UNIVERSIDADE DE LISBOA

Faculdade de Ciências

Departamento de Informática

IViHumans Platform
The Graphical Processing Layer

Ricardo Lince Amaral Farto e Abreu

Mestrado em Engenharia Informática

2008

Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática

IViHumans Platform
The Graphical Processing Layer

Ricardo Lince Amaral Farto e Abreu

DISSERTAÇÃO

Orientada pela Prof. Dra. Ana Paula Cláudio

Mestrado em Engenharia Informática

2008

Abstract

Virtual environments that are inhabited by agents with a human like embodiment have many practical applications nowadays, in areas such as entertainment, education, psychotherapy, industrial training, or reconstitution of historical environments. These are examples of areas that may benefit from a flexible platform that supports the generation and rendering of animated scenes with intelligent virtual humans.

The IViHumans platform is currently being built with this perspective in mind. The platform is divided in two layers: one for graphical processing and another for artificial intelligence computation. It was projected to provide a set of features which automatically takes care of many issues that are common to applications integrating virtual humans and virtual environments. This document focuses on the conception and development of the Graphical Processing layer, which constitutes the ground for the Artificial Intelligence layer. The connection between the two layers is also addressed. The layers were projected to run in different processes, communicating by means of a simple, yet effective and extensible client/server protocol that we idealized and implemented.

The tasks of the graphical processing layer rely, first of all, on graphical representations. For that matter, we highlight the techniques used in 3D object modeling. We also focus on our design and implementation and on how we applied the principles of object oriented design to confer flexibility to the platform. Reynolds' conception of movement is applied according to our own view, to make virtual humans and other objects steer autonomously in the world, while displaying consistent animations that are automatically chosen according to character specific rules. We expose our solution for facial expressions that can be mixed to transmit complex emotions and that are subject to automatic smooth transitions. We show how virtual objects can be characterized with default and custom properties. We discuss the integration of perception through synthetic vision, including how it is coupled with distinct kinds of automatic memory that recalls any attributes of the objects that inhabit the virtual world.

KEYWORDS:
Virtual Humans, Virtual Environments, Emotional Expression, Synthetic Perception, Steering, Locomotion.

Resumo

Os ambientes virtuais habitados por agentes com uma aparência humanóide têm diversas aplicações práticas nos dias que correm, em áreas como o entertenimento, a educação, a psico-terapia, o treino industrial ou a reconstituição de ambientes históricos. Estes são exemplos de áreas que podem beneficiar de uma plataforma flexível que suporte a geração e produção de cenas animadas com humanos virtuais inteligentes.

A plataforma IViHumans está actualmente a ser construída com esta perspectiva. A plataforma divide-se em duas camadas: uma para o processamento gráfico e outra para a computação de inteligência artificial. A sua concepção pressupõe a inclusão de um conjunto de funcionalidades que cobrem muitos aspectos comuns a aplicações que integram humanos virtuais em ambientes virtuais. Este documento atenta na concepção e no desenvolvimento da camada de processamento gráfico, que constitui a base para a camada de inteligência artificial. A ligação entre as duas camadas é também abordada. As camadas foram projectadas de modo a correr em diferentes processos que comunicam por meio de um protocolo cliente/servidor eficaz e extensível, que idealizámos e implementámos.

As tarefas da camada de processamento gráfico baseiam-se, antes de mais, em representações gráficas. Assim sendo, destacamos as técnicas usadas na modelação de objectos tridimensionais. Também nos focamos no desenho e na implementação da plataforma e explicamos como aplicámos os princípios do desenho orientado a objectos para lhe conferir flexibilidade. A concepção de movimento de Reynolds é aplicada de acordo com a nossa interpretação, para que humanos virtuais e outros objectos possam conduzir-se autonomamente pelo mundo, enquanto reproduzem animações coerentes que são automaticamente escolhidas com base em regras específicas para cada personagem. Expomos também a nossa solução para expressões faciais que podem ser misturadas de modo a transmitir emoções complexas e que são objecto de transições suaves e automáticas. Mostramos ainda como os objectos virtuais podem ser caracterizados com propriedades pré-definidas ou atribuídas pelo utilizador e discutimos a integração da percepção através de visão sintética, incluindo como ela é acoplada a diferentes tipos de memória que recordam quaisquer atributos dos objectos que habitam o mundo virtual.

PALAVRAS-CHAVE:
Humanos Virtuais, Ambientes Virtuais, Expressão Emocional, Percepção Sintética, Condução, Locomoção.

Contents

List of Figures

Chapter 1

Introduction

Virtual environments that are inhabited by agents with a human like embodiment have many practical applications nowadays, in areas such as entertainment, education, psychotherapy, industrial training or reconstitution of historical environments. These are examples of areas that can benefit from a flexible platform that supports the generation and rendering of animated scenes with intelligent virtual humans, specially if it is suitable for applications that have diverse purposes.

The IViHumans (**I**ntelligent **Vi**rtual **Humans**) platform is currently being developed with this perspective in mind. It was projected to integrate a sufficiently wide set of elements, in order to reach the degree of completeness that is indispensable to its effective applicability. Its purpose is to be used as a grounding base for the development of applications that integrate Virtual Humans (VHs) and virtual environments. We try to accomplish that by providing a set of features that automatically takes care of many issues that are common to such applications.

The architecture of the IViHumans platform comprises two separate layers that have distinct purposes: graphical processing and artificial intelligence. The Graphical Processing (GP) layer is not only responsible for rendering scenes that incorporate VHs and other objects, it also manages many logical aspects that are related to the virtual world and its components. The AI layer, on the other hand, implements the intelligent processing that commands virtual humans, through a multi-agent system (MAS). This document addresses the GP layer of the platform, which was the object of study and development in this research work. However, the GP layer is not isolated and some aspects of the AI layer are sometimes brought up.

The project for the conception of the IViHumans platform had began before the author joined it. Prior to the beginning of this research work, much had already been done. The first steps of projecting the platform happened in 2004/2005 [71, 70, 17]. Later on, a tool for mixing facial expressions was created [25, 24], along with the face for a female VH and her basic expressions. A synthetic vision algorithm was also conceived [67] and a proof of concept implementation was accomplished [68].

The actual implementation of both the GP layer and the AI layer, however, had not been done yet.

Building the GP layer was the global task of this work. Some of its aspects were redesigned and it was implemented from scratch, to include features that had not yet been planned for. We followed the general architecture that had already been established. The software that was originally appointed to aid us in our work was used. Namely, OGRE is used as a rendering library and Blender as the 3D modeling software. ODE is the library that will enable the inclusion of rigid body dynamics and JADE is the environment that will support the MAS of the AI layer. We also built on top of *boost* libraries and another tool was added to the content production pipeline: Poser. The former was chosen mainly for networking. The latter was used for the first stage of the creation of our first VH prototype. The content that had already been created for the platform – the face of a female VH – was integrated in this prototype in a second stage, with Blender. Another VH was also derived from an original model of *aXYZ design*(`http://www.axyz-design.com/`). We obtained free motion capture data from the website `www.mocapdata.com`, adapted it, and imbued the latter model with it. The graphical representation of virtual humans is addressed in Section 3.2. Documentation on the creation of the female VH can be found in [5, 4].

We can now use two VH models with the IViHumans platform, but it was developed in such a way that many other models, with different characteristics, can be used. In fact, one of the main goals we always kept in mind when deciding on how to develop the GP layer, was to make it as flexible and extensible as possible, so that it could effectively be used in different contexts. This concern is reflected in the way we conceived and developed the features that the GP layer now includes, which can be grouped in four major topics: movement, expression, perception, and networking.

The ability to move is one of the main base features that must definitely characterize VHs. In the IViHumans platform, VHs' movement is based on Reynolds' behavioral concepts, which we adapted and implemented according to our own view. VHs, and potentially other beings, can have their movement dictated by steering behaviors, displaying consistent animations that are automatically chosen and updated according to custom rules, as explained in Section 3.3.1.

In the future, VHs' cognition will be accomplished by the AI layer, which may include complex models of emotion. If modeled in a way that approximates its impact on real life, emotion will have profound effects on the decisions of the characters and, ultimately, on the overall progress of events within the virtual world. To achieve a higher degree of believability, however, the practical consequences of emotion should be accompanied by the physical expression that it naturally entails. This physical expression of emotion is now supported by the GP layer. Emotion's first reflection was projected for the face of the VHs. We propose an approach that supports default and custom facial expressions that can be mixed together to originate other, more complex, expressions, and that even withstands certain kinds of body expressions. The transition between expressions is fully automatic and it happens smoothly, even when multiple face deformations are involved. Expressions are discussed in Section 3.3.2. Our approach for movement and emotional expression is documented in [6].

Another trait of VHs in our platform is that they can perceive their environment through synthetic vision. For that purpose, the vision algorithm that had already been devised was fully re-implemented and integrated in the platform. VHs can observe the objects of the world and extract their properties, either as an effect of explicit command or with distinct kinds of optional automatic memories that we coupled with the vision sense. This subject is exposed in Section 3.3.3.

Finally, the GP layer was given a server to communicate with the AI layer through a client/server paradigm. A simple yet effective protocol was designed and implemented, along with stub clients that allowed us to test and demonstrate how the services that the GP layer provides can be used by external programs. Networking is the topic of Section 3.3.4.

In Chapter 4 we expose the main conclusions we draw from this research work and we suggest possible future paths for the IViHumans platform. Chapter 3 contains a thorough explanation of the work that was performed during the execution of our research work. In the next Chapter, we present and discuss important work that greatly influenced our progress.

Chapter 2

Related Work

The area of virtual environments inhabited by intelligent virtual humans is extremely vast and has inspired a great amount of research work, especially since around two decades ago. Since the end of the 80's until today, several contributions have been made, directly or indirectly related to the subject. Some of the efforts contribute to general perspectives on how the virtual environments and agents should be architectured and implemented. However, the vast majority follows a trend of specialization, focusing deeply on specific topics that incisively enhance the overall knowledge.

The task of building a platform as the one we have been conceiving draws great benefit from the awareness of problems that were already identified and from the solutions found for them. Generally, an expanded notion of the progress that has already been achieved allows us to benefit from knowledge that helps the design of a, hopefully, useful system.

This chapter briefly describes some of the major works that focus on virtual environments and VHs or on related aspects that we consider relevant to the point of view we adopt. We dedicate special attention to the ones we consider the most influential and we focus on the aspects that are the most relevant to the approach exposed here. However, we also describe some research courses that essentially diverge from ours but that we consider substantial for the full picture and that contributed to the grounding base we build upon.

Even though this work concentrates on the graphical layer, understanding some issues that relate mainly to the field of artificial intelligence is crucial to harmonize the behavior and the interaction of both layers, so that they fit properly. Bearing this fact in mind, we also devote much attention to subjects that are not strictly classified as belonging to the area of computer graphics.

We include some figures in the following discussions that we believe can be an aid in understanding key concepts. All of them were drawn from the respective publications.

2.1 Most Relevant Work

Although the last two decades were prolific in scientific publications that disclose distinct approaches to the problems involved in the subject of virtual environments, in our opinion, the most striking contributions are not always the most recent. Nevertheless, we consider it fair to place them prominently because they were foundational and set the courses that oriented subsequent research. In this section we include the work we distinguish this way.

2.1.1 Reynolds – *Boids*

The work of Craig Reynolds definitely marked the history of computer graphics by introducing the concept of behavioral animation, repeatedly employed ever since. In 1987 he released a *distributed behavioral model* for the simulation of animated flocks, herds, and schools [61], breaking up with traditional approaches of scripting the movement of virtual actors and objects. In the model he built, “boids” decide their routes autonomously and at runtime. The decision of each individual is achieved according to a set of behavioral rules and using simple models of local perception of a dynamic environment.

Using this model for flock animation, Reynolds avoided having to predetermine the path of each individual, a tedious and error-prone task, largely automatizing the generation of the animation. The behavior of each boid is modeled according to the following three principles:

Collision avoidance – each individual tries to avoid collisions with nearby flock mates and objects;

Velocity matching – each individual tries to match velocity with nearby partners;

Flock centering – each individual tries to stay close to its neighbors.

In order to have a consistent behavior, boids act according to a limited awareness of the surrounding environment, which is obtained through what the author calls *simulated perception* – a means of controlling the information that a character can access by filtering irrelevant data and knowledge that should naturally be inaccessible due to any physical limitation (e.g. an imperceptible sound due to the distance from its source, or an invisible object due to some obstruction). This kind of technique is widely used today and it is also often called *synthetic perception*.

In Craig Reynolds’ work, the actors can only access such properties as velocity and position from mates that stand nearer than a given threshold. This way, the volume that a boid can reach with its perception is a sphere centered in itself. However, the author acknowledges that a deformation of this volume in the direction of movement would increase the similarity with what happens in the real world. In what concerns external obstacles, a boid is only sensitive to the objects that are in front of him. It detects them by looking for incidental intersections with its local Z-axis. When an intersection is discovered, the boid diverts its movement by an amount that is calculated on the basis of the distance to the object that was intercepted. The benefits of local perception can be exemplified, in this context, by the ability that a flock demonstrates of separating, when before an obstacle, to eventually merge again. Clearly, this emergent behavior would not come up if flock centering enclosed all the group as one.

Boids are given steering specifications in the form of acceleration suggestions. Each behavior returns acceleration vectors that are somehow combined to derive the net acceleration that the actor is submitted to. Therefore, the evolution of the flock is dictated by the search of a balance situation among all behavioral principles. Global movement exhibits a behavior that emerges from the individual ones and that surpasses their sum: an unit behavior displayed by the group that appears to have a centralized intention source.

In some occasions, disabling conflicts may erupt between individual behaviors. As a way of solving undesired conflicts, a priority schema for behaviors may be followed. In this work, Reynolds embodies behavior priority by imposing a limit for the magnitude that the net force can achieve and by adding the individual forces in the order that is established by their relative priorities. When the threshold is exceeded, the excess is removed and remaining additions are discontinued. Hence, less urgent behaviors are temporarily left unsatisfied.

The author claims that, with correct parameterization of the rules, the results can be quite realistic and prove to be in accordance with zoological observation. Some demonstrations of these results can be found at [13].

2.1.2 Reynolds – *Steering Behaviors*

After little more than a decade later, Craig Reynolds deepened the concept he had applied in the creation of flocks of virtual animals, generalizing it to imbue different characters with realistic and spontaneous navigation capabilities [62]. The new behaviors were varied and had different purposes, in order to extend the autonomy of virtual characters. The rules for the movement of the boids were early examples of what the author came to call *Steering Behaviors*, name by which they are still emblematically known.

To clarify the scope of his behavioral model, the author proposed a conceptual division of the behaviors of a virtual character in three separate layers, ranked by degree of abstraction, as depicted in Figure 2.1.

The *locomotion layer* includes low-level tasks that carry the movement of the

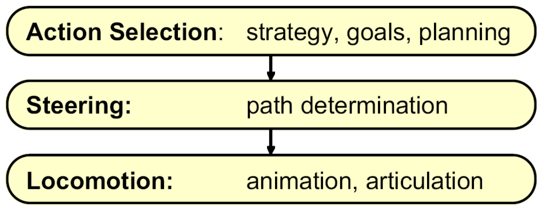

Action Selection: strategy, goals, planning
Steering: path determination
Locomotion: animation, articulation

Figure 2.1: Hierarchy of the conceptual division for the movement of a virtual character.

character; the *steering layer* establishes the way to go to reach an objective; the *action selection* layer defines objectives and sub-objectives. According to this hierarchical layout, the aims of the action selection layer are achieved throw the features that the steering layer provides, and the same relation holds between the steering layer and the locomotion layer. In Reynolds' vision, steering behaviors are a part of the steering layer.

By categorizing three movement control layers, Reynolds views them as independent, anticipating a plugin-like implementation that enables, for instance, the exchange of one locomotion module for another, maintaining the viability of commands from the steering layer. In practice, some adjustments may be necessary to deal with differences in agility and intrinsic characteristics of distinct locomotion models. The author sustains that such adjustments can be made by parameter tweaking, in the same way a driver quickly adapts to a new car, instinctively correcting differences in the mapping between the vehicle's commands and responses.

Steering behaviors stand on the assumption that the underlying layer can be approximated by a simplified concept of vehicle that can equally be applied to planes, cars, horses, or the legs of a human being (with some abuse of nomenclature, as the author points out). The vehicle is modeled as a point mass whose main characteristics are position, mass, velocity, and orientation. The intentions associated with steering behaviors are transmitted to the vehicle as force vectors to be applied on itself. As the force is applied on the vehicle by itself and since, in the real world, it should be limited in spendable energy, the forces' magnitudes must not be arbitrarily large. Thus, a maximum applicable force attribute is also included in the point mass model, as is a maximum speed attribute, for analoguous reasons. In fact, in the real world the forces that a body is subject to are, from a certain speed on, canceled by friction and air resistance and so do not produce an acceleration to further increase speed.

The state of a vehicle is defined by the values of its variable attributes (position, velocity and orientation). It is updated discretely, as a function of the elapsed time since the previous update and it can be changed by external or internal forces. Internal forces – the ones that are produced by steering behaviors – are combined to originate a net force whose intensity is truncated to the maximum allowed value. Applying Newtonian laws for kinetics, an acceleration is derived and used to update velocity which, in its turn, is used to update position.

Taken together with its position, the orientation of the vehicle determines its local space: the position defines the origin of a frame whose axis are defined by orientation. The orientation is partially dependent on velocity, since the depth axis of the vehicle is always aligned with its velocity. Therefore, the vehicle has its own velocity-aligned local coordinate space. Nevertheless, the orientation is not totally established by velocity. Indeed, there is an additional degree of freedom that is left undetermined, namely the one that corresponds to rotation around the depth axis – roll. This additional information can be obtained recursively, for timestep n, from the rotation of the velocity vector since timestep $n-1$, applied to the orientation of timestep $n-1$.

Although it is a simple approximation to reality, this model leaves space for sudden rotations of the object because it does not impose a limit to angular velocity. The author suggests that this problem can be solved, for instance, with a restriction for variation in the orientation of the object of by limiting the side component of the net force.

Assuming that the implementation for locomotion is in accordance with this simple vehicle model, Reynolds described several steering behaviors that operate by geometrically computing the intended forces. Some examples of the behaviors he invented are *seek*, that tries to direct the vehicle to a certain target, *pursuit*, that leads the vehicle into the computed future position of a moving target, and *leader following*, which steers the vehicle so that it follows the actor appointed as leader.

2.1.3 Tu and Terzopoulos – *Artificial Fishes*

Xiaoyuan Tu and Demetri Terzopoulos proposed a framework for automatic animation of artificial fish that mimics the complexity of movement in natural ecosystems [80]. In their approach, several properties of fish and of their interaction with the real world are faithfully approximated by a system that holistically embraces rules for physics, locomotion, perception, and behavior. To believably simulate these animals, they modeled them as autonomous agents that are composed by features that belong to these four layers of abstraction, contrastingly with the approaches that, like Reynold's simple vehicle, aim solely and directly on the final appearance of the animation.

The physics layer encloses the vast dynamics of natural forces, particularly those that are produced in an underwater world. It is over this layer that all the system is built. At this level, the artificial fish is modeled as a sophisticated spring-mass model that imitates its muscular structure and mechanics. The contraction of some sections of the body gives rise to forces that are applied over the water and that originate reaction forces that drive the fish forward. The authors claim that the model keeps structural stability of the body while allowing it to flex.

In the locomotion layer, motor controllers are implemented to give the fish the ability of swimming forward or changing direction. These controllers command both the actions of the muscles and the pose of the pectoral fins, so that the fish can move with precision in the 3D world.

The artificial fish has a vision sensor that, while not imitating the highly developed eye of real fish, provides him information about the environment, such as colors, sizes, and distances. The vision is cyclopean and covers a spherical angle of 300^o along a radius that is parameterized according to the translucence of the water. The artificial fish has also got a temperature sensor that samples the ambient water temperature at the center of its body.

The final animation relies on yet another layer for behavior that amounts to an intention generator that has a simple memory mechanism for recording interrupted actions, so they can be resumed later. The character of a fish may be defined by specifying its tastes and habits (e.g. whether it likes light or whether it normally incorporates schools). These specifications are used by the intention generator, together with the attributes that determine the state of the fish, to define his immediate behavior. Behavior priorities are defined with chained rules. By specifying different priority schemes, Tu et al. created three kinds of fish – predator, prey, and pacifist.

With this framework, underwater animations where rendered with reduced interference by the animator. The authors claim that the framework is easily extensible. The computational cost does not seem too high, given the wide range of features that are implemented. Nonetheless, the number of fish that can be animated with a real-time rendering is forcibly low, when compared with approaches that simplify the modeling of lower-lever layers to pay special attention to higher-level ones.

2.1.4 Funge et al. – *Cognitive Modeling*

In [33], Funge et al. present a cognitive model to direct artificial characters, with the aim of increasing their autonomy and, thus, the automatism of animation creation. With their method, they attempt to outdo behavioral approaches that are directly built over physical or motor layers. While some of the previous works already involve a certain degree of autonomous decision-making and objective picking, they are closely bound to some particular system and may be categorized as essentially pertaining to the behavioral level (see, for instance, how the intention generator deals directly with collision possibilities, in [80], or how, in [57], low-level actions are pointed to actors by the typical script).

The cognitive paradigm that the authors propose transcends the behavioral one mainly because it originates deliberative attitudes in place of reactive attitudes. Funge et al. construct a cognitive layer that is clearly separate from the behavioral layer, albeit seating on it. Cognitive modeling is supported by behavioral modeling and, hence, rises one level above it. Figure 2.2 shows how the compartmentalization of the functionality that is involved in automatic animation can be seen. The figure highlights the authors' contribution.

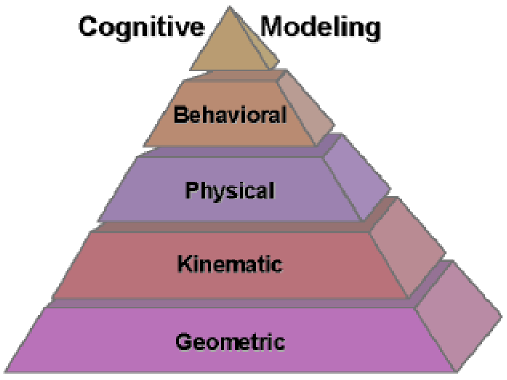


Figure 2.2: Hierarchy of the modeling layers that are involved in automatic animation generation.

The assignment of cognitive capabilities to characters entails the need for granting them with the power of acquiring and using knowledge, namely to plan action sequences that can be mapped into commands for lower layers. The *Cognitive Modeling Language* (CML) is developed as a means of constructing cognitive actors that can elaborate plans to achieve their own goals and whose decisions depend on their knowledge. The cognitive models that CML enables the user to create withstand complex behaviors of interaction with the environment and with other characters. Still, the language does not prevent the user from including as much detail as he wants into an animation, since a section for the direction of the animation should also be defined, leaving the author to directly handle any aspect in the world. Also, the scope of the language is not limited to the use of features from the behavioral layer, but it allows resorting to lower-level services.

In this work, the author's also came up with an alternative formalism for interval arithmetics. This formalism is fully integrated in CML and it supports reasoning with uncertainty about the state of the world. CML also includes perception mechanisms and it allows reasoning over specific domain knowledge as well as general planning algorithms.

2.1.5 Perlin and Goldberg – *Improv*

Perlin and Goldberg built a system, which they name *Improv*, for the creation of animated actors whose behavior is synthesized, to a vast extent, in real rime, following user's intentions [57]. These are defined in scripts whose content ranges from simple actions, which can be specified as postures of the skeleton that is associated with a character, to decision rules and composite actions.

Improv is constituted by two subsystems: an animation engine and a behavior engine. The animation engine controls the body of the actors and it is responsible for the generation of animations. It allows the actors to switch between animations in a smooth and natural fashion. Base animations are combined with coherent noise, which was originally conceived by Perlin for procedural textures, in order to render non-repetitive animations. Noise is stochastically applied as slight displacements of the limbs of a character, so that any two repetitions of the same animation are never equal. Additional believability is thus bestowed on the characters.

The system is robust, dealing effectively with interference between animations. To this end, animations are grouped together and a priority order is established among the groups. Animations that are in the same group compete with each other and, when one of them is selected, the remaining ones are progressively deactivated. If two animations that belong to different groups are played simultaneously, the animation that belongs to the group with higher priority invalidates any interfering movements from the other one.

Chronological dependence between actions is solved with the concept of *action buffering*. Inconsistent action transitions are circumvented this way. Suppose that, at some instant, a virtual actor had his hands behind his back an that the next action was to clasp his hands. If linear interpolation was applied, the hands of the actor would pass through his body. To prevent this, a dependency must be declared that states that, before clasping his hands, the actor has to assume a posture in which his arms are laterally placed along his torso. An auxiliary action is thus inserted in the transition sequence (Figure 2.3).

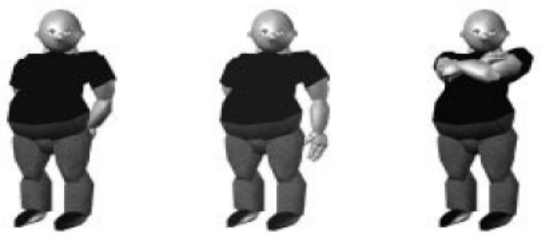


Figure 2.3: Example of action buffering.

The behavior engine allows the user to create sophisticated rules that govern how actors communicate, change, and make decisions. The flow of decision of an actor can depend on properties of the world and of other actors. For instance, it is possible to create an actor that interacts only with actors that he finds interesting, based on their characteristics.

The system is comprehensive and allows the creation of complex interactive stories that may unfold in a non-deterministic way. Stochastic behaviors may be declared at distinct levels of abstraction. For instance, when participating in a game, a character may choose the following play randomly or according to some probability distribution.

Improv is implemented as a set of distributed programs that connect through TCP/IP and through UNIX pipes, with a flexible synchronization but also with guaranteed consistency. If the system is distributed along a LAN, it is possible to have several animation engines, although the behavioral engine must be unique. Hence, albeit executing the same abstract operations that the behavior engine determines, the animation engines produce slightly different results. The authors illustrate the situation explaining that, although the body of the actor exists in *parallel universes*, exhibiting small differences in posture at each instant, his mind is unique and comprises always one single current state that is compatible with what is being rendered by all animation engines.

2.1.6 Vosinakis et al. – *SimHuman*

Vosinakis et al. created a platform for virtual agents with planning capabilities that run in real-time in an arbitrary virtual environment – *SimHuman* [83]. The authors seek to have performance and autonomy well balanced with the believability of the agents, which have a graphical representation in a 3D world.

The visualization module includes features such as *keyframing* animation, collision detection and response, and an alternative form of inverse kinematics. It relies on an engine for rendering and physics that is implemented in C++ and based on OpenGL. This engine applies kinematics' rules to the objects that are present in the environment to determine their following state. Collision detection is performed with a non-exact method [45], relying on automatically calculated bounding cylinders (Figure 2.4). This type of collision detection is considered perfectly adequate for the purpose, considerably reducing the use of computational resources, when compared to exact collision detection.

Virtual agents consist mainly of a high-level planner, which is implemented in Prolog. An agent senses, decides and acts simultaneously. The sensing process is carried out by a ray-casting synthetic vision algorithm – in each call, the agent casts a series of rays from the position of his eyes and accesses only the information of the first objects to be intersected by any ray. The agent records this information

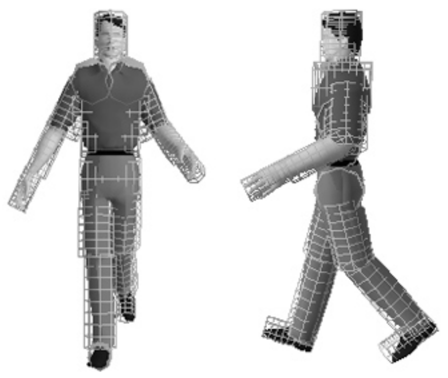


Figure 2.4: Bounding cylinders for VHs in SimHuman

in memory, until the next perception yields new data to substitute it. This data is subject to an abstraction process that returns symbolic information. The information that results from this abstraction is handed over to the planner to constitute the agent's beliefs, so that plans can be generated (e.g. the position of a ball might be translated to a belief that states that the ball is over some table).

The decision process corresponds to planning. Since the world is dynamic, the planning is carried out with monitoring, which is performed by constantly comparing what the agent perceives with its beliefs. Aspects like the type of planner or the scope of its domain are not addressed by the authors. The planning process uses a set of all possible actions the agent is able to perform, which are executed by standard algorithms and whose specification must include preconditions and effects.

At this time, the Prolog planner was directly called from within C++ code. Later on [8], however, the architecture underwent some changes to fit it to the Unreal Engine [75]. The cognitive layer was then implemented in an external controller, but the way this controller was connected with the rest of the framework is not explained.

