Toward life-like agents: integrating tasks, verbal communication and behavioural engines

Angela Caicedo ***Jean-Sébastien Monzani***

Daniel Thalmann

Computer Graphics Laboratory

EPFL-Lausanne

{caicedo,jmonzani,thalmann}@lig.di.epfl.ch

Abstract

We present a platform for animating communicative autonomous virtual humans. This is indeed a great technical challenge that involves the low-level animation capabilities (using the notion of *tasks* to handle concurrent gestures) to the high-level behaviour simulation as computed by our Intelligent Virtual Agent. In order to exchange information, a verbal inter-agents communication is also possible. Motivations are triggered by a “Beliefs, Desires and Intentions” architecture, and these notions do not only apply to the virtual world, but also to other agents with a simulation of trust. Finally, since the 3D animation and behaviour modules are separated, we also describe in details how their integration is managed.

Keywords: Conversational characters, tasks, behavioral architecture, approximated sound propagation, human-like behavioral agents, distributed and multi-languages development.

1. Introduction

During the last years, the entertainment industry has produced a lot of exciting movies, games or TV shows involving realistic virtual humans. However, most of the work is hardly designed by artists and these impressing animations still require huge efforts. Furthermore, since movies are now integrating more and more virtual humans, there is a need for authoring tools specifically dedicated to *autonomous* agents' animation. This has been clearly demonstrated by the famous Improv system [17] or similar commercial tools, such as Motion Factory’s Motivate [15] or Virtools’ NeMo [19]. Efforts are continuously spent in order to obtain more and more realism:

the use of speech, better animation, and improved autonomy contribute to go toward life-like characters. Target applications do not only include the entertainment industry, but any inhabited virtual world might benefit from this kind of work. For example, we are now working on a simulator into which policemen have to deal with panic situations, with virtual humans running all around: this kind of training into a virtual environment is a good test for realistic autonomous agents.

Unfortunately, the animation of a virtual human is not an easy process: it actually involves various topics such as: motion control, action selection and verbal communication. Consequently, the *integration* of these domains altogether is a motivating technical challenge. The work presented by Bindiganavale *et al.* [2] is a good illustration of this goal. Our research is focusing on the same topic, that is the animation of autonomous virtual humans that are able to communicate verbally as we do. We are now going to briefly summarise the contributions and previous research for these domains.

From the animator's point of view, it is difficult for one agent to handle concurrent motions at the same time: how can one walk while carrying a box and looking around? If we are able to do this everyday, the simulation of simultaneous gestures and motions is a particular research subject. Models have been proposed to deal with that, such as Granieri's Parallel Transition Networks [10]. For the specific case of gestures involved in virtual humans conversation, Cassel *et al* [8] studied an automatic generation of movements and facial expressions (during conversation), based on the content of the dialog itself.

Regarding realistic verbal communication, we also need some sound propagation models. While Tsingos and Gascuel [22] and more recently, Funkhouser, Min and Carlbom [9] introduced interesting algorithms for fast rendering of sound occlusion and diffraction effects, we think that simpler models simulating sound within a room and taking almost no CPU time have many useful applications in social simulations. A good example would be the simulation of a party, with many people speaking at the same time, and background music disturbing them. Our model is able to simulate such situations, without high computational cost.

Finally, an autonomous agent has to select its actions by itself. Research has been driven by people from different areas: ethologists such as Tinbergen [20], and computer scientists such as Brooks [6], Maes [13] and Minsky [14] who lead the school of Behaviour-Based Artificial Intelligence (BBAI). Our model, as proposed in the BBAI, does not attempt to build models of the world, and the agent has to re-evaluate its course of action on every slot of time. Some points are not directly addressed by the BBAI such as the interplay between internal factors (emotional levels) and external factors (common world situations). Other authors such as Travers [21] have modelled a behavioural system where the agents are described in terms of *if-then* rules. However, we show in this paper that a simple predicate approach is not sufficient for modelling complex human behaviours based on different levels of emotions.

We are now going to present briefly our system and the various components embedded into it. The next section will focus on combining concurrent actions in order to create higher-level tasks. We will continue with a brief overview of our verbal communication model in section 4, and address in section 5 the integration of this module with the *agent's brain*. Finally we describe in section 6 the agent's brain implementation in LISP, before concluding.

2. Agent Common Environment

We have developed a system called: the *Agent Common Environment* (ACE) which animates virtual humans able to perceive their shared environment, perform different motions and have facial expressions. It also provides an easy way to plug-ins different behavioural modules.

ACE understands a set of different commands to be able to control the simulations:

- Creation and location of 3D objects, virtual humans, and smart objects [12].
- Performance of different motion motors and facial expression: playing key-frames animation, using inverse kinematics [1], walking actions, etc.
- Virtual human interactions with smart objects.
- Query of perception pipelines for a given virtual human [4]

All these commands are easily accessible from Python scripts, where different behavioural libraries can be created and plugged into ACE. Those scripts are basically ensuring the low level 3D animation of the virtual humans, while the high level decisions and behaviours are selected by the external **Intelligent Virtual Agent** behavioural module (see section 6). Thanks to the available packages coming with Python, one can manage easily concurrent processes with threads (such as, walking while looking at something), while a TCP/IP connection is maintained between the scripts and the Intelligent Virtual Agent.

We are now going to describe the Agent Common Environment in details.

2.1. Agent design

The agent's specification and implementation is decomposed into two modules: the low-level animation and the high-level decisions taking.

As many 3D environments, ACE is mainly coded in C++ to ensure high performances. For convenient user-interaction, it also provides a **Python layer** that interprets commands on the fly and animates the virtual humans. Python is an all-purposes scripting language that we have extended to fit our needs. More precisely, when the application is launched, a simple environment is created and displayed in a window, and a command shell is prompted, ready for entering commands in Python. ACE provides the basic commands for loading, moving, animating humans and objects, giving a powerful set of functionalities straight from the scripting language. It is very convenient indeed to reuse a language and extend it to match our purposes, rather than developing a new syntax from scratch: this saves time and gives the opportunity to reuse third-party modules, which have been already implemented and tested by others.

