NICE project (IST-2001-35293)

Natural Interactive Communication for Edutainment

NICE Deliverable D1.2b

Formal representation of domain information, personality information and dialogue behaviour for the NICE fairy-tale game

18 November 2004

Authors

Johan Boye, Joakim Gustafson and Mats Wirén

Voice Technologies, TeliaSonera

Project ref. no.	IST-2001-35293
Project acronym	NICE
Deliverable status	Internal
Contractual date of delivery	1 Nov 2004
Actual date of delivery	18 Nov 2004
Deliverable number	D1.2b
Deliverable title	Formal representation of domain information, personality information and dialogue behaviour for the fairy-tale domain in the second prototype
Nature	Report
Status & version	1.2
Number of pages	23
WP contributing to the deliverable	1
WP / Task responsible	TeliaSonera
Editor	
Author(s)	Johan Boye, Joakim Gustafson and Mats Wirén
EC Project Officer	Mats Ljungqvist
Keywords	
Abstract (for dissemination)	

Table of Contents

1 Introduction

1.1 Definitions and scope

This deliverable outlines a method for specifying domain information, as well as the behaviour and personality of virtual animated characters, in the second NICE fairytale-world prototype. Such specifications are used for *scripting* the dialogue management kernel (described in deliverable D5.2b) to decide what a given character is supposed to do and say in all the various situations that might arise during execution of the system.

The word "scripting" above merits some explanation. The second prototype of the fairy-tale game contains two fairy-tale characters with different personalities. Moreover, the behaviour of each character is supposed to change over time, reflecting supposed changes in the characters' knowledge, attitudes and states of mind. However, when considered at an appropriate level of abstraction, many of the functions a dialogue manager needs to be able to carry out remain constant regardless of the character or the situation at hand (for instance interactions with other modules in the system, reference resolution, procedures that reason about beliefs, goals and actions, response generation, etc.). Thus there is a good case to be made for organizing the software of the dialogue manager into a *kernel* laying down the common functionality, and *scripting code* modifying the dialogue behaviour to be suitable for different characters and different situations. Such a model of code organization is common in the computer games field (see e.g. Varanese and LaMothe 2003). In the spoken dialogue systems field, it is desirable for both practical and theoretical reasons. On the practical side, it allows for the development of systems that are simpler to understand and maintain. On the theoretical side, it helps distinguishing the dialogue management concepts that are actually generic from those that are situation-specific. Such knowledge can then increase our understanding of dialogue in general.

Thus, to get the full picture on how dialogue management is carried out in the NICE fairy-tale world system, the reader is encouraged to consult deliverable D5.2b, which describes the workings of the dialogue management kernel, as well as this deliverable, which describes the scripting possibilities.

1.2 Structure

Section 2 describes the game scenario in the fairy-tale world system. Section 3 and 4 how respecively domain information and task information is coded. Section 5 describes how to script the dialogue behaviour and personality of a fairy-tale character.

2 Game scenario

2.1 Background

The game scenario of the NICE system is basically that of a player interacting with embodied fairy-tale characters in a 3D world via spoken dialogue as well as graphical gestures via a mouse-compatible input device to solve various problems. The fairy-tale characters communicate with the player using spoken dialogue and gestures. The appearances of the characters, their voices, actions and ways of expressing themselves all contribute to give the player the impression of fairy-tale characters with distinct personalities.

Figure 1. Prelude to the game: Cloddy Hans saying farewell to H. C. Andersen on his embarking to Copenhagen.

2.2 First scene

The first scene of the fairy-tale game includes a single embodied character, Cloddy Hans (loosely inspired by the character from H. C. Andersen's story with the same name). Cloddy Hans is adapted as follows: He is a bit retarded, or so it seems. He cannot read and only understands spoken utterances and graphical gestures at a rather simple level. He does not take a lot of initiatives, but is honest and anxious to try to help the user. In spite of his limited intellectual capabilities, he may sometimes provide important clues through sudden flashs of insight. Most importantly, he is the user's faithful assistant who will follow him/her throughout the game.

The game begins in H. C. Andersen's house in Copenhagen in the 19th century. Andersen has just left on a trip to Odense, and has asked one of his fairy-tale characters, Cloddy Hans, to guard his fairy-tale laboratory while he is away (see Figure 1). The key device in the laboratory is a fairy-tale machine, which nobody except Andersen himself is allowed to touch (Figure 2). On a set of shelves beside the machine, various objects, such as a key, a hammer, a diamond and a magic wand, are located (Figure 3). By removing objects from the shelves, putting them into suitable slots in the machine and pulling a lever, one lets the machine construct a new fairy-tale in which the objects come to life.

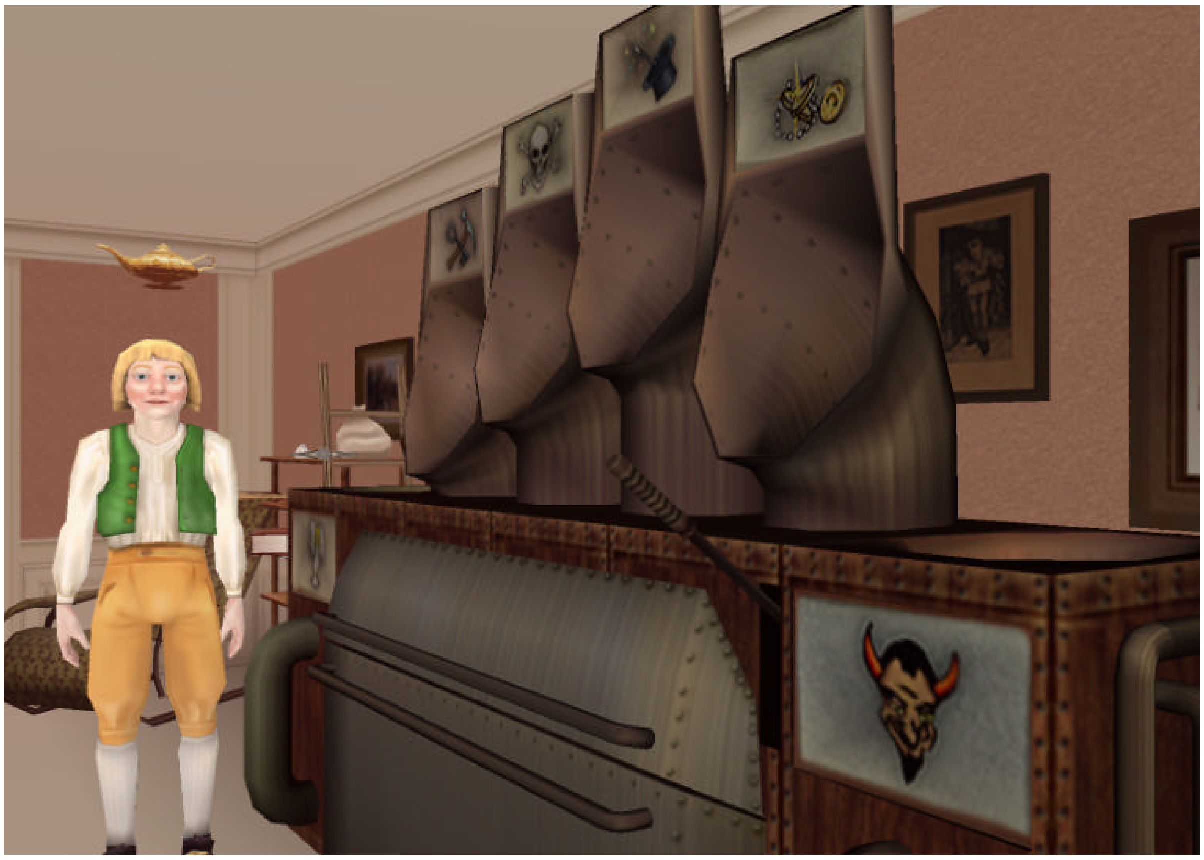


Figure 2. Inside the first scene: Cloddy Hans with the fairy-tale machine.

