ABSTRACT
Current Agent Toolkits are too restricting, by preventing their Users to choose the Agent’s Decision Making Process and Testing/Deployment Environment. I propose a flexible Framework capable of integrating Embodied Intelligent Agents with any Virtual Environment. By maintaining a clear and abstract structure of the interaction between the Decision Making Process Centre and the Environment, Framework Users will also be able to choose each Agent’s Decision Making Process. The Agent will always maintain its full feature specification, not mattering what Environment it is. This article describes the proposed Framework’s basic concepts, architecture and several implementation approaches, and presents Use Cases that will prove the soundness of the Framework and its capabilities.

Keywords
Agent, Environment, Simulation, Coherency, Actuator, Sensor

1. INTRODUCTION

1.1 Motivation and Objectives
Recently we’ve been able to witness a rapid expansion of Virtual Environments, each one more complex than the last one, due to the ease of content creation by their virtual community. Although this content is becoming more and more dynamic, the introduction of Intelligent Agents into this kind of Environment is still scarce.

This work presented here proposes a Framework that, by combining the flexibility and modularity of its components, allows for a rapid and easy change of Environment. It also enables a fast Embodied Agent development, enhancing the interaction level between Agents and User’s virtual representation known as Avatars.

By structuring the interaction between the Decision Making Process centre and the Agent’s Body, Sensors and Actuators, located in the Environment, this Framework allows the creation of any Agent, using any Decision Making Process or Environment the User wishes.

The popular virtual environment Second Life was chosen for the development of this Framework and, as a result, it comes with a pack of Sensors, Actuators and Bodies specifically designed for it.

1.2 Outline
This article will present a brief description of Second Life, followed by the Related Work, where several Agent Toolkits and Frameworks are analyzed. The Concept Model will show the key concepts used for the Framework as well as their relations.

The Architecture section will explore the Framework’s distribution and communication model. The Implementation section will show a layered structure for the Framework along with several problems and justified solutions. To prove the Framework’s capabilities, the Results section will present two Use Cases and their analysis.

2. Second Life
Second Life (1) was developed by Linden Labs (2) and released in 2003, having at the moment up to 20 million registered accounts. Users can access this virtual environment via an application called Second Life Viewer and interact with it via a visual representation of the User called Avatar. This Avatar can be modified to resemble almost any shape, creature or person.

Avatars can communicate with each other via public chat or instant messaging. Public chat is used to maintain a conversation between two or more Avatars that are within a certain range from the talking Avatar. Unlike public chat, instant messaging is not distance-constraint and can even be sent via email, in case the receiving Avatar is offline.

Second Life has its own virtual Economy, being its currency the Linden Dollar, represented by L$. Users can buy and sell objects, services and even own virtual land. A User is considered to be a Resident if the User owns virtual land and is a paying customer for at least three months. Avatars have an Inventory where they can store Sounds, Textures, Notecards, Scripts and Objects. Objects are composed of simple geometric shapes called Prims, short for Primitives. Users can easily modify Prims’ properties such as Scale, Rotation, Position, and Squew. They can also apply colors and textures to the Prims’ sides.

Linden Script Language (LSL) is an event and state driven programming language used by Second Life Objects’ Scripts, controlling their behavior. Even though it has a lot of server-side limitations, it allows communication between Avatars, Objects and even third-party applications outside the Second Life via HTTP and XML-RPC. LSL has a similar structure like C or JavaScript using integers, strings, floating points, vectors, quaternions and lists.

3. Related Work
There were two solutions for my work: either adopt an existing Framework or make one from scratch. Each solution has its advantages and setbacks.

By adopting a Framework, I would be also adopting its Agent concept, and there is a learning curve regarding all Framework operations. There is also the problem of communication between several already adopted tools, each one written in a different programming language. The clear advantage by adopting a Framework is that all tools are already made and ready to be used.

By creating a Framework, I can define the internal structure of the Framework itself, according to the specific problem at hand. I’d have greater control over the Framework, since I know and can change the Framework’s internal mechanics. It causes some setbacks, because I would take some time to define all the
necessary protocols, mechanics and tools as well as how they work with each other. Problems in the form of Errors and Bugs would emerge during all stages of the Framework’s development.

Before deciding which solution to take, I researched the available Agent Toolkits and Frameworks.

During each analysis, three main factors were considered: communication, learning curve and information support.

Communication between the Framework and the Environment is the most important aspect, since the bulk of Actuators and Sensors will be located in the Environment, separated from the Agent’s Decision Making Process. The adopted Framework needs to have HTTP or XML-RPC communication capabilities.

