Assignment 4: An Original Sota Behavior

In this assignment you will design and implement an original Sota behavior using all that you have learned so far in this course. You have a lot of freedom here which can be liberating but also challenging if you are not used to open-ended assignments.

The main goals of this assignment are:

- Gain comfort working with behavior trees to structure complex interactive behaviors.
- Practice integrating a range of inputs into an interactive behavior.
- Practice designing and implementing a branching, interactive robot dialog.
- Gain experience with designing and managing a larger programming project.

Build Environment

Make sure to update your Sota build environment to be sure you have the most recent details in the Sota primer, including the updated LD_LIBRARY_PATH. Also, you need to make sure to use the `upload_resources` target as it contains all the libraries you have not yet used.

Integrating everything will be very challenging, and, you may hit CPU limits. Note that the Sota has two cores and so leveraging threading cleverly may help resolve some of this. Otherwise you need to be clever about when to turn on and off certain features.

Also note that all the inputs are very unstable and unreliable. You need to design your behavior to work well and to handle all the many cases that the robot may find.

At the end of the file I have included additional build environment sections to help you get started with the various libraries.

Task 1: High-level Design of A new Sota Behavior! (20%)

In this task you will engage with HRI and Socal HRI principles and plan out a high level social robot behavior. You have a lot of freedom here with your task; if you're stuck, consider a companion to chat with, or a lonely robot kiosk at a train station. Don't skip this, put real work into it before you start to build. Don't start coding yet, beyond possibly testing the included library functionalities to understand possibilities.

Your behavior must include the following:

- Branching static or dynamic dialog using keyword or keyphrase recognition.
- Evolving interaction over time.
- Robot speech (using dynamic TTS or fully pre-rendered audio)[1].
- Robot gestures and body language[2], including use of color.
- Some inclusion of the face detection library into the behavior.

---

[1] You can use other sound, like the robot playing theme music or background music.

[2] You can use your IK to do this. However, depending on your interaction goals it may be simpler to use pre-programmed behaviors. It should be simple to extend your AS3.1 to record keyframes in a Java class file and store a file using Serialization (e.g., use the up button to set the next keyframe). Then load these with hand-coded parameters such as speed, eye color, etc.

- Anything else you think is interesting (e.g., other libraries, your FK, etc.)

You can finish and polish this before working on the next sections, and leave it behind. You will most certainly make changes later, but you don't need to come back and re-frame this unless you completely changed your project direction.

Deliverable

- **Writeup /5: a one page, high-level introduction to your behavior. Focus on the motivation and use of the HRI and Social HRI principles, without necessarily giving the implementation details. What are your goals? What is your behavior trying to achieve, from both a task and a user experience lens?**

Task 2: Design your new Sota Behavior! (15%)

In this task you will simultaneously design the specifics of your novel Sota behavior, and an initial behavior tree to plan it out. Plan it first before you build it, to the best of your ability. Don't start coding yet.

Focus on the specifics of what the robot will do, when. Develop your dialog, what the inputs and outputs will be. Plan out your body language, speech, colors, etc., throughout your behavior. Use whatever format makes sense for you here (text, flow diagram, a mixture, etc.).

Develop an initial behavior tree, on paper (or using a digital tool), that describes how this will be implemented.

Keep in mind that not all the logic has to be embedded within the tree, e.g., a node may use a state machine or other such logic. The tree is the high-level structure. That said, you should aim to use trees to abstract behaviors. Note that the sample code includes an example of how to embed a tree into another tree, which can help simplify implementation and debugging.

Deliverables

- **Writeup /5: a half page to one page description of the implementation specifics – what the robot will do, when. (30%)**
- **Writeup /5: your initial, pre-programming behavior tree. (70%)**

Task 3: Implement Your Behavior (50%)

Now start coding. Start writing your code to implement your planned out behavior tree. Of course, you need to be flexible – things won't work (or won't work as expected), and you'll hit roadblocks. Update your behavior tree as you go and keep great notes.

Deliverables

- **Demonstration /5**
-

Task 4: Show off Your Final Behavior (15%)

Create a video that shows your behavior. Make it fun, and plan out how you will explain and sell it. If you can, editing will make it feel polished.