However, just before the user enters the game, Cloddy Hans has violated the rules by taking one of the objects and putting it into the machine. As nothing harmful has happened, Cloddy Hans gets the idea of surprising H. C. Andersen with a new fairy-tale on his coming back. There is a problem, however: Each slot is labelled with a symbol which tells which type of object is supposed to go there, but since Cloddy Hans is not very bright, he needs help from the user with understanding these. There are four slots, which are labelled with symbols denoting “useful”, “magical”, “precious” and “dangerous” things, respectively. Which object goes in which slot is sometimes more obvious (provided you understand the symbols), like the diamond belonging in “precious”, and sometimes less obvious, like the knife belonging in “useful” rather than “dangerous”.

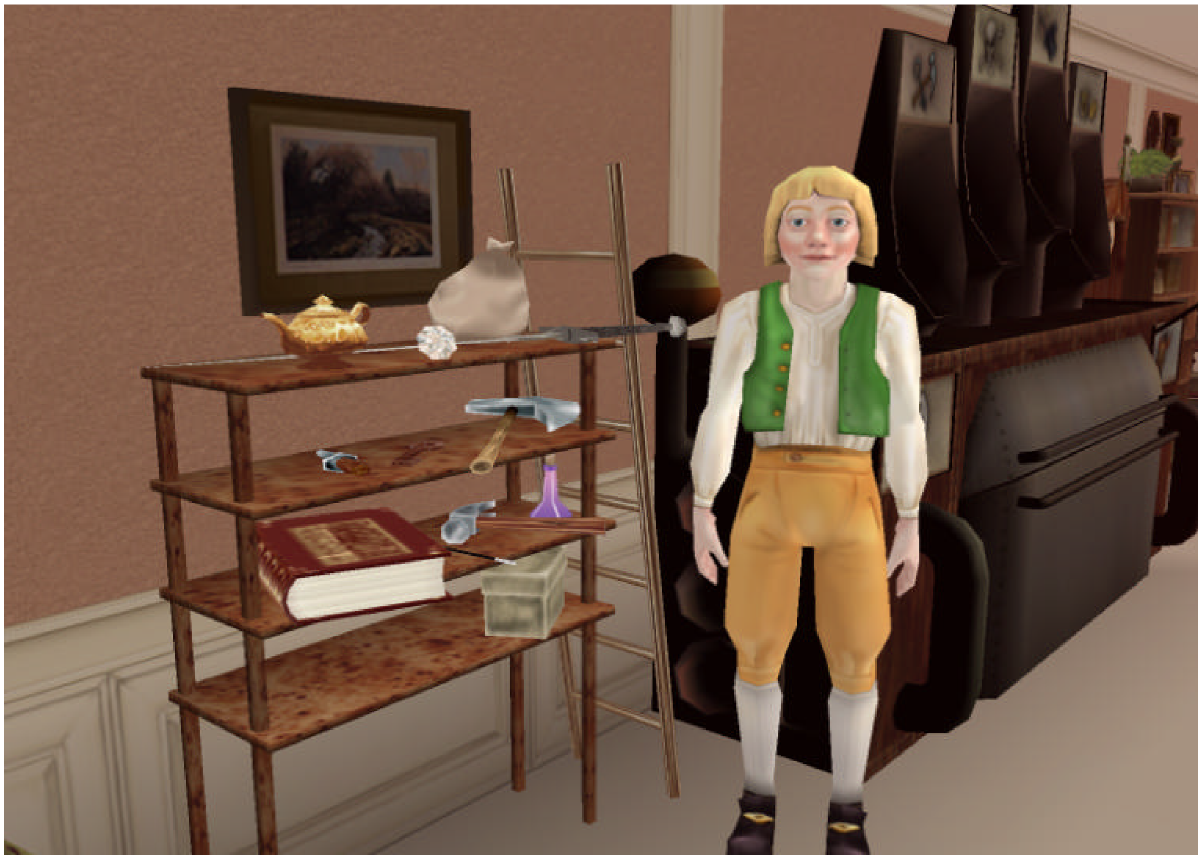


Figure 3. The first scene: Cloddy Hans beside the shelves with objects.

The first scene thus develops into a kind of "put-that-there" game, where it is the task of the user to instruct Cloddy Hans; tell him where to go, which objects to pick up and where to put them down, etc. If the user does not understand what to say, Cloddy Hans will encourage him or her, give suggestions, and eventually take matters into own hands. However, experiences from earlier data collections in the project indicate that the players almost immediately understand the idea (see deliverable D2.2b).

Because the initial scene is task-oriented in a straightforward way, the system is able to anticipate what the user will have to say to solve it. The real purpose is not to solve the task, but to engage in a collaborative grounding conversation where the user learns what the fairy-tale objects can be used for and how they should be referred to. This process also lets the players find out (by trial-and-error) how to adapt in order to make it easier for the Cloddy Hans to understand them, e.g. by using multimodal input in certain contexts. The intention is to make the interaction smoother in the subsequent scenes in the fairy-tale world, since the objects that appear in it already have been grounded in the initial scene.

2.3 Second scene

In the second scene, the player enters the actual fairy-tale world for the first time, together with Cloddy Hans. The fairy-tale world is a large 3D virtual world, see Figure 4.