On the other hand, the **Intelligent Virtual Agent (IVA)** is in charge of taking decisions such as choosing the next action to take place, deciding what are the new subgoals to be achieved, managing the dynamics of the agent's emotions during the simulation, and so on. Information is stored here in an abstract way, leaving the high to low level binding to the Python layer. For instance, to indicate a specific furniture in an office, we will specify it as *the chair next to the window* rather than x, y and z coordinates: this mapping is handled directly in Python. To conclude, the IVA can be considered as the agent's *brain*.

2.2. Merging capabilities

Running into ACE, the script for each agent should handle various capabilities, such as: perception, verbal communication, performing actions and connecting to the IVA behavioural module. Thus, we split each capability into one class and merged all of them into the definition of what an agent should be able to do. Using UML [3], we present in Figure 1 the definition of the **Agent** class, as implemented in Python.

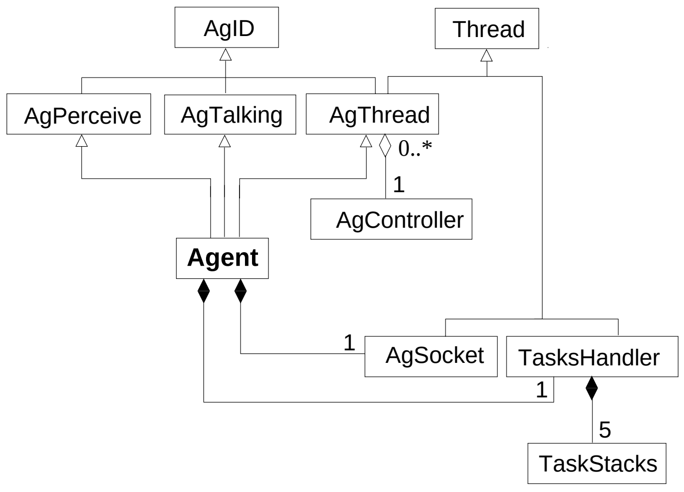


Figure 1. Multiple inheritance architecture defining the agent capabilities

Since each agent has a unique ID, we start by defining the AgID class as a super class, sharing the ID among the inherited classes. From this, we derive three basic classes, for the various capabilities, as pointed out before:

- **AgPerceive**. This class encapsulates all the methods that allow the agent to visually perceive objects and remembers when objects get on/out of focus.
- **AgTalking** lets the agent communicate by speaking to and hearing other agents.
- **AgThread** is the basic class for running one thread per agent, which means that each agent is running its own code in its thread (these functions are provided by the standard **Thread** class). Each thread is registered into an AgController, which is then in charge of monitoring them. It also provides a shared space for exchanging information between threads.

The final **Agent** class inherits from these three basic classes, which of course means that our **Agent** is able to speak to someone, hear when someone speaks and perceive the objects in the environment. But the **Agent** still need to use some other modules:

- **TasksHandler**: This class is in charge of handling parallel tasks like walking, looking, playing keyframes, applying facial expressions or interacting with objects. The next section presents in details what we call tasks.
- **AgSocket**: Each agent should be connected in some way to its IVA behavioural module and this class achieves this. The AgSocket class is able to decode orders coming for the IVA or send stimuli like visual perception back to it. By using sockets and TCP/IP connection, the system can run in a distributed way, reducing the CPU cost on the machine that is responsible of the 3D environment display.

The communication between the **Agent** object and the corresponding IVA is summarised in Figure 2.

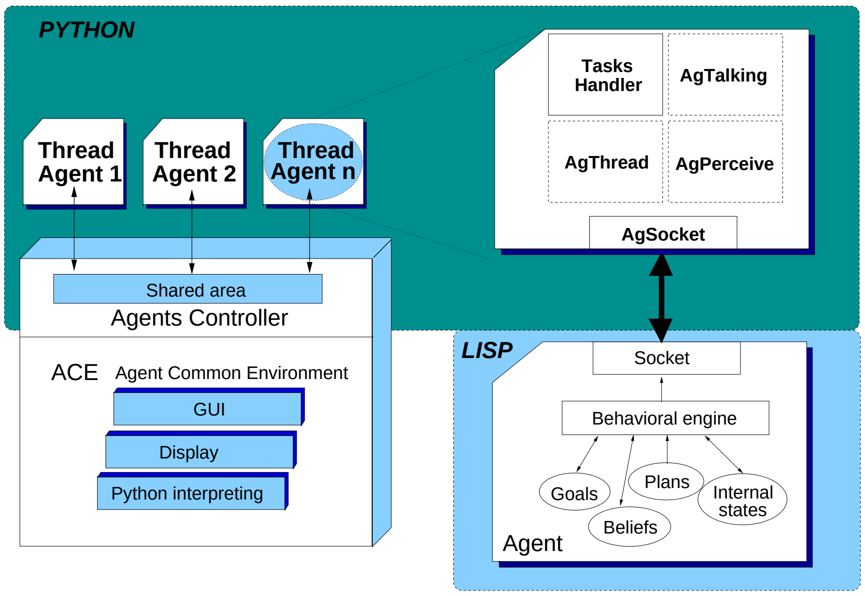


Figure 2. ACE system and connections to the Intelligent Virtual Agent (IVA)

2.3. The use of threads

One major improvement in adding the Python interpreter is the easy way of creating threads within it. Threads all run in parallel and efficient synchronisation primitives are available, such as events. This is a very convenient way to perform actions in parallel. Such threads could easily handle blocking actions such as waiting for data or event (for instance, a task to finish). While it is very tempting to use threads to mimic human capabilities of performing various actions at the same time, one should take care of not creating too many threads (let's say, one per action), since it might take too much CPU time. That is why we are concerned in the next sections by simulating parallel behaviours within non-concurrent instructions too.

