RFC - Interactions between Entities

Metacosm Development Team

Revision History

Revision M1	July, 15th 2001	
DocBook conversion, types of actions, uses cases examples, beginning of interaction framework, actior		
Revision 0.13	2000/03/21	Revised by: Entropizer
Re-organization, improved the implementation section		
Revision 0.12	2000/03/16	Revised by: Entropizer
Full rewrite (Entropizer) - Based on Metacosm-DTD		
Revision 0.11	1999/11/27	Revised by: Entropizer
Minor changes (corrections by Ruffy, thanks)		
Revision 0.10	1999/11/20	Revised by: Horus
Document rewrite		
Revision 0.9.1	1999/08/18	Revised by: Horus
First SGML version		
Revision First discussions	Summer 1999	

This document describes how interactions between entities could be handled in Metacosm. This is an important part of the design since it will define how the Metacosm engine could work.

1. License

2. Metacosm

For more information on Metacosm, check our site: Metacosm (http://metacosm.org).

3. Foreword

This document relies heavily on the notion of Entity. To be able to understand all of it, the reader should be familiar with Entity-RFC. Several references are made to that document in the present one. It can therefore be useful to keep a copy of it at hand. Enjoy!

4. Introduction

Interactions are (with the concept of Entity) the core of the Metacosm game engine. Designing an efficient and flexible interaction model is really important because so much will run on interactions in the game engine. Next, we define what is an Interaction in Metacosm.

Interaction

Interactions are the modality through which Entities modify (or at least try to...) their surrounding (including themselves). Everything that has an observable result in the game domain, is the result of an Interaction between one (an Entity interacting with itself) or several Entities.

The approach that we decided to take in this document is to account for the thought process that we followed to design the interaction system. More specifically, we will try to define interesting use cases that raise issues that we will try to address by designing the interaction system. We believe that following this process will help us design an efficient interaction engine and will serve as documentation and history of the choices that were made. This document will be finished when all identified use cases can either be implemented using Metacosm's interaction engine or rationale will be given as to why a given use case won't be supported.

5. Action concept

An Entity can influence its environment in very various way depending on its type, its abilities, etc. To create an efficient interaction model, we introduce the concept of Action.

Action

Actions are the basic piece in the Interaction model. They define at the smallest level how Entities interact with their surroundings. They encapsulate what the Entity does and how it does it. Entity influence their environment via Actions.

6. Use cases

Nothing can drive development better than well chosen use cases. In this section, we describe use cases that we think could help us direct our effort towards creating a good interaction system.

6.1. A flying gnome

6.1.1. Description

Gob is a (not very intelligent) gnome that decided that he wanted to fly. He asked his friend Mardus, a magician, to cast a fly spell on him. The spell is successful but has a limited duration. Gob decides to try his new ability by trying to fly from trees to trees, discarding his friend's advice to be cautious. Delighted by the possibility to fly, he forgets about a time limit and found himself in middle air between two trees as the spell wears off. He cannot fly anymore and falls to the ground.

6.1.2. Technical challenges

The focus of this use case is to come up with an elegant solution to temporary Influence Actions and how to deal with interrupted Actions.

- Off-line to enable use to focus on the logical aspects before having to deal with synchronization and time management issues.
- Action interruption and dynamic continuity.
- Dynamic adding and removal of Action via Influences.

7. Requirements

The interaction system is at the core of Metacosm's game engine. It is therefore very important to implement it in an efficient and extensible way. We will rely on design patterns to insure that the Interaction system will offer an acceptable level of performance. Next, we examine the requirements for the interaction system.

- *Small memory footprint*: the system must be able to handle a large amount of different Actions without impacting too much on memory requirements for the game engine.
- *Dynamically extensible*: it should be possible to add new Actions at runtime. It should also be possible to reload any number of Actions to replace them with a newer implementation.
- *Shareable among a large number of Entities*: many Entities have the possibility to perform the same Actions.
- *Customizable on an Entity basis*: an Entity can be influenced by several different Influences and therefore be able to perform all (or part) of the Actions define by its Influences but can also provide a custom implementation for a given Action that will override the default implementation.
- *Fast enough to insure a good interactivity with players*: look up and retrieval of an Action should be performed in a reasonable lapse of time (100 ms?) so that interactivity is preserved.

