UNIVERSIDAD CARLOS III DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

PROYECTO DE FIN DE CARRERA

Acceso manual e interfaz gráfica para el juego AI-LIVE

Alumno: Iván Uzquiano Mateo
Tutores: Susana Fernández Arregui
Javier Ortiz Laguna
Año: 2010

Agradecimientos

A mis tutores, Javier Ortiz, por adentrarme en el mundo de las interfaces gráficas, y Susana Fernández, por haberme enseñado las técnicas de inteligencia artificial basada en reglas, que me han servido para la comprensión de la estructura y el funcionamiento del juego AI-LIVE.

A los iniciadores del proyecto AI-LIVE, que han proporcionado una base sobre la cual he podido realizar este Proyecto de Fin de Carrera.

A los actuales y futuros contribuyentes del proyecto AI-LIVE, por seguir desarrollando y mejorando este juego tan interesante.

Y por último, pero no por ello menos importante, a mi familia y amigos, por apoyarme en esta etapa final de mi carrera.

Muchas gracias a todos por vuestro apoyo.

Índice de contenido

Índice de figuras

1. Introducción

Este capítulo muestra un resumen sobre el ámbito en el que se ha desarrollado este Proyecto de Fin de Carrera, su objetivo, y cómo está organizado este documento.

1.1. Entorno del Proyecto de Fin de Carrera

Este Proyecto de Fin de Carrera (Acceso manual e interfaz gráfica para el juego AI-LIVE) consiste en la creación de un cliente GUI (*Graphic User Interface* – Interfaz Gráfica de Usuario) que implemente una interfaz gráfica utilizando un motor gráfico 3D. Esta interfaz gráfica debe permitir al usuario observar cómo interactúan los personajes con su entorno en el juego AI-LIVE, desarrollado anteriormente como Proyecto de Fin de Carrera por otros alumnos de la Universidad Carlos III de Madrid.

AI-LIVE es un juego de simulación social, basado en el videojuego *Los Sims*, que fue creado con el objetivo de simular situaciones del mundo real. En el juego, los personajes, denominados actores, toman decisiones para realizar determinadas acciones dentro del entorno donde se encuentran. Estas decisiones se ordenan según las características e implementación propias de cada personaje.

El juego sigue el modelo cliente-servidor, de manera que el servidor es el encargado de asignar los turnos a los clientes. También éste se encarga de actualizar el estado de las entidades y los actores según las acciones solicitadas y enviárselo a los clientes para que puedan desarrollar su labor.

Además, el servidor consta de un motor emocional, encargado de controlar las emociones, los gustos y las relaciones de cada actor cuando establece una comunicación verbal con otro actor dentro del escenario.

Hasta el momento existen cuatro tipos de clientes: el cliente CLIPS decide qué acción quiere realizar utilizando un sistema de IA (Inteligencia Artificial) basado en reglas, el cliente Prodigy trata de solucionar un problema mediante un planificador de tareas, el cliente manual está basado en el cliente CLIPS y en él es el usuario el encargado de elegir la acción que desea realizar, y el cliente GUI es el encargado de mostrar la interfaz gráfica 2D que representa los estados del juego.

1.2. Cometido del Proyecto de Fin de Carrera

Inicialmente AI-LIVE tenía desarrollado un cliente GUI con una interfaz gráfica en 2D implementada en lenguaje C. El propósito de este Proyecto de Fin de Carrera es desarrollar un cliente GUI que implemente una interfaz gráfica en 3D, que introduzca mejoras en la visualización respecto a la interfaz gráfica 2D anterior, permitiendo representar el comportamiento de los actores en el juego bajo una perspectiva más similar a la realidad.

Para la implementación de la interfaz gráfica se ha utilizado una herramienta de software libre para el desarrollo de aplicaciones de visualización 3D en tiempo real. Concretamente, esta herramienta es el motor gráfico OGRE3D, que utiliza una interfaz de programación en lenguaje C++ para la codificación de aplicaciones, siendo de gran utilidad en el desarrollo de este tipo de programas gráficos, al tratarse de un lenguaje orientado a objetos.

La función principal de la interfaz gráfica es, una vez que el cliente GUI ha recibido el estado del juego enviado por el servidor en el turno de cada cliente, generar las entidades en caso de que sean nuevas, actualizarlas si han sido modificadas o eliminarlas si ya no existen. Este proceso requiere una sincronización perfecta entre el servidor y el cliente GUI, ya que si no existiera tal sincronización, se producirían errores en la representación gráfica.

1.3. Estructura del documento

Capítulo 1 - Introducción: se expone un breve resumen al entorno del Proyecto de Fin de Carrera, su cometido, y cómo está estructurado este documento.

Capítulo 2 - Estado del arte: se ofrece información sobre la generación de gráficos 3D, los tipos de APIs gráficas, las herramientas de desarrollo de aplicaciones 3D, y los videojuegos de simulación de relaciones sociales.

Capítulo 3 - Objetivos: se definen las metas que se desean conseguir con la realización de este Proyecto de Fin de Carrera.

Capítulo 4 - Trabajo realizado: se describe el trabajo realizado para la elaboración del Proyecto de Fin de Carrera, como son la arquitectura de AI-LIVE, el modelo de conocimiento del juego AI-LIVE, la descripción de los módulos principales, el manual de usuario y el manual de referencia.

Capítulo 5 - Resultados: se proporciona información sobre los escenarios creados para las pruebas, las diferentes vistas de la interfaz gráfica y las acciones de los actores mostradas por el cliente GUI.

Capítulo 6 - Conclusiones: se muestran las resoluciones a las que se ha llegado tras el estudio del juego AI-LIVE, la implementación del cliente GUI y la utilización de OGRE3D como motor gráfico.

Capítulo 7 - Futuras líneas: se plantean posibles mejoras y ampliaciones para la interfaz gráfica implementada.

Capítulo 8 - Bibliografía: se contemplan las fuentes de información utilizadas para el desarrollo del Proyecto de Fin de Carrera.

2. Estado del arte

El objetivo de este capítulo es ofrecer información sobre el estado actual de la tecnología utilizada en la elaboración de interfaces gráficas en 3D, similares a la usada en este proyecto.

Principalmente se hará un repaso a la generación de gráficos 3D, los tipos de APIs gráficas, las herramientas de desarrollo de aplicaciones 3D en tiempo real, y los videojuegos de simulación de relaciones sociales.

2.1. Gráficos 3D

Un gráfico en tres dimensiones (3D o tridimensional), según *[GRÁFICOS_3D]*, se genera mediante un proceso de cálculos matemáticos sobre entidades geométricas tridimensionales producidas por un ordenador, cuyo objetivo es crear una proyección en dos dimensiones (2D o bidimensional) para ser visualizada.

La diferencia entre los gráficos 3D y 2D es el número de dimensiones que contiene cada uno. Los gráficos 3D tienen tres dimensiones (longitud, profundidad y altura), permitiendo una representación volumétrica, mientras que los gráficos 2D sólo tienen dos (longitud y profundidad), permitiendo una representación plana.

En las posteriores subsecciones, se van a describir las diferentes fases para la creación de gráficos 3D.

2.1.1. Modelado

La fase de modelado consiste en dar forma a los objetos que posteriormente serán representados en la escena.

Existen diversos tipos de geometría para modelar, algunos de ellos son:

- NURBS (*Non Uniform Rational B-Splines*): se utiliza para representar curvas y superficies. Son representaciones matemáticas de geometría 3D capaces de describir cualquier forma con precisión.

- Modelado poligonal o de subdivisión de superficies: método para representar una superficie a través de una malla poligonal menos detallada. Es el tipo de modelado más utilizado.

- Modelado basado en imágenes: consiste en transformar una imagen en un objeto 3D mediante el uso de diversas técnicas. Es el tipo de modelado menos utilizado.

Algunas herramientas utilizadas en el modelado de objetos 3D son: Autodesk 3D Studio Max, Blender, Autodesk Maya, 3D Canvas,…

Un ejemplo de modelado de una cabeza humana se muestra en la Figura 1.

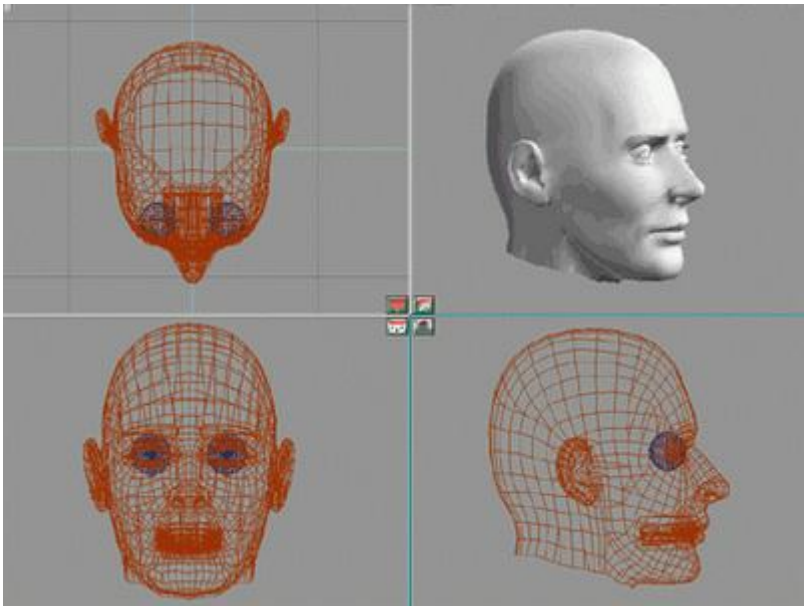

Figura 1: Ejemplo de modelado.

2.1.2. Animación

Una vez terminado el proceso de modelado, es necesario realizar la animación del modelo 3D construido para dar sensación de movimiento. Para eso se lleva a cabo la fase de animación, que consiste en definir la descripción del estado de los objetos de la escena en los diferentes instantes de tiempo.

Hay diversos métodos de animación, como son:

- Transformaciones respecto a los ejes X, Y, Z:
  - Rotación: giro del objeto sobre uno o varios de los ejes.
  - Escala: modificación del tamaño del objeto.
  - Traslación: modificación de la posición del objeto en la escena.

- Transformaciones de la forma:
  - Mediante esqueletos: un esqueleto es una estructura capaz de modificar la forma y el movimiento de un objeto asociado a éste.
  - Mediante deformadores: como son las cajas de deformación u otros.
  - Dinámicas: para simular texturas en movimiento, como son la ropa, el pelo, etc.

Un ejemplo de animación de un esqueleto humano se muestra a continuación en la Figura 2.

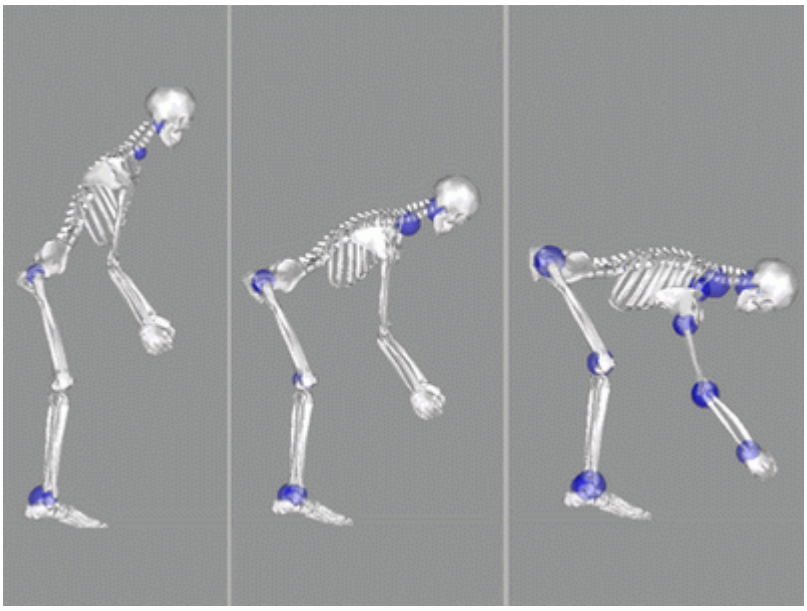

Figura 2: Ejemplo de animación.

2.1.3. Renderizado

Tras la animación del modelo 3D, es necesario asignarle texturas y luces al modelo y representarlo en un entorno bidimensional, para poder visualizarlo correctamente en un monitor. Es por ello por lo que se realiza la fase de renderizado, que consiste en generar una imagen 2D a partir de la escena 3D modelada.

El renderizado utiliza principalmente cuatro técnicas:

- Reflexión: consiste en cómo la luz interacciona con la superficie de un objeto en punto dado.

- Texturizado: consiste en definir la distribución de los diferentes tipos de reflexión a través de la superficie de un objeto. Para ello se utilizan unos materiales denominados *shaders*[1].

- Transporte: describe cómo la iluminación de una escena va de un lugar a otro.

- Proyección: define cómo se deben representar los objetos 3D en dos dimensiones.

Un ejemplo de renderizado de un ser humano corriendo se muestra posteriormente en la Figura 3.

[1] Shader: conjunto de instrucciones gráficas utilizadas por el acelerador gráfico, las cuales se encargan de proporcionar el aspecto final de los objetos.

Figura 3: Ejemplo de renderizado.

2.2. APIs gráficas

Las APIs (*Application Programming Interface* – Interfaz de Programación de Aplicaciones) gráficas proporcionan una abstracción software a la GPU (*Graphics Processing Unit* – Unidad de Procesado de Gráficos), y aprovechan el hardware de aceleración de gráficos 3D perteneciente a la tarjeta gráfica, haciendo así un uso más efectivo de los recursos y mejorando la calidad de los gráficos.

Según la definición proporcionada en *[API]*, una API gráfica es el conjunto de funciones y procedimientos que ofrece una determinada biblioteca para ser usada por un software de generación de gráficos como una capa de abstracción. Esto permite al programador utilizar las funciones y procedimientos mencionados sin la necesidad de conocer su implementación, pudiendo elaborar así una interfaz gráfica programada a muy bajo nivel.

Los dos tipos de APIs gráficas más importantes se detallan en *[SEOANE, 2007]*, y se van a comentar en las siguientes subsecciones.

2.2.1. OpenGL

OpenGL (*Open Graphics Library*) (http://www.opengl.org) es una especificación estándar y libre, desarrollada por Silicon Graphics Inc. (SGI) en 1992, que define una API multilenguaje y multiplataforma para implementar aplicaciones que generen gráficos 2D y 3D.

Una especificación es un documento que describe un conjunto de funciones y su comportamiento exacto. A partir de ella, los fabricantes de hardware crean implementaciones, es por ello que aunque la especificación es libre, la implementación desarrollada por los fabricantes de ésta no suele serlo.

OpenGL se utiliza en campos como CAD (*Computer Aided Design* – Diseño Asistido por Computador*)*, realidad virtual, representación científica y de información, simulación de vuelo y desarrollo de videojuegos.

2.2.2. Direct3D

Direct3D es parte de Microsoft DirectX (http://www.microsoft.com/games/en-US/aboutGFW/pages/directx.aspx), una API disponible tanto en los sistemas Windows de 32 y 64 bits, como para sus consolas Xbox y Xbox 360 en la programación de gráficos 3D.

El objetivo de esta API es facilitar el manejo y trazado de entidades gráficas elementales, como líneas, polígonos y texturas, en cualquier aplicación que utilice gráficos en 3D, así como efectuar de forma transparente transformaciones geométricas sobre dichas entidades.

Pese a que DirectX tiene una especificación cerrada al público, se trata de un software gratuito, a diferencia de OpenGL, cuya especificación es completamente libre.

Direct3D se usa principalmente en los videojuegos, donde el rendimiento es fundamental.

2.3. Herramientas para el desarrollo de aplicaciones 3D

Las herramientas para el desarrollo de aplicaciones 3D son una nueva forma de procesar y trabajar la información vectorial, permitiendo modelar matemáticamente los elementos de la realidad en sus tres dimensiones.

Este tipo de herramientas permiten la interacción del usuario con las aplicaciones implementadas, ofreciendo un mayor realismo y una mejor calidad de visualización. Asimismo, son capaces de proporcionar a los gráficos generados niveles de iluminación y sombras, efectos de cámaras, animaciones, soporte para audio y vídeo, y una gran gama de posibilidades además de las ya mencionadas.

A diferencia de las APIs gráficas, ofrecen un nivel de abstracción mucho mayor a la hora de realizar una interfaz gráfica. Pero, a pesar de ello, la mayoría de estas herramientas utilizan las APIs gráficas como base en la implementación de aplicaciones.

Algunos campos en los que se utilizan herramientas para el desarrollo de aplicaciones de visualización 3D en tiempo real son: videojuegos, simulación, entretenimiento, educación, divulgación, visualización científica, etc.

Las herramientas para el desarrollo de aplicaciones 3D se pueden clasificar en tres grandes tipos, según *[SEOANE, 2007]*, definidos a continuación.

2.3.1. Frameworks gráficos

Los *frameworks* intentan facilitar el desarrollo de software, permitiendo a los diseñadores y programadores emplear más tiempo identificando requisitos software en vez de tratar los detalles de bajo nivel para proporcionar un sistema funcional.

Un *framework*, según *[FRAMEWORK]*, es un entorno de trabajo (programas, bibliotecas, lenguajes,…) que proporciona funciones de alto nivel para el desarrollo de aplicaciones. Representa una arquitectura de software, es decir, un conjunto de patrones y abstracciones que permiten modelar las relaciones de las entidades del dominio.

El *framework* gráfico más utilizado es Microsoft XNA (http://msdn.microsoft.com/es-es/aa937791.aspx), se trata de un grupo de bibliotecas basado en .NET Framework 2.0 que mejoran la productividad en el desarrollo de juegos para Windows y Xbox 360. Utiliza el API de Microsoft DirectX, está implementado en lenguaje C#, y aunque es gratuito, no se trata de software libre.

2.3.2. Scene graphs

Un *scene graph* o grafo de escena, según *[SCENE_GRAPH]*, es una estructura de datos jerárquica y reutilizable que describe los objetos de una escena y sus relaciones (tamaño, posición, orientación,…). Se centra en representar la escena y su visualización de forma eficiente.

Su estructura en forma de árbol (como en la Figura 5) permite que toda operación realizada en un nodo padre, afecte a todos y cada uno de sus nodos hijo, simplificando de esta manera la codificación para las transformaciones de estos últimos.

A diferencia de los *frameworks* gráficos, los *scene graphs* no representan un entorno de trabajo, sino una herramienta a utilizar para el desarrollo de aplicaciones 3D.

Figura 4: Escena de un videojuego en primera persona.

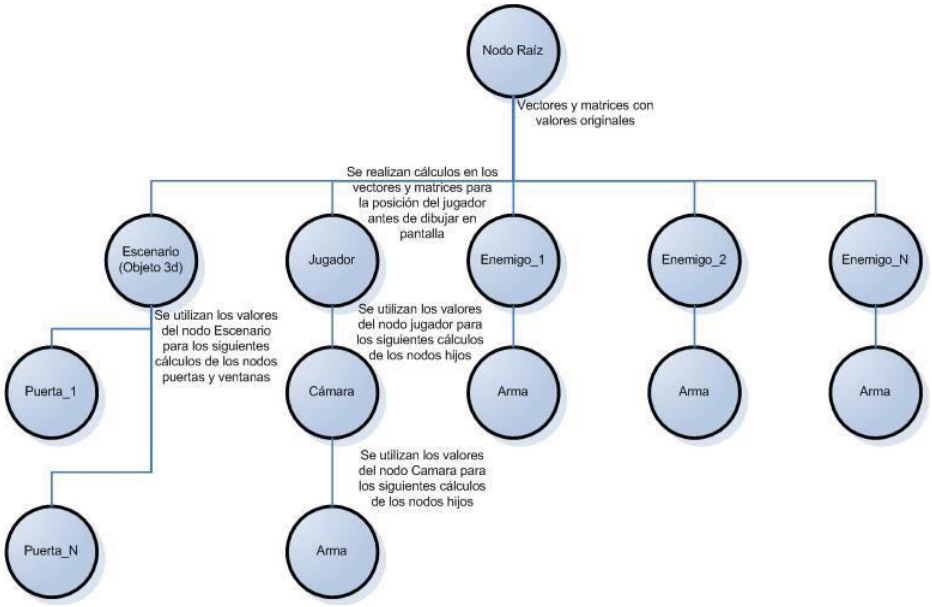

Figura 5: Scene graph que representa la escena de la Figura 4.

Seguidamente se van a comentar algunas herramientas que usan *scene graphs*.

2.3.2.1. NVIDIA SceniX

NVIDIA SceniX (http://developer.nvidia.com/object/scenix-home.html) es una librería de programación orientada a objetos utilizada en aplicaciones basadas en *scene graphs*.

Está implementada en lenguaje C++, puede utilizarse tanto para sistemas operativos Windows como Linux, está basada en el API de OpenGL, y pese a que es un software gratuito, no es libre.

En la Figura 6 se muestran ejemplos de visualización con NVIDIA SceniX.

Figura 6: Ejemplos de visualización con NVIDIA SceniX.

2.3.2.2. OpenSG

OpenSG (http://www.opensg.org) es un sistema de *scene graphs* portable para crear aplicaciones gráficas en tiempo real.

Está implementado en lenguaje C++, puede utilizarse en sistemas operativos Windows, Linux, Solaris y MacOS, utiliza el API de OpenGL, y se trata de software libre.

En la Figura 7 se muestran ejemplos de visualización en distintos campos con OpenSG.

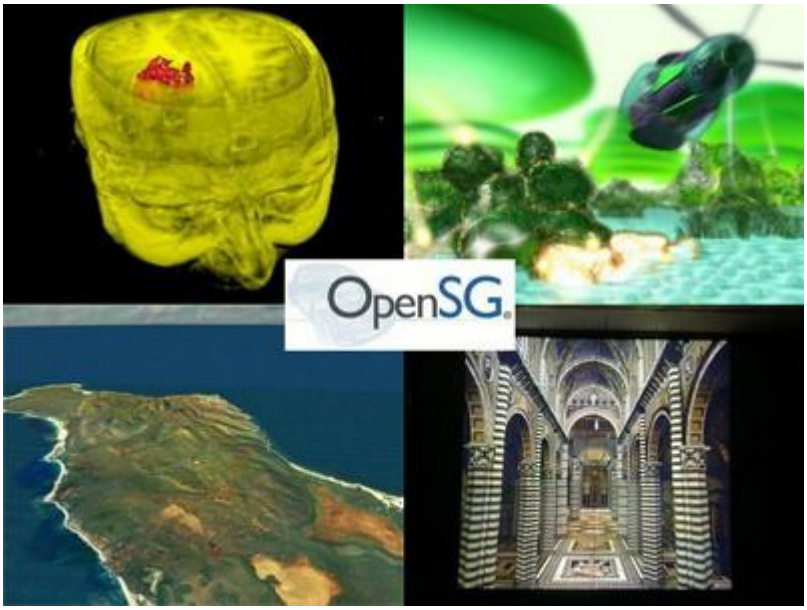

Figura 7: Ejemplos de visualización con OpenSG.

2.3.2.3. OpenSceneGraph

OpenSceneGraph (http://www.openscenegraph.org) es un conjunto de herramientas gráficas 3D de alta definición. No se debe confundir con OpenSG (citado anteriormente), cuyo nombre y características son similares pero pertenecen a proyectos distintos.

Está implementado en lenguaje C++, puede utilizarse en sistemas operativos Windows, Linux, Solaris y MacOS, IRIX, HP-Ux, AIX y FreeBSD, utiliza el API de OpenGL, y se trata de software libre.

En la Figura 8 se muestran ejemplos de visualización con OpenSceneGraph.

Figura 8: Ejemplos de visualización con OpenSceneGraph.

2.3.2.4. OpenGL Performer

OpenGL Performer (http://oss.sgi.com/projects/performer) es una potente interfaz de programación desarrollada por SGI para la creación de simulaciones visuales a tiempo real por parte de desarrolladores y para otras aplicaciones gráficas 3D de carácter profesional.

Está implementado en lenguaje C++, puede utilizarse en sistemas operativos Windows, Linux e IRIX, utiliza el API de OpenGL, y se trata de software comercial.

En la Figura 9 se muestran ejemplos de visualización en distintos campos con OpenGL Performer.

Figura 9: Ejemplos de visualización con OpenGL Performer.

2.3.3. Motores gráficos

Los motores gráficos se han convertido en una fuente de ayuda para los programadores de aplicaciones gráficas, ya que facilitan su diseño ofreciendo un conjunto de herramientas de desarrollo que reducen el coste, la complejidad y el tiempo de implementación.

Un motor gráfico, según *[GAME_ENGINE]*, es un conjunto de herramientas vinculadas entre sí que permiten desarrollar aplicaciones 3D. Además de la gestión de las escenas mediante *scene graphs*, incorpora otras funcionalidades como animación, scripting, IA, gestión de audio y vídeo, sistemas de red, editores, sistemas físicos,…

Pese a que los *frameworks* y los motores gráficos se pueden utilizar con la misma finalidad, estos últimos proporcionan una mayor funcionalidad a la hora de desarrollar aplicaciones 3D.