Our **Agent** has mainly three threads: the **Agent itself**, the **Tasks Handler**, and the **Agent Socket**. The main task of the **Agent** is to be alert of what he sees, or hears, and to give the appropriate response when one of these events happens. Even if the agent is managing socket connections and parallel tasks, it has not to worry about these matters, because separated threads continuously handle this. **The Tasks Handler** is a thread that is managing the stacked tasks performed or to be performed by the **Agent**. This thread is in charge of choosing the tasks that will be triggered in the next time slot. The **Agent Socket** monitors the activity of the socket, this means, is in charge of reading from the socket the incoming data, and writing the outgoing data or feedback data to the **IVA brain**.

3. Low-level 3D animation using Tasks

Our approach for individual animation of a virtual human relies on a layered architecture: **Actions** provide basic behaviours such as *walk*, *look* and coherently mix them. **Tasks** and **Tasks Stacks** ease the automatic activation and inhibition of actions, all under the responsibility of the **Tasks Handler**. This is discussed in the next sections.

3.1. Actions

At the lowest level, we have a C library of actions that directly controls the posture of the virtual human. Each action is applied to a subset of joints (called the *scope* of this action), and action weights and priorities are used to mix various motions altogether. In-depth details have already been discussed in a previous article [5], therefore we will not go into details but briefly summarise what is already available from our *agent animation library*: actions can be either activated or not, and smooth transitions between these states are computed by adjusting the individual actions weights. By taking into account the various tasks scopes and performing weighted sums, the system is then able to compute the various joints values, and combine the actions.

Actions provided by the library are:

- walk to location
- look at location
- interact with an object
- play a *keyframe sequence* (usually, a motion captured on a real human)
- change the facial expression

While this approach elegantly avoids conflicts and produces smooth animations, it is not sufficient to specify high level behaviours, since every action has to be triggered by hand. For instance, chaining actions like *"do action 1 then do action 2"* requires to check by hand when *action 1* is finished, and then remove it and activate *action 2*. Therefore, making a virtual human follow a path (which we are decomposing into *go to location 1*, then *go to location 2*, etc...) forces one to look if the virtual human has finally arrived at the location before triggering the next action. This becomes quickly complicated when we try to mix various behaviours. To manage in parallel the chaining of actions while performing tests (in order to delete actions that are terminated), we introduce our *Tasks*.

3.2. Combining Actions into Tasks

Tasks are a convenient way to execute actions and monitor their evolution over time. They are implemented as Python classes, and all inherit from the same generic task which contains the following attributes: the **task call-back** is the key element of the task; it generally just calls one action (as presented in the previous section), but more complex behaviours can be easily implemented (for instance, switching from various keyframes depending on the context). The **termination callback** is responsible for testing the end of the task and is called at each frame. This enables the automatic removing of terminated tasks from the **Tasks stacks**, as we will see later on. Of course, together with the own task **id**, you will find a reference to the controlled virtual human (**vh_id**), timing attributes (**time_start** and **duration**), and the **state** of the task (*Suspended* or *Activated*)

There are two more important attributes: the **activation**, which takes one of the values {*Reactivated, Repeated, Once*}, and the **next tasks** to trigger once the task is terminated. This will be discussed in details in next section.

One has to notice that the term **task** is here used as the notion of performing some actions in a specific way: e.g. since the only action that we have for walking corresponds to something like *walk to this location and then stop*, then following a trajectory (that is, going sequentially to various locations) can be seen as a task. Therefore, it is definitively not a synonym of *threads* or *concurrent processes*, as one can think, but rather a general way to consider some virtual human actions.

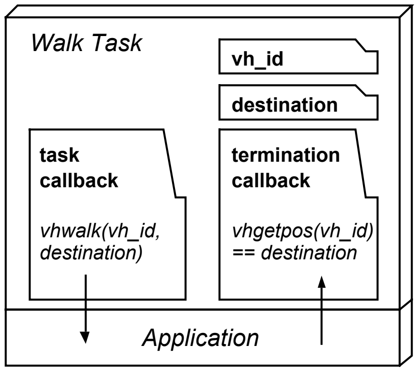


Figure 3. Example callback for a walking task

As an example, Figure 3 shows some of the attributes for a **Walk Task**. This task first stores a reference to the virtual human it is controlling (**vh_id**) and the **destination** point, which is the location that the virtual human should reach. The **task callback** makes the virtual human walk: in order to activate the corresponding action, we are using the vhwalk function provided by ACE. The task is terminated when the virtual human is at the correct location. Once again, **the termination callback** uses vhgetpos to get the position of the agent and regularly checks if it corresponds to the **destination**.

3.3. Managing priorities with Tasks Stacks

Tasks of the same type are organised into stacks, with one stack of each type per agent. Typical stacks that we have in our application are: walking, looking, interacting with objects, playing a keyframe, and manipulating the agent face. Into each stack, only one task can be *executed* at a specific time (the *top task*, see bellow), and tasks on top of the stack have higher priorities than those bellow. At each frame, Tasks Stacks are responsible for updating Tasks, activate them, delete terminated ones, etc... Since Tasks have two states (*Suspended* or *Activated*), only *Activated* tasks are taken into account, as one can expect. *Executing* tasks (that is, the ones which are calling their **task callback** attribute) depends on the type of task to perform: the **activation** attribute of the task is set to *Once* if the task is activated once (playing a keyframe, for example), *Repeated* for continuous tasks which should be performed at each frame (visual tracking of a moving object) and *Reactivated* if the task callback has be executed each time the task becomes active again (typically, walking to a location).

The Task inspection algorithm for each individual Tasks Stack is the following: it starts from the top of the stacks and looks for the first *Activated* task (*Suspended* ones are simply ignored). This task is called the *top task*, that is, the first activated one, starting from the top. Now, depending on the activation of the task:

- if set to *Once* and the task has never been executed, execute it.
- if set to *Repeated*, execute the task.
- if set to *Reactivated* and the top task is not the same than for the previous frame, execute the task.

Once the top task has been found, we do not execute the pending tasks anymore, but we still go through the stack in order to detect tasks which are terminated, by testing their **termination callback**. Terminated tasks are removed from the stack, and eventually activate other suspended tasks stored in their **next task** list. This is a very convenient way to chain the tasks: to make the agent follow a path composed of three locations, we put three Walking Tasks on the stack, chain them by setting the **next tasks** of Walking Task 1 to Walking Task 2, then for Walking Task 2 to Walking Task 3 and only activate the first Task. Once it is terminated, Walking Task 1 is removed from the stack and Walking Task 2 is activated. Same for the next one.