8. Action concept summary

We elaborate on this section a little bit on the concept of Action, following what we have learned in the previous sections.

- Each Entity can only perform (at a given time) a given number of different Actions. It is possible for an Entity to learn new ways to influence its environment and thus to expand its number of possible Actions.
- Each Entity maintains a list of Actions that it is able to perform. This list is in fact the composition of several different lists. In fact, each Influence can potentially

define its own particular list of Actions to which Entities gain access when becoming influenced by this particular Influence.

- All of these lists are dynamic and can be modified at runtime to follow the evolution of the Influences. However, the level that is most likely to be altered is Individual level since an Entity's state influence on what it can accomplish. Moreover, for Entities able to learn, their available Actions will follow their progresses. An Entity able to see can become blind... An Entity can learn how to swim. For some Entities, it is likely that they won't have (at least most of the time) an individual level (this is true for Entities that cannot individually evolve).
- Actions can be complex objects that act like scripts. They are able to handle a given action by themselves. In this way, Entities don't have to implement numerous interfaces to be able to perform actions. Moreover, the needed code for a given Action is only written once and can be used by numerous entities. This way entities are kept pretty simple and their capabilities can be expanded (or restricted) dynamically.
- In fact, Actions could be as simple or as complex as needed. One could start by implementing Actions that do nothing and that just return something like "OK, you did it!" as a result... Moreover, it is possible to think of changing an Action implementation while the game is running. If more realism is needed, it is possible to change the Action implementation and all Entities will be able to use the new, more realistic version in no time and while the game is running! Since the result of an action is computed only in the Action object, it is very easy to modify it. Actions' responsibility is to collect the collection of needed information for the interaction and then compute the result according to the data retrieved.

9. Design

9.1. Simulation Scheduling

As described in Game-Loop RFC, action simulation will be realized using a quantum approach: Simulated time will be incremented globally by steps. All action occuring inside this quantum will be simulated during the same global simulation pass.

Note that there is no hard correspondance between game time and real time. Under heavy load, simulated time will slow down. Under such conditions, Server response time should stay globally constant, but simulated quanta length will become smaller.

Quantum length need only to be fixed at the beginning of the simulation pass. This mean quantum length can be dynamicly determined according to game load by the server.

9.2. Controllers and Decision Tree

The purpose of this section is to describe how an Entity controller communicates its will to the game.

Each entity controller will have to maintain a list of the actions the entity is trying to do in the immediate future. This 'TODO-List' will be called here a Decision Tree (DT).

The Decision Tree is part of the Entity model. It will be accessible to both the action simulation code and the entity controller with different level of priviledges. In other words, the object will present two different interfaces, one for the controller (limited access) and one to the interaction engine (full access).

The controller will access through the DT:

- Current Action during last turn, and progress indicator in the case of a long, multiple turn action.
- Ordered list of already planned actions by the controller.
- Ordered list of failed/impossible actions during last turn.

The action simulation engine will at each turn:

- Evaluate planned actions for execution through the use of predicates. Accepted actions are started. Process goes on until action simulation time meet TURN_DURATION or all planned actions are processed.
- Update the current action reference in the DT. If all planned actions are finished (failed or successful), then there is no current action. If an action is not finished at the end of the turn, it becomes the current action, and its progress attribute should be available to the controller. If an action finish exactly at the end of the turn and there is a next one, which is accepted by the predicates, then the latter becomes the new current action, with a null progress attribute.
- The list of failed actions is updated as well. This is needed because the controller needs to differentiate between actions that could not be performed due to time constraints, actions that appeared to be impossible when the entity tried to perform them, and actions that were started and just failed without any clear stimuli being emitted.

