Navigating with the Tekkotsu Pilot

Owen Watson[1] and **David S. Touretzky**[2]

[1]Department of Computer and Information Sciences
Florida A&M University
Tallahassee, FL 32301
owen1.watson@gmail.com

[2]Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
dst@cs.cmu.edu

Abstract

Tekkotsu is a free, open source software framework for high-level robot programming. We describe enhancements to Tekkotsu's navigation component, the Pilot, to incorporate a particle filter for localization and an RRT-based path planner for obstacle avoidance. This allows us to largely automate the robot's navigation behavior using a combination of odometry and landmark-based localization. Beginning robot programmers need only indicate a destination in Tekkotsu's world map and the Pilot will take the robot there. The software has been tested both in simulation and on Calliope, a new educational robot developed in the Tekkotsu lab in collaboration with RoPro Design, Inc..

Introduction

The Tekkotsu framework, available at Tekkotsu.org, features a set of four interacting software components, called the Crew, that help relieve programmers of the need to directly specify low-level behaviors (Touretzky and Tira-Thompson 2010). The Crew use a request/event architecture, where requests describe desired outcomes as opposed to a specific set of actions for the robot to take. Programmers specify a high level behavior, such as "walk to this location," or "find all the blue lines in the scene," in the form of a request data structure that each Crew member provides. The Crew members execute these requests on the programmer's behalf, posting events that signify the outcome. This paper describes navigation enhancements to the Pilot, the Crew member responsible for locomotion and navigation.

Providing programmers an interface to express navigation in terms of high level goals simplifies the construction of complex behaviors. To achieve this the Pilot incorporates a particle filter for localization and an RRT-based path planner for obstacle avoidance. Using odometry combined with landmark-based localization allows us to largely automate navigation in Tekkotsu.

The software has been tested both in simulation and on Calliope, a new educational robot developed in the Tekkotsu lab in collaboration with RoPro Design, Inc. The navigation enhancements integrate well with Calliope, which provides a camera on a pan-tilt mount (useful for locating landmarks), wheel-based mobility, and sufficient on-board processing power to handle data-intensive tasks.

The Tekkotsu Crew

Each member of the Crew has its own specialty, summarized below.

The Lookout

The Lookout manages the robot's sensor package and provides camera images and possibly other types of sensor observations, such as rangefinder scans. A robot's sensor package is usually mounted on some type of moveable "head" which the Lookout can point.

The MapBuilder

The MapBuilder is responsible for building representations of the world. The MapBuilder calls on the Lookout whenever it needs to obtain images. Depending on the application, the user may choose to work in any of three coordinate systems: camera space, local (body-centered) space, or world space (Touretzky et al. 2007). The MapBuilder provides methods for recognizing shapes such as lines or ellipses, generic blobs, or navigation markers. Using shapes that it extracts from camera images, the MapBuilder can construct local or world representations. It invokes Tekkotsu's kinematics engine to determine the camera pose based on the robot's current posture, then performs coordinate transformations from camera to local space. Building a body-centered representation may require multiple camera images due to the robot's fairly narrow field of view

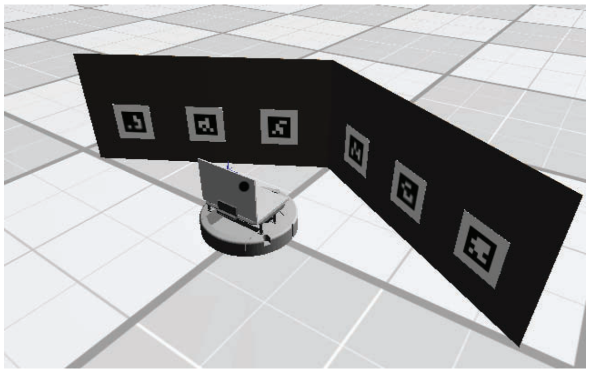


Figure 1: A Create/ASUS robot examining AprilTag navigation markers in the Mirage simulator environment. This particular virtual world consists of six distinct AprilTags affixed to a vee-shaped barrier.

The Pilot

The Pilot interacts with the MapBuilder to find landmarks for use in navigation. In some modes it may also communicate with the Lookout to obtain rangefinder readings.

The Grasper

For those robots that include an arm, the Grasper is the Crew member responsible for manipulation. The Grasper invokes an inverse kinematics solver to calculate the joint angles necessary to put the "fingers" at a specified point in space, and a path planner to calculate arm trajectories (Coens 2010). The Grasper is the only member of the Crew that is not used in navigating.