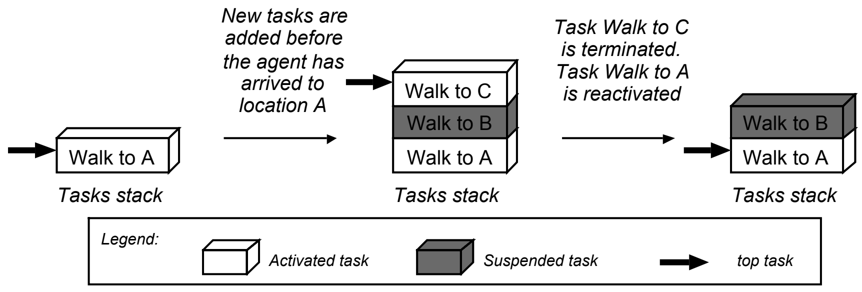


Figure 4. Reactivated tasks - Task stack for the Walking tasks

As an example of *Reactivated* tasks, consider Figure 4: we have represented the stack of Walking Tasks for one agent. At the beginning, there is only one activated task, which asks the agent to go to location A. But before the agent could actually arrive there, two new tasks are appended on the top of the stack: one order to go to location B (which is ignored, since it is *Suspended*) and an order to go to location C, which becomes the *top task* and consequently initiates the lower level action “go to location C”. When location C has been reached, the task is removed, and the Tasks stack reactivates “go to location A” again.

An important thing to note is that if the agent reaches location A while going to location C, then task Walk to A is considered to be terminated and removed from the stack. To prevent this kind of behaviour, one can suspend tasks and use **the next tasks** lists as we have described previously in our example on how to follow a path.

Repeated tasks are illustrated in Figure 5 with the visual tracking of an object. We first track object A (this is the first *Activated* task) and when the tracking of object C is stacked with a higher priority, it becomes the *top task* and prevents the execution of others tasks. If the tracking of object C is removed and the tracking of D queued, then the top task becomes the tracking of A again, and consequently executes this task at each frame.

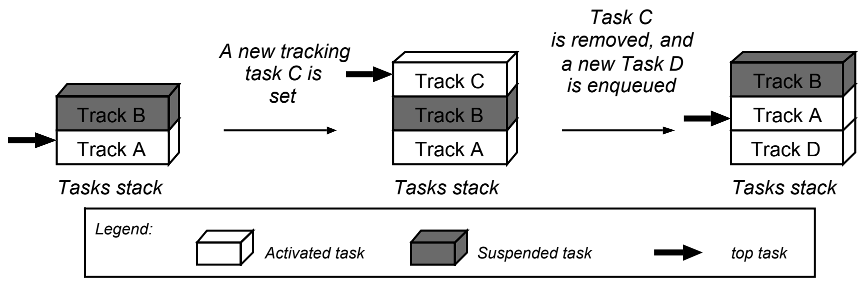


Figure 5. Repeated tasks - Task stack for the Looking tasks

3.4. Multiple tasks altogether: the Tasks Handler

The Tasks Handler gathers all the Tasks stacks for one agent and repetitively activates sequentially each stack, in order to let them execute/purge their tasks. Tasks stacks are launched into threads so that the user only has to append tasks and do not matter to check when they are terminated or not. Since all stacks are regrouped into one object, it is easier to link them, as shown in Figure 6, into which the **next tasks** lists sequentially activates two Walking Tasks and a keyframe. As expected, the generated behaviour drives the agent from location 1, then 2, then 3 and once the agent is arrived, make it applause. In parallel, the visual attention of the agent is focusing on a car.

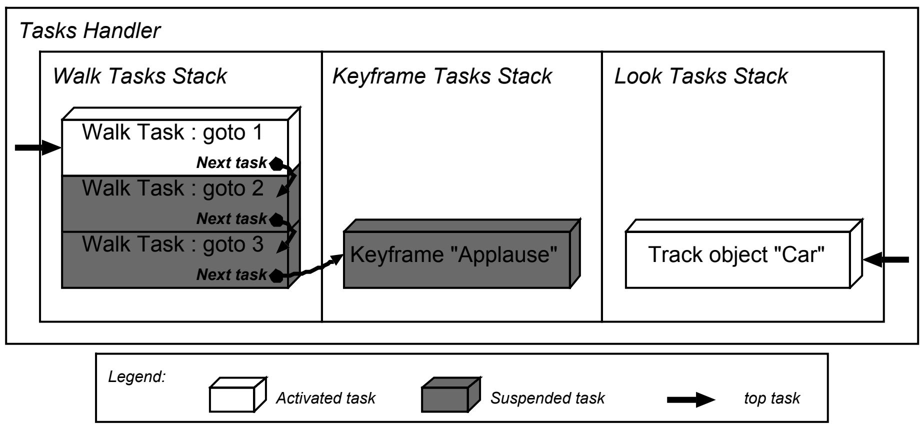


Figure 6. The Tasks Handler

3.5. Life-like behaviours: idle state example

One complex behaviour that we have implemented is the agent "idle state" parameterised by the agent anxiety. Using motion capture, we have recorded various postures of someone waiting and then split keyframes from the upper, the lower body and the hands. This gave us a library of postures for the legs, the spine and the fingers. By randomly switching between them, we manage to produce a realistic feeling for our virtual human. Blinking the eyes is also an important feature, together with changes of expressions on the face, random rotation of the head and breathing. All of these concurrent tasks are simply parameterised by the agent anxiety (ranging from 0 to 1). When the user is changing this value into the graphical user interface, he/she will notice that the amplitude and frequency of motions are updated accordingly.

4. Verbal Communication

We have extended an events based communication model with an approximate model of sound propagation, which is less accurate than real sound simulation but suitable for real-time applications [16]. We are not going to describe in details the various messages exchanged between agents, but to briefly summarise, a sentence is split into the *time to understand* it and the *remaining* time (to complete the sentence). For each utterance, three messages are sent: *is-speaking* first warns the others that someone starts to speak, but the semantic itself is not sent yet, until the time to understand has been reached. The *message-interchange* actually carries the content of the message and when the sentence is terminated, an *end-of-message* is sent to finish the communication.