In [9] the authors report how they changed the design of the system in order in include emotion. To achieve that, they pursued a hybrid approach of cognitive and sub-cognitive theories for categorizing emotion, following someone else's [35] classification of emotion as partly mental and partly physical. The framework was divided in an *execution subsystem* and in a *behavioral subsystem*. The latter is constituted by a physical layer, in which basic sensory and motor functions are included, while the former is divided in two layers: cognitive and non-cognitive. Basic beliefs and emotions reside in the non-cognitive layer. The cognitive layer is responsible for high-level behavior, which is determined by high-level beliefs, affective experiences and deliberative processes.

Finally, in [84], the authors develop SimHuman even further. They explain that characters' behavior is now specified by the programmer through callback functions that can implement intelligent behavior. The callback functions call also map user input into characters' actions, to create avatars.

The walk animation receives special attention, being implemented by a FSM[1] that attends necessary movements for a character to change its path direction. All transitions between states are carefully established so that no inconsistent movement may be generated. The authors also focus on their own Inverse Kinematics algorithm that successively changes skeleton position to approach the target, instead of calculating the final position at once. They argue that this approach is beneficial because it produces more realistic animations than the ones that result from a simple interpolation of initial and final states of the skeleton, and because it copes successfully with moving objects, avoiding obstacles between the character and the target.

2.2 Other relevant work

There is a great amount of published work that is somehow related to the subjects that the IViHumans platform explores. This section summarizes a selection of them, focusing on the characteristics that matter the most to our work.

2.2.1 Noser et al. – *Synthetic Vision*

In [52], a solution for the navigation of digital actors in a 3D world is proposed, based on a single means of perception: synthetic vision. The authors present a Z-buffer based method for synthetic vision. They argue that vision is a sufficient source of information for the agents to create a mental model of the environment that can be searched and used for path planing and obstacle avoidance.

The internal environment map of an agent has a dynamic octree structure. Information is kept about which nodes are occupied by objects. Other relevant data can be also associated with octree nodes, depending on application needs. When an object is detected, the octree is refined so that the corresponding node has an adequate dimension and the node is marked as occupied and timestamped. As time goes by, the deeper nodes of the octree can be deleted, so that the agent *forgets* what he saw.

The map can be used to what the authors distinguish as global navigation tasks (e.g. path planning). For local navigation tasks like obstacle avoidance, no representation of this kind is needed, nor the knowledge of the agent's position. The perception of objects that stand close by is enough for the agent to immediately react if these objects pose a collision threat to him. The reactions are caused only

[1]Finite State Machine

by objects that are in the vision field of the agent and closer to him than a threshold that is proportional to the speed of the agent.

The agent periodically probes the world with vision, being the period called *attention rate*. This rate should be adjusted to balance the quickness of reflexes of the agents with computational weight.

2.2.2 Peters and Sullivan – *Synthetic Vision*

C. Peters and C. O' Sullivan propose a combination of synthetic vision and memory as a means for the VHs to access their environment [16]. The synthetic vision module is based on the one that was described by Noser et al. and uses false-coloring rendering from the point of view of the VHs.

The features of the vision module are extended with the inclusion of multiple *vision modes*. With the *distinct mode*, a unique color is assigned to each object so that it can be identified and looked up in the *scene database*. In the *grouped mode*, single colors are attributed to groups of objects. These groups are selected according to a number of different criteria (e.g. proximity; brightness). In this mode, the sight of one object suffices for its group to be identified. Further querying is then necessary to establish which objects are actually being seen.

Some features of the examined objects are recorded in the memory of the VH that saw them. The detail of extracted data is dependent on the attention that was paid to the corresponding object or group of objects. The memory model distinguishes sensory memory, short-term memory, and long-term memory. The information is transferred among different types of memory, depending on the action of filters whose permeability is determined by logic assertions about specific attributes (e.g. age of the observation; attention).

2.2.3 Szarowicz et al. – *Freewill*

Freewill was presented in [73] as a prototype that means to liberate animators from some of the decision responsibilities that are involved in the creation of animated scenes with human-like characters, by transferring them to the characters themselves. By empowering it with perception and decision making capabilities, the authors give the virtual character the ability of acting according to its goals and to a certain degree of awareness of its environment. The animation is simulated in the Freewill software and then translated into a format that can be used by some animation package, like Autodesk 3ds Max. The perception is done with a vision cone and the implementation of goal based planning is inspired by STRIPS [26]. The authors also discuss an example of the application of Freewill in a setting of avatars that walk in either direction of a city street, stopping to shake hands. In

[30], the authors discuss the application of their prototype to the generation of crowd scenes, comparing it with other alternatives. By this time, Freewill had already been updated, as they state that their model built on the cognitive architecture of John Funge (see section 2.1.4). Finally, in [74] it is shown that the system was further perfected by combining Funge's cognitive model with Winikoff's SAC (Simplified Agent Concepts) [85], thus employing a simplification of the BDI (Belief-Desire-Intention) architecture.

2.2.4 Torres et al. – *Autonomous Characters with BDI architecture*

Torres et al. propose the use of the Belief-Desire-Intention (BDI) model for cognitive agents that control animated characters [79]. They bring together an articulated model for character animation and an interpreter for *AgentSpeak(L)* [59], an object-oriented language that implements the BDI architecture.

Each character is controlled in real-time by a cognitive agent that runs in an individual process. The agents are implemented as clients of the environment, which plays the role of the server. The environment is responsible for managing the data that the agents can access, that is, the perception information. Each agent runs a perception-reasoning-execution cycle that determines its conduct. The beliefs of the agent are reviewed whenever it acquires new sensory data and the reasoning process may yield a single intention per iteration. When an intention is executed, new beliefs and objectives may arise and an action can be performed, which translates to an animation. The agents communicate with the environment through sockets [77]. Nor the way the graphical environment is implemented, nor its characteristics are explained.

In [78] new features are reported. Agents can now concentrate on multiple foci of attention. They are still only able to execute one intention at a time but they can now alternate between intentions along successive iterations of the cycle. New agents that are not responsible for animating characters are also added to the simulation. The authors explain that all possible perceptions of any agent at any state of the world have to be previously listed as facts that can be true or false. When a change occurs, this list is updated and an agent obtains sensory information by consulting it. This may be viewed as a drawback, in the sense that the domain over which agents operate must be very limited and, apparently, even discrete.

In [60] this work is further extended and the authors clarify that the graphical environment is now built on top of Cal3D API [44] and that the agents can now run in different interpreters of AgentSpeak(L). The multi-agent system is distributed and has built-in communication between agents.

2.2.5 Karla et al. – *Real-Time Virtual Humans*

Nadia Magnenat-Thalmann and Daniel Thalmann have long been researching in the field of VHs and they are unquestionable authorities in the field. Their teams have achieved amazing results that are documented in countless publications.

In [40] the authors synthesize the complex methods they were using for modeling and animating VHs in real-time, as early as 1998. These methods successfully combine artistic skills with the products of intense and diverse scientific research, to simulate humans with real-time visualization and animation. In this paper, they highlight the different constraints that are involved in modeling VHs for *frame-by-frame* versus real-time rendering, focusing on the real-time approach.

They choose to divide the modeling of VHs in separate parts that require distinct techniques. The hands and the head/face are modeled with polygon meshes whilst the body follows a multilayer approach. On the other hand, the body and the hands are subject to skeletal animation while head and face are animated using *Minimum Perceptible Actions* (MPAs) that are associated with basic motion parameters for elementary deformations[2].

The modeling of VHs for real-time applications is constrained by computational resources. The authors express this concern, in head modeling, noting that the number of polygons employed should reflect an equilibrium between appearance and efficiency, and observing that some of the detail can be achieved with texture mapping. Also the regions of the mesh that are less involved in animations allow sparing some polygons that can be used on those that participate the most. They suggest that head modeling should start from a prototype or from an hemisphere that can be copied for the other half, before some small changes are made on the whole head, since slightly asymmetric faces look more realistic.

The body is modeled with three layers. The first layer provides the skeleton, that allows the definition of postures. The second layer consists of groups of volume primitives known as *metaballs*. The metaballs simulate muscles' shapes and behavior and can be joined smoothly and gradually to give a realistic organic look to the body. The bodies are constructed by positioning, scaling and rotating these volumes, as well as by attaching them to a desired joint articulation, a process that requires strong skills in anatomy or drawing and sculpting, as the authors mention. The third layer is the body envelope, the equivalent of human skin, and it is defined as spline surfaces, through a ray-casting method. Textures are mapped on it, as well as on the head and hands.

For hand modeling, the authors begin with two basic sets of 3D hands, one for each gender. The muscular and skeleton layers can be parameterized to obtain morphological variations, such as hand thickness or finger length. The muscle layer

[2]Examples of MPAs are *open_mouth*, *close_upper_eyelids*, or *raise_corner_lip*

is first fitted to the skeleton, then the muscular operators are applied to the hand surface.

2.2.6 Ulicny and Thalmann – *Crowd Simulation*

In [81], Branislav Ulicny and Daniel Thalmann present a system projected for crowd simulation in which each individual is an autonomous VH. The system comprises two clearly separate layers: the *model layer* and the *visualization layer*. The model layer consists of a set of agents that are associated with VHs an on an abstract model of the environment. This abstract model contains the symbolic representation of the static part of the environment, along with information about the dynamic objects that inhabit it. In the visualization layer, graphical representations of the world, of the objects, and of the VHs are included, as well as the user interface.

The agents interact with the environment, with other agents, and with human participants of the simulation. They do so with events that trigger behavior rules and that are incorporated into changes of the state of elements that constitute the world. Excluding those that are introduced by the user, the events are created and processed in the model layer. The visualization layer is bound to exhibit the graphical results of the simulation, which is, therefore, independent. This rigid limit between the layers allows the simulation to executed offline, leaving the output as a log of its execution. The log can later be used to render the simulation graphically, with high quality and different possible 3D models.

2.2.7 Conde and Thalmann – *ALifeE*

Toni Conde and Daniel Thalmann created *ALifeE* (*Artificial Life Environment*) [18]. It is based on the multi-sensory integration approach of the standard theory of neuroscience, where signals of a single object coming from distinct sensory systems are combined. They equip *Autonomous Virtual Agents* (*AVAs*) with the main virtual sensors in the form of a small nervous system. These senses are *synthetic vision*, *synthetic audition*, and *synthetic touch*.

After obtaining sensory information, filtering, selection, and simplification processes are carried out to obtain a cognitive map. Behavioral animation is employed to have the AVA react to its virtual environment and to decide according to its perceptive system.

The AVAs are imbued with *proprioception*, which is the faculty of capturing information related to their internal state. An AVA has an inherent set of variables whose values must be kept within pre-determined boundaries and proprioception allows it to take actions to prevent them to exceed these boundaries. An AVA has also got *active perception* which, based on prediction capabilities, allows it to direct its attention elsewhere.

2.2.8 Rickel et al. – *Mission Rehearsal Exercise*

In [63], the *Mission Rehearsal Exercise* system is presented. A virtual world is inhabited by VHs that can interact with it, with the user, and among them. Because their minds are accomplished with intelligent agents whose abilities are domain-independent, the VHs can play multiple roles. Together with their minds, their bodies lend the VHs the typical realism of motion capture animation and the flexibility of procedural animation.

The VHs also have specialized valencies that, along with the graphical visualization and with an involving sound environment, lead the user to immerse in the military training experience. Dialog capacities are accomplished with a grammar-based approach that uses domain-specific knowledge to reach semantic meaning. Agents can assess events according to their knowledge and objectives and, as a result, they experience emotions that lead them to change their body language and that can have an effect on their beliefs, desires, and intensions.

The perception model reproduces many natural human sensory limitations, both visual and aural. The scope of an agent's vision resembles that of human vision and the detail of perceived objects varies on the basis of their position in the visual field. Hearing functions are modeled by sound pressure estimation that takes individual and cumulative effects into account, as well as distances and directions of sound sources. This allows, on the one hand, consciousness of some events that are not captured by vision and, on the other hand, masking of some sound events by others (for instance, an helicopter could prevent a VH from listening to someone speaking in normal tone).

2.2.9 Si et al. – *Thespian*

Thespian, a framework for realizing interactive drama that seeks to reduce programming effort, is presented in [69]. It aims at enabling the creation of interactive stories with minimal technical knowledge. To start, an author provides linear scripts of the drama. The environment and the possible actions are specified, along with the sequences of actions that allow the story to unfold. These sequences of actions are used by an automated fitting algorithm that configures agents to behave as intended. The character of each agent is modeled by tunning its goal parameters. The fitting algorithm assigns a suitable weight to each objective, according to the scripts that were originaly provided.

Thespian is built over *PsychSim* [49], a social simulation system that allows modeling characters as intelligent agents. Therefore, it provides features that are suitable for the creation of interactive stories. For instance, different policies can be chosen for action selection, such as reactive or bounded optimality (which involves limited lookahead). Another example of a PsychSim feature that is incorporated in Thespian is the possibility of having agents acting accordingly to subjective beliefs about the world or about other agents.

2.2.10 Barella et al. – *JGOMAS*

JGOMAS [11, 10, 82] is a game-oriented multi-agent system built over *JADE* [31]. It comprises a module for the multi-agent system, which is the main one, and one module for graphic visualization. Currently, JGOMAS implements a particular simulator for a *capture-the-flag* kind of game. Nevertheless, the authors express the intention of extending and changing the simulator to make it general and applicable to any type of multi-agent simulation. They also state that the framework must allow users to add their own code modifications (*mods*) so that it can be used for various purposes.

The multi-agent system includes not only the agents that play the role of *bots*, but also agents that are responsible for the behavior of game objects (e.g. health packs). There is also a special agent that has no graphical representation or *physical existence* in the game but that is responsible for the management of all game logics, as well as for providing interfaces for the visualization module. This agent acts as a server that can accept connections from one or more instances of the graphic viewer.

The graphic viewer is implemented in C++, using the graphic library *OpenGL* [54], and it communicates with the multi-agent system through sockets. Its only task is to display the information it gets from the multi-agent system (e.g. positions or velocities of agents or objects) in a 3D representation. As a matter of fact, the visualization module has no influence what so ever over the state of the world or over the progress of the game. Therefore, the user can interact with the simulation only through the multi-agent system.

The authors intend to generalize the networking interface of the multi-agent system module so that other visualization units can connect to the system, particularly other viewers that are built over complete rendering engines. Although JGOMAS is a very promising framework, some of its traits are left unmentioned. For instance, its not clear whether arbitrary 3D models can be associated with the agents, how are they animated, and what are their motor characteristics.

2.2.11 Other works

Longhi et al. present an architecture for an *Embodied Conversational Agent* that inhabits a virtual environment that can be built and edited by the user [46]. The agent, called *Maga Vitta*, is an entity that was conceived with cognitive and emotional capabilities for interacting with the user that deals with the environment *CIVITAS*, which was created by the team to allow children to interactively build virtual cities. The interactive features are embedded in the graphical representation of the agent as a *talking head*, in the authors' own words. The agent interprets the action that the user performs over the virtual world and expresses itself through speech and through the six facial expressions that were identified by Paul Ekman.

Multon et al. propose a framework for animating humans in virtual reality, capable of performing real-time motion editing in interactive complex environments [51]. It offers efficient and morphology independent motion representation [42] that sustains synchronization, retargeting and adaptation. Blending is achieved by an algorithm that is driven by priorities and states.

2.3 Discussion of Related Work

By comparing the work of Tu and Terzopoulos with the one of Reynolds, it becomes clear that there are two essentially distinct approaches to simulate reality in automatically generated animation. While Tu and Terzopoulos closely model the interaction between fish and the virtual world employing a holistic approximation of reality's processes, Reynolds abstracts lower-level details to produce generally applicable models, leaving the final appearance of the animation to be tweaked according to particular cases. Since then, research has been situated between these two approaches, leaning towards one or the other. Even though the tendency to abstract low-level aspects is more frequent, attempts of faithfully reproducing natural operations, with models that range from physics to cognition, are also found in the literature. Among these are the integration of sensory systems by Conde et al., the simulation of the aural sense with sound pressure estimation by Rickel et al., and the complex motions that the framework of Multon et al. is capable of producing.

Our approach with the IViHumans platform is more inclined to focus on the appearance of the simulation, even though it is produced by processes that may diverge from what happens in the real world. Indeed, Reynolds' work is probably the one that influenced us the most. We consider his conception of virtual movement as a grounding base for our approach and we adapt and implement it in our own way. We pay special attention to locomotion and our characters display consistent animations that are automatically chosen according to their own rules.

Reynolds' boids already apply an early version of synthetic perception (that he calls simulated perception) in which the actors detect companions and obstacles that are closer than a given threshold. Even then, the author recognizes that this threshold should not be homogeneous in all directions, but greater in the direction of movement. Evolution continued the trend of trying to improve to make virtual perception have the limitations that are intrinsic to real perception. Nonetheless, synthetic perception is a typical case in which the effects of these limitations are accomplished with models that completely bypass the complexity of real perception systems. Vosinakis et al., Noser et al., and Peters and Sullivan, all apply a model of virtual vision that acts by filtering out the objects that should not be seen by the characters, granting them access to the information that characterizes the ones that are seen (though sometimes its detail depends on attention). In the IViHumans platform, we use a model of ray-casting synthetic vision, like Vosinakis et al. did. We do so with an original algorithm that tries to maximize precision without any meaningful impairment of efficiency and that provides parameters that can be tweaked to find the correct balance.

Like these authors, we also have a memory model coupled with perception. It is able to remember default and custom properties of objects with three different memory management techniques. Our memory is capable of recalling the different states an object goes through in time, and not only one instantaneous state. This allows the extraction of conclusions about the evolution of the world.

Currently, this memory model is incomplete, as it still keeps only raw data that will be processed later, in the AI layer. The symbolic representation that will thus be produced will be used as the base of agents' reasoning processes. These agents will rely on the features of an agent-specific framework to compose a layer whose role is analogous to the top cognitive layer of Funge et al.

Like Vosinakis et al., Rickel et al. and Longhi et al., we also account for emotions on the IViHumans platform. Emotion will integrate cognitive models on the AI layer, according to what is described in [50]. On the GP layer, it will be indirectly expressed in the behavior of the characters and, directly, through their facial expressions. Unlike these authors, however, we propose a solution for facial expressions that allows the exhibition of complex expressions and that supports smooth transitions between them.

In the overall perspective, our architecture is also different from others. Torres et al. implement agents as individual processes. The agents play the role of clients of an environment that acts like server. Vosinakis et al. have the AI code supplied by the user through callback functions. The IViHumans platform seeks to evenly distribute computation tasks between both layers. The GP layer even includes some behavioral features that are sometimes classified as belonging to the field of artificial intelligence [15]. This allows reactive behavior to happen faster, without having to wait for the decisions of the AI layer.

The IViHumans platform is projected to be generally applicable, in order to enable different applications to use and build upon it. Our aim is to allow user applications to focus and extend either one of the layers, or both. Interaction with the user can be included in either side, as both layers have a determinant influence in the course of events, unlike what happens in the works of Perlin et al., Ulicny et al., and Barella et al. Also, in the approaches of these authors, the server is on the artificial intelligence side, while in ours the server is on the GP layer. Agents operate according to the basic services that the graphical layer, which is at a lower level, provides, instead of having the latter with the sole responsibility of rendering the effects of agents' operations. In this sense, our view is similar to the one manifested by Funge et al., in the sense that cognition is built on top of lower-level functionality.

Albeit having significant differences from the research work summed up here, the IViHumans platform is greatly influenced by it. Besides partially owing our perspectives to them, previous works like these can still originate ideas for future work on the platform that would increase its value. Our aim for flexibility is, in part, driven by the potential inclusion of great ideas that were already explored, and that still are, like inverse kinematics, anatomic modeling of VHs, or synthetic audition.

Chapter 3

The Conception of the IViHumans Platform

In this chapter, the IViHumans platform is globally laid out. First, its architecture is described, along with the way its components relate to each other. Then we proceed to some more detailed descriptions of its graphical side, paying special attention to the topics covered by the tasks we carried out. In the second part of this chapter, some implementation details are explained and the main related design options are discussed and justified.

Some of the issues that are involved in the work this chapter focuses on are solved by simplified approximations that aim at solution prototyping. These could be improved in the future. We sometimes expose some frequent techniques and solutions to problems that are shared by this project, even though they are not currently fulfilled by the platform. In these cases, it is discussed how they could be applied in the respective contexts, as a means of clarifying how the current implementation complies with the inclusion of more elaborate procedures and contents.

3.1 The Architecture of the IViHumans Platform

The IViHumans platform is composed by two layers: the Graphical Processing (GP) layer and the Artificial Intelligence (AI) layer [17]. The communication between them happens by means of TCP sockets (Figure 3.1). The GP layer is mainly responsible for visually representing the virtual environment and the elements that compose it, among which VHs are the most significant. Given that the environment is dynamic, the GP layer must properly exhibit adequate animations to carry on the flow of events, clearly and consistently displaying the changes that the world and its components undergo.

Whilst the GP layer hosts the "bodies" of the VHs, the AI layer manages their "minds". The AI layer consists mainly of a Multi-Agent System (MAS) whose agents may play several roles. High-level decisions of the VHs are performed by of one or

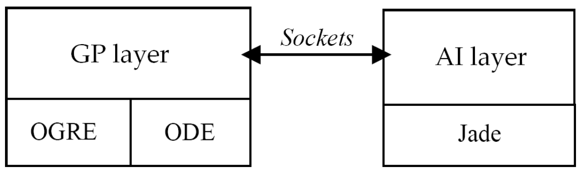


Figure 3.1: The Graphical Processing and Artificial Intelligence layers

more agents that can bring intelligent behavior into being.

Modifications in the scene can be lead by both layers. The GP layer manages a great deal of events and provides services of perception and action to the AI layer. So, an agent in the AI layer can ask the GP layer for the sensory information obtained by the VH it controls and command it, by ordering it to execute certain actions or to assume certain behaviors.

Instead of developing any of the layers from scratch, open-source softwares were selected, since the beginning of the platform's conception, to serve as a basis for subsequent development. Previous work established that the GP layer should be built over *OGRE* [55], as the essential library that enables the fulfillment of the graphics engine, and over *ODE* [22], a library that supports the completion of the physics engine through rigid body dynamics simulation. The AI layer, on the other hand, is meant to be implemented over the FIPA [27] compliant agent middleware that the *JADE* framework [31] provides, as a base environment for the MAS that underlies this layer.

The GP layer of the IViHumans platform is the one that manages the core aspects of the virtual world and that is responsible for rendering the scenes. Its duties range from the management of lights and cameras to the movement of the objects that are present in the world. Applying textures and materials to the models, computing mesh deformations, animating all the elements, handling collisions, updating the internal state of all the entities and reflecting it in the rendering, are some examples of the tasks we want it to cope with. Besides this, it must be able to engage in formal communication with the process that runs the AI layer, through sockets, according to a specific protocol, providing perception and actuation services, among others, to the agents that lie on the AI layer.

We realized, from about the time we began this work, that we had to assign priorities to the objectives of the project, because many could not be accomplished with the time we had available. We ended up establishing one main orientation: *we should begin by creating a small set of features that made the GP layer functional and that would allow the connection to the AI layer; only then should new features be iteratively added.*

One of the grounding issues we had to solve was how were the VHs to be rendered.

Their graphical representations, as for any other objects, rely on digital content that is produced offline and loaded at runtime. *Blender* [12] is the tool suite that was chosen for 3D content creation, particularly for the 3D models that are used in the visual representation of VHs, remaining objects, and scenery. Recently another tool was added to the content creation pipeline: *Poser* [3] (the choice of this tool is explained ahead). In fact, any 3D modeling tools can be used to the same end, as long as there is some way to translate the models from the original format of that particular tool to a format that *OGRE* can understand. The choice of all these softwares happened before the beginning of the work conveyed by this document. It obeyed well defined criteria – such as what were the cost, offered features, or the activity and dimension of any related user community – and it is documented in [71]. Figure 3.2 illustrates how content creation tools articulate with the remaining architecture of the platform.

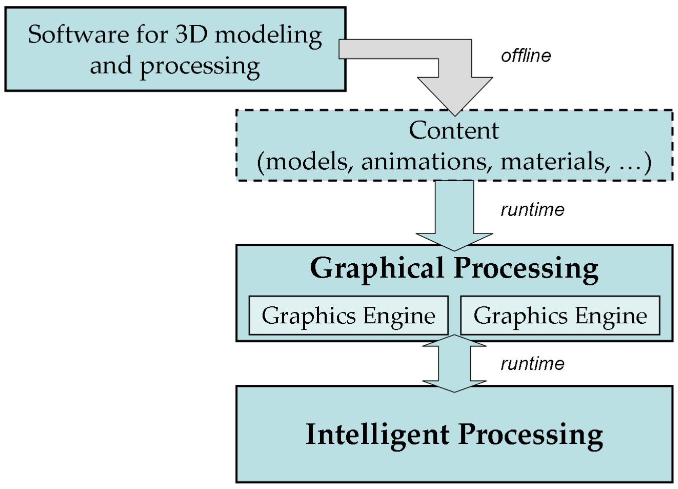


Figure 3.2: Global architecture of the IViHumans platform

The next section describes the content we used, how we obtained it, and some techniques that lie on the basis of both its production and rendering.

3.2 Visualization of Objects and Virtual Humans

One of the essential features that a virtual environment should have is the propensity for capturing the user's attention so that he "forgets" his real environment and focuses on the simulated environment. In other words, and borrowing an expression from the field of virtual reality, the virtual environment should provoke a sensation of *immersion* on the user. To that end, one of the issues to account for is the realism of 3D models and the naturalness of animations associated with interaction movements among all the elements of the environment, particularly in what concerns VHs. On the other hand, the models that are used in real-time applications of virtual environments should be “light”, that is, they should not be defined with excessive detail, nor depend on too heavy algorithms, due to the inherent restrictions to graphical processing in real-time. The combination of these two requirements has driven much of the recent progress concerning graphical simulation of human beings, both static and dynamic, and led to diverse techniques, such as skeletal animation, metaballs, anatomic modeling (with bones, muscles and skin), motion capture, Levels Of Detail (LODs), Inverse Kinematics (IK), hair simulation, or cloth simulation [40, 47].

In the following subsection we briefly introduce some of the techniques that are commonly used in 3D modeling for real-time applications and describe the models we used in the platform.

3.2.1 Static Representation

In the IViHumans platform, and in a preliminary approach, a prototypical model of a VH was defined as a polygon mesh to which some simple materials were applied. The model represents a female human and its body has an associated skeleton that enables its deformation, hence allowing easier movement definition. A set of poses that affect this VH’s face is also defined for the exhibition of facial expressions.

However, modeling VHs is a task that requires refined artistic skills and, for purposes other than early testing and demonstration, it is best left to talented specialists. Acknowledging this evidence, we later integrated a higher-quality commercial VH model into the platform, for tests and demonstrations. This model was obtained for free from *aXYZ design* as a textured, fully rigged mesh[1] that represents a male human. We applied some animations to it that we derived and adapted from free motion capture data obtained from the website `www.mocapdata.com`[2].

The models that figure in mainstream animation movies have an astonishing 3D detail, in part at the expense of a large number of polygons. However, rendering films takes a lot of time, even when it is carried out in the so called *render farms* – sets of computers that work in parallel in the generation of frames. This option is obviously not viable when real-time rendering is needed. In the IViHumans platform the objects and characters that are used should not be modeled by resorting to too many polygons, due to efficiency requirements, specially if one intends to include many objects and characters in a simulation without impairing the performance of the visualization. Indeed, low-polygon modeling is a frequent approach to video game design, since video games share requirements of fast processing and real-time execution [36]. In spite of that, the term *low-poly* denotes a concept that is intrinsi-

[1]Copyright ©2008 aXYZ design (`http://www.axyz-design.com/`)

[2]The motion capture data that we used is licensed under a Creative Commons Attribution-Share Alike 2.1 Japan License

cally subjective and whose interpretation varies parallelly with what a fast computer is considered to be. These observations were kept in mind when we developed our first model, which has around 12500 triangles, with a great concentration of them in the face, and when we decided to use the second model, which has around 4600 polygons. These might be considered to be relatively low-poly character models, when compared with many modern ones [3].

Like the majority of rendering libraries, OGRE allows the definition of complex materials that may rely on programs for the Graphical Processing Unit (GPU), called *shaders*, that can take full advantage of the rendering capacities of this specialized hardware [29]. Great quality and spectacularity may be reached by defining *pixel shaders* or *vertex shaders*, but this kind of programming requires specialized knowledge to which we do not attribute much priority and that is not included in our current abilities. In spite of that, some complex materials were already defined in a previous work [25], through composition of procedural textures, in Blender, namely for the skin and for the eyes. However, this materials cannot be directly translated into the shaders that OGRE supports, even though the way that the platform is projected does not raise any obstacle to the potential definition of more elaborate materials, as long as their formats can be assimilated by OGRE. For now, the materials we use are limited to the application of colors, to the moderate use of transparencies and to *UV mapping* of textures. Different colors are sometimes assigned to a polygon, depending on the way they react to light. In OGRE, each polygon can have one color for each of the following attributes: *ambient*, *diffuse*, *specular*, and *emissive*. By combining these attributes, nuances of light and material interaction can be approximated. Figure 3.3 illustrates the effect that can be achieved by combining only specular and diffuse color attributes.