Learning curve is also an important factor due to my time-related restraints. The adopted Framework needs to be easy to learn and use, while providing the features needed to expand the current work.

Information support is directly related to the learning curve. Information in high quantity and quality, either being manuals, forums or even direct line of communication with the Framework’s creators would help the work at hand.

Several Agent Toolkits were found during research, however only few were considered relevant for my work. Toolkits like AdventNet SNMP Agent Toolkit (3), Agent Sheets (4) and Botql (5) are more related to Interface Agents for Web Site usage than Embodied Agent for Virtual Environments.

3.1 Agent Toolkits

NetLogo (6) is a programmable modeling environment with the capability of simulation emergent behaviors. Created in 1999 by Uri Wilensky and under constant development by the Center For Connected Learning and Computer-Based Modeling. Also contains a vast collection of tutorials and a well-documented manual. It uses a distinct language that doesn’t allow for the same flexibility as C# or Java, restraining the integration with the remaining tools and libraries. Also, by not being able to communicate via HTTP, it’s impossible to connect this Framework with Second Life.

JADE (7), created and developed by Telecom Italia, is a Framework for the development of Agent-based applications, specially designed for multi-agent systems. JADE also provides flexible communication architecture with a Message Queue for each Agent. If necessary, it can also provide a multi-thread solution, by supporting cooperative behaviors. For debug purposes, it has a Sniffer Agent that tracks all messages exchanged between Agents. JADE comes equipped with a great variety of communication and debug options. Provides a multi-thread system and multi-tasking capabilities, which is a plus for multi-agent environments. Since the Agent building process doesn’t allow for behavior reutilization, we need to change each individual Agent’s behavior to change the Agent Society’s behavior.

Zeus (8) is an Open-Source, Java written, Multi-Agent System developed by British Telecom Labs. It consists of an API, core generator, Agent and Society monitoring tools and programming documentation. A Zeus Agent comes with several features such as coordination engine for negotiation protocols and a task manager. All inter-Agent communication is done via string message exchange. Zeus also provides an Agent Facilitator which works like a Yellow Pages Service. You can monitor Agents individually or the entire Society, using the several graphical interfaces. According to the Zeus Technical Analysis Report, this Framework has two main problems: the lack of interfaces for the Agent’s subsystem definition and the abstraction level for the API methods were too low. By allowing high granularity code we gain flexibility but we overburden the programmers.

Open Agent Architecture (9) (OAA), created by the Stanford Research Institute, is a Framework designed for Agent Collaboration. OAA Agents can be defined, using several programming languages such as Java, Perl and Prolog. To increase communication efficiency between Agents, OAA provides an Agent Facilitator, much like the Yellow Pages services. OAA defines agents as “independent processes which communicate and cooperate with each other across a distributed network... [agents can] actively contribute to a computation as opposed to being only passive participants” (10). Although the OAA Framework provides mechanisms for concepts similar to objectives and plans, it doesn’t provide for alternate methods of reaching a goal even after it fails. For example, if the Facilitator tool allocates a task for repairing the Agent’s car and if the Agent’s callback method fails, the Framework cannot provide for another way to reach the goal.

Agent Builder (11) is an integrated toolkit for building intelligent software agents, comprised of two main components, both written in Java: Toolkit and the Run-Time System. The Agent Builder Toolkit provides tools behavior definition, debug and tests, making it an integrated environment for a fast and easy Agent development. The Run-Time System provides an Environment for Agent testing and execution. Agents created by the Agent Build communication using the language and communication protocols Knowledge Query and Manipulation Language (KQML). Agent Builder was the only non-Open-Source Framework to be analyzed. It’s an extremely complex and expensive Framework. Agent Factory, Zeus, Jade and OAA were developed in Java and, having all the necessary libraries in C#, the integration between them would create even more problems.

3.2 Frameworks and Libraries

ION (12) is an Open-Source Framework, developed by GAIPS at INESC-id, built on the goal of code reusability. It uses a discrete simulation environment and provides the necessary tools for maintaining the communication and synchronization between all elements. The four ION principles are: coherent information access, active and passive information gathering, conflict resolution, and dynamic configuration changes. The coherent information access principle dictates that, during the same update cycle, all simulation’s elements must have access to the same information. The conflict resolution principle dictates that we can define the method used to resolve each conflict. We can access information in a passive or active way. Active Information Access is common to all Frameworks, through the systematic variable access. The Passive Information Access is done through the notification of the change, making unnecessary the systemic variable access. Dynamic configuration changes principle allows for modifications to occur to elements while the simulation is still running.

libOMV (13) is a Client-Server model based library for .Net. It’s used for accessing and creating 3D Virtual Environments. It’s fully compatible with Second Life protocols and can be used to create and operate Second Life Avatars.
Both ION and libOMV are in a different category as the previous Frameworks. They’re not Agent Toolkits but, can be combined to provide a simple and fast access to the Virtual Environment using libOMV and provide a Framework with a synchronous simulation for Event propagation. Both Frameworks can be easily extended to create a new Agent Framework on top of them.

