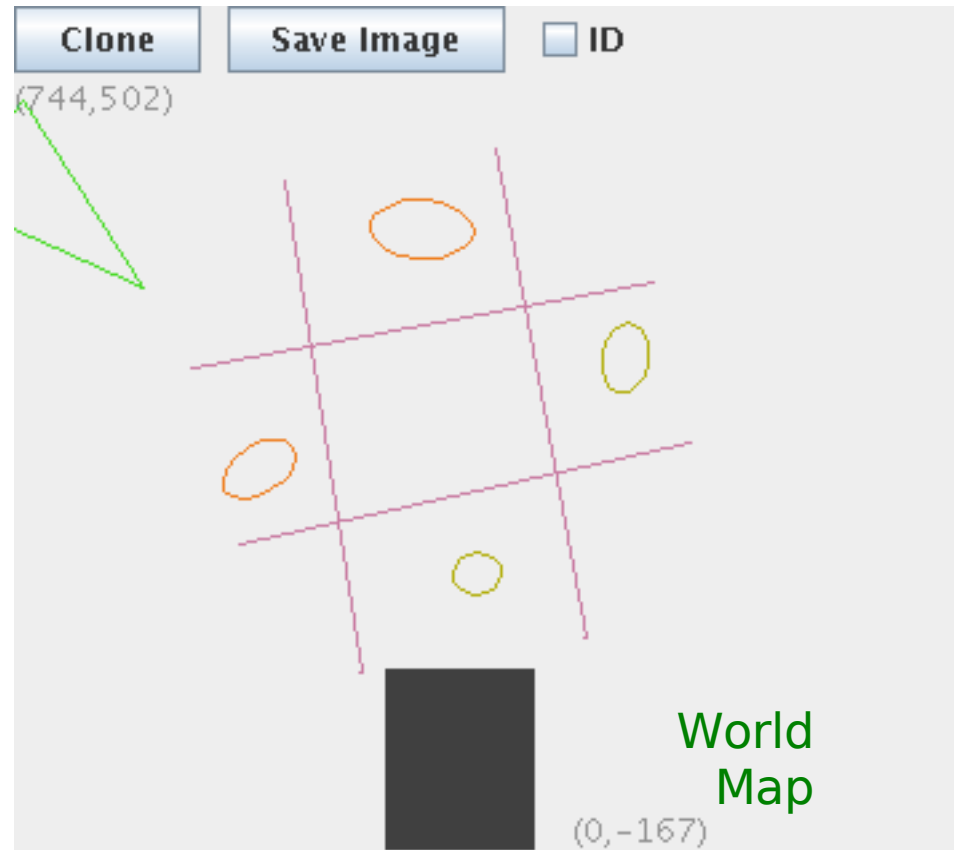
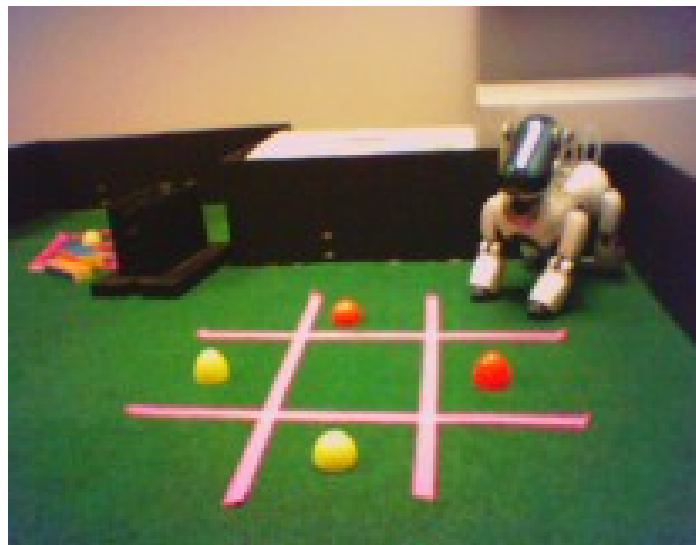
+ means addition  
x means deletion  
means match



