Outline for Revised Sensor / Data Logging system in NTRTsim

Drew Sabelhaus 2017-01-03

We could do the redesign in multiple ways. These are split into sections below…

1) Full-featured

Sensors themselves have a reference to an individual tgModel of a certain type.

For example, one tgRodPositionSensor has a reference to one tgRod.

So, that means that multiple sensors can sense the same tgModel.

An individual sensor knows what classes it can sense.

There should be some check in the constructor about what type of reference is passed in.

(This also implies that a pointer to a specific tgModel, for example some specific tgRod that is to be sensed, is passed in to the constructor of the tgRodPositionSensor for example.)

An individual sensor knows how to get specific information from its tgModel.

For example, tgRodPositionSensor will call a specific method within tgRod (something like getPosition or whatever the equivalent would be.)

An individual sensor also knows how to make a header for its data for a CSV file, as well as how to output the data itself.

For example, tgRodPositionSensor.getSensorHeading might return something like “(tags,etc).X,(etc).Y,(etc).Z”.

Then, tgRodPositionSensor.getData might return something like “(dataX),(dataY),(dataZ)”.

In other words, an individual sensor can output MULTIPLE COLUMNS.

There can be sensors that have other sensors attached.

For example, a tgRodSensor might have a tgRodPositionSensor and a tgRodOrientationSensor.

(^NOT SURE ABOUT THIS QUITE YET…)

(The advantage of this is for ease of coding. We could have, for example, a tgEverythingSensor that includes all other sensors. This would make it easy for new people to use the sensors: just attach a tgEverythingSensor and no need to worry about individual ones.)

(Maybe this means that tgSensors should optionally have a list of other sensors, similar to how tgModel works?)

An individual sensor is created by some method like buildSensors, similar to buildInto for models.

This will need to be called by ..

(^ Here, we want to have the logger of some sort do the creating of the sensors. The tgModel(s) themselves should not create the sensors. The sensor creation should happen externally from creating the model, probably: think similar to how controllers are created.)

There are Info classes for each sensor.

For example, the tgRodPositionSensor would also have a tgRodPositionSensorInfo class.

In general, these info classes will take either zero or one argument: if no arguments, then sensors are created for each possible model.

So, it’s this Info class that has a hard-coded check on the type of tgModel passed in: it would look through all the children of the base tgModel, and if any fit the criteria, it creates a specific type of sensor while passing in a pointer to that (for example) specific tgRod.

If there is one argument to the Info class, that would be a list of tags, and only models that have those tags will get sensors attached to them.

For example, if we create a tgRodPositionSensorInfo with an argument like [“t2”,”t3”], then only rods with those two tags will have a tgRodPositionSensor created for them.

On the logging side of things, the actual tgDataLogger2 (or whatever we call the new class) should take a list of pointers to sensors.

This list has to be kept ordered – and maybe tgDataLogger2 alphabetizes it or something if we want to get fancy.

Then, during setup, this logger class (of which there is only one instance) creates the log file, and calls on each sensor to create a header, which it then appends to the log file.

Similarly, during step, the logger just calls the getData method or something like that inside each sensor, and just outputs the returned strings to a log file.

(This is similar to the current structure, just with the individual sensors providing the information instead of hard-coding it into the logger class.)

This allows for some other types of logging classes to be made – maybe don’t call them loggers.

For example, we could have a tgDataLoggerSTDOUT class that outputs the data to the command line instead of a log file.

YAML files should be able to have sensors specified.

Parsing the YAML file should create Info objects for the sensors, just like for the tgModels.

In the YAML file itself, sensors should be specified similar to how a Info class is specified: that is, with the name of the sensor, and optionally a list of tags of the models that should be sensed.

What’s not clear to Drew at the moment is how the sensors are created exactly, and how this would look (for example) in an App file. We could do a few different things…

2) Simplified, without Info classes.

Sensors themselves have a reference to an individual tgModel of a certain type.

