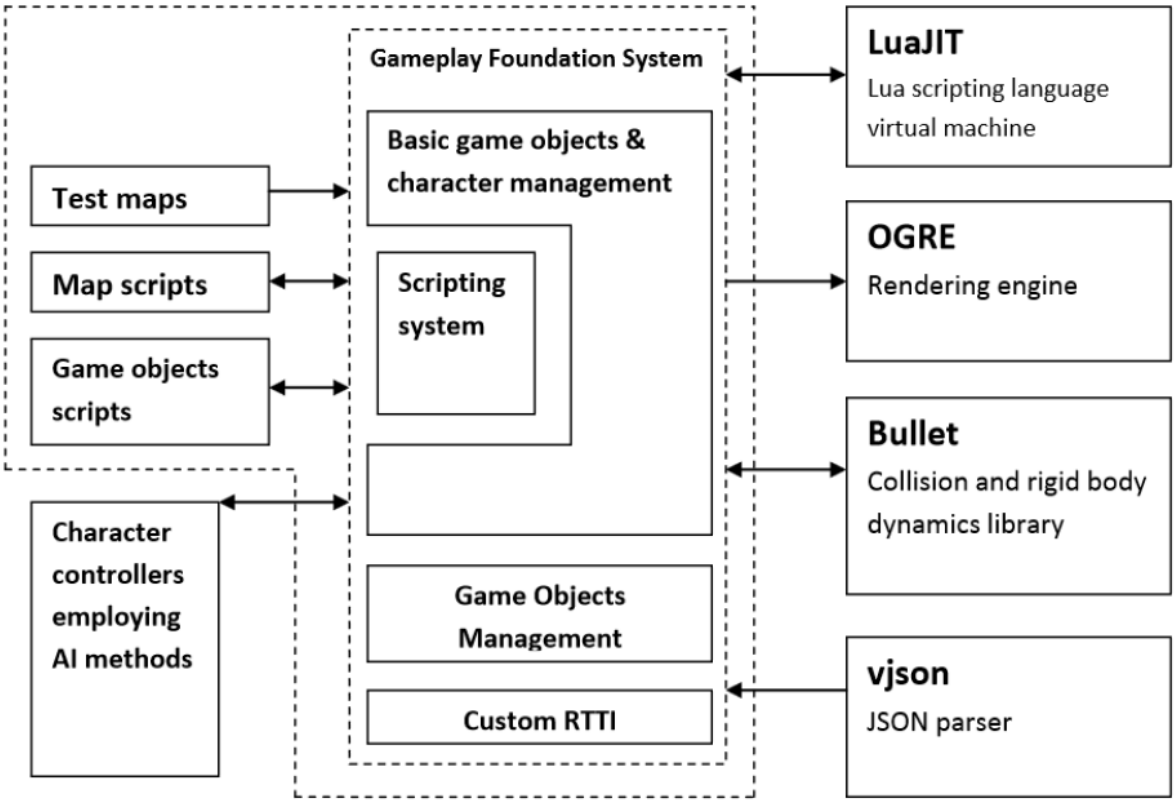
14.04.2014

FRAIL specification

Developer's manual

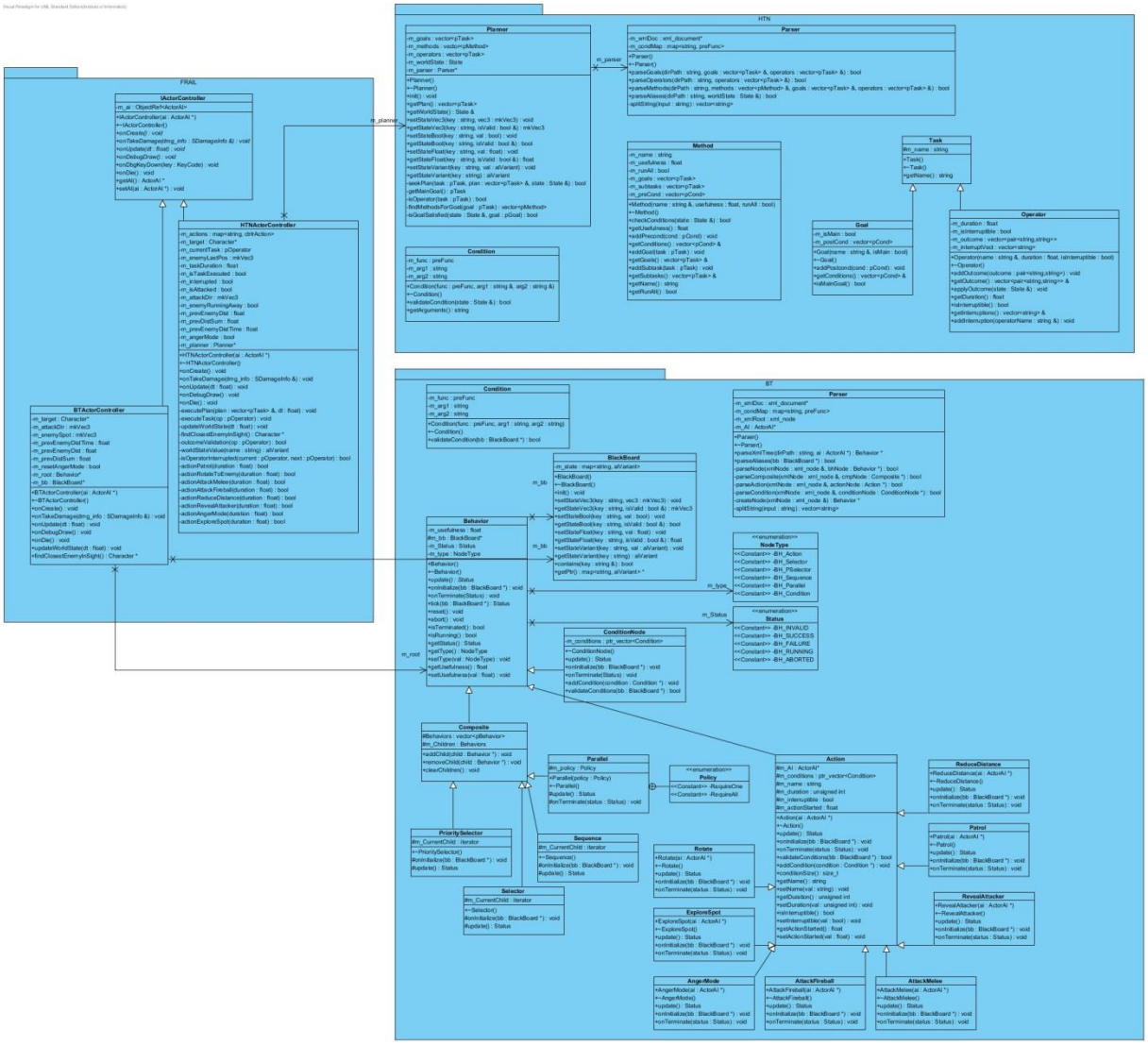
Table of Contents

FRAIL's architecture



FRAIL architecture is based on well-known schemas used in game development, thus making it easier to use for people familiar with existing solutions. Some functionality is built using open-source **middleware**. FRAIL architecture schema is presented above. Scene rendering and low-level visualization is done using OGRE, an Object-Oriented Graphics Rendering Engine. OGRE is "a scene-oriented, flexible 3D engine written in C++ designed to make it easier and more intuitive for developers to produce applications utilizing hardware-accelerated 3D graphics". Besides rendering, OGRE handles most of operating system-specific boilerplate, such as window creation, making it trivial to port FRAIL to Linux and Mac OS X – please, note, however that there are precompiled dependencies only for Win32/MSVC, and compiling on Linux/OSX would require minor code adjustments.

AI architecture for BT and HTN



Actor controllers are classes included into FRAIL framework which communicates with specific AI mechanism: Behavior Tree, Hierarchical Task Network, FSM, etc. The same diagram is located in main FRAIL's directory.

Behavior Tree

To create BT-based AI you need to create controller file in FRAIL first.
All controllers' name must end with "**ActorController**" phrase (ex. BTActorController.cpp)!

1. In your fresh actor controller class, implement all abstract methods from inherited class **IActorController**.
2. Create members which will point to BT root node and blackboard.
3. In actor controller file in constructor body create new blackboard and initialize it by calling **init()** method.

