UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

INGENIERÍA

EN INFORMÁTICA

PROYECTO FIN DE CARRERA

MASYRO: Un sistema multi-agente para la optimización del renderizado

David Vallejo Fernández

Septiembre, 2006

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

Departamento de Informática

PROYECTO FIN DE CARRERA

MASYRO: Un sistema multi-agente para la optimización del renderizado

Autor: David Vallejo Fernández
Director: Carlos González Morcillo

Septiembre, 2006.

TRIBUNAL:

Presidente:
Vocal:
Secretario:

FECHA DE DEFENSA:

CALIFICACIÓN:

PRESIDENTE	VOCAL	SECRETARIO
Fdo.:	**Fdo.:**	**Fdo.:**

Este documento fue compuesto con LaTeX. Imágenes generadas con OpenOffice.

Resumen

Se puede entender el proceso de *Render* como el mecanismo utilizado para obtener una imagen a partir de la descripción de una escena tridimensional. Actualmente dicho proceso ha alcanzado un alto grado de realismo, mientras que el gran problema sigue siendo el tiempo de ejecución empleado para el mismo. Dentro del ámbito de este proyecto se ha desarrollado una aplicación que permite optimizar dicho tiempo de ejecución, empleando para ello una arquitectura multi-agente de propósito general y basándose en sistemas de reglas difusos.

Este proyecto está dedicado a mis padres,
por apoyarme todos estos años de estudio,
y a mi novia Alba,
por comprender mi dedicación
a estas pequeñas máquinas.

Agradecimientos

A Carlos González Morcillo, por su ayuda, dedicación, y profesionalidad.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Se puede entender el proceso de **síntesis de imagen fotorrealista** como aquel proceso que persigue la creación de imágenes sintéticas que no se puedan distinguir de aquellas captadas en el mundo real. Dicho proceso está divido en las siguientes fases: modelado, asignación de materiales y texturas, iluminación, y render.

La última de las fases mencionadas, es decir, la que se refiere al proceso de **renderizado**, es la que determina el color más apropiado que se asigna a cada píxel de la escena, y cuyo resultado es una imagen que representa la escena tridimensional. Dicho color vendrá determinado por diversos factores, como por ejemplo la geometría del objeto, la posición, el color de las fuentes de luz, la posición y orientación de la cámara, las propiedades de las superficies, etc.

La fase de renderizado ha sufrido una gran evolución a lo largo del tiempo. En los primeros años de estudio de este campo, la investigación centró su foco en cómo resolver problemas básicos, como por ejemplo la detección de superfices visibles, o el sombreado básico. Conforme el tiempo iba pasando y dichos problemas se iban solventando, el estudio se fue centrando en la creación de algoritmos de síntesis más realistas, que simularan el comportamiento de

la luz de la forma más fideligna posible. Entender la naturaleza de la luz y cómo se dispersa sobre el entorno es esencial para simular correctamente la iluminación.

Una importante distinción surge a la hora de elegir un modelo de iluminación, diferenciando entre **iluminación local**, donde sólo se considera iluminación desde las fuentes de luz, e **iluminación global**, donde se considera el transporte de luz en su conjunto. Así mismo, existen algoritmos de renderizado asociados a dichos modelos de iluminación. Sin embargo, se puede considerar que todas las alternativas de renderizado pretenden solucionar la ecuación general de renderizado, propuesta en 1986 por **Kajiya** [45]:

$$L_0(x, \overrightarrow{w}) = L_e(x, \overrightarrow{w}) + \int_\Omega f_r(x, \overrightarrow{w}', \overrightarrow{w}) L_i(x, \overrightarrow{w}')(\overrightarrow{w}'\overrightarrow{n}) d\overrightarrow{w}' \tag{1.1}$$

Dicha ecuación puede interpretarse como: "dada una posición x, y dirección $\overrightarrow{w}$ determinada, el valor de iluminación saliente L_0 es el resultado de sumar la iluminación emitida L_e y la luz reflejada. La luz reflejada (el segundo término de la suma de la ecuación anterior) viene dado por la suma de la luz entrante L_i desde todas las direcciones, multiplicando por la reflexión de la superficie y el ángulo de incidencia.

Para expresar la interacción de la luz en cada superficie, se utiliza la función de distribución de reflectancia biridireccional, que correspondería con el término f_r de la expresión anterior, y que es particular de cada superficie.

La etapa de renderizado, dentro del proceso de síntesis de imagen fotorrealista, supone uno de los principales cuellos de botella en lo que a desarrollo se refiere. La razón se debe a que los algoritmos empleados requieren una gran cantidad de cómputo. Además, estos algoritmos han sido ya muy estudiados, por lo que resulta complicado conseguir mejoras sustanciales en lo que a resultados se refiere. Por lo tanto, resulta interesante plantear métodos que reduzcan el tiempo empleado en el proceso de renderizado desde otro enfoque, es decir, desde otro punto de vista que no se centre en el método de simulación del comportamiento de la luz en cuestión.

Un aspecto que resulta interesante resaltar, ya en lo relativo a los parámetros que intervienen en la etapa de renderizado, es que existen ciertos parámetros cuyos valores hacen que el tiempo empleado en el renderizado se incremente, mientras que la diferencia de calidad con respecto a valores distintos de esos parámetros resulta inapreciable. Visto de otro modo, exis-

ten ciertas configuraciones que hacen que el tiempo final de renderizado sea alto, sin mejorar sustancialmente la calidad del resultado final en lo referente a la percepción del usuario.

1.1. Justificación del trabajo

La justificación de este trabajo reside en utilizar un enfoque distinto a la hora de abordar la etapa de renderizado, con el objetivo principal de reducir el tiempo empleado para ello. Debido a que existe una gran comunidad de desarrollo en lo que a síntesis de imagen fotorrealista se refiere, proyectos como éste son importantes porque las mejoras conseguidas afectan beneficiosamente a un gran número de personas. De hecho, existen muchas empresas y entidades que utilizan mecanismos de representación realista, como centros de investigación científica, empresas de publicidad, y entidades relacionadas con el sector del ocio (videojuegos y películas de animación).

1.2. Objetivos perseguidos

Con este proyecto se pretende distribuir el trabajo asociado a la etapa de renderizado entre distintos agentes *inteligentes*, utilizando para ello la instanciación concreta de un sistema multi-agente de propósito general. Bajo este enfoque general, se pretenden alcanzar los siguientes objetivos generales:

- Construir la arquitectura básica de un sistema multi-agente, siguiendo las guías del comité de estándares FIPA [18], que podrá ser adaptada a cualquier desarrollo del sistema multi-agente.

- Utilizar dicha arquitectura para llevar a cabo el proceso de renderizado de manera distribuida, es decir, instanciar la arquitectura básica creada en el punto anterior para realizar el renderizado paralelo de una escena de forma distribuida.

La consecución de estos objetivos generales se llevará a cabo mediante la realización de una serie de subobjetivos particulares:

- Reducir el tiempo empleado en el proceso de renderizado.

- Utilización de técnicas de estudio de la escena, previas al renderizado.
- Utilización de técnicas de *soft-computing* para ajustar los parámetros del renderizado final.
- Construcción de un sistema escalable a otras etapas del proceso de síntesis.
- Asegurar la portabilidad entre los principales sistemas operativos existentes.
- Uso de herramientas y tecnologías libres que aseguren la portabilidad.
- Empleo de estándares para la construcción del sistema.

Bajo este conjunto de objetivos generales, el objetivo principal consiste en realizar una primera aproximación a un sistema que optimice el proceso de renderizado en lo que a tiempo de ejecución se refiere, obteniendo parámetros de calidad similares a los que se obtendrían con los parámetros predefinidos por el usuario. Dicha optimización será posible gracias al uso de sistemas expertos, en concreto en la implementación actual del sistema se emplean sistemas de reglas difusos.

Por otra parte, también se pretende utilizar este proyecto como propuesta inicial para continuar con el desarrollo de un sistema multi-agente que optimice de manera general el proceso de renderizado, teniendo en cuenta los distintos factores que en el mismo intervienen.

1.3. Estructura del documento

Este documento está estructurado en seis capítulos y en dos anexos. El capítulo actual, relativo a la introducción, pretende ser una presentación, comentando el marco bajo el que se elabora este proyecto, los objetivos perseguidos, y la terminología empleada.

El segundo capítulo pretende ofrecer al lector una visión general de las distintas tecnologías y herramientas que se han utilizado para llevar a cabo la elaboración de este proyecto. Dicho capítulo se divide en los siguientes subapartados:

- Ingeligencia Artificial distribuida.
- *Middlewares*.

- Síntesis de imagen tridimensional.
- Análisis de tecnologías multiplataforma.

En lo que a Inteligencia Artificial distribuida se refiere, se realizará una introducción a los sistemas multi-agente y a las distintas metodologías que subyacen bajo el concepto de Ingeniería del Software en sistemas multi-agente. Así mismo, se hará un recorrido por los distintos estándares propuestos por FIPA, principal organismo responsable en estándares para la construcción de sistemas multi-agente. También se dedicará una sección a la representación del conocimiento, centrándose en los sistemas expertos.

En la sección de *middlewares* se realizará un recorrido por las distintas tecnologías existentes, haciendo especial hincapié en la elegida para abordar la parte de comunicaciones de este proyecto, ZeroC ICE.

En la parte de síntesis de imagen tridimensional se expondrá una introducción a dicho proceso, para pasar posteriormente a realizar un breve recorrido por los principales métodos de renderizados utilizados actualmente.

Por último, en la sección de análisis de tecnologías multiplataforma se estudiarán las tecnologías empleadas para llevar a cabo el procesamiento de imagen, utilizando *Python Imaging Library*, y para abordar el lenguaje de intercambio de información (DOM y XML).

El tercer capítulo trata de abordar de una manera más completa los objetivos que se persiguen con la elaboración de este proyecto, comentando las hipótesis de trabajo. El cuarto capítulo versa sobre el procedimiento de trabajo empleado para alcanzar los objetivos propuestos. En el quinto capítulo se realiza un análisis de los resultados obtenidos, que darán pie a las conclusiones y propuestas expuestas.

1.4. Terminología

En esta sección se tratará de definir de una manera clara los distintos términos utilizados a lo largo de este proyecto, centrándose principalmente en aquellos cuyo traducción al castellano no está muy clara o el término original en inglés está ampliamente aceptado y adoptado en la bibliografía en castellano.

- **Render**: proceso por el cual se determina el color más apropiada a cada píxel de la escena, y cuyo resultado es una imagen que representa al entorno tridimiensional.

- **Soft-computing**: colección de técnicas computacionales en informática que pretende estudiar, analizar, y modelar fenómenos complejos para los cuales las técnicas tradicionales no obtienen resultados completos o el tiempo o coste empleado es muy elevado. Tradicionalmente se han denominado técnicas de Inteligencia Artificial.

- **Middleware**: *software* de conectividad que ofrece unos determinados servicios para hacer posible el funcionamiento de aplicaciones distribuidas en distintos entornos de red.

- **Fuzzy**: difuso o borroso. Este adjetivo aparece en términos como *fuzzy sets* o *fuzzy logic*, traducidos como conjuntos difusos o lógica difusa, respectivamente.

- **Fuzzyfication**: proceso por el cual se lleva a cabo el paso de una variable física a una difusa o borrosa. En este documento se ha traducido como *borrosificación*.

- **Desfuzzyfication**: proceso inverso a la *borrosificación*, y traducido como *desborrosificación* en este documento. Sin embargo, también está ampliamente aceptado en castellano *Desfuzzyfication*.

Capítulo 2

Antecedentes, estado de la cuestión

2.1. Inteligencia Artificial distribuida

2.1.1. Introducción a los sistemas multi-agente

2.1.1.1. El concepto de agente

El término **agente** proviene del latín *agere*, que significa hacer. Agente deriva del participio *agens*, y expresa la capacidad de acción o actuación de una entidad.

Tras esta introducción etimológica, es importante resaltar que la mayoría de libros y tratados relacionados con los agentes no mantienen una definición común del concepto de agente. Sin embargo, y al estar hablando de productos de ingeniería, se pueden establecer unos criterios que permitan distinguir lo que es un agente de lo que no, ofreciendo un modelo razonable de sus características y de su comportamiento.

Una de las definiciones más aceptadas es la de Wooldridge [50], que concibe un agente como un sistema que reúne las siguientes características:

- **Autonomía**: un agente encapsula un estado (no accesible a otros agentes), y toma decisiones sobre qué hacer basándose en su estado, sin la intervención directa de otros.
- **Reactividad**: un agente está situado dentro de un entorno, y es capaz de realizar percepciones de ese entorno y responder en consecuencia a los cambios que en él se producen.
- **Pro-actividad**: un agente no actúa simplemente en respuesta a los cambios que se producen en su entorno, sino que también tiene la capacidad de tomar la iniciativa en lo que a ejecutar acciones se refiere.
- **Habilidad social**: un agente interactúa con otros agentes (y posiblemente también con personas) a través de algún tipo de lenguaje de comunicación de agentes, y típicamente tiene la habilidad de involucrarse en actividades sociales (como por ejemplo la resolución de problemas de forma cooperativa).

Además, existen ciertos atributos que los agentes pueden poseer, como por ejemplo:

- Razonamiento y aprendizaje, mediante los cuales los agentes son capaces de comportarse de manera inteligente.

- Movilidad, que permite a los agentes desplazarse entre los nodos de una red y ejecutarse en distintas plataformas.

2.1.1.2. Clasificación. Tipos de agentes

Se pueden establecer los criterios de clasificación de agentes desde distintos puntos de vista, desde sus características individuales hasta su utilidad, pasando por su comportamiento externo. Un modelo de clasificación bastante característico es el modelo de clasificación llamado de las **vocales**, propuesto por Y ves Demazeau. La **A** (de agente) caracteriza sus rasgos individuales: arquitectura, funcionamiento interno, complejidad, etc. La visión externa se desglosa en varios apartados: la **E** (de entorno) caracteriza los requisitos computacionales para que el agente funcione correctamente; la **I** (de interacción) considera las capacidades de comunicación del agente; la **O** (de organización) considera el papel del agente en el conjunto del sistema; y, por último, la **U** (de utilidad) hace referencia a la aplicación de la que forma parte el agente.

Según sus características individuales, los agentes pueden clasificarse en agentes reactivos y agentes cognitivos. Los **agentes reactivos** realizan tareas sencillas y su modelo computacional está basado en un ciclo *recepción de eventos externos/reacción*. La reacción consiste en la ejecución de procedimientos según el estado interno del agente. Además, un agente reactivo no realiza procesos de razonamiento, ni tiene ningún mecanismo explícito de representación del conocimiento. Por otra parte, los **agentes cognitivos** realizan tareas complejas. De hecho, utilizan algún tipo de representación explícita del conocimiento. Para realizar las tareas necesitan llevar a cabo procesos de razonamiento y otros procesos cognitivos, como por ejemplo la planificación y el aprendizaje. El modelo computacional de un agente cognitivo está basado en un ciclo *percepción-asimilación-razonamiento-actuación*.

Según el entorno en el que funcionan, existen agentes que requieren un entorno especial, o una plataforma *software* específica, y agentes que se ejecutan en las plataformas computacionales existentes. Un ejemplo de los **primeros** son los agentes móviles, donde cada plataforma proporciona los mecanismos para gestionar su ciclo de vida y para que se desplacen de un nodo a otro. Los **últimos** se crean mediante los recursos del sistema operativo y siguen el ciclo de vida de cualquier aplicación.

Según el modo de interacción, tenemos agentes especializados en la interacción con el usuario o **agentes de interfaz**, y agentes especializados en la interacción con los **recursos**.

Según el modo de organización, podemos considerar **agentes individuales**, que no tienen capacidad de cooperación y realizan sus tareas solos, y **agentes cooperativos**, que pueden realizar tareas solos o colaborando con otros agentes.

Según la utilidad, se puede clasificar a los agentes de acuerdo con la **finalidad** o el propósito con el que fueron construidos. Para ello se pueden seguir dos criterios: el dominio de aplicación y el tipo de tarea que realizan dentro de él.

2.1.1.3. Arquitecturas para construir agentes

En esta sección se presentarán arquitecturas para la construcción de agentes, destacando las arquitecturas deliberativas y comentando las arquitecturas reactivas.

Las **arquitecturas deliberativas** utilizan modelos de representación simbólica del conocimiento, y suelen estar basadas en la teoría clásica de planificación. Estos agentes parten de un estado inicial y son capaces de generar planes para alcanzar sus objetivos. Por tanto, un agente deliberativo es aquel que contiene un modelo simbólico del mundo, en donde las decisiones se toman utilizando mecanismos de razonamiento lógico basados en la correspondencia de patrones y la manipulación simbólica, con el propósito de alcanzar los objetivos del agente.

Quizás la arquitectura deliberativa más estudiada sea la arquitectura deliberativa BDI (*Belief, Desire, Intention*), caracterizada por el hecho de que los agentes que la implementan están dotados de los estados mentales de *creencias, deseos, e intenciones*. El éxito de esta arquitectura se debe a que combina elementos interesantes: un modelo filosófico del razonamiento humano fácil de comprender, un número considerable de implementaciones, y una semántica lógica abstracta y elegante, aceptada por gran parte de la comunidad científica. El modelo BDI se ha desarrollado para proporcionar soluciones en entornos dinámicos o inciertos, en los que el agente o los agentes sólo tienen una visión parcial del problema y, posiblemente, manejen un número limitado de recursos. Las creencias, los deseos, las intenciones, y los planes son una parte fundamental del estado de ese tipo de sistemas. Las creencias representan el conocimiento que se tiene del entorno. Los deseos u objetivos representan el estado

final deseado, y para alcanzarlos a partir de las creencias existentes es necesario definir un mecanismo de planificación que permita identificar las inteciones.

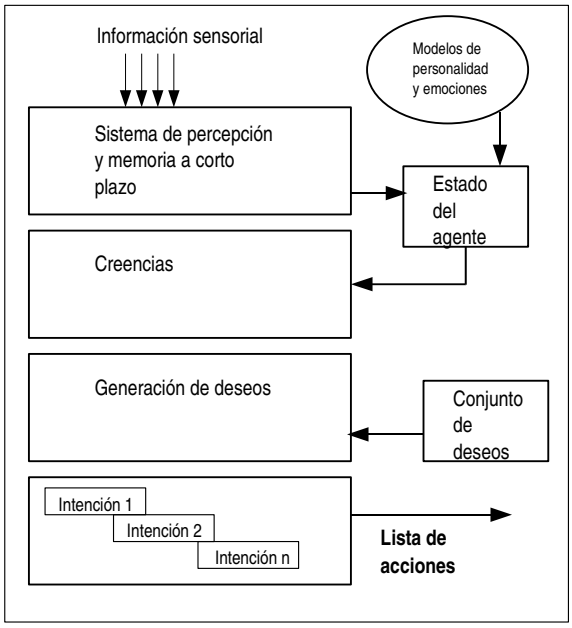

Figura 2.1: Arquitectura BDI

Como consecuencia de los numerosos problemas que lleva asociados utilizar una representación simbólica se han estudiado modelos más efectivos de representación del conocimiento. Por tanto, las **arquitecturas reactivas** se caracterizan por no tener como elemento central de razonamiento un modelo simbólico y por no utilizar razonamiento simbólico complejo.

Por último, mencionar que existen **arquitecturas híbridas** que tratan de solventar las limitaciones que ambas arquitecturas presentan por separado. Una posible propuesta es construir un agente compuesto de dos subsistemas: uno deliberativo, que emplee un modelo simbólico y que genera planes, y otro reactivo, centrado en reaccionar ante los eventos que tengan lugar en el entorno, que no requiera un mecanismo de razonamiento complejo.

2.1.1.4. La arquitectura de pizarra

Uno de los principales modelos utilizados en los sistemas cooperantes, y cuyos estudios se remotan a la década de los 80, es la **arquitectura de pizarra**. Este modelo permite la cooperación entre fuentes de conocimiento heterogéneas mediante un mecanismo sencillo de comunicación: **la pizarra**. Cada fuente de conocimiento contiene parte del conocimiento necesario para resolver un problema. Cuando en la pizarra existen datos sobre los que la fuente de conocimiento puede trabajar, los toma de la pizarra, los elabora, y deja en ella los resultados. Cada fuente de conocimiento puede trabajar de forma independiente y en paralelo con otras fuentes para producir sus soluciones. En la práctica, el principal problema de este paradigma es el control. No es suficiente con producir resultados parciales, además es necesario hacerlo en el orden y en el momento adecuados.

Utilizando una arquitectura de pizarra ha de existir un modelo de coordinación que sincronice los resultados producidos por las distintas fuentes de conocimiento. El control también puede modelarse como una fuente de conocimiento más que mira los datos de la pizarra, tiene información sobre las fuentes que pueden intervenir, y decide cuál de ellas lo debe hacer. Este tipo de modelo es un modelo centralizado.

En un entorno de procesamiento distribuido el modelo de pizarra da lugar a múltiples variantes. Se pueden descentralizar las fuentes, el control o la pizarra, agrupar las fuentes de conocimiento en función de las prestaciones necesarias para resolver el problema, tener en cuenta aspectos de tolerancia a fallos entre fuentes o redundancia de información, etc.

2.1.1.5. Arquitecturas multi-agente

El hecho de desarrollar aplicaciones complejas compuestas de distintos subsistemas que colaboran entre sí obliga a distribuir la inteligencia entre diversos agentes, y a construir **sistemas multi-agente** que permitan la gestión inteligente de un sistema complejo. Dicha gestión se logra mediante la coordinación de los distintos subsistemas que lo componen e integrando los objetivos particulares de cada subsistema en un objetivo común.

Si cada subsistema tiene una capacidad de decisión local, el problema de la gestión se puede abordar definiendo políticas de cooperación, coordinación, y negociación entre agentes.

En general, un sistema multi-agente cooperante presentará las siguientes características:

- Estará formado por un conjunto de agentes, cada uno de los cuales mantienen sus propias habilidades.
- El sistema multi-agente tiene una misión común, la cual puede descomponerse en diferentes tareas independientes, de forma que se puedan ejecutar en paralelo.
- Cada agente del sistema tiene un conocimiento limitado.
- Cada agente del sistema tiene cierta especialización para realizar determinadas tareas, en función de lo que conoce, la capacidad del proceso, y la habilidad requerida.

Por otra parte, la distribución de las decisiones entre los distintos nodos del sistema permite fundamentalmente:

- Mantener la autonomía de cada agente.
- Eliminar la necesidad de que toda la información del sistema se encuentre en un único agente.
- Descentralizar la toma de decisiones.
- Llevar a cabo acciones coordinadas.
- Organizar dinámicamente la arquitectura de agentes, lo que permite la adaptación de la arquitectura a los cambios que se produzcan en el entorno en tiempo real.

En el problema de la coordinación entre los agentes es tan importante el proceso de razonamiento interno como el proceso de comunicación. El proceso de razonamiento interno consistirá en la toma de decisiones y en la identificación de la información que se debe compartir. El proceso de comunicación debe prefijar cómo y cuándo debe producirse la comunicación entre los nodos. La comunicación, el qué y el cómo entre los distintos agentes, es muy importante a la hora de definir un sistema multi-agente.

En cuanto al análisis de una organización compuesta de agentes es importante distinguir sus elementos fundamentales. Por ello, se diferencian tres métodos distintos de análisis:

1. **Análisis funcional**, que permite describir las diferentes funciones del sistema multi-agente en sus diferentes dimensiones.

2. **Análisis estructural**, que permite distinguir entre las distintas formas de organización e identificar los parámetros estructurales fundamentales.

3. **Análisis de parámetros concretos**, centrado en las diferentes cuestiones que aparecen al pasar de una estructura abstracta a una organización concreta de agentes.

Una vez que se tiene una descripción del sistema multi-agente es posible analizar las relaciones abstractas entre los agentes que existen en la organización, los modos de enlace de los agentes y las subordinaciones que existen entre los agentes para la toma de decisiones. Las relaciones abstractas describen las formas de interacción entre las distintas clases de agentes, es decir, entre los roles. Dichas relaciones son:

- Relación de conocidos.
- Relación de comunicación.
- Relación de subordinación.
- Relación operativa.
- Relación de información.
- Relación de conflicto.
- Relación competitiva.

Las relaciones abstractas pueden ser estáticas o dinámicas. Las primeras se determinan al definir la organización y no cambian con la ejecución, mientras que las segundas cambian en función de la ejecución. Los modos de enlace definen cómo están unidos los agentes entre sí y la propia capacidad del agente para automodificarse. Dichos enlaces pueden ser fijos, variables, o evolutivos. La relación de subordinación se puede definir mediante una estructura jerárquica o una estructura igualitaria.

Además del análisis llevado a cabo desde un punto de vista funcional y estructural, queda por analizar cómo distribuir las habilidades entre los agentes. Los casos extremos son:

- Que todos los agentes tengan todas las habilidades posibles.
- Que cada agente tenga una única habilidad.

Para analizar esta distribución se pueden utilizar dos índices: el grado de especialización y el grado de redundancia. El primero evalúa la relación existente entre el número de habilidades que tiene el agente para un problema dado y el número de habilidades existente para ese problema. El segundo evalúa la relación existente entre el número de agentes que tienen la misma habilidad y el número de agentes. A partir de estos dos parámetros se definen los siguientes casos:

- Organización no redundante e hiperespecializada, en la que cada agente tiene una única habilidad y nadie más posee esa habilidad. Es una organización típica de una descomposición funcional, donde cada función es un agente.
- Organización redundante y especializada, donde cada agente sólo tiene una habilidad y todos los agentes la poseen.
- Organización redundante y general, donde cada agente tiene varias habilidades y varios agentes las poseen.
- Organización no redundante y general, en la que cada agente tiene todas las habilidades y nadie más que ese agente las posee.

2.1.2. Ingeniería del Software en sistemas multi-agente

2.1.2.1. Introducción

Uno de los principales problemas en el área de los sistemas multi-agente es que cada vez son más necesarios métodos, técnicas, y herramientas que faciliten el desarrollo de aplicacaciones basadas en el paradigma de los sistemas multi-agente. Dentro del campo de la Ingeniería del Software, se podría pensar en el concepto de agente como una nueva abstracción con el suficiente potencial como un paradigma en la Ingeniería del Software, de forma que es necesario desarrollar técnicas de Ingeniería del Software que sean aplicables de forma específica a este paradigma.

En los últimos tiempos han ido apareciendo distintas aproximaciones con el objetivo de proporcionar una metodología apropiada para el desarrollo de sistemas multi-agente. De hecho, han aparecido nuevos conceptos, como por ejemplo la **Ingeniería del Software Orientada a Agente** (AOSE - *Agent Oriented Software Engineering*) [50]. La mayoría de trabajos existentes intentan buscar métodos de desarrollo para modelar sistemas reales complejos y con características claramente distribuidas.

Según Jennings [49], las técnicas orientadas a objetos están capacitadas para desarrollar sistemas *software* complejos. Este argumento consta de tres partes:

- Las descomposiciones orientadas a agentes son una forma efectiva de particionar el espacio del problema en un sistema complejo.
- La naturaleza de la abstracción del concepto de agente implica una manera natural de modelar sistemas complejos.
- La filosofía de la orientación a agentes para identificar y gestionar relaciones de organización es apropiada para tratar con las dependencias e interacciones que existen en un sistema complejo.

2.1.2.2. Comparativa con la orientación a objetos y la orientación a componentes

El enfoque de la orientación a agentes se puede comparar al análisis y al diseño de la orientación a objetos, e incluso al de la orientación a componentes. De hecho, existen ciertas similitudes entre la orientación a objetos y la orientación a agentes en lo que a desarrollo de sistemas se refiere, como por el ejemplo que ambos enfoques enfatizan la importancia de las relaciones entre entidades. Sin embargo, también existen ciertas diferencias. En primer lugar, los objetos son normalmente entidades pasivas por naturaleza, es decir, es necesario que se les envíe un mensaje previamente a que desarrollen una acción. En segundo lugar y, aunque los objetos encapsulan un estado y un comportamiento, no encapsulan la activación de comportamiento (elección de acciones). De este modo, cualquier objeto puede invocar cualquier método de acceso público de cualquier otro objeto. Una vez que el método ha sido invocado, se desarrollan las acciones correspondientes. En este sentido, los objetos son *obedientes* los unos con los otros.

Mientras este último enfoque de la orientación a objetos puede ser suficiente para aplicaciones pequeñas en entornos bien controlados y cooperativos, no es suficiente para entornos grandes y competitivos, ya que su *modus operandi* localiza toda la responsabilidad de invocación de comportamiento en la parte del cliente. El servidor queda al margen del asunto. Parece mejor un enfoque en el que el ejecutor de la acción tenga algo que decir en el asunto. Así, por ejemplo, el ejecutor, quien por definición está más concienciado con los detalles de las acciones a desarrollar, puede tener el conocimiento necesario para decidir si la situación actual permite llevar a cabo una invocación o no.

Otra diferencia entre los enfoques de la orientación a objetos y de la orientación a agentes, y en tercer lugar, es que el primero de ellos falla en el sentido de que no proporciona un conjunto adecuado de conceptos y mecanismos para modelar los distintos tipos de un sistema complejo. Este hecho se justifica debido a que las clases y los módulos no proporcionan un nivel de abstracción acorde con lo necesario para modelar las distintas entidades que nos encontramos en un sistema complejo.

Una tecnología cercana a la orientación a objetos es la orientación a componentes, o el *software* basado en componentes. Dicha orientación hereda la mayoría de las características de la orientación a objetos, pero añadiendo el principal objetivo de la reusabilidad del componente. Por lo tanto, y como los objetos, los componentes no son autónomos en el sentido contrario en el que se entienden los agentes. Además, como ocurre con los objetos, no mantienen ninguna noción de re-actividad, pro-actividad, o comportamiento social.

2.1.2.3. El ciclo de vida del *software* orientado a agentes

Una vez mostrado que las técnicas orientadas a agentes representan un enfoque ingenieril para abordar sistemas complejos, se puede pasar a describir la Ingeniería del Software Orientada a Agentes. En particular, se estudiará como se realiza la especificación de un sistema de agentes, se discutirá cómo se implementa tal especificación, y cómo se verifica que realmente esa implementación es acorde con la especificación previa.

El enfoque predominante para especificar agentes implica tratarlos como sistemas intencionales que pueden entenderse caracterizándolos con estados mentales, como por ejemplo creencias, deseos, e intenciones. Siguiendo esta idea, se han desarrollado una gran variedad de

enfoques capaces de representar los siguientes aspectos de los sistemas basados en agentes:

- Las creencias que tiene un agente.
- Los objetivos que trata de conseguir.
- Las acciones que un agente desarrolla y los efectos que éstas tienen.
- Las interacciones entre agentes.

Una vez creada la especificación, se ha de implementar un sistema acorde con la misma. Visto de otro modo, se trata de dar el paso de las especificaciones abstractas a sistemas computacionales concretos. Existen al menos dos posibilidades para lograr este objetivo:

- Ejecutar de alguna manera la especificación abstracta.
- Trasladar o compilar la especificación en una forma computacional concreta empleando una técnica de traducción automática.

El último paso consiste en la verificación, es decir, se necesita comprobar que el sistema es correcto con respecto a la especificación inicial. Los enfoques de verificación pueden dividirse en dos grandes ramas: axiomáticos y semánticos.

2.1.2.4. Metodologías de desarrollo

En la sección actual se realizará un breve recorrido por las metodologías de desarrollo de sistemas multi-agente, comentando las principales características de cada una de ellas y el enfoque adoptado. Aunque en el presento proyecto no se ha seguido ninguna de ellas, sí que se han tomado como buenas prácticas de ingeniería y se han utilizado como guías en distintos momentos de la elaboración del proyecto.

Una primera aproximación es el **modelado y diseño de sistemas multiagente en BDI**. Esta aproximación trata de explorar cómo las técnicas de modelado de orientación a objetos se pueden extender para aplicarse a sistemas de agentes basados en la arquitectura BDI. Por ello, se proponen técnicas de modelado para describir las distintas perspectivas internas y externas de sistemas multi-agente basados en la arquitectura BDI.

Otro trabajo es el llamado **método de Burmeister** [34], en el que se especifican tres modelos para el análisis orientado a agentes de un sistema y se dan ciertas guías para la construcción del sistema. Este método está basado en técnicas orientadas a objetos y, aunque está poco detallada sobre todo en su fase de diseño, introduce elementos clave a detectar y modelar en el proceso de desarrollo de un sistema multi-agente: organización, agentes, e interacciones.

Otra metodología es la denominada **MAS-CommonKADS**, basada en la metodología CommonKADS. Dicha metodología aporta una serie de modelos para desarrollar las fases de análisis y diseño de sistemas multi-agente. Su principal característica es la incorporación de técnicas orientadas a objetos, la cual es tomada como eje fundamental a lo largo de todo el proceso. Además, se desarrolla a través de la construcción de siete modelos: *el modelo de agente*, que describe las características de cada agente; *el modelo de tarea*, que describe las tareas realizadas por los agentes; *el modelo de la experiencia*, que describe el conocimiento que necesitan los agentes para llevar a cabo los distintos objetivos; *el modelo de coordinación*, que describe las relaciones dinámicas entre los agentes *software*; *el modelo de comunicación*, que describe las relaciones dinámicas entre los agentes humanos y los agentes *software*; *el modelo de organización*, que describe la organización humana en la que el sistema multi-agente se introduce y la estructura de la organización de agentes *software*; y *el modelo de diseño*, que refina los modelos anteriores y decide qué arquitectura de agentes es más adecuada para cada agente.

La metodología **GAIA** [49] se centra en la idea de que la construcción de sistemas basados en agentes es un proceso de diseño organizacional. Se trata de una metodología muy genérica, y que no indica posibles mecanismos para llegar a un sistema directamente ejecutable. Está basada en la idea de tratar el sistema multi-agente como una organización computacional consistente en varios roles interactivos. Al aplicar esta metodología, el analista pasa de conceptos abstractos a conceptos más concreto de forma incremental. GAIA importa terminología y notación de la orientación a objetos, pero no es un intento de adaptar la tecnologías de objetos a la tecnología de agentes. Como contrapartida, proporciona un conjunto de conceptos específicos a la tecnología de agentes a través de los cuales un ingeniero del *software* pueda entender los modelos y el sistema en su conjunto. De hecho, GAIA trata de hacer entender la

construcción de un sistema basado en agentes como un proceso de diseño organizacional.

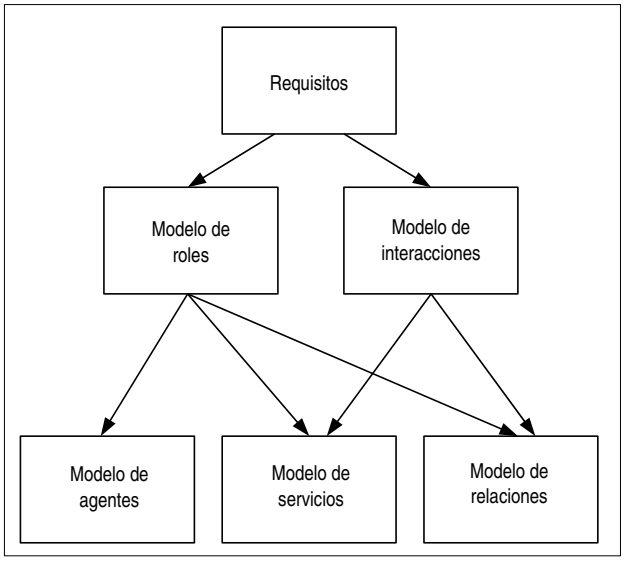

Figura 2.2: Relaciones entre los modelos de GAIA

Otra metodología es el método de desarrollo de sistemas multi-agente **MASSIVE** (*Multi-Agent SystemS Iterative View Enginering*) [46]. Dicho método está constituido por un conjunto de vistas diferentes del sistema a construir donde el desarrollo que se sigue consiste en una visión iterativa del mismo. En él se combinan procesos de reingeniería con un método en cascada mejorado con el objetivo de llevar a cabo distintos refinamientos.

Una iniciativa muy interesante y que cobra especial importancia es la iniciativa **AUML** [41]. Se trata de una visión de los agentes como siguiente paso a la tecnología de objetos, de forma que se exploran extensiones a UML para acomodar los requisitos distintivos de los agentes. Una de las principales áreas de extensión, y la que en este documento se comenta, es la representación de los protocolos de los agentes.

UML es insuficiente para modelar agentes y sistemas basados en agentes, ya que comparados con los objetos, los agentes son entidades activas. De este modo, se plantea el uso de AUML con el objetivo de capturar patrones en lo que a la organización de agentes se refiere. Por ejemplo, se podría expresar un protocolo de interacción mediante un diagrama de secuencia en UML, en el que un agente asume la iniciativa y otro agente responde a las solicitudes del primero de ellos, de forma que el agente que responde puede decidir en un

momento dado si aceptar o no la proposición. Como se puede apreciar en la figura adjunta, el protocolo de interacción es tratado como una entidad y puede utilizarse como un patrón que puede personalizarse en dominios de problemas similares.

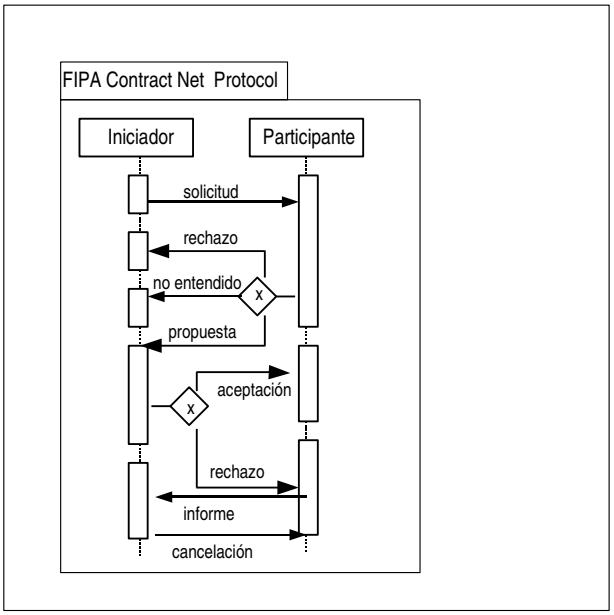

Figura 2.3: Protocolo de interacción entre agentes expresado como una plantilla

La conclusión que se puede extraer de la anterior idea es que los patrones son ideas que son útiles en un determinado contexto y que, probablemente, puedan ser útiles en contextos similares. Los patrones nos ofrecen ejemplos que podemos usar como soluciones a problemas relacionados con el análisis y el diseño de sistemas. Por lo tanto, los protocolos de interacción entre agentes nos proporcionan soluciones reutilizables que pueden aplicarse a varios tipos de secuencias de mensajes que encontramos entre agentes. Para ello, se pueden emplear las dos técnicas que mejor expresan esta reutilización en UML: paquetes y plantillas.

Además de lo comentado anteriormente, también se puede hacer uso de los modelos dinámicos de UML para expresar las interacciones entre los agentes, como por ejemplo los diagramas de interacción, los diagramas de actividad, y las máquinas de estados.