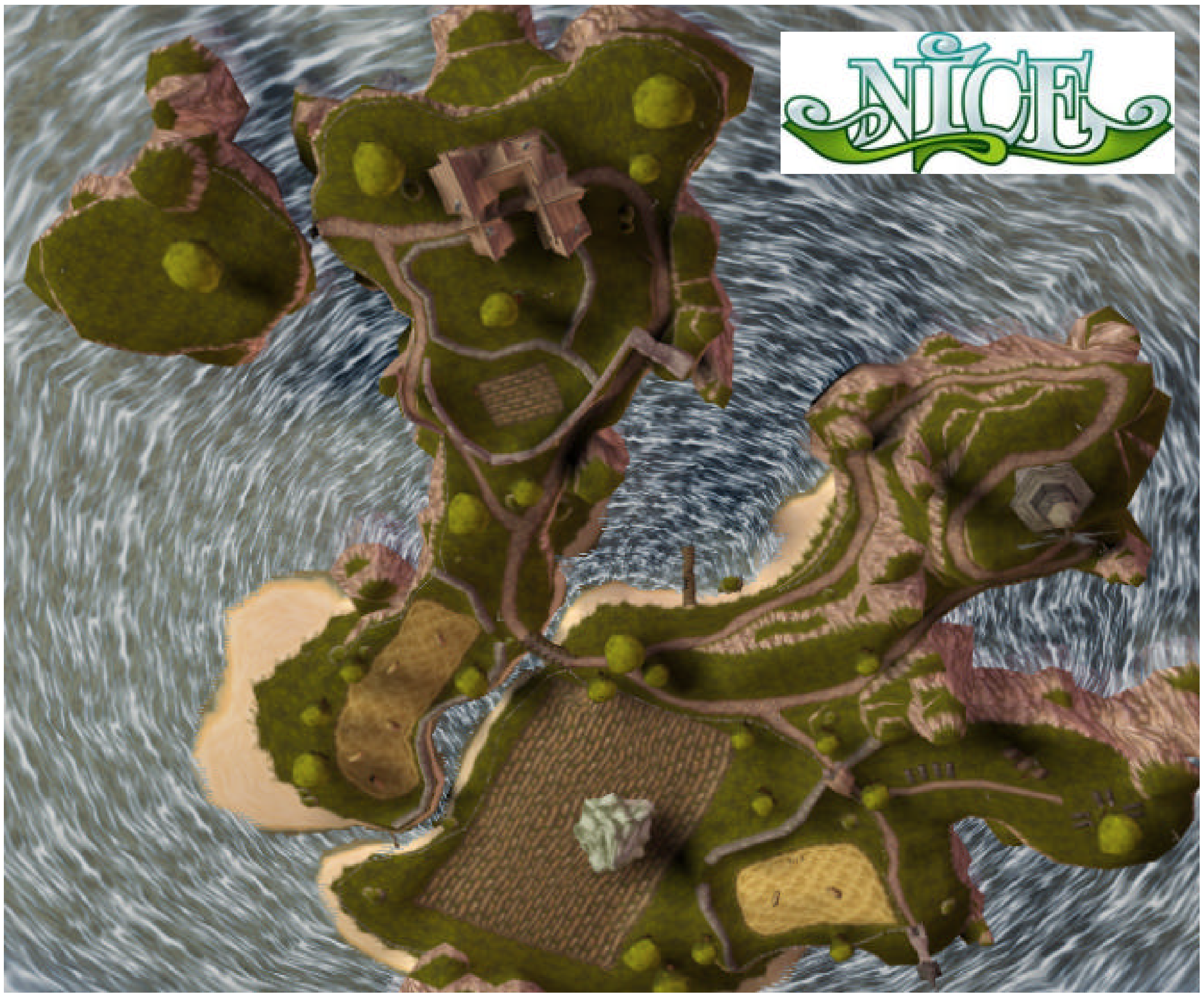


Figure 4. An overview map of the fairy-tale world.

The second scene takes place on the small island in the upper left part of this world, see Figure 5.

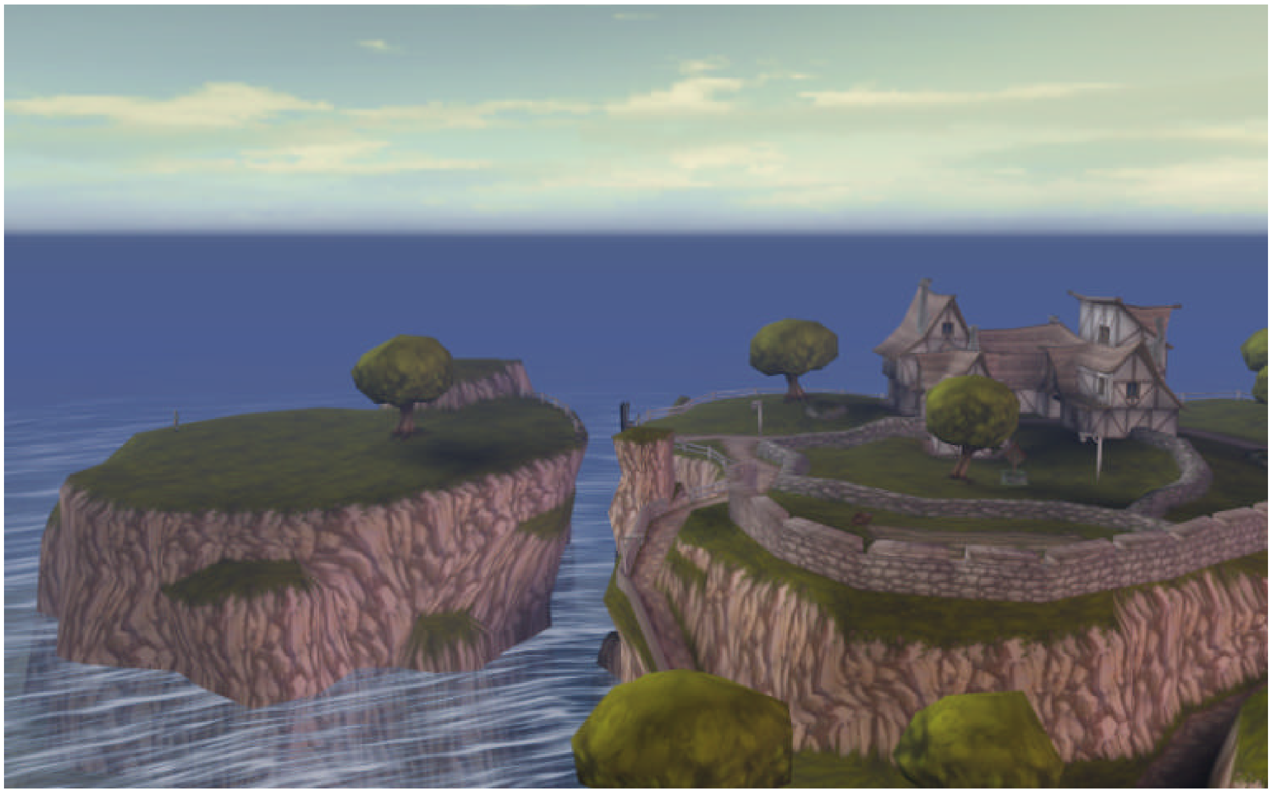


Figure 5. The second scene: A small part of the fairy-tale world. The player and Cloddy Hans start off on the small island on the left hand side.

At the beginning of the second scene, Cloddy Hans encourages the player to explore the immediate surroundings on the small island. While wandering about and looking around, the player discovers that the objects that were put in the fairy-tale machine in the preceding scene are now lying scattered in the grass. Although it is not completely clear to the player at this point, these objects will actually constitute valuable assets when solving various tasks in the world. Cloddy Hans is able to refer multimodally to object found in the grass, and if the user tells him he will pick them up, see Figure 6.

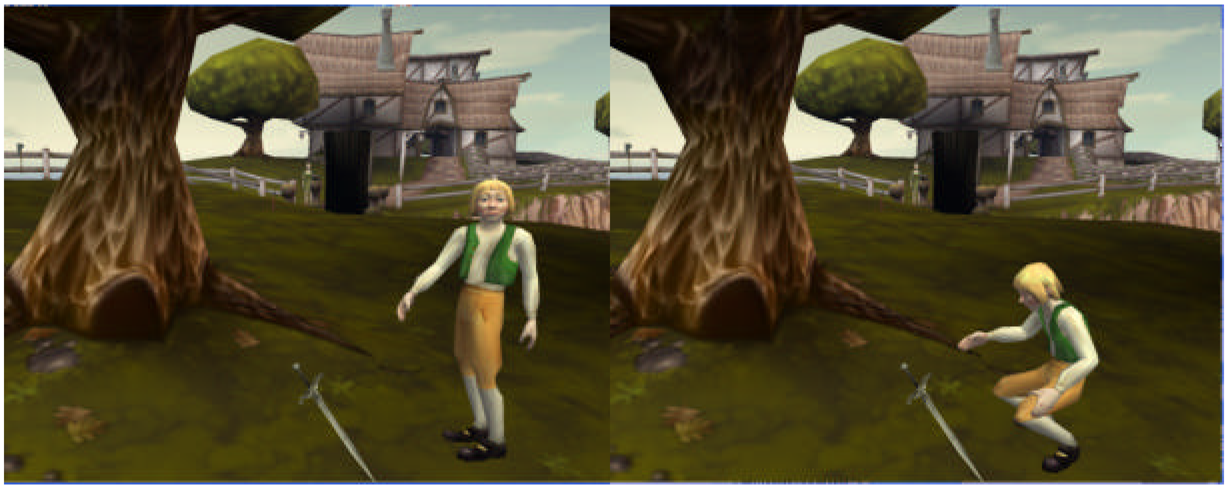


Figure 6. The second scene: Cloddy Hans finds an sword, he first points at it asking if the user what to do with it. When the user tells him to, he picks up the sword.

The player soon encounters the first problem. Together with Cloddy Hans, he is trapped on a small island, from which he can see the marvels of the fairy-tale world – houses, fields, a wind mill, and many more things – but they are all out of reach. A deep gap separates him from these wonders. There is a drawbridge, which can be used for the crossing, but it is open, and the handle that operates it is on the other side. Fortunately, a girl, Karen, is standing on the other side (see Figure 7).