4. In **onCreate()** method create local parser member and assign root node to result of parser's **parseXmlTree(std::string, ActorAI* ai)** method.
5. Call also **parseAliases(BT::BlackBoard *bb)** method to create user defined aliases.
6. In actor controller's method called **onUpdate(float dt)** you need to update blackboard values to let actions run properly.

   Example:
   ```
   m_bb->setStateFloat("ActorHealth", getAI()->getHealth());
   ```

7. Now you have to run **tick()** method in your root node object to start evaluating the tree. To do this, simply put **m_root->tick(m_bb)** in your **onUpdate(float dt)** method. You need to remember to pass blackboard object to the root node, so BT actions can transfer informations and evaluate preconditions.
8. The most important part is to create actions which can be called by their parent nodes. Go to **Actions.h/Actions.cpp** and create new class which extends **Action** class and implements its virtual methods.
9. The last step is to assign xml, action node's name to specific object. Navigate to **Parser.cpp** file and put new "**else if**" statement into **createNode(pugi::xmlNode& xmlNode)**.

   Example:
   ```
   else if(std::strcmp(xmlNode.attribute("name").value(),"MY_ACTION_NAME") == 0){
            result = new BT::MY_ACTION_CLASS(m_AI);
   }
   ```

Now you just need to put your actions into **tree.xml**, assign your controller name to AI "Preset" object in **ActorAI.json** file and run the game!

Hierarchical Task Network

1. Create new actor controller file containing "**ActorController.cpp**" ending.
2. **Planner** object lets you get current plan based on previously designed methods, goals and operators. In your new actor controller you need to implement your own plan executor or copy existing one from **HTNActorConroller.cpp** file.
3. In your actor controller you need to implement abstract, inherited methods and create HTN::Planner object.
4. First initialize Planner by calling its **init()** method. (**init()** method call parser and initialize world state).
5. In your controller's **onUpdate(float dt)** method, you need to update world state and execute new plan by calling **m_planner->getPlan()**.
6. To create new action you need to assign your xml operator's name to unary function by placing it into m_actions std::map.
   Example:
   ```
   m_actions["opPatrol"] = &HTNActorController::actionPatrol;
   ```

7. Now declare that function in your actor controller's file and define it as:

```
bool HTNActorController::actionName(float duration){}
```

Sample HTN-based actor controller and BT-based actor controller implementation can be found in **SampleHTNActorController.cpp** and **SampleBTActorController.cpp** files.

Finite-state Machine

Sample FSM controller can be viewed in **SampleFSMActorController.cpp** file.

1. New header file and source file have to be created within project sources, with class name ended with "*ActorController.*" phrase.
2. New FSM controller needs to extend **StateMachineActorController** class.
3. It is recommended to include states in controller file to avoid mess.
4. You should create namespace for your AI states to avoid conflicts with other states' names.
5. You have to create your own base state which extends **sm::State** class from **StateMachineActorController** file.
6. Defining new method of your base class is required for derived states to communicate with actor controller. It should be called e.g. YourController* getController() const, where YourController is your controller class name.
7. Every frame **updateStateTransition()** method from parent class should be called.
8. To change state use **sheduleTransitionInNextFrame(new State())** method.
9. Every state should have, at least **onEnter(State*)** and **onUpdate(float dt)** virtual methods extended.

AI actor interface

Every AI actor is an instance of ActorAI class. It has one controller which decides if any action has to be performed.

Basic AI actor's methods.

Base class	Name	Parameters	Return value	Description
ActorAI	IsDead	-	bool	true if actor is dead, false otherwise
ActorAI	getHealth	-	float	returns value referring to actor's current health
ActorAI	getMaxHealth	-	float	value reffering to actor's maximum health
ActorAI	isSeenByEnemy	Character* *enemy*	bool	true if is seen by *enemy*, false otherwise
ActorAI	isInShootingRange	Character* *enemy*	bool	true if is in *enemy's* shooting range, false otherwise
Character	jump	-	-	performs action jump without animation