4.1. Model for the speaker

We define the speech amplitude $Amplitude_i(x,\alpha)$ when agent *i* is speaking to *j* by the product of the radial distribution $D_{radial}(x)$ times the angular distribution $D_{angular}(\alpha)$. α and x are the angle and distance between the listener and the speaker. Both distributions have thresholds: when $x \leq x_{unders\tan d}$ or $x < x_{hear}$, D_{radial} returns respectively 1 or 0, while in-between values are computed with a cubic Hermite interpolation. Similarly, if $\alpha \leq \alpha_{full}$ or $\alpha_{low} \leq \alpha$, $D_{angular}(\alpha)$ returns 1 or A_{behind} (sound amplitude at the back of the head), and in-between input is also interpolated. **Error! Reference source not found.** presents the value of the Amplitude in the 2D plane. As one can expect, the value decreases over the distance (radial distribution) and increases for the “gaze” (mouth) direction (angular distribution).

The *sound quality of the environment* Q is defined as follow: the higher the noise, the lower the quality will be. Thus, each time an actor speaks, the value decreases for a certain amount and increases again once the sentence is over. We have defined that $\alpha_{full}, \alpha_{low}, \alpha_{behind}$ are linear functions of Q. $x_{unders\tan d}, x_{hear}$ depend on the volume of the voice of the speakerand are also affected by Q.

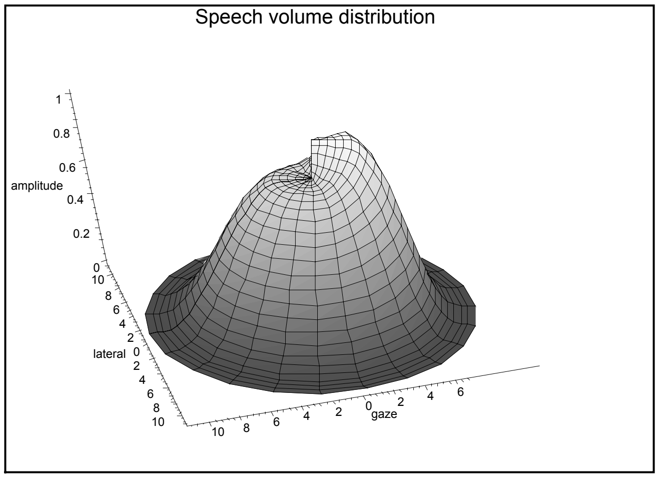


Figure 7. Speech amplitude distribution

4.2. Model for the listener

To evaluate if the listener is able to understand a message, each actor will set two thresholds, $A_{unders\tan d}$ and A_{hear}, between 0 and 1. When evaluating the amplitude *A* of a message, the listener will understand the message if $A \in [A_{unders\tan d}, 1]$, hear but not understand when $A \in [A_{hear}, A_{unders\tan d}]$ and will not perceive anything if $A \in [0, A_{hear}]$.

5. Interconnecting the Animation and Behavioural modules

As we have mentioned before, the agent animation is handled by Python scripts (specifically by the **Agent** class) while the behavioural selection and the decisions making process are handle by the Intelligent Virtual Agent. Both modules are interconnected through sockets. In the python side the **Agent Socket** is in charge of managing the socket and translating high level orders (coming from the IVA) to low level ones understandable by the Python **Agent**, and vice-versa. We can basically distinguish three kinds of communications:

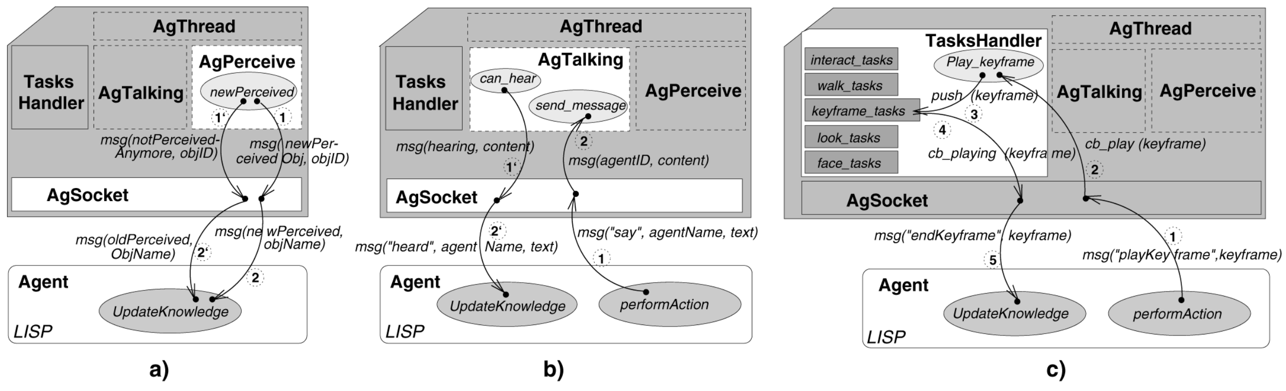


Figure 8. Communication between the Agent Python class and the IVA

5.1. Perceiving *an object or another agent.*

One of the main activities of the Python **Agent** is to watch the surrounding environment: if any new object is perceived, the method *newPerceived* inherited from AgPerceive returns *true*, and a new message is created for the **AgSocket** (see Figure 8a). This message consists of a short description of what happened and the ID of the perceived object. The **AgSocket** receives this message and translates it into a new one understandable by the IVA: the ID of the perceived object is mapped to the corresponding object's name. Similarly, the method *newPerceived* is also used to update the objects that get out of focus.

5.2. Speaking to *and* hearing *another agent*

The Python **Agent** also handles the verbal communication: when someone starts to speak, the method *can-hear* inherited from **AgTalking** returns *true*, and the Agent receives the incoming message. The *is-speaking* and the *end-of-message* messages are ignored, because these ones are just used for synchronisation purposes. The **AgSocket** again is in charge of extracting the relevant information for the IVA, and creating a new message that contains the name of the agent who spoke with the message utterance.

