Universitat d'Alacant
Universidad de Alicante

MINERVA: Modelo Integral para Entorno de Realidad Virtual y Agentes

Gabriel López García

Tesis Doctorales

UNIVERSIDAD de ALICANTE

www.eltallerdigital.com

MINERVA

Modelo Integral para Entornos de Realidad Virtual y Agentes

Gabriel López García

Universitat d'Alacant
Universidad de Alicante

Universidad de Alicante

DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL

TESIS DOCTORAL:

Minerva

Modelo Integral para Entornos de Realidad Virtual y Agentes

Universitat d'Alacant
Universidad de Alicante

Autor

Gabriel López García

Director

Rafael Molina Carmona

Alicante, septiembre de 2014

Universitat d'Alacant
Universidad de Alicante

Resumen

Uno de los problemas más importantes en los sistemas de realidad virtual es la diversidad de los dispositivos visuales y de interacción que existen en la actualidad. Junto a esto, la heterogeneidad de los motores gráficos, los motores físicos y los motores de inteligencia artificial, propicia que no exista un modelo que aúne todos estos aspectos de una forma integral y coherente. Con el objetivo de unificar toda esta diversidad, presentamos un modelo formal que afronta de forma integral el problema de la diversidad en los sistemas de realidad virtual, así como la definición de los módulos principales que los constituyen.

El modelo propuesto se basa en la definición de varias gramáticas que integra su actividad, su visualización y su interacción con el usuario.

La descripción de un mundo se presenta como una cadena de un lenguaje L(M) que representa una secuencia ordenada de agentes que aceptan eventos y que definen la actividad del sistema. Por otro lado, la visualización del estado actual del sistema se traduce en otra cadena del lenguaje de visualización L(V) y que está compuesta de primitivas y transformaciones.

Los conceptos de primitiva y transformación son mucho más amplios de lo que es habitual en estos sistemas. Las primitivas no son simples primitivas de dibujo sino acciones que se deben ejecutar en un determinado sistema de representación que es definida mediante otro lenguaje L(G) y que representa las acciones ordenadas en un determinado dispositivo visual o no. Las transformaciones, por otro lado, modifican las primitivas, dependiendo de su naturaleza.

Los agentes desarrollan una o varias actividades en el mundo virtual, se visualizan mediante primitivas y transformaciones, y usan eventos que también se definen en sentido amplio.

El modelo presentado tiene como virtud la separación de la actividad del sistema de los dispositivos visuales y de interacción concretos que lo componen. Esto supone varias ventajas: los dispositivos pueden ser sustituidos por otros dispositivos o por simulaciones de estos, los elementos del sistema pueden ser reutilizados, y la representación gráfica puede ser diferente dependiendo del dispositivo visual.

En definitiva, se ha pretendido diseñar un sistema integral, adaptativo, reutilizable y genérico.

Por último, se presentan varios casos prácticos que permiten concretar cómo se utiliza el modelo propuesto.

Universitat d'Alacant
Universidad de Alicante

Abstract

One of the major problems in virtual reality systems is the diversity of visual and interaction devices today. Besides, the heterogeneity of the graphics engines, physics engines and artificial intelligence engines, does not favour a model that combines all these aspects in a comprehensive and coherent manner. In order to unify all this diversity, we present a formal model that comprehensively faces the problem of diversity in virtual reality systems, and the definition of the main modules that constitute them.

The proposed model is based on the definition of several grammars that integrates its activities, visualization and user interaction.

The world description is presented as a string of a language L(M) representing an ordered sequence of events and agents that accept events and define system activity. Furthermore, for displaying the current state of the system, the string is translated into another string of the display language L(V), which is composed of primitives and transformations.

The primitive and transformation concepts are broader than is usual in these systems. The primitives are not just simple drawing primitives but actions to be executed in a given display system. It is defined by another language L(G), representing the actions ordered in a particular visual or not visual device. The transformations, moreover, modify primitives, depending on their nature.

The agents develop one or more activities in the virtual world, visualized by primitives and transformations, and use events that are also defined broadly.

The model has the virtue that it separates system activity from visual devices and concrete component interaction. This has several advantages: the devices can be replaced by other devices or simulations, system components can be reused, and the graphical devices may be different depending on the visual device.

In short, we have designed a comprehensive, adaptive, reusable and generic system.

Finally, we present several examples that clarify how the proposed model is used is presented.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

Resum

Un dels problemes més importants en els sistemes de realitat virtual és la diversitat dels dispositius visuals i d'interacció que existeixen en l'actualitat. Al costat d'això, l'heterogeneïtat dels motors gràfics, els motors físics i els motors d'intel·ligència artificial, propicia que no existeixi un model que uneixi tots aquests aspectes d'una manera integral i coherent. Amb l'objectiu d'unificar tota aquesta diversitat, presentem un model formal que afronta de manera integral el problema de la diversitat en els sistemes de realitat virtual, així com la definició dels mòduls principals que els constitueixen.

El model proposat es basa en la definició de diverses gramàtiques que integra la seva activitat, la seva visualització i la seva interacció amb l'usuari.

La descripció d'un món es presenta com una cadena d'un llenguatge L(M) que representa una seqüència ordenada d'agents que accepten esdeveniments i que defineixen l'activitat del sistema. D'altra banda, per a la visualització de l'estat actual del sistema es tradueix en una altra cadena del llenguatge de visualització L(V) i que està composta de primitives i transformacions.

Els conceptes de primitiva, transformació són molt més amplis del que és habitual en aquests sistemes. Les primitives no són simples primitives de dibuix sinó accions que s'han d'executar en un determinat sistema de representació que és definit mitjançant un altre llenguatge L(G) i que representa les accions ordenades en un determinat dispositiu visual o no. Les transformacions, d'altra banda, modifiquen les primitives, depenent de la seva naturalesa.

Els agents desenvolupen una o diverses activitats en el món virtual, es visualitzen mitjançant primitives i transformacions, i usen esdeveniments que també es defineixen en sentit ampli.

El model presentat té com a virtut la separació de l'activitat del sistema dels dispositius visuals i d'interacció concrets que el componen. Això suposa diversos avantatges: els dispositius poden ser substituïts per altres dispositius o per simulacions d'aquests, els elements del sistema poden ser reutilitzats, i la representació gràfica pot ser diferent depenent del dispositiu visual.

En definitiva, s'ha pretès dissenyar un sistema integral, adaptatiu, reutilitzable i genèric.

Finalment, es presenten diversos casos pràctics que permeten concretar com s'utilitza el model proposat.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

“Un informático es un matemático especializado
en sistemas formales.”

Edsger Wybe Dijkstra

Universitat d’Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

Agradecimientos

Esta tesis no podría haber salido a luz si durante su realización no me hubieran acompañado personas que desde su cariño y desde su interés han aportado objetividad, corrección y modos de exposición lo suficientemente claros como para que cualquier persona con ciertos conocimientos pudiera acceder al objetivo principal de la tesis.

Es por ello que doy las gracias a todas las personas del grupo de investigación de Informática Industrial e Inteligencia Artificial por todas sus aportaciones y colaboraciones que han ayudado en la corrección de esta tesis y otras publicaciones.

En particular agradezco a Rafael Molina Carmona su interés, su dedicación y su nivel de exigencia para que esta tesis pudiera estar a la altura de lo esperado. Gracias por tus aportaciones, tus sugerencias y tu buen hacer a la hora de dirigir esta tesis.

También quisiera agradecer a Antonio Javier Gallego Sánchez por su amistad y su objetividad a la hora de sugerir cambios en mi trabajo para que éste quede lo más legible y lo más coherente posible a los ojos de un especialista de lenguajes y autómatas.

Agradezco también a Gaspar Cano Esquibel por escuchar pacientemente mis disquisiciones cuando estaba inmerso en la realización de la tesis y por todas sus aportaciones que desde un punto de vista de no especialista en sistemas de realidad virtual pudo ofrecer. Gracias por tu amistad y tu paciencia.

Por último quiero agradecer a mis profesores de carrera Juan Manuel García Chamizo y Ramón Rizo Aldeguer porque en mis primeros años de carrera insuflaron en mí esta pasión por las ciencias de la computación que aún hoy en día sigue en mi vida profesional.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

Índice de contenido

I.Introducción..27
1.La realidad virtual..29
1.1.Definición de realidad virtual..29
1.2.Arquitectura de un sistema de realidad virtual..31
1.3.Problemática de la realidad virtual..33
2.Sistemas multiagentes..34
2.1.Definición de sistema multiagente..34
2.2.Problemática de los sistemas multiagentes..35
3.Lenguajes formales..36
4.Hipótesis de partida..37
5.Estructura de la tesis..38
II.Motivación y objetivos..41
1.Motivación..43
2.Objetivos..45
III.Estado del arte..49
1.Evolución de los sistemas de realidad virtual..51
2.Definición de sistema de realidad virtual..54
3.Retos de los sistemas de realidad virtual..56
3.1.Retos en el motor gráfico..56
3.1.1.Árboles de escena..58
3.1.2.Sistemas basados en lenguajes formales..59
3.1.3.Sonorización como parte del sistema de renderizado..60
3.2.Retos en los dispositivos de entrada..61
3.3.Retos en el motor físico..63
3.4.Retos en el motor de inteligencia artificial..67
3.4.1.Sistemas reactivos..68
3.4.2.Sistemas evolutivos..69
3.4.3.Sistemas estadísticos..70
3.4.4.Sistemas basados en reglas..70
3.4.5.Sistemas multiagentes..71
Sistemas multiagentes y realidad virtual..74
Simulación con sistemas multiagentes..74
Sistemas multiagentes en la industria del entretenimiento..75
Herramientas para sistemas multiagentes..76
4.Entornos de desarrollo para mundos virtuales..77
IV.Definición del modelo..81
1.Descripción de los elementos básicos del sistema..83

1.1.Elementos para la descripción del sistema (primitivas y transformaciones)........83
1.2.Elementos para la manipulación y la evolución del sistema (agentes y eventos)........86
2.Formalización del sistema........87
2.1.Gramáticas, autómatas y lenguajes formales........88
2.2.Definición de gramática independiente del contexto........89
2.3.Definición de autómata finito con pila con aceptación de pila vacía........90
2.4.Definición de traductor finito con pila........90
3.El modelo MINERVA........91
3.1.Definición de los símbolos de MINERVA........92
3.1.1.Agentes y Eventos........92
3.1.2.Primitivas y transformaciones........92
3.2.Traductores de MINERVA........93
3.3.Definición de la gramática M y su autómata finito con pila........94
3.4.Definición de la Gramática V y su autómata finito con pila........96
3.5.Traductores de Evolución (TE)........97
3.6.Traductor de Visualización (TV)........99
3.7.Traductor de Renderizado (TR)........100
3.8.Algoritmo del sistema........102
V.Patrones de diseño........105
1.Patrones de diseño........107
1.1.Definición de agentes........108
1.1.1.Según su función de evolución........110
Agentes pasivos........111
Agentes reactivos........112
Agentes deliberativos........113
1.1.2.Según el tipo de eventos a los que responden........114
Agentes de usuario........114
Agentes autónomos........115
1.1.3.Según su representación........116
Agentes visuales........117
Agentes no visuales........117
1.1.4.Según el tipo de comunicación........117
Agentes generadores de eventos........117
Agentes receptores de eventos........119
2.Definición de los eventos........120
3.Definición de primitivas y transformaciones........121
4.MINERVA como motor gráfico........123

4.1.Isomorfismo entre L(V) y el árbol de escena....123
4.2.La importancia del traductor TR....126
5.MINERVA como motor físico....127
6.MINERVA como sistema de realidad virtual....129
7.MINERVA como motor de inteligencia artificial....131
7.1.MINERVA como sistema multiagente....132
8.MINERVA como simulador....134
9.Relación entre los motores....135
VI.Experimentación....139
1.Experimentación....141
2.Metodología....142
3.Simulador de incendios forestales....142
3.1.Definición del lenguaje L(V)....143
3.1.1.Traductor OpenGL....144
Primitiva pBosque....144
....145
Primitiva pArbol....145
Primitiva pArbolArdiendo....146
Primitiva pRayo....146
Transformación tDesplaza....146
Transformación tEscala....147
Transformación tCamara....147
3.2.Definición del lenguaje L(M)....147
3.2.1.Definición de los eventos....147
3.2.2.Definición de los agentes....147
Agente aGenerador....149
Agente aBosque....150
Agente aRayo....151
Agente aArbol....152
Agente aArbolQuemado....153
Agente AKeybCam....153
Agente aCamara....154
3.3.Cadena inicial....155
4.Sistema robótico....155
4.1.Definición del lenguaje L(V)....156
4.1.1.Traductor OpenGL....157
Primitiva pRobot....157
Transformación tDesplaza....158
Transformación tGira....158
Transformación tCamara....159

Universitat d'Alacant
Universidad de Alicante

4.1.2.Traductor Real....159
Primitiva pRobot....159
Transformación tDesplaza....159
Transformación tGira....159
Transformación tCamara....159
4.2.Definición del lenguaje L(M)....160
4.2.1.Definición de eventos....160
4.2.2.Definición de los agentes....161
Agente aLaser....162
Agente aVideo....162
Agente aOperador....163
Agente aAccion....164
Agente aRobot....164
4.3.Cadena inicial....167
5.Juego de carreras de coches....168
5.1.Definición del lenguaje L(V)....170
5.1.1.Traductor OpenGL....171
Primitiva pCircuito....171
Primitiva pPaisaje....172
Primitiva pCocheId....172
Traducción de las transformaciones....173
5.1.2.Traductor de sprites....173
Primitivas pCircuito....174
Primitivas pCocheId....174
Primitivas pPaisaje....174
Traducción de las transformaciones....174
5.1.3.Traductor de sonido....175
5.1.4.Traductor de fichero....175
5.2.Definición del lenguaje L(M)....175
5.2.1.Definición de los eventos....176
5.2.2.Definición de los agentes....178
Agente aPaisaje....179
Agente aCircuito....180
Agente aKeybUser1 y aKeybUser2....180
Agente aJoystUser1 y aJoystUser2....182
Agente aFisica....183
Agente aIAPiloto....185
Agente ACoche....188
5.3.Cadena inicial....192
VII.Discusión del modelo....195

Universitat d'Alacant
Universidad de Alicante

1.Discusión del modelo....197
2.Comparación con otros sistemas....203
3.Comparación de los motores gráficos....204
4.Comparación de los motores físicos....205
5.Comparación de los motores de inteligencia artificial....208
6.Otros elementos....209
7.Tabla resumen comparativa....210
VIII.Conclusiones y futuras líneas de investigación....213
1.Conclusiones....215
2.Artículos publicados....219
3.Futuras lineas de investigación....222
3.1.Paralelización de MINERVA....222
3.2.Nuevas líneas de investigación que abre el modelo formal....225
IX.Bibliografía....229

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

Índice de figuras

Figura 1: Ejemplo de HMD....................................30
Figura 2: Sistema de realidad virtual con HMD y guante virtual....................................31
Figura 3: Arquitectura de un sistema de realidad virtual....................................32
Figura 4: Modelo de un agente....................................35
Figura 5: Árboles generados con L-System....................................43
Figura 6: Sutherland usando Sketchpad....................................51
Figura 7: Sistema de Realidad Virtual....................................52
Figura 8: Componentes típicos de un sistema de realidad virtual....................................54
Figura 9: Diversidad de dispositivos de salida....................................57
Figura 10: Árbol de escena....................................58
Figura 11: Jugando con la Wii....................................62
Figura 12: Virtual CAVE....................................62
Figura 13: Sistema de Partículas con cálculo de colisiones....................................63
Figura 14: Diferentes motores físicos....................................66
Figura 15: Regla usada por Craig Reynolds para bandadas de pájaros....................................69
Figura 16: Ejemplo de videojuego en primera persona....................................70
Figura 17: Una imagen de NetLogo....................................76
Figura 18: Esquema de MINERVA como Render....................................85
Figura 19: Gestión de Eventos en MINERVA....................................86
Figura 20: Dependencias entre traductores....................................94
Figura 21: Algoritmo de ejecución de MINERVA....................................104
Figura 22: Agente Reactivo....................................112
Figura 23: Agente Deliberativo....................................113
Figura 24: Figura 3: Ejemplo árbol de escena....................................123
Figura 25: Árbol sintáctico....................................125
Figura 26: Paso del mundo abstracto al sistema geométrico....................................127
Figura 27: Relación entre los motores....................................137
Figura 28: Simulador de incendios forestales....................................143
Figura 29: Primitiva pBosque en OpenGL....................................145
Figura 30: Primitiva pArbol en OpenGL....................................145
Figura 31: Primitiva pRayo en OpenGL....................................146
Figura 32: Simulador robótico....................................156
Figura 33: Primitiva pRobot en OpenGL....................................158
Figura 34: Actualización del atributo mem....................................166
Figura 35: Juego de carreras de coches....................................168
Figura 36: Primitiva pPaisaje en OpenGL....................................172
Figura 37: Primitiva pCocheId en OpenGL....................................173

Universitat d'Alacant
Universidad de Alicante

Figura 38: Máquina de estado del agente aIPiloto..187
Figura 39: Relación entre los motores de un sistema de realidad virtual. .201
Figura 40: Paralelización de procesamiento de cadenas................................223

Universitat d'Alacant
Universidad de Alicante

Índice de tablas

Tabla 1: Definición del lenguaje L(V) para el simulador de incendios forestales.. 144
Tabla 2: Definición de los eventos para el simulador de incendios...........147
Tabla 3: Definición de los agentes para el simulador de incendios............148
Tabla 4: Definición del lenguaje L(V) para el simulador robótico..............157
Tabla 5: Definición de los eventos para el simulador robótico...................160
Tabla 6: Definición de los agentes para el simulador robótico...................161
Tabla 7: Definición del lenguaje L(V) para el juego de carreras de coches 171
Tabla 8: Traducción de los símbolo de L(V) en cadena de caracteres.........175
Tabla 9: Definición de los eventos para el juego de carreras de coches......176
Tabla 10: Definición de los agentes para el juego de carreras de coches....179
Tabla 11: Comparativa de los motores gráficos..210
Tabla 12: Comparativa de los motores físicos..211
Tabla 13: Comparativa motores de inteligencia artificial.............................211
Tabla 14: Comparativa de otros aspectos..212

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

I. Introducción

En este capítulo hacemos una introducción a lo que es un sistema de realidad virtual, un sistema multiagente y un lenguaje formal y establecemos las problemáticas de cada uno de ellos.
Definimos cuál es la hipótesis de partida y explicamos la estructura de capítulos de la tesis.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

1. La realidad virtual

1.1. Definición de realidad virtual

Es indudable que la realidad virtual y la realidad ampliada han tenido últimamente una difusión que hasta hace unos años no tenían. Durante mucho tiempo, este tipo de sistemas ha pertenecido al ámbito de la investigación entre otros motivos por el alto coste de sus componentes. El abaratamiento de este tipo de sistemas ha ayudado a su industrialización y, por consiguiente, los ha hecho accesibles a todo tipo de usuarios.

Existen muchas definiciones de realidad virtual, pero básicamente, un sistema de realidad virtual consiste en la simulación de un escenario a través de un conjunto de dispositivos que están orientados a engañar los sentidos de sus usuarios para que tengan la sensación de que están inmersos en una realidad ficticia generada por ordenador. Para conseguir esta inmersión:

1. Se debe tener un mundo virtual ficticio simulado por un ordenador. Este mundo virtual debe contar con una colección de objetos en un espacio y un conjunto de reglas y relaciones que dirigen el comportamiento coherente de los elementos del mundo.

2. Se debe tener una sensación de inmersión. Es decir, todos los dispositivos que muestran el mundo virtual deben tener como objetivo principal el de engañar a los sentidos para que el usuario tenga la sensación de estar inmerso dentro del mundo virtual.

3. Se debe tener un conjunto de dispositivos que permitan una sencilla interacción. Con estos dispositivos el usuario tiene la sensación de poder interactuar con el mundo virtual de forma tan natural como en el mundo real.

4. Se debe tener un procesamiento en tiempo real para poder simular la sensación de inmersión.

Universitat d'Alacant
Universidad de Alicante

Como podemos comprobar todos los elementos están orientados a un objetivo principal: la inmersión. Para conseguirla se necesita de una inmersión sensorial. Es decir, debemos introducir dispositivos para sustituir los estímulos reales por estímulos ficticios en algunos o todos los sentidos. Esta inmersión sensorial busca una inmersión mental, esto es, que el usuario tenga una sensación de presencia en el mundo virtual generado por el ordenador. El problema que tiene esta sensación mental es que depende en gran medida de la capacidad del usuario de abstraerse. Para conseguir la inmersión sensorial existen dispositivos de salida típicos de realidad virtual que buscan la simulación de sensaciones.

Por ejemplo, para la inmersión visual del usuario tenemos dispositivos visuales tales como las gafas de realidad virtual o HMD (*Head-Mounted Display*) que es un dispositivo de visualización similar a un casco que permite reproducir, cercanas al ojo, imágenes generadas por ordenador (Figura 1). También tenemos los *Wall Display* que consisten en un conjunto de monitores puestos en rejilla que dibujan un escenario a tamaño natural dibujando en cada uno de los monitores una parte del mismo.

Universitat d'Alacant
Universidad de Alicante

Figura 1: Ejemplo de HMD

Para la inmersión auditiva, se pueden utilizar diferentes altavoces puestos en puntos estratégicos para generar un ambiente auditivo tridimensional

pudiendo producir efectos tales como el de Doppler.
También existen dispositivos orientados al tacto que recrean la sensación del tacto mediante vibraciones y aplicaciones de fuerzas. Un ejemplo típico de un dispositivo táctil son los guantes de realidad virtual.

Como se puede observar la diversidad de los dispositivos que tiene que gestionar un sistema de realidad virtual (como se muestra en la Figura 2, por ejemplo), la cantidad de información que debe procesar y además todo ello en tiempo real hace que este tipo de sistema tengan una alta complejidad. Sin embargo, hoy en día los nuevos procesadores dedicados para sistemas gráficos (GPU), el abaratamiento de los dispositivos de entrada y el aumento de las capacidades de las CPUs hacen que los sistemas de realidad virtual se hayan popularizado.

Figura 2: Sistema de realidad virtual con HMD y guante virtual

1.2. Arquitectura de un sistema de realidad virtual

Existen varios modelos para estructurar un sistema de realidad virtual. Una arquitectura general de este tipo de sistema se compone de los siguientes elementos:

1. Es necesario un modelo que represente al mundo de realidad virtual. En este modelo se incluirán todos los objetos que pertenezcan al mundo virtual y la forma en que interactúan

entre ellos. También es necesario las reglas que rigen el mundo virtual.

2. Es necesario un modelo que represente a los usuarios. De esta manera podemos definir usuarios con diferentes características. Entre estas características tendremos la forma en que se visualiza el usuario, qué tipo de dispositivos tiene configurado y si el usuario es real o virtual.

3. Es necesario un motor de realidad virtual. Este motor es el encargado de recopilar la información de los dispositivos de entrada, procesarla y generar en cada dispositivo de salida el estado actual del mundo virtual.

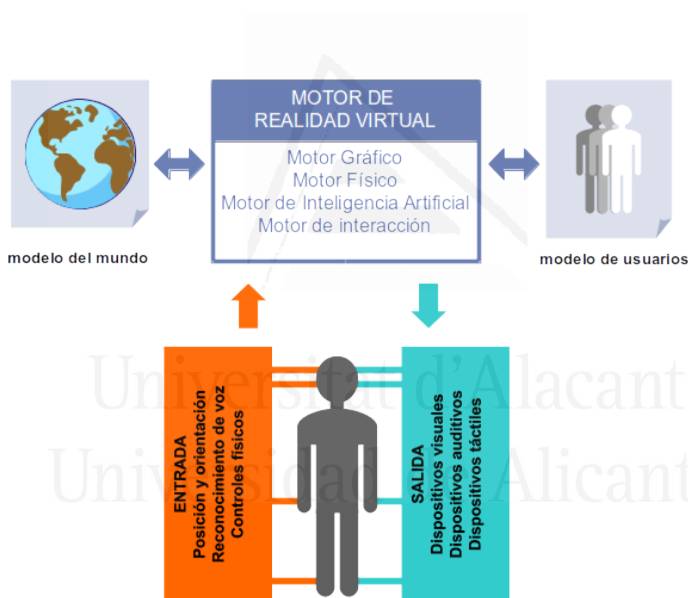


Figura 3: Arquitectura de un sistema de realidad virtual

Como vemos en la Figura 3, todos los elementos del sistema están entrelazados. Así, podemos ver que el motor de realidad virtual obtiene el estado del mundo virtual, lee las entradas de los dispositivos de interacción del usuario, actualiza aquellos objetos que son modificados y procesa la información del estado del mundo virtual para generar la salida a cada uno de los dispositivos orientados a los sentidos, es decir, a los dispositivos visuales, auditivos y táctiles.

Este motor de realidad virtual puede ser dividido en 4 partes: motor gráfico, motor físico, motor de inteligencia artificial y motor de interacción.

El primero hace referencia a la parte de los sistemas de realidad virtual que se encarga de dibujar la escena en dispositivos visuales. Estos dispositivos tienen diferentes características y capacidades de forma que, dependiendo de éstas, se generan imágenes de mayor o menor calidad.

El motor físico es el encargado de simular las leyes físicas del mundo virtual. Este componente simula los diferentes procesos físicos que acontecen dentro de la realidad sintética, dando una mayor sensación de realidad al mundo virtual. Por ejemplo, será el responsable de que los movimientos de los objetos cumplan ciertas condiciones físicas, de detectar las colisiones entre los diferentes cuerpos que componen el mundo virtual, así como de su reacción dependiendo del material del que están compuestos.

El motor de inteligencia artificial es el responsable de imitar comportamientos inteligentes por parte de los personajes que existen dentro del mundo virtual. Estos personajes tienen la capacidad de poder modificar su entorno, comunicarse con otros personajes e incluso con los usuarios del sistema de realidad virtual. Este motor también está encargado de ayudar al usuario en la toma de decisiones y en la búsqueda de soluciones óptimas para el trabajo desarrollado.

Por último, está el motor de interacción que posibilita que el usuario pueda interactuar con el entorno, modificándolo, explorándolo o cualquier otra acción que desee realizar facilitando la interacción hombre-máquina.

Universitat d'Alacant
Universidad de Alicante

1.3. Problemática de la realidad virtual

Podemos observar que el diseño de un sistema de realidad virtual conlleva un conjunto de problemas que deben ser abordados.

Uno de ellos es la diversidad de los dispositivos que se debe manejar. Esta diversidad debe ser gestionada de forma homogénea cuando se opera a nivel del mundo virtual.

Por otro lado, debe resolver el hecho de que la salida del mundo virtual debe procesarse de forma diferente dependiendo del dispositivo de salida elegido y el tipo de representación (por ejemplo, visual o sonora).

Todo este procesamiento debe hacerse en tiempo real y no debe ser apreciado por el usuario ningún retraso de procesamiento, simulando que

todas las acciones que se realizan en el mundo virtual suceden en una secuencia temporal coherente.

Todo esto supone una alta complejidad intrínseca. Esta alta complejidad junto con el alto coste del hardware asociado, ha hecho que hasta hace poco los sistemas de realidad virtual hayan formado parte generalmente del ámbito de la investigación. Con el actual abaratamiento de tarjetas gráficas que desarrollan una alta capacidad de procesamiento y con el abaratamiento de los dispositivos de entrada, los sistemas de realidad virtual empiezan a formar parte del mercado en masa dando lugar a un incremento a su vez en el desarrollo de estos sistemas.

2. Sistemas multiagentes

2.1. Definición de sistema multiagente

Un sistema multiagente es un sistema formado por un conjunto de agentes inteligentes que interactúan entre sí y se desenvuelven en un determinado entorno. Cada uno de los agentes que conforman el sistema debe tener las siguientes características:

- Autonomía: Es capaz de tener un comportamiento autónomo regido por sus propias reglas o motivaciones.
- Habilidad social: Es capaz de interactuar con otros agentes y comunicarse con ellos, pudiendo influirse mutuamente.
- Reactividad: Es capaz de reaccionar apropiadamente a cualquier estímulo producido por el entorno o por otro agente.
- Proactividad: Es capaz de actuar bajo la influencia de uno o varios objetivos planeando sus acciones.

Universitat d'Alacant
Universidad de Alicante

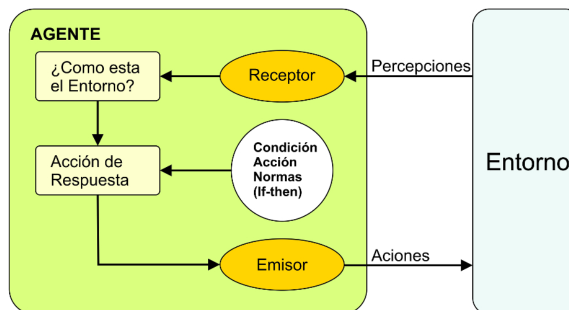


Figura 4: Modelo de un agente

En la Figura 4, vemos que además de agentes, es necesario un entorno. Este es el espacio donde se localizan los agentes y realizan sus actividades, como la comunicación. En el entorno puede existir otra clase de objetos no reactivos tales como obstáculos. Estos objetos pueden ser modelados también como agentes pero que no tienen la característica de la reactividad.

El tiempo es otro de los elementos necesarios para un sistema multiagente. Este tiempo puede ser continuo, o discreto, de forma que en cada instante o frame se ejecuta una acción por parte del agente. El modelado del tiempo también es importante para sincronizar las acciones de los agentes y para que las acciones y la comunicación entre ellos tengan un orden correcto.

Gracias a estas características los sistemas multiagentes tienen la capacidad de modelar sistemas dinámicos con comportamientos heterogéneos muy necesarios en cualquier problema que intente simular una situación real. Por ejemplo, se han utilizado sistemas multiagentes para modelar simulaciones sociales gracias a su capacidad de modelar agentes con diferentes tipos de estrategias.

Resumiendo, podemos decir que un sistema multiagente es un sistema formado por un conjunto de agentes inteligentes que interactúan entre sí y se desenvuelven en un determinado entorno.

2.2. Problemática de los sistemas multiagentes

Existen varios problemas a resolver en los sistemas multiagentes. Uno de los más importantes es el idioma que se utiliza para poder comunicarse. Ya

se ha dicho antes que la capacidad de comunicación entre agentes es una de las principales características de este tipo de sistemas. Sin embargo, para que dos agentes puedan comunicarse entre ellos deben utilizar el mismo idioma. Para asegurar que el idioma utilizado por los agentes es el mismo, existen aproximaciones para utilizar la misma ontología a la hora de generar un mensaje. Esta ontología hace referencia a una formulación exhaustiva y rigurosa de un esquema conceptual para poder facilitar la comunicación. El problema es que la definición de esta ontología es compleja y existen varias formas de definirlas y si dos agentes utilizan ontologías diferentes no pueden comunicarse. Por eso es tan importante que cuando se diseña un sistema multiagente, los agentes utilicen la misma ontología.

Otro de los problemas en los sistemas multiagentes es la reproducción de resultados. En muchos sistemas, los agentes utilizan variables aleatorias para decidir qué acciones tomar en el entorno. Esta aleatoriedad hace que la reproducción de los resultados tenga cierta dificultad.

Los sistemas multiagentes, a pesar de los retos que plantean, se están utilizando en muchos ámbitos como los de investigación, videojuegos y sistemas de realidad virtual y gracias a sus características son muy atractivos.