Character	setDirection[1]	mkVec3 *dir*	-	sets new direction *dir*
Character	setSpeed	float *max_speed_part*	-	sets new speed relative to maximum speed
Character	setMaxSpeed[2]	float *val*	-	sets new maximum speed
Character	getMaxSpeed	-	float	returns character's maximum speed
Character	getRealSpeed	-	float	returns character's current speed
Character	getSimPos	-	mkVec3	returns character's current position
Character	getSimDir	-	mkVec3	returns character's current movement direction
Character	raycast	mkVec3 *dir*, float *height*, float *ray_len*	RayCastResult[3]	casts ray into selected direction with specified length and height
Character	isPositionVisible	mkVec3 *pos*	bool	true if position is visible, false otherwise
Character	isEnemy	Character* *other*	bool	true if *other* character is an enemy, false otherwise
Character	isAlly	Character* *other*	bool	true if *other* character is an ally, false otherwise
Character	getShootingRange	-	float	returns value referring to character's shooting range
Character	getMeleeRange	-	float	returns value referring to character's melee range
Character	lookAt[1]	mkVec3 *target_pos*	-	sets new direction, opposite *target_pos*
Character	startSmoothChangeDir	mkVec3 *destinationDir*, unsigned int *stepCount*, float *taskDuration*	-	smoothly rotates to *destinationDir* with *stepCount* steps in *taskDuration* time
Character	stopSmoothChangeDir	-	-	stops smooth rotation invoked by startSmoothChangeDir method
Character	runAnimation[4]	mkString *animName*, float *duration*, float *animDuration*	-	runs *animName* animation in *duration* time with *animDuration* – animation time specified
Character	hitMelee	-	-	performs melee attack without animation within melee range
Character	hitFireball	mkVec3 targetPos	-	casts fireball towards specified position
Character	hasBuff	-	bool	true if character holds buff, false otherwise
Character	isInPowerLake	-	bool	true if character is in power lake, false otherwise
Character	setMovementDir	mkVec3 val	-	sets movement direction
Character	getMovementDir	-	mkVec3	gets movement direction

Character	resetMovementDir	-	-	sets movement direction to mkVec3::ZERO

[1] – for more realistic rotation use startSmoothChangeDir(…) instead
[2] – some methods are prohibited in tournament gamemode
[3] – RayCastResult is a structure which holds values which determine if hit was occurred, etc.
[4] – runs animation only if model has specified in *animName* animation, animations can be browsed in game with ctrl+z (previous animation), ctrl+x (current animation), ctrl+c (next animation)

mkVec3

mkVec3 is a type definition referring to Ogre::Vector3. In FRAIL(in Ogre as well) mkVec3 corresponds to position and direction. Direction is described as normalized vector, when position is a vector holding three float values (x,y - vertical,z).

Movement direction

Character's movement direction is a normalized mkVec3 vector which defines where character will move in next frame. When movement direction attribute holds mkVec3::ZERO value (x=0, y=0, z=0), character will move in direction it's facing. To reset movement direction you can use Character::resetMovementDir() or use Character::setMovementDir(mkVec3::ZERO) methods.

Tournament arena actor interface

Tournament mode is a specific FRAIL's gamemode. It is strictly connected with tournament_arena map and has some restrictions. Methods listed below should only be used within tournament_arena map.

AI actor's methods on tournament map.

Base class	Name	Parameters	Return value	Description
ActorAI	isMedkitAvailable	-	bool	**true** when medkit is available on map, **false** otherwise
ActorAI	isBuffAvailable	-	bool	**true** when buff is avaialbe on map, **false** otherwise
ActorAI	getMedkitPosition	-	mkVec3	medkit position if available, mkVec3::Zero otherwise
ActorAI	getBuffPosition	-	mkVec3	buff position if available, mkVec3::Zero otherwise
ActorAI	getPowerLakePosition	-	mkVec3	power lake position
ActorAI	getBarrels	-	std::vector<ModelObject*>	returns collection of ModelObjects referred to barrels on tournament map

Building FRAIL from source

Microsoft Visual C++ 2010 (MSVC 10) is required to build FRAIL source files.
All third-party libraries are included in "**src/deps**" directory.

1. Open **mkd.sln** file, which is located in **"src/code/mkd"** directory
2. Choose preferred configuration (**Hybrid/Release**)
3. Press **F7** button to build project

Currently debug profile isn't available due to lack of third-party libraries and *.dll files for debug profile.
Hybrid profile is the release mode without optimization. It is recommended whenever quick build is needed.