For example, one tgRodSensor has a reference to one tgRod.

So, that means that multiple sensors can sense the same tgModel.

However, for ease of use, let’s only do one sensor type per class of tgModel at the moment: whatever type of sensor would output ALL the data from that tgModel. Ex., only one tgRodSensor instead of a tgRodPositionSensor/tgRodOrientationSensor.

An individual sensor knows what classes it can sense.

There should be some check in the constructor about what type of reference is passed in.

(This also implies that a pointer to a specific tgModel, for example some specific tgRod that is to be sensed, is passed in to the constructor of the tgRodSensor for example.)

An individual sensor knows how to get specific information from its tgModel.

For example, tgRodSensor will call a specific method within tgRod (something like getPosition or whatever the equivalent would be.)

An individual sensor also knows how to make a header for its data for a CSV file, as well as how to output the data itself.

For example, tgRodSensor.getSensorHeading might return something like “(tags,etc).X,(etc).Y,(etc).Z,(etc).RotX,...”.

Then, tgRodSensor.getData might return something like “(dataX),(dataY),(dataZ),(dataRotX),...”.

In other words, an individual sensor can output MULTIPLE COLUMNS.

There is one tgDataLogger2 class.

This class does the creation and management of sensors, as well as the logging of data to a text file.

In its constructor, it takes in:  
 - a reference to a tgModel (CHECK THIS! Is this possible to do? Will it work before the tgStructureInfo calls buildInto?)

- a list of sensors to create and attach to the various tgModel objects, parents and children.

In its setup method, the tgDataLogger2 iterates through all the sensors, as well as all the tgModels, creates a sensor if the sensor name matches the model (NOTE that this comparison needs to be done by the sensor itself, maybe a static method since the sensor isn’t instantiated yet?), and stores all the references to sensors in an ordered list.

Then, the setup method calls on all those sensors to create a heading for a log file.

The onStep method for tgDataLogger2 looks similar to the onStep for tgDataLogger, but asks the sensors for data instead of calling on individual tgModels itself.

Here is exactly the flow of commands:

1. A specific tgModel (e.g., tgBoxAnchorDebugDemo) is created with ‘new’ inside an App.
2. A tgDataLogger2 is created, passing in a reference to the tgModel (maybe a vector of pointers to tgModels?), as well as a list of all the sensors to create/attach to the tgModel. (This is maybe a vector of strings.)
3. simulation.addModel is called, which calls the ‘setup’ method inside the tgModel.
   1. Note that no tgRods, etc. are present inside the tgModel until simulation.addModel is called, since the tgModel’s setup method is what calls tgStructureInfo.buildInto.
4. tgDataLogger2.addSensors is called, which looks through its pointers to tgModels and creates the sensors. This also creates the text file and headers maybe?
   1. An alternate infrastructure here would be to have a list of info classes for the sensors, and have those info classes do the checking (“is this my tgModel?”). This will definitely be cleaner than passing in a vector of strings.

3) Simplified, with Info classes.

This is needed because C++ does not allow for re-definition of static member functions.

This would be required in the check-and-create-sensor-if-it-fits stage of setup.

My initial idea was to have a static method for each sensor, one that could be called for a yes-no answer of if that sensor should be created for a specific sense-able object, but this won’t work because the method (a) has to be present in the parent class, so that we can maintain a list of tgSensors in the tgDataLogger2 during setup time, and (b) a static member function in the parent wouldn’t be able to determine if an object that has inherited from it would be able to answer this question.

SO! Let’s do three new base classes:

1. tgSenseable. This class exists so that we can have pointers to a specific subset of objects. It will serve as a mixin for tgModel, for now. This makes it so that we don’t have tgModel\* pointers everywhere in tgDataLogger2, which would require changing all that code in the case that something other than a tgModel should be sensed.
2. tgSensor. This is an abstract base class for a sensor. It will have two methods, and a constructor which takes a pointer to a tgSenseable and assigns it to a protected pointer.
3. tgSensorInfo. This abstract base class serves as the go-between for creating sensors. It will have nothing in its constructor. It will have two methods: (a) canSense, which takes a pointer to a tgSenseable, and returns a boolean if that tgSenseable is compatible with this sensor. (b) createSensor, which creates one of “these” sensors for a specific tgSenseable.