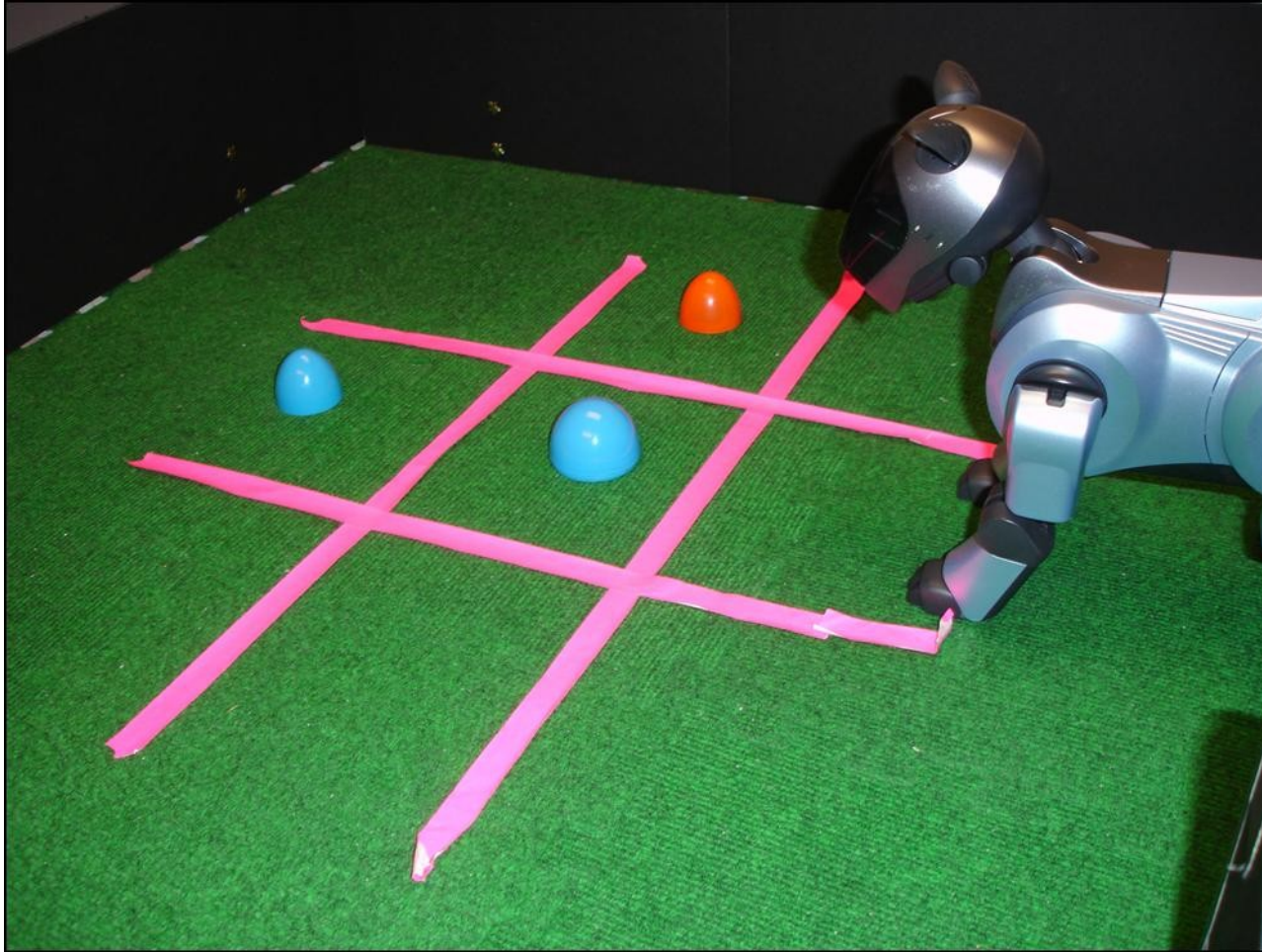
# Local and World Maps on the Dog



# Localization After Movement

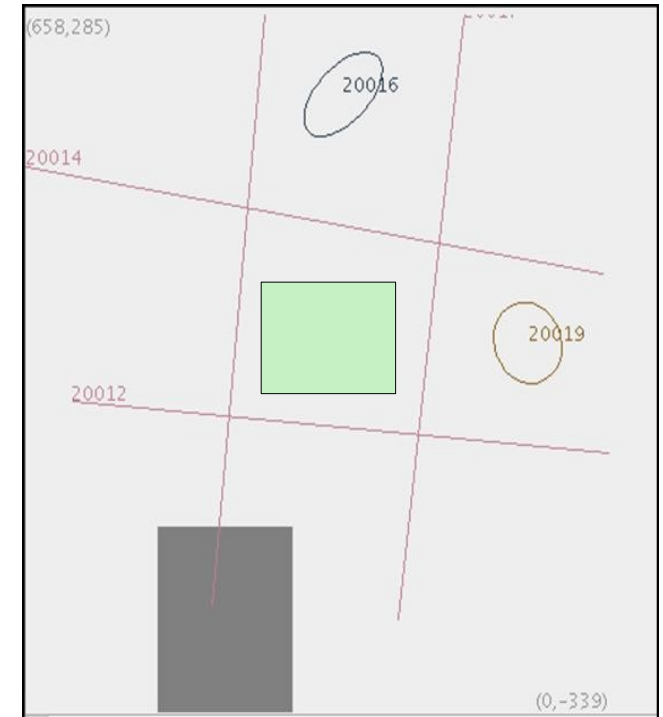
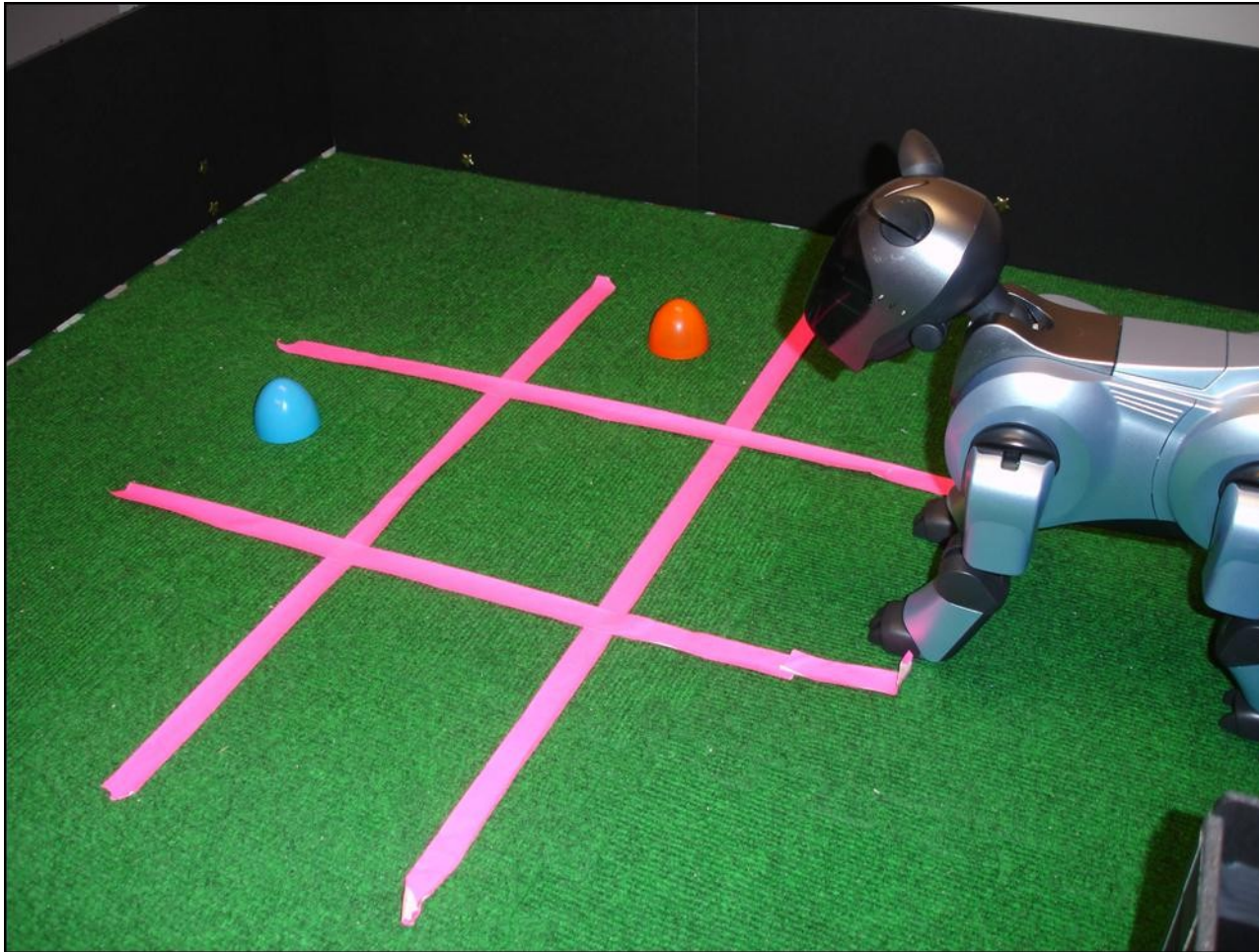


# Construct World Map



Three pieces on the board. Let's delete one.