The speaking process is a little bit different. It is the IVA this time who starts the conversation, as presented in Figure 8b. The message consists of the action that will take place (in that case, the action *say*), the agent receiver's name, and the text that the agent wants to say. The **AgSocket** receives this message and generates three new *SpokenMessages*: *is-speaking*, *message-interchange* (which carries the semantic) and *end-of-message* to finish the communication.

5.3. Walking, looking, playing keyframes or applying face actions

All these mentioned tasks have something in common: the Python **Agent** treats them in the same way, specifically by the **Agent's Tasks Handler**. Again, the IVA triggers the need of performing one of these tasks, sending a message to the **AgSocket**. Then the **AgSocket** activates the corresponding **task callback** associated with the task and push it into its **Tasks Stack**. The **Tasks Handler** keeps checking for the termination callback of all the tasks inside the Tasks Handler, and when the

termination callback is triggered, a new message is sent to AgSocket to reflect the changes into the Agent's brain (see Figure 8c).

6. The IVA Brain: Intelligent Virtual Agent

The Intelligent Virtual Agent is based on a BDI architecture (Beliefs, desires and intentions), widely described by Georgeff [18]. This architecture is promising but needs some extensions for achieving our goal: giving to the virtual human the ability to act by itself in a dynamic environment relying on its beliefs, internal states, current state of the surrounded world and assumptions about other agents. It should also allow us to control it in real time [7].

6.1. IVA's components

An IVA has all its knowledge organised into sets, which are distributed according to their functionality (Figure 9): the set of *Beliefs*, the set of *Goals*, the set of *Competing Plans*, the set of *Internal states*, the set of *Beliefs About Others*. Based on all its knowledge, the IVA is able to select the correct action to perform in order to achieve its goal. The Behavioural Engine that will be explained later in this paper does this process.

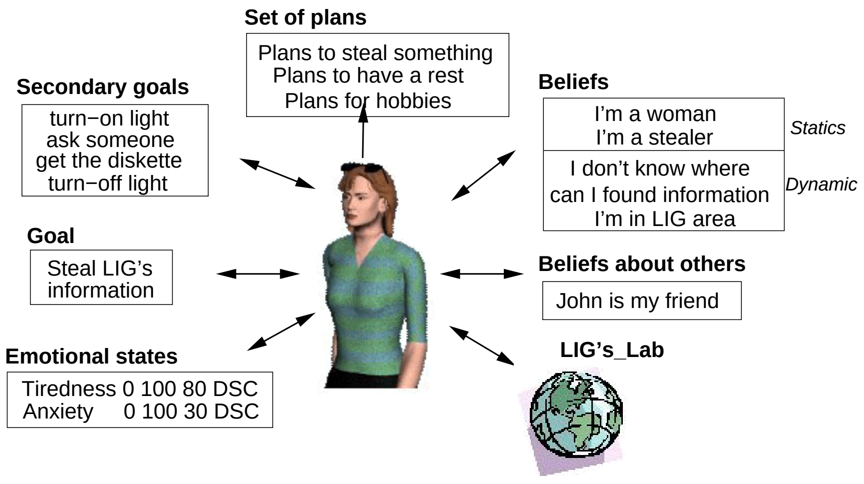


Figure 9. The intelligent virtual agent IVA

- **Beliefs.** Beliefs are a set of statements that the IVA believes to be true. The agent's beliefs are organised in such a way that allows us to simulate *short-term memory* (**Short-term beliefs, STB**), and *everlasting memory* **(Long term beliefs, LTB**).
- **Goals.** IVAs have one main **Goal** and one or several **Subgoals**. The main goal is the objective that the IVA is trying to achieve at a certain moment. During this process, an IVA has to deal with smaller subgoals on which the outcome of the larger one relies on.
- **Competing plans** An IVA uses a set of competing plans that specified a sequence of actions required to reach its main goal (see Figure 10). A competing plan $\mathbf{P_i}$ is described as: $\mathbf{P_i = (is_i, pc_i, ef_i)}$, where:

```
(RememberPlan
   (newPlan 'inspect−place
     '((curiosity 50 >))
     '((is at (? place))
       (! (has been is (? place))))
     '((Act (inspect the (? place)))
       (Add (inspecting the (? place)))
       (Add (has been in (? place)))
      ))
    *P_Walker*)
```

Figure 10. Plan example

- **is$_i$** is a list of internal states to be checked before the plan can be executed. Each of the internal states has an associated valid value or range.
- **pc$_i$** is a list of preconditions that have to be true before the competing plan can be triggered. The preconditions belong either to the agent's beliefs or to the general knowledge stored in the world.
- **ef$_i$** is a list that contains the effects of a plan execution. When a plan is selected, changes at agent or world level will occur (new knowledge will be added and old one will be deleted). These changes are consequences of the plan's effects, as shown in Figure 11.

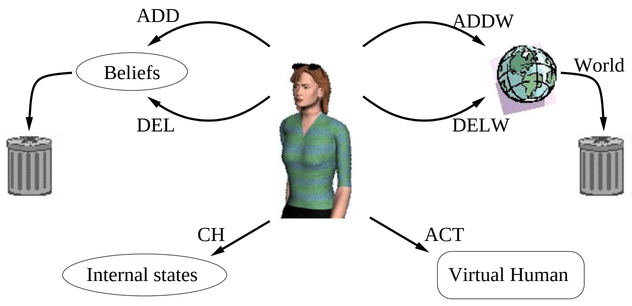


Figure 11. Plans' effects

- **Internal states** The agent stores a set of internal states representing physiological or psychological variables of the virtual human. Internal state act as stimulus for the agent, i.e. *a high hunger level will stimulate the agent to eat*. An internal state is_i is described as a tuple: (n_i, min_i, max_i, c_i, cat_i), where for any given internal state *i*: n_i is its name, min_i is its minimum accepted value, max_i, the maximum accepted value, c_i the current value, and cat_i is its category.