The World and Local Maps

World maps are the robot's representation of the physical world in an allocentric "shape space," part of Tekkotsu's "dual-coding" vision system (Touretzky et al. 2007). These maps describe the world's boundaries, landmarks, and obstacles.The world boundary for a maze-type environment is defined by constructing a polygon shape in the world shape space. The robot is also represented in this world map, using an "agent" shape with both position and orientation attributes. The placement of landmarks in the world map gives the particle filter a means to evaluate hypotheses about the robot's position based on what the Pilot is currently seeing. The path planner uses the world map to search for a collision-free path from the robot's current position to the desired destination.

Local maps represent the spatial configuration of the objects the robot currently sees, in body-centered coordinates. Due to the camera's limited field of view (roughly 60° for a webcam vs. 200° for a binocular human), local maps are typically constructed incrementally by the MapBuilder from multiple camera images by moving just the head. Once the robot moves its body, the local map is no longer valid. However, on the Create/ASUS robot, the forerunner of Calliope, which lacks a pan/tilt and instead relies on the netbook's built-in camera, the Pilot will rotate the body if necessary to acquire enough landmarks for localization. It will then rotate back to the original heading. The heading error induced by this operation, averaging 5 to degrees, was judged a reasonable tradeoff in return for being able to see more of the world.

Functions Of The Pilot

The Pilot provides a unified interface for instructing robots to navigate through the world, abstracting away the details of how each type of robot moves and how it acquires landmarks. Before the advent of the Pilot, users were responsible for doing their own path planning, obstacle avoidance, and localization. Now users are able to invoke these functions with minimal effort, and create routines that can be used on all platforms supported by Tekkotsu.

Locomotion

The fundamental function of the Pilot is to move the robot's body through the world. This includes both translational motion (forward and backward), sideways motion (for holonomic robots), and rotation. In the future the Pilot will also accommodate requests for postural changes, e.g., legged robots can rear up, hunker down, lean in a specific direction, etc. The Pilot also allows users to specify the speed at which to move and/or the distance to travel. It is the Pilot's responsibility to interface with the walk engine that controls each type of robot.

Odometry

The Pilot keeps a running estimate of the displacement and heading of the robot. On platforms that provide hardware odometry, such as the iRobot Create, it makes use of this. Otherwise it does dead reckoning by integrating the robot's velocity vector over time. The particle filter's motion model makes use of this odometry information to drag the particles along as the robot moves, adding noise to reflect increasing uncertainty over time, until the next resampling occurs.

Localization

As previously described, the Pilot uses the MapBuilder to acquire landmarks and a particle filter to maintain its position estimate. A variety of landmark types can be used, but we find the most reliable are AprilTags (Olson 2010), fiducial markers inspired by Augmented Reality Tags. See Figure 1.

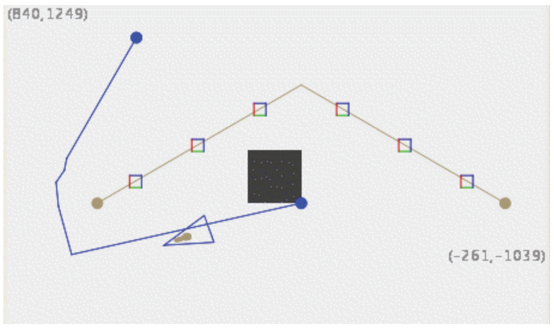


Figure 2: The world shape space displayed using Tekkotsu's SketchGUI tool. The Pilot has planned a path from the robot's starting position to a goal location on the other side of the barrier. The small squares show the locations of AprilTags. The triangle shape indicates the robot's estimated position and heading. A cloud of particles is visible within the triangle.

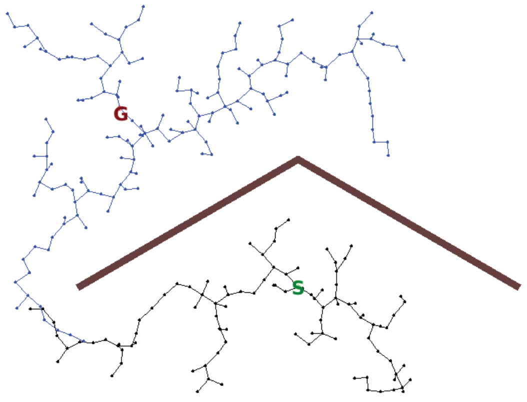


Figure 3: The RRT-Connect path planner grows search trees simultaneously from the start (S) and goal (G) locations, until they meet. This figure, showing actual Tekkotsu path planner output, was prepared in Matlab, but we are presently extending Tekkotsu's SketchGUI to provide a similar display, to help students visualize the RRT algorithm.