# Delete a Game Piece

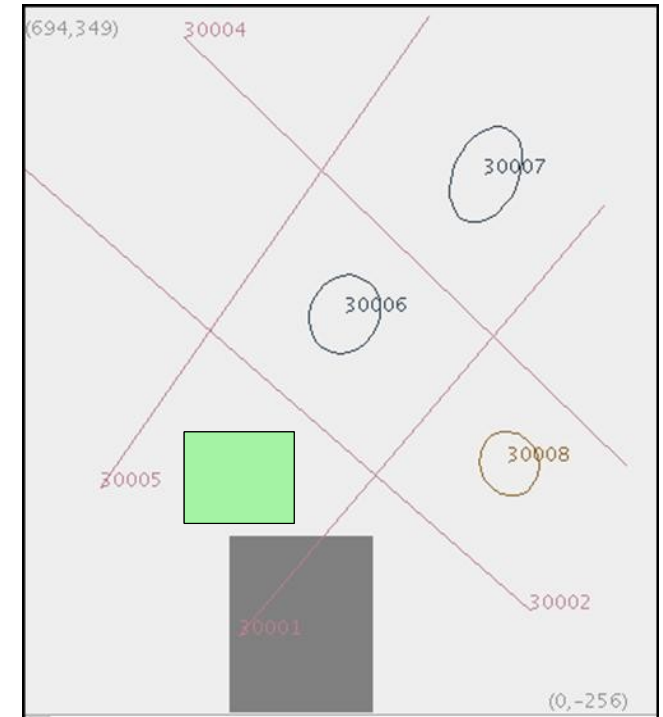
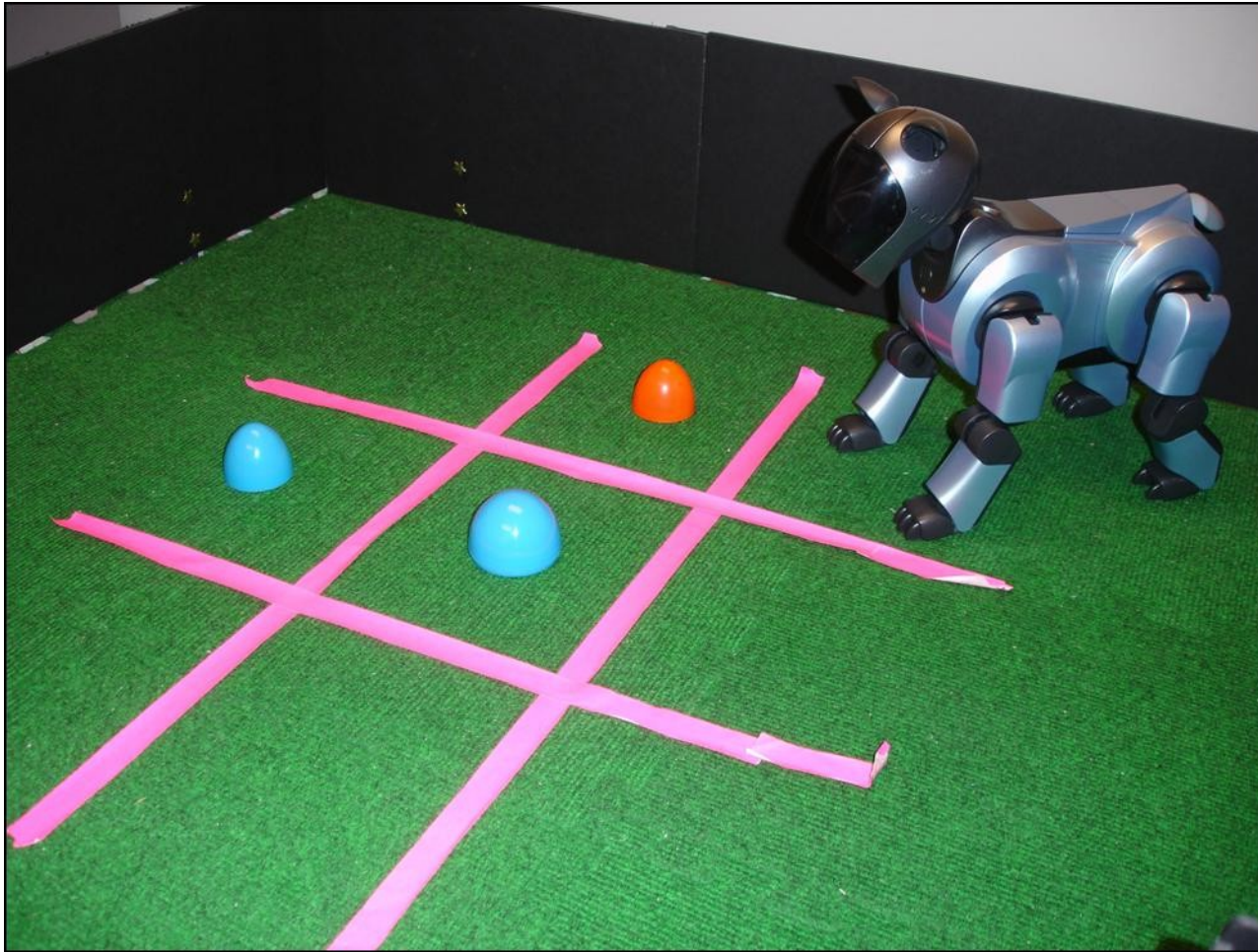


Actual change:  $dx = 0$  mm,  $dy = 0$  mm,  $\theta = 0^\circ$ , delete shape 30005

Particle filter:  $dx = 9$  mm,  $dy = 13$  mm,  $\theta = -0.2^\circ$ , delete shape 30005

