Navigating with the Pilot

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

Carnegie Mellon
Spring 2015

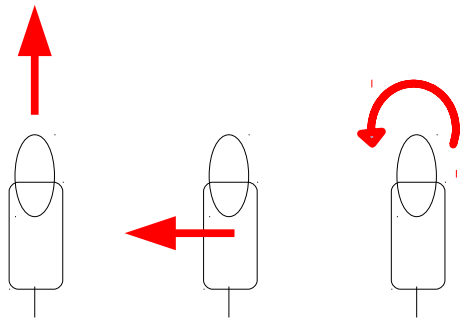
How Does the Robot Move?

- Multiple walk engines are incorporated into Tekkotsu:
  - “Wheeled walk” for the Create
  - XWalk engine by Ethan Tira-Thompson for the Chiara
  - CMPack '02 AIBO walk engine from Veloso et al. (CMU), with modifications by Ethan Tira-Thompson
  - UPennalizers AIBO walk engine from Lee et al. (U. Penn)
- Basic idea is the same for the legged walk engines:
  - Cyclic pattern of leg motions
  - Parameters control leg trajectory, body angle, etc.
  - Many different gaits are possible by varying phases of the legs
  - “Open loop” control: no force feedback
  - Can't adapt to rough terrain
  - Can move quickly, but not very accurately

WalkMC

- WalkMC is a motion command that uses the walk engine to move the robot.
  - For legged robots this means calculating leg trajectories.
  - For wheeled robots, calculate motor speeds.

- Walking is controlled by three parameters:
  - xvel: forward velocity
  - yvel: lateral velocity (strafing) requires legs or roller-wheels
  - avel: angular velocity (rotation)



WalkNode

- Subclass of StateNode
- Activates a WalkMC on start()
- Deactivates the WalkMC on stop()
- User can specify (x,y,a) velocity or displacement
- WalkNode(xvel, yvel, avel, time, WalkNode::VEL)
  - Velocities xvel, yvel in mm/sec; avel in rad/sec; time in sec
- WalkNode(xdisp, ydisp, adisp, time, WalkNode::DISP)
  - Displacements in mm and radians; time in sec.
  - Calculates velocity to reach destination in time
  - Use a time of 0 to request maximum velocity.

Requirements for Navigation

There's more to navigation than just walking.

The robot should be able to:

- Move in a specified direction at a specified speed
- Keep track of its location and heading
- Avoid cliffs
- Avoid obstacles or report collisions
- Localize using landmarks
- Plan paths using a world map
- Execute planned paths with error correction

The Pilot

- Higher level approach to locomotion.
- Specify effect to achieve, rather than mechanism:
  - Go to an object.
  - Search for an object.
- Specify policies to use:
  - Cliff detection (AIBO IR sensor)
  - Obstacle avoidance (turn off to knock down soda cans)
  - Localization procedure
- Experimental code; changing rapidly.

Pilot Request Types

- walk – essentially a WalkMC displacement request
- goToShape – plan path and travel to the location of a shape on the world map
- localize – look for landmarks and invoke the particle filter to update position estimate
- visualSearch – turn body to look for an object
- pushObject – plan path and push an object from its present location to a target location (in progress)
- setVelocity – set speed and go forever
- waypointWalk – provides Waypoint walk functionality
- *More functions are planned...*

PilotNode

- The PilotNode class provides an interface to the Pilot:
  - Creates a PilotRequest in the member variable pilotreq
  - Transmits the request to the Pilot
  - Listens for completion of the request
  - Posts a completion event when the request is finished
- Users typically create their own subclasses of PilotNode so they can fill in request parameters in the doStart() method.

Trivial Pilot Example

```
$nodeclass MyPilotDemo {

  $nodeclass Goer : PilotNode(PilotTypes::walk) : doStart {
    pilotreq.dx = 500;   // forward half a meter

  }

  $setupmachine{
     Goer =C=> SpeechNode("I have arrived")
  }



}

REGISTER_BEHAVIOR(MyPilotDemo);
```

Fields of a PilotRequest

- dx, dy, da — Displacement distance in mm or rad
- forwardspeed — Forward speed in mm/sec
  strafespeed — Sideways speed in mm/sec
  turnspeed — Turning speed in rad/sec
- collisionAction — What to do if a collision is detected
- landmarks — Landmarks to use for localization

Lots of other options; see the online documentation.