3. Lenguajes formales

Universitat d'Alacant
Universidad de Alicante

Un lenguaje formal es un lenguaje cuyas cadenas están generadas gracias a una gramática o sintaxis formalmente especificada. Para definir un lenguaje formal es necesario:

1. Un alfabeto definido mediante un conjunto de símbolos primitivos necesarios para formar cadenas.
2. Unas reglas gramaticales que especifican la forma en que se generan las cadenas del lenguaje formal.

A las cadenas generadas por la gramática se les denomina fórmulas bien formadas y son cadenas que forman parte del lenguaje.

Este sistema formal tiene la capacidad de tratar un conjunto de cadenas con un algoritmo bien definido y elimina las peculiaridades de cada uno los símbolos que forman el alfabeto para tratarlos de forma homogénea.

Uno de los aspectos más interesantes de los lenguajes formales es que dada una cadena podemos saber si pertenece o no al lenguaje aplicando las reglas gramaticales. Además, existen lenguajes formales que no tienen ambigüedades, es decir, que para una cadena no existen varios significados. Esto lo hace especialmente interesante para la descripción de elementos como por ejemplo el lenguaje X3D que está basado en XML y que se usa para describir mundos virtuales en la web.

4. Hipótesis de partida

A partir de las descripciones que se han hecho sobre los sistemas de realidad virtual, los sistemas multiagente y los sistemas formales, esta tesis plantea como hipótesis de partida la posibilidad de utilizar un lenguaje formal para definir, manipular y visualizar escenas virtuales formadas por elementos constructivos en forma de agentes.

Esto significa que trataremos de aprovechar la capacidad de abstracción que tienen los lenguajes formales para gestionar la diversidad de los sistemas de realidad virtual. Cada símbolo del lenguaje será un agente que podrá comunicarse con otros agentes mediante eventos, generando la actividad propia de los elementos que forman parte del mundo virtual.

El uso de símbolos para identificar objetos del mundo virtual facilitará la traducción de estos símbolos a cada uno de los dispositivos de salida. De esta manera, tendríamos una única descripción para varios dispositivos de salida.

Además, el uso de un lenguaje formal para el diseño del modelo de un mundo virtual tiene como principal ventaja el considerar los sistemas de realidad virtual desde otro punto de vista, estratificando en niveles de detalle, donde cada estrato tiene su propia función. Así, estaría la parte descriptiva representada por cadenas donde se puede ver de un vistazo los elementos que forman parte del sistema, así como la forma que tiene cada elemento de relacionarse con los demás. También, se podría ver la estructura jerárquica que existe entre los diferentes elementos del sistema.

El hecho de que los elementos del sistema tengan el mismo marco de desarrollo permitiría que la reutilización de elementos definidos pueda ser factible e inmediata.

Esta tesis, es por tanto, una definición formal de los sistemas de realidad virtual para la sistematización del desarrollo de este tipo de sistemas. Con

ello, se pretende por un lado el aumento de productividad a la hora de implementar dichos sistemas gracias a la reutilización, y tener una visión diferente gracias al modelo formal para poder cuestionar ciertas características que desde otro punto de vista no es fácil plantearse.

5. Estructura de la tesis

La estructura de la tesis queda de la siguiente manera. En el capítulo 2, nos centraremos en las motivaciones que hemos tenido para desarrollar un modelo formal para modelar sistemas de realidad virtual y enumeraremos los objetivos principales de la tesis.

En el capítulo 3, estado del arte, se va a presentar la evolución de los sistemas de realidad virtual desde sus inicios hasta la actualidad analizando los diferentes logros conseguidos hasta el momento y los desafíos que se presentan para el futuro. Para realizar esta presentación, se va a tener en cuenta la descomposición de un sistema de realidad virtual en cuatro elementos principales: motor gráfico, motor físico, motor de inteligencia artificial y sistema de interacción.

En el capítulo 4, presentamos el Modelo INtegral para Entornos de Realidad Virtual y Agentes (MINERVA), concebido como un sistema abierto compuesto por dos tipos principales de elementos. Por un lado están los elementos que describen geométricamente el mundo virtual, dando como resultado una descripción visual. Por otro lado, encontramos los elementos que permiten que el sistema evolucione y definen, en conjunto, el estado del mismo.

En el capítulo 5, se va a proponer un conjunto de patrones de diseño que podrán ser usados, utilizando el lenguaje formal descrito en los anteriores apartados. Estos patrones de diseño servirán como guías para empezar a modelar un sistema con MINERVA.

En el capítulo 6, se van a realizar los experimentos necesarios para demostrar que MINERVA cumple con los objetivos que se habían establecido. Se plantean tres experimentos: un simulador de incendios forestales, un simulador robótico y un juego de carreras.

En el capítulo 7, se analizan las ventajas e inconvenientes que tiene el modelo comparándolo con dos sistemas: OGRE y Unity. Esto sistemas tienen algunos objetivos similares a los de MINERVA, como la gestión de los motores físicos y de inteligencia artificial o la desvinculación de los

detalles de la API de renderizado.

Por último, en el capítulo 8, se presentan las conclusiones que incluyen un análisis del grado de cumplimiento de los objetivos planteados. También sugerimos posibles líneas de investigaciones futuras que aporta el hecho de que MINERVA se haya diseñando como un modelo formal.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

II. Motivación y objetivos

En este capítulo explicamos cuáles son las motivaciones que nos ha llevado a trabajar en el campo de los sistemas de realidad virtual y cuál es la filosofía que nos ha guiado en la investigación.
También describimos los principales objetivos que vamos a estudiar y porqué vemos necesario plantear un modelo formal para alcanzar dichos objetivos.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

1. Motivación

Desde mi adolescencia, siempre he tenido un especial interés por los sistemas gráficos porque conectan con la parte más artística, humanista y creativa de mi personalidad. Que haya podido desarrollar toda mi carrera profesional trabajando con estos sistemas ha sido una gran satisfacción.

Los sistemas gráficos nos permiten crear paisajes imaginarios que hasta hace poco sólo podíamos recrear en nuestra imaginación. Gracias a los gigantescos avances que se han desarrollado en las últimas décadas, podemos pasearnos por planetas tan fascinantes como Pandora de la película de Avatar, por mundos fantásticos como en El Señor de los Anillos o por universos como el de La Guerra de las Galaxias.

Pero si me apasiona el mundo de los sistemas gráficos ya no es sólo por sus espectaculares resultados, o por su belleza visual. También es porque entrelaza en una simbiosis perfecta el arte, la ciencia y la tecnología o dicho de otra manera, las humanidades y la ciencia.

Gracias a esta simbiosis han surgido nuevos movimientos artísticos al rebufo de los sistemas gráficos y en especial de los sistemas de Realidad Virtual como el Arte Virtual. Arte y ciencia empiezan a entablar un diálogo con un nuevo lenguaje que es el lenguaje utilizado por los sistemas gráficos donde geometría y arte se entremezclan de una forma que ni Miguel Angel ni Leonardo da Vinci podrían haber imaginado. Así, existen desarrollos matemáticos tan interesantes como los fractales (Figura 5) para simular la geometría de la naturaleza, como bien los llamaba Maldelbrot en [Mandelbrot 1983], o desarrollos relacionados con las curvas de Bézier.

Figura 5: Árboles generados con L-System

Pero, además, esta simbiosis entre arte y tecnología conecta con otra de mis pasiones personales que son las matemáticas y en particular la estética

matemática.

En la carrera de Ingeniería Informática pude estudiar los sistemas formales por primera vez en su más extensa realidad. Quedé maravillado ya no sólo por sus posibilidades, sino por la elegancia a la hora de expresar problemas tan complejos como los desarrollados por las Bases de Datos que conectaba directamente con la formalización de la Teoría de Conjuntos, la Teoría de Grafos que podía ser utilizada en infinidad de problemas o la Teoría de Lenguajes y Autómatas que establecía un marco bien definido sobre la expresión matemática de problemas cotidianos y que todos los días los ingenieros informáticos utilizamos. Sistemas formales que modelaban realidades bien concretas. Las matemáticas dejaban de ser un juego para convertirse en el motor fundamental del desarrollo de mi carrera profesional.

Con la experiencia en sistemas gráficos, pronto me di cuenta que encontrar un modelo formal adecuado y que expresara con exactitud el problema que pretendía resolver, hacía que la solución al problema emergiera de una forma elegante y casi sin esfuerzo y que todas las piezas que formaban parte del problema empezaran a armonizarse para encontrar la solución. Estas observaciones tuvieron dos consecuencias principales en mi personalidad como profesional. La primera fue que no podía conformarme con la primera solución que encontrara por mucho que solucionara el problema si no estaba bien establecido el modelo formal. Un modelo formal bien establecido podía adelantar posibles problemas en el futuro y solucionarlos rápidamente gracias a la formalización. Y la segunda consecuencia fue que si tenía bien establecido mi modelo formal, mi productividad aumentaba de forma espectacular.

Pero ¿qué significaba tener el modelo bien establecido? Aquí es donde la estética matemática entra en acción. Durante un verano estuve leyendo cómo Paul Dirac descubrió el positrón [Dirac 1934]. La historia cuenta que Dirac no estaba contento con la estética de las fórmulas que describían el electrón y que redefinió las ecuaciones para que éstas fueran estéticamente mejores. Al hacer este ejercicio descubrió, tiempo más tarde, una nueva partícula que fue el positrón. Esta historia influenció profundamente en mi perfil profesional cuestionándome en adelante si el modelo formal diseñado para solucionar un problema era estéticamente correcto.

He aquí donde están todos los elementos que me motivaron a realizar esta tesis. Lo que he buscado es un modelo formal donde se pudiera definir un sistema de realidad virtual al completo que fuera estéticamente

Universitat d'Alacant
Universidad de Alicante

equilibrado. El fin de este esfuerzo tenía como objetivo conseguir una mayor productividad a la hora de desarrollar un sistema de realidad virtual gracias a una fuerte formalización, y que por otro lado, pudiéramos formularnos preguntas, gracias al sistema formal, sobre características que de otra manera no podíamos formularnos. Se trata de buscar, al igual que Dirac, algún positrón.

Este es el principal objetivo de la tesis y la principal motivación que me ha llevado a desarrollar lo que en adelante describo, haciendo honor a la frase de Edsger Dijkstra: "Un informático es un matemático especializado en sistemas formales".

2. Objetivos

El sistema MINERVA (Modelo INtegral de Entornos de Realidad Virtual y Agentes) tiene como objetivo general definir y desarrollar un modelo formal con el que se pueda modelar de forma integral sistemas de realidad virtual, incluyendo todos sus elementos. Además, se desea estudiar desde un punto de vista formal la relación que existe entre todos ellos.

El modelo propuesto deberá proporcionar la integración de todos los módulos que forman parte de este tipo de sistemas, unificándolos en una misma descripción, pero sin perder la independencia funcional de cada uno de ellos. Es decir, deberá integrar los diferentes módulos constitutivos, proporcionando una total independencia de dispositivos hardware, tanto visuales como de interacción, pudiendo, si fuera necesario, sustituir un dispositivo por otro, o por una simulación, sin afectar a los mecanismos internos del sistema.

El sistema que se propone describe un modelo formal que permitirá definir y construir de una forma integral, flexible y eficiente los diferentes módulos, adaptándose a la diversidad de los sistemas de realidad virtual. De forma más concreta, nuestro modelo deberá cumplir los siguientes objetivos específicos:

- Definir el motor gráfico que maneje la diversidad de los diferentes dispositivos de salida, tanto visuales como no visuales (sonoros, salidas de ficheros, etc). Es decir, una única descripción de la escena se debe poder procesar en distintos dispositivos con un nivel de detalle acorde con las

Universitat d'Alacant
Universidad de Alicante

características del mismo. Además, se le dotará de la suficiente flexibilidad como para cambiar la representación de los elementos y dar otro tipo de información, a petición del usuario, sin necesidad de modificar la descripción misma de la escena.

- Definir el motor físico que modele toda la actividad del sistema, adaptándose a los diferentes componentes hardware donde se va a computar. Si se dispone de dispositivos hardware que implementen algoritmos físicos, el sistema los aprovechará. Si por el contrario no existen, los sustituirá mediante software. Por otro lado, en cualquier momento, será capaz de demandar al motor gráfico los datos geométricos que describen la escena, pudiendo realizar modificaciones de los elementos de la escena sin necesidad de conocer con detalle la definición gráfica de los mismos.
- Definir el motor de inteligencia artificial que facilite la integración de diversos modelos de inteligencia en distintos elementos de la escena. Además, deberá relacionarse con el motor físico y considerar en sus acciones las limitaciones impuestas por las leyes físicas establecidas por este último.
- Definir un modelo de interacción con el sistema, de forma que se abstraiga del origen de la interacción. Esta interacción no sólo resultará de diferentes dispositivos físicos, sino de procesos internos del sistema que producen interacción con la escena, tales como elementos del sistema que realizan comportamientos inteligentes.
- Reutilizar los elementos. Es decir, cualquier elemento diseñado para un determinado mundo virtual o aplicación, utilizando los mecanismos proporcionados por nuestro modelo, se podrá reutilizar en cualquier otro.

Para el cumplimiento de todos estos objetivos se utilizarán modelos matemáticos que formalizarán los diferentes componentes del sistema, abstrayendo las características que definen los elementos importantes de un modelo como el descrito.

Esta fuerte formalización ayudará a que fácilmente se puedan construir

sistemas de generación automáticos y/o visuales que ayudarán en el desarrollo de los elementos que componen un escenario virtual.

Por otro lado, el uso de modelos matemáticos ya existentes, podrá introducir en los sistemas de realidad virtual resultados de esos modelos matemáticos. Esto dará una nueva visión y nos permitirá definir nuevas características o relaciones que de otro modo no pueden establecerse.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

III. Estado del arte

En el presente capítulo, se va a presentar la evolución de los sistemas de realidad virtual desde sus inicios hasta la actualidad analizando los diferentes logros conseguidos hasta el momento y los desafíos que se presentan para el futuro. Para realizar esta presentación, se va a tener en cuenta la descomposición de un sistema de realidad virtual en cuatro elementos principales: motor gráfico, motor físico, motor de inteligencia artificial y sistema de interacción.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

1. Evolución de los sistemas de realidad virtual

La evolución de los sistemas gráficos e interactivos ha sido espectacular desde que en 1963 Ivan Shutherland presentó Sketchpad (Figura 6) en su trabajo [Sutherland 1963] sentando las bases de la programación gráfica. Se empezaba a hablar de objeto de dibujo, selección de objeto en una pantalla, puntero de selección y demás elementos que poco después se incorporarían al lenguaje técnico de dichos sistemas. Eran los inicios de un mundo que se dibujaba nuevo, donde se vislumbraban nuevas formas de interacción entre hombre y máquina.

Figura 6: Sutherland usando Sketchpad

Universitat d'Alacant
Universidad de Alicante

Pasaron décadas desde este primer paso, desarrollando sistemas cada vez más sofisticados e incorporando nuevos conceptos como el paradigma de escritorio, la invención del ratón, el sistema de ventanas y otros muchos elementos conceptuales que hoy en día están totalmente incorporados a nuestra vida diaria.

Pero con la llegada de los entornos de realidad virtual (Error: No se encuentra la fuente de referencia), se dio un salto cualitativo considerable y se abrieron nuevos desafíos. Ya no sólo era importante lo que se mostraba en una pantalla, sino que además debía parecerse a la realidad, intentando confundir los sentidos del usuario, tanto visuales como auditivos, produciendo una percepción de inmersión nunca vista en la imagen generada por ordenador. Es entonces, cuando los mundos que sólo se dibujaban en sueños y se escribían en novelas, comienzan a presentarse ante los ojos de la sociedad en una realidad sintética generada por

ordenador. Toda una revolución para una civilización donde las imágenes animadas daban sus primeros pasos con el cine a principios del siglo XX.

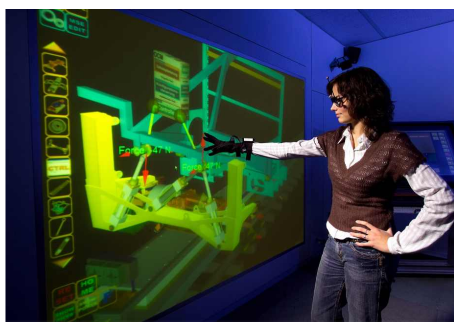


Figura 7: Sistema de Realidad Virtual

Este nuevo tipo de imagen, se convertiría en una nueva forma para transmitir noticias, ideas y proyectos, siendo así una pieza clave para entender las nuevas formas de visualizar la información e interactuar con ella. En un mundo donde la imagen se sitúa como elemento central de transmisión, los sistemas gráficos se convierten en una herramienta vital para la comunicación de ideas.

Es fácil advertir cómo los sistemas de realidad virtual han transformado numerosos ámbitos. Desde el arte hasta la ciencia, pasando por el mundo del entretenimiento, los sistemas gráficos y en ocasiones los sistemas de realidad virtual han producido un gran impacto allí donde se han aplicado. Por ejemplo, en el trabajo de J.Bates [Bates 1992] se habla de las potencialidades de los sistemas de realidad virtual en el arte y el entretenimiento o trabajos como el de M.Cavazza, J.Lugrin, S.Hartley y otros [Cavazza et al. 2005] donde se explica un sistema para realizar instalaciones de arte usando sistemas de realidad virtual.

Pero si hay una industria que se destaca como motor fundamental en la evolución de los sistemas de realidad virtual es la industria del entretenimiento y en concreto las empresas de videojuegos. Gracias a esta industria, se puede advertir un aumento casi exponencial de la capacidad de procesamiento de elementos visuales y de cálculo, así como de diferentes formas de interactuar con los mundos virtuales mediante nuevos dispositivos de entrada. Y todo ello, con un encomiable esfuerzo por abaratar costes. Hoy en día, se puede disfrutar de la Unidades de

Procesamiento Gráficas (*Graphics Processing Unit – GPU*), procesadores especializados en generar imágenes con una calidad excepcional a una velocidad significativa y un precio asequible.

Este enorme esfuerzo ha hecho que los videojuegos tengan un profundo impacto sobre otras áreas para la visualización de datos como las científicas. En el trabajo de T. Rhyne [Rhyne 2000] , por ejemplo, se hace un análisis de cómo el juego de Sim City influyó en la manera de presentar planes de distribución urbana de edificios. Además, también en este trabajo, se analiza "cómo cada tipo de videojuego facilita el desarrollo de conceptos de pensamientos visuales", conceptos que influyen en las tecnologías de visualización de la información y en la aplicación científica.

El aumento de las capacidades de las GPUs, el considerable volumen de información a tratar, la simulación de los fenómenos físicos, el comportamiento inteligente de sus componentes y todo ello en tiempo real, hace que los retos que se perfilan en los entornos de realidad virtual sean múltiples y provocan la actividad investigadora, máxime cuando está respaldada por una industria tan fuerte como la del entretenimiento.

La rápida evolución de los sistemas gráficos, ha contribuido a que la información de nuestro alrededor se presente visualmente, dando a conocer nuevas formas de análisis en aquellos campos donde se han aplicado y aumentando el volumen de información que somos capaces de analizar a la vez. Los programas CAD/CAE (*Computer-Aided Design/Computer-Aided Engineering*), los videojuegos, los sistemas de realidad virtual y los sistemas gráficos han sido aplicados en numerosas ocasiones y seguirán desarrollándose con el tiempo. Todo ello, ha contribuido a un aumento de la complejidad de los problemas a solucionar, originando nuevos retos en la visualización e interacción del usuario con la información.

2. Definición de sistema de realidad virtual

La definición de sistema de realidad virtual ha ido evolucionando con el tiempo. En el trabajo de J. Steuer [Steuer 2006] se hace un estudio detallado de la definición, así como de su evolución. Según este autor, al principio, la definición más popular de realidad virtual consistía en un sistema que incluía un ordenador con capacidad para generar imágenes animadas, un conjunto de sensores que rastreaba los movimientos del usuario, un guante de realidad virtual y unas gafas de realidad virtual donde se visualizaba el entorno generado por el ordenador (Figura 8).

Figura 8: Componentes típicos de un sistema de realidad virtual

Universitat d'Alacant
Universidad de Alicante

Sin embargo, esta definición estaba muy limitada a las tecnologías asociadas a los entornos de realidad virtual. Existen otras definiciones que huyen de esta dependencia tecnológica y se basan más en la experiencia humana, intentando generalizar el concepto de realidad virtual. Estas definiciones hablan del concepto de presencia como en la definición de J. Gibson en [Gibson 1979]:

> "La presencia se define como la sensación de estar en un entorno"

Muchos factores contribuyen a generar esta sensación y no sólo las

aportaciones generadas por los sentidos. También otros procesos mentales como la atención, nos ayudan a asimilar los datos sensoriales.

El concepto de presencia está estrechamente relacionado con el fenómeno de atribución, un fenómeno que se refiere a la percepción por parte de nuestros sentidos de la existencia de un espacio exterior más allá de la información recibida por los órganos sensoriales como nos explica J. Steuer. Así, cuando en la percepción median sistemas tecnológicos de comunicación, entonces uno percibe dos realidades: la realidad física en la que está y la realidad presentada por el medio tecnológico. A la presencia en esta última realidad, es lo que se denomina telepresencia.

Considerando este último concepto de telepresencia, una realidad virtual se define como un entorno real o simulado en el que el perceptor experimenta la telepresencia, definición finalmente aportada por J. Steuer. Es decir, confundir de alguna forma la percepción del usuario mediante componentes tecnológicos para hacerle sentir que está en otra realidad que no sea la física o lo que de otro modo es denominado inmersión en los sistemas de realidad virtual.

Para provocar esta inmersión al usuario, los sistemas de realidad virtual van a necesitar principalmente de los siguientes componentes, anteriormente considerados:

- Un Motor Gráfico que genera en tiempo real un entorno virtual que simule los efectos de luz producidos por la realidad.
- Un Motor Físico que simula las leyes físicas que gobiernan la realidad para recrear los movimientos de un mundo virtual con coherencia.
- Un Motor de inteligencia artificial que gestiona la inteligencia de personajes virtuales que viven en el mundo virtual e intercambian información con el usuario.
- Un conjunto de dispositivos tecnológicos de interacción para que el usuario pueda manipular los objetos que existen en la realidad virtual generada.

Estos componentes implican en sí mismos campos de estudio propios en las ciencias de la computación y un conjunto de retos que hace que los sistemas de realidad virtual sean uno de los más activos.

Universitat d'Alacant
Universidad de Alicante

3. Retos de los sistemas de realidad virtual

Según B. Hibbard los sistemas de realidad virtual se enfrentan a importantes retos [Hibbard 2004]. En primer lugar, cita la cantidad de datos con los que deben tratar los sistemas de visualización que en la mayoría de los casos es colosal. La necesidad de altas capacidades de computación se enfrenta directamente con el segundo gran reto: todo el procesamiento de datos debe ser en tiempo real. El autor nos habla como tercer reto de la necesidad de que la visualización tenga un gran contenido estético, no sirviendo cualquier tipo de visualización, sino que además debe ser agradable estéticamente a los usuarios. Cantidad de datos, visualización en tiempo real, belleza: características que van ser básicas en el desarrollo de los sistemas de realidad virtual y donde se centrarán sus retos.

Hasta ahora se ha hecho un recorrido sobre los desafíos que plantean los sistemas de realidad virtual en general. Sin embargo, se puede establecer una clasificación de estos desafíos según el componente del sistema de realidad virtual. A continuación, se describirán los retos que existen en el motor gráfico, en los dispositivos de entrada, en el motor físico y en el motor de inteligencia artificial.

3.1. Retos en el motor gráfico

La diversidad de dispositivos visuales va en aumento sobre todo en los últimos años. Ahora se aspira a obtener la misma información visual, ya no sólo en una pantalla de ordenador sino en nuestros teléfonos móviles, tabletas, portátiles, netbooks, pantallas pared, televisores, etc. La Figura 9 ilustra esta aspiración.

Distintos tipos de pantalla con diferentes tipos de formatos, tamaños y potencialidades, cuyos sistemas gráficos deben adaptarse a la información mostrada, de tal manera que ésta sea igual o parecida, ajustada a las capacidades de los dispositivos. Todo este esfuerzo tiene como objetivo que el usuario obtenga la sensación de que puede realizar las mismas tareas y visualizar los mismos datos en cualquier sitio y sobre cualquier dispositivo. Es decir, tener una experiencia visual homogénea.

Figura 9: Diversidad de dispositivos de salida

Esto no significa que se deban implementar distintos sistemas de visualización para cada uno de los distintos dispositivos. En primer lugar, el coste de desarrollo del sistema se multiplicaría por el número de tipos de dispositivos, un coste que no sólo debe ser medido en coste económico, sino también en coste temporal de desarrollo. Es especialmente importante este coste, a raíz de que la industria de dispositivos visuales se mueve a una gran velocidad, sacando al mercado tal cantidad de dispositivos que intentar implementar para cada uno de ellos un sistema propio de renderizado se hace casi imposible. Es necesario un sistema que sea fácilmente transportable, con un diseño multiplataforma reduciendo el coste tanto temporal, como económico, para poder ajustarse lo más rápidamente posible a las necesidades del mercado.

En segundo lugar, el hecho de tener diferentes arquitecturas para diferentes sistemas de renderizado, hace que el mantenimiento sea complicado, entre otras cosas, porque cualquier cambio de diseño se debe implementar en todos los tipos de dispositivos. Esto hace que la posibilidad de cometer errores de diferente naturaleza en cada una de las implementaciones del sistema se multiplique. Esto nos lleva a un sistema de renderizado unificado que se adapte al dispositivo.

Universitat d'Alacant
Universidad de Alicante

Existen ya algunos proyectos consolidados para un sistema de renderizado genérico independiente del dispositivo, como las API (Application Programming Interface) OpenGL [Kronos 2014] y Direct3D [Microsoft 2014]. Librerías que ya se han convertido en un estándar dentro de los sistemas gráficos.

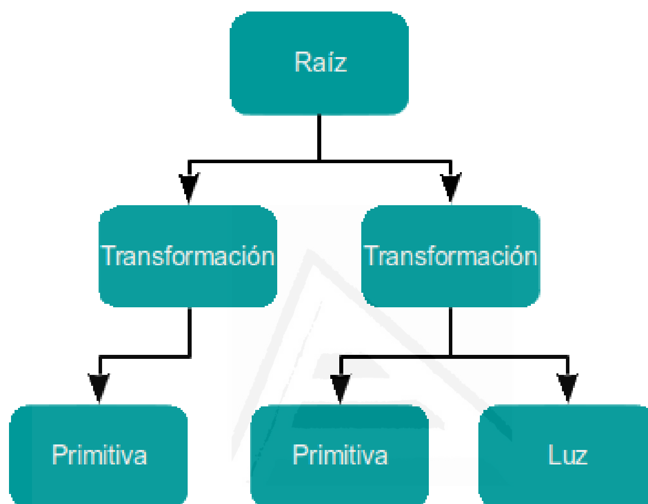


Figura 10: Árbol de escena

Universitat d'Alacant
Universidad de Alicante

3.1.1. Árboles de escena

La búsqueda de la mejor organización de los elementos de la escena para que estos se dibujen de forma eficiente, siempre ha estado presente en el desarrollo de los sistemas gráficos. Fruto de la investigación se encontraron sistemas de organización en árbol, denominado árboles de escena (Figura 10), que desde los inicios fueron una alternativa interesante y que organizan los elementos de una forma jerarquizada. En el trabajo de A. Bar-Zeev [Bar-Zeev 2003], se puede obtener un exhaustivo estudio sobre este tipo de organización. Ahí se explica la existencia de árboles de escenas de un único padre y n hijos o grafos acíclicos. Cada grafo de escena se compone de un nodo raíz, con uno o varios hijos y cada hijo a su vez podrá tener otros hijos. Los nodos podrán tener elementos dibujables o elementos que faciliten la organización de la escena como transformaciones, agrupaciones, etc.
Estas primeras aproximaciones tenían una dependencia muy cercana con la

API del dispositivo gráfico, como se puede ver en OpenScene, trabajo desarrollado por P. Strauss y R.Carey en [Strauss and Carey 1992], donde se describe un sistema de organización de escenas orientado a objetos. Existen otras soluciones que pretenden eliminar estas dependencias con las API de los dispositivos gráficos. J. Dollner y K. Hinrich en [Dollner and Hinrichs 2002] y G. Junker en [Junker 2006] buscan eliminar esta dependencia, definiendo las diferentes partes del árbol de escena y desarrollando una capa por encima para poder ocultar las dependencias con las librerías que tocan con el dispositivo gráfico (OpenGL y Direct3D) . En esta misma línea existen múltiples soluciones de las que se puede destacar OGRE, una librería que desarrolla un motor gráfico tanto para OpenGL como para Direct3D [OGRE 2014].

En torno a la visualización y los árboles de escena, se están planteando también otras aproximaciones en las que se describa, no sólo información visual de los elementos de la escena, sino también información relativa a la interacción con el usuario y diferentes tipos de organización como la espacial o la semántica, todo ello para intentar organizar los elementos de escena para que sea lo más eficaz posible y que el dibujado se pueda ejecutar en tiempo real. Una herramienta de este tipo es OpenSceneGraph, inicialmente desarrollado por Don Burns y Robert Osfield y que organiza la escena y gestiona la interacción con el usuario [Burns and Osfield 2014] y en el trabajo de A. Bar-Zeev existen otros ejemplos. En este sentido también existe el entorno de desarrollo de videojuegos Unity [Unity 2014], un sistema que utiliza un editor visual definiendo tanto los componentes que definen la escena, como la interacción con el usuario, y la forma que tienen los elementos de interactuar los unos con los otros.

Universitat d'Alacant
Universidad de Alicante

3.1.2. Sistemas basados en lenguajes formales

Además de la organización de la escena, desde los inicios de los motores gráficos se han buscado formas de describir mundos virtuales mediante lenguajes formales de alto nivel.

Uno de los primeros lenguajes descriptivos de escenas lo desarrolló la herramienta de renderizado POV-Ray [POV-Ray 2014]. Su lenguaje de descripción se basa en objetos compuestos por primitivas que forman parte del lenguaje. Existe una gran variedad de primitivas como esferas, cubos, toros, primitivas polinómicas, etc. Además podemos modelar condiciones atmosféricas, introducir varias luces, posicionar cámaras, etc. Una de las

desventajas que tiene es que no se pueden modelar superficies irregulares ya que sólo podemos componer objetos a partir de las primitivas.

Otro de los lenguajes de descripción de escenas es X3D cuyas especificaciones se describen en [X3D 2014]. Es un lenguaje descriptivo de escenas que están compuestas por objetos tanto 3D como 2D basado en XML. Es un estándar ratificado por la ISO y puede ser aplicado a una gran variedad de ámbitos como la visualización para ingeniería y ciencia, arquitectura y CAD, visualización médica, entretenimiento, multimedia, etc. Las fuentes de audio puede ser insertadas dentro de la descripción geométrica, es capaz de resolver colisiones, etc.

Por último, tenemos el Lindenmayer System o L-System desarrollado por el biólogo Aristid Lindenmayer [Lindenmayer et al. 1988]. Este sistema esta basado en gramáticas y consiste en un alfabeto de símbolos que puede construir cadenas, una colección de reglas de producción y un axioma inicial que es el inicio de la construcción y una traducción de cadenas generadas a estructuras geométricas. Está especialmente diseñado para modelar plantas y fractales.