Otra metodología a tener en cuenta es **MaSE** (*Multiagent System Engineering*) [59], desarrollada en el Air Force Institute of Technology. Dicha metodología trata de cubrir todas las

etapas en el proceso de construcción de un sistema multi-agente, partiendo de la especificación del mismo hasta su implementación. Además, MaSE dispone de un lenguaje de especificación basado en UML+OCL y una herramienta de desarrollo denominada *AgentTool*, la cual trata de cubrir la totalidad de fases de la metodología.

MESSAGE (*Methodology for Engineering Systems of Software Agents*) [37] es una metodología orientada a agentes que incorpora técnicas de Ingeniería del Software cubriendo el análisis y diseño de sistemas multi-agente. La metodología proporciona un lenguaje, un método, y unas guías de cómo aplicar la metodología, centrándose en las fases de análisis y diseño y ofreciendo ideas sobre el resto de etapas, como por ejemplo la etapa de implementación.

Por último, en esta sección se comentará la metodología **INGENIAS** [10]. Dicha metodología ha sido desarrollada a partir de la metodología MESSAGE, a la que mejora en tres aspectos:

- Integración de las vistas de diseño del sistema.
- Integración de resultados de investigación.
- Integración con el ciclo de vida de desarrollo de *software*.

Esta metodología propone un lenguaje de especificación de sistemas multi-agente, así como su integración en el ciclo de vida. Al igual que en MESSAGE, el lenguaje se especifica con meta-modelos y lenguaje natural. Un meta-modelo es una representación de los tipos de entidades que pueden existir en un modelo, sus relaciones, y sus restricciones de aplicación. Los meta-modelos se presenta de forma separada:

- Meta-modelo de agente, que describe agentes particulares y los estados mentales en que se encontrarán a lo largo de su vida.
- Meta-modelo de tareas y objetivos, que se usa para asociar el estado mental del agente con las tareas que ejecuta.
- Meta-modelo de organización, que define cómo se agrupan los agentes, la funcionalidad del sistema, y qué restricciones hay que imponer sobre el comportamiento de los agentes.

- Meta-modelo de interacción, que detalla cómo se coordinan y comunican los agentes.
- Meta-modelo de entorno, que especifica qué existe alrededor del nuevo sistema y cómo lo percibe cada agente.

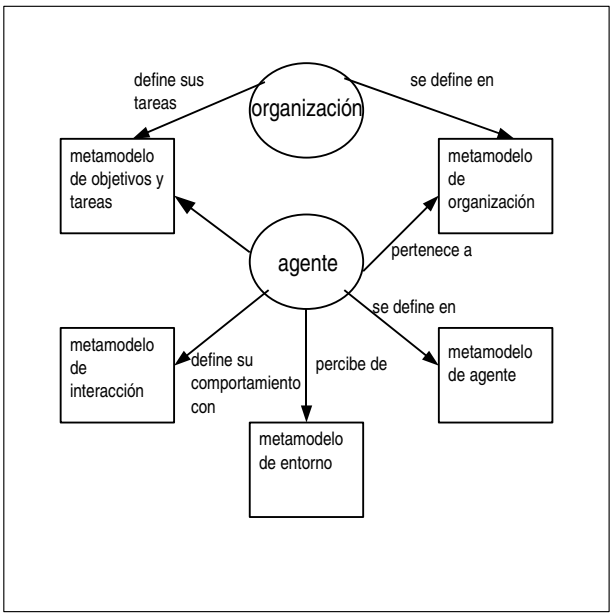

Figura 2.4: Meta-modelos y entidades en INGENIAS.

2.1.3. Estándares para sistemas multi-agente

2.1.3.1. FIPA

FIPA (**The Foundation for Intelligent Physical Agents**) [18] es un comité del IEEE para la promoción de la tecnología basada en agentes y en la interoperabilidad de sus estándares con otras tecnologías. Las especificaciones FIPA representan un conjunto de estándares que persiguen el objetivo de promover la interacción entre agentes heterogéneos y los servicios que representan los mismos.

FIPA mantiene un ciclo de vida para sus distintas especificaciones, compuesto por las etapas *Preliminary*, *Experimental*, *Standard*, *Deprecated*, y *Obsolote*. A cada especificación

se le asigna un identificador único, en función del estado en el que se encuentran dentro del ciclo de vida.

La elección de FIPA como estándar para el desarrollo del sistema multi-agente subyacente a MASYRO se basa en que es la especificación más seria en lo que a desarrollo de sistemas multi-agente se refiere, junto con la alta escalabilidad que ofrece.

A continuación se irán comentando alguno de los documentos más importantes, que definen las partes principales de la arquitectura propuesta por FIPA.

Uno de los principales documentos es el **FIPA *Absctract Architecture Specification*** [2]. Dicho documento, y las especificaciones derivadas del mismo, definen la arquitectura abstracta propuesta por FIPA. Las partes de dicha arquitectura incluyen:

- Una especificación que define elementos arquitectónicos y sus relaciones.
- Guías para la especificación de sistemas de agentes en lo que se refiere a *software* en concreto y a tecnologías de comunicación (guías para la instanciación).
- Especificaciones que gobiernan la interoperabilidad y conformidad de los agentes y los sistemas multi-agente (guías para la interoperabilidad).

El propósito principal de este documento es garantizar la interoperabilidad y la reusabilidad. Para lograr este propósito es necesario identificar los elementos de la arquitectura que deben codificarse. En concreto, si dos o más sistemas usan distintas tecnologías para alcanzar algún propósito funcional es necesario identificar las características comunes de los distintos enfoques. Este hecho lleva a la identificación de abstracciones arquitectónicas, es decir, a diseños abstractos que pueden ser formalmente trasladados a cualquier implementación válida. Describiendo los sistemas de forma abstracta, se pueden explorar las relaciones entre los elementos principales de los sistemas multi-agente. Además, describiendo las relaciones entre estos elementos se llega a obtener de forma clara cómo los sistemas multi-agente pueden crearse y cómo son capaces de interoperar. De este conjunto de elementos arquitectónicos y sus relaciones se puede obtener un amplio conjunto de posibles arquitecturas concretas, que pueden interoperar porque comparten un diseño abstracto.

De una forma más específica, el objetivo de este documento es definir el intercambio de mensajes entre agentes, los cuales pueden utilizar distintos protocolos de transporte de

mensaje, distintos lenguajes de comunicación de agentes, y diferentes lenguajes de contenido. El proposito de esta arquitectura incluye:

- Un modelo de servicios y de descubrimiento de servicios disponibles a los agentes y a otros servicios.
- Interoperabilidad en el transporte de mensajes.
- Soporte para varias formas de representación de lenguajes de comunicación de agentes.
- Soporte para varias formas de lenguajes de contenido.
- Soporte para múltiples representaciones de servicios de directorio.

Debido a que una arquitectura abstracta no puede implementarse directamente, existe un proceso de conversión de una arquitectura abstracta a implementaciones concretas.

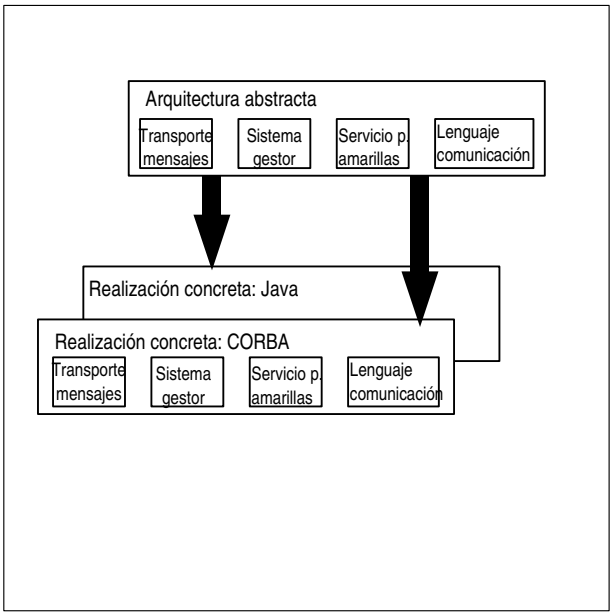

Figura 2.5: Realización concreta de la arquitectura abstracta

Según la arquitectura abstracta de FIPA, los **agentes** se comunican intercambiando mensajes que representan actos comunicativos, los cuales son codificados en un lenguaje de comunicación de agentes. Los **servicios** proporcionan servicios de soporte para los agentes, siendo

necesario un mecanismo por el que los agentes obtengan un conjunto de servicios básicos cuando los primeros son creados.

Otro de los documentos que constituye el pilar principal de FIPA es el documento **FIPA *Agent Management Specification*** [4]. Dicho documento contiene especificaciones para la gestión de agentes, incluyendo servicios de gestión de agentes, ontologías, y transporte de mensajes dentro de la plataforma de agentes. Además, este documento está especialmente vinculado a la definición de interfaces estándares abiertas para el acceso a los servicios de gestión de agentes. El diseño interno, la implementación de los agentes, y la infraestructura de la gestión de agentes quedan al margen de este documento.

El modelo de referencia para la gestión de agentes contiene entidades que son conjuntos de capacidades lógicas (es decir, servicios), y que no implican configuraciones físicas. Además, los detalles de implementación de los agentes y las plataformas de agentes son decisiones particulares del desarrollador. Dicho modelo de referencia consiste distintos componentes lógicos, cada uno de los cuales representa un conjunto de capacidades.

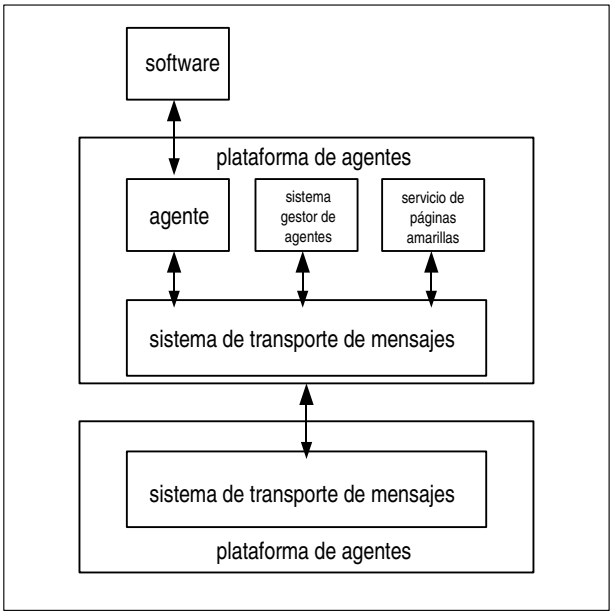

Figura 2.6: Modelo de referencia para la gestión de agentes

Un **agente** (*agent*) es un proceso computacional que implementa la funcionalidad de

comunicación y la autonomía de una aplicación. Los agentes se comunican empleando un **lenguaje de comunicación de agentes** (*Agent Communication Language*). Además, un agente se puede concebir como el actor fundamental en la plataforma de agentes, y que combina una o más capacidades (servicios) dentro de un modelo de ejecución integrado. La noción de identidad para un agente es el **identificador de agente** (*Agent Identifier* o *AID*), que identifica a un agente de manera unívoca dentro de la plataforma de agentes.

El ***Directory Facilitator*** es un componente opcional de la plataforma de agentes, pero que si está presente debe implementarse como un tipo específico de servicio. Dicho componente es el encargado de proporcionar un servicio de páginas amarillas a los agentes, y puede existir más de uno dentro de una plataforma de agentes. Si este componente está presente en la plataforma de agentes tiene un AID reservado:

```
(agent-identifier
  :name df@hap_name
  :addresses (sequence hap_transport_address))
```

Cada *Directory Facilitator* ha de ser capaz de proporcionar las siguientes funciones:

- register
- deregister
- modify
- search

El protocolo de interación *fipa-request* [5] debe usarse por los agentes que deseen llevar a cabo alguna de estas acciones. Además, el *Directory Facilitator* puede proporcionar el siguiente mecanismo adicional:

- subscribe mechanism

Para el mecanismo de subscripción el *Directory Facilitator* debe implementar el protocolo de interación *fipa-subscribe* [6] con el objetivo de notificar a los agentes subscritos eventos como el registro, la eliminación de la subscripción, o la modificación de ciertas descripciones de los agentes.

El ***Agent Management System*** es un componente obligatorio de la plataforma de agentes, y que ejerce como controlador de dicha plataforma. Además, sólo se permite un AMS por plataforma de agentes. Este servicio mantiene un directorio con los AIDs de los agentes registrados en la plataforma. El AMS proporciona un servicio de páginas blancas a los agentes, y es necesario que cada agente se registre en el AMS para obtener un AID válido. Dicho componente representa la autoridad de gestión en la plataforma de agentes, y es responsable de todas las acciones de gestión relacionadas con los agentes. El AMS tiene un AID reservado:

```
(agent-identifier
  :name ams@hap_name
  :addresses (sequence hap_transport_address))
```

El AMS debe ser capaz de ofrecer las siguientes funciones de gestión:

- register
- deregister
- modify
- search
- get-description

Además, el AMS tiene la capacidad para llevar a cabo las siguientes acciones:

- Suspend agent
- Terminate agent
- Create agent
- Resume agent execution
- Invoke agent
- Execute agent
- Resource management

El ***Message Transport Service*** [7] representa el método de comunicación por defecto entre los agentes de distintas plataformas de agentes. El modelo de referencia para el transporte de mensajes entre agentes está asociado con tres niveles:

- El *Message Transport Protocol* (MTP) es usado para llevar a cabo el transporte físico de mensajes entre dos ACCs (entidades que proporcionan el servicio directo de transporte a los agentes).
- El *Message Transport Service* (MTS) es un servicio proporcionado por la plataforma de agentes a la que un agente está vinculado. El MTS soporta el transporte de mensajes FIPA ACL entre agentes de una misma o distintas plataformas de agentes.
- El ACL representa la carga útil de los mensajes intercambiados entre el MTS y el MTP.

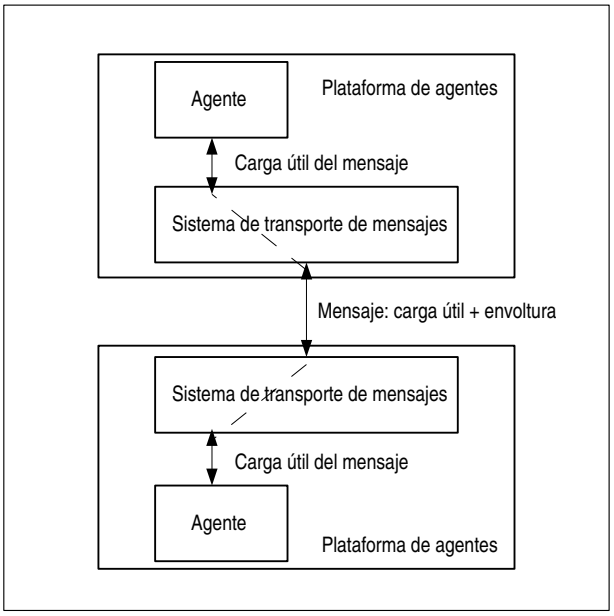

Figura 2.7: Modelo de referencia para el transporte de mensajes

En cuanto a lo que a la estructura de un mensaje se refiere, un mensaje está compuesto de dos partes. Por un lado, la envoltura del mensaje expresa la información de transporte y mantiene un conjunto de definiciones:

- Contiene una colección de parámetros.
- Un parámetro es una pareja clave-valor.
- Contiene los parámetros obligatorios *to*, *from*, *date*, y *acl-representation*.
- Puede contener parámetros opcionales.

Por otro lado, la carga útil del mensaje representa el mensaje ACL en lo relativo a la comunicación entre agentes.

Un agente tiene diversas opciones a la hora de enviar mensajes a agentes de otras plataformas:

1. El agente A envía un mensaje a su ACC local usando una interfaz estándar propietaria. Éste se encarga de enviar el mensaje al ACC remoto a través del MTP. Por último, el ACC remoto entregará el mensaje a su destinatario.
2. El agente A envía un mensaje directamente al ACC remoto, el cual entregará el mensaje al agente B.
3. El agente A envía directamente el mensaje al agente B.

El modelo de referencia de nombrado de agentes identifica un agente a través de una colección extensible de parejas parámetro-valor, denominada *Agent Identifier* (AID). La extensible naturaleza del AID permite aumentar de una forma cómoda el número de parámetros de dicho elemento. Por defecto, el AID comprime los siguientes parámetros:

- Un parámetro *name*, que es el identificador único y global que se emplea como expresión para referirse al agente.
- Un parámetro *addresses*, consistente en una lista de direcciones de transporte donde se pueden entregar los mensajes.
- Un parámetro *resolvers*, que constituye una lista de direcciones de servicios de resolución de nombres.

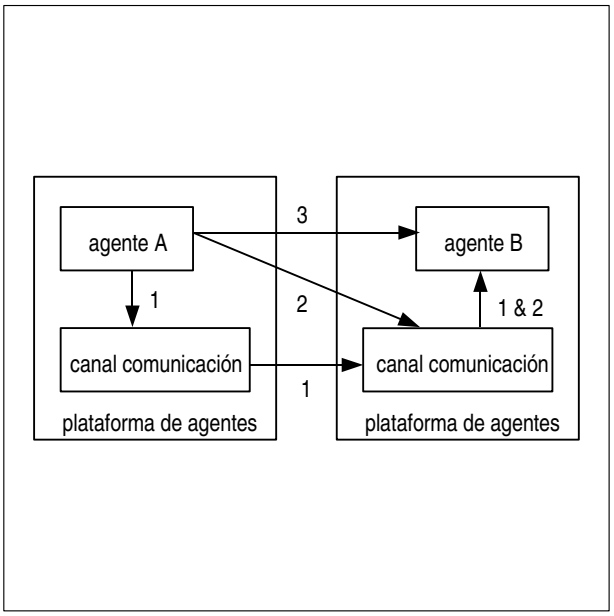

Figura 2.8: Métodos de comunicación entre agentes

Se considera que dos AIDs son equivalentes si lo son los valores de sus parámetros *name*.

En cuanto al ciclo de vida de un agente, se debe tener en cuenta que un agente existe físicamente en la plataforma de agentes y utiliza la funcionalidad que ofrecen los distintos servicios de la plataforma. Dentro de este contexto, un agente, como proceso *software*, mantiene un ciclo de vida físico que ha de ser gestionado por la plataforma de agentes. Los estados en los que un agente puede encontrarse son los siguientes:

- Active
- Initiated
- Suspended
- Waiting
- Transit

Otra cuestión importante la representa el conjunto de marcos que representa las clases de objetos en el dominio del discurso asociado a la ontología de la gestión de agentes en FIPA.

Dicha ontología no especifica un orden en la codificación de los parámetros de los objetos, pero si que es necesario codificarlos en la forma parámetro-valor.

El identificador del agente está representado por el marco del objeto asociado al *Agent-Identifier Description*.

Marco (ontología)	agent-identifier (fipa-agent-management)			
Parámetro	**Descripción**	**Presencia**	**Tipo**	**Valores reservados**
name	Nombre simbólico del agente.	Obligatorio	word	dfhap_name amshap_name
addresses	Secuencia de direcciones de transporte en las que el agente puede ser contactado.	Opcional	Secuencia de url	
resolvers	Secuencia de AIDs donde los servicios de resolución de nombres para el agente pueden ser contactados.	Opcional.	Secuencia de *agent-identifier*	

Cuadro 2.1: *Agent Identifier Description*.

La descripción del agente que se registra en el *Directory Facilitator* está representada por el marco del objeto asociado al *Directory Facilitator Agent Description*.

Marco (ontología)	df-agent-description (fipa-agent-management)			
Parámetro	**Descripción**	**Presencia**	**Tipo**	**Valores reservados**
name	Identificador del agente.	Opcional	agent-identifier	
services	Lista de servicios soportados por el agente.	Opcional	Conjunto de *service-description*	
protocols	Lista de protocolos de interacción soportados por el agente.	Opcional.	Conjunto de string	
ontologies	Lista de ontologías conocidas por el agente.	Opcional.	Conjunto de string	*fipa-agent-management*
languages	Lista de lenguajes de contenido conocidos por el agente.	Opcional.	Conjunto de string	fipa-sl fipa-sl0 fipa-sl1 fipa-sl2
lease-time	Duración del registro.	Opcional.	datetime	
scope	Visibilidad de la descripción.	Opcional.	Conjunto de string	global local

Cuadro 2.2: *Directory Facilitator Agent Description*.

La descripción del agente que se registra en el *Agent Management System* está represen-

tada por el marco del objeto asociado al *Agent Management System Agent Description*.

Marco (ontología)	ams-agent-description (fipa-agent-management)			
Parámetro	**Descripción**	**Presencia**	**Tipo**	**Valores reservados**
name	Identificador del agente.	Opcional	agent-identifier	
ownership	Propietario del agente.	Opcional	string	
state	Estado del ciclo de vida del agente.	Opcional.	string	

Cuadro 2.3: *Agent Management System Agent Description*.

Todo este marco conceptual estaría incompleto sin un lenguaje de contenidos. Para ello, FIPA define el *Semantic Language (SL) Content Language*, el cual define una sintaxis específica. Sin embargo, dicha sintaxis y su semántica son sugeridas como candidatas para usarse junto con el lenguaje de comunicación de agentes, lo cual permite al desarrollador utilizar otro tipo de lenguaje de contenidos si así lo desea.

El lenguaje está especificado en sintaxis EBNF, y su gramática asociada tiene como punto de entrada el elemento *Content Expression*:

```
ContentExpresion   = IdentifyingExpression
                   | ActionExpression
                   | Proposition
```

La proposición define un valor que puede darse en un determinado contexto, la acción se refiere a un evento a desarrollar, y la expresión de identificación identifica a un objeto dentro de un contexto.

2.1.4. SDKs existentes para sistemas multi-agente

2.1.4.1. JADE

JADE (***Java Agent DEvelopment Framework***) [11] es un *framework* completamente implementado en el lenguaje Java, que simplifica la implementación de sistemas multi-agente a través de un *middleware* acorde con las especificaciones del conjunto de estándares FIPA y de un conjunto de herramientas gráficas que soportan las fases de desarrollo y depuración. JADE está disponible bajo una licencia LGPL (*Lesser General Public License*) versión 2 y detrás de su desarrollo están varias empresas.

La plataforma de agentes puede distribuirse entre distintas máquinas (no necesariamente con el mismo sistema operativo) y la configuración puede llevarse a cabo de forma remota. Dicha configuración puede realizarse a cabo en tiempo de ejecución, permitiendo la movilidad de agentes entre distintas máquinas.

Cada instancia de ejecución del entorno JADE se denomina contenedor, y puede contener distintos agentes. El conjunto de contenedores activos se denomina plataforma. Los agentes están identificados por un nombre, y pueden comunicarse de forma transparente a su localización. JADE sigue el conjunto de estándares definidos por FIPA, por lo que maneja los servicios básicos como el *Agent Management System*, el *Directory Facilitator*, y el *Agent Communication Channel*.

Crear un agente que desarrolle una específica tarea dentro de la plataforma de agentes pasa por definir una clase que extiende a la clase *jade.core.Agent*, e implementar el método *setup()*:

```
import jade.core.Agent;

public class MyAgent extends Agent {
  protected void setup () {
    System.out.println('Hello World!');
  }
}
```

El trabajo actual de un agente se lleva a cabo a través de sus *comportamientos*. Un *comportamiento* representa una tarea que un agente puede llevar a cabo, y es implementada como un objeto de una clase que extiende de la clase *jade.core.behaviours.Behaviour*. Para conseguir que un agente ejecute una tarea implementada por un objeto del tipo *comportamiento* es suficiente añadir un comportamiento empleando el método *addBehaviour()*. Además, cada clase que herada de la clase *Behaviour* ha de implementar el método *action()*, que define las operaciones a desarrollar cuando el comportamiento está en ejecución.

Una de las características más importantes de los agentes en JADE es la habilidad de comunicación. Para ello se adoptó un paradigma de comunicación basado en el intercambio de mensajes asíncrono. Cada agente tiene un tipo de buzón de correo (cola de mensajes del agente) en el que el núcleo de ejecución de JADE almacena los mensajes enviados por otros agentes. Cada vez que un mensaje entra en la cola del mensaje el agente recibe una notificación.

Otro elemento importante de JADE es uno de los principales servicios definidos por FIPA, el *Directory Facilitator*. Dicho servicio proporciona un servicio de páginas amarillas para que los agentes publiquen los servicios que ofrecen, de forma que otros agentes puedan solicitar dichos servicios. El servicio de páginas amarillas está implementado en JADE a través de un agente, y pueden existir distintos agentes de este tipo dentro de una plataforma de agentes.

2.1.5. Representación del conocimiento

2.1.5.1. Introducción a los sistemas expertos

Los **sistemas expertos** son programas de ordenador, diseñados con el objetivo de proporcionar ciertas habilidades de un experto a un no experto. Debido a que dichos programas tratan de emular el comportamiento de un experto, es natural que el primer trabajo de estas características fuese desarrollado en los círculos de la Inteligencia Artificial. Entre los primeros sistemas expertos destacan los programas *Dendral*, en 1965, que determinaron la estructura molecular a partir de la información de un espectrómetro de masas, empleados para configurar sistemas de ordenador y para el diagnóstico médico.

Existen numerosas formas para crear programas de ordenador que simulen el comportamiento de un experto. Las primeras maneras utilizaban **sistemas basados en reglas**, que empleaban reglas *si-entonces* para representar el proceso de razonamiento de un experto (*si* los datos reunen ciertas condiciones específicas, *entonces* se deben tomar las acciones apropiadas). Por otra parte, existe un consenso entre gran parte de la comunidad científica en lo relativo a que una parte significativa del razonamiento humano puede expresarse mediante reglas, lo cual le aporta un gran interés a los sistemas basados en reglas. Otras aproximaciones incluyen redes semánticas o asociativas y redes neuronales, actualmente muy populares en una gran variedad de campos.

Sea cual sea el tipo de sistema experto empleado, se debe considerar cuales son los prerrequisitos para construir un sistema experto con éxito. Las dos fuentes principales de conocimiento son las habilidades de un experto en la materia, y los datos históricos disponibles. Los sistemas expertos basados en reglas dependen considerablemente de incorporar las habilidades de un experto en el dominio del problema, mientras que dependen en mucha menor

medida de los datos históricos; mientras que en las redes neuronales ocurre exactamente lo contrario.

Hay que establecer una distinción entre dos tipos de sistemas expertos; sistemas procedimentales, escritos en lenguajes procedimentales convencionales como C++, y sistemas de producción, que emplean reglas del tipo *si-entonces*. La parte *si* de la regla se denomina antecedente, mientras que la parte *entonces* es el consecuente. Un ejemplo de regla es el siguiente:

```
Si Complejidad es 10 y Tamaño es 15 Entonces Nivel de recursión es 5
```

En la regla anterior, los antecedentes serían las variables *Complejidad* y *Tamaño*, mientras que el consecuente sería la variable *Nivel de recursión*. Un sistema de producción puede contener docenas, cientos, e incluso miles de reglas.

Un tipo específico de sistemas expertos son los **sistemas expertos difusos**, que se categorizan en dos tipos principales: los sistemas de control difuso y los sistemas de razonamiento difuso. Aunque ambos hacen uso de los conjuntos difusos, se diferencian cuantitativamente en su metodología. En las siguientes secciones se profundizará más sobre estos conceptos.

El control difuso de procesos fue éxitosamente introducido por Mamdani (1976) con un sistema difuso para controlar una cementera. Desde entonces, el control difuso ha sido ampliamente aceptado, primero en Japón y luego en el resto del mundo. Un sencillo sistema difuso de control acepta números como entrada, convierte estos números a variables lingüísticas como *lento*, *normal*, o *rápido* (proceso de *borrosificación* o *fuzzyfication*). A continuación, las reglas establecen las variables lingüísticas de salida en función de las de entrada y, finalmente, los términos lingüísticos de salida son convertidos a números, en un proceso inverso a la *borrosificación* que se denomina *desborrosificación* o *defuzzyfication*. Una regla típica de control difusa podría ser la siguiente:

```
IF input1 is High AND input2 is Low THEN output is Zero.
```

Los sistemas basados en reglas requieren que el conocimiento experto y los patrones de pensamiento estén especificados explícitamente. Normalmente, dos personas (o grupos) desarrollan un sistema juntos. Una de ellos es el experto en el dominio, que sabe cómo resolver el problema pero que raramente está vinculado a la programación. La otra mitad es el ingeniero del conocimiento, que está familiarizado con la tecnología de los computadores y los

sistemas expertos, pero que normalmente no tiene conocimiento del dominio del problema o mantiene un conocimiento escaso del mismo. Obtener este conocimiento y escribir las reglas apropiadas se denomina *fase de adquisición de conocimiento*. Después de escribir el sistema, éste debe afinarse de una forma similar al entrenamiento en una red neuronal. Después de este paso, el sistema debe validarse.

Los sistemas basados en reglas tienen la ventaja de que no es necesario un largo *entrenamiento*, pero la desventaja de que la fase de adquisición de conocimiento puede resultar complicada. Además, una gran ventaja de los sistemas expertos difusos es que las reglas pueden escribirse a partir del conocimiento del experto del dominio. Además son altamente escalables y permiten modificaciones de una manera rápida y sencilla.

2.1.5.2. Conjuntos difusos

Un **conjunto difuso** o borroso se puede entender como una clase de objetos con un grado de pertenencia continuo. Dicho conjunto se caracteriza por una función de pertenencia (función característica) que asigna a cada objeto un grado de pertenencia evaluable entre cero y uno. Las nociones de inclusión, unión, intersección, complemento, relación, etc. se extienden a estos conjuntos, al mismo tiempo que establecen propiedades de estas nociones en el contexto de los conjuntos borrosos. A continuación se expondrán algunas definiciones básicas que ayudarán a entender la noción de conjunto difuso.

Sea X un espacio de puntos (objetos), con un elemento genérico denotado por x. Así, $X = \{x\}$.

Un *conjunto (clase) borroso o difuso* A en X se caracteriza por una función de pertenencia $f_A(x)$ que asocia a cada punto de X un número real del intervalo $[0, 1]$ [1], donde el valor $f_A(x)$ en x representa el *grado de pertenencia de* x *a* A. De este modo, cuanto más se aproxima el valor de $f_A(x)$ a la unidad, mayor es el grado de pertenencia de x a A.

Como ejemplo, se puede suponer X la recta real $\Re$ y A un conjunto borroso de números mucho mayores que 1. Se puede dar una caracterización precisa, aunque subjetiva, de A especificando $f_A(x)$ como una función en $\Re$. Algunos valores podrían ser $f_A(0) = 0$, $f_A(1) =$

[1] En un escenario más general, el rango de la función de pertenencia puede ser un conjunto parcialmente ordenado.

0, $f_A(5) = 0{,}01$, $f_A(100) = 0{,}95$, y $f_A(500) = 1$.

Es importante resaltar que, aunque la función de pertenencia de un conjunto borroso tiene cierto parecido con una función de probabilidad cuando X es un conjunto numerable, hay diferencias sustanciales entre ambos conceptos que se aclararán a la hora de establecer las reglas de combinación de las funciones de pertenencia y sus propiedades básicas. De hecho, la noción de conjunto borroso es de naturaleza no estadística.

Un conjunto borroso es *vacío* si y sólo si su función de pertenencia es igual a cero en X.

Dos conjuntos borrosos A y B son *iguales*, escrito como $A = B$, si y sólo si $f_A(x) = f_B(x)$ para todo x de X.

El *complemento* de un conjunto borroso A se representa por A' y se define como

$$f_{A'} = 1 - f_A \tag{2.1}$$

La propiedad de *inclusión* dice que A está contenido en B, o A es un subconjunto de B si y sólo si $f_A \leq f_B$. En símbolos

$$A \subset B \Leftrightarrow f_A \leq f_B \tag{2.2}$$

La *unión* de dos conjuntos borrosos A y B con funciones de pertenencia f_A y f_B, respectivamente, es un conjunto borroso C, escrito como $C = A \cup B$, cuya función de pertenencia se relaciona con las de A y B mediante

$$f_C(x) = Max[f_A(x), f_B(x)], x \in X \tag{2.3}$$

o, de forma más abreviada $f_C = f_A \vee f_B$. Es importante resaltar que $\cup$ tiene la propiedad asociativa, es decir, $A \cup (B \cup C) = (A \cup B) \cup C$. De una forma más intuitiva, la unión de A y B es el menor de los conjuntos borrosos que contienen a A y a B.

La *intersección* de dos conjuntos borrosos A y B con funciones de pertenencia f_A y f_B, respectivamente, es un conjunto borroso C, escrito como $C = A \cap B$, cuya función de pertenencia se relaciona con las de A y B mediante

$$f_C(x) = Min[f_A(x), f_B(x)], x \in X \tag{2.4}$$

o, de forma más abreviada $f_C = f_A \wedge f_B$.

Con las operaciones de unión, intersección, y complementación definidas en 2.3, 2.4, y en 2.1, es sencillo extender a los conjuntos borrosos muchas de las identidades básicas válidas en los ordinarios. Por ejemplo, tenemos

Leyes de De Morgan

$$(A \cup B)' = A' \cap B' \tag{2.5}$$

$$(A \cap B)' = A' \cup B' \tag{2.6}$$

Leyes distributivas

$$C \cap (A \cup B) = (C \cap A) \cup (C \cap B) \tag{2.7}$$

$$C \cup (A \cap B) = (C \cup A) \cap (C \cup B) \tag{2.8}$$

Además de las operaciones de unión e intersección, se pueden definir otras maneras de obtener combinaciones de conjuntos borrosos y relacionarlas con otro. Entre las más importantes encontramos las que a continuación se comentan.

El *producto algebraico* de A y B se escribe AB y se define a partir de las funciones de pertenencia de A y B mediante la relación

$$f_{AB} = f_A f_B, AB \subset A \cap B \tag{2.9}$$

La *suma algebraica* de A y B se escribe $A + B$ y se define como

$$f_{A+B} = f_A + f_B \tag{2.10}$$

a condición de que la suma $f_A + f_B$ sea menor o igual que la unidad. De este modo, y a diferencia del producto algebraico, la suma algebraica sólo tiene sentido cuando $f_A(x) + f_B(x) \leq 1$ se satisfaga para todo x.

La *diferencia absoluta* de A y B se denota como $|A - B|$ y se define por

$$f_{|A-B|=|f_A-f_B|} \tag{2.11}$$

Hay dos tipos especiales de conjuntos difusos necesarios en los sistemas expertos difusos, los conjuntos difusos discretos, y los números difusos.

2.1.5.3. Conjuntos difusos discretos

Si X es finito, el conjunto difuso discreto D es un subconjunto difuso de X, que se escribe como

$$D = \{\frac{\mu_1}{x_1}, \frac{\mu_2}{x_2}, \ldots, \frac{\mu_n}{x_n}\} \tag{2.12}$$

donde el valor de pertenencia de x_1 en D es μ_1. Además, si X no es finito pero $D(x) \neq 0$ para todo $x = \{x_1, x_2, \ldots, x_n\}$ escribimos D como en la ecuación 2.12. Normalmente, el valor de verdad de un miembro en un conjunto difuso se denomina grado de pertenencia.

2.1.5.4. Número difusos

Los números difusos representan un número cuyo valor conocemos de manera incierta. Son un tipo especial de conjuntos difusos cuyos miembros son número reales y, por lo tanto, infinitos en extensión. La función que relaciona dichos números con su grado de pertenencia se denomina **función de pertenencia**, y un ejemplo está representado en la figura 2.9.

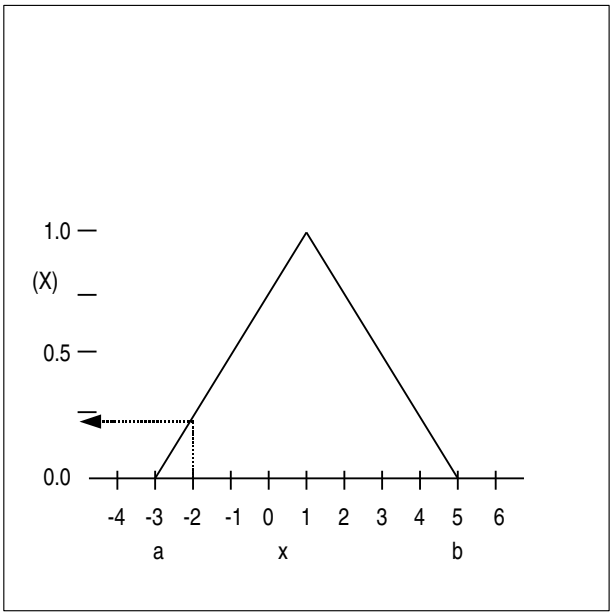

Figura 2.9: Función triangular de pertenencia

La pertenencia de un determinado número x se suele denotar como $\mu(x)$. Normalmente,

las funciones de pertenencia tienen distintas formas, siendo las más comunes las funciones de pertenencia triangulares y las funciones de pertenencia trapezoidales. Un ejemplo de función triangular podría comenzar su lado izquierdo en $x = a$, alcanzar su máximo valor en $x = b$, y descender hasta $x = c$, donde su valor de pertenencia sería 0. Dicha función estaría representada por las siguientes ecuaciones:

$$\mu(x) = 0, x \leq a \tag{2.13}$$

$$\mu(x) = (x - a)/(b - a), a < x \leq b \tag{2.14}$$

$$\mu(x) = (c - x)/(c - b), b < x \leq c \tag{2.15}$$

$$\mu(x) = 0, x > c \tag{2.16}$$

2.1.5.5. Variables lingüísticas y funciones de pertenencia

Actualmente, los conjuntos difusos discretos han sido formalizados con **variables lingüísticas**, consistentes en un nombre que se asocia a un conjunto difuso, los nombres de sus miembros (conocidos como valores lingüísticos), y para cada valor lingüístico una función de pertenencia como las definidas para los números difusos.

Un ejemplo de variable lingüística es la velocidad, cuyos miembros son *lenta*, *media*, y *rápida*. Las variables lingüísticas han provocado mucha atención debido a la importancia que tienen en el proceso de control difuso.

2.1.5.6. Álgebra de los conjuntos difusos

Dados los conjuntos difusos $A, B, C, \ldots$ todos subconjuntos de X, puede que queramos averiguar $A \cup B$, $B \cap C$, etc. En lógica difuso se utilizan los operadores AND y OR de la lógica clásica. Sin embargo, su nomenclatura es distinta ya que por un lado tenemos las *t-normas* (AND), y por otro lado las *t-conormas* (OR).