Internal states are constantly being adjusted, as the simulation evolves and plans are adopted. Changes in the internal state are consequences of: the **autonomous growth** or **damping** associated with the internal state and **the side effects** of an active behaviour.

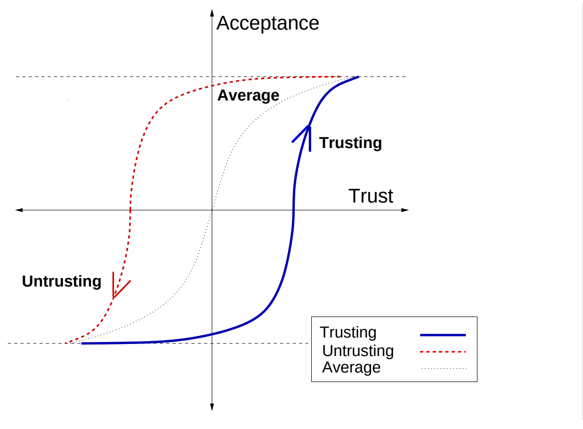
We categorise the internal states as ascendant (the higher the level the better), descendants (the lower the level the better) and not categorised (-), as shown in Figure 12.

Emotion	Category	Emotion	Category
Impatience	DSC	Love	ASC
Enthusiasm	ASC	Curiosity	—
Boredom	DSC	Excitement	—

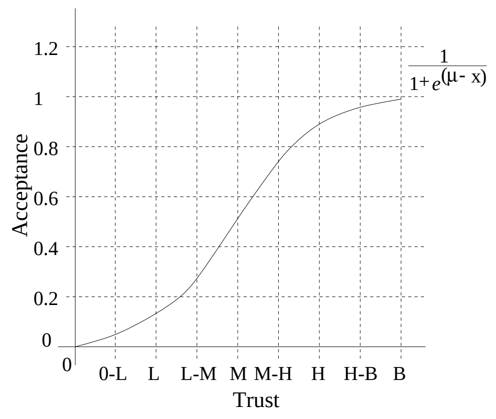
Figure 12. Categorising the internal states

- **Beliefs about others** In our model each IVA is autonomous, and it can accept or reject an order coming from the user or from another agent. Each IVA includes a set of Beliefs about others into which it stores the trust levels associated with them (Figure 9). An IVA sees the user as another agent, and depending on the user's category it will accept an order or not.

  The levels of trust will evolve during the simulation [8], following the Hinde statement: *"Trust, once established in some degree, is often self-reinforcing because individuals have stronger tendencies to confirm their prior beliefs than to disprove them."* [11]. This characteristic is applied to evolve the beliefs about others through the hysteresis curve as shown in Figure 13a.



a



b

Figure 13. Trusting curve.

To be able to show this behaviour we have chosen some categories, from the lowest trusting level to the highest trusting levels: 0-Low, Low, Low-Medium, Medium, Medium-High, High, High-Blind, Blindly.

All IVAS contain the name of the other agents and the level of trust associated to them. The value of acceptance for any order coming from a user with certain trusting level can be seen in Figure 13b: the higher/lower the trust level, the higher/lower the possibility of accepting the order. This means that it could arrived that a user's order is rejected by the agent, showing how societies really work: Once your reliability on someone is corrupted your won't believe him anymore and you won't accept its orders that easy.

For the user to be able to recover the agent's reliability, it should try slowly to obtain a higher level of confidence (climb in the trust curve) or just interact with the agent through another agent who has a high level of confidence[1].

[1] More details of the Trust model are presented in [7]

6.2. The Behavioural Engine (BE)

The behavioural engine is in charge of updating the internal states of the IVA and selecting its next action to perform. It is composed of some controllers as shown in Figure 14.