3.1.3. Sonorización como parte del sistema de renderizado

Una parte necesaria para que la sensación de realidad sea más precisa consiste en generar sonidos en el mundo virtual.

En la parte relativa a la sonorización, los adelantos han sido interesantes. Se puede observar como partiendo de sistemas donde el sonido tenía una sola fuente de sonorización, se pasa a varias fuentes de sonido (sistema estéreo) para pasar a simular el efecto Doppler de objetos visuales virtuales. Todo ello, para generar, al igual que la renderización, sonidos cada vez más parecidos a los sonidos generados por la realidad, con el fin de aumentar la sensación de inmersión al usuario. Así, surgen los sistemas de sonorización, el proceso gemelo de la renderización. P. Cook en su trabajo [Cook 2002], describe un sistema de sonorización donde se emplean modelos de sonidos en 3D. Estos modelos pueden ser aplicados según el entorno de audición, como se describe en el trabajo de T. Looki y otros [Lokki et al. 2002]. En este trabajo, se aplica un sistema de sonorización según el lugar donde se vaya a escuchar el sonido.

3.2. Retos en los dispositivos de entrada

Toda esta evolución y diversidad de soluciones en los motores gráficos contrastaba con la evolución de los dispositivos de interacción que llevaba un ritmo más lento. Sin embargo, se puede observar, en los últimos años, un aumento en la evolución de los dispositivos de entradas, gracias, otra vez, al empuje de la industria del entretenimiento.

Durante mucho tiempo no hubo un desarrollo destacable en los dispositivos de entrada. Se podría hablar de soluciones como los guantes de realidad virtual pero sin mucha repercusión. Otras tentativas se podían ver en el trabajo de Y. Sriboonruang [Sriboonruang 2006] que detecta los gestos de las manos usados en juegos de tablero como el ajedrez utilizando una cámara. En esta línea, el trabajo de K. Oka, Y. Sato y H. Koike [Oka et al. 2002], describe un sistema que detecta los gestos de los dedos para realizar acciones. Para la detección del gesto, localiza los dedos en cada uno de los frames usando una cámara y un detector de infrarrojos. Una vez detectados los dedos, mide las trayectorias a través de las imágenes de los frames y detecta el gesto de los dedos.

Los dispositivos de entrada en 3D son otro tipo de dispositivos explorados, pero su desarrollo evolucionó de una forma lenta. Tal vez, el problema fundamental de estos dispositivos consiste en sus grados de libertad y en encontrar una forma universal de interactuar con un dispositivo de entrada 3D como bien lo analiza S. Julier, J. Hochstrate y A. Kulik en [Julier et al. 2006].

Se ha explorado también la posibilidad de interacción entre varios usuarios. Por ejemplo, K. Ryall, A. Esenther y C. Forlines describen un sistema compuesto por una mesa en la que pueden interactuar varias personas en una misma zona de trabajo [Ryall et al. 2006].

Sin embargo, todos estos proyectos no han tenido la repercusión esperada. Tal vez por su elevando coste o porque sus resultados no son del todo cómodos para el usuario final.

Pero todo cambió con la llegada de la videoconsola Wii de Nintendo, que revolucionó la industria, gracias a su nueva forma de interactuar mediante su mando, Wii Remote. Este incorpora acelerómetros y detección infrarroja permitiendo reconocer gestos del usuario y abriendo nuevas posibilidades de interacción [Nintendo 2014].

Universitat d'Alacant
Universidad de Alicante

Con la llegada de la Wii (Figura 11), otras compañías la siguieron. Así, Microsoft sacó al mercado su Kinect que es capaz de detectar los gestos de todo el cuerpo del usuario, así como su voz, objetos e imágenes [Kinect 2014]. Sony para su PlayStation 3 también desarrolló su propio sistema de interacción: PlayStation Move. Este sistema combina mando y cámara digital y es capaz de detectar la posición del mando principal [Sony 2014].

Figura 11: Jugando con la Wii

Actualmente, relacionados con los trabajos de Y. Sriboonruang y K. Oka, ha salido al mercado Leap Motion [Buckwald and Holz 2014]. Este sencillo dispositivo USB detecta los movimientos de las manos y los dedos convirtiéndose en un dispositivo similar al ratón pero sin ningún tipo de contacto. Usa dos cámaras monocromáticas IR y tres LEDs de infrarrojos y gracias a estos sensores y a un algoritmo no revelado por la compañía es capaz de detectar los gestos de las manos y los dedos.

Figura 12: Virtual CAVE

En otra línea, las pantallas multitáctiles están siendo otra forma de explorar nuevas técnicas de interacción. Por ejemplo, J. Kim y D. Gracanin desarrollan en [Kim and Gracanin 2009] un sistema que utiliza un iPod/iPhone como dispositivo de entrada para poder navegar en el entorno de realidad virtual CAVE como el de la Figura 12.

Uno de los problemas que se presenta en las pantallas multitáctiles, sobre todo en pantallas pequeñas, lo plantea el tamaño de los dedos. A veces, los componentes visuales con los que el usuario debe interactuar no son lo suficientemente grandes, debido a que están pensados para el uso del ratón.

Universitat d'Alacant
Universidad de Alicante

Es necesario, entonces, utilizar otras estrategias. Por ejemplo, H. Benko, A. Wilson y P. Baudish en [Benko et al. 2006], describen un sistema que utiliza componentes visuales adaptados a los dedos cuando se usan pantallas multitáctiles.

El desarrollo de los dispositivos de entrada para los sistemas de realidad virtual es muy importante ya que se agudiza la sensación de inmersión con un buen dispositivo de entrada.

3.3. Retos en el motor físico

Los motores físicos son los componentes encargados de simular ciertas leyes físicas como la dinámica de cuerpos rígidos, incluyendo la detección de colisiones y su respuesta (Figura 13), las fuerzas físicas newtonianas, la

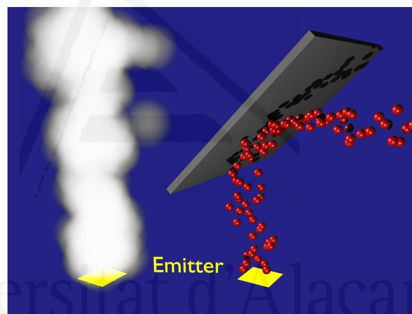


Figura 13: Sistema de Partículas con cálculo de colisiones

Universitat d'Alacant
Universidad de Alicante

dinámica de fluidos, la elasticidad, etc. Según D. Eberly [Eberly 2003], todos estos fenómenos físicos deben tenerse en cuenta cuando intentamos reproducir, dentro de los entornos virtuales, un comportamiento coherente de los objetos y lo más fiel posible a la realidad.

Uno de los principales problemas dentro de los motores físicos consiste en la detección de colisiones entre los diferentes elementos de la escena y que se considera uno de los cuellos de botella existentes en los sistemas de animación según analiza J. Lombardo, M. Cani y F. Neynet en su trabajo [Lombardo et al. 1999]. El principal escoyo en la detección de colisiones consiste en que el sistema debe hacer los cálculos para la detección y su respuesta asociada en tiempo real. El cálculo para la detección de colisiones va a depender del número de objetos que compone la escena y la forma en

que se organicen dichos objetos. Además, la respuesta a las colisiones puede aumentar considerablemente el coste computacional dependiendo del tipo de actuación que se desee realizar.

Para solucionar este problema se han estudiado bastantes alternativas y éste ha sido uno de los motores en las investigaciones en geometría computacional y siempre ha estado presente como problema activo dentro de los sistemas gráficos. Ya en 1988 M. Moore y J. Wilhelms desarrollaban un sistema de detección de colisiones basado en modelos poligonales, con respuesta a dichas colisiones [Moore et al. 1988]. Se trataba de unos algoritmos que se basaban en métodos de cálculos altamente optimizados y que se focalizaban en el método de calcular la colisión objeto a objeto.

Más tarde, se ha intentado optimizar el proceso mediante una organización jerárquica en árbol de los elementos de la escena y utilizando el cálculo de colisiones entre esferas, que es el cálculo de colisión más eficiente, se descartan elementos según se avanza en la jerarquía. Es decir, si en un nodo de la jerarquía no existe colisión, no hace falta comprobar los elementos hijos de la jerarquía. Soluciones de este tipo han sido desarrolladas por C. Fünzig, T. Ullrich y D. Feller en su trabajo [Fünfzig et al. 2006], donde se estudia la detección de colisiones mediante esferas que están preprocesadas o en [Ullrich et al. 2007] donde también se descompone en diferentes trozos los objetos de la escena y se organizan en una estructura de árbol. Otra aproximación se pueden ver en el trabajo de D. James y D. Pai pero organizadas en un árbol BDTree (*Bounded Deformation Tree*) para la detección de objetos con superficies con cierta deformación [James and Pai 2004]. Otro trabajo en la misma línea es el de M. Otaduy y otros, donde se organizan los elementos en árboles equilibrados (o *balanced trees*) [Otaduy et al. 2007]. Estos objetos se pueden cortar y se reorganizan los árboles equilibrados para que los trozos sesgados puedan incorporarse en la detección de colisiones de forma óptima. Otro tipo de organización es la organización espacial presentada por M. Teschner, B. Heidelberger y otros en su trabajo [Teschner et al. 2003] donde existe la posibilidad de detectar auto colisiones entre objetos flexibles.

Teniendo en cuenta el rendimiento de las GPUs, se están abordando soluciones a la detección de colisiones que explotan estos procesadores de alto rendimiento. Así, en el trabajo [Lombardo et al. 1999] J. Lombardo, M. Cani y F. Neyret explican un sistema de detección de colisiones usando GPUs. Para ello, lo que se hace es renderizar la escena de tal manera que

después del proceso de renderización y examinando los píxeles generados, se puede establecer qué elementos han colisionado.

Sin embargo, el motor físico no sólo se encarga de las colisiones, también debe simular las fuerzas que actúan en la escena, la velocidad y aceleración y así hasta el nivel de detalle de realidad que se desee, dependiendo de si se necesita que el motor sea en tiempo real o no.

Dentro de la simulación de los fenómenos físicos existen motores donde lo importante ya no es que sean en tiempo real, sino que describan con la mayor precisión posible la actividad simulada para conocer sus resultados. Por ejemplo, en aplicaciones CAD/CAE es importante realizar cálculos precisos para poder saber el comportamiento de los componentes una vez que se fabriquen. De esta manera, los ingenieros pueden adelantarse a posible fallos de diseño antes de la fabricación. A estos motores físicos se les denomina motores físicos de alta precisión. Existen pocas implementaciones de este tipo de motores teniendo *Working Model* como ejemplo [MSC Software 2014] .

Es evidente que cuanto más cercanía a la realidad se desea, mayor será el coste temporal de la simulación de los fenómenos físicos. Por eso, en muchos casos la estrategia hasta ahora para modelar los motores físicos consistía en calcular aproximaciones de los fenómenos físicos que acontecen. A menor detalle de los fenómenos mayor rendimiento temporal. Son motores que se utilizan, en gran medida, en videojuegos donde la fidelidad del fenómeno físico no es tan necesaria y podemos utilizar modelos igualmente efectistas pero con mucho menos detalles. Por el uso que se le da a estos motores menos precisos en los videojuegos, se les suelen denominar motores de juegos.

Universitat d'Alacant
Universidad de Alicante

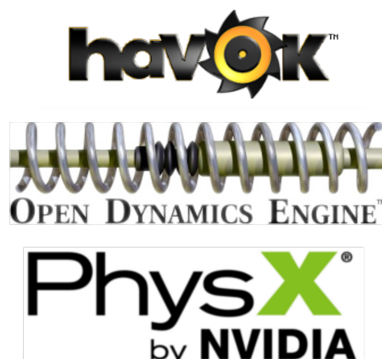


Figura 14: Diferentes motores físicos

En los motores de juegos existe una gran variedad de librerías apoyadas por la potente industria del entretenimiento (Figura 14). En primer lugar, está Physics Abstraction Layer (PAL) definido por A. Boeing en [Adrian Boeing 2014] que es una capa abstracta para el desarrollo de motores físicos en tiempo real. Soporta un determinado número de metodologías de simulación dinámica que incluye cuerpos rígidos, cuerpos flexibles, vehículos dinámicos y lo que se denomina *ragdoll* (conjunto de cuerpos rígidos que forman un esqueleto). PAL es una API en C++, flexible, multiplataforma, escalable y que soporta formatos de ficheros tales como COLLADA, Scythe Physics Editor y XML. Otra de estas librerías es Open Dynamics Engine (ODE) definido por R. Smith en [Russell Smith 2014], que implementa parte de las especificaciones de PAL, pero que soporta principalmente detección de colisiones y dinámica de cuerpo rígidos.

Todos los motores anteriores se han desarrollado dentro de la comunidad de software libre. Dentro del software propietario existen otras tantas API. Se puede destacar por un lado PhysX, cuyo propietario es NVidia [Phyxs 2014], y es utilizado en la PlayStation 3, y Havok [Havok 2014], perteneciente a la compañía Havok y que está implementado para Windows, Xbox 360, Wii, PlayStation3 y PlayStation Portable, Mac OS X y Linux.

En los últimos años, la tendencia de los motores físicos de poca precisión para videojuegos está cambiando por la importancia que está adquiriendo un motor físico cada vez más realista. La industria del entretenimiento está tratando de llevar todos los elementos del motor

físico a la GPU para mejorar considerablemente la eficiencia del sistema y el nivel de detalle de las simulaciones. Esto se está consiguiendo gracias a la capacidad de cálculo que tienen las GPU. Como ejemplos de estos pasos está la tecnología Physx de Nvidia, Havok y Bullet Physics Engine [AMD 2014] donde participa ATI, abriéndose paso la simulación de efectos físicos que hasta ahora eran intratables en tiempo real por su coste computacional, como por ejemplo la dinámica de fluidos y la dinámica de partículas o los cálculos más precisos usados en las aplicaciones CAD/CAE.

3.4. Retos en el motor de inteligencia artificial

Desde que en 1950 Alan Turing diseñara su test de inteligencia en el artículo [Turing 1950], el objetivo de la inteligencia artificial ha sido imitar la inteligencia humana. Desde los inicios, el campo de la inteligencia artificial siempre ha suscitado mucho interés en el ámbito de la investigación, tratando de encontrar diferentes modelos que puedan simular un concepto tan sutil como la inteligencia. Los sistemas expertos basados en reglas, las redes neuronales, los algoritmos evolutivos, los modelos estadísticos bayesianos y los sistemas multiagentes son algunas de las propuestas que han sido planteadas.

En un sistema de realidad virtual, donde la imitación de la realidad es uno de los objetivos principales, es evidente que una parte importante de estos sistemas deben ser los modelos utilizados para hacer que los personajes o avatares que 'viven' en el mundo virtual desplieguen un comportamiento inteligente. Es donde el test de Turing se plantea con mayor naturalidad, intentando que el usuario del sistema pueda confundir avatares artificiales con avatares guiados por usuarios del sistema.

Al conjunto de estrategias, algoritmos y modelos orientados a desarrollar comportamientos inteligentes en un sistema de realidad virtual se denominará Motor de inteligencia artificial.

En el mundo del videojuego, las aportaciones a los motores de inteligencia artificial han sido muy importantes. El desarrollo de avatares artificiales es crucial para que un videojuego sea interesante y divertido para sus usuarios. Sin embargo, el desarrollo de avatares o bots, como se denominan en el mundo del entretenimiento, es duro y costoso. J. Gemrot, R. Kadlec y otros hacen un análisis del desarrollo de estos avatares y explican la dificultad que tiene la implementación, el testeo y la

Universitat d'Alacant
Universidad de Alicante

depuración de un bot inteligente que exige muchas horas de trabajo [Gemrot et al. 2009]. En este mismo artículo, se divide el ciclo de desarrollo de un bot en 5 pasos: inventar un personaje, implementar, depurar la implementación, adaptar los parámetros del personaje y realizar un conjunto de experimentos para validar si el personaje tiene el comportamiento inteligente esperado. Pasos que en su mayor parte constituyen una carga de trabajo importante para los equipos de desarrollo de videojuegos.

Se pueden observar varias técnicas entre las que se destacan los sistemas reactivos, los sistemas evolutivos, los sistemas estadísticos, los sistemas de reglas, y los sistemas multiagentes dependiendo del nivel de inteligencia que se desea imitar.

3.4.1. Sistemas reactivos

Los sistemas reactivos son los que desarrollan una inteligencia básica. Su esquema de funcionamiento consiste en recibir un conjunto de percepciones a las que se reacciona de forma inmediata. No existe plan, ni objetivos y sólo se ejecutan acciones reactivas sin memorizar ninguna de las llevadas acabo anteriormente. Son los sistemas más simples que se usan para la simulación de comportamientos en grupos.

Existen varios ejemplos como por ejemplo el trabajo de C. Reynolds donde se simulan los movimientos de bandadas de pájaros [Reynolds 1987] usando este tipo de sistemas. Otro ejemplo al respecto es el sistema que implementaron S. Paris y S. Donikian en [Paris and Donikian 2009] para simular y analizar movimientos grupales de personas. En esta misma linea, S. Lemercier, A. Jelic, R. Kulpa y otros en el artículo [Lemercier et al. 2012] estudian los comportamientos de multitudes buscando fenómenos macroscópicos utilizando sistemas reactivos.

Universitat d'Alacant
Universidad de Alicante

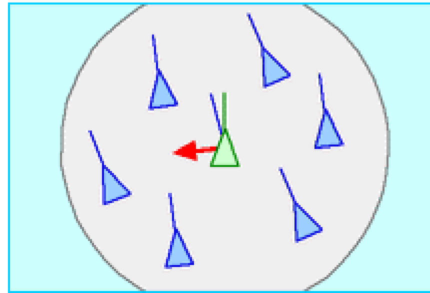


Figura 15: Regla usada por Craig Reynolds para bandadas de pájaros

Los sistemas reactivos también son usados en juegos depredador/presa donde los depredadores intentan alcanzar a las presas y las presas buscan escaparse reaccionando a los estímulos primarios del entorno, como lo describe D. Richards, M. Jacobs, y J. Porte en [Richards et al. 2012].

3.4.2. Sistemas evolutivos

En este tipo de sistemas se utilizan algoritmos evolutivos para decidir qué acciones deben ser ejecutadas con las condiciones actuales del mundo. Son sistemas adaptativos capaces de modificar su comportamiento, adaptándose a los cambios del mundo que les rodea, entre los que se encuentran los cambios de comportamiento de los otros miembros del mundo virtual. Este tipo de sistemas son usados en los juegos de depredador/presa donde tanto la presa como el depredador son capaces de modificar sus comportamientos para llegar a sus objetivos (presa: escapar; depredador: atrapar). Una aplicación de este tipo de modelo lo realizó G. Yannakakis en el artículo [Yannakakis 2004].

Otro tipo de juego donde se utilizan sistemas evolutivos son los juegos en primera persona (Figura 16). J. Murphy en [Murphy 2009] desarrolla bots que deben adaptarse a los cambios del entorno si desean sobrevivir, un tipo de juego similar al de depredador/presa, salvo que aquí todos los personajes son a la vez depredador y presa.

Figura 16: Ejemplo de videojuego en primera persona

Dentro de los sistemas evolutivos hay alguna herramienta que facilita el uso de dichos algoritmos. Podemos destacar EO Evolutionary Computation Framework [Merelo 2014] desarrollado en C++ y que implementa lo necesario para el desarrollo de algoritmos evolutivos. También existe CILib [CILib 2014], entorno de desarrollo implementado en Java.

Universitat d'Alacant
Universidad de Alicante

3.4.3. Sistemas estadísticos

Los sistemas estadísticos bayesianos son sistemas adaptativos, como los evolutivos, pero en este caso se utiliza la teoría bayesiana como elemento de adaptación, modificando las distintas probabilidades dependiendo de lo acontecido en el mundo virtual. La base de los sistemas bayesianos consiste en calcular la probabilidad de ejecutar una acción condicionada a los diferentes estados que es capaz de captar la percepción del bot. En [Gorman and Thurau 2006], por ejemplo, B. Gorman y C. Thurau modelan estos sistemas capaces de aprender de acciones pasadas y anticipar los movimientos de otros personajes a partir de su posición anterior, además de imitar comportamientos.

3.4.4. Sistemas basados en reglas

Los sistemas basados en reglas consisten en modelar un conjunto de

acciones que se deben ejecutar cuando se producen ciertas condiciones en el mundo virtual. Basados en modelos cognitivos, son reglas del tipo Si *condición* entonces *acción*. Las acciones están orientadas a lograr un objetivo dentro de un plan de acción para cada uno de los personajes y existen modelos que son capaces de organizar planificaciones. J. Funge, X. Tu y D. Terzopoulos descomponen el modelo cognitivo en dos subtareas: gestión del conocimiento y dirección del bot [Funge et al. 1999]. A partir de esta descomposición proponen un lenguaje para ayudar a desarrollar un lenguaje que son capaces de definir modelos cognitivos que denominan CML.

La gestión del conocimiento implica una administración de la percepción que el personaje tiene del mundo que le rodea y la gestión de cómo puede cambiar a lo largo del tiempo. La dirección del bot hace referencia al conjunto de acciones que debe realizar para alcanzar sus objetivos. Esto supone que de alguna forma se debe gestionar eficientemente el plan que va a ejecutar el bot buscando las acciones más prometedoras para alcanzar sus objetivos.

3.4.5. Sistemas multiagentes

Otros modelos de inteligencia artificial que han recibido especial atención, han sido los sistemas multiagentes. Como su nombre indica, son sistemas compuestos por agentes y que tienen la peculiaridad de englobar los anteriores sistemas. La definición de un agente es un tanto confusa entre otras cosas porque los sistemas multiagentes se han aplicado a un conjunto muy diverso de problemas como se nos indica en el trabajo de N. Gilbert y P. Terna [Gilbert and Terna 2000].

Existe, sin embargo, un cierto consenso sobre ciertas características generales que un agente debe tener y que se recogen en el trabajo de F. Caire, A. Poggi y G. Rimasa [Caire et al. 2003] donde se implementa una librería de sistemas multiagentes usando Java, muy extendida, denominada JADE [JADE 2014].

En primer lugar, los agentes deben ser autónomos, es decir, deben tener un cierto grado de control sobre sus acciones, deben gobernar sus tareas de control y en algunas circunstancias pueden tomar sus propias decisiones. En segundo lugar, deben ser proactivos: no sólo deben ser capaces de reaccionar a eventos externos, sino que deben exhibir un comportamiento orientado a objetivos y en algunas circunstancias pueden tomar la

Universitat d'Alacant
Universidad de Alicante

iniciativa. Y en tercer lugar, son sociales y por tanto pueden cooperar, interactuar o necesitar la ayuda de otros agentes para poder completar sus objetivos usando canales de comunicación entre los mismos.

N. Gilbert enumera ciertas características interesantes de los sistemas multiagentes, tales como modelar comportamientos heterogéneos, representar explícitamente sus objetivos mediante reglas y situarse en una posición dentro de un espacio virtual como se estudia en [Gilbert 2007]. Su carácter social y la capacidad de comunicarse con otros agentes hace que sean especialmente interesantes para estudiar los modelos de comunicación cognitiva entre individuos o flujos de información entre los miembros de un grupo. Así, pueden observar y analizar el comportamiento de los demás agentes que existen en el mundo virtual. Esto es una característica que otros sistemas de inteligencia artificial no tienen.

La capacidad de comunicación de los agentes, su disposición de cooperar entre ellos y la habilidad que tienen de interactuar con el entorno, hacen a los sistemas multiagentes muy atractivos para los sistemas de realidad virtual, sobre todo si estos se implementan para programas de entrenamiento y aprendizaje. Por ejemplo, el artículo de L. Edward, D. Lourdeaux y D. Lenne [Edward et al. 2008] tiene un ejemplo de aplicación de sistemas multiagentes para este propósito.

Como se ha hablado anteriormente, una de las capacidades más importantes de los sistemas multiagentes es que pueden percibir el entorno, reaccionar ante él y modificarlo. Esto sugiere que modelar de forma adecuada el entorno y la comunicación de éste con los agentes es uno de los procesos fundamentales en los sistemas multiagentes.

El entorno es el mundo virtual donde los agentes actúan. Este puede ser un medio neutral que afecta poco o nada a la actividad del agente o puede ser tan importante como los mismos agentes. N. Gilbert clasifica los entornos en dos tipos: los espacios de conocimiento y los entornos espacialmente explícitos.

Los primeros son entornos que no representan espacios geográficos sino que cada zona especifica unas determinadas características. Esto no quiere decir que no tengan una posición espacial, pero ésta no es relevante. Dentro de estos entornos también existen los que sitúan los agentes en un grafo cuyas aristas representan las relaciones entre los agentes.

Los entornos espacialmente explícitos son entornos que representan espacios geográficos y cada agente está situado en una posición. Un agente en una determinada posición puede relacionarse con los más cercanos a su

posición geográfica. Estos entornos son especialmente interesantes para sistemas de realidad virtual porque vienen a representar mundos virtuales con una estructura geográfica.

En los entornos con características geográficas, se evidencia la relación que existe entre el motor físico y el motor de inteligencia artificial. Hay varias soluciones para resolver la forma en que se relacionan los agentes con el motor físico que define el entorno donde están los agentes. Por ejemplo, P. Maes y T. Darrell en [Maes and Darrell 1995] implementan el sistema ALIVE, un sistema de realidad virtual que relaciona agentes autónomos con usuarios, teniendo como uno de sus requerimientos, relacionar el motor físico del mundo virtual con las acciones de los agentes.

La parte social y la comunicación entre los agentes es otra de las características más importantes de los sistemas multiagentes. La comunicación entre agentes debe tener 3 principales características según F. Caire, A. Poggi y G. Rimasa que explican en [Caire et al. 2003]:

1. Selección de los mensajes recibidos. Es decir un agente puede ignorar un mensaje de otro agente o incluso puede negarse a cooperar. Esto significa que los mensajes son asíncronos y no se puede bloquear el agente que envía el mensaje esperando infinitamente una respuesta del agente receptor.

2. La comunicación es una acción más de los agentes. Es decir, el acto de comunicar está al mismo nivel de prioridad que las acciones del agente. Dentro de la capacidad de planear acciones por parte de los agentes, se debe tener en cuenta en la planificación la acción de comunicarse.

3. Los mensajes tienen significado semántico. El agente debe entender el significado de la acción y por qué debe ser realizada (la intención del agente emisor). Esta propiedad hace necesaria una semántica estándar y universal.

Esta última propiedad de la comunicación exige que los mensajes sean entendibles por todos los agentes mediante algún tipo de estándar. Para ello, se crea la FIPA (*Foundation for Intelligent Physical Agents*) que pretende la estandarización de las tecnologías de los sistemas multiagentes.

Universitat d'Alacant
Universidad de Alicante

Sistemas multiagentes y realidad virtual

Gracias al desarrollo de los sistemas de realidad virtual y las características de los sistemas multiagentes existen ya trabajos que integran agentes en mundo virtuales. Las ventajas de esta fusión son explicadas por T. Trescak, M. Esteva y I. Rodriguez en [Trescak et al. 2010] y consisten en:

1. Los sistemas multiagentes regulan los sistemas de realidad virtual definiendo una norma dentro del mundo virtual.
2. La representación en 3D de los agentes facilita un mejor entendimiento del comportamiento global del sistema.
3. Los participantes del mundo virtual pueden ser tanto usuarios como agentes de software facilitando una participación directa de los usuarios en los sistemas de realidad virtual.

Ya se ha mencionado el amplio espectro de aplicaciones que están teniendo los sistemas multiagentes, pero centrando el estudio en los sistemas de realidad virtual existen dos campos donde los resultados han sido fructíferos. Por un lado, la aplicación de los sistemas multiagentes a la simulación y en los últimos años en los videojuegos.

Simulación con sistemas multiagentes

Los sistemas multiagentes por su capacidad de comunicación, de implementar modelos heterogéneos y de cambiar el entorno que los rodea son muy interesantes para simular todo tipo de sistemas.

R. Axelrod hace un estudio exhaustivo sobre el impacto que tienen los sistemas multiagentes en las simulaciones [Axelrod 2003]. En este trabajo se explica, que dichas simulaciones son importantes para el estudio de entornos donde las variables de entrada y sus relaciones forman un entramado complejo y cuyo estudio se hace dificultoso. Se pueden realizar simulaciones para un sistema complejo con diferentes hipótesis de trabajo para ver como reacciona el sistema ante estas hipótesis. Por ejemplo, en economía, el uso de los sistemas multiagentes son ideales para simular varios escenarios gracias a que se pueden implementar agentes con diferentes comportamientos, tanto racionales como irracionales y realizar previsiones de futuro de cómo se comportará el mercado según diferentes escenarios.

Universitat d'Alacant
Universidad de Alicante

Sin embargo, dentro de las simulaciones existen varios problemas a resolver según R. Axelrod. Uno de ellos es la reproducción de los resultados. A causa de que ciertas variables de los agentes son aleatorias (por ejemplo, para modelar comportamientos no racionales o en sistemas estadísticos bayesianos) la reproducción y la validación de los resultados por parte de otros investigadores es una tarea delicada y afanosa. Además, la implementación del sistema puede llevar asociados algunos errores no detectados que afecten a los resultados de la simulación, mostrando diferencias entre modelos implementados por diferentes equipos de desarrollo.

Sistemas multiagentes en la industria del entretenimiento

En los últimos años, el mundo del entretenimiento está interesado en los sistemas multiagentes para el desarrollo de sus personajes. Cierto es que estos sistemas se adaptan muy bien a las necesidades de un personaje de videojuego. Pueden implementar diferentes niveles de inteligencia acordes con el personaje del juego y existe un entorno que pueden modificar y observar. Además, hoy en día, sobre todo con los videojuegos en red, la comunicación entre avatares es necesaria. Dentro del entretenimiento, también se están utilizando en la realización de películas, teatro, juguetes, etc. En [Maes 1995], P. Maes analiza cómo los sistemas multiagentes se están usando en el mundo del entretenimiento.

Por otro lado, los sistemas multiagentes dentro de los videojuegos están llevando a los investigadores en inteligencia artificial a interesarse por realizar sus experimentos en videojuegos. Este uso es especialmente interesante porque en un mismo escenario, el videojuego, se pueden ejecutar diversos algoritmos de inteligencia artificial. Existen ejemplos con videojuegos tan populares como Quake II donde J. Laird desarrolla en [Laird 2001] algoritmos de inteligencia artificial para bots de dicho juego. En [Khoo and Zubek 2002], A. Khoo y R. Zubek hacen lo mismo pero con el videojuego HalfLife utilizando un enfoque cognitivo con capacidad de aprendizaje y predictivos. Por último, en el trabajo [Miles and Quiroz 2007] C. Miles y J. Quiroz hacen uso de algoritmos genéticos para la implementación de estrategias basados en sistemas de razonamiento utilizando árboles de búsqueda, aplicados a videojuegos como Starcraft, Dawn of War o Age of Empire.

Herramientas para sistemas multiagentes

El uso de librerías comunes que implementen las características más importantes de los sistemas multiagentes hace que la reproducción de resultados sea mucho más sencilla. Efectivamente, al tener implementada toda la infraestructura necesaria para el desarrollo del sistema, la probabilidad de errores en la implementación es menor y más fácil de detectar ya que los errores pueden estar sólo en el módulo particular del problema que se está modelando y no en la estructura interna de la implementación de los sistemas multiagentes. De entre estas librerías de herramientas para desarrollar sistemas multiagentes destacan JADE, NetLogo y MASON.