Not everything works for every robot.

Motion Nodes

- Shorthand for common Pilot calls.
- Defined in /usr/local/Tekkotsu/Crew/MotionNodes.h.fsm
- Common motions with convenient constructors:
  - WalkForward(distance)
  - WalkSideways(distance)
  - Turn(angle)
- For legged robots:
  - Rock(distance)
  - Sway(distance)
  - Twist(angle)

  etc...

Odometry

- From Greek word meaning “measuring a journey”.
- The robot estimates its position and heading by integrating position and heading changes over time.
- For most robot types, Tekkotsu assumes that position and heading changes match the commanded velocity.
- But the Create measures the changes itself:
  - Go to Root Control > Status Reports > Sensor Observer > Sensors
  - Click on Distance and Angle menu items
  - Go to Realtime View
  - Drive the robot around and watch the values change

Limitations of Odometry

Odometry can be inaccurate for several reasons:

- Acceleration/deceleration error: the robot may not always be moving at the commanded velocity.
  - Encoders on the wheels can measure actual distance traveled.
- Wheel or foot slip: varies with speed & type of surface.
  - An IMU (Inertial Measurement Unit) could be used to estimate true body displacement.
  - But IMU's also have accuracy limitations
- Integration error is cumulative: it increases over time.

Create Odometry Bug

- The Create reports angle values in integer degrees.
- Values are updated several times per second.
- Turning at low speeds gives an angle change of zero at each update because the software does not correctly accumulate fractional heading changes.
- This problem has been fixed in the Create 2, which reports separate encoder values for each wheel, as does the Kobuki from Yujin Robot (used in Turtlebot 2).

Solutions to the Odometry Bug

- Use a default angular velocity high enough for decent odometry, but not high enough to cause other types of problems (e.g., slipping due to high accelerations).

- Don't try to advance and turn at the same time.
  - Curved arcs generally require small da relative to dx, which triggers the odometry bug.

- New/experimental: try to use optic flow to supplement hardware odometry.

PilotDemo

- PilotDemo is a built in demo framework for demonstrating various features of the Pilot.
- It uses a simple, extensible command line interface.
- You can run it directly, or define your own behavior as a subclass of PilotDemo to make use of its features.
- To run it, go to:
  Root Control > Framework Demos >
  Navigation Demos > PilotDemo
- Use “msg” to send commands to the robot, e.g.,
  msg fwd 2000
- To use the PilotDemo class, you must do:
  #include “Behaviors/Demos/Navigation/PilotDemo.h”
  But note that the actual source file is PilotDemo.h.fsm

Robot Positioning

- Use PilotDemo commands to move the robot instead of using the Walk Controller.
  - F/B forward/backward 500 mm
  - f/b forward/backward 100 mm
  - fwd <*distance*>
  - L/R left/right 90 degrees
  - l/r left/right 10 degrees
  - turn <*angle*>
- Reason to avoid the Walk Contoller: too low or too high a velocity will mess up the robot's odometry.
- PilotDemo uses “good” velocity values so odometry works reasonably well.

The =PILOT=> Transition

- If you only care about when a Pilot request has completed, you can use a =C=> transition.
- But the Pilot reports additional information about what happened as a result of the request. To access this information you must use a =PILOT=> transition.
- Example: you may want to take special action if a collision was detected. Use:
  =PILOT(collisionDetected)=>
- A plain =PILOT=> transition will act as a default case if none of the other cases match.
- You should not use a =C=> transition if you have any =PILOT(...)=> transitions exiting the node.