3.3 Second Life Related Work
During the Related Work research I came across some articles regarding the development of Intelligent Agents Frameworks for Second Life. Kathryn Merrick (14) (15) developed a Framework to cultivate emergent behavior on Agents that took the appearance of sheep. As I was developing this Framework, two similar works were also being developed in Japan, by Helmut Prendinger.

AstroSim (16) is Framework designed for the Second Life Environment, capable of representing star clusters and other astronomical phenomena in a tridimensional fashion.

Twin-World Mediator (17), also developed by Helmut Prendinger, uses a communication solution between his Framework and the Second Life Environment much similar to the one used by me. Twin-World Mediator is a Framework that establishes a direct link to real life agricultural-related Sensors and Actuators to its equivalent representation in Second Life.

This Framework maintains a direct line of communication with the National Agricultural Research Centre (NARC), receiving weather information, soil temperature and humidity levels, among other parameters and represents in the information gathered in Second Life, so Avatars can analyze it. Avatars can also control a remote camera to explore real life terrain.

All the previously shown works had exploratory and learning objectives in mind, regarding the capabilities of Virtual Environments for the fast and efficient information sharing.

Even though they started in Second Life, customizing virtual terrain for multinational companies, Electric Sheep Company (18) decided to take it to another level. In February 2007, Columbia Broadcasting System (CBS) network television, paid 7 million dollars to Electric Sheep Company to create a Virtual Environment for multinational companies, Electric Sheep Company (18) decided to take it to another level. In February 2007, Columbia Broadcasting System (CBS) network television, paid 7 million dollars to Electric Sheep Company to create a Virtual Environment that resembled Second Life, with the entire cast and set of CSI: New York, having won two Emmys as a result.

3.4 Summary
Having finished my Framework analysis, I concluded that both JADE and ION were the best Frameworks suited for the job at hand.

My final choices were extending ION’s capabilities and creating my Framework from it. I feel that, by adopting a pre-existing Framework, I’d also be adopting the Framework’s Agent, Society, Actuator, Sensor and Behavior definition, imposed by the Framework itself.

My work differs from the type of Agents and Frameworks that are currently available. These Frameworks define the Agent as an ethereal entity, with a set of tools and an identifier. My Agent’s concept is visual, embodied and interactive with the Environment, Avatars and Objects. All analyzed Agent Toolkits already had their built-in Decision Making Process Centre. One purpose of this work is to let the User decide how the Decision Process should be executed and facilitate it by structuring its interaction with the Environment.

In conclusion, my Framework will extend from ION, bridging the Decision Making Process Centre and the Virtual Environment, both chosen by the User.

4. Concept Model

4.1 Description
At a first glance, we can organize the concepts into three categories: Decision Making Process Centre, Framework and Environment. Figure 1 shows the interaction between these categories. Framework bridges a connection between the Virtual Environment and the Decision Making Process Centre. If we explore the Environment we see Avatars, Acts and Objects. On the Framework side, we see Agents being comprised of Actuators, Sensors, a Mind and a Body.

![Figure 1 - Framework Distribution](image)

The Decision Making Process Centre (DMPC) remains in constant communication with the Framework, receiving information regarding the Virtual Environment. If a deliberation is made, the DMPC sends the resulting action to the Framework, which it relays it to the Virtual Environment so the Agent can execute it.

![Figure 2 - Agent Composition and Deliberation Sequence](image)

As show in Figure 2, the Agent is an entity composed of Sensors, Actuators, a Mind and a Body. Sensors, located both in the Framework and the Virtual Environment, are responsible for gathering information about the state of the Virtual Environment. Actuators, also located in the Framework and Virtual Environment, are responsible for changing the state of the Virtual Environment. The Mind, located in the DMPC, uses the information gathered by the Sensors to generate Actions, which are sent to the Actuators. The Body is the visual representation of the Agent in the Virtual Environment.

