Implementing Segmented Vision using Tekkotsu
Shawn Turner – Robotics Seminar – Spring 2003

This document aims at examining steps in taking a segemented vision scheme created using the Tekkotsu segmenting tools and applying them to a running Aibo through an arbitrary behavior. It is assumed before reading this that the reader is extremely familiar with Behaviors and Events in Tekkotsu. It would also be a wonderful idea to read over the camera cailbration tutorial. There are good tutorials and papers available at http://www.tekkotsu.org as well as http://www.ils.albany.edu/robotics This document concludes with a brief summary of things to keep in mind.

First, let's take one second to define what color segmentation is, and what's it's apparent usefullness is. CMU's robobits OPEN-R programming course gives the following definition of vision- "Vision can be thought of as the art of throwing out the information you don't want, while keeping the information you care about for you task." This is a very useful definition when thinking about segmentated vision. *Segmentating* can then be thought of as the process by which we *define* those pieces of information that you *do* want by *color*. The process is done in Tekkotsu by taking a series of sample images that represent those good objects and feeding them through a calibration tool that allows us to build a threshhold and color file(s) needed for setup later on. Tekkotsu has some tools for helping to automate this process- as of this paper, they were an open area for improvement. The camera calibration tutorial (http://www-2.cs.cmu.edu/~tekkotsu/CameraSetup.html) steps through the process of generating the necessary threshhold and color files based on a set of sample images.

There are two java classes that we'll use to handle creating the segmentation setup. The first is VisionTrain, which allows us to send it a series of images (taken from the aibo). VisionTrain then creates a color palette based on the colors in those images, the user then has the option of selecting which colors we actually want to keep. Again, the steps for doing this can be found in the color calibration tutorial. The second tool is called VisionSegment, which lets us check our test segmentation, by feeding it the configuration that we just created in VisionTrain and the same set of sample images. The purpose is in seeing the effect that our segmentation had on the set of sample images. Take your time and carefully handle this bit of calibration. It'll directly impact how well your robot performs later. It's worth noting that the tutorial makes mention of a useThresh option included with the java tools that allows us to apply the test segmentation to a running aibo- I've had little luck getting it to work correctly (Tekkotsu 2.0.1); the visionSegment check is fairly indicative of how the segmentation will come through Aibo.

Once we've created an acceptable calibration, the next step is in how to start to apply it to an existing Tekkotsu project. Let's assume that we saved our setup in visionTrain as "test". The result of this would generate two files, test.col and test.tm, as of Tekkotsu 2.0.1, no matter what the path you specified during save, those files will be found in "Tekkotsu_Root"/tools/seg. These files first need to be copied in to the

configuration folder inside our local Tekkotsu project. The appropriate destination will be “Project_Root”/ms/config . After copying those files, the next logical step is to make appropriate changes to the tekkotsu.cfg configuration file. It's location is project/ms/config. Here's a snippet of the file with noted deviations from the default.

```
# tekkotsu.cfg
# ... ...
# gain          low | mid | high
# higher gain will brighten the image, but increases noise
gain=high

# shutter_speed  slow | mid | fast
# slower shutter will brighten image, but increases motion blur
shutter_speed=slow

# ... ...

### Color Segmentation Threshold files: ###
# Threshold (.tm) files define the mapping from full color to indexed color
# You can uncomment more than one of these - they will be loaded into
# separate channels of the segmenter.  The only cost of loading more
# threshold files is memory - the CPU cost of actual segmenting is
# only done when the channel is accessed.

# Included options for color threshold file:
<ERS-2*>
# phb.tm – pink, skin (hand), and blue
#   note: "skin" is just of people who work in our lab - not a general

# sampling... :(
# general.tm - general colors, previously 'default'
# ball.tm - standard Sony pink ball definition
# pb.tm - pink and blue

# orginally on, turned of to avoid color conflict (explained later)
# thresh=/ms/config/phb.tm

#thresh=/ms/config/general.tm

# off again for a truly independent setup
# thresh=/ms/config/ball.tm

# thresh=/ms/config/pb.tm
```

```
# thresh=/ms/config/ttt.tm

# this is our test.tm, created using visionTrain
thresh=/ms/config/test.tm

</ERS-2*>
# ... ...
# the .col file gives names and a "typical" color for display
# the indexes numbers it contains correspond to indexes in the .tm file
# We've disabled the default colors to only account for a very simple one
# color scheme
# colors=/ms/config/default.col

# our segmentation
colors=/ms/config/test.col

# ... ...
# end Tekkotsu.cfg
```