Collision Detection

```
$nodeclass PilotLab3 {

  $nodeclass Backup : PilotNode(PilotTypes::walk) : doStart {
    pilotreq.dx = -90;              // negative displacement means back up
    pilotreq.forwardSpeed = 30;     // speeds are always non-negative
  }

  $setupmachine {


    forward: WalkForward(500)


    forward =PILOT(collisionDetected)=>
       SpeechNode("Ouch! I hit something.") =C=> Backup


    forward =PILOT=> SpeechNode("done")
  }
}
```

Collision Policies

- Defined in /usr/local/Tekkotsu/Crew/PilotTypes.h
- Collisions are detected either from the bump switches or from a motor stalling.
- Choices of collision action:
  - collisionIgnore — Do nothing
  - collisionReport — Post an event but continue
  - collisionStop — Stop (default action)
  - collisionReplan — Replan; not yet implemented

Push A Ball & Keep Going

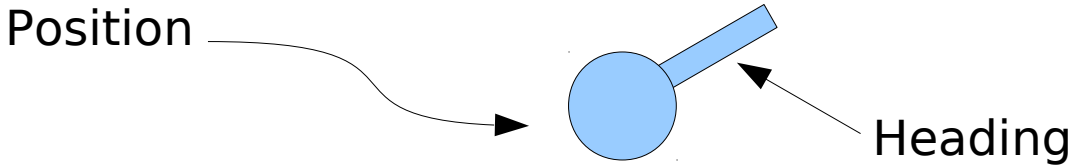
```
#include “Behaviors/Demos/Navigation/PilotDemo.h”

$nodeclass SlamAndScram : PilotDemo {

   $nodeclass Slammer : PilotNode(PilotTypes::walk) : doStart {
      pilotreq.dx = 2000;
      pilotreq.collisionAction = collisionIgnore;
   }

   $setupmachine{
      rundemo: Slammer =PILOT=> SpeechNode(“Got it”)
   }

}
```

Special node named **rundemo** can be activated using the Pilot's rundemo command after using other Pilot commands to position the robot.

Use **startdemo** instead to make the demo start automatically.