Localization is a discrete activity for the Pilot, for two reasons. First, we're using identifiable landmarks rather than a continuous stream of raw sensor readings such as a sonar array or laser scanner would provide. Therefore we must point the camera at the locations where landmarks are expected to appear and measure their distances and bearings, one at a time. Second, if the robot uses a webcam that suffers from motion blur, as most do, it cannot move and localize at the same time. Localization is therefore a planned activity, so that the Pilot can optimize the locations where it occurs. For example, if the robot is about to turn a corner, it makes sense to delay localization until the turn has been made.

Planning

When the user specifies a destination, the Pilot first plans a path to that location that avoids world boundaries (walls and cliffs) and obstacles. For this it uses the RRT-Connect algorithm (Kuffner and Lavalle 2000). Figure 2 shows a path to a goal location behind the barrier of Figure 1, and Figure 3 shows the search trees that generated this path.

Navigation is goal-directed locomotion informed by perception. The Pilot constructs a *navigation plan* as a series of motion segments along the planned path, interspersed with localization steps. Path analysis is used to optimize the placement of localization operations. For example, long straight segments are broken up by localizations to guard against the robot drifting. Once the robot is on the far side of the barrier there will be no landmarks visible, so the Pilot must rely on dead reckoning from then on, and further localization steps are omitted. In situations where this is not acceptable, the path planner would have to consider landmark availability as part of its route planning process. We leave this to future work.

Our current mode of using the Create triggers a firmware bug that prevents accurate odometry for slow turns. Therefore, the navigation planner does not try to smoothly round out the corners of a path. Instead it generates separate steps for heading changes vs. translational motion.

Obstacle and Cliff Detection

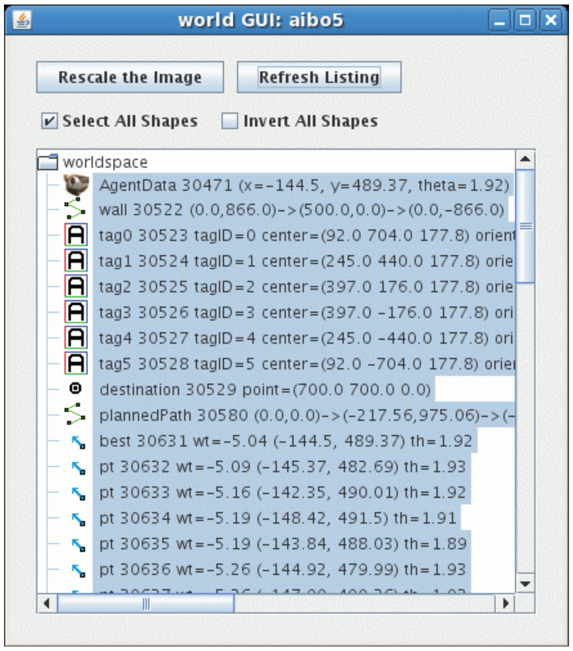
We provide a variety of modes of obstacle and cliff detection. On the Create we use the left and right bump sensors to detect collisions, plus the motor current sensors to detect when the robot is encountering resistance, typically because it's pushing against a wall. On the AIBO we used the chest IR sensor to detect cliffs. This could be done on the Create using the wheel drop sensor, but at present we have wheel drop events putting the robot into Emergency Stop mode as a safety precaution. Calliope has an IR rangefinder which could potentially be used for both obstacle and cliff detection; this will be explored in future work. We are also experimenting with the Microsoft Kinect sensor as a source of depth information.

Execution

Once the navigation plan has been constructed, the Pilot begins to execute the plan. (There is also an option to have it return the results of its path planning computation without execution, which is useful if the user simply wants to know whether it's possible to get from A to B given the current state of the world.)

The Pilot uses Tekkotsu's "waypoint walk engine" to follow a sequence of waypoints up to the next localization step. If localization reveals the robot has drifted, the next invocation of the waypoint walk engine calculates a correction to bring the robot back onto the desired path.