A continuación se van a describir las técnicas más importantes utilizadas por los motores gráficos, según *[TALÓN, 2005]*:

- Renderización: proceso de cálculo complejo destinado a generar una imagen 2D a partir de una escena 3D, con el fin de imitar un espacio 3D formado por estructuras poligonales, comportamiento de luces, texturas, materiales y animación, simulando ambientes y estructuras físicas verosímiles.

- Árboles BSP (*Binary Space Partitioning* – Particionado Binario del Espacio): estructuras de datos usadas para organizar objetos dentro de un espacio. Permiten determinar el orden de dibujado de polígonos para lograr la ocultación de una superficie, comprobar si un punto está en la parte sólida del modelo, así como detectar colisiones.

- Radiosidad: conjunto de técnicas para el cálculo de la iluminación global que trata de buscar el equilibrio de la energía que es emitida por los objetos emisores de luz y la energía que es absorbida por los objetos en el ambiente.

- *MipMapping*: sirve para cambiar la textura de un polígono en un objeto 3D según el ángulo de visión o las condiciones del juego. Se utiliza para conseguir mayor rendimiento, mejorando el nivel de resolución cuando el objeto está más cerca, y reduciéndolo cuando está más lejos.

- *Lightmaps*: estructura de datos que contiene el brillo de las superficies de una escena. Consiste en añadir una textura a una escena. Se suele utilizar para evitar el uso del cálculo de la radiosidad.

- *Ray tracing*: consiste en trazar rayos para determinar las superficies visibles con un proceso de sombreado que tiene en cuenta efectos globales de iluminación, como pueden ser reflexiones, refracciones o sombras arrojadas.

En general todos los motores gráficos tienen características muy parecidas, por lo que se van a exponer a continuación sólo los más importantes (tanto de software libre como comercial) junto con sus propiedades.

2.3.3.1. Crystal Space

Crystal Space (http://www.crystalspace3d.org/) es una herramienta portable de desarrollo de juegos 3D implementada en C++ que fue creada por Jorrit Tyberghein. Es un motor gráfico de los más antiguos.

Se puede utilizar en sistemas operativos Windows, UNIX y MacOS, utiliza el API de OpenGL o Microsoft DirectX, incluye también renderización 3D por software, así como la utilización de plugins, y se trata de un software libre.

Algunos videojuegos creados con Crystal Space son: *Bonez Adventures* (2004), *KeepSake* (2006), *PlaneShift* (2007),…

En la Figura 10 se muestran ejemplos de visualización de videojuegos creados con Crystal Space.

Figura 10: Ejemplos de visualización con Crystal Space.

2.3.3.2. Delta3D

Delta3D (http://www.delta3d.org/) es un motor gráfico que puede ser usado en juegos, simulación u otras aplicaciones gráficas, que está implementado en C++.

Tiene integrados como módulos OpenSceneGraph, Open Dynamic Engine (ODE), Character Animation Library (CAL3D) y la librería de audio OpenAL.

Se puede utilizar en sistemas operativos Windows y Linux, utiliza el API de OpenGL y se trata de un software libre.

Algunos videojuegos creados con Delta3D son: *DCOS Fire Fighter*, *SurfTacs*, *HuntIR* (2007),…

En la Figura 11 se muestran ejemplos de visualización creados con Delta3D.

Figura 11: Ejemplos de visualización con Delta3D.

2.3.3.3. OGRE3D (*Object-oriented Graphics Rendering Engine* 3D)

OGRE3D (http://www.ogre3d.org/) es un motor gráfico 3D flexible y orientado en escenas, implementado en C++, que está diseñado para hacer más fácil e intuitiva la producción de aplicaciones que utilizan aceleración hardware de gráficos 3D.

Se puede utilizar en sistemas operativos Windows, Linux y MacOS, utiliza el API de OpenGL o Microsoft DirectX, es compatible con una gran cantidad de plugins desarrollados, y se trata de un software libre.

Algunas aplicaciones y videojuegos creados con OGRE3D son: *Tomb Raider Viewer* (2004), *Ankh* (2005), *Pacific Storm* (2006), *The Legend of Beowulf* (2007),…

En la Figura 12 se muestran ejemplos de visualización creados con OGRE3D.

Figura 12: Ejemplos de visualización con OGRE3D.

2.3.3.4. RenderWare

RenderWare (http://www.renderware.com/) es un API 3D y un motor de renderización de gráficos para computador y videojuegos creado por Criterion Software.

Se puede utilizar en sistemas operativos Windows y MacOS, así como en el desarrollo de videojuegos para videoconsolas de Nintendo (GameCube y Wii), Sony (PlayStation2, PlayStation3 y PlayStationPortable) y Microsoft (Xbox y Xbox360).

Utiliza el API de OpenGL o el de Microsoft DirectX, y se trata de un software comercial.

Algunos videojuegos creados con RenderWare son: *Broken Sword IV: Sleeping dragon* (2003), *Call of Duty: Finest hour* (2004), *Commandos: Strike force* (2004), *Grand Theft Auto: San Andreas* (2004), Mortal *Kombat: Armaggedon* (2006), *The Godfather* (2006),…

En la Figura 13 se muestran ejemplos de visualización de videojuegos creados con RenderWare.

Figura 13: Ejemplos de visualización con RenderWare.

2.3.3.5. Torque Game Engine (TGE)

Torque Game Engine (http://www.torquepowered.com/products/torque-3d) es una versión modificada del motor 3D para el videojuego *Tribes 2* (2001) de Dynamix, creada por GarageGames e implementada en C++.

Se puede utilizar en sistemas operativos Windows, Linux y MacOS, y puede ser portado a videoconsolas Nintendo Wii y Microsoft Xbox360.

Utiliza el API de OpenGL o el de Microsoft DirectX, y se trata de un software comercial.

Algunos videojuegos creados con Torque Game Engine son: *The Odyssey: Winds of Athena* (2001), *Wildlife Tycoon: Venture Africa* (2005), *Marble Blast Ultra* (2006), *Dreamlords* (2007),…

En la Figura 14 se muestran ejemplos de visualización de videojuegos creados con Torque Game Engine.

Figura 14: Ejemplos de visualización con Torque Game Engine.

2.3.3.6. Unreal Engine

Unreal Engine (http://www.unrealtechnology.com/) proporciona una plataforma y herramientas para el desarrollo de videojuegos 3D de vanguardia, también utilizado para la realización de simulaciones militares. Fue creado por Epic Games para el juego *Unreal* (1998), está implementado en C++ y utiliza un tipo de script diseñado por la empresa creadora, llamado *UnrealScript*.

Se puede utilizar en sistemas operativos Windows, Linux y MacOS, así como en videoconsolas Dreamcast, Microsoft Xbox y Xbox360, Sony PlayStation2 y PlayStation3, y Nintendo Wii.

Utiliza el API de OpenGL o el de Microsoft DirectX, y se trata de un software comercial.

Algunos videojuegos creados con Unreal Engine son: *Unreal* (1998), *Harry Potter and the Philosopher's Stone* (2001), *Tom Clancy's Splinter Cell* (2002), *XIII* (2003), *Stargate Worlds* (2008),…

En la Figura 15 se muestran ejemplos de visualización de videojuegos creados con Unreal Engine.

Figura 15: Ejemplos de visualización con Unreal Engine.

2.3.3.7. CryEngine

CryEngine (http://www.crytek.com/) es un motor gráfico diseñado específicamente para el videojuego *FarCry* (2004), e implementado en C++, aunque fue desarrollado originalmente como una tecnología de demostración para NVIDIA.

Se puede utilizar en sistemas operativos Windows, así como en videoconsolas Microsoft Xbox y Xbox360, Nintendo GameCube y Sony PlayStation2, utiliza el API de OpenGL o el de Microsoft DirectX, y se trata de un software comercial.

Además de *FarCry*, se ha creado con CryEngine el videojuego *Crysis* (2007).

En la Figura 16 se muestran ejemplos de visualización de videojuegos creados con CryEngine.

Figura 16: Ejemplos de visualización con CryEngine.

2.3.3.8. Source

Source (http://source.valvesoftware.com/) es un motor gráfico 3D desarrollado por Valve Corporation.

Se puede utilizar en sistemas operativos Windows, así como en videoconsolas Microsoft Xbox y Xbox 360, y Sony PlayStation3, utiliza el API de Microsoft DirectX, y se trata de un software comercial.

Algunos videojuegos creados con Source son: *Counter-Strike: Source* (2004), *Half-Life: Source* (2004), *Vampire: The Masquerade – Bloodlines* (2004), *Day of Defeat: Source* (2005),…

En la Figura 17 se muestran ejemplos de visualización de videojuegos creados con Source.

Figura 17: Ejemplos de visualización con Source.

2.4. Videojuegos de simulación de relaciones sociales

Como se menciona en *[HISTORIA_VIDEOJUEGOS]*, la industria de los videojuegos ha evolucionado considerablemente en los últimos cuarenta años. Esta evolución ha transcurrido desde la utilización de tubos de rayos catódicos y ordenadores analógicos con osciloscopio, pasando por las máquinas recreativas (denominadas ARCADE), hasta llegar a la era actual de los ordenadores personales, que utilizan software y hardware especializado para su uso, sin olvidar las videoconsolas, diseñadas específicamente para la utilización de videojuegos.

Existe una gran variedad de videojuegos, como son los de acción, deportes, aventuras, estrategia, lucha, plataformas, etc. A pesar de ello, esta sección va a mostrar ejemplos de videojuegos de simulación o realidad simulada, pero sólo aquellos que tienen que ver con las relaciones sociales, ya que están implicados directamente con el objetivo de este Proyecto de Fin de Carrera.

Los videojuegos de simulación de relaciones sociales, también llamados de simulación social, consisten en el manejo de uno o más personajes virtuales que simulan la vida y el comportamiento de los seres humanos mediante la interacción con objetos y otros personajes de su entorno.

En las posteriores subsecciones se van a comentar los videojuegos de simulación social más relevantes.

2.4.1. Los Sims (The Sims)

Los Sims (http://www.portalmix.com/lossims/) es un videojuego de simulación humana representado en 2D, diseñado por Will Wright para Maxis, y distribuido para Electronic Arts en el año 2000. Es el primero de esta categoría, en el que los personajes tienen personalidad propia y se controlan individualmente de forma directa.

El videojuego consiste, según *[SIMS]*, en crear personajes, denominados *sims*, y asignarles unos rasgos físicos y de personalidad (pulcritud, extroversión, actividad, nivel lúdico y cordialidad) además de diseñarles una casa en donde vivir dentro de un barrio con su respectivo mobiliario.

Una vez hecho esto, el jugador deberá encargarse de satisfacer las necesidades básicas (hambre, energía, comodidad, diversión, higiene, sociedad, vejiga y entorno) de los *sims* que controla interactuando con los objetos de la casa, así como las necesidades económicas buscando y asignándoles un empleo en el cual podrán ir ascendiendo en su cargo según sus habilidades (creatividad, cocina, mecánica, estado físico, carisma y lógica), las cuales se pueden aumentar realizando diferentes acciones, y según el número de amigos que tenga.

Además, cada *sim* puede relacionarse con otros personajes, saludando, hablando, alabando o contando sus intereses (viajes, dinero, política, años 60, tiempo, deportes, música, aire libre y ejercicios). De esta manera pueden establecerse distintos tipos de relaciones, como amistad, romance, matrimonio, e incluso pueden tener hijos.

Los *sims* que no son controlados por el jugador son manejados mediante un programa basado en IA, el cual guiará sus comportamientos según las características asignadas a cada *sim*.

Pese a que no hay ningún objetivo final en el juego, se intenta maximizar la felicidad de los *sims*, aunque existe la posibilidad de que éstos mueran, debido a diversas circunstancias.

Las principales ventajas del videojuego son: la similitud con el mundo real y la gran variedad de acciones disponibles y de relaciones con otros personajes.

Algunas desventajas del videojuego son: los personajes sólo pueden estar en la casa, los personajes no crecen con el paso del tiempo, no existen días festivos,…

Tras el gran éxito de *Los Sims*, se desarrollaron siete actualizaciones para éste:

- *Los Sims: Más vivos que nunca* (*The Sims: Livin' large* - 2000): se añaden nuevos objetos, acciones y habilidades para relacionarse con los vecinos, así como nuevos trabajos y temas de conversación.

- *Los Sims: House party* (*The Sims: House* party - 2001): se añade la capacidad de realizar fiestas en casa, además de objetos y ropa de temática fiestera.

- *Los Sims: Primera cita* (*The Sims: Hot date* - 2001): los personajes ahora pueden salir de sus casas, yendo a la ciudad para salir por discotecas y tener citas en restaurantes románticos.

- *Los Sims: De vacaciones* (*The Sims: Vacation* - 2002): los personajes ahora pueden elegir distintos destinos para ir de vacaciones, como son la playa, la montaña o el campo.

- *Los Sims: Animales a raudales* (*The Sims: Unleashed* – 2002): permite adoptar una gran variedad de mascotas, además expande el barrio con la posibilidad de construir nuevas tiendas, parques, etc.

- *Los Sims: Superstar* (*The Sims: Superstar* - 2003): aparece la denominada Ciudad Estudio, donde los personajes pueden grabar discos, modelar, actuar, relajarse en los balnearios y conocer a gente famosa.

- *Los Sims: Magia potagia* (*The Sims: Makin' magic* - 2003): permite utilizar magia y aprender poderes, además se añade el Barrio Mágico.

En la Figura 18 se muestran capturas del videojuego *Los Sims*.

Figura 18: Capturas de Los Sims.

2.4.2. Los Sims 2 (The Sims 2)

Los Sims 2 (http://lossims2.ea.com/) es la secuela del videojuego de simulación social *Los Sims*, creado por Andres Cuartas para Maxis, representado en 3D, y distribuido por Electronic Arts en el año 2004.

En *[SIMS2]* se muestra que, a diferencia de *Los Sims*, en *Los Sims 2* los personajes pueden crecer con el paso del tiempo existiendo varias etapas de vida (bebé, infante, niño, adolescente, adulto y anciano). También se ha introducido el concepto de ADN virtual, lo que hace que los hijos se parezcan a los padres, y a otros familiares.

Además, se ha introducido la característica de que según la etapa de vida en la que se encuentre el *sim*, éste puede ir al colegio, a la universidad, trabajar, o jubilarse.

En este videojuego existen días festivos, como los fines de semana, en donde los *sims* ni trabajan ni van al colegio o a la universidad. Ahora los *sims* pueden realizar tareas más autónomamente, y son más conscientes de lo que ocurre a su alrededor.

Una nueva característica que implementa *Los Sims 2*, es el medidor de aspiraciones (familiar, conocimientos, romántica, dinero y popularidad), en donde entra en juego los deseos y miedos de cada *sim*. Estos puntos de aspiración se pueden canjear por objetos especiales.

Al igual que en *Los Sims*, se han creado varias expansiones para *Los Sims 2*:

- *Los Sims 2: Universitarios* (*The Sims 2: University* - 2005): se añade la etapa de la vida "joven adulto", nuevas metas en la vida, bromas e influencias.

- *Los Sims 2: Noctámbulos* (*The Sims 2: Nightlife* - 2005): se añade la opción de realizar citas, la aspiración de "placer", la química entre los personajes y la utilización de vehículos.

- *Los Sims 2: Abren negocios* (*The Sims 2: Open for business* - 2006): se añaden nuevos negocios, ascensores y robots.

- *Los Sims 2: Mascotas* (*The Sims 2: Pets* - 2006): permite la adopción de mascotas, y la posibilidad de ir a la tienda de mascotas.

- *Los Sims 2: Las cuatro estaciones* (*The Sims 2: Seasons* - 2007): se añade un sistema de clima, ropa para exteriores, el cambio de estaciones del año y la posibilidad de realizar pesca y jardinería.

- *Los Sims 2: Bon voyage* (*The Sims 2: Bon voyage* - 2007): permite realizar viajes, reservar hoteles, acudir a eventos de la región, y conseguir artículos.

- *Los Sims 2: Hobbies* (*The Sims 2: FreeTime* - 2008): se añade un sistema de hobbies con diferentes acciones.

- *Los Sims 2: Comparten piso* (*The Sims 2: Apartment life* - 2008): se añaden nuevos estilos de vida, magia y brujería.

En la Figura 19 se muestran capturas de *Los Sims 2*.

Figura 19: Capturas de Los Sims 2.

2.4.3. Singles: ¿En tu casa o en la mía? (Singles: Flirt up your life)

Singles: ¿En tu casa o en la mía? (http://www.singles-the-game.de/english/girls/index_girls.php) es un videojuego europeo desarrollado por Rotobee, representado en 3D, y publicado por Deep Silver en el año 2004, similar a *Los Sims*, pero con un objetivo específico, que dos compañeros de piso establezcan relaciones sexuales.

El videojuego, según *[SINGLES]*, muestra la vida de dos compañeros de piso elegidos por el jugador de entre un total de 15, con diferentes personalidades. Ambos deberán realizar las rutinas diarias, como cocinar, ver la televisión, limpiar, etc., así como conseguir un trabajo para poder subsistir. También pueden conocer a otros personajes, con los cuales pueden relacionarse y flirtear, invitándolos a su casa.

El jugador deberá realizar diferentes objetivos, el primero es satisfacer las necesidades básicas de los personajes que controla, el segundo es conseguir que los dos personajes entablen una conversación, y el objetivo final es que tengan relaciones sexuales el uno con el otro.

Al igual que ocurre en *Los Sims*, cuando el jugador no está controlando un personaje, éste realiza acciones seleccionadas mediante técnicas de IA, basadas en sus características y necesidades.

Como datos curiosos, en este videojuego los personajes nunca crecen ni mueren, permite establecer relaciones entre personas del mismo sexo, y no existe censura en los gráficos cuando los personajes están desnudos.

En la Figura 22 se muestran capturas del videojuego *Singles: ¿En tu casa o en la mía?*.

Figura 20: Capturas de Singles: ¿En tu casa o en la mía?.

2.4.4. Singles 2: ¿Tres son multitud? (Singles 2: Triple trouble)

Singles 2: ¿Tres son multitud? (http://www.singles2.com/englisch/index.html) es la secuela del videojuego *Singles: ¿En tu casa o en la mía?*, desarrollado por Robotee, publicado por Deep Silver y distribuido por Ubisoft en el año 2005.

Como nuevos aspectos en el juego, aparece un tercer personaje en la convivencia, además mejora los gráficos con respecto a su versión anterior, y añade nuevos contenidos, como poder visitar el bar donde los personajes pueden interactuar con otros.

El modo historia narra las aventuras de Josh, un músico que ha sido dejado por su novia y despedido de su trabajo, que decide mudarse de ciudad para dejar atrás su vida anterior.

Una vez llegado a la nueva ciudad, Josh encuentra un piso compartido para tres personas, en el que se encuentra con Kim, una joven asiática.

Casualidades del destino, ocurre que Anna, la exnovia de Josh, se ha mudado a la misma ciudad, y además a la habitación que queda libre en la casa donde Josh vive.

En el modo libre el jugador podrá crear sus propias historias de amor. Éste permite elegir a tres personajes entre un total de 17 para poder realizar todo tipo de relaciones, como son la amistad, el flirteo, el amor, etc., al igual que ocurre en *Singles: ¿En tu casa o en la mía?*.

En la Figura 21 se muestran capturas del videojuego *Singles 2: ¿Tres son multitud?*.

Figura 21: Capturas de Singles 2: ¿Tres son multitud?.

2.4.5. Second Life (SL)

Second Life (http://secondlife.com/) es un videojuego de software libre basado en un mundo virtual online 3D desarrollado por Linden Lab en el año 2003.

El videojuego, según *[SECOND_LIFE]*, está inspirado en el movimiento literario *cyberpunk*, más concretamente en uno de los mundos virtuales que aparecen en la novela de ciencia ficción *Snow crash* (1992), de Neal Stephenson.

El objetivo inicial de *Second Life* fue crear un mundo diseñado completamente por los jugadores, como el Metaverso[2] descrito por el autor de la novela, en donde los personajes pudieran interactuar, jugar, trabajar y comunicarse.

Cada jugador, denominado residente, debe crear su propio avatar, un personaje configurable con el que va a interactuar en el juego. Éstos podrán explorar el mundo virtual, conociendo a otras personas, socializándose, participar en actividades grupales, y tener sexo virtual, entre otras cosas.

A diferencia de otros juegos de simulación social como *Los Sims*, los avatares no tienen necesidades básicas, y la comunicación con otros usuarios se hace mediante un chat. Además, pueden crear objetos virtuales con los que intercambiar con otros personajes en un mercado abierto.

[2] Metaverso: entorno donde los humanos interactúan como avatares con otros semejantes y agentes software en un espacio tridimensional que actúa como una metáfora del mundo real.

Second Life tiene una economía muy particular con un mercado financiero que permite cambiar dólares Linden, la moneda utilizada en el videojuego, por dólares americanos reales.

El acceso e instalación del videojuego es gratuito, pero la compra y edificación de terrenos requiere de una cuota mensual de aproximadamente 10 dólares americanos.

Actualmente hay una gran cantidad de empresas y famosos que se han apuntado al mundo virtual de *Second Life*.

En la Figura 22 se muestran capturas del videojuego online *Second Life*.

Figura 22: Capturas de Second Life.

3. Objetivos

El objetivo principal de este Proyecto de Fin de Carrera es elaborar un cliente GUI que implemente una interfaz gráfica en 3D, de manera que permita al usuario observar cómo interaccionan los actores entre sí y con su entorno, proporcionando una visión que se asimile más a la realidad que la utilizada hasta ahora.

Anteriormente en el juego AI-LIVE se almacenaban las acciones que había realizado cada uno de los actores en un archivo de traza, desde el momento en que se conectaban al servidor hasta que finalizaban su ejecución.

Ésta era la única manera de saber cómo tomaban las decisiones los actores, creándose la necesidad de visualizar todo el archivo de traza de cada uno de los actores que habían intervenido en el juego para saber cómo habían influido las decisiones de unos actores sobre otros. Por lo que surgió la necesidad de obtener una interfaz gráfica que permitiera visualizar el comportamiento de cada uno de éstos a tiempo real.

Para ello, el servidor debe generar y enviar en el turno de cada cliente el estado actual del escenario al cliente GUI, pero no toda la información, sino sólo la referente a los aspectos gráficos, como son la posición, el tamaño y el estado de cada uno de los objetos y actores dentro del escenario, así como las acciones que realizan estos últimos.

Una vez hecho esto, el cliente GUI debe ser capaz de crear el escenario inicial y las entidades a partir del estado recibido en la primera iteración, y de actualizar el estado de éstas o generar un nuevo escenario con sus propias entidades en caso de haber un cambio de escenario, en posteriores iteraciones.

Hay que tener en cuenta que debe existir un control de conexión y sincronización entre el servidor y el cliente GUI para que no haya problemas en la actualización del estado de los objetos visualizados, por lo que es necesario crear un mecanismo de comunicación entre ambos que permita realizar este propósito.

Además, el cliente GUI debe proporcionar una serie de características, como son la portabilidad y la adaptabilidad, ya que se plantea que en un futuro el juego AI-LIVE pueda ejecutarse bajo sistemas operativos Microsoft Windows, aunque actualmente funciona bajo sistemas operativos Linux.

4. Trabajo realizado

En este capítulo se va a describir el trabajo realizado para la elaboración del cliente GUI que implementa la interfaz gráfica en 3D.

En las posteriores secciones se exponen: una introducción sobre cómo se ha desarrollado el Proyecto de Fin de Carrera; la arquitectura de AI-LIVE, en donde se describen generalmente los distintos módulos del juego, el protocolo de comunicación entre éstos y las entradas y salidas de cada módulo; el modelo de conocimiento de la aplicación, que representa el diagrama de clases con la definición de éstas, sus atributos y sus relaciones; la descripción detallada del cliente GUI implementado; el manual de usuario, y por último, el manual de referencia.

4.1. Introducción

Antes de la implementación del cliente GUI, ha sido necesario realizar un estudio previo sobre la arquitectura y el funcionamiento del juego AI-LIVE, utilizando la bibliografía *[PÉREZ, 2006]*, *[BENITO, 2007]* y *[JIMÉNEZ, 2008]*.

Posteriormente, se ha realizado un estudio de las posibles herramientas de desarrollo de aplicaciones 3D a utilizar y sus características, para elegir de entre ellas la más adecuada para implementar la interfaz gráfica perteneciente al cliente GUI. De todas ellas, se ha optado por el motor gráfico OGRE3D por las siguientes razones:

- Se trata de software libre, ya que posee una licencia GNU LGPL (*GNU Library General Public License* – Licencia Pública General para Librerías de GNU), por lo que no requiere ningún coste económico para su utilización.

- Es portable y adaptable, permitiendo su utilización tanto en sistemas operativos Microsoft Windows como en Linux, lo que es una gran ventaja, ya que en un futuro se plantea la portabilidad de la aplicación a Windows.

- Es capaz de utilizar tanto la API gráfica de OpenGL como la de Microsoft DirectX, dependiendo del sistema operativo en el que se esté trabajando.

- Proporciona una gran calidad gráfica y realismo en los objetos visualizados.

- Tiene una interfaz orientada a objetos clara, simple y fácil de usar, basada en el lenguaje C++.

- Posee una gran cantidad de documentación.

- Permite la incorporación de otras librerías.

- Le da soporte una gran comunidad de usuarios.