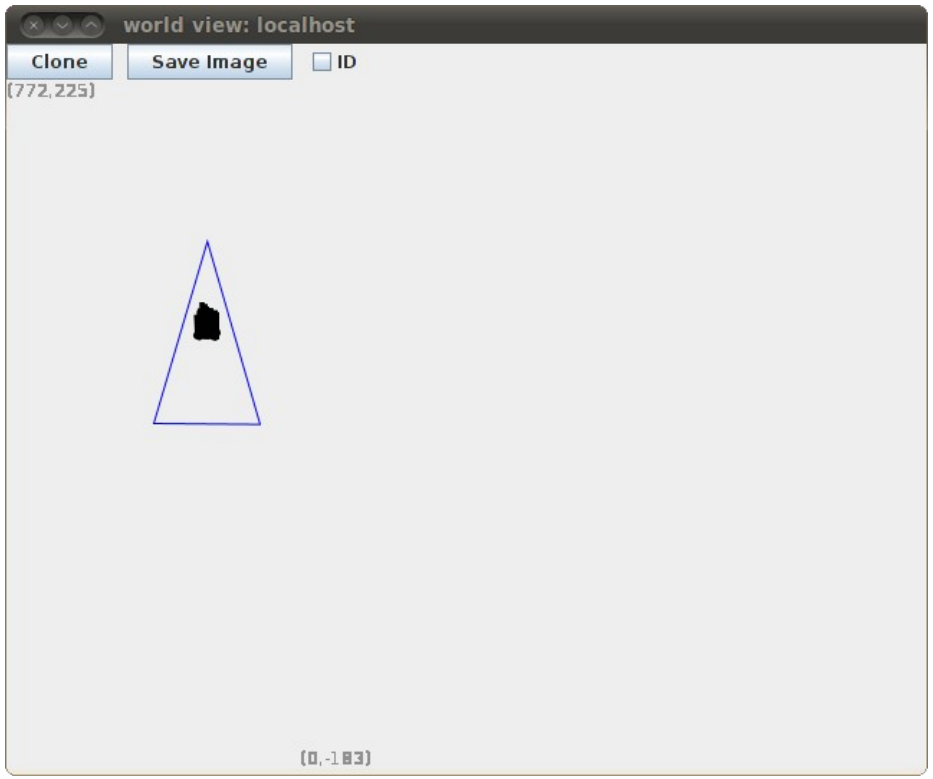
Localization Particles

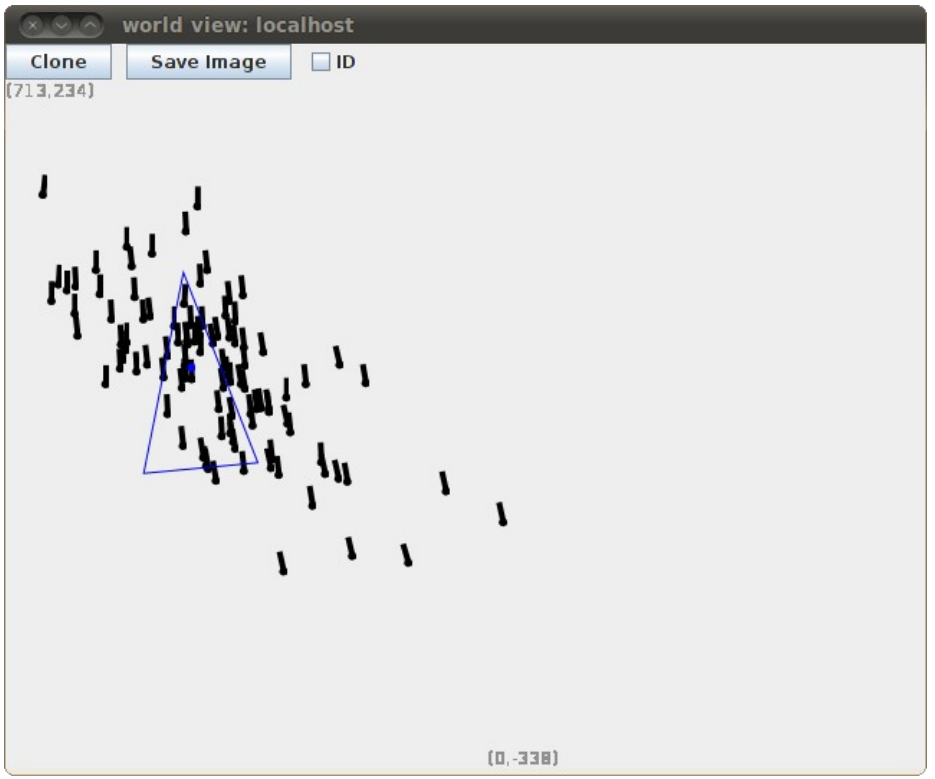
- A localization particle is an object that represents a hypothesis about the robot's position and heading.



- The Pilot's *particle filter* maintains a collection of these particles.
- The particles all start out at the robot's initial position and orientation.
- As the robot moves, the particles are updated by odometry, and noise from a *motion model is* added to simulate the effect of integration error.
- The particle “cloud” disperses, reflecting the growing uncertainty in the robot's pose.

Particle Dispersion Over Time





The robot and particles are displayed on the *world map*.