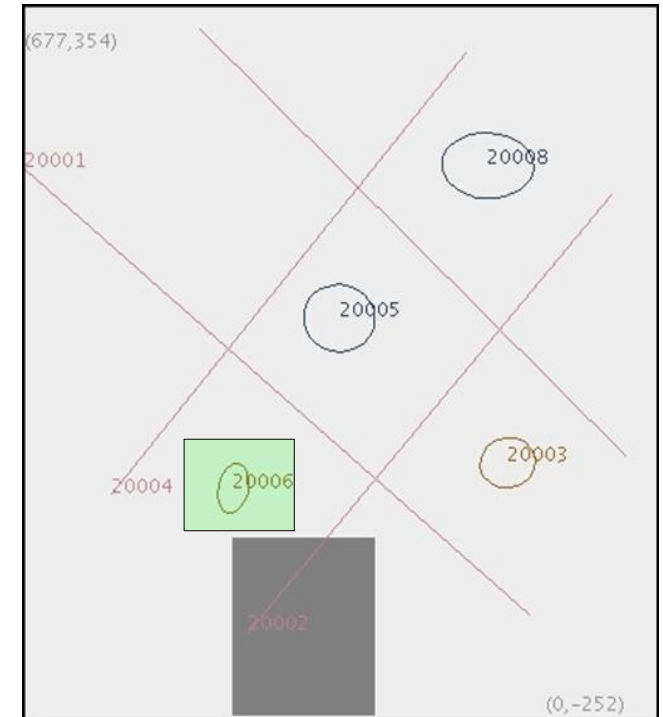
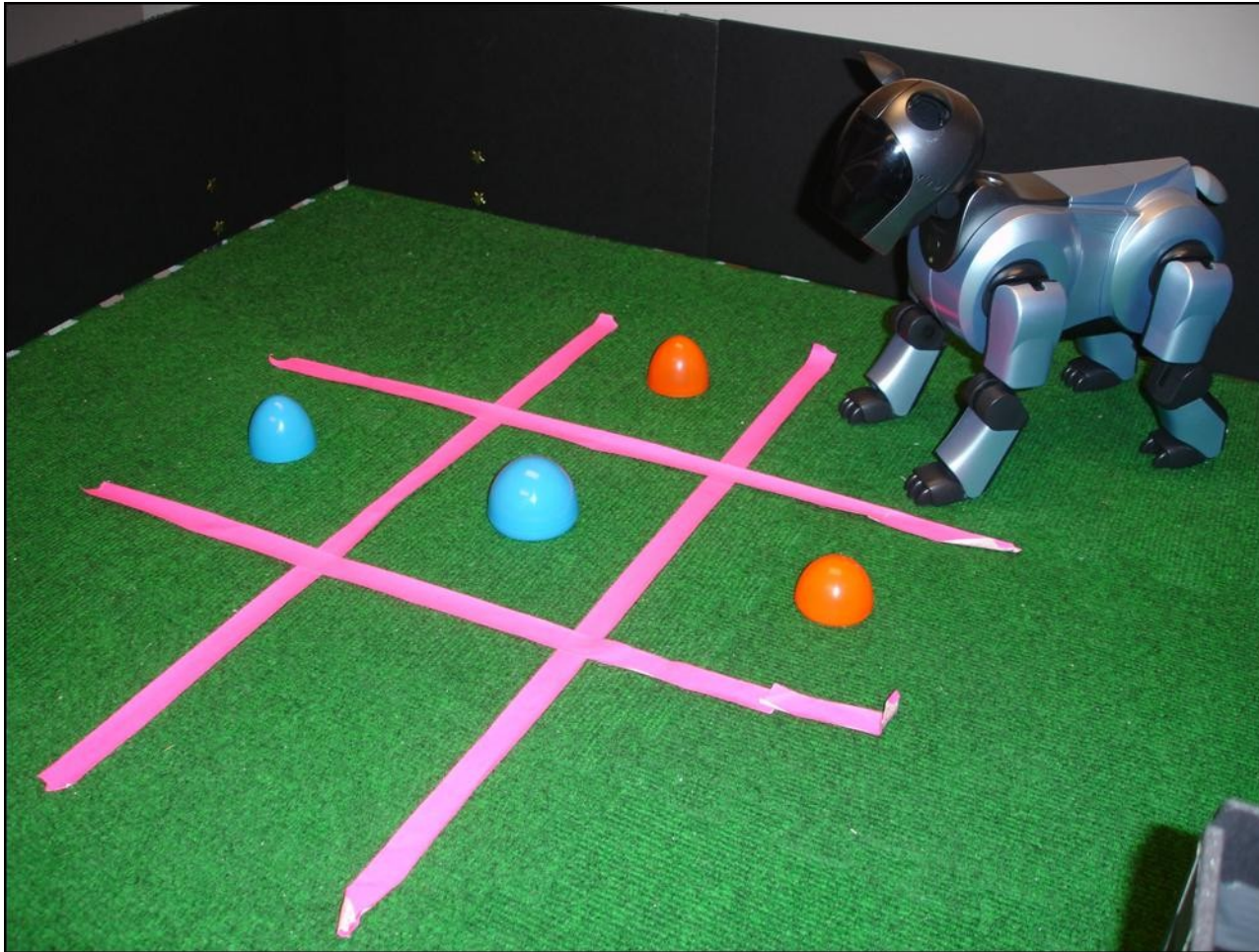


# Construct World Map



Three pieces on the board. Let's add one.