Debido a que el motor gráfico utiliza para la implementación de aplicaciones el lenguaje de programación C++, se ha requerido una iniciación a este lenguaje, utilizando la documentación *[ALTADILL, 2004] y [OSORIO, 2006]*.

Seguidamente, se ha procedido a la instalación y al aprendizaje de utilización de OGRE3D, mediante el libro *[JUNKER, 2006]* y la realización de los tutoriales mencionados anteriormente, en donde se explican paso a paso los conceptos y técnicas que se pueden emplear.

Hecho esto, se ha realizado la implementación de la parte encargada de la conexión y del intercambio de información entre el servidor y el cliente GUI, siguiendo el protocolo de comunicación utilizado por los demás clientes, aunque se han incluido unos comandos que permiten la sincronización entre el servidor y la interfaz gráfica.

Por último, se ha desarrollado la parte de la interfaz gráfica encargada de procesar el estado del juego recibido del servidor en el turno de cada cliente, y así generar, actualizar o eliminar gráficamente los objetos y actores que hay en el escenario actual, según el estado en el que se encuentren.

4.2. Arquitectura de AI-LIVE

El juego AI-LIVE sigue el modelo cliente-servidor, en donde la comunicación entre los clientes y el servidor se realiza mediante *sockets*[3], sobre TCP/IP.

La división de la arquitectura del juego en los módulos de servidor y clientes (mostrada en la Figura 23) otorga una estructuración y organización del contenido, así como una transparencia en el diseño del sistema, proporcionando además una separación de las responsabilidades de cada uno.

Esto hace que cada módulo pueda ser implementado mediante diferentes técnicas y lenguajes de programación, permitiendo la ampliación de cada uno con independencia del resto.

Pero, a pesar de esta división, hay que tener en cuenta que tanto el servidor y los clientes poseen una ontología común, que contiene las clases, con sus respectivos atributos, y las relaciones entre éstas, representando así el universo virtual del juego.

En las siguientes subsecciones, se van a describir a alto nivel los módulos que componen la arquitectura de la aplicación, el protocolo de comunicación para el envío de mensajes entre el servidor y los clientes, y las entradas y salidas de cada uno de los módulos.

[3] Socket: es una interfaz a través de la cual dos entidades pueden intercambiarse flujos de datos de manera fiable y ordenada. Para ello se requiere: un protocolo de comunicaciones, una dirección de red y el número de un puerto.

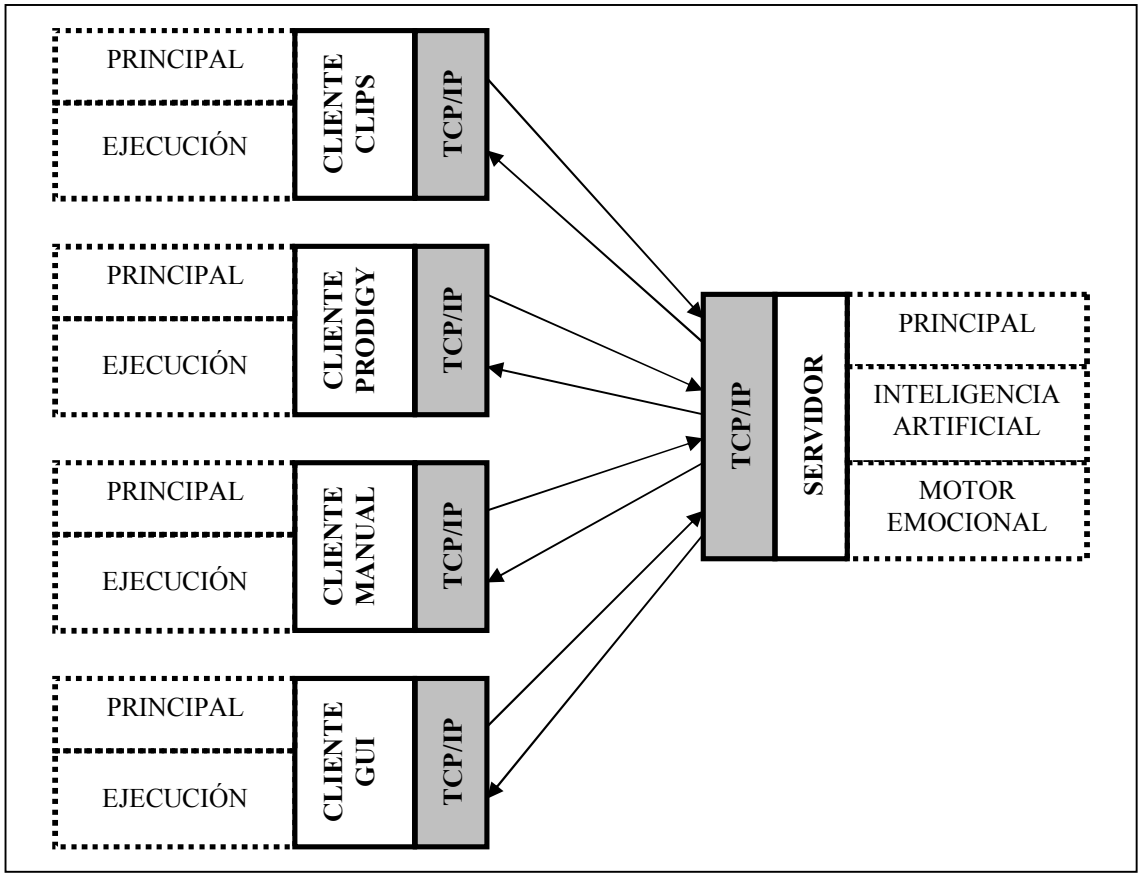

Figura 23: Arquitectura de la aplicación.

4.2.1. Servidor

El servidor está continuamente en escucha, a la espera de peticiones de conexión por parte de los clientes. Este proceso es independiente, por lo que no afecta a las demás funciones del servidor, como el proceso de validación de la conexión para los clientes, en el cual, cuando se valida a un cliente, pasa a formar parte de la lista de clientes para establecer la asignación de turnos, mediante el algoritmo de planificación *Round Robin*.

Además, el servidor contiene el estado completo del juego, recibiendo las peticiones realizadas por los clientes, procesándolas, y modificando el estado actual, para luego enviárselo a cada cliente.

Los submódulos que componen el servidor son:

- Submódulo principal: se encarga de establecer las conexiones con los clientes, llevar a cabo el protocolo de comunicaciones, asignar los turnos de éstos, y hacer llamadas al submódulo de inteligencia artificial.

- Submódulo de inteligencia artificial: contiene el estado completo del juego, y ejecuta las acciones recibidas por los clientes, actualizando el estado.

- Submódulo del motor emocional: controla las emociones, gustos y relaciones de los actores cuando tiene lugar una acción comunicativa.

4.2.2. Clientes

Cuando un cliente se conecta al servidor, se le asigna un *socket* a través del cual pueden intercambiarse información siguiendo el protocolo de comunicación establecido.

Los submódulos que forman los clientes son:

- Submódulo principal: se encarga de establecer la conexión con el servidor y de llevar a cabo el protocolo de comunicaciones.

- Submódulo de ejecución: realiza la funcionalidad específica de cada cliente, dependiendo de su tipo. Actualmente existen cuatro tipos de clientes:

    - Cliente CLIPS: es una entidad capaz de elegir la acción que quiere realizar e interactuar con otros clientes y objetos. Utiliza un motor de IA llamado CLIPS[4].

    - Cliente Prodigy: al igual que el cliente CLIPS, utiliza un motor de IA, pero éste integra un planificador de tareas llamado Prodigy, en lugar de emplear CLIPS.

    - Cliente manual: se basa en el cliente CLIPS, pero tiene la diferencia en que es una persona física la encargada de elegir la acción que quiere que realice el actor asociado a éste.

    - Cliente GUI: es el encargado de representar gráficamente el estado del juego. A diferencia de los clientes basados en IA y el manual, éste no selecciona ni envía la acción a realizar por el servidor.

Siguiendo una política LILO (*Last Input - Last Output* – el último en entrar es el último en salir), el cliente que tenga el turno, ya sea de IA o manual, recibirá el estado del servidor, teniendo un tiempo ilimitado para decidir la acción que desea ejecutar, para después enviársela al servidor. Mientras, los otros clientes permanecerán a la espera de su turno.

Sin embargo, los clientes GUI no tienen que esperar su turno, ya que deben actualizar, en los turnos de los demás clientes, la información gráfica recibida en el estado del juego que le envía el servidor.

[4] CLIPS (*C Language Integrated Production System*): herramienta libre de programación, creada por la NASA (*National Aeronautics Space Administration*), que permite realizar sistemas expertos basados en IA. Soporta programación lógica (para utilizar reglas de cumplimiento: Si A -> B), imperativa (para ejecutar algoritmos) y orientada a objetos (para diseñar sistemas complejos mediante módulos).

4.2.3. Protocolo de comunicación

Para el intercambio de información entre el servidor y los clientes, existe un protocolo de comunicación que utiliza una serie de etiquetas que identifican el contenido o la función de cada mensaje:

- **HOLA**: utilizada tanto por el servidor como los clientes para iniciar la conexión. En él se indica la versión y las diversas opciones soportadas por los elementos que intervienen en la comunicación.

- **STAG**: utilizada por los clientes para indicar el escenario en donde entrar.

- **ACTR**: utilizada por los clientes de IA y los manuales para especificar el identificador del actor que están controlando.

- **ACTN**: utilizada por los clientes de IA y los manuales para indicar la acción que desean que ejecute el servidor.

- **STAT**: utilizada por el servidor para enviar el estado del juego a un cliente.

- **GO**: utilizada por el cliente GUI implementado y el servidor para indicar que el módulo actual ha terminado la actualización de la información y que el otro módulo puede continuar su ejecución.

- **EX**: utilizada por el cliente GUI implementado para indicarle al servidor que ha finalizado su ejecución.

A continuación se describe el conjunto de pasos a seguir, tanto por el servidor como por los clientes, para establecer y mantener una conexión entre éstos a través del protocolo de comunicación.

4.2.3.1. Pasos a realizar por el servidor

1. Iniciar el motor de IA del servidor.
2. Iniciar el escuchador de red.
3. Conectarse con el cliente a través del protocolo establecido.
4. Bucle principal:

   - Seleccionar al cliente que posea el turno.
   - Enviar el estado al cliente actual y al GUI.
   - Recibir la acción del cliente.
   - Enviar la acción al servidor de IA.
   - Recibir el estado de IA.

4.2.3.2. Pasos a realizar por los clientes de IA y manual

1. Conectarse al servidor.
2. Bucle principal:

- Recibir el estado completo.
- Interpretar el estado recibido.
- Elegir la acción a realizar (de distinta forma según el tipo de cliente)
- Enviar la acción a realizar.

4.2.3.3. Pasos a realizar por el cliente GUI

1. Conectarse al servidor.
2. Bucle principal:

- Recibir el estado referente a los objetos gráficos.
- Representar el estado recibido.

4.2.3.4. Inicio de la conexión

El cliente se conecta al servidor de escucha. El servidor puede cerrar la conexión en cualquier momento si el cliente no es adecuado para él.

1. El servidor envía al cliente **HOLA*xxyz***. Este comando se utiliza para iniciar la conexión con el cliente:

- ***xx***: indica la versión del servidor.
- ***y***: opciones que soporta el servidor (todos los *flags* de tipo cliente soportados).
- ***z***: reservado (actualmente es 0).

2. El cliente envía al servidor **HOLA*xxyz***. Este comando se utiliza para iniciar la conexión con el servidor:

- ***xx***: indica la versión del cliente.
- ***y***: opciones que soporta el cliente (tipo de cliente).
- ***z***: reservado (actualmente es 0).

3. El cliente envía al servidor **STAG*xxxxstage*0**. Este comando se utiliza para enviar el escenario elegido por el cliente:

- ***xxxx*:** entero de 32 bits que indica la longitud en bytes del escenario enviado.
- ***stage*:** identificador del escenario.
- El comando termina con un byte 0, que no se tiene en cuenta.

4. Sólo los clientes de IA y manual envían al servidor **ACTR*xxxxactor*0**. Este comando se utiliza para el envío del identificador del actor controlado por el cliente:

- ***xxxx***: entero de 32 bits que indica la longitud en bytes del identificador del actor
- ***actor***: identificador del actor generado aleatoriamente.
- El comando termina con un byte 0, que no se tiene en cuenta.

5. Los clientes de IA y manual envían al servidor **ACTN*xxxxprofile*(id [*a_id*])(stage [*stg*])0**. Este comando se utiliza para el envío del perfil del actor:

- ***xxxx***: entero de 32 bits que indica la longitud en bytes del perfil del actor.
- ***profile***: perfil del actor que se envía al servidor.
- ***a_id***: identificador del actor.
- ***stg***: escenario en el que está el actor.
- El comando termina con un byte 0, que no se tiene en cuenta.

4.2.3.5. Bucle de la conexión

1. El servidor envía al cliente **STAT*xxxxstate*0**. Este comando se utiliza para enviar el estado del juego al cliente:

- ***xxxx***: entero de 32 bits que indica la longitud en bytes del estado enviado.
- ***state***: estado enviado del servidor al cliente.
- El comando termina con un byte 0, que no se tiene en cuenta.

2.a. Si se trata de un cliente de IA o manual, éste responde con **ACTN*xxxxaction*0**. Este comando se utiliza para el envío de la acción del cliente al servidor:

- ***xxxx*:** entero de 32 bits que indica la longitud en bytes de la acción enviada.
- ***action*:** acción enviada del cliente al servidor.
- El comando termina con un byte 0, que no se tiene en cuenta.

2.b. Si se trata del cliente GUI, éste responde con **GO** para indicarle al servidor que ha terminado de representar el estado y que continúe con su ejecución, o **EX** para indicarle que ha finalizado su ejecución y que no le envíe el estado en próximas iteraciones.

4.2.4. Entradas y salidas de los módulos de la aplicación

A continuación, se describen el proceso de inicialización de cada módulo de la arquitectura para realizar su ejecución, las entradas que recibe y las salidas que produce.

4.2.4.1. Servidor

Entradas

- Ontología (*ontology.clp*)
- Código del servidor CLIPS (*server.clp*)
- Código del motor emocional (*emotional_engine_server.clp*)
- Estado inicial del juego (*initial.state*)

En la Figura 24 se proporciona un ejemplo del contenido del archivo *initial.state*.

```
([DEFAULT_STAGE_0_0_0] of  Cell
(stage [DEFAULT_STAGE])
(inCell [DS_WALL_SOUTH_ID])
(occupied TRUE)
(x 0)
(z 0)
(y 0))

([DEFAULT_STAGE_0_0_1] of  Cell
(stage [DEFAULT_STAGE])
(inCell [DS_WALL_SOUTH_ID])
(occupied TRUE)
(x 0)
(z 0)
(y 1))

([DEFAULT_STAGE_0_0_2] of  Cell
(stage [DEFAULT_STAGE])
(inCell [DS_WALL_SOUTH_ID])
(occupied TRUE)
(x 0)
(z 0)
(y 2))
```

Figura 24: Ejemplo del contenido del archivo initial.state.

Ejecución normal

El servidor almacena en un archivo dinámico la información necesaria para cada cliente, cuyo nombre tiene el identificador del actor (*id_actor.state*), para luego enviársela mediante un mensaje, usando la etiqueta **STAT**. Esta información contiene el estado de los objetos del escenario actual, así como las características del actor que controla el cliente, incluyendo su psique, sus gustos y sus relaciones con otros actores.

En la Figura 25 se muestra un ejemplo del contenido del archivo *id_actor.state*.

```
([DEFAULT_STAGE] of Stage
(name_ "sala")
(entities [CLIPS_ACTOR_7DG312ZVKB] [Book_1_id] [Computer_1_id] [Sofa_1_id]
[Refrigerator_1_id] [Bed_1_id] [Bed_2_id] [Ball_1_id] [ExitDoor_1_id]
[Teleporter_1_id] [DS_WALL_NORTH_ID] [DS_WALL_EAST_ID] [DS_WALL_SOUTH_ID]
[DS_WALL_WEST_ID] [DS_FLOOR_ID])
(stageGraphic [nil])
(length 14)
(width 20)
(height 0)
)
([BATH_STAGE] of Stage
(name_ "baño")
(entities [Shower_1_id] [Teleporter_2_id] [BS_WALL_NORTH_ID]
[BS_WALL_EAST_ID] [BS_WALL_SOUTH_ID] [BS_WALL_WEST_ID] [BS_FLOOR_ID])
(stageGraphic [nil])
(length 4)
(width 4)
(height 0)
)
```

Figura 25: Ejemplo del contenido del archivo id_actor.state.

Para el cliente GUI, el servidor almacena la información del estado en un archivo dinámico, cuyo nombre tiene el identificador del escenario (*id_stage.state*) en el que está el actor actual, envía la etiqueta **GO** para que el cliente GUI continúe con su ejecución, y seguidamente le envía el estado, a través de un mensaje con la etiqueta **STAT**. Esta información contiene el estado de todos los objetos y actores en el escenario, y la acción que realiza el actor que controla el cliente actual.

En la Figura 26 se ofrece un ejemplo del contenido del archivo *id_stage.state*.

```
([DEFAULT_STAGE] of Stage
(name_ "sala")
(entities [Computer_1_id] [Sofa_1_id] [Refrigerator_1_id] [Bed_1_id]
[Bed_2_id] [Ball_1_id] [Book_1_id] [ExitDoor_1_id] [Teleporter_1_id]
[DS_WALL_NORTH_ID] [DS_WALL_EAST_ID] [DS_WALL_SOUTH_ID] [DS_WALL_WEST_ID]
[DS_FLOOR_ID])
(stageGraphic [nil])
(length 14)
(width 20)
(height 0)
)
([Book_1_id] of Book
(stage [DEFAULT_STAGE])
(name_ "Book_1")
(x 1)
(y 10)
(z 0)
(sizeX 1)
(sizeY 1)
(sizeZ 1)
(angle north)
(weight 1)
(entityGraphic "book_table")
(entityState "on")
(volume 0)
(type book)
(value_hunger 1)
(value_tiredness 1)
(in [DEFAULT_STAGE_0_1_10])
)
```

Figura 26: Ejemplo del contenido del archivo id_stage.state y gui.state.

El servidor recibe, por parte de los clientes, exceptuando el cliente GUI, un mensaje con la etiqueta **ACTN**, que contiene la acción que desea realizar cada uno.

Cuando la acción recibida es comunicativa, el submódulo de inteligencia artificial del servidor realiza una llamada al submódulo del motor emocional, que se encarga de actualizar las emociones, las relaciones y los gustos de los actores.

4.2.4.2. Clientes CLIPS y manual

Entradas

- Ontología (*ontology.clp*)
- Código del cliente CLIPS (*client.clp*)
- Perfil del actor (*DEFAULT_ACTOR.profile*)

En la Figura 27 aparece un ejemplo del contenido del archivo *DEFAULT_ACTOR.profile*.

```
([DEFAULT_ACTOR] of AddClient
(name_ "Amy")
(gender F)
(entityGraphic "casual woman")
(maxLoad 5)
(maxVolume 10)
(volume 1)
(weight 55)
(agreeableness 1.0)
(conscientousness 1.0)
(extraversion 1.0)
(neuroticism 0.0)
(openness 1.0)
(age 0)
(status 0)
(sex 0)
(arousal 0.0)
(valence 0.0)
(dislikenessList Read meat bathtub [Working])
(likenessList Shower Bed [Resting])
(neutrallikeList Ball)
(inter_likeList fruit)
(inter_dislikeList Sofa [Washed])
(maxhunger 30)
(maxthirst 30)
(maxdirtness 30)
(minwealth 30)
(maxtiredness 30)
(maxboredom 30)
(maxsolitude 30)
(continueProbability 80))
```

Figura 27: Ejemplo del contenido del archivo DEFAULT_ACTOR.profile.

Ejecución normal

El cliente selecciona la acción que desea ejecutar, ya sea a través de IA o manualmente (según se trate de un cliente CLIPS o manual), almacenándola en un archivo dinámico (*current.action*), el cual es enviado al servidor mediante un mensaje con la etiqueta **ACTN**.

En la Figura 28 se proporciona un ejemplo del contenido del archivo *current.action*.

```
([Washed] of Washed
(stage [BATH_STAGE])
(actor [CLIPS_ACTOR_7DG312ZVKB])
(bathObject [Shower_1_id])
)
```

Figura 28: Ejemplo del contenido del archivo current.action.

Éste recibe del servidor la información relevante del estado actual del juego, a través de un mensaje con la etiqueta **STAT**.

Una vez ha finalizado su ejecución, el cliente genera un archivo de traza (*id_actor-TRAZA.txt*) con la secuencia de acciones realizadas por éste.

En la Figura 29 se muestra un ejemplo del contenido del archivo *id_actor-TRAZA.txt*.

```
********** regla WorkObjectGoal **********

     - vamos a por: [Computer_1_id]
     - situado en x: 12 y: 10
     - coordenadas actuales del actor: X: 2 Y:15
     - celda de acceso : [DEFAULT_STAGE_0_11_9]

[DEFAULT_STAGE_0_11_9] of Cell
(stage [DEFAULT_STAGE])
(occupied FALSE)
(inCell [nil])
(x 11)
(y 9)
(z 0)


********** regla move **********

     - Se va a mover el actor con el Id: [CLIPS_ACTOR_7DG312ZVKB]
     - Coordenadas futuras del actor: X= 11 Y= 9

********** regla Working **********

     - objeto de trabajo : [Computer_1_id]
```

Figura 29: Ejemplo del contenido del archivo id_actor-TRAZA.txt

4.2.4.3. Cliente Prodigy

Entradas

- Dominio del problema (*domain.lisp*)
- Problema a resolver (*problem.lisp*)
- Perfil del actor (*DEFAULT_ACTOR.profile*)

Ejecución normal

El cliente almacena la acción obtenida en un archivo dinámico (*current.action*), el cual será enviado al servidor mediante un mensaje con la etiqueta **ACTN**.

Seguidamente traduce el estado del juego recibido del servidor de CLIPS a LISP para poder manejar el problema.

A continuación genera el objetivo del problema, que es obtener el mayor número de objetos que hay en el escenario.

Al igual que los clientes CLIPS y manual, éste recibe del servidor la información relevante del estado actual del juego, a través de un mensaje con la etiqueta **STAT**.

4.2.4.4. Cliente GUI

Entradas

- Núcleo de OGRE
- Plugins (CEGUI y OIS)
- Recursos de imagen necesarios (objetos, texturas,...)

Ejecución normal

El cliente recibe del servidor la información relevante del estado actual del juego a través de un mensaje con la etiqueta **STAT**, y lo almacena en un archivo dinámico (*gui.state*), del cual se muestra un ejemplo de su contenido en la Figura 26.

Una vez hecho esto, lee el archivo y genera, actualiza o elimina gráficamente las entidades del escenario actual, según el contenido del estado.

El cliente le enviará al servidor la etiqueta **GO** para indicarle que la interfaz gráfica ha terminado de actualizar la información gráfica, y que éste puede continuar con su ejecución.

Una vez que ha terminado su ejecución, el cliente le enviará la etiqueta **EX** al servidor para indicarle que debe borrarle de la lista de clientes, y así no enviarle un nuevo estado en los próximos turnos.

4.3. Modelo de conocimiento de AI-LIVE

En las Figuras de la 30 a la 34 se muestra la ontología, que representa el modelo de conocimiento del juego AI-LIVE, compuesta por un conjunto de clases, con sus respectivos atributos, y las relaciones que hay entre éstas. Esta ontología es común tanto para clientes como para el servidor.