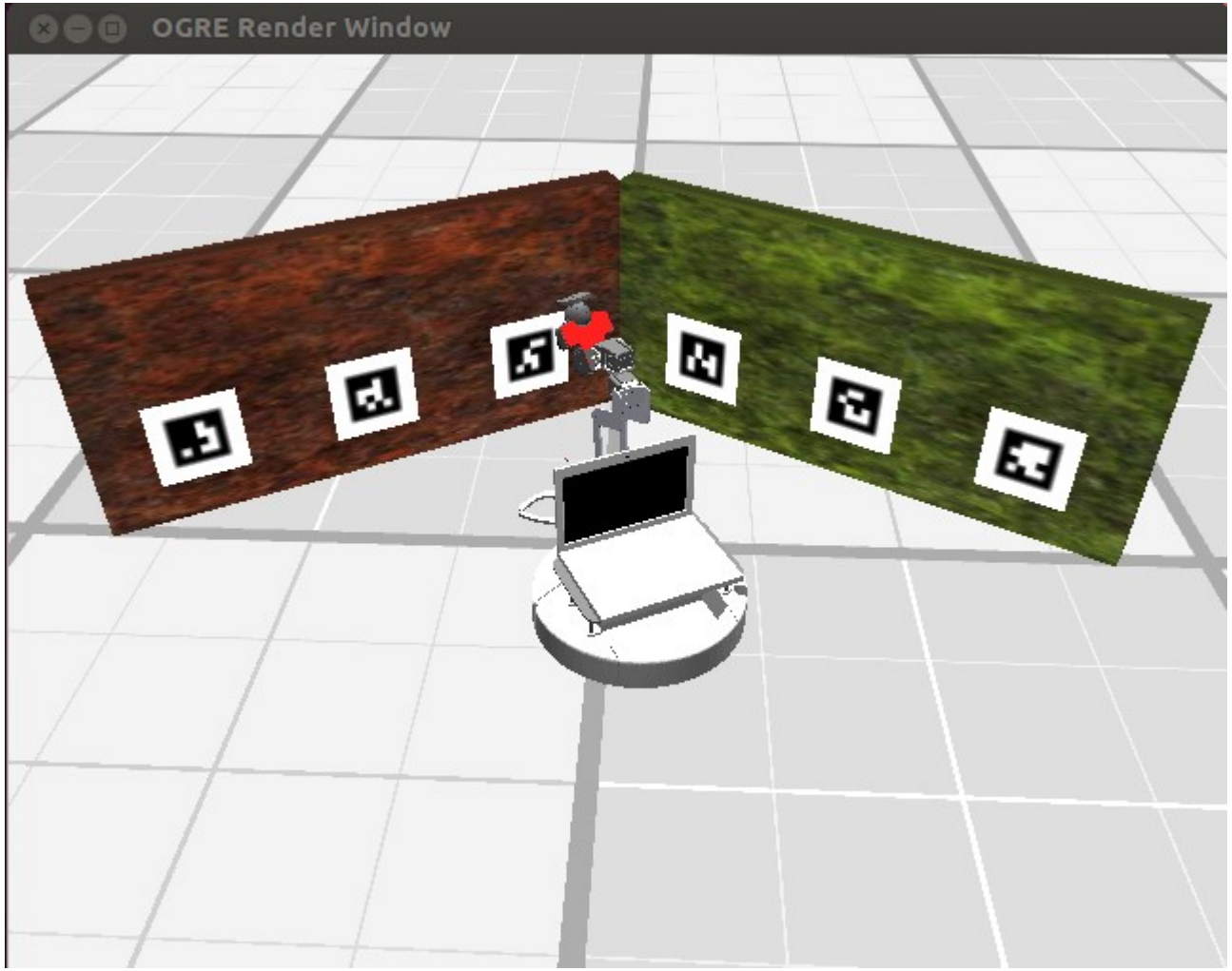
Click on “W” in the Sketch row of the ControllerGUI to bring up the map. Click on Refresh to update it.

Localization

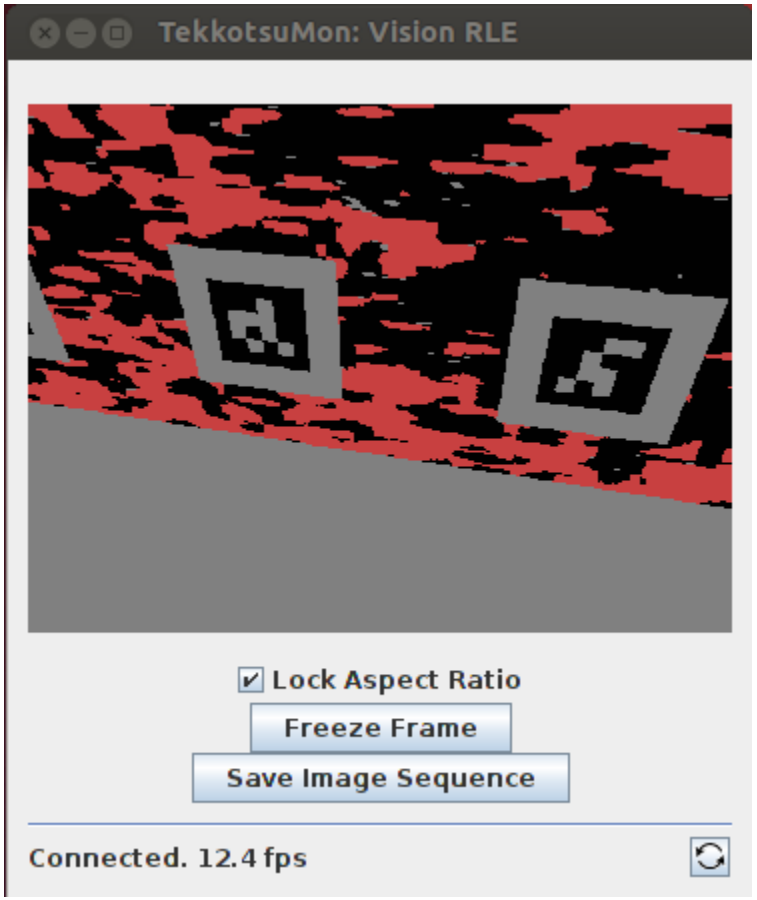
- As the robot moves, the particle cloud disperses and position uncertainty grows.
- How can we reduce our uncertainty?
- **Use visual landmarks and the particle filter to make a new estimate of position.**
- Requirements:
  - Landmarks such as AprilTags that the robot can recognize.
  - A world map giving the landmarks' locations.
- What does the Pilot do?
  - Look around for landmarks and build a local map.
  - Use particle filter to match local against world map.
  - Particles giving the best match get the highest score.

