Boku Programming Object Model

By Michael C. Miller

This document describes the current programming model used in the Boku prototype as of 9/13/2007. It is meant as an introduction into how the objects are organized and used; and many small details are left to the reader to find in the code.

## Programming Organization

Brain

Tasks

Modifiers

Selector

Actuator

Filters

Sensor

Reflexs

Sensors

Actuators

Memory

Brain: This represents the programming for one GameActor. It does all the work to manage the tasks (activities), programming elements that are per actor (sensors, actuators), and reflexes. It contains all the tasks that a given actor is programmed to do.

Task: This represents a set of reflexes (like a set of instructions) for the brain. There can only be one active at any given time within the brain. All reflexes within it should be considered to be run in parallel. They are ordered for the purpose of providing a priority when the actuators of the reflex are shared.

Reflex: This represents one behavior or instruction. It contains a reference to the GameActor’s sensor and/or actuator that it uses. It also contains the filters to apply to sensor input, the selector to convert the input into output, and modifiers to change the output.

Sensors: This is a list of the active sensors used by the active task. When the task changes, the sensors are released and created as needed for the active task.

Actuators: This is a list of the active actuators used by the active task. When the task changes, the actuators are released and created as needed for the active task.

Memory: This will manage the time based memory system. It will act to retain information about actively sensed objects for a period time relative to the time they were originally sensed. The longer they are actively sensed, the longer they will remain in the memory (to a point) to be able to be passively sensed. Currently there is only one memory used primarily for sight.

## Programming pieces Class Hierarchy

Modifier

Sensor

Selector

Filter

Actuator

ActorElement

ProgrammingElement

Atom

Atom: This represents the base for all CardSpace classes. Its use other than the base to ProgrammingElement is for the management and serializing of a group atom; which is used for purposes of visual grouping elements for selectors.

ProgrammingElemenent: This is the base class for all programming elements.

ActorElement: It originally was meant to provide any needed member variables or methods when a ProgrammingElement class was required to be instanced per Actor. It is rare that these methods are not used and having this information called out outside of a ProgrammingElement class doesn’t seem useful longer term.

Sensor: This represents the base class for all input. There are currently three types of sensors that define how they are used. They are FastPoll, Event, and polled (not defined). FastPoll will update the sensor on every brain/task update. Event relies on external events to cause the sensor to update and thus is excluded from the list of sensors updated in brain/task. And if neither is defined, then the sensor will be updated by the brain/task at its normal reaction time (currently every 0.33 seconds).

Filter: This represents the mechanism to narrow and define the input that comes from the sensors before said input is handed to the selector. There are cases where it used as a parameter (PlayerFilter, TeamFilter). Further there are cases where it is used as an originator of the input (a dynamic parameter) (like GamePadButtonFilter, GamePadStickFilter) for a custom Sensor.

Selector: This represents the behavior of the reflex. It will take sensor information (after being filtered) and create output that the modifiers are applied too before being registered with the Actuators arbitrator.

Modifier: This represents the mechanism to modify the output that comes from the selector before it is registered with the Actuators arbitrator. In some cases it is used purely as parameter to the Selector or Actuator (ColorModifier).

## Programming Functional Description

### Brain Update

The brain update is called on every game update cycle. Inside this update the brain will do work to correctly update sensors and actuators and run reflexes as needed based upon its reaction time. The steps are listed below in order.

1. Every brain will have a random start offset before it will do any work at all. This offset is always less than a second and keeps the GameActors from being synchronized in their first reactions.
2. All reflexes of the active task with Sensors that are FastPoll are updated. This is primarily used for User Controlled sensors like GamePads that are based upon polling to react quickly to state change. See Sensor Update below for details.
3. The reaction time is checked against the number reflexes that are left to run and an estimate of the number to run this update is made and those reflexes are updated. See Reflex Update below for details.
4. All actuators of the current task are updated. This is done by having the associated arbitrator run and provide the priority action that the actuator will apply to the GameActor. See Actuator Update below for details.

### Sensor Update and ComposeTargetSet

The work a sensor does is split into two pieces; Update and ComposeTargetSet.

Sensor

Update

Sensed Things

All GameThings

Since a sensor is shared by all reflexes in the brain/task, some of the work it does is common to all reflexes. This common work is done in the Update. For an example, take sight, during the update it will use the sensors properties (range and view angle) and calculate the sub set of all of the GameThings that are within its possible range.

filters

Reflex

Sensor

ComposeTargetSet

Sensed Things

Target Things

Then there is work that is specific to each reflex and done at Relfex update time. This is done in the ComposeTargetSet. Often this is applying the reflexes Filters to the sub-set that calculated in the Update to provide the final set of objects to the Selector.

### Reflex Update

When a reflex is updated, it will ask the Sensor to ComposeTargetSet with the reflexes filters.

Arbitrator

Reflex

modifiers

Action Things

ComposeActiontSet

Action Things

Action Things

Target Things

Then it will ask the Selector to ComposeActionSet with this target set and the reflex modifiers. The exact work the Selector does depends on the selector. But often it entails picking one of the items from the Target set and applying the modifiers.

Then the target set is attached to the arbitrator of the Actuator for possible later handling.

### Actuator Update

Action Things

Actuator

Arbitrator

Reflex 1

Apply to Actor

Action Things

Reflex 3

ComposeEffector

Action Things

Reflex 4

The first thing the actuator does is to ask its associated arbitrator to “Pick” an action set based upon priority and then apply the value to the GameActors state that the actuator represents. Currently there are two types of arbitrators; a Movement Arbitrator and an Action Arbitrator.

The Action arbitrator will simply walk the sets of Action Things and take the highest priority one. The priority is set by the Reflex when it adds it to the arbitrator. The value it uses is the index from the set tasks set of reflexes, thus a reflex at the top will have a higher priority.

The Movement arbitrator is a little more complex. Its set of Action Things includes information that defines the Action Things as Attractors or Repulsors. The Selector that added the Action things will define the type. The movement arbitrator will find the closest Attractor heading vector and merge all Repulsors vectors.

## Programming Elements

### CardSpace.Xml

All Programming Elements exposed in the game as Programming Tiles are define in the CardSpace.Xml file. Many of exposed Tiles use the same Programming Element class, just with different properties. This CardSpace.Xml file allows for easy property management like changing the Tiles text or icon without changing code. Further, it includes properties that allow a tile to be archived (hidden) and properties to hint at how the tiles should be grouped. Lastly the file includes a section to define replacements for “upgraded” tiles and a section to define the UI groups Icon and labels.

### Hidden Defaults

A programming element can be marked as a hidden default. This allows a tile to be automatically added if the user didn’t specifically add it or if it wasn’t even exposed for the user to select so that functionally can be maintained. It hides implementation elements that do not clarify anything for the user.

An example of this is the Selector Closest. It will select the GameThing that is closest to the actor as the input for the Actuator. Although alternates could be provided (like furthest), the complexity of the selector is hidden from user but is still present in the programming definition.

### Tile Compatibility

When creating a Reflex, it is required to maintain that all included Programming Elements are compatible and usable. To allow the UI to correctly filter the selectable list there are several methods exposed off the classes and properties defined that should be used to filter the available Programming Elements. The UI should first call ReflexCompatible and if it returns true then call ActorCompatible.

#### bool ProgrammingElement.ReflexCompatible( Reflex reflex, ProgrammingElement replacement )

This method is called on an element, with parameters of the reflex it is being added to and the element that it is replacing (null if being added).

The implementation is different for each type and is outlined in the table below. Exact implementation depends on the needs of the ProgrammingElement this table is just a generalization.

|  |  |  |
| --- | --- | --- |
| Element Type | Test 1 | Test 2 |
| Sensor | no tests |  |
| Filter | check Reflex.Sensor.Category against Filter.SensorCategory | confirm count/duplication |
| Actuator | call ElementCompatible for each filter |  |
| Selector | check Reflex.Acutator.Category against Selector.ActuatorCategory | call ElementCompatible for each filter |
| Modifier | check Reflex.Actuator.Category against Modifier.ActuatorCategory | confirm count/duplication |

The confirm count/duplication means that the ProgrammingElement will check the Reflex if there is another element of the same type and allow or disallow as needed. Today, you can use the modifier Quickly and Slowly three times.

#### bool ProgrammingElement.ElementCompatible( Reflex reflex, ProgrammingElement other )

This method is called on a secondary element, with parameters of the reflex it is being used with and the primary element that is being tested against. It is currently always used within other Compatible calls.

#### bool ProgrammingElement.ActorCompatible( GameActor actor )

This method is called on the ProgrammingElement to find out if it is compatible with the GameActor it is going to be added to.

The default implementation will if check the Actors Classification Name is present within the ProgrammingElements MountKey property. For sensors, it also will check if the GameActor actually has that Sensor as a device.

### The List

The following are all the currently developed programming elements with a brief description of their unique nature.

#### Sensors

AudioSensor – Currently this sensor is archived and thus not used. It was originally planned to be the hearing for any gameactor; but without sound in the game and further another sensor that was a little abstract it was found not to be as useful. Further, to support sound as a programming concept, other programming elements would be needed to filter them and actually make them.

BumpSensor – This is exposed as the “Touch” sensor. Currently very simple collision test is done against the Bump Devices that the actor exposes. This work should get redone and use a true collision system to trigger this sensor rather than the current polling.

GamePadSensor – This sensor acts more like a manager than the true source of the sensor event. It will request the GamePadStick and GamePadButton filters to provide the actual input. This sensor demonstrates a break in the normal use of the model but demonstrates how other elements can be used to solve problems.

GameScoredSensor – This sensor demonstrates one of the only Event based sensors. It will only get updated when the actual score changes.

GivenObjectSensor – This sensor demonstrates another of the few Event based sensors. It will only get updated when an actual item is given to the game actor.