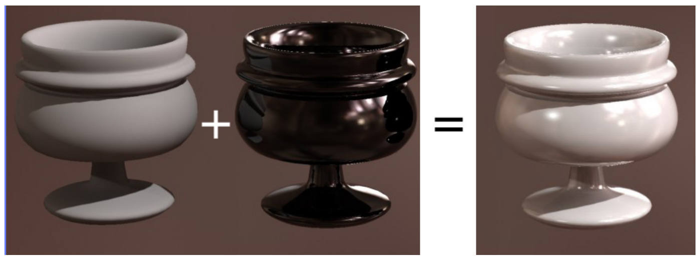


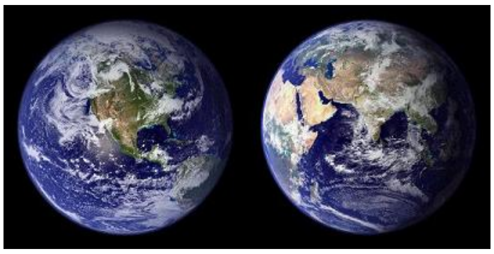
Figure 3.3: Combination of specular and diffuse reflection

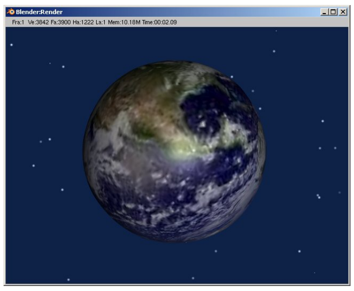
UV mapping is a technique for mapping 2D images over 3D surfaces. Texture mapping is done with a 2D coordinate system that establishes a relation between the image and the 3D model. Each pixel of the image is identified with a pair of coordinates (U, V), hence the name. When one of the pixels is applied to a vertex

[3]For instance, polygon counts of native Poser characters often amount to numbers of the order of hundreds of thousands

of a mesh, a *texture element*, or *texel*, is originated. Typically, U and V may assume values on the range $[0, 1]$ [4], so that the four corners of the image are identified by the pairs $(0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$. Given a polygon whose vertices are associated with UV coordinates, the remaining texels are simply derived from linear interpolation. How the information contained in a texel is interpreted and applied to the model varies, raising different types of texture mapping that can be combined or used separately. *Color maps*, *transparency maps* and *bump maps* are examples of different types of texture mapping.

The UV mapping technique is also used in the models that we use in the IViHumans platform, in its most direct form – color maps. While in transparency maps and in bump maps the color of the texels are translated into transparencies or surface normals, respectively, in color maps the color of each texel is directly applied to the surface. Figure 3.4 shows the effect of correctly mapping images of the earth over a sphere. An introductory, yet detailed, description of UV mapping can be consulted in [56].


(a) 2D images of the earth



(b) The result of mapping images of the earth over a sphere

Figure 3.4: UV Mapping of images into a mesh

The virtual woman prototype we created (Figure 3.5) combines colors, transparencies, and texture mapping. The eyes, whose composition and modeling is documented in [25], employ all these techniques (Figure 3.6). However, their transference to OGRE's format implied the loss of some quality that cannot be restituted without resorting to shaders. The modeling of this female character started from the face, which had been previously created. A body was then extracted and adapted from Poser and merged with the head. This process also involved fitting a skeleton for animation purposes and it is discussed in the following section.

The male VH was obtained from *aXYZ design*, as mentioned above. Its materials are essentially made up of color maps that give it much more realistic appearance because they are properly done and include such details as cloth folds and wrinkles or facial hair (Figure 3.7). The aspect of this character while standing still can be

[4]Values outside of this range are specified when the image is applied repeatedly, as a tile

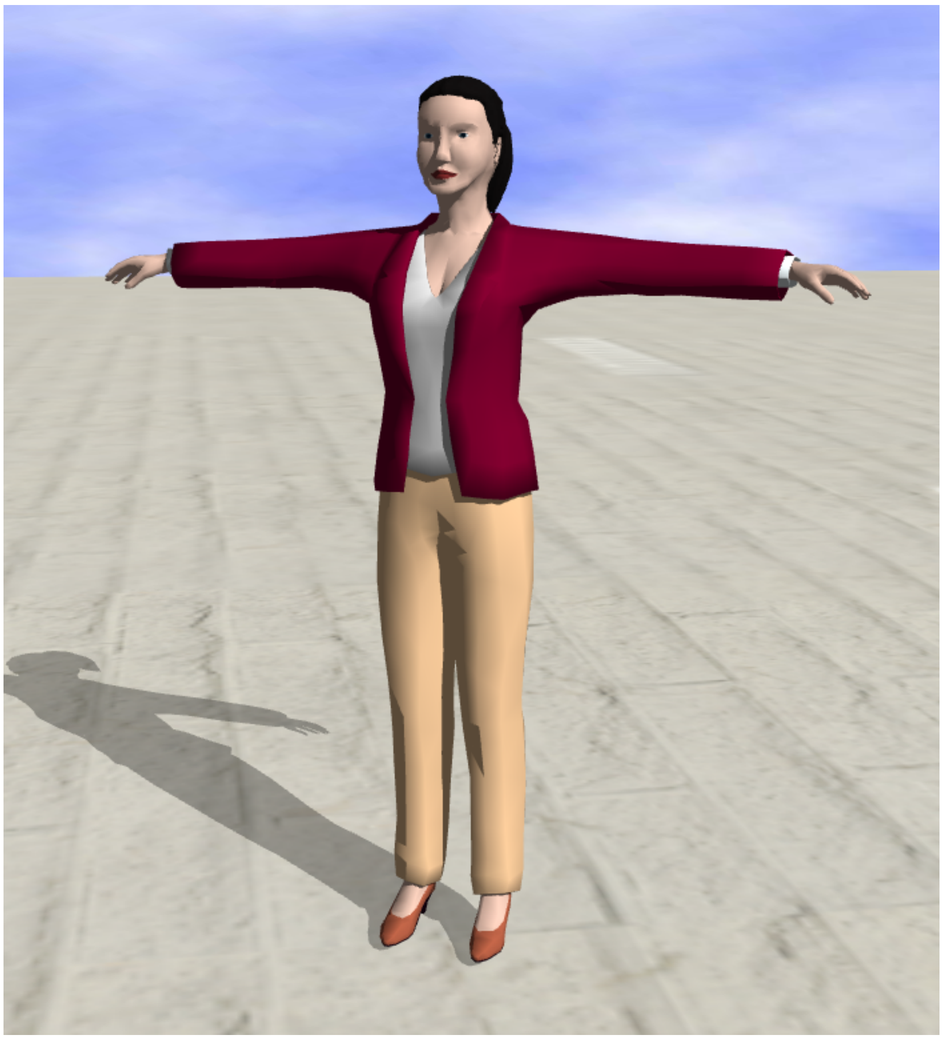


Figure 3.5: The virtual woman prototype, in its original posture

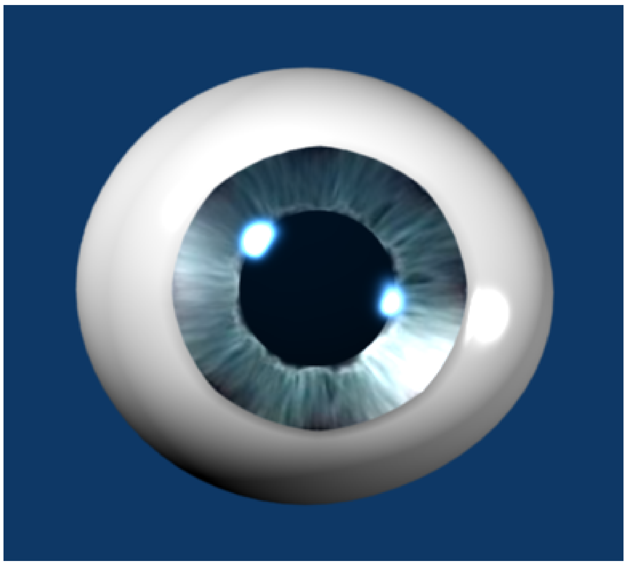


Figure 3.6: The eye that was modeled for the female virtual woman

seen in Figure 3.8.

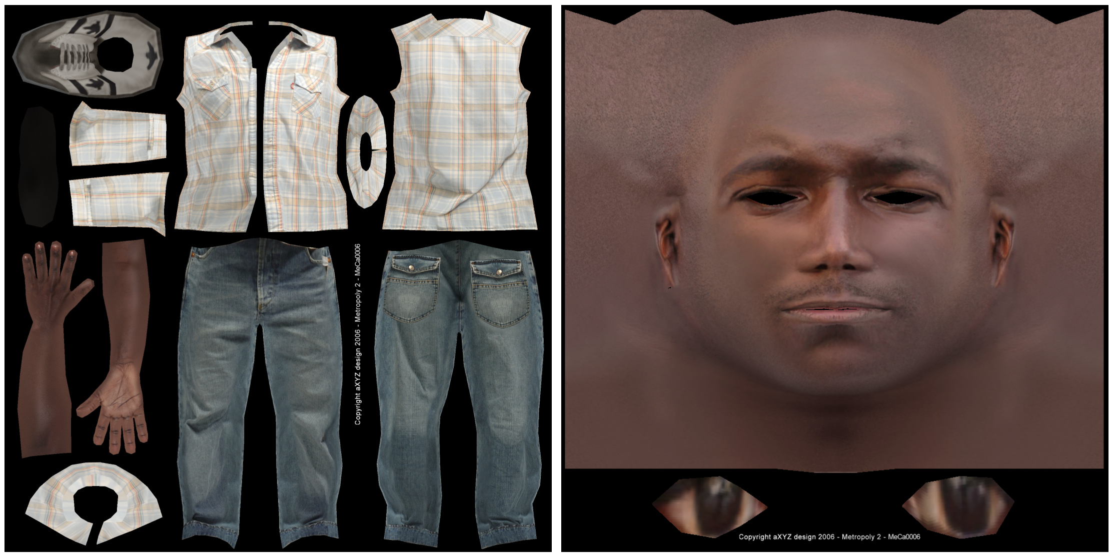

Copyright aXYZ design 2006 - Metropoly 2 - MeCa0006
Copyright aXYZ design 2006 - Metropoly 2 - MeCa0006

Figure 3.7: The images that are used for color-mapping the virtual man. Copyright ©2008 aXYZ design

3.2.2 Dynamic Representation

The creation of the female VH began with face modeling and with the definition of the six basic expressions identified by Paul Ekman [24, 25]. The face was modeled as a polygon mesh and the deformations that correspond to the basic expressions were codified as *poses* of this mesh. A pose records a set of displacement vectors for the vertices of the mesh. When the vertices are translated by the corresponding displacement vectors that a given pose specifies, the geometry of the mesh changes so that the intended expression is exhibited. If one intends to show an expression just partially, it suffices to reduce the length of the vectors that specify the translation of each vertex. On the other hand, the visualization of complex expressions can also be easily achieved by mixing basic expressions with desired intensities. Suppose, for instance, that two poses are defined for a face mesh: joy and surprise. An expression that is composed by a partial joy expression with 80% intensity and by a partial surprise expression of 30% can be obtain this way: for each vertex of the mesh, multiply the corresponding displacement vector, in each pose, by 0.8 and 0.3, respectively; then, add the two resulting vectors and translate the vertex according to the final vector.

In the same work, a tool was created for obtaining complex expressions by mixing basic ones, observing the effect in real-time. This tool was called *Faces*. The user can vary the intensity of each basic expression intuitively by playing with scroll buttons, and the results are immediately shown as a deformation of the face. In order to

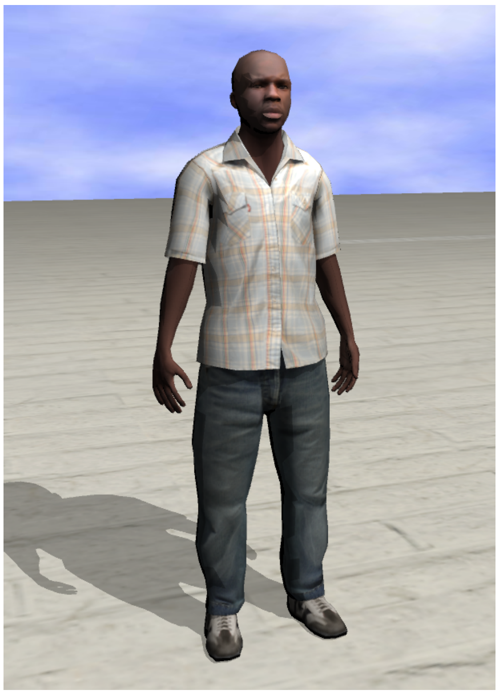


Figure 3.8: The male VH in a static position.
Copyright ©2008 aXYZ design

prevent excessive deformations being applied, the intensity of each expression must be comprised in the interval $[0, 1]$. These limits are implicitly imposed by the Faces' Graphical User Interface (GUI) since the positions of scroll buttons can only be varied so much. The sum of individual intensities is not limited, however. Indeed, one can even create a complex expression by mixing, in the limit, all the basic expressions with 100% intensity. Obviously, the results would not be the most natural. Even though there is no limit for how much deformation is applied, pushing it to far will create uninteresting results and should be avoided, unless it is done with an experimental resolution. It is up to the user to find the correct balance that produces the intended results. This tool was later perfected and completed. The user can now save new expressions as individual poses. This way, rich libraries of facial expressions can be created for each model.

Once we had the face of the female VH, it was neccessary to give it a body and to animate the resulting model, so that it had the ability of wandering about the scene. The body, cloth, and hair of the VH was derived from an model we obtained in Poser as a polygon mesh. The geometries of the meshes (body, cloth, hair, and face) were adjusted so that they would fit together and could be merged into a single continuous mesh.

We prototyped only one animation for our virtual woman: a walk animation. To produce it, we used a very common technique: skeletal animation. The method relies on the assignment of a skeleton to the mesh whose animations should be produced.

According to this technique, specially suitable for the animation of vertebrate characters, the abstract representation of a character is composed by two elements: the mesh and the skeleton. The mesh is a structured set of polygons that shapes the body of the character and coincides with its rendered aspect. The skeleton, which is not visible in the rendered images, consists on a set of bones organized in one or more tree structures. Figure 3.9 shows the mesh and the skeleton, still separate and slightly misfit, during an intermediate modeling phase.

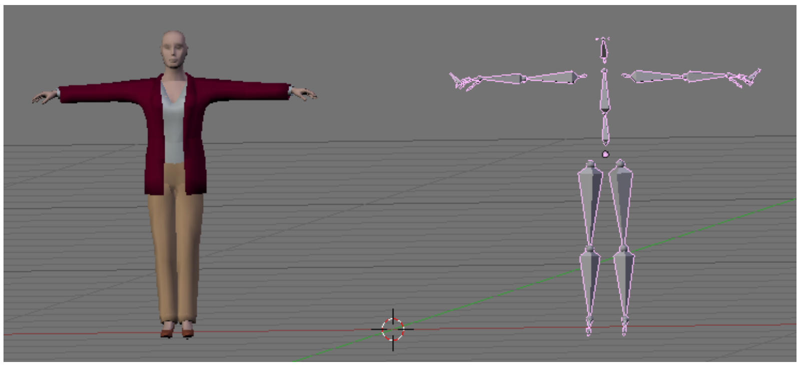


Figure 3.9: Skeleton and mesh of the virtual woman, still disconnected and slightly misfit

The skeleton is organized hierarchically so that parent bones have their movement automatically inherited by their children. Typically, the skeleton consists only of a single tree. It has one bone as root and other bones descend, directly or indirectly from it. As in any tree structure, every bone as a parent, except for the root, and all have one or more descendants, except for the leaves. The animation of the skeleton results from the animation of its bones, in such a way that each bone inherits the movement of its parent. For instance, in a simple skeleton whose root is the hip bone, which has the leg bones as direct children, these latter ones have all the movements of the former and, potentially, others too. If one applies, to the hip bone, a translation of 10 units along the X-axis and a rotation of 90^o about the Y-axis, the leg bones will also suffer these movements. If each leg also has a foot bone as child, and if each one is subject to an additional rotation of -60^o about the Y-axis, the bones of the feet will inherit the translation of the hip and a rotation of 30^o about the Y-axis. This kind of approach enables fast creation of the movements that characterize vertebrate beings, since it is a good conception of the way their skeleton behaves when in absence of external forces (particularly, gravity).

The movement of a bone, by itself, does not meet any function, since the skeleton is not even a part of the rendered images. Its purpose is only that of being applied to a mesh object. For the movement of a skeleton to be reflected on the mesh, they must be connected. This is done by associating each vertex of the mesh to one or more bones of the skeleton and by assigning a real value to each such connection.

This value is usually comprehended in the interval $(0, 1]$. It can be regarded as the intensity, or as the weight, with which the movement of the bone affects the movement of the vertex. The displacement of each vertex of the mesh is then calculated as a function of the movement of the bones it is associated with and of the weight that describes each one of these associations.

The walk animation that was mentioned before was also created in *Poser*. To this end, we used the skeleton that was predefined for the chosen model and we defined its motion by parameterizing a base movement, relying on a set of features provided by that tool. The skeleton, already imbued with the walk animation, was imported into Blender and associated to the mesh. This process involved consecutive adjustments of the mesh, to eliminate any residual interpenetrations among body and clothes and among different parts of the body. The skeleton and the mesh were then exported to *OGRE* formats and went through a series of additional procedures that we conceived to overcome other obstacles. At last, we were able to achieve a prototype of a VH, ready to be integrated in the platform and able to walk and to assume several facial expressions. The elaboration of this virtual woman faced some other difficulties that are not mentioned in this document, though the process we devised to create her is documented in [5] and, in more detail, in [4]. The result of applying a skeleton posture to the mesh is shown in Figure 3.10.

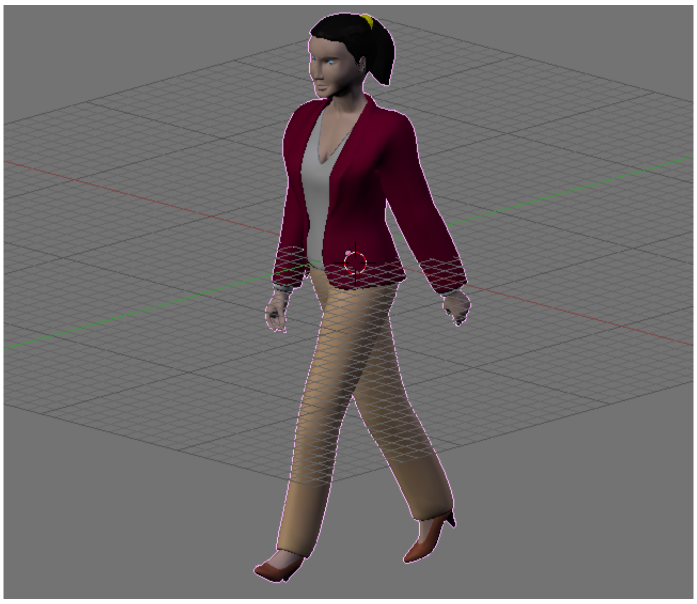
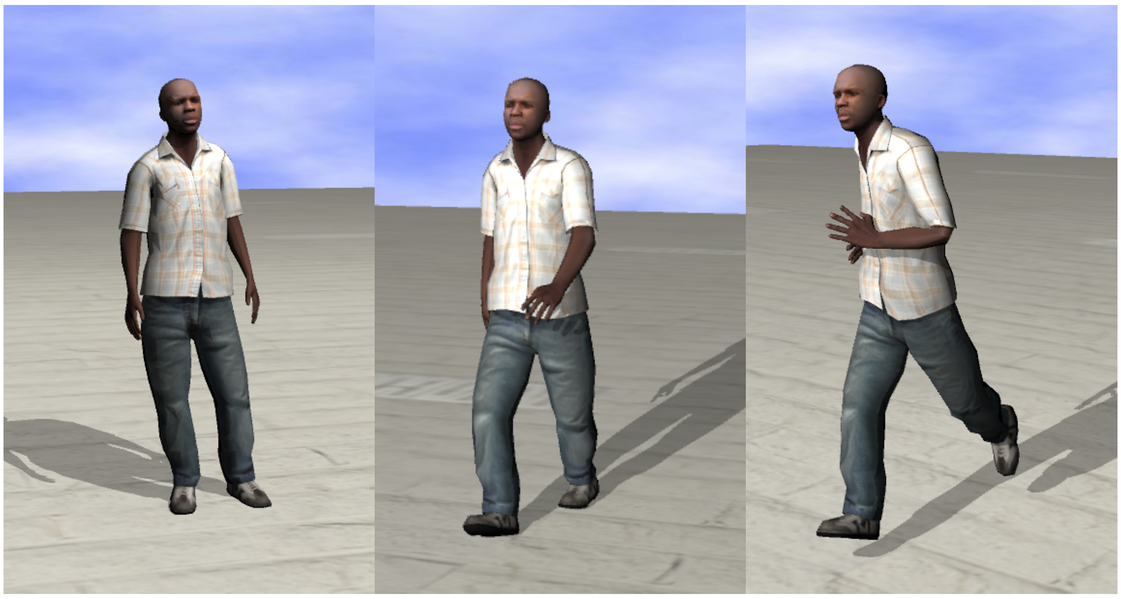


Figure 3.10: The virtual woman in a posture determined by her skeleton

The static posture of the female VH – when no skeletal deformation is applied – is the one that was originally defined for the character we got from Poser and, even though it may be adequate for modeling tasks, it is not a rest pose that a real person would assume. We wanted to establish a position for the skeleton that corresponded to a more realistic pose and we performed several attempts with that purpose, experimenting different connections between the skeleton and the mesh, but to no avail, due to interpenetrations that always appeared among different parts of the mesh. In order for the postures of a skeleton to reflect properly in the mesh, the associations between the vertices and the bones, as well as the respective weight values, must be accurately determined, which requires specialist skills. Moreover, an association between the skeleton and the mesh will often fully serve a particular animation while being inadequate for the rest. Hence, the connection of the two components is usually defined iteratively, being target of successive improvements, in order to support the generation of a varied collection of consistent poses. In the particular case of the creation of our VH, we also faced this common issue. Unfortunately, it is a problem we were unable to solve, since we could not reconcile the necessary artistic skills.

Aiming at a male VH prototype that could move around the scene conveying a better sense of believability, we searched the internet for a free model that would not require much modeling work. We came to a model of a male VH that was provided for free by *aXYZ design* and we found it very realistic while still perfectly suitable for real-time demonstrations. This model was already textured and fully rigged. The association between mesh and skeleton proved to be quite perfect and we could easily create several poses without incurring the issues we had with our prototype. We downloaded the model in Autodesk 3ds Max [1] format and readied it in a trial version of this software.

We imbued the model with three animations: an idle animation (for when the VH stood in the same place), a walk animation, and a run animation. These animations were derived from motion capture data we got from `www.mocapdata.com`. We obtained them in a 3ds Max proprietary format that could directly be applied to the skeleton of the VH – the *biped* format. However, this format only records the movement of some bones, which sometimes can originate strange animations in which some parts of the body move in a very realistic fashion, while others stand rigid. For instance, whenever the arms of the model moved, his fingers and hands would stay stiff and still relatively to their bone's parents. We tried to mitigate this problem by merging some custom keyframes, concerning the bones that did not move, with the original animations. We also had to select sections of the original animations and change them so that they could be played cyclically. For that, we modified the poses of the bones in the frames of the beginning and of the end of these sections, in order to have the first frame be a natural sequence of the last one. The position and orientation of the root bone were also adjusted in every frame, so that the animations did not move the VH from his original place. Once these tasks were complete, the model was exported and immediately ready to be used with OGRE's library. In the end, and considering our amateur abilities in what concerns 3D modeling, the result turned out to be quite alright. However, this was only possible due to the high quality rig of the original model. Figure 3.11 shows screenshots of the male VH in each of the animations we created for him.



(a) Idle (b) Walking (c) Running

Figure 3.11: The virtual man while idle, walking, and running
Copyright ©2008 aXYZ design

In both VH prototypes, the animations are composed by a set of keyframes that record successive postures of the skeleton. Each keyframe encodes a set of transformations that, when applied to the bones, put the skeleton in the corresponding pose. The keyframes are assigned unique instants of the animation, normally separated by regular time intervals. The posture of the skeleton in any animation is then generated by linear interpolation of the two closest keyframes, chronologically speaking (one that precedes it and another that comes after it). Each animation is created in such a way that it would remain coherent if its last frame was placed in the begining of the animation and the rest were shifted. So, the animation can be repeatedly played with the first frame perfectly fit to succeed the last one. That is, the animations are *cyclic* and can be played unlimitedly.

Besides the animations the models currently have, several more would be desirable, to enable actions like sitting, grasping, and moving the head. In fact, the typical number of animations of a videogame character, according to [28], is a hundred, for central role characters, and about twenty, for supporting characters.

Additionally, an IK solver could be included, either to deviate existing poses or to create new ones on the fly. The skeletal animation methods we have been talking until now are included in the *Forward Kinematics* (FK) approach. According to it, the movement of the bones are transmitted in the top-down direction – from the root to the leaves – of the bone hierarchy. In FK, each bone determines the original position of its children but it has no influence over its parent. However, in real life, the movement of a bone propagates influences in both ways and, indeed, the notion of parent or child bone, which is imposed in 3D animation only for convenience, does not make much sense. IK is the converse of FK and it is used for deriving interactions in the bottom-up direction of the skeleton tree. In other words, IK allows the derivation of movements of a bone chain that put its utmost point in some intended position. Usually, an IK solver operates with skeletons that impose movement restrictions that limit the chains of bones, so that inconsistent positions are prevented. For instance, an IK solver could find ways of moving the bones of a leg so that the tip of the foot would reach a certain position. For that, it would work its way from the foot bones to the femur, which would probably end the bone chain.

An IK algorithm could also be used to, for instance, understand how a VH could put his hand in a certain position and with a certain orientation. If the VH had an animation for grasping, it could be combined with the additional offsets that the IK prescribed so that objects in different locations could be grasped. The IViHumans platform does not integrate IK but we see it as an hypothesis of future work that would greatly benefit it.

The subject of IK is really vast and there are various approaches for its application. Some of the techniques that were developed to solve the issues it deals with even have their own taxonomy. For a small introduction to the subject and a set of other references concerning it, see [47].

The walk and run animations of the VHs we use are designed for moving in a straight line, but we also use them when the characters turn. To increase believability, the VHs should be animated in distinct ways depending on their angular velocities and accelerations, so that their movements more closely corresponded to the impulses that should generate them. A possible solution would be to have a set of different animations for different turn arcs, so that these base animations could be blended to generate a proper animation for each arc tightness. Some kind of IK could also be applied to deviate the movement of the original straight line movement animations and achieve the intended result. To find and apply a precise solution for this issue is left as future work.

3.3 Implementing the Graphical Processing Layer

During the time we were involved with the platform, we could only complete a fraction of all the features we wish some day will be provided by the GP layer.

Nonetheless, the core set of features we implemented give the GP layer a substantial body, and we believe it can be a great aid for creating applications that involve virtual environments and VHs. Moreover, we have reached a state of development that, for the first time, allows the AI layer to be implemented. This was not possible until now, since the AI layer must rely on the services of the GP layer and these were not defined yet.

The GP layer is developed with the C++ programming language [72], since this is the language in which OGRE and ODE are implemented. Although the ODE is projected to be one of the base libraries of the GP layer, we did not use it yet. Although physics is an essential part of graphical processing, we did not have the time to implement it. Therefore, no collision handling exists yet: when objects collide, they simply interpenetrate. In our opinion, it is the greatest priority for the near future, on what concerns the GP layer.

This section focuses on the issues we devoted most effort to and on the features that are currently implemented. We explain our main design and implementation choices and we try to make apparent how they reflect our two greatest concerns, which are intimately interconnected: to make the platform extensible and to give it a wide range of applicability. We considered these properties very important, since they should characterize any library that is used as a base for creating applications and because of the iterative fashion we adopted for implementing the platform. The features that are added to the platform at any time should not prevent subsequent development, even if for pursuing goals that are not yet planned. Whenever possible, existing code should be self contained so that there is no need to change it when adding new code. It should also be general and suitable for applications that have different purposes.

The organization of the main features that we included in the platform can be roughly abbreviated by the following scheme:

- VH's Action
    - Movement
        - Steering
        - Locomotion
    - Expressions
- VH's Perception
    - Synthetic Vision
    - Memory
- Networking

– IViHumans' Server

In the IViHumans platform, the VHs can perceive the environment through ray-casting synthetic vision. It is accomplished with an algorithm that was devised in a previous work [67]. That work also involved the implementation of a preliminary version of the algorithm that is documented in [68]. Although, as nothing else was implemented at that time, apart from the tool *Faces*, this early implementation laid directly on top of OGRE and served only as a proof of concept. Recently, we implemented it again from scratch and included it in the platform. We also created an automatic memory mechanism associated with it. These aspects are discussed in Section 3.3.3.