The VeeTags Demo

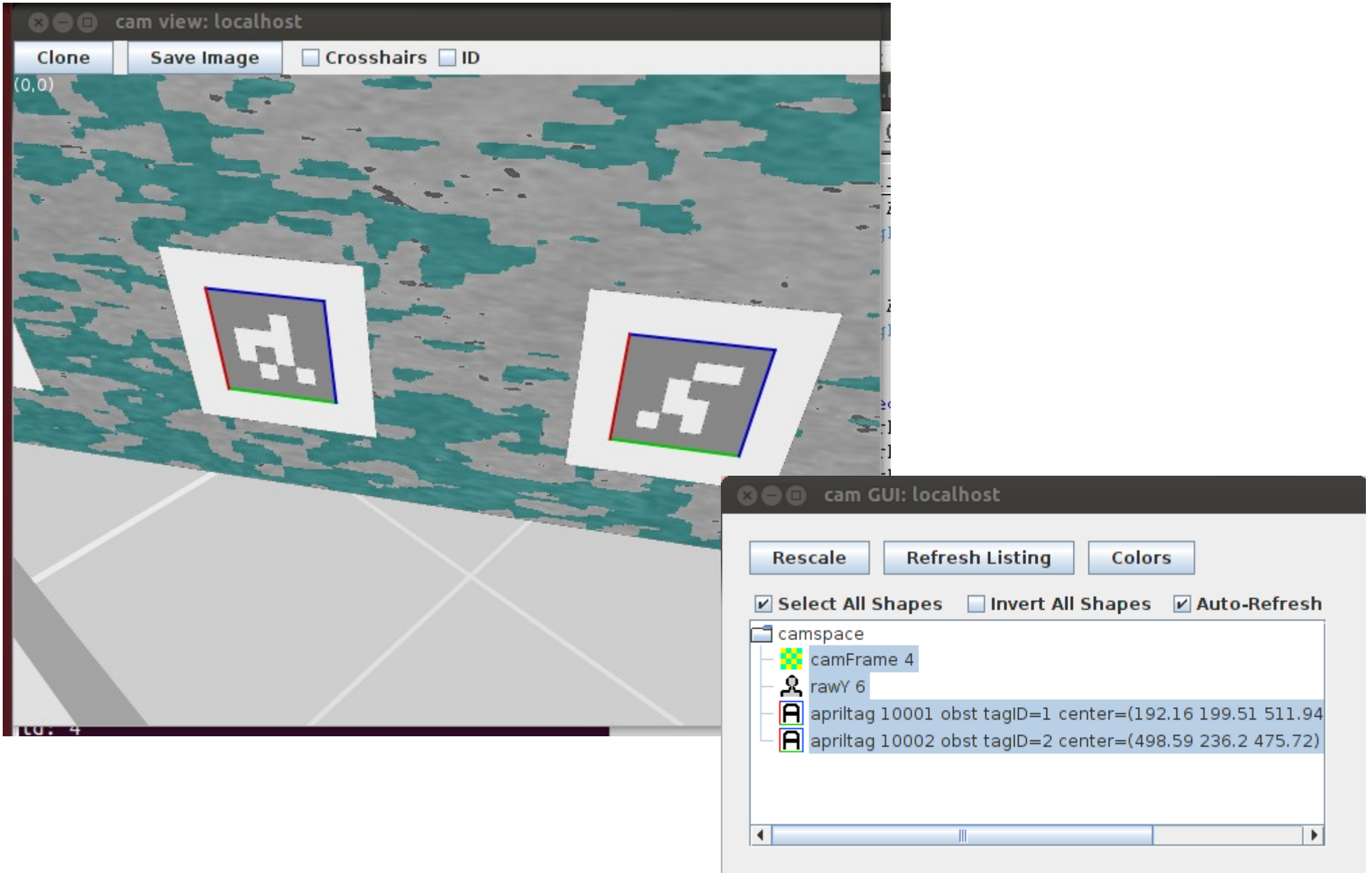
- Unique AprilTags serve as landmarks.
- Use PilotDemo's “loc” command to localize.