9.3. Action lookup

The design that we came up with makes use of the following design patterns. Next, we motivate this choice.

- Command
- Dynamic Linkage
- Object Pool

Actions are implemented using the Command pattern. This will insure:

- *modularity* since every Action implements the same "Action" interface (proposed name, not necessarily definitive),
- *extensibility* since new Actions can be created by implementing the defined interface or extending an already-created Action and overriding the methods required by the

Action interface,

- *flexibility* since for a given Action, a better implementation can be created by having to solely re-write the implementation of the methods defined by the Action interface without having to modify the rest of the game engine. We can therefore provide incremental functionality very easily.

The Command pattern already gives us much of the required flexibility. However, we also need the Dynamic Linkage pattern to be able to add an Action to an Action Pool at runtime or even dynamically modify an Action implementation. The Dynamic Linkage pattern allows us to dynamically load Actions without requiring that the game engine or Actions know each other. We will however need to refine this pattern a little bit to adapt it to our needs. Moreover, by using a special Class loader, we will be able to load Actions from a local or remote repository. For more information on the Dynamic Linkage pattern, see "Patterns in Java, volume 1" by Mark Grand.

We still have to address memory and interactivity requirements. We need to be able to share Actions between a large number of Entities in a memory- and speed-efficient way. A possible solution is to use the Object Pool pattern (as described by Mark Grand, see above). An Object Pool manages reusable instances. In our case, Action Pools will be implemented as Object Pools. How does it work and why is it efficient? Not too complex:

- Object Pools act as containers of reusable instances. Specific methods allow to get and release an instance from and to the pool.
- Object Pools can be shared by several clients (here Entities). The only requirement is to acquire a reference to a given Object Pool to use it. In our case, Entities will be initialized with a list of all Action Pools to which it belongs for each Influence.
- Retrieving an Action, once the appropriate Action Pool has been located, will be call a get-type method. However, this method will ask the pool if an Action object is available to be reused. If yes, no need to create a new one and it is returned directly to the Entity that asked for it. If no, a new Action is created and returned to the Entity. This new Action is then added to the pool.
- Several options are available concerning the management of reusable instances. We

can define a max number of Action of a given type for each Action Pool to reduce memory requirements and initialize the pool with a given number of instances (possibly the max, in which case no instance is likely to be created). This system is very flexible and accommodate for memory and speed needs. It is possible to augment the needed memory to accelerate the process by pooling a greater number of instances to avoid the overhead of instantiation or limit the number of pre-instantiated objects to reduce memory footprint.

- Using pools of reusable instances can be a huge performance factor by avoiding instantiation overhead most of the time but also by preventing garbage collection to happen until needed. Moreover, we don't need to create an Action instance per Entity which also reduces the memory footprint.

It could be also possible to use the Cache Management pattern (see Grand), to decrease lookup and increase speed performance but this will increase memory requirements.

This system seems to answer our needs rather efficiently. However, we will need to experiment to find if it keeps it promises when implemented.

9.4. Action simulation

Once the pertinent Action object has been found, control is transferred to it by the game through the perform command. This function is the base of the command pattern. As parameters, it takes:

- Entity[] invokers: a set of subject entities, trying to perform the action.
- Entity[] receivers: a set of target entities, object of the action.
- Object[] parameters: a set of extra parameters, which completely depends on the Action. They are provided by the controller. For each action it knows, an Entity controller will know the type of its required parameters. If an action receive bad parameters, perform will throw an ActionBadParameterException.

Concrete action simulation will be done by:

- Impacting state of various entities in the game.
- Sending stimuli (directly or indirectly while impacting entities).
- Sending events.

DT Action objects provided by Instances (and then Java compiled) will be able to use the following interfaces:

- Read Access to the Properties, the Capabilities, the Skills of any Entities. Write access when meaningful.
- Full Access to the Decision Tree of any entity.
- Full Access to the Stimuli Dispatching system.
- Full Access to the Event Dispatching system (if applicable).