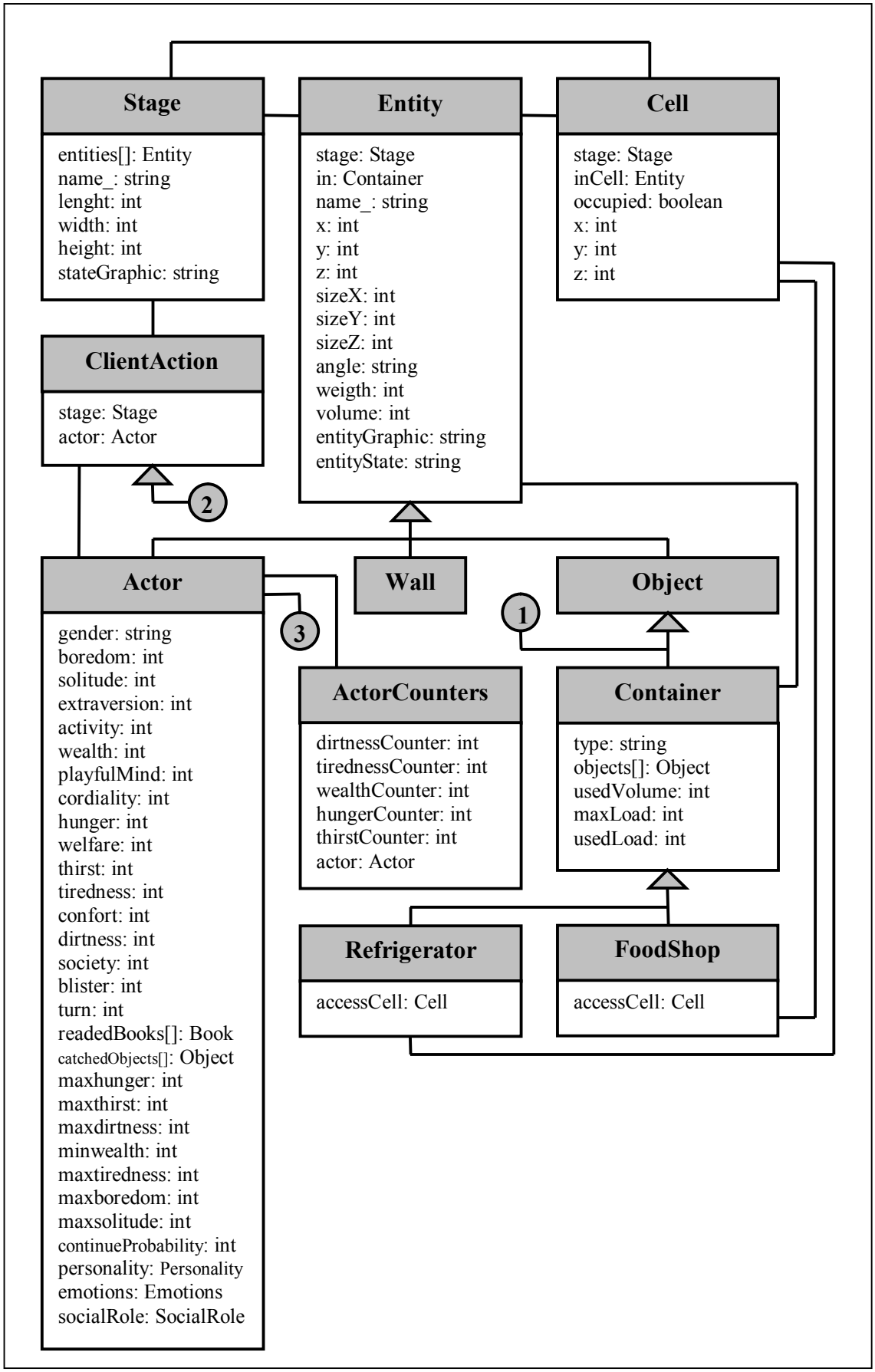

Figura 30: Modelo de conocimiento - General.

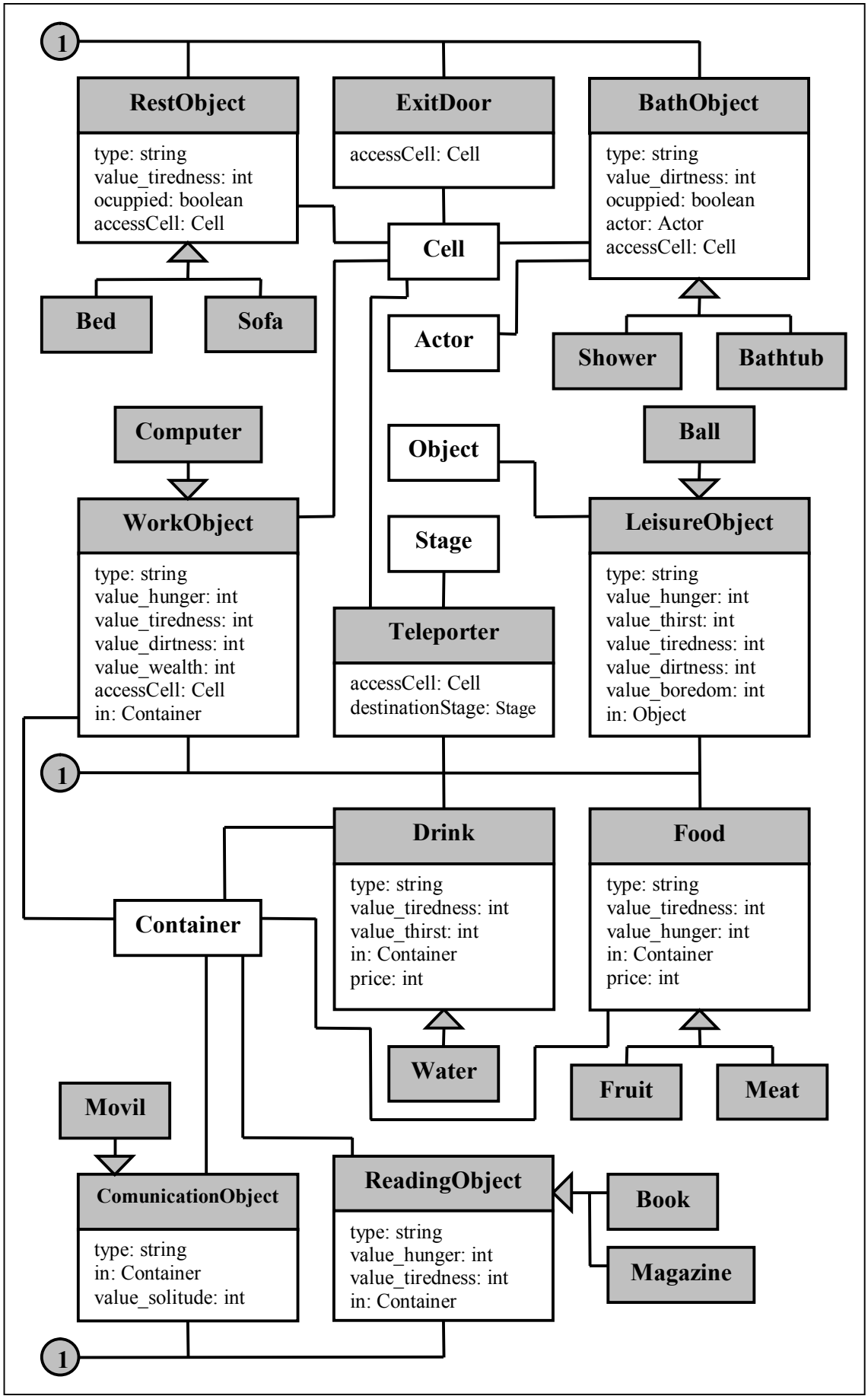

Figura 31: Modelo de conocimiento - Resto de subclases de Object.

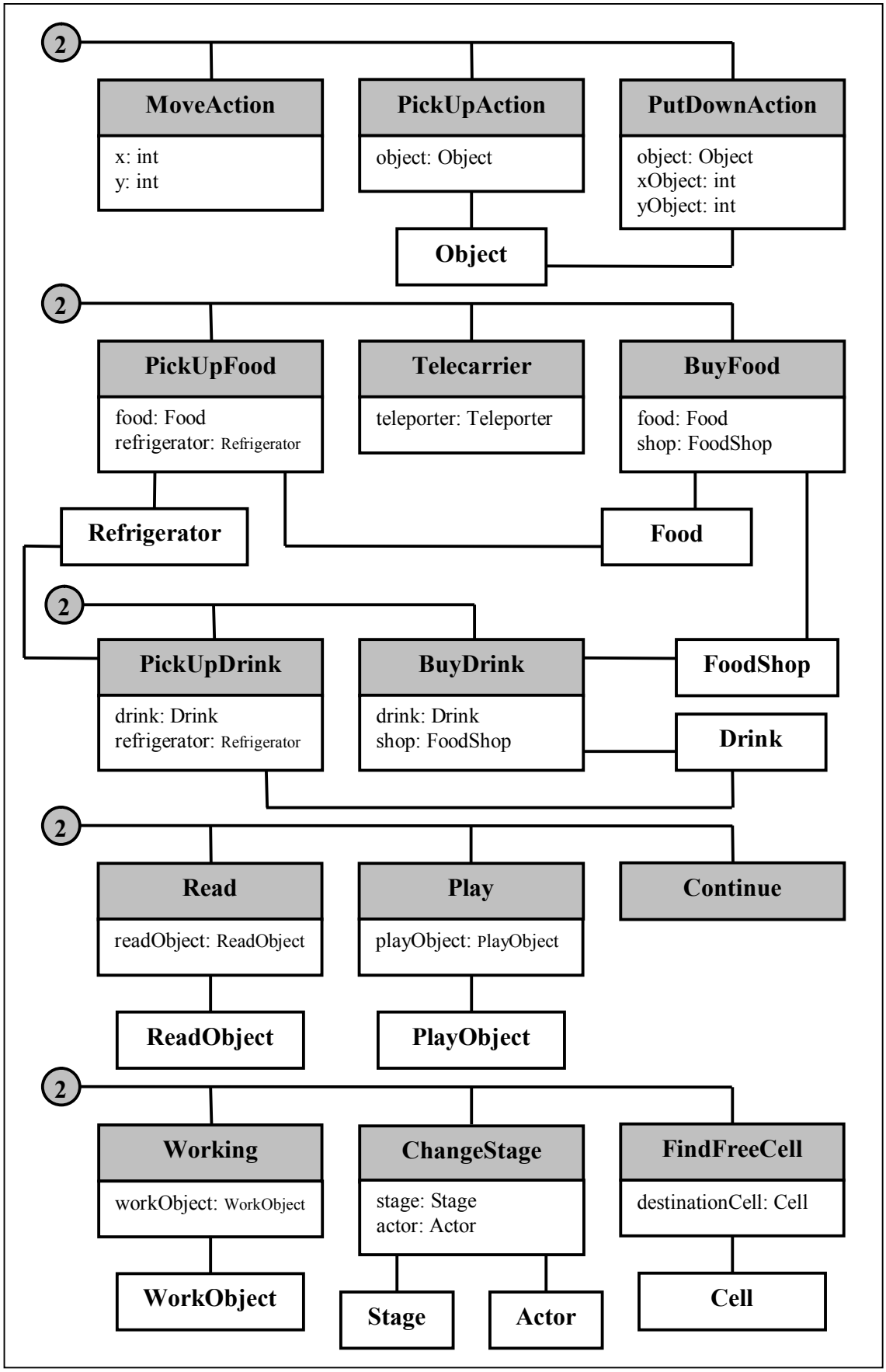

Figura 32: Modelo de conocimiento - Subclases de ClientAction I.

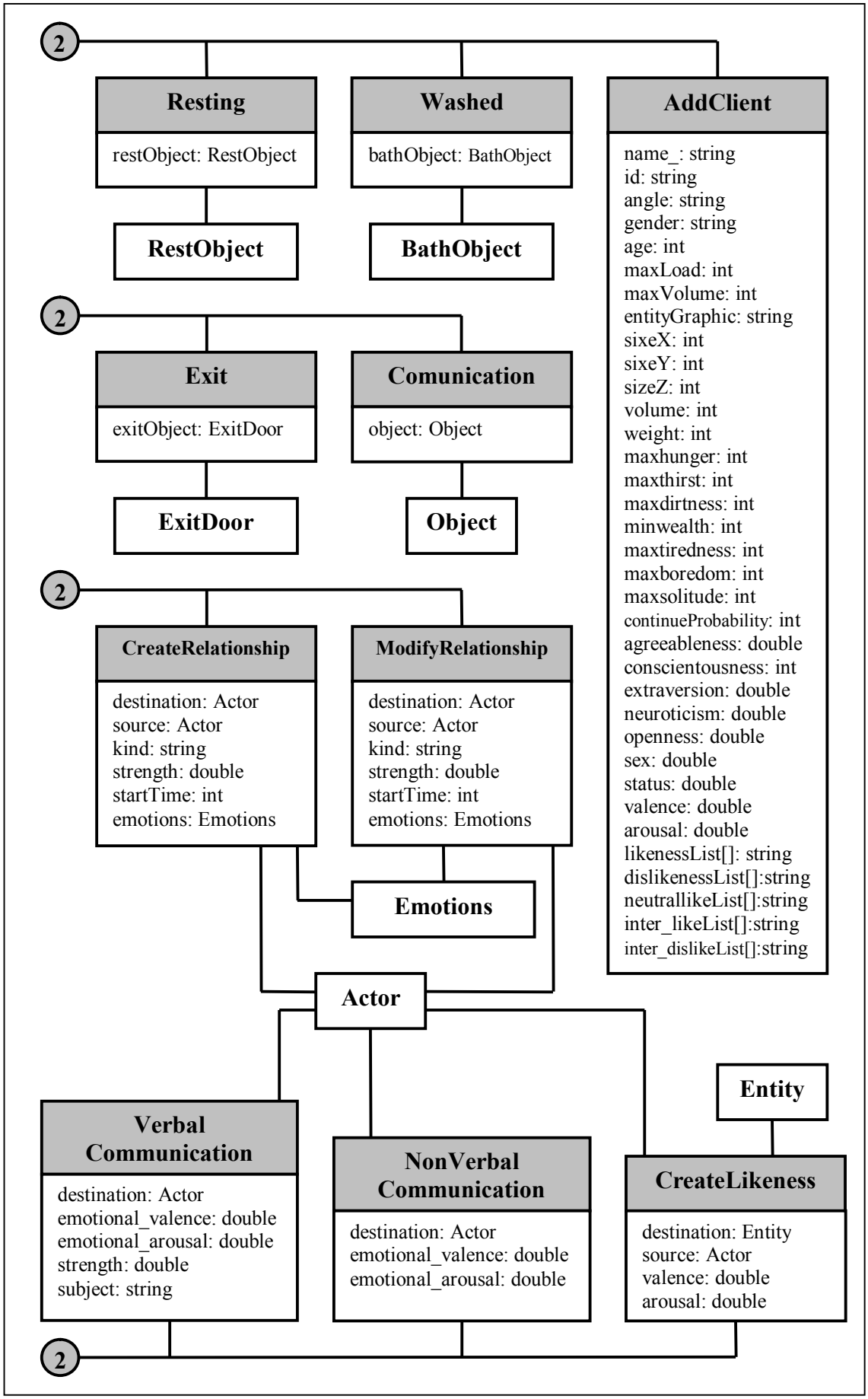

Figura 33: Modelo de conocimiento - Subclases de ClientAction II.

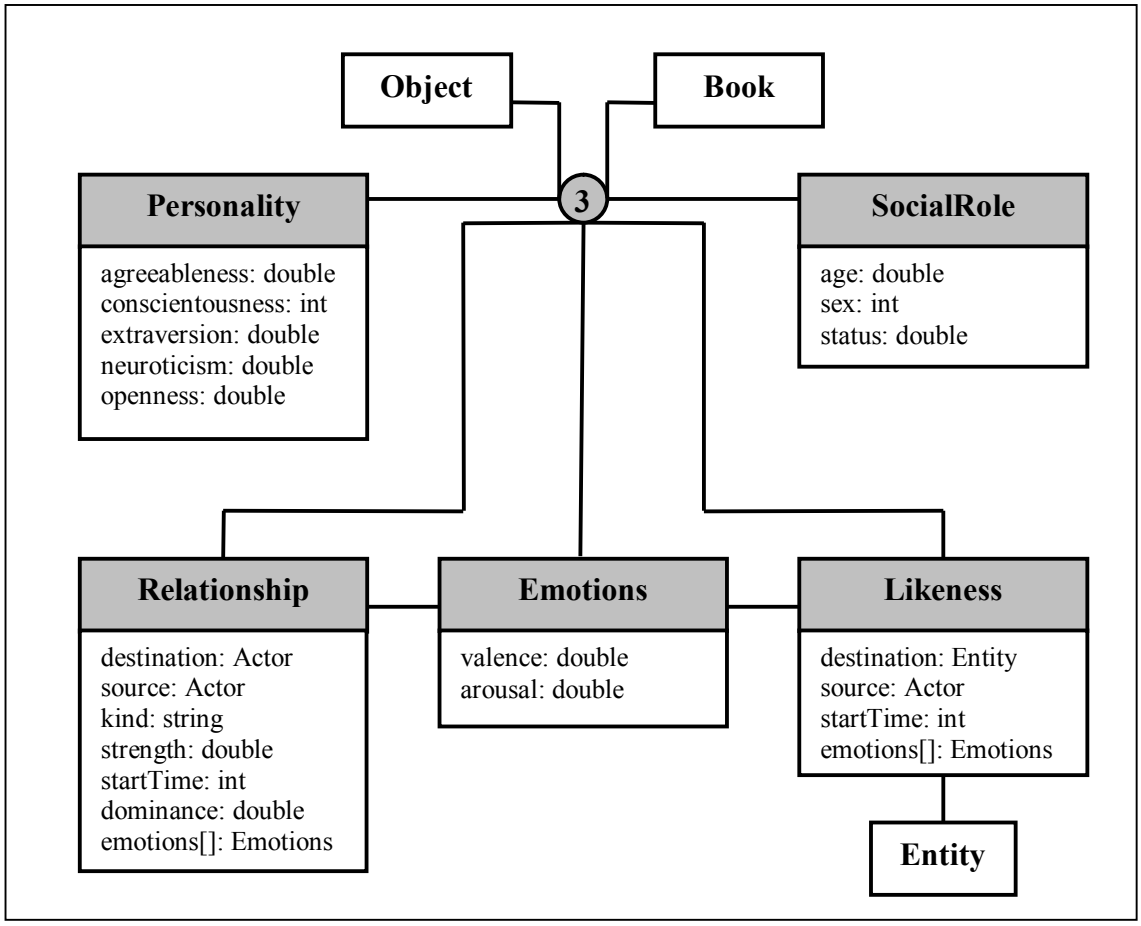

Figura 34: Modelo de conocimiento - Psique y relaciones sociales del actor.

Seguidamente se van a comentar las clases de la ontología, con sus respectivos atributos, que se van a utilizar en la creación de entidades en la interfaz gráfica:

Stage

Representa los escenarios que pueden existir en el estado del juego. Un escenario contiene una colección de objetos de la clase **Entity**. Todo el espacio que comprende un escenario esta representado con objetos de la clase **Cell**. Los atributos que componen esta clase son:

- entities: lista que representa el conjunto de entidades pertenecientes a un escenario.
- name_: nombre del escenario.
- length: longitud del escenario expresada en número de celdas.
- width: anchura del escenario expresada en número de celdas.
- heigth: altura del escenario expresada en número de celdas.
- stageGraphic: textura para representar el escenario en la interfaz gráfica.

Cell

Representa una porción mínima e indivisible de espacio dentro de un escenario. Se utiliza para la detección de colisiones y como posición de acceso a otros objetos, lo que hace que no se haya implementado gráficamente. Los atributos que componen esta clase son:

- stage: escenario al que pertenece la celda.
- inCell: en caso de que la celda no esté ocupada el valor del atributo será *nil*, en otro caso, contendrá el identificador del objeto situado en ésta.
- occupied: indica si la celda esta ocupada (TRUE) o libre (FALSE).
- x: coordenada X de la celda dentro del escenario.
- y: coordenada Y de la celda dentro del escenario.
- z: coordenada Z de la celda dentro del escenario.

Actor

Especialización de la clase **Entity** que define las características de un actor. Se explican a continuación los atributos propios más relevantes:

- gender: género del actor. Puede ser masculino (M) o femenino (F).
- boredom: nivel de aburrimiento del actor.
- solitude: nivel de soledad del actor.
- extraversion: nivel de extroversión del actor.
- activity: nivel de actividad del actor.
- wealth: riqueza del actor.
- playfullMind: nivel de creatividad del actor.
- cordiality: nivel de cordialidad del actor.
- hunger: nivel de hambre del actor.
- welfare: nivel de bienestar del actor. Se calcula a partir de los atributos hunger, thirst, solitude, dirtness, tiredness y boredom).
- thirst: nivel de sed del actor.
- tiredness: nivel de cansancio del actor.
- confort: nivel de comodidad del actor.
- dirtness: nivel de suciedad del actor.
- society: nivel de sociabilidad del actor.
- blister: nivel de vejiga del actor.
- turn: turno del actor. Se utiliza para controlar las acciones que conlleven más de un turno.
- readedBooks: lista de los libros leídos por el actor.
- catchedObjects: lista de los objetos cogidos por el actor.
- continueProbability: probabilidad para que el actor ejecute la regla Continuar.
- personality: personalidad del actor.
- emotions: estado emocional del actor.
- socialRole: estado social del actor.

ActorCounters

Representa los valores de suciedad, cansancio, riqueza, hambre y sed de un actor determinado. Los atributos que componen esta clase son:

- dirtnessCounter: valor de suciedad del actor.
- tirednessCounter: valor de cansancio del actor.
- wealthCounter: valor de riqueza del actor.
- hungerCounter: valor de hambre del actor.
- thristCounter: valor de sed del actor.
- actor: actor al que pertenecen los valores.

Entity

Es una generalización abstracta de todos los tipos de entidades que pueden existir en el juego. Contiene los atributos comunes a todas ellas. Los atributos que componen esta clase son:

- stage: escenario al que pertenece la entidad.
- in: objeto contenedor dentro del cual está la entidad, o *nil* en caso de no pertenecer a ninguno.
- name_: nombre de la entidad.
- x: coordenada X de la entidad dentro del escenario en el que se encuentra.
- y: coordenada Y de la entidad dentro del escenario en el que se encuentra.
- z: coordenada Z de la entidad dentro del escenario en el que se encuentra.
- sizeX: tamaño respecto al eje X.
- sizeY: tamaño respecto al eje Y.
- sizeZ: tamaño respecto al eje Z.
- angle: orientación de la entidad dentro del escenario. Los posibles valores que pueden tomar son: north, south, east, west.
- weight: peso de la entidad.
- volume: volumen de la entidad.
- entityGraphic: nombre del *mesh* asociado a la entidad para poder representarla en la interfaz gráfica.
- entityState: estado de la entidad utilizado para representarla en la interfaz gráfica.

Object

Especialización de la clase **Entity** que especifica las propiedades de un objeto genérico. No contiene atributos propios.

Container

Especialización de la clase **Object** que sirve para representar a aquellos objetos que tienen la propiedad de contener a otros objetos. Los atributos propios que componen esta clase son:

- type: especifica el tipo de objeto contenedor, puede ser de dos tipos: refrigerator y shop.
- objects: lista de objetos contenida por el contenedor.
- usedVolume: representa el volumen usado dentro del contenedor.
- maxLoad: representa el número máximo de objetos que puede tener el contenedor.
- usedLoad: representa el número de objetos que tiene en el contenedor.

Refrigerator

Especialización de la clase **Container** que representa un frigorífico que contiene objetos de comida. El único atributo propio que compone esta clase es:

- accessCell: celda de acceso al frigorífico.

FoodShop

Especialización de la clase **Container** que representa una tienda de comida que contiene objetos de comida. El único atributo propio que compone esta clase es:

- accessCell: celda de acceso a la tienda de comida.

RestObject

Especialización de la clase **Object** que representa un objeto de descanso. Los atributos propios que componen esta clase son:

- type: tipo de objeto de descanso.
- value_tiredness: valor en que disminuye el cansancio del actor debido a la utilización del objeto.
- ocuppied: indica si el objeto está ocupado (yes) o no (no).
- accessCell: celda de acceso al objeto.

Bed

Especialización de la clase **RestObject** que representa una cama. No contiene atributos propios.

Sofa

Especialización de la clase **RestObject** que representa un sofá. No contiene atributos propios.

Teleporter

Especialización de la clase **Object** que representa un objeto de teletransporte entre escenarios. Los atributos propios que componen esta clase son:

- accessCell: celda de acceso al teletransportador.
- destinationStage: escenario destino.

ExitDoor

Especialización de la clase **Object** que representa una puerta de salida utilizada cuando un actor termina su ejecución. El único atributo propio que tiene esta clase es:

- accessCell: celda de acceso a la puerta de salida.

ReadingObject

Especialización de la clase **Object** que representa un objeto de lectura, portado siempre en una bolsa asignada a un actor. Los atributos propios que componen esta clase son:

- type: tipo de objeto de lectura.
- value_hunger: valor en que aumenta el hambre del actor debido al uso la utilización del objeto.
- value_tiredness: valor en que aumenta el cansancio del actor debido a la utilización del objeto.
- in: contenedor en el que se encuentra el objeto. En caso de no encontrarse en ningún contenedor, el valor del atributo será *nil*.

Book

Especialización de la clase **ReadingObject** que representa un libro. No contiene atributos propios.

Magazine

Especialización de la clase **ReadingObject** que representa una revista. No contiene atributos propios.

ComunicacionObject

Especialización de la clase **Object** que representa un objeto de comunicación. Los atributos propios que componen esta clase son:

- type: tipo de objeto de comunicación.
- value_solitude: valor en que aumenta la soledad del actor debido a la utilización del objeto.
- in: contenedor en el que se encuentra el objeto. En caso de no encontrarse en ningún contenedor, el valor del atributo será *nil*.

Movil

Especialización de la clase **ComunicationObject** que representa un teléfono móvil. No contiene atributos propios.

WorkObject

Especialización de la clase **Object** que representa un objeto destinado a la realización de una acción de trabajo. Los atributos propios que componen esta clase son:

- type: tipo de objeto de trabajo.
- value_hunger: valor en que aumenta el hambre del actor debido a la utilización del objeto.
- value_tiredness: valor en que aumenta el cansancio del actor debido a la utilización del objeto.
- value_dirtness: valor en que aumenta la suciedad del actor debido a la utilización del objeto.
- value_wealth: valor en que aumenta la riqueza del actor debido a la utilización del objeto.
- accessCell: celda de acceso al objeto.
- in: contenedor en el que se encuentra el objeto. En caso de no encontrarse en ningún contenedor, el valor del atributo será *nil*.