According to Russel and Norvig (19), Agents can be divided into five classes: simple reflex, model-based reflex, goal-oriented, utility-based and learning. Simple reflex Agent’s actions are based simply on the immediate perception of the Environment. For this type of Agent to work successfully, its Environment needs to be completely observable. A model-based reflex Agent works the same way a simple-reflex Agent, however, it keeps an internal representation of the Environment, allowing it to work well on partially observable Environments. A goal-oriented Agent works the same way a model-based reflex Agent but all Actions that will
lead the Agent to a Goal take priority. A utility-based Agent assigns a score to each Action, prioritizing those that increase its final score. A learning Agent is the most complex type of Agent, being capable of performing on unknown Environments, becoming more and more efficient as it collects data about the Environment.

The Mind is the Agent’s most important component because it’s what defines the Agent’s behavior. The five types of Agents described earlier are the result of five different types of Minds.

Sensors can extract information from the Environment in a passive or active way. A Sensor is considered to be passive if the Agent receives information from the Sensor when a change occurs in the Environment. A Sensor is considered to be active if the Agent uses the Sensor to retrieve information about the Environment, even when no change occurs. A camera is a good example for a passive Sensor, because it only records the information that the lens pick up. The sonar is a good active Sensor, because it sends a ping to the Environment, collecting information caused by that ping.

Actuators are responsible for the changes on the Environment. An Actuator represents an action an Agent can do. There are two types of actions: a sequenced action and an instant action.

The sequenced action has three states: Idle, Active and Paused. When an Actuator representing a sequenced action is created, the action’s state is Idle. When the action starts, the state changes to Active. If the action, while active, is paused, the state is changed to Paused. If the action is stopped, while Active, or simply terminates, the action’s state goes back to Idle. The Instant Action, which is used to represent simple actions, doesn’t need the state transition.

The Environment is the location where all interactions between Agents, Objects and Avatars occur. An Environment can be classified according to the following features: completely or partially observable, deterministic or non-deterministic, sequential or episodic, static or dynamic, discrete or continuous.

An Environment is completely observable if the Agent has full access to the state of the Environment, partially observable otherwise. An Environment is deterministic if the next state of the Environment can be calculated using the current state and action to be applied. An Environment is episodic if the taken action only depends on the current state of the Environment, otherwise it’s sequential. An Environment is considered to be static if the state remains unchanged while the Agent is deliberating on which action to take. An Environment is discrete if there are a finite number of states for the Environment, otherwise it’s continuous.

Considering the Environment selected for the Framework development, Second Life is partially observable, non-deterministic, sequential, dynamic and discrete.

Most Virtual Environments were created so that Avatars could interact with it. An Avatar is the visual representation of a User in the Environment. They can interact with each other using messages, animations or trading objects, among other features.

The most complex Environments allow Avatars to create objects for interaction purposes. Usually these objects are unmanned and don’t possess any communication or interaction capability.

While using Second Life, we can insert LSL Scripts into the Objects, giving them a partial capability for communication and interaction.

4.2 Summary

The main concepts used for the developed solution are: Framework, Decision Making Process Centre and Environment. The Virtual Environment selected for the development of the Framework is Second Life, which is a partially observable, non-deterministic, sequential, dynamic and discrete Environment. Being partially observable, we need a way to maintain the collected information.

Agents will be represented in the Environment by a Prim composed Object, named Body that, with the help of LSL Scripts, will be able to communicate and interact with other entities in Second Life. There are many ways of communication within the Environment, such as public chat, instant messaging and email.

5. Architecture

5.1 Description

As it was presented on the Concept Model, the entity called Agent is comprised of Sensors, Actuators, a Mind and a Body and is distributed between the Decision Making Process Centre, The Framework and the Environment, as you can see in Figure 3.

By leaving the choice of Decision Making Process and Environment to the Framework User, it was necessary to separate these components from the Framework.

Since the Decision Making Process is usually located in the same machine as the Framework, we assume a local communication between the two parties. However, since the researched Virtual Environments are comprised of server clusters, located all around the World, we need to assume a remote communication between the Framework and the Environment.

Figure 4 shows one of the early founded solutions for the communication of the Agent’s Components, presenting a scalability problem. For each new Agent, a new remote communication line had to be created and maintained between the Framework and the Environment.
In Figure 5, it’s shown the solution found to resolve the scalability problem. By implementing an Agent Manager in the Environment and a Broker in the Framework, we just need a single remote communication, independent from how many Agents have been created. Should this connection fail, all communication between the Framework and the Environment fails, however the previous solution had the same problem because the end-systems are the same.