Currently, the VHs have two ways of acting: moving and exhibiting expressions. As collision handling is not included yet, and since no modules of interpretation of expressions exist, both movement and the exhibition of expressions are considered actions of a special kind. Indeed, the state of the world is left unchanged by them, apart from what concerns the entity itself.

In what regards movement, we did not try to mimic all natural processes in our models, from physics to decision making, as did Tu and Tersopoulos with their artificial fishes (see Section 2.1.3). Instead, we focus on the final appearance of the simulation. We follow Reynolds' conception of movement and his division of it in the three layers of locomotion, steering and action selection (see Sections 2.1.1 and 2.1.2). We are not concerned whether motion is generated by natural processes, only that it looks natural in the end. We follow the simple vehicle model, as an abstraction of locomotion over which the steering layer can be implemented. We based our implementation on the explanation of [15], but we adopt a rather different approach in design. Once having the steering layer, distinct entities can map locomotion into actual movement in different ways. Our design and implementation of movement is explained in Section 3.3.1.

The VHs of the IViHumans platform were given the ability of conveying emotions through either basic or composite facial expressions. Transitions from an expressionless face to any expression, and conversely, happen smoothly, as do transitions among different expressions, even if several expressions are displayed sequentially in a short time period. Event though we only have expressions that are modeled as poses, we took into account that other ways of defining them should also be allowed and that is reflected in our design decisions, which are exposed in Section 3.3.2.

As explained earlier, the GP layer has to provide services to the AI layer. In fact, the connection between the two layers follows the client-server architecture, being the GP layer left with the role of the server, while the AI layer is the client. Besides managing the graphical aspects of the environment, as well as many logical ones, the GP layer has to accept orders from the AI layer, to control VHs, the environment, or its elements. TCP was defined, early on, as the protocol in which the communication between the layers should take place, and we designed and implemented a simple, yet extensible, protocol on top of it, for networking on the IViHumans platform. The server was implemented so that it accepts and handles requests automatically, as explained in Section 3.3.4.

Besides OGRE and the C++ standard library [37] we also used *boost* libraries [14] and a modified version of *OPCODE* [21]. OPCODE was used indirectly, through an OGRE wrapper called *OgreOpcode* [53], to implement the ground for collision detection in the IViHumans platform. Collision handling – which includes collision detection and response – is far from finished, but collision detection is already supported. However, until now we only used collision detection for the implementation of the vision algorithm, as will be explained ahead.

The main boost library we used was *Boost.Asio*, to implement networking with the AI layer, but we also used other libraries as programming aids. *Boost.Foreach*, *Boost.Bind*, *Boost.Lambda*, *Boost.SmartPtr*, *Boost.Timer*, and *Boost.Utility* are some examples. From these, Boost.Bind and Boost.SmartPtr were included in the technical report *ISO/IEC TR 19768: C++ Library Extensions TR1* [7], more commonly known only as *C++ Technical Report 1*, that suggests a series of extensions for the C++ standard libraries that will probably be included in the new standard, aimed for 2009 by the C++ Standards Committee. Either way, all the libraries we used for the implementation of the GP layer are standard or open-source, with public implementations for all major operating systems. So, in principle, the platform can be used with any operating system, although we only tested it on Microsoft Windows XP. In practice, some minor changes might be necessary to compile and execute it in other operating systems.

In the following sections we discuss the design and implementation of the GP layer in more detail, paying special attention to its main features. We allude to some contents of the OGRE's library, so we suggest consulting the partial documentation of its API[5]. The fully documented API of the GP layer may also be helpful[6]. There are also several places that document the standard C++ library, but we suggest Josuttis' book [37].

3.3.1 Movement

As stated, the GP layer is implemented in C++, hence we tried to guide our design according to the *Object-Oriented* paradigm, seeking to take full advantage from it. The VHs are the central element the IViHumans platform has to deal with and we put them at the heart of the GP layer's implementation. Concretely, the VHs

[5]http://www.ogre3d.org/docs/api/html/

[6]http://labmag.di.fc.ul.pt/virtual/IViHumans-GPLayerAPI/api/html/

are implemented, in the GP layer, by the class *IViHuman*. Objects of this class are particular VHs that can perceive their environment with synthetic vision, move about it, and exhibit expressions.

The most obvious attributes that a VH must have are the ones that describe its basic physical state, such as position, velocity, or orientation. Values for these attributes can be derived by applying laws of classical physics. Using these laws, an IViHuman can update its own physical state if it is given the time that elapsed since the last state. However, this ability is not exclusive of a VH. It is part of every entity with physical existence that moves in the virtual world, be it a ball, a car, or a camera. Thus, we created a base class that aggregates this functionality, so that every moving entity can inherit from it, and we called it *IViEntity*. This class has also got other functionality, discussed ahead, that should be common to various elements. Its inheritance diagram is shown in Figure 3.12.

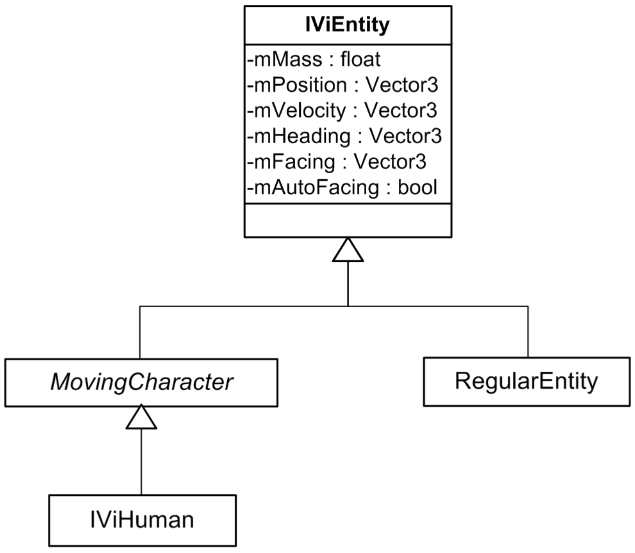


Figure 3.12: Class Diagram – *IViEntity*

In what concerns movement, an IViEntity models a point mass that has a *mass*, a *position* and a *velocity*. A vector that specifies the direction of movement is also kept and updated. This *heading vector* is always normalized and tangent to the entity's path at every instant. Another vector, called *facing vector*, is also included. It specifies the direction the IViEntity is facing and it can be automatically updated to match the *heading vector*, if that is what is intended. For that, the IViEntity's *auto facing mode* should be set on. If these two vectors coincide, the entity faces the direction it moves. Otherwise, the entity moves in one direction while facing another. We conventionalize that the heading vector is the null vector when the entity is not moving. The facing vector is left intact in this case. The movement of an IViEntity can be automatically updated as a function of the time elapsed since the last update, with a call to the method *IViEntity::updateMovement*. This involves updating the position of the entity according to the formula $\overrightarrow{v_m} = \frac{\Delta \overrightarrow{x}}{\Delta t}$ and, in case there was a change in velocity, updating the heading vector. If the auto facing mode is active, the facing vector is also updated to coincide with the heading vector, unless the latter is null.

The class IViEntity implements only an abstract representation of movement. It is not even connected with OGRE in any way other then by using its basic tipes (e.g. vectors). It serves as a base class that provides some basic movement functionality to other classes that inherit from it. Its functionality can be identified with part of the functionality of Reynolds' vehicle model and. Once mapped into rendered motion, it is enough for entities whose movement is generated by some external force or will, that is, entities that do not need to appear as being self-powered or as moving on their own.

The ***MovingCharacter*** and the Steering Behaviors

MovingCharacter is one of the classes that inherit from IViEntity and it extends its movement functionality to the point that the vehicle model achieves, completing an abstract interface of the locomotion layer, over which the steering layer can be built. Besides updating its position on the basis of velocity, the MovingCharacter can also compute a new velocity from the acceleration that, on its turn, results from a force. This is performed according to the laws $\overrightarrow{a_m} = \frac{\Delta \overrightarrow{v}}{\Delta t}$ and $\overrightarrow{F} = m\overrightarrow{a}$, except for the fact that a MovingCharacter is restricted by limits for the maximum velocity it can reach and for the maximum force it can apply on itself.

To construct an instance of MovingCharacter, certain knowledge is needed. This class's constructor receives a series of parameters that configure a particular MovingCharacter, along with its initial state. We make this distinction between a character's configuration and its initial state because some of these parameters are used to define the characteristics that will change during the lifetime of the MovingCharacter, while others are used to establish traits that will not change at all, being an intrinsic part of the character. These constant attributes do not depend on the context. Therefore, they are directly obtained from a file in the constructor. The file's name is provided as the name of the intended template character and its extension is `.mcharacter`. Its format follows the one that is defined by the class *Ogre::ConfigFile* and it defines such attributes as mass, maximum velocity, or maximum force. The attributes that define the state of the character, or that are just dependent on the context, are directly passed to the constructor[7].

The MovingCharacter class also implements the features of the steering layer. It

[7]for more information on the data that this type of file includes, refer to the class MovingCharacter on the API of the IViHumans platform. An example of this file can be found on Appendix A.1.

models a character that can move by producing forces to steer itself. These forces produce accelerations that are used to update the MovingCharacter's state. Steering behaviors determine forces that the character applies on himself, simulating how a self-powered entity uses energy to move by itself. The restrictions on velocity or applicable force are used as makeshifts for the effects of having limited energy to steer and of being vulnerable to friction and to air resistance.

The vehicle model is used so that the steering layer can operate independently of the actual vehicle it is driving. The abstract vehicle moves as an effect of forces. A steering behavior decides on a force for the character to apply on himself in each iteration, impelling him towards his goal, but simulating the production of these forces is up to the locomotion layer. This way, the operation of the steering layer is always the same, no matter what kind of entity is the subject. Thus, the point mass model can be used in conjunction with steering behaviors to move a man driving a car or just walking, although in practice some changes might be necessary.

In the IViHumans platform, a MovingCharacter can have one instance of the class *SteeringBehavior* managing its motion. This steering behavior can either be a basic one or what we called a *CombineBehavior*, which is actually a *composite* [34, 32, 38, 43] steering behavior (see Figure 3.13).

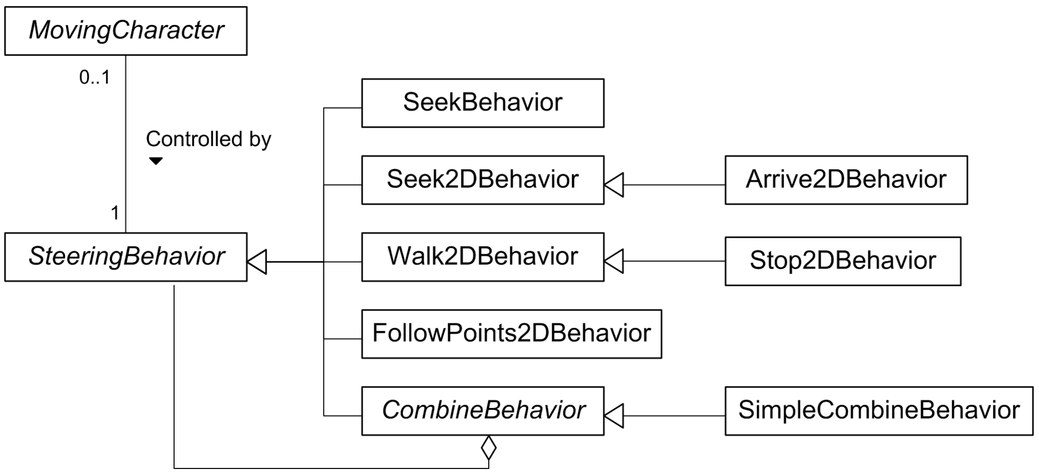


Figure 3.13: Class Diagram – Steering behaviors

We implemented six basic steering behaviors: *SeekBehavior*, *Seek2DBehavior*, *Arrive2DBehavior*, *Walk2DBehavior*, *Stop2DBehavior*, and *FollowPoints2DBehavior*. We decided to distinguish behaviors that operate in three dimensions from those that operate in two, by including the term "2D" in their name. Behaviors that operate in 3D are intended for characters that can move freely in the three dimensions of an environment (e.g. birds, that can fly in the sky; fish, that can swim in the sea). On the other hand, characters whose movement is constrained to one plane need behaviors that operate in 2D. In our approach, the *2D behaviors* still operate with vectors that are defined in three dimensions, but they are implemented so that they always return forces that have a null height component. This allows all behaviors to be represented under one common type. Also, it does not prevent characters from changing their height when necessary. Steering behaviors that operate in 2D never influence the height component of a character's position or velocity, because the character is assumed not to be able to vertically change his position on his own. However, the environment can do so, through the physics layer. For instance, if a VH was steering to reach some target that stood at whatever height and he found stairs on his way, his steering behavior would still return an horizontal force. So, if the velocity was already horizontal, the activity of the 2D steering behavior would still leave it so. But, on a second phase, external influences on the velocity of the character would have to be taken into account, as the physics layer performed its duties. The environment would change the character's position or velocity automatically and he would climb up the stairs.

The force that a SteeringBehavior instance applies on its owner is obtained by the MovingCharacter with the *template method* [34] *SteeringBehavior::getForce*[8], which is called in each update of the MovingCharacter's movement. This method checks whether the behavior is on or off. If it is off, a null force is returned; otherwise, the core computation is delegated on the protected method *SteeringBehavior::calcForce*[9]. This is a pure virtual method, that is, a virtual[10] method that is not defined in the base class. The classes that inherit from SteeringBehavior can then implement this method whichever way is appropriate, and that is how steering behaviors are distinguished. Besides computing a force to apply on the character, this method can also have *side effects*. For instance, some kinds of *SteeringBehavior* will remove themselves from the character once their job is complete.

Before being returned, the force is truncated to the maximum magnitude the character allows. The most elementary steering behavior we implemented is *SeekBehavior*. It drives the character in the direction of a target position, independently of the position of the character or of the obstacles that may interpose in its straight path. In its pure form, it leads the character to orbit around the target, even after the target has been reached, since it has no stop condition. This behavior is usually applied repeatedly to have a character walk through a series of points. In that case, the upper layer – the action selection layer – can assume the responsibility of updating the target once it has been reached. An alternative is to have a special behavior for that effect, as we did with *FollowPoints2DBehavior*, which is explained ahead.

Seek2DBehavior is a particularization of SeekBehavior that functions only in

[8]The declaration of this method is `virtual Ogre::Vector3 getForce();`

[9]The declaration is `virtual Ogre::Vector3 calcForce() = 0;`

[10]to allow polymorphic behavior [72, 23, 41]

two dimensions. The movement that this behavior generates is restricted to the horizontal plane that stands at the same height as the character. For that purpose, this steering behavior computes forces considering, not the original target, but its projection on the character's horizontal plane. Thus, it only yields forces that have a null height component.

As the previous one, *Arrive2DBehavior* operates on the horizontal plane of the character. For most of the time, this behavior does the same thing as Seek2DBehavior but, when the character approaches the target, the computation differs. So, we had Arrive2DBehavior as a subclass of Seek2DBehavior, in order to reuse the implementation of this last one. It also drives the character in the direction of the target, but it makes him slow down linearly as he approaches it. In the original *Arrive* behavior, the agent is subject to constant deceleration once he reaches a certain distance from the target, eventually stopping in a position that coincides with the target. In our version, the deceleration is applied only until the character's speed reaches some predetermined value, after what the speed is maintained. When the character comes very close to the target, as defined by another threshold, the behavior stops the character's movement and deactivates itself. In our opinion, this version produces more realistic results, in what concerns the movement of VHs. Their speed decreases when they approach the target but they stop at once, like real humans do with one last step. In the original version, characters tend to move at an unrealistically low speed when approaching the target, for too long a period, before coming to a halt, unless they decelerate too quickly, which is not desirable either.

We also implemented a steering behavior that drives the character at a given velocity and we called it *Walk2DBehavior*. This behavior has a target, like the previous ones, although this target is a velocity vector instead of a position. In each iteration, it calculates a force that will accelerate the character towards a velocity that can be obtained from the supplied target velocity as follows. Let $\overrightarrow{V}$ and $\overrightarrow{v}$ be the target and current velocities, respectively. The velocity that will actually be taken into account is $\overrightarrow{V_h} = (V_X, v_Y, V_Z)$, in a system where height is defined by the Y component[11]. The returned force can be viewed as $(\overrightarrow{V_h} - \overrightarrow{v}) \times 1Kg/s$, thus it will always have a null height component. So, this steering behavior will also maintain the character moving horizontally (if he was already moving horizontally). The velocity of the character eventually comes to match the projection of the target velocity. This steering behavior is a 2D version of one of the three behaviors that Reynolds created for his flocking boids. We find it very useful to have a user directly controlling an avatar with a keyboard or a joystick (instead of an action selection layer).

[11]This is the case of OGRE

A steering behavior to stop the character – *Stop2DBehavior* – was also created. This behavior is a particularization of Walk2DBehavior, but the target velocity is defined *a priori* as the null vector. When the character stops moving, that is, when his velocity drops bellow a very low threshold, the behavior deactivates and removes itself from the MovingCharacter. One may wonder why would this behavior also operate in two dimensions but we figured that, if a character only has the ability to steer in two dimensions, he would also be able to stop only his horizontal movement, since he can only affect his horizontal velocity. In other words, one character that can speed up by generating only horizontal forces should also be able to reduce only the horizontal components of his velocity. The vertical component of velocity could only be generated by external forces and this behavior will not influence it.

The last steering behavior we implemented was *FollowPoints2DBehavior*. It will drive the character across a sequence of n target positions and it will stop him once he comes to the last one. It does so by making the character *seek* the first $n-1$ targets and *arrive* at the n^{th} target. In fact, it aggregates instances of Seek2DBehavior and Arrive2DBehavior to accomplish that goal, but that is completely transparent to the user. The task this behavior performs can also be delegated to the action selection layer. This more abstract layer could also activate instances of the elementary behaviors Seek2DBehavior and Arrive2DBehavior successively but, when the path is known beforehand, the same effect can be achieved with only one command, which contributes to reduce communication overhead. Besides, monitoring and changing the behavior of the character is always possible.

Besides these basic steering behaviors, many more may be implemented in the future, simply by extending the abstract class *SteeringBehavior* and implementing the calcForce method appropriately.

Sometimes the movement of the MovingCharacter must be driven by different behaviors at once. For this purpose, further steering behaviors could be implemented to mix the operation of existing ones, but that would be cumbersome and very inflexible, requiring implementation, compilation and linking, each time a new mix was to be tried. The MovingCharacter could also simply be able to aggregate several steering behaviors directly, but then the issue would be how to combine them. The moving character would be left with one single way of combining SteeringBehaviors, but there are at least four ways [62, 15]:

- **Truncated Sum** – simply add the forces and truncate the resulting net force. This method is the most simple but it is costly since the contribution of every behavior is calculated in each time step. It is not the best option in most cases, because all behaviors are considered equally important.

- **Weighted Truncated Sum** – multiply the forces by a relative weight, before adding them, and truncate the resulting force. This method also requires the computation of every steering behavior and the weights can be difficult to tweak. Its biggest problem, however, happens with strange results produced by conflicting forces. This problem is also shared by the simple Truncated Sum, which is just a particular instance of the Weighted Truncated Sum method, when the total weight is evenly distributed among all active behaviors.

- **Weighted Truncated Running Sum with Prioritization** – sequentially add each force vector, truncating the result in each iteration to make sure the maximum force was not exceeded. Each behavior is prioritized and processed in the corresponding order and the forces can also be added with weighting. Before definitely adding the contribution of each vector, it is checked whether there is enough "remaining force" relatively to the maximum allowed magnitude. If there is, the new force is added and the process continues. Otherwise, if the maximum amount has been reached, the current net force is returned and the process ends. If there is enough space for only a portion of the last force, then only that portion is added. In time steps that involve large steering activity, the contribution of some behaviors is disregarded. This method gives a good compromise between speed and accuracy.

- **Prioritized Dithering** - besides a priority, each steering behavior is assigned a probability for evaluation. In each time step, it is checked whether the next highest priority behavior is going to be evaluated, according to the assigned probability. If it is, and if it returns a non-null force, that force is all that is returned. Otherwise, the behavior is skipped and the next highest priority behavior is considered. This method requires far less CPU time than others, but that comes at the cost of accuracy. It is also necessary to tweak the probabilities until the desired effect is reached.

In addition to these methods, other variations, or even new methods, may be suitable to combine steering behaviors, depending on the circumstances. In the IViHumans platform we considered this issue and chose an implementation that allows different methods for combining steering behaviors. For that, we created the abstract class *CombineBehavior* which can aggregate other steering behaviors and we left the method *CombineBehavior::calcForce* still undefined. This class has a series of features that are related to its purpose of aggregating steering behaviors, but the duty of actually combining them is left to descending classes, which can apply whatever technique by implementing the method calcForce. This way, different procedures for combining steering behaviors can be chosen at runtime, depending on application needs.

The class CombineBehavior has another pure virtual method, besides calcForce, called *CombineBehavior::prototypeClone*[12]. The purpose of this member is discussed ahead, when the need for it becomes apparent. The only subclass of CombineBehavior we created was *SimpleCombineBehavior*, which implements the most simple combination method – Truncated Sum. Other methods can be implemented at any time with no changes to existing code. The design of how an IViHuman calculates its steering force was inspired on the Strategy Pattern [34, 32].

The *RegularEntity*, the *IViHuman*, and their Locomotion

In an architecture that tries to deeply mimic reality, the locomotion layer would produce forces as a result of some intrinsic physical mechanism, as in [80]. Instead, the forces do not have a specific natural origin. In our case, the locomotion layer simply maps the state of the abstract vehicle into observable motion.

The class IViEntity represents a point in space. MovingCharacter completes the abstract vehicle model and groups the steering functionality for entities that can be represented by it. Both these classes are abstract, so they allow no direct instances to be created, and neither of them is associated with OGRE. The actual observable locomotion must be embodied in subclasses.

The class *RegularEntity* accomplishes this with a general and simple approach. It is intended for entities that "do not steer", that is, for whom the steering layer is absent. Therefore, it is a direct subclass of IViEntity (Figure 3.12). To provide a visual representation, it is intimately associated with an instance of *Ogre::Entity* and with an instance of *Ogre::SceneNode*. These are the main structures that enable an object to be rendered. The class Entity encapsulates the features and contents that are associated with the graphical representation of an object, such as meshes, materials, poses or animations. The class SceneNode represents a node in the scene graph and has an inherent local frame of reference that admits the usual transforms of translation, rotation, and scale. As the class RegularEntity extends the class IViEntity, it has a position and a facing vector that are managed by that class, besides the position and orientation of its SceneNode.

The state of an IViEntity is updated with the method *IViEntity::updateMovement*[13]. In the base class, this method updates the attributes that define the IViEntity's state, such as position and velocity, according to classical physics, as explained earlier. In the class RegularEntity, this method is overridden to update the state of the scene node, so that its attributes coincide with the attributes of the IViEntity it also is. Of course, the updates that are already available in the parent class are also performed. So, the state of the RegularEntity is always consistent with the state of its SceneNode.

[12] `virtual CombineBehavior * prototypeClone() const = 0;`

[13] `virtual bool updateMovement(Ogre::Real timeSinceLastUpdate);`

For the RegularEntity to always face the direction that is specified by its facing vector, the orientation of the scene node must be set accordingly. This vector can change as an effect of an external order or because the auto facing mode is on. Either way, since it is only a vector, it is not enough to completely specify an orientation. That would require another degree of freedom, namely the one that would specify the angle of rotation around the local axis that the facing vector, itself, defines. For flying entities, further heuristics would have to be used, as Reynolds points out [62]. However, given the previous orientation of the entity, and assuming that the *roll*[14] component of the orientation should remain the same, the scene node can be transformed by the smallest rotation that would transform the previous facing vector in the new one. This is enough to approximate the movement of most 2D constrained entities, but requires the initial facing vector to be properly defined, so that it is aligned with the facing direction that the observer actually sees. For RegularEntities, an initial rotation transform may be provided in the constructor to account for this[15].

The minimum rotation can be easily found, except when the new vector points in the exact opposite direction of the old one. When the old and new vectors are the same, no rotation needs to be applied whatsoever, but when they point in the opposite direction, there is an infinity of planes that contain them both, therefore, there are infinite rotations with the same magnitude that would transform the old one in the new one. This may seem like a really rare case, especially if entities are commanded by steering behaviors and updated frequently, because, in this case, their velocity and, thus, their facing direction, should change very little in each time step. But there are some circumstances that happen often enough and that give rise to this issue. For instance, suppose that an entity, commanded by steering behaviors, is still, facing the direction of the positive X-axis. Suppose that, afterwards, the seek behavior is activated for the target $(X, 0, 0), X \in \mathbb{R}$. The character would reach the target and, eventually, surpass it, at which time the force returned by the behavior would have the opposite direction of movement. The character would have entered a particular straight orbit around the target position and, in some time steps, his movement would invert its direction suddenly. In the case of entities that move on 2D and whose roll component is not supposed to change, a 180^{o} degree rotation around the up axis can be applied and, for now, this is what happens in the IViHumans platform. We are aware that this is not the best solution because sudden rotations like this do not happen in the real world and it compromises the intended sensation of believability. The investigation of how to overcome this problem is

[14]Rotation about the depth axis

[15]Actually, a scale transform may also be specified. With this option, *the modeler* does not have to worry with the right proportions or orientations of the objects *he* models. In the case of VHs, initial transforms can be specified in a configuration file that is discussed ahead.

suggested as future work. In the case of flying entities, the roll component should not be assumed as null, as they would incline to make turns and, as mentioned above, another method should be used.

The movement of a RegularEntity is simply translated to the change of its position and orientation. In the case of a VH, he also needs a means of locomotion to look convincing. Real humans move by contracting and distending muscles, executing complex movements that make them walk, run, climb, crawl, etc. In the case of our VHs, instead of deeply simulating these complex processes, synchronized animations are played in a way that looks like they are actually pushing themselves forward, which gives a pretty fair appearance of realism while significantly reducing the use of computational resources. Unfortunately, the VHs we use have very few animations, so they are restricted to only about a couple of movements. In fact, our virtual woman can only walk, while the virtual man can walk and run (besides staying in the same place with a consistent animation).

The VHs are represented, in the IViHumans platform, by the class *IViHuman*, which descends from MovingCharacter, inheriting steering abilities. The transforms that are applied to the bones of the VHs we use consist only on operations of rotation, even in the special case of the root of the skeleton – in both models, the hip bone. Thus, the animations preserve the original position of the entity they apply to. On the other hand, the orientation of the whole skeleton is also approximately maintained, even though small rotations of the hip bones take place in the animations. This way, the attributes of an IViHuman are always consistent with what is actually observed and, indeed, to change its position and orientation, its SceneNode has to be transformed.

Along with the transformations of their scene nodes, the animations of the VHs drive their motion. To play the animations, a specialized object – of the type *Ogre::AnimationState* – receives indications in each update that tell it the instant it should use to derive the current transforms for the skeleton. For instance, suppose that some simple animation lasts one second and that it relies on two keyframes: one for the instant $0s$ and another for the instant $1s$. Suppose that the first keyframe encodes a null transform and that the second encodes a 90° rotation of the root bone, about some arbitrary axis. If right before rendering a particular frame it is known that $\frac{1}{3}s$ elapsed since the beginning of the animation, the responsible AnimationState instance can be commanded to put the animation in the instant $\frac{1}{3}s$ and it will ensure that the right measures are taken to deform the skeleton accordingly, that is, to rotate the hip bone, and therefore the whole body, by 30° relatively to its original orientation.

The vast majority of the updates that need to be performed over the virtual environment are a function of the time elapsed since the previous updates and the methods that are responsible for the individual updates receive this information through a parameter. In the case of the class IViHuman, the elapsed time is added to the current time registered by the AnimationState object that controls some animation. The repetition of this process in every frame or, at least, in most of the frames, allows animations to be played at their original speed. However, playing an animation at its original speed while the character moves is not enough to get good results, unless the character is restricted to move at a single constant speed that matches the speed of the animation. In the IViHumans platform we wanted every character to be able to move at any speed comprised in the interval whose upper limit is his maximum velocity and whose lower limit is zero velocity (when the character is still). Therefore, it was needful that the animations could be played at variable speeds on the basis of the speed of the VH.