Como ya se a comentado anteriormente, JADE es una librería implementada en Java que permite modelar sistemas multiagentes [JADE 2014]. Puede ser ejecutada en varias máquinas conectadas en red. JADE tiene como mayor aporte un sistema de comunicación entre agentes bastante desarrollado, que implementa la comunicación semántica entre agentes utilizando FIPA. Una de las desventajas que tiene es que si se desea mostrar en un sistema gráfico los agentes entonces se debe implementar aparte. Es una de las librerías más usadas para modelar sistemas multiagentes.

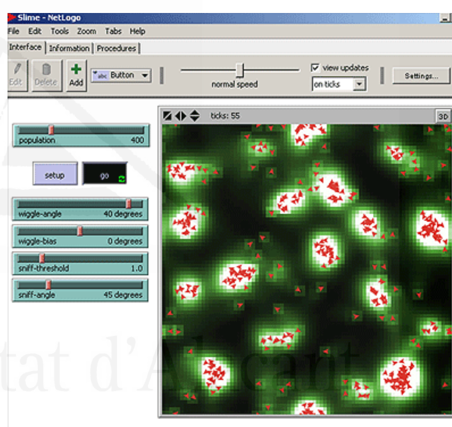


Figura 17: Una imagen de NetLogo

Universitat d'Alacant
Universidad de Alicante

NetLogo (Figura 17), presentado por S.Tisue y U. Wilensky en [Tisue and Wilensky 2004], sin embargo, sí que tiene una parte visual de los agentes pero sólo en 2D. La visualización de los agentes es bastante simple, siendo representados por una simple flecha. No se puede implementar una visualización personalizada de los agentes. Además, el sistema de comunicación no puede realizarse en red y sólo se comunican los agentes dentro de la misma máquina. Sin embargo, es un sistema muy liviano.

S. Luke y C. Cioffi-Revilla presentan en [Luke and Cioffi-Revilla 2004] MASON, una de las librerías con mayor soporte gráfico.

Implementada en JAVA, los agentes se pueden disponer tanto en 2D como en 3D y puede ser ejecutada con o sin visualización. Como principal característica es capaz de gestionar miles de agentes de forma eficiente y garantiza que los resultados son independientes de la plataforma en la que se ejecuta.

4. Entornos de desarrollo para mundos virtuales

Existen múltiples entornos de desarrollo para diseñar e implementar mundo virtuales que están orientados, sobre todo, al desarrollo de videojuegos. Estos entornos buscan, entre muchas características, ser transparentes a las API de tarjetas gráficas como Direct3D y OpenGL, tener herramientas que faciliten requerimientos de un motor físico y ayudar al desarrollo del motor de inteligencia artificial.

Como principales entornos de desarrollo analizamos Unreal Engine Kit, IrrLicht Engine, OpenSceneGraph, OGRE y Unity.

Unreal Engine Kit [Epic Games 2014] es una suite profesional para construir videojuegos que pueda ser ejecutado en un conjunto de sistemas operativos de forma transparente (Windows, Apple, Linux, Plataformas móviles, etc). Esta orientado a mundos virtuales en 3D y se basa en un sistema de elementos denominados actores. Estos actores pueden realizar acciones sobre el mundo virtual y otras operaciones asociadas a los actores como el dibujado o el cálculo de colisiones entre otros. Las características principales de este kit de desarrollo son:

- Controladores para implementar el motor de inteligencia artificial. Con estos controlares podemos analizar los elementos que hay cerca del actor o planificar nuestros movimientos según un plan establecido.
- Sistema de audio para definir sonidos en la escena.
- Herramientas de animación para sistemas de esqueletos asociados a mallas de triángulos. Con estas herramientas también podemos detectar colisiones en la escena.
- Herramientas para crear escenarios virtuales.
- Editor para el desarrollo de sistemas virtuales.

Unreal Engine Kit es por tanto un sistema para el desarrollo de videojuegos y simulaciones con muchas facilidades sobre todo si se tiene que desarrollar mundos en 3D.

IrrLicht Engine [Gebhardt 2014] es una librería implementada en C++ que nos abstrae la capa de renderizado al igual que Unreal Engine. Más modesta que el anterior es capaz de renderizar para Direct3d y OpenGL y está implementado para Windows, Linux y Apple. Entre sus características principales tenemos:

- Sistemas de animación basados en esqueletos.
- Renderizado tanto en 2D como en 3D.
- Importación de tipos de archivos de mallas de triángulos.
- Algoritmo de detección de colisiones.

Esta librería está desarrollada sobretodo para abstraer el renderizado. Esto hace que no exista ninguna referencia a los sistemas de inteligencia artificial y en cuanto al motor físico sólo tiene la detección de colisiones.

OpenSceneGraph [Burns and Osfield 2014] es otra librería implementada para Windows, Linux, Apple, iOS y Android. Es una librería basada en un árbol de escena (anteriormente explicado) y que sobretodo tiene como característica principal el eliminar las dependencias del sistema operativo. Su implementación esta basada en OpenGL y no tiene nada para el cálculo de colisiones, ni ningún sistema que ayude en la implementación del motor de inteligencia artificial. Es una librería que sobre todo se focaliza en la renderización.

OGRE [OGRE 2014] es otra herramienta para el desarrollo de videojuegos. Está implementada en C++ y principalmente tiene como principal objetivo la abstracción en el renderizado. Así podemos ver que OGRE puede dibujar escenarios en 3D utilizando tanto la librería OpenGL como Direct3D. Además tiene herramientas para la animación de mallas de triángulos, un árbol de escena similar al de OpenSceneGraph, etc. Sin embargo, no tiene implementado nada del motor físico. Sí tiene, por otra parte, varias librerías a parte de OGRE que si implementan algoritmos de detección de colisiones y cálculo de sistemas cinemáticos. También tiene librerías que soportan modelos de sistemas de inteligencia artificial.

Por último tenemos Unity [Technologies 2014]. Esta herramienta de

desarrollo es un editor gráfico donde se puede desarrollar todo tipo de mundos virtuales. Implementado para múltiples plataformas tanto de escritorio como móviles es un sistema completo donde podemos definir elementos tanto del renderizado, como del motor físico o del motor de inteligencia artificial. Es uno de los más completos y se pueden desarrollar mundos en 2D y en 3D. Su programación es principalmente visual aunque en algunas partes podemos utilizar su lenguaje Script para desarrollar, sobretodo, partes del motor físico y del motor de inteligencia artificial.

Todos los entornos de desarrollo explicados anteriormente tienen objetivos similares a los de MINERVA y sería conveniente que una vez definido el modelo formal pudiéramos comparar MINERVA con algunos de estos entornos de desarrollo. Por ser los más utilizados y porque forman parte de dos tipos de herramientas, las herramientas que son sólo librerías y las herramientas que tienen editor de desarrollo, se va a seleccionar OGRE y Unity como base para comparar MINERVA con los entornos de desarrollo ya existentes. De esta manera, podremos comparar las ventajas y desventajas de tanto de MINERVA como de Unity y OGRE.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

IV. Definición del modelo

En este capítulo presentamos el Modelo INtegral para Entornos de Realidad Virtual y Agentes (MINERVA), concebido como un sistema abierto compuesto por dos tipos principales de elementos. Por un lado están los elementos que describen geométricamente el sistema, dando como resultado una descripción visual del estado actual. Por otro lado, encontramos los elementos que permiten que el sistema evolucione y definen, en conjunto, el estado del mismo.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

1. Descripción de los elementos básicos del sistema

Existen varios tipos de sistemas según se relacionen con el exterior o evolucionen de forma independiente. Según la Teoría de Sistemas planteada por L. Bertalanffy se denomina sistemas abiertos a los primeros, mientras que a los segundos se les llama sistemas cerrados [Bertalanffy 1986]. Los sistemas de realidad virtual son sistemas abiertos que interactúan con el usuario para darle una sensación de realidad sobre una visualización de un mundo sintético procesado gráficamente, según el criterio de J. Steuer establecido en [Steuer 2006]. Si el sistema es cerrado y sólo muestra un mundo sintético donde no se actúa con el usuario, entonces el sistema deja de ser un sistema de realidad virtual. Dentro de estos sistemas cerrados, se pueden implementar simulaciones virtuales donde se analizan la evolución de sistemas, demostración de hipótesis, etc.

MINERVA es un sistema abierto que se compone principalmente de dos tipos de elementos. El primer tipo son las primitivas y las transformaciones que describen el estado actual del sistema. El segundo tipo son los agentes que tienen la capacidad de evolucionar en el tiempo y establecen el estado global del sistema. Son capaces de modificar variables del sistema, introducir o eliminar otros agentes o establecer distintos criterios de decisión dependiendo de los agentes que tienen a su alrededor.

1.1. Elementos para la descripción del sistema (primitivas y transformaciones)

Para describir el aspecto gráfico de un mundo virtual se necesitan principalmente dos tipos de elementos: primitivas y transformaciones. Tanto unas como las otras pueden ser combinadas creando una gran variedad de escenas.

Las primitivas describen la representación de un objeto dentro del mundo virtual. En los sistemas gráficos, siempre se han considerado primitivas a esferas, cubos, mallas de puntos, y todo cuerpo geométrico definido tanto en 2D como en 3D. Estas primitivas siempre se han asociado a elementos visuales de una escena que son dibujados utilizando un sistema de renderizado. Sin embargo, además de los dispositivos

visuales, se pretende definir primitivas cuya forma de representación no tenga porqué ser visual, tales como el sonido. Así, el sonido de un tren, una explosión o un disparo, debería ser representado como una primitiva dentro de la escena.

En lo sucesivo hablaremos principalmente de sistemas gráficos, por ser los más habituales, pero los conceptos a los que se hace referencia son extensibles a cualquier sistema de representación, tanto visual como de otro tipo.

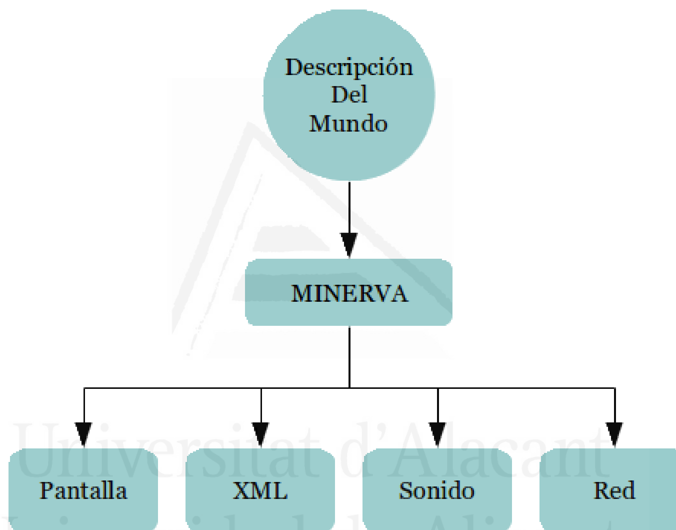
El procesamiento de una primitiva depende de las capacidades gráficas y de la resolución del dispositivo. Hasta ahora, las primitivas se definían según las características del sistema de renderizado. Esto es, si el sistema podía dibujar escenas 3D, las primitivas se implementaban considerando las tres dimensiones. Sin embargo, si el sistema de dibujo sólo podía dibujar elementos 2D entonces las primitivas dibujaban elementos bidimensionales.

Ahora con este modelo, se pretende que el sistema gráfico sea capaz de evaluar cómo pueden visualizarse las primitivas en el dispositivo de salida dependiendo de sus capacidades gráficas. Puede adaptarlas a las capacidades del dispositivo para ver la escena en las mejores condiciones que puede proporcionar el dispositivo gráfico. Por ejemplo, si existe un dispositivo con capacidades de renderización en 3D, entonces la primitiva seleccionada sería una malla de triángulos o, si por el contrario, sólo puede dibujarse en 2D, entonces optaría por una primitiva de tipo imagen. Planteado de esta manera, el sistema gráfico es quién decide el tipo de primitiva entre varias opciones de representación, y así se adapta en tiempo de ejecución al dispositivo gráfico.

Además de las primitivas, para modelar un mundo virtual, son imprescindibles las transformaciones que modifican el comportamiento de una o varias primitivas. Éstas tienen un ámbito de aplicación. Pueden ser combinadas para realizar composiciones y aumentar así las formas de modificación. Como ejemplos de transformaciones se tienen como habituales el posicionamiento espacial, el escalado, los giros, etc. Sin embargo, al igual que en las primitivas, no sólo queremos aplicar las transformaciones geométricas sobre cuerpos geométricos, sino que se desea extender la definición de transformación a propiedades del dibujado como el color, la textura, gradientes, etc, o a transformaciones no visuales como el volumen del sonido.

Las primitivas y las transformaciones no sólo tienen por qué ser

dibujadas en dispositivos gráficos. Puede ser necesario tener un tratamiento más abstracto para poder procesarla de otras formas que no sean gráficas. Por ejemplo, pueden tratarse las escenas para que su descripción sea codificada y enviada por red, para luego mostrarla en otro ordenador o escribir la escena en algún tipo de formato de dibujo como DWG, DXF o X3D. Teniendo en cuenta esta diversidad de formas para el tratamiento de una descripción de un mundo virtual, se va a establecer una forma más abstracta para la definición de una primitiva como representación de un objeto. La Figura 18 muestra el esquema que sigue MINERVA para realizar



Universitat d'Alacant
Universidad de Alicante

Figura 18: Esquema de MINERVA como Render

un render de la escena.

Con estos elementos sólo se puede modelar una representación descriptiva del mundo en la que la interacción del usuario no es posible. Muchos de los lenguajes descriptivos de mundos virtuales solo definen este tipo de elementos. Sin embargo, también se quiere modelar las diferentes formas que tiene el usuario de manipular un mundo virtual, incluso qué elementos del propio sistema pueden modificar otros elementos de la escena.

1.2. Elementos para la manipulación y la evolución del sistema (agentes y eventos)

Para que un usuario pueda manipular el mundo virtual necesita de dispositivos que comuniquen al sistema, de alguna forma, el modo en que el usuario quiere alterar el mismo. Para ello, se recurrirá a un modelo de eventos que son capturados por el sistema para hacer las transformaciones necesarias, ejecutando la orden que el usuario ha enviado mediante el evento. Este modelo de eventos necesita de un emisor y un receptor.

En otros sistemas, los eventos están orientados a los dispositivos que los generan. Por ejemplo, si en un joystick se pulsa un botón, se genera un tipo de evento que nos dice el botón que ha sido pulsado y se manda para ser procesado. Sin embargo, lo que se propone en MINERVA es que los eventos no estén tan vinculados a los dispositivos, sino que definen un tipo de manipulación que establece la forma de cambiar los objetos de la escena. Así, si el botón de un joystick tiene como función disparar, en vez de enviar el evento del botón y que el sistema decida qué hacer, lo que se enviará es un evento que especifique el concepto disparo.

Figura 19: Gestión de Eventos en MINERVA

La ventaja de definir así el evento es que disparar ya no sólo está asociado a

un joystick, sino que puede estar asociado a un teclado, a un guante o a cualquier otro dispositivo que signifique disparar, sin tener que cambiar la recepción del evento, tal y como se muestra en la Figura 19.

Estos eventos van a ser enviados por emisores que transformarán la información del dispositivo en información más genérica. Estos emisores van a comunicarse con el dispositivo hardware y van a traducir los datos generados por el dispositivo en un evento del tipo que los objetos de la escena pueden manipular. Así, por ejemplo, se podrá definir un emisor que captura los datos generados por el joystick o generados por un teclado y los transformará en un evento de disparo. Esto lo que pretende es trasladar la gestión del evento del dispositivo hardware al emisor de eventos y reducir la complejidad asociada a los distintos dispositivos hardware de donde proceden los eventos.

Los eventos por sí solos no sirven para manipular una escena. Necesitan de receptores que reciban dicho evento y hagan las variaciones oportunas en los elementos involucrados para que esa alteración sea efectiva.

Un agente es el elemento que es capaz de capturar un evento y modificar su comportamiento según su definición. Así, si un agente recibe el evento de disparo, éste podrá explotar si es una nave espacial, morir si es un personaje o no hacer nada si es un muro. Los agentes tienen un estado definido mediante atributos. Estos atributos también pueden ser actualizados según el evento recibido.

Además de recibir eventos, el agente es capaz de enviar eventos a otros agentes para que estos puedan capturar el evento. Son emisores de eventos a la vez que receptores.

Universitat d'Alacant
Universidad de Alicante

2. Formalización del sistema

Una vez presentados los elementos más importantes del sistema, se debe utilizar algún tipo de modelo para poder diseñar su forma descriptiva así como la relación que existe entre ellos. Para describir el sistema se utilizará un modelo matemático. La formalización de un modelo matemático consiste en describir de forma sistemática todo lo necesario, eliminando cualquier tipo de ambigüedad. Para ello, utilizaremos un lenguaje formal libre de ambigüedades que especifica los diferentes aspectos que se pretende modelar. De esta manera, reducimos el sistema a meros símbolos que pueden ser manipulados de forma automática mediante reglas de

inferencia. Los sistemas formales tienen la capacidad de abstraer los detalles necesarios de los elementos, para captar la esencia de cada una de sus unidades y mediante reglas de inferencia, sacar las características globales más importantes.

Para modelar nuestro sistema formal es necesario establecer los símbolos que lo definen, un lenguaje libre de ambigüedades estructurado mediante una gramática y una semántica que le dé significado al lenguaje.

2.1. Gramáticas, autómatas y lenguajes formales

La teoría de lenguajes y autómatas es la teoría que define formalmente un lenguaje, de tal manera que se puede conocer si una cadena forma parte del lenguaje o no. Un lenguaje formal esta definido por un conjunto de símbolos, alfabeto, que permite construir cadenas que pertenecen al lenguaje. Existen varias formas de definir un lenguaje. Entre ellas se encuentran los autómatas, las expresiones regulares y las gramáticas. En particular, nos interesan las gramáticas independientes del contexto, notación natural y recursiva que establece el conjunto de cadenas que definen un lenguaje [E.Hopcroft et al. 2002].

En MINERVA, se van a generar cadenas a partir de primitivas, transformaciones, agentes y eventos. Dependiendo de la situación, se va a modelar una gramática que nos indicará la forma de generar cadenas dentro del lenguaje de manera que representen sistemas de realidad virtual bien formados.

Se definirán dos lenguajes. El primer lenguaje, denotado por L(M), es el lenguaje que define cadenas de agentes bien formadas. Este lenguaje se usará para generar cadenas que representan la evolución del sistema a lo largo del tiempo. El segundo lenguaje, denotado por L(V), es el lenguaje cuyas cadenas están formadas por primitivas y transformaciones y que representan la visualización del sistema en un estado determinado. Para transformar las cadenas de L(M) en cadenas de L(V) se necesitará de un traductor.

El uso de la teoría de lenguajes y autómatas hace que el modelo pueda aprovechar todas las propiedades de los lenguajes formales. Esto ayudará a establecer un sistema de realidad virtual bien definido.

A continuación, se van a realizar las descripciones generales sobre los elementos de la teoría de lenguajes y autómatas imprescindibles para la definición del modelo. Para ello, se describe la parte formal de una

gramática, se muestra la definición de un autómata con pila y finalmente se establece lo que es un traductor con pila, los tres elementos que son empleados para diseñar el modelo MINERVA.

2.2. Definición de gramática independiente del contexto

A. Aho, R. Sethi y J. Ullman definen una gramática independiente del contexto G en [Aho et al. 1990] . Para ello, definen la sintaxis del lenguaje mediante la tupla $G=<\Sigma,N,R,S>$ donde:

- Σ: Es un conjunto finito de símbolos terminales o alfabeto del lenguaje, que juntos forman cadenas. De ahora en adelante, las letras minúsculas representarán símbolos del alfabeto.
- *N:* Es un conjunto finito de símbolos no terminales o variables que representan subcadenas del lenguaje. Para representar variables de una gramática se utilizan letras mayúsculas.
- *S:* $S \in N$ es la cadena inicial.
- *R:* Es el conjunto de reglas sintácticas o de producción que describen cómo un símbolo no terminal se define en función de símbolos terminales y no terminales. Una regla de producción es una aplicación $r: N \rightarrow W^*$ donde $W=\Sigma \cup N$ y $r \in R$.

Se dice que una gramática es recursiva por la izquierda si existe alguna regla tal que $A \rightarrow A\alpha$, siendo α una cadena formada tanto por variables no terminales como por símbolos terminales de la gramática.

Por otro lado, se demuestra que las gramáticas independientes del contexto y no recursivas por la izquierda aseguran la existencia de un procedimiento que verifica, dada una cadena, si esta pertenece o no a la gramática. Este procedimiento es un autómata finito con pila con aceptación de pila vacía AP. Esto significa que una cadena dada w es aceptada por AP si el autómata consume los símbolos de entrada hasta vaciar la pila. Si la cadena termina con la pila vacía entonces dicha cadena está aceptada por AP y es generada por la gramática.

2.3. Definición de autómata finito con pila con aceptación de pila vacía

La definición formal de un autómata con pila con aceptación de pila vacía se define mediante la tupla $AP=<\Sigma,Q,\Gamma,f_t,S,Z_0>$ donde:

- Σ: Es un conjunto finito de símbolos
- Q: Es un conjunto de estados del autómata de entrada.
- Γ: Es un conjunto finito de símbolos o alfabeto de la pila.
- f_t: Es la función de transición definida como:

  $f_t: Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow P(Q \times \Gamma^*)$ donde P define el conjunto potencia y ε la cadena vacía.
- $S \in Q$: Es el estado inicial.
- $Z_0 \in \Gamma$: Es el símbolo inicial de la pila.

En general, una gramática $G=<\Sigma,N,R,S>$ independiente del contexto y no recursiva por la izquierda, es equivalente a un autómata finito con pila. Este autómata está definido por $AP=<\{q\},\Sigma,N \cup \Sigma,f_t,q,S>$ donde la función de transición f_t se define de la siguiente forma:

1. Para cada variable no terminal A

   $f_t(q,\varepsilon,A) = \{(q, \beta) \mid A \rightarrow \beta$ *es una regla de producción de R* $\}$
2. Para cada símbolo terminal a

   $f_t(q,a,a) = \{(q, \varepsilon)\}$

Si las cadenas aceptadas por el autómata con pila AP se denotan por N(AP) y el lenguaje generado por la gramática G se denota por L(G) entonces se demuestra en que $N(AP) = L(G)$ [E.Hopcroft et al. 2002].

2.4. Definición de traductor finito con pila

Un autómata, además de identificar las cadenas que pertenecen a un determinado lenguaje, puede generar como salida otra cadena. Si un

Universitat d'Alacant
Universidad de Alicante

autómata finito con pila tiene como salida otra cadena, entonces es denominado traductor finito con pila. Estos traductores van a ser los componentes algorítmicos que se van a utilizar para poder procesar tanto el renderizado como la evolución del sistema en MINERVA.

La definición formal de un traductor es la de un autómata finito, que además de tener la cadena de entrada genera una cadena de salida. La cadena de salida puede estar formada por símbolos del alfabeto de entrada o puede ser una cadena de otro alfabeto.

La diferencia entre un autómata finito con pila y un traductor finito con pila, es que al traductor finito le añadimos un nuevo conjunto de símbolos de salida y una función de traducción que a partir de un símbolo de entrada obtiene un símbolo del nuevo alfabeto de salida. Así pues, la definición del traductor finito con pila es la tupla $TP=<\Sigma,\Gamma,Q,f_t,S,Z_0,Out,f_{out}>$ siendo los dos nuevos parámetros:

- *Out:* Es el conjunto de símbolos o alfabeto de la cadena de salida.
- f_{out}: Es la función de traducción definida por: $f_{out}: Q \times \Sigma \rightarrow S^*$

Por la forma que tiene la función de traducción f_{out}, se puede afirmar que el traductor, así definido, es un máquina de Mealy [Mealy 1955]. Es un traductor cuyo símbolo de salida depende tanto del estado del autómata como del símbolo de entrada.

3. El modelo MINERVA

Una vez establecidas las definiciones generales de gramática, autómata finito con pila y traductor finito con pila, se van a aplicar dichas definiciones a la gramática concreta de MINERVA.

En MINERVA se van a definir dos gramáticas. La primera gramática denotada por M, es la gramática que describe la generación de cadenas de agentes correctamente escritas y es la encargada de evolucionar el sistema en el tiempo. La segunda gramática denotada por V, es la que da formato a las cadenas formadas por transformaciones y primitivas que serán tratadas por el sistema de renderizado.

En primer lugar presentamos los principales elementos que conforman MINERVA (agentes, eventos, primitivas y transformaciones) para, a

Universitat d'Alacant
Universidad de Alicante

continuación, describir las gramáticas M y V y los demás elementos del modelo.

3.1. Definición de los símbolos de MINERVA

3.1.1. Agentes y Eventos

La gramática M está formada por los agentes y los eventos que hacen evolucionar el mundo virtual a lo largo del tiempo. En esta gramática, los agentes están representados por el conjunto A_{ATR}^{E}, donde *ATR* es el conjunto de atributos que representan los estados de los agentes y los datos de un evento y *E* es el conjunto de los eventos que el agente recibe.

Para facilitar la comprensión del modelo, proponemos una notación homogénea de los diferentes elementos, formada por un prefijo, que indica el tipo de elemento, un identificador y una lista de atributos en forma de subíndices. En el caso de los agentes, el prefijo es la letra 'a' e incorporan, además del identificador y los atributos, una lista de eventos que el agente puede procesar en forma de superíndice. En el caso de los eventos, utilizaremos la letra 'e' como prefijo e incorporaremos, como en los demás elementos, un identificador y una lista de atributos. Por ejemplo, $aNave_{\langle id, velocidad \rangle}^{eDisparo}$ es un agente que recibe el evento de *eDisparo* y tiene como atributos un identificador y la velocidad de la nave.

3.1.2. Primitivas y transformaciones

Los elementos descriptivos que forman la gramática V son las primitivas y las transformaciones. Se define *P* como el conjunto de símbolos que representan primitivas y que forman parte de la representación del sistema donde cada primitiva se denota con el prefijo 'p' seguido de un identificador. Por ejemplo, *pEsfera* sería la primitiva que simboliza el dibujo de una esfera. Por otro lado, las transformaciones están representadas por el conjunto de símbolos T_{ATR}, donde *ATR* es el conjunto de atributos necesarios que especifican los parámetros de la transformación y sus símbolos están prefijados por la letra 't' seguido de un identificador. Así, $tDesplazamiento_{<dx,dy,dz>}$, es un símbolo perteneciente a T_{ATR}, que representa un desplazamiento en 3D donde *dx, dy* y *dz* son los atributos que establecen la cantidad de desplazamiento en cada uno de los ejes.

3.2. Traductores de MINERVA

Gracias a la teoría de lenguajes y autómatas se puede establecer una gramática que define la estructura de las cadenas del lenguaje L(M) y otra gramática que define la estructura de las cadenas del lenguaje L(V). Ambas gramáticas son independientes del contexto y no recursivas por la izquierda.

Con una gramática de estas características se puede establecer un autómata que puede determinar, dada una cadena, si ésta pertenece o no al lenguaje. Sin embargo, lo más importante es que se puede diseñar un traductor que dada una cadena genera otra cadena.

Así, es posible construir un traductor que dada una cadena de agentes genere otra cadena de agentes representando la evolución del sistema. A este traductor lo hemos llamado Traductor de Evolución (TE).

Al usar otro lenguaje para el sistema de visualización, también nos interesa un traductor que transforme una cadena de L(M) a otra que pertenezca a L(V), es decir, existe un proceso que transforma agentes en primitivas y transformaciones. A este traductor se le denominará Traductor de Visualización (TV).

Por último, para que cada primitiva se pueda dibujar en un dispositivo gráfico y aprovechando que L(V) es un lenguaje formado por cadenas, se puede diseñar otro traductor que transforme las primitivas y las transformaciones en las acciones necesarias para dibujar la escena en un dispositivo gráfico. Este traductor va a variar según el dispositivo, y lo vamos a llamar Traductor de Renderizado (TR). Aunque el TR sea distinto para cada dispositivo, sus características quedan encerradas en el traductor. Gracias al lenguaje L(V) la reutilización está asegurada, ya que sólo reimplementando las primitivas y las transformaciones para el nuevo dispositivo se puede conseguir que todo el modelo representado por MINERVA puede ser trasladado a otro dispositivo gráfico de forma eficaz.

Por tanto, el modelo está estructurado por dos lenguajes L(M) y L(V) y sus algoritmos asociados tanto a la evolución del sistema como al dibujado del mismo serán traductores que están bien especificados por la teoría de lenguajes y autómatas. En la Figura 20, se representa gráficamente la forma en que los traductores actúan sobre las cadenas de los diferentes lenguajes.

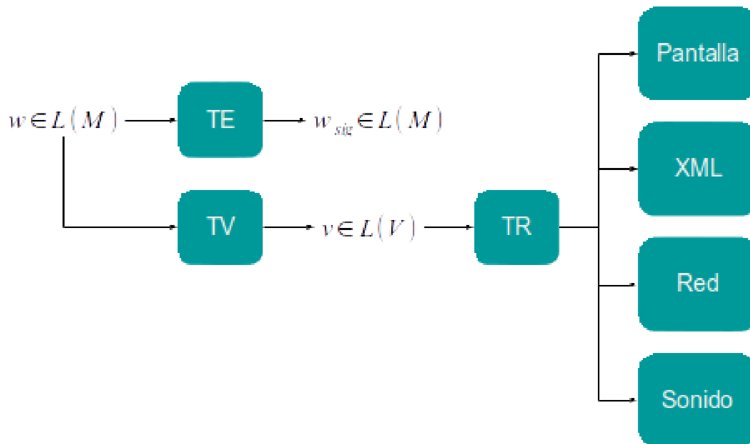


Figura 20: Dependencias entre traductores

A continuación, se va a describir de forma exhaustiva cada uno de los componentes que forma parte del modelo utilizando el formalismo especificado por la teoría de lenguajes y autómatas.

3.3. Definición de la gramática M y su autómata finito con pila

Ahora, se procede a definir la gramática M y el autómata finito de pila que nos dice si una cadena forma parte de L(M). Establecida la forma general de la gramática de un lenguaje, se define la gramática $M=<\Sigma,N,S,R>$ donde:

- $\Sigma = O \cup A^{E}_{ATR}$ es el conjunto de símbolos no terminales donde $O=\{\cdot()\}$ es el conjunto de separadores y operadores y A^{E}_{ATR} el conjunto de símbolos que representan agentes que van a ejecutar actividades dependiendo de los eventos definidos por *E*, el conjunto de eventos que pueden ser recibidos por el agente. *ATR* define el conjunto de atributos que establece el estado del agente en un determinado instante de tiempo y los datos asociados al evento.
- *N={MUNDO, AGENTE}* es el conjunto de símbolos no terminales.
- *S = MUNDO.*

Universitat d'Alacant
Universidad de Alicante

- *R* es el conjunto de reglas gramaticales donde:

Regla 1.- MUNDO → AGENTE · MUNDO | AGENTE Regla 2.- AGENTE → a^{e}_{atr} | a^{e}_{atr}(MUNDO) donde $a^{e}_{atr} \in A^{E}_{ATR}$

Una cadena $w \in \Sigma^*$ es generada por la gramática M si se puede obtener aplicando sucesivamente las reglas de producción del conjunto *R* a partir del símbolo inicial *MUNDO*. El conjunto de cadenas así generadas forman parte del lenguaje generado por la gramática M y lo vamos a denotar como L(M). Es decir que:

$$L(M) = \{ w \in \Sigma^* | MUNDO \overset{*}{\rightarrow} w \}$$

La gramática M es una gramática independiente del contexto, y además no es recursiva por la izquierda. Esto asegura la existencia de un autómata finito con pila que verifica si dada una cadena está definida correctamente o no. Es decir, existe un autómata finito con pila que podrá determinar, por ejemplo, que la cadena $a_1\ a_2\ (a_3\ a_1) \in L(M)$ y la cadena $a_2\ (a_3 \notin L(M)$.