Revisit your earlier tasks, and update your behavior tree, and reflect on your first page plan.

Deliverables

- **Video /5: either upload the video or provide a link. The prof would love a copy to keep for records. (50%)**
- **Writeup: /5: updated behavior tree (25%)**
- **Writeup: /5 half page reflection on what worked and what went wrong, and why. (25%)**

Build Environment: Behavior Tree Library

The distribution pack includes a library `gdx.ai.btree` which provides a simple tick-based behavior tree implementation.

Examine the included `GDXaiSample.java` program. Sketch out the behavior tree specified in the file. Run the program in VSCode (not on the Sota) and follow along using your sketch, making sure you understand everything that is happening.

Also, check out the `GDXaiSubtree.java` file, which shows you how to wrap a tree in a task to be a node in a new tree. This is really important for modularity and lets you build and test parts of the tree independently.

Although in this example everything is in one file, I suggest considering to create separate files for each class. If you find this is too many files, at least for the robot blackboard. You can clean up the structure by making packages to keep things separate.

The library has a very nice documentation, including the section "A Simple Example."

Behavior Trees · libgdx/gdx-ai Wiki · GitHub (https://github.com/libgdx/gdx-ai/wiki/Behavior-Trees)

JavaDoc: gdx-ai 1.8.2 javadoc (com.badlogicgames.gdx) (https://javadoc.io/doc/com.badlogicgames.gdx/gdx-ai/latest/index.html)

Build Environment: MaryTTS

MaryTTS is a lightweight, simple TTS (text to speech) engine. Included is an English voice pack. This is old technology, and so don't expect cutting edge synthesis – but, it's simple enough to run on the Sota 😊. The gotcha with MaryTTS is that its slow. So if you don't need dynamic generation, you should pre-gen them and save as wave files, or, just use a different offline generator. You could even cleverly cache them for generated results (e.g., in wav files).

Examples of usage are provided in `marysample.java`.

MaryTTS has support for fine-grained control of prosody, but again, this is old tech so probably there are better tools available.

You can install your own local Mary setup using git, and use gradle to run a few of the utilities. https://github.com/marytts/marytts Only if you're really into it. You probably have enough variation given the speech effects as outlined below. This also has a tool to let you download other voices. If you are interested and can't get it working, I downloaded them, so feel free to ask.

MarySpeech | MyRobotLab – section 3. Voice Effects, gives an overview of the effects.

Overview (MaryTTS 5.2 API) – the JavaDocs API (probably not needed)

MaryTTS – Documentation – main website, probably not needed.

There are additional voices you can install as well for more variety. You need to install a downloader to get them, but I have them cached if anyone is interested. I haven't tried them.

Build Environment: PocketSphinx Keyword Recognition

PocketSphinx is a portable, lightweight speech to text system, focused on being deployable on embedded systems. As with MaryTTS, this is old tech, but it works on this old platform.

A few examples are provided in the `pocketsphinxSample` package. To get these working you need to make sure that the language model provided is loaded on the Sota, uploaded with the "upload resources" command.

You need to configure the keyphrases.txt file to look for specific phrases. As outlined at Building a language model – CMUSphinx Open Source Speech Recognition, for robust applications you should try to autotune the values using pre-recorded audio but that is not needed here (probably). You may want to move it elsewhere so you can edit it on the Sota without it being overwritten if you upload resources. In this case make sure to keep a backup.

General transcription does not work well, as outlined in the sample file. You can improve it as noted. You can also dig into the pocketsphinx documentation online and see how to create grammars – basically trees of keywords – to more robustly read sentences.

Build Environment: Sota Face Recognition and Processing

First you need to be sure that your camera focus is working properly. Check the Sota primer (newest version) on how to do this. Note that having a bright light behind the person confuses the image processing.

Check the provided `FaceDetection.java` file for how to access Sota's face detection features. This is pretty unreliable and slow, so you will need to consider that in your designs. Consider using the smiling detection, face size (e.g., closeness?), motion detection, etc.