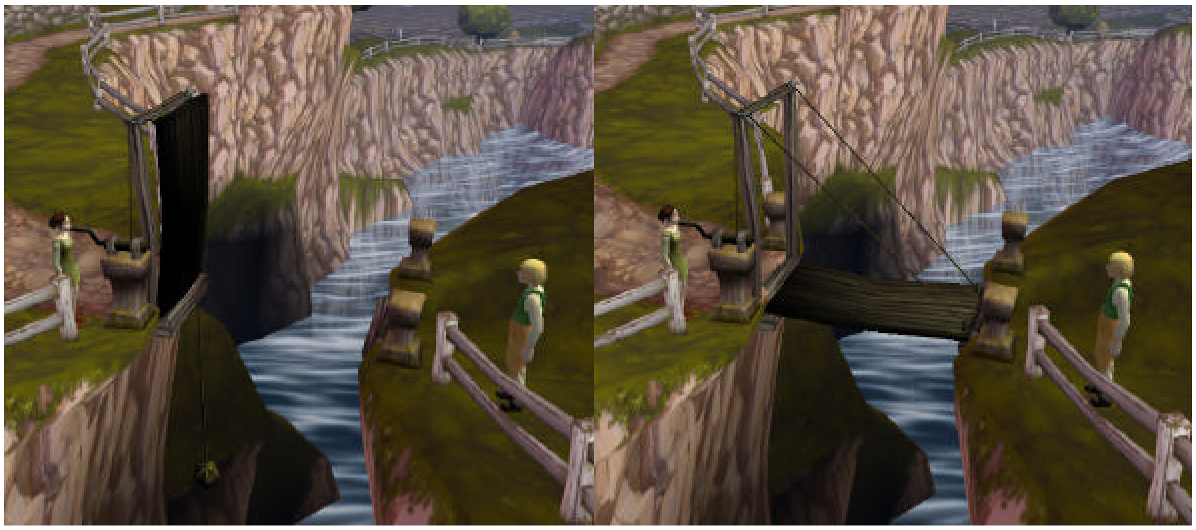


Figure 7. The second scene: Cloddy Hans and Karen at the gap and the open drawbridge.

Karen is a feature character in the game with a gatekeeper function. She is loosely inspired by the main character in H. C. Andersen’s story “The Red Shoes”. Karen has a different kind of personality compared to Cloddy Hans. Instead of having Cloddy Hans's positive attitude, she is sullen and uncooperative, and refuses to close the drawbridge. The key to solving this deadlock is for the player to find out that Karen will comply if she is paid: she wants to have one of the fairy-tale objects that are lying in the grass on the player's side of the gap (which object she wants will change each time the game is restarted). Thus, it is the task of the player to find the appropriate object, and use this object to bargain with Karen. It turns out that what she is especially interested in jewels. There are three jewels (a diamond, a ruby and an emerald) lying in the grass on the island, see Figure 8.

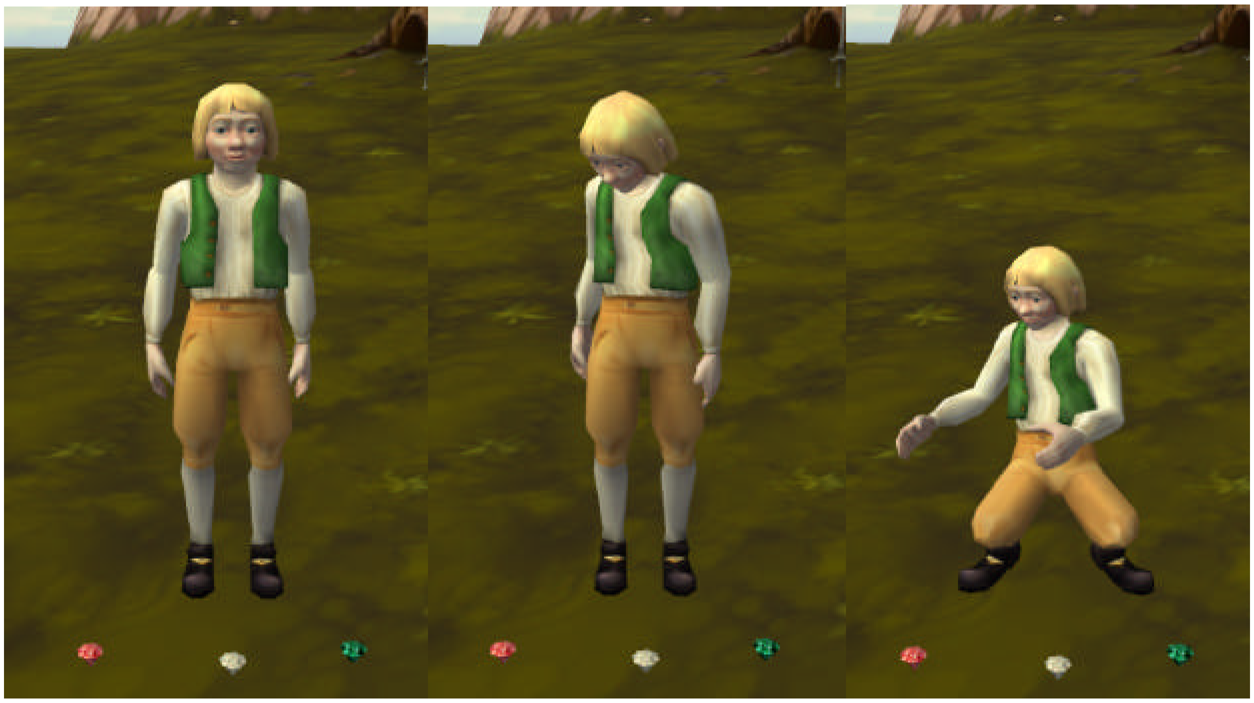


Figure 8. The second scene: Cloddy Hans looks at ruby and the crouches to pick it up.

In this phase it is possible to encourage graphical gesture references by letting Cloddy Hans say that he doesn't know what a ruby looks like, and if the user says "*pick up the red jewel*" he might state that he cannot see the difference between green and red. Another possibility is to have more than one ruby.

When the users has identified which jewel Karen wants, gotten Cloddy Hans to fetch that jewel to the drawbridge, and promised Karen that they will give it to her when they get over, she will lower the bridge, and let the player and Cloddy Hans pass. As in the first scene, Cloddy Hans will provide the appropriate hints if the user does not understand what to do.

2.4 Role of the user

The user perceives the fairy-tale world through a first-person perspective. Hence, there is no user avatar, but the user is still perceived as appearing in the world by other characters in the game. The user's means of action in the world are:

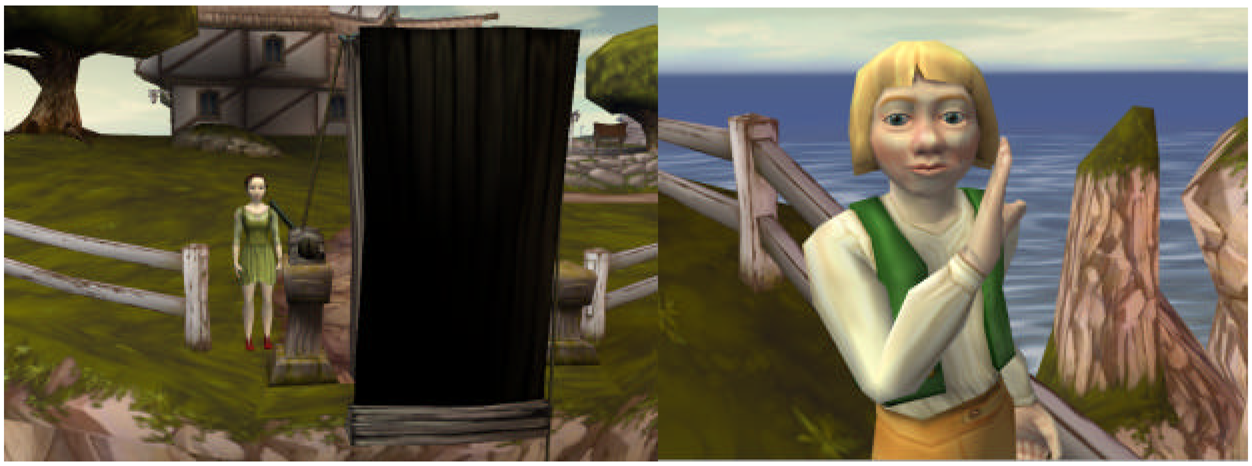
- speaking to other characters in the game
- pointing and gesturing at arbitrary characters, objects and locations.