So handling this much of the setup prepares us to start making code changes that will result in the run-time aibo being able to make use of this l e n g t h y amount of preperation. There are two major components in high-level vision programming in the Tekkotsu framework. The simplist way to conceptualize what goes on is that we need a DetectionGenerator to recognize images in the field of vision (think perception) and a Behavior to turn that perception into action. The easiest running demo of this process is the StareAtBallBehavior. It uses a class called BallDetectionGenerator (inherited from EventGeneratorBase:BehaviorBase) to recognize pink objects of a minimum size and throw an event to the StareAtBallBehavior which can get a relative offset from center of the object, and it updates the head joints appropriately. What follows is not an exhaustive look at how to create an appropriate DetectionGenerator, instead, a look at how to implement this detection is presented using StareAtBallBehavior as an example.

First, let's take a look at Shared/ProjectInterface.h and .cc:

```
// ProjectInterface.h
// ... ...
//! Default source IDs for the various generators; These are given default values, but you can reassign them if you like.
/*! @name Vision SIDs */
//! source id for event
extern unsigned int visRawCameraSID;
extern unsigned int visInterleaveSID;
extern unsigned int visColorJPEGSID;
extern unsigned int visGrayscaleJPEGSID;
extern unsigned int visSegmentSID;
```

```
        extern unsigned int visRLESID;
        extern unsigned int visRegionSID;
        extern unsigned int visPinkBallSID;
        extern unsigned int visBlueBallSID;
        extern unsigned int visHandSID;
        //@}
// ... ...  ...
```

What's of concern here are the externs relating to the “balls”. visPinkBallSID, visBlueBallSID, etc. are going to wind up specifying exactly what ball, aibo, marker, etc. that we're trying to detect when we implement our behavior. So if one wishes to detect a goal for instance, you would want to define a constant like:

```
        extern unsigned int visGoalSID;
```

What you actually call it doesn't matter of course, it's just nice to be consistent.
You give this new variable value in ProjectInterface.cc:

```
        // .... .... ...
        unsigned int ProjectInterface::visGoalSID=17;
        // .... .... ....
```

The numbers are completely arbitrary, the current version of Tekkotsu uses up to like 3 or 4, so pick a number far higher to help minimize potential conflicts. Also, be aware that ProjectInterface is a shared file across all project in the installation. That means variables you declare will be able to be seen by other programmer's, so you'll want to comment appropriately so everyone's on the same page.

The other file that helps to implement a segmentation is StartupBehavior_SetupVision.cc. This file declares, among other things, specific pointers to DetectionGenerators for each channel of the segmentation that we currently care about. i.e. :
StartupBehavior_SetupVision.cc :

```
        // .... .... this is already in the file, but take note of the scope
        using namespace ProjectInterface;

        BallDetectionGenerator * pball=NULL;
        BallDetectionGenerator * bball=NULL;
        // .... ... declare two different BallDetection pointers, one for a pink ball, one
for blue
```

So here we've got two different pointers for detection, they both use the same generator, but we'll assign them different channels from low level vision:

```
// ... ...
//this line here associates an int to the color channel defined as “red” (from the .col file)
unsigned int pinkIdx=segcol->getColorIndex("red");
```

//if the channel associated successfully, allocate a new generator to our pink ball pointer

```
        if(pinkIdx!=-1U) {
                pball = new
BallDetectionGenerator(EventBase::visRegionEGID,visRegionSI
D,visPinkBallSID,pinkIdx,threshChan,noiseFiltering,confiden
ceThreshold);

        pball->setName("PinkBallDetectionGenerator");
        }
// ... ...
```