Computer

Especialización de la clase **WorkObject** que representa un ordenador personal. No contiene atributos propios.

BathObject

Especialización de la clase **Object** que representa un objeto de aseo personal. Los atributos propios que componen esta clase son:

- type: tipo de objeto de aseo.
- value_dirtness: valor en que disminuye la suciedad del actor debido a la utilización del objeto.
- ocuppied: indica si el objeto está ocupado (yes) o no (no).
- actor: actor que está ocupando el objeto. En caso de no encontrarse ningún actor dentro, el valor del atributo será *nil*.
- accessCell: celda de acceso al objeto.

Bathtub

Especialización de la clase **BathObject** que representa una bañera. No contiene atributos propios.

Shower

Especialización de la clase **BathObject** que representa una ducha. No contiene atributos propios.

LeisureObject

Especialización de la clase **Object** que representa un objeto de entretenimiento. Los atributos propios que componen esta clase son:

- type: tipo de objeto de entretenimiento.
- value_hunger: valor en que aumenta el hambre del actor debido a la utilización del objeto.
- value_thirst: valor en que aumenta la sed del actor debido a la utilización del objeto.
- value_tiredness: valor en que aumenta el cansancio del actor debido a la utilización del objeto.
- value_dirtness: valor en que aumenta la suciedad del actor debido a la utilización del objeto.
- value_boredom: valor en que aumenta el aburrimiento del actor debido a la utilización del objeto.
- in: contenedor, actor o suelo en el que se encuentra el objeto. En caso de no encontrarse en ninguno, el valor del atributo será *nil*.

Ball

Especialización de la clase **PlayObject** que representa una pelota. No contiene atributos propios.

Food

Especialización de la clase **Object** que representa un objeto de comida. Los atributos propios que componen esta clase son:

- type: tipo de objeto de comida.
- value_hunger: valor en que disminuye el hambre del actor debido a la utilización del objeto.
- value_tiredness: valor en que disminuye el cansancio del actor debido a la utilización del objeto.
- container: contenedor en el que se encuentra el objeto. En caso de no encontrarse en ningún contenedor, el valor del atributo será *nil*.
- price: precio del objeto de comida, utilizado para su compra en una tienda de comida.

Fruit

Especialización de la clase **Food** que representa una fruta. No contiene atributos propios.

Meat

Especialización de la clase **Food** que representa una comida. No contiene atributos propios.

Drink

Especialización de la clase **Object** que representa un objeto de bebida. Los atributos propios que componen esta clase son:

- type: tipo de objeto de comida.
- value_thirst: valor en que disminuye la sed del actor debido a la utilización del objeto.
- value_tiredness: valor en que disminuye el cansancio del actor debido a la utilización del objeto.
- container: contenedor en el que se encuentra el objeto. En caso de no encontrarse en ningún contenedor, el valor del atributo será *nil*.
- price: precio del objeto de bebida, utilizado para su compra en una tienda de comida.

Water

Especialización de la clase **Drink** que representa una botella de agua. No contiene atributos propios.

Wall

Especialización de la clase **Entity** que representa una entidad de tipo pared. No puede ser atravesada por un actor. No contiene atributos propios.

ClientAction

Representa de forma genérica una acción. Toda nueva acción que se implemente debe heredar de esta clase. Los atributos que componen esta clase son:

- stage: escenario en el que va a tener lugar la acción.
- actor: actor encargado de realizar la acción.

MoveAction

Especialización de la clase **ClientAction** que representa el movimiento de un actor de una celda a otra. Los atributos propios que componen esta clase son:

- x: coordenada X de destino del actor.
- y: coordenada Y de destino del actor.

PickUpAction

Especialización de la clase **ClientAction** que representa la acción de recoger un objeto del escenario. El único atributo propio que compone esta clase es:

- object: objeto a recoger del escenario.

PutDownAction

Especialización de la clase **ClientAction** que representa la acción de dejar un objeto en el escenario. Los atributos propios que componen esta clase son:

- object: objeto a dejar en el escenario.
- xObject: coordenada X de la celda donde dejar el objeto.
- yObject: coordenada Y de la celda donde dejar el objeto.

PickUpFood

Especialización de la clase **ClientAction** que representa la acción de recoger un objeto de comida de un frigorífico. Los atributos propios que componen esta clase son:

- food: objeto de comida que va a ser extraído del frigorífico.
- refrigerator: frigorífico donde se encuentra almacenado el objeto a extraer.

BuyFood

Especialización de la clase **ClientAction** que representa la acción de comprar un objeto de comida en una tienda. Los atributos propios que componen esta clase son:

- food: objeto de comida que va a ser comprado.
- shop: tienda donde se encuentra almacenado el objeto a comprar.

PickUpDrink

Especialización de la clase **ClientAction** que representa la acción de recoger un objeto de bebida de un frigorífico. Los atributos propios que componen esta clase son:

- drink: objeto de bebida que va a ser extraído del frigorífico.
- refrigerator: frigorífico donde se encuentra almacenado el objeto a extraer.

BuyDrink

Especialización de la clase **ClientAction** que representa la acción de comprar un objeto de bebida en una tienda. Los atributos propios que componen esta clase son:

- drink: objeto de bebida que va a ser comprado.
- shop: tienda donde se encuentra almacenado el objeto a comprar.

Telecarrier

Especialización de la clase **ClientAction** que representa la acción de usar un teletransportador. El único atributo propio que compone esta clase es:

- teleporter: objeto de teletransporte que va a ser usado.

Working

Especialización de la clase **ClientAction** que representa la acción de trabajar con un objeto. El único atributo propio que compone esta clase es:

- workObject: objeto de trabajo a utilizar.

Play

Especialización de la clase **ClientAction** que representa una acción de juego del actor. El único atributo propio que compone esta clase es:

- playObject: objeto de juego a utilizar.

Read

Especialización de la clase **ClientAction** que representa una acción de lectura del actor. El único atributo propio que compone esta clase es:

- readingObject: objeto de lectura a utilizar.

Resting

Especialización de la clase **ClientAction** que representa una acción de descanso del actor. El único atributo propio que compone esta clase es:

- restObject: objeto de descanso a utilizar.

Washed

Especialización de la clase **ClientAction** que representa una acción de higiene personal. El único atributo propio que compone esta clase es:

- bathObject: objeto de higiene personal a utilizar.

ChangeState

Especialización de la clase **ClientAction** que representa la acción de teletransportarse a otro escenario. Los atributos propios que componen esta clase son:

- stage: escenario al cual se va a cambiar.
- actor: actor que va a cambiar de escenario.

FindFreeCell

Especialización de la clase **ClientAction** que representa la acción de desplazamiento a una celda libre. El único atributo propio que compone esta clase es:

- destinationCell: celda de destino libre a la que se desplazará el actor.

Continue

Especialización de la clase **ClientAction** que indica que un actor continúa realizando la acción anterior. No contiene atributos propios.

Comunication

Especialización de la clase **ClientAction** que representa la acción de utilizar un objeto de comunicación. El único atributo propio que compone esta clase es:

- object: objeto de comunicación.

Exit

Especialización de la clase **ClientAction** que representa la acción de salida de un actor en el juego. El único atributo propio que compone esta clase es:

- exitObject: objeto de salida del juego.

AddClient

Especialización de la clase **ClientAction** que representa la acción de inicialización de un cliente, añadiéndole al estado del juego gestionado por el servidor. Los atributos propios más relevantes que componen esta clase son:

- name_: nombre del actor.
- id: identificador del cliente, único para cada actor.
- angle: orientación inicial del actor.
- gender: género del actor.
- age: edad del actor.
- maxLoad: número máximo de objetos que puede tener el actor.
- maxVolume: volumen máximo de objetos que puede llevar el actor.
- entityGraphic: nombre del *mesh* asociado a la entidad para poder representarla en la interfaz gráfica.
- sixeX: tamaño respecto al eje X del actor.
- sixeY: tamaño respecto al eje Y del actor.
- sizeZ: tamaño respecto al eje Z del actor.
- volume: volumen del actor.
- weight: peso del actor.
- sex: relaciones sexuales con otros actores.
- status: estatus social del actor.
- valence: valencia emocional del actor.
- arousal: activación emocional del actor.
- likenessList: lista de cosas que le gustan al actor.
- dislikenessList: lista de cosas que le disgustan al actor.
- neutrallikeList: lista de cosas que ni le gustan ni le disgustan al actor.
- inter_likeList: lista de cosas que le medio gustan al actor.
- inter_dislikeList: lista de cosas que le medio disgustan al actor.

VerbalCommunication

Especialización de la clase **ClientAction** que representa una acción de comunicación verbal. Los atributos propios que componen esta clase son:

- destination: actor hacia el que se dirige la acción comunicativa (receptor).
- subject: tema de la comunicación. Puede ser un objeto, acción, idea o actor.
- emotion_valence: valencia de la emoción transmitida.
- emotion_arousal: activación de la emoción transmitida.
- strength: fuerza de la emoción transmitida.

NonVerbalCommunication

Especialización de la clase **ClientAction** que representa una acción de comunicación no verbal. Los atributos propios que componen esta clase son:

- destination: actor hacia el que se dirige la acción comunicativa (receptor).
- emotion_valence: valencia de la emoción transmitida.
- emotion_arousal: activación de la emoción transmitida.

CreateLikeness

Especialización de la clase **ClientAction** que representa la acción de crear explícitamente un gusto de un actor. Los atributos propios que componen esta clase son:

- destination: objeto, idea o acción que le gusta al actor.
- source: actor que posee el gusto.
- valence: valencia emocional del gusto.
- arousal: activación emocional del gusto.

CreateRelationship y ModifyRelationship

Especialización de la clase **ClientAction** que representa la creación o modificación explícita (según la clase) de una relación entre dos actores. Los atributos propios que componen esta clase son:

- destination: actor destino al que va dirigida la relación.
- source: actor origen de la relación.
- kind: tipo de relación que tiene lugar entre los dos actores. Puede ser basada en edad (AgeBased), estatus social (StatusBased) o parentesco (BloodBased).
- strength: fuerza del lazo afectivo entre los actores.
- startTime: momento en que se inicia la relación.
- emotions: emociones que suscita el actor destino en el actor origen.

Personality

Recoge los cinco factores que componen la personalidad de un actor. Los atributos que componen esta clase son:

- agreeableness: valor de afabilidad del actor.
- conscientousness: valor de meticulosidad o autodisciplina del actor.
- extraversion: valor de extroversión del actor.
- neuroticism: valor de volubilidad del actor.
- openness: valor de mentalidad abierta del actor.

SocialRole

Representa el estado social de un actor. Los atributos que tiene esta clase son:

- age: edad del actor.
- sex: cantidad de actividad sexual del actor.
- status: estatus social, nivel que tiene dentro de la jerarquía social.

Emotions

Representa las emociones vinculadas a una persona, objeto o idea. Los atributos que componen esta clase son:

- valence: valencia de la emoción, nivel de satisfacción o agrado.
- arousal: activación de la emoción, nivel de actividad.

Likeness

Representa el gusto de un actor por un objeto, acción, idea u otro actor del juego. Los atributos que componen esta clase son:

- source: actor origen al que pertenece el gusto.
- destination: objeto, acción, idea o actor que le gusta al actor origen.
- startTime: momento en que se crea el gusto.
- emotions: emociones que despierta el objeto del gusto en el actor.

Relationship

Representa una relación social unidireccional de un actor origen con otro destino. Los atributos que componen esta clase son:

- source: actor origen del cual parte la relación.
- destination: actor destino hacia el que se dirige la relación.
- kind: tipo de relación que tiene lugar entre los dos actores. Puede ser basada en edad (AgeBased), estatus social (StatusBased) o parentesco (BloodBased).
- strength: fuerza del lazo afectivo entre los actores.
- dominance: grado de predominancia del actor origen sobre el destino.
- startTime: momento en que se inicia la relación.
- emotions: emociones que suscita el actor destino en el actor origen.

4.4. Cliente GUI

El cliente GUI está programado en lenguaje C++, utilizando el motor gráfico OGRE3D, el cual le proporciona un conjunto de métodos y funciones implementados previamente, que facilitan su desarrollo.

A diferencia del resto de clientes, el cliente GUI recibe el estado del juego en el turno de cada uno de éstos para representar la información gráfica, ya sea creando, actualizando o eliminando entidades gráficas. Además, éste no envía ninguna petición de realización de acción para que la ejecute el servidor.

A pesar de que el cliente GUI no usa la ontología al igual que lo hacen los demás clientes al no tener integrado CLIPS, la utiliza a la hora de representar las entidades según la clase a la que pertenezcan.

A continuación se expone el diagrama de clases del cliente GUI implementado, el contenido de los submódulos que lo forman, las ampliaciones realizadas en el módulo del servidor, los archivos de configuración necesarios para su ejecución, y la representación del estado del juego por la interfaz gráfica.

4.4.1. Diagrama de clases

En las Figuras 35 y 36 se muestra el diagrama de clases del cliente GUI implementado, que contiene el conjunto de clases necesarias para la elaboración de la interfaz gráfica, así como sus atributos y relaciones con otras clases.

Para cada clase sólo se indican los métodos y funciones que se utilizan en la implementación de la interfaz gráfica.

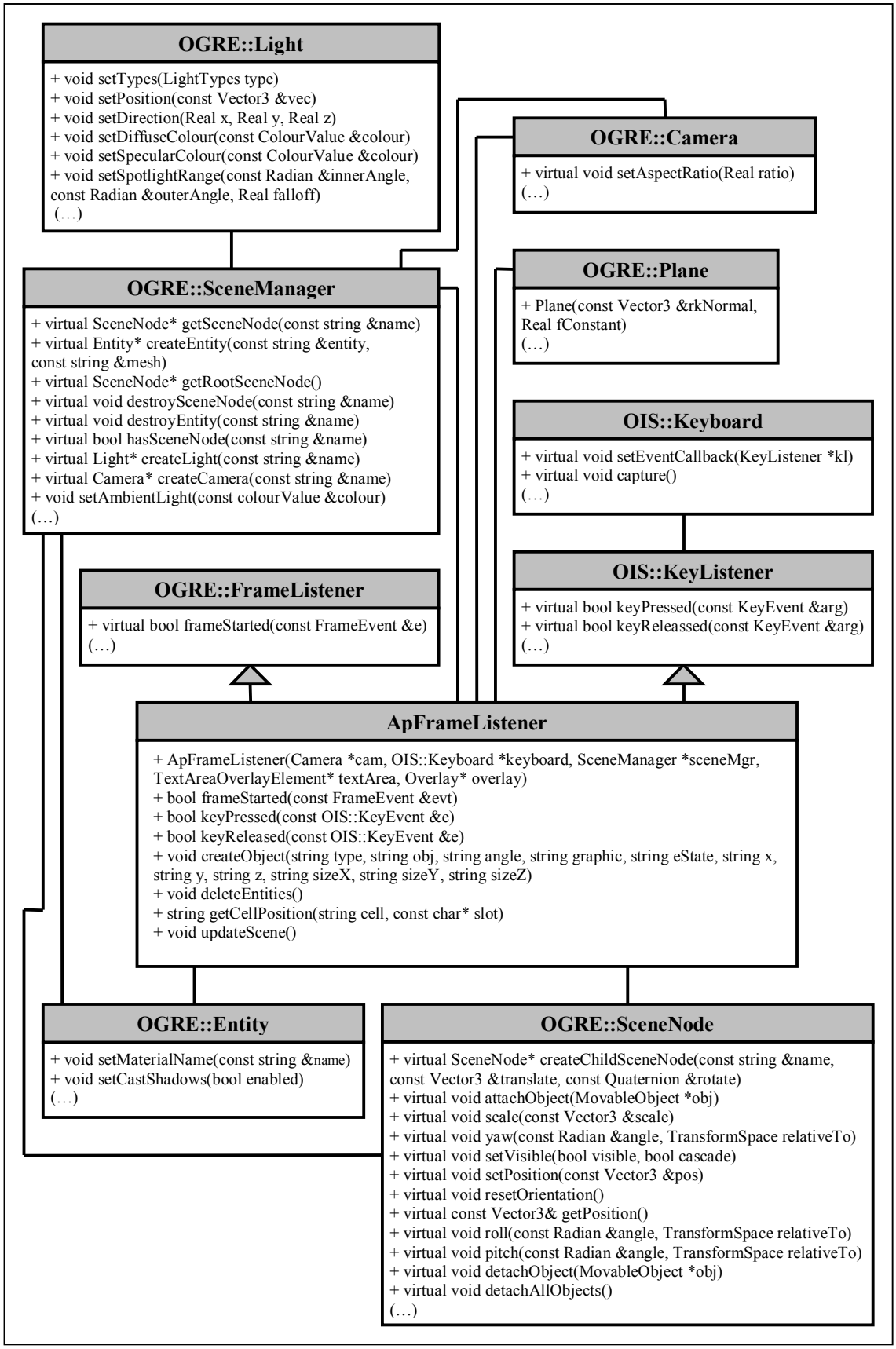

Figura 35: Diagrama de clases del cliente GUI I.

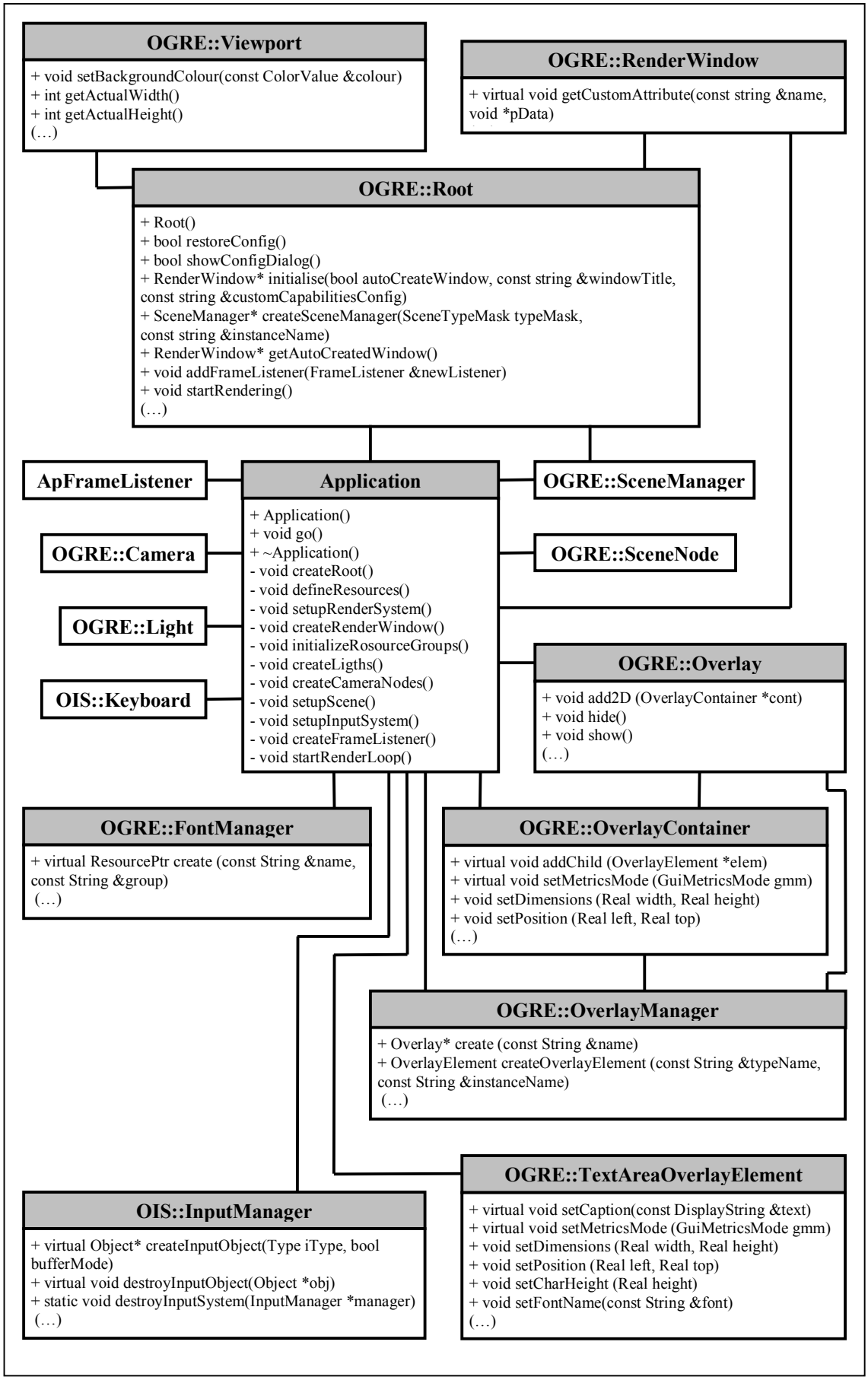

Figura 36: Diagrama de clases del cliente GUI II.

4.4.2. Submódulo principal y de ejecución: gui.cpp

El submódulo principal y de ejecución del cliente GUI (*gui.cpp*) está formado por dos clases: **ApFrameListener**, que contiene los métodos y funciones encargados de representar las entidades a partir del estado recibido del servidor, y **Application**, que posee los métodos y funciones necesarios para la inicialización de la interfaz gráfica y la ejecución de ésta.

La clase **ApFrameListener** hereda de las clases **OGRE::FrameListener**, encargada de estar a la espera de eventos de imagen, y **OIS::KeyListener**, encargada de estar a la espera de eventos de teclado, pertenecientes a los API de OGRE3D y OIS respectivamente. Los métodos y funciones que posee son:

```
ApFrameListener(Camera* cam, Entity *ent, OIS::Keyboard *keyboard,
SceneManager *sceneMgr, TextAreaOverlayElement* textArea, Overlay*
overlay);
```

Constructor de la clase que se encarga de inicializar sus variables privadas con los valores pasados por parámetro, así como de conectarse con el servidor, a partir de los valores del archivo de configuración *gui.ini*.

```
bool frameStarted(const FrameEvent &evt);
```

Bucle principal del cliente GUI que se encarga de actualizar cada *frame*[5] de la interfaz gráfica. Devuelve `true` en caso de continuar con la ejecución, y `false` en caso contrario.

El algoritmo sigue la siguiente estructura:

- Capturar los eventos de teclado.
- Recibir el estado del servidor y almacenarlo en el archivo *gui.state*.
- Llamar al método `updateScene()` para actualizar las entidades gráficas.
- Eliminar el archivo *gui.state* para evitar conflictos.
- Esperar 1 segundo para visualizar la acción realizada por el actor actual.
- Llamar al método `sendCommand(…)` con el string **GO** como parámetro para que el servidor continúe su ejecución.

```
bool keyPressed(const OIS::KeyEvent &e);
```

Detecta cuándo y qué tecla ha sido pulsada, para poder realizar alguna acción, como el cambio del ángulos de las cámaras (con las teclas de la 1 a la 5) o la salida del cliente GUI (con la tecla ESC). Siempre devuelve `true`.

```
bool keyReleased(const OIS::KeyEvent &e);
```

Detecta cuándo y qué tecla ha dejado de ser pulsada, para poder realizar alguna acción. Siempre devuelve `true`. Actualmente esta función no se utiliza.

[5] Frame: fotograma o imagen particular dentro de la secuencia de imágenes que compone una animación.

```
void createObject(string type, string obj, string name, string angle, string graphic, string eState, string x, string y, string z, string sizeX, string sizeY, string sizeZ);
```

Crea una entidad, con su respectivo nodo asociado, según la clase **type** de la ontología a la que pertenezca, a partir del resto de atributos pasados por parámetro. También añade el identificador del objeto al archivo *createdEntities*, del cual se muestra un ejemplo de su contenido en la Figura 37.

```
CLIPS_ACTOR_VEDSGAPAIY
Refrigerator_1_id
Book_1_id
Computer_1_id
Ball_1_id
Bed_1_id
Bed_2_id
Sofa_1_id
DS_WALL_NORTH_ID
DS_WALL_EAST_ID
DS_WALL_SOUTH_ID
DS_WALL_WEST_ID
DS_WALL_1_ID
DS_WALL_2_ID
DS_WALL_3_ID
DS_WALL_4_ID
DS_WALL_5_ID
DS_WALL_6_ID
DS_WALL_7_ID
DS_WALL_8_ID
DS_WALL_9_ID
DS_WALL_10_ID
DS_WALL_11_ID
DS_FLOOR_ID
```

Figura 37: Ejemplo del contenido del archivo createdEntities.

```
void deleteEntities();
```

Elimina las entidades creadas que ya no se encuentran en el estado actual, a partir de los archivos *createdEntities*, que contiene el nombre de las entidades creadas, y *stateEntities*, que contiene el nombre de las entidades del estado actual del juego, del cual se muestra un ejemplo de su contenido en la Figura 38.

```
Refrigerator_1_id
Book_1_id
Computer_1_id
Ball_1_id
Bed_1_id
Bed_2_id
Sofa_1_id
DS_WALL_NORTH_ID
DS_WALL_EAST_ID
DS_WALL_SOUTH_ID
DS_WALL_WEST_ID
DS_WALL_1_ID
DS_WALL_2_ID
DS_WALL_3_ID
DS_WALL_4_ID
DS_WALL_5_ID
DS_WALL_6_ID
DS_WALL_7_ID
DS_WALL_8_ID
DS_WALL_9_ID
DS_WALL_10_ID
DS_WALL_11_ID
DS_FLOOR_ID
CLIPS_ACTOR_VEDSGAPAIY
```

Figura 38: Ejemplo del contenido del archivo stateEntities.

```
string getCellPosition(string cell, const char* slot);
```

Devuelve un string con el valor del slot **`slot`** del objeto de la clase **Cell** **`cell`** de la ontología, perteneciente al archivo auxiliar *gui.state* que contiene el estado actual del juego.

```
void updateScene();
```

Se encarga de representar el estado del juego según la clase de la ontología a la que pertenecen las entidades.