Una ***t-norma*** T es una función $[0, 1]x[0, 1]$ en $[0, 1]$. Es decir, si $z = T(x, y)$, entonces x, y, y z pertenecen al intervalo $[0, 1]$. Todas las *t-normas* cumplen las siguientes cuatro propiedades:

1. $T(x, 1) = x$ (límite)

2. $T(x, y) = T(y, x)$ (conmutatividad)

3. Si $y_1 \leq y_2$, entonces $T(x, y_1) \leq T(x, y_2)$ (unicidad)

4. $T(x, T(y, z)) = T(T(x, y), z)$ (asociatividad)

Las *t-normas* generalizan el AND de la lógica clásica. Este hecho significa que $tv^2(PANDQ) = T(tv(P), tv(Q))$ para cualquier *t-norma*. Las *t-normas* básicas son:

$$T_m(x, y) = min(x, y) \tag{2.17}$$

$$T_L(x, y) = max(0, x + y - 1) \tag{2.18}$$

$$T_p(x, y) = xy \tag{2.19}$$

T_m se denomina la interseción estándar (Zadeh [60]), y se trata de una de las más comúnmente utilizadas.

Por otra parte, una *t-conorma* generalizan la operación OR de la lógica clásica. Como en el caso de una *t-norma*, una *t-conorma* $C(x, y) = z$ tiene a x, y, y z siempre en el intervalo $[0, 1]$. Las propiedades básica de una *t-conorma* C son:

1. $C(x, 0) = x$ (límite)

2. $C(x, y) = C(y, x)$ (conmutatividad)

3. Si $y_1 \leq y_2$, entonces $C(x, y_1) \leq C(x, y_2)$ (unicidad)

4. $C(x, C(y, z)) = C(C(x, y), z)$ (asociatividad)

Las *t-conormas* básicas son las siguientes:

$$C_m(x, y) = max(x, y) \tag{2.20}$$

$$C_L(x, y) = min(0, x + y - 1) \tag{2.21}$$

$$C_p(x, y) = x + y - xy \tag{2.22}$$

[2]Valor de verdad

Las definiciones anteriores están referidas a dos varibles, pero en un sistema experto es necesario extenderlas para n variables:

$$T_m(x_1, \ldots, x_n) = min(x_1, \ldots, x_n) \tag{2.23}$$

$$C_m(x_1, \ldots, x_n) = max(x_1, \ldots, x_n) \tag{2.24}$$

2.1.5.7. Razonamiento aproximado

El **razonamiento aproximado** es el término empleado para referirse a la inferencia lógica difusa empleando el *modus ponens* difuso generalizado, una versión difusa del *modus ponens* clásico. La versión difusa del *modus ponens* se formula de la siguiente forma:

Si X es A entonces Y es B

de X = A'

se infiere que Y = B'

A' y A' son conjuntos difusos definidos en el mismo universo, y B y B' también son conjuntos difusos definidos en el mismo universo, pudiendo ser este universo distinto al primero. En el control difuso, normalmente las funciones de pertenencia de los conjuntos difusos están definidas en un eje real, y por lo tanto son números difusos.

El cálculo de B' a partir de $A, B,$ y A' es directo. Primero, se elige el operador de implicación difuso. La implicación $A(x) \rightarrow B(x)$ define la relación lógica $A \rightarrow B$ entre A y B. Después, B' se calcula mediante la composición de A' con $A \rightarrow B$, siguiendo el método del centroide:

$$B' = A' \circ (A \rightarrow B) \tag{2.25}$$

La conclusión difusa B' se obtiene usando la regla de composición de inferencia $B' = AoR$. Esta expresión define la función de pertenencia para la conclusión lógica B'.

Los sistemas de razonamiento aproximado se pueden clasificar de la siguiente forma:

- Métodos **directos**.
  - Método directo de **Mamdani**.
  - Modelado borroso de Takagi y Sugeno.

- Método simplificado.

- Métodos **indirectos**.

En el presente proyecto se utilizará un mecanismo de razonamiento aproximado de tipo Mamdani. Un ejemplo de funcionamiento con dos reglas es el siguiente:

Regla 1: Si x es A_1 e y es B_1 Entonces z es C_1
Regla 2: Si x es A_2 e y es B_2 Entonces z es C_2

$A_1, A_2, B_1, B_2, C_1, C_2$ son conjuntos borrosos, y el proceso de razonamiento se expone a continuación. Se supone que x_0 e y_0 son las entradas para las variables x e y de las premisas, es decir, los antecendentes de las reglas. El proceso consta de los siguientes pasos:

1. Obtener el grado de satisfacción de los antecendentes, a partir de la aplicación de operadores estándar sobre conjuntos borrosos. En este caso, se aplicarían las *t-normas*.

$$\text{Regla 1: } W_1 = min\lfloor \mu_{A_1}(x_0), \mu_{B_1}(y_0) \rfloor$$
$$\text{Regla 2: } W_2 = min\lfloor \mu_{A_2}(x_0), \mu_{B_2}(y_0) \rfloor$$

2. Aplicar los resultados obtenidos a los conjuntos borrosos en la parte del consecuente, de forma que se obtenga la conclusión de cada regla, utilizando el operador de implicación de Mamdani.

$$\text{Conclusión Regla 1: } \mu_{C_1}(x_0) = min\lfloor W_1, \mu_{C_1}(z) \forall z \in Z$$
$$\text{Conclusión Regla 2: } \mu_{C_2}(x_0) = min\lfloor W_2, \mu_{C_2}(z) \forall z \in Z$$

3. Construir la conclusión final a partir de la composición de todas las conclusiones parciales, utilizando para ello el operador estándar *t-conorma*.

$$\text{Conclusión final: } \mu_C(z) = max\lfloor \mu_{C_1}(z), \mu_{C_2}(z) \rfloor$$

Hay que tener en cuenta que la salida obtenida es un conjunto borroso, lo cual no nos sirve como salida, ya que en un sistema necesitamos un valor concreto. Por tanto, ha de existir un

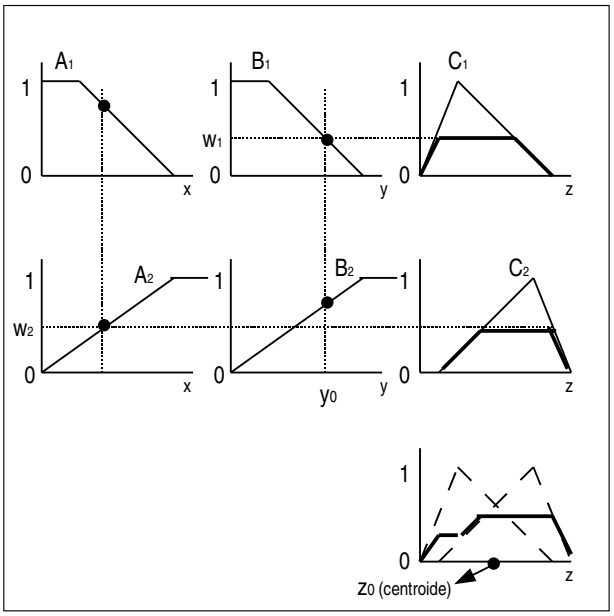

Figura 2.10: Razonamiento con el método de Mamdani

proceso que convierte el conjunto borroso de salida en un valor concreto. Esta operación es la *desborrosificación*, y entre los métodos más usados destacan:

$$z_0 = \frac{\int \mu_C(z) z dz}{\int \mu_C(z) dz} \tag{2.26}$$

$$z_0 = max(\mu_C(z)) \tag{2.27}$$

2.2. *Middlewares*

2.2.1. Introducción

Se puede entender un ***middleware*** como un *software* de conectividad que hace posible que aplicaciones distribuidas pueden ejecutarse sobre distintas plataformas heterogéneas, es decir, sobre plataformas con distintos sistemas operativos, que usan distintos protocolos de red y, que incluso, involucran distintos lenguajes de programación en la aplicación distribuida.

Desde otro punto de vista distinto, un *middleware* se puede entender como una abstracción en la complejidad y en la heterogeneidad que las redes de comunicaciones imponen. De hecho, uno de los objetivos de un *middleware* es ofrecer un acuerdo en las interfaces y en los mecanismos de interoperabilidad, como contrapartida de los distintos desacuerdos en *hardware*, sistemas operativos, protocolos de red, y lenguajes de programación.

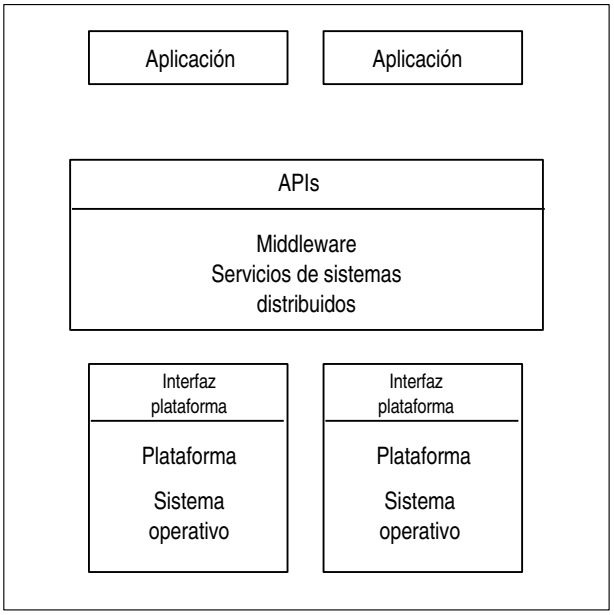

Figura 2.11: Esquema general de un *middleware*

En los distintos apartados que se exponen a continuación se realizará un estudio de los principales *middlewares* que actualmente están en uso, haciendo un mayor hincapié en ZeroC ICE, ya que ha sido la opción elegida para abordar la parte de comunicación de este pro-

yecto. Así mismo, se dedicará una sección a la exposición de los fundamentos básicos de un *middleware*.

2.2.2. Fundamentos básicos

La mayoría de los *middlewares* tratan de acercar el modelo de programación a un punto de vista local, es decir, enmascarando la llamada a los procedimientos remotos. Por otra parte, el enfoque más extendido es el de la generación de un *proxy* en la parte del cliente y de un *esqueleto* en la parte del servidor. Para ello, el cliente utiliza un objeto *proxy* con la misma interfaz definida en la parte del servidor, y que actúa como intermediario. El servidor, por otro lado, utiliza un *esqueleto* encargado de traducir los eventos de la red a invocaciones sobre el objeto en cuestión. Como se puede apreciar, existe un gran acercamiento de la versión distribuida a la versión centralizada.

Las principales responsabilidades de un *proxy* son las siguientes:

- Codificar la invocación y los argumentos en un mensaje.
- Esperar la respuesta y decodificar el valor o los valores de retorno.

Por otra parte, las principales responsabilidades de un *esqueleto* son las siguientes:

- Esperar una invocación y decodificar el mensaje y los argumentos.
- Invocar el método real.
- Codificar el valor o los valores de retorno en un mensaje de respuesta.

Una cuestión importante a tener en cuenta es la codificación de los datos en la red. Por ejemplo, es posible tener distintas representaciones de un mismo tipo de datos dependiendo del computador empleado. Para solucionar este problema, se suele definir una representación externa canónica, y transformar a y desde un formato binario a partir de la representación externa. Este proceso es conocido como proceso de *marshalling* y *unmarshalling*.

Por otra parte, el *middleware* también ha de gestionar problemas inherentes a las comunicaciones, como por ejemplo la identificación de mensajes, la gestión de retransmisiones, la gestión de conexiones, y la identificación de objetos. Por todo ello, la solución más extendida

se basa en un núcleo de comunicaciones genérico y de un generador automático de *proxies* y *esqueletos*.

2.2.3. CORBA

CORBA [17] es el acrónimo de **Common Object Request Broker Architecture**, y es un estándar que establece una plataforma para el desarrollo de sistemas distribuidos con el objetivo de facilitar la invocación a procedimientos remotos y utilizando un paradigma de orientación a objetos. CORBA utiliza el protocolo estándar IIOP, de forma que un programa que utilice CORBA de cualquier vendedor, en cualquier computador, lenguaje de programación, y red puede interoperar con otro programa del mismo u otro vendedor.

Dentro del consorcio **OMG** [22], que es el acrónimo de **Object Management Group**, CORBA representa la arquitectura de comunicaciones del OMG. Su objetivo es proporcionar un *bus software* independiente de quién fabricó la aplicación, de qué plataforma o plataformas la soportan, cómo se programó, y dónde se ejecuta.

Las aplicaciones CORBA están compuestas de objetos, que se definen como unidades individuales de ejecución que combinan funcionalidad y datos, y que frecuentemente representan algo en el mundo real. Típicamente, hay muchas instancias de un objeto de un determinado tipo, por ejemplo un sitio web de comercio electrónico podría tener múltiples instancias de objetos de tipo tárjeta de crédito, todos con la misma funcionalidad pero diferenciándose en que cada uno es asignado a un cliente distinto.

En *CORBA* se distinguen los siguientes actores:

- Los objetos que representan los modelos de objetos del mundo real, y que definen su interfaz a través de **IDL** (**Interface Description Language**).
- Los clientes que invocan métodos sobre los objetos de manera transparente, es decir, sin conocer su localización, ni el sistema operativo subyacente, ni el *hardware* en el que se ejecutan.
- El **ORB** (**Object Request Broker**), que es el mediador entre los clientes y los objetos encargado de garantizar la interoperabilidad.

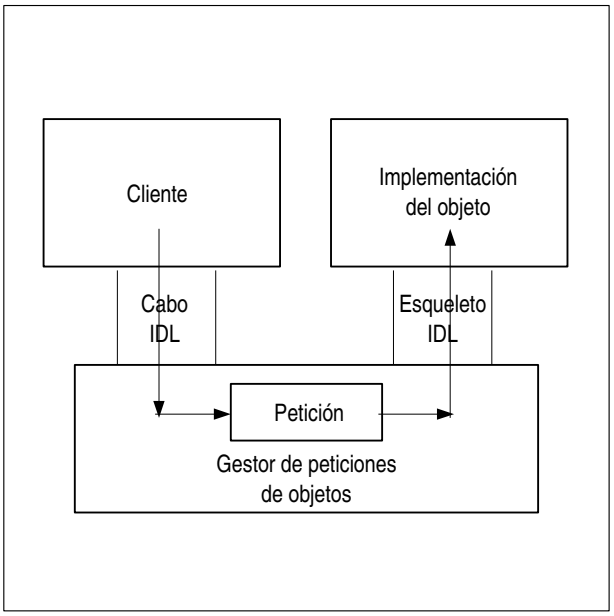

Figura 2.12: Funcionamiento general de CORBA

Para cada tipo de objeto se define una interfaz en OMG IDL. La interfaz define el contrato que el objeto servidor ofrece al cliente, y está especificada con una determinada sintaxis acorde con OMG IDL. Cualquier cliente que quiera invocar una operación en el objeto debe usar esta interfaz IDL especificar la operación que quiera llevar a cabo, realizando un proceso de *marshalling* con los argumentos a enviar. Cuando la invocación alcanza el objeto destino, la misma definición de interfaz es usada en el lugar remoto para llevar a cabo el proceso inverso, es decir, el proceso de *unmarshalling* de los argumentos. A partir de ahí, el objeto puede desarrollar la acción solicitada y realizar el mismo proceso para enviar los resultados de la operación (si así fuera).

La interfaz IDL es independiente de los lenguajes de programación empleados por el cliente y por el servidor, pero existe un *mapping* a los lenguajes de programación más populares, a partir de estándares del OMG, como por ejemplo C, C++, Java, Cobol, Smaltalk, Ada, Lisp, o Python. Esta separación de la interfaz de la implementación habilita la interoperabilidad de los distintos componentes de manera transparente.

Uno de los componentes importantes de CORBA es el **POA** (**Portable Object Adapter**),

que permite conectar peticiones sobre una referencia a un objeto con el objeto propiamente dicho. El POA está compuesto a su vez por tres componentes:

- La referencia al objeto.
- El mecanismo de conexión de peticiones.
- El código asociado que realiza la operación.

En CORBA, las referencias a los objetos se denominan **IOR** (**Interoperable Object Reference**), y garantizan que cualquier ORB las entienda.

Por otra parte, CORBA distingue entre dos conceptos distintos: el objeto virtual sobre el que se pueden invocar métodos y el sirviente o código que respalda a dicho objeto. El POA permite un sirviente por objeto, un único sirviente para varios objetos, múltiples sirvientes para un mismo objeto, y activación implícita de objetos.

Un ejemplo de interfaz podría ser la siguiente:

```
module Demo {
  interface hello {
    void puts(in string str);
  };
};
```

La anterior interfaz ofrece una operación *puts*, la cual tiene un parámetro de entrada de tipo *string*. Al compilar este ejemplo con un determinado compilador para un cierto lenguaje, se obtendrán distintos archivos que representan al interfaz en ese lenguaje, la definición de los *proxies* y *esqueletos*, entre otros.

En la parte del servidor se han de implementar todos los métodos definidos en la interfaz. Un ejemplo de implementación en Java acorde con la anterior interfaz podría ser el siguiente:

```
class HelloImpl extends Demo.helloPOA {
  public void puts(String str) {
    System.out.println(str);
  }
}
```

El siguiente paso consistiría en implementar las aplicaciones cliente y servidora, de forma que en la aplicación cliente se cree una referencia al objeto remoto y se invoque la operación, y en la aplicación servidora se instancie un objeto del tipo anteriormente expuesto.

OMG IDL permite la definición de tipos básicos de datos, como por ejemplo *short*, *long*, o *double*, además de tipos agregados como estructuras o enumeraciones.

Así mismo, CORBA permite características avanzadas como la activación implícita de objetos, invocación asíncrona, migración transparente, balanceo de carga entre servidores, calidad de servicio, políticas de seguridad, garantías de tiempo real, tolerancia a fallos, etc.

El problema técnico más obvio de CORBA es su complejidad, especialmente en lo referido a la complejidad de sus *APIs*. Otra área problemática es el *mapping* al lenguaje C++, sobre todo en lo referido a seguridad en el manejo de hilos, en seguridad en las excepciones, y en gestión de la memoria. Por otra parte, las referencias a objetos son entidades opacas cuyo contenido se supone que permanece oculto por parte de los desarrolladores [40].

2.2.4. SOAP/Web Services

Un **Web Service** [27] o servicio web se define como un sistema *software* diseñado para soportar interacciones entre dos máquinas de forma interoperable sobre una red. Tiene una interfaz descrita en un formato procesable a nivel máquina, que se denomina **WSDL** (**Web Services Description Language**). Otros sistemas interactúan con el servicio web a partir de su descripción usando mensajes **SOAP** (**Simple Object Adapter Protocol**), típicamente transmitidos usando HTTP con una serialización en XML, y en conjunto con otros estándares relaciones con Internet.

Un servicio web se puede entender como una abstracción que debe ser implementada por un agente en concreto. El agente es el *software* o el *hardware* en concreto que envía y recibe mensajes, mientras que el servicio es el recurso caracterizado por el conjunto abstracto de funcionalidad que proporciona. Desde otro punto de vista, la implementación de un agente puede cambiar con el tiempo pero manteniendo el mismo servicio.

Los mecanismos de intercambio de mensajes están documentados en una descripción de servicio web, que es una especificación procesable a nivel máquina de la interfaz del servicio web, y que está escrita en WSDL. Dicha especificación define los formatos del lenguaje, los tipos de datos, los protocolos de transporte, y los formatos de serialización de transporte que deberían usarse entre el agente que solicita un servicio y el agente que lo proporciona.

Además, también especifica una o más localizaciones de transporte en las que un agente que proporciona un servicio puede invocarse, e incluso puede proporcionar información sobre el patrón de intercambio de mensaje esperado. En esencia, la descripción del servicio representa una especie de acuerdo que rige los mecanismos de interacción con el servicio. A continuación se muestra un ejemplo de descripción de un servicio web:

```
<?xml version="1.0"?>
<definitions name="StockQuote"
 targetNamespace=
   "http://example.com/stockquote.wsdl"
 xmlns:tns="http://example.com/stockquote.wsdl"
 xmlns:xsd1="http://example.com/stockquote.xsd"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns="http://schemas.xmlsoap.org/wsdl/">
<types>
  <schema targetNamespace=
  http://example.com/stockquote.xsd
  xmlns="http://www.w3.org/2000/10/XMLSchema">
  <element name="TradePriceRequest">
   <complexType>
    <all>
     <element name="tickerSymbol"
      type="string"/>
    </all>
   </complexType>
  </element>
  <element name="TradePrice">
   <complexType>
    <all>
     <element name="price" type="float"/>
    </all>
   </complexType>
    </element>
     </schema>
 </types>
  <message name="GetLastTradePriceInput">
   <part name="body" element=
    "xsd1:TradePriceRequest"/>
  </message>
  <message name="GetLastTradePriceOutput">
   <part name="body" element="xsd1:TradePrice"/>
  </message>
  <portType name="StockQuotePortType">
   <operation name="GetLastTradePrice">
    <input message="tns:GetLastTradePriceInput"/>
    <output message="tns:GetLastTradePriceOutput"/>
   </operation>
  </portType>
   <binding name="StockQuoteSoapBinding"
    type="tns:StockQuotePortType">
    <soap:binding style="document"
```

```
    transport=
     "http://schemas.xmlsoap.org/soap/http"/>
  <operation name="GetLastTradePrice">
  <soap:operation
    soapAction=
     "http://example.com/GetLastTradePrice"/>
     <input>
      <soap:body use="literal"/>
     </input>
     <output>
      <soap:body use="literal"/>
     </output>
  </operation>
  </binding>
  <service name="StockQuoteService">
   <documentation>My first service</documentation>
   <port name="StockQuotePort"
   binding="tns:StockQuoteBinding">
   <soap:address location=
    "http://example.com/stockquote"/>
  </port>
 </service>
</definitions>
```

Como se puede apreciar en el ejemplo anterior, la especificación de una interfaz es bastante extensa y tediosa, e involucra aspectos internos que son ajenos a lo inherentemente relacionado con el concepto de interfaz. De hecho, se puede expresar el contraste con otros *middlewares* como ZeroC ICE si se expone su análogo al lenguaje de descripción empleado por los servicios web [39]:

```
interface StockQuoteService
{
  float GetLastTracePrice(string tickerSymbol):
};
```

2.2.5. .NET

.NET Remoting [21] es un API de *Microsoft* para la comunicación entre procesos. Fue lanzado en el año 2002 con la versión 1.0 del *framework* .NET. Dicho *framework* tiene una máquina virtual bastante eficiente, hace uso de una especificación pública, y es independiente del lenguaje de programación empleado. El objetivo principal de .NET es ofrecer una plataforma de desarrollo de *software* transparente a la red subyacente, con independencia de la plataforma empleada, y que permite el desarrollo de aplicaciones de una manera rápida. Actualmente existen dos implementaciones libres de .NET: *Mono* y *DotGNU*.

.NET Remoting implementa la invocación remota, y ofrece transparencia de acceso, de localización, y de protocolo de transporte al cliente. El mecanismo de funcionamiento es análogo al de la mayoría de *middlewares* actuales, y se basa en el uso de un *proxy* que actúa como representante remoto (o local) de un objeto que puede estar respaldado por uno o más servidores. .NET Remoting permite servidores con activación implícita, es decir, servidores que se activan en el momento en el que reciben una petición por parte de un cliente.

Existen distintos tipos de objetos que se pueden configurar como objetos de servicios remotos .NET, de forma que se puede seleccionar aquél que mejor se adapte a las necesidades de la aplicación. Cabe destacar los siguientes:

- Objetos de tipo *Single Call*: son aquellos que sólo pueden cubrir una solicitud entrante.
- Objetos de tipo *Singleton*: son aquellos que sirven a varios clientes y, por lo tanto, pueden compartir información al almacenar el estado entre las llamadas de los clientes.
- Objetos activados en el cliente: son aquellos objetos del lado del servidor que se activan cuando el cliente realiza la solicitud.

Para llevar a cabo el envío de mensajes .NET Remoting requiere de un objeto canal, entre los que destacan HTTP/SOAP, TCP/binario, e IPC/binario. Cada canal es capaz de establecer las conexiones, de traducir la invocación a un mensaje, y de traducir el resultado a un mensaje.

Como especificación de la interfaz, .NET Remoting permite utilizar cualquier lenguaje soportado por .NET. Además, no es necesario compilar la interfaz y no se necesitan convenios especiales de nombrado. Como ejemplo, podemos estudiar el siguiente caso:

```
namespace Demo {
  interface Hello {
    void puts(string str);
  }
}
```

La implementación del objeto remoto es muy sencilla:

```
public class Server
  : System.MarshalByRefObject,
    Demo.Hello {

  public voi puts(string str) {
    Console.WriteLine(str);
  }
}
```

La aplicación servidora ha de crear un canal de comunicación, como por ejemplo un canal TCP, crear un objeto remoto, y crear un proxy a ese objeto remoto. Por otra parte, la aplicación cliente ha de crear un canal de comunicación y un proxy al objeto remoto, para posteriormente llevar a cabo la invocación remota. Como se puede apreciar, el funcionamiento es totalmente análogo al resto de *middlewares* estudiados.

.NET utiliza metadatos y ensambladores para almacenar la información sobre los componentes, habilitando la programación en varios lenguajes. Además, con dichos metadatos, los servicios remotos de .NET pueden crear *proxies* de manera dinámica, lo cual es una gran ventaja. Los *proxies* creados en el lado del cliente presentan los mismos miembros que la clase original, reenviandos las solicitudes al objeto original a través de los servicios remotos de .NET. De hecho, dichos metadatos son también utilizados para interceptar mensajes entrantes.

2.2.6. Java RMI

Java RMI (**Java Remote Method Invocation**) [12] permite al programador crear aplicaciones distribuidas basadas en tecnología Java., en las que los métodos de los objetos remotos pueden invocarse desde otras máquinas virtuales Java, que posiblemente estarán en equipos remotos. Java RMI utiliza la serialización de objetos para llevar a cabo el proceso de *marshalling* y *unmarshalling* de los parámetros sin truncar los tipos, de manera que haya un soporte verdadero del polimorfismo.

El mecanismo de comunicación empleado en Java RMI es esencialmente análogo al empleado en Corba:

1. Se define una interfaz, con la peculiaridad de que se hace directamente en Java y que debe heredar de la interfaz *java.rmi.Remote*.
2. Se utiliza un esqueleto como base en el servidor.
3. La compilación se lleva a cabo con un generador de *proxies*.
4. Se emplea un *proxy* en el lado del cliente para ejecutar las invocaciones remotas.

Un ejemplo de interfaz sería la siguiente:

```
package Demo;

public interface Hello extends java.rmi.Remote {
  void puts (String str) throws java.rmi.RemoteException;
}
```

La implementación del objeto remoto consistiría en crear una clase que heredara del esqueleto generado al compilar la interfaz anterior y, posteriormente, efectuar la implementación de las operaciones definidas en dicha interfaz.

Java RMI también ofrece características avanzadas como la activación implícita, pero tiene ciertas restricciones. Al estar completamente implementado en Java no es posible emplear otros lenguajes de programación y necesita del conocimiento de ciertos convenios de nombrado. De hecho, no necesita un lenguaje de descripción de interfaces, ya que emplea directamente el lenguaje Java.

2.2.7. ZeroC ICE

2.2.7.1. Introducción

ICE (**Internet Communication Engine**) [30] es un *middleware* orientado a objetos, es decir, ICE proporciona herramientas, *APIs*, y soporte de bibliotecas para construir aplicaciones cliente-servidor orientadas a objetos. Una aplicación ICE se puede usar en entornos heterogéneos: los clientes y los servidores pueden escribirse en diferentes lenguajes de programación, pueden ejecutarse en distintos sistemas operativos y en distintas arquitecturas, y pueden comunicarse empleando diferentes tecnologías de red. Además, el código fuente de estas aplicaciones puede portarse de manera independiente al entorno de desarrollo. Los principales objetivos de diseño de ICE son los siguientes:

- Proporcionar un *middleware* listo para usarse en sistemas heterogéneos.
- Proveer un conjunto completo de características que soporten el desarrollo de aplicaciones distribuidas reales en un amplio rango de dominios.
- Evitar una complejidad innecesaria, haciendo que ICE sea fácil de aprender y de usar.

- Proporcionar una implementación eficiente en ancho de banda, en uso de memoria, y en carga de CPU.
- Proporcionar una implementación basada en la seguridad, de forma que se pueda usar sobre redes no seguras.

Se puede decir que la filosofía de ICE se basa en construir una plataforma tan potente como CORBA, pero sin cometer todos los fallos de ésta y evitando una complejidad innecesaria.

La justificación de la elección de ICE como plataforma para el desarrollo de la parte de comunicaciones de MASYRO se basa en los siguientes puntos:

- Permite la separación de las interfaces y de la implementación.
- Es fácil de aprender y de usar.
- Tiene licencia GPL.
- Es multilenguaje y multiplataforma.
- Tiene uno de los conjuntos de características más amplio en lo que a *middlewares* se refiere.
- Tiene una comunidad de desarrollo muy activa y que ofrece mucho soporte.

En cuanto a terminología, ICE emplea unos conceptos muy comunes con el resto de *middlewares* existentes. Se emplea el término **cliente** para denotar a una entidad activa, es decir, una entidad que solicita un servicio a un **servidor**. Por el contrario, el servidor es una entidad pasiva que proporciona servicios en respuesta a las peticiones de los clientes. Normalmente, los servidores no son servidores puros, es decir, que también actúan como clientes, haciéndose más frecuente el uso del concepto cliente-servidor. El concepto de **objeto** también es importante, y hace referencia a una abstracción caracterizada por los siguientes factores:

- Un objeto ICE es una entidad en el espacio de direcciones local o remoto, y que puede responder a las peticiones de los clientes.

- Un objeto ICE puede instanciarse en un servidor o, de manera redundante, en múltiples servidores.
- Cada objeto ICE tiene una o varias interfaces, y una interfaz es una colección de operaciones proporcionadas por un objeto. Los clientes emiten peticiones invocando estas operaciones.
- Una operación tiene cero o más parámetros y puede tener o no un valor de retorno. Así mismo, los parámetros son de un determinado tipo y pueden ser de entrada o de salida.
- Un objeto ICE tiene una interfaz principal, y puede proporcionar cero o más interfaces conocidas como facetas.
- Cada objeto ICE tiene una identidad única, que lo distingue del resto de objetos.

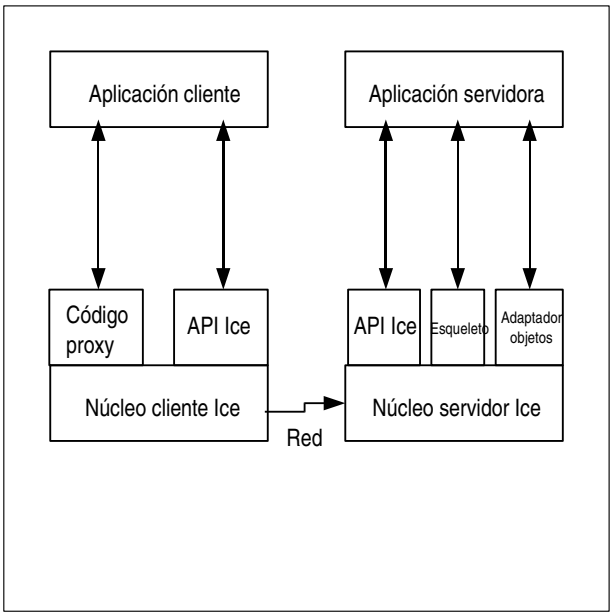

Figura 2.13: Estructura cliente-servidor en ICE

Al igual que en otros *middlewares*, ICE maneja el concepto de *proxy*, que es un artefacto local al espacio de direcciones del cliente y que representa al posiblemente remoto objeto

ICE para el cliente. Visto de otro modo, un *proxy* se puede ver como el embajador local de un objeto ICE. Existen tres tipos de *proxies*:

1. *Stringfied proxies*: la información del *proxy* se expresa mediante una cadena de texto, como por ejemplo *SimplePrinter:tcp -h <ip>-p <port>*. El API de ICE proporciona operaciones para convertir un *proxy* a su representación textual y viceversa.

2. *Direct proxies*: es un *proxy* que encapsula la identidad del objeto junto con la dirección en la que el servidor se ejecuta. Dicha dirección queda completamente especificada por un identificador de protocolo (como TCP/IP) y por una dirección específica al protocolo (nombre del equipo y puerto).

3. *Indirect proxies*: puede verse de dos formas, proporcionanado sólo la identidad del objeto, o proporcionando dicha identidad más un identificador de adaptador de objetos. Un objeto que es accesible empleando sólo su identidad se conoce como *well-known object* (objeto bien conocido). El adaptador de objetos representa el nexo entre el núcleo de ejecución de ICE y el código de la aplicación servidora.

La replicación en ICE está basada en los adaptadores de objetos, especificando múltiples direcciones. Un ejemplo sería el siguiente:

```
SimplePrinter:tcp –h server1 –p 10001:tcp –h server2 –p 10002
```

Además de dicha replicación, ICE soporta una tipo de replicación más interesante conocida como *replica group* y que requiere el uso de un servicio de localización. Dicha replicación mantiene un único identificador y consiste en un número indeterminado de adaptadores de objetos, de manera que el servicio de localización podría devolver todas las direcciones de los adaptadores de objetos, en cuyo caso el núcleo de ejecución en la parte del cliente podría decidir cuál usar en función de una heurística.

ICE garantiza una semántica *at-most-once*, de manera que el núcleo de ejecución hace todo lo posible para entregar una petición al destino correcto, e incluso si se diera el caso puede tratar de volver a realizar la petición. Por otra parte, ICE también permite invocación asíncrona de operaciones en el lado del cliente, y su homólogo en el lado del servidor, aunque por defecto el modelo de peticiones usado por ICE es síncrono.

2.2.7.2. Slice

Slice (**Specification Language for ICE**) es el lenguaje de descripción de interfaces de ICE. Permite, entre otras cosas, definir interfaces, operaciones, y los parámetros empleados por estas últimas. Slice permite los contratos entre los clientes y los servidores de una forma independiente del lenguaje de programación empleado. Las reglas que determinan cómo cada construcción de Slice se translada a cada lenguaje de programación específico son conocidas como *language mappings*. Actualmente, ICE proporciona *language mappings* para los lenguajes C++, Java, C#, Visual Basic .NET, Python, y PHP para el lado del cliente.

Un ejemplo de parte de un archivo de especificación en Slice sería el siguiente:

```
#ifndef _MASYRO
#define _MASYRO

module MASYRO
{

/************************/
/*Estructuras generales.*/
/************************/

// Estructura que define una zona a renderizar.
struct TZone {

   // Identificador de la zona.
   int id;
   // Coordenadas que definen la zona.
   int x1;
   int y1;
   int x2;
   int y2;
   // Desviación estándar del color de la zona.
   float d;
   // Media de color en la zona.
   float m;

};

sequence <TZone> TZones;

....
....

/**************************/
/*Definición de excepciones*/
/**************************/

// Excepción base.
```

```
exception BaseException
{
   string Reason;
};

....
....

/**************************/
/*Descripción de interfaces.*/
/**************************/

// RenderAgent es la interfaz a un agente especializado en el render.
interface RenderAgent
{

   // La operación notifyNewWork permite al RenderAgent
      conocer la existencia de un nuevo trabajo.
   void notifyNewWork(TZones zones, int idWork, int benchmarkValue);
   // La operación notifyZone asigna una zona del nuevo trabajo al RenderAgent,
      para que éste haga un estudio previo.
   ["ami"] void notifyZone(TZone zone, int idWork);
   // La operación beginRenderProcess notifica el comienzo del trabajo.
   void beginRenderProcess();
   // La operación render notifica al agente el comienzo del renderizado
      para la zona idZone del trabajo idWork.
   ["ami"] void render(int idWork, int idZone, string agent);

};
```

Como se puede apreciar, se pueden definir tipos de datos de usuario, como por ejemplo las estructuras o las secuencias. Además, Slice también permite la definición de excepciones, interfaces, e incluso de clases.

2.2.7.3. El protocolo ICE

ICE proporciona un protocolo RPC que puede usar TCP/IP o UDP como protocolo de transporte. Además, ICE también permite usar SSL como transporte, de forma que todas las comunicaciones entre el cliente y el servidor están encriptadas. El protocolo ICE define:

- Un número de tipos de mensajes, como los mensajes de petición y respuesta.
- Una máquina de estados de protocolos que indica qué secuencias de distintos tipos de mensajes se intercambian entre el cliente y el servidor, junto con los establecimientos de conexión asociados.

- Reglas de codificación que determinan como se representa cada tipo de dato en el medio.
- Una cabecera para cada tipo de mensaje que contiene detalles tales como el tipo de mensaje, el tamaño del mismo, y el protocolo y versión de codificación empleados.

El protocolo ICE también ofrece operaciones bidireccionales: si un servidor quiere enviar un mensaje a un objeto de retrollamada proporcionado por el cliente, la retrollamada puede llevarse a cabo por la conexión que el cliente estableció con el servidor previamente. Esta característica es de especial importancia cuando el cliente está detrás de un cortafuegos que permite conexiones salientes pero no entrantes.

2.2.7.4. Servicios avanzados de ICE

El núcleo de ICE proporciona una plataforma cliente-servidor para el desarrollo de aplicaciones distribuidas. Sin embargo, las aplicaciones reales normalmente demandan servicios como la activación de servidores bajo demanda, la distribución de *proxies* a clientes, distribución de eventos asíncronos, configuración de aplicaciones, etc. ICE ofrece una serie de servicios muy útiles para el desarrollo de aplicaciones distribuidas. Los utilizados en este proyecto se comentarán a continuación, y son **IceGrid**, **IceStorm**, y **Glacier2**.

IceGrid es el servicio de localización y ativación de aplicaciones ICE. Sus características principales son las siguientes:

- Servicio de localización.
- Activación del servidor bajo demanda.
- Distribución de aplicaciones.
- Replicado y balanceado de carga.
- Recuperado automático.
- Preguntas dinámicas.
- Monitorización del estado.

- Administración.
- Facilidad de desarrollo.

La arquitectura básica de IceGrid consiste en un sencillo registro y un número cualquiera de nodos. Ambos cooperan para gestionar la información y para servir los procesos que incluyen aplicaciones, de forma que cada aplicación asigna servidores a los nodos. El registro contiene una tabla persistente de esta información, donde los nodos son los responsables de arrancar y monitorizar sus procesos de servidor asignados. En una configuración típica, cada nodo corre sobre un computador que provee servidores ICE. El registro no consume mucho tiempo de CPU, por lo que comúnmente corre en el mismo computador que un nodo.