Some actions will require coordination between two or more entities. For example Buying something requires that the shopkeeper is able to sell its products and not busy doing something else. Joining a group requires that the existing group is offering the newcomer to join.

In such cases, two actions will work in pair. The "Buy" action will only succeed if the shopkeeper is performing the "Sell" action. The Buy action will simply look at the current action of the Shopkeeper entity. If it finds a "Sell" Action there, It will contact this action, using a second method: Object [] interact(Object []).

For example, the interaction between the buyers and sellers could looks like:

- Buy -> Sell.interact("BuyProposal","WoodArmor", 12 credits);
- Buy <- Sell ("BuyResponse","OK", "WoodArmor", 12 credits);
- The Buy actions then does the transaction, effectively removing the Wood armor from the shop inventory to the traveller's one. Money is transferred to the shop.

This proposal is just a beginning to define real interactions between entities. More use cases need to be reviewed, like:

- Flying gnome falling on the ground. Will he get killed?
- Adventurer joining a group.
- Simple tavern fight.
- Carrying someone on your shoulder. (up and down).
- Two magicians casting a spell together, effectively joining their power.
- Two character trying to pick up an object at approximatively the same time.

9.5. Action interruption

Another difficult point to design in which case an action being performed will be interrupted. We can do the following remarks:

- At the end of each turn, the controllers are given the opportunity to update their Decision Tree. In the case the entity is performing a long action, the controller is free to cancel it. The cancel decision will go through the predicate system to check if it's acceptable. Some action are beyond the will of the controller: a falling gnome can not decide to stop falling. A dying character can not decide to stop his agony...
- An aggressive action from another entity can interrupt the current action. (This is exactly the definition of an aggressive action). The target entity will be only able to react at the next turn.
- The subject entity can enter or quit an influence. If it impacts the current action, most of the time, this means that the action will become impossible and will be interrupted. This is exactly the use case of the flying gnome. When the gnome leaves the spell influence, it loose the ability to fly and as a consequence, the action "Fly". The remove process check wheter the current action is affected. In this case, it is. The "Fly" action is interrupted, and a "Fall" action is scheduled for the next turn.

10. Types of Actions

10.1. Framework Actions

Metacosm's interaction engine provides a set of primitive Actions that provide game developers with a sort of assembly language to create new Actions without having to write code. See Dynamic Actions for more details. Framework Actions are closely tied to the implementation of the framework since they provide the basic facilities to act on Entities. They are coded directly in Java.

10.2. Instance Actions

Instance Actions are Actions that have been coded in Java and then compiled. They are made available to the interaction engine via dynamic loading. At the implementation level, they are really close to framework Actions since both are compiled code. However, framework Actions are part of the framework and provide, most of the time, means of interacting on Entities that would otherwise be impossible. Instance Actions are Metacosm instance specific even though it is possible to envision that really useful instance Actions could mature to the point of becoming framework Actions.

10.3. Dynamic Actions

Dynamic Actions are Actions that are NOT compiled and that don't need to be written in Java to be used in Metacosm. They are the means via which game developers can create new Actions without knowing how to program in Java. More precisely, they can be defined via a graphic editor that allows the chaining of already defined Actions (whether those are framework, dynamic or instance Actions) to create new and/or more complex Actions. They could also be automatically generated (without human input) as well. We can imagine that dynamic Actions could be re-written in Java over time to improve efficiency when a programmer is available. However, such an operation shouldn't be needed to enjoy Metacosm and create new Actions. Moreover, dynamic

Actions are the only (current) possibility for the interaction engine to create new Actions on the fly when so needed.

10.4. Automatic Actions

It is pretty difficult to find an adequate system for hard-time constraints Interactions. A more elegant solution than this problem would be to enable Actions to be made automatically when a request cannot arrive in due time or has been lost. This solution has the advantage to be relatively simple and efficient. Metacosm's Interaction engine could automatically provide default Actions. It could however be more gratifying for users if they were able to define which Action to be undertaken in case they cannot respond in a short enough time to maintain tight timing constraints.