El algoritmo del método se detalla a continuación:

- Mientras que no sea fin de fichero del archivo *gui.state*:
  - Leer un objeto del archivo *gui.state*.
  - Si la clase del objeto leído es **Wall**, **Bed**, **Sofa**, **Computer**, **Refrigerator**, **Book**, **Ball**, **Shower** o **Actor**:
    - Añadir el identificador del objeto al archivo *stateEntities*.
    - Almacenar los valores de los atributos: name_, angle, entityGraphic, entityState, x, y, z, sizeX, sizeY y sizeZ.
    - Si existe el nodo asociado a la entidad del objeto:
      - Si el objeto es de la clase **Actor** o **Book** o **Ball**:
        - Actualizar la nueva posición.
    - Sino (la entidad no ha sido creada):
      - Llamar al método **`createObject`**`(…)` utilizando como parámetro los valores de los atributos guardados.
  - Sino, si la clase del objeto leído es **BuyFood**, **BuyDrink**, **PickUpFood**, **PickUpDrink**, **Read**, **Play**, **Washed**, **Resting**, **Working, PickUpAction**, **PutDownAction**, **Comunication** o **VerbalCommunication**:
    - Almacenar los valores de los atributos: actor y objeto al que se refiere la acción.
    - Si existen el nodo asociado al actor:
      - Hacer visible la esfera del actor, para indicar que está realizando una acción.
      - Indicar la acción que está realizando el actor con el objeto utilizado.
      - Si existe el nodo asociado al objeto y la clase del objeto leído es **BuyFood**, **PickUpFood**, **BuyDrink**, **PickUpDrink**, **Washed**, **Resting**, **Working** o **VerbalCommunication**:
        - Orientar al actor hacia el objeto referido.
  - Sino, si la clase del objeto leído es **MoveAction** o **FindFreeCell**:
    - Almacenar los valores de los atributos: actor y la celda destino.
    - Si existe el nodo asociado al actor:
      - Hacer invisible la esfera del actor, para indicar que no está realizando ninguna acción (sólo se está moviendo).
      - Orientar al actor hacia la celda destino.
      - Actualizar la posición del actor
  - Leer un nuevo objeto del archivo *gui.state*.
- Llamar al método **`deleteEntities`**`()` para eliminar las entidades que ya no se encuentran en el estado actual.

La clase **Application** posee los siguientes métodos y funciones:

```
void createRoot();
```

Crea un objeto de la clase **Root** perteneciente al API de OGRE3D, encargado de controlar toda la interfaz gráfica.

```
void defineResources();
```

Genera una estructura con todos los recursos definidos en el archivo *resources.cfg*.

```
void setupRenderSystem();
```

Configura el sistema de renderizado de la interfaz gráfica.

```
void createRenderWindow();
```

Crea la ventana de renderizado donde se visualizará la interfaz gráfica.

```
void initializeResourceGroups();
```

Inicializa los recursos anteriormente definidos.

```
void createLights();
```

Crea las luces del escenario a partir del archivo de configuración *lights*.

```
void createCameraNodes();
```

Crea los nodos de cámara, necesarios para elegir entre las distintas vistas del escenario, a partir del archivo de configuración *cameraNodes*.

```
void setupScene();
```

Configura la escena creando la cámara, las luces y los nodos de cámara.

```
void setupInputSystem();
```

Configura el sistema de entrada, formado por el teclado y el ratón, para que pueda interactuar con la interfaz gráfica.

```
void createFrameListener();
```

Crea un objeto de la clase **ApFrameListener** y lo inicializa.

```
void startRenderLoop();
```

Inicia el bucle de renderizado de la interfaz gráfica.

```
void go();
```

Realiza una llamada a todos los métodos anteriores en el orden dado (excepto **createLights()** y **createCameraNodes()** que son llamados en **setupScene()**).

```
~Application();
```

Destructor de la clase que se encarga de eliminar las variables privadas.

Además de las clases anteriores, el cliente GUI tiene un programa principal para iniciar la ejecución de éste:

```
int main(int argc, char **argv);
```

Función principal del cliente GUI que se encarga de hacer una llamada al método **go()** para iniciar el bucle de renderizado. No se requieren parámetros para iniciar la ejecución del cliente GUI.

4.4.3. Comunicación con el servidor: serverCommunication

Para establecer y mantener la comunicación entre el cliente GUI y el servidor, se han implementado un conjunto de métodos y funciones pertenecientes al archivo *serverCommunication.ccp*, definidos en su cabecera *serverCommunication.hpp*.

La mayoría de los métodos y funciones que implementan estos archivos son los mismos que los utilizados en los demás clientes, pero optimizados para C++. Los métodos y funciones más importantes son:

```
int serverConnect(const char *serverhost, int serverport,
const char *stage);
```

Devuelve el descriptor del *socket* de la máquina que contiene el servidor con la dirección IP **serverhost** al cual intenta conectarse el cliente mediante el puerto **serverport** para acceder al escenario **stage**.

```
buffer *receiveState(int ss);
```

Devuelve una estructura de tipo *buffer*[6] que contiene el estado recibido del *socket* del servidor **ss**.

[6] Buffer: región de la memoria de un ordenador reservada para el almacenamiento temporal de información digital.

```
void sendCommand(int ss, const char *c);
```

Envía la cadena `c` al servidor del *socket* cuyo descriptor es `ss`. Se utiliza para enviarle los comandos del protocolo de comunicación **GO** y **EX** al servidor.

En caso de que ocurra un error en alguno de estos tres métodos o funciones previos y produzca la finalización del cliente GUI, se generará el archivo *serverCommunication.log*, que contendrá información sobre el error ocasionado.

4.4.4. Funciones string auxiliares: auxiliarStringFunctions

Para facilitar la utilización de estructuras de tipo string, se han implementado las siguientes funciones, pertenecientes al archivo *auxiliarStringFunctions.cpp*, definidas en su cabecera *auxiliarStringFunctions.hpp*, y utilizadas en el archivo *gui.cpp*:

```
int countCharacter(string str, char c);
```

Devuelve el número de apariciones del carácter `c` dentro del string `str`.

```
string getSlotValue(string str, const char* s, bool c);
```

Devuelve un string con el valor del slot `s` dentro del string `str` que debe tener el siguiente formato: `(slotName slotValue)`. El parámetro `c` se utiliza para indicar si el valor del slot va a contener comillas al inicio y al fin, para eliminarlas (`true`) o no (`false`) antes de devolverlo.

```
string getObjectName(string str);
```

Devuelve un string con el nombre del objeto que está entre corchetes dentro del string `str`.

4.4.5. Ampliación del módulo del servidor

La implementación del cliente GUI ha llevado consigo varias ampliaciones en los siguientes archivos del directorio del servidor ("*ai-live/server-source/*"):

server.c

En el submódulo principal del servidor se ha modificado el código dentro del bucle principal del programa, para que el servidor se mantenga a la espera, una vez ha enviado el estado al cliente GUI, hasta que reciba de éste la confirmación de que puede continuar con la ejecución, o bien que ha finalizado su ejecución.

Además, se ha modificado el hecho de que el servidor envíe en cada iteración el estado al cliente GUI, haciendo que sólo lo haga en caso de haber algún actor interactuando en el sistema.

server.clp

En el submódulo de IA del servidor se ha implementado un método llamado **`printInstances()`**. Este método imprime las instancias que tienen información gráfica del escenario en el archivo de texto *id_stage.state*, donde *id_stage* es el identificador del escenario, para representarlas con la interfaz gráfica del cliente GUI. No se debe confundir con el método **`printInstances(?actorId)`** anteriormente implementado, el cual imprime las instancias referidas a un actor.

El método ha de llamarse en el consecuente del resto de reglas, aunque debe aparecer antes de eliminar la acción para aquellas reglas que utilizan objetos de clases que heredan de **ClientAction** (*véanse las Figuras 32 y 33*).

A continuación se muestra el algoritmo del método implementado:

- Para todos los objetos *id_stage* de la clase **Stage**:
  - Redirigir el flujo de información al archivo *id_stage.state*.
  - Imprimir la información del escenario.
  - Para todos los objetos de la clase **Entity** o que hereden de ésta, cuyo escenario sea *id_stage*:
    - Imprimir la información del objeto de la clase **Entity**.
  - Para todos los objetos de la clase **Cell**, cuyo escenario sea *id_stage*:
    - Imprimir la información del objeto de la clase **Cell**.
  - Para todos los objetos de la clase **ClientAction** o que hereden de ésta, cuyo escenario sea *id_stage*:
    - Imprimir la información del objeto de la clase **ClientAction**.
  - Cerrar redirección del flujo de información al archivo *id_stage.state*.

emotional_engine_server.clp

En el submódulo de IA emocional del servidor se ha introducido una llamada al método **`printInstances()`**, incluido en el archivo *server.clp*, en el consecuente de las reglas **`Gossip`** y **`TellLikenessShared`**, para poder representar acciones comunicativas verbales en la interfaz gráfica.

4.4.6. Archivos de configuración

A continuación se muestran los archivos de configuración necesarios para la ejecución del cliente GUI implementado, tanto los de configuración general, como los específicos para cada escenario.

4.4.6.1. Archivos de configuración general

gui.ini

Es el archivo de configuración inicial del cliente GUI, similar al archivo de configuración inicial del resto de clientes. Posee el siguiente formato:

- `host=hostName`: `hostName` es la dirección IP del ordenador donde se encuentra el servidor.
- `port=portNumber`: `portNumber` es el puerto al que se va a conectar el cliente GUI con el servidor.
- `stage=stageName`: `stageName` es el escenario que va a representar la interfaz gráfica del cliente GUI.

media.cfg

Contiene todos los nombres de los modelos que se pueden utilizar en la interfaz gráfica, con el formato: `Mesh=modelName.mesh`, donde `modelName` es el nombre del modelo con la extensión `.mesh`.

Deben aparecer, con el formato anterior, una lista con los nombres de los archivos que están en el directorio "*ai-live/client-gui/Media/models/*".

plugins.cfg

Contiene todos los plugins usados en la interfaz gráfica.

Se debe indicar inicialmente el directorio donde se encuentran estos plugins, con el formato: `PluginFolder=pluginPath`, donde `pluginPath` es el directorio donde se encuentran los plugins que utiliza la interfaz gráfica.

Los plugins utilizados deben aparecer en una lista, con el siguiente formato: `Plugin=pluginName.so`, donde `pluginName` es el nombre del plugin utilizado con la extensión `.so`.

resources.cfg

Contiene los directorios donde buscar los recursos necesarios para representar las entidades en la interfaz gráfica. Está dividido en dos secciones, encabezadas con las siguientes etiquetas:

- `[Bootstrap]`: en ella se indica la lista de archivos que contienen los packs de recursos, con el formato: `Zip=packsPath/zipName.zip`, donde `packsPath` es el directorio donde se encuentran los packs y `zipName` es el nombre del pack con extensión `.zip`.

- `[General]`: en ella se indica la lista de directorios que contienen los recursos, con el formato: `FileSystem=resourcePath`, donde `resourcePath` es el directorio donde se encuentran los recursos a utilizar.

4.4.6.2. Archivos de configuración específicos

cameraNodes

Archivo de configuración que contiene una estructura repetida cinco veces, una para cada nodo de cámara, con distintos valores para representar las cinco posibles vistas (global, noreste, noroeste, suroeste y sureste) del escenario. Esta configuración debe ser modificada según las dimensiones del escenario.

A continuación se muestra el formato de la estructura:

- `node=camNode`: `camNode` es el nombre del nodo de cámara.
- `x=xPosition`: `xPosition` es un número real que representa el valor X del nodo de cámara.
- `y=yPosition`: `yPosition` es un número real que representa el valor Y del nodo de cámara.
- `z=zPosition`: `zPosition` es un número real que representa el valor Z del nodo de cámara.
- `roll=xRotation`: `xRotation` es un número entero que representa el ángulo de rotación de la cámara en grados respecto al eje X.
- `yaw=yRotation`: `yRotation` es un número entero que representa el ángulo de rotación de la cámara en grados respecto al eje Y.
- `pitch=zRotation`: `zRotation` es un número entero que representa el ángulo de rotación de la cámara en grados respecto al eje Z.

lights

Archivo de configuración que contiene una estructura para representar las luces del escenario. Ésta se puede reutilizar para representar más luces. Esta configuración debe ser modificada en función de las dimensiones del escenario.

A continuación se muestra el formato de la estructura:

- `light=lightName`: `lightName` es el nombre de la luz.
- `x=xPosition`: `xPosition` es un número real que representa el valor X de la luz.
- `y=yPosition`: `yPosition` es un número real que representa el valor Y de la luz.
- `z=zPosition`: `zPosition` es un número real que representa el valor Z de la luz.

4.4.7. Representación del estado del juego

Una vez que el cliente GUI recibe el estado gráfico del juego, almacena los valores leídos de cada uno de los objetos en variables, para luego poder representarlos mediante entidades gráficas, ya sea creándolas, actualizándolas o eliminándolas.

4.4.7.1. Ejes de coordenadas, tamaño y posición de los objetos

Para representar las entidades gráficas, hay que tener en cuenta que en la ontología se utiliza una configuración distinta de los ejes de coordenadas a la usada en la interfaz gráfica, mostrada posteriormente en la Figura 39.

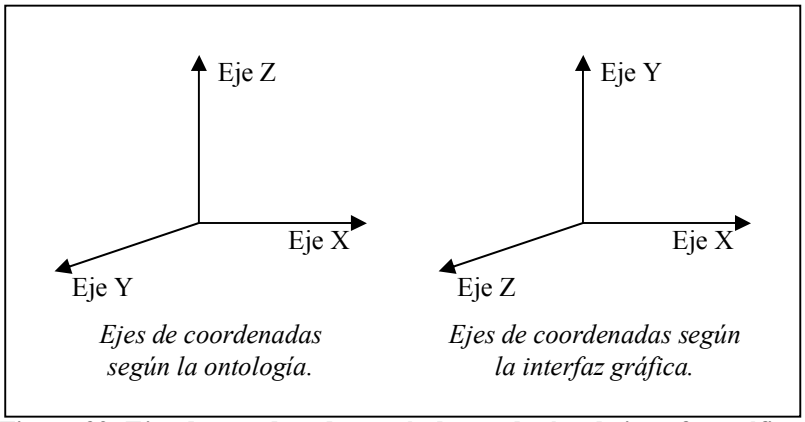

Figura 39: Ejes de coordenadas según la ontología y la interfaz gráfica.

Tanto la ontología como la interfaz gráfica utilizan el eje X para indicar la longitud de los objetos, pero con respecto a los otros dos ejes, la ontología utiliza el eje Y para indicar la anchura y el eje Z para indicar la altura del objeto, a diferencia de la interfaz gráfica, que lo hace al contrario.

Es por ello por lo que se realiza una trasposición de las variables *y* por *z*, y *sizeY* por *sizeZ*, para poder representar las entidades de forma correcta.

Con respecto al tamaño (*sizeX*, *sizeY* y *sizeZ*) y a la posición (*x*, *y* y *z*) de los objetos, 1 unidad en la ontología se representa con 100 pixeles en la interfaz gráfica.

4.4.7.2. Entidades gráficas asociadas a los objetos

Otro aspecto que afecta a la representación del estado del juego es el tamaño por defecto de las entidades gráficas utilizadas. Éstas están asociadas al objeto que se quiere representar y, en general, su tamaño no va acorde con las dimensiones del escenario, por lo que requieren de la aplicación de un factor de escala para visualizarlas correctamente.

Con respecto a la orientación de las entidades gráficas utilizadas, debe aplicarse el factor de rotación (*angle*) proveniente del estado del juego, teniendo en cuenta la orientación inicial que posee cada objeto.

4.4.7.3. Acción realizada por el actor

En el momento en el que un actor está realizando una acción, se muestra ésta mediante un cuadro de texto en la parte superior izquierda de la interfaz gráfica.

El cuadro de texto muestra tanto el actor que realiza la acción como el objeto que está usando o el actor con el que se está comunicando.

4.4.7.4. Esfera asociada al actor

Cuando se crea una entidad de la clase **Actor**, se le asigna a ésta otra entidad que representa una esfera sobre la cabeza del actor.

Esta esfera se hace visible en el momento en que el actor realiza una acción. Una vez que el actor se mueve de una celda a otra, o de un escenario a otro, la esfera desaparece hasta que el actor realice una nueva acción que no sea de desplazamiento.

Cuando se termina la ejecución del cliente que controla al actor, se elimina la entidad tanto de la esfera, como la del actor.

4.4.7.5. Cambio de escenario

En el juego se pueden utilizar diversos escenarios a la vez para que los actores interactúen entre ellos y con los objetos.

Cuando un actor realiza un cambio de escenario, se eliminan todos los objetos y actores del escenario del que provenía, y se generan los objetos y actores pertenecientes al nuevo escenario.

4.5. Manual de usuario

El manual de usuario muestra los requisitos generales de la aplicación, así como el conjunto de pasos que hay que realizar para la instalación, configuración y ejecución del juego AI-LIVE, incluyendo el cliente GUI que implementa la interfaz gráfica.

4.5.1. Requisitos generales

Se requiere una distribución de **Linux** (Debian, Ubuntu,…) para poder ejecutar la aplicación.

Se necesita también **PHP**[7] **5** o superior (http://www.php.net/downloads.php) con **CLI**[8] para poder ejecutar los archivos *run* encargados de la ejecución del juego.

No se requerirá ninguna versión de CLIPS (http://clipsrules.sourceforge.net/) para la ejecución de los archivos *.clp*, ya que se encuentra integrado en el lenguaje C. Para poder llamar desde C a las funciones de CLIPS se incluyen las siguientes librerías: "*ai-live/common/clips/clips.h*" y "*ai-live/common/clips/main.h*".

[7] PHP (*PHP Hypertext Pre-processor* – Preprocesador de Hipertexto PHP): lenguaje interpretado, multiplataforma, libre, y de propósito general, que está diseñado especialmente para desarrollo Web, pero también puede emplearse para la creación de scripts.

[8] CLI (*Command Line Interface* – Interfaz de Línea de Comando): interfaz de programación de uso del servidor para PHP, que permite ejecutar aplicaciones o scripts implementadas en lenguaje PHP bajo línea de comandos.

4.5.2. Instalación

Para la ejecución del cliente GUI se requiere la distribución **Ubuntu 8.04** de **32 bits**, y se debe seguir la siguiente guía de instalación de **OGRE3D 1.6.5**:

- Acceder desde un terminal al directorio "*~/Desktop/*":
  ```
  cd ~/Desktop
  ```
- Instalar los paquetes necesarios:
  ```
  sudo apt-get install pkg-config build-essential autoconf automake libtool libzzip-dev libxt-dev libxxf86vm-dev libxrandr-dev libfreeimage-dev nvidia-cg-toolkit checkinstall libfreetype6-dev libpcre3-dev libopenexr-dev freeglut-dev mesa-common-dev libtiff4-dev libglademm-2.4-dev libcppunit-dev libxaw7-dev libxaw-headers libois-dev
  ```
- Descargar el plugin CEGUI:
  ```
  wget http://downloads.sourceforge.net/crayzedsgui/CEGUI-0.6.2.tar.gz
  ```
- Extraer el plugin CEGUI:
  ```
  tar xzf CEGUI-0.6.2.tar.gz
  ```
- Acceder al directorio "*CEGUI-0.6.2/*":
  ```
  cd CEGUI-0.6.2
  ```
- Ejecutar aclocal, bootstrap y configure:
  ```
  aclocal && ./bootstrap && ./configure
  ```
- Compilar el código e instalarlo:
  ```
  make && sudo make install
  ```
- Volver al directorio anterior:
  ```
  cd ..
  ```
- Descargar OGRE3D:
  ```
  wget http://downloads.sourceforge.net/ogre/ogre-v1-6-5.tar.bz2
  ```
- Extraer OGRE3D:
  ```
  tar xjf ogre-v1-6-5.tar.bz2
  ```
- Acceder al directorio "*ogre/*":
  ```
  cd ogre
  ```
- Ejecutar aclocal, bootstrap y configure:
  ```
  aclocal && ./bootstrap && ./configure
  ```
- Compilar el código e instalarlo:
  ```
  make && sudo make install
  ```
- Volver al directorio anterior:
  ```
  cd ..
  ```
- Ejecutar *ldconfig*:
  ```
  sudo ldconfig
  ```

Una vez instalado todo, se pueden borrar las carpetas iniciales del directorio "*~/Desktop/*".

Para la instalación de OGRE3D en otras distribuciones, se puede consultar el siguiente sitio Web: http://www.ogre3d.org/wiki/index.php/Building_From_Source.

4.5.3. Configuración

Seguidamente se detalla la configuración de los distintos módulos del juego AI-LIVE para su correcto funcionamiento.

4.5.3.1. Servidor

La configuración del servidor está definida en el archivo *server.ini*, incluido en el directorio "*ai-live/server-source/*". Ésta puede ser modificada, aunque se muestran a continuación los valores por defecto:

- `ontology=../ontology.clp`: `ontology.clp` contiene la ontología del juego.
- `initial_state=initial.state`: `initial.state` contiene el estado inicial del juego.
- `port=6969`: `6969` es el puerto al que tienen que conectarse los clientes.

4.5.3.2. Clientes CLIPS y manual

Los clientes CLIPS y manual requieren de la configuración del archivo *client.ini*, incluido en el directorio "*ai-live/client-source/*" para el cliente CLIPS o "*ai-live/client-manual-source/*" para el cliente manual. Ésta puede ser modificada, mostrándose a continuación los valores por defecto:

- `ontology=../ontology.clp`: `ontology.clp` contiene la ontología del juego. Debe ser la misma que la del servidor.
- `stage=DEFAULT_STAGE`: `DEFAULT_STAGE` es el escenario inicial del actor.
- `actor=DEFAULT_ACTOR`: `DEFAULT_ACTOR` es el perfil del actor que va a controlar el cliente.
- `host=localhost`: `localhost` es la dirección IP del ordenador donde se encuentra el servidor.
- `port=6969`: `6969` es el puerto al que se va a conectar con el servidor.

Los perfiles de cada actor (*DEFAULT_ACTOR.profile* y *DEFAULT_ACTOR2.profile*) se encuentran en el directorio "*ai-live/client-source/profiles/*" y "*ai-live/client-manual-source/profiles/*". En ellos se deben indicar las características básicas del actor (name_, gender, weight,…), otras relacionadas con el rol social (age, sex, status), y otras más específicas basadas en la personalidad (conscientiousness, extraversion, neuroticism, openness, agreeableness) y las emociones (valence y arousal).

4.5.3.3. Cliente GUI

El cliente GUI requiere de la configuración de los archivos *gui.ini*, *lights* y *cameraNodes*, incluidos en el directorio "*ai-live/client-gui/*":

- *gui.ini*: contiene la configuración para comunicarse con el servidor. Esta configuración puede ser modificada, mostrándose los valores por defecto:
    - `host=localhost`: `localhost` es la dirección IP del ordenador donde se encuentra el servidor.
    - `port=6969`: `6969` es el puerto al que se va a conectar con el servidor.
    - `stage=DEFAULT_STAGE`: `DEFAULT_STAGE` es el escenario inicial que va a representar la interfaz gráfica.

- *lights*: contiene una estructura para representar las luces. La estructura se puede reutilizar para representar más luces. Esta configuración debe ser modificada en función de las dimensiones del escenario. A continuación se muestra la estructura de una luz de ejemplo:

    - `light=Light1`: `Light1` es el nombre de la luz.
    - `x=6.5`: `6.5` es el valor X de la luz.
    - `y=9.5`: `9.5` es el valor Y de la luz.
    - `z=10`: `10` es el valor Z de la luz.

- *cameraNodes*: contiene una estructura repetida cinco veces, una para cada vista (global, noreste, noroeste, suroeste y sureste), con distintos valores para representar los nodos de cámara. Esta configuración debe ser modificada en función de las dimensiones de los escenarios. A continuación se muestra la estructura de una vista global de ejemplo:

    - `node=CamNode`: `CamNode` es el nombre del nodo de cámara.
    - `x=6.5`: `6.5` es el valor X del nodo de cámara.
    - `y=9.5`: `9.5` es el valor Y del nodo de cámara.
    - `z=25`: `25` es el valor Z del nodo de cámara.
    - `roll=0`: `0` es la rotación en grados respecto al eje X de la cámara.
    - `yaw=-90`: `-90` es la rotación en grados respecto al eje Y de la cámara.
    - `pitch=-90`: `-90` es la rotación en grados respecto al eje Z de la cámara.