Para el caso de la gramática M, el autómata con pila $APM = \langle Q, \Sigma, N \cup \Sigma, f_t, S, Z0 \rangle$ *se define* como:

- $Q = \{q\}$.
- $\Sigma = \{ \{ \cdot\ , (\ ,) \} \cup A^{E}_{ATR} \}$
- $N \cup \Sigma = \{ \{ \cdot\ , (\ ,), MUNDO, AGENTE \} \cup A^{E}_{ATR} \}$
- $S = \{q\}$.
- $Z_0 = MUNDO$.
- f_t *definido por:*

  $f_t(q, \varepsilon, MUNDO) = \{(q, AGENTE)\}$

  $f_t(q, \varepsilon, MUNDO) = \{(q, AGENTE \cdot MUNDO)\}$

  $f_t(q, \varepsilon, AGENTE) = \{(q, a^{d}_{atr})\}$

  $f_t(q, \varepsilon, AGENTE) = \{(q, a^{d}_{atr}\ (MUNDO))\}$

  $f_t(q,\ a^{d}_{atr}, a^{d}_{atr}) = \{(q,\ \varepsilon)\}$; $f_t(q,\ (, () = \{(q, \varepsilon)\}$; $ft(q,),)) = \{(q,$

$\varepsilon)\}$; $ft(q, \cdot, \cdot) = \{(q, \varepsilon)\}$

Ya está especificado el autómata finito APM que nos dice si una cadena pertenece o no al lenguaje L(M) que define mundos virtuales especificados con agentes. Es decir, según se ha explicado anteriormente, las cadenas aceptadas por APM son las mismas que genera el lenguaje L(M).

3.4. Definición de la Gramática V y su autómata finito con pila

Ahora se va a definir la gramática V que es la gramática que genera cadenas necesarias para definir los elementos que van a ser renderizados en el dispositivo gráfico. La gramática es $V=<\Sigma, N, S, R>$ donde:

- $\Sigma=O\cup T\cup P$ es el conjunto de símbolos no terminales donde $O=\{\cdot()\}$ es el conjunto de separadores, T_{ATR} el conjunto de símbolos que representan transformaciones con atributos y P el conjunto de símbolos que representan primitivas.
- $N = \{ESCENA, OBJETO\}$ es el conjunto de símbolos no terminales.
- $S = ESCENA$.
- R es el conjunto de reglas gramaticales donde:

Regla 1.- ESCENA → OBJETO · ESCENA | OBJETO
Regla 2.- OBJETO → t_{atr}(ESCENA) | p donde $t_{atr}\in T_{ATR}$, $p\in P$.

El conjunto de cadenas generadas por la gramática V se denota como L(V) donde:

$$L(V)=\{w\in\Sigma^{*} | ESCENA \overset{*}{\rightarrow} w\}$$

La gramática V, al igual que M, es una gramática independiente del contexto y además no es recursiva por la izquierda. Esto significa que se puede asegurar la existencia de un autómata finito con pila que verifica si dada una cadena está definida correctamente o no.

Para el caso de la gramática V, el autómata con pila $APV=<Q,\Sigma,N\cup\Sigma,f_t,S,Z_0>$ será definido como:

Universitat d'Alacant
Universidad de Alicante

- $Q=\{q\}$.
- $\Sigma=\{\{\cdot\ , (\ ,)\}\cup T\cup P\}$
- $N\cup\Sigma= \{\{\cdot\ , (\ ,), ESCENA, OBJETO\}\cup T\cup P\}$
- $S=\{q\}$.
- $Z_0 = ESCENA$.
- f_t *que se define:*

  $f_t(q,\ , ESCENA) = \{(q, OBJETO)\}$

  $f_t(q,\ , ESCENA) = \{(q, OBJETO\quad ESCENA)\}$

  $f_t(q,\ , OBJETO) = \{(q, t_{atr}(ESCENA))\}$

  $f_t(q,\ , OBJETO) = \{(q, p)\}$

  $f_t(q,\ t_{atr}, t_{atr}) = \{(q, \varepsilon)\}; ft(q, p, p) = \{(q, \varepsilon)\};$

  $f_t(q,\ (, () = \{(q,\ )\}; f_t(q,),)) = \{(q, \varepsilon)\}; ft(q, \cdot, \cdot) = \{(q, \varepsilon)\}$

Ya está especificado el autómata finito APV que nos dice si una cadena pertenece o no al lenguaje L(V) que define la representación de los mundos virtuales en forma de primitivas y transformaciones. Es decir, según se ha explicado anteriormente, las cadenas aceptadas por APV son las mismas que genera el lenguaje L(V).

Llegados hasta aquí, se diseñarán los traductores necesarios para que el sistema realice sus funciones, a saber, que evolucione en el tiempo y que se dibuje la escena en los dispositivos gráficos. Para ello, como ya se ha comentado, son necesarios tres traductores: el traductor de evolución, el traductor de visualización y el traductor de renderizado.

A continuación, se va a describir cada uno de los traductores partiendo de la definición general de traductor ya especificada anteriormente.

3.5. Traductores de Evolución (TE)

En un sistema de realidad virtual es necesario que el mundo evolucione con el tiempo para que tenga un cierto interés. En el caso de MINERVA, la parte evolutiva del sistema la resuelve el traductor de evolución del sistema o TE. En este traductor, se utilizan las cadenas generadas por la gramática

Universitat d'Alacant
Universidad de Alicante

M como cadenas de entrada y tiene como salida una cadena que también pertenece a L(M). Ya hemos visto como construir un traductor finito de pila a partir de un autómata finito de pila. Por ello, sólo nos detendremos en los tres nuevos elementos de la tupla $<\Sigma,N\cup\Sigma,Q,f_t,S,Z_0,Out,f_r>$ que debemos definir para cada traductor: el conjunto de estados Q, el alfabeto Out y la función de traducción f_r. Cada agente va a tener una función de traducción f_r distinta. En esta función se transformará el agente actual en la traducción evolucionada. Además podrá añadir al estado del autómata los eventos necesarios para la evolución. Por otro lado, en el caso del traductor TE $Out=N\cup\Sigma$, es decir, se traduce en cadenas del mismo alfabeto.

La función de traducción f_r se define de la siguiente forma:

$$f_r(q,s)=\begin{cases} s & Si\ s\in O \\ f_{evolucion}(q,s) & Si\ s\in A_{ATR}^{E} \end{cases}$$

Donde se especifica la función de traducción $f_{evolucion}$ que es la función de evolución del agente. La función de evolución del agente es una función que transformará un agente en cadenas del lenguaje L(M) y se puede interpretar como la evolución del agente en el tiempo. Su definición es:

$$f_{evolucion}: Q\times A_{ATR}^{E}\rightarrow\left(A_{ATR}^{E}\right)^{*}$$

La función $f_{evolucion}$ tendrá diferentes expresiones dependiendo de cómo evolucione en el tiempo. Sin embargo, se puede definir la expresión general como:

$$f_{evolucion}(q,a_{atr})=\begin{cases} u_1\in L(M) & e_1\in q.eventos \\ u_2\in L(M) & e_2\in q.eventos \\ \dots & \\ a_{atr} & \forall e_i\notin q.eventos \end{cases}$$

El resultado de la función de evolución puede contener o no al propio agente. Esta función modifica la cadena de salida dependiendo de los eventos actuales y puede decidir evolucionar de una forma o de otra según los eventos. También existe la posibilidad de que para ciertos eventos se genere la cadena vacía.

En el traductor TE, el conjunto de estados Q está determinado por un conjunto de tuplas $<q,(e_1,e_2...e_n)>$ donde q es el estado de transición del autómata finito ya especificado en el autómata con pila y $(e_1,e_2...e_n)$ es una lista de eventos donde $e_1,e_2...e_n$ son los eventos actuales del sistema.

Universitat d'Alacant
Universidad de Alicante

La función de evolución puede cambiar el estado q para añadir eventos del sistema. Efectivamente, como se vio con anterioridad los estados de Q son tuplas compuestas por un estado de transición y por una lista de eventos. El estado de transición es la configuración necesaria para que el autómata finito verifique que la cadena que se está procesando está dentro de las reglas gramaticales y que es siempre *q*. Por otro lado, la lista de eventos puede ser modificada por la función de evolución. Para la modificación de la lista de eventos se definen las siguientes operaciones:

$$q.insert: Q \times E \rightarrow Q$$
$$q.delete: Q \times E \rightarrow Q$$

La primera función, *q.insert*, añadirá un evento a la lista, mientras que la segunda, *q.delete* borrará el evento de la lista si existe el evento. Esto significa que el agente tiene la posibilidad de generar eventos con determinadas características. Por ejemplo, se podría definir un agente que captura los eventos del botón de un joystick y añade a la lista de eventos un evento que representa un disparo. Esta lista modificada pasaría a las siguientes fases del autómata para que otros agentes gestionen su comportamiento según los eventos que estén en la lista.

Una vez definido el traductor de evolución del sistema se puede concluir que este traductor es el responsable de la actividad de todo el sistema. Esto supone que la cadena generada por el traductor TE representa el estado del sistema en un instante determinado. Por otro lado, analizando la cadena en un instante de tiempo se podría averiguar si el sistema es abierto o cerrado simplemente analizando si existen agentes que están relacionados con dispositivos de usuarios. Como ya sabemos, el traductor es un algoritmo que representaremos con la siguiente función:

$$TE: L(M) \rightarrow L(M)$$

El traductor de evolución es donde se modelará tanto el motor físico como el motor de inteligencia artificial emitiendo eventos o recibiéndolos, según el agente.

3.6. Traductor de Visualización (TV)

El objetivo que tiene el traductor de visualización (TV) es el de traducir los agentes de una cadena perteneciente a L(M) y transformarlos en primitivas y transformaciones dependiendo del estado del agente representado por sus

atributos. Esto quiere decir que dada una cadena perteneciente a L(M) devolverá otra cadena que pertenezca a L(V).

El TV es parecido al TE, salvo que la función de evolución del agente en vez de devolver cualquier tipo de cadena perteneciente al lenguaje L(M), devuelve una cadena que pertenece a L(V). Esto significa que el traductor TV se define de la misma forma que TE salvo que la función de evolución cambia y el conjunto *Out* del traductor es $Out = T_{ATR} \cup P$. Ahora, la cadenas se traducen en cadenas que están formadas por primitivas y transformaciones. A esta función de evolución del agente, y a fin de diferenciarla de la función de evolución, se la denominará en adelante función de visualización de un agente y está denotada por $f_{visualizacion}$ y cuya definición es $f_{visualizacion} : Q \; x \; A^{E}_{ATR} \rightarrow (T_{ATR} \cup P)^{*}$.

Al igual que ocurre con la función de evolución, la función de visualización tendrá una forma genérica que puede ser descrita de la siguiente forma:

$$f_{visualizacion}(q, a_{atr}) = \begin{cases} u_1 \in L(V) & e_{v1} \in q.eventos \\ u_2 \in L(V) & e_{v2} \in q.eventos \\ \dots & \\ \varepsilon & \forall e_{vi} \notin q.eventos \end{cases}$$

Con esta última expresión se pueden observar varias diferencias con respecto a la función de evolución. En primer lugar, lo que se ha especificado antes: las cadenas que devuelve la función de visualización son cadenas del lenguaje L(V). En segundo lugar, si el agente no tiene ningún evento que pertenezca a la lista de eventos, entonces, la cadena devuelta es la cadena vacía. Por último, señalar que los eventos que se alojan en el estado del autómata son eventos de visualización.

En definitiva, el traductor de visualización es el responsable de preparar la escena para que pueda ser visualizada por el traductor de renderizado, pudiendo realizar varias vistas de la escena según convenga. El algoritmo asociado a este traductor lo definimos con la función:

$$TV : L(M) \rightarrow L(V)$$

3.7. Traductor de Renderizado (TR)

El proceso de renderizado consiste en procesar un conjunto de datos que

representan figuras y transformaciones geométricas y transformarlo en acciones sobre un sistema gráfico. En el caso del modelo formal MINERVA, los datos geométricos están representados y estructurados mediante cadenas de la gramática V que sólo consisten en primitivas y transformaciones.

El último paso consiste en traducir una cadena *v* perteneciente al lenguaje L(V) en un conjunto de acciones sobre el sistema gráfico. Si definimos G como la gramática que define el conjunto de acciones sobre dicho sistema gráfico entonces se puede definir un traductor que transforma las cadenas de L(V) al lenguaje L(G). A dicho traductor se le denominará traductor de renderizado (TR) y queda definido por la tupla $<\Sigma,N\cup\Sigma,Q,f_t,S,Z_0,Out,f_r>$. La definición del traductor es igual que la definición general, salvo que el conjunto de símbolos terminales Σ es el alfabeto compuesto por primitivas P y transformaciones T, *Out=L(G)* es el conjunto de las acciones sobre el sistema gráfico y la función de traducción f_r traduce las cadenas de primitivas y transformaciones en cadenas del conjunto L(G).

La ventaja de usar el TR es significativa ya que cambiando la gramática G se puede, para una misma cadena, generar varias salidas a distintos sistemas geométricos. Por ejemplo, se puede diseñar un TR para que una cadena de L(V) se convierta en primitivas y transformaciones de la API de OpenGL y otro TR para traducir la misma cadena en primitivas y transformaciones de la API de DirectX sólo cambiando la gramática G.

En el caso de las primitivas, la traducción entre un símbolo del alfabeto y la acción sobre el sistema gráfico será directa. Así, por ejemplo, si es *pEsfera* una primitiva perteneciente al conjunto de primitivas P y `glutSolidSphere` la función de la API de OpenGL que dibuja una esfera, la función f_r se encargará de transformar el símbolo *pEsfera* en la ejecución de `glutSolidSphere`. Si el caso es el del traductor de Direct3D y `sphereDirectX` es la función que dibuja una esfera en Direct3D entonces la función f_r ejecutará dicha función.

Por otro lado, la traducción en el caso de las transformaciones no es tan directa como en el caso de las primitivas. Esto se debe a que las transformaciones tienen un ámbito de aplicabilidad que debe ser respetado. Por ejemplo, no es lo mismo hacer un giro y luego un desplazamiento que primero hacer un desplazamiento y después un giro.

En el lenguaje, este ámbito se corresponde con los paréntesis que define

Universitat d'Alacant
Universidad de Alicante

la transformación. Esto significa que el traductor TR debe traducir los paréntesis a dos funciones:

1. Función de Inicio de Transformación (IT) y que se ejecutará cuando TR traduzca el símbolo de apertura de paréntesis '(' .
2. Función de Fin de Transformación (FT) y que se ejecutará cuando TR traduzca el símbolo de cierre de paréntesis ')'.

Estas funciones serán las encargadas de velar por la consistencia de la aplicación de las transformaciones dentro de la escena y que están definidas en el lenguaje por los paréntesis. Por ejemplo, en el caso de un TR que renderice la escena con la API OpenGL, la función IT se define como `glPushMatrix` que apila la matriz de transformación actual y después aplicaría la transformación correspondiente. En el caso de encontrar el símbolo ')', entonces la función FT debe ejecutar la función `glPopMatrix` de OpenGL.

El algoritmo que traduce las cadenas del lenguaje L(V) en cadenas del lenguaje L(G) se define como:

$$TR: L(V) \rightarrow L(G)$$

3.8. Algoritmo del sistema

Una vez descritos todos los traductores utilizados en el sistema, se debe establecer el orden de ejecución de los traductores para que el sistema del mundo virtual pueda ejecutarse de forma correcta. Para ello, se definen las siguientes variables y funciones muchas de ellas ya introducidas a lo largo de este capítulo:

- TE: Función que representa el algoritmo del traductor de evolución.
- TV: Función que representa el algoritmo del traductor de visualización.
- TR: Función que representa el algoritmo del traductor del renderizado.
- w_0 = Cadena inicial perteneciente a L(M).

- w = Cadena evolucionada perteneciente a L(M).
- v = Cadena de visualización perteneciente a L(V).
- g = Cadena de renderización perteneciente a L(G).

Una vez definidas las variables, proponemos el siguiente algoritmo para la ejecución de MINERVA:

Paso 1: $w = w_0$.
Paso 2: while ($w <> \varepsilon$)
Paso 3: $w_{sig} = TE(w)$
Paso 4: $v = TV(w)$
Paso 5: $g = TR(v)$
Paso 6: dibujar(g)
Paso 7: $w = w_{sig}$
Paso 8: endwhile
Paso 9: end

El primer paso del algoritmo consiste en asignar a la cadena actual la cadena inicial del sistema

En el paso 2 verificamos la condición de finalización del algoritmo de tal manera que el algoritmo finaliza cuando la cadena actual es vacía. Si no es vacía entonces realizamos los pasos de un frame.

En el paso 3 la cadena actual *w* evoluciona al siguiente estado gracias al traductor TE y se almacena en la cadena w_{sig}. En el paso 4 la misma cadena *w* se transforma en una cadena del lenguaje L(V) para poder dibujarla en el dispositivo de salida mediante el paso 5 gracias la traductor de renderizado.

En el paso 7 asignamos la cadena evolucionada, resultado del traductor TE, a *w* para la siguiente iteración del bucle.

Universitat d'Alacant
Universidad de Alicante

La Figura 21 presenta de forma gráfica el algoritmo de ejecución de MINERVA.

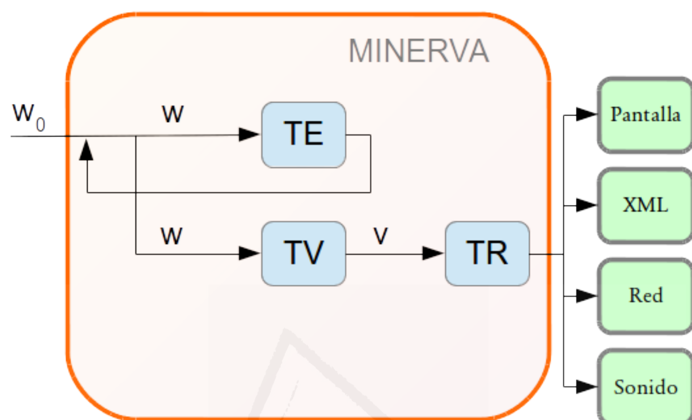


Figura 21: Algoritmo de ejecución de MINERVA

Hasta ahora se ha hecho una descripción exhaustiva de todo el modelo formal de MINERVA incluido el algoritmo que relaciona todos los traductores. Llegado a este punto se puede empezar a analizar el sistema y ver qué características puede aportar a los sistemas de realidad virtual.

Universitat d'Alacant
Universidad de Alicante

V. Patrones de diseño

La Teoría de Lenguajes y Autómatas que da soporte al modelo hacen de MINERVA un modelo singular, debido a la aplicación de modelos lingüísticos a sistemas de realidad virtual.
Sin embargo, este tipo de formalismos exige que en el proceso de diseño de las aplicaciones, se encuentren ciertas estructuras que normalmente son ajenas a los sistemas de realidad virtual y que, por lo tanto, hacen necesario un esfuerzo de adaptación.
Para ayudar en este cambio de estructuras se va a proponer un conjunto de patrones de diseño que servirán como guías para empezar a modelar un sistema con MINERVA.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

1. Patrones de diseño.

Ya se ha presentado el modelo formal que sustenta a MINERVA. Es una descripción de los componentes necesarios para definir un sistema de realidad virtual, así como la forma en que deben ensamblarse los distintos elementos para que el sistema funcione.

Para diseñar un sistema con este modelo formal, se recomendarán unos patrones de diseño que, por un lado, demostrarán la aplicabilidad de MINERVA a varios tipos de problemas, y por otro, darán lugar a una metodología para construir de forma estructurada y metódica una solución al problema planteado. Esta metodología no es, ni mucho menos, de obligado cumplimiento y no modifica en ningún punto el modelo formal, pero sí ayudará en el diseño estructural, marcando ciertos caminos ya explorados que nos llevan a soluciones correctas.

En lineas generales, la metodología que se aplica para desarrollar un sistema con MINERVA se puede descomponer en los siguientes pasos:

a) Especificar los agentes del modelo: Detectar cuáles son los agentes necesarios que van a formar parte del sistema y cómo se relacionan entre sí.

b) Enumerar el conjunto de eventos: Estudiar cuáles son los eventos que van a desencadenar la modificación de los atributos de los agentes y generalizarlos lo máximo posible para que tengan el menor acoplamiento posible con los dispositivos de entrada.

c) Definir la función de evolución de los agentes: Diseñar la función de evolución del agente en el tiempo, es decir, la forma en que se modifica el estado del agente dependiendo de los eventos que puede procesar.

d) Definir primitivas y transformaciones: Establecer qué primitivas y transformaciones serán necesarias para poder representar el mundo virtual en un dispositivo de salida según su estado.

e) Determinar los dispositivos de salida: Identificar los dispositivos de salida tanto gráficos como de otro tipo.

Universitat d'Alacant
Universidad de Alicante

Esta identificación va a tener una relación directa con las primitivas y las transformaciones.

f) Detallar los dispositivos de entrada: Estudiar qué dispositivos de entrada va a tener el sistema y modelarlos en agentes que generarán los eventos necesarios para que otros agentes actúen según estos dispositivos de entrada.

Estos pasos, como se puede observar, son muy generales, pero son puntos de partida para iniciar el proceso de análisis y diseño del sistema. Hay que destacar que los puntos del a al d establecen los elementos del sistema más abstractos y por tanto deben estar lo más alejados posible de los dispositivos. Si el nivel de abstracción es el adecuado, los elementos podrán ser reutilizados en entornos de ejecución totalmente diferentes.

En los puntos e y f se trata con los elementos que tocan directamente con los dispositivos tanto de entrada como de salida. Son los puntos más delicados a la hora de llevar el sistema de un entorno de ejecución a otro. Si el diseño de estos elementos es correcto, el impacto de sustitución de un elemento que gestione un dispositivo por otro, tendrá un coste mínimo y hará el sistema fácilmente extensible.

Aunque se han numerado los pasos, éstos no tienen porque ir en orden e incluso se puede establecer un bucle de refinamiento de los elementos entre los diferentes pasos.

Con estas consideraciones se va a detallar cada uno de los pasos. Después, se estudiará cómo utilizar MINERVA como motor gráfico, como sistema de realidad virtual, como sistema multiagente y como simulador.

1.1. Definición de agentes

Los agentes son los principales elementos que usa MINERVA para poder desarrollar toda la actividad dentro del sistema de realidad virtual. Son los elementos constitutivos de la evolución del sistema y el núcleo de procesamiento.

Existen varias formas de clasificar los agentes, como por ejemplo la clasificación de Nigel Gilbert en [Gilbert 2007]. Pero examinando la formalización de los agentes de MINERVA, se puede sugerir una clasificación de agentes dependiendo de cómo se defina su función de evolución, qué tipo de eventos va a manejar, su representación en forma de

primitivas y transformaciones, cómo se comunica con el resto de agentes y su relación con los agentes que lo componen.

Proponemos la siguiente clasificación según los criterios comentados.:

1. Según su función de evolución.

    a) Agentes pasivos: son agentes cuya función de evolución es la identidad. Por lo tanto no evolucionan a lo largo del tiempo. Es el caso de los elementos estáticos de un mundo virtual.

    b) Agentes reactivos: su función de evolución les confiere un comportamiento reactivo, es decir, su proceso de evolución sigue un patrón estímulo/respuesta, de forma que se reacciona al estímulo (evento) a través de acciones sin deliberación.

    c) Agentes deliberativos: su función de evolución incorpora comportamientos con una inteligencia más elaborada. La evolución del agente introduce una función deliberativa entre la percepción del evento y la ejecución para elegir la acción correcta.

2. Según al tipo de eventos a los que responde

    a) Agentes de usuario: responden únicamente a eventos generados por los dispositivos de interacción del usuario. Un caso típico son los personajes de videojuegos manejados por usuario.

    b) Agente autónomos: no requieren la participación del usuario para su evolución, sino que tienen su propio comportamiento autónomo. Un ejemplo son los bots presentes en los videojuegos.

3. Según su representación.

    a) Agentes visualizables: agentes que responden a eventos de visualización para generar una cadena de primitivas y

transformaciones que pueda visualizarse en un sistema de representación gráfico o de otro tipo.

b) Agentes no visualizables: agentes que no responden a eventos de visualización y, por lo tanto, no tienen una representación en el mundo virtual. Un ejemplo son los agentes que generan los eventos de usuario o los que controlan las funciones del motor físico.

4. Según el tipo de comunicación.

a) Agentes generadores de eventos: agentes cuya misión es lanzar eventos al sistema. El ejemplo más claro es el de los agentes que traducen eventos de dispositivo en eventos que pueden ser interpretados por el sistema.

b) Agentes receptores de eventos: agentes centrados en la percepción, es decir, que reaccionan ante los eventos que se producen en el sistema.

La anterior clasificación no establece tipos disjuntos, sino que los agentes pertenecen a varios tipos de los descritos anteriormente. Es más, rara vez los agentes tienen características puras de un tipo determinado. Casi todos los agentes útiles que se pueden describir son híbridos, compartiendo características de varios tipos.

Una vez establecida la clasificación, se explicará cómo se diseñan los tres elementos que definen un agente según su tipo, es decir, las características que tiene su función de evolución, los tipos de eventos a los que debe reaccionar y cómo se transforma en primitivas y transformaciones.

1.1.1. Según su función de evolución

La función de evolución es la que determina cómo los agentes evolucionan a lo largo del tiempo. Esta función puede implementar varias características desde la generación de eventos, hasta la toma de decisiones, pasando por la reacción a determinados sucesos que acontecen en la escena.

Hay que considerar que el coste temporal de la función de evolución es muy importante ya que impactará sobre el rendimiento global del sistema. Existen agentes con una actividad intensa dentro del mundo virtual. Por

ejemplo, aquellos que deben reaccionar a casi todos los eventos generados por el sistema y deben adaptar su comportamiento según lo que acontece a su alrededor. Por otro lado, el número de agentes que pueden formar parte de un sistema puede ser grande. Por ejemplo, se puede imaginar un juego de realidad virtual ejecutado en red con cientos de jugadores interaccionando a la vez.

Por tanto, si se tiene en cuenta por un lado el número de agentes, por otro el número de eventos a procesar, que algunos eventos están relacionados entre sí a la hora de ser procesados por el agente, aumentando así su complejidad y que todo esto debe ejecutarse en un tiempo lo suficientemente corto para que no afecte al rendimiento del sistema, se puede afirmar que el diseño de la función de evolución para cierto tipo de agentes es delicado, extremadamente sensible y su procesamiento debe ser extremamente rápido si se desea que el sistema se ejecute en tiempo real.

Agentes pasivos

Los agentes pasivos son agentes cuya función de evolución es vacía. No reaccionan a ningún tipo de eventos y sólo existen para representar obstáculos para otro tipo de agentes. De ahí su función pasiva.

Sin embargo, este tipo de agentes generalmente sí que deben ser detectados por otros agentes para realizar, por ejemplo, una labor de obstáculo. Además, habitualmente tienen una representación visual en el mundo virtual para indicar su presencia dentro del mismo.

Un ejemplo de agente pasivo puede ser un árbol. El árbol puede ser considerado como una decoración dentro del mundo virtual que además tiene una presencia dentro del mismo. Sirve como obstáculo ya que existirán agentes que no deben atravesar el árbol. Es decir, el árbol tiene una presencia y una función, obstaculizar a los demás elementos del sistema. Al igual que un árbol, se pueden considerar agentes pasivos muros, montañas y todo tipo de agentes que no se muevan o no ejecuten algún tipo de acción sobre el mundo virtual.

En este tipo de agentes sólo se debe tener especial cuidado a la hora de transformarlos en primitivas y transformaciones ya que el número de elementos pasivos dentro de un sistema de realidad virtual puede ser alto. Por ejemplo, si se necesita diseñar un bosque de árboles, el número de agentes pasivos puede ser alto, generando un gran número de elementos que luego el traductor de visualización y el traductor de renderizado deben

Universitat d'Alacant
Universidad de Alicante

procesar aumentando el tiempo de procesamiento de la escena.

Es posible optimizar los procesos asociados a los agentes pasivos. Por ejemplo, este tipo de agente puede ser agrupado y procesado al inicio de la evolución ya que no van a modificar su estado en todo el tiempo que dure la ejecución del sistema.

Agentes reactivos

Los agentes reactivos ocupan el siguiente nivel de complejidad en las funciones de evolución. Dependiendo de cómo se plantee la función de evolución, el grado de inteligencia que puede tener el agente es menor o mayor. Considerando la gama más baja de inteligencia se puede diseñar un agente independiente totalmente reactivo. En este caso, sólo ejecutará acciones sencillas sin ningún tipo de evaluación y con una reacción impulsiva a los eventos que le lleguen del entorno.

Una típica función de evolución reactiva sería una función del tipo “si condición entonces acción”, donde tanto la condición como la acción son de complejidad sencilla, tales como reaccionar a un evento sin evaluar sus atributos o una modificación de un atributo del agente.

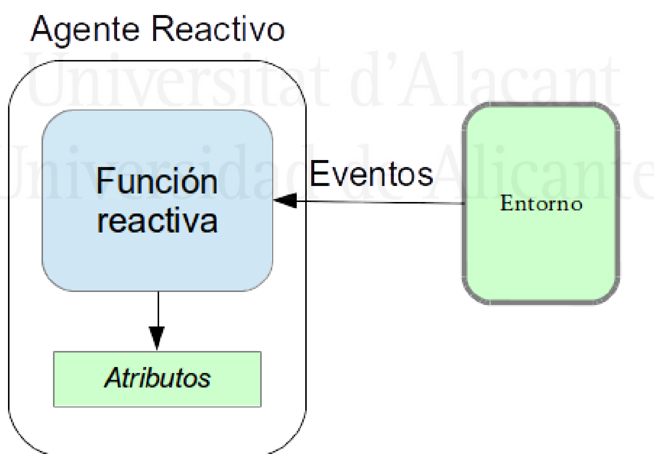


Figura 22: Agente Reactivo

Un ejemplo claro de los agentes reactivos son los manejados por los usuarios. Este tipo de agentes reaccionan con acciones simples a los eventos

lanzados por los dispositivos de entrada manejados por los usuarios.

Agentes deliberativos

Los agentes deliberativos son agentes con un grado de complejidad alto. Su función de evolución suele ser compleja tanto desde un punto de vista computacional como desde un punto de vista cognitivo.

Estos agentes tienen la habilidad de decidir qué acciones realizar dependiendo de criterios establecidos en su función de evolución mediante una valoración exhaustiva de los datos recibidos del entorno a través de los eventos.

En las funciones de evolución de estos agentes se pueden implementar todas las técnicas de inteligencia artificial que existen con la apropiada adaptación.

Tanto los agentes reactivos como los deliberativos tienen la capacidad de modificar su entorno mediante las acciones que ejecutan sus funciones de evolución. Por ejemplo, un agente deliberativo puede ejecutar un disparo cuando tiene un enemigo cerca produciendo un nuevo agente en el sistema, la bala que puede colisionar o no contra el enemigo.