The user can refresh the world shape space during execution to see the robot's progress on the path and some of the particle filter's particles (Figures 2 and 4).



```
worldspace
AgentData 30471 (x=-144.5, y=489.37, theta=1.92)
wall 30522 (0.0,866.0)->(500.0,0.0)->(0.0,-866.0)
tag0 30523 tagID=0 center=(92.0 704.0 177.8) orient
tag1 30524 tagID=1 center=(245.0 440.0 177.8) orie
tag2 30525 tagID=2 center=(397.0 176.0 177.8) orie
tag3 30526 tagID=3 center=(397.0 -176.0 177.8) ori
tag4 30527 tagID=4 center=(245.0 -440.0 177.8) ori
tag5 30528 tagID=5 center=(92.0 -704.0 177.8) orie
destination 30529 point=(700.0 700.0 0.0)
plannedPath 30580 (0.0,0.0)->(-217.56,975.06)->(-
best 30631 wt=-5.04 (-144.5, 489.37) th=1.92
pt 30632 wt=-5.09 (-145.37, 482.69) th=1.93
pt 30633 wt=-5.16 (-142.35, 490.01) th=1.92
pt 30634 wt=-5.19 (-148.42, 491.5) th=1.91
pt 30635 wt=-5.19 (-143.84, 488.03) th=1.89
pt 30636 wt=-5.26 (-144.92, 479.99) th=1.93
```

Figure 4: SketchGUI panel showing the contents of the world shape space displayed in Figure 2: the robot, the wall, six AprilTags, the destination (a point), the planned path (an open polygon), and some of the particle filter's particles (pt). The particles are tightly clustered inside the triangle in Figure 2 because the robot had recently localized and resampled.

Table 1: Pilot request types.

walk	Walk/turn a specified distance.
waypointWalk	Execute the specified waypoint engine waypoint list.
setVelocity	Move at the specified velocity until told to stop.
localize	Acquire landmarks and estimate the robot's position.
goToShape	Plan a path to the specified destination.
visualSearch	Perform a search until the search exit test returns success.

If all plan steps execute without error, the Pilot posts an event indicating successful completion of the request. Otherwise, the event indicates via an error code what went wrong. One possibility is planning failure, which can occur if there is no collision-free path from the start to the destination, presumably because some object is blocking the way. Another possibility is that a collision occurred.

The user can specify what action the Pilot is to take when a collision is detected. The default is to stop the robot and report the event. Alternatives include ignoring the collision (useful if we're trying to push an object), or relocalizing and then replanning a path to the desired destination.

Pilot Requests

Users communicate with the Pilot by filling in the fields of a PilotRequest instance and submitting it for execution. A

Table 2: PilotRequest fields.

requestType	Type of action the Pilot is to perform.
dx, dy, da	Displacement for walk request.
waypointList	Used by waypointWalk.
forwardSpeed, strafeSpeed, turnSpeed	Speeds for locomotion actions.
walkParameters	Parameter file to load for special gait or walk posture.
targetShape	Destination to walk to.
landmarks	Landmarks to use to localize.
landmarkExtractor	MapBuilderRequest used to acquire landmarks.
landmarkExitTest	Returns true if enough landmarks to localize.
collisionAction	How to respond to collisions.
searchObjectExtractor	MapBuilderRequest used to acquire objects when doing a visual search.
searchExitTest	Returns true if the visual search has succeeded.
executePath	If false, return the planned path but don't execute it.
displayParticles	# of particles to display in world shape space.

"request type" field that describes the effect to be achieved, and additional fields supply parameters for the effect. Table 1 lists the request types, and Table 2 summarizes the major parameters.

A PilotRequest can include two forms of higher level information. One is a MapBuilderRequest that the Pilot should pass to the MapBuilder to tell it what to look for, e.g., during a visual search. The other is a functor that the Pilot can call to test whether some condition holds. For example, the "landmark exit test" tells the Pilot whether it has acquired enough landmarks to perform a localization, and the "search exit test" indicates when a visual search should terminate.