When the Sensors located in the Environment receive new information, instead of relaying that information via an exclusive communication channel, it relays that information to the Agent Manager.

The Framework’s Broker works the same way as the Agent Manager. It receives information from all Agents’ Bodies, relaying it to each respective Entity. When an Entity receives an action from its Mind, it’s sent to the Broker, which relays it to the Agent Manager.

Having dealt with all communication problems between the various components, a coherency problem was detected.

Due to the fact of the Environment being partially observable and dynamic, Agents don’t have access to all information about the Environment and the information they currently know is probably being changed while they deliberate.

It was important to enforce a coherency on the actions of all Agents that belong to the Framework, so that they all have access to the same information while they deliberate. For example, two Agents of the same Framework, named Albert and Brooklyn. They both had access to the same information about the Environment at the same time, however Albert deliberated first and changed the Environment’s State while Brooklyn is still deliberating. Brooklyn’s actions are now not coherent with the current state of the Environment, because the generated actions are now based on outdated information.

Because of this problem, we need to introduce two new concepts: IONSIM and ESIM.

In Figure 6, a simulation was implemented on the Framework named ION Simulation (IONSIM) which contains the observable representation of the Environment’s State. This simulation also allows for information sharing among all Agents belonging to this Framework.

The Decision Process Making Centre will connect to this Simulation to access the available information and emit its actions.

The Environment Simulation (ESIM) is responsible for the Environment’s management. It’s completely independent from the Framework and has its own updating cycle for the Environment. Due to the fact that ESIM contains all information regarding the Environment’s State and is independent from the Framework, IONSIM is dependent on ESIM to keep a correct observable representation of the Environment’s State.

When the Decision Making Process Centre emits an action, that action needs to be executed in ESIM. The information resulting from executing the action needs to be transferred to the IONSIM, so the Framework can update its representation.

For this reason, we implemented a Client-Server communication model, where ESIM is the Server and IONSIM is the Client.

With this coherency enforcement, if the Mind decides to change the Agent’s Position, for example, the Agent’s Position will remain the same in IONSIM until that action is executed in ESIM and information regarding the position change is relayed back to IONSIM.

5.2 Summary
This Framework’s purpose is to create a bridge between the Decision Making Process Centre and the Virtual Environment.

There are two simulations running at the same time, IONSIM and ESIM. ESIM is the independent simulation managing the Virtual Environment and IONSIM is the Framework’s Simulation, dependent on ESIM that maintains the observable representation of the Environment’s State.

To create and maintain the remote communication between IONSIM and ESIM, a Broker, named Second Life Manager, was created on the Framework and an Agent Manager was implemented on the Virtual Environment.

The communication model between the Framework and the Environment is Client-Server, with the Environment, represented by the ESIM, being the Server and the Framework, represented by the IONSIM, the Client.

6. Implementation
6.1 Description
As shown Figure 7, there are five main components to the Framework’s Client Side. SLATE.Core, which contains IONSIM, defines all abstract concepts shown on the Concept Model section.

The Broker, named Second Life Manager, was implemented using a library called OpenMetaverse (libOMV). This library simulates the interaction made by an Avatar using the Second Life...
Environment, by recreating the packet communication with the Second Life Servers. The Environment’s Agent Manager is the extension of the Second Life Manager, creating the link between Framework and Environment.

SLATE.SecondLife is built using SLATE.Core and Second Life Manager, establishing the connection between the abstract concepts of SLATE.Core and the Actions and Events provided by the Second Life Manager.

Second Life Manager comes with a variety of Managers such as Inventory, Network, Communication, Action and Event Managers.

Inventory Manager is responsible for the creation and destruction Agents’ Bodies in the Environment. Network Manager is responsible for maintaining the session for the Avatar controlled by the Framework. Communication Manager is responsible for maintaining the communication between the Environment and the Framework, relaying information between the Agents’ Mind and the Agents’ Body.

The Action Manager is a collection of actions that can be executed in the Environment, used by the Framework’s Actuators. The Event Manager parses the information received by the Communication Manager and informs the Framework’s Sensors of changes in the Environment.

6.2 Problems and Solutions

6.2.1 Environment’s Limitations

Second Life uses a realistic approach regarding public chat. A message sent via public chat can only be “heard” up to 20 meters from the emitter. The range can be between 10 meter, by whisper, or 100 meters, by shouting. Given this limitation, the Avatar controlled by the Agent Manager needs to stay within 100 meter from every Agent created by the Framework. There is no range limit when using Instant Messaging, however Second Life Objects cannot receive Instant Messages.