# Add a Game Piece

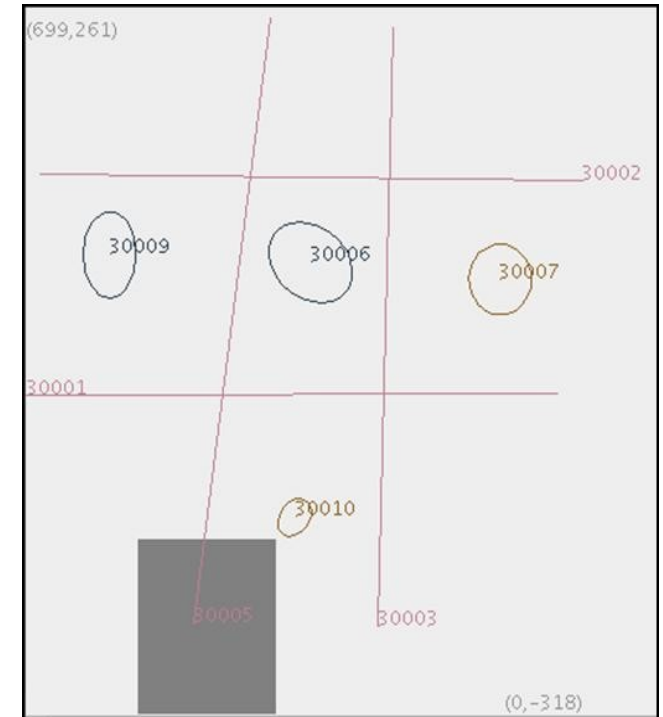
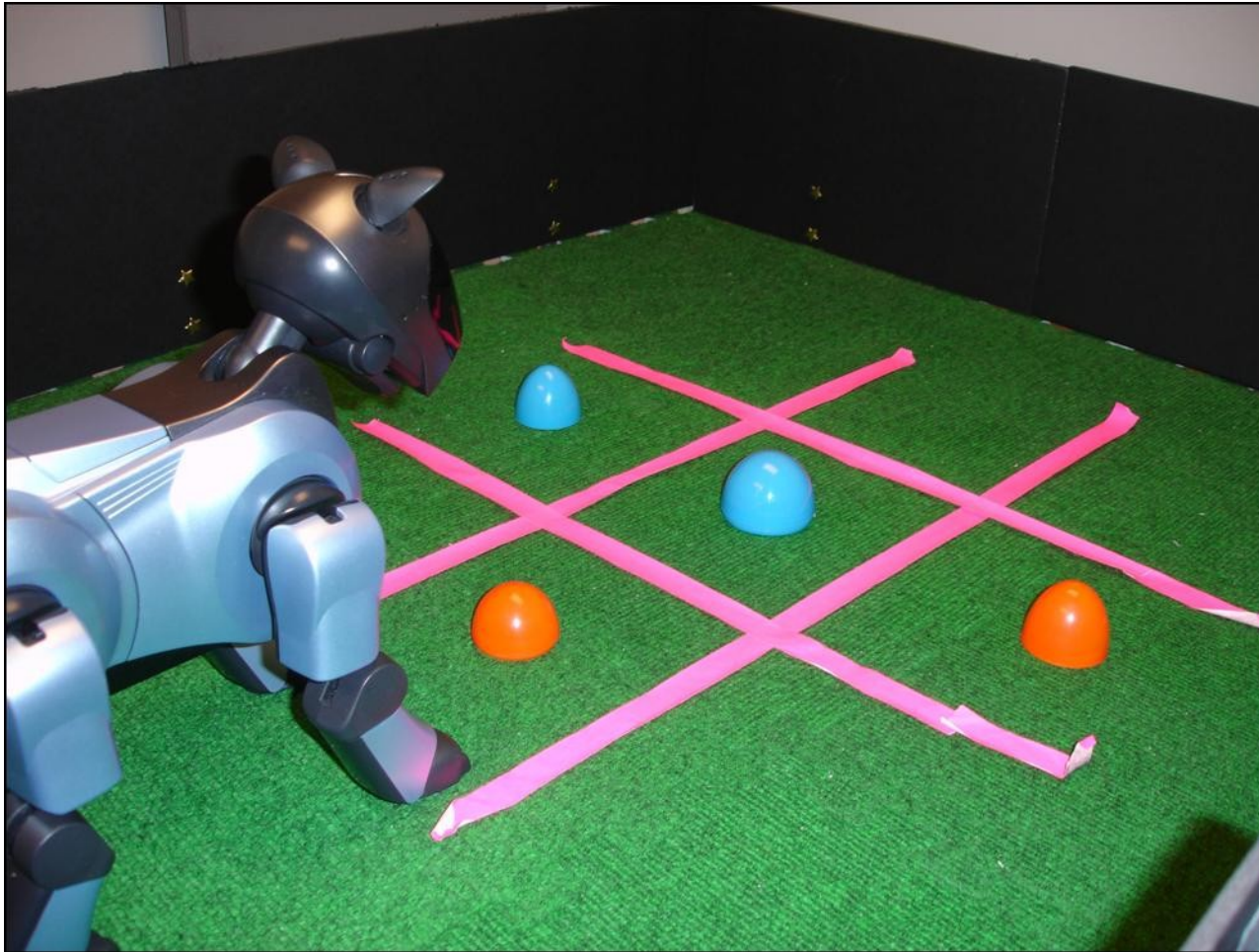


Actual change:  $dx = 0$  mm,  $dy = 0$  mm,  $\theta = 0^\circ$ , add shape 20006

Particle filter:  $dx = 2$  mm,  $dy = -.5$  mm,  $\theta = -0.6^\circ$ , add shape 20006

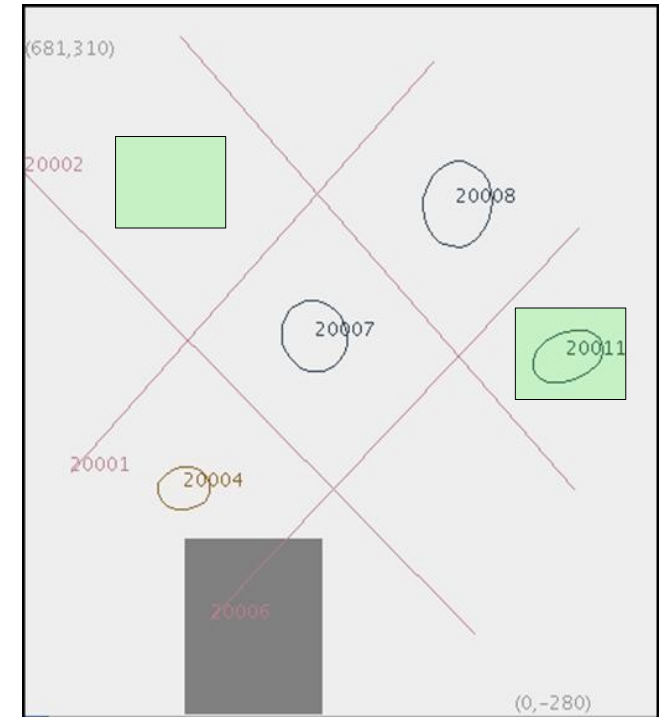
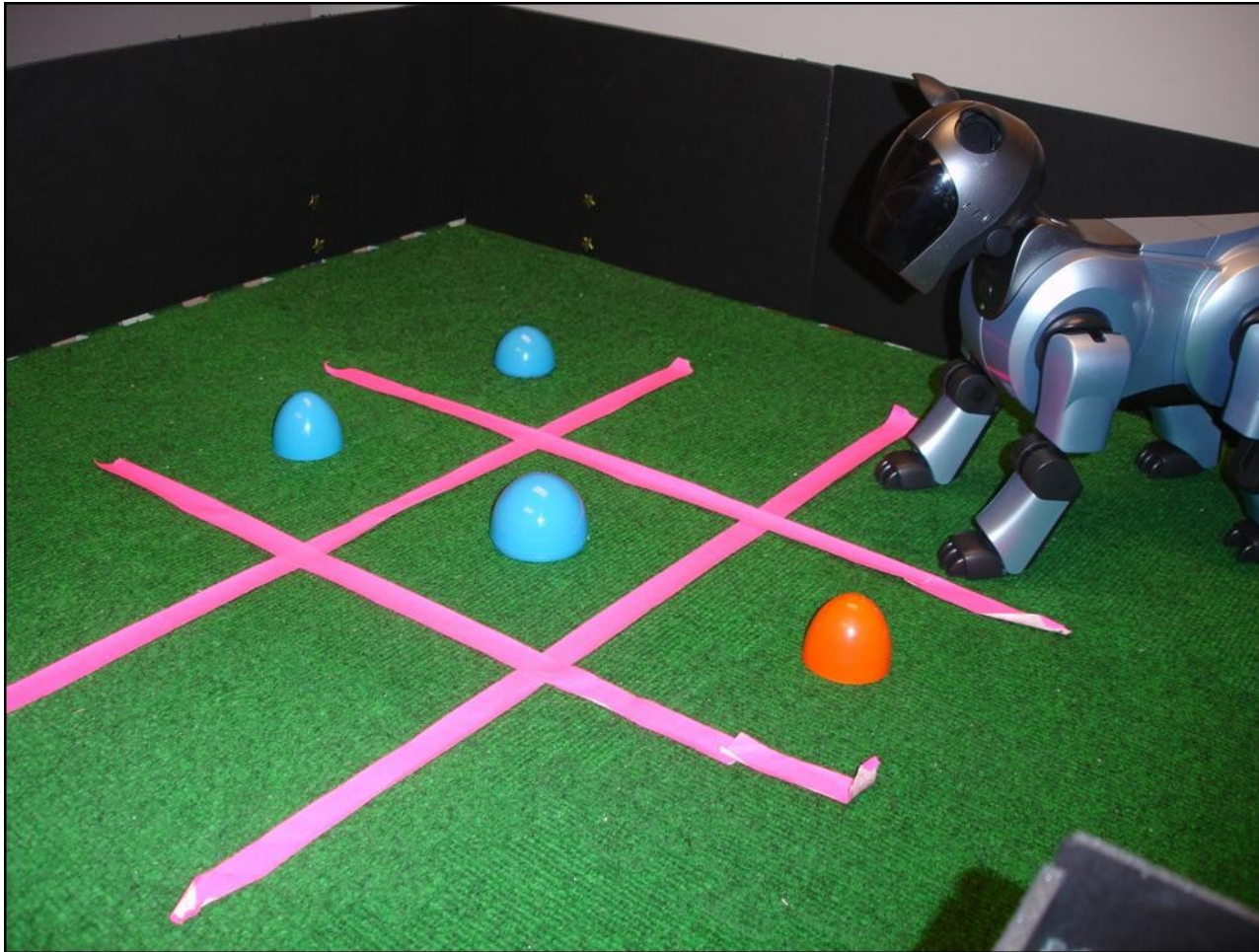