When a real human walks naturally, his instantaneous speed varies little in distinct phases of his steps. The horizontal speed at which a person's center of mass moves when he puts the foot on the ground is approximately the same as when the same foot is in middle air. Similarly, for each speed of the scene node the VH is attached to, a speed for the animation can be found that does not compromise believability. For instance, take the walk animation of the female VH and let D_o be her displacement after walking for T_o time at the natural speed of the animation, where T_o is the time of one full cycle of the animation. Obviously, the average speed of the virtual woman was $V_o = \frac{D_o}{T_o}$. This is the speed that maximizes the synchronization between the translation of the IViHuman's scene node and the continued playing of the animation at its original speed. With this relationship in mind, the speed at which the animation is played can be adjusted in order to have it synchronized with the VH's movement. The time it takes to play some portion of the animation is inversely proportional to the character's speed. For any speed V_x of the VH, we have

$$V_x = \frac{V_x}{V_o} V_o = \frac{V_x}{V_o}\frac{D_o}{T_o} = \frac{1}{\frac{V_o}{V_x}}\frac{D_o}{T_o} = \frac{D_o}{\frac{V_o}{V_x}T_o}$$

On the other hand, we know that there is an amount of time T_o' for which the equality $V_x = \frac{D_0}{T_o'}$ holds. T_0 is exactly the time that the animation should take to complete one full cycle when the VH is moving at a speed of V_x. From these facts, we can extract the equation $\frac{1}{\frac{V_o}{V_x}}\frac{D_o}{T_o} = \frac{D_o}{T_o'}$. Since the problem we are trying to solve is about animations that are played when the VH is displaced, we know that $D_o \neq 0$. Therefore, we obtain

$$\left(\frac{1}{\frac{V_o}{V_x}}\frac{D_o}{T_o}\right)^{-1} = \left(\frac{D_o}{T_o'}\right)^{-1} \Longleftrightarrow T_o' = \frac{V_o}{V_x}T_o$$

So, to update the movement of a VH, the animation should be forwarded an amount of time given by the function $f(t, v) = \frac{V_o}{v}t$, where t is the time that actually went by since the last update and v is the current speed of the VH. We call the constant V_o *Translation to Animation time Ratio*, or simply *T2ARatio*.

Figure 3.14 shows that the class IViHuman delegates to the class *TranslateAnimController* the responsibility of choosing and updating the animations that enable the translation of the VHs. An instance of this class chooses the right animation for the IViHuman that owns it according to specified rules.

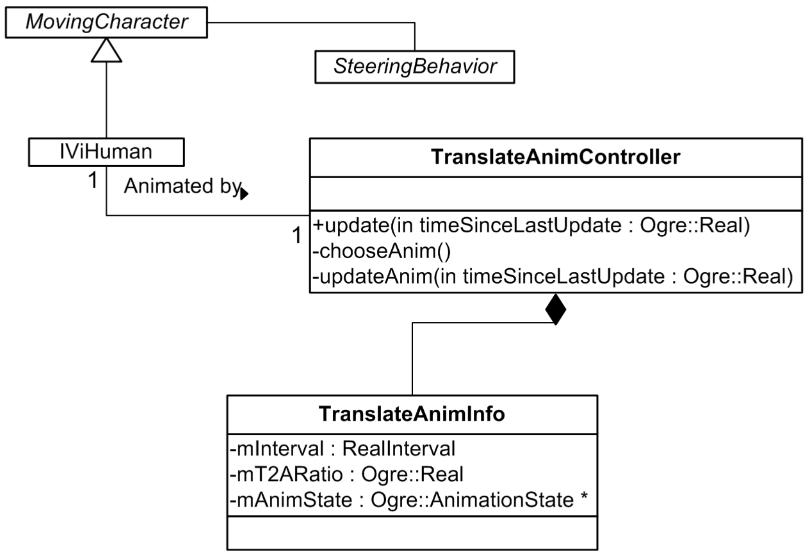

MovingCharacter
SteeringBehavior
IViHuman
1
Animated by
1
TranslateAnimController
+update(in timeSinceLastUpdate : Ogre::Real)
-chooseAnim()
-updateAnim(in timeSinceLastUpdate : Ogre::Real)
TranslateAnimInfo
-mInterval : RealInterval
-mT2ARatio : Ogre::Real
-mAnimState : Ogre::AnimationState *

Figure 3.14: Class Diagram – *TranslateAnimController*

TranslateAnimController is composed by instances of TranslateAnimInfo. There is one and only one TranslateAnimInfo object for each translation animation. This class is characterized by the attributes *mInterval*, *mT2ARatio* and *mAnimState*, as shown in the figure. The animation to be played is chosen according to its current speed. Each animation has a corresponding range of speed that is expressed in the attribute mInterval. If the VH is moving with a speed that belongs in this interval, this is the animation to be played. mT2ARatio specifies the Translation to Animation time Ratio and mAnimState is a pointer to the AnimationState object that controls the animation. When an IViHuman is commanded to update its movement, it forwards the animation update to its TranslateAnimController. This object will first choose the correct animation and then update it, both operations performed on the basis of the speed of its owner. It checks in its TranlateAnimInfo objects for the first animation whose speed interval includes the speed of the VH and, if one is found, it updates the animation by putting it in the correct instant, through the AnimationState object, according to what was described earlier. The period of time the AnimationState object is told that elapsed since the last update is as much larger (or smaller) as the speed is smaller (or larger, respectively). For instance, if the character A moves at twice the speed of character B, the AnimationState of B is given twice the amount of the elapsed time as that of character A (as long as they both play the same animation).

If the right animation is not found, no animation is played. Whenever a new animation is selected, it is advanced a random amount of time so that no synchronized animations are seen. Otherwise, when different instances of the same VH are in a similar state, their animations would appear synchronized (e.g. when all the VHs are told to walk at the same time).

As for MovingCharacter, instances of IViHuman are configured according to some characteristics that are not supposed to vary at runtime and that are therefore obtained from a file. The file that contains these specifications, that apply only to VHs and not to all characters, has the extension `.ivihuman`[16]. The rules for *translation animations* should be included in this file, one rule per animation. These rules follow the format `<anim_name>=<x> <y> <z>`. `<anim_name>` should be substituted by the actual name of the animation. `<x>` and `<y>` specify the speed interval for the animation: $(x, y], x, y \in \mathbb{R} \wedge x < y$. Finally, `<z>` specifies the value T2ARatio for the corresponding animation.

The `.ivihuman` file can also contain initial transforms to be applied to the VH before rendering the first frame. The need for two further parameters – *eyeHeight* and *headBode* – is discussed ahead, in Section 3.3.3.

3.3.2 Expressions

In the IViHumans platform, a VH has the capability of showing emotions through facial expressions. As explained before, the facial expressions are conceived as poses that encapsulate a set of displacement vectors. When the translations these vectors declare are applied to the corresponding vertices of the mesh, the character assumes the intended facial expression.

Each VH should have a set of basic facial expressions that, when mixed together, originate composite expressions. The conceptual distinction between basic and composite expressions is essential for the design of the fraction of the platform that implements the features required for emotional expression. As the name indicates, composite expressions are composed of other expressions. However, from the point of view of the observer, the effects of activating basic or complex expressions are identical: both originate changes on the shape of the face of a VH. In fact, even for a programmer that deals with expression objects, their use should be identical in most aspects.

The class IViHuman, which is a client of the class *Expression* must deal with basic and complex expressions uniformly, needing only to know that they are expressions. Nevertheless, some distinctions obviously exist. The process of creating

[16]For a detailed description of this file, refer to the class IViHuman, in the API of the IViHumans platform. An example of this file can be found on the appendix A.2.

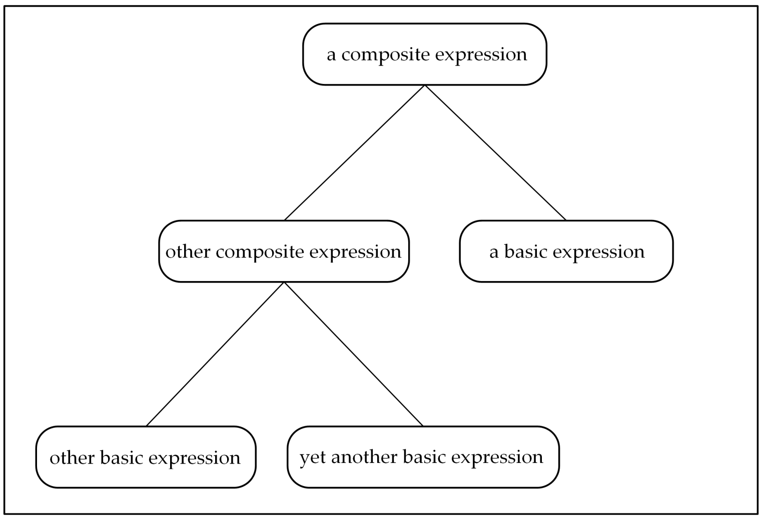


Figure 3.15: Example of a tree of expressions.

expressions is different for these two types, as a basic expression does not need to know or interact with any other expression, while a *composite* expression is created, as the name suggests, by composition of other expressions. Besides, the type of expressions that a composite expression can aggregate is not restricted to basic expressions. The elaboration of composite expressions by combining other composite expressions is perfectly plausible, as long as no cycles are formed. As a matter of fact, the constitution of a composite expression can actually be represented by a tree structure, as Figure 3.15 exemplifies. Thus, our implementation follows the Composite Pattern[17], as shown in Figure 3.16. This figure also includes some important parameters and methods of the involved classes, though less relevant information is omitted.

For clarity sake of the arguments exposed here, we assume that the common interface of the composite pattern receives the name *component*, that the basic components are called *leaf* and that the components that aggregate other components are called composite[18]. In the present case, these roles are played by the classes *Expression*, *BasicExpression* and *CompositeExpression*, respectively.

As Figure 3.16 shows, all the expressions have a common interface, except for the fact that CompositeExpression has also got operations for managing its children (the main ones are *CompositeExpression::addChildExpression and CompositeExpression::getChildExpression*). To aggregate other expressions, CompositeExpression has an attribute that maps Expression objects by their name. These methods add expressions to and obtain them from this map. For a very short discussion on two

[17]Assuring that no cycles exist is a responsibility of the client programmer. This can be viewed as a drawback of the composite pattern, which does not prevent cycles but does not support them either.

[18]This naming convention is withdraw from the *GoF* (*Gang of Four*) book [34]

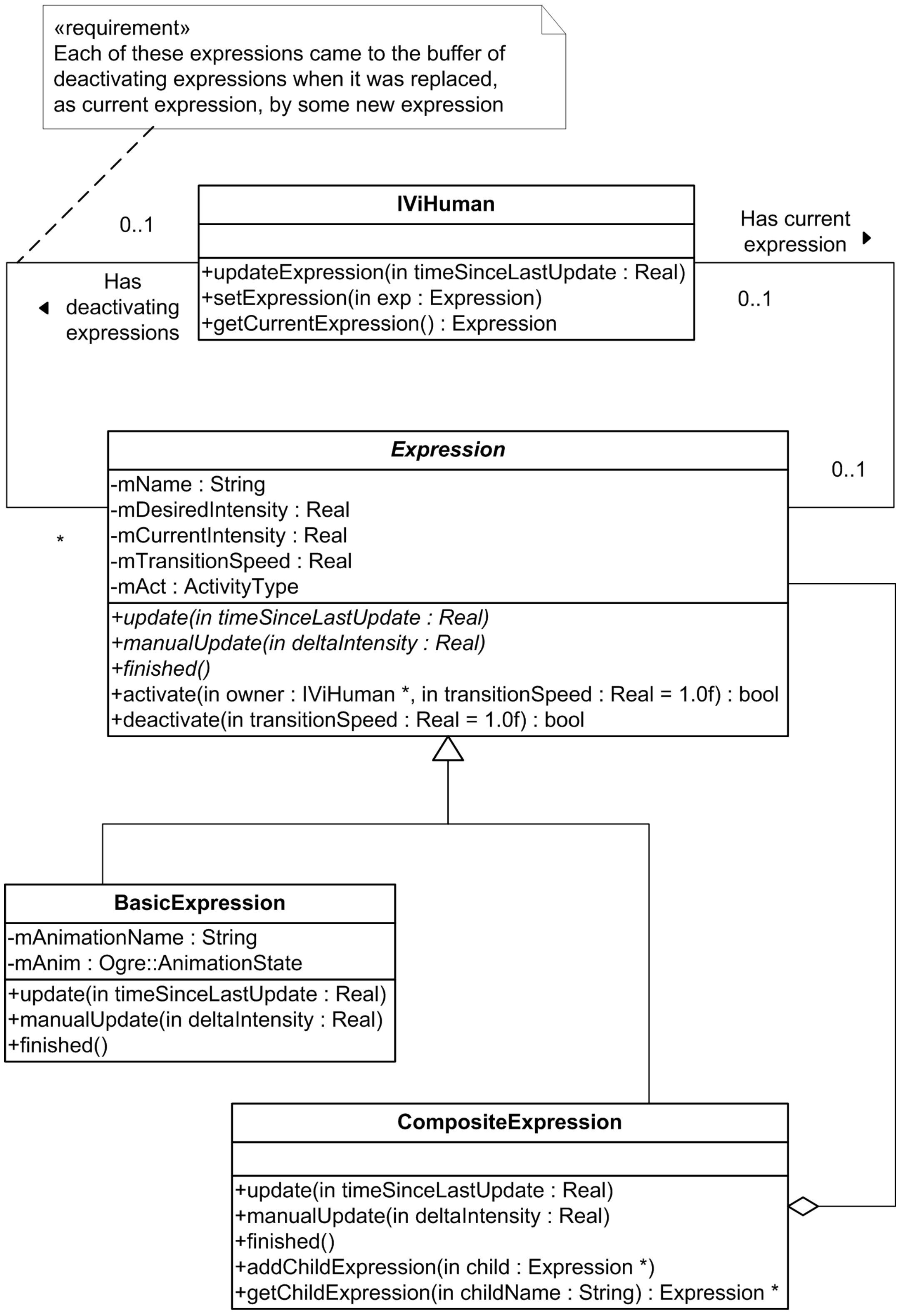

«requirement»
Each of these expressions came to the buffer of deactivating expressions when it was replaced, as current expression, by some new expression
IViHuman
+updateExpression(in timeSinceLastUpdate : Real)
+setExpression(in exp : Expression)
+getCurrentExpression() : Expression
0..1
Has deactivating expressions
*
Has current expression
0..1
0..1
Expression
-mName : String
-mDesiredIntensity : Real
-mCurrentIntensity : Real
-mTransitionSpeed : Real
-mAct : ActivityType
+update(in timeSinceLastUpdate : Real)
+manualUpdate(in deltaIntensity : Real)
+finished()
+activate(in owner : IViHuman *, in transitionSpeed : Real = 1.0f) : bool
+deactivate(in transitionSpeed : Real = 1.0f) : bool
BasicExpression
-mAnimationName : String
-mAnim : Ogre::AnimationState
+update(in timeSinceLastUpdate : Real)
+manualUpdate(in deltaIntensity : Real)
+finished()
CompositeExpression
+update(in timeSinceLastUpdate : Real)
+manualUpdate(in deltaIntensity : Real)
+finished()
+addChildExpression(in child : Expression *)
+getChildExpression(in childName : String) : Expression *

Figure 3.16: Class Diagram – Expressions

variants of the Composite Design Pattern, in what concerns the management of components, as well as for a justification of our option, see Appendix B.

An important attribute of an expression is its intensity. The class Expression declares an attribute that specifies the current intensity of its instances – *mCurrentIntensity* – and another that records the value that they should ultimately achieve – *mDesiredIntensity*. As previously mentioned, an expression can be partially applied to a VH's face. Here, the intensity of an expression is seen as a value that represents the relative amount of application to the VH's face: a real number that should be comprised in the interval $[0, 1]$. Put another way, it is the percentage of activation of an expression. In the class Expression, the current intensity is relative to the maximum intensity and not to the desired intensity, so it is considered that the expression has reached the desired intensity when `mCurrentIntensity == mDesiredIntensity`, instead of `mCurrentIntensity == 1.0f` being the condition.

Recall that expressions are recorded as poses of the mesh and let a be some arbitrary value in the specified range ($a \in [0, 1]$). Suppose that $M = \{\overrightarrow{v_i} \mid i \in \mathbb{N} \wedge i \leq N\}$ is the set of vertices of the mesh, where N is the number of vertices. Suppose also that $P = \{\overrightarrow{d_i} \mid i \in \mathbb{N} \wedge i \leq N\}$ is the set of displacement vectors that codifies one pose for that mesh. When the mesh is deformed by the pose, each displacement vector $\overrightarrow{d_i}$ comprised in the pose, is used to displace the corresponding vertex $\overrightarrow{v_i}$[19]. If the pose is applied with an intensity a, the set of vertices that compose the mesh becomes $M' = \{\overrightarrow{v_i'} \mid \overrightarrow{v_i'} = \overrightarrow{v_i} + a\overrightarrow{d_i}\}$, that is, each vertex becomes $\overrightarrow{v_i'} = (v_{ix} + ad_{ix}, v_{iy} + ad_{iy})$.

For instance, suppose that some polygon in 2D space (instead of a 3D mesh, for simplicity sake) was composed by the vertices

$$\overrightarrow{v_1} = (-5, -5); \quad \overrightarrow{v_2} = (-5, 5); \quad \overrightarrow{v_3} = (5, -5); \quad \overrightarrow{v_4} = (5, 5).$$

Suppose also that some "2D pose" for that polygon comprised the displacement vectors

$$\overrightarrow{d_1} = (-2, -2); \quad \overrightarrow{d_2} = (-2, 2); \quad \overrightarrow{d_3} = (2, -2); \quad \overrightarrow{d_4} = (2, 2),$$

and that each displacement $\overrightarrow{d_i}$ was applied to the vertex $\overrightarrow{v_i}$. If this pose was to be applied with an intensity of 100%, the polygon's vertices would end up being

$$\overrightarrow{v_1'} = (-7, -7); \quad \overrightarrow{v_2'} = (-7, 7); \quad \overrightarrow{v_3'} = (7, -7); \quad \overrightarrow{v_4'} = (7, 7).$$

If, on the other hand, the pose was applied with an intensity $a = 0.5$ (50%), the vertices would become

$$\overrightarrow{v_1''} = (-6, -6); \quad \overrightarrow{v_2''} = (-6, 6); \quad \overrightarrow{v_3''} = (6, -6); \quad \overrightarrow{v_4''} = (6, 6).$$

[19]Usually a pose does not change all the vertices of a mesh. In this discussion we assume that each vertex is displaced by some vector, even if it is the null vector. In practice, the poses only record non-null vectors, along with data that enables the identification of the vertices they apply to

By constructing an Expression object with the adequate desired intensity, we can have a VH showing weak or strong emotions. For instance, he can be very joyful or somewhat surprised.

The transition of a VH's expressions should be gradual, so that no sudden changes on his face compromise believability. To achieve this, an Expression object goes through various states of activity, in what regards the variation of its intensity. During its lifetime, whether the intensity of an expression increases, decreases, or stays the same is controlled by a fundamental attribute that states its state of activity: mAct. This attribute's type – *ActivityType* – enumerates five values: `INACT`, `ACT`, `SKIP`, `DEACT`, and `DONE`. When the value of the attribute mAct is INACT, the expression is in the inactive state. At this moment, though the object was already created, it was not associated with any VH yet. The object symbolizes only an abstract expression and it has zero intensity. When the expression is activated, with the method *Expression::activate*, it goes to the state ACT, being necessarily associated with an IViHuman. When in this state, its intensity is increased in each call to the method *Expression::update*. The expression remains in this state until its intensity reaches the intended value or until it is explicitly commanded to change to the state DEACT. If the desired intensity is reached, the state of the expression becomes SKIP. While at this state, the intended expression is fully applied to its owner and its intensity is maintained. The expression eventually changes its state to DEACT, when *Expression::deactivate* is called. At this state, its behavior is symmetrical to that of state ACT. Its intensity is decreased in each update, until it becomes zero again, or until an activation order is issued again. When the intensity decreases to zero, the expression is put in the state DONE. At this time, its job is fulfilled and it can be dissociated from the VH. That happens in the following update and the expression's state becomes INACT again. If desired for any reason, it can be used again with any VH that has the pose(s) the expression relies upon. Figure 3.17 depicts the life cycle of Expression objects with a Finite State Machine (FSM), whilst Figure 3.18 shows how the intensity of an expression varies on the basis of its activity state.

As shown in Figure 3.16, the class IViHuman relates to the class Expression through two distinct associations: one for the most recent expression (the current expression of the VH) and one for the set of expressions under deactivation. When the method *IViHumans::updateExpression* is called, the IViHuman is held responsible for the update of all his expressions. That responsibility is then delegated by him on the Expression objects themselves, through their update method, and the updates occur in accordance with what was just explained.

The association between the VH and his current expression is the one that plays the central role. This association is established when the expression is activated.

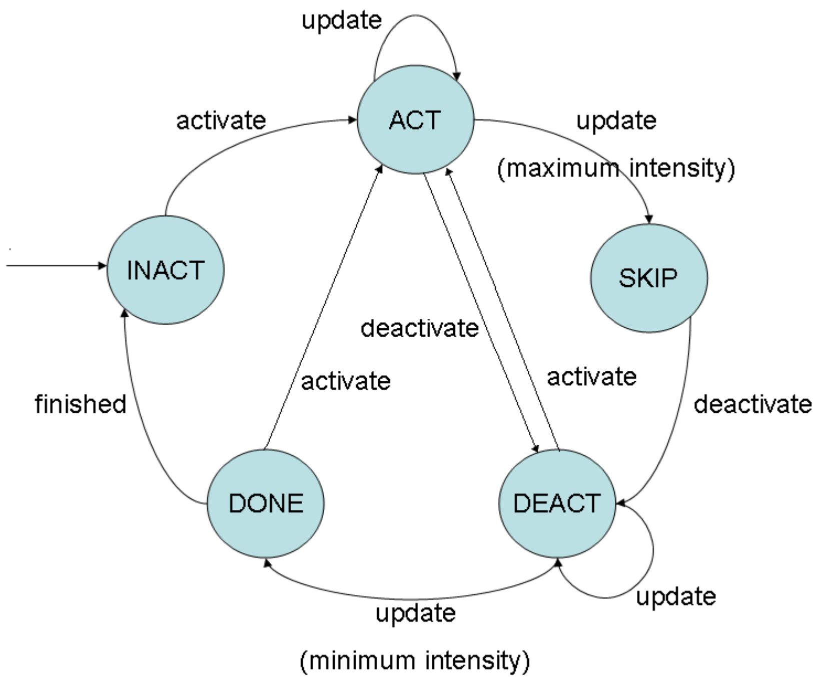


Figure 3.17: FSM for the Activity of an Expression

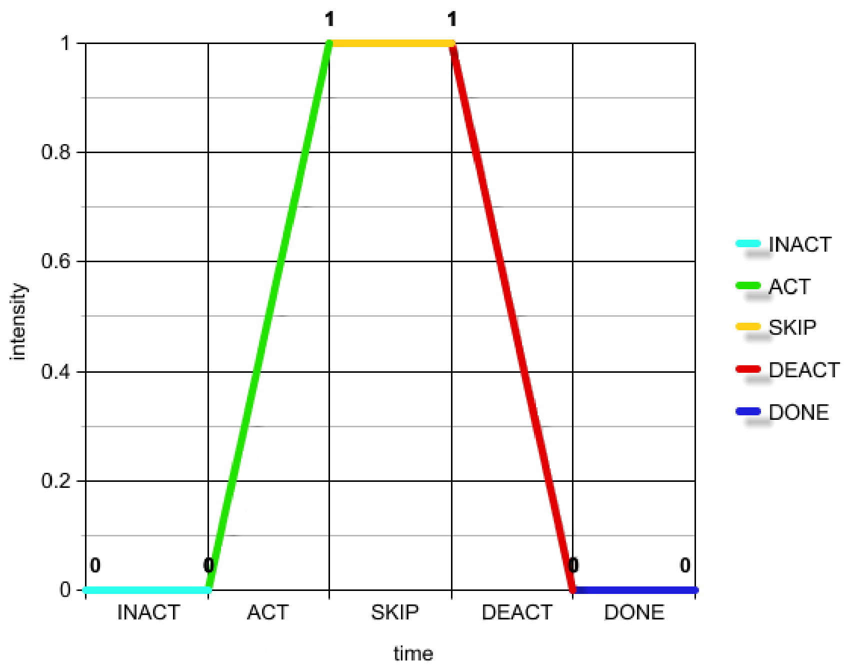


Figure 3.18: Graph of the intensity of an expression, with respect to time, in its diverse states of activity

While the association lasts, the expression is in the state ACT, to begin with, and SKIP, following (if the desired intensity is reached before deactivation). This first association ceases, being replaced by the second association, when an order is issued for the expression to deactivate. This order may be given explicitly to the IViHuman or placed by himself when another expression is set as the current one.

When an expression is deactivated, it is still related to the same IViHuman. In fact, from the point of view of the Expression object, no change occurs with the association, as it sees it only as being tracked by an attribute that points to its *owner*. Yet, from the point of view of the IViHuman object, the association is replaced by a new one that withstands multiple deactivating expressions. This second association is implemented with what may be seen as a buffer of deactivating Expressions, which is kept by the IViHuman. So, when the IViHuman commands an expression to deactivate, it puts it in the buffer of deactivating expressions, simultaneously, and that is when the Expression goes from the state SKIP (or act, in case its activation was not completed) to the state DEACT. When updating his facial expression, the IViHuman issues the update command, not only to the current expression, but also to those that are still deactivating. He also checks, from these, which have achieved the state DONE, and terminates any such associations, notifying the expressions and removing them from the buffer. This is when the expressions' states become INACT once again.

Figure 3.19 shows the variation of three composite expressions that received the names *combo1_100*, *combo2_100*, and *combo3_75*. The desired intensity for the first couple is 1 (or 100%) while, for the third, it is 0.75. The data used for the creation of this graph was obtained from a small program we implemented for that purpose. The program uses our framework and the assignement of an expression as the current one is triggered by a simple keyboard event – pressing a single determined key. The instants in which the key is pressed are also those in which an active expression is deactivated. They can be identified in the graph as the moments when the intensity of one expression starts to increase, along with the decrease of the intensity of an expression that has just been put in the deactivating buffer.

How an Expression is internally translated into a deformation of the mesh depends on its ultimate type, that is, on whether it is a BasicExpression or a CompositeExpression. Whilst BasicExpressions can be directly translated into an effective expression, the application of CompositeExpressions depends on the translation of the leafs that, at the bottom of the tree, compose it.

To materialize a visible expression, each BasicExpression relies on a simple animation that gradually intensifies the corresponding pose. The animation lasts 1s and is composed by only two keyframes: one at the instant 0s, in which the face has its default appearance, with no pose applied, and one at the instant 1s, in which

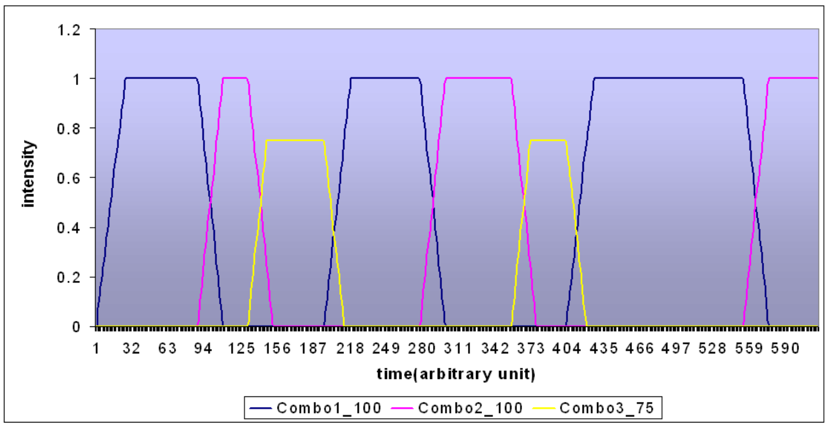


Figure 3.19: Variation of intensity of three expressions with time

the pose is applied at 100% of its intensity. If this animation was to be played without any modification, the face of the VH would show 30% of the corresponding expression in a frame that was rendered at the instant $0.3s$, 50% at $0.5s$ and so on. Generally speaking, if one intends to show an expression with $\frac{1}{x}$ of its maximum intensity, it sufices to render the scene as if the animation was at the instant $\frac{1}{x}$. As we already saw, this can be very easily performed through the AnimationState object of each animation.

When an Expression instance receives an order of activation or deactivation, it can also receive a parameter that indicates the speed desired for the transition, as a factor to multiply by the original speed. This parameter is not required, as it has the default value of 1.0. Let I be the desired intensity for an expression. The transition between 0 intensity and the intensity I then lasts Is. If the supplied factor is c, the transition will be c times faster or, put another way, it will last $\frac{I}{c}s$. If the time that elapses between two consecutive calls to the update method of an Expression is Δt, that Expression will forward the animation by $c\Delta t$.

CompositeExpressions are not directly associated with any pose or animation. Instead, when their update method is called, they figure out the proper intensity variation for each direct component Expression as a function of the intended intensity, of the elapsed time since the last update and of the factor to apply to the default transition speed. The responsibility of the update is then passed on to the component Expressions, through the method *Expression::manualUpdate*, that directly receives the variation of intensity as a parameter.

Besides being a simpler way of manipulating an expression, the idea of wrapping it into an animation has also got the advantage of encapsulating any deformation that may represent an expression. This way, it is very easy to deal with more complex expressions. For instance, there is no need for any change in the code, for an expression that is built by two poses to be displayed. An expression could also be setup by deforming a skeleton, a lattice or any kind of auxiliary object whose shape can be referenced at a keyframe in any way OGRE supports. Thus, certain kinds of body expressions can also be used with the existing framework. For instance, if we want some VH to express fear by also rising his hands a bit, that posture would only have to be modeled and referenced in the last frame of the animation that corresponds to that emotion.