To prevent server overloading, there are built-in time-delays in all LSL functions. They range from a 0.2 second delay from moving an Object to 20 seconds from sending an email.

Given these limitations, the LSL Scripts will be mainly used for features that are only available via LSL Scripts such as, movement using the physics engine provided by Second Life, Event detection and Communication.

6.2.2 Communication

One of the first detected problems when we started the communication between the Framework and the Environment was the detection of newly created Object within the terrain’s bounds.

The commands used to create a Second Life Object do not return any kind of information that could be used to detect the newly created Object or even if the Object was actually created.

To prevent this problem, when an Object creation request is made, the creation request, containing Object shape, position, rotation, is stored temporarily. When a new Object appears on the terrain, a match is attempted between it and all the pending requests. If a match is made, the pending request is removed and the correct Object’s name is given.

There were also some communication problems while attempting to retrieve Inventory Information from the Avatar controlled by the Agent Manager. Sometimes, while retrieving the identification of an item in the Inventory, it either timed out or retrieved a null identification. To prevent this problem a loop had to be implemented. This loop is controlled by two variables, located in the configuration file, which determines how many attempts will be made to retrieve the item and how long will it wait for each attempt before a time out is issued.

![Figure 8 - Framework’s Class Hierarchy](image)

By implementing my Framework using the ION Framework, my core class, named CoreElement, inherited all synchronization mechanisms implemented by ION’s core class, Element.

A CoreElement has a list of CoreElements which it depends on, i.e., the CoreElement is ready to use when all of its dependencies are also ready to be used.

Figure 8 shows the Framework’s Class Hierarchy for the main classes used for building an Agent. An Agent is composed of a Body, Actuators and Sensors. An Agent is only ready to be used if its Body, Actuators and Sensors are also ready to be used.

![Figure 9 - CoreElement’s Life Cycle](image)

Figure 9 shows the CoreElement’s life cycle while in the IONSIM. When a CoreElement is created, it’s located outside the IONSIM. When it’s about to be added to the Simulation, it adds all of its dependencies. When all dependencies are ready, the CoreElement informs the Simulation that it’s ready to be used. When the CoreElement is about to be removed from the Simulation, it removes all of its dependencies.
### 6.2.3 Platform

One of the Framework’s features is information sharing. A CoreElement is able to listen for changes on another CoreElement. For example, if Agent Albert wants to know when Agent Brooklyn’s position is changed, he can listen for the Position Change Event that originates from Agent Brooklyn.

There are two distinct ways to enable or disable a Sensor. You can use the Enable and Disable mechanism, provided in every Sensor or the Add and Remove Sensor mechanism, provided in every Agent. When the Sensor is disabled, all Events are ignored and Environment monitoring, in case it’s an active sensor, is disabled.

When you use the Disable mechanism, the Sensor remains in the possession of the Agent. When you use the Remove Sensor mechanism, the disabled Sensor no longer remains attached to the Agent. Any Agent can pick up this removing of the Sensor.

The available Sensors are Communication, Interaction and Vision. The Vision Sensor detects the presence of entities within the Agent’s field of vision, raising Events indicating when an entity enters, leaves or changes position or rotation. The Communication Sensor raises Events whenever the Agent hears a message in the public chat. The Interaction Sensor raises Events whenever a collision occurs or when an Avatar tries to interact with the Agent by touching it.

There are two types of Actuators, as show on Figure 8. Actuator with State and Without State. An Actuator with State represents a sequenced action with the States Active, Inactive, and Paused, as previously presented in the Concept Model Section. The Actuator without State represents an action where the transition Inactive-Active is done within the same Simulation update cycle.

#### 6.2.4 Bodies

During the Framework’s development, an Actuator was created to animate the Agent’s Body. During the first phase, the animation was made via LSL Script, which manipulated each Prim of the Agent’s Body.

There were several problems with body part movement, and synchronization calculations due to my lack of LSL experience, so I decided to purchase a tool called Puppeteer. This tool uses the concept of an animation as a collection of slides, known as key frames and the built-in mechanism interpolates the transition between frames. By using this tool, I only needed to create the Start, Stop, Pause and Resume mechanisms on the Framework side.

However, this Animation Actuator limits the Users freedom. By using this Actuator, the Agent’s Body needs to be of a specific type, one with already the built-in key frames and this Body cannot be changed in any way. If the User decided to implement this Actuator in a non-prepared Body, it became useless.