Building a Navigation Application

To encourage user experimentation with the Pilot, we provide a PilotDemo class from which simple navigation applications can be built. PilotDemo implements a command interpreter that allows users to manually construct requests to move the robot, localize, navigate to a point in space, or

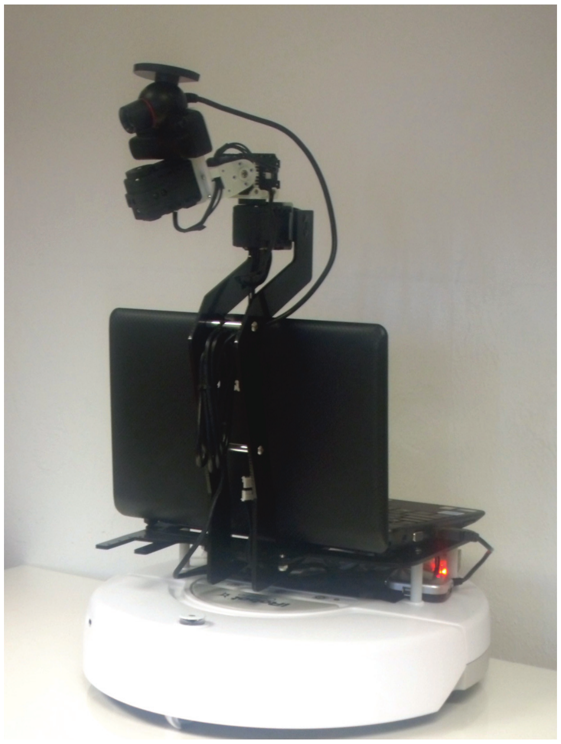


Figure 5: A version of Calliope with the pan/tilt, camera, and IR rangefinder, but no arm.

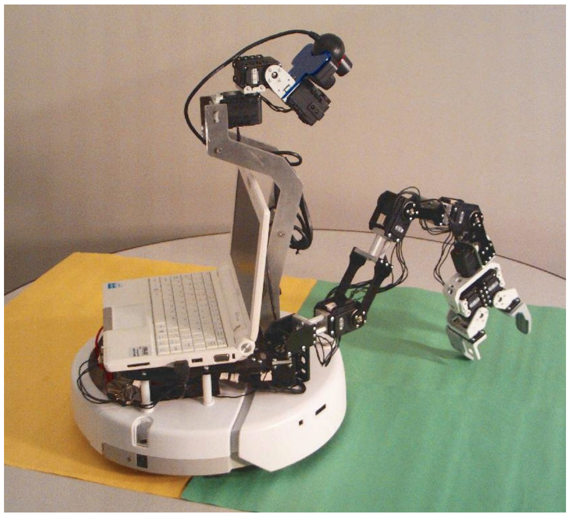


Figure 6: The latest Calliope prototype with a five-degree of freedom arm and two independently controllable fingers. Several arm configurations are being investigated.

manipulate the particle filter. Users can make their own application, e.g., to test out a particular environment and landmark configuration, by creating a state machine that inherits from the PilotDemo class.

Figure 7 shows sample code that produces the path shown in Figure 2. The code is written in Tekkotsu's state machine language, a convenient mixture of C++ and state machine shorthand. It is translated into pure C++ by a preprocessor. In this example, the Figure2Demo behavior inherits from VeeTags, a built-in class that defines the world configuration shown in Figures 1 and 2. (VeeTags in turn inherits from PilotDemo.) We define a new node class GoToGoal as a subclass of PilotNode. (A PilotNode constructs a PilotRequest instance and automatically submits it to the Pilot for execution.) GoToGoal fills in the PilotRequest with a point shape that specifies the destination.

The user's actual state machine contains just three nodes: an instance of GotoGoal and two SpeechNodes to announce the result of the navigation action.. One of the transitions between these nodes, denoted **`=PILOT=>`** in Figure 7, will fire when the Pilot signals completion of the request submitted by GoToGoal.

The Calliope Robot

The Create/ASUS and Calliope robots (Touretzky et al. 2010) have been the primary testing platforms for the navigation facility. Calliope consists of an iRobot Create mobile base on which is mounted an ASUS Eee PC 1001 netbook, a "neck" with a pan/tilt mount holding a camera and IR rangefinder, and an optional arm (Figures 4 and 5). The Create provides odometry, bump sensors, cliff sensors, and user-accessible buttons. It communicates with the netbook via a serial cable. The ASUS netbook has a 1.66 GHz Intel Atom processor, 1 GB of RAM, and a 250 GB hard drive, providing plenty of computing power. It runs Ubuntu Linux. Use of a netbook rather than a simple processor board has several advantages: the netbook provides both WiFi and Ethernet networking, speakers for audio output, and an internal battery to power USB devices. In a pinch it's also possible to program the robot directly from the ASUS keyboard, although it's usually more convenient to use an ssh connection from a full-size laptop or workstation.