3.3.3 Perception

If one wants VHs to be autonomous, they have to include reactive and deliberative behavior. Reactive behavior happens when VHs engage in predetermined activities that are triggered by specific world states or events. Deliberative behavior, on the other hand, is much more complex, involving reasoning that is dependent on the mental state of the agent, which is influenced and also influences his internal conception of the world in a feedback cycle that builds up along time. Any actual attitudes involved in both these kinds of behavior rely on the simulation of external stimuli that are given to the VH as input. In some cases, internal stimuli are also determinant to the agents conducts, specially in deliberative behavior.

A proper perception model is key to simulating VHs' intelligence, and *an agent's awareness of its environment should be consistent with his embodiment* [15]. A character that has ears should behave as if he was hearing, one with eyes should behave like he was seeing, and so on. In the case of VHs, and in the limit, they should have all the senses that a real human has, since real humans are exactly what they try to simulate. However, depending on the purpose of the simulation, some senses are much more important than others. For instance, the absence of the sense of taste would probably have no effect on an application in which VHs did not eat or drink; the sense of smell would probably not be missed if no extraordinary scent sources were present in the scene. In our case, and due to the need of establishing priorities among all desired features of the IViHumans platform, we chose to include vision as the primary perception mechanism for VHs.

The inclusion of simulation models for other senses is suggested as future work. It is our opinion that the second most important sense for the majority of simulation applications would be hearing. The tactile sense, is obviously very important but it would be already simulated in its most important implications by collision handling and by intrinsic abilities of VHs, so we consider it as a special case. Some complex model would probably have to be created if VHs were to distinguish objects only by their touch, but the mere perception of whether or not they collided with something and of whether or not they successfully grabbed a tool, may be directly included in their simulated abilities simply by providing them with knowledge of the external events they are involved in through contact. In what concerns the awareness that comes from these effects of the tactile sense, which we regard as the most important ones, a particular model of touch can be simply skipped.

The same is not certainly true for other senses. With the sense of touch, the main manifestation can be viewed as a simple boolean event (whether or not anything was touched), maybe with some further information, such as heat. Contrastingly, with taste, evaluation of other characteristics of what was being tasted would have to be performed, so that the inseparable qualitative appreciation could take place. With the remaining senses, an even more prominent issue must be dealt with: which events should and should not be perceived.

If with touch and taste we can simply state that events are perceived by a real human if he is in contact with the respective objects, that is certainly not the case for other senses. Vision, hearing, and smell are the result of specialized sensitivity to particular effects of certain events that may happen far away from who notices them, but they are all subject to physical obstacles or to medium characteristics that can prevent the perception of these events. For instance, we might or might not smell a distant fire depending on the direction of the wind; we might or might not see the formation of a supernova depending on how cloudy was the sky when its light arrived at Earth; and we might or might not hear a whale call, depending on whether we were inside or outside the water and on whether there was any motorboat around. As vision was the only sense that was already included in the IViHumans platform, it is the only one that will be discussed from now on.

Synthetic Vision

Some approaches to games and simulations implement virtual agents with unrestricted access to the world data, bestowing them with *sensory omnipotence* [15]. These approaches incur in a flaw that originates unnatural events in the behavior of the intelligent agents. Their actions demonstrate they have knowledge they should not have, like being aware of objects in different rooms. This is not a good approach when believability is sought. It is essential to restrain from the agents the information they should not be capable of accessing, otherwise it will be apparent that the AI is *cheating*.

In other approaches, classified as *classical* by Conde et al. [18], *different [steering] behaviors implement their own perception mechanisms*. In the IViHumans platform, VHs sense the environment through the single means of synthetic vision and the perceptions generated by this centralized approach can be used to whatever purposes. The VHs watch the world with an algorithm that was devised in previous works [67, 68]. It is not the goal of this algorithm to simulate the complexity of real visual systems. Instead, it accomplishes a means of filtering out the information that cannot be perceived by the VHs. It works by casting rays from the point of view of the VH and by checking which objects are intersected by them, to detect which are visible to the agent. These rays are calculated in such a way that their concentration is greater in the center of the Field of View (FoV). They are also displaced in each iteration of the vision algorithm, to minimize the chances of undetected objects over a series of iterations. On their turn, the displacement vectors that are applied to the rays also change in each iteration, in such a way that each ray describes a spiral path. Figure 3.20 shows an example of such a spiral, yielded by our implementation of the algorithm.

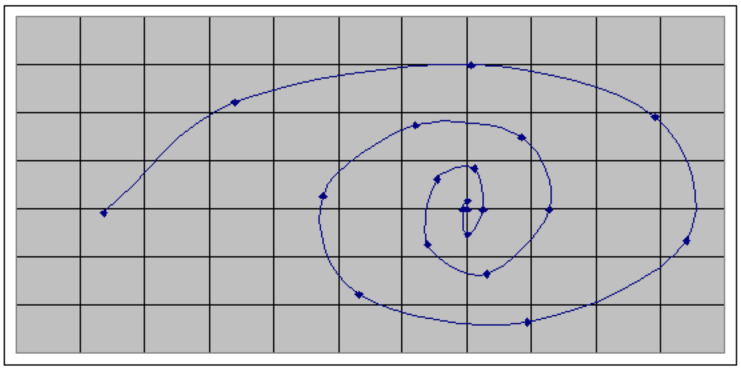


Figure 3.20: A spiral formed by successive points that represent the displacement vectors

A demonstration of the vision algorithm had already been implemented. It was implemented directly over OGRE, however, as the GP layer was yet to be built, and we had to implement it again from scratch to fit it into the platform. We also had an idea for an alternative vision algorithm that we did not have the opportunity to explore. In the vision algorithm we currently use, rays are sent to all over the FoV of the VH, regardless of the disposition of objects relatively to it. An object is considered as being seen if it is the first to be intersected by at least one ray, that is, if there is at least one ray whose intersection with the object is closest to the VH than any other intersection with the same ray. Our idea was to concentrate the rays in the areas where objects are known to be. In the IViHumans platform, VHs already have a camera that can render the scene from their point of view, so that it can be shown to the user. This camera is also used to derive the direction of rays, as explained ahead. In the alternative we imagined, intersections of world objects with the camera's frustum would first be checked for. A parameterizable number of rays would then be cast only to the objects that were found to be, at least partially, within the FoV. For each object, the faces that were facing the camera could be found the traditional way, by checking whether the angle between their normals and the camera's normal were acute. Rays would then be cast only to these faces. We speculate that this procedure could yield a more efficient and precise algorithm. This idea was not explored in any way, though, and a study of its viability would be required. It should also be compared thoroughly with the existing method, to check if it would be advantageous. This is suggested as future work and, if it was concluded that this idea could be profitable, it could be deepened and implemented.

To position the camera that is associated with each VH, a parameter is obtained from the correspondent `.ivihuman` file: `eyeHeight`. As the name suggests, this parameter indicates the height at which the eyes of the VH stand, relatively to the base of his feet (the vertex with the lowest height component). After positioning the camera, it is rotated in order to have it facing the same direction as the VH. Only then is it attached to the head bone of the VH, whose name is also specified in its configuration file by the parameter `headBone`. The camera inherits all the transforms of the head bone, thus we ensure that the camera always captures what the VH would see.

To create the rays that are to be cast, a matrix of 2D points is first computed. In each iteration of the algorithm a displacement vector is added to all the points. The resulting points indicate the positions in the viewport that are used to calculate the rays. Each 2D viewport point corresponds to a 3D point in world space. This point is found, along with the origin of the camera, and both points are used to derive the line that will carry the ray. Figure 3.21 gives and example of such a ray.

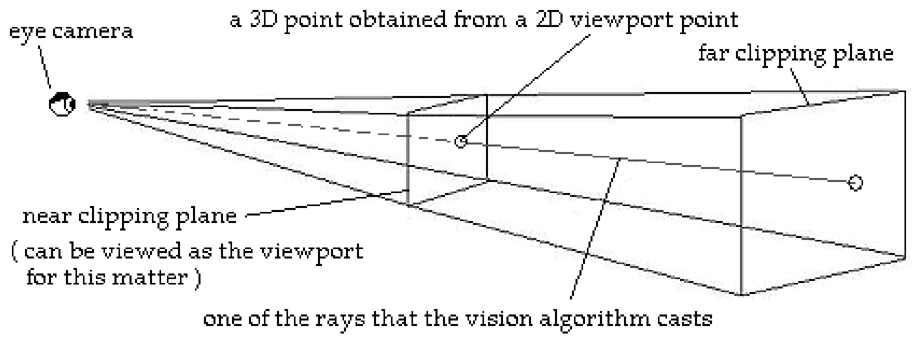

eye camera
a 3D point obtained from a 2D viewport point
far clipping plane
near clipping plane
(can be viewed as the viewport for this matter)
one of the rays that the vision algorithm casts

Figure 3.21: Example of how rays are generated in the vision algorithm

The coordinates of the points included in the original matrix vary in the interval $[0+k, 1-k]$, where k is a security margin to ensure that the rays do not get out of the FoV because of the displacements that are applied in each iteration. The original matrix is square and symmetrical with respect to its center – the point $(0.5, 0.5)$ – which might or might not be included in it, depending on whether it has an odd or even number of points. An explanation of how this matrix is computed can be found in [68]. Figure 3.22 shows the point distribution of a 20×20 such matrix.

A synthetic vision algorithm acts as a filter that distinguishes which objects are seen and which are unseen by some VH. Collisions between the rays that the algorithm casts and the bounding boxes of the objects are checked for. To detect

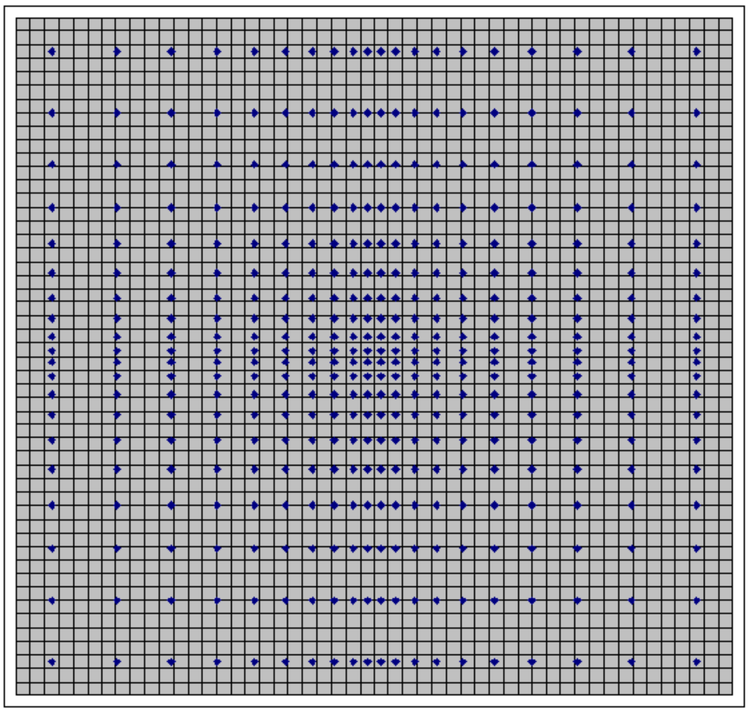
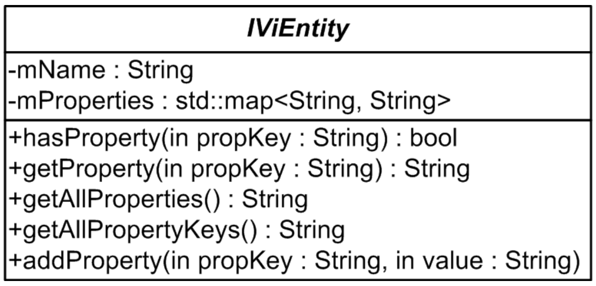


Figure 3.22: Point distribution of a base matrix for synthetic vision

the collisions, Axis Aligned Bounding Boxes (AABBs) are used in the first place. When a collision is detected with an AABB, a much more tight bounding box is used to determine whether the object is really intersected by the ray. This allows the algorithm to act as a more precise filter, so that VHs can detect objects that were occluded by an AABB but that are not really occluded by the corresponding object. For instance, with this approach the VHs can see through the hole in a torus.

The vision algorithm only checks which objects are visible. After executing it, a VH can access the properties of the objects he sees but, for him to be able to extract them, every object that he can see must be accessible through a common interface. In our platform, this common interface is provided by the class *IViEntity*, which was already partially presented. The attributes and methods of this class that matter the most for this purpose are shown in Figure 3.23.



```
IViEntity
-mName : String
-mProperties : std::map<String, String>
+hasProperty(in propKey : String) : bool
+getProperty(in propKey : String) : String
+getAllProperties() : String
+getAllPropertyKeys() : String
+addProperty(in propKey : String, in value : String)
```

Figure 3.23: A portion of the class *IViEntity*

Direct and indirect instances of the class IViEntity can have built-in properties and custom properties. Eventually, these properties have to be sent to the AI layer, where they will be processed by the intelligent agents. So they must have a common representation, to be transmitted over the network and parsed on the other side. Here we decided for simplicity and we have all the properties represented as string key-value pairs that are human readable. One string keeps the name of the property, which is used as its key, and another keeps its value. When sending a property, or a list of properties, over the network, the single strings are joined together, interweaving them with other predefined separator strings.

The properties are already kept as string key-value pairs in the IViEntity. This class has an attribute – *mProperties* – that maps the name of each property to its value. Nonetheless, this map is used only for custom properties. Any custom properties can be added to any IViEntity object and changed, at runtime, provided that the key-value string representation is used. Objects can even have descriptions of how they can be handled, which would make them similar to *smart objects* [39]. However, these properties are static, unless they are externally changed.

Built-in properties are also included. For instance, any IViEntity is already created with properties like *speed*, *velocity*, or *facing*. Classes that descend from IViEntity add their own built-in properties. For instance, some of the properties that are included in every MovingCharacter are *characterName*, *mass*, and *maxSpeed*. These properties cannot be externally changed. Some of them are completely static and do not ever change within the lifetime of the object (e.g characterName). Others are dynamic and their string representation is generated only when asked for (e.g. speed).

Figure 3.24 shows a diagram the includes the main classes and associations involved in the vision process. An IViHuman has an instance of *SyntheticRayVision*, which is an abstract class. This class declares the interface that the IViHuman uses to see. The main portion of this interface is accomplished by the methods *SyntheticRayVision::see*, *SyntheticRayVision::doSee*, and *SyntheticRayVision::recallLastSeen*. The class SyntheticRayVision has an attribute that *remembers* the results of the last iteration of the algorithm – *mShortMemory*. This attribute has the type *VisibleObjects* which, as the name indicates, is a data structure that can hold relevant information about visible objects, among which are their identifiers. Two methods can be used to execute one iteration of the vision algorithm. *SyntheticRayVision::doSee* is the basis one. It does not return anything, but it records its results in the attribute mShortMemory. This method is called by *SyntheticRayVision::see*, which returns the results afterwards. The method *SyntheticRayVision::recallLastSeen* can be used to obtain the results of the last iteration.

SyntheticRayVision is still an abstract class, since it leaves the actual imple-

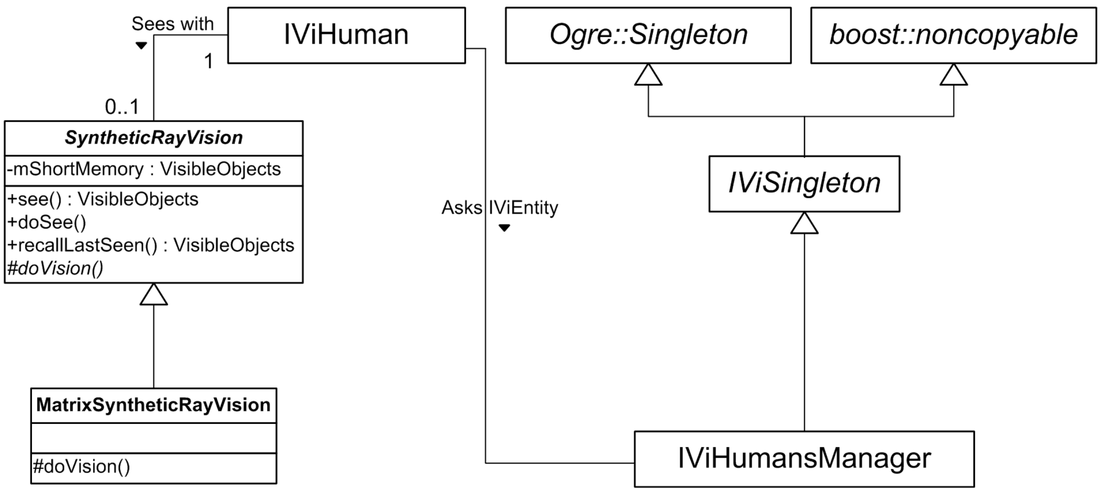


Figure 3.24: Class Diagram – Vision and *IViHumansManager*

mentation of the algorithm to its subclasses. This implementation should be determined by the method doVision. The class *MatrixSyntheticRayVision* implements this method according to the algorithm that Semião et al. created. Other algorithms can be implemented, though, as long as they return the same data for objects that are seen in each iteration. We chose this design so that different synthetic ray vision algorithms can be added without changing existing code.

When the owner of the SyntheticRayVision object gets the results of casting vision, its client becomes aware of the identifiers of the objects whose data the IViHuman is allowed to access. To get information about the objects, he first obtains the references to the actual objects through the singleton object [34] of *IViHumansManager*, a class that has several managing duties related to the entities that inhabit an IViHumans' world (e.g. creation, memory release, indexation, update). The client can then use the properties interface to obtain the required data.

Memories

Synthetic vision can be explicitly called upon, to obtain information about the objects that a VH sees at any given moment. The AI layer can post such requests on the GP layer any time, and the vision algorithm will be executed. Memories and beliefs can then be managed by the intelligent agents. However, we included an alternative in the GP layer that we believe significantly reduces the communication overhead and that releases the AI layer from the burden of constantly asking the VHs what they see. It may also be used for reactive behavior directly handled by the GP layer (e.g. by creating steering behaviors that use it). This alternative consists on an automatic vision mode with custom memory associated.

An IViHuman can be told to activate the "auto vision" mode, so that he periodically executes his vision algorithm on his own. This can be done with the method

IViHuman::activateAutoVision (see Figure 3.25), which receives a real number that specifies the frequency (f), in Hertz, for the periodical execution of vision. A timer is initialized when the auto vision mode is activated. The timer is updated when the IViHuman object is updated and, if more than a whole period ($T = \frac{1}{f}$) went by since the last iteration of the algorithm, it is executed again and the timer is reset. The first time this timer is initialized, it is assigned a random value in the interval $[0, T)$, so that, if several VHs were given auto vision and are updated in the same time-steps, vision is carried asynchronously to all of them. Otherwise, and if they all had the same vision frequency, the algorithm would be executed for all of them at the same time, which would greatly increase processing time before rendering certain frames. With this simple solution we get a uniform statistical distribution of the execution of the algorithm over time.

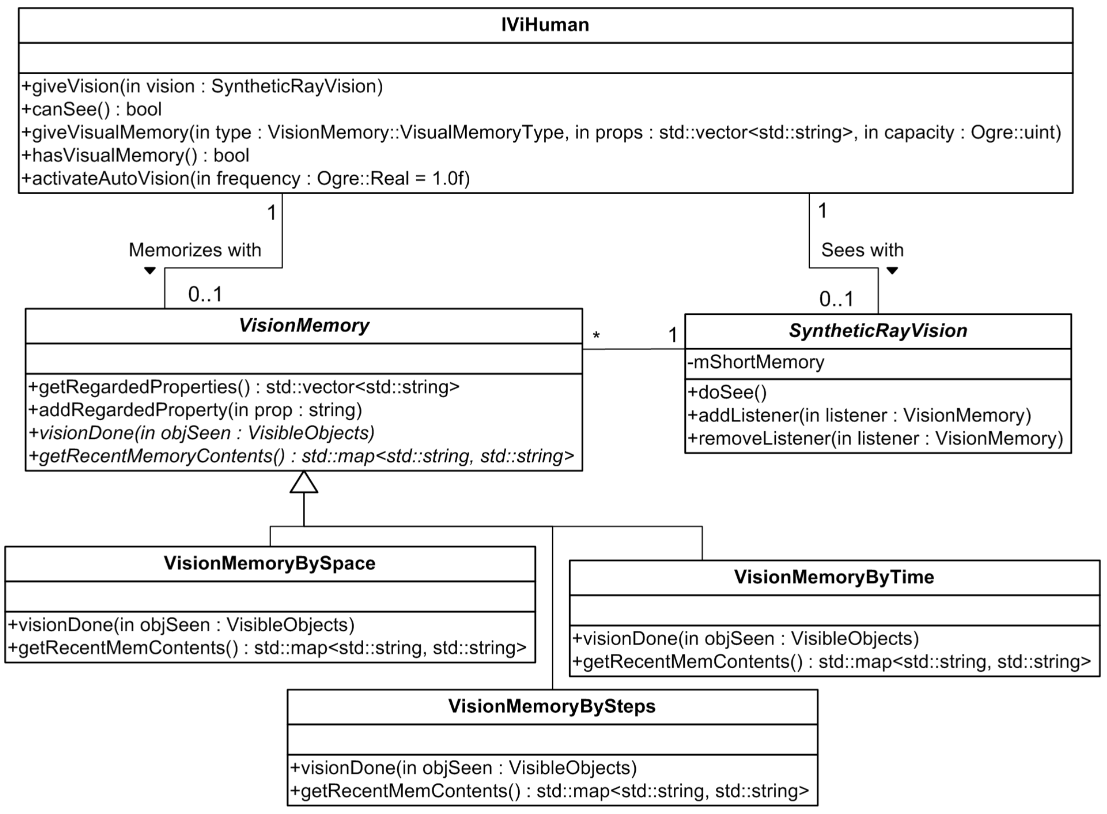

IViHuman
+giveVision(in vision : SyntheticRayVision)
+canSee() : bool
+giveVisualMemory(in type : VisionMemory::VisualMemoryType, in props : std::vector<std::string>, in capacity : Ogre::uint)
+hasVisualMemory() : bool
+activateAutoVision(in frequency : Ogre::Real = 1.0f)
1
Memorizes with
0..1
1
Sees with
0..1
VisionMemory
+getRegardedProperties() : std::vector<std::string>
+addRegardedProperty(in prop : string)
+visionDone(in objSeen : VisibleObjects)
+getRecentMemoryContents() : std::map<std::string, std::string>
*
1
SyntheticRayVision
-mShortMemory
+doSee()
+addListener(in listener : VisionMemory)
+removeListener(in listener : VisionMemory)
VisionMemoryBySpace
+visionDone(in objSeen : VisibleObjects)
+getRecentMemContents() : std::map<std::string, std::string>
VisionMemoryByTime
+visionDone(in objSeen : VisibleObjects)
+getRecentMemContents() : std::map<std::string, std::string>
VisionMemoryBySteps
+visionDone(in objSeen : VisibleObjects)
+getRecentMemContents() : std::map<std::string, std::string>

Figure 3.25: Class Diagram – Vision and Memory

There would be no point in having the vision algorithm automatically executed if the results were not used. As it is, the IViHuman class does not analyze the results of the vision algorithm, as this is assumed to be mainly a responsibility of the AI layer. So, the auto vision mode cannot be executed unless the IViHuman has a memory that can record relevant data and provide it whenever needed. *VisionMemory* is the abstract class that represents this memory, as shown in Figure 3.25.

Every VisionMemory instance is a *listener* of a SyntheticRayVision object. The former is notified whenever the vision algorithm is executed. VisionMemory objects register themselves with SyntheticRayVision objects with the method *SyntheticRayVision::addListener* and they can stop listening by passing themselves to the method *SyntheticRayVision::removeListener*. The method *SyntheticRayVision::doSee* is a *template method* that, besides calling the polymorphic method *doVision*, notifies every registered listener by calling *VisionMemory::visionDone*, passing the results along. VisionMemory objects can then process these results and save the information they have been commanded to. A VisionMemory can be told what properties to get from the objects that are seen by its owner. This can be done upon construction or through the method *VisionMemory::addRegardedProperty*, by providing the names (the keys) of the intended properties. When the VisionMemory instance is notified of the results of one iteration of the vision algorithm, they obtain the value of each property they were commanded to focus on, for each object. If any object does not have any of these properties, the property is simply ignored for that object.

VisionMemory is still only an abstract class that aggregates the main features that must be provided by every actual memory. Subclasses implement the pure virtual methods that had no implementation, according to their specific memory model. We implemented three subclasses that store and manage data in their own way: *VisionMemoryBySpace*, *VisionMemoryByTime*, and *VisionMemoryBySteps*. Event though they manage their storage space in distinct ways, all the implementations have some limit for their capacity, which is provided upon construction. Thus, they function only as short or medium term memories. A more persistent memory model should be built in the AI layer, possibly storing data after it was processed to a more meaningful symbolic representation that may be more useful in reasoning and cognition.

VisionMemoryBySpace stores information for a certain number of objects. Its capacity is determined in terms of the number of object entries that can be stored at any given time. It has a counter that tracks the number of iterations of the vision algorithm. The entries are ordered according to their chronological relation and, when the capacity is exceeded, the results from the oldest iterations are erased, until there is some space left again.

VisionMemoryByTime stores all the information obtained in a certain time span, which is specified by its capacity. An absolute timer is kept and updated when the object is notified of new vision results. Only the results obtained in the last C milliseconds are kept at any time, where C is the capacity of the VisionMemoryByTime instance.

VisionMemoryBySteps is very similar to VisionMemoryByTime, but it uses relative time instead of absolute time, as VisionMemoryBySpace does. It stores all the information obtained in a certain number of vision iterations. As with VisionMemoryBySpace, a counter that tracks the number of iterations of the vision algorithm is kept. The step of each set of results, that is, the iteration it was obtained at, is used as a relative timer and only the results from the last C timesteps are kept, where C is the capacity of the VisionMemoryBySteps instance.

The contents of the IViHuman's memory can be requested by the AI layer. With this memory model, agents can ask for the most recent value of some particular property, for a particular object or for all objects that were seen. They can trace how a certain property varied recently for some object or draw conclusions from relating properties of different objects. They can also request information, from time to time, to monitor the world while reasoning, or to build a persistent memory, at the pace that suits them best.

3.3.4 Networking

As explained earlier, our platform was conceived in such a way that its two layers run in different processes, possibly in completely distinct places, and that they connect by TCP sockets. As the layers are built in different languages and rely on distinct libraries, we had to restrict our use of networking features to the point strictly specified by global standards. For instance, we could not use the mechanism of serialization that Sun's libraries provide for Java, unless we also implemented it in C++ for the GP layer. Instead, we chose to use only the TCP protocol and build upon it.

The communication between the two layers follows the client/server approach. The GP layer provides the server, and the AI layer connects to it through a client. Although this work only comprised the implementation of the server, in C++, some client stubs were also implemented in Java, to ensure that the server was working properly and to produce demonstrations.

Our server relies heavily on the boost libraries, mainly on Boost.Asio, an open source *cross-platform C++ library for network programming that provides developers with a consistent asynchronous I/O model using a modern C++ approach.* This library is based on the Proactor design pattern, whose description can be found elsewhere [66].

A Simple Protocol

The layers of the IViHumans platform communicate with a very simple protocol that we built over TCP. With this protocol, the server can be connected with only one client. The client sends requests to the server, which handles them and replies back to the client, in such a way that every request message from the client triggers one and only one reply message from the server. This allows the computation of both sides to be sequential: the client only sends a new request after receiving the reply for the previous one and the server deals with only one request at a time.

Messages are composed by two fields: the header and the body of the message. The header has a fixed size of $3 bytes$ and it indicates the size of the body. The body has a variable size, but it must not exceed $512 bytes$. These requirements are imposed in the class *IViMessage*, which implements a message in the GP layer, but they may easily be changed. We also implemented this class in Java, to be used by the AI layer, and we gave it the same name.

The body of a message is a human readable array of characters. The first thing that the body of any message includes is an identifier. For request messages, this identifier is generated by the client as an unique number. Reply messages are identified with this number followed by the string `"(R)"`. For request messages sent by the client, the rest of the body is composed by a sequence of keywords that the server can univocally identify as a particular request, possibly along with the parameters of the request. In the replies, and after the identifier, the body can carry string representations of the values that answer client's questions, or simply success or error messages.

One of our main goals for this protocol was that it was extensible. On the one hand, the GP layer is not finished and we hope it comes to provide more features in the future than it does currently. On the other hand, the development of the AI layer is only at its early stages and, in time, we may find that more services than we predicted are needed. So, we need to project existing code in order to welcome subsequent extension. In what concerns the communication protocol, we needed it to be able to cope with semantics that is yet to be defined. To meet this requirement, we conceived the protocol so that it specifies the structure and organization of messages but in a way that does not restrict the meaning of their contents. This was accomplished with an approach that was inspired on the Interpreter pattern [34]. We believe we managed to create a flexible and extensible protocol that is independent of the meaning of messages.