# Construct World Map



Four pieces on the board. Let's move, add, and delete.

# Change Position and Add/Delete



Actual change:  $dx = 670$  mm,  $dy = -260$  mm,  $\theta = 45^\circ$ , add 20011, del. 30010

Particle filter:  $dx = 678$  mm,  $dy = -306$  mm,  $\theta = 42^\circ$ , add 20011, del. 30010



# Invoking the Particle Filter

```
#include "DualCoding/DualCoding.h"
```

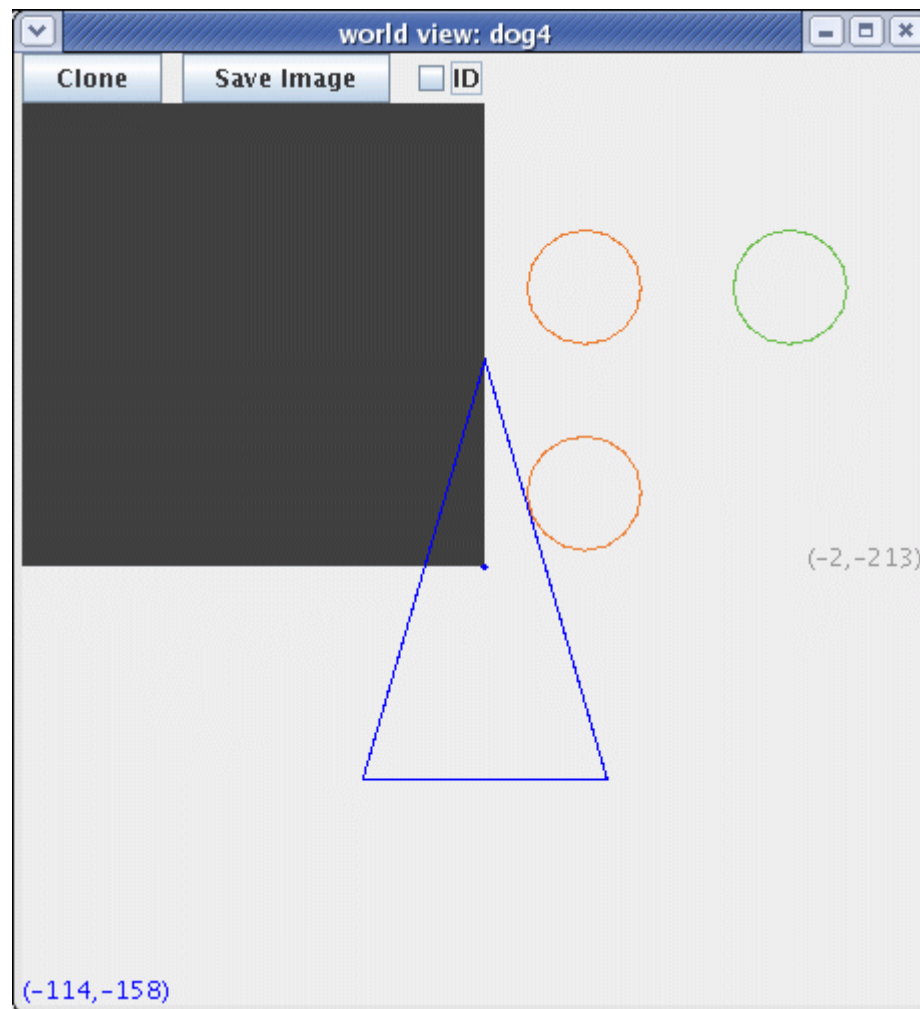
```
ShapeLocalizationPF filter(localShS,worldShS,1000);  
mapBuilder.executeRequest(...);
```

```
for (int i=0; i<10; i++)  
    filter.update()
```

```
filter.setAgent();  
filter.displayParticles();
```

# Particle Filter Demo

Set up a world with three landmarks (worldShS):



```
class ParticleDemo : public VisualRoutinesBehavior {
public:
    ParticleDemo() : VisualRoutinesBehavior("ParticleDemo") {}
```

```
void DoStart() {
    VisualRoutinesBehavior::DoStart();
```