Note the bold faced visPinkBallSID and pinkIdx, the BallSID really just acts as a unique marker for the color as far as I can tell, the pinkIdx role should hopefully be apparent- this creates the association inside the BallDetection class between the regions we'll be searching and the segmented channel that we care about. The blue ball declaration is identical, save the use of the blue channel and BallSID.
The last piece of work to do in SetupVision is that if we want any behaviors to be able to capture our detection event, we had better make sure it's on. The inheritence from BehaviorBase makes this possible.

```
        // StartupBehavior_SetupVision.cc:
        // ... ..
// just plug the name of your detection pointer here and run the
// “start” method from BehaviorBase ... ...

addItem((new BehaviorSwitchControlBase(pball))->start());
        // ... ...
```

This is similar for all detection that you'd like to do.
This is a good spot to mention that if you change the default segmentation, you'll want to make sure that the SetupVision file isn't trying to access and activate channels that no longer exist. For instance, if you only segment a goal color, running the above ->start() call will naturally result in a program abort (Castlevania noise). The simple solution is just to comment out any detection initialization code that you haven't setup explicitly during your segmentation process. The reason why this would happen stems from the code snippets in SetupVision above. Take pball for instance, if we define a goal as green, and only segment that color, the color index (.col) file won't have any entry for red or pink. SetupVision begins by initializing it's pointers to NULL. The next step is to attempt to associate an integer with a value from the color file, after that the `if (pinkIdx != -1U)` controls whether or not we allocate memory to this pointer. A failed attempt to associate would return the -1U value, and nothing more would be done with our pointer until we tried to activate it- and what happens when we dereference a NULL pointer?

So if all has gone well, he have a running detection generator succesfully implemented in a Tekkotsu project, all that's left now if to create a behavior capable of

catching this event and acting on it. Again, let's look at StareAtBallBehavior for a simple example:

StareAtBallBehavior.cc (.h is standard boilerplate):

```
// ... ... ...

void StareAtBallBehavior::DoStart() {
	BehaviorBase::DoStart();
	headpointer_id = motman->addMotion(SharedObject<HeadPointerMC>());
erouter->addListener(this,EventBase::visObjEGID,ProjectInterface::visPinkBallSID);
}
	// ... ...
	// ... processEvent(... ) ...
	static float horiz=0,vert=0;
	if(event.getGeneratorID()==EventBase::visObjEGID &&
event.getTypeID()==EventBase::statusETID) {

	horiz=static_cast<const VisionObjectEvent*>(&event)->getCenterX();
	vert=static_cast<const VisionObjectEvent*>(&event)->getCenterY();
	}

	//cout << horiz << ' ' << vert << endl;
	// .... the remainder of the code just updates the head joints
```

The three things of primary interest are the bold-faced erouter->addListener call above, the third argument of this should always correspond to that unique value we created for our colored object. The rest should always remain the same. The second is the if check in processEvent above, this if check should pretty much remain constant, but does seem the most convienent way to make sure that we're checking the correct event, the statusETID portion of this may be the least intuitive- take a look at BallDetectionGenerator.cc for a more in-depth look at why this is necessary. Lastly, look and see how the location of the object is retrieved from the event, this will give back an x and y value in a scale from -1 to 1 across both axis in the aibo's field of vision. This is very useful if we want to chase or follow in any way.

This concludes the bare necessities in getting a segmentation onto Aibo. Take a deep breath, and take a look at the summary below:

1. Segmented Vision works on the principle of including only certain colors.
2. Not all colors are appropriate for segmentation, experiment in YUV raw camera mode to get a good idea of what the Aibo can “see”.
3. The visionTrain and visionSegment tools are provided in Tekkotsu as a

way to define the needed colors by creating a color file (.col) and a threshold map (.tm) that Tekkotsu low level vision can use to create good channel mappings.

4. The tekkotsu.cfg configuration file in Shared/ is where the specific .col and .tm are named for the project, as well as where shutter and gain settings are set.
5. Every object of a specific color that needs detection should be given a unique unsigned int ID inside the ProjectInterface namespace.
6. The interface between low level and high level vision can be observed in the StartupBehavior_SetupVision file- this is where RLE actually takes place, where channel associations are created, and where your detection generators are given memory and activated.
7. Make certain that your behavior registers for the event according to the visID that you've created, and make sure that the processEvent method verifies that statusEGID has been posted as well as visObjEGID.

Comments and Questions welcomed at st2750@albany.edu.