Inside the Game, Automatic actions can be seen as the Reflexes of an Entity. As regular actions, Automatic actions are defined by Influences. Additionnaly, they can be parametred by the controller (ie the player for a player Entity) and activate themselves when the context requires it. More precisely, Automatic Actions are actions with a minimal AI code which is able to activate the Action automatically.

Practically it means that controlled Characters will be able to defend themselves spontaneously to attacks even if their controller do not react in time.

11. Implementation

11.1. Interaction system

1. Controller receives Stimuli and has access to both failed and successful actions tried during last quantum.

2. Using the Entity's memory, the Controller's memory and the Entity's state, the Controller chooses an Action (and its parameters) for the next turn.
3. At the next turn, the Entities trie to retrieve the chosen Action (in Actions pools). If it fails, the Entity does nothing; an error could be logged to detect broken Controller, but it's more probably a context change (ex: you can't fly anymore because the spell ended).
4. The Action is resolved/performed. It implies Stimuli are sent to the StimuliDispatcher, self perception is sent to the Controller, and Entities are modified.

11.2. Performing an Action

Here we describe the simple sequence of messages needed for an Entity to perform an Action.

1. The controller asks the Entity to perform an Action. When it does so, it invokes the `perform(String actionName, Entity[] invokers, Entity[] receivers, Object[] parameters)` method on the Entity.
2. The Entity that is asked to perform the action looks up for the given Action in its individual list of Actions. If the Action is not available, the Entity looks up in the list maintained by the Influences it is influenced by in the priority order of these Influences. If the Action is still not found, it launches an Exception to indicate that it can not perform the action asked.
3. Once the Action is located, the Entity retrieves the given Action object.
4. The Entity then delegates the execution of the action to the retrieved Action object by invoking the `perform(Entity[] invokers, Entity[] receivers, Object[] parameters)` method on the Action.

11.3. The `perform` methods

In the case of Entity's `perform` method, the first argument is the name for the Action to be performed and not the Action in itself. This name will allow efficient lookup in actions lists and will not require the controller to obtain a reference to an action. By using this indirection level, it is possible to have several implementations of the same Action. It is the responsibility of the Entity to retrieve the appropriate Action. More precisely, the Action that will be used is the one that is first retrieved according to the priority order of the Entity's Influences.

Moreover, the controller needs to know about the result of an action, so, there should be some kind of return but not in a form of a simple return value. The controller will have two sources of information:

- The entity's perceptions
- The Decision Tree which will publish list of actions that the entity has been table to try performing and the other it was unable to.

Hence the complete signature for the `perform` method would be: `int perform(String actionName, Entity[] invokers, Object[] parameters) .`

Parameters will a reference to an array of Objects, potentially null. Each Action will require a different set of parameters. The simplest case of parameter will be an Entity ID. (Example for action "take object": parameter = object EID). Other possible parameters type will be:

1. - a Typed Value (time delay, date, distance, weight, etc..)
2. - a Position (coordinates) in a place.
3. - a Type of Entity. (an influence)
4. - a Message
5. - a Group of Entity (Group ID/Reference).

Return value of perform will be one of the following constants:

1. - COMPLETED_ACTION
2. - FAILED_ACTION
3. - CONTINUING_ACTION

11.4. Decision Tree

11.4.1. Context

Each controlled entity maintains a list of actions planned for next turns. We will call an intention a reference to an action with all context parameters. The next action to perform will be determined by the success or failure of the previous actions: Intentions will be organized in structure called the Decision Tree

Obviously this list will be managed by the entity controller during the last phase of each turn. The controller will be able to know which actions have been declared impossible by the predicate system, which one have been executed during the turn, which one were delayed and remain in the list.