In the App, then, the workflow is the following. First, the tgModel (e.g. the tgSenseable) is created, then the sensors are created.

1. A specific tgModel (e.g., tgBoxAnchorDebugDemo) is created with ‘new’ inside an App.
2. simulation.addModel is called, which calls the ‘setup’ method inside the tgModel.
   1. Note that no tgRods, etc. are present inside the tgModel until simulation.addModel is called, since the tgModel’s setup method is what calls tgStructureInfo.buildInto.
3. A tgDataLogger2 is created, passing in a reference to the tgModel (maybe a vector of pointers to tgModels?).
4. tgSensorInfo classes are created with ‘new’ for all the sensors that should be created. E.g., a tgRodSensorInfo, a tgSpringCableActuatorSensorInfo, etc.
5. These sensor info classes are added to the tgDataLogger2.
6. The tgDataLogger2 is added to the simulation, via something like simulation.addDataLogger, which then calls something like the createSensors method in tgDataLogger2 and sets everything up.
   1. Used to be: “The setup method in tgDataLogger2 is called, which creates the sensors via the tgSensorInfos’ createSensor methods.” This is insufficient, because even though everything is set up properly, there isn’t any way to get data while the simulation is running.
7. Then, when the simulation is running, there should be an additional ‘step’ or something like that to get the tgDataLogger2 to record data. This has to be separate from tgSubject/tgObserver, unfortunately.

This requires some additional code inside tgSimulation that calls some onStep method within data logger(s), like with tgModels.

Motivation behind all of this: it would be better, ideally, to use the current tgSubject/tgObserver paradigm for the sensors. However, there are massive polymorphism issues with having tgModel be a tgSubject. Specifically, for controllers, a controller needs to be templated as an observer for a specific tgModel, not a general one. So, the base tgModel class is NOT a subject, but all its children ARE subjects templated against themselves. This creates problems for observers that should act on ALL tgModels. It also will create massive issues if we have, for example, a tgSenseable be a tgSubject, then have tgModel inherit from tgSenseable: in this case, a child of tgModel (such as VerticalSpineModel) will be a tgSubject in two ways (first, from tgSubject → tgSenseable → tgModel → VerticalSpineModel, and second directly as tgSubject → VerticalSpineModel). We could get around this with careful namespace resolution, but that makes our code horrible.

The alternative is to have an entirely separate structure for sensing than we do for observer/subject patterning, e.g., something different between sensors and controllers. This is probably OK for now, since these two functionalities (sensing, controlling) are the only things that NTRTsim users really do in this way. Also, this is probably OK because if the polymorphism with tgSubject/tgModel is ever truly fixed, then all the underlying code for tgSensor(s) would not take too much change to switch over to tgSubject: it will all work the same, just changing over ‘addDataLogger’ to attach to a model. (< but maybe not… what if we have multiple models??? That’s the case with some of Drew’s horizontal spine work!)

Aside: do I need to pull in the controller code for TensegrityModel, the YAML model? Only the old stuff seems to be present in master + SensorRedesign (that “TensegrityModelController” class that now doesn’t make sense – I think I had individual controllers for e.g. the HorizontalSpine) .

Weds afternoon: next step is to start on the infrastructure for having the simulation hold a tgDataLogger2 and call it at each timestep. Write out the classes and make sure they’re called. After that, then link it all together with tgSensorInfo classes.

TgDataManager needs:

* tgDataManager
* ~tgDataManager
* setup()
* teardown()
* step(double dt)
* addSensorInfo(tgSensorInfo\* pSensorInfo)

tgSensorInfo needs:

* bool isThisMySenseable
* tgSensor\* createSensor