Seeing this problem at hand, I decided to create a new Animation Actuator that allowed more control. I defined a Body Part hierarchy in the form of a tree, with just one Root.

There is a Father-Child relation between the tree elements, which dictate how the transformations are propagated throughout the tree.

#### Figure 10 - Body Part Hierarchy Example

Using Figure 10 as an example, if a rotation is applied to the Left Shoulder, the same rotation is applied to the Left Shoulder’s Child, and so on till there are no more Childs.

When the Father-Child link is established, the Child Body Part stores the transformation needed to change its Father position and rotation into its position and rotation.

I kept the same Animation concept as the Puppeteer tool, by using Key Frames and an interpolation calculator between them.

During the test phase, it became clear that the constant update for each Body Part of each Agent per Update Cycle was causing a strain in communication between Framework and Environment.

Several optimization methods were implemented to diminish communication traffic, such as a threshold update value, which prevents moving a Body Part when the difference between the current and the future value is below the threshold value.

Even with all optimization mechanisms implemented, this Animation Actuator was still slower than the one by just using the Puppeteer tool. So I decide to provide the Framework with both and let the User decide which one to use: a faster but restraining, or a free but slower Animation Actuator.

### 6.3 Summary

This Framework is comprised of two Cores: Slate.Core and the Broker. Slate.Core contains all the abstract definitions for all Simulation’s Elements. The Broker contains all the necessary tools to establish a link between the Framework and the Virtual Environment. IONSIM, located in Slate.Core, was built with the ION Framework and it’s used to maintain a representation of the Environment’s observable state.

The Broker, named Second Life Manager, was built with the libOMV library. Slate.SecondLife is the layer created by linking Slate.Core with the Second Life Manager, establishing the link between the abstract concepts and the actions and events available for the Second Life Environment.

During the development of the Framework, there were some problems that were divided into four categories: Environment, Communication, Platform and Bodies.

LSL, the script programming language provided by the Environment, has a lot of built-in server-loading prevention mechanisms such as delays from 0.2 seconds up to 20 seconds. These and other limitations made the Framework responsible for a
lot of the Object Management, leaving only the physics engine movement and the sensor information collection in the Environment.

There were implemented several fail-safe loop retrieval mechanisms to retrieve Inventory Information, Object creation, detection and tagging, with easily changed parameters via the configuration file or done while in execution.

Regarding Platform problems, CoreElement was implemented using a strict and simple life-cycle, dependency management to enforce the synchronization.

The Agent’s Body problems were somewhat complex, that a category was created just for them. It was necessary to implement a Body Part hierarchy to apply recursively the transformations applied to a single Body Part. There are two Animation Actuators, one that is fast but restricts the User’s choice of Agent’s Body; and one that it’s free of restraints but it’s slower and generates more communication traffic.

7. Results
To prove the Framework’s capabilities, two Use Cases were made.

Boids is the first Use Case, considered to be a load test, testing the communication capabilities between the Framework and the Environment. For each Agent, an action is sent from the Decision Making Process Centre to the Environment, via Framework, each Simulation’s update cycle.

Simon Says is the second Use Case, testing the interaction between Avatars and Agent in the Environment. This Use Case also allowed further testing of Communication and Vision Sensors as well of Movement and Transport Actuators.

7.1 Use Case – Boids
7.1.1 Description
Boids is a computational model for coordinated animal-like movement, developed in 1986 by Craig Reynolds. It’s a good example of emergent behavior. When a simple set of rules applied to each Agent, the Agent-comprised society starts showing a complex behavior. The Decision Making Process is purely reactive, i.e., Agent doesn’t have any internal representation of the Environment and act solely on the current available information.

Figure 11 shows the three main rules applied to an Agent: separation, alignment and cohesion. Separation Rule steers the Agent to avoid crowding local flock mates. Alignment Rule steers the Agent toward the average heading of local flock mates. And Cohesion Rule steers the Agent to move toward the average position of local flock mates.

For this Use Case a fourth rule was enforced, a Boundary Rule, steering the Agent to stay within the bounds of the terrain. The number of Agents used for this Use Case ranged from five to twenty.

7.1.2 Analysis
Although the Agents didn’t have much room to move around, the resulting emergent behavior was the expected one. This Use Case was tested several times, with different configurations regarding the Separation, Alignment, Cohesion factors and Agent quantity, returning different results.

