Now that you have learned the basics of behaviors and Aibo movements, it’s time to put movement together with sensing to create a truly interesting behavior. For this assignment the aibo will be placed at an unknown location in a square pen. At another unknown location in the pen (hidden from view) is the pink aibo bone. Your aibo should explore the pen alternately “sniffing the floor” in search of the bone and “looking around” for the location of the walls. A path leading to the hidden bone has been laid out on the floor of the pen. The path is marked with green and blue post-it notes, laid side-by-side. You aibo should explore the pen until it finds the path and then follow the path to the bone. When your aibo finds the bone note it should raise it’s head, bark and wag its tail. Part of the grading will be a competition. The solution that finds the post-it note the fastest in three trials, wins and extra 1/2 letter grade.

### New Aibo motions for this lab.

The first step in your solution will be to add head movement and walking motions to the lab1 behavior.

**Head Movement:** If you haven’t done so already, refer to the documentation for the `HeadPointerMC` class ([http://www-2.cs.cmu.edu/~tekkotsu/dox/classHeadPointerMC.html](http://www-2.cs.cmu.edu/~tekkotsu/dox/classHeadPointerMC.html)). This motion command gives you control over the three joints in the aibo’s neck: tilt, pan, and roll. Tilt controls the joint at the base of the neck moving the head up and down; pan turns the head left and right; and roll controls the joint at the top of the neck and moves the head in a nodding motion. Use the `setJoints(tilt,pan,roll)` method to set the head position. Don’t forget the mutual exclusion access wrapper and **before setting the head position, make sure that your settings are within limits by checking the outputRanges[] array**. The specific outputRanges array for your Aibo model is included with the `Shared/WorldState.h` file. The indexes for the neck joint values are also symbolically defined in this file.

**Walking Movement:** Walking the aibo is relatively easy using the `WalkMC` class. Refer to [http://www-2.cs.cmu.edu/~tekkotsu/dox/classWalkMC.html](http://www-2.cs.cmu.edu/~tekkotsu/dox/classWalkMC.html) for details. Walking is set up similarly to other motions by adding it to the motion manager, accessing it through the mutual exclusion wrapper, and setting the dog in motion using the `setTargetVelocity(dx,dy,dtheta)` method. The aibo can walk forward, backwards, sideways, and can turn while walking by setting dx to the x velocity, dy to the y velocity and dtheta to the turning angular velocity.

Let’s look at this project in more detail. You will use the motion commands outlined above and two of the sensor systems on the aibo, the IR distance sensors, and the vision system including the CCD camera and color segment detection hardware. There are actually two IR distance sensors, one tuned for near field sensing of close-up objects, and one for far field object sensing. You will use these sensors to keep track of where you are relative to the walls of the pen. For the vision system you also have three object detectors set up, one that will sense blue post-its, one from green post-its, and one for the pink bone/ball object. Before we discuss the specifics of how to access the sensors, read the following overview of event processing on the aibo.
Events and event processing is the primary control structure for Tekkotsu behaviors on the Aibo. Each event in Tekkotsu is identified by three values: the group ID or EGID, the type ID or ETID, and the source ID or SID.

**The Group ID:** The EGID identifies a behavior or set of behaviors that are the source of the event. For example, the `timerEGID` is the group ID and event from an interval timer behavior, the `powerEGID` is the group ID for the events from the power monitor behavior, and the `visObjEGID` is an event sent from an object detector behavior (see below). A complete listing of EGID for events defined in Tekkotsu are listed in [http://www-2.cs.cmu.edu/~tekkotsu/dox/classEventBase.html](http://www-2.cs.cmu.edu/~tekkotsu/dox/classEventBase.html).

**The Type ID:** The ETID is used to communicate specific state information about event sequences. There are three types of events currently defined in Tekkotsu. The `activateETID` is usually the first in an event sequence. The `deactivateETID` tells the listener not to expect any further events in the current sequence and the `statusETID` identifies an intermediate event in the sequence. For example, consider a sequence of events in the visual object identification, `visObjEGID`, group. An activate event is the event sent when the object is initially acquired, status events send new position information as the object moves within the visual field, and a deactivate is sent when the object is not longer in the visual field.

**The Source ID:** The SID is specific to the behavior and identifies sub-events within the group. For example, within the `buttonEGID` group, the SID identifies the specific button, head, back, paw etc., that has been depressed. Specific SID values are assigned by the generating behavior.

As you learned in the last lab, behaviors have three basic components, `DoStart`, `DoStop`, and `ProcessEvent`. Before an event can be received and processed you must do the following in your behavior:

1. add an include statement for the `Events/EventBase.h` file.
2. add a listener that will route the event to your behavior in your `DoStart` method by invoking the `addListener(listenerObject, eventEGID, eventSID)` method.
3. remove the listener in your `DoStop` method by invoking the `removeListener(ListenerObject, eventEGID, eventSID)` method.

Once a listener is added, you will catch the event in the `processEvent(event)` method of your behavior. The event argument is a pointer to an `event` object that is of the `EventBase` class or a class specific to the event group that descends from `EventBase`. Always identify the group and source of the event using the `getTypeID()` and `getSourceID` methods and then cast the event to the particular event subclass.

An aibo behavior never reads the value of a sensor directly. Instead, every 30ms a background process reads the values of all of the sensors (called a frame in OPEN-R) and transfers them to a shared memory region object called `WorldState`. The behavior can access this information in either of two ways. In cases where a behavior needs to read a sensor value synchronously, (i.e. the value is read when the program decides that it needs it) the value can be read directly via a
pointer to the WorldState object. In cases where the program needs to know each time that a sensor value changes or reaches a specific value, asynchronous access. To do this second behavior is generated to monitor the sensor at each update, (signaled by a frame update event) checks it’s value according to a specific rule, and post an event to the first behavior whenever the rule is satisfied.

For this lab we have implemented three behaviors that monitor the color segmentation filter of the vision processor and will post an event to the your behavior if objects larger that a specific size are detected in the visual field. The event parameter includes a pointer to the region that caused the event (or the largest object in the field is multiple regions were detected)

To include these behaviors into your solution you should do the following.

1) Replace the StartupBehavior_SetupVision.cc file in your project file with a replacement that you will be provided for this lab. This program will start the BlockObjectDetection behaviors in the background when Tekkotsu boots on the aibo.

2) Add listeners for event group: visObjEGID, for source IDs: visGreenPostItSID, visBluePostItSID and visPinkBallSID to your DoStart and remove listeners for the same in your DoStop.

3) In your processEvent procedure, after you check the EGID to make sure that it’s a vision object event, recaste the event argument to a vision event argument as follows.

const VisionObjectEvent *ve = dynamic_cast<const VisionObjectEvent*>(&event);

ve is now a pointer to the vision object event from which you can extract the position of the object using the getCenterX(), getCenterY() methods.

For this lab, you will access the near field and far field distance sensors by reading them directly from the WorldState object. To make this object available and access this data do the following:

1) Add and include statement for Shared/WorldState.h into your behaviour.

2) Once included you will have access to the extern (global) pointer call state that is initialized at startup to point to the WorldState object.

3) Within this object, these sensor values are stored in an array at state->sensors[].

4) The index of each sensor depends on the model of your dog. You can retrieve it from constants defines in the config file includes from each dog as follows (xxx = 7 for ERS7, 210 for ERS210)

ERSxxxInfo::IRDistOffset is the index of the far field sensor

ERSxxxInfo::NearIRDistOffset is the index of the near field sensor

It is possible to dynamically check the dog model in your code with the following statement

if(state->robotDesign&WorldState::ERSxxxMask)
{
    /* model xxx specific code here */
}