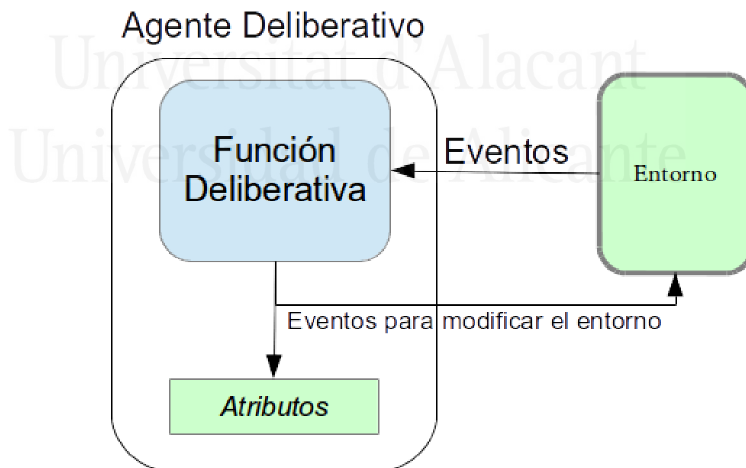


Figura 23: Agente Deliberativo

1.1.2. Según el tipo de eventos a los que responden

Otra forma de clasificar los agentes es según el tipo de eventos a los que responde. Así se tiene que si responde a eventos generados por dispositivos de entrada, se tendrá un agente manejado por un usuario. Por otro lado, si los eventos a los que reacciona son eventos generados por sucesos que se producen internamente en el sistema como el de una colisión y no reaccionan a ningún evento de dispositivo de entrada, entonces los agentes se clasifican como autónomos.

Agentes de usuario

Este tipo de agente deberá ser manejado por el usuario mediante los dispositivos de entrada que generan los eventos oportunos, dependiendo de las órdenes de usuario. Los atributos que componen el agente reflejarán el estado del agente dependiendo de las acciones generadas por el usuario y un tipo de evento podrá modificar uno o varios atributos.

Este tipo de agentes, no sólo debe responder a los eventos del usuario. Dentro del sistema podrán generarse eventos que aunque no están producidos por el usuario pueden afectar a dichos agentes. Son eventos dedicados a establecer las leyes físicas que rigen en el mundo virtual y que deben cumplir las acciones de los diferentes personajes que componen la escena. Así, por ejemplo, si al agente le llega un evento de colisión, debe igualmente reaccionar con coherencia a este tipo de eventos, de la misma forma reactiva que a los eventos generados por el usuario.

El nivel de complejidad de las acciones es variado dependiendo de lo que el usuario desee realizar. Por ejemplo, se puede establecer un comportamiento totalmente reactivo del agente. Es decir, va a realizar lo que le indica directamente el usuario. Las acciones de usuario sobre estos agentes son acciones muy concretas y básicas (mover a la izquierda, mover a la derecha, adelántate un paso, etc). En el caso opuesto, están los agentes que tienen un comportamiento más complejo y que tienen capacidad de planificación. Estos agentes responden a acciones tales como busca el tesoro, busca el camino más corto para llegar a una determinada posición, etc.

Así, resumiendo, un agente de usuario se caracteriza por una función de evolución que sólo ejecuta acciones según determinados tipos de eventos diseñados para tal fin.

Universitat d'Alacant
Universidad de Alicante

Agentes autónomos

Los agentes autónomos deben realizar acciones por su propia cuenta, teniendo en cuenta sus objetivos, sus recursos y lo que acontece a su alrededor. Esto exige que su comportamiento se rija por determinado nivel de inteligencia que se implementará dentro de la función de evolución según los eventos que reciba.

Los agentes autónomos tienen que manejar principalmente dos tipos de eventos: los eventos de recogida de información de su entorno y los eventos de evaluación de la información recogida. Con esta información deben ejecutar la acción adecuada según consideren oportuno. Aquí se debe prestar especial atención al diseño de los eventos tanto de recogida de información como los que van a demandar del agente que realice una determinada acción.

Los eventos de recogida de información transportarán los datos necesarios que describen el entorno en el que se mueve el agente y los sucesos que afectan directamente al agente como por ejemplo una colisión. Esta información, debe tener dos características principales.

La primera es que la información del evento debe describir de una forma sencilla lo necesario para que su procesamiento en la función de evolución tenga el menor coste posible y no pierda tiempo en el procesamiento de los datos del evento. Por ejemplo, imagínese un agente que representa un robot que tiene como sensor una cámara que capta imágenes y que en esas imágenes se desea detectar unos marcadores. Podemos definir un evento *eMarcador*$_{<datos>}$ para que el agente detecte estos marcadores. Pues bien, si en los atributos del evento se envía la imagen del vídeo directamente, el agente tendrá que procesar la imagen buscando si tiene o no el marcador dentro de la función de evolución, con el gran coste que supone esto ya que se debe evaluar la imagen en cada agente. Sin embargo, si el procesamiento de la imagen se realiza en el agente que la capta y en los atributos del evento se incorpora directamente un identificador del marcador que el agente debe procesar, sólo tendrá que evaluar qué hacer cuando vea ese identificador, sin perder tiempo en el procesamiento de los datos para extraer la información que le interesa.

La segunda característica es que los datos de los eventos que describen el entorno deben alejarse lo máximo posible de los sensores. Esto facilitará la separación entre la forma en que se detecta el entorno y la forma en que se procesa por parte de los agentes. De esta manera, la escalabilidad del

Universitat d'Alacant
Universidad de Alicante

sistema aumenta ya que, por ejemplo, los marcadores antes descritos pueden ser detectados por una cámara o mediante un sensor láser o incluso mediante ambos sensores para aumentar la calidad de la información.

Dependiendo de cómo se plantee la función de evolución, el grado de inteligencia que puede tener el agente autónomo es menor o mayor. Considerando la gama más baja de inteligencia se puede diseñar un agente reactivo anteriormente descrito. En el lado opuesto de los agentes reactivos existen los agentes deliberativos, agentes que tienen la habilidad de decidir qué acciones llevar a cabo dependiendo de criterios establecidos en su función de evolución mediante una valoración exhaustiva de los datos recibidos del entorno.

Lo más interesante que tiene la función de evolución es que se pueden diseñar diferentes agentes con distintos paradigmas de inteligencia artificial (algoritmos genéticos, redes neuronales, sistemas bayesianos, etc). De esta manera, se pueden comparar fácilmente cada uno de los modelos bajo los mismos parámetros que definen el entorno.

Uno de los principales atributos de que disponen los agentes autónomos es la memoria, necesaria para tener un comportamiento inteligente. La memoria debe almacenar información que sea fácilmente tratable para que se pueda acceder a ella y facilitar la toma de decisiones. Esta memoria va a ser definida en los atributos del agente y su implementación va a depender del diseño que se establezca para definir el modo en que el agente autónomo procese la información memorizada.

Resumiendo, los agentes autónomos son agentes con funciones de evolución que recogen información de su entorno y que en menor o mayor medida evalúan la acción que deben realizar. Dependiendo del grado de tratamiento de la información recogida, la capacidad de memorizar información y de lo elaborada que sea la toma de decisiones del agente, así será el grado de inteligencia con el que se dota al agente autónomo.

1.1.3. Según su representación

La representación visual de un agente nos indica su estado, en lo que se refiere a su visualización. Sin embargo, existen agentes que son necesarios para la evolución del sistema que no son visibles. Por ejemplo, un agente de usuario debe tener una visualización para que el usuario pueda ver en que lugar se encuentra y cuál es su estado. Pero agentes como los que están relacionados con los dispositivos de entrada pueden no necesitar una

visualización.

Agentes visuales

Los agentes visuales son agentes que tienen una función de visualización que traduce el estado del agente en una cadena de transformaciones y primitivas del lenguaje L(V). Una vez traducida, el traductor de visualización podrá ejecutar las acciones oportunas para transformar los símbolos del lenguaje L(V) en acciones sobre un dispositivo de salida.

A partir de estos agentes visuales, se definirán las transformaciones y primitivas. Este aspecto se estudiará en profundidad en capítulos siguientes.

Agentes no visuales

Los agentes no visuales son aquellos cuya función de visualización devuelve la cadena vacía. Esto hará que para el traductor de renderización, tal agente no exista y por tanto no será dibujado.

Existen varios ejemplos de agentes no visuales como los generadores de eventos que están asociados a dispositivos de entradas o agentes que generan eventos para comunicarlos a los otros agentes que forman parte del sistema. Un ejemplo de tales agentes es el que modela el motor físico.

1.1.4. Según el tipo de comunicación

Universitat d'Alacant
Universidad de Alicante

Otra forma de establecer una clasificación entre los agentes es en función de si genera eventos o si recibe eventos. Esta clasificación no significa que sólo existan agentes que generen eventos exclusivamente o que sólo reciban eventos, más bien al contrario, la mayoría de los agentes podrán tanto generar como recibir eventos.

Agentes generadores de eventos

Los agentes generadores de eventos tienen dos características esenciales. En primer lugar, son los agentes más dependientes de los dispositivos de entrada. Van a recibir eventos de los dispositivos de entrada, como un teclado o un ratón, y van a procesarlos y traducirlos a eventos que los agentes pueden manejar como un tipo de evento más abstracto. Al fin y al cabo, son los agentes que establecen una barrera entre los dispositivos de entrada y los demás agentes. Por ejemplo, se puede diseñar un generador de

eventos que reciba los eventos de un ratón denominado *aRaton*. En este caso, el agente generador de eventos recibirá un evento de pulsado de botón derecho y generará un evento de tipo *eDisparo* que representará la acción de disparo.

En segundo lugar, los agentes generadores de eventos deben establecer a qué agentes pueden enviarles el evento. Como se vio en la descripción de la sintaxis, un agente puede tener a su vez subagentes. Entonces, aprovechando la sintaxis del lenguaje se puede decir que los eventos de un generador de eventos sólo serán procesados por los subagentes que componen el agente. Es decir, en el caso de la cadena $aRaton(a_1 \cdot a_2 \cdot a_3) \cdot a_4$ el agente *aRaton* enviará el evento sólo a a_1, a_2 y a_3 pero no le llegará al agente a_4. De esta manera, se puede reducir el procesamiento sólo a los agentes que van a tener que procesar el evento *eDisparo* optimizando el tratamiento de eventos del sistema.

Hay que tener en cuenta que un agente que genera eventos no tiene porqué generar un solo tipo de evento. Puede generar tantos tipos de eventos como se desee. Siguiendo con el agente *aRaton*, se puede generar no sólo el evento *eDisparo*, también se puede generar *eSalta* si pulsas el botón derecho, $eMueve_{<izq>}$ si se mueve el ratón hacia la izquierda y $eMueve_{<der>}$ si se mueve el ratón a la derecha.

Los eventos del mismo tipo que pueden generar varios agentes pueden solaparse entre ellos. Cuando un agente genera un evento que ya ha sido generado, este no será sustituido por el anterior. De esta forma, se puede especificar una relación de prioridades entre los agentes generadores de eventos. Esta prioridad se establece en la propia cadena que representa el mundo virtual. Por ejemplo, supóngase que se tiene otro agente generador de eventos *aTeclado* que genera los mismos eventos que *aRaton* y que además se define la siguiente cadena: $aRaton(aTeclado(a_1 \cdot a_2 \cdot a_3))$. En este caso, *aRaton* tiene prioridad sobre *aTeclado* ya que si *aTeclado* genera por ejemplo *eDisparo* y este también es generado por *aRaton*, *eDisparo* generado por *aTeclado* no será añadido a la lista de eventos que el traductor de evolución TE tiene en su variable de estado *q*. Sin embargo, si se define la cadena como $aTeclado(aRaton(a_1 \cdot a_2 \cdot a_3))$, aquí *aTeclado* tendrá más prioridad que *aRaton*.

La prioridad de los agentes generadores de eventos va a depender del diseño del sistema y MINERVA, a este respecto, deja total libertad para establecer la prioridad. Sin embargo, parece lógico que los dispositivos de

entrada con mayor inmersión dentro del mundo virtual tengan más prioridad que los que tienen una menor inmersión. Por ejemplo, es lógico establecer con mayor prioridad los eventos generados por un guante de realidad virtual que por un teclado.

Descritos de esta manera, se puede suponer fácilmente que este tipo de agentes no tienen una traducción en primitivas y transformaciones ya que su cometido es el de comunicar los dispositivos de entrada con los agentes del sistema. Sin embargo, esto no tiene porque ser siempre así. Por ejemplo, se pueden generar primitivas y transformaciones para que el traductor de renderizado dibuje un esquema del dispositivo y mostrar las acciones que el sistema está detectando en el dispositivo de entrada.

Los agentes generadores de eventos pueden también evaluar la calidad de la información recibida de los dispositivos de entrada, enviando o no el evento en función de si la información recibida tiene la suficiente importancia como para tener que enviarla a los agentes del sistema. Si la información es insuficiente, o no tiene importancia para los objetivos del sistema, puede filtrar esta información no enviándola a los demás agentes aumentando la eficacia general del sistema.

Se puede concluir que los agentes generadores de eventos son la pieza clave que nos facilita la reutilización de los demás agentes del sistema porque traducen los eventos de diferentes procedencias, en eventos que el sistema diseñado puede manejar. Además, son importantes para que el sistema se adapte fácilmente a la introducción de nuevos dispositivos de entrada o incluso a la sustitución de un dispositivo por otro. Por otro lado, este tipo de agentes es importante para el rendimiento del sistema. Pueden filtrar información espuria o procesar volúmenes importantes de información en varias iteraciones del modelo.

En definitiva, el diseño de los agentes generadores de eventos requiere una especial atención por el impacto en el rendimiento, por su importancia en la reutilización de los demás agentes del sistema y por su característica de adaptar el sistema a nuevos dispositivos.

Agentes receptores de eventos

Los agentes receptores de eventos son agentes centrados en la percepción del sistema, es decir, que reaccionan ante los eventos que se producen en el sistema. Dependiendo de cómo se reaccione tendremos un tipo de agente u otro. Así, por ejemplo, los agentes que reaccionan a eventos de dispositivos

de entrada son los agentes de usuario, los agentes que reciben eventos evaluando su estado y los eventos recibidos son agentes autónomos, etc.

A la hora de diseñar una función de evolución que recibe eventos se debe cuestionar qué coste tiene el tratamiento del evento. Si el tratamiento de los eventos es muy costoso va a impactar sobre el rendimiento del sistema y se debe prestar especial atención sobre la forma en que se reciben los eventos.

2. Definición de los eventos

Los eventos son los encargados de transportar la información de unos agentes a otros y de desencadenar la ejecución, en las funciones de evolución de los agentes, de aquellas acciones relacionadas con la acción de los eventos.

La información que transporta un evento contiene, por un lado, la acción que representa el evento y que está íntimamente relacionada con la función de evolución y por otro los datos necesarios que definen la acción y que están establecidos en los atributos del evento.

Los tipos de eventos que manejan los agentes deben tener un significado específico que depende de la acción a tomar. No es recomendable establecer tipos de eventos que estén relacionados con los dispositivos de entrada para luego en la función de evolución ejecutar la acción. Dada la importancia de esta cuestión, es necesario insistir en que deben ser tipos de eventos que representen la acción misma. Volviendo a un ejemplo ya tratado, si se desea que un personaje dispare cuando se pulsa la tecla espaciadora no se debe diseñar un evento tal como *eTeclado*$_{<tecla>}$, para luego capturar este evento y ejecutar la acción de disparar. Lo recomendable es definir el evento *eDisparo* para luego ejecutar la acción de disparar en la función de evolución. Esto hace que la separación entre la acción y el dispositivo sea total eliminando cualquier dependencia del origen del evento. En el ejemplo, cuando *eDisparo* es tratado en la función de evolución ya no se sabe si este evento se generó por el teclado o por un guante. Una consecuencia de este tipo de patrón es que si se quiere extender el disparo a un joystick pulsando un botón del mismo, sólo deberá generar el evento *eDisparo* para que el agente actúe de la misma forma. La ventaja de diseñar así el evento es que no se tiene que realizar ningún cambio en la función de evolución, siendo totalmente compatible con el nuevo dispositivo de

Universitat d'Alacant
Universidad de Alicante

entrada.

El diseño de eventos está especialmente relacionado con el diseño de los agentes. Por tanto, la creación de los diferentes eventos estará sujeta a lo que se requiera por parte del agente. Como los eventos describen acciones sobre los agentes, se puede obviar el origen del evento y focalizar el diseño en el significado de la acción, los atributos necesarios si los tiene y cómo estos atributos deben estar optimizados para que el procesamiento de la función de evolución sea lo más rápido posible.

Como se puede observar, los eventos son el mecanismo que tiene MINERVA de ejecutar las acciones dentro del sistema, es decir, es el mecanismo que posibilita que el sistema evolucione con el tiempo. Si diseñamos un mundo sin eventos, se está diseñando un sistema sin actividad y por lo tanto un sistema que sólo describe una escena estática.

3. Definición de primitivas y transformaciones

Como se vio en la descripción de MINERVA, las primitivas y transformaciones son los elementos descriptivos de los diferentes agentes que forman la escena. El traductor de visualización es el encargado de traducir los agentes de la escena en elementos del alfabeto del lenguaje L(V).

Sin embargo, en el modelo no se habla en ningún momento de cuáles son los elementos del alfabeto del lenguaje L(V). Se podrían haber establecido unas primitivas y unas transformaciones lo suficientemente genéricas como para poder abarcar la mayoría de los escenarios que pueden encontrarse al renderizar un mundo virtual. Es lo que se hace en muchos lenguajes descriptivos tales como X3D o POV. Por el contrario, esto no se ha hecho en la definición de MINERVA para no sacrificar la generalización del modelo, en aras de poder adaptarlo a todo tipo de aplicaciones y que cada aplicación establezca los elementos del alfabeto que necesita dependiendo de las circunstancias.

Los elementos que van a formar el alfabeto del lenguaje L(V) están directamente relacionados con los agentes que se han definido en pasos anteriores, pero teniendo en cuenta que estos símbolos pueden ser reutilizados en cualquier momento por otras aplicaciones. También hay que tener en cuenta que las primitivas y las transformaciones serán dibujadas en diferentes dispositivos de salida que no tienen nada que ver

entre ellos. Por ejemplo, no es lo mismo utilizar un dispositivo de salida que permite un renderizado en 3D que otro que sólo dibuja sprites como elementos de dibujo.

Por tanto, a la hora de diseñar las primitivas y las transformaciones se deben tener en cuenta dos importantes características.

La primera característica consiste en que tanto las primitivas como las transformaciones deben tener una cierta independencia de los agentes. Es decir, es muy recomendable que la primitiva o la transformación sólo sea una representación del estado de un determinado agente, pero que por otro lado sea lo suficientemente genérica como para que pueda ser utilizado en otros entornos. Por ejemplo, si se decide establecer *pRobot* como una primitiva que dibuja un robot, está primitiva debe ser diseñada de tal manera que pueda ser utilizada en un videojuego o en un simulador robótico que represente el robot que se está simulando. Si se diseñan así las primitivas y transformaciones, se podrá elaborar un conjunto de símbolos del lenguaje L(V) totalmente reutilizable y expansivo a modo de biblioteca.

La segunda característica estriba en que las primitivas y transformaciones pueden tener diferentes implementaciones dependiendo del dispositivo de salida. La traducción la realiza el traductor de renderizado, y como se estableció en MINERVA si el traductor no puede traducir el símbolo de L(V) entonces simplemente no será visualizado. Sin embargo, esta característica afecta a la forma en que se tienen que diseñar tanto las primitivas como las transformaciones. Por ejemplo, sea $\mathit{tMover}_{<dx,dy,dz>}(\mathit{pNave})$ una transformación que desplaza las primitivas, como *pNave*, en los tres ejes espaciales. La forma en que el traductor va a tratar esta transformación va a ser diferente, si el traductor de renderizado dibuja la escena en un dispositivo 3D o si se dibuja en un dispositivo 2D. Mientras que en primer dispositivo se puede mover la nave de forma natural en el mundo 3D, en el segundo dispositivo, si la escena se dibuja desde arriba, para simular el efecto 3D, el desplazamiento en el eje Y se representaría con un conjunto de sprites de diferentes tamaños que hacen que el objeto se vea más grande cuanto más cerca del observador está.

Las primitivas y transformaciones suelen relacionarse con la parte visual de un mundo virtual. Sin embargo, MINERVA puede traducir las primitivas y las transformaciones a un dispositivo no visual. Un ejemplo claro de esto sería la representación sonora de los elementos del lenguaje L(V). Utilizando el ejemplo anterior, el traductor TR para sonidos, traduciría la cadena $\mathit{tMover}_{<dx,dy,dz>}(\mathit{pNave})$ en sonidos convirtiendo la

Universitat d'Alacant
Universidad de Alicante

primitiva *pNave* en un sonido de motores y la transformación *tMover* realizaría un desplazamiento del foco de este sonido de motores. Así, desplazando varias veces el foco del sonido, se podría generar el efecto Doopler, usando la misma cadena de L(V) que para la representación visual. De esta manera, la representación visual y la sonora se traducen con la misma cadena origen y, por tanto, sonido y representación visual van unidas. Es decir, no nos puede suceder que mientras visualmente vemos a la nave acercarse, desde un punto de vista sonoro parece que se aleja.

4. MINERVA como motor gráfico

En este apartado veremos el isomorfismo existente entre MINERVA y un árbol de escena, estructura muy utilizada en los motores gráficos. También analizaremos cómo construir con MINERVA los elementos propios del motor gráfico y qué patrones de diseño debemos adoptar para que las primitivas y transformaciones de L(V) puedan ser reutilizadas en distintos dispositivos de salida.

4.1. Isomorfismo entre L(V) y el árbol de escena.

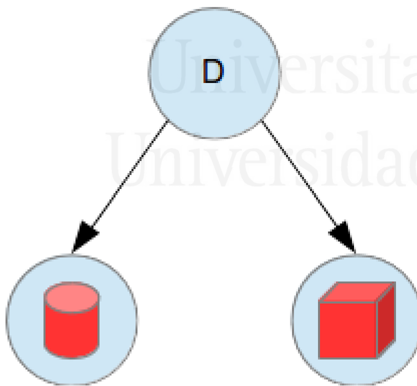


Figura 24: Figura 3: Ejemplo árbol de escena

Tradicionalmente, en los sistemas gráficos, una escena no se representa como una cadena, sino como un árbol denominado árbol de la escena [Strauss and Carey 1992]. En el árbol del ejemplo que se presenta en la Figura 24, los nodos corresponden a elementos de dibujo y las aristas reflejan la dependencia de unos elementos con otros. Así, por ejemplo, si se define una transformación como el desplazamiento (D), que desplaza una esfera y un cubo, en el árbol de escena el nodo padre tanto de la esfera como el del cubo es el nodo de la transformación.

El árbol de la escena define principalmente la relación que existe entre las diferentes partes de la escena. Esta relación puede ser de varios tipos [Bar-Zeev 2003]:

1. Relación de composición: Es una relación definida por un nodo que tiene como hijos partes que definen al padre. Por ejemplo, un personaje puede estar definido por una relación de composición entre su cabeza, su tronco y sus extremidades.
2. Relación lógica: Es una relación que hace que una determinada operación se defina mediante la composición lógica de varios elementos. Así, si se pretende mover un caballero con su caballo se podrá definir una composición lógica para que el movimiento afecte tanto al caballo como al caballero.
3. Relación espacial: Es una relación que estructura la escena por subespacios donde cada subespacio viene representado por un nodo y en el nodo se apunta a los elementos que están en ese subespacio.

Una escena definida como un árbol puede aprovechar las características propias del procesamiento de árboles. Así, por ejemplo, la búsqueda de un elemento del árbol, si la relación es espacial, va a ser una búsqueda dicotómica con un coste temporal mucho más reducido que si la escena se estructura como una simple lista. Por otra parte, el árbol de la escena tiene una importancia relevante en cuanto al coste de renderizado ya que se puede discriminar mediante el árbol, qué partes del dibujo están fuera de la zona de dibujo y, por tanto, si un nodo no se ve, se pueden descartar todos sus hijos en el proceso de dibujado. Por último, se puede considerar el cálculo de colisiones como otro algoritmo de procesamiento en el cual el árbol de escena tiene un efecto relevante sobre el coste. Así, para calcular la colisión de un elemento que tiene una relación de composición con otros elementos, primero calcularíamos si la colisión afecta al volumen total que define la relación de composición, para luego si hay una colisión, entrar en los nodos hijos. Sin embargo, si no existe una colisión en el volumen total, entonces se descartarían los hijos para el cálculo de colisión.

Como se puede comprobar, el árbol de escena tiene un valor en el procesamiento de la escena que no se puede ignorar. Por tanto, se debe establecer la forma de transformar una cadena del lenguaje L(V) en un árbol de escena.

Como se ha descrito anteriormente, el lenguaje L(V) viene definido por una gramática libre de contexto, por lo que es posible saber si una cadena pertenece o no al lenguaje aplicando sucesivamente las reglas gramaticales.

Al conjunto de reglas que se aplican para una cadena dada se le denomina derivación. Esta derivación puede ser expresada mediante la lista de reglas que se aplican para obtener la cadena o mediante una estructura jerárquica sobre la cadena que está siendo derivada. Esta última estructura jerárquica puede ser expresada mediante un árbol denominado árbol sintáctico. Por ejemplo, la cadena $t_1(p_1 \cdot p_2) \cdot p_3$ se puede representar mediante el árbol sintáctico de la Figura 25.

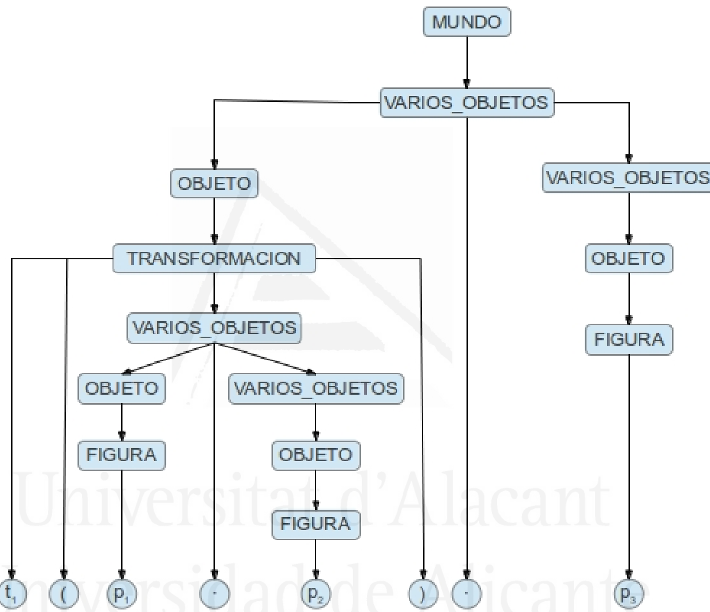


Figura 25: Árbol sintáctico

Pues bien, este árbol sintáctico modela las mismas relaciones que modela un árbol de escena constituyendo ambos tipos de árbol un isomorfismo. Es decir, las relaciones de composición, lógica y espacial son relaciones que están integradas dentro de la gramática, de forma que los paréntesis son los símbolos que permiten acotar el ámbito de las relaciones entre los diferentes símbolos de la cadena que define la escena.

Todo esto significa que las optimizaciones antes tratadas para los árboles de escena puede ser aplicadas a las cadenas generadas por el lenguaje L(V) y se pueden aplicar por tanto los algoritmos de colisión, búsqueda y cualquier algoritmo que use árboles de escena.

4.2. La importancia del traductor TR

La consecuencia principal que tiene utilizar un traductor para el renderizado como el traductor TR, consiste en independizar la ejecución de las diferentes acciones del renderizado para dibujar una escena, de la definición misma de la escena representada por la cadena. Esto significa que una misma cadena de L(V) se puede visualizar en cualquier sistema gráfico solamente cambiado el traductor. Para ser más exactos, la parte del traductor que se debe modificar es el alfabeto del lenguaje de salida L(G).

Sin embargo, el cambio del traductor no sólo afecta a la salida en un sistema gráfico. Cambiando la salida L(G) se consigue que la cadena sea ya no dibujada en un sistema gráfico sino que la cadena pueda estar escrita en un formato de archivo como DWG, DXF, X3D, etc, o transformar la cadena en una estructura de datos necesaria para un cálculo geométrico de las posibles colisiones entre objetos de la escena.

Es decir, el traductor TR proporciona dos características esenciales: la primera consiste en constatar que la estructura de la escena es correcta, característica propia de cualquier traductor, y la segunda, y tal vez más importante, abstraer el proceso de generación de la escena del proceso de obtención de la escena en el sistema geométrico concreto definido por la gramática G.

Las implicaciones que tiene esta separación son varias. Se pueden definir mundos virtuales y estos pueden reutilizarse en diferentes entornos de aplicación. Por ejemplo, si se diseña la descripción de objetos como lámparas, armarios, etc... mediante cadenas, estas cadenas pueden ser utilizadas en diferentes juegos o programas de CAD con la misma descripción. Esta separación hace que podamos trabajar sobre un videojuego y por un lado definir la forma de los diferentes personajes del juego y por otro desarrollar diferentes implementaciones del juego según la consola sobre la cual se esté desarrollando. También se pueden tener varios equipos de desarrollo que trabajen sobre diferentes secuencias de cadenas descriptivas.

Todo esto sugiere que la composición de una cadena es como una descripción de conceptos abstractos donde el objetivo del traductor es concretar la abstracción que representa una cadena en el sistema geométrico G. Con estas consideraciones se puede proponer que el traductor TR es la conexión del mundo de la abstracción con la realidad concreta, como se

sugiere en la Figura 26.

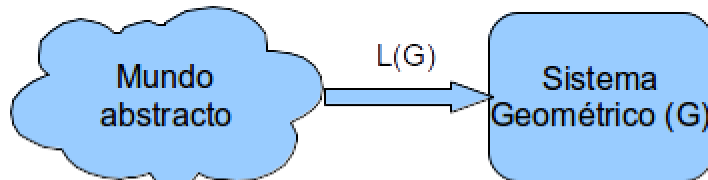


Figura 26: Paso del mundo abstracto al sistema geométrico

En resumen, con el traductor TR se pretende obtener un procedimiento que establece una separación entre la descripción de la escena formulada con un lenguaje común y las particularidades propias del dispositivo gráfico traduciendo el lenguaje común descriptivo en instrucciones propias del dispositivo gráfico adaptadas a las características del dispositivo.

5. MINERVA como motor físico

Los motores físicos son los componentes encargados de simular el efecto de un conjunto de leyes físicas sobre el comportamiento de los elementos de un entorno virtual. Es por tanto necesario mostrar cómo se modela un motor físico en MINERVA para constatar que con el modelo formal propuesto este importante motor puede ser diseñado.

En primer lugar, el motor físico debe comunicar a los elementos los sucesos físicos que acontecen en el mundo virtual. Para ello, debe existir una comunicación entre los elementos del mundo virtual y el motor físico. En MINERVA, este problema puede ser resuelto fácilmente mediante los eventos. Es decir, se puede diseñar un conjunto de eventos que comuniquen a los agentes aquellos sucesos físicos que les afecten.

En segundo lugar, una vez acontecido el suceso físico que comunica el evento, los agentes deben reaccionar ante estos sucesos. En este caso, MINERVA utiliza la función de evolución del agente para definir la reacción del mismo ante dicho suceso asignando una acción al evento del suceso físico.