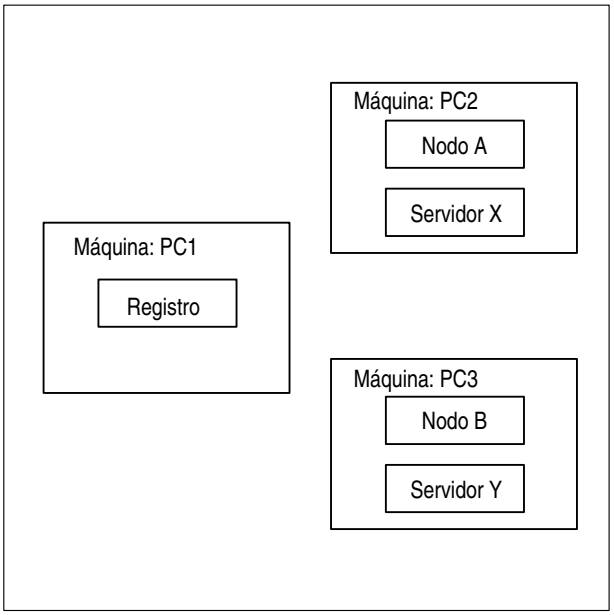

Figura 2.14: Aplicación simple con IceGrid

La principal responsabilidad del registro es resolver *proxies* indirectos como un servicio de localización de ICE. Cuando un primer cliente trata de usar un *proxy* indirecto, la parte de ICE relativa al cliente contacta con el registro para convertir la información simbólica del *proxy* en un *endpoint* que permita al cliente establecer una conexión. En IceGrid el despliegue es el proceso de describir una aplicación en el registro, la cual se especifica con el lenguaje

XML. Dicha acción está compuesta de la siguiente información:

- Grupos de réplica, que son colección de adaptadores de objetos replicados.
- Nodos, a los cuales se deben asignar los servidores.
- Servidores, con un nombre único, una ruta al archivo ejecutable, y los adaptadores de objetos que crea.
- Adaptadores de objetos, que contienen los *endpoints*, los objetos bien conocidos que anuncia, y el identificador del grupo si pertenece a un grupo de réplica.

Un ejemplo de descriptor sería el siguiente:

```
<icegrid>

 <application name="MASYRO">

  <node name="localhost">

   <server id="StartService" exe="./StartService" activation="on-demand">
    <adapter name="StartServiceAdapter" id="StartServiceAdapter"
             register-process="true" endpoints="tcp">
     <object identity="startService" type="::FIPA::StartService"/>
    </adapter>
    <property name="IdentitySS" value="startService"/>
    <property name="InputSS" value="inputSS.xml"/>
   </server>

   ....
   ....

  </node>

 </application>

</icegrid>
```

IceStorm es un eficiente servicio de publicación-subscripción para aplicaciones ICE. Actúa de mediador entre el publicador y el subscriptor, ofreciendo varias ventajas:

- Para distribuir información a los subscriptores sólo es necesaria una simple llamada al servicio IceStorm.
- Los monitores interactúan con el servidor IceStorm para llevar a cabo la subscripción.

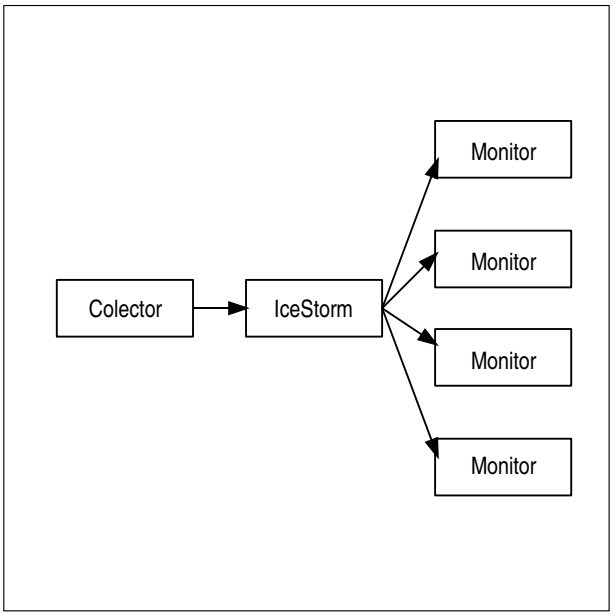

Figura 2.15: Ejemplo de aplicación con IceStorm

- Los cambios de la aplicación para incluir IceStorm son mínimos.

Una aplicación indica su interés en recibir mensajes subscribiéndose a un *topic*. Un servidor IceStorm soporta cualquier número de *topics*, los cuales son creados dinámicamente y distinguidos por un nombre único. Cada *topic* puede tener varios publicadores y subscriptores. Un *topic* es equivalente a una interfaz Slice: las operaciones del interfaz definen los tipos de mensajes soportados por el *topic*. Un publicador usa un *proxy* al interfaz *topic* para enviar sus mensajes, y un subscriptor implementa la interfaz *topic* (o derivada) para recibir los mensajes. Realmente dicha interfaz representa el contrato entre el publicador (cliente) y el subscriptor (servidor), excepto que IceStorm encamina cada mensaje a múltiples receptores de forma transparente.

Glacier2 es una solución de cortafuegos ligera para aplicaciones ICE, que permite a los clientes y a los servidores comunicarse a ravés de un cortafuegos de una manera segura. El tráfico cliente-servidor queda totalmente encriptado usando certificados de clave pública de forma bidireccional. Además, Glacier2 proporciona soporte para autenticación y gestión de sesiones. Este servicio trata de solventar las dificultades que suponen escenarios comunes en

Internet, simulando un escenario menos complejo:

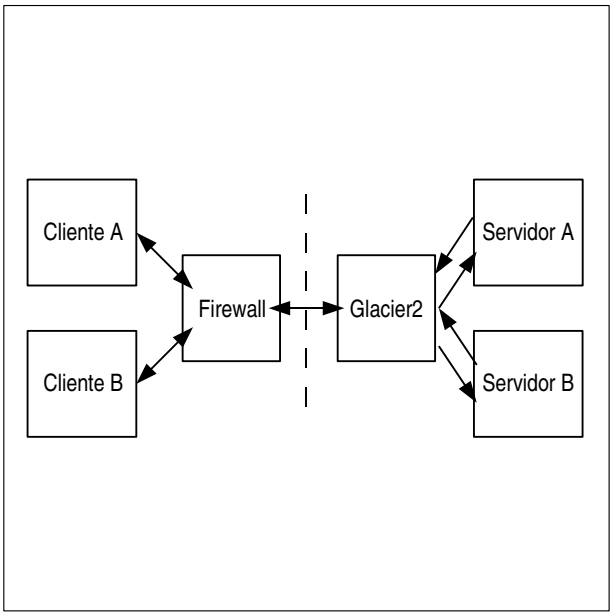

Figura 2.16: Escenario creado por Glacier2

Glacier2 proporciona las siguientes ventajas:

- Los clientes requieren un cambio mínimo.
- Sólo es necesario un puerto para soportar cualquier número de servidores.
- El número de conexiones a los servidores es reducido.
- Los servidores no se dan cuenta de la presencia de Glacier2, ya que éste actúa como un cliente más. Además, IceGrid puede seguir usándose de forma transparente.
- Las retrollamadas de los servidores se envían sobre la conexión establecida entre el cliente y el servidor.
- Glacier2 no requiere conocimiento de la aplicación.
- Glacier2 ofrece servicios de gestión y autenticación de sesiones.

2.3. Síntesis de imagen tridimensional

2.3.1. Introducción al proceso de síntesis

Se puede definir el proceso de **síntesis de imagen fotorrealista** como aquel que persigue la creación de imágenes sintéticas que no se puedan distinguir de aquellas captadas en el mundo real. Dentro de este campo de los gráficos por computador, se ha producido una gran evolución. Desde algoritmos como el de *Scan-Line*, propuesto por Bouknight [33] y que no realizaba ninguna simulación física de la luz, hasta los comienzos del fotorrealismo, en el artículo de Whitted [58] con la propuesta del método de trazado de rayos y, posteriormente, la introducción de la radiosidad [35], haciendo uso de técnicas habituales en el campo de la óptica y el cálculo de transferencia de calor.

Uno de los puntos clave a la hora de llevar a cabo la generación de imagen fotorrealista es la simulación de la luz. Dicha simulación siguiendo las leyes de la física de toda la luz dispersa en un modelo 3D sintético se denomina **iluminación global**. El objetivo de la iluminación global es simular todas las reflexiones de la luz y obtener en cada punto del modelo el valor de la intensidad de luz que vendrá determinado por la interacción de la luz con el resto de elementos de la escena. De este modo, el objetivo del algoritmo de iluminación global es calcular la interacción de todas las fuentes de luz con todos los elementos que conforman la escena. Esta técnica es importante porque permite calcular la llamada *iluminación indirecta*, ya que sin esta iluminación el modelo parece artificial y sintético.

Existen varias técnicas de iluminación global, pero la mayoría están basadas en el **Trazado de rayos** (o Muestreo de Puntos) o en **Radiosidad** (o Elementos Finitos), o en ambas a la vez (técnicas híbridas). Como se puede suponer, cada uno de estos métodos tiene sus ventajas y sus inconvenientes.

Mediante la **técnica de trazado de rayos** se trazan un gran número de rayos de luz a través del modelo tridimensional que se pretende renderizar. En este modelo, introducido en 1980 por Whitted [58], se trazan rayos desde el observador hacia las fuentes de luz. Algunas de las importantes limitaciones de este algoritmo es que no se pueden representar profundidad de campo, caústicas, o iluminación indirecta. Para simular estos efectos, es necesario extender la técnica base introduciendo **métodos de Monte Carlo** [52], que lancen rayos adicionales para

simular todos los posibles caminos de la luz. Sin embargo, los métodos de Monte Carlo tienen un problema importante debido a su varianza, que se percibe como ruido en las imágenes generadas.

La eliminación de este ruido requiere aumentar el número de muestras. Además, se han llevado a cabo estudios para distribuir los rayos de forma que el ruido sea lo menor posible, concluyendo en que los mejores métodos son los producidos por los métodos guiados de trazado de rayos, cuyas propiedades de convergencia son mejores.

En las técnicas de trazado de rayos, la geometría se trata como una *caja negra*, de forma que los rayos se trazan en la escena y obtienen un valor de iluminación. Esta es una importante ventaja en escenas con una geometría más o meno compleja, ya que las técnicas de muestreo de puntos sólo tienen que gestionar la complejidad de la iluminación, y no de la geometría. Sin embargo, esta independencia de la geometría también tiene sus inconvenientes, como por ejemplo la necesidad de un alto número de muestras para obtener unos resultados aceptables.

El otro gran tipo de técnicas son las **técnicas de radiosidad**. En este tipo de técnicas se calcula el intercambio de luz entre superficies. Este cálculo se consigue subdividiendo el modelo en pequeñas unidades denominadas *parches*, que serán la base de la distribución de luz final.

El modelo básico de radiosidad calcula una solución independiente del punto de vista. Sin embargo, el cálculo de la solución es muy costoso en tiempo y en espacio de almacenamiento. No obstante, cuando la iluminación ha sido calculada, puede utilizarse para renderizar la escena desde diferentes ángulos, lo que hace que este tipo de soluciones se utilicen en visitas interactivas y videojuegos en primera persona actuales.

En el modelo de radiosidad, cada superficie tiene asociados dos valores: la intensidad luminosa que recibe, y la cantidad de energía que emite (energía radiante). En este algoritmo se calcula la interacción de energía desde cada superficie hasta el resto. Si tenemos n superficies, la complejidad del algoritmo será $O(n^2)$. El valor matemático que calcula la relación geomérica entre superficies se denomina *factor de forma*. Será necesario calcular n^2 factores de forma ya que no se cumple la propiedad conmutativa debido a que se tiene en cuenta la relación de área entre superficies. La matriz que contiene los factores de forma relacionando todas las superficies se denomina *matriz de radiosidad*.

En general, el cálculo de la radiosidad es eficiente para el cálculo de distribuciones de luz en modelos simples con materiales difusos, pero resulta bastante costoso para modelos complejos, ya que se calculan los valores de energía para cada parche del modelo, o con materiales no difusos. Además, la solución del algoritmo se muestra como una nueva malla poligonal que tiende a desenfocar los límites de las sombras. Esta malla poligonal puede resultar inmanejable en complejidad si la representación original no se realiza con cuidado, o si el modelo es muy grande, por lo que este tipo de técnicas no se utilizan en la práctica con modelos complejos.

La radiosidad es buena para simular las reflexiones difusas, mientras que el trazado de rayos da mejores resultados en reflexión especular. De este modo, se puede utilizar el trazado de rayos para el cálculo de las reflexiones especulares y radiosidad para superficies con reflexión difusa. Existen varias aproximaciones en las que se utiliza radiosidad para el cálculo de iluminación indirecta, ya que las técnicas de Monte Carlo requieren un alto número de muestras para que no se perciba el ruido. El uso de algoritmos basados en radiosidad tienen un importante inconveniente, y es que limitan la complejidad de los modelos que pueden ser renderizados. Una aproximación a estos problemas se basa en simplificar la malla origen para utilizarla únicamente en el cálculo de la iluminación indirecta, cuyos cambios suelen ser suaves y no requieren un alto nivel de detalle en la superficie.

2.3.2. Taxonomía de métodos de render

En las sucesivas etapas se realizará un breve recorrido por los principales métodos de render utilizados actualmente, haciendo especial hincapié en los métodos de renderizado píxel a píxel.

Se puede considerar que todas las alternativas de renderizado pretenden dar solución a la ecuación de renderizado, propuesta en 1986 por Kajiya [45]:

$$L_0(x, \overrightarrow{w}) = L_e(x, \overrightarrow{w}) + \int_\Omega f_r(x, \overrightarrow{w}', \overrightarrow{w}) L_i(x, \overrightarrow{w}')(\overrightarrow{w}' \overrightarrow{n}) d\overrightarrow{w}' \tag{2.28}$$

Dicha ecuación puede interpretarse como: “dada una posición x, y dirección $\overrightarrow{w}$ determinada, el valor de iluminación saliente L_0 es el resultado de sumar la iluminación emitida L_e y la luz reflejada. La luz reflejada (el segundo término de la suma de la ecuación anterior)

viene dado por la suma de la luz entrante L_i desde todas las direcciones, multiplicando por la reflexión de la superficie y el ángulo de incidencia.

Para expresar la interacción de la luz en cada superficie, se utiliza la función de distribución de reflectancia biridireccional, que correspondería con el término f_r de la expresión anterior, y que es particular de cada superficie.

2.3.2.1. Scanline

El algoritmo de **Scanline** es un método que fue propuesto en 1970 [33], como algoritmo para la representación punto a punto de una imagen. Cuando se calculan los píxeles que forman cada línea de la imagen, se continúa con la siguiente.

Las ventajas que ofrece el algoritmo Scanline giran en torno a su rapidez, ya que permite simular cualquier tipo de sobreado, funcionando correctamente con texturas. Entre los inconvenientes, destaca que no permite simular de forma realista reflexiones y refracciones de la luz

2.3.2.2. Raytracing

El método de **Raytracing** es una propuesta elegante y recursiva que permite calcular superficies con reflejos y transparencia de una manera sencilla. Este método fue propuesto en 1980 por Whitted [58].

La idea base en Raytracing es la de trazar rayos desde el observador a las fuentes de luz. En realidad, son las fuentes de luz las que emiten fotones que rebotan en la escena y llegan a los ojos del observador. Sin embargo, sólo una pequeñísima fracción de los fotones llegan a este destino, por lo que el cálculo en esta forma directa resulta demasiado costoso. Los rayos que se emiten a la escena son evaluados respecto de su visibilidad trazando nuevos rayos desde los puntos de intersección (rayos de sombra).

Existen distintos métodos de aceleración del algoritmo de Raytracing, como el uso de árboles octales, grids jerárquicos, árboles BSP, etc. Además, se pueden utilizar técnicas que permiten interpolar los valores de píxeles cercanos si la variación es pequeña.

La principal ventaja de este método es que permite simular reflexiones de espejo y materiales transparentes.

Los inconvenientes se obtienen de la naturaleza del Raytracing, ya que no es un método de iluminación global. Por lo tanto, no permite la simulación de iluminación indirecta ni sombras difusas. Además, sólo funciona perfectamente en superficies con brillo especular donde la luz entrante tenga una única dirección de reflexión.

2.3.2.3. Ambient Occlusion

La técnica de **Ambient Occlusion** es un caso particular del uso de pruebas de oclusión en entornos con iluminación local para determinar los efectos difusos de iluminación. Fueron introducidos inicialmente por Zhurov en 1998 como alternativa a las técnicas de radiosidad para aplicaciones interactivas, por su bajo coste computacional [53]. Se puede definir la ocultación de un punto de una superficie mediante la siguiente ecuación:

$$W(P) = \frac{1}{\pi} \int_{w \in \Omega} \rho(d(P, \omega)) \cos \Theta d\omega \tag{2.29}$$

La principal ventaja de Ambient Occlusion es que es una técnica de al menos un orden de magnitud más rápido que el Pathtracing, y en muchos caso, la simulación obtenida es suficientemente buena y con menos ruido. Sin embargo, no es un método de iluminación global, aunque ofrece aproximaciones bastante buenas. Tampoco permite la simulación de caústicas, y puede resultar costoso si aumentamos el número de muestras (con el objetivo de eliminar ruido).

2.3.2.4. Radiosidad

En este tipo de técnicas se calcula el intercambio de luz entre superficies. Inicialmente fue propuesto por Goral en 1984 [35]. El cálculo del intercambio de luz entre superficies se hace mediante el factor de forma, calculado mediante la siguiente ecuación:

$$F_{i,j} = \frac{1}{A} \int_{A_i} \int_{A_j} \frac{\cos \Theta_i \cos \Theta_j}{\pi r^2} H_{ij} dA_j dA_i \tag{2.30}$$

Siendo $F_{i,j}$ el factor de forma de la superficie i a la superficie j, $\cos \Theta_i \cos \Theta_j$ es el ángulo entre las normales de los planos de cada parche. πr^2 mide la distancia entre los parches y H_{ij} es el factor de visibilidad, entre 0 y 1, que indica cuánta superficie de j ve la superficie de i debido a la oclusión de otras superficies. Finalmente, dA_x es el área de la superficie x.

Las principales ventajas de este algoritmo se resumen en que ofrece buenos resultados con superficies difusas y, que al ofrecer una solución independiente del punto de vista, pueden emplearse para renderizar animaciones, incluso de forma interactiva. Como contrapartida, limita la complejidad de la geometría con la que se trabaja, y tiende a suavizar las zonas de penumbra, lo cual no siempre es deseable. Además, el tiempo de cálculo de la matriz de radiosidad puede ser muy alto, incluso trabajando con técnicas incrementales.

2.3.2.5. Pathtracing

El algoritmo de **Pathtracing** es una extensión del algoritmo de Raytracing clásico, que permite calcular una solución de iluminación global completa. Fue formulado por Kajiya en 1986 como solución a la ecuación de renderizado (en el mismo artículo donde era propuesta) [45], basándose en las ideas de Cook en su artículo de 1984 [52].

El Pathtracing se basa en la idea de calcular todos los posibles caminos de la luz en sus rebotes en superficies que no tengan reflejo especular. Estas superficies requieren la evaluación de integrales complejas, las cuales se solucionan trazando rayos aleatorios dentro del dominio de la integral para estimar su valor (empleando integración de Monte Carlo). Realizando la media de un alto número de muestras por cada píxel se puede obtener una estimación del valor de la integral para ese píxel.

Una cuestión importante del Pathtracing es que sólo utiliza un rayo reflejado para estimar la iluminación indirecta. Visto de otro modo, cuando un rayo choca sobre una superficie difusa, sólo se realizará una llamada con un nuevo rayo de cálculo de la iluminación indirecta, cuya dirección será aleatoria dentro del dominio de definición de la superficie.

La principal ventaja de este método es que si la iluminación varía poco (como en una escena exterior), ofrece resultados sin ruido con pocas muestras. Sin embargo, y para que no se perciba ruido en la imagen final, hay que aumentar el número de muestras por píxel, sobre todo en escenas donde hay variaciones muy fuertes en las fuentes de luz.

2.3.2.6. Photon Mapping

Photon Mapping es un método que desacopla la representación de la iluminación de la geometría. Se realiza en dos pasos, primero se construye la estructura del mapa de fotones

(trazado de fotones), desde las fuentes de luz al modelo. En una segunda etapa de render se utiliza la información del mapa de fotones para realizar el renderizado de manera más eficiente.

Este método fue propuesto por Jensen en 1996 [42]. En una primera etapa, un elevado número de fotones son lanzados desde las fuentes de luz, dividiendo la energía de las fuentes de luz entre los fotones emitidos (cada fotón transporta una fracción de la energía de la fuente de luz).

Cuando se emite un fotón, éste es trazado a través de la escena de igual forma que se lanzan los rayos en el método de Raytracing, excepto por el hecho de que los fotones propagan *flux* en lugar de radiancia. Cuando un fotón choca con un objeto puede ser reflejado, transmitido, o absorbido, en función de las propiedades del material.

Los fotones son almacenados cuando impactan sobre superficies no especulares. Este almacenamiento se realiza sobre una estructura de datos que está desacoplada de la geometría del modelo, y se utilizará en la etapa de render para obtener información sobre qué impactos de fotones están más cerca del punto del que queremos calcular su valor de iluminación.

2.3.3. Análisis de sistemas existentes

En este apartado se realizará un estudio del distinto *software* existente en lo relativo a síntesis de imagen 3D, como por ejemplo motores de renderizado y *suites* de modelado, iluminación, renderizado, etc.

2.3.3.1. *Yafray: Yet Another Free Raytracer*

Yafray [29] es un poderoso *raytracer*, liberado bajo licencia LGPL. *Yafray* permite al usuario crear imágenes y animaciones de calidad fotográfica, y utiliza XML como lenguaje de descripción de escenas. *Yafray* está integrado como motor de renderizado en el proyecto *Blender*. Sus principales desarrolladores son Alejandro Conty Estévez y Alfredo de Greef. Las principales características de *Yafray* son las siguientes:

- Iluminación global completa: *Yafray* puede iluminar escenas mediante un sistema de iluminación global completa, empleando aproximaciones basadas en el método de Mon-

te Carlo.

- Iluminación *Skydome*: este sistema de iluminación está basado principalmente en que la luz proviene de un cielo emisor, causando sombras suaves en la escena.
- Iluminación HDRI: este sistema de iluminación está basado en la información contenida en una imagen HDR. Este tipo de iluminación puede utilizarse en conjunto con alguno de los anteriores métodos.
- Caústicas: *Yafray* permite simular el comportamiento de los materiales reflectantes y transmisores de luz, como por ejemplo el cristal.
- DOF real: gracias a este sistema es posible reproducir en la escena el efecto de enfoque que tendría una lente real, de forma que se produce un desenfoque en los objetos más lejanos y cercanos del punto de enfoque.
- Reflexiones borrosas: ciertos objetos producen una distorsión en la reflexión debida a una microscopia superficie rugosa.

Además, *Yafray* mantiene una estructura modular, se puede integrar en programas de modelado 3D mediante un *plug-in*, es multiplataforma, mantiene un motor de render independiente, y permite realizar un renderizado distribuido y multihilo.

2.3.3.2. *POV-Ray*

POV-Ray (*Persistance of Vision Ray-tracer*) es un *raytracer* gratuito, en constante evolución, y que funciona en una gran variedad de plataformas, como por ejemplo en entornos Windows, Mac OS X, y Linux. A pesar de no ser un programa libre, su código fuente está disponible bajo las condiciones de la licencia *POV-Ray*.

POV-Ray es capaz de interpretar ficheros de código ASCII de extensión *.pov* en los que se describe la escena. Dicha descripción ha de contener la posición y los parámetros de las fuentes de luz, la cámara, y los objetos. Además, *POV-Ray* soporta un ligero lenguaje de programación que permite realizar operaciones más avanzadas con los distintos objetos de la escena.

2.3.3.3. *Toxic*

Toxic [24] es un motor de render de iluminación global disponible bajo licencia GPL, ideado con el objetivo de producir imágenes y animaciones de calidad fotográfica. La principal característica de *Toxic* es la corrección física. Desde la especificación de la escena hasta el renderizado final, todas las cantidades se expresan en unidades físicas. *Toxic* está construido utilizando técnicas geométricas avanzadas, debido a que la corrección física implica usar algoritmos que requieren una gran cantidad de procesamiento.

2.3.3.4. *Indigo*

Indigo [19] es un *raytracer* gratis (*freeware*), disponible originalmente para plataformas Windows, aunque también se puede emular en Linux. Utiliza el método *pathtracing* bidireccional y, aunque obtiene resultados muy buenos, es excesivamente lento. Al igual que en otros *raytracers*, la escena está especificada utilizando XML.

2.3.3.5. *Mental Ray*

Mental Ray permite generar imágenes de gran calidad y puede desplegarse en redes o en sistemas multi-procesador. Este *software* emplea técnicas propietarias para acelerar el proceso de renderizado, utiliza su propio lenguaje de descripción de escenas, y soporta una gran variedad de primitivas geométricas, como por ejemplo polígonos, superficies asimétricas, pelo, y división de superficies. Además, permite al usuario incluir código en C y C++ para la creación de texturas, materiales, modelos de iluminación, efectos de volumen, etc.

2.3.3.6. *Brazil*

Brazil [23] es una arquitectura de render basada en *plugins* e independiente de la plataforma empleada. Entre sus características principales destacan su escalabilidad y su adaptabilidad a distintos entornos de desarrollo 3D, y la incorporación de un sistema de iluminación global completo. *Brazil* tiene una licencia comercial, y su última versión (2.0) fue lanzada en el año 2005.

2.3.3.7. *Blender*

Blender es *software* libre multiplataforma, cuyas funciones se centran en el modelado y la creación de gráficos tridimensionales [16]. Su creador es Ton Roosendaal, creador de la fundación *Blender*.

Las principales características de *Blender* son:

- Es multiplataforma, libre, gratuito, y con un tamaño mucho menor que otros entornos de desarrollo 3D.
- Tiene capacidad para una gran variedad de primitivas geométricas, incluyendo curvas, mallas poligonales, vacíos, NURBS, *metabolls*, etc.
- Junto con la herramienta de animación se incluyen funciones como la cinemática inversa, deformaciones por armadura o cuadrícula, vértices de carga, y partículas estáticas y dinámicas.
- Edición de audio y sincronización de vídeo.
- Características interactivas para juegos, como por ejemplo detección de colisiones, recreaciones dinámicas, y lógica.
- Posibilidad de realizar el renderizado con su motor interno o con el motor de renderizado *Yafray*.
- Utilización del lenguaje de *script Python* para automatizar o controlar tareas.
- Aceptación de una gran variedad de formatos gráficos.
- Motor de juegos 3D integrado, con un sistema de ladrillos lógicos.
- Simulaciones dinámicas para *softbodies*, partículas y fluidos.
- Modificadores apilables para la aplicación de transformación no destructiva sobre mallas.
- Sistema de partículas estáticas para simular cabellos y pelajes, permitiendo opciones de *shaders* para lograr texturas realistas.

2.3.3.8. 3D Studio Max

3D Studio Max es un entorno de desarrollo 3D utilizado para el desarrollo de videojuegos comerciales. Conocido también como *3D Max* o *Max*, este programa dispone de una sólida capacidad de edición, una arquitectura basada en *plugins*, y muy buen soporte para la captura de movimiento.

Las principales características de *3D Studio Max* son:

- Motor de IK robusto, facilitando la captura de movimiento.
- Es un *software* relativamente barato.
- Gran cantidad de *plugins* en el mercado.
- Gran cantidad de bibliografía, al ser uno de los entornos más utilizados.
- Interfaz de usuario confusa y nada modular.
- Mala gestión de restricciones.
- Lenguaje de *script* algo confuso.

2.3.3.9. Maya

Maya es un *software* de creación de gráficos 3D, efectos especiales, y animación. Surgió del desarrollo de *Power Animator* y de la fusión de *Alias* y *Wavefront*. *Maya* se caracteriza por tener el mejor sistema de *script*: MEL (*Maya Embedded Language*).

En *Maya todo es un nodo, todo nodo puede ser enlazado con otro nodo, y todo nodo es animable*. *Maya* se jacta de ser el sistema de modelado 3D más completo del mercado, y es ampliamente usado en el ámbito comercial.

2.4. Análisis de tecnologías multiplataforma

2.4.1. Procesamiento de imagen

2.4.1.1. Python Imaging Library (PIL)

PIL (***Python Imaging Library***) es una biblioteca de funciones que proporciona capacidades para procesar imágenes desde el intérprete Python. Esta biblioteca tiene soporte para distintos tipos de archivos, mantiene una representación interna eficiente, y ofrece potentes funcionas para el tratamiento y composición de imágenes.

PYL contiene algunas funcionalidades básicas para el procesamiento de imágenes, incluyendo operaciones de puntos, manejo de filtros, y conversiones entre colores. Además, existen funciones básica para redimensionar imágenes, rotarlas, y realizar otro tipo de transformaciones. Inluso existe un método para obtener estadísticas de la imagen, con el objetivo de usarlas para automatizar el tratamiento del contraste y de analizar estadísticas globales.

La clase más importante en PYL es la clase *Image*, definida en el módulo con el mismo nombre. Dicha clase puede ser instanciada de varias formas: cargando imágenes a partir de archivos, a partir del procesamiento de imágenes, e incluso seleccionando trozos de las imágenes. Para cargar una imagen a partir de un archivo se utiliza la función *open* del módulo *Image*:

```
>>> import Image
>>> im = Image.open('imagen.png')
```

Si la apertura resultó exitosa la función devuelve un objeto del tipo *Image*. A partir de este objeto se pueden obtener los atributos principales de la imagen:

```
>>> print im.format, im.size, im.mode
PNG (640, 480) RGB
```

El atributo formato identifica la fuente del archivo, el atributo tamaño devuelve una tupla con las dimensiones de la imagen, y el atributo modo define el número y los nombres de las bandas de la imagen junto con el tipo de *pixel* y la profundidad. Los modos más comunes son L para imágenes en escala de grises, RGB para imágenes con color verdadero, y CMYK para imágenes del tipo *pre-press*.

Si un fichero no puede abrirse, se lanza una excepción del tipo *IOError*. Una vez que se ha obtenido una instancia de la clase Image se pueden emplear las funciones definidas para procesar y manipular imágenes. Por ejemplo, se puede emplear la función *show* para mostrar la imagen cargada:

```
>>> im.show()
```

PYL soporta una amplia variedad de formatos de imágenes. Para leer ficheros de disco se utiliza la función *open* del módulo *Image*. No es necesario conocer el formato de un archivo para abrirlo, ya que PYL determina de manera automática el formato basándose en el contenido del archivo. Para guardar un fichero se utiliza el método *save* del módulo *Image*. Cuando se guarda un fichero el nombre del mismo resulta importante, ya que si no se especifica el formato PYL utiliza la extensión de dicho nombre para determinar qué formato utilizar.

Un ejemplo de conversión de archivos a JPEG es el siguiente:

```
import os, sys
import Image

for infile in sys.argv[1:]:
   outfile = os.path.splitext(infile[0] + '.jpg')
   if infile != outfile:
      try:
         Image.open(infile).save(outfile)
      except IOError:
         print 'No se pudo convertir ' + infile
```

La clase *Image* contiene funciones para manipular regiones dentro de una imagen. Para extraer una porción de una imagen se utiliza la función *crop*:

```
box = (100, 100, 400, 400)
region = im.crop(box)
```

Las regiones se definen con tuplas de cuatro elementos, donde las coordenadas son (izquierda, arriba, derecha, abajo). PYL utiliza un sistema de coordenadas con (0, 0) en la esquina superior izquierda. La región anterior podría ser ahora procesada de una determinada manera y vuelta a pegar:

```
region = region.transpose(Image.ROTATE_180)
im.paste(region, box)
```

Cuando se copian regiones, el tamaño de la región debe coincidir con el de la región determinada anteriormente. Además, dicha región no puede extenderse más allá de las dimensiones de la imagen. Sin embargo, los modos de la imagen original y de la región no tienen por qué coincidir. Un ejemplo es el siguiente:

```
def roll (image, delta):

   xsize, ysize = image.size

   delta = delta % xsize
   if delta == 0:
      return image

   part1 = image.crop((0, 0, delta, ysize))
   part2 = image.crop((delta, 0 xsize, ysize))
   image.paste(part2, (0, 0, xsize - delta, ysize))
   image.paste(part1, (xsize - delta, 0, xsize, ysize))

   return image
```

Para cuestiones más avanzadas, la función *paste* puede tomar una máscara de transparencia como un argumento opcional. En esta máscara, el valor 255 indica que la imagen pegada es opaca en esa posición. El valor 0 indica que imagen pegada es completamente transparente. Los valores intermedios indican diferentes niveles de transparencia.

PYL también permite trabajar con bandas individuales de una imagen multi-banda, como las imágenes RGB. La función *split* crea un conjunto de nuevas imágenes, donde cada una contiene una banda de la imagen multi-banda. La función *merge* toma como parámetros el modo y una tupla de imágenes, y las combina en una nueva imagen. Un ejemplo es el siguiente:

```
r, g, b = im.split()
im = Image.merge('RGB', (b, g, r))
```

La clase *Image* también contiene funciones para redimensionar y rotar imágenes. El primero toma una tupla como argumento definiendo el nuevo tamaño, mientras que el último toma el ángulo en grados en sentido opuesto de las agujas del reloj:

```
out = im.resize((128, 128))
out = im.rotate(45)
```

Función	Significado
Image.new(mode, size, color) ==>image	Crea una nueva imagen
Image.open(infile, mode) ==>image	Abre e identifica la imagen dada
Image.blend(image1, image2, alpha) ==>image	Crea una nueva imagen interpolando las imágenes dadas
Image.composite(image1, image2, mask) ==>image	Crea una nueva imagen interpolando las imágenes dadas
Image.eval(function, image) ==>image	Aplica la función a cada píxel de la imagen
Image.fromstring(mode, size, data, decode, parameters) ==>image	Crea una imagen en memoria a partir de data
Image.merge(mode, bands) ==>image	Crea una nueva imagen a partir de un número de bandas
im.convert(mode, matrix) ==>image	Devuelve una copia convertida de la imagen
im.copy(image) ==>image	Copia la imagen
im.crop(image) ==>image	Devuelve una región rectangular de la imagen actual
im.draft(mode, size) ==>image	Devuelve una versión de la imagen lo más cercana posible a los parámetros mode y size
im.filter(filter) ==>image	Devuelve una copia de la imagen con un filtro aplicado
im.fromstring(data, decoder, parameters) ==>image	Idéntica a la función fromstring pero relativa a la imagen actual
im.getbands() ==>tuple of strings	Devuelve una tupla que contiene el nombre de cada banda
im.getbox() ==>4-tuple or None	Calcula los perímetros de las regiones de la imagen
im.getdata() ==>sequence	Devuelve el contenido de la imagen como una secuencia de objetos de tipo píxel
im.getextrema() ==>2-tuple	Devuelve una tupla con los valores mínimo y máximo de la imagen
im.getpixel(xy) ==>value or tuple	Devuelve el valor del píxel de la posición dada
im.histogram(mask) ==>list	Devuelve un histograma de la imagen
im.load() ==>list	Carga una imagen de memoria
im.offset(xoffset, yoffset) ==>image	Devuelve una copia de la imagen en la que los datos han sido desplazados
im.paste(image, box)	Copia otra imagen en la imagen actual
im.paste(image, box, mask)	Copia otra imagen en la imagen actual
im.point(table) ==>image	Devuelve una copia de la imagen en la que cada píxel es mapeado con la tabla dada
im.putalpha(band)	Copia la banda dada a la imagen actual
im.putdata(data, scale, offset)	Copia los valores de los píxeles en la imagen actual
im.putpalette(sequence)	Asocia una paleta a una imagen 'P' o 'L'
im.putpixel(xy, color)	Modifica el píxel de la posición dada
im.resize(size, filter) ==>image	Devuelve una copia redimensionada de la imagen actual
im.rotate(angle, filter) ==>image	Devuelve una copia rotada de la imagen actual
im.save(outfile, format, options)	Almacena una copia de la imagen
im.seek(frame)	Busca el frame dado en un archivo de secuencia
im.show()	Muestra la imagen
im.split() ==>sequence	Devuelve una tupla con las bandas individuales de la imagen
im.tell() ==>integer	Devuelve el número del frame actual
im.thumbnail(size, filter)	Modifica la imagen para contener una versión parecida de la propia imagen
im.tobitmap() ==>string	Devuelve la imagen convertida a un mapa de bits X11
im.tostring(encoder, parameters) ==>string	Devuelve una cadena conteniendo los datos de los píxeles
im.transform(size, method, data, filter) ==>image	Crea una nueva imagen con los parámetros dados
im.transpose(method) ==>image	Devuelve una copia de la imagen según el método aplicado
im.verify()	Trata de determinar si la imagen es correcta

Cuadro 2.4: Funciones de la clase *Image*.

2.4.2. Lenguaje de intercambio de información

2.4.2.1. Document Object Model (DOM)

DOM (***Document Object Model***) es una interfaz de programación de aplicaciones para documentos HTML y XML. Define la estructura lógica de los documentos y la forma en la que se accede y se manipula el documento. En la especificación DOM, el término *documento* es usado en el sentido amplio de la palabra, ya que lenguajes como XML extienden la representación de distintos tipos de información que puede almacenarse en distintos sistemas, de manera que el concepto de documento se extiende al concepto de *datos*.

A través de DOM, un programador puede construir documentos, navegar por su estructura, y añadir, modificar, o eliminar elementos y contenidos. Cualquier cosa incluida en un documento HTML o XML puede ser accedida, actualizada, eliminada, o añadida utilizando DOM.

Como especificación del *World Wide Web Consortium*, un objetivo importante de DOM es proporcionar un API que pueda usarse en una gran variedad de entornos y aplicaciones. De hecho, DOM está diseñado para usarse con cualquier lenguaje de programación. Con la idea de proporcionar una especificación precisa e independiente del lenguaje para las interfaces de DOM, se elige la especificación 2.2 de CORBA para la definición de las interfaces.

DOM está asociado a la estructura de los documentos que modela. Como ejemplo, podemos considerar la siguiente tabla tomada de un documento HTML:

```
<TABLE>
 <TBODY>
  <TR>
   <TD>Shady Grove</TD>
   <TD>Aeolian</TD>
  </TR>
  <TR>
   <TD>Over the river, Charlie</TD>
   <TD>Dorian</TD>
  </TR>
 </TBODY>
</TABLE>
```

Empleando DOM los documentos tienen una estructura lógica parecida a la de un árbol; aunque, para ser más preciso, es más cercana a un *bosque* que contiene varios árboles.

Existen distintas implementaciones de DOM para distintos lenguajes de programación, pero una de las más importantes es la implementación para C++ basada en la recomendación Apache para C++, cuyo diseño ofrece las siguientes características:

- Buena gestión de memoria.
- Rapidez.
- Gran escalabilidad en sistemas multiprocesador.
- Más cercano a C++ y menos cercano a Java.

El proyecto en cuestión se denomina *Xerces-C++* y ofrece un analizador sintáctico con validación para XML escrito en un subconjunto portable de C++. Para acceder al API desde el código de la aplicación es necesario incluir la siguiente sentencia:

```
#include <xercesc/dom/DOM.hpp>
```

El fichero de cabecera DOM.hpp incluye todas las cabeceras individuales de las distintas clases del API DOM.