Las dimensiones del escenario inicial se pueden obtener accediendo al archivo *initial.state* del directorio "*ai-live/server-source/*", y buscando los atributos length y width del objeto `DEFAULT_STAGE` (el valor dado al atributo `stage` en la configuración del archivo *gui.ini*) de la clase **Stage**.

4.5.4. Ejecución

Antes de ejecutar cualquiera de los módulos, se debe iniciar un terminal de comandos, accediendo desde éste al directorio "*ai-live/*" y ejecutando la sentencia `make`, lo cual creará los archivos ejecutables del servidor y los clientes.

Seguidamente, hay que comprobar en las propiedades de los archivos *run* del directorio del servidor y el de los clientes (excepto para el cliente GUI), que tienen permisos de ejecución. Si no es el caso, se les debe asignar esta propiedad.

En caso de que no permita ejecutar alguno de los módulos, una vez creados los archivos ejecutables *server*, *client*, *client-manual* y *gui*, se debe comprobar que tienen asignados permisos de ejecución.

4.5.4.1. Servidor

Para ejecutar el servidor es necesario iniciar un terminal de comandos, accediendo desde éste al directorio del servidor "*ai-live/server-source/*" y a continuación ejecutar la sentencia `./run` .

Para acabar con la ejecución, el usuario debe pulsar `Control+C` en el terminal donde ha ejecutado éste. Se recomienda terminar la ejecución de los clientes antes.

4.5.4.2. Cliente CLIPS

Para utilizar el cliente CLIPS es necesario haber ejecutado previamente el servidor.

Para ejecutarlo se requiere iniciar un terminal de comandos, accediendo desde éste al directorio "*ai-live/client-source/*" y seguidamente ejecutar la sentencia `./run` para ejecutar el actor por defecto o `./run client2` para ejecutar el segundo actor.

Para terminar la ejecución, basta con pulsar `Control+C` en el terminal donde se ha ejecutado el cliente CLIPS, en cuyo momento se generará un archivo de traza que contiene la secuencia de acciones realizadas por éste. Si sólo están el cliente GUI y el actual ejecutando, debe cerrarse antes el cliente GUI.

4.5.4.3. Cliente manual

Para utilizar el cliente manual es necesario haber ejecutado previamente el servidor.

Para ejecutarlo se requiere iniciar un terminal de comandos, accediendo ahora al directorio "*ai-live/client-manual-source/*" y después ejecutar la sentencia `./run` para ejecutar el actor por defecto o `./run client2` para ejecutar el segundo actor.

Por cada turno, se le mostrará al usuario el conjunto de acciones que puede realizar el actor en ese momento. Cada acción contiene un número que la identifica, su nombre, así como el conjunto de objetos que están involucrados en ella. Es por ello por lo que no se introducen parámetros para la ejecución de las acciones, ya que éstas contemplan todas las posibilidades que puede realizar el actor en cada turno.

El usuario debe introducir el número de la acción que desea que realice el actor, y pulsar `ENTER` para que se envíe la acción al servidor y éste pueda ejecutarla.

Si el usuario desea terminar la ejecución, debe pulsar `0` y después `ENTER`, en cuyo momento se generará el archivo de traza. Si sólo están el cliente GUI y el actual ejecutando, debe cerrarse antes el cliente GUI.

4.5.4.4. Cliente GUI

Para utilizar el cliente GUI es necesario haber ejecutado previamente el servidor y al menos un cliente (CLIPS o manual).

Para ejecutarlo se requiere iniciar un terminal de comandos, accediendo desde éste al directorio "*ai-live/client-gui/*" y a continuación ejecutar la sentencia `./gui`.

Una vez iniciado el cliente GUI, se le mostrará al usuario la ventana de configuración de la interfaz gráfica, en donde deberá seleccionar inicialmente el API OpenGL Rendering Subsystem, y después deberá configurar las siguientes opciones:

- Display Frequency: indica la frecuencia de actualización de la interfaz gráfica.
- FSAA (*Full-Scene Anti-Aliasing*): indica el nivel de difuminación de la escena.
- Full Screen: se utiliza para elegir si se desea ejecutar la interfaz gráfica en pantalla completa o en ventana.
- RTT (*Render To Texture*) Preferred Mode: indica el tipo de *buffer* de renderizado a emplear para la representación de la escena.
- VSync (*Vertical Synchronization*): se utiliza para elegir si se desea ejecutar la sincronización vertical o no.
- Video Mode: muestra las opciones de resolución de pantalla (en pixeles) para ejecutar la interfaz gráfica.

Tras aceptar la configuración anterior, se abrirá la ventana que muestra la interfaz gráfica. Con las teclas `1`, `2`, `3`, `4` y `5` se puede cambiar el ángulo de la cámara, y con la tecla `ESC` se termina la ejecución del cliente GUI (aunque no del juego).

4.6. Manual de referencia

El manual de referencia será de gran utilidad para que un desarrollador pueda mantener y ampliar la aplicación, ya que en él se muestran aspectos estructurales, técnicos y de contenido de ésta.

En las siguientes subsecciones se expone la jerarquía de archivos del juego AI-LIVE, y el formato y la descripción del contenido de los archivos de cada uno de los módulos del sistema.

4.6.1. Jerarquía de archivos

En las Figuras 40 y 41 se muestra la jerarquía de archivos del juego AI-LIVE, a excepción del directorio del cliente Prodigy ("*ai-live/client-prodigy/*") que ha quedado obsoleto. La jerarquía está estructurada en orden alfabético, teniendo en cuenta que los subdirectorios de cada directorio están situados al principio de éste.

Se le ha asignado a cada archivo un código compuesto de dos partes separadas por una barra diagonal (/). La primera indica la ubicación del archivo en el conjunto de directorios, representados mediante la inicial de las palabras que componen sus nombres; para diferenciar los subdirectorios, éstos se separan con un guión (-). La segunda parte muestra el orden alfabético que dicho archivo ocupa en su directorio.

"ai-live/"		
"client-gui/"		
"Media/"		
"materials/"	Recursos para representar los modelos	
"scripts/"	Scripts para representar los modelos	
"textures/"	Imágenes para representar los modelos	
"models/"	Modelos de entidades en formato *.mesh*	
"packs/"	Conjunto de recursos agrupados	
auxiliarStringFunctions.cpp	Funciones string auxiliares	**[CG/0]**
auxiliarStringFunctions.hpp	Funciones string auxiliares (cabecera)	**[CG/1]**
cameraNodes	Configuración de los nodos de cámara	**[CG/2]**
gui.cpp	Código C de la interfaz gráfica	**[CG/3]**
gui.ini	Configuración de la interfaz gráfica	**[CG/4]**
lights	Configuración de las luces	**[CG/5]**
Makefile	Script para compilar la interfaz gráfica	**[CG/6]**
media.cfg	Configuración de los *meshes*	**[CG/7]**
plugins.cfg	Configuración de los plugins	**[CG/8]**
resources.cfg	Configuración de los recursos	**[CG/9]**
serverCommunication.cpp	Funciones de comunicación	**[CG/10]**
serverCommunication.hpp	Funciones de comunicación (cabecera)	**[CG/11]**
"client-manual-source/"		
"profiles/"		
DEFAULT_ACTOR.profile	Perfil por defecto del actor	**[CMS-P/0]**
DEFAULT_ACTOR2.profile	Perfil 2 por defecto del actor	**[CMS-P/1]**
client.c	Código C del cliente	**[CMS/0]**
client.clp	Código de IA del cliente	**[CMS/1]**
client.ini	Configuración del cliente	**[CMS/2]**
client2.ini	Configuración del cliente 2	**[CMS/3]**
Makefile	Script para compilar el cliente	**[CMS/4]**
run	Script PHP para ejecutar el cliente	**[CMS/5]**
"client-source/"		
"profiles/"		
DEFAULT_ACTOR.profile	Perfil por defecto del actor	**[CS-P/0]**
DEFAULT_ACTOR2.profile	Perfil 2 por defecto del actor	**[CS-P/1]**
client.c	Código C del cliente	**[CS/0]**
client.clp	Código de IA del cliente	**[CS/1]**
client.ini	Configuración del cliente	**[CS/2]**
client2.ini	Configuración del cliente 2	**[CS/3]**
Makefile	Script para compilar el cliente	**[CS/4]**
run	Script PHP para ejecutar el cliente	**[CS/5]**

Figura 40: Jerarquía de archivos de AI-LIVE I.

"common/"		
"clips/"	Librerías de CLIPS	
"emotional_engine/"		
ee_clips_adapter.c	Adaptador de datos CLIPS	[C-EE/0]
ee_clips_adapter.h	Adaptador de datos CLIPS (cabecera)	[C-EE/1]
emotional_engine.c	Código C del motor emocional	[C-EE/2]
emotional_engine.h	Código C del motor emocional (cabecera)	[C-EE/3]
Makefile	Script para compilar el motor emocional	[C-EE/4]
buffer.c	Funciones para buffer de datos	[C/0]
buffer.h	Funciones para buffer de datos (cabecera)	[C/1]
clienttypes.h	Definición de tipos de clientes	[C/2]
rinput.c	Funciones de entrada y salida	[C/3]
rinput.h	Funciones de entrada y salida (cabecera)	[C/4]
"server-source/"		
clientlist.c	Código C para la lista de clientes	[SS/0]
clientlist.h	Código C para la lista de clientes (cabecera)	[SS/1]
emotional_engine_server.clp	Código de IA para el motor emocional	[SS/2]
initial.state	Estado inicial del juego	[SS/3]
Makefile	Script para compilar el servidor	[SS/4]
run	Script PHP para ejecutar el servidor	[SS/5]
server.c	Código C del servidor	[SS/6]
server.clp	Código de IA del servidor	[SS/7]
server.ini	Configuración del servidor	[SS/8]
Makefile	Script para compilar el juego	[0]
ontology.clp	Ontología del juego	[1]

Figura 41: Jerarquía de archivos de AI-LIVE II.

4.6.2. Archivos del módulo del servidor

Seguidamente se describe el formato y el contenido de los archivos que componen el directorio del servidor.

clientlist.c [SS/0]
Tipo: Código fuente
Lenguaje: ISO C99

Se encarga de gestionar la lista de clientes que interaccionan con el servidor. Implementa una lista dinámica que irá creciendo a medida que aumente el número de clientes conectados.

clientlist.h [SS/1]
Tipo: Cabecera
Lenguaje: ISO C99

Es la cabecera del archivo *clientlist.c* [SS/0] y define los tipos de datos y prototipos de sus métodos y funciones. Todos los módulos que utilicen la lista de clientes tendrán que incluir esta cabecera.

emotional_engine_server.clp **[SS/2]**
Tipo: Código fuente
Lenguaje: CLIPS

Éste es el submódulo de IA emocional del servidor. Posee distintos métodos, funciones y reglas de interacción social y comunicativa.

initial.state **[SS/3]**
Tipo: Definición de instancias
Lenguaje: CLIPS

Define el estado inicial de los objetos en el servidor. Todos los escenarios que formen parte del sistema deben definirse en este archivo, junto con su contenido.

Makefile **[SS/4]**
Tipo: Script
Lenguaje: Bash

Este script se utiliza para compilar y generar el archivo ejecutable del servidor (*server*).

run **[SS/5]**
Tipo: Script
Lenguaje: PHP

Este script carga las opciones del archivo *server.ini* **[SS/8]** y lanza la ejecución del servidor.

server.c **[SS/6]**
Tipo: Código fuente
Lenguaje: ISO C99

Submódulo principal del servidor. Se encarga de:

- Inicializar el estado de los objetos, cargando el archivo *initial.state* **[SS/3]**.
- Atender las conexiones de los clientes e iniciar sus sesiones.
- Mantener una lista de clientes conectados, utilizando el archivo *clientlist.h* **[SS/1]**.
- Dirigir los turnos de los clientes en el juego para enviarles el estado.
- Ejecutar las peticiones de los clientes, mediante el archivo *server.clp* **[SS/7]**.
- Actualizar el estado de los objetos.

server.clp **[SS/7]**
Tipo: Código fuente
Lenguaje: CLIPS

Éste es el submódulo de IA del servidor. Posee distintos métodos, funciones y reglas para realizar las acciones solicitadas por los clientes.

Genera los archivos: *id_client.state*, donde *id_client* es el identificador de un actor, el cual contiene el estado actual de éste; y *id_stage.state*, donde *id_stage* es el identificador de un escenario, el cual contiene el estado gráfico actual de éste.

server.ini **[SS/8]**
Tipo: Configuración
Lenguaje: INI

Archivo de configuración del servidor, en el cual se debe especificar: el directorio donde se encuentra la ontología (*ontology.clp* **[1]**), el directorio donde se encuentra el estado inicial del juego (*initial.state* **[SS/3]**) y el puerto de conexión al servidor.

4.6.2. Archivos del módulo del cliente CLIPS

A continuación se describe el formato y el contenido de los archivos que componen el directorio del cliente CLIPS.

DEFAULT_ACTOR.profile **[CS-P/0]** y *DEFAULT_ACTOR2.profile* **[CS-P/1]**
Tipo: Definición de instancias
Lenguaje: CLIPS

Este archivo se utiliza para definir el perfil del actor controlado por el cliente. Contiene una instancia de la clase **AddClient**, con los valores de los atributos del actor.

client.c **[CS/0]**
Tipo: Código fuente
Lenguaje: ISO C99

Submódulo principal del cliente. Se encarga de:

- Mantener la comunicación con el servidor.
- Recibir el estado que envía el servidor.
- Seleccionar mediante técnicas de IA la acción que se va realizar, utilizando el archivo *client.clp* **[CS/1]**.
- Enviar al servidor la acción que debe ejecutarse.

client.clp [CS/1]
Tipo: Código fuente
Lenguaje: CLIPS

Éste es el submódulo de ejecución del cliente CLIPS que utiliza IA. Posee distintos métodos, funciones y reglas para gestionar el comportamiento del cliente.

Genera el archivo auxiliar *current.action* que contiene la acción del actor que posee el turno.

client.ini [CS/2] y *client2.ini* [CS/3]

Tipo: Configuración
Lenguaje: INI

Archivo de configuración del cliente CLIPS, en el cual se debe especificar: el directorio donde se encuentra la ontología (*ontology.clp* [1]), el escenario donde debe acceder el actor, el perfil del actor controlado por el cliente, la dirección IP del ordenador que tiene el servidor, y el puerto de conexión al servidor.

Makefile [CS/4]
Tipo: Script
Lenguaje: Bash

Este script se utiliza para compilar y generar el archivo ejecutable del cliente CLIPS (*client*).

run [CS/5]
Tipo: Script
Lenguaje: PHP

Este script carga las opciones del archivo *client.ini* [CS/2] o *client2.ini* [CS/3] y lanza la ejecución del cliente CLIPS. Tras el fin de la ejecución, se generará el archivo de traza, que contiene la secuencia de acciones realizadas por el actor, así como los objetos utilizados en ellas.

4.6.3. Archivos del módulo del cliente manual

En esta subsección se describe el formato y el contenido de los archivos que componen el directorio del cliente manual.

DEFAULT_ACTOR.profile [CMS-P/0] y *DEFAULT_ACTOR2.profile* [CMS-P/1]
Tipo: Definición de instancias
Lenguaje: CLIPS

Este archivo se utiliza para definir el perfil del actor controlado por el cliente. Contiene una instancia de la clase **AddClient**, con los valores de los atributos del actor.

client.c **[CMS/0]**
Tipo: Código fuente
Lenguaje: ISO C99

Submódulo principal del cliente manual. Se encarga de:

- Mantener la comunicación con el servidor.
- Recibir el estado que envía el servidor.
- Seleccionar manualmente la acción que se desea realizar, utilizando el archivo *client.clp* **[CMS/1]**.
- Enviar al servidor la acción que debe ejecutarse.

client.clp **[CMS/1]**
Tipo: Código fuente
Lenguaje: CLIPS

Éste es el submódulo de ejecución del cliente manual que utiliza IA. Posee métodos, funciones y reglas para gestionar el comportamiento del cliente.

El archivo es el mismo que utiliza el cliente CLIPS (*client.clp* **[CS/1]**), aunque puede ser modificado definiendo nuevas reglas para variar las acciones a elegir por el cliente manual, teniendo en cuenta que estas reglas tienen que estar controladas en el archivo *server.clp* **[SS/7]** del servidor.

Genera el archivo auxiliar *current.action*.

client.ini **[CMS/2]** y *client.ini* **[CMS/3]**
Tipo: Configuración
Lenguaje: INI

Archivo de configuración del cliente manual, en el cual se debe especificar: el directorio donde se encuentra la ontología (*ontology.clp* **[1]**), el escenario donde debe acceder el actor, el perfil del actor controlado por el cliente, la dirección IP del ordenador que tiene el servidor, y el puerto de conexión al servidor.

Makefile **[CMS/4]**
Tipo: Script
Lenguaje: Bash

Este script se utiliza para compilar y generar el archivo ejecutable del cliente manual (*client-manual*).

run **[CMS/5]**
Tipo: Script
Lenguaje: PHP

Este script carga las opciones del archivo *client.ini* **[CMS/2]** o *client2.ini* **[CMS/3]** y lanza la ejecución del cliente manual. Tras el fin de la ejecución, se generará el archivo de traza.

4.6.4. Archivos del módulo del cliente GUI

Seguidamente se describe el formato y el contenido de los archivos que componen el directorio del cliente GUI.

auxiliarStringFunctions.cpp **[CG/0]**
Tipo: Código fuente
Lenguaje: ISO C++98

Implementa un conjunto de funciones auxiliares que facilitan la utilización de estructuras de tipo string.

auxiliarStringFunctions.hpp **[CG/1]**
Tipo: Cabecera
Lenguaje: ISO C++98

Es la cabecera del archivo *auxiliarStringFunctions.cpp* **[CG/0]** y define los prototipos de sus funciones.

Además, incluye la cabecera *clienttypes.h* **[C/2]** perteneciente al directorio "*ai-live/common/*", que define los tipos de clientes.

cameraNodes **[CG/2]**
Tipo: Configuración
Lenguaje: INI

Archivo de configuración en donde se especifican los datos de los nodos de cámara, utilizados para representar las cinco vistas posibles del escenario (global, noreste, noroeste, suroeste y sureste).

gui.cpp **[CG/3]**
Tipo: Código fuente
Lenguaje: ISO C++98

Submódulo principal y de ejecución del cliente GUI. Se encarga de:

- Mantener la comunicación con el servidor.
- Recibir el estado que envía el servidor.
- Representar las entidades obtenidas del estado recibido.
- Actualizar el estado de las entidades según el caso.

Genera los archivos auxiliares: *gui.state*, que contiene el estado actual del juego en cada turno; *createdEntities* que contiene el nombre de las entidades creadas, y *stateEntities*, que contiene el nombre de las entidades del estado actual del juego.

gui.ini **[CG/4]**
Tipo: Configuración
Lenguaje: INI

Archivo de configuración del cliente GUI, en el cual se debe especificar: la dirección IP del ordenador que tiene el servidor, el puerto de conexión al servidor, y el escenario que va a representar el cliente GUI.

lights **[CG/5]**
Tipo: Configuración
Lenguaje: INI

Archivo de configuración en donde se especifican los datos para representar las luces del escenario.

Makefile **[CG/6]**
Tipo: Script
Lenguaje: Bash

Este script se utiliza para compilar y generar el archivo ejecutable del cliente GUI (*gui*).

media.cfg **[CG/7]**
Tipo: Configuración
Lenguaje: OGRE3D config

Archivo de configuración que contiene todos los nombres de los modelos que se pueden utilizar en la interfaz gráfica.

plugins.cfg **[CG/8]**
Tipo: Configuración
Lenguaje: OGRE3D config

Archivo de configuración que contiene los plugins usados en la interfaz gráfica.

resources.cfg **[CG/9]**
Tipo: Configuración
Lenguaje: OGRE3D config

Archivo de configuración en donde se indican los directorios donde buscar los recursos necesarios para representar las entidades en la interfaz gráfica.

serverCommunication.cpp **[CG/10]**
Tipo: Código fuente
Lenguaje: ISO C++98

Implementa un conjunto de métodos y funciones necesarios para la lectura y escritura de *sockets*, y el uso de *buffers*.

Estos métodos y funciones son similares a los utilizados en los archivos comunes, pero optimizados para el lenguaje C++. Además contiene métodos y funciones para la comunicación con el servidor.

En caso de producirse un error en alguno de los métodos y funciones implementados, se generará el archivo *serverCommunication.log*, que contendrá la información relativa a dicho error.

serverCommunication.hpp **[CG/11]**
Tipo: Cabecera
Lenguaje: ISO C++98

Es la cabecera del archivo *serverCommunication.cpp* **[CG/10]** y define los tipos de datos y prototipos de sus métodos y funciones.

4.6.5. Archivos comunes

En esta subsección se van a describir los archivos comunes tanto para el servidor como los clientes.

Makefile **[0]**
Tipo: Script
Lenguaje: Bash

Este script se utiliza para compilar y generar los archivos ejecutables de todo el juego (*server*, *client*, *client-manual* y *gui*).

ontology.clp **[1]**
Tipo: Código fuente
Lenguaje: CLIPS

Define la jerarquía de clases que forma el modelo de conocimiento del sistema AI-LIVE. Esta ontología es común para el servidor y los clientes.

El directorio "*ai-live/common/clips*" contiene los archivos de códigos fuente y cabeceras necesarios para embeber funciones CLIPS en lenguaje C.

4.6.5.1. Archivos de los submódulos principales

buffer.c **[C/0]**
Tipo: Código fuente
Lenguaje: ISO C99

Implementa los métodos y funciones usados en la lectura y escritura de *buffers*.

buffer.h **[C/1]**
Tipo: Cabecera
Lenguaje: ISO C99

Es la cabecera para el archivo *buffer.c* **[C/0]** que define los tipos de datos y prototipos de sus métodos y funciones.

clienttypes.h **[C/2]**
Tipo: Cabecera
Lenguaje: ISO C99

Especifica las constantes utilizadas para definir los *flags* de los tipos de cliente. Estos *flags* son un entero de 8 bits que especifican las características de cada cliente, determinando cómo el servidor va a tratarlo y qué parte del protocolo utilizar.

rinput.c **[C/3]**
Tipo: Código fuente
Lenguaje: ISO C99

Implementa las funciones necesarias para la lectura y escritura de *socket*s.

rinput.h **[C/4]**
Tipo: Cabecera
Lenguaje: ISO C99

Es la cabecera para el archivo *rinput.c* **[C/3]** y define los tipos de datos y prototipos de sus funciones.

4.6.5.2. Archivos del motor emocional

ee_clips_adapter.c **[C-EE/0]**
Tipo: Código fuente
Lenguaje: ISO C99

Define métodos y funciones destinados a gestionar los datos proporcionados por CLIPS y el núcleo del motor emocional.

ee_clips_adapter.h **[C-EE/1]**
Tipo: Cabecera
Lenguaje: ISO C99

Es la cabecera para el archivo *ee_clips_adapter.c* **[C-EE/0]** y define los tipos de datos y prototipos de sus métodos y funciones. Todos los módulos que utilicen el motor emocional tendrán que incluir esta cabecera.

emotional_engine.c **[C-EE/2]**
Tipo: Código fuente
Lenguaje: ISO C99

Contiene todos los métodos y funciones necesarios para actualizar el estado emocional de los actores tras una acción verbal comunicativa.

emotional_engine.h **[C-EE/3]**
Tipo: Cabecera
Lenguaje: ISO C99

Es la cabecera para el archivo *emotional_engine.c* **[C-EE/2]** y define los tipos de datos y prototipos de sus métodos y funciones. Todos los módulos que utilicen el motor emocional tendrán que incluir esta cabecera.

Makefile **[C-EE/4]**
Tipo: Script
Lenguaje: Bash

Este script se utiliza para compilar el motor emocional.

5. Resultados

En este capítulo se muestran los resultados obtenidos tras la realización y pruebas del cliente GUI implementado.

5.1. Escenarios del juego AI-LIVE

Para la realización de las pruebas de la interfaz gráfica, se han creado dos escenarios: uno principal (Figura 42) de dimensiones 20x14x6 unidades, que contiene dos dormitorios, un salón, una sala de estudio, una sala de lectura y una cocina; y otro secundario (Figura 43) de dimensiones 4x4x1 unidades que simula el baño.

Figura 42: Escenario principal – Vista global.

Figura 43: Escenario secundario – Vista global.

En el escenario principal cada dormitorio posee una cama, el salón tiene un sofá y una pelota, en la sala de estudio hay un ordenador, la sala de lectura posee una mesilla con un libro, y la cocina tiene un frigorífico. El escenario secundario posee únicamente una ducha.

La idea de utilizar un escenario distinto para representar una habitación más (el baño), es debida a la necesidad de probar la funcionalidad de cambio de escenario implementada en el juego AI-LIVE.

Se ha utilizado una textura para el suelo que permite representar las posibles casillas que pueden ocupar los objetos de la interfaz gráfica, siendo cada una de éstas de tamaño 1x1 unidades.

Además, para hacer más visible el contenido de los dos escenarios, se han hecho tanto las paredes internas del escenario principal como las paredes externas del escenario secundario con una altura de 1 unidad.