The controller will be able to cancel intentions which were not already started, and interrupt currently executed actions.

The result of an action will be known to the controller only through the perception system.

Long actions whose length is more than one turn will have a progress attribute, which is known to the controller. When the entity is the target of an external action while busy with an action, two situations are possible:

1. the external action is coercive (for example, an attack) and the current long actions is immediately interrupted.

2. the external action is not coercive, and the current action continues at least for the current turn.

In both cases, at the end of the turn, the controller can choose to react to the external action and interrupt the current action or to continue this action.

11.4.2. Interfaces exposed to entity controllers

At the end of each turn, all entity controller will be given the opportunity to update their Decision Tree. They will use the DecisionTree interface.

DecisionTree interface:

1. IntentionRef program (long date, ActionId action,...); define an intention, schedule it in the list for date "date" and returns a reference for it. If many intentions are defined for the same date, this means, do the following actions as soon as possible, from this date in the order they have been defined.
2. Iterator iterateOnIntentions(); Return an iterator on the set of all defined Intention in chronological order.
3. void deprogram (IntentionRef ref); cancel an intention before it becomes active using its reference.
4. void cancel (IntentionRef ref); cancel the current action (for long actions)
5. Intention getIntention(IntentionRef ref); Retrieve the complete Intention data object using its reference
6. IntentionRef getCurrentIntention(); Retrieve the reference of the intention of the current action.
7. int getProgress(IntentionRef ref); Retrieve the value of the progress attribute of a long action.

Event message Each time an event concerning its decision tree happens, the controller will receive messages through a call back function: HandleEvent(Event e). The different decision tree messages will be:

1. Intention_Started
2. Intention_Completed
3. Intention_Failed
4. Intention_Interrupted

11.4.3. Interface exposed to the game scheduler

As the game controller will only manage the Decision Tree through a well defined interface, the Decision Tree implementation is a protected data structure. It can then be used by the framework to schedule actions which are not directly under the control of the controller: for example, the game can schedule the contact between the character and the ground after a jump, at the time determined for the end of the fall..After a deadly wound, the combat management code can schedule an action "die" for the next turn.

This means that only a subset of the defined actions are accessible to the entity controllers. The other actions will be used internally by the game system.

To make use a this facility, the simulation classes will have access to more functionalities of the Decision Tree than the controller itself. We will have to provide a more powerful interface in this case. In the context of the simulation, this interface will be called ActionTreeInterface. This interface will extend DecisionTreeInterface and will also define:

1. Iterator iterateOnActions(): returns an iterator on the set of all defined Actions (Intentions + Actions) in chronological order.
2. setStarted(IntentionRef ref): marks an intention as current.
3. setCompleted(IntentionRef ref): marks an intention as completed; removes from list and send event to the controller.
4. setProgress(IntentionRef ref, int value): updates the progress attribute of an intention.

5. setFailed(IntentionRef ref): marks an intention as failed. Removes from list and send event to the controller.
6. setInterrupted(IntentionRef ref): marks an intention as interrupted. Sends event to the controller.

11.5. Action scheduling

Once the action has been chosen by the controller, it is processed by the framework in three steps:

1. Actions scheduling: what are the planned actions for the entity, are these actions possible, does the entity has enough time to perform them?
2. Actions simulations: potential modifications of game and entities properties, planning of possible consequent actions.
3. The action impacts the game and stimuli are send, allowing the controllers to plan future actions.

Actions scheduling is done by the framework, by example in the Entity class. A simple way to express this is to speak of the 'Action scheduler'.

The Action scheduler examines the Decision Tree at each turn, and decides which actions are performed. We can list the following cases:

1. Simplest case: no current action inherited from the last turn, and one action in IL. The actions is checked for acceptability using the 'MayStart' Predicate. If the results is negative, the action is marked as failed. If the result is positive, the control is given to the action, with a reference to the Context.
2. Long action case: one action is inherited from the last turn and marked current. The Action scheduler check if this action can continue using the 'MayContinue' predicate. If ok, the action continues, if not it is marked as interrupted.