Los nombres de las clases de DOM utilizan el prefijo DOM, de manera que se eviten conflictos de nomenclatura por el resto de clases definidas por el usuario:

```
DOMDocument* myDocument;
DOMNode* aNode;
DOMText* someText;
```

Las aplicaciones normalmente usan punteros para acceder directamente a los distintos objetos empleando DOM:

```
DOMNode* aNode;
DOMNode* docRootNode;

aNode = someDocument->createElement(anElementName);
docRootNode = someDocument->getDocumentElement();
docRootNode->appendChild(aNode);
```

Con el objetivo de utilizar Xerces-C++ para realizar un análisis sintáctico utilizando DOM se utiliza la clase XercesDOMParser. Un sencillo ejemplo es el siguiente:

```
#include <xercesc/parsers/XercesDOMParser.hpp>
#include ....

int main (int argc, char *args[]) {

  XMLPlatformUtils::Initialize();

  XercesDOMParser* parser = new XercesDOMParser();
  char* xmlFile = 'x1.xml';
  try {
    parser->parse(xmlFile);
  }
  catch (...) {
    ...
  }

  delete parser;
  return 0;

}
```

Además de la anterior implementación de DOM, existen implementaciones más ligeras, como por ejemplo el módulo xml.dom.minidom para Python. El objetivo principal de este módulo es ofrecer una implementación más sencilla y significativamente más pequeña que la definida por DOM.

Las aplicaciones relacionadas con DOM comienzan típicamente con el análisis de algún documento XML. En el entorno minidom se utiliza la función *parse*:

```
from xml.dom.minidom import parse, parseString

dom1 = parse('test.xml')
datasource = open('test.xml')
dom2 = parse(datasource)
dom3 = parseString('<myxml> Some data<empty/> some more data </myxml>')
```

Ambas funciones devuelven un objeto *Document* que representa el contenido del documento. Lo que realmente hacen las funciones *parse* y *parseString* es conectar un analizador XML con un constructor DOM que puede aceptar eventos relativos al análisis de cualquier analizador SAX y convertirlos en un árbol DOM.

Un ejemplo más elaborado de creación de documentos según DOM es el siguiente:

```
from xml.dom.minidom
from xml.dom.ext.c14n import Canonicalize

    def createFIPAACLMessageXML (self):
```

```
        aclMessage = xml.dom.minidom.Document()

        rootElem = aclMessage.createElement('fipa-message')

        rootElem.setAttribute('act', self.getPerformative())
        aclMessage.appendChild(rootElem)

        senderElem = aclMessage.createElement('sender')
        rootElem.appendChild(senderElem)
        agentIdentifierElem = aclMessage.createElement('agent-identifier')
        senderElem.appendChild(agentIdentifierElem)

        nameElem = aclMessage.createElement('name')
        agentIdentifierElem.appendChild(nameElem)
        nameElem.setAttribute('id', self.getFrom().Name)

        addressesElem = aclMessage.createElement('addresses')
        agentIdentifierElem.appendChild(addressesElem)

        for i in range(len(self.getFrom().Addresses)):
            urlElem = aclMessage.createElement('url')
            addressesElem.appendChild(urlElem)
            urlElem.setAttribute('href', self.getFrom().Addresses[i])

....
....

        return 0, Canonicalize(aclMessage)
```

2.4.2.2. Extensible Markup Language (XML)

XML (***Extensible Markup Language***) describe una clase de objetos de datos denominados documentos XML y, parcialmente, también describe el comportamiento de los programas de ordenador que los procesan. XML conforma un parte de SGML (*Standard Generalized Markup Language*).

Existen dos tipos de documentos XML: válidos y bien formados. Los documentos bien formados son aquellos documentos que cumplen todas las especificaciones del lenguaje en cuanto a reglas sintácticas se refiere, sin estar fijados a unos elementos fijados en un DTD. Además, los documentos XML deben tener una estructura jerárquica muy estricta que los documentos bien formados han de cumplir. Por otra parte, los documentos válidos son documentos bien formados que siguen una estructura y una semántica determinada por un DTD, es decir, sus elementos y su estructura jerárquica deben ajustarse a lo que define el DTD.

Un DTD (*Document Type Definition*) constituye una definición de los elementos que conforman un documento XML, junto con su relación entre ellos, sus atributos, sus valores, etc. Visto de otro modo, el DTD constituye la definición de la gramática asociada al documento XML.

Un ejemplo de documento XML es el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>

<agent name="Mulder">

 <agentDescription>

  <service name="render" type="render" />
  <protocol value="fipa-request" />
  <ontology value="fipa-agent-management" />
  <language value="fipa-rdf0" />
  <leaseTime value="86400" />
  <scope value="global" />

 </agentDescription>

</agent>
```

La primera línea indica la versión de XML empleada en el documento, junto con la codificación del mismo. Además, también es posible incluir un parámetro *standalone* que indica si el documento va acompañado o no de un DTD.

En cuanto a la sintaxis del documento, es importante detallar ciertos detalles:

- Toda etiqueta no vacía ha de tener su etiqueta de cerrado asociada, es decir, <etiqueta>ha de estar seguida de <etiqueta>.
- Todos los elementos ha de estar perfectamente anidados, es decir, no es válido <ficha><nombre>Adolfo <ficha><nombre>
- Los elementos vacíos son aquellos que no tienen contenido dentro del documento. La sintaxis correcta para estos documentos implica que la etiqueta siempre tenga la forma <etiqueta/>.

Si un DTD acompaña a un documento XML, existen varias formas de referenciarlo. Por un lado, se puede incluir dentro del documento XML una referencia al documento DTD en forma de URI, empleando la siguiente sintaxis:

```
<!DOCTYPE ficha SYSTEM
"http://www.dat.etsit.upm.es/~abarbero/DTD/ficha.dtd">
```

En este ejemplo, la palabra SYSTEM indica que el DTD se obtendrá a partir de un documento externo al documento, e indicado por el URI que lo sigue. Por otra parte, también es posible incluir el DTD dentro del propio documento XML:

```
<?xml version="1.0"?>
 <!DOCTYPE ficha [
  <!ELEMENT ficha (nombre+, apellido+, direccion+, foto?)>
  <!ELEMENT nombre (#PCDATA)>
   <!ATTLIST nombre sexo (masculino|femenino) #IMPLIED>
  <!ELEMENT apellido (#PCDATA)>
  <!ELEMENT direccion (#PCDATA)>
  <!ELEMENT foto EMPTY>
]>
<ficha>
 <nombre>David</nombre>
 <apellido>Vallejo</apellido>
 <direccion>C/Dorada, 32</direccion>
</ficha>
```

En lo relativo a la definición de los elementos se utiliza la claúsula <!ELEMENT, y después distintos parámetros en función del elemento. Entre paréntesis, siempre que el elemento no sea vacío, se indica el posible contenido del elemento: la lista de elementos hijos o descendientes, separados por comas, o el tipo de contenido, que normalmente es #PCDATA que indica datos de tipo texto. Además, si el elemento es vacío, se indica con la claúsula EMPTY.

A la hora de especificar los elementos descendientes (entre paréntesis) se utilizan distintos caracteres especiales, que sirven para indicar qué tipo de uso se permite hacer de esos elementos dentro del documento:

- +: uso obligatorio y múltiple.
- *: uso opcional y múltiple.
- ?: uso opcional y singular.
- —: uso opcional de un elemento.

Para la definición de atributos se utiliza la declaración <!ATTLIST, seguida de:

- El nombre del elemento del que se declaran los atributos.

- El nombre del atributo.
- Los posibles valores del atributo, entre paréntesis y separados por el carácter —.
- De forma opcional y entre comillas el valor por defecto del atributo.
- La claúsula #REQUIRED si es obligatorio declarar el atributo.

Como se comentó anteriormente, las entidades o *entities* son los elementos que proporcionan la modularidad de los documentos XML. Al igual que los DTDs, se pueden definir de forma interna o externa al documento XML. Por ejemplo, se puede definir una entidad que referencia a un nombre largo:

```
<!ENTITY IEEE 'Institute of Electrical and Electronics Engineers'>
```

De esta forma, cada vez que se quiera que aparezca el nombre 'Institute of Electrical and Electronics Engineers' sólo será necesario escribir &IEEE.

Otra característica importante de XML es la posibilidad de incorporar hojas de estilo a través de la tecnología **XSL** (*eXtensible Style Language*). XSL permite describir cómo la información contenida en un documento XML cualquiera debe ser transformada o formateada para su presentación en un medio específico. La familia de lenguajes XSL está formada por tres lenguajes:

- XSLT (*eXtensible Stylesheet Language Transformations*), que permite convertir documentos XML de una sintaxis a otra.
- XSL-FO (*eXtensible Stylesheet Language Formatting Objects*), que permite especificar el formato visual con el cual se quiere presentar un documento XML.
- XPath, que es una sintaxis para acceder o referirse a porciones de un documento XML.

Capítulo 3

Objetivos del proyecto e hipótesis del trabajo

Con este proyecto se pretende distribuir el trabajo asociado a la etapa de renderizado entre distintos agentes *inteligentes*, utilizando para ello un sistema multi-agente de propósito general. Bajo este enfoque se pretenden alcanzar los siguientes objetivos generales:

- Construir la arquitectura básica de un sistema multi-agente, siguiendo las guías del comité de estándares FIPA [18].
- Utilizar dicha arquitectura para llevar a cabo el proceso de renderizado de manera distribuida.

El hecho de utilizar una arquitectura multi-agente facilita el planteamiento de la solución del problema comentado, ya que la naturaleza de dicho problema es inherentemente distribuida. Además, la reutilización de dicha arquitectura supone una gran ventaja a la hora de abordar otros problemas de distinta naturaleza pero de índole distribuida.

Para lograr estos objetivos más generales se alcanzarán los siguientes subobjetivos:

- Reducir el tiempo empleado en el proceso de renderizado.
- Utilización de técnicas de estudio de la escena, previas al renderizado.
- Utilización de técnicas de *soft-computing* para ajustar los parámetros del renderizado final.

- Construcción de un sistema escalable a otras etapas del proceso de síntesis.
- Asegurar la portabilidad entre los principales sistemas operativos existentes.
- Uso de herramientas y tecnologías libres que aseguren la portabilidad.
- Empleo de estándares para la construcción del sistema.

Bajo este conjunto de objetivos, el objetivo principal consiste en realizar una primera aproximación a un sistema que optimice el proceso de renderizado en lo que a tiempo de ejecución se refiere, obteniendo parámetros de calidad similares a los que se obtendrían con los parámetros predefinidos por el usuario.

El hecho de realizar un estudio previo de la escena permite obtener un conocimiento que posteriormente será empleado para abordar el problema de *rendering* distribuido de una forma más correcta. Además, el uso de técnicas de *soft-computing* permite ajustar los parámetros del renderizado final, utilizando para ello conocimiento experto.

En cuanto a la escalabilidad del sistema, ésta se logra gracias al enfoque elegido, es decir, gracias a la elección de un sistema multi-agente como esqueleto de la aplicación. Por otra parte, el uso de tecnologías libres y de estándares asegura la portabilidad entre distintas arquitecturas y sistemas operativos.

Como se ha comentado anteriormente, toda la lógica de la aplicación descansará bajo al arquitectura de un sistema multi-agente. La implementación inicial de dicha arquitectura se realizará utilizando el lenguaje de programación C++, debido a su portabilidad entre distintas plataformas y a su eficiencia, motivos que lo hacen adecuado para este tipo de proyectos. Debido a la interacción con la *suite* de modelado Blender [16] (aplicación junto con la que se realizará el desarrollo de las pruebas), cuya API está hecha utilizando el lenguaje Python, también se realizará una implementación del agente en dicho lenguaje de programación, con el objetivo de facilitar la integración.

Las pruebas de renderizado se realizarán con el motor Yafray [29], aunque se preparará el entorno para la posibilidad de interactuar con otros motores de renderizado.

En cuanto al intercambio de datos entre distintos componentes y a la necesidad de los mismos para llevar a cabo la inicialización de servicios y agentes, se utilizará el metalenguaje

XML. Esta elección se justifica mediante la sencillez, escalabilidad, e interoperabilidad que ofrece XML.

En lo relativo al *middleware* de comunicaciones se utilizará ZeroC Ice, elección que se justificó anteriormente y que, básicamente, se debe al hecho de que Ice es multiplataforma, multilenguaje, sencillo de aprender y usar, y se distribuye bajo una licencia GPL.

En un principio, el entorno de desarrollo será el sistema operativo GNU/Linux. Sin embargo, y debido a que todas las tecnologías empleadas son multiplataforma, la portabilidad con respecto a otros sistemas operativos será inmediata. En cuanto al equipo físico, el sistema se podrá ejecutar utilizando para ello cualquier máquina que tenga una conexión a Internet e instalados todas las herramientas anteriormente mencionadas. De hecho, lo que se pretende con sucesivas versiones del proyecto es alcanzar un entorno similar al proyecto *Seti@Home* [14], de forma que haya un número indeterminado de agentes ejecutándose en máquinas de Internet contribuyendo al renderizado distribuido de numerosos proyectos, de forma transparente a las distintas dificultades que puedan plantear diversos entornos de red hostiles.

Capítulo 4

Metodología de trabajo

La metodología de trabajo está condicionada por la naturaleza de MASYRO, ya que el método escogido para realizar el *rendering* distribuido por Internet está basado en el uso de una arquitectura multi-agente. Es importante destacar que este trabajo está dividido en dos partes completamente independientes. Por un lado, se ha construido una arquitectura multi-agente genérica, la cual se puede utilizar en cualquier aplicación que se desee, y que además garantiza la interoperabilidad gracias a que es compatible con el conjunto de estándares definidos por FIPA. Por otro lado, se ha realizado una instanciación de dicha arquitectura multi-agente, añadiendo la funcionalidad necesaria, es decir, los servicios y agentes necesarios para

tal efecto, con el objetivo de proporcionar un sistema dedicado al *rendering* distribuido por Internet.

Una vez comentada esta importante cuestión, este capítulo se dividirá en dos apartados principales:

- Construcción de la arquitectura básica del sistema multi-agente compatible con el conjunto de estándares FIPA.
- Instanciación de la arquitectura genérica multi-agente para llevar a cabo el renderizado distribuido con optimizaciones basadas en conocimiento experto.

En ambos apartados se comentarán los pasos necesarios para solucionar los problemas propuestos, haciendo especial hincapié en los mecanismos empleados en lugar de incluir grandes cantidades de código.

4.1. Construcción de la arquitectura básica del sistema multi-agente

4.1.1. Definición de los servicios básicos

De acuerdo con el estudio previo que se hizo en el capítulo dos, una plataforma de agentes según FIPA [18] está formada por un conjunto de agentes y por un cierto número de servicios básicos (ver figura 4.1).

El primer paso que hubo que dar fue definir las interfaces básicas de estos servicios y de un agente (ver figura 4.2). Estas interfaces suponen el contrato entre el ente que actúa como servidor y el ente que actúa como cliente. No obstante, suele darse la situación en la que un mismo ente actúa como cliente-servidor. A continuación se irán estudiando las interfaces definidas para cada uno de los elementos anteriores, pero antes es necesario solucionar un problema básico. Cuando un agente se vincula a una plataforma de agentes, es necesario que dicha plataforma ofrezca un servicio que permita al agente obtener una referencia a los servicios básicos. FIPA no impone un mecanismo concreto para desarrollar este servicio, por lo que en este proyecto se ha decidido incorporar un servicio muy simple y escalable, que

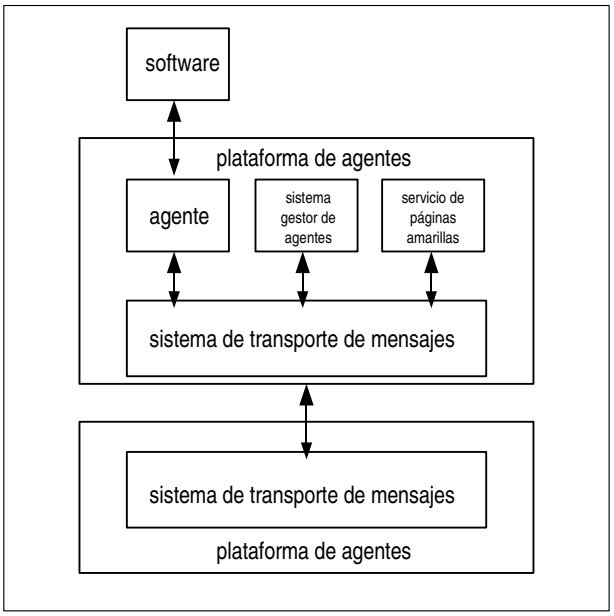

Figura 4.1: Modelo de referencia para la gestión de agentes

permita que un agente conozca los servicios básicos de una plataforma cuando comienza su ejecución en la plataforma de agentes. La siguiente interfaz muestra la funcionalidad de este servicio básico, cuyo nombre es ***StartService***:

```
interface StartService
{
   nonmutating TServiceDirectoryEntries getServiceRoot();
   void supplyBasicService(TServiceDirectoryEntry sde);
};
```

Como se puede apreciar, la interfaz consta de dos operaciones:

- La operación *getServiceRoot* permite obtener un conjunto de entradas que definen a los servicios básicos, y que viene determinado por el parámetro de retorno de la operación. El parámetro *TServiceDirectoryEntries* se corresponde con una secuencia de entradas, cada una de ellas a los distintos servicios básicos que se registraron en el servicio *StartService*.

- La operación *supplyBasicService* permite que un servicio básico notifique al servicio *StartService* que en realidad es un servicio básico, y que necesita ser conocido por un

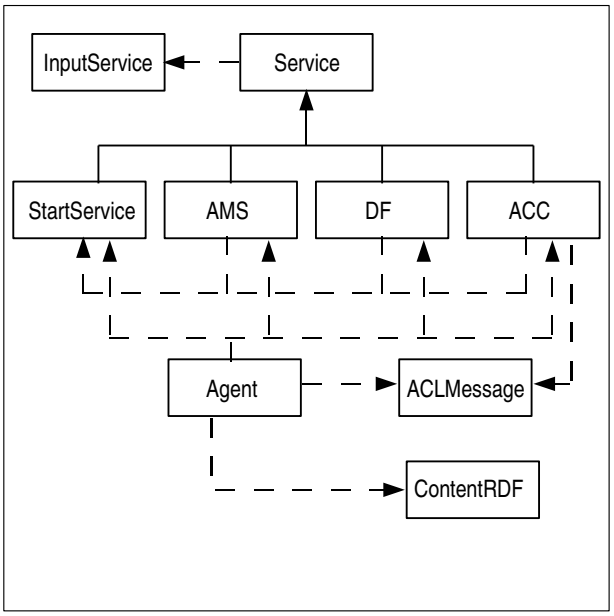

Figura 4.2: Diagrama de clases del sistema multi-agente

agente cuando éste comienza su ejecución. El parámetro que se le pasa a la operación consiste en la entrada que describe el servicio básico.

Un servicio básico queda definido con la siguiente estructura:

```
struct TServiceDirectoryEntry
{
   string ServiceType;
   TServiceLocator ServiceLocator;
   string ServiceId;
};
```

El parámetro más interesante es *ServiceLocator*, que se corresponde con una secuencia de direcciones de transporte a las que responde el servicio en cuestión.

Con este sencillo mecanismo, se puede añadir cualquier servicio básico a la plataforma de agentes de forma escalable, con el objetivo de que un agente o un servicio sea capaz de obtener los servicios básicos incluidos en dicha plataforma. Una vez que se ha solucionado este primer problema, el siguiente paso consiste en definir los distintos servicios básicos de una plataforma de agentes. Para ello es necesario especificar la funcionalidad que dichos servicios proporcionan, es decir, la interfaz que ofrecen al exterior.

El principal servicio, y que además actúa como gestor, de la plataforma de agentes es el ***Agent Management System***. Este elemento es el encargado de controlar todo lo que ocurre en la plataforma de agentes, ofreciendo un servicio de páginas blancas a los agentes y manteniendo un control sobre cada uno de los estados en los que se encuentra un agente. A continuación, se expone la interfaz de este servicio con la funcionalidad que proporciona, de manera que se irán estudiando una por una las operaciones de las que se compone dicha interfaz.

```
interface AMS
{
   void register(TAID aid,
         out int explanation, out string newName, out int state);
   void deregister(TAID aid, out int explanation);
   idempotent void modify(TAID aid, out int explanation);
   nonmutating void search(TAID aid, int match,
         out int explanation, out TAIDs aids);
   nonmutating string getDescription();
};
```

La primera operación definida en el interfaz, es decir, la operación *register*, permite que un agente se registre en la plataforma de agentes. Para que tal registro sea válido el identificador del agente debe ser único y verificado por el *Agent Management System*. Como se puede apreciar, el parámetro de entrada es una estructura del tipo TAID, que define el *agent identifier* o identificador de agente asociado al agente que intenta registrarse. Dicha estructura tiene la siguiente forma:

```
struct TAID
{
   string Name;
   Sstring Addresses;
};
```

El parámetro *Name* define el nombre del agente, mientras que el parámetro *Addresses* representa a una secuencia de direcciones de transporte en las que el agente puede ser contactado.

Tras un intento de registro, el *Agent Management System* devuelve un parámetro *explanation* que indica si el registro tuvo éxito o no, señalando en tal caso su motivo, un parámetro *state* que define el estado del agente tras el registro, y un parámetro *newName* en caso de

que exista un agente subscrito con el nombre con el que se intenta registrar el agente. En este caso, el *Agent Management System* es el encargado de asignar el nombre al agente.

Las operaciones *deregister* y *modify* permiten eliminar una subscripción y modificarla, respectivamente. Como se puede apreciar, es necesario proporcionar el *agent-identifier* del agente que solicita tal funcionalidad. Por otra parte, la operación *search* permite buscar agentes que cumplan unas determinadas convenciones de nombrado.

El *Agent Management System* mantiene una estructura del tipo *TAgentDirectory* con las descripciones de los agentes que mantienen una subscripción con dicho servicio:

```
struct TAMSAgentDescription
{
   TAID Name;
   int State;
};

sequence <TAMSAgentDescription> TAgentDirectory;
```

Por último, la operación *getDescription* devuelve una descripción de la plataforma de agentes.

El siguiente servicio básico es el denominado ***Directory Facilitator***, cuya principal función es proporcionar un servicio de páginas amarillas a los agentes. La interfaz de este servicio es parecida a la del *Agent Management System*:

```
interface DirectoryFacilitator
{
   void register(TDFAgentDescription ad, out int explanation);
   void deregister(TDFAgentDescription ad, out int explanation);
   idempotent void modify(TDFAgentDescription ad, out int explanation);
   nonmutating void search(TDFAgentDescription ad, int match,
         out int explanation, out TDFAgentDescriptions ads);
};
```

Como se puede observar, las operaciones representan la misma funcionalidad que sus homólogas en el servicio *Agent Management System*, pero con la diferencia de que la estructura de datos utilizada para representar a un agente es distinta:

```
struct TDFAgentDescription
{
   TAID Name;
   TDFServiceDescriptions Services;
   Sstring Protocols;
   Sstring Ontologies;
```

```
   Sstring Languages;
   int LeaseTime;
   Sstring Scope;
};

sequence <TDFAgentDescription> TDFAgentDescriptions;
```

En este caso, FIPA especifica parámetros para distinguir los servicios que proporciona un agente, la lista de protocolos de interacción que conoce, la lista de ontologías que maneja, la lista de lenguajes de contenido que soporta, el tiempo máximo de duración del registro, y la visibilidad de la descripción del agente en el DirectoryFacilitator.

Llegados a este punto, y con las correspondientes implementaciones subyacentes, ya están solucionados los problemas relativos a la representación de los servicios de páginas blancas y de páginas amarillas de la plataforma de agentes. Sin embargo, falta por definir el tercer servicio básico, el *Agent Communication Channel*. Dicho servicio es el encargado de recibir y enviar los distintos mensajes enviados y recibidos por los agentes. En realidad, la única funcionalidad que debe soportar este servicio es la capacidad de recibir mensajes, ya que de cara al envío de mensajes el agente incorporará una operación que permite recibir un mensaje, y que será utilizada por el sistema de transporte de correo para tal efecto. La interfaz que define al *Agent Communication Channel* es la siguiente:

```
interface ACC
{
   int receive(TMessage message);
};
```

La estructura *TMessage* se corresponde con un mensaje, y está dividida en dos partes:

- La carga útil del mensaje, es decir, el contenido del mismo que está especificado en un lenguaje de comunicación.
- La envoltura del mensaje, es decir, aquel elemento que define a quién va dirigido el mensaje y de quién proviene.

Siguiendo la analogía del correo tradicional, la envoltura del mensaje se correspondería con el tradicional sobre que se utiliza para enviar una carta, y la carga útil estaría vinculada a la carta en cuestión:

```
struct TEnvelope
{
   TAIDs To;
   TAID From;
   TDate Date;
   int ACLRepresentation;
};

struct TMessage
{
   TEnvelope Envelope;
   string Payload;
};
```

Con la definición de estos cuatro servicios, ya tenemos el esqueleto básico de una arquitectura acorde con las especificaciones FIPA. El siguiente paso consiste en especificar la arquitectura básica de un agente.

4.1.2. Definición del agente

Dentro de una plataforma de agentes, los agentes representan las unidades computacionales autónomas encargadas de llevar a cabo la funcionalidad en la que están especializados. Sin embargo, un agente también ha de soportar cierta funcionalidad de gestión, de forma independiente al cometido para el que fue diseñado. La siguiente interfaz especifica la funcionalidad mínima requerida para un agente:

```
interface Agent
{
   idempotent void suspend();
   void terminate();
   idempotent void resume();
   void receiveACLMessage(string ACLMessage);
};
```

Es importante destacar la operación *receiveACLMessage*, que permite a un agente recibir un mensaje. Aunque ya se comentó anteriormente, toda la información intercambiada entre agentes está especificada en XML, y los DTDs asociados en las especificaciones FIPA.

Una vez implementada la arquitectura básica de la arquitectura multi-agente, toda esta infraestructura se puede reutilizar para cualquier propósito. Para ello, basta con realizar especializaciones del agente ya implementado (en los lenguajes C++ y Python) de forma que desarrolle la funcionalidad que se requiera.

La arquitectura básica implementada se ha definido teniendo en cuenta las características de Internet, por lo que es posible que cualquier agente o servicio se pueden ejecutar en cualquier máquina, siendo solamente necesario ajustar los archivos de configuración apropiados. Para lograr este comportamiento, se han utilizado dos servicios avanzados que proporciona Ice:

- IceGrid.
- Glacier2.

Mediante Icegrid, que es un servicio de localización, se indican dónde están localizados los servicios, es decir, las máquinas en las que se ejecutan. Por otra parte, los agentes utilizan el servicio de localización de IceGrid para contactar con dichos servicios. Glacier2 se utiliza para solventar las dificultades que suponen los distintos entornos de red hostiles, de forma que es posible que un agente esté en una red local detrás de un router (con su cortafuegos), mientras que los servidores están bajo la infraestructura de una red universitaria.

4.2. Construcción de MASYRO

Una vez terminada la implementación básica de una arquitectura multi-agente, ya es posible reutilizarla para la implementación del sistema de renderizado distribuido. Desde un punto de vista abstracto tendremos agentes especializados en desarrollar distintas funcionalidades y servicios a los que acceden otros agentes. En la siguiente sección se realizará una breve introducción, describiendo los distintos servicios y agentes implicados en el sistema de renderizado distribuido, mientras que en las sucesivas secciones se dedicarán a analizar cada uno de los distintos pasos involucrados en el *flujo de trabajo*.

4.2.1. Introducción a MASYRO

Desde un punto de vista abstracto, MASYRO se puede ver como una caja negra cuya entrada es un modelo y la salida una imagen que representa la escena tridimiensional, como se puede apreciar en la figura 4.3. Visto desde otro punto de vista, el usuario tiene la impresión de

que está usando un motor de renderizado local, mientras que realmente su trabajo está siendo distribuido y desarrollado por los distintos agentes subscritos a MASYRO.

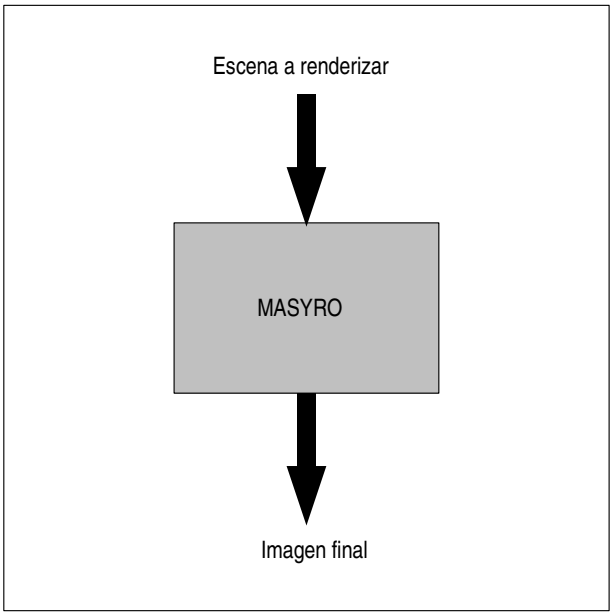

Figura 4.3: Vista abstracta de MASYRO

Dentro de esa caja negra reside el sistema multi-agente, con un número indeterminado de agentes que cooperan y compiten para desarrollar un trabajo. Los agentes en cuestión son los encargados de llevar a cabo dicha función, pero son necesarios un cierto número de servicios que aseguren el correcto funcionamiento del sistema (gestores), servicios que se encargan del análisis previo de la entrada, servicios que representen a repositorios de modelos, y servicios que permitan a los agentes ir notificando sus progresos.

En MASYRO existe una instancia de cada uno de los servicios anteriormente mencionados:

- El gestor del sistema está representado por la entidad ***Master***.
- El servicio encargado de llevar a cabo el análisis previo de la escena está representado por la entidad ***Analyst***.

- El servicio que soporta el repositorio de modelos está representado por la entidad ***ModelRepository***.

- El servicio que permite notificar sus progresos a los agentes y, que realmente representa a una arquitectura de pizarra, está representado por la entidad ***Blackboard***.

Como se puede apreciar en la figura 4.4, existe un número determinado de agentes vinculados a un gestor. Dichos agentes pueden ejecutarse en máquinas con distintas características, como por ejemplo en máquinas con distinto sistema operativo. También existe la posibilidad de tener distintos gestores dentro de la plataforma, los cuales se encargarán de coordinar distintos grupos de trabajo. De esta forma, dentro del mismo sistema pueden coexistir distintos grupos de trabajo desarrollando diferentes trabajos de manera completamente independiente. Además, es posible tener distintos agentes especializados en uno o varios métodos de renderizado. Para ello, simplemente es necesario modificar el sistema de reglas difuso interno a cada agente.

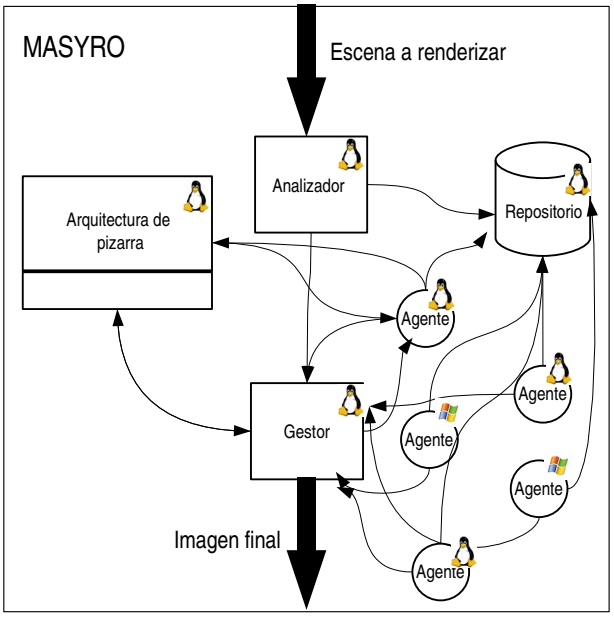

Figura 4.4: Vista detallada de MASYRO

El flujo básico de trabajo es el siguiente (ver figura 4.5):

- Los agentes especializados en el renderizado se subscriben a MASYRO a través de subscripciones al servicio gestor (Master).
- El sistema recibe un nuevo trabajo.
- El analizador lleva a cabo un análisis previo de la escena, realizando una división lógica de la escena en zonas en función de la complejidad de las mismas.
- El analizador notifica la existencia de un nuevo trabajo al gestor y envía el modelo al repositorio de modelos.
- El gestor notifica la existencia de un nuevo trabajo a los agentes subscritos especializados en el renderizado.
- Cada uno de los agentes obtiene el modelo del repositorio y comienza el proceso de subasta de trozos-renderizado.
- Los agentes van enviando los resultados parciales al gestor.
- El gestor lleva a cabo la composición de todos los resultados parciales, obteniendo el resultado final.
- El gestor envía el resultado final al usuario.

MASYRO dispone de una interfaz web que permite enviar nuevos trabajos y de un visualizador de imágenes integrado que va mostrando la evolución del trabajo conforme el tiempo va avanzando. Utilizando este enfoque, el usuario tiene la posibilidad de encargar un nuevo trabajo a MASYRO desde cualquier máquina con conexión a Internet.

A continuación se describirá con más detalle el flujo básico de trabajo.

4.2.2. Paso 1: Subscripción de los agentes

Una vez que un agente se ha integrado dentro de la plataforma de agentes, registrando en los servicios principales definidos por FIPA, el siguiente paso es llevar a cabo la funcionalidad específica para la cual ha sido creado. En el caso de MASYRO, dicha funcionalidad es el

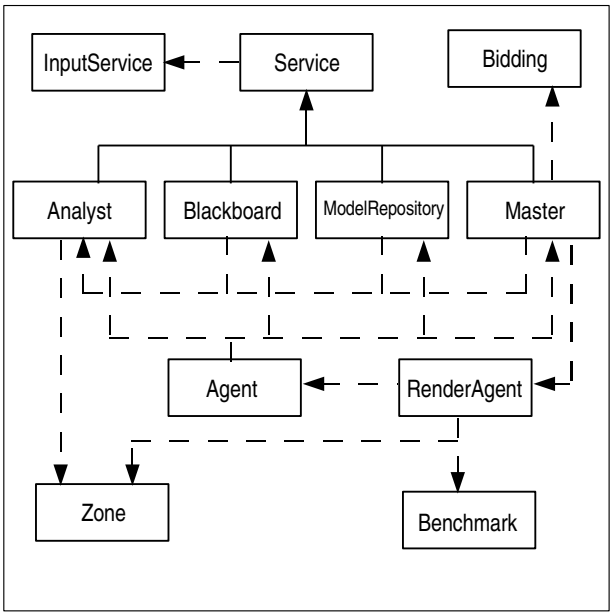

Figura 4.5: Diagrama de clases de MASYRO

renderizado. Según el diseño actual, el agente especializado en el renderizado notifica su voluntad para realizar dicha acción al gestor (o Master), que se corresponde con un servicio.

Dentro del conjunto de operaciones que soporta un gestor, existen dos operaciones de gestión relacionadas con la subscripción y eliminación de la subscripción por parte de un agente:

```
interface Master
{
   void subscribe(string agentName, RenderAgent* agent);
   void unsubscribe(string agentName);
   ...
};
```

Por una parte, la operación *subscribe* permite a un agente la subscripción con el elemento gestor. Para ello, el primero debe proporcionar dos parámetros:

- *agentName*: especifica el nombre del agente en cuestión.
- *agent*: especifica un *proxy* al propio agente. Dicho *proxy* representa al objeto agente en la parte del cliente, que en este caso se corresponde con la parte del gestor. A través de

este elemento, el gestor podrá llevar a cabo las invocaciones a operaciones remotas con respecto al agente.

Por otra parte, la operación *unsubscribe* permite que un agente elimine la subscripción que tiene con el elemento gestor. Para ello debe proporcionar su propio nombre. De forma interna, el elemento gestor mantiene un diccionario con parejas <nombre agente, *proxy* agente>, añadiendo o eliminando nuevos registros en función de las subscripciones o eliminación de las mismas por parte de los agentes.

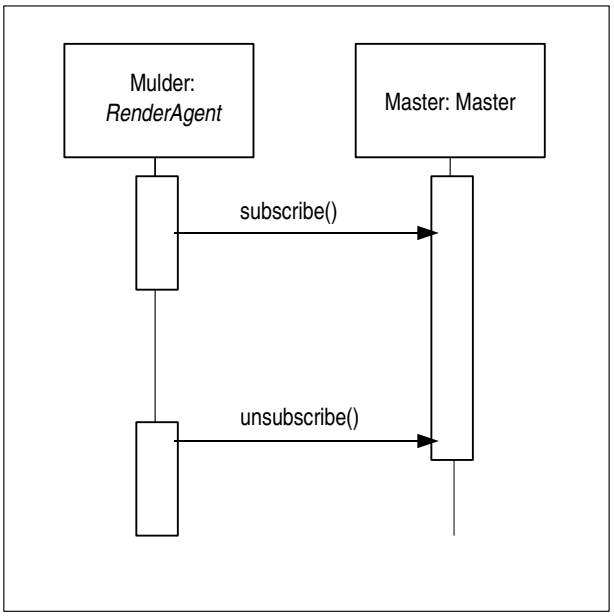

Figura 4.6: Diagrama de secuencia Agente-Master

En lo que a cuestiones de implementación se refiere, un agente especializado en el renderizado es una especialización del agente básico definido acorde con las especificaciones FIPA. De esta forma, toda la funcionalidad desarrollada en el agente básico es reutilizada en el agente especializado en el renderizado a través de la herencia.

4.2.3. Paso 2: Recepción de un nuevo trabajo por parte del sistema

El punto de entrada para un nuevo trabajo está representado por el servicio Analizador, el cual lleva a cabo el análisis previo de la entrada. Para ello, el Analizador proporciona la operación *processWork*. Dicha operación es invocada desde un *script* escrito en PHP, que reside en el servidor web. Cuando un cliente envía un nuevo trabajo, es el servidor web el que notifica al Analizador de la existencia de un nuevo trabajo. El código relevante del *script* PHP es el siguiente:

```
Ice_loadProfile();

try {
   $analystPrx = $ICE->stringToProxy("analyst")
      ->ice_checkedCast("::MASYRO::Analyst");
   $workName = $_POST['workName'];
   $analystPrx->processWork($work, $workName,
      $divisionLevel + 0, $optimizationLevel + 0);
}
catch (Ice_LocalException $ex) {
   print_r($ex);
}
```

Dicho código le permite al servidor web comunicarse con el analizador y suministrarle la información necesaria para llevar a cabo el análisis previo de la escena.

4.2.4. Paso 3: Análisis previo de la escena

Una vez que se conoce la existencia de un nuevo trabajo por parte del sistema, el primer paso consiste en realizar un análisis previo de la escena, con el objetivo de obtener una representación lógica por zonas de la complejidad de la escena de entrada. Dicho análisis previo lo lleva a cabo el **Analizador** (***Analyst***) para alcanzar los siguientes objetivos:

- División del trabajo en bloques de complejidad semejante. Con este objetivo se pretende conseguir que las zonas de trabajo repartidas en los agentes sean aproximadamente equivalentes, de forma que se eviten situaciones en las que la terminación de un trabajo queda retardada debido a una zona de complejidad elevada.
- Separación de la parte de análisis con respecto al renderizado final.

La interfaz pública del Analizador, comentada a continuación, es la siguiente:

```
interface Analyst
{
   void processWork(ByteSeq work, string workName, int level,
         int optimization);
};
```

La única operación de dicha interfaz es la operación *processWork*, que contiene los siguientes parámetros:

- *work*: se trata de una secuencia de *bytes* relativa a los ficheros necesarios para representar la escena de entrada.
- *workName*: se trata del nombre asignado al trabajo.
- *level*: se refiere al nivel de división en el análisis inicial.
- *optimization*: se refiere al nivel de optimización definido por el usuario.

La primera acción que realiza el Analizador es llevar a cabo un render inicial cambiando las propiedades de los materiales involucrados en la escena, de forma que el mapa de complejidad sea una imagen en escala de grises representando la complejidad aproximada teórica de la escena *a priori* (ver figura 4.7). Las zonas más complejas estarán representadas por colores cercanos al blanco, mientras que las zonas menos complejas estarán representadas por colores más cercanos al negro. Esta será la imagen utilizada para realizar la división inicial en zonas.

Si el valor del parámetro *level* es 1, el Analizador lleva a cabo una división en zonas en función de dos parámetros:

- El tamaño mínimo de una zona.
- La complejidad máxima de una zona.

Según estos parámetros, el Analizador realiza una división recursiva en zonas dividiendo una zona si está tiene un tamaño lo suficientemente grande y si la desviación estándar de la complejidad es mayor que la establecida por defecto. Esta primera división (ver figura 4.8) tiene ciertas carencias, destacando que no agrupa zonas de complejidad parecida y, sobre todo, que no establece una división equitativa en función de la complejidad del modelo.

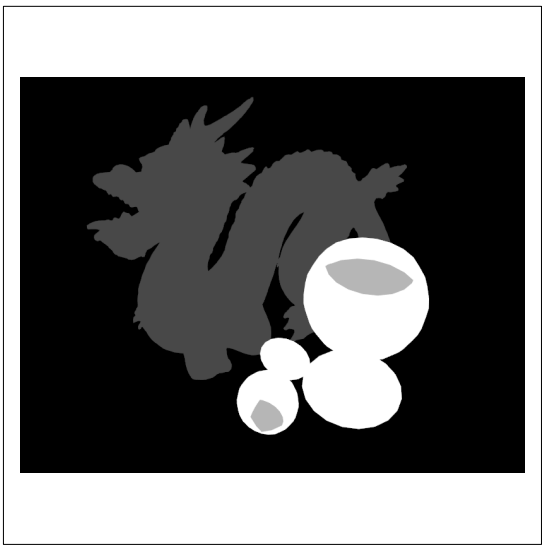

Figura 4.7: Imagen que representa la complejidad de una escena

Si el valor del parámetro *level* es 2, el Analizador efectúa una fusión de las zonas, previa división de la imagen inicial. Dicha fusión consiste en unir zonas vecinas en el caso de que éstas tengan una complejidad parecida (ver figura 4.9).

Por último, si el valor del parámetro *level* es 3, además de realizar la división inicial y la fusión, el Analizar realiza un equilibrado entre zonas, dividiendo aquellas zonas cuyo *ratio* complejidad-tamaño es elevado. Mediante esta nueva división (ver figura 4.10), se soluciona el problema planteado en la división inicial, relativo a que no se establecía una división equitativa en función de la complejidad del modelo. El parámetro del tamaño de un trozo también es relevante a la hora de llevar a cabo una estimación de la complejidad global de un trozo.

Llegados a este punto, el proceso de análisis previo de la escena por parte del Analizador llega a su fin. Con este proceso de análisis se alcanzan los siguientes objetivos:

- Obtener una representación inicial de la complejidad de la escena, en función de los materiales que la componen.
- Obtención de una división de la escena de entrada en zonas, en función de la complejidad y el tamaño de las mismas.

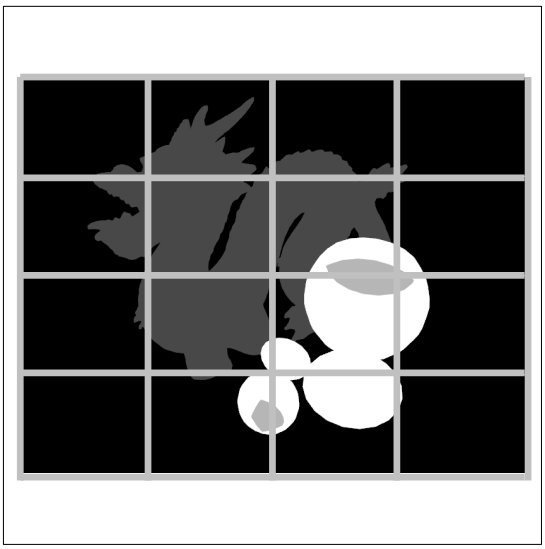

Figura 4.8: Análisis con un valor 1 del parámetro *level*

- Aislar el proceso de análisis previo del renderizado en cuestión.

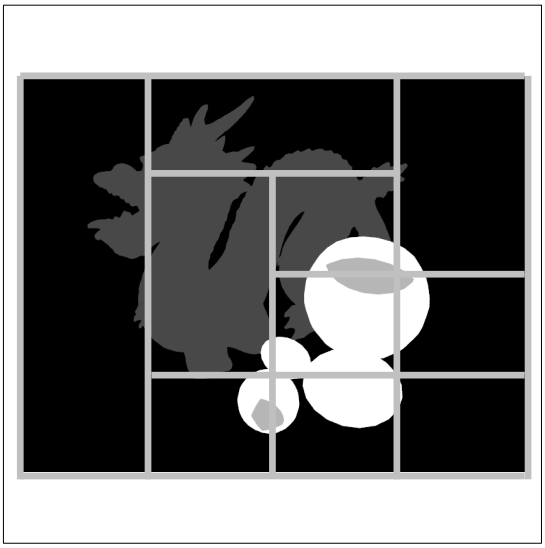

Figura 4.9: Análisis con un valor 2 del parámetro *level*

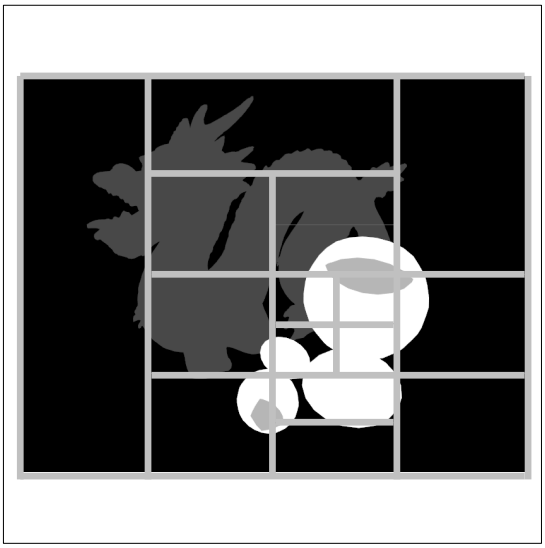

Figura 4.10: Análisis con un valor 3 del parámetro *level*

4.2.5. Paso 4: Notificación de la existencia de un nuevo trabajo

Una vez que el Analizador ha realizado el análisis previo, el siguiente paso consiste en notificar la existencia de un nuevo trabajo al gestor (Master). Para ello, el Analizador utiliza una operación de la interfaz pública del gestor:

```
interface Master
{
   void notifyNewWork(TZones zones, int idWork, int optimization);
   ...
};
```

La operación *notifyNewWork* tiene los siguientes parámetros:

- *zones*: representa una lista de zonas, en función de la división llevada a cabo por el Analizador.
- *idWork*: representa el identificador número asociado al trabajo, y que es necesario para obtener el modelo en el repositorio de modelos.
- *optimization*: representa el nivel de optimización definido por el usuario.

El parámetro *zones* es una secuencia de estructuras del tipo *TZone*:

```
struct TZone {
   int id;
   int x1;
   int y1;
   int x2;
   int y2;
   float d;
   float m;
};
```

Dicha estructura consta de los siguientes parámetros:

- *id*: se refiere al identificador asociado a la zona en cuestión. Es un valor numérico único que distingue a esa zona del resto de zonas.
- *(x1, y1, x2, y2)*: representa el cuadrante asociado a la zona en cuestión, donde (x1, y1) representa la esquina superior izquierda y (x2, y2) representa la esquina inferior derecha.

- *d*: representa la desviación estándar del color (complejidad) asociada a la zona en cuestión.

- *m*: representa la media del color (complejidad) asociada a la zona en cuestión.

Además de llevar a cabo la notificación de un nuevo trabajo al gestor, el analizador ha de enviarlo previamente al repositorio de modelos. Para ello hace uso de la interfaz pública del servicio que representa al repositorio de modelos:

```
interface ModelRepository
{
   int put(string name, ByteSeq model);
   nonmutating string get(int idModel, out ByteSeq model)
         throws ModelNotExistsException;
};
```

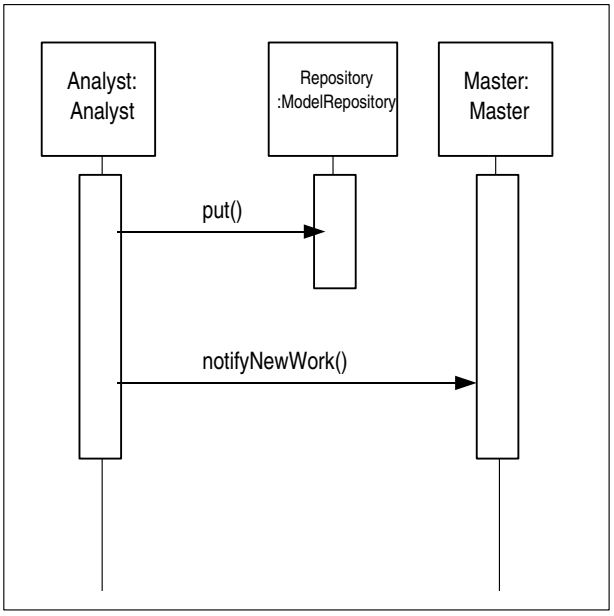

Figura 4.11: Diagrama de secuencia asociado al gestor

notifyWorkMaster

La interfaz *ModelRepository* consta de las siguientes operaciones:

- *put*: permite enviar un nuevo modelo al repositorio de modelos. Para ello es necesario proporcionar el nombre del modelo y una secuencia de *bytes* que lo represente. Esta

operación devuelve un parámetro de tipo entero, asociado al identificador asignado por el repositorio de modelos al modelo en cuestión. De esta forma, los distintos servicios y agentes podrán obtener el modelo asociado.

- *get*: permite obtener un modelo del repositorio de modelos. Para ellos es necesario proporcionar el identificador del modelo. En caso de que este no exista, se lanza una excepción que controla dicha situación.

Una vez que el modelo está disponible en el repositorio de modelos y que el gestor conoce la existencia de un nuevo trabajo, el siguiente paso consiste en notificar la existencia del mismo a los trabajadores finales, es decir, a los agentes especializados en el renderizado. Debido a que el gestor mantiene un diccionario referente a los agentes subscritos, puede utilizar la interfaz pública de éstos para informar de la existencia de un nuevo trabajo:

```
interface RenderAgent
{
   void notifyNewWork(TZones zones, int idWork, int benchmarkValue);
   ...
};
```

Dicha operación es análoga a la definida por el gestor, pero con una salvedad: la inclusión de un parámetro denominado *benchmarkValue*. Dicho parámetro representa el valor medio asociado al tiempo que los agentes subscritos al gestor emplearon al utilizar un *benchmark*. De hecho, la primera acción que los agentes desarrollan después de haberse subscrito al gestor es la ejecución del *benchmark*, y la notificación al mismo del tiempo empleado para ello.

El objetivo que se persigue con la inclusión de un *benchmark* es poder representar de una forma aproximada la capacidad que tiene cada uno de los agentes. Dicha capacidad será utilizada en el proceso de renderizado por parte de los agentes y tendrá una cierta importancia que se detallará en el apartado siguiente. Este valor relativo a la capacidad se va ajustando en tiempo de ejecución, ya que inicialmente se parte de un valor, pero los agentes van realizando modificaciones para obtener un valor más correcto. Por lo tanto, el valor obtenido de la ejecución del *benchmark* es únicamente un valor inicial, es decir, una primera estimación.

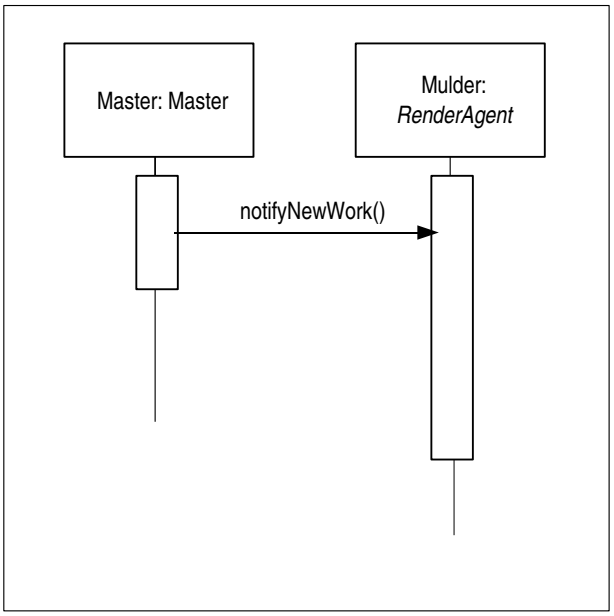

Figura 4.12: Diagrama de secuencia Master-Agente

4.2.6. Paso 5: Proceso de renderizado

Este paso constituye el paso más importante dentro del funcionamiento de MASYRO, y permite realizar el renderizado final de la escena de entrada entre el conjunto de agentes subscritos al gestor. En este paso se lleva a cabo el proceso de optimización del renderizado, permitiendo obtener unos tiempos inferiores y con una calidad similar en comparación con el renderizado tradicional, ya sea en una sola máquina o lanzando el trabajo en un sistema *grid*.

Una vez que el gestor (Master) ha notificado la existencia de un nuevo trabajo a sus agentes subscritos, el siguiente paso es realizar la distribución inicial por parte del gestor. Para ello, el gestor (que conoce en qué zonas se ha dividido la escena de entrada) notifica a los agentes subscritos los paquetes de trabajo que se les asigna *a priori*. A modo de ejemplo, si la escena está dividida en 16 zonas y hay 4 agentes subscritos, el gestor asignará 4 zonas a cada uno de los agentes sin ningún tipo de criterio. De esta forma, cada agente obtiene su paquete de trabajo inicial. El gestor utiliza la siguiente operación remota para desarrollar esta funcionalidad:

```
interface RenderAgent
{
   ["ami"] void notifyZones(TZones zones, int idWork, int optimization);
   ...
};
```

El parámetro *zones* representa la secuencia de zonas asignadas al agente, el parámetro *idWork* representa el identificador del trabajo y, por último, el parámetro *optimization* representa el nivel de optimización definido por el usuario. En la signatura de la operación también destaca el modificador *ami*, que quiere decir que la operación se invocará de forma asíncrona. De esta forma, el gestor no tiene que esperar a que un agente termine la ejecución de dicha operación para notificar el siguiente paquete de zonas al siguiente agente.

Cuando un agente recibe un paquete de zonas, lo que hace es llevar a cabo una estimación de cada una de las zonas de dicho paquete. Esta estimación consiste en realizar un renderizado a baja resolución, en concreto a un 10 % de la resolución predefinida, con el objetivo de obtener un tiempo de renderizado estimado para ese trozo. Llegados a este punto entra en juego la representación de la capacidad de la máquina en la que el agente se ejecuta, como se comentó en el apartado anterior. El objetivo es tener una estimación del tiempo estimado de renderizado en una medida universal, de forma que aunque un agente haga la estimación, el resto de agentes tengan una estimación *válida* aunque las capacidades de las máquinas en las que se ejecutan sean diferentes a la del primer agente. Con esta representación universal, todos los agentes mantienen una estimación similar para todos las zonas en las que se dividió la escena de entrada.

El ser capaz de tener una estimación de cada una de las zonas permite, posteriormente, trabajar en primer lugar sobre las zonas *a priori* más complejas. Mediante este enfoque se logra aún más equilibrar el trabajo entre los distintos agentes, ya que se trabajo primero sobre las partes más complejas de la escena.

El lugar en el que todos los agentes y el gestor escriben sus resultados reside como estructura de datos en el servicio *Blackboard*, la cual está vinculada a una arquitectura de pizarra. Gracias a este elemento, los agentes y el gestor comparten un lugar donde escribir, leer, modificar, y borrar toda aquella información relevante para el proceso de renderizado. La interfaz de este servicio se presenta a continuación.

```
interface Blackboard
{
   void write(TRegister register);
   TRegister read(int idWork, int workUnit)
         throws RegisterNotExistsException;
   void update(int idWork, int workUnit, int test)
         throws RegisterNotExistsException;
   void clear();

   bool isWorkPartiallyEstimated();
   bool isCurrentWorkFinished();
   string show();
   int getMaxTest(int idWork);
   int getMaxComp(int idWork);

   void setWorkUnit(int idWork, int workUnit, string agent)
         throws RegisterNotExistsException;
   void finishWorkUnit(int idWork, int workUnit, int treal,
         int ibs, int ls, int rl) throws RegisterNotExistsException;
};
```

Las cuatro primeras operaciones representan la funcionalidad asociada a la escritura y lectura de datos en la pizarra. El servicio encargado de escribir los datos inicialmente es el gestor, utilizando para ello la operación *write*. La operación *update* es ejecutada por los agentes, en el momento en que terminan de hacer la estimación de una zona.

El segundo bloque de operaciones son operaciones de consulta de datos:

- *isWorkPartiallyEstimated*: devuelve un valor de tipo *boolean*, en función de si el trabajo está parcialmente estimado, con el objetivo de comenzar el proceso de renderizado final.
- *isCurrentWorkFinished*: devuelve un valor de tipo *boolean*, en función de si todos los trozos se han terminado de renderizar o no.
- *show*: devuelve una representación textual del contenido de la pizarra.
- *getMaxTest*: devuelve el valor del mayor tiempo de estimación relativo a las zonas existentes.
- *getMaxComp*: devuelve el valor del mayor tiempo de renderizado final relativo a las zonas existentes.

Una vez que se ha llevado a cabo la distribución inicial y que los agentes han realizado

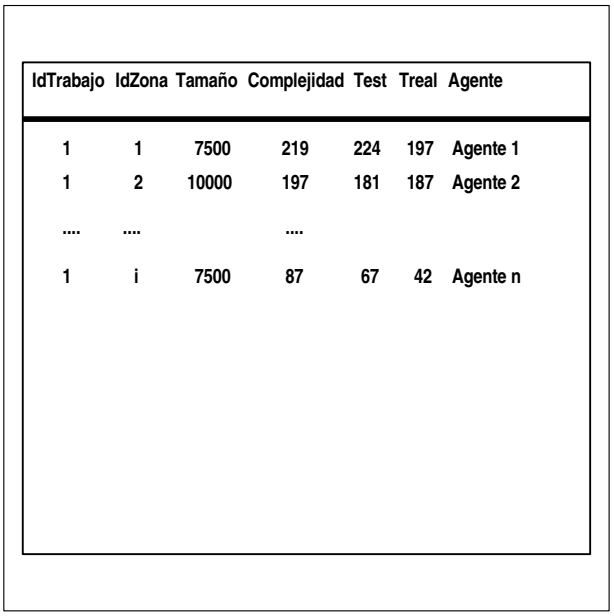

Figura 4.13: Arquitectura de pizarra

la estimación de cada una de las zonas, el sistema comienza el proceso de renderizado final. Para ello, el gestor utiliza la siguiente operación:

```
interface RenderAgent
{
   void beginRenderProcess();
   ...
};
```

Para que un agente puede efectuar el renderizado final de un trozo, es necesario que puje por el mismo. Para ello, se utiliza un mecanismo de subasta entre todos los agentes que en un momento determinado se encuentran *ociosos*, es decir, que aún no han realizado un trabajo o que ha realizado alguno o algunos y ya los ha terminado. Como se comentó anteriormente, los agentes siempre tratan de pujar por el trozo más complejo que aún no se ha renderizado. Si dos o más agentes pujan por un mismo trozo, se tienen en cuenta dos factores:

- El **número de créditos** del que dispone el agente. Dicho parámetro representa los éxitos y fracasos del agente, es decir, si terminó con anterioridad los trabajos en un tiempo menor o igual al que se estimó previamente. Si un agente termina un trozo antes del

tiempo estimado, dicho agente obtiene un premio que se traduce en un número determinado de créditos. Por el contrario, si un agente no termina un trozo antes del tiempo estimado, dicho agente es penalizado y su número de créditos disminuye.

- El **histórico** del agente que realiza la puja. Dicho parámetro representa la secuencia de éxitos o fracasos más recientes del agente en cuestión. Resulta interesante incorporar este tipo de parámetro, ya que es importante dar un peso al histórico más reciente del agente, con el objetivo de representar situaciones en las que un agente mantiene una secuencia de éxitos ante trozos de una misma naturaleza.

Suponiendo que un agente ha pujado por un trozo y el gestor se lo ha asignado, el agente usa un sistema de reglas difuso para simular el comportamiento de un experto en el sentido de configurar parámetros que afectan al renderizado. Dichos parámetros permiten disminuir el tiempo empleado para dicho proceso, a la vez que no experimentan apenas cambios en lo que se refiere a la calidad final obtenida. Los parámetros que se ajustan con el sistema de reglas, es decir, los que se corresponden con los consecuentes de las reglas del sistema difuso son los siguientes:

- **Nivel de recursión** (*recursion level*): parámetro que define el número de rebotes de los rayos de luz.
- **Número de *samples* por luz** (*light samples*): párametro que define el número de *samples* por luz.
- **Tamaño de la banda de interpolación** (*interpolation band size*): parámetro que especifica el tamaño de la banda de interpolación en píxeles.

Los dos primeros parámetros se obtendrán de la escena 3D a renderizar, por lo que el acceso a esto valores es inmediato. El tercer parámetro será empleado a la hora de hacer la composición final de la imagen, acción que se estudiará en el apartado siguiente.

Por otra parte, hay una serie de parámetros que determinan el valor de los consecuentes de las reglas, estudiados anteriormente. Estos parámetros se corresponden con los antecedentes de las reglas, y son los siguientes:

- **Complejidad**: parámetro que indica la relación complejidad-tamaño de la unidad de trabajo.
- **Diferencia con los vecinos** (*neighbour difference*): parámetro que define la diferencia de complejidad de la unidad de trabajo con el resto de vecinos.
- **Tamaño**: parámetro que representa el tamaño de la unidad de trabajo en píxeles.
- **Nivel de optimización** (*optimization level*): parámetro que indica el nivel de optimización deseado por el usuario.

A continuación se expone una tabla que resume la lista de variables del sistema de reglas difuso.

Variable	Tipo	Etiqueta	Conjunto de variable lingüísticas
Complejidad	Entrada	C	{VB, B, N, S, VS}
Diferencia vecinos	Entrada	Nd	{VB, B, N, S, VS}
Tamaño	Entrada	S	{B, N, S}
Nivel optimización	Entrada	Op	{VB, B, N, S, VS}
Tamaño banda interpolación	Salida	Ibs	{VB, B, N, S, VS}
Samples por luz	Salida	Ls	{VB, B, N, S, VS}
Nivel recursión	Salida	Rl	{VB, B, N, S, VS}

Cuadro 4.1: Variables del sistema difuso.

El conjunto de reglas es el que define el comportamiento del sistema, es decir, el que establece el valor de las variables de salida en función de las variables de entrada:

1. R1: If C is B,VB and S is B,N and Op is VB then Ls is VS and Rl is VS
2. R2: If C is N and S is B,N and Op is VB then Ls is VS and Rl is VS
3. R3: If C is S,VS and S is B,N and Op is VB then Ls is S and Rl is S
4. R4: If C is VB,B,N,S,VS and S is S and Op is VB then Ls is S and Rl is N
5. R5: If C is B,VB and S is B,N and Op is B then Ls is S and Rl is VS
6. R6: If C is N and S is B,N and Op is B then Ls is S and Rl is VS
7. R7: If C is S,VS and S is B,N and Op is B then Ls is N and Rl is S
8. R8: If C is VB,B,N,S,VS and S is S and Op is B then Ls is N and Rl is N
9. R9: If C is B,VB and S is B,N and Op is N then Ls is N and Rl is S
10. R10: If C is N and S is B,N and Op is N then Ls is N and Rl is S
11. R11: If C is S,VS and S is B,N and Op is N then Ls is N and Rl is N
12. R12: If C is VB,B,N,S,VS and S is S and Op is N then Ls is B and Rl is B

13. R13: If C is B,VB and S is B,N and Op is S then Ls is B and Rl is N
14. R14: If C is N and S is B,N and Op is S then Ls is B and Rl is N
15. R15: If C is S,VS and S is B,N and Op is S then Ls is B and Rl is B
16. R16: If C is VB,B,N,S,VS and S is S and Op is S then Ls is VB and Rl is B
17. R17: If C is B,VB and S is B,N and Op is VS then Ls is B and Rl is N
18. R18: If C is N and S is B,N and Op is VS then Ls is VB and Rl is B
19. R19: If C is S,VS and S is B,N and Op is VS then Ls is VB and Rl is VB
20. R20: If C is VB,B,N,S,VS and S is S and Op is VS then Ls is VB and Rl is VB
21. R21: If C is VB,B and Nd is VB then Ls is VB
22. R22: If Nd is VB then Ibs is VB
23. R23: If C is VB,B and Nd is B then Ls is VB
24. R24: If Nd is B then Ibs is B
25. R25: If C is VB,B and Nd is N then Ls is B
26. R26: If Nd is N then Ibs is N
27. R28: If Nd is S then Ibs is S
28. R30: If Nd is VS then Ibs is VS

Mediante este conjunto de reglas se obtienen valores para las variables de salida, de forma que aplicando este conjunto de reglas a cada unidad de trabajo obtenemos los valores de los parámetros que configuran el renderizado final.

El siguiente paso en lo relativo a la utilización del sistema de reglas conlleva la definición de los conjuntos difusos asociados a cada una de las variables. La definición de los conjuntos difusos para las variables de entrada se hace de forma dinámica, es decir, los intervalos de dichos conjuntos se calculan en función de los valores de entrada de todas las zonas. Dicho de otro modo, el escalado de los intervalos asociados a los conjuntos difusos se realiza en tiempo de ejecución.

Para las variables de salida sí que se definen los distintos conjuntos difusos asociadas a cada una de las variables. Para ello, se definen funciones triangulares y trapezoidales para cada una de las etiquetas lingüísticas asociadas a las variables lingüísticas.

La variable de salida asociada al tamaño de la banda de interpolación se define en función de los valores mostrados en la figura 4.14.

La variable de salida asociada al número de *samples* por luz se define en función de los valores mostrados en la figura 4.15.

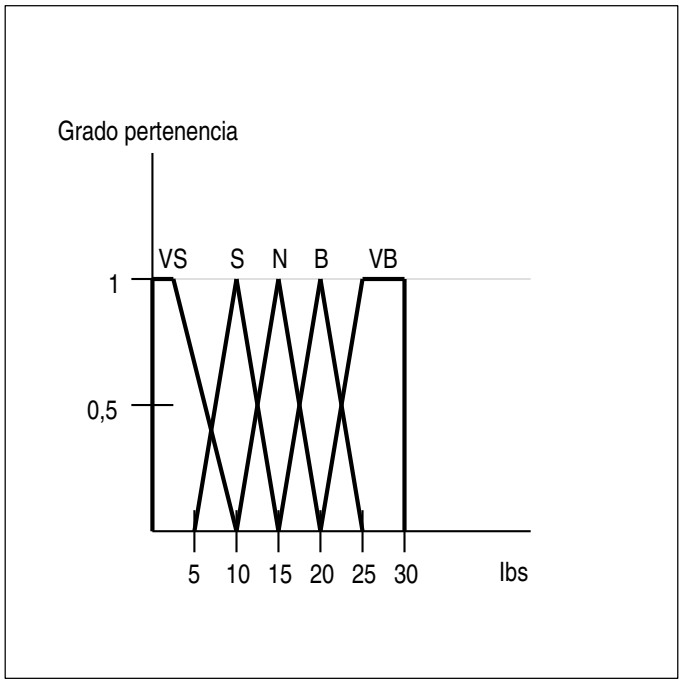

Figura 4.14: Definición de la variable *Tamaño de la banda de interpolación*

La variable de salida asociada al nivel de recursión se define en función de los valores mostrados en la figura 4.16.

Continuando con el flujo de trabajo, en el proceso de premio-penalización también existe un ajuste de un parámetro interno al agente, que se inicializó en función del tiempo obtenido al ejecutar el *benchmark*. De esta forma, se trata de corregir dicho factor con el objetivo de hacer estimaciones más precisas en sucesivos trabajos. El pseudo-código asociado a este proceso se expone a continuación:

```
begin renderProcess (nivelOptimizacion):

  zonaMasCompleja := obtenerZonaMasCompleja();
  exitoEnPuja := pujar(zonaMasCompleja);

  if exitoEnPuja == True:
  then tiempoFinal := renderFinal(zonaMasCompleja, nivelOptimizacion);
       ajusteInterno(tiempoFinal);
  renderProcess(nivelOptimizacion);

end renderProcess;
```

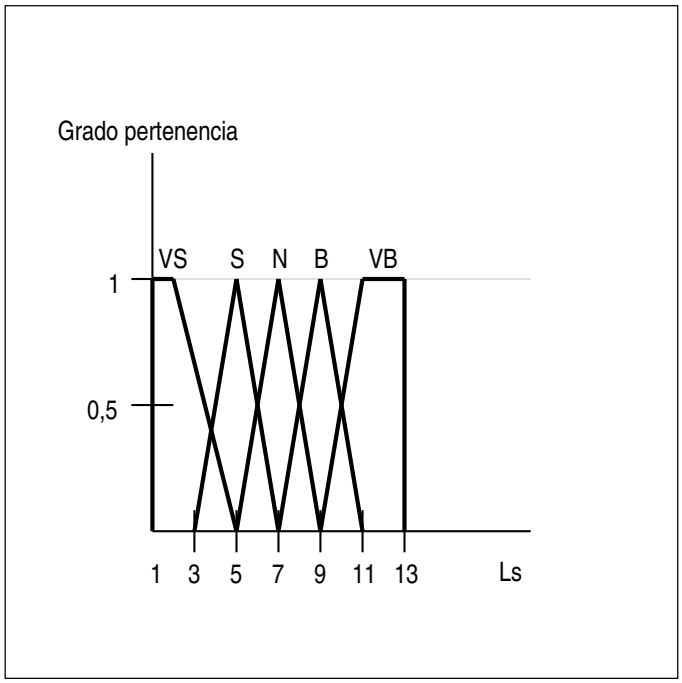

Figura 4.15: Definición de la variable *Número de samples por luz*

Siempre que un agente termina de renderizar un trozo, lleva a cabo dos acciones:

- Actualización en la pizarra.
- Envío al gestor del resultado parcial, es decir, de la parte de la imagen renderizada.

Para enviar un resultado parcial al gestor, el agente utiliza la siguiente operación:

```
interface Master
{
   void giveFinalImage(int idWork, int idZone, ByteSeq partialImage,
         int x1, int y1, int x2, int y2, int ibs);
   ...
};
```

De esta forma, el agente envía al gestor la información necesaria para que éste pueda llevar a cabo la composición del resultado final:

- Los parámetros *idWork* e *idZone* identifican a la zona.

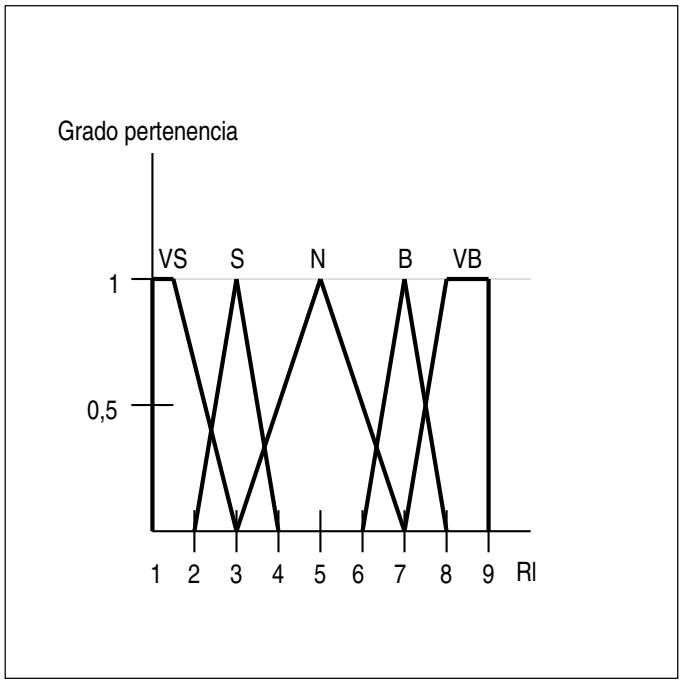

Figura 4.16: Definición de la variable *Nivel de recursión*

- El parámetro *partialImage* representa el flujo de *bytes* asociado al trozo de la imagen final.
- Los parámetros *(x1, y1, x2, y2)* representan al recuadro bajo el que se encierra el trozo de la imagen final.
- El parámetro *ibs* representa el tamaño de la banda de interpolación del trozo.

4.2.7. Paso 6: Composición del resultado final

El gestor es el encargado de llevar a cabo la composición de la imagen final, una vez que todos las unidades de trabajo han sido terminadas. Cada vez que un agente termina una unidad de trabajo notifica este hecho al gestor, para que éste vaya construyendo la imagen de salida progresivamente. Utilizando este enfoque, y cuando los agentes hayan terminado todas las unidades de trabajo, el gestor tendrá la imagen de salida final, correspondiente con el renderizado de la escena de entrada.

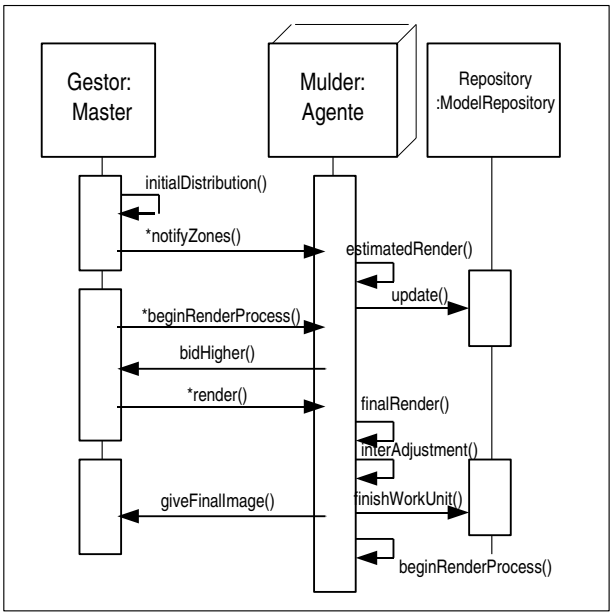

Figura 4.17: Proceso de renderizado final

El proceso de composición del resultado final es en realidad un proceso de unión de los resultados obtenidos al procesar las unidades de trabajo. Llegados a este punto, el parámetro a tener en cuenta es el tamaño de la banda de interpolación de cada trozo. Es importante recordar que el valor de este parámetro es uno de los valores de salida como consecuencia de aplicar el sistema de reglas difuso.

En la figura 4.18 se puede apreciar como existen dos trozos, uno vecino del otro, delimitados por franjas. La sección de la imagen que tienen en común representa la parte de la misma a la que se le aplicará una función de composición.

Para componer la imagen final, el gestor va *pegando* los distintos trozos teniendo en cuenta la vecindad de los mismos. A modo de ejemplo, si un trozo es el vecino derecho de otro trozo se ejecutarán las siguientes acciones:

1. Se *pega* en la imagen final el trozo que tiene un vecino por la derecha.

2. Se *pega* en la imagen final el trozo que se corresponde con el vecino derecho del anterior trozo.

Figura 4.18: Composición del resultado final

3. Se realiza una interpolación lineal entre los resultados de las dos unidades de trabajo, en función del tamaño de las bandas de interpolación, utilizando una función de la biblioteca PYL (*Python Imaging Library*) que realiza una composición de ambos trozos, de forma que el usuario no aprecia el cambio de un trozo a otro.

4.2.8. Paso 7: Visualización de resultados por parte del usuario

La visualización de los resultados por parte del usuario es posible gracias al interfaz web de MASYRO. Dicho interfaz web está compuesto de dos partes principales:

- Un visualizador de imágenes.
- Unas ventanas dedicadas a mostrar el estado de los agentes involucrados en el renderizado en tiempo de ejecución y a las actividades que éstos van desarrollando.

El visualizador de imágenes (ver figura 4.19) le permite al usuario visualizar cómo se va completando el trabajo, de forma que cuando una unidad de trabajo ha sido terminada el resultado aparece en el navegador.

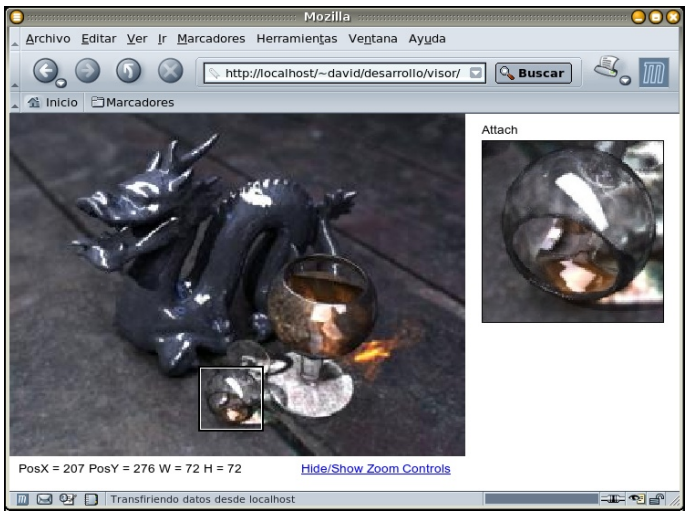

Figura 4.19: Aspecto gráfico del visualizador de MASYRO

Por otra parte, la interfaz también permite observar en todo momento el estado de los agentes. Dicho estado puede ser:

- Estimando (*Estimating*).
- Pujando (*Bidding*).
- Renderizando (*Rendering*).
- Descansando (*Resting*).
- Finalizando (*Finishing*).

Así mismo, se puede observar qué acciones ha ejecutado cada uno de los agentes en la parte dedicada al *log*, el tiempo empleado en realizar el análisis inicial, el tiempo dedicado a realizar las estimaciones de las distintas unidades de trabajo, o el tiempo empleado en el proceso de subasta.

Capítulo 5

Resultados obtenidos

5.1. Introducción

En este capítulo dedicado a los resultados obtenidos al ejecutar la aplicación se establece una comparativa entre los tiempos e imágenes renderizadas de manera tradicional y los resultados obtenidos aplicando MASYRO[1]. Así mismo, se pretende ilustrar al lector con el

[1]Las distintas imágenes obtenidas a partir de la realización de estas pruebas se incluyen en el anexo B.

enfoque elegido al desarrollar esta aplicación, diferenciándolo de otros enfoques ya establecidos. Este capítulo se ha estructurado en las siguientes secciones:

- **Renderizado tradicional**: en esta sección se expondrán los resultados obtenidos tras realizar un renderizado tradicional de la escena de prueba en una única máquina, definiendo los parámetros asociados al renderizado y que se mantendrán para el resto de pruebas.
- **Renderizado utilizando MASYRO y aplicando distintos niveles de particionado**: en esta sección se mostrarán los resultados de las pruebas utilizando los distintos niveles de particionados definidos en MASYRO (particionado estándar, particionado con fusión, y particionado con fusión y equilibrado final).
- **Renderizado utilizando MASYRO y aplicando distintos niveles de optimización**: en esta sección se mostrarán los resultados de las pruebas utilizando los distintos niveles de optimización que MASYRO ofrece (rango de 1 a 5).
- **Comparativa de resultados**: en esta sección se pretende ofrecer una comparativa de los distintos enfoques utilizados a la hora de renderizar.

La escena de prueba será una escena en la que aparecen un mono y un *famoso* robot (ver figura B.1). Es importante mencionar que en el análisis de resultados sólo se ha tenido en cuenta el tiempo empleado en el renderizado, es decir, que el tiempo dedicado al análisis previo y a la latencia de la red no se sumado al tiempo de renderizado. La razón se debe a que este tiempo es mínimo en comparación con el tiempo de renderizado, sobre todo en modelos más complejos que el utilizado para llevar a cabo las pruebas.

5.2. Renderizado tradicional

En la tabla 5.1 se exponen los distintos parámetros con los que se ha lanzado el render sin utilizar MASYRO, y utilizando una sola máquina.

Las características del equipo en el que se ha lanzado el renderizado se exponen en la tabla 5.2.

Parámetro	Valor
Motor de renderizado	Yafray
Método de renderizado	*Ray tracing*
Nivel de *oversampling*	8
Nivel de recursión	5
Método de iluminación global	*Full*
Calidad de iluminación global	*High*
Otros	Uso de *Irradiance cache*

Cuadro 5.1: Parámetros del renderizado sin utilizar MASYRO

Parámetro	Valor
Procesador	Intel Centrino
Frecuencia del procesador	1,6 GHz
Memoria del sistema	1GB
Sistema operativo	Debian GNU/Linux

Cuadro 5.2: Características de la máquina utilizada

Bajo estas situaciones, el tiempo empleado para obtener la imagen de la figura B.1 ha sido de 19 minutos y 28 segundos.

5.3. Renderizado con MASYRO y aplicando distintos niveles de particionado

En esta sección se estudiarán los resultados obtenidos utilizando MASYRO y distintos esquemas a la hora de particionar la escena de entrada en distintas unidades de trabajo. Los posibles esquemas están en función del parámetro que define el nivel de particionado:

- Un nivel de particionado 1 hace que MASYRO divida recursivamente la escena de entrada en unidades de trabajo en función de dos parámetros. El primero de ellos es la complejidad de la unidad de trabajo, y el segundo parámetro es el tamaño de la misma. De esta forma, si una unidad de trabajo tiene una complejidad mayor que la definida y un tamaño superior al mínimo definido, la unidad de trabajo se divide en cuatro unidades de trabajo.

- Un nivel de particionado 2 hace que MASYRO realice el particionado de nivel 1 y, a continuación, realice otra pasada fusionando unidades de trabajo adyacentes que man-

tienen una complejidad semejante. De esta forma, el particionado es mucho más homogéneo y acorde con las características de la escena.

- Por último, un nivel de particionado 3 hace que MASYRO realice un particionado de nivel 1, después un particionado de nivel 2 y, a continuación, realice una tercera pasada de equilibrado de zonas. Este equilibrio consiste en dividir unidades de trabajo de forma que todas tengan un *ratio* de tamaño-complejidad similar.

5.3.1. Particionado de primer nivel

Llegados a este punto, es importante reflexionar sobre el enfoque utilizado a la hora de realizar el estudio previo de la escena. Actualmente, existen proyectos de *rendering* distribuido (como por ejemplo *Yafrid* [54]) que realizan un particionado de la escena de entrada de forma que todas las unidades de trabajo tengan el mismo tamaño. Este enfoque tiene la ventaja de que realizar dicho particionado es muy sencillo, ya que consiste en definir un tamaño de unidad de trabajo y hacer la división acorde con el mismo; pero el inconveniente de que no tiene en cuenta la naturaleza de la escena, es decir, no realiza ningún tipo de estudio para averiguar la complejidad las distintas zonas que conforman la entrada.

El enfoque utilizado en MASYRO es radicalmente opuesto (véase figura 5.1), principalmente por tres motivos:

- El estudio de la escena de entrada y, por lo tanto su particionado, se realiza en función de la complejidad de la misma.

- MASYRO es una instanciación de una arquitectura multi-agente de propósito general, por lo que la naturaleza de la aplicación es totalmente distinta a la del enfoque mencionado al comienzo de la sección.

- Los agentes de MASYRO incorporan conocimiento experto que selecciona los mejores parámetros de render para cada unidad de trabajo, consiguiendo optimizar el tiempo de render sin pérdida de calidad perceptiva.

Una vez aclarada esta diferencia relativa al particionado de primer nivel de MASYRO, se procederá al estudio de los resultados obtenidos. Es importante destacar que el nivel de opti-

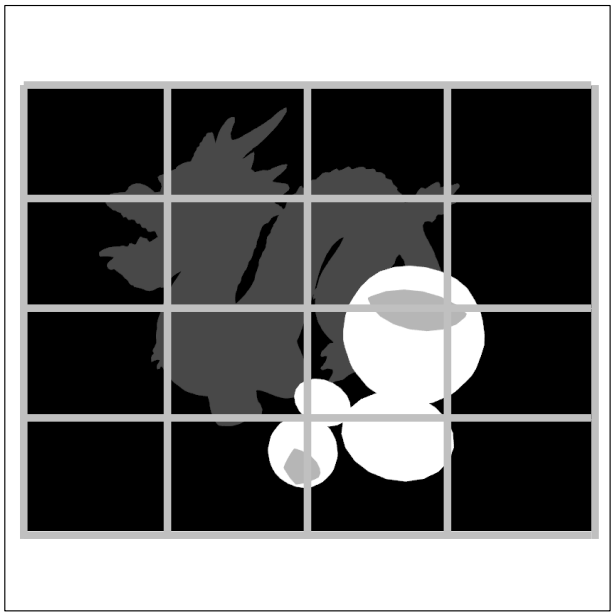

Figura 5.1: Modelo del dragón aplicando un particionado de primer nivel

mización, es decir, el parámetro que interviene en los antecedentes de las reglas del sistema difso, se establecerá en 3 dentro de un rango de 1 a 5. En sucesivos apartados se aplicarán distintos niveles de optimización.

En la tabla 5.3 se exponen los resultados obtenidos utilizando un particionado de primer nivel.

Número de agentes	Nivel de particionado	Nivel de optimización	Tiempo empleado (hh:mm:ss)
1	1	3	00:19:10
2	1	3	00:09:47
4	1	3	00:05:27
8	1	3	00:05:27

Cuadro 5.3: Resultados obtenidos con particionado de primer nivel

5.3.2. Particionado de segundo nivel

En la tabla 5.4 se exponen los resultados obtenidos utilizando un particionado de segundo nivel (véase figura 5.2).

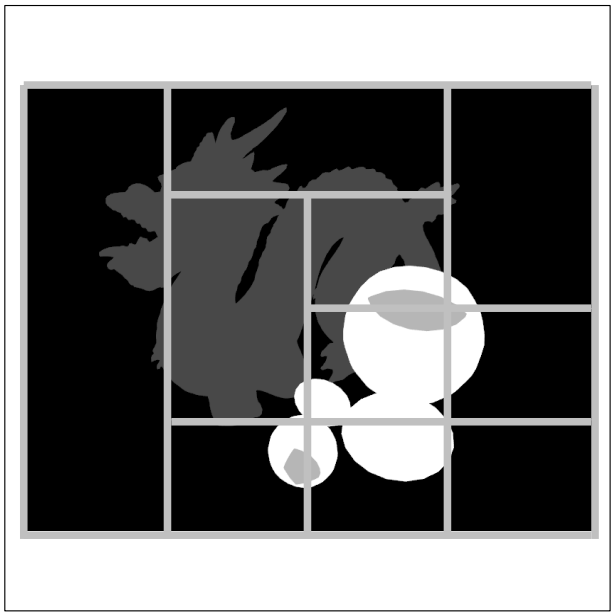

Figura 5.2: Modelo del dragón aplicando un particionado de segundo nivel

Número de agentes	Nivel de particionado	Nivel de optimización	Tiempo empleado (hh:mm:ss)
1	2	3	00:17:04
2	2	3	00:08:31
4	2	3	00:04:52
8	2	3	00:04:52

Cuadro 5.4: Resultados obtenidos con particionado de segundo nivel

5.3.3. Particionado de tercer nivel

En la tabla 5.5 se exponen los resultados obtenidos utilizando un particionado de tercer nivel (véase figura 5.3).

5.4. Renderizado con MASYRO y aplicando distintos niveles de optimización

En esta sección se estudiarán los resultados obtenidos utilizando MASYRO con distintos niveles de optimización. El nivel de optimización más agresivo es el número 5, mientras que

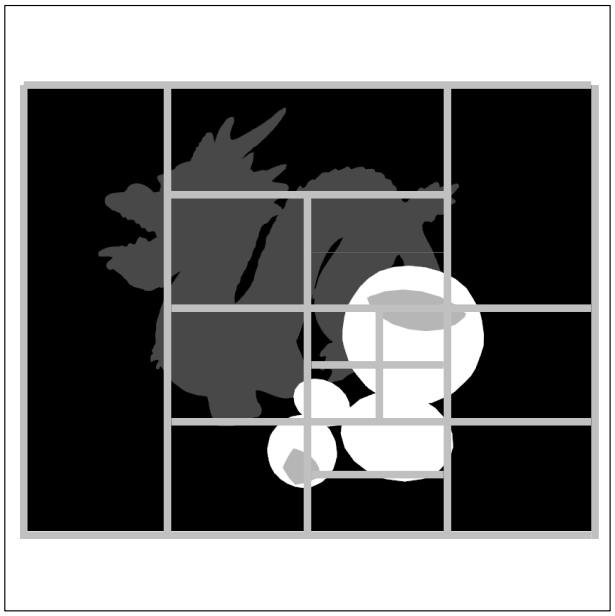

Figura 5.3: Modelo del dragón aplicando un particionado de tercer nivel

Número de agentes	Nivel de particionado	Nivel de optimización	Tiempo empleado (hh:mm:ss)
1	3	3	00:21:42
2	3	3	00:10:53
4	3	3	00:05:29
8	3	3	00:03:29

Cuadro 5.5: Resultados obtenidos con particionado de tercer nivel

el nivel de optimización menos agresivo es el número 1. El nivel de particionado elegido para realizar este conjunto de pruebas será el nivel 3 (ver figura 4.10).

5.4.1. Renderizado con MASYRO y aplicando un nivel 1 de optimización

En la tabla 5.6 se exponen los resultados obtenidos aplicando un nivel de optimización 1.

La imagen bidimensional obtenida al realizar el renderizado con nivel de optimización 1 se muestra en la figura B.2.

Número de agentes	Nivel de particionado	Nivel de optimización	Tiempo empleado (hh:mm:ss)
1	3	1	00:25:50
2	3	1	00:12:56
4	3	1	00:06:28
8	3	1	00:04:54

Cuadro 5.6: Resultados obtenidos aplicando optimización de nivel 1

5.4.2. Renderizado con MASYRO y aplicando un nivel 2 de optimización

En la tabla 5.7 se exponen los resultados obtenidos aplicando un nivel de optimización 2.

Número de agentes	Nivel de particionado	Nivel de optimización	Tiempo empleado (hh:mm:ss)
1	3	2	00:22:54
2	3	2	00:11:33
4	3	2	00:05:37
8	3	2	00:04:19

Cuadro 5.7: Resultados obtenidos aplicando optimización de nivel 2

La imagen bidimensional obtenida al realizar el renderizado con nivel de optimización 2 se muestra en la figura B.3.

5.4.3. Renderizado con MASYRO y aplicando un nivel 3 de optimización

En la tabla 5.8 se exponen los resultados obtenidos aplicando un nivel de optimización 3.

Número de agentes	Nivel de particionado	Nivel de optimización	Tiempo empleado (hh:mm:ss)
1	3	3	00:21:42
2	3	3	00:10:53
4	3	3	00:05:29
8	3	3	00:03:29

Cuadro 5.8: Resultados obtenidos aplicando optimización de nivel 3

La imagen bidimensional obtenida al realizar el renderizado con nivel de optimización 3 se muestra en la figura B.4.

5.4.4. Renderizado con MASYRO y aplicando un nivel 4 de optimización

En la tabla 5.9 se exponen los resultados obtenidos aplicando un nivel de optimización 4.

Número de agentes	Nivel de particionado	Nivel de optimización	Tiempo empleado (hh:mm:ss)
1	3	4	00:17:44
2	3	4	00:08:52
4	3	4	00:04:43
8	3	4	00:03:22

Cuadro 5.9: Resultados obtenidos aplicando optimización de nivel 4

La imagen bidimensional obtenida al realizar el renderizado con nivel de optimización 4 se muestra en la figura B.5.

5.4.5. Renderizado con MASYRO y aplicando un nivel 5 de optimización

En la tabla 5.10 se exponen los resultados obtenidos aplicando un nivel de optimización 5.

Número de agentes	Nivel de particionado	Nivel de optimización	Tiempo empleado (hh:mm:ss)
1	3	5	00:17:35
2	3	5	00:08:49
4	3	5	00:04:41
8	3	5	00:03:18

Cuadro 5.10: Resultados obtenidos aplicando optimización de nivel 5

La imagen bidimensional obtenida al realizar el renderizado con nivel de optimización 5 se muestra en la figura B.6.

5.5. Comparativa de resultados

En esta sección se irán estudiando los resultados expuestos en las tablas anteriores, haciendo especial hincapié en los tiempos de renderizado finales. Así mismo, también se es-

tablecerá una comparativa de los mismos comentando el por qué de la diferencia de dichos tiempos.

La figura 5.4 muestra la relación de los tiempos empleados para llevar a cabo el renderizado utilizando distintos números de agentes, aplicando un particionado de primer nivel. Como se expuso en la tabla 5.3, el tiempo utilizado es ligeramente menor al tiempo empleado con un renderizado sin optimización, lo cual se debe a la optimización aplicada por MASYRO. Por otra parte, y debido a la naturaleza de los métodos no deterministas de Monte Carlo y a la diferencia de calidad entre regiones, es necesario el uso de una banda de interpolación, es decir, una banda utilizada a la hora de interpolar distintas unidades de trabajo. Otra cuestión interesante es que a la hora de utilizar más de cuatro agentes (ver tabla 5.3), el tiempo final de render no se mejora. Este hecho se debe a que el trabajo en su conjunto está condicionado por el tiempo empleado en la unidad de trabajo más compleja. Esta situación refleja la debilidad de un particionado que no tiene en cuenta el equilibrio entre las distintas unidades de trabajo, es decir, la unidad de trabajo más compleja retrasa el trabajo completo.

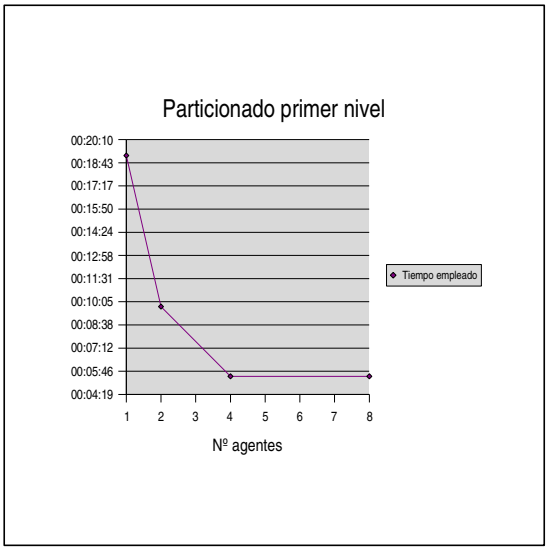

Figura 5.4: Gráfica asociada al particionado de primer nivel

La figura 5.5 hace referencia al particionado de segundo nivel, en el que se lleva a cabo la fusión de trozos. Este particionado sufre el mismo problema que el particionado de primer nivel, es decir, el utilizar un número de agentes mayor que 4 no supone ningún beneficio para el tiempo de renderizado final.

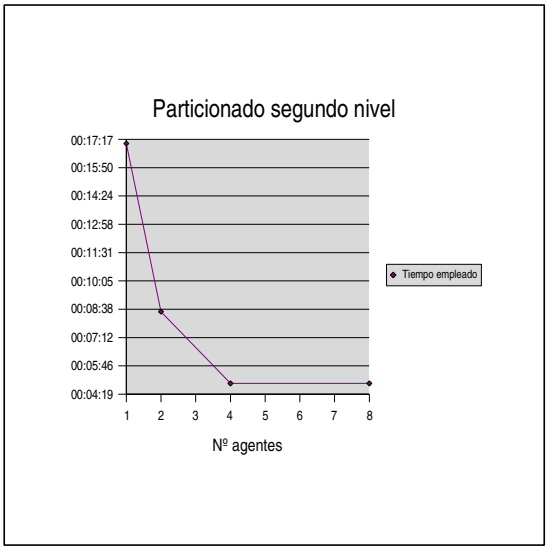

Figura 5.5: Gráfica asociada al particionado de segundo nivel

Como contrapartida, y como se muestra en la figura 5.6, un particionado que tenga en cuenta el equilibrio entre las distintas unidades de trabajo sí reporta un mayor beneficio a la hora de utilizar un mayor número de agentes. En este caso, la mejora obtenida es sustancial con respecto al tiempo obtenido utilizando 4 agentes. Por lo tanto, una conclusión importante que se puede obtener es que la utilización de un enfoque que lleve a cabo un estudio previo de la escena, y que haga un particionado *inteligente* que divida la escena en unidades de trabajo de una complejidad teórica parecida, reporta grandes beneficios a la hora de realizar *rendering* distribuido.

En cuanto a los tiempos obtenidos con distintos niveles de optimización, merece la pena resaltar la disminución de tiempos conforme el usuario utiliza un nivel de optimización más

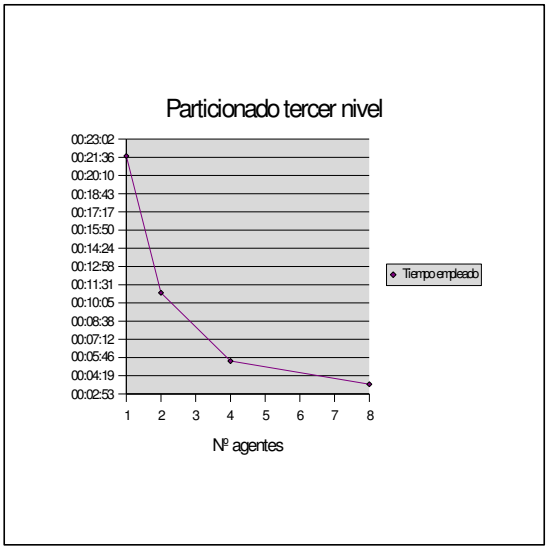

Figura 5.6: Gráfica asociada al particionado de tercer nivel

agresivo. Sin embargo, cuanto mayor es el nivel de optimización, más apreciables son las diferencias con respecto al renderizado sin optimización. Este hecho se debe a que la variación de los parámetros que intervienen en el renderizado, como por ejemplo el número de *samples* por luz, hacen que la calidad del resultado final sea menor. Por otra parte, al utilizar un particionado con equilibrado, un mayor número de agentes hace que el trabajo se realice en menor tiempo (ver tabla 5.7). Utilizando un nivel de optimización 2 y empleando 8 agentes, se obtiene un tiempo final de renderizado de 4 minutos y 19 segundos, obteniendo un resultado de calidad similar al obtenido con un renderizado sin optimización, cuyo tiempo es de 19 minutos y 28 segundos. El tiempo empleado ha sido cuatro veces menor. En la figura 5.7 se muestra una comparativa de tiempos en relación a los distintas optimizaciones aplicadas por MASYRO.

Otra reflexión importante surge a la hora de pensar que si utilizamos n agentes, el tiempo teórico final se reduzca en un factor n. Utilizando MASYRO, no sólo es posible obtener esta reducción, sino que también es posible obtener un tiempo final menor que T/n, donde T

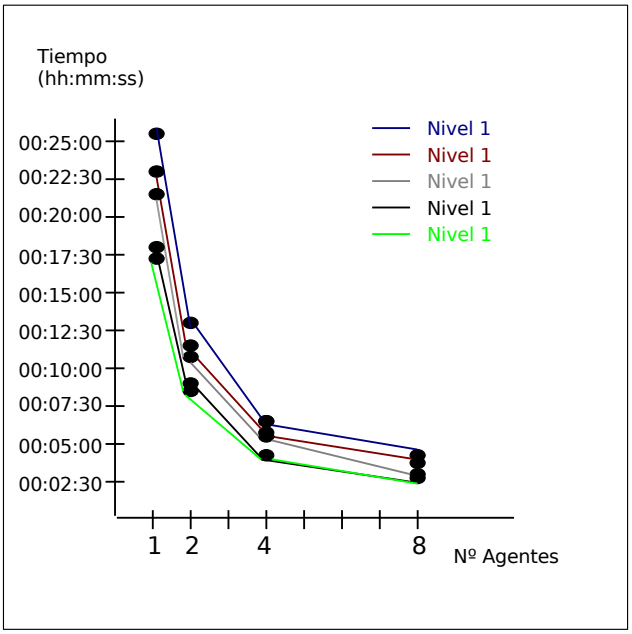

Figura 5.7: Gráfica asociada a las optimizaciones de MASYRO

representa el tiempo empleado en realizar el renderizado sin optimizaciones en una máquina, y n representa el número de agentes. Visto de otro modo, con MASYRO es posible obtener una optimización en tiempo mayor que el factor indicado por el número de agentes.

En el capítulo posterior de conclusiones y propuestas se estudiarán distintas posibilidades que en teoría mejoran el proceso de análisis previo de la escena y, como consecuencia, obtienen una mejor división de la escena en unidades de trabajo.

Capítulo 6

Conclusiones y propuestas

6.1. Conclusiones

La consecución de este proyecto supone una primera aproximación en lo relativo a la aplicación de técnicas de Inteligencia Artificial con respecto al proceso de renderizado. Los objetivos alcanzados son los siguientes:

1. Obtención de un sistema multi-agente de propósito general que sigue las ideas propuestas por el comité FIPA.

2. Instanciación de dicho sistema con el objetivo de construir una aplicación de renderizado distribuido.

3. Obtención de un módulo de análisis que determine la complejidad teórica de una escena.

4. Aplicación de técnicas de Inteligencia Artificial al problema del *rendering* distribuido.

El hecho de construir un sistema multi-agente básico abre las puertas a reutilizar dicho sistema para la aplicación que el desarrollador requiera, como por ejemplo:

- Sistemas de comercio electrónico.
- Sistemas de televigilancia.
- Aplicaciones en entornos industriales.
- Sistemas de monitorización de propósito general.
- Robótica.
- Etc.

La parte relativa a la arquitectura multi-agente se ha preparado para que el desarrollador pueda implementar sus propios agentes, de una forma escalable y sin preocuparse de las actividades de gestión relativas al conjunto de estándares FIPA.

Por otra parte, la obtención de un sistema que permite acelerar el proceso de renderizado es un logro importante, ya que los tiempos de renderizado en síntesis de imagen fotorrealista suelen ser muy elevados. La inclusión de un sistema experto que permite optimizar dicho proceso supone una aproximación muy interesante, porque normalmente siempre se pueden configurar los parámetros de renderizado de forma que la calidad obtenida es prácticamente la misma al mismo tiempo que el tiempo final de renderizado es mucho menor. Además, esta inclusión supone una gran ventaja, debido a que en cualquier momento es posible remodelar el conocimiento de un experto gracias a la utilización de un sistema de reglas. Este tipo de sistemas son muy escalables, y permiten al desarrollador hacer cambios con un impacto nulo en la aplicación en cuestión.

MASYRO se ha desarrollado separando la parte de configuración de la parte de código, de forma que los servicios que proporciona y los agentes involucrados pueden ejecutarse en cualquier máquina conectada a Internet. Para lograr tal objetivo, el usuario simplemente debe ajustar unos cuantos parámetros en los archivos de configuración y ejecutar la aplicación. Además, el usuario ha de preocuparse mínimamente de cuestiones asociadas a las redes, como por ejemplo Internet, de forma que pueda centrar su atención en la aplicación en cuestión.

6.2. Propuestas y líneas de investigación futuras

Debido a que MASYRO supone una primera aproximación al campo del *rendering* distribuido con la utilización de una arquitectura multi-agente de propósito general subyacente, junto con el uso de técnicas de *soft-computing*, el número de propuestas y líneas de investigación asociadas es muy amplio. En esta sección se comentarán una serie de ideas, empezando por el hecho de utilizar un arquitectura multi-agente que se guía por el conjunto de estándares definidos por FIPA, pasando por el uso de elementos como un sistema de reglas difuso, y terminando por el amplio número de optimizaciones que se pueden aplicar en cada uno de los pasos definidos en el *flujo de trabajo* de MASYRO.

6.2.1. Propuestas relativas al sistema multi-agente

El hecho de guiarse por el conjunto de estándares definidos por FIPA supone la implementación de numerosos servicios, la implementación de diferentes protocolos de interacción, la implementación de un lenguaje de comunicación entre agentes, y la implementación de una gran variedad de elementos involucrados en la creación de un sistema multi-agente de acuerdo a los estándares definidos por FIPA. En este proyecto se ha realizado una implementación de los elementos básicos de la arquitectura propuesta por FIPA. Por lo tanto, en este sentido se abren numerosas vías a la hora de continuar con el desarrollo de MASYRO:

- Completar la especificación de todas las operaciones de gestión definidas por FIPA, como por ejemplo la creación y destrucción de agentes, o la movilidad de agentes entre distintas plataformas.
- Integración del servicio encargado de la transferencia de mensajes, cuya implementación básica se adjunta con este proyecto, dentro de la arquitectura multi-agente.
- Integración del concepto de mensaje, cuya implementación se adjunta con este proyecto, dentro de la arquitectura multi-agente.
- Creación de un lenguaje de comunicación entre agentes, bien utilizando especificaciones definidas por FIPA o partiendo desde cero.

- Integración del contenido de un mensaje, cuya implementación básica se adjunta con este proyecto, dentro de la arquitectura multi-agente.
- Implementación de los distintos protocolos de interacción definidos por FIPA.

La mayor complejidad en lo que a la implementación se refiere estaría asociada al primer punto, sobre todo de cara a gestionar la movilidad de los agentes entre distintas plataformas. En cuanto a la opción de crear un lenguaje de comunicación entre agentes, una opción es utilizar la especificación FIPA ***FIPA RDF Content Language Specification*** [3]. Dicha especificación describe cómo el *framework* de descripción de recursos (*Resource Description Framework* [26]) se puede utilizar como lenguaje de contenidos en un mensaje FIPA. Los esquemas RDF pueden definirse extendiendo sus modelos para expresar:

- Objetos que representan una entidad identificable en el dominio del discurso.
- Proposiciones que expresan que una sentencia de un lenguaje es verdadera o falsa.
- Acciones que expresan actividades que los objetos pueden realizar.

Una vez integradas todas estas características, se podría pensar en un entorno que facilitara la instanciación de la arquitectura multi-agente al usuario. Para ello, se podría diseñar una aplicación gráfica que simplificara dicha tarea, de forma similar a la empleada en otras herramientas (véase [11]).

6.2.2. Propuestas relativas a MASYRO

Una vez que MASYRO recibe un nuevo trabajo, se llevan a cabo distintas acciones hasta la obtención de la imagen bidimensional que representa a la escena tridimensional incluida en el nuevo trabajo. De forma resumida, dichas acciones son las siguientes:

1. Proceso de análisis de entrada.
2. Notificación del trabajo al gestor.
3. Notificación del trabajo a los agentes subscritos al gestor.

4. Proceso de renderizado distribuido utilizando conocimiento experto.

5. Composición del resultado final.

Todos estos procesos abren una gran variedad de posibilidades en lo que a optimizaciones y mejoras se refiere. A continuación se irán estudiando distintas propuestas que contribuirán a la mejora de MASYRO como sistema de *rendering* distribuido.

6.2.2.1. Proceso de análisis de entrada

El proceso de análisis de entrada consiste en realizar un estudio de la complejidad teórica de la escena a analizar. Dicho proceso es ejecutado por el Analizador, encargado de notificar la existencia de un nuevo trabajo al gestor. Como se expuso en capítulos anteriores, el proceso de análisis de entrada admite tres niveles de particionado:

- Particionado de primer nivel, dividiendo la escena en unidades de trabajo de acuerdo a la complejidad y tamaño de los mismos.

- Particionado de segundo nivel, fusionando unidades de trabajo adyacentes de complejidad parecida.

- Particionado de tercer nivel, equilibrando las unidades de trabajo en lo que a complejidad final se refiere.

La primera mejora que se podría estudiar estaría vinculada con los parámetros que rigen el particionado de primer nivel. Dichos parámetros son la complejidad de la unidad de trabajo y su tamaño. Actualmente, la división de la escena de entrada en unidades de trabajo se realiza mediante valores predefinidos de complejidad y tamaño de la unidad de trabajo mínima. Para que una unidad de trabajo se vuelva a dividir deben cumplirse las siguientes restricciones:

1. Que la desviación estándar asociada a la complejidad sea mayor que un factor definido previamente.

2. Que la anchura y altura de la unidad de trabajo sean mayores que las definidas en sendos factores de anchura y altura mínimas.

Como se puede apreciar, la división de la escena en unidades de trabajo se realiza utilizando un mecanismo puramente estático. Una posible propuesta sería la de incluir algún tipo de realimentación para que a la hora de realizar sucesivos análisis de entrada se utilizará la información previamente almacenada de análisis anteriores. De esta forma, el esquema utilizado se adaptaría en función de los resultados obtenidos, y sería útil para los tres niveles de particionado. Una posible implementación de este esquema estaría basada en la utilización de la información asociada al proceso de premio-penalización involucrado en el renderizado por parte de los agentes. Por ejemplo, si la consecución de una unidad de trabajo supone una penalización más o menos grave para un agente, se podría establecer un proceso de realimentación al Analizador, de forma que éste *aprendiera* y fuera consciente de que la división para esa unidad de trabajo en cuestión no fue acertada. Así mismo, si la consecución de la unidad de trabajo supuesto un éxito para el agente y el tiempo final de renderizado fue mucho menor que el tiempo estimado, también se podría informar al Analizador de este hecho.

Otra mejora importante, previa al proceso de particionado, podría estar relacionada con el *script* utilizado para obtener la imagen que representa la complejidad teórica de la escena a renderizar. Actualmente, dicho *script* contempla parámetros importantes relativos a los materiales empleados, como son el nivel de recursión y el nivel de refracción de los mismos. Sin embargo, una mejora sería la de contemplar más elementos relacionados con la escena, como por ejemplo la complejidad de la malla poligonal, o qué materiales tienen texturas y, de ser así, el número de imágenes utilizadas en las mismas, estudiar las distancias empleando mapas de profundidad, estudiar las distancias entre los distintos objetos que componen la imagen, etc.

6.2.2.2. Notificación del trabajo al gestor

Una vez que el Analizador ha realizado el proceso de análisis inicial, el siguiente paso consiste en notificar la existencia de un nuevo trabajo a un gestor. Actualmente, la implementación de MASYRO mantiene un único gestor, accesible a través de un identificador único y al que los agentes se subscriben. En este paso del flujo de trabajo, se podría escalar la arquitectura de MASYRO incluyendo un determinado número de gestores, de forma que representarán (junto a sus agentes subscritos) grupos de trabajo independientes, como se aprecia en la figura

6.1.

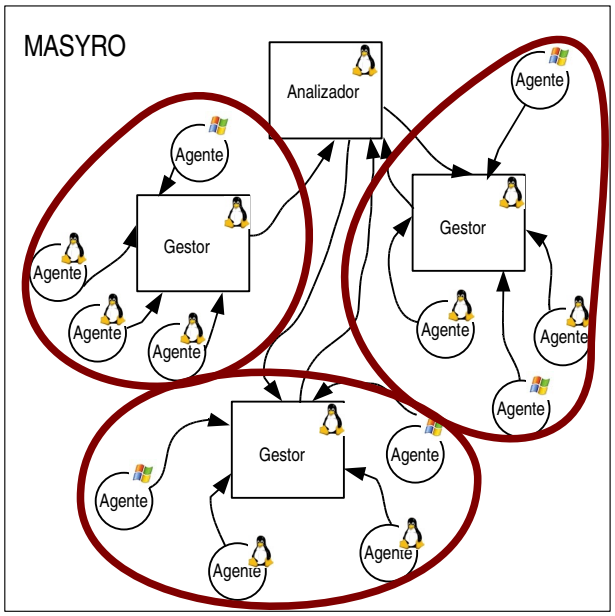

Figura 6.1: Arquitectura de MASYRO con múltiples grupos de trabajo.

6.2.2.3. Notificación del trabajo a los agentes subscritos al gestor

El proceso de notificación de un trabajo a los agentes subscritos a un gestor requiere que éstos accedan al repositorio de modelos con el objetivo de obtener el trabajo en cuestión. Normalmente, los trabajos no sólo incluyen el modelo representado tridimensionalmente, sino que también requieren de otro tipo de archivos externos, como por ejemplo texturas. Estos elementos incrementan notablemente el tamaño físico del trabajo en cuestión. Actualmente, MASYRO utiliza una máquina para ejecutar el servicio asociado al repositorio de modelos, por lo que si el repositorio está situado en una red doméstica, la carga de red implicaría que el tiempo empleado por los agentes para obtener el modelo sería demasiado elevado. Una posible mejora en relación a esta etapa sería utilizar algún tipo de mecanismo de distribución del modelo entre pares (esquema tipo *peer to peer*), es decir, que los propios agentes fueran los encargados de enviarse entre sí los distintos trozos que componen el trabajo. De este modo, se pasaría de una perspectiva centralizada a una perspectiva descentralizada. Un método con-

creto podría ser utilizar algún tipo de aplicación basada en el protocolo *peer-to-peer*, como por ejemplo *bittorrent* (aunque éste también hace uso del protocolo FTP).

6.2.2.4. Proceso de renderizado distribuido

Esta etapa del flujo de trabajo supone el eje central de MASYRO, por lo que el número de posibles propuestas es mayor en esta parte del proceso. Una importante fuente de propuestas estará directamente vinculada con la utilización de conocimiento experto mediante un sistema de reglas difuso. Este sistema permite realizar cambios de una manera directa y altamente escalable en lo relativo a los parámetros que intervienen en el proceso de renderizado.

La primera acción que lleva a cabo un agente especializado en el renderizado es obtener el modelo del repositorio. A continuación, carga el sistema de reglas difuso, el cual está especificado en el lenguaje XML. Actualmente, dicha especificación está pensada para utilizar el método de render definido por el usuario, pero optimizando el valor de distintos parámetros que intervienen en el proceso de render. Estos parámetros son los siguientes:

- El nivel de recursión de los rayos de luz empleado.
- El número de *samples* por luz.
- El tamaño de la banda de interpolación.

Llegados a este punto, el número de optimizaciones aplicables posibles se define en función de los cambios especificados en el sistema de reglas. Por ejemplo, se podría pensar en ajustar distintos parámetros que también intervienen en el proceso de renderizado:

- Resolución espacial de la imagen.
- Nivel de *oversampling*.
- Resolución del árbol octal en métodos de renderizado basados en *raytracing*.
- Parámetros de radiosidad, como el número de parches, número de elementos, tamaño máximo y mínimo de los mismos, etc.
- Número de muestras en métodos de iluminación global.

- Número de fotones.
- Radio de mezclado entre fotones.
- Profundidad de rebote en el interior de objetos con caústicas.
- Profundidad máxima de rebote de rayos secundarios en métodos de iluminación global.
- Número máximo de píxeles sin muestras, si se utiliza el método de *irradiance cache*.
- Nivel de refinamiento de sombras.
- Etc.

Otra posible línea de investigación sería ir un paso más alla en la definición del sistema de reglas, permitiendo que el método de renderizado empleado para cada unidad de trabajo fuese distinto. Sin embargo, esta característica tendría que ser cuidadosamente estudiada, para que la diferencia de calidad entre unidades de trabajo adyacentes no fuese apreciable por el usuario. Una vez más, se demuestra que la utilización de un sistema de reglas difuso permite representar fácilmente el conocimiento experto, a la vez que posibilita incluir nuevas mejoras de una forma muy escalable. Esta opción descrita sería una de las posibles, aunque también se podrían tener distintas especializaciones de agentes en función de los diferentes métodos de render utilizados actualmente. De este modo, cada agente tendría su propio sistema de reglas que representaría su propio conocimiento. Sin embargo, utilizando este enfoque habría que realizar un reparto de zonas empleando un estudio previo más exhaustivo, con el objetivo de que la imagen bidimensional final sea lo más aproximada posible al resultado ideal. Estas dos aproximaciones mencionadas supondrían mejoras muy importantes en términos de tiempo de ejecución, pero el modelado del problema debería hacerse de manera completa y concisa.

El siguiente paso tras la carga del sistema de reglas por parte del agente es llevar a cabo el renderizado *ligero* de las distintas unidades de trabajo asignadas por el gestor. Dicho renderizado, como se expuso en anteriores capítulos, consiste en un renderizad a baja resolución de la unidad de trabajo en cuestión. Las pruebas ejecutadas se han realizado al 10 % de la resolución predefinida por el usuario, es decir, se han ejecutado utilizando un parámetro estático. En este caso, quizás incluir algún tipo de optimización para variar este parámetro no

tenga mucho sentido, ya que el tiempo empleado para este tipo de renderizados es mínimo. Sin embargo, y como se estudiará posteriormente, lo que sí resulta interesante es establecer la relación de tiempo estimado que hay entre el renderizado a baja resolución y el renderizado final, es decir, el renderizado con la resolución predefinida.

Otro tema importante, y que puede enfocarse de muchas otras formas distintas, es el método de subasta de unidades de trabajo empleado en MASYRO. Actualmente, el enfoque utilizado para el mecanismo de subasta se basa en que los agentes siempre pujan por la unidad de trabajo más compleja que aún queda por realizar. La unidad más compleja se define en función de una ponderación, donde el 75 % del peso lo representa el tiempo de renderizado estimado para esa unidad de trabajo y el 25 % del peso lo representa el parámetro que define la complejidad de la misma. El por qué de esta elección se justificó en el capítulo dedicado a la metodología de trabajo, pero básicamente consiste en que terminando antes los trabajos parciales más complejos, la finalización del trabajo en su conjunto se realizará de forma más simétrica entre los distintos agentes que participan en dicha labor. El gestor es el mediador encargado de asignar las unidades de trabajo a los agentes que pujaron con anterioridad. Nuevamente, quizás un enfoque más adaptativo produciría mejores resultados, de forma que la ponderación del peso entre el tiempo estimado y la complejidad se determinara en tiempo de ejecución, en función del histórico de pujas de cada agente. Lo mismo ocurre con el proceso de premio-penalización, ya que un enfoque que ajustara los premios en función de las características del trozo y del agente en cuestión quizás se comportaría de una mejor forma, es decir, convergería mejor al comportamiento de subasta ideal.

6.2.2.5. Composición del resultado final

En cuanto al tema de la composición final de la imagen existen menos mejoras a comentar, ya que es un proceso más mecánico y, por lo tanto, da menos pie a utilizar nuevas alternativas. Sin embargo, si se podría estudiar un mecanismo que ajustase mejor el tamaño de la banda de interpolación de cada unidad de trabajo, y que encontrara la proporción adecuada para realizar la interpolación entre la zona común a dos unidades de trabajo.

Apéndice A

Anexo A

A.1. Código fuente

Debido la longitud del código fuente (en torno a 8000 líneas), a continuación se expondrá el código principal de MASYRO, comenzando por la definición de las interfaces en Slice y continuando con el código más relevante de MASYRO.

A.1.1. FIPA.ice