To explain how messages are handled in the server, we begin by explaining our parser architecture. Figure 3.26 shows that our message parsers are organized according to a variation of the Composite pattern. *IViParser* is the base class of all parsers. It is characterized by a string attribute, that can be viewed as its key. The key of a parser is used to identify the message sections it should interpret and to identify the parser itself, since there is no point in having more than one parser for each command section. The IViParser class accepts the key in its constructor.

IViParser is an abstract class since it provides no implementation for its main method: *IViParser::parse*. This is the method whose implementation determines the behavior of the parser. It receives an *IViTokenizer*, which is an auxiliary type

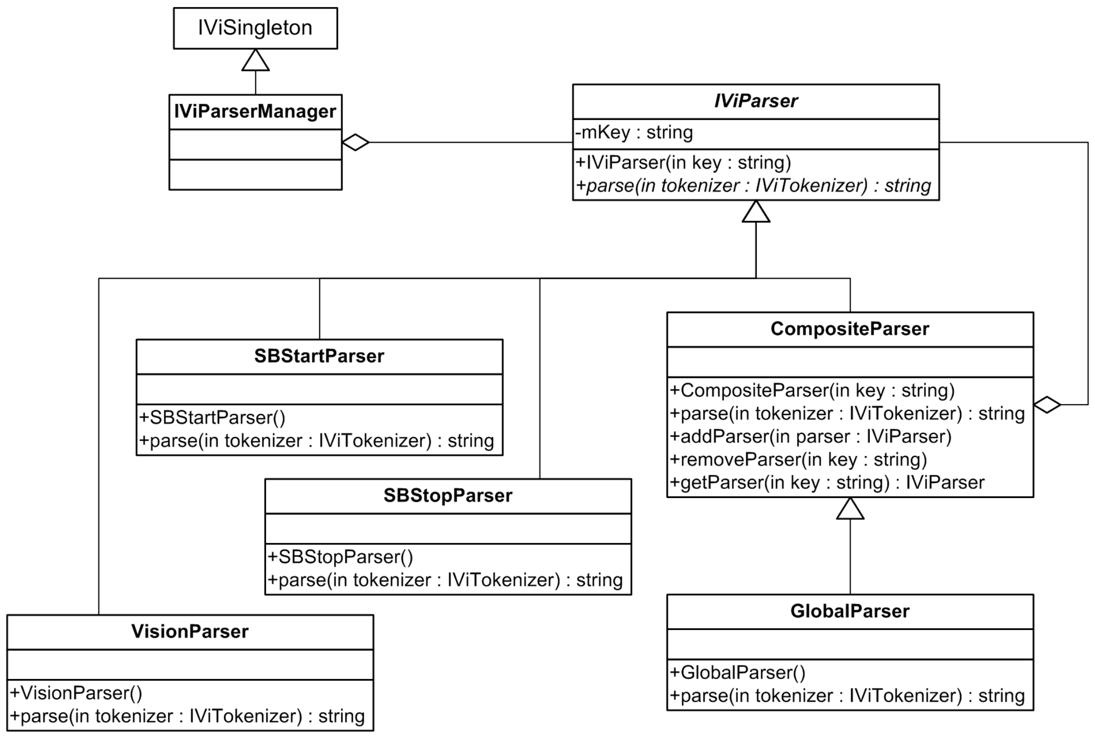

IViSingleton
IViParserManager
IViParser
-mKey : string
+IViParser(in key : string)
+parse(in tokenizer : IViTokenizer) : string
CompositeParser
+CompositeParser(in key : string)
+parse(in tokenizer : IViTokenizer) : string
+addParser(in parser : IViParser)
+removeParser(in key : string)
+getParser(in key : string) : IViParser
SBStartParser
+SBStartParser()
+parse(in tokenizer : IViTokenizer) : string
SBStopParser
+SBStopParser()
+parse(in tokenizer : IViTokenizer) : string
VisionParser
+VisionParser()
+parse(in tokenizer : IViTokenizer) : string
GlobalParser
+GlobalParser()
+parse(in tokenizer : IViTokenizer) : string

Figure 3.26: Class Diagram – Parsers

that can encapsulate messages and split them into tokens, and it returns a string that is used in the construction of the reply message.

All the existing subclasses of IViParser are concrete classes that have implementations for all their methods. *SBStartParser*, *SBStopParser*, and *VisionParser* are leaf parsers. They rely on no other parsers to complete the interpretation of one message, as opposed to *CompositeParsers*. CompositeParser instances aggregate other IViParsers and, after parsing their share of the message, they forward the remainder of it to their components. *GlobalParser* is a special kind of CompositeParser, meant to be the first to receive the message. Put another way, it is used as the root of the parser tree.

The classes IViParser, CompositeParser, and GlobalParser are not tied to any particular message semantics. Rather, they suggest a tree structure organization for the set of possible messages. SBStartParser, SBStopParser, and VisionParser, on the other hand, are parsers that know how to parse a single kind of message. While the key of IViParser and CompositeParser instances must be defined upon creation, their descendants automatically initialize this key according to their particular purpose. The key of GlobalParser is simply the empty string – `""`. SBStartParser, SBStopParser, and VisionParser have their keys automatically initialized to `"start"`, `"stop"`, and `"vision"`, respectively.

CompositeParser is the class that most parsers are created from, as it is meant for those that are responsible for parsing the middle sections of a message, that is, for those parsers that constitute the middle nodes of the parsers tree. To understand how the method *CompositeParser::parse* works, look at the example of Figure 3.27. The root of the parsers tree is, as always, an instance of GlobalParser. This instance can aggregate any number of IViParsers, as the general CompositeParsers. In this case, it has two composite parsers, whose keys are `"foo"` and `"bar"`. The CompositeParser `"foo"` aggregates two leaf parsers with keys `"a"` and `"b"`. The CompositeParser `"bar"` has one descendant parser, keyed `"c"`.

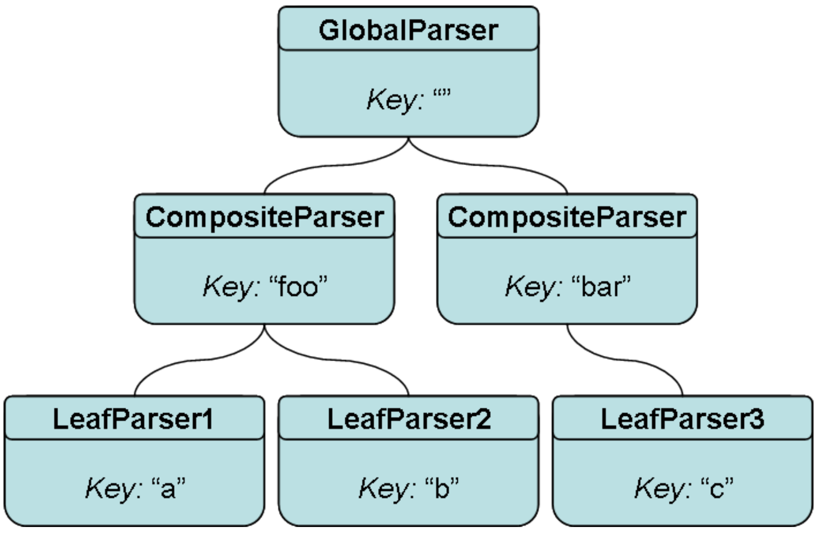


Figure 3.27: Example of a tree of parsers

In this example, *LeafParser1*, *LeafParser2*, and *LeafParser3* are imaginary leaf parser classes. This parser tree would be capable of parsing any message that could be generated by the following context-free grammar, followed by any string[20]

```
   <message> ::= <foo> | <bar>
       <foo> ::= "foo" <separator> ("a" | "b")
       <bar> ::= "bar" <separator> "c"
 <separator> ::= <sepliteral> <separator>
<sepliteral> ::= " " | "," | ";" | ":"
```

To parse any such message, after keeping the id of the message, the GlobalParser reads the first token and checks which of the parsers it aggregates has that token as key. The following parser, being also a composite one, will do the same thing. When the last key string is reached, the corresponding leaf parser is delegated the responsibility of parsing the remainder of the message, which may include an arbitrary string in which any parameters can be supplied. After completing the computation that is required by the client's request, the leaf parser returns a string that is used

[20]The grammars are expressed in *Augmented Backus-Naur Form* (ABNF) [2], sometimes called *Extended Backus-Naur Form* (EBNF)

to form the reply message. The control goes back up in the chain and any involved parser can add information to the returned string. Finally the GlobalParser adds the string `"<id>(R)"` to the beginning of the string, before returning it.

In some cases, it would be important to have some of the CompositeParsers do a part of the processing, eventually consuming more from the original message than their key. This can be very easily accomplished by subclassing CompositeParser and overriding the method *CompositeParser::parse* accordingly, so that parameters are interpreted and the right tasks are carried before forwarding the rest of the message to the proper component parser. This is exactly what GlobalParser does.

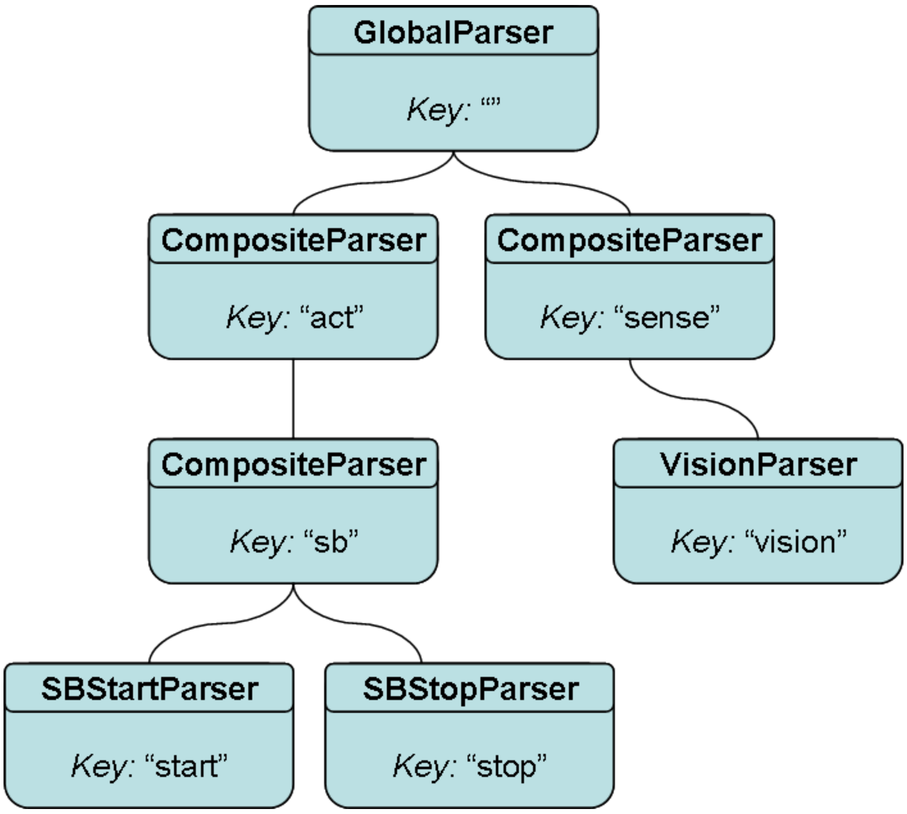


Figure 3.28: Our default tree of parsers

The leaf parsers we implemented allow us to parse a small set of messages, that can be used for demonstration purposes and extended in the future. Our default parser tree looks like what is show in Figure 3.28 and it is fit to parse the messages that can be generated by the following grammar, where some intuitive rules are informally defined:

```
<message> ::= <id> <separator> (<sense> | <act>)
  <sense> ::= "sense" <separator> <vision>
 <vision> ::= "vision" <separator> <targetIV> <separator>
                  (<obj> | "all")
```

```
    <obj> ::= <objID> <separator> (<property> | "propKeys" | "props")
    <act> ::= "act" <separator> <sb>
     <sb> ::= "sb" <separator> (<start> | <stop>)
  <start> ::= "start" <separator> <targetMC> <separator>
                   <steeringBehavior> <separator> <params>
   <stop> ::= "stop" <separator> <targetMC> <separator>
                   (<steeringBehaviorID> | "all")
  <separator> ::= <sepliteral> <separator>
 <sepliteral> ::= " " | "," | ";" | ":"
```

`<id>` – a number that identifies the request message

`<targetIV>` – the name that identifies an IViHuman

`<targetMC>` – the name that identifies a MovingCharacter

`<objID>` – the name that identifies an IViEntity that is seen by the target IViHuman

`<property>` – the name of the property whose value is requested

`<steeringBehavior>` – the name of a (type of) steering behavior (e.g. Arrive2D-Behavior)

`<params>` – a list of the parameters that are used to create the steering behavior

`<steeringBehaviorID>` – the id of a particular steering behavior

The rules of this grammar could be simplified, but we chose to present them so that each IViParser in our structure had a corresponding rule. The root parser – GlobalParser – acts over messages described by the rule `<message>`. `<sense>`, `<act>`, and `<sb>` are the rules for the other composite parsers. Finally, `<vision>`, `<start>`, and `<stop>` are the rules for the leaf parsers.

The replies to these requests are very simple. First of all comes the reply id (`<id>(R)`). Then, when the client expects a value, or a list of values, their string representation forms the remainder of the message. For instance if the server received a request message with the body `1 sense vision Human1 object1 speed`, and the speed at which `Human1` saw `object1` was $s = 20$, the reply would be simply `1(R) 20`. If the request was `2 sense vision Human1 object1 position`, and the position seen for `object1` was $\overrightarrow{P} = (5, 0, 10)$, the reply would be `2(R) 5 0 10`. If the server successfully handles a request regarding a particular SteeringBehavior object, its id is used for the body of the reply. When an error occurs and the server is not able to cope with the request, an error description is put after the reply id.

This parser structure is used for a particular set of messages that account only for a part of the features that the GP layer already provides. The main portion that is not addressed by currently implemented messages is the one that deals with expressions. However, the existing parser structure can be extended to support further semantics. For instance, suppose that it was to be extended to deal with expressions, with a similar approach to the one is used for steering behaviors. A possible solution would be to use a CompositeParser with the key `"expression"` that would have leaf parsers for the keywords `"activate"` and `"deactivate"`. Of course, the corresponding leaf classes would have to be implemented, extending IViParser directly. Suppose also that the `"expression"` parser had to read a parameter before forwarding parsing control to its components. The class CompositeParser alone could not do the trick, but a class *ExpressionParser* that inherited from it could be created, overriding the method *ExpressionParser::parse* so that it consumed the parameter from the IViTokenizer that encapsulated the message. When this method was called, the keyword `"expression"` would already have been consumed. The ExpressionParser would consume and process the parameter. It would then consume the following keyword to check whether it should call the parse method on *ActivateExpressionParser* or *DeactivateExpressionParser*. The ExpressionParser instance would be given as a component to the `"act"` CompositeParser, since emotion expression can be viewed as a special form of acting.

Other extensions could be added, even to control other aspects of the virtual world. Also, this protocol could be used by any client that followed it, even if it had no intent of dealing with artificial intelligence. For instance, a GUI client could be connected to the GP layer to have a real human remotely control not only the VHs but also other world or application aspects (e.g. lighting conditions, viewport configuration).

IViParserManager

Instead of leaving them for the application layer, we concentrate the duties of managing parsers in the class IViParserManager (see Figure 3.29). This class deals with low-level memory management for IViParsers and provides the correct parsers by key, ensuring that only one parser is created for the same purpose. Clients of this class can access any parser at any time, without worrying about whether it was already created or not. Further, it encapsulates parser creation at a single place, by accepting *creator functions* that determine its behavior when parsers that were not created yet are requested. These creator functions have to be registered with the IViParserManager by the layer that establishes the semantics of the messages, which is built on top of existing code, to relieve the application layer from that duty. This way, as long as a single set of messages has been defined as enough for the intended purposes, multiple applications can use it without always having to rebuild parser structure. Besides, a default parser structure is provided by the method *IViParser-Manager::registerDefaultCreators*. As the current parser structure (that of Figure 3.28) is not complete yet, the implementation of this method is still not final. In the future, it should be augmented to include new developments in what regards general purpose message semantics.

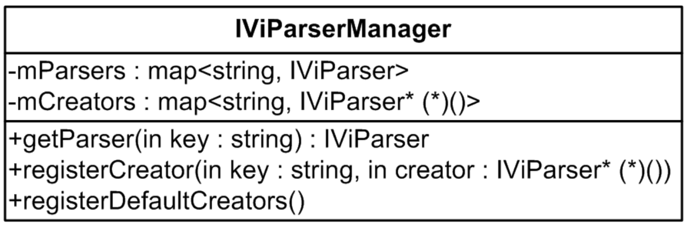

IViParserManager
-mParsers : map<string, IViParser>
-mCreators : map<string, IViParser* (*)()>
+getParser(in key : string) : IViParser
+registerCreator(in key : string, in creator : IViParser* (*)())
+registerDefaultCreators()

Figure 3.29: The class *IViParserManager*

IViParserManager aggregates IViParsers in the attribute *mParsers*. As it deals with only one parser per key, it uses the key as their identifier to index them in a map. The method *IViParserManager::getParser* provides parsers by their key. When this method is called, the IViParserManager checks if it already has any parser with the supplied key. If it does, it simply returns the parser, otherwise, it creates one before returning it. However, as parser creation differs according to the intended parser structure, the IViParserManager cannot have hardcoded implementations for every possible parser creation process. Instead, it registers what we call *creator functions*, that is, functions that are called back when the corresponding parsers need to be created. In fact, IViParserManager aggregates these creator functions, in the attribute *mCreators*, also indexing them by the key of the parser they create. The manager must have a creator function for every parser it is asked for, otherwise an exception is thrown when it realizes that it does not know how to create them.

Recall that, to create a parser, most of the time it is not enough to declare and initialize the parser. CompositeParsers must be given their components, which might also have to be created. On the other hand, any IViParser subclass could declare a constructor that received obligatory configuration parameters. What the method *IViParserManager::registerDefaultCreators* does is spare programmers from having to make this effort when the current parser structure is fit for their intentions, by automatically registering creator functions for the involved parsers. Appendix C shows the implementation of this method and of the creator functions that are registered by default[21].

[21]These functions are local to the compilation unit used for the IViParserManager class and cannot be used externally

Services and Leaf Parsers

The current leaf parsers provide the grounding interface between the server and the rest of the GP layer. Although our architecture enables composite parsers to play a role in interpreting and mapping messages to commands, currently the leaf parsers are the ones that actually fulfill the requests.

VisionParser can handle two essential kinds of requests related to the VHs' sense of vision: obtaining the identifiers of the objects that are seen by a certain VH; obtaining characteristics for a certain object. The first request is carried by the message `<id> sense vision <targetIV> all`. When the vision parser is asked to parse its section of such a message, it commands the IViHuman with the name `<targetIV>` to execute its vision algorithm and returns a string with the sequence of identifiers of the objects that were seen (their names). The other kind of requests comprehends the three messages `<id> sense vision <targetIV> <objID> (<property> | propKeys | props)`. In the first case, `<property>` assumes the name of a property and the VisionParser replies with the value of that property for object `<objID>`, if this object is being seen by `<targetIV>`. The IViHumans and the other IViEntities are obtained through the IViHumansManager. If the object is not seen by the IViHuman, if it does not exist, or if it does not have the specified property, an error message is returned.

SBStartParser can add steering behaviors to a moving character. For that, it asks the singleton instance of *SBFactory* to create them (see Figure 3.30). SBFactory is the class that manages the construction and destruction of SteeringBehaviors and, for that, it also aggregates instances of another class – *SBCreator*. The ideia is similar to that of IViParserManager, in which creator functions are registered with the manager so that he knows how to create custom IViParsers. Here, the SBCreator instances have an analogous role to that of creator functions. Due to implementation details, SBCreators benefit from being instances of a class instead of functions.

When a request is made to add some SteeringBehavior to an IViHuman, it is handled differently based on whether the new behavior is a CombineBehavior or not, on whether any steering behavior is set for the IViHuman currently, and, if so, which is. The pseudocode shown in Figure 3.31 illustrates how the addition of a steering behavior is processed depending on these factors.

This pseudocode shows that there are cases in which a new CombineBehavior must be created, even if it was not explicitly asked for. Even though the current implementation only includes one CombineBehavior, namely SimpleCombineBehavior, clients of the platform may wish to provide and use other alternatives. In order for the parser to obtain a CombineBehavior automatically, one must have been appointed as default. Our solution to this issue is based on the *Prototype* de-

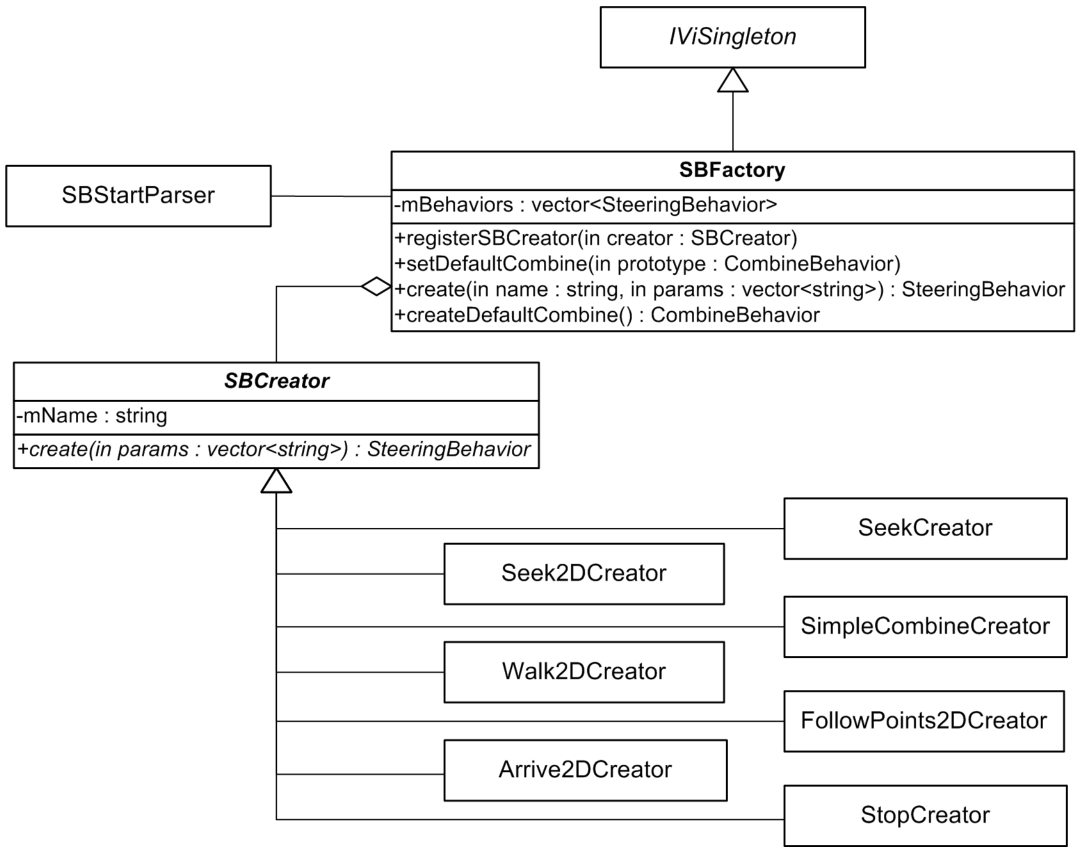

IViSingleton
SBFactory
-mBehaviors : vector<SteeringBehavior>
+registerSBCreator(in creator : SBCreator)
+setDefaultCombine(in prototype : CombineBehavior)
+create(in name : string, in params : vector<string>) : SteeringBehavior
+createDefaultCombine() : CombineBehavior
SBStartParser
SBCreator
-mName : string
+create(in params : vector<string>) : SteeringBehavior
SeekCreator
Seek2DCreator
SimpleCombineCreator
Walk2DCreator
FollowPoints2DCreator
Arrive2DCreator
StopCreator

Figure 3.30: Class Diagram – *SBStartParser*, *SBFactory* and *SBCreators*

```
If(!human.hasSteeringBehavior())
    human.giveSteeringBehavior(requestedBehavior);
Else
    If(requestedBehavior.isCombine())
        If(human.getCurrentBehavior().isCombine())
            For(SteeringBehavior sb In human.getCurrentBehavior())
                human.getCurrentBehavior().removeSteeringBehavior(sb);
                requestedBehavior.addSteeringBehavior(sb);
            human.giveSteeringBehavior(requestedBehavior);
        Else
            requestedBehavior.addSteeringBehavior(human.
                getCurrentBehavior());
            human.giveSteeringBehavior(requestedBehavior);
    Else
        If(human.getCurrentBehavior().isCombine)
            human.getCurrentBehavior().addSteeringBehavior(
                requestedBehavior);
        Else
            CombineBehavior comb;
            comb.addSteeringBehavior(human.getCurrentBehavior());
            comb.addSteeringBehavior(requestedBehavior);
            human.giveSteeringBehavior(comb);
```

Figure 3.31: Pseudocode for the addition of a *SteeringBehavior* to a *MovingCharacter*

sign pattern [34]. The default CombineBehavior can be appointed by calling the method *SBFactory::setDefaultCombine* and passing in a standard instance of the CombineBehavior that should be used. This instance will have to be copied every time the default CombineBehavior is requested by the SBStartParser. To that end, the class CombineBehavior declares the pure virtual method *CombineBehavior::prototypeClone*, which returns a new CombineBehavior that is intended to be a shallow copy of the instance the method is called upon. We say that the copy should be shallow because no component behaviors should be included in the new CombineBehavior. This method should be implemented so that the returned copy is of the same ultimate type as the original and so that any particular configurations are kept. In the case of our SimpleCombineBehavior, it simply returns a new SimpleCombineBehavior.

SBStopParser can handle two different kinds of requests: to deactivate a particular steering behavior from a MovingCharacter; to completely stop the movement of the MovingCharacter. To cope with the second kind of request, the parser exchanges the current steering behavior of the moving character for an instance of StopBehavior obtained from the SBFactory singleton instance. To stop a single behavior, the parser first checks whether the supplied id corresponds to the current behavior of the MovingCharacter. If it is, the parser removes it. Otherwise, if the current behavior is a CombineBehavior, the parser executes a depth-first search over the corresponding behavior tree. If no behavior with the id is found, an error message is returned.

The Server

The server of the IViHumans platform is implemented in the GP layer, by the class *IViServer*. The sessions that are established with the client also have a class representation – *IViServSession*. The IViServer relies on the Boost.Asio library and it is most dependent on an object of the type *boost::asio::io_service*, which is essential for asynchronous networking operations. With the help of this object, the server behaves almost automatically.

Figure 3.32 shows the classes IViServer and IViServSession. The class IViServer is able to accept connections right after it is created, albeit only one at a time. When a connection is accepted, the IViServer creates an instance of IViServSession, which will be expecting messages from the client right away.

Networking operations are executed asynchronously, so that processing time is properly shared with the other tasks of the GP layer. To execute an asynchronous Input/Ouput (I/O) operation, clients of the Boost.Asio library post the corresponding request to the io_service object, providing a *handler method* to be called right after the I/O operation completes. The io_service registers the request and im-

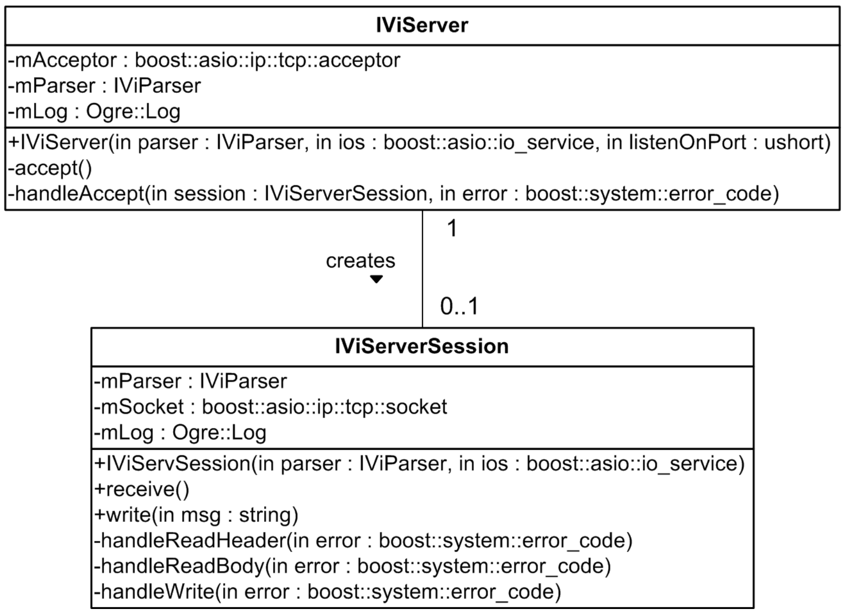

IViServer
-mAcceptor : boost::asio::ip::tcp::acceptor
-mParser : IViParser
-mLog : Ogre::Log
+IViServer(in parser : IViParser, in ios : boost::asio::io_service, in listenOnPort : ushort)
-accept()
-handleAccept(in session : IViServerSession, in error : boost::system::error_code)
1
creates
0..1
IViServerSession
-mParser : IViParser
-mSocket : boost::asio::ip::tcp::socket
-mLog : Ogre::Log
+IViServSession(in parser : IViParser, in ios : boost::asio::io_service)
+receive()
+write(in msg : string)
-handleReadHeader(in error : boost::system::error_code)
-handleReadBody(in error : boost::system::error_code)
-handleWrite(in error : boost::system::error_code)

Figure 3.32: Class Diagram – *IViServer* and *IViServSession*

mediately returns control to the client, so that he can continue with other duties. The requests are later dealt with by the io_service, when the right method is called upon it. In the case of the IViHumans platform, the main loop of any GP layer application must adequately distribute processing time by all the tasks. Thus, at each iteration, the io_service should be commanded to execute at most one pending operation, without blocking. This should be done in the main loop by calling the method *io_service::poll_one*[22]. The implementation of the server follows the *continuation-passing style*, in which the continuation of execution is represented by a handler method, in order to avoid blocking the main loop with I/O operations.