When the user meets fairy-tale characters of the game, they are typically shown in full-body camera angle. Various 3D objects also appear in the environment. The user can ask the characters to manipulate objects by referring to them verbally and/or by using the mouse. To understand the reason for these rather limited capabilities of the user (in particular, the lack of direct manipulation), we have to recall what distinguishes NICE from other games, namely, spoken multimodal dialogue. We thus want to ensure that multimodal dialogue is appreciated by the user not just as an "add-on" but as *the primary means of progressing in the game*. Our key to achieving this is to deliberately limit the capabilities of the key actors, the user and Cloddy Hans, in such a way that they can succeed only by cooperating through spoken multimodal dialogue.

2.5 Role of Cloddy Hans

The users only way of manipulating objects in the 3D world is to get one of the fairy tale characters to do it. However, most characters have their own goals and plans that might conflict with the user's desires. The only character that is always willing to help the users is Cloddy Hans. He has no goals and plans of his own other than being the user's friend and helper. He is friendly, helpful, thorough and calm, but a bit stupid and uncertain. If the users doesn't seem to know what to do (or if the understanding modules have failed for a certain number of turns), he is able to guide the users by giving them suggestions on what to do next (apparently through sudden flashes of insight). This means that Cloddy Hans and the user form a team: Cloddy Hans can perform physical action, but he does not know what to do, while the user knows what has to be done, but he cannot perform the physical actions needed. This means that the user and Cloddy Hans have to cooperate to solve the proble ms put before them.

In the second scene the user is supposed to give Karen something to make her want to lower the drawbridge. This is a bargain situation, where Karen will refuse to accept unattractive objects, like the sack. If the user is stuck in this deadlock situation for some time, Cloddy Hans can attract the user's attention by saying "psst". The camera then turns from Karen to Cloddy Hans, who then will give the user a hint on what to tell Karen to change her mind (see Figure 9). After the Cloddy Hans has given his clue the camera turn to Karen again, allowing the user to talk to her.



Karen: I don't want an old sack! **User:** *is silent* **Cloddy:** Psst…tell her that it is magical.

Figure 9. The second scene: Cloddy Hans and Karen at the gap and the open drawbridge.

3 Domain model

To be able to reason about what to do and say in a given situation, each character has a *domain model*, effectively coding that character's ontology (view of the world). The different characters' domain models are identical as far as the organization of classes of objects are concerned, but may differ as concerns knowledge about object instances. So, for example, it is conceivable that Cloddy Hans but not Karen is aware of the existence of a particular object, say a hammer. But both the domain model of Cloddy Hans and that of Karen contains a representation of the class to which the hammer belongs (the class Thing, in this case). Figure 10 shows a subset of Cloddy Hans's domain model.

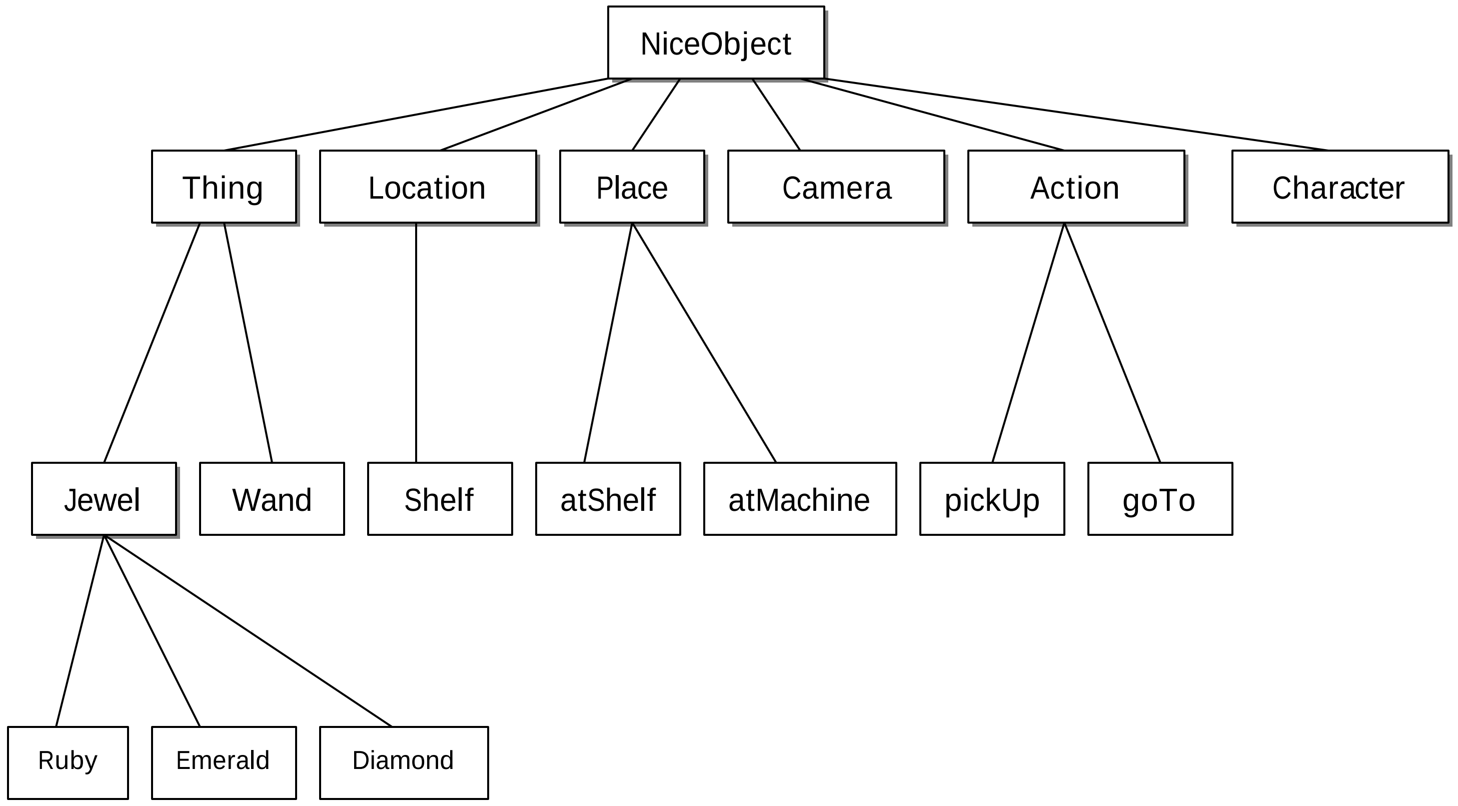


Figure 10. Some classes (shaded boxes) and objects (non-shaded boxes) in Cloddy Hans's domain model, and their subclass/superclass relations.

Classes are organized in a subclass relation. This means that if Jewel is a subclass of Thing, then everything that is true of a Thing is also true of a Jewel. In particular, subclasses inherit all the *attributes* of their respective superclasses (see below). Multiple inheritance is not allowed; a given class can be a direct subclass of one class only.

Relations between objects (apart from the subclass relation) are coded by means of attributes and values. The following table shows the classes of some objects appearing in the first scene, together with some of their attributes and the classes of those attributes (in the cases where the classes of attributes belong to the domain model themselves).

Class	**Extends**	**Attribute**	**Class of attribute**
NiceObject		name	
Thing	NiceObject	position	Location
Jewel	Thing	color	
Camera	NiceObject		
Location	NiceObject	nextTo	Place
Place	NiceObject	preferredCamera	Camera
Slot	Location	contents	set of Thing
Machine	NiceObject	usefulSlot	Slot
		magicSlot	Slot
		valuableSlot	Slot
		dangerousSlot	Slot
Character	NiceObject	carrying	Thing
		age	
		profession	
		health	
		home	
NonMovable	NiceObject		
Action	NiceObject		

Figure 11. Classes and attributes in Cloddy Hans's domain model.