En tercer lugar, el motor físico necesita actualizar el estado de los elementos. Por ejemplo, dependiendo de los movimientos realizados por cada agente, el motor físico necesita disponer de las coordenadas de los agentes actualizadas para el cálculo de colisiones. Para realizar esta

actualización, se puede utilizar otro evento con el fin de actualizar el estado de dichos agentes y cualquier otro tipo de atributo necesario para la gestión de las acciones físicas.

En último lugar, el motor físico actuará sobre un determinado número de elementos de la escena, no teniendo porqué ser todos. Con el modelo MINERVA se puede utilizar la propia sintaxis de M para definir el ámbito de aplicación del motor físico. Por ejemplo, si *aMFisico* es el agente que modela el motor físico, la cadena $aMFisico(a_1 \cdot a_2 \cdot a_3) \cdot a_4$ define el ámbito de aplicación del agente *aMFisico* mediante los paréntesis del lenguaje L(M). Es decir, los agentes a_1, a_2 y a_3 reaccionarían ante los eventos del motor físico, mientras que el agente a_4 no.

Por tanto, para modelar un motor físico se debe crear un agente *aMFisico*, por ejemplo, con una función de evolución que genera eventos: eventos asociados a los sucesos físicos y el evento que actualiza los valores de los atributos de los elementos. En general esta función de evolución puede quedar representada como sigue:

$$f_{evolucion}(q, aMotor_{\langle atributos \rangle}) = \begin{cases} aMotor_{\langle atributos \rangle} & q.insertar(caracteristicaFisica_1(atributos)) \\ aMotor_{\langle atributos \rangle} & q.insertar(caracteristicaFisica_2(atributos)) \\ aMotor_{\langle atributos \rangle} & q.insertar(caracteristicaFisica_3(atributos)) \\ aMotor_{\langle atributos \rangle} & q.insertar(EUpdate_{\langle atributos \rangle}) \end{cases}$$

En este caso las funciones `caracterisicaFisica`$_i$`(Registro)` generarían los eventos que modelan la característica física *i*, y estos eventos se añadirían al a lista *q* de eventos mediante la función `q.insertar`. Por otro lado, está el evento *eUpdate<registro>* que le dirá a los agentes qué atributos deben actualizar.

Universitat d'Alacant
Universidad de Alicante

En el caso de los agentes que reaccionan a algún tipo de evento proveniente del motor físico, la función de evolución puede tener el siguiente aspecto general:

$$f_{evolucion}(q, aAgente_{\langle attr \rangle}) = \begin{cases} \ldots & \\ w & Si\; eFisica_{1\langle attr \rangle} \in q \\ v & Si\; eFisica_{2\langle attr \rangle} \in q \\ \ldots & \end{cases}$$

donde *w* es la cadena que define la reacción al suceso físico del tipo 1 y *v* es la cadena que define la reacción al suceso físico 2.

Un ejemplo claro de esto se puede ver en el cálculo de colisiones. El motor físico calcularía todas las colisiones de los elementos de la escena y

generaría tantos eventos como colisiones resultantes, poniendo en los atributos del evento de colisión un atributo que identifique los elementos que están colisionando. Por otro lado, en la función de evolución de los agentes se definirá la reacción a la colisión cuando se detecte el evento de colisión.

En el ejemplo descrito, la expresión del motor físico esta centralizada en un solo agente que genera todos los posibles eventos. Sin embargo, MINERVA podría separar este agente en varios agentes especializados en diferentes funciones físicas. Por ejemplo, se podría tener el agente *aColisionar*, *aGravedad*, etc. donde cada agente trataría un tipo de suceso físico. De esta forma, se podrían combinar ciertos sucesos en la definición en la cadena. Así, es posible definir cadenas como $aColisionar(aGravedad(a_1 \cdot a_2 \cdot a_3))$ donde se generarían tanto eventos de gravedad como eventos de colisión, mientras que en la cadena definida como $aColisionar(a_1 \cdot a_2 \cdot a_3)$ sólo se generarían los eventos de colisión. Hay que tener en cuenta que las funciones de evolución de los agentes a_i no se modifican y sólo combinando las cadenas se puede conseguir un mundo con gravedad o un mundo sin gravedad.

Por último, decir que dentro de los eventos generados por los agentes que modelan motores físicos, se deben evitar características unidas a la geometría de la física en aras de tener un mayor grado de reutilización. Por ejemplo, tener en los atributos del evento la posición de la colisión definida mediante un punto 3D hace que dicho evento sólo sirva para sistemas físicos que modelen mundos 3D y nuestros agentes deben saber que dicho mundo están en 3D. Entonces, si queremos cambiar de un mundo 3D a un mundo 2D debemos cambiar tanto el motor físico como las funciones de evolución del agente. Sin embargo, si eliminamos esta dependencia geométrica en el evento de colisión, es fácil comprobar que aunque tengamos que cambiar el motor físico, la función de evolución no es necesario cambiarla ya que no importa si el mundo es 3D o 2D para gestionar dicha colisión.

6. MINERVA como sistema de realidad virtual

El principal objetivo de un sistema de realidad virtual es el de proporcionar al usuario una completa experiencia de inmersión dentro del mundo virtual. Para ello, este tipo de sistemas tiene una gran variedad de

dispositivos tanto de entrada como de salida. Esta diversidad de dispositivos hace que los sistemas de realidad virtual sean complejos de desarrollar. Uno de los objetivos de MINERVA precisamente consiste en homogeneizar esta diversidad de dispositivos tanto de salida como de entrada.

En el caso de los dispositivos de entrada ya se ha comentado que los eventos no deben estar asociados a los dispositivos. Es decir, si dentro del sistema existe un guante de realidad virtual, no se generarán eventos relacionados con el guante sino que el agente encargado de gestionar el guante virtual traducirá los eventos del guante a eventos más abstractos y alejados del dispositivo de entrada tales como coger, avanzar, etc.

Otra característica importante dentro de los sistemas de Realidad Virtual es que en cualquier momento puede salir al mercado un dispositivo con mayor grado de inmersión. La integración de este nuevo dispositivo puede suponer, en algunos casos, el cambio de muchas partes del sistema que utilicen eventos cercanos a los dispositivos. Sin embargo, en el caso de MINERVA, la introducción de un nuevo dispositivo se limita a crear un nuevo agente con su función de evolución generadora de eventos compatible con nuestro sistema e introducirlo en la cadena. Por ejemplo, si se quiere introducir el mando de la Wii en el sistema sólo será necesario introducir un nuevo agente *aWii* en la cadena: $aWii(aGuante(aRaton(a_1 \cdot a_2)))$ que como se indicó anteriormente tendría la mayor prioridad.

Todo esto está referido a los dispositivos de entrada. Sin embargo, existen también fórmulas para tratar la gran diversidad de los dispositivos de salida dentro de los sistemas de Realidad Virtual.

Uno de los principales problemas que tienen los dispositivos de salida es que las capacidades gráficas, las resoluciones y los tamaños de las pantallas, etc. son muy diversos. Por ejemplo, la escena puede ser dibujada en una pantalla de ordenador, en unas gafas de realidad virtual o en una pantalla que ocupe toda una pared. Una de las soluciones que adoptan algunos sistemas es identificar las capacidades mínimas de todos los dispositivos en cuestión y desarrollar el renderizado de la escena con las mínimas capacidades desperdiciando muchas de las capacidades gráficas en ciertos entornos. MINERVA, sin embargo, resuelve todo este tipo de diversidad implementando varios traductores gráficos dependiendo de cada una de las capacidades del sistema. Así, por ejemplo, si el sistema sólo es capaz de dibujar un cierto número de polígonos por frame se puede

Universitat d'Alacant
Universidad de Alicante

dibujar una malla de polígonos con un menor o mayor número de caras dependiendo de la capacidad del sistema gráfico. Incluso el cambio puede ser más radical. Se podría pasar de 3D a 2D utilizando un traductor para el modelado 3D y otro traductor para el modelado en 2D. Lo más interesante que tiene este modelo es que no hay que cambiar nada de la cadena inicial del sistema, ni del traductor del lenguaje L(M) al lenguaje L(V), centrándose las modificaciones en una parte muy concreta del sistema.

En conclusión, se puede observar que MINERVA se adapta muy bien a las necesidades de resolver la gran variedad de dispositivos que entran en juego en un sistema de realidad birtual.

7. MINERVA como motor de inteligencia artificial

Un motor de inteligencia artificial se puede resumir como un sistema que, dado un conjunto de entradas de diversa procedencia, es capaz de tomar una decisión en un momento dado acorde con un plan. Dependiendo de lo complejo que sea el proceso de toma de decisión así será el motor más o menos inteligente.

El modelado del motor de inteligencia artificial con MINERVA se define en las funciones de evolución. Las entradas están representadas por los eventos que es capaz de atender el agente que desarrolla la inteligencia artificial. Dependiendo de la complejidad con la que se ha definido la función de evolución así será el nivel de inteligencia del agente. En anteriores apartados de este capítulo ya se habló de los diferentes tipos de agentes según su función de evolución y se utilizaba un vocabulario que estaba relacionado con el motor de inteligencia artificial.

Así, se puede implementar un modelo reactivo que realiza una simple acción como respuesta inmediata de un evento. Este evento debe ser generado por un agente que modele sensores, siendo de esta manera la forma de especificar en MINERVA la gestión de un sensor por un agente. Es decir, por un lado existirá el agente que genera el evento del sensor y por otro lado el agente que desarrolla la parte de inteligencia.

En el caso de los agentes deliberativos se pueden implementar los demás modelos de inteligencia artificial. Así si la función de evolución utiliza un modelo de fusión para su toma de decisión utilizando modelos bayesianos se tendría un agente con inteligencia probabilística, si se utiliza un proceso cognitivo orientado a una planificación se tendría un agente cognitivo y

Universitat d'Alacant
Universidad de Alicante

así para cada uno de las técnicas de inteligencia artificial.

Por otro lado, hay muchos modelos de inteligencia artificial que utilizan memoria, planificaciones, modelos de aprendizaje, etc. En este caso todos estas estructuras que forman parte del agente pueden ser modelados en MINERVA en los atributos. De esta manera, si se desea un agente que gestione un robot con memoria y con una planificación puede ser definido como $aRobot_{<memoria,\ plan>}$, donde en este caso el atributo *memoria* sería un tipo de dato donde se estructura la memoria y en el atributo *plan* se establecerá la planificación del robot como convenga.

Otro de los principales paradigmas de inteligencia artificial son los sistemas multiagentes que se tratará en el siguiente capítulo de forma particular debido a su importancia, complejidad y relación directa con el modelo presentado.

7.1. MINERVA como sistema multiagente

Un sistema mulitiagente es un sistema en el que concurren varios elementos denominados agentes. Existen muchas definiciones para establecer qué es un agente pero de todas ellas se puede extraer que es un componente autónomo, puede modificar su entorno y tiene capacidad social. Un agente es autónomo si tiene un grado de control sobre sus acciones, sus tareas y toma decisiones bajo ciertas circunstancias. Además un agente puede modificar el entorno que le rodea realizando acciones para tal caso. Por último, cuando se habla de que un agente es social quiere decirse que colaborará con otros agentes si para completar sus objetivos es necesario.

Que en MINERVA se les llame agentes a los elementos de actividad del modelo no es por casualidad. En definitiva, un agente de MINERVA es análogo a un agente de un sistema multiagente.

Un agente de MINERVA es autónomo ya que tiene control sobre sus tareas y sus acciones y sobre cualquier elemento que esté definido en sus atributos tomando sus decisiones mediante la función de evolución teniendo en cuenta su estado.

Además cualquier agente de MINERVA es capaz de modificar el entorno añadiendo nuevos elementos a la escena o modificando parte de ella. Por ejemplo, para añadir nuevos elementos en la escena sólo se tiene que especificar el nuevo elemento del alfabeto mediante la función de evolución como por ejemplo:

Universitat d'Alacant
Universidad de Alicante

$$f_{evolucion}(q, aAgente_{\langle atr \rangle}) = \begin{Bmatrix} \ldots \\ w \cdot aAgente_{\langle atr \rangle} \quad Si\ eEvento_{\langle atr \rangle} \in q \\ \ldots \end{Bmatrix}$$

donde *w* podría ser cualquier cadena que define un nuevo conjunto de elementos.

En el caso de alterar elementos de las escena, como por ejemplo coger los objetos de un videojuego, la función de evolución tiene una expresión algo más compleja. Por ejemplo, imagínese que se desea coger una moneda de oro por parte de un personaje. Sean *aPersona* y *aMoneda* los dos agentes que representan cada uno de los elementos. Para la acción de coger se necesitaría un evento *eCoger* y además se necesitaría del agente *aEntorno* que tendría la parte del motor físico donde se registra la posición de todos los elementos. Entonces, las funciones de evolución de cada uno de los elementos tendrían la siguiente expresión:

$$f_{evolucion}(q, aPersona_{\langle id \rangle}) = \begin{Bmatrix} \ldots \\ aPersona \quad Si\ eCoger_{\langle moneda \rangle} \in q \rightarrow RegistraCogido(Registro, id, moneda) \\ \ldots \end{Bmatrix}$$

Con esta función lo que se hace es si existe el evento *eCoger* entonces se registra que se desea coger una moneda. La función `RegistraCogido(Registro, id, moneda)`, mirará si hay una moneda cerca de la persona identificada mediante id. Si existe una moneda entonces registrará que se ha cogido y si no, no hará nada. En el caso de que se haya cogido, la función de evolución de *aEntorno* generará un evento *eCoger* con el *id* de la moneda cogida. En la función de evolución de la moneda, cuando exista este evento, desaparecerá de la escena. Las funciones de evolución quedan expresadas de la siguiente manera:

$$f_{evolucion}(q, aEntorno) = \begin{Bmatrix} \ldots \\ aEntorno \quad q.insertar(GeneraEventosElemCogidos(Registro)) \\ \ldots \end{Bmatrix}$$

$$f_{evolucion}(q, aMoneda_{\langle id \rangle}^{ECoger}) = \begin{Bmatrix} \ldots \\ \varepsilon \quad Si\ eCoger_{\langle idMoneda \rangle} \in q \wedge eCoger.idMoneda = aMoneda.id \\ \ldots \end{Bmatrix}$$

La cadena que define la escena en un momento dado sería *aEntorno(aPersona*$_{<id>}$ · *aMoneda*$_{<idMoneda>}$*)* y después de coger la moneda quedaría como *aEntorno(aPersona*$_{<id>}$*).*

Por último, hay que demostrar que los agentes de MINERVA son sociales, es decir pueden comunicarse con otros agentes. Esto es evidente gracias al uso de los eventos. Sólo hay un pequeño detalle a resolver. Una de las características de la comunicación entre agentes es que ésta debe ser también entre agentes del mismo nivel. Sin embargo, hasta ahora en MINERVA podemos enviar mensajes desde un agente a sus subagentes, es decir, en la cadena $a_1(a_2 \cdot a_3)$ sólo el agente a_1 puede mandar eventos a a_2 y a_3, pero el agente a_2 no puede enviar ningún un evento ni al agente a_1 ni al agente a_3. Esta relación de prioridad que tiene la sintaxis de MINERVA puede ser en muchos casos interesante ya que se puede dividir la comunicación por zonas, como por ejemplo con la cadena $zona_1(a_1 \cdot a_2) \cdot zona_2(a_3 \cdot a_4)$. Sin embargo, se debe resolver el primer caso planteado de comunicación entre agentes del mismo nivel. Para solucionar este problema, es necesario definir un nuevo agente *aMensajes* a modo de gestor de mensajes, cuyo cometido es recoger los mensajes que se desea enviar y mandarlos a los agentes receptores, quedando la cadena como $aMensajes(a_1 \cdot a_2 \cdot a_3)$

Se ha comprobado, por tanto, que MINERVA cumple con todas las propiedades esenciales de los sistemas multiagentes, pero añadiendo todas las características necesarias para definir completamente sistemas de realidad virtual. Es decir, de forma natural, y utilizando el mismo modelo se pueden describir los elementos de realidad virtual (dispositivos de entrada, dispositivos de salida, motor físico, etc.) como parte de un sistema multiagente.

Universitat d'Alacant
Universidad de Alicante

8. MINERVA como simulador

Otro de los sistemas importantes que se puede modelar con MINERVA es un simulador. Estos sistemas lo que pretenden es probar ciertos algoritmos con diferentes características en entornos virtuales controlados para validar que son correctos y para después implementarlos con mayor seguridad en entornos reales, con los mínimos cambios posibles.

La forma de modelar un simulador con MINERVA es la siguiente. En primer lugar, se modelaría todo el sistema virtual simulado con el modelo formal de MINERVA. La diferencia entre un simulador y un sistema real está principalmente en la salida: mientras que en el primero las acciones se realizan sobre un mundo virtual, en el segundo las acciones se llevan a

cabo en el mundo real. Sin embargo la toma de decisiones se seguirá realizando en la parte de evolución del sistema modelado con MINERVA, es decir, la función de evolución. Por tanto, si se realiza un traductor cuyas acciones se traduzcan en un mundo virtual tendremos el simulador, pero si el traductor transforma la salida en acciones sobre el entorno real se tendrá el sistema real, solamente cambiando el traductor. Sin cambiar nada de la cadena que representa el sistema, se trasladaría el sistema de lo simulado a lo real, minimizando el impacto de este cambio.

Otra de las características que tienen las simulaciones, sobre todo simulaciones sociales, es la de plantear ciertas hipótesis de entorno y ciertos comportamientos heterogéneos y observar cómo se comporta el sistema simulado. Con MINERVA las hipótesis de entorno podrían ser modeladas mediante un agente que implementara estas hipótesis, mientras que los comportamientos heterogéneos estarían modelados en las diversas funciones de evolución de cada uno de los agentes con comportamientos. Por ejemplo, imagínese el problema de presa/depredador. En este caso las hipótesis de entorno podría ser diferentes escenarios donde se desarrolle la escena, como por ejemplo, escena con muchos árboles *aMuchosArboles* o escena con pocos árboles *aPocosArboles*. Por otro lado, se implementarían varios agentes con diferentes funciones de evolución para los depredadores dependiendo de su agresividad *aAgresivo*, *aPocoAgresivo* y otros tantos agentes para las presas *aPresaRapida*, *aPresaLenta*. Entonces, se podrían simular varios casos dependiendo de la combinación de los elementos definidos.

aMuchosArboles(aAgresivo·aPresaRapida·aPresaRapida·aPocoAgresivo)
aPocosArboles(aAgresivo·aPresaRapida·aPresaRapida·aPocoAgresivo)

Con estas dos cadenas se están haciendo dos simulaciones con los mismos elementos pero con hipótesis de entornos diferentes.

En definitiva, con el lenguaje L(M) de MINERVA se pueden establecer tantas combinaciones simuladas como permita el lenguaje definido y podrán ser reproducidas en diferentes circunstancias dependiendo de la forma de la cadena.

9. Relación entre los motores

Hasta ahora se ha visto como se modelan con MINERVA el motor gráfico, el motor físico y el motor de inteligencia artificial. Gracias al modelo

formal se puede observar cómo existe una relación estrecha entre estos tres motores.

Ya hemos visto que el motor gráfico es el encargado de definir la descripción geométrica de una escena virtual. Esta misma descripción geométrica es la que se suele emplear para poder simular algunos efectos físicos. Por ejemplo, para el cálculo de colisiones es necesario saber la geometría de los elementos para comprobar si existen conflictos entre los elementos de la escena. Esta relación entre motor gráfico y motor físico está claramente definida con MINERVA mediante el traductor TV que nos da la descripción geométrica del mundo virtual y cuyo resultado puede utilizarse en los agentes que modelan los motores físicos.

Un motor de inteligencia artificial debe solucionar los problemas de control sobre los agentes, establecer las estrategias en función de sus objetivos y adaptarse correctamente a los cambios dinámicos que se producen. Para seleccionar la estrategia adecuada, el agente en cuestión debe evaluar su entorno. Este entorno es la descripción geométrica que se utiliza para su visualización, además estas descripciones geométricas estás sometidas a simulaciones que ejecuta el motor físico que también se deben considerar en la evaluación del entorno. Por último, el motor de inteligencia artificial cambia la geometría de su entorno modificando su descripción geométrica y por tanto su visualización. Esto nos lleva a constatar la existencia de conexiones entre el motor de inteligencia artificial y los dos motores anteriores.

Universitat d'Alacant
Universidad de Alicante

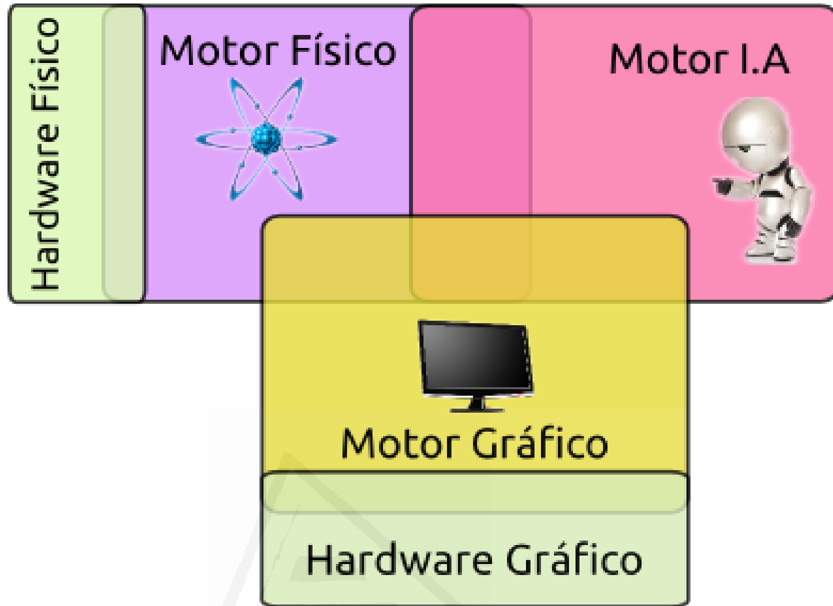


Figura 27: Relación entre los motores

Como representamos en la Figura 27, los tres motores están influenciados mutuamente y esta influencia se manifiesta en el modelo MINERVA a través de la relación entre los traductores. Así el traductor de evolución puede utilizar el traductor de visualización para saber cómo es la geometría de la escena y las cadenas resultantes pueden ser evaluadas por agentes que forman parte del motor de física o del motor de inteligencia artificial.

Universitat d'Alacant
Universidad de Alicante

VI. Experimentación

Una vez definido todo el sistema, se van a realizar los experimentos necesarios para demostrar que MINERVA cumple con los objetivos que en un principio se habían establecido.
Se plantean tres experimentos. El primero es un simulador de incendios forestales, el segundo es un simulador robótico y el tercero un juego de carreras. Tres experimentos que nos mostrarán las potencialidades de MINERVA.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

1. Experimentación

Este capítulo presenta la experimentación realizada. Su objetivo es comprobar las posibilidades de MINERVA al aplicarlo en entornos con diferentes características para así verificar que con un único modelo formal se puede atacar una gran variedad de problemas con múltiples dispositivos de adquisición de datos y diversos sistemas de visualización. Los objetivos de la experimentación son:

- Mostrar cómo MINERVA es capaz de gestionar varios dispositivos de entrada de diferente naturaleza y cómo se gestionan los eventos independientemente de su origen.
- Mostrar cómo MINERVA es capaz de modelar simulaciones basadas en agentes.
- Mostrar cómo MINERVA es capaz de modelar los motores gráfico, físico y de inteligencia artificial de un sistema de realidad virtual como un juego.

Entre los sistemas que se han seleccionado planteamos en primer lugar un simulador de incendios forestales para demostrar que MINERVA puede modelar sistemas de simulación basados en agentes. Es uno de los más sencillos y trata de introducirnos en el diseño de sistemas basados en MINERVA.

El siguiente sistema desarrollado es una aplicación para sistemas robóticos. En este ejemplo, se podrá observar cómo cambiando sólo uno de los traductores se puede trasladar todo el sistema robótico de un entorno de simulación a un entorno real. Además, se podrán apreciar las diferentes formas que tiene MINERVA de tratar con un conjunto de dispositivos de entrada diferentes.

Por último, se desarrollará un videojuego con todos los motores necesarios para tales aplicaciones: motor gráfico, motor físico y motor de inteligencia artificial. Además, se podrán analizar las diferentes relaciones que existen entre los motores del videojuego.

Universitat d'Alacant
Universidad de Alicante

2. Metodología

MINERVA se basa en tres traductores: el traductor de evolución TE, el traductor de visualización TV y el traductor de renderizado TR. Tanto el traductor TE como el traductor TV están definidos dentro del modelo. El algoritmo de traducción es igual para todos los casos. Lo único que cambia en cada caso son las funciones de transición de los traductores, es decir, en el caso del traductor TE cambia la función de evolución, mientras que en el caso del traductor TV cambia la función de visualización. A diferencia de estos traductores, el traductor TR debe ser desarrollado para cada uno de los dispositivos ya que es el traductor que comunica MINERVA con cada uno de los dispositivos de salida. Esto nos obliga a que el traductor TR se implemente para cada uno de los dispositivos a diferencia de los traductores TE y TV.

Aunque el orden en que se definan los elementos es indiferente, en todos los experimentos vamos a plantear una misma secuencia. Empezaremos, en todos los casos, por definir los símbolos del lenguaje L(V). Después, se necesitan los traductores de renderizado que transforman los elementos del lenguaje L(V) en operaciones del dispositivo de salida. Por último, se determinarán los símbolos del lenguaje L(M) que representan los agentes que componen nuestro mundo. Cada agente está constituido por un símbolo, unos eventos ante los que reacciona, unos atributos, una función de evolución que nos indica cómo se comporta cada agente en cada momento y cómo va a evolucionar con el tiempo. Por último, presentaremos la función de visualización que transforma los símbolos del lenguaje L(M) al lenguaje L(V).

3. Simulador de incendios forestales

El primer sistema que hemos desarrollado con MINERVA es un simulador de incendios forestales causados por rayos que caen aleatoriamente y que es descrito por J.Miller y S.Page en [Miller and Page 2009]. El sistema se basa en un agente, el bosque, que es capaz de generar otros agentes con una cierta probabilidad: árboles o rayos. Esto quiere decir que en el bosque pueden crecer nuevos árboles, pero que de vez en cuando puede caer un rayo sobre una zona del bosque. Si un rayo coincide en una zona donde hay un árbol entonces este árbol comienza a quemarse. Después, este fuego

se propaga a los árboles vecinos mientras haya árboles adyacentes que quemar. Vemos un ejemplo en la Figura 28:

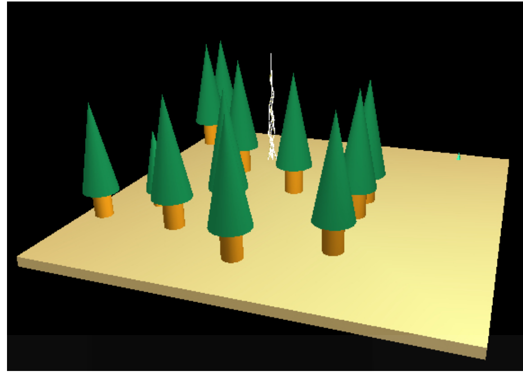


Figura 28: Simulador de incendios forestales

Para modelar este sistema se especificará en primer lugar el alfabeto que compone el lenguaje L(V). Después se implementará un traductor del lenguaje L(V) que dibujará la escena en OpenGL. Una vez definido este traductor se explicará los elementos que forman parte del lenguaje L(M) del sistema.

3.1. Definición del lenguaje L(V)

Existen dos tipos de elementos que forman el lenguaje L(V) del simulador de incendios: las primitivas y las transformaciones. Estos elementos serán los símbolos que luego el traductor de OpenGL dibujará para mostrar la situación del sistema. A continuación, en la tabla 1 se describen cada uno de los símbolos:

Símbolo	Descripción
pBosque	Primitiva que representa el suelo del bosque donde se situarán los árboles y se lanzarán los rayos. Se trata de una rejilla de *NxN* celdas
pArbol	Primitiva que representa un árbol.
pArbolArdiendo	Primitiva que representa un árbol ardiendo.
pRayo	Primitiva que representa un rayo cayendo hacia el suelo.
$tDesplaza_{<f,c>}$	Transformación que sitúa un elemento en la fila *f* y columna *c* de la rejilla *NxN* del bosque.
$tEscala_{<s>}$	Transformación que escala un elemento, siendo *s* el factor de escala.
$tCamara_{<x,y,z,giroX,giroY,giroZ>}$	Transformación del punto de vista de la cámara. La coordenadas *x, y, z* corresponden con la posición de la cámara y los atributos *giroX, giroY, giroZ* los giros en los diferentes ejes de la cámara.

Tabla 1: Definición del lenguaje L(V) para el simulador de incendios forestales

Estos símbolos serán procesados por el único traductor para este sistema que es el traductor que dibuja con OpenGL.

3.1.1. Traductor OpenGL

El traductor de OpenGL es el encargado de traducir las primitivas y las transformaciones del lenguaje L(V) en el conjunto de acciones necesarias para dibujar con la librería gráfica OpenGL.

Primitiva pBosque

Esta primitiva representa el suelo sobre el que se asienta el bosque y se dibujará como una rejilla de *NxN* casillas, cada una de las casillas de un determinado tamaño. Si *TAM_CASILLA* es el tamaño de la casilla, *PBosque* será traducido a una superficie plana de tamaño *TAM_CASILLAxN* de ancho por *TAM_CASILLAxN* de alto usando las funciones de la API de OpenGL para dibujar tal superficie. Al final la primitiva *pBosque* queda renderizada como muestra la Figura 29.

Universitat d'Alacant
Universidad de Alicante

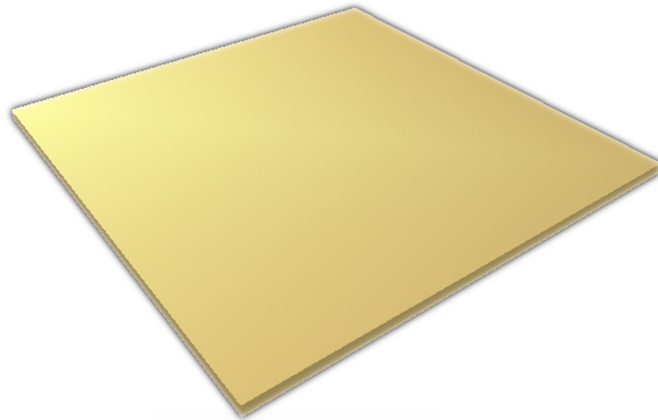


Figura 29: Primitiva pBosque en OpenGL

Primitiva pArbol

Universitat d'Alacant
Universidad de Alicante

Figura 30: Primitiva pArbol en OpenGL

En el caso de la primitiva *pArbol*, este traductor dibujará un cono en la parte superior representando la copa del árbol y un cilindro representando el tronco. Tanto para dibujar el cono, como para dibujar el cilindro, se utilizarán las funciones pertinentes que la API de OpenGL tiene. Podemos ver el resultado en la Figura 30.

Primitiva pArbolArdiendo

Para dibujar el árbol ardiendo, simplemente se dibujará el árbol igual que la anterior primitiva, salvo que en este caso se dibujará la copa del árbol en rojo representando que está ardiendo. Además se añadirán ciertas superficies simulando las llamas del fuego. Para dibujar estas superficies se utilizarán bandas de triángulos usando las funciones de la API de OpenGL.