The camera is a Sony PlayStation Eye, whose high frame rate (up to 125 frames per second) promises to eliminate motion blur and allow a future version of the Pilot to do landmark tracking on the fly. The pan/tilt uses two Robotis AX-12 servos, and the rangefinder is a Robotis AX-S1 that looks forward, left, and right simultaneously.

Several arm configurations are being investigated; all are constructed from Robotis AX and RX series servos. The arm in Figure 6 is kinematically similar to a Lynxmotion arm with five degrees of freedom: base yaw, shoulder, elbow, and wrist pitch, and wrist rotate, plus two independently controllable fingers. A future version of the Pilot will take the current arm configuration into account when calculating collision boundaries for path planning.

Calliopes with just a pan/tilt draw power from the Create's rechargeable battery to drive the servos. Calliopes with an arm have a separate 5000 mAH NiMH battery in

```
$nodeclass Figure2Demo : VeeTags {

  $nodeclass GoToGoal : PilotNode($, PilotTypes::goToShape) : doStart {
    NEW_SHAPE(destination, PointData,
              new PointData(worldShS, Point(700, 700, 0, allocentric)));
    pilotreq.targetShape = destination;
  }

  $setupmachine{
    go: GoToGoal
    go =PILOT=> SpeechNode($, "destination reached")
    go =PILOT(collisionDetected)=> SpeechNode($,"I hit something")
  }

}
```
Figure 7.

the Create's cargo bay. This auxiliary battery not only powers the arm and pan/tilt, but also serves as an effective counterweight to the arm.

Discussion

The Pilot offers a high level approach to robot navigation by combining path planning, localization, and locomotion. Users can customize the Pilot's behavior by supplying it with MapBuilderRequests and exit test functors.

Other robotics frameworks, such as Player/Stage or ROS, also provide path planning and localization facilities. Unique to Tekkotsu is its emphasis on integrating these components rather than trying to maintain them as independent, orthogonal software modules. In addition, Tekkotsu emphasizes making vision-based behaviors work on relatively low cost robots.

Robotics education at the high school and undergraduate levels is presently hampered by a lack of affordable robots with serious vision, navigation, and manipulation capabilities (Touretzky 2010). Platforms such as Calliope, when supported by the right high-level software, promise to make sophisticated robotics accessible to a much broader population of students, and to researchers who want to use robots in their work without becoming roboticists themselves.

Acknowledgments

Ethan Tira-Thompson implemented the particle filter and Alex Grubb implemented the navigation path planner used by the Pilot. RoPro Design, Inc. is a co-developer of the Calliope robot and supplied several key components.

This work was supported by the National Science Foundation under awards DUE-0717705 and CNS-0742106.

References

Coens, J. 2010. *Taking Tekkotsu Out of the Plane.* Masters thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA. Available online at http://Chiara-Robot.org/Chess.

Kuffner, J. J. Jr., and LaValle, S. M. 2000. RRT-Connect: An efficient approach to single-query path planning. *Proc. IEEE International Conference on Robotics and Automation (ICRA-2000).*

Olson, E. 2010. AprilTag: A robust and flexible multi-purpose fiducial system. Technical report, University of Michigan APRIL Laboratory, May, 2010.

Touretzky, D. S. 2010. Preparing computer science students for the robotics revolution. *Comm. ACM,* **53**(8):27-29.

Touretzky, D. S., Halelamian, N. S., Tira-Thompson, E. J., Wales, J. J., and Usui, K. 2007. Dual-coding representations for robot vision in Tekkotsu. *Autonomous Robots,* **22**(4):425-435.

Touretzky, D. S., and Tira-Thompson, E. J. 2010. The Tekkotsu "Crew": Teaching robot programming at a higher level. First AAAI Symposium on Educational Advances in Artificial Intelligence. Menlo Park, CA: Association for the Advancement of Artificial Intelligence.

Touretzky, D. S., Watson, O., Allen, C. S., and Russell, R. 2010. Calliope: Mobile manipulation from commodity components. Technical report WS-10-09: Papers from the 2010 AAAI Robot Workshop. Menlo Park, CA: Association for the Advancement of Artificial Intelligence.