Assume that the hammer (of type Thing) is placed on the shelf. Then we model that by letting the value of the position attribute in the hammer object take the value shelf (which is of type Location). Here we make a distinction between Locations, where objects can be put, and Places, where characters can stand. Assume further that we wish to model the fact that in order to pick up an object from the shelf, a character must stand next to it. Then in the object shelf (of type Location), the value of the attribute nextTo is set to atShelf (of type Place).

Class	**Object**
Thing	axe, book, box, hammer, key, knife, lamp, poison, sack, sword, wand, drawbridge, house, tree
Gem	diamond, emerald, ruby
Camera	shelfCamera, origoCamera, machineCamera, … (*several more cameras of different types, see D4.2*)
Location	shelf
Place	atOrigo, atMachine, atShelf
Slot	dangerousSlot, magicSlot, usefulSlot, valuableSlot
Machine	machine
Character	cloddy, user, karin
NonMovable	readingChair, candle, featherPen, pictureHCAMother, pictureJennyLind, pictureLittleMermaid, pictureUglyDuckling, table, window, writingDesk, boots, coat, door, hat
Action	goTo, pickUp, putDown, lowerBridge, raiseBridge

Figure 12. Objects appearing in the first scene

Object	**Type**	**Initital values for attributes**

atShelf	Place	name=atShelf
		preferredCamera=shelfCamera

shelf	Location	name=shelf
		nextTo=atShelf

axe	Thing	name=axe
		position=shelf

diamond	Jewel	name=diamond
		color=white
		position=shelf

Fairytale Machine	Machine	name=machine

Figure 13. Initial values for some attributes.

4 A rule language for specifying goals and actions

4.1 Expressions and propositions

Expressions come in two flavours: *evaluable* or not evaluable. Examples of evaluable expressions are arithmetic expressions like 1+1, which evaluates to 2, or “dot” expressions like axe.position, which evaluates to the current value of the attribute position of the object axe. Examples of expressions that are not evaluable are the terms that encode dialogue acts, like request(user, pickup(cloddy,axe)).

For the purpose of evaluating expressions, we stipulate the existence of a function eval, mapping expressions to objects, which is such that eval(a)=a, whenever a is not evaluable. For the dot notation, we define

eval(a.b) = c

if the object a has an attribute b whose current value is c. The dot associates to the left, so that

eval(a.b.c) = eval(eval(a.b).c)

(This evaluation order is standard in object-oriented formalisms and programming langagues.) For instance, assuming that objects of the type location have an attribute nextTo, whose value is a set containing all neighbouring positions. Then the value of the expression

axe.position.nextTo

is retrieved by first retrieving the object o which is the value of axe.position, and then retrieving o.nextTo.

If a and b are expressions, then a=b and a!=b are *propositions*. By “evaluation” of a given proposition, we mean the mechanical process of determining whether, or under which circumstances, the proposition is true. To this end, we extend the function eval as follows:

eval(a=b) = unify(eval(a), eval(b)) otherwise

where unify(s,t) returns a substitution for the variables in s and t that make the expressions syntactically equal . For instance, if s is the expression request(user, pickup(cloddy, axe)), and t is the expression request(user, pickup(cloddy, x)), where x is a variable, then unify(s,t) = [x:=axe]. Applying this substitution to both s and t (i.e. binding all occurrences of the variable x to axe) results in syntactically identical expressions. Given a substitution σ and an expression s, we denote the application of σ to s by σ(s).

If two identical expressions are unified, this results in the empty substitution, which binds no variables. The empty substitution is denoted by true. If there is no substitution to make s and t equal, unify(s,t) returns false. Applying the empty substitution to an expression yields the same expression, i.e. true(s)=s, for all s.

As an example, assume that the attribute latestUserUtterance of the character cloddy has the value request(user, pickup(cloddy, axe)). Then

```
eval( cloddy.latestUserUtterance = request(user, cloddy, pickup(cloddy, x)) ) =
unify(eval(cloddy.latestUserUtterance), eval(request(user, cloddy, pickup(cloddy, x))) =
unify(request(user, cloddy, pickup(cloddy, axe)), request(user, cloddy, pickup(cloddy, x))) =
[x := axe]
```

We further extend eval as follows:

eval(a!=b) = true	if unify(a,b)=false
eval(a!=b) = false	otherwise

There are special propositions pertaining to collections of objects. For example, if a is a set and b is not a collection, then a.member(b) and a.isEmpty() are propositions. We extend eval as follows:

eval(a.contains(b)) = true	if a is a set and b is a member of this set
eval(a. contains(b)) = false	otherwise
eval(a.isEmpty()) = true	if a is a set which has no members
eval(a.isEmpty()) = false	otherwise

In general, we will consider any term $p(a_1^{t1}, a_2^{t2}, \ldots, a_n^{tn})$ a well-formed proposition as long as there is a procedure $proc_p$, taking n arguments of types $t_1, t_2, \ldots, t_n$ respectively, such that $proc_p(a_1, \ldots, a_n)$ returns true in the cases where $p(a_1, \ldots, a_n)$ is true, and returns false in the cases where $p(a_1, \ldots, a_n)$ is false. The NICE system is easily extendible to include new such procedures. Thus:

eval($p(a_1, \ldots, a_n)$) = true	if $proc_p(a_1, \ldots, a_n)$ returns true
eval($p(a_1, \ldots, a_n)$) = false	otherwise

The logical connectives are dealt with in the following way (recall that true is considered equivalent to the empty substitution):

eval(P && Q) = eval(σ(Q))	if eval(P) = σ
eval(P && Q) = false	if eval(P) = false
eval(P || Q) = σ	if eval(P) = σ
eval(P || Q) = eval(Q)	if eval(P) = false
eval(!P) = true	if eval(P) = false
eval(!P) = false	otherwise

Here we assume that P and Q are propositions. We have preferred the standard C/C++/Java notation to denote the logical connectives, rather than the "$\wedge$", "$\vee$" and "$\neg$" which are common in logic textbooks.

4.2 Goal satisfaction rules

The purpose of goal selection rules is to provide recipes for how to go about satisfying goals; how goals are broken down into subgoals, and how subgoals eventually are satisfied by a sequence of operations.

A *goal satisfaction rule* is an expression defined by the following BNF grammar:

```
<goal satisfaction rule> ::=   <satify-expression> <operation>
<satisfy-expression>     ::=   satisfy(<proposition>)
<operation>              ::=   convey(<dialogue act>);  |
                               perform(<action>);  |
                               <satisfy-expression>;  |
                               (<operations-list)  |
                               {<operations-list>}
<operations-list>        ::=   <operation>  |
                               <operation> <operations-list>
```

where <proposition> is defined as in Section 5.1, and where <dialogue act> and <action> are terms of type dialogue_act and action, respectively, as defined in deliverable 3.5, Section 3. An example of a goal satisfaction rules is:

```
satisfy(scenario_1_ready)  {satisfy(in_machine(axe)); satisfy(in_machine(wand))}
```

This rule can be read: "In order to satisfy the goal scenario_1_ready, satisfy the two subgoals in_machine(axe) and in_machine(wand), in any order." Thus, the first satisfy expression, called the *head* of the rule, tells what goal is being addressed by the rule, and the ensuing list, called the *body* of the rule, specifies what should be done in order to satisfy the goal of the rule head. The curly brackets "{" and "}" mean "in any order", whereas parentheses "(" and ")" mean "in this order". Thus by replacing the brackets in the rule, we obtain a rule with a different interpretation:

```
satisfy(scenario_1_ready)  (satisfy(in_machine(axe)); satisfy(in_machine(wand)))
```

This new rule can be read: "In order to satisfy the goal scenario_1_ready, first satisfy the goal in_machine(axe), then satisfy the goal in_machine(wand).