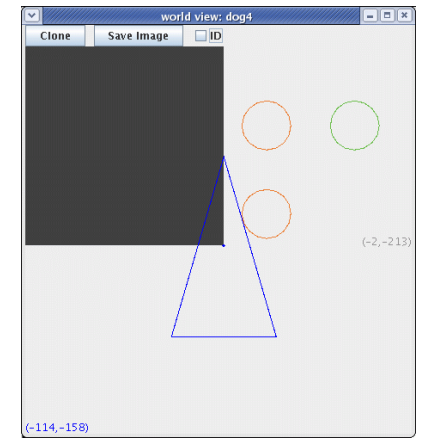
```
    const int orange_index = ProjectInterface::getColorIndex("orange");
    const int green_index = ProjectInterface::getColorIndex("green");
```

```
    // Build the world map
```

```
    NEW_SHAPE(orange1, EllipseData,
        new EllipseData(worldShS,Point(35,-50,0,allocentric),27.5,27.5));
    orange1->setColor(orange_index);
```

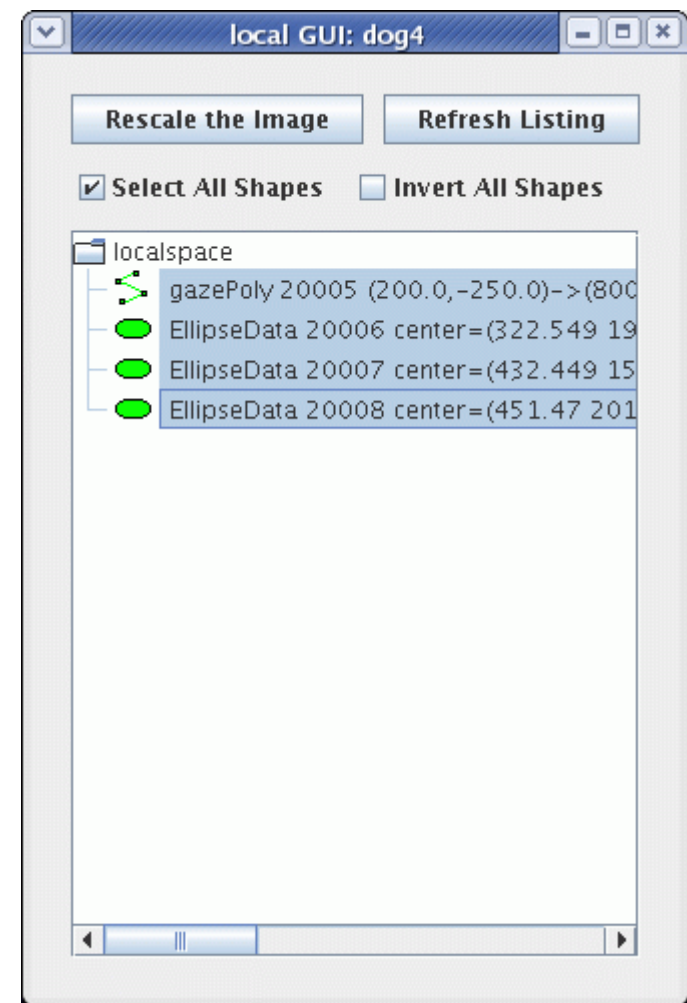
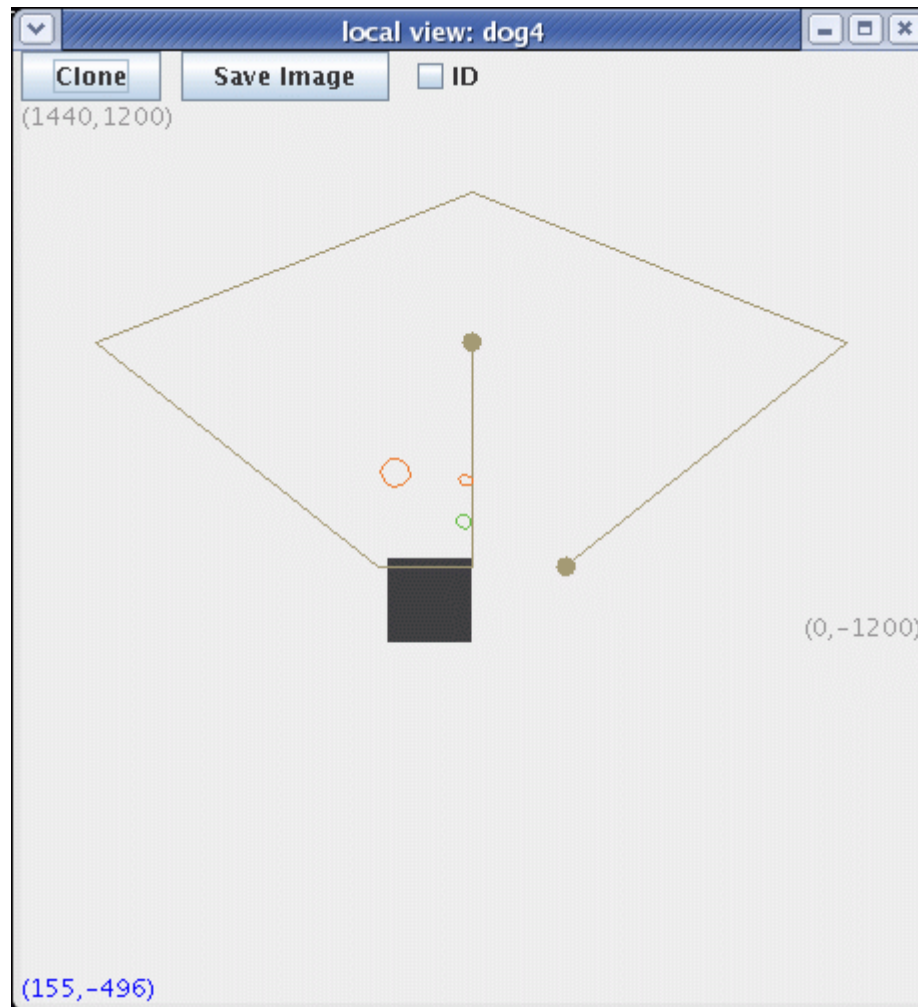
```
    NEW_SHAPE(orange2, EllipseData,
        new EllipseData(worldShS,Point(135,-50,0,allocentric),27.5,27.5));
    orange2->setColor(orange_index);
```

```
    NEW_SHAPE(green1, EllipseData,
        new EllipseData(worldShS,Point(135,-150,0,allocentric),27.5,27.5));
    green1->setColor(green_index);
```