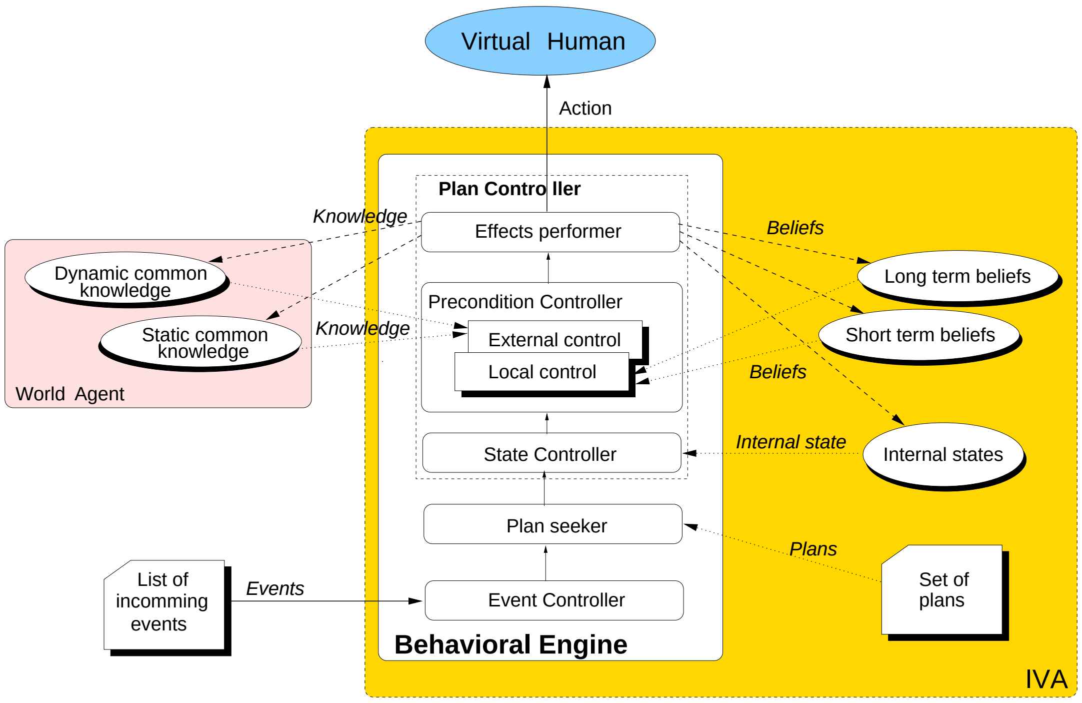


Figure 14. Behavioural engine

First the **Event Controller** checks in the pending events list for those events that trigger in a specific time slot to be integrated in the IVA's knowledge. Then the **Plan Seeker** sequentially passes the plans to the **Plan Controller** who verifies if the plan will be trigger or not. A plan to be triggered needs to have the suitable internal states levels and to full-fill all the preconditions. The **State Controller** checks the internal states levels and if all of them have the appropriate values it will give the control to the **Precondition Controller**, otherwise the **Plan Seeker** will search for the next plan to evaluate. The **Precondition Controller** searches if all the preconditions are full-filled from its local knowledge, or from the external knowledge (*World Agent*). If the **Precondition Controller** agrees with all the preconditions the **Effects Performer** will be called, in order to perform all the necessaries updates inside the IVA or in the *World Agent*, and send the selected action (if there is one) to the *Virtual Human*.

6.3. World Agent

The world agent manages the general information about the environment, such as the names and IDs of all active virtual humans. Each IVA maintains a reference to the world agent so that some information can be exchanged through it. Information is organised in two different groups: **Static Common Knowledge** for the world's information, which is not subject to change, while the **Dynamic Common Knowledge** manages the evolving events.

7. Discussion

In comparison with systems such as Improv, Motivate or NeMo, ACE focuses more on autonomy than animation: while previous applications were targeted to designers and animators, we propose better autonomy by using results from A. I. research in BDI architectures. It is therefore closer to the Smart avatars from Bindiganavale *et al.* [2]. Their PAR architecture is somewhat similar to what we propose with our Tasks. They also integrated natural language interaction, which was off-topic for us. Once again, differences are in the behaviours: the trust model that we adopted is for instance one of our improvements.

8. Conclusion

We have presented how ACE, the Agents' Common Environment can successfully integrate both abstract behavioural decisions with virtual humans 3D animations. At the animation level, we proposed the notion of **Tasks** to handle and synchronise concurrent motions and gestures. For more realism, we have included verbal communication using an approximate propagation of sound. We have shown how a high level **Intelligent Virtual Agent**, independent of graphics specification, is able to intelligently interact with its environment. And to enhance the inter-agents relations, we added a model for Trust. Finally, one can notice how the overall integration can successfully end up with a multi-layered and distributed multi-languages architecture.

9. Reference

[1] P. Baerlocher and R. Boulic. Task priority formulations for the kinematic control of highly redundant articulated structures. In *IEEE IROS' 98*, pages 323-329, 1998.

[2] R. Bindiganavale, W. Schuler, Allbeck J., Badler N., Joshi A., and M. Palmer. Dynamically altering agent behaviours using natural language instructions. In *Autonomous Agents 2000 Proceedings*, 2000.

[3] Grady Booch, Ivar Jacobson, James Rumbaugh, and Jim Rumbaugh. *The Unified Modelling Language User Guide*. Addison-Wesley, 1998.

[4] C. Bordeux R. Boulic and D. Thalmann. An efficient and flexible perception pipeline for autonomous agents. In *Proceedings of Eurographics' 99*, pages 23-30, 1999.

[5] R. Boulic, P. Becheiraz, L. Emering, and D. Thalmann. Integration of motion control techniques for virtual human and avatar real-time animation. *ACM Symposium on Virtual Reality Software and Technology*, September 1997.

[6] R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation RA-2*, 1986.

[7] A. Caicedo and D. Thalmann. Virtual humanoid: Let them be autonomous without losing control. In *The fourth international conference of computer graphics and artificial intelligence*, Limoges, France, May 2000.

[8] Justine Cassell, Catherine Pelachaud, Norman Badler, Mark Steedman, Brett Achorn, Tripp Bechet, Brett Douville, Scott Prevost, and Matthew Stone. Animated conversation: Rule-based generation of facial expression gesture and spoken intonation for multiple. In Andrew Glassner, editor, *Proceedings of SIGGRAPH 94*, pages 413-420. ACM Press, 1994.

[9] Thomas A. Funkhouser, Patrick Min, and Ingrid Carlbom. Real-time acoustic modeling for distributed virtual environments. *Proceedings of SIGGRAPH 99*, pages 365-374, August 1999.

[10] John P. Granieri, Welton Becket, Barry D. Reich, Jonathan Crabtree, and Norman L. Badler. Behavioral control for real-time simulated human agents. *1995 Symposium on Interactive 3D Graphics*, pages 173-180, April 1995.

[11] Robert Hinde and Jo Groebel. Cooperation and prosocial behaviour. Cambridge University Press, 1991.

[12] M. Kallmann and D. Thalmann. A behavioural interface to simulate agent-object interactions in real-time. In IEEE Computer Society Press, editor, *Proceedings of Computer Animation 99*, pages 138-146, 1999.

[13] P. Maes. How to do the right thing. *Connection Science Journal*, 1:291-323, Dec 1989.

[14] M. Minsky. *The society of mind*. Simon and Schuster, 1988.

[15] Karen Moltenbrey. All the right moves. *Computer Graphics Word*, 22, October 1999.

[16] J.-S. Monzani and D. Thalmann. Verbal communication: Using approximate sound propagation. In *Autonomous Agents'2000 Conference Proceedings*, 2000.

[17] Ken Perlin and Athomas Goldberg. Improv: A system for scripting interactive actors in virtual worlds. *Proceedings of SIGGRAPH 96*, pages 205-216, August 1996.

[18] A. S. Rao and M. P. Georgeff. Modelling rational agents within a bdi-architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann, 1991.

[19] Dan Teven. Virtools' nemo. *Game Developer Magazine*, September 1999.

[20] N. Tinbergen. *The study of Instinc*. Oxford University Press, 1951.

[21] M. Travers. *Agar: An animal construction kit*. PhD thesis, The Media Lab, MIT, 1988.

[22] Nicholas Tsingos and Jean-Dominique Gascuel. Sound rendering in dynamic environments with occlusions. *Graphics Interface '97*, pages 9-16, May 1997.