A rule definition can contain variables, for example:

```
satisfy( holding( x^character, y^thing))
    (
        satisfy( available( y ));
        satisfy( freeHands( x ));
        satisfy( standingAt( x, y.position.nextTo ));
        perform pickUp( x, y );
    )
```

That is, in order to end up in a state where character `x` is holding thing `y`, `x` must be available (i.e. not already put in the fairy-tale machine), `y` must not already hold anything (since characters can only carry one thing at a time), and `x` must stand at a place next to `y` 's location `x`. If all these subgoals are satisfied, `x` is in a position to pick up `y`. This rule can then be instantiated in many ways (one way for each `Character` and each `Thing` in the domain).

5 Dialogue and personality scripting

As described in Section 2, the NICE fairy-tale game is divided into different scenes. These scenes may be divided into phases, the phases further divided into subphases, and so on, to an arbitrary level of nesting. From a dramatic point of view, a (sub)phase can be thought of as a plot element, and the transition from one phase to another marks the passing of some significant event (for instance, the introduction of a new character, or the change of locale). From a gaming point of view, a scene corresponds to a *level*, each new level introducing a new environment and a new set of problems. In any case, there is a need for a method of defining scenes and phases in a modular way, so that new scenes and phases can be added to the system without the need to modify the dialogue management kernel. Here, "adding a scene/phase" should mean modifying the behaviour, or adding new behaviour, to the characters participating therein. Thus we need to find primitives at an appropriate level of abstraction, in which to express this modified behaviour. Our solution is based on the concept of a *dialogue event*, outlined in Section 5.2.

5.1 Scenes and phases

Internally, the system has an hierarchical representation of scenes and phases. For convenience, we will use the word *scene* to refer to the root nodes of such a hierarchy, while the other nodes are referred to as *phases*, no matter of its place in the hierarchy.

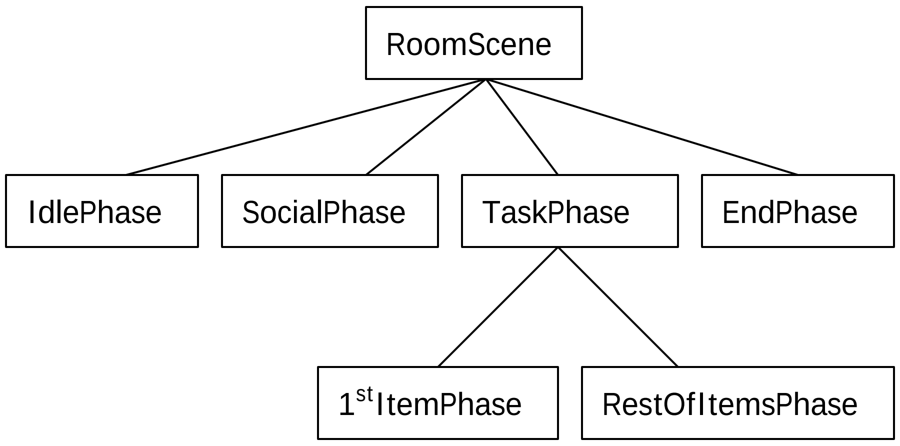


Figure 14. The phases of the first scene.

The first scene is divided into four phases:

- *Idle phase.* This is the scene played when the game is started. Cloddy Hans is walking about in the room doing this and that, seemingly unaware of the user.
- *Social phase.* As soon as the user starts talking (or gesturing at the screen), the game enters the social phase. In this scene, it is the goal of Cloddy Hans to give the user some information about himself (his name, age, profession, residence, etc.), and get the user to talk a little bit about himself. Cloddy Hans also introduces the fairy-tale machine and the overall task of putting objects in it.
- *Task phase.* As soon as the user starts talking about the task, the game enters the task phase. This in turn consists of two phases: Before the game has reached the point where Cloddy Hans has put the first item in the machine, the game is in the *1stItemPhase*. After

that, when the user presumably has got the idea of how to talk to Cloddy Hans, the game enters the *RestOfItemsPhase*. The main difference between these two scenes is that Cloddy Hans will use a more cautious dialogue strategy in the former scene (i.e. asking more confirmation questions).

- *EndPhase*. The scene in the study ends when Cloddy Hans has put enough items in the machine, or if enough time has elapsed since the game started.

As hinted above, at any moment there is exactly one *current phase* in the game, and this current phase changes occasionally according to some algorithm. In the scene above, the phases were arranged in a predefined sequence, but this needs not be the case. For instance, the order in which the phases of the second scene (of section 2.3) are played also depends on the geographic position of Cloddy Hans.

Internally, a scene or phase is defined by a Java object, containing scripting code defining the particulars of the scene. The internal representation of a scene or phase contains the following components:

- an *intro*: a procedure which is run when the scene starts
- an *outro*: a procedure which is run just before the scene ends
- a procedure *changePhase* implementing an algorithm for changing the current phase. If there are no more phases to play, the scene is considered to be finished.

Furthermore, the scene/phase contains code to catch *dialogue events* (see below).

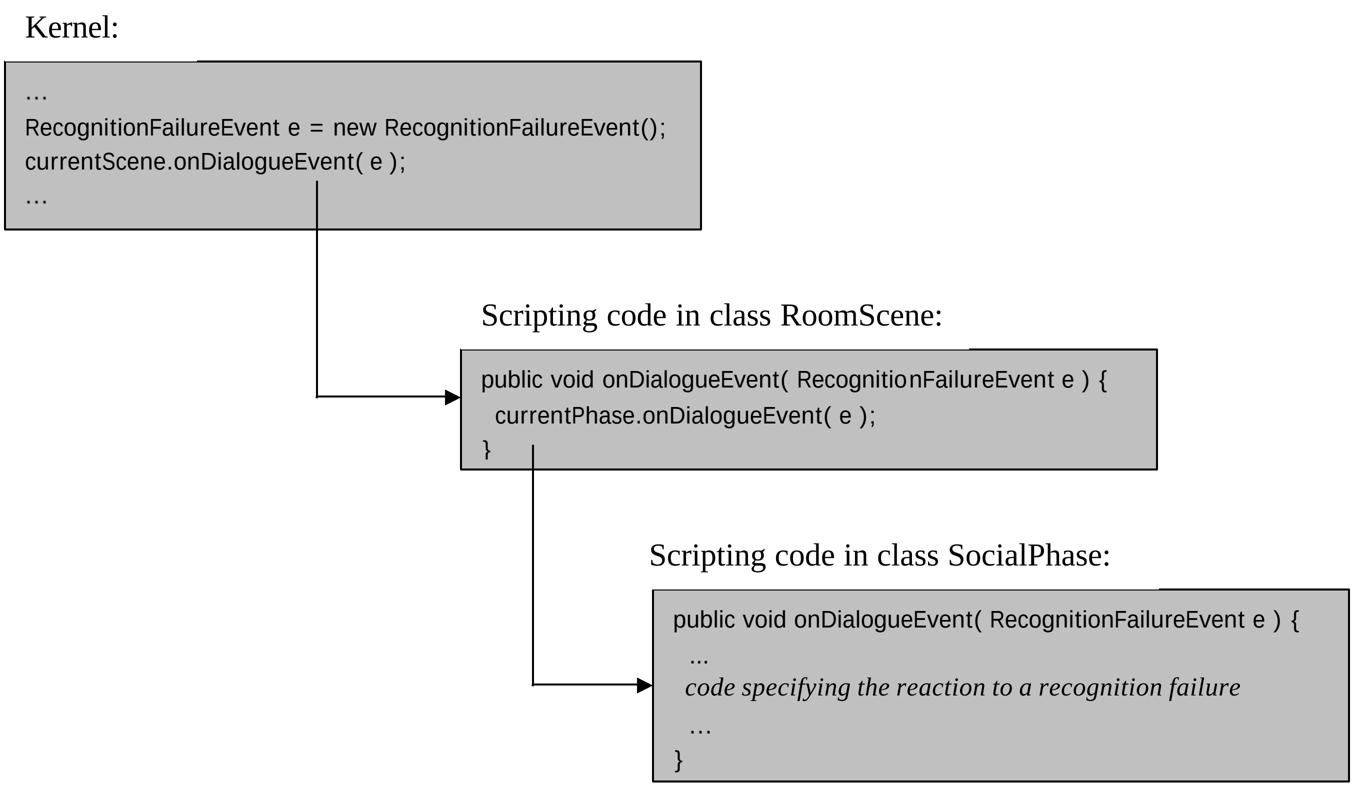
5.2 Dialogue events

The dialogue management kernel issues *dialogue events* at important points in the processing. Some kinds of dialogue events, the so-called *external events*, are triggered from an event in a module outside the dialogue manager (for instance, a recognition failure in the speech recognizer), whereas for