3. Consecutive actions case: more than one action can be performed in one turn, if they are short enough. The time taken by each performed action is known to the AS because this information is provided in the return of the perform function.
4. No action case. Nothing to do. Current action reference remains null.

For each phase, we will provide some examples to make things clearer.

11.6. Action simulation

The action can be implemented as singleton. All state will be managed externally, through the Intention objects. So it is the time to define more precisely what an Intention is:

- An action reference (can be a string = the action name)
- The action state: reference to context, Subject and Target Entities, Action parameters, internal progress attributes...

Each Action will be free to define what it wants to store in its state data. This data will be private and only useful to the Action code.

In the this section we should describe exactly which interfaces are available to performed actions. We already proposed a list of them:

- Read Access to the Properties, the Capabilities, the Skills of any Entities. Write access when meaningful.
- Full Access to the Decision Tree of any entity.
- Full Access to the Stimuli Dispatching system.
- Full Access to the Event Dispatching system.

We needs to define full access interfaces to the Perception System and the Event Dispatching System.

11.7. Controller planning

The controller interface will contain a call back method use to provide CPU time to the AI in order to plan or reconsider its Decision Tree. This function may be called plan.

11.8. Example

Example 1. Raoul trying to pick up flowers

Raoul the baker is in vacation and walk in the mountains. He sees some beautiful flowers and thinks they would make a great gift for his girlfriend Gertrude. So Raoul's fine AI controller plans the following consecutive actions:

1. walk towards the flowers.
2. pick up the flowers.

This plan is translated into the Decision Tree of the Raoul Entity. More precisely, the IL now contains two elements: - an Intention object referencing the Action "go" with a parameter equal to the EntityID of the flowers. - an Intention object referencing the Action "take" with a parameter equal to the EntityID of the flowers. Both Intention object are still simple only intention, the action have not started, and they don't reference any Action state objects.

First scenario, Raoul successfully perform both actions. During the next quanta, the game check for the possibility to perform the first action, and compute a positive answer. The entity Raoul really tries to perform this"go" action. First we goes through Action initialisation code, which takes 2 parameters: the subject Entity, the Intention Object. This initialisation phase creates an ActionState instance where it puts data it needs. In this example the data object can contain Raoul's initial position, current position and target position in terms of place coordinates. The target position is extracted from the Memory of Raoul: the flowers belong to his field of vision, if they would not, the action initialization would have failed (and the action too). After initialization, the Intention object in the tree contains the reference to the newly created ActionState object.

Then Raouls starts to actually perform the action, "Go" becomes the current action. perform() is called. The method checks the path and the distance between current and target position. If there is no way that exist and that Raoul can use, the Action fails (exemple if target = a cloud). If there is a way, the Action computes the time the move will take. If this delay is longer than the remaining time in the quanta, the action updates its state (current position = position at the end of the quanta)and the entity position in the place and return the "Action continuing" value. If the action ends before the end of the quanta, the action update the Entity state, discard its state and return the "Action completed" value.

The same procedure applie for the second Action. We suppose the quanta is still not over for the entity. First the game check if the action is possible. We suppose it is. Then it initialize it. Again, the initialization succeeds, before the flowers are known to Raoul from his visual memory.

The perform method this time does not compute the length of the action, because the "Take" action is supposed to take a constant time. But it checks that the object of the action is close enough from Raoul. The check succeed, then the flowers are removed from the scene and put into Raoul’s inventory. The action returns the constant "Action performed".

In fact, Raoul may not have totally used its time quanta, but his Decision Tree is now empty. So he does not perform anything more. The simulation thread can process another entity.

Next time it will be activated, Raoul’s controller will know that both actions in IL have been tried and in some way completed. From his perceptions, he will see that the flowers are gone from the place, and that they are now in his backpack.