5.2. Vistas de la interfaz gráfica

Con el fin de mejorar la visualización del contenido de los escenarios, se han creado cinco posibles vistas: global, noreste, noroeste, suroeste y sureste. Estas vistas se pueden configurar a través del archivo *cameraNodes* perteneciente al directorio "*ai-live/client-gui/*"

A continuación se muestran las distintas vistas para el escenario principal (Figuras de la 44 a la 47), aunque no aparecerá la vista global, ya que ésta se muestra en la Figura 42.

Figura 44: Escenario principal – Vista noreste.

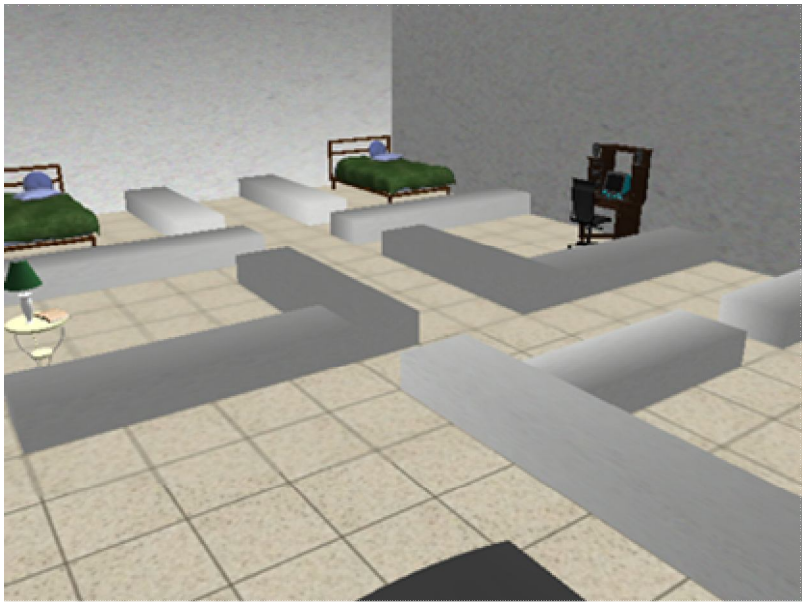

Figura 45: Escenario principal – Vista noroeste.

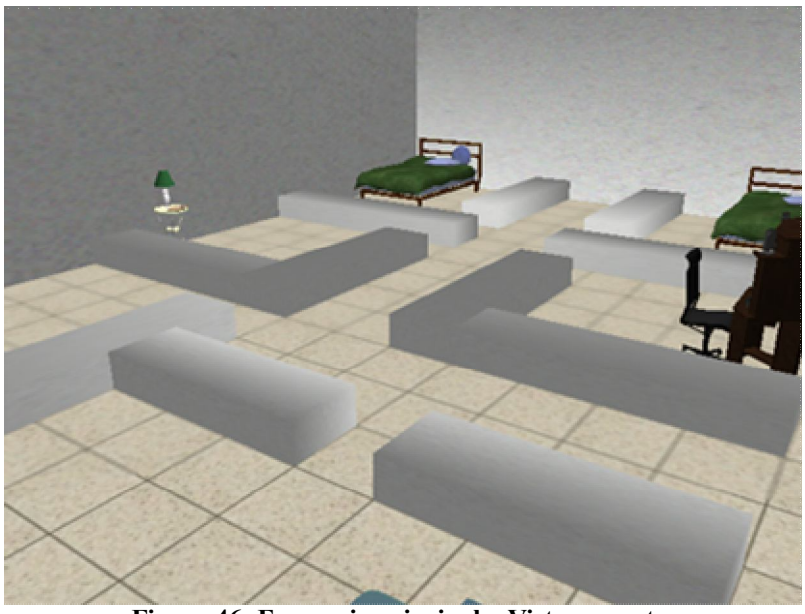

Figura 46: Escenario principal – Vista suroeste.

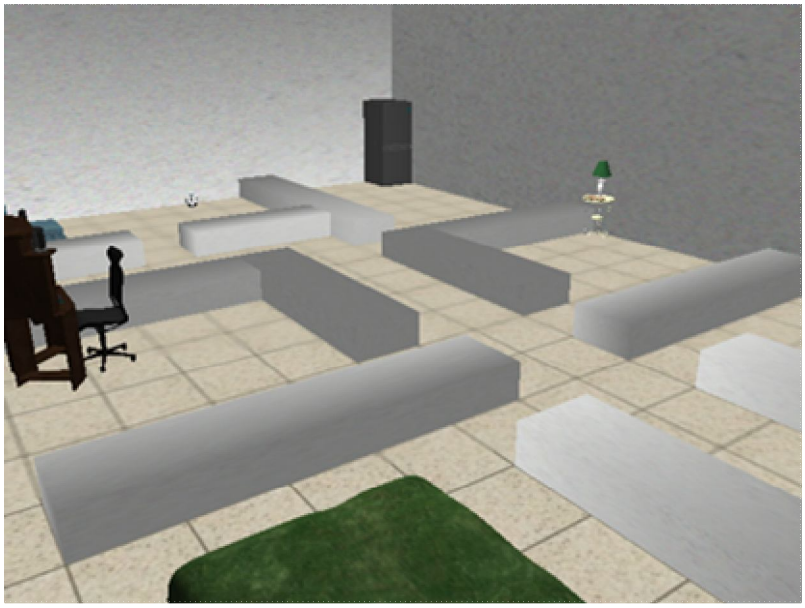

Figura 47: Escenario principal – Vista sureste.

Como se puede observar, estas vistas representan cada esquina del escenario principal, cubriendo todos los espacios posibles, para que el usuario pueda ver en todo momento dónde se encuentra el actor o los actores participantes.

Seguidamente se muestran las vistas para el escenario secundario (Figuras de la 48 a la 50), a excepción de la vista global, que se muestra en la Figura 43, y la vista sureste, que no aporta información gráfica.

Figura 48: Escenario secundario – Vista noreste.

Figura 49: Escenario secundario – Vista noroeste.

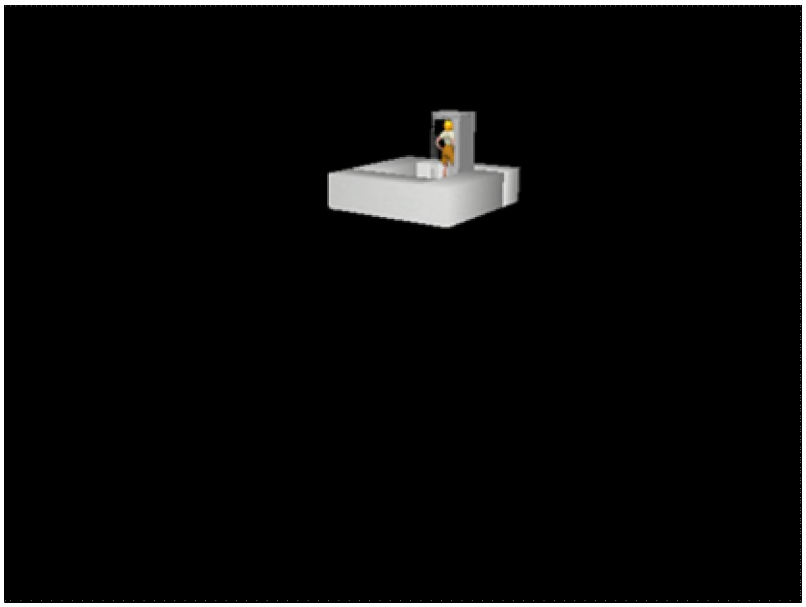

Figura 50: Escenario secundario – Vista suroeste.

A diferencia de las vistas del escenario principal, en el escenario secundario existen vistas que no son útiles, como son las vistas noreste, noroeste y sureste. Esto es debido a que el escenario secundario es más pequeño que el principal, y las posiciones de las cámaras son fijas para ambos escenarios.

5.3. Acciones de los actores

A continuación se van a mostrar algunos ejemplos de cómo se visualizan las acciones realizadas por los actores en el escenario.

Actualmente no existen animaciones de los actores, por lo que se ha creado una esfera sobre la cabeza del actor que aparecerá cuando éste esté realizando una acción, y un texto en la parte superior izquierda de la pantalla que muestra qué acción se está realizando y los actores y/u objetos intervinientes en ella.

Se debe mencionar que se han omitido la aparición de la esfera sobre la cabeza del actor y el texto que indica la realización de acciones, para las acciones de movimiento y cambio de escenario, ya que se pueden comprobar directamente viendo la interfaz gráfica y no aportan información útil para el usuario.

También cabe destacar que, por lo general, los actores se sitúan cerca del objeto (concretamente en la celda de acceso a éste), o bien sobre el objeto para poder interactuar con él, dependiendo si son objetos de gran tamaño o son objetos que se pueden coger.

En la Figura 51 se puede ver la representación de cómo el actor *Mike* trabaja con el objeto *Computer_1*.

Como se puede observar, el actor no se sienta en la silla del escritorio, ni el ordenador se enciende, debido a la falta de las animaciones para el actor y los objetos.

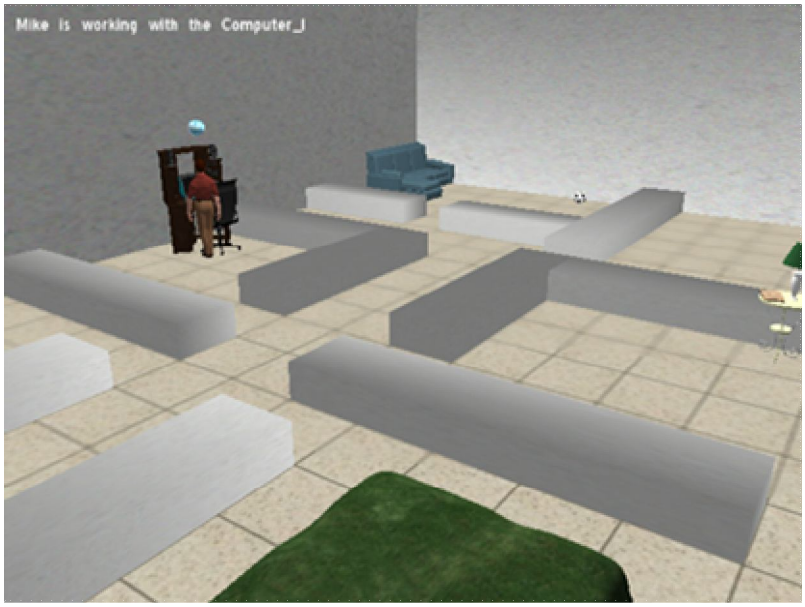

Figura 51: Representación de la acción trabajar (Working).

En la Figura 52 se puede percibir la representación de cómo el actor *Amy* lee el objeto *Book_1*.

Se debe mencionar para este ejemplo que el actor está situado sobre el objeto. Esto es debido a que el objeto tendría que representar un libro, y no una mesilla con un libro como es este caso.

Figura 52: Representación de la acción leer (Read).

En la Figura 53 se puede ver la representación de cómo el actor *Mike* coge el objeto *Water_6_id* del frigorífico.

Como se puede observar, el actor sólo se acerca al frigorífico, sin coger realmente el objeto, ya que aún no existe la representación del objeto *Water_6_id*. De la misma forma, el frigorífico no está abierto, debido a que no están generadas las animaciones de los objetos.

Figura 53: Representación de la acción coger una bebida (PickUpDrink).

En la Figura 54 se puede observar la representación de cómo el actor *Amy* se lava utilizando el objeto *Shower_1*.

Se puede ver que el actor no se desnuda, ni sale agua de la ducha. Esto es debido a lo mencionado anteriormente, la falta de las animaciones del actor y los objetos.

Figura 54: Representación de la acción lavarse (Washed).

En la Figura 55 se puede percibir la representación de cómo el actor *Mike* habla con el actor *Amy*.

Cabe destacar que para el actor *Amy* no aparece una esfera sobre su cabeza, indicando que está realizando una acción, ya que la acción de hablar en el juego AI-LIVE no es recíproca, es decir, que el actor *Amy* puede estar realizando otra acción o ninguna (en este caso) en su turno, mientras que el actor *Mike* le está hablando.

Figura 55: Representación de la acción hablar (VerbalCommunication).

Además de las acciones mostradas previamente, se han implementado para su visualización las siguientes acciones de la ontología: BuyFood, BuyDrink, PickUpFood, Play, Resting, PickUpAction, PutDownAction y Comunication.

6. Conclusiones

En este capítulo se muestran las conclusiones a las que se ha llegado tras la utilización del juego AI-LIVE, la elaboración del cliente GUI que implementa la interfaz gráfica 3D para éste, y la utilización de OGRE3D como motor gráfico.

6.1. Conclusiones respecto al juego AI-LIVE

El juego AI-LIVE ofrece la posibilidad de simular el comportamiento humano a través de seres virtuales que utilizan técnicas de IA, las cuales pueden ser completamente diferentes, haciendo que el juego sea más realista y creíble. Estos personajes son capaces de interactuar unos con otros y con su entorno mediante la realización de acciones.

A lo largo de la historia del proyecto AI-LIVE se han ido implementando nuevas funcionalidades, aumentando la complejidad del sistema, para poder conseguir, tanto en el comportamiento de los personajes como en la visualización del juego, una mayor aproximación a la realidad.

El modelo cliente-servidor utilizado en el juego permite una gran independencia respecto a la estructura y el comportamiento de los distintos módulos que componen la aplicación, proporcionando así una gran autonomía en caso de querer ampliarlos o introducir algún módulo nuevo.

El empleo de una ontología común para el servidor y los clientes es muy útil a la hora de definir el universo del juego a representar, ya que permite ampliar o modificar fácilmente las clases que lo forman.

La implementación previa del cliente manual para el trabajo dirigido *Acceso manual al juego AI-LIVE [UZQUIANO, 2008]* ha facilitado el entendimiento de la arquitectura del juego AI-LIVE y proporcionado conocimientos sobre la comunicación entre cliente y servidor.

6.2. Conclusiones respecto al cliente GUI implementado

La interfaz gráfica 2D previamente implementada ha servido de ayuda para la creación del nuevo cliente GUI, proporcionando la reutilización de funciones de comunicación entre el servidor y el cliente GUI. Sin embargo, todo lo relacionado con la visualización de los objetos ha sido elaborado desde el principio, ya que estas interfaces gráficas utilizan motores gráficos distintos.

El cliente GUI implementado para la elaboración de la interfaz gráfica 3D permite al usuario visualizar cómo interactúan los personajes unos con otros y con su entorno, de una manera más rápida y sencilla que como se hacía anteriormente.

Además de esto, la interfaz gráfica otorga una mayor interactividad con el usuario, ofreciendo la posibilidad de seleccionar entre distintos puntos de vista para obtener una visión íntegra del escenario.

Actualmente la interfaz gráfica se limita a construir gráficamente las entidades que recibe en el estado del juego enviado por el servidor, facilitando así su representación sin tener que controlar algunos aspectos gráficos, como son el control de colisiones entre entidades, o que éstas estén situadas dentro de los límites del escenario.

Un inconveniente que tiene la interfaz gráfica es que el código debe modificarse en posteriores ampliaciones del juego AI-LIVE en caso de que cambie la ontología, ya que la aplicación se encarga de crear, modificar o eliminar los gráficos en función de la clase a la que pertenezca cada entidad recibida en el estado del juego.

Con respecto a la eficiencia, la interfaz gráfica reduce la velocidad de ejecución del juego, debido a que tiene que recibir el estado del juego en el turno de cada cliente y actualizar la información gráfica, dejando al servidor en pausa hasta que finalice la representación de ésta. No obstante, mejora el tiempo de análisis de los resultados, al poder ver en tiempo real la interacción de los actores con su entorno.

6.3. Conclusiones respecto a la utilización de OGRE3D

OGRE3D posee una documentación muy completa, con una API *[API_OGRE3D]* que posee una amplia jerarquía de clases bien definidas, una Wiki *[WIKI_OGRE3D]* que proporciona ayuda tanto para la instalación como para la creación de contenido mediante tutoriales *[TUTORIALES_OGRE3D]*, así como foros *[FOROS_OGRE3D]* muy útiles a la hora de encontrar una solución a algún problema.

También hay que destacar que OGRE3D puede utilizar diferentes librerías que mejoran su funcionamiento o lo complementan, como son: los grafos de escena (véase el apartado **2.3.2. Scene graphs**); el motor de físicas, que simula modelos físicos a través de un conjunto de variables que controlan el estado de los objetos de la interfaz gráfica, como puede ser el control de colisiones; el motor de IA, que dota a ciertos elementos de la interfaz con comportamiento pseudointeligente, desde máquinas de estados y algoritmos de búsqueda hasta redes neuronales y algoritmos genéticos; el motor de sonido, que se encarga de reproducir y sincronizar la música y los efectos sonoros de la interfaz gráfica; y el motor de comunicaciones, que se encarga de gestionar las redes conectadas con la interfaz gráfica.

Debido a la restricción que tiene OGRE3D con respecto al formato de las entidades gráficas, se produce una desventaja: a pesar de que existen objetos bien definidos en la ontología, no hay suficientes entidades gráficas asociadas a éstos para poder representarlos en la interfaz gráfica. Pero esto se puede suplir utilizando programas de modelado de objetos 3D, como por ejemplo Blender, y plugins que permiten transformar objetos de otros tipos de archivo al formato que utiliza OGRE3D.

Hay que tener en cuenta que al ser OGRE3D un sistema multiplataforma, se requiere de unos conocimientos previos para su instalación, dependiendo de la plataforma en la que se vaya a ejecutar las aplicaciones. Esto suele ser un problema, ya que es de gran dificultad saber cuáles son los archivos o paquetes necesarios para su correcto funcionamiento. Sin embargo, existen guías de instalación muy útiles, en la Wiki de OGRE3D anteriormente mencionada, que subsanan este inconveniente.

Sopesando las ventajas (incluidas las citadas en el apartado **4.1. Introducción**) y los inconvenientes indicados previamente, se puede llegar a la conclusión de que la utilización de OGRE3D como motor gráfico 3D ha sido apropiada.

7. Futuras líneas

En este capítulo se proponen ampliaciones del trabajo realizado para ofrecer mejoras en la funcionalidad, la visualización y la eficiencia de la interfaz gráfica perteneciente al cliente GUI.

7.1. Mejorar los procesos de movimiento del actor

Debido a que en el juego AI-LIVE todavía no se ha implementado el concepto de tiempo, se ha creado la interfaz gráfica de manera que actualice en cada *frame* el estado del juego que recibe del servidor, impidiendo que un actor pueda realizar distintos tipos de movimiento.

Convendría que un actor adoptara distintos gestos según el estado de ánimo en el que se encuentre.

Con respecto a los movimientos de desplazamiento, sería interesante que cuando un actor se desplazara, fuera andando de una celda a otra, a través de un camino por el que pueda pasar sin obstáculos, no como lo hace actualmente mediante teletransporte.

7.2. Crear animaciones para los objetos

Actualmente los objetos que se encuentran en el escenario carecen de estados, por lo que no modifican su forma cuando se realiza alguna acción con ellos.

Convendría que cada objeto, o bien tenga animación propia, o bien existan otros objetos que sean iguales pero con distinta forma para poder cambiar unos por otros en función del estado en el que se encuentre.

Un ejemplo de esto sería un frigorífico, que permanezca cerrado cuando no se esté utilizando y abierto cuando se está cogiendo comida de él.

7.3. Mostrar los medidores de los actores

En la ontología del juego AI-LIVE existen actualmente unos medidores sobre el estado físico y psíquico de cada actor. Podría ser de gran utilidad mostrarlos en la interfaz gráfica para saber cuál es el estado de cada actor en todo momento.

7.4. Crear nuevas entidades

Sería conveniente la creación de nuevos objetos para que puedan interactuar los actores, ya sea diseñándolos utilizando una aplicación de modelado 3D, o bien exportándolos de otros formatos al formato *.mesh* utilizado por OGRE3D.

Para hacer el juego más realista, ayudaría la introducción de distintos tipos de actores según sus características propias (edad, sexo, peso, altura, etc).

7.5. Eliminar la utilización de archivos de texto intermedios

Aunque por ahora no existen problemas de eficiencia, para mejorar ésta vendría bien eliminar los procesos de lectura y escritura en archivos de texto intermedios, necesarios para la creación, actualización y eliminación de las entidades en la interfaz gráfica, como son el archivo de estado que recibe del servidor en cada turno (*gui.state*), así como los archivos de creación y estado de las entidades (*createdEntities* y *stateEntities*).

Una solución a este problema sería la utilización de estructuras de almacenamiento dinámico.

7.6. Crear dinámicamente los nodos de cámara y luces

En la interfaz gráfica implementada los nodos donde se sitúa la cámara que proporcionan las distintas vistas, son introducidos manualmente a través del archivo de texto *cameraNodes*. Convendría crear los nodos de cámara dinámicamente, a partir de las dimensiones del escenario.

Lo mismo ocurre con las luces, mediante el archivo de texto *lights*, ya que para que el escenario tenga una buena iluminación necesita de uno o varios focos de luz distribuidos correctamente.

7.7. Agregar funciones gráficas para el cliente manual

Debido a que el cliente manual permite al usuario elegir la acción que desea que ejecute el actor que controla, podría ser de gran ayuda mostrarle la lista de acciones que puede realizar mediante un menú dentro de la interfaz gráfica.

También estaría bien permitir al usuario interactuar con los objetos y otros actores mediante menús que tuvieran opciones seleccionables utilizando el ratón.

8. Bibliografía

A continuación se muestra la documentación que ha servido de ayuda para la elaboración del cliente GUI.

Documentación obtenida de Proyectos de Fin de Carrera y Trabajos dirigidos previos:

- *[PÉREZ, 2006] Proyecto de Fin de Carrera: AI-LIVE.*
  Miguel Alfonso Pérez Bonomini.
  Universidad Carlos III de Madrid. 2006.

- *[BENITO, 2007] Proyecto de Fin de Carrera: Necesidades básicas de actores en universos de realidad simulada.*
  Miguel Benito García.
  Universidad Carlos III de Madrid. 2007.

- *[JIMÉNEZ, 2008] Proyecto de Fin de Carrera: Diseño e implementación de un modelo comunicativo emocional para agentes virtuales en el universo AI-LIVE.*
  Marta Jiménez Matarranz.
  Universidad Carlos III de Madrid. 2008.

- *[UZQUIANO, 2008] Trabajo dirigido: Acceso manual al juego AI-LIVE.*
  Iván Uzquiano Mateo.
  Universidad Carlos III de Madrid. 2008.

Documentación obtenida de sitios Web:

- *[GRÁFICOS_3D] Wikipedia - Gráficos 3D por computadora.*
  http://es.wikipedia.org/wiki/Gráficos_3D_por_computadora

- *[API] Wikipedia - API.*
  http://es.wikipedia.org/wiki/API

- *[FRAMEWORK] Wikipedia - Framework.*
  http://en.wikipedia.org/wiki/Framework

- *[SCENE_GRAPH] Wikipedia - Scene graph.*
  http://en.wikipedia.org/wiki/Scene_graph

- *[GAME_ENGINE] Wikipedia - Game engine.*
  http://en.wikipedia.org/wiki/Game_engine

- *[TALÓN, 2005] Motores gráficos.*
  Rubén Talón Argente.
  Universidad de Valencia. 2005.
  http://informatica.uv.es/iiguia/IG/motores_graf.pps

- *[SEOANE, 2007] Herramientas de software libre para el desarrollo de aplicaciones de visualización 3D en tiempo real.*
  Antonio Seoane.
  II Jornadas de informática gráfica y software libre. 2007.
  http://stuff.gpul.org/2007_graficos/doc/2007_JGRAF_03_3Dtreal.pdf

- *[HISTORIA_VIDEOJUEGOS] Wikipedia - Historia de los videojuegos.*
  http://es.wikipedia.org/wiki/Historia_de_los_videojuegos

- *[SIMS] Wikipedia - Los Sims.*
  http://es.wikipedia.org/wiki/Los_Sims

- *[SIMS2] Wikipedia - Los Sims 2.*
  http://es.wikipedia.org/wiki/Los_Sims_2

- *[SINGLES] Wikipedia - Singles: Flirt up your life.*
  http://en.wikipedia.org/wiki/Singles:_Flirt_Up_Your_Life

- *[SECOND_LIFE] Wikipedia - Second Life.*
  http://es.wikipedia.org/wiki/Second_Life

Documentación de apoyo para la implementación del cliente GUI:

- *[ALTADILL, 2004] Tutorial de C++ - El diario de Peter Class.*
  Pello Xabier Altadill Izura. 2004.
  http://www.pello.info/filez/tutorialcpp.pdf

- *[OSORIO, 2006] Manual teórico-práctico de C++.*
  Alan D. Osorio Rojas. 2006.
  http://slent.iespana.es/docs/manualC++Public.pdf

- *[JUNKER, 2006] Pro OGRE 3D programming.*
  Gregory Junker. 2006.

- *[API_OGRE3D] API de OGRE3D.*
  http://www.ogre3d.org/docs/api/html/

- *[WIKI_OGRE3D] Wiki de OGRE3D.*
  http://www.ogre3d.org/wiki/index.php/Main_Page

- *[FOROS_OGRE3D] Foros de OGRE3D.*
  http://www.ogre3d.org/forums/

- *[TUTORIALES_OGRE3D] Tutoriales de OGRE3D.*
  http://www.ogre3d.org/wiki/index.php/Ogre_Tutorials