Final Project Structure

See the Project Proposal for the task description.

File Setup

state.c - the high-level state machine logic and the main method

follow.c - functions defining how to follow a person

voices.c - the `SoundManager` class that handles playing sounds on a separate thread and several `voice` structs for different personae

sensors.c - sensor-processing functions to translate sensor readings into useful descriptions about the robot's surroundings

motion.c - functions for motion control, such as keeping a velocity cap

All files except the first have header files that allow them to reference each other. We have one additional header file **Robot.h** that defines the robot state struct (for more information about the state machine architecture, see project 4).

Person Following (`follow.c`)

The primary action for our system is to follow the person displaying a known marker (a rotated Italian flag pattern). We use a PTZ camera to identify and locate the target's position relative to straight ahead and a laser and sonars to determine the effective distance.

Generally, the robot enters its `FOLLOW` state when the marker comes within the robot's field of view and exits upon losing sight of it. We register a pre-defined object detector from SVM to notify `dataHandler_trackMarker` whenever it finds the object. This function updates the state variables `Robot.sx, Robot.sy` with the centroid and resets `Robot.timeSinceMarker` to 0. The latter variable is incremented during the state update step, so it always records how many cycles have past since seeing the object at location `(sx,sy)`. If it grows too large, the robot will exit the follow state. As long as it is small, though, we assume that the object is in the most recent location. We set the rotational velocity using a proportional control law on the difference between `sx` and the horizontal center of the image. The translational velocity follows the same type of control based on the difference between the free distance in front of the robot and the desired following distance.

Audio Output (`voices.c`)

The robot "speaks" by playing one of its collection of sound files (.wav format). We use SoX, which we briefly describe in Project 3. In order to play the files, we must start a new thread so the robot can keep functioning while playing the sound. The `p_thread` interface allows you to create a new thread by passing in a callback function to execute. We have a whole series of callback functions `characterPhrase` in the file `greet.c` that play appropriate comments in a character's voice. Our first character voice chosen is Dug, the primary dog from *Up*.

In order to allow for run-time switching between voices, we created a struct `voice` that has a field for each type of comment (eg: greet or joke). We then make a struct for each of the characters, so the main program can simply swap out instances of the struct.

In order to avoid needing to reproduce the thread creation code throughout our main file, we created a class `SoundManager` that executes on the main thread and controls the audio thread. It maintains the audio thread as a private field, has two public methods, `playSound(callbackFunction)` and `isFree()`, and provides the voice structs as public static fields. By keeping track of a single audio thread, the SoundManager can cancel current sounds to play a newly requested file. We could even modify the class to allow for high and low priority requests.

This setup has the advantage that we do not lots of threads and provides structure for interruptions. We must create a new function for each new comment, but by passing general function names rather than file paths we can more easily switch out phrases and voices. In addition, our callback functions initially each play a single file, but we can easily add in randomness to select, for example, any of a selection of greetings or jokes.