Primitiva pRayo

Figura 31: Primitiva pRayo en OpenGL

Universitat d'Alacant
Universidad de Alicante

En el caso del rayo (Figura 31), la traducción a OpenGL consistirá en un conjunto de cilindros de tamaños aleatorios y que irán ramificándose poco a poco hasta dibujar un rayo. Para dibujar dichos cilindros se utilizará la primitiva de la API de OpenGL.

Transformación tDesplaza

Para la transformación *tDesplaza* se utilizará la función de la API de OpenGL que desplaza el sistema de coordenadas. Sin embargo, aquí el desplazamiento será *f * TAM_CASILLA* en el eje Z y *c * TAM_CASILLA* en el eje X, mientras que la transformación en el eje Y será siempre 0. De esta manera, transformaremos la fila y la columna de la rejilla que representa el bosque a coordenadas del mundo que representa OpenGL.

Transformación tEscala

La siguiente transformación es *tEscala* que será transformada en la función de escalado de la API de OpenGL. Hay que considerar que *s* estará comprendido entre 0 y 1.

Transformación tCamara

Por último $tCamara_{<x,y,z,giroX,giroY,giroZ>}$ representa la posición de la cámara de la escena. Estos tres símbolos ejecutarán las rutinas de la API de OpenGL que desplazan, giran y posicionan la cámara.

3.2. Definición del lenguaje L(M)

Una vez definida la parte visual del simulador de incendios, se deben precisar los agentes que forman parte del lenguaje L(M) que modelan el simulador y que son los responsables de hacer evolucionar el sistema y los eventos que son generados por los agentes.

3.2.1. Definición de los eventos

A continuación, se definen los eventos que vamos a utilizar en los agentes. Todos los eventos se muestran en la tabla 2:

Evento	Descripción
$eArbol_{<f,c>}$	Este evento es el encargado de crear un árbol en la posición *<f,c>* de la rejilla.
$eRayo_{<f,c>}$	Este evento lanza un rayo sobre la celda *<f,c>* de la rejilla.
$eQuemar_{<f,c>}$	Este evento indica que el árbol pasa al estado quemado.
$eCamDesplaza_{<dx,dy,dz>}$	Evento que desplaza la posición de la cámara.
$eCamGira_{<rotX,rotY,rotZ>}$	Evento que gira la cámara.

Tabla 2: Definición de los eventos para el simulador de incendios

3.2.2. Definición de los agentes

Para definir un agente en MINERVA se necesita definir su función de evolución y su función de visualización. Si un agente no tiene visualización o no tiene evolución entonces su función será vacía.

Universitat d'Alacant
Universidad de Alicante

A continuación, se enumeran todos los agentes que forman parte del sistema de simulación de incendios en la tabla 3. Más tarde, se hará una descripción detallada de cada uno de los agentes especificando los eventos que genera, su función de evolución y su función de visualización.

Símbolo	Descripción
$aGenerador_{<regBosque>}$	Agente que genera los eventos para crear árboles, lanzar rayos y quemar árboles. Como atributo tiene el registro de los árboles quemados y no quemados que hay en el bosque.
aBosque	Agente que define el bosque.
$aRayo_{<t,f,c>}$	Agente que define un rayo. El atributo *t* pertenece al intervalo [0,1] y representa la vida del rayo dentro de la escena. Por otro lado, el atributo *f* representa la fila de la rejilla del bosque y *c* la columna.
$aArbol_{<t,f,c>}$	Agente que define un árbol. El atributo *t* está entre 0 y 1 y representa el grado de crecimiento del árbol. Los atributos *f* y *c* representan la posición del árbol dentro del bosque.
$aArbolQuemado_{<t, f,c>}$	Agente que define un árbol quemándose. El atributo *t* está entre 0 y 1 y representa el grado de quemado del árbol. Los atributos *f* y *c* representan la posición del árbol dentro del bosque.
aKeybCam	Agente que gestiona la posición de la cámara con el teclado.
aCamara	Agente que define la posición de la cámara y cómo se gestiona el cambio de cámara.

Tabla 3: Definición de los agentes para el simulador de incendios

Ahora, se detallarán cada uno de los agentes con la descripción de su función de evolución y su función de visualización. Para una mejor comprensión, al principio de cada agente se realizará una tabla especificando cada uno de los elementos definidos MINERVA.

Universitat d'Alacant
Universidad de Alicante

Agente aGenerador

Símbolo:	$aGenerador_{\langle regBosque \rangle}$
Función de Evolución:	*Sí.*
Eventos Generados:	$eArbol_{<f,c>}$,$eRrayo_{<f,c>}$,$eQuemar_{<f,c>}$
Eventos aceptados:	-
Atributos:	*regBosque.*
Función de Visualización:	*No.*

Este agente es el encargado de generar los eventos que luego van a ser tratados por los agentes. Para generar estos eventos necesitaremos de determinadas funciones de apoyo que utilizarán como parámetro de entrada el atributo *regBosque*.

Este atributo es una matriz de *NxN* números reales en el intervalo [-1,1]. Si una de las celdas del registro vale 0, significa que no hay árbol, si es negativo significa que existe un árbol quemándose, si es mayor que 0 entonces significa que existe un árbol creciendo y si es igual a 1 es que hay un árbol crecido.

Para generar un árbol se va a usar una función `crearArbol(regBosque)`. Esta función devolverá el evento $eArbol_{<f,c>}$ que se generará con una cierta probabilidad siempre que no exista ningún árbol ya en la celda *f,c*. Además, asignará a la celda *(f,c)* de *regBosque* el valor 0.1 indicando que hay un árbol creciendo.

Por otro lado, existe la función `crearRelampago(regBosque)`, muy similar a la anterior. En este caso, devolverá el evento $eRayo_{<f,c>}$ con otra cierta probabilidad y se pasará a valor negativo el valor de la casilla *(f,c)* de *regBosque* indicando que ha caído un árbol.

Por último, definiremos la función `crearQuemar(regBosque)`. En este caso, esta función devolverá una lista de eventos $eQuemar_{<f,c>}$ que se generará si se cumple alguna de las siguientes condiciones:

1. Si el valor de la casilla *(f,c)* de *regBosque* ha cambiado de positivo a negativo.
2. Si en las casillas vecinas de *(f,c)* hay algún valor negativo.

Además de generar los eventos de $eQuemar_{<f,c>}$ esta función actualizará los

valores de la celda *(f,c)*. Si el valor es negativo, lo incrementará hasta 0 que es cuando desaparecerá el árbol quemado; y si es mayor que 0 se incrementará hasta 1 que es cuando el árbol está totalmente crecido.

A continuación se muestra como queda la función de evolución del agente:

$$f_{evolucion}(q, aGenerador_{\langle regBosque \rangle}) = \left\{ aGenerador_{\langle regBosque \rangle} \quad \begin{array}{l} q.insertar(crearArbol(regBosque)) \\ \wedge q.insertar(crearRayo(regBosque)) \\ \wedge q.insertar(crearQuemar(regBosque)) \end{array} \right\}$$

Como se puede observar esta función lo único que hace es insertar en q los eventos que genera.

Agente aBosque

Símbolo:	$aBosque^{eArbol, eRayo}$
Función de Evolución:	*Sí.*
Eventos Generados:	-
Eventos Aceptados:	*eArbol, eRayo*
Atributos:	-
Función de Visualización:	*Sí.*

aBosque es el agente que genera los demás agentes. A partir de éste se van a ir generando los árboles y los rayos aceptando los eventos de creación generados por *aGenerador*.

A continuación, se define la función de evolución del agente *aBosque*.

$$f_{evolucion}(q, aBosque^{eArbol, eRayo}) = \left\{ \begin{array}{ll} aArbol_{\langle 0.1, eArbol.f, eArbol.c \rangle} \cdot aBosque^{eArbol, eRayo} & Si\ eArbol_{\langle f,c \rangle} \in q \\ aRayo_{\langle 0.1, eRayo.f, eRayo.c \rangle} \cdot aBosque^{eArbol, eRayo} & Si\ eRayo_{\langle f,c \rangle} \in q \end{array} \right\}$$

Se puede observar que para generar un nuevo agente, basta con añadir a la cadena resultante, el nuevo agente que se quiere añadir a la escena. Es por lo tanto, un agente generador de nuevos agentes. Por ejemplo, cuando recibimos el evento $eArbol_{<f,c>}$ podemos ver cómo se añade un nuevo agente, *aArbol* añadiendo a la cadena de evolución el nuevo árbol. De igual modo, se procede con los rayos recibiendo el evento *eRayo* y creando el agente *aRayo*.

Una vez definida la función de evolución, se describe la función de

visualización:

$$f_{visualizacion}(q, aBosque_{\langle Registro \rangle}) = \{pBosque \quad Para\ todos\ los\ casos\}$$

Agente aRayo

Símbolo:	$aRayo_{\langle t,f,c \rangle}$
Función de Evolución:	*Sí.*
Eventos Generados:	-
Eventos Aceptados:	-
Atributos:	*t,f,c*
Función de Visualización:	*Sí.*

Este es el agente encargado de procesar y visualizar un rayo dentro de la escena. Los atributos del agente se describen de la siguiente forma: el atributo *t* toma los valores entre 0.1 y 1 y especifica el nivel de evolución del rayo. Por otro lado, los atributos *f* y *c* definen la posición dentro del bosque del rayo.

Explicados los atributos, la función de evolución del agente *aRayo* queda como sigue:

$$f_{evolucion}(q, aRayo_{\langle t,f,c \rangle}) = \begin{cases} aRayo_{\langle t+0.1,f,c \rangle} & Si\ t < 1. \\ \varepsilon & Si\ t = 1. \end{cases}$$

Con este agente se puede observar cómo MINERVA nos permite eliminar agentes de la escena con la función de evolución. Para tal caso, lo único que se debe hacer es devolver la cadena vacía. Por otro lado, se puede ver cómo va evolucionando el agente incrementando el atributo *t* en 0.1.

A continuación definimos la función de visualización:

$$f_{visualizacion}(q, aRayo_{\langle t,f,c \rangle}) = \{tDesplaza_{\langle f,c \rangle}(tEscala_{\langle t \rangle}(pRayo)) \quad Para\ todos\ los\ casos\}$$

Se puede destacar en la función de visualización como los atributos del agente son usados en las transformaciones para modificar el rayo conforme evoluciona en la escena.

Agente aArbol

Símbolo:	$aArbol_{\langle t,f,c\rangle}^{eQuemar}$
Función de Evolución:	*Si.*
Eventos Generados:	*No.*
Eventos Aceptados:	$eQuemar_{<f,c>}$
Atributos:	*t,f,c*
Función de Visualización:	*Sí.*

Este agente acepta el evento *eQuemar* que genera el agente *aGenerador*. Por otro lado, los atributos del árbol tienen el mismo significado que en el rayo, es decir, *t* representa la evolución del árbol y *f* y *c* es la posición del árbol dentro del bosque.

Así pues, la función de evolución del árbol está representada por la siguiente expresión:

$$f_{evolucion}(q, aArbol_{\langle t,f,c\rangle}^{eQuemar}) = \begin{Bmatrix} aArbolQuemado_{\langle t,f,c\rangle} & Si\ eQuemar_{\langle f,c\rangle} \in q \\ & \wedge aArbol.f = eQuemar.f \wedge aArbol.c = eQuemar.c \\ aArbol_{\langle t+0.1,f,c\rangle}^{eQuemar} & Si\ t<1 \\ aArbol_{\langle 1,f,c\rangle}^{eQuemar} & Si\ t=1 \end{Bmatrix}$$

En este caso, en la función de evolución se puede observar como se pasa del agente *aArbol* al agente *aArbolQuemado* cuando pertenece a *q* el evento de quemado con la misma posición que el agente. La evolución de crecimiento del árbol es muy similar a la del rayo, salvo que en este caso cuando $t = 1$ se mantiene en la escena el agente.

Ahora, la función de visualización es muy parecida a la del rayo y queda expresada como:

$$f_{visualizacion}(q, aArbol_{\langle t,f,c\rangle}^{eQuemar}) = \{ tDesplaza_{\langle f,c\rangle}(tEscala_{\langle t\rangle}(pArbol)) \quad Para\ todos\ los\ casos \}$$

Se puede ver que la expresión es prácticamente la del rayo salvo que la primitiva es diferente.

Agente aArbolQuemado

Símbolo:	$aArbolQuemado_{\langle t,f,c\rangle}$
Función de Evolución:	*Sí.*
Eventos Generados:	-
Eventos Aceptados:	-
Atributos:	*t,f,c*
Función de Visualización:	*Sí.*

El árbol quemado representado por este agente tiene los mismos atributos que los agentes *aRayo* y *aArbol* salvo que este, a diferencia de *aArbol* no acepta ningún evento. Así pues, la función de evolución del árbol quemado se expresa como:

$$f_{evolucion}(q, aArbolQuemado_{\langle t,f,c\rangle}) = \begin{cases} aArbolQuemado_{\langle t-0.1,f,c\rangle} & Si\, t>0 \\ \varepsilon & Si\, t=0 \end{cases}$$

En este caso, la función de evolución elimina el agente cuando el atributo t es igual a 0, a diferencia del rayo. La evolución del agente queda representado por la resta $t - 0.1$ que va disminuyendo el tamaño del árbol gracias a la función de visualización que es muy similar a la del árbol:

$$f_{visualizacion}(q, aArbol^{EQuemar}_{\langle t,f,c\rangle}) = \{ tDesplaza_{\langle f,c\rangle}(tEscala_{\langle t\rangle}(pArbolQuemado)) \quad Para\, todos\, los\, casos \}$$

Agente AKeybCam

Símbolo:	$AKeybCam$
Función de Evolución:	*Sí.*
Eventos Generados:	*eCam*
Eventos Aceptados:	-
Atributos:	-
Función de Visualización:	*No.*

Este agente gestiona los eventos de teclado que permiten mover la cámara. No tiene ningún atributo ni función de visualización. En este caso sólo

tiene función de evolución y es la encargada de generar los eventos pertinentes. Su función de evolución es la siguiente:

$$f_{evolucion}(q, aKeybCam) = \left\{ \begin{array}{ll} aKeybCam & Si\,Key = 'home'\, q.insertar(eCamDesplaza_{\langle 0,\, Step,\, 0\rangle}) \\ AKeybCam & Si\,Key = 'end'\, q.insertar(eCamDesplaza_{\langle 0,-Step,\, 0\rangle}) \\ aKeybCam & Si\,Key = '\uparrow'\, q.insertar(eCamDesplaza_{\langle ,\, 0,0,\, Step\rangle}) \\ aKeybCam & Si\,Key = '\downarrow'\, q.insertar(eCamDesplaza_{\langle 0,0,-Step\rangle}) \\ aKeybCam & Si\,Key = '\rightarrow'\, q.insertar(eCamDesplaza_{\langle Step,\, 0,0\rangle}) \\ aKeybCam & Si\,Key = '\leftarrow'\, q.insertar(eCamDesplaza_{\langle -Step,0,0\rangle}) \\ aKeybCam & Si\,Key = 'x'\, q.insertar(eCamGira_{\langle StepAng,0,0\rangle}) \\ aKeybCam & Si\,Key = 'y'\, q.insertar(eCamGira_{\langle 0,StepAng,0\rangle}) \\ aKeybCam & Si\,Key = 'z'\, q.insertar(eCamGira_{\langle 0,0,StepAng\rangle}) \\ aKeybCam & En\,cualquier\,otro\,caso \end{array} \right\}$$

En este caso, las flechas de arriba y abajo del teclado harán que la cámara avance sobre el eje Z, las flechas izquierda y derecha moverán la cámara sobre el eje X y las teclas 'home' y 'end' moverán la cámara en el eje Y. Por otro lado, la tecla 'x' gira en el eje X, la tecla 'y' gira en el eje Y y la tecla 'z' gira en el eje Z.

Agente aCamara

Símbolo:	$aCamara^{eCamDesplaza,\, eCamGira}_{\langle x,\, y,\, z,\, rotX,\, rotY,\, rotZ\rangle}$
Función de Evolución:	*Sí.*
Eventos Generados:	-
Eventos Aceptados:	*eCamDesplaza, eCamGira*
Atributos:	-
Función de Visualización:	*Sí.*

Universitat d'Alacant
Universidad de Alicante

El agente *aCamara* es el encargado de posicionar la cámara de la escena. La posición de la cámara está especificada por los atributos *x, y, z* y los atributos *rotX, rotY y rotZ* definen el ángulo de rotación de la cámara para cada uno de los ejes.

Este agente acepta los eventos *eCamDesplaza* y *eCamGira* que modifican los atributos de la cámara. Estas modificaciones están gestionadas por la función de evolución del agente que queda expresada de

la siguiente manera:

$$f_{evolucion}\left(q, aCam^{eCamDesplaza,eCamGira}_{\langle x,y,z,rotX,rotY,rotZ\rangle}\right)=\left\{\begin{array}{ll} aCam^{eCamDesplaza,eCamGira}_{\langle x+dx,y+dy,z+dz,rotX,rotY,rotZ\rangle} & Si\ eCamDesplaza_{\langle dx,dy,dz,\rangle}\in q \\ aCam^{eCamDesplaza,eCamGira}_{\langle x,y,z,rotX+aX,rotY+aY,rotZ+aZ\rangle} & Si\ eCamGira_{\langle aX,aY,aZ\rangle}\in q \\ aCam^{eCamDesplaza,eCamGira}_{\langle x,y,z,rotX,rotY,rotZ\rangle} & En\ cualquier\ otro\ caso \end{array}\right\}$$

Se puede observar, en la función de evolución, cómo se modifican los atributos dependiendo del tipo de evento de *eCamDesplaza* o *eCamGira*. Cuando llega un evento de la cámara del tipo desplazar, sólo modifica la posición de la misma, mientras que si llega un evento de girar lo que modifica son los atributos de los ángulos de la cámara.

La función de visualización del agente se encarga de traducir el agente en la transformación adecuada del lenguaje L(V) y se expresa de la siguiente manera:

$$f_{visualizacion}\left(q, aCam_{\langle x,y,z,rotX,rotY,rotZ\rangle}\right)=\left\{tCamara_{\langle x,y,z,rotX,rotY,rotZ\rangle}\right\}$$

3.3. Cadena inicial

Por último, queda por definir la cadena inicial que queda representada por:

$$aKeybCam(aCam_{<0,0,0,0,0,0>}(aGenerador_{<regBosque>}(aBosque^{eArbol,eRayo})))$$

Gracias a que *aBosque* puede generar más agentes la cadena del sistema podrá ir evolucionando a cadenas tales como:

$$aKeybCam(aCamara_{<0,020,0,0,0>}(aGenerador_{<regBosque>}(aArbolQuemado_{<0.2,2,2>}\cdot aRayo_{<0.8,2,2>}\ aArbol_{<0.3,1,2>}\cdot aBosque_{<regBosque>})))$$

4. Sistema robótico

El siguiente ejemplo (Figura 32) que se va a tratar consiste en un robot que se desplaza a determinados lugares de un edificio indicados por un operador. Estos lugares están marcados con unas etiquetas y el robot debe buscar dichas etiquetas por todo el edificio hasta encontrar el lugar ordenado por el operador.

El robot está equipado con un sensor láser y una videocámara y a partir de esta información ejecutará acciones dependiendo de lo que perciba en su entorno y el objetivo que ha definido el operador.

Figura 32: Simulador robótico

Este ejemplo busca por un lado modelar un algoritmo de búsqueda de caminos, muy usado en videojuegos, y por otro, poder trasladar el sistema de un entorno simulado a un entorno real solamente cambiando el traductor de visualización del sistema y sustituyendo los sensores simulados por los sensores reales. Además, se verá cómo se puede modelar la combinación de la información adquirida por varios sensores.

Además, se van a reutilizar varios elementos del anterior experimento mostrando así la facilidad de reutilización de los símbolos ya definidos por MINERVA. El caso concreto de reutilización es la transformación *tCamara* del lenguaje L(V) y los agentes *aKeyCam* y *aCam* que gestionan la cámara.

4.1. Definición del lenguaje L(V)

En primer lugar, en la tabla 4 se muestran los símbolos del lenguaje L(V):

Símbolo	Descripción
pRobot	Primitiva que representa el robot.
$tDesplaza_{<dist>}$	Transformación que desplaza el robot una distancia *dist*.
$tGira_{<ang>}$	Transformación que gira el robot un ángulo *ang* en grados.
$tCamara_{<x,y,z,giroX,giroY,giroZ>}$	Transformación del punto de vista de la cámara. Es la misma que se definió para el simulador de incendios forestales.

Tabla 4: Definición del lenguaje L(V) para el simulador robótico

Al final se tienen tres símbolos que deberán traducir los dos traductores siguientes:

1. Traductor de Simulación basado en OpenGL que traducirá las primitivas y las transformaciones a un entorno simulado y dibujado por OpenGL.
2. Traductor Real que traducirá los símbolos del lenguaje L(V) a comandos de ejecución en un robot real.

Ambos traductores se describirán en los apartados posteriores especificando su diseño e implementación.

Universitat d'Alacant
Universidad de Alicante

4.1.1. Traductor OpenGL

Este traductor se encargará de traducir los símbolos del lenguaje L(V) a un sistema que dibujará el estado del robot en un entorno de simulación basado en OpenGL. Para ello, tendrá almacenado en una tupla la posición actual *x* e *y* del robot y el *angulo* de giro en el estado *<x, y, angulo>* del traductor. Esta tupla representa la posición y el ángulo del robot dentro del espacio virtual que vamos a representar en el simulador. Al procesar la cadena el traductor aplicará las operaciones de desplazamiento en la posición *x, y* y aplicará el ángulo *angulo* a la orientación del robot.

Primitiva pRobot

Cuando al traductor le llega la primitiva *pRobot* (Figura 33), aplicará primero el ángulo de giro del robot y luego la posición del robot utilizando la transformación de giro en el eje Z y la transformación de

desplazamiento de la API de OpenGL. Los valores de rotación y traslación se encuentran en el estado *<x,y,angulo>*.

Para dibujar el robot se utilizará un modelo definido mediante un conjunto de mallas de triángulos y se utilizarán las primitivas de bandas de triángulos que tiene la API de OpenGL para dibujar dichas mallas.

Figura 33: Primitiva pRobot en OpenGL

Universitat d'Alacant
Universidad de Alicante

Transformación tDesplaza

Con esta transformación se moverá el robot una determinada distancia especificada por el atributo *dist*. Para ello, se modificará la posición del robot en el estado del traductor considerando el ángulo actual del siguiente modo:

```
x = x + dist · cos(angulo)
y = y + dist · sin(angulo)
```

Transformación tGira

En el caso de la transformación de giro, lo único que se debe hacer es actualizar el atributo *angulo* de la tupla del traductor. Así, la modificación del ángulo queda como:

```
angulo = angulo + ang
```

Transformación tCamara

Es exactamente igual que en el simulador de incendios forestales y se puede reutilizar esta parte de su traductor de renderizado.

4.1.2. Traductor Real

Este traductor ejecutará un conjunto de comandos sobre un robot real. En particular, supongamos que está disponible el comando `girar` que rota un determinado ángulo y el comando `avanzar` que desplaza el robot una determinada distancia. El traductor traduce los símbolos como se especifica en los siguientes apartados.

Primitiva pRobot

Este traductor no hará nada con la primitiva *pRobot*. Realmente lo único que nos interesa de este traductor son las transformaciones que se traducirán en comandos enviados al robot.

Transformación tDesplaza

En este caso la transformación *tDesplaza* ejecutará el comando `avanza` del robot que mueve los motores y le permite desplazarse la distancia definida en el atributo *dist*. Esta transformación terminará cuando termine el comando de avanza.

Transformación tGira

Al igual que la anterior transformación, *tGira* ejecutará un comando del robot real, en este caso el comando `gira`. La cantidad de giro a realizar está definida mediante el atributo *ang*. El comando terminará cuando haya terminado esta transformación.

Transformación tCamara

En este traductor la cámara no tiene efecto y por tanto no hace nada.

Universitat d'Alacant
Universidad de Alicante

4.2. Definición del lenguaje L(M)

Una vez establecido el lenguaje L(V) y sus traductores, se explicará cuáles son los eventos que se van a necesitar y qué agentes son necesarios para hacer evolucionar el sistema.

4.2.1. Definición de eventos

Existen dos tipos de eventos en este sistema robótico. Por un lado, están los eventos relacionados con los sensores y en cuyos atributos está la información obtenida por el sensor. Por otro lado, están los eventos que se utilizarán para sincronizar la ejecución de las acciones del robot y la de toma de decisiones del robot a partir de los datos obtenidos por los sensores. Todos los eventos se muestran en la tabla 5:

Evento	Descripción
$eObstaculo_{<dist,ang>}$	Este evento se emite cuando se detecta un obstáculo. El atributo *dist* nos dice la distancia del obstáculo y *ang* el ángulo donde se sitúa.
$eMarca_{<marca>}$	Este evento informa de que se ha encontrado la marca almacenada en el atributo *marca*.
$eObjetivo_{<objetivo>}$	Este evento indica el nuevo objetivo del robot, almacenado en el atributo *objetivo*.
eEjecuta	Este evento ejecuta la acción que ha decidido el robot.
$eCamDesplaza_{<dx,dy,dz>}$	Evento reutilizado del simulador de incendios forestales.
$eCamGira_{<dx,dy,dz>}$	Evento reutilizado del simulador de incendios forestales.

Tabla 5: Definición de los eventos para el simulador robótico

Se puede comprobar que los eventos están desvinculados de los sensores (*eObstaculo* y *eMarca*). De esta manera, se pueden tener varios sensores que emiten el mismo evento. Por ejemplo, se pueden tener además de un láser, un radar que también nos indica el obstáculo. Esto permite al robot abstraerse del origen de la información y simplemente efectúa las operaciones necesarias dependiendo de si hay o no hay un obstáculo.

El evento *eObjetivo* es un evento emitido por el operador e indica el nuevo objetivo que se debe asignar al robot.

Universitat d'Alacant
Universidad de Alicante

eCamDesplaza y *eCamGira* son los mismos eventos utilizados en el ejemplo del simulador de incendios forestales y servirá para mover el punto de vista de la cámara en el caso de la simulación virtual del sistema robótico.

Por último, está el evento que tiene que ver con la inteligencia artificial: *eEjecuta*. Con este evento el robot evaluará su situación actual y su objetivo y decidirá que acción tomar. Esta acción consistirá en girar o desplazar.

4.2.2. Definición de los agentes

Con los eventos definidos ya se pueden especificar los agentes que formarán parte del sistema que se muestran en la tabla 6:

Símbolo	Descripción
aLaser	Agente que define el sensor de láser.
aVideo	Agente que define el sensor de vídeo.
aOperador	Agente que representa el operador.
aAccion	Agente que define la ejecución de la decisión tomada.
aRobot$_{<memoria, x, y, o, acc, decis>}$	Agente que define el robot.
aKeybCam	Agente reutilizado del simulador de agentes forestales.
aCamara	Agente reutilizado del simulador de agentes forestales.

Tabla 6: Definición de los agentes para el simulador robótico

Universitat d'Alacant
Universidad de Alicante

Los agentes *aLaser* y *aVideo* tienen que ver con los sensores del robot. Destacar que si se desea introducir un nuevo sensor sólo se tiene que definir un nuevo agente. El agente *aAccion* realiza la toma de decisiones y la ejecución de las acciones. *aRobot* define el agente que representa al robot.

Por otro lado, está el agente *aOperador* que será el que representa las acciones que va a tomar el operador con respecto al robot. Será quién nos diga qué objetivo debe buscar el robot.

Los agentes *aKeybCam* y *aCamara* son reutilizados del simulador de incendios forestales y su función de evolución y de visualización son iguales, por tanto no se volverán a explicar. De esta manera, comprobamos el potencial de reutilización que tiene MINERVA.

A continuación, se van a definir las funciones de evolución y visualización, así como una descripción más detallada del diseño de los

agentes.

Agente aLaser

Símbolo:	$aLaser$
Función de Evolución:	*Sí.*
Eventos Generados:	$eObstaculo_{<dist,ang>}$
Eventos Aceptados:	-
Atributos:	-
Función de Visualización:	*No.*

El agente *aLaser* es el encargado de gestionar el sensor láser del robot. En este agente se necesita de una función `buscaObstaculos` que devuelve una lista de eventos *eObstaculo* que representan los obstáculos que ha detectado el sensor. La función `buscaObstaculos` es la encargada de realizar el procesamiento de la señal de láser y detectar los obstáculos, mediante los algoritmos de IA correspondientes.

La función de evolución de este agente se expresa de la siguiente manera:

$$f_{evolucion}(q, aLaser) = \{aLaser \quad q.insertar(buscaObstaculos())\}$$

Universitat d'Alacant
Universidad de Alicante

Este agente no es visible para el traductor del lenguaje L(V) y por tanto no tiene función de visualización.

Agente aVideo

Símbolo:	$aVideo$
Función de Evolución:	*Sí.*
Eventos Generados:	$eMarca_{<marca>}$
Eventos Aceptados:	-
Atributos:	-
Función de Visualización:	*No.*

El agente *aVideo* es prácticamente igual al de *aLaser* ya que, al fin y al cabo,

es otro sensor del robot. En este caso, se tiene la función `buscarMarcas` que generará los eventos $eMarca_{<marca>}$ cuando detecte en la imagen del vídeo una marca determinada. En nuestro caso, hemos utilizado un algoritmo sencillo que detecta distintos colores de forma que cada color se identifica con una marca. No obstante, la función `buscarMarcas` podrá implementar los algoritmos de visión para detección de marcadores que el diseñador del sistema considere convenientes.

Así pues, la función de evolución queda muy parecida a la de *ALaser:*

$$f_{evolucion}(q, aLaser)=\{aLaser \quad q.insertar(buscaObstaculos())\}$$

Al igual que el anterior, este agente no tiene función de visualización.

Agente aOperador

Símbolo:	*aOperador*
Función de Evolución:	*Sí.*
Eventos Generados:	$eObjetivo_{<objetivo>}$
Eventos Aceptados:	-
Atributos:	-
Función de Visualización:	*No.*

Universitat d'Alacant
Universidad de Alicante

El agente *aOperador* es el responsable de comunicarle al robot el objetivo asignado por el operador, es decir, la marca que el robot debe buscar a continuación. Para ello, en nuestro caso hemos utilizado el teclado del ordenador del operador y los objetivos están asignados a las teclas '1', '2', '3' y '4'.

Así pues, la función de evolución queda expresada de la siguiente manera*:*

$$f_{evolucion}(q, aOperador)=\left\{\begin{array}{ll} aOperador & Si\,teclaPulsada()='1'\,q.insertar(eObjetivo_{\langle 1\rangle}) \\ aOperador & Si\,teclaPulsada()='2'\,q.insertar(eObjetivo_{\langle 2\rangle}) \\ aOperador & Si\,teclaPulsada()='3'\,q.insertar(eObjetivo_{\langle 3\rangle}) \\ aOperador & Si\,teclaPulsada()='4'\,q.insertar(eObjetivo_{\langle 4\rangle}) \\ aOperador & En\,cualquier\,otro\,caso \end{array}\right\}$$

Al igual que el anterior, este agente no tiene función de visualización.

Agente aAccion

Símbolo:	$aAccion$
Función de Evolución:	*Sí.*
Eventos Generados:	*eAccion*
Eventos Aceptados:	-
Atributos:	-
Función de Visualización:	*No.*

El agente *