# Move to New Location and Use MapBuilder to Look Around

Results are constructed in localShS:



```

// Build a local map from what we can see

localShS.clear();
NEW_SHAPE(gazePoly, PolygonData,
    new PolygonData(localShS, Lookout::groundSearchPoints(),
        false));

MapBuilderRequest mapreq(MapBuilderRequest::localMap);
mapreq.searchArea = gazePoly;
mapreq.doScan = true;
mapreq.pursueShapes = true;
mapreq.maxDist = 2000;
mapreq.clearShapes = false; // to preserve gazePoly

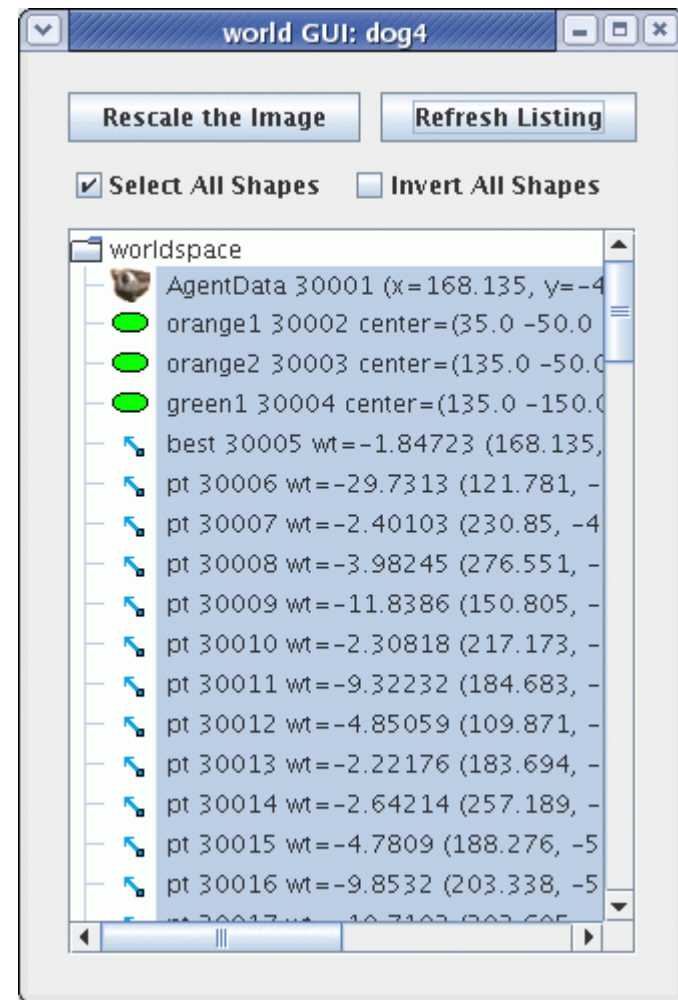
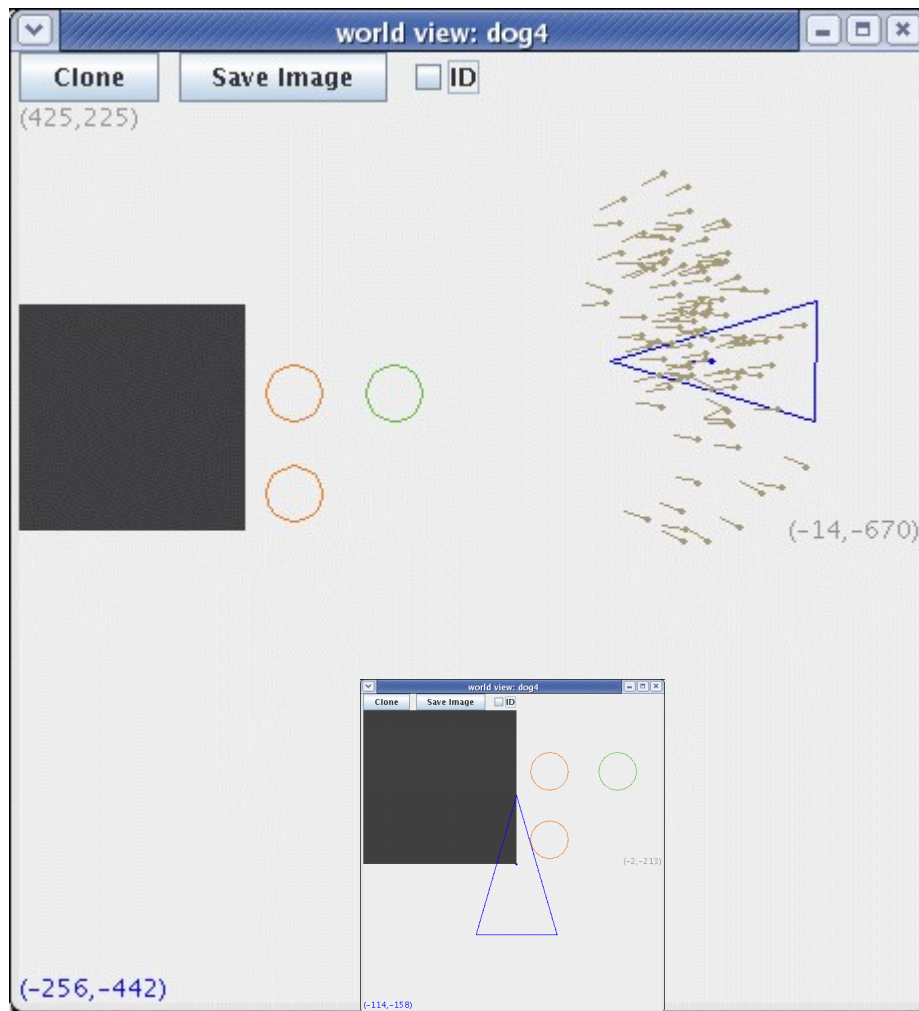
mapreq.objectColors[ellipseDataType].insert(orange_index);
mapreq.objectColors[ellipseDataType].insert(green_index);

unsigned int mapreq_id = mapBuilder.executeRequest(mapreq);

erouter->addListener(this, EventBase::mapbuilderEGID,
    mapreq_id, EventBase::deactivateETID);
}

```

# Use Particle Filter to Localize on the World Map



```
void processEvent(const EventBase &) {  
    // map complete; now we can localize  
  
    particleFilter->setMinAcceptableWeight(-3);  
    for (int i=0; i<5; i++)  
        particleFilter->update();  
    particleFilter->setAgent();  
    particleFilter->displayParticles();  
  
    cout << "Done!" << endl;  
}  
  
}; // class ParticleDemo
```