Due to the fact that, when Agents move using the physics engine in Second Life, they tend to collide with one another, the Agents had a tendency to get stuck in corners. Even though they had a command to move to a designated position, the collisions prevented them from reaching it. By increasing the Separation factor, this problem was minimized. The optimal number of Agents for the terrain’s dimension was ten. Lower than ten, the Agents formed small groups that never intercepted. Above ten Agents, they had a tendency to get stuck in corners, even with a high Separation Factor.

7.2 Use Case – Simon Says
7.2.1 Description
Simon Says is a simple Use Case, where one or more User controlled Avatars order Agents to follow them or to pick up other Agents, using a reactive decision process with state.

Agents can follow any Avatar that uses the command follow with the syntax <agentName> follow start|stop. Avatars can also make an Agent grab another Agent and bring it to where the Avatar is using the command fetch with the syntax <agentName> fetch <targetAgent>.

The commands may seem simple, but they were specifically created to test the Framework capabilities regarding Event treatment.

For this Use Case, the Agents were placed in specific positions, so that not all Agents would be able to hear the same messages from public chat. This was meant to test the Communication Sensor as well. Also, all available Sensors were active on all Agents, to test the information gathering and Event generation from the Environment to the Framework.

7.2.2 Analysis
All tests were successful, with all Agents behaving within the expected. The main purpose of this Use Case was to test the Framework how would react with the high amount of information being generated from the Environment, by having all Agents with active Sensors.

7.3 Summary
Two Use Cases were created to test the Framework’s capabilities as well as Avatar/Agent interaction in the Environment.

The first Use Case is the classic application Boids, which applies a reactive decision process to an Agent, as well as a simple set of rules that allows the Agent to move around within the terrain bounds.

The second Use Case adds the Avatar/Agent interaction component, allowing Avatars to control Agents through simple commands.

Both returned satisfactory results, validating the expected Framework’s capabilities.

8. Conclusion and Future Work
With this work, I developed a Reusable Framework supporting the development of Embodied Intelligent Agents, while allowing the User to choose Virtual Environment and Decision Making Process.
The Framework creates a link between the Virtual Environment and the Decision Making Process Centre. To ease the Agent’s development, this Framework also provides a pack of Sensors, Actuators and Bodies for the Virtual Environment Second Life. To prevent scalability problems, two Managers were created. The Broker, located in the Framework and the Agent Manager, located in the Environment.

A Simulation, called IONSIM, was implemented in the Framework, creating a representation of the observable Environment, allowing for the Agents to share information.

Two Use Cases were made to test the Framework’s capabilities. Use Case Boids was a load test and Use Case Simon Says was an interaction test as well as an Event treatment test. Both Use Cases performed well. My work differs from the ones presented in Related Work section, due to the fact that I’m allowing the User to create and deploy an Embodied Agent using a Virtual Environment and Decision Making Process of his choosing.

Although the Framework’s development took place in Second Life, this Framework can adapt to any Virtual Environment. For example, to adapt this Framework to Active Worlds (20), you just need to switch the current Broker, named Second Life Manager, for one that connects with this new Virtual Environment. After creating a new layer that connects the abstract concepts of Slate.Core to this new Broker, Slate.ActiveWorlds is ready to use. There is still room for improvement regarding Agent’s Body’s animations. The ideal was the creation of a simple to use script that kept all the animation computation on the server side, reducing the communication between the Framework and the Environment while the animation was playing to a bare minimum. The Agent’s Body creation process could also be made simpler, enabling some sort of import from known 3D modeling software.

9. References

8. Fonseca, Steven, Griss, Martin and Letsinger, Reed.
   Evaluation of the ZEUS MAS Framework. s.l. : HP -
10. International, SRI. Open Agent Architecture (OAA)
12. ION Framework --- A Simulation Environment for
    Worlds with Virtual Agents. Vala, Marco, et al. s.l. :
13. Library OpenMetaVerse. [Online]
14. Motivated reinforcement learning for adaptive
    characters in open-ended simulation games. Merrick,
    Kathryn and Maher, Mary Lou. s.l. : ACM International
15. Motivated reinforcement learning for non-player
    characters in persistent computer game worlds. Merrick,
    Kathryn and Maher, Mary Lou. s.l. : ACM International
16. AstroSim: Collaborative visualization of an astrophysics
    simulation in Second Life. Nakasone, Arturo, et al. s.l. :
17. Co-presence, collaboration, and control in
    environmental studies. A Second Life based approach.
    Attasiriluk, Songpol, et al. s.l. : Virtual Reality J, 2009.9,
    Vol. 13.
19. Russel, Stuart J. and Norvig, Peter. Artificial