Other scenario: the action fails: at the next turn, the game checks the predicates for the first action: ’walking’. It appears that Raoul is not experienced enough to climb up to the flowers. So the first actions fails. As a consequence, the second action is also impossible. Both fails, and Raoul does nothing this turn.

Let’s describe more precisely what happens: the initialization of the first action " succeeds as in the first case. But the perform method, while evaluating the actions, estimates that the path to the flowers is almost vertical, and that would be too difficult for the beginner alpinist that Raoul is. So the methods returns the constant value "Action Failure". Raoul's time is also translated with an estimation of the time it would take to him to give up climbing to the flowers.

The second action "pick up" initializes too. But again, while the perform method is called, it returns "Action Failure" because the flowers are out of reach from Raoul's hand. And no other actions are in Raoul's Decision Tree.

During the next quanta, Raoul's controller will know from the Decision Tree, that both actions "failed". Raoul will know its new position through the perception system.

Other examples: complex action parameters.

1. Raoul open his door with his key: perform("Open" ActionID, Raoul's Entity ID, [door of Midgaard's bakery EID, Raoul's key EID]) Object []: object to open, tool/object used to open (optional).
2. Raoul prepares bread pasta: perform("Cooking/Mix" ActionID, Raoul's Entity ID, [kneader (container) EID, flour bag EID, water container EID, salt container EID]) Object []: 1st element: target container, other elements: ingredients containers.
3. Raoul prepares bread from pasta: perform("Cooking/Setup" ActionID, Raoul's Entity ID, [kneader EID, Bread TypeId]) Object []: 1st element:ingredient which to give form, 2st element: Type of product to be prepared.
4. Raoul puts breads into the oven: perform("Move" ActionID, Raoul's Entity ID, [Oven EID, Bread1 EID, Bread2 EID, Bread3 EID,...]) Object []: 1st element: Target position, other: Objects to be moved.
5. Raoul wait 10 minutes perform("Wait" ActionID, Raoul's Entity ID, [time delay=10min Value]) Object []: 1st element: time delay Value.
6. Raoul says to his wife the breads are ready: perform("Talk" ActionID, Raoul's Entity ID, [Raoul's wife EID, Message]) Object []: 1st element: target Entity of message, next: message.

11.9. Action consequences

Example 2. Picking up a flower

The flower is removed from the place, and stimuli are emitted. The (cut) flower is added to the character’s inventory.

Example 3. Jumping in a deep hole

Stimuli describing the move are emitted. An action ’falling’ is planned for the time of crash.

Example 4. Falling on a surface

Stimuli describing the actions are emitted. If the shock is strong enough, the entity will be killed. In such a case, an action "die" is planned for the next turn.

11.10. Long actions and "hibernation mode" of entities

(Not in M1? To be completed.) Place in the game in where no players are active and where there is no perception should save CPU power. This can be done by using special long actions called hibernation actions, during which no action planning is done by AI controllers. The Stimuli Dispatcher will be able to interrupt these actions, which will effectively "wake up" the entities. Again, the stimuli dispatcher will also be responsible to decide to start of the hibernation.

Example 5. In the bakery

Raoul the baker has not seen any visitors for the last few turns. The StimuliDispatcher of the ’Bakery’ Place knows there is no visitor around. It sends a special message to the

Entity Raoul to signify that the place will start to hibernate. Raoul's controller then starts a long action "continuous baking": continuous baking means that Raoul will bake bread continuously until something interrupts him. The result of the actions will only be computed at the time of this interruption. For example, if Raoul bakes continuously for 30 turns, he will obtain 20 pieces of breads and 10 sandwiches.

It should be possible to avoid this extra complexity and at the same time building a game CPU savvy: build the AI of all common entities so that their common state would be a long action similar to 'continuous baking'. In the case of Raoul, the Dispatcher will interrupt the action when someone enters the Bakery, we don't need to provide extra code for CPU resource management.

11.11. Implementation data

The DecisionTree will be implemented with (To be completed).