```
#ifndef _FIPA
#define _FIPA

module FIPA
{

/************************/
/*Estructuras generales.*/
```

```
/************************/

sequence <string> Sstring;

/*************************************************************/
/*Definición de la estructura básica para definir un servicio.*/
/*************************************************************/

// Estructura del tipo TServiceLocationDescription.
struct TServiceLocationDescription
{
   // ServiceSignature indica la firma vinculante a un servicio.
   string ServiceSignature;
   // ServiceAddress indica como unirse a un servicio.
   // Se entiende como la cadena de texto que representa al proxy
   // asociado al objeto que proporciona un servicio,
   // es decir, al objeto bien conocido.
   string ServiceAddress;
};

// Estructura del tipo TServiceLocator.
// Sirve para acceder y hacer uso de un servicio.
sequence <TServiceLocationDescription> TServiceLocator;

// La estructura TServiceDirectoryEntry es un conjunto de parámetros
// que definen a un servicio.
struct TServiceDirectoryEntry
{
   // ServiceType define el tipo de servicio.
   string ServiceType;
   TServiceLocator ServiceLocator;
   // ServiceId sirve para identificar un servicio de forma única dentro
   // de la plataforma de agentes.
   string ServiceId;
};

sequence <TServiceDirectoryEntry> TServiceDirectoryEntries;

struct TProperty
{
   // Name representa el nombre de la propiedad.
   string Name;
   // Value representa el nombre de la propiedad.
   string Value;
};

sequence <TProperty> TProperties;

/*****************************************/
/*Descripción del identificador de un agente.*/
/*****************************************/

// El AID es una colección extensible de parámetros que identifican a un agente.
struct TAID
```

```
{
   // Name identifica a un agente de manera única en la plataforma de agentes.
   // Es de la forma nombre-agente@plataforma-agentes.
   string Name;
   // Addresses es una lista de direcciones de transporte donde
   // un mensaje puede ser entregado.
   Sstring Addresses;
};



sequence <TAID> TAIDs;

/*********************************************************/
/*Descripción de un agente en el Agent Management System.*/
/*********************************************************/

enum EState {Initiated, Active, Suspended, Waiting, Transit};
enum EExplanation {Duplicate, Access, Invalid, Success, NotFound};

// AMSAgentDescription representa la información por la cual
// el AMS conoce a los agentes.
struct TAMSAgentDescription
{
   // Name representa el identificador del agente.
   TAID Name;
   // State representa el estado del agente.
   int State;
};

sequence <TAMSAgentDescription> TAgentDirectory;

/*********************************************************/
/*Descripción de un servicio en el DirectoryFacilitator.*/
/*********************************************************/

struct TDFServiceDescription
{
   // Name representa el nombre del servicio.
   string Name;
   // Type representa el tipo de servicio.
   string Type;
};

sequence <TDFServiceDescription> TDFServiceDescriptions;

/*********************************************************/
/*Descripción de un agente en el Directory Facilitator.*/
/*********************************************************/

struct TDFAgentDescription
{
   // Name es el identificador del agente.
```

```
   TAID Name;
   // Services es una lista de servicios soportados por el agente.
   TDFServiceDescriptions Services;
   // Protocols es una lista de protocolos de interacción
   // soportados por el agente.
   Sstring Protocols;
   // Ontologies es una lista de ontologías soportadas por el agente.
   Sstring Ontologies;
   // Languages es una lista de lenguajes de contenido
   // soportados por el agente.
   Sstring Languages;
   // Lease-Time representa el tiempo en segundos que dura
   // el registro del agente.
   int LeaseTime;
   // Scope define la visibilidad de la descripción
   // del agente en el DirectoryFacilitator.
   Sstring Scope;
};

sequence <TDFAgentDescription> TDFAgentDescriptions;

/*********************************************/
/*Elementos de Message Transport Specification*/
/*********************************************/

// Estructura para representar una fecha.
struct TDate
{
   int hour;
   int minutes;
   int seconds;
   int day;
   int month;
   int year;
};

// Estructura para representar el "sobre" del mensaje.
struct TEnvelope
{
   TAIDs To;
   TAID From;
   TDate Date;
   int ACLRepresentation;
};

// Estructura para representar un mensaje.
struct TMessage
{
   TEnvelope Envelope;
   string Payload;
};

// Representaciones del mensaje ACL.
enum EAclRepresentation {bitefficientRep,
```

```
stringRep,
xmlRep};

/**************************/
/*Definición de excepciones*/
/**************************/

// Excepción base.
exception BaseException
{
   string Reason;
};

/**************************/
/*Descripción de interfaces.*/
/**************************/

// StartService es el servicio que provee de los servicios básicos
// a un agente cuando éste comienza su ejecución.
interface StartService
{

   // La operación getBasicServices permite que un agente
   // descubra los servicios básicos.
   // como el AMS, el MTS, o el DF.
   nonmutating TServiceDirectoryEntries getServiceRoot();
   // La operación supplyBasicService permite que un servicio
   // notifique al StartService que es un servico básico .
   void supplyBasicService(TServiceDirectoryEntry sde);

};

enum Matching {SAME, ANY};

// Agent Management System es el servicio controlador
// de la plataforma de agentes.
interface AMS
{

   // La operación register permite que un agente
   se registre en la plataforma de agentes.
   void register(TAID aid,
      out int explanation, out string newName, out int state);
   // La operación deregister permite que un agente elimine su registro
   // en la plataforma de agentes.
   void deregister(TAID aid, out int explanation);
   // La operación modify permite que un agente modifique sus datos en el AMS.
   idempotent void modify(TAID aid, out int explanation);
   // La operación search permite buscar uno o varios agentes
   // según unos criterios.
   nonmutating void search(TAID aid, int match,
      out int explanation, out TAIDs aids);
   // La operación getDescription permite obtener la descripción del AP.
   nonmutating string getDescription();
```

```
};

enum DFOperation {REGISTER, DEREGISTER, MODIFY};

// Agent representa a un agente en la plataforma de agentes.
interface Agent
{

   // La operación suspend permite suspender la ejecución de un agente.
   idempotent void suspend();
   // La operación terminate permite terminar la ejecución de un agente.
   void terminate();
   // La operación resume permite reanudar la ejecución de un agente.
   idempotent void resume();
   // La operación receiveACLMessage permite recibir un mensaje ACL.
   void receiveACLMessage(string ACLMessage);

};

// DirectoryFacilitator es un servicio de páginas amarillas
// dentro de la plataforma de agentes.
interface DirectoryFacilitator
{

   // La operación register permite que un agente
   // se registre en el DirectoryFacilitator.
   void register(TDFAgentDescription ad, out int explanation);
   // La operación deregister permite que un agente
   // elimine su registro en el DirectoryFacilitator.
   void deregister(TDFAgentDescription ad, out int explanation);
   // La operación modify permite que un agente
   // modifique sus datos en el DirectoryFacilitator.
   idempotent void modify(TDFAgentDescription ad, out int explanation);
   // La operación search permite buscar uno o varios agentes
   // según unos criterios.
   nonmutating void search(TDFAgentDescription ad, int match,
      out int explanation, out TDFAgentDescriptions ads);

};

// Agent Communication Channel provee el servicio
// del Message Transport Service en la plataforma de agentes.
interface ACC
{

   // La operación receive permite recibir un mensaje.
   int receive(TMessage message);

};

};

#endif
```

A.1.2. MASYRO.ice

```
#ifndef _MASYRO
#define _MASYRO

module MASYRO
{

/************************/
/*Estructuras generales.*/
/************************/

// Estructura que define una zona a renderizar.
struct TZone {
   // Identificador de la zona.
   int id;
   // Coordenadas que definen la zona.
   int x1;
   int y1;
   int x2;
   int y2;
   // Desviación estándar del color de la zona.
   float d;
   // Media de color en la zona.
   float m;
};

sequence <TZone> TZones;

enum StateRegister {Done, NotDone, InWork};

// Estructura que define un registro en la arquitectura de pizarra.
struct TRegister {
   // Identificador del trabajo.
   int IdWork;
   // Identificador de la unidad de trabajo (zona de la imagen).
   int WorkUnit;
   int Size;
   // Complejidad del trozo.
   int Comp;
   // Tiempo estimado por el agente tras un render "tipo sello"
   // (tiempo empírico).
   int Test;
   // Tiempo final empleado por el agente.
   int Treal;
   // Nombre del agente encargado de la unidad de trabajo.
   string Agent;
   // Variable que indica si se completó la unidad de trabajo.
   StateRegister State;
   // Ibs representa el tamaño de la banda de interpolación.
   int Ibs;
   // Ls representa el número de samples por luz.
   int Ls;
   // Rl representa el nivel de recursión.
```

```
   int Rl;
};

sequence <TRegister> TRegisters;

sequence <byte> ByteSeq;
sequence <int> IntSeq;

/**************************/
/*Definición de excepciones*/
/**************************/

// Excepción base.
exception BaseException
{
   string Reason;
};

// Excepción lanzada en caso de que al solicitar un modelo éste no exista.
exception ModelNotExistsException extends BaseException
{
   IntSeq ExistingModels;
};

// Excepción lanzada en caso de que no exista un registro
// a la hora de actualizarlo.
exception RegisterNotExistsException extends BaseException
{
};

/**************************/
/*Descripción de interfaces.*/
/**************************/

// Analyst es el servicio encargado de llevar a cabo el análisis de la entrada.
interface Analyst
{

   // La operación processWork procesa un trabajo asociado a una escena.
   // work representa el flujo de bytes asociado al trabajo.
   // workName es el nombre dado al trabajo.
   // level representa el número de pasadas para llevar a cabo
   // la división de la imagen inicial en trozos.
   // optimization es el nivel de optimización definido por el usuario.
   void processWork(ByteSeq work, string workName, int level, int optimization);

};

// RenderAgent es la interfaz a un agente especializado en el render.
interface RenderAgent
{

   // La operación notifyNewWork permite al RenderAgent
   // conocer la existencia de un nuevo trabajo.
```

```
   void notifyNewWork(TZones zones, int idWork, int benchmarkValue);
   // La operación notifyZone asigna una lista de zonas del nuevo trabajo
   // al RenderAgent, para que éste haga un estudio previo.
   // optimization ==> Nivel de optimización definido por el usuario (1-5).
   ["ami"] void notifyZones(TZones zones, int idWork, int optimization);
   // La operación beginRenderProcess notifica el comienzo del trabajo.
   void beginRenderProcess();
   // La operación render notifica al agente el comienzo del renderizado
   // para la zona idZone del trabajo idWork.
   ["ami"] void render(int idWork, int idZone, string agent);

};

// Master es el servicio publicador de la existencia
// de nuevos trabajos en la plataforma.
interface Master
{

   // La operación subscribe permite a un agente especializado
   // en el renderizado subscribirse al Master.
   void subscribe(string agentName, RenderAgent* agent);
   // La operación unsubscribe permite a un agente especializado
   // en el renderizado darse de baja con el Master.
   void unsubscribe(string agentName);

   // La operación notifyNewWork permite al Master conocer
   // la existencia de un nuevo trabajo.
   // optimization ==> Nivel de optimización definido por el usuario (1-5).
   void notifyNewWork(TZones zones, int idWork, int optimization);
   // La operación benchmarkValue incrementa el valor medio del
   // tiempo empleado en la ejecución del benchmark.
   void benchmarkValue(int value);

   // La operación bidHigher permite a un agente pujar por un trozo.
   ["ami"] void bidHigher(string agent, int idWork, int idZone,
      int credits, IntSeq historic);
   // La operación noMoreBiddings le indica al Master
   // la posible finalización del trabajo.
   void noMoreBiddings();
   // La operación giveFinalImage le proporciona al Master
   // un trozo de la imagen final.
   void giveFinalImage(int idWork, int idZone, ByteSeq partialImage,
      int x1, int y1, int x2, int y2, int ibs);

   // La operación showAgentsState devuelve el estado de los agentes.
   nonmutating string showAgentStates();
   // La operación getLog devuelve el log del Master.
   nonmutating string getLog();

};

// ModelRepository es la interfaz relativa
// al almacenamiento/recuperación de modelos.
interface ModelRepository
```

```
{

   // put devuelve el identificador asignado al modelo enviado.
   int put(string name, ByteSeq model);
   // get devuelve el nombre del modelo y
   // el propio modelo como secuencia de bytes.
   nonmutating string get(int idModel, out ByteSeq model)
      throws ModelNotExistsException;

};

// Blackboard es la interfaz relativa a la arquitectura de pizarra.
interface Blackboard
{

   // write escribe un registro en la pizarra.
   void write(TRegister register);
   // read lee un registro de la pizarra, identificado por
   // el id del trabajo y el id de la zona.
   TRegister read(int idWork, int workUnit)
      throws RegisterNotExistsException;
   // update actualiza el valor de un registro en la pizarra.
   void update(int idWork, int workUnit, int test)
      throws RegisterNotExistsException;
   // clear limpia la pizarra.
   void clear();

   // setAnalysisTime establece el tiempo empleado para
   // el análisis de la escena más reciente.
   void setAnalysisTime(int time);
   // getAnalysisTime devuelve el tiempo empleado
   // para el análisis de la escena más reciente.
   nonmutating int getAnalysisTime();
   // incrementEstimatedRenderTime incrementa el tiempo
   // de estimación de la escena.
   void incrementEstimatedRenderTime(int time);
   // getEstimatedRenderTime devuelve el tiempo de
   // estimación de la escena.
   nonmutating int getEstimatedRenderTime();

   // isWorkEstimated indica si las distintas partes
   // del trabajo se han estimado.
   bool isWorkPartiallyEstimated();
   // isEnd indica si el trabajo actual ha terminado.
   bool isCurrentWorkFinished();
   // show muestra el contenido de la pizarra.
   string show();
   // getMaxTest devuelve el valor del trozo cuyo
   // tiempo estimado es el mayor del trabajo idWork.
   int getMaxTest(int idWork);
   // getMaxComp devuelve el valor del trozo
   // cuya complejidad es la mayor del trabajo idWork
   int getMaxComp(int idWork);
```

```
   // setWorkUnit permite a un agente hacerse cargo
   // de una unidad de trabajo.
   void setWorkUnit(int idWork, int workUnit, string agent)
      throws RegisterNotExistsException;
   // finishWork permite a un agente notificar
   // la finalización de una unidad de trabajo.
   void finishWorkUnit(int idWork, int workUnit, int treal,
      int ibs, int ls, int rl) throws RegisterNotExistsException;

};

};

#endif
```

A.1.3. Proceso de división de unidades de trabajo por parte del analista

```
def divideZone (self, zone):
    """Divide la zona pasada como parametro o
    la almacena como zona no divisible"""

    x1, y1 = zone.getX1(), zone.getY1()
    x2, y2 = zone.getX2(), zone.getY2()

    # Estudiamos si es divisible.
    med, d = self.getMediumStandardDesviation(zone)

    # Si la desviacion estandar es mayor que la establecida
    # y el trozo es lo suficientemente grande, se divide.
    if (d > DESV) and zone.getWidth() > self.getSmallestPieceX() and
       zone.getHeight() > self.getSmallestPieceY():
       self.divideZone(Zone(-1, x1, y1, x1 + (x2 - x1) / 2, y1 + (y2 - y1) / 2))
       self.divideZone(Zone(-1, x1 + (x2 - x1) / 2, y1, x2, y1 + (y2 - y1) / 2))
       self.divideZone(Zone(-1, x1, y1 + (y2 - y1) / 2, x1 + (x2 - x1) / 2, y2))
       self.divideZone(Zone(-1, x1 + (x2 - x1) / 2, y1 + (y2 - y1) / 2, x2, y2))

    # Si la zona no es divisible, la insertamos en la lista de zonas.
    else:
       zone = Zone(self.getId(), x1, y1, x2, y2)
       zone.setDesv(d)
       zone.setMed(med)
       self.addZone(zone, self.getInitialZones())
```

A.1.4. Notificación de un nuevo trabajo al gestor

```
def notifyNewWork (self, zones, idWork, optimization, current = None):
    """Obtiene la información asociada a un trabajo:
    zones representa la división en zonas e idWork es el id del trabajo"""

    self.openLogFile()
    self.getLogFile().write(self.getServiceId() + ' -->
       Recibido un nuevo trabajo con id ' + str(idWork) + '\n')
    # Obtención de los proxies a los servicios necesarios.
    self.obtainProxies()
    # Limpieza de la pizarra y de las anteriores apuestas.
    self.getBlackboard().clear()
    self.clearBiddings()
    # Se establecen los valores para las zonas de trabajo actuales.
    self.setCurrentZones(zones)
    self.setCurrentIdWork(idWork)
    self.setCurrentImage()
    # Aún no ha llegado ningún trozo de la imagen final.
    self.setNumberOfPieces(0)

    # Escritura en la pizarra del nuevo trabajo.
    self.writeWorkInBlackboard(idWork)
    # Notificación de un nuevo trabajo a los agentes subscritos a este Master.
    self.notifyNewWorkToAgents(zones, idWork, int(self.getBenchmarkValue() /
```

```
       len(self.getRenderAgents())))
    # Reparto inicial de las zonas del trabajo actual entre los
    # agentes especializados en render subscritos.
    self.initialDistribution(optimization)

def notifyNewWorkToAgents (self, zones, idWork, benchmarkValue):
    """Notifica un nuevo trabajo a los agentes subscritos al master"""

    for prxAgent in self.getRenderAgents().values():
        prxAgent.notifyNewWork(zones, idWork, benchmarkValue)

def initialDistribution (self, optimization):
    """Lleva a cabo la distribución inicial de zonas
    entre los agentes especializados en el render"""

    # Se distribuyen los trozos a los agentes para el render tamaño "sello".
    # La distribución se hace en grupos de trozos.
    zonesPerBlock = max(int(len(self.getCurrentZones()) /
       len(self.getRenderAgents())), 1)

    for a in range(len(self.getRenderAgents())):

        if a <> len(self.getRenderAgents()) - 1:
            zones = self.getCurrentZones()
       [a * zonesPerBlock: (a + 1) * zonesPerBlock]
        else:
            zones = self.getCurrentZones()
       [a * zonesPerBlock: len(self.getCurrentZones())]

        agentPrx = self.getRenderAgents().values()[a]
        cb = AMI_Model_notifyZonesI()
        agentPrx.notifyZones_async(cb, zones,
    self.getCurrentIdWork(), optimization)

    # Comienza el proceso de subasta (se notifica a todos los agentes).
    self.beginRenderProcess()
```

A.1.5. Proceso de renderizado por parte de un agente

```
def beginRenderProcess (self, current = None):
    """Notifica el comienzo del proceso de puja-render"""

    # Lectura de los registros de la pizarra para pujar por una zona.
    idMostComplexZone, maxComplex = -1, -1
    for x in self.getCurrentZones():
        register = self.getBlackboard().read(self.getCurrentIdWork(), x.id)
        # Zona más compleja ==> 75% peso Test 25% peso Comp.
        # Si la zona es más compleja que la actual
        # y no ha sido cogida por ningún agente...
        maxTest = self.getBlackboard().getMaxTest(self.getCurrentIdWork())
        maxComp = self.getBlackboard().getMaxComp(self.getCurrentIdWork())
        currentZoneComplexity = self.getComplexityRatio(register.Test,
```

```
            maxTest, register.Comp, maxComp)
        if currentZoneComplexity > maxComplex and register.Agent == '':
            idMostComplexZone = x.id
            maxComplex = currentZoneComplexity

        # El agente puja por el trozo más complejo.
        if idMostComplexZone != -1:
            print self.getName() + ' --> Pujando por ' +
                str(self.getCurrentIdWork()) + ' ' + str(idMostComplexZone)
            cb = AMI_Model_bidHigherI()
            self.getMaster().bidHigher_async(cb,
                self.getName(), self.getCurrentIdWork(),
                idMostComplexZone, self.getCredits(), self.getHistoric())
        # En caso de que no pueda pujar, lo notifica al Master.
        else:
            self.getMaster().noMoreBiddings()
```

A.1.6. Renderizado inicial para estimar la complejidad de la escena

```
import Blender
from Blender import *
from Blender.Scene import Render

editmode = Window.EditMode()    # If we are in edit mode; exit edit mode
if editmode: Window.EditMode(0)

world = World.GetCurrent()
if (world == None):
world = World.New("")

world.setHor([0.0, 0.0, 0.0])   # Black colour for Horizont

scn = Scene.GetCurrent()
context = scn.getRenderingContext()
context.setRenderPath("//")
context.setImageType(Render.PNG)
context.enableGrayscale()

context.enableRayTracing(0)
context.setRenderer(Render.INTERNAL)

nameList = NMesh.GetNames()
complexList = {}
maxValue = 0
j = 1

for mat in Material.Get():
    compDict = {'RayMirrorLevel': 0, 'RayTranspLevel':0}

    # Removes all textures of the material
    texture = Texture.New()
    texture.setType('None')
```

```
    for i in range(0,9):
        mat.setTexture(i, texture)

        # Obtaining poperties for complexity
        if (mat.mode & Material.Modes.RAYMIRROR):
            compDict['RayMirrorLevel'] = mat.getMirrDepth()

        if (mat.mode & Material.Modes.RAYTRANSP):
            compDict['RayTranspLevel'] = mat.getTransDepth()

        # Calculates the material complexity
        complexList[j] = compDict['RayMirrorLevel'] + compDict['RayTranspLevel']
        if (complexList[j] > maxValue):
            maxValue = complexList[j]
        j = j+1

j = 1
for mat in Material.Get():
    mat.setMode()   # Remove all flags (even UV Mapping)
    mat.setAlpha(1)
    mat.emit = 0
    complexity = (complexList[j] * 1.0) / maxValue * (1.0)
    print complexity
    mat.setRGBCol (complexity, complexity, complexity)
    mat.mode |= Material.Modes.SHADELESS
    j = j+1
```

Apéndice B

Anexo B

B.1. Figuras

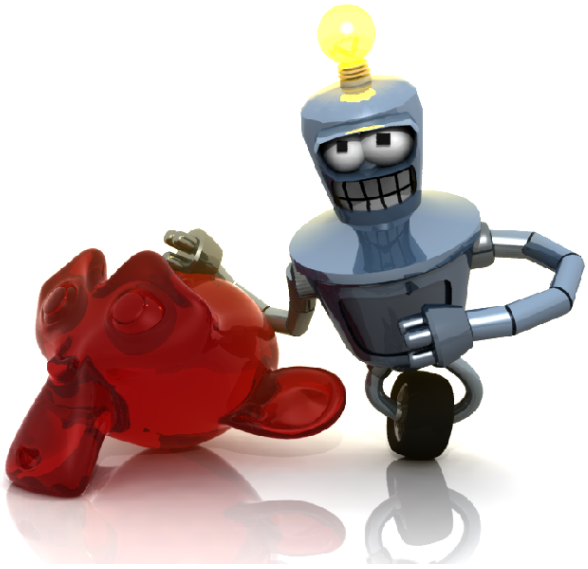

Figura B.1: Imagen renderizada sin aplicar las optimizaciones de MASYRO

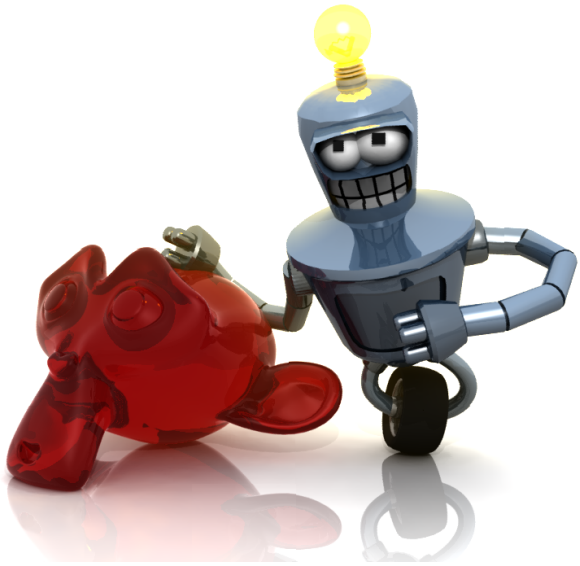

Figura B.2: Imagen renderizada utilizando MASYRO con nivel de optimización 1

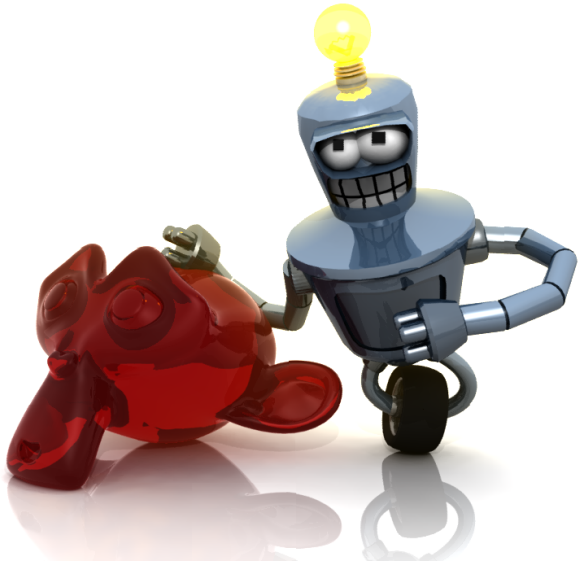

Figura B.3: Imagen renderizada utilizando MASYRO con nivel de optimización 2

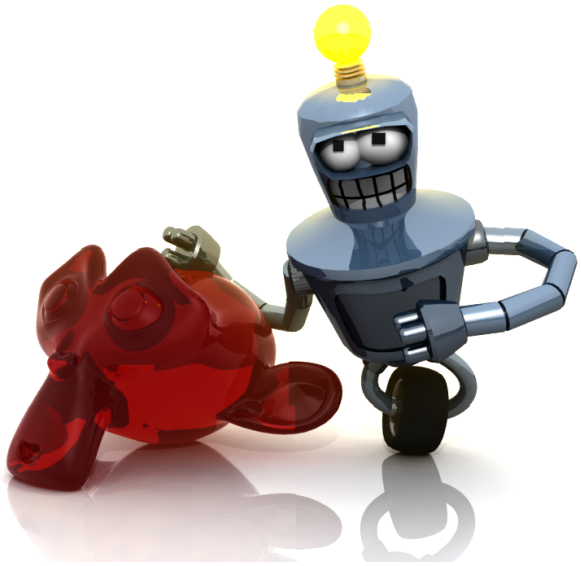

Figura B.4: Imagen renderizada utilizando MASYRO con nivel de optimización 3

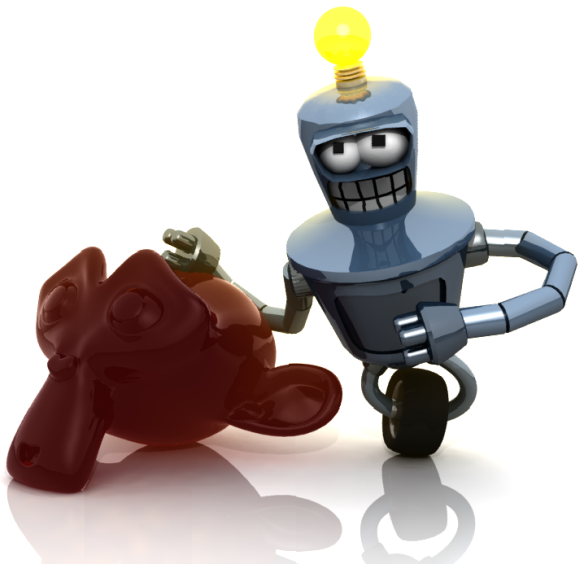

Figura B.5: Imagen renderizada utilizando MASYRO con nivel de optimización 4

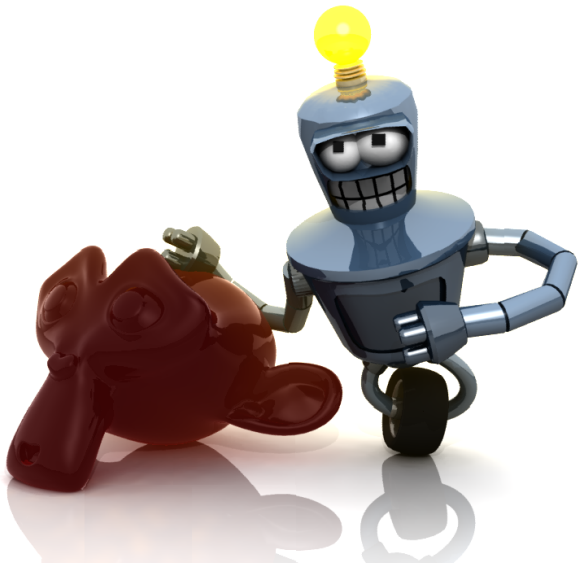

Figura B.6: Imagen renderizada utilizando MASYRO con nivel de optimización 5

Apéndice C

Anexo C

C.1. Manual de usuario

En esta sección se expondrán una serie de instruccionas para desplegar MASYRO en cualquier máquina con conexión a Internet. El usuario solamente ha de ajustar unos parámetros de configuración tanto para arrancar los servicios como para arrancar los agentes. Junto con el código fuente se incluye el siguiente *script (initMASYRO.sh)* para inicializar MASYRO:

```
#!/bin/sh

echo ""
echo "Script to automate the MASYRO deployment"
echo ""

# Compilación del código fuente.
slice2cpp --output-dir ../FIPA/generated/ ../FIPA/FIPA.ice
make clean
echo ""
echo "Compiling the source code..."
make
echo ""
```

Para arrancar MASYRO se utiliza el *script startMASYRO.sh*:

```
#!/bin/sh

echo ""
echo "Script to start MASYRO"
echo ""

# Arrancando el nodo y el registro.
echo "Starting the node and the registry..."
icegridnode --Ice.Config=config/icegrid.cfg --daemon --nochdir
sleep 5
echo "Deploying MASYRO..."

# Desplegando la aplicación
icegridadmin --Ice.Config=config/icegridadmin.cfg
   -e "application add ./init/MASYRO.xml"
icegridadmin --Ice.Config=config/icegridadmin.cfg
   -e "server start AMS"
icegridadmin --Ice.Config=config/icegridadmin.cfg
   -e "server start DirectoryFacilitator"

# Arrancando glacier2router
echo ""
echo "Starting glacier2router"
glacier2router --Ice.Config=config/router.cfg
```

Por defecto, y utilizando este *script*, todos los servicios de MASYRO se ejecutarán en la misma máquina en la que se ejecute el *script*. Sin embargo, si es necesario ajustar el archivo de configuración *router.cfg*, de forma que se especifiquen la dirección IP pública del equipo que ofrece la salida a Internet, y la IP privada en la que se ejecutan los servicios. Dicho archivo ha de quedar de la siguiente forma:

```
Glacier2.Client.Endpoints=tcp -h <ip_privada> -p <puerto_maquina_ip_privada>
Glacier2.Client.PublishedEndpoints=tcp -h <ip_publica> -p <puerto_publico>
Glacier2.Server.Endpoints=tcp -h <ip_privada>
Glacier2.CryptPasswords=config/passwords.cfg

Ice.Trace.Network=2

Ice.Default.Locator=IceGrid/Locator:tcp -h 127.0.0.1 -p 10000

#Other properties
Ice.Warn.Leaks=0
Ice.MessageSizeMax=20480
```

Para detener la aplicación basta con ejecutar el *script (stopMASYRO.sh)* siguiente:

```
#!/bin/sh

echo ""
echo "Script to stop MASYRO"
echo ""

echo ""
echo "Closing the MASYRO application..."
icegridadmin --Ice.Config=config/icegridadmin.cfg -e "application remove MASYRO"
echo "Shutdowning localhost..."
icegridadmin --Ice.Config=config/icegridadmin.cfg -e "node shutdown localhost"
```

Para iniciar un agente simplemente es necesario ejecutar la siguiente orden:

```
python RenderAgent.py -i <xml_specification_file>
```

La opción *-i* permite al usuario indicar el fichero de configuración para llevar a cabo la inicialización del agente.

Bibliografía

[1] delegados.dat.etsit.upm.es/ abarbero/curso/xml/xmltutorial.html.

[2] FIPA00001 FIPA Abstract Architecture Specification. Foundation for Intelligent Physical Agents, 2000.

[3] FIPA00011 FIPA RDF Content Language Specification. Foundation for Intelligent Physical Agents, 2001.

[4] FIPA00023 FIPA Agent Management Specification. Foundation for Intelligent Physical Agents, 2004.

[5] FIPA00026 FIPA Request Interaction Protocol Specification. Foundation for Intelligent Physical Agents, 2002.

[6] FIPA00035 FIPA Subscribe Interaction Protocol Specification. Foundation for Intelligent Physical Agents, 2002.

[7] FIPA00067 FIPA Agent Message Transport Service Specification. Foundation for Intelligent Physical Agents, 2002.

[8] http://docs.python.org/lib/module-xml.dom.minidom.html.

[9] http://es.wikipedia.org.

[10] http://grasia.fdi.ucm.es/ingenias/spain/index.php.

[11] http://jade.tilab.com/.

[12] http://java.sun.com/products/jdk/rmi/.

[13] http://macr.cis.ksu.edu/projects/mase.htm.

[14] http://setiathome.berkeley.edu.

[15] http://www-leibniz.imag.fr/magma/publications/index.php?author=demazeau.

[16] http://www.blender.org.

[17] http://www.corba.org.

[18] http://www.fipa.com.

[19] http://www.indigorenderer.com.

[20] http://www.mentalimages.com.

[21] http://www.microsoft.com/spanish/msdn/articulos/archivo/091101/voices/remoting.asp.

[22] http://www.omg.org.

[23] http://www.splutterfish.com/sf/.

[24] http://www.toxicengine.org/index.php.

[25] http://www.w3.org/dom/.

[26] http://www.w3.org/rdf/.

[27] http://www.w3.org/tr/ws-arch/gengag.

[28] http://www.w3.org/tr/xml11/.

[29] http://www.yafray.org.

[30] http://www.zeroc.com.

[31] http://xml.apache.org/xerces-c/.

[32] *Tecnologías libres para Síntesis de Imagen Digital Tridimensional.* 2006.

[33] W.J. Bouknight. A procedure or generation of three-dimensional half-tone computer graphics. *Communications of the ACM*, 1970.

[34] B. Burmeister. Models and methodologies for agent-oriented analysis and design. 1996.

[35] et al C. Goral. Modelling the interaction of light between diffuso surfaces. *Proceedings of SIGGRAPH*, 1984.

[36] M. Georgeff D. Kinny. Modelling and design of multi-agent systems. 1996.

[37] EURESCOM. MESSAGE: Methodology for engineering systems of software agents. 2001.

[38] C. A. Iglesias Fernández. Definición de una metodología para el desarrollo de sistemas multi-agente. 1998.

[39] Michi Henning. To Slice or Not to Slice. *Connections*, 2005.

[40] Michi Henning. The Rise and Fall of Corba. 2006.

[41] B. Bauer J. Odell, H. V. D. Parunak. Representing Agent Interaction Protocols in uml. 2000.

[42] H.W. Jensen. Global Illumination using photon map. *Eurographics Rendering Workshop*, 1996.

[43] H.W. Jensen. *Realistic Image Synthesis using Photon Mapping.* 2001.

[44] Botti J. Vicente Julián J. Vicente. Estudio de métodos de desarrollo de sistemas multiagente. 1998.

[45] J.T. Kajiya. The rendering equation. *Computer Graphics*, 1986.

[46] J. Lind. MASSIVE: Software Engineering on Multi-agent Systems. 1999.

[47] F. Garijo et al M. Belmonte, V. J. Botti. *Agentes Software y Sistemas Multiagente. Conceptos, arquitecturas, y aplicaciones. Ana Mas.* Prentice Hall, 2005.

[48] M. Spruiell M. Henning. *Distributed Programming with Ice.* 2006.

[49] D. Kinny M. Wooldridge, N. R. Jennings. The GAIA methodology for agent-oriented analysis and design. 1998.

[50] M. Wooldridge N. R. Jennings. Agent-oriented software engineering. Handbook of Agent Technology. 2000.

[51] Nils J. Nilsson. *Inteligencia Artificial. Una nueva síntesis.* Mc Graw Hill, 2001.

[52] L. Carpenter R.L. Cook, T. Porter. Distributed ray tracing. *Computer Graphics*, 1984.

[53] G. Kronin S. Zhukov, A. Iones. An ambient light illumination model. *Proc. of Eurographics Rendering Workshop '98*, 1998.

[54] José Antonio Fernández Sorribes. *Yafrid: Sistema Grid para Render.* 2006.

[55] PeterÑorvig Stuart J. Russell. *Artificial Intelligence. A Modern Approach.* Prentice Hall, 1995.

[56] J. J. Buckley W. Siler. *Fuzzy Expert Systems and Fuzzy Reasoning.* 2005.

[57] Gerhard Weiss. *Multiagent Systems. A Modern Approach to Distributed Modern Approach to Artificial Intelligence.* 1999.

[58] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 1980.

[59] M. F. Wood. Multiagent Systems Engineering:A Methodology for Analysis and Design of Multiagent System. Air Force Institute of Technology. 2000.

[60] L. A. Zadeh. Fuzzy Sets. 1965.