others, the *internal events,* an internal event takes place within the dialogue kernel. Dialogue events can be caught by the scripting code by writing a callback procedure, e.g.

```
public void onDialogueEvent( RecognitionFailureEvent e ) {
        …
        code specifying the reaction to a recognition failure
        …
}
```

As an example, if the speech recognizer detects that the user is speaking but cannot recognize any words, it sends a "recognition failure" message to the dialogue manager. The dialogue management kernel receives this message, generates a RecognitionFailureEvent, and calls the onDialogueEvent procedure of the current scene. The current scene may then redirect the procedure call to its current phase. Figure 15 shows this example pictorally (we assume here that the current scene is the RoomScene, and that the RoomScene's current phase is SocialPhase).


Kernel:
Scripting code in class RoomScene:
Scripting code in class SocialPhase:

```
…
RecognitionFailureEvent e = new RecognitionFailureEvent();
currentScene.onDialogueEvent( e );
…
```

```
public void onDialogueEvent( RecognitionFailureEvent e ) {
  currentPhase.onDialogueEvent( e );
}
```

```
public void onDialogueEvent( RecognitionFailureEvent e ) {
  ...
  code specifying the reaction to a recognition failure
  …
}
```

Figure 15. Catching dialogue events.

In this way, different pieces of scripting code can be provided for different characters, scenes and phases, facilitating the creation of different personalities and scene-dependent behaviour.

The tables in Figure 16 and 17 show all the classes of dialogue events used in the second prototype. Some events contain a message with information pertaining to the event.

Name	**Explanation**	**Message**
AlreadySatisfiedEvent	A goal which already is satisfied has been added to the character's agenda.	The already satisfied goal
CannotSolveEvent	An unsolvable goal has been added to the character's agenda.	The unsolvable goal
ConfirmedEvent	The user has confirmed a proposition.	The confirmed proposition
DisconfirmationEvent	The user has disconfirmed a proposition.	The disconfirmed proposition
ExplanationEvent	The user has asked for an explanation.	The proposition to be explained
IntentionEvent	The character has an intention to say or do something.	The utterance or action the character is intending
NoReactionEvent	The character has nothing on the agenda.	
PossibleGoalConflictEvent	A goal is added to the agenda, but the agenda contains a possibly conflicting goal.	The possibly conflicting goal
RequestEvent	The user has requested the character to do something.	The request
TimeOutEvent	A timeout has expired	The ID of the timeout
QuestionEvent	The user has asked the character a question.	The question

Figure 16. Internal dialogue events

Name	Explanation	Message
GestureEvent	The Gesture Interpreter has recognized a gesture, and found one or several objects gestured at.	The object(s) gestured at.
NluInputEvent	The NLU has arrived at an analysis of the latest utterance.	The semantic representation of the utterance.
PerformedEvent	The animation system has completed an operation, either an utterance or an action such as goTo, pickUp etc.	The ID of the operation.
RecognitionFailureEvent	The speech recognizer has detected that the user has said something, but failed to recognize it.	
SlotEvent	An object has been inserted into one of the slots of the fairytale machine	The inserted object
UnparsableEvent	The speech recognizer has recognized an utterance, but the NLU failed to deliver an analysis.	
TriggerEvent	The animation system has detected that the character has moved into a trigger (see deliverable 3.6).	The ID of the trigger.

Figure 17. External dialogue events

5.3 Kernel interface

The kernel provides a number of operations through which the scripting code can influence the dialogue behaviour of the character. These are:

- interpret the user's latest utterance in its context (see deliverable D5.2b).
- convey (a dialogue act).
- perform (an action).
- add a goal (to the character's agenda).
- remove a goal (from the character's agenda).
- find the next goal on the agenda, and pursue it.

The contextual interpretation process and the agenda is described in detail in deliverable D5.2b. The `convey` operation ultimately leads to an utterance with accompanying gestures from the character (via text generation, graphics generation, and speech synthesis; see deliverable D3.7). The `perform` operation ultimately leads to an action being performed by the character.

5.4 The event-driven dialogue model and its implications

The interplay between the instructions in the scripting code and the dialogue events generated by the dialogue management kernel creates the overall dialogue behaviour of the character. For instance, consider the case where the user requests something from Cloddy Hans in the middle of a conversation; "Go to the fairy-tale machine". It would lead to the following sequence of events:

1. A message from the NLU module arrives and generates an NluInputEvent.
2. The NluInputEvent is caught by the scripting code of the current scene, which calls the contextual interpretation procedure of the dialogue kernel.
3. Contextual interpretation establishes that the user's utterance is a request from the user that Cloddy Hans should go to a specific spot (the fairy-tale machine, in this case). A RequestEvent is generated.
4. The RequestEvent is caught by the scripting code, which calls convey to produce an utterance acknowledging the request, and then adds to Cloddy Hans's agenda the goal that he should be standing next to the fairy-tale machine.
5. When eventually Cloddy Hans has reached his destination, a message arrives from the animation system. This message generates a PerformedEvent, which can again be caught to produce a new utterance from Cloddy Hans, etc.

This event-driven model allows for asynchronous dialogue behaviour (see e.g. Boye et al 2000). That is, unlike many existing dialogue systems, a character in the fairy-tale system is not confined to a model where the user and character have to speak in alternation. Rather, a character may take the turn and speak for a number of reasons: because the user has said something (signalled by an NluInputEvent), because of an event in the fairy-tale world (PerformedEvent, TriggerEvent or SlotEvent), or because a certain amount of time has elapsed (TimeOutEvent). Such events arrive asynchronously; hence they give rise to a more flexible dialogue model. For instance, in the example above, more input from the user may arrive when Cloddy Hans is walking over to the fairy-tale machine. Using the event-based model outlined above, that is no problem; a new line of dialogue can be opened and the user's new utterance can be answered. Eventually the PerformedEvent in (5) above will arrive, and Cloddy Hans can then be made to switch back to the original line of dialogue.

The organization of the dialogue manager into kernel and scripting code, and the hierarchical organization of the scripting code into scenes and phases, also allow for situation-dependent dialogue behaviour. For instance, in the idle phase described above, all events that signal the presence of a user (NluInputEvent, RecognitionFailureEvent, GestureEvent, etc.) lead to the same reaction from Cloddy Hans: he looks up, approaches the camera, and greets the user. By contrast, in the task phase, the three events are handled quite differently.

Also personality-dependent behaviour is made possible by the interplay between dialogue events and the scripting code that catches them. Cloddy Hans, who is supposed to be a helpful character, is scripted to answer a question whenever the user asks it, and to try to fulfill any request from the user. Karen, the girl in the second scene, is scripted to try to satisfy her own goals in the first place. This automatically creates a more selfish and less accomodating personality.

6 References

Boye, J., Hockey, B.A., and Rayner M. (2000) Asynchronous dialogue management: Two case studies. *Proc. Götalog, 4th Workshop on Formal Semantics and Pragmatics of Dialogue*.

Varanese, A. and LaMothe, A. (2003) *Game scripting mastery*. Premier Press.