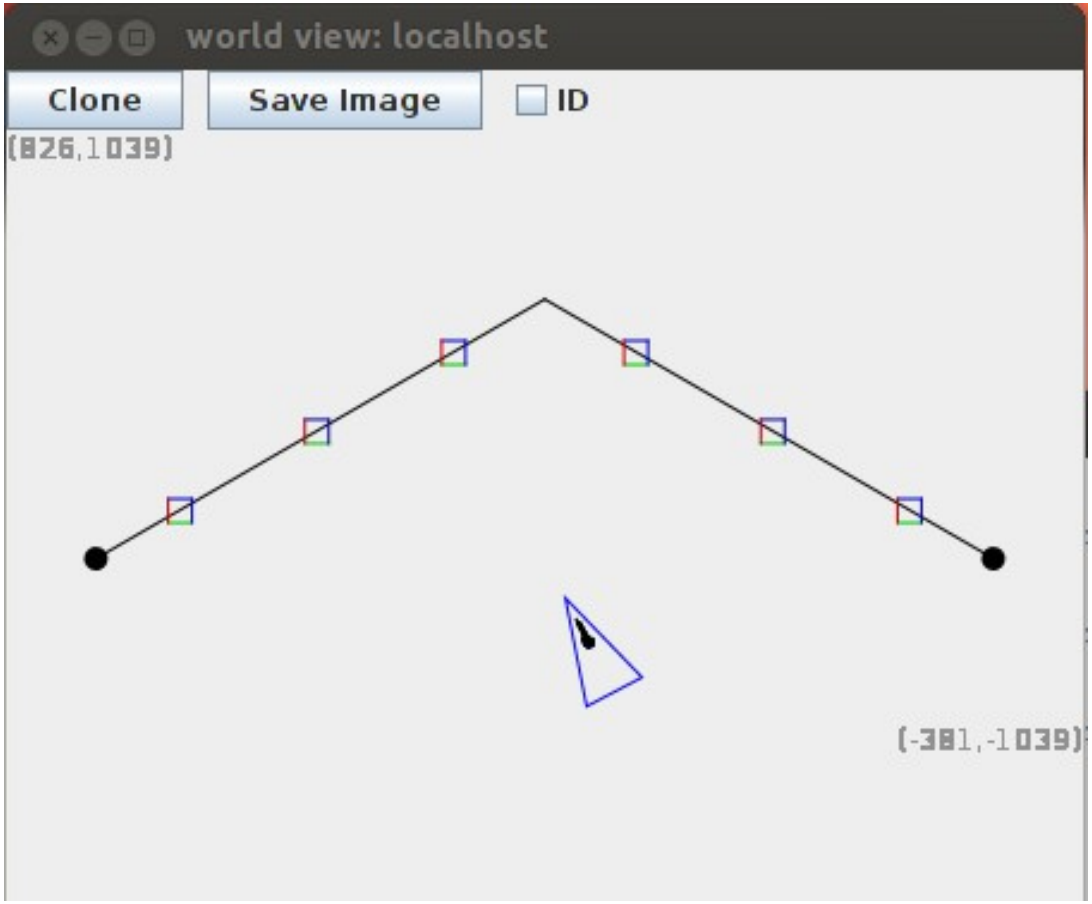
Camera Views: Raw and Color Segmented

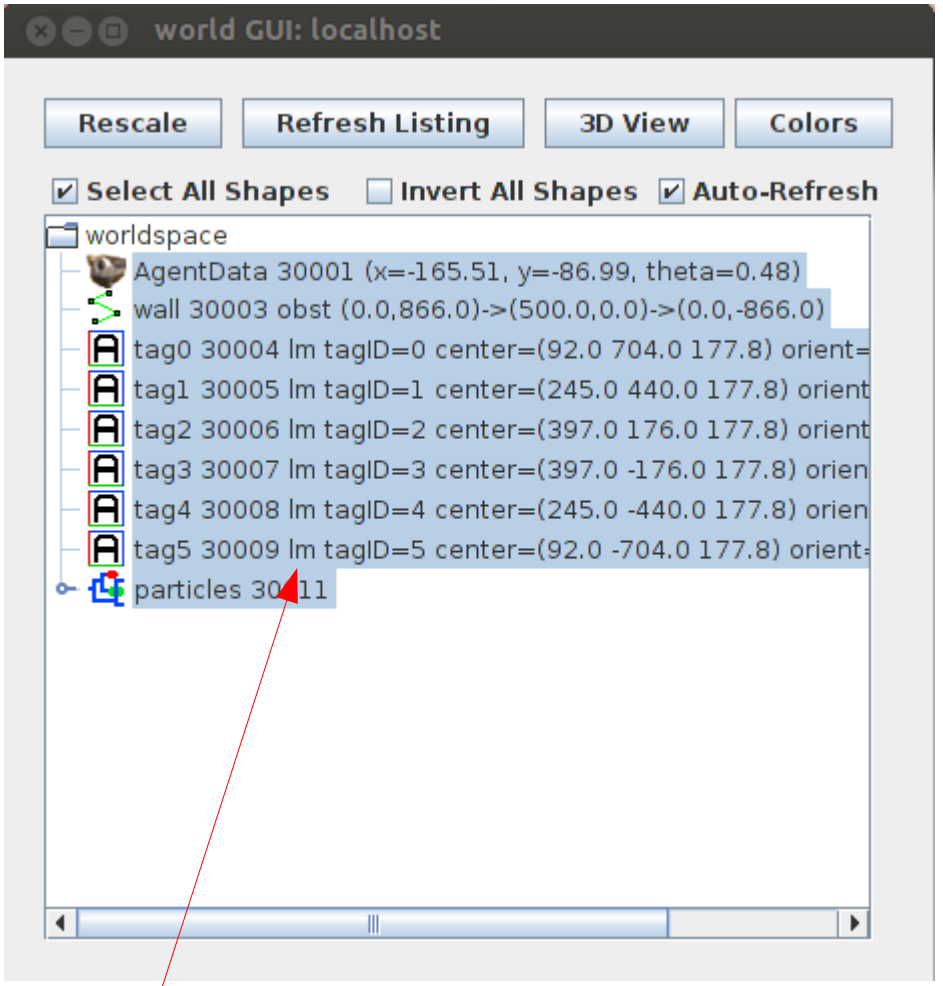


Camera Shape Space



World Map After Localization





AprilTags are marked as “lm” for “landmark”.

Retrieving the Robot's Pose

```
cout << “Robot is at x=“
     << theAgent->getCentroid().coordX()
     << “ y=”
     << theAgent->getCentroid().coordY() << endl;

cout << “Robot heading is “
     << theAgent->getOrientation() << endl;
```

Summary

- The Pilot is part of the Crew, and is responsible for:
  - Moving the robot through the world.
  - Tracking its position using odometry.
  - Localization using landmarks and a particle filter.
  - Path planning to avoid obstacles (to be covered later).
- Use a PilotNode to send a request to the Pilot.
- Use a =PILOT=> transition to check the results.
- Common cases:
  =PILOT(collisionDetected)=>
  =PILOT(noError)=>
  =PILOT(someError)=>
  =PILOT=>