When the IViServer object is created, it automatically posts an asynchronous task to accept a client connection, providing the method *IViServer::handleAccept* as handler. Control is immediately returned to the method that created the IViServer object to carry on with any other operations. When *io_service::poll_one* is hit, the io_service checks whether there are any operations awaiting completion and it notices the accept operation, executing it. If the client did not try to communicate with the server yet, the operation is delayed again and control is returned. Otherwise, the connection is accepted and the handler is called. In IViServer::handleAccept, an IViServSession object is created and an operation is posted to the io_service to read h bytes, where h is the number of bytes of the header of an IViMessage. The method *IViServSession::handleReadHeader* is passed along as a handler. Until something is received from the client, this read operation is the only operation pending on the

[22]which should be followed by the method *io_service::reset*, due to design details of the underlying library

io_service. When something arrives, the io_service reads x bytes, where $x <= h$, and returns. This action is repeated in each call to *io_service::poll_one* until $x = h$, at which point the read operation is complete. IViServSession::handleReadHeader is then called. In this method, the header is decoded to check how many bytes long is the body of the message. Still in this method, a new read operation is posted, to read n bytes, where n is the number of bytes that the body of the message occupies. Again, control is returned and the application can carry on with anything else. When io_service::poll_one is hit, again only an undefined number of bytes is read. Durring the following iterations of the main loop, eventually n bytes are read and *IViServSession::handleReadBody* is called. In this method, the IViServSession has a complete IViMessage object. The message is encapsulated in an IViTokenizer and passed to the global parser. The parser parses the message, forwarding it down the parser tree. The request is handled and, eventually, a reply string is returned back up the tree, to the method that is handling the reception of the message (*IViServSession::handleReadBody*). A request for the io_service to write the reply is made and the control is returned. Again, several iterations of the main loop may go by until all the bytes are written to the socket stream. When the last byte is written, the method *IViServSession::handleWrite* is called. In this method, a new post to read a message header is made and everything happens again.

If the io_service fails to execute any of the tasks he is assigned to perform, the corresponding handler is called with a meaningful error code (recall the declaration of the mentioned handlers, in Figure 3.32). The handler uses this code to create and throw an exception. This exception can be caught, in the main loop and, if the error occurred because the client terminated the connection, the server can be initiated again, so that the GP layer recovers from the problem. Other errors can be also dealt with or ignored by the application. It can be stated that the server carries on its duties autonomously as, besides updating the io_service, the only time the application needs to worry about the server is when it creates it and when it is restarted.

Chapter 4

Conclusions

Many ideas involved in the IViHumans platform are borne by current scientific context. Thus, we present the contributions that we consider the most important for the topic of virtual humans and virtual environments. Although current scientific research in this topic is shaped in large proportions, we share the prevailing opinion that there is still much space for further efforts to help leveraging it. It is with this hope that we seek to create a generally applicable platform to support the development of applications that integrate virtual humans and virtual environments, and that are conceived for various purposes.

The IViHumans platform comprises one layer for graphical processing and one for artificial intelligence. The layers were projected to run in different processes, communicating by means of a simple, yet effective and extensible client/server protocol that we projected and implemented. In this framing, the graphical processing layer plays the role of server, while the artificial intelligence layer occupies the position of the client. Therefore, the graphical processing layer is the base of the platform, providing services for the intelligent agents that populate the artificial intelligence layer.

We expose the methods we applied to achieve the main goals that were projected for the IViHumans platform, focusing on the layer that accomplishes the graphical processing, along with the theory that sustains them. The tasks of the graphical processing layer rely, first of all, on graphical representations. For that matter, we highlight the techniques used in object modeling, mainly in what concerns the virtual humans we use, and we explain how their graphical representation reflects in static and dynamic scene rendering. We also focus on our design and implementation and on how we applied the principles of object oriented design to confer flexibility to the platform and to enable different applications to use and build upon its core features, which support the exploration of different kinds of simulations.

We pay special attention to the central capabilities of the virtual humans. We explain our approach to the application of Reynolds' conception of movement to make virtual humans and, potentially, other objects steer autonomously in the world, while displaying consistent animations that are automatically chosen according to character specific rules. We expose the solution of our framework for facial expressions that can be mixed together to transmit complex emotions, being subject to multiple blending that automatically drives smooth transitions on characters' faces. We also discuss how facial expressions are encapsulated in convenience animations that allow easy manipulation and that can be extended to reference objects that aid the production of deformations. We show how virtual objects can be characterized with default and custom properties and we discuss the integration of perception through synthetic vision, as well as how it is coupled with distinct kinds of automatic memory that recalls arbitrary attributes of the objects that inhabit the virtual world. Some of these features are depicted in the most recent video we created, that can be found at [20].

Due to the scarceness of human resources assigned to this project, its diverse sub-topics are tackled somewhat slowly when compared to our wishes. In time, we came to the solid conclusion that the configuration of this project demands a judicious layout of priorities. The priority scheme we followed in the development of the graphical processing layer of the IViHumans platform was driven by a central idea: *to start by creating a conducting wire that crosses this layer and unites it to the artificial intelligence layer; then, to broaden and consolidate this wire, in an iterative way, so that it becomes a fully consistent bridge.* Many other paths could have been followed, to explore different ideas, and many will certainly still be. However, to achieve solid results it is essential to stand on solid ground. We hope that we built accurately shaped foundations that can sustain ever higher results in the future.

4.1 Future Work

Since the first stages of conception of the IViHumans platform, much effort was put in projecting its main aspects but, as we believe it is natural in scientific research, the course of this project was not completely defined beforehand. Instead, we established a series of aims that was often revisited while we pursued their accomplishment. During the study of the platform's context and during its development, we came up with relatively detailed ideas that we believe would enrich its applicability to different purposes. Most of these ideas, alike some of the goals that were previously set, could not be put into practice yet. These are left as future work and discussed here.

The integration of the physics engine with the remainder of the graphical processing layer is probably the most important goal for this layer in the short-term. It will supply it with necessary capabilities for the natural progress of the events that underlie any dynamic world. From these, we emphasize the automatic application of forces to the objects that are present in the virtual environment, with special attention to collision handling mechanisms.

As explained in Section 3.2.2, the movement animations that the platform applies show the virtual humans moving in a straight line. It is our opinion that the believability of virtual humans would be increased if, when they were turning, they were animated in different ways on the basis of how steep was their turn. In the same section we suggested that a possible solution would imply that the models could have distinct animations for different turn arcs and that these animations could be blended to generate the right animation for any case. Of course, this would require that the animations were correctly synchronized and that all of them fitted, so that no sliding feet or other artifacts would emerge.

An Inverse Kinematics algorithm could also be added to the graphical layer, to solve issues like these and others as well. This algorithm would be particularly helpful for having characters interacting with the environment in any way that involved touch, be it for grasping, sitting and even walking on rugged terrain.

Transitions between animations should also be accounted for. Currently, they are only correct for the exchange or update of facial expressions, but not in what concerns other features, namely walking and running. When a virtual human goes from an animation to another, the change in his speed is gradual, as is the change in animation speed. However, the transition between the two animations happens instantly and it should not. This problem may be hard to solve, again mainly due to sliding feet artifacts, and finding its solution would require significant research.

Another improvement we proposed, in Section 3.3.3, aims at increasing the efficiency of the vision algorithm. We believe the distribution of the rays that the algorithm casts into the scene could be improved by using information of the objects that are known to be in the field of view of a character. Instead of casting rays to all over the field of view, they could be concentrated in the zones that were known to be occupied by objects. We speculate that this procedure could yield a more efficient and precise algorithm. This idea was not explored yet though and it would require a study to ensure its viability and to check whether it was better than the existing approach. The inclusion of simulation models for other senses besides vision could also be very interesting, even though it should probably be left for a more distant future.

For the user to be able to interact with the graphical processing layer, a GUI should be provided by applications. Some GUI features that are common to applications that display 3D worlds could be developed over a carefully chosen GUI library, to create a standard GUI framework for this layer. Selecting objects by pointing and clicking with the mouse, highlighting and showing information for them, and positioning entities, lights, and cameras in the world, are examples of such features.

For the development of rich applications, much more digital content should be provided to be used in the creation of virtual worlds. Complex material shaders, realistic object and scenery meshes, and accurately rigged and animated character models would be the priorities.

For the whole of the platform, the most urgent effort should clearly be put on projecting and developing the artificial intelligence layer. For now, the graphical processing layer can already be used to quickly create applications that do not include agents with cognitive capabilities. We believe that, when both layers reach a solid state of development and interconnection, the IViHumans platform can be even more fruitful in diverse contexts.

Appendix A

Examples of Configuration Files for Characters

A.1 An Example of a `.mcharacter` File

```
## Parameter definition file for MovingCharacter CMan_v1_1

[General]
mass=0.3
maxSpeed=75.0
maxForce=100.0

[SteeringBehaviorsSettings]
walkMinSpeed=5.0
arriveDecDist=20.0
arriveStopDist=2.0
arriveSpeed=9.0
arriveDecel=6.0
```

A.2 An Example of a `.ivihuman` File

```
## Parameter definition file for IViHuman BizWoman_v1_3

meshFileName=BizWoman_v1_2.mesh
defaultFacing=<default>
headBone=Head
eyeHeight=10.3

# Translation Animations
[TranslateAnims]
WalkForward=0.00001 30 18.5

# Transformation operations to be applied at load time.
# The transformations are applied in alphabetical order of the keys.
[Transform]
a=translate 0 16.9 0
```

Appendix B

Expressions and the Composite Pattern - Two Alternatives

The Composite Design Pattern is used in the IViHumans platform for implementing expressions. There is an issue about this pattern that demands attention, towards which there is no universally accepted correct solution. The question is where should the operations for managing children components be put and opinions diverge regarding this matter.

Since the composite class aggregates other components, it must have operations for adding, removing, and obtaining children components, typically accomplished by methods with names like *addComponent*, *removeComponent*, and *getComponent*. However, it is not unanimous whether this methods should belong only to the composite class or be declared in the component class. It is clear, though, that both approaches have advantages and drawbacks, as the analysis found in [34] explains. The option we followed was to declare these operations only on the composite class. This implies less transparency, since the components cannot be dealt with uniformly, but more security, because any attempt to allude to children of a leaf node is excluded at compilation time. Moreover, in our case, the alteration of an expression that is already fully created is not justified. Indeed, adding components to an expression that is already conceived would most likely deprive it of its essence, effectively turning it into another expression. The class CompositeExpression even leaves the remove operation out, remaining faithful to this opinion.

The time when the use of this operations makes most sense is immediately after the creation of Expression objects. Because in the moment of creation of the expressions their types are necessarily distinguished (since calls to the respective constructors were just made) there is no real loss of transparency in this situation. With this choice we are able to respect a basic principle of object oriented design that determines that a class should only declare operations that are meaningful for every descendant.

Appendix C

Default Creator Functions for *IViParserManager*

```


	//-----------------------------------------------------------------
	IViParser * createSBStartParser()
	{
		return new SBStartParser;
	}

	//-----------------------------------------------------------------
	IViParser * createSBStopParser()
	{
		return new SBStopParser;
	}

	//-----------------------------------------------------------------
	IViParser * createVisionParser()
	{
		return new VisionParser;
	}

	//-----------------------------------------------------------------
	IViParser * createSBParser()
	{
		CompositeParser * ret = new CompositeParser("sb");
		IViParser * child1 = IViParserMgr.getParser("start");
		IViParser * child2 = IViParserMgr.getParser("stop");
		ret->addParser(child1);
		ret->addParser(child2);

		return ret;
	}

	//-----------------------------------------------------------------
	IViParser * createActParser()
	{
		CompositeParser * ret = new CompositeParser("act");
		IViParser * child = IViParserMgr.getParser("sb");
		ret->addParser(child);
```

```

                return ret;
        }

        //------------------------------------------------------------
        IViParser * createSenseParser()
        {
                CompositeParser * ret = new CompositeParser("sense");
                IViParser * child = IViParserMgr.getParser("vision");
                ret->addParser(child);

                return ret;
        }

        //------------------------------------------------------------
        IViParser * createGlobalParser()
        {
                GlobalParser * ret = new GlobalParser();
                IViParser * child1 = IViParserMgr.getParser("act");
                IViParser * child2 = IViParserMgr.getParser("sense");
                ret->addParser(child1);
                ret->addParser(child2);

                return ret;
        }

        //---

        //------------------------------------------------------------
        void IViParserManager::registerDefaultCreators()
        {
                registerCreator("start", createSBStartParser);
                registerCreator("stop", createSBStopParser);
                registerCreator("vision", createVisionParser);
                registerCreator("sb", createSBParser);
                registerCreator("act", createActParser);
                registerCreator("sense", createSenseParser);
                registerCreator("", createGlobalParser);
        }

        //---
```

Bibliography

[1] Autodesk 3ds Max. `http://usa.autodesk.com/adsk/servlet/index?id=5659302&siteID=123112`.

[2] RFC 4254. Augmented BNF for Syntax Specifications: ABNF. `ftp://ftp.rfc-editor.org/in-notes/rfc4234.txt`.

[3] Poser 7. `http://www.e-frontier.com/go/poser`.

[4] Ricardo Abreu, Ana Paula Cláudio, and Maria Beatriz Carmo. Desenvolvimento de Humanos Virtuais para a Plataforma IViHumans. Technical Report DI-FCUL TR-07-32, Departamento de Informática da Faculdade de Ciências da Universidade de Lisboa, November 2007.

[5] Ricardo Abreu, Ana Paula Cláudio, and Maria Beatriz Carmo. Humanos Virtuais na Plataforma IViHumans – a Odisseia da Integração de Ferramentas. In *Actas do 15º Encontro Português de Computação Gráfica, 15º EPCG*, pages 217–222, Outubro 2007.

[6] Ricardo Abreu, Ana Paula Cláudio, Maria Beatriz Carmo, Luís Moniz, and Graça Gaspar. Virtual Humans in the IViHumans Platform. In *Proc. of 3IA 2008, International Conference on Computer Graphics and Artificial Intelligence, in cooperation with Eurographics*, pages 157–162, Athens, Greece, May 2008.

[7] Matt Austern. Draft Technical Report on C++ Library Extensions. Technical Report PDTR 19768, ISO/IEC, 2005.

[8] N. Avradinis, S. Vosinakis, T. Panayiotopoulos, A. Belesiotis, I. Giannakas, R. Koutsiamanis, and K. Tilelis. An Unreal Base Platform for Developing Intelligent Virtual Agents. In *WSEAS Transactions on Information Science and Applications*, pages 752–756, August 2004.

[9] Nikos Avradinis, Themis Panayiotopoulos, and Spyros Vosinakis. Synthetic Characters with Emotional States. In George A. Vouros and Themis Panayiotopoulos, editors, *Methods and Applications of Artificial Intelligence*,

volume 3025 of *Lecture Notes in Computer Science*, pages 505–514. Springer, 2004.

[10] A. Barella, C. Carrascosa, V. Botti, and M. Martí. Multi-Agent Systems Applied to Virtual Environments: a Case Study. In *VRST '07: Proceedings of the 2007 ACM symposium on Virtual reality software and technology*, pages 237–238. ACM, 2007.

[11] Antonio Barella, Carlos Carrascosa, and Vicente Botti. JGOMAS: Game-Oriented Multi-Agent System based on Jade. In *ACE '06: Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology.* ACM, 2006.

[12] Blender. `http://www.blender.org/`.

[13] Craig W. Reynolds. Boids. `http://www.red3d.com/cwr/boids/`.

[14] boost C++ Libraries. `http://www.boost.org/`.

[15] Mat Buckland. *Programming Game AI by Example.* Wordware Game Developer's Library. Wordware Publishing, 2005.

[16] C. O Sullivan C. Peters. Synthetic vision and memory for autonomous virtual humans. In *Computer Graphics Forum*, volume 21, pages 743–752. Blackwell Publishing, November 2002.

[17] M. B. Carmo, A. P. Cláudio, J. D. Cunha, H. Coelho, M. Silvestre, and M. Pinto-Albuquerque. Plataforma de Suporte à Geração de Cenas Animadas com Agentes Inteligentes. In *13º Encontro Português de Computação Gráfica*, pages 79–84, 2005.

[18] Toni Conde and Daniel Thalmann. An Artificial Life Environment for Autonomous Virtual Agents with multi-sensorial and multi-perceptive features. *Comput. Animat. Virtual Worlds*, 15(3-4):311–318, 2004.

[19] LabMAg. Laboratório de Modelação de Agentes. `http://labmag.di.fc.ul.pt/`.

[20] IViHumans Demos. `http://labmag.di.fc.ul.pt/virtual/demos-images.htm`.

[21] OPCODE. OPtimized COllision DEtection. `http://www.codercorner.com/Opcode.htm`.

[22] ODE. Open Dynamics Engine. `http://www.ode.org/`.

[23] Bruce Eckel. *Thinking in C++*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[24] Janete Faustino. Faces: Expressão de Emoções em Humanos Virtuais. Master's thesis, Faculdade de Ciências da Universidade de Lisboa, July 2006.

[25] Janete Faustino, Ana Paula Cláudio, and Maria Beatriz Carmo. Faces – Biblioteca de Expressões Faciais. In *Actas da 2^a Conferência Nacional em Interacção Pessoa-Máquina, Interacção 2006*, pages 139–142, Outubro 2006.

[26] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and Executing Generalized Robot Plans. *Readings in knowledge acquisition and learning: automating the construction and improvement of expert systems*, pages 485–503, 1993.

[27] FIPA. The Foundation for Intelligent Physical Agents. `http://www.fipa.org/`.

[28] Tom Forsyth. Character animation. In Steve Rabin, editor, *Introduction to Game Development*, pages 495 – 537. Charles River Media, 2005.

[29] Tom Forsyth. Graphics. In Steve Rabin, editor, *Introduction to Game Development*, pages 443 – 494. Charles River Media, 2005.

[30] Peter Forte and Adam Szarowicz. The Application of AI Techniques for Automatic Generation of Crowd Scenes. In *The Eleventh International Symposium on Intelligent Information Systems*, pages 209–216, 2002.

[31] JADE. Java Agent DEvelopment Framework. `http://jade.tilab.com`.

[32] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. *Head First Design Patterns*. Head First Series. O'Reilly, 2004.

[33] John Funge, Xiaoyuan Tu, and Demetri Terzopoulos. Cognitive Modeling: Knowledge, Reasoning and Planning for Intelligent Characters. In *Siggraph 1999, Computer Graphics Proceedings*, pages 29–38, Los Angeles, 1999. Addison Wesley Longman.

[34] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[35] C. Izard. Four Systems for Emotion Activation: Cognitive and Noncognitive Processes. *Psychological Review*, 100(1):68–90, 1993.

[36] David Johnson. 3d modeling. In Steve Rabin, editor, *Introduction to Game Development*, pages 675 – 696. Charles River Media, 2005.

[37] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[38] Jean-Marc Jézéquel, Michel Train, and Christine Mingins. *Design Patterns and Contracts.* Addison-Wesley, 2000.

[39] Marcelo Kallmann and Daniel Thalmann. Modeling Objects for Interaction Tasks. In *Proc. Eurographics Workshop on Animation and Simulation*, pages 73–86, 1998.

[40] P. Karla, N. Magnenat-Thalmann, L. Moccozet, G. Sannier, A. Aubel, and D. Thalmann. Real-Time Animation of Realistic Virtual Humans. *IEEE Computer Graphics and Applications*, 18(5):42–56, 1998.

[41] Andrew Koenig and Barbara E. Moo. *Accelerated C++: practical programming by example.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[42] R. Kulpa, F. Multon, and B. Arnaldi. Morphology-independent representation of motions for interactive human-like animation. *Computer Graphics Forum*, 24(3):343–351, 2005.

[43] Graig Larman. *Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design.* Prentice-Hall, 1998.

[44] Cal3D. 3D Character Animation Library. `http://home.gna.org/cal3d/`.

[45] Ming C. Lin and Stefan Gottschalk. Collision Detection Between Geometric Models: A Survey. In *In Proc. of IMA Conference on Mathematics of Surfaces*, pages 37–56, 1998.

[46] Magalí Longhi, Luciana Nedel, Rosa Viccari, and Margarete Axt. Especificação e Interpretação de Gestos Faciais em um Agente Inteligente e Comunicativo. In *SBC Symposium on Virtual Reality*, São Paulo, 2004.

[47] Nadia Magnenat-Thalmann and Daniel Thalmann. *Handbook of Virtual Humans.* John Wiley & Sons, 2004.

[48] George Mandler, James A. Russel, and Josi-Miguel Fernandez-Dols. *The Psychology of Facial Expressions.* Cambridge University Press, 1997.

[49] Stacy C. Marsella, David V. Pynadath, and Stephen J. Read. PsychSim: Agent-based Modeling of Social Interactions and Influence. In *Proceedings of the International Conference on Cognitive Modeling*, pages 243–248, 2004.

[50] Luís Morgado and Graça Gaspar. Abstraction Level Regulation of Cognitive Processing Through Emotion-Based Attention Mechanisms. In Lucas Paletta and Erich Rome, editors, *WAPCV*, volume 4840 of *Lecture Notes in Computer Science*, pages 59–74. Springer, 2007.

[51] Franck Multon, Richard Kulpa, and Benoit Bideau. Mkm: A Global Framework for Animating Humans in Virtual Reality Applications. *Presence: Teleoper. Virtual Environ.*, 17(1):17–28, 2008.

[52] Hansrudi Noser, Olivier Renault, Daniel Thalmann, and Nadia Magnenat Thalmann. Navigation for Digital Actors based on Synthetic Vision, Memory, and Learning. *Computers and Graphics*, 19(1):7–19, 1995.

[53] OgreOpcode. `http://conglomerate.berlios.de/wiki/doku.php?id=ogreopcode`.

[54] OpenGL. `http://www.opengl.org/`.

[55] OGRE. Object Oriented Graphics Rendering Engine. `http://www.ogre3d.org`.

[56] Tito Pagan. 2d textures and texture mapping. In Steve Rabin, editor, *Introduction to Game Development*, pages 705 – 717. Charles River Media, 2005.

[57] Ken Perlin and Athomas Goldberg. Improv: A System for Scripting Interactive Actors in Virtual Worlds. *Computer Graphics*, 30(Annual Conference Series):205–216, 1996.

[58] Rob Pooley and Perdita Stevens. *Using UML – Software Engineering with Objects and Components*. Object Technology Series. Addison-Wesley, 1999.

[59] A. S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In *MAAMAW '96: Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away*, pages 42–55. Springer-Verlag New York, Inc., 1996.

[60] D. S. Reis, L. P. Nedela, C. Freitas, and J. F. Hübner. Uma Arquitectura para Simulação de Agentes Autônomos com Comportamento Social. In *SBC Symposium on Virtual Reality*, pages 39–50, 2004.

[61] Craig W. Reynolds. Flocks, Herds, and Schools: A Distributed Behavioral Model. *Computer Graphics*, 21(4):25–34, 1987.

[62] Craig W. Reynolds. Steering Behaviors for Autonomous Characters. In *Game Developers Conference 1999*, 1999.

[63] Jeff Rickel, Stacy Marsella, Jonathan Gratch, Randall Hill, David Traum, and William Swartout. Toward a New Generation of Virtual Humans for Interactive Experiences. *IEEE Intelligent Systems*, 17:32–38, 2002.

[64] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1999.

[65] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[66] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000.

[67] Pedro Miguel Semião, Maria Beatriz Carmo, and Ana Paula Cláudio. Algoritmo de Visão para Humanos Virtuais. In *Actas da 2^{a} Conferência Nacional em Interacção Pessoa-Máquina, Interacção 2006*, pages 133–138, Outubro 2006.

[68] Pedro Miguel Semião, Maria Beatriz Carmo, and Ana Paula Cláudio. Implementing Vision in the IViHumans Platform. In *Ibero-American Symposium on Computer Graphics, SIACG 2006*, pages 56–59, July 2006.

[69] Mei Si, Stacy C. Marsella, and David V. Pynadath. Thespian: Using multi-agent fitting to craft interactive drama. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 21–28. ACM, 2005.

[70] M. Silvestre, M. Pinto-Albuquerque, M. B. Carmo, A. P. Cláudio, J. D. Cunha, and H. Coelho. Platform for the Generation of Virtual Environments Inhabited by Intelligent Virtual Humans, *student poster*. In *ACM ITiCSE'05*, page 402, Monte da Caparica, Junho 2005.

[71] Miguel Silvestre, Maria Pinto-Albuquerque, Maria Beatriz Carmo, Ana Paula Cláudio, João Duarte Cunha, and Helder Coelho. Arquitectura de Suporte à Geração de Cenas Animadas com Agentes Inteligentes. Technical Report DI-FCUL TR-04-7, Departamento de Informática da Faculdade de Ciências da Universidade de Lisboa, 2004.

[72] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[73] Adam Szarowicz, Juan Amiguet-Vercher, Peter Forte, Jonathan Briggs, Petros Gelepithis, and Paolo Remagnino. The Application of AI to Automatically Generated Animation. In *Advances in AI, Proceedings of the 14th Australian Joint Conf. on Artificial Intelligence*, pages 487–494. Springer LNAI, 2001.

[74] Adam Szarowicz and Peter Forte. Combining Intelligent Agents and Animation. In *AI*IA 2003 : advances in artificial intelligence*, pages 275–286, 2003.

[75] Unreal Technology. `http://www.unrealtechnology.com/`.

[76] IViHumans. **I**ntelligent **Vi**rtual **Humans**. `http://labmag.di.fc.ul.pt/virtual/`.

[77] J. A. Torres, A. Maciel, and L. P. Nedel. Uma Arquitectura para Animação de Humanos Virtuais com Racioncínio Cognitivo. In *SVR 2002 – Symposium on Virtual Reality*, pages 341–352, 2002.

[78] J. A. Torres, L. P. Nedel, and R. H. Bordini. Autonomous Agents with Multiple Foci of Attention in Virtual Environments. In *Proceedings of 17th International Conference on Computer Animation and Social Agents*, pages 189–196, 2004.

[79] Jorge A. Torres, Luciana P. Nedel, and Rafael H. Bordini. Using the BDI architecture to produce autonomous characters in virtual worlds. In *4th International Working Conference on Intelligent Virtual Agents (IVA) 2003*, pages 197–201. Springer Verlag, 2003.

[80] Xiaoyuan Tu and Demetri Terzopoulos. Artificial Fishes: Physics, Locomotion, Perception, Behavior. *Computer Graphics*, 28(Annual Conference Series):43–50, 1994.

[81] Branislav Ulicny and Daniel Thalmann. Crowd Simulation for Interactive Virtual Environments and VR Training Systems. In *Eurographics Workshop of Computer Animation and Simulation '01*, pages 163–170. Springer-Verlag, 2001.

[82] GTI-IA (DSIC UPV). Jgomas user manual. `http://www.dsic.upv.es/users/ia/sma/tools/jgomas/archivos/documentation/JGOMAS_User_Manual.pdf`.

[83] Spyros Vosinakis and Themis Panayiotopoulos. SimHuman: A platform for Real-Time Virtual Agents with Planning Capabilities. In *IVA 2001 3rd International Workshop on Intelligent Virtual Agents*, pages 210–223. SpringerVerlag, 2001.

[84] Spyros Vosinakis and Themis Panayiotopoulos. A Tool for Constructing 3D Environments with Virtual Agents. *Multimedia Tools Appl.*, 25(2):253–279, 2005.

[85] Michael Winikoff, Lin Padgham, and James Harl. Simplifying the Development of Intelligent Agents. In *AI2001: Advances in Artificial Intelligence. 14th Australian Joint Conference on Artificial Intelligence*, pages 557–568. Springer, LNAI, 2001.