HeldSensor – nothing unique.

HoldingObjectSensor - nothing unique.

MemorySensor – this sensor is archived. The currently plan is to move its functionality into other sensors as inherent functionality than expose the complexity.

NullSensor – this sensor is not exposed and was used to make the locations that were empty.

ScanSensor – This sensor demonstrates the MountKey feature, and is only usable on a Saucer or Blimp.

SightSensor – nothing unique

StallSensor – this sensor is currently archived for simple usability testing. It normally checks if the GameActor is not moving but is expecting to be moving. There are thoughts that this should be inherent in Selectors that apply to movement; but there are some interesting cases that you specifically do not include such behavior.

TimerSensor – this sensor is one of the few FastPoll sensors. This means that it will be updated on every Update rather than on every brain reaction cycle (about every 0.33 seconds).

#### Filters

ClassificationFilter - this very general class allows for defining a filter on any of the properties stored in the GameActors classification class.

CountFilter - this class allows for defining a filter that checks the count of items in a Target Thing set. Including the lack of anything in the set. It is also used as a hidden default to represent a filter to check the presence of ANY in the set (included on all reflexes if not already present in another form).

DistanceFilter - this class allows for defining a filter that checks the distance of the item in a Target Thing set.

NullFilter - this filter is not exposed and was used to make the locations that were empty

AnythingFilter – not used anymore, deprecated by the count filter.

NothingFilter - not used anymore, deprecated by the count filter

GamePadButtonFilter - A hybrid filter that provides the source of gamepad button input

GamePadStickFilter - a hybrid filter that provides the source of the gamepad stick input

PlayerFilter - this class is used to define a parameter filter that will set which player gamepad should be used.

ScoreFilter – this class is used to define a filter for the amount of score that must be met or exceeded.

TeamFilter - this class is used to define a parameter filter that will set which team the score is sensed from

TimerFilter - a hybrid filter that provides the source of a timer input.

#### Actuators

GameResetActuator – this class exposes the ability to have the InGame state reset.

GameScoreActuator – this class exposes the ability to have the InGame score changed

GameVictoryActuator - this class exposes the ability to have the InGame victory condition set

MovementActuator – this class exposes the ability to update a GameActors movement (heading and speed)

NullActuator - this actuator is not exposed and was used to make the locations that were empty

VerbActuator- this general purpose class can be used to expose actions that GameActors support. There are three types (valency) of Verbs supported; Intransitive, Transitive, and Ditransitive (see definitions of a verb). The triggering of the different valency will cause the GameActor and an alternate to be called with one of several Verb handling calls. These are DoDirectObjectVerb and DoSubjectVerb.

#### Selectors

AwayFromAllSelector – this selector will calculate a vector that is away from all of the Action Things and add it as a replusor to the actuators arbitrator.

CameraRelativeSelector - this selector will calculate a vector that provides movement based upon the input vector (from a stick) and the camera view vector.

CircleSelector – this selector will do one of two things. First, if the distance to the first Action Thing is greater than the circling distance, then it will move toward the first Action Thing. Otherwise if the distance is equal or less than the circling distance, it will calculate a set of waypoints that circle the target and start navigating those points.

ClosestSelector – this selector is known as “nearest” and is hidden unless there is another valid selector. It was known as “Use” for a bit. It will just select the closest Action Thing from the set and add that to the actuators arbitrator.

EscapeSelector - this selector will spin the GameActor around and then move the GameActor straight after spinning around. The angle and distance are properties and a randomizer is used.

FollowWaypointsSelector – this selector is known as “Path”, was “Follow”, and “Waypoints”. It will interact with a Waypoint set and provide target positions for the actuator based upon the state of following said Waypoint set.

NullSelector - this selector is not exposed and was used to make the locations that were empty

SpinSelector – this selector will spin the GameActor clockwise

TowardClosestSelector – this selector will find the closest Action Thing and calculate a vector toward it and hand this to the actuator’s arbitrator.

WanderSelector – this selector will pick a random location to wander to based upon properties. Once it gets near the location or gets stalled it will pick another.

#### Modifiers

ColorModifier – this modifier acts like a parameter and provides a color for the selector or actuator

DirectionModifier – this modifier both acts like a parameter providing a fixed direction for the selector or actuator; and modifiers the heading to be the direction.

ExpressModifier – this modifier acts like a parameter and provides the expression to the actuator.

NullModifier - this modifier is not exposed and was used to make the locations that were empty

ObjectModifier – this modifier acts like a parameter and provides the object type to the actuator

PayloadVerbModifier – this modifier acts like a parameter and provides a Verb to be used as a payload to the actuator

ScoreModifier – this modifier acts like a parameter and provides a score amount to the actuator

SpeedModifier – this modifier will modify the selector output vectors length

TaskModifier – this modifier acts like a parameter and provides the task id to the actuator

TeamModifier – this modifier acts like a parameter and provides the team id to the actuator