06.04.2014

FRAIL specification

User's manual

Table of Contents

Creating AI actor

Configuring AI

To create an actor you need to edit **ActorAI.json** file, which is located in "**build\data\presets**" directory. Actor is defined by JSON "Preset" object.

AI actor is represented by specified parameters:

- *PresetName* – actor identification name
- *m_maxSpeed* – floating point value which sets actor's maximum movement speed
- *m_canJump* – boolean value which determines if actor can jump
- *m_jumpSpeed* – floating point value which sets actor's jump height
- *m_sightDist* – floating point value which sets actor's sight range
- *m_horSightAngleRad* – floating point value which sets actor's sight range degree in radians
- *m_shootingRange* – floating point value which sets actor's shooting range
- *m_visibleInSightQueries* – boolean value which determines if actor will be seen by another AI
- *m_shootingDamage* – floating point value which sets shooting damage value (doesn't apply to fireball damage)
- *m_prefabName* – prefab name from prefab database
- *m_conflictSide* – conflict side name (currently available: "BlueTeam", "RedTeam", "Unknown")
- *m_characterCtrlName* – sandbox AI controller name
- *m_rangedLaunchPosHelperName* – actor's element which shoots ranged missiles
- *m_health* – actor's health on game start
- *m_maxHealth* – actor's maximum health
- *m_animMultiplierMelee* – floating point value which sets melee animation speed (deprecated since animation can be configured through xml file)
- *m_damageMultiplier* – floating point value which sets damage multiplication value
- *m_meleeConeSize* – floating point value which sets cone size (degrees), located in front of actor; objects in cone receive damage when melee attack is executed
- *m_meleeRange* – floating point value which sets actor's melee attack maximum range
- *m_smellRange* – floating point value which sets actor's smell sense range
- *m_btTreePath* – relative path to behavior tree xml file (if BT controller was chosen)
- *m_htnMethodsPath* – relative path to HTN methods xml file (if HTN controller was chosen)
- *m_htnOperatorsPath* – relative path to HTN operators xml file (if HTN controller was chosen)
- *m_htnGoalsPath* – relative path to HTN goals xml file (if HTN controller was chosen)

Setting AI on map

To set AI actor on level you need to edit desired level JSON file. Sample levels are stored in "**build\data\levels**" directory. Actor's spawner object is called "AISpawner".

AISpawner parameters:

- *m_spawnOrigin* – position where AI will be placed
- *m_spawnRadius* – range radius in which AI will be placed
- *m_aiNum* – number of desired AI actors to be spawned
- *m_presetName* – actor's preset name mentioned earlier in this section

Creating BT-based controller

To create an actor which uses behavior tree as a controller you need to design AI first.
(You can try sample BT-based controller which is located in "**build\data\AI\BT**" directory.)

Behavior Tree Structure

The whole tree consists of nodes. There are several types of node implemented into sandbox. Those are:

- ***Selector*** - runs first child node which satisfies its preconditions

Example:
```
<node type="Selector"> ... </node>
```

- ***Priority Selector*** - runs first, most useful child node which satisfies its preconditions; child nodes are sorted by "usefulness" parameter

Example:
```
<node type="PrioritySelector"> ... </node>
```

- ***Sequence*** - runs every child node one after another and returns success status when all children ended successfully

Example:
```
<node type="Sequence"> ... </node>
```

- ***Parallel*** - runs every child in parallel and returns status specific to its policy

Example:
```
<node type="Parallel" policy="requireOne"> ... </node>
```

Parallel node result depends on its policy. Policy "**requireOne**" returns success status if one node from children nodes returned success status. However, policy "**requireAll**" returns success only when all nodes from children nodes returned success status.

- ***Condition*** - returns success status if all preconditions are satisfied

Example:
```
<node type="Condition">
```

```
        <pre>Equal IsEnemyVisible False</pre>
    </node>
```

Condition node contains at least one "pre" statement, which stands for precondition. Nodes of condition type returns success status only when all of its preconditions are satisfied.

- ***Action*** - node which manages in game actions

Example:

```
    <node type="Action" name="Patrol" usefulness="1.0" duration="3000"
    interruptible="1">
        <pre>Equal IsEnemyAttack False</pre>
        <pre>Equal IsEnemyVisible False</pre>
    </node>
```

Action node is an action which is executed in game. It returns success status only when all of its preconditions are satisfied. It requires 4 additional attributes: **name**, **usefulness**, **duration**, **interruptible**. Attribute **name** reflects action implemented into in-game controller. **Usefulness** is required only when parent node is of type "Priority Selector". **Duration** stands for time spent on executing specific action. **Interruptible** defines if action can be interrupted by any other action.

Interruptible

If **interruptible** action was intended to interrupt other action from different parent node, then all preconditions must be specified. **Action interruptions rely on preconditions.**

Creating HTN-based controller

To create an AI actor based on HTN controller files: goals.xml, methods.xml and operators.xml need to be edited or created. Those files are located in "**build\data\AI\HTN"** directory.

Methods

File **methods.xml** contains at least one method. Methods decompose compound tasks into other compound tasks or operators. It is design dependent.

Example:

```
    <methods>
      <method name="mthPatrol" usefulness="1.0">
          <goals>
              <goal>glPatrol</goal>
          </goals>
          <preconditions>
              <pre>Equal IsEnemyVisible False</pre>
          </preconditions>
          <subtasks>
              <sub>opPatrol</sub>
          </subtasks>
      </method>
```

```
        </methods>
```

Method's attributes:

- *name* – methods identification name
- *usefulness* – floating point value which determines methods evaluation sequence
- *runAll* – optional parameter which determines if all methods operators should be applied when preconditions are satisfied

Method's members:

- *goals* – determines which goals can be decomposed by this method
- *preconditions* – conditions that need to be satisfied before compound task can be decomposed
- *subtasks* – result of decomposition; subtask can be compound task or operator

Operators

Operators are defined in **operators.xml** file. Operators are atomic tasks which can be executed by HTN controller.

Example:

```
<tasks>
  <task name="opPatrol" duration="3000" interruptible="1">
      <outcome>
          <out>IsEnemyVisible True</out>
      </outcome>
      <interruptions>
          <int>opRevealAttacker</int>
          <int>opExploreSpot</int>
      </interruptions>
  </task>
</tasks>
```

Operator's attributes:

- *name* – operator's identification name
- *duration* – time spend on executing operator
- *interruptible* – determines if operator can be interrupted
- *isAnim* – determines if operator is an animation

Operator's members:

- *outcome* – predictions about effects of task execution (important to create reasonable plan!)
- *interruptions* – when preconditions can't determine if task should be interrupted, these operators can interrupt current task

Goals

Goals are defined in **goals.xml** file. They are also known as compound tasks.

Example:

```
<tasks>
    <task name="glKillEnemy" main="1">
        <postconditions>
            <post>Equal IsEnemyVisible True</post>
            <post>Equal IsEnemyDead True</post>
        </postconditions>
    </task>
</tasks>
```

Goal's attributes:

- *name* – goal's identification name
- *main* – optional attribute which determines if goal is the main compound task (starting task)

Goal's members:

- *postconditions* – conditions which need to be satisfied for plan to be complete

Common specification

Preconditions

Preconditions are statements build from 3 separate aliases. First is always binary function, which may be: Equal, NotEqual, More, MoreEqual, Less, LessEqual. Next two aliases are values which will be compared, using binary function, every time node will be updated.

Aliases

Aliases are objects which hold values. There are two types of values that alias can hold: boolean value and floating point value. There are a few types of aliases: consistent, actor specific, world state, user.

Consistent aliases:

- *True* – which corresponds to true boolean value
- *False* – which corresponds to false boolean value
- *Zero* – floating point value which is used for operators' outcome, representing number 0

Actor specific aliases (controller specific):

- *rngMelee* – actor melee range
- *rngFbMax* – actor fireball range

World state aliases (controller specific):

- *IsEnemyVisible* – determines if enemy is in range of actor's sight
- *IsEnemyDead* – determines if enemy is already dead
- *EnemyDistance* – floating point value which holds distance from actor to enemy
- *IsEnemyAttack* – determines if actor was attacked by enemy

- *IsEnemyRunningAway* – determines if enemy ran further than 1.0 distance unit since last 0.5 second
- *ActorHealth* – floating point value which holds actor's current health points
- *HealthAMLimit* – floating point which holds anger mode limit health points (currently ActorHealth/2)
- *IsActorAM* – determines if actor is or already was in anger mode
- *IsEnemySeen* – determines if last enemy position is set and valid
- *EnemyDgrDiff* – floating point value which holds angle between AI direction and enemy in degrees

User aliases are declared in tree.xml file. Every user alias can be a redirection to already existing alias or a new value. User aliases are evaluated only once when application starts!

Example:

```
<aliases>
    <alias>boolExample true</alias>
    <alias>sampleRange rngMelee</alias>
    <alias>FBRange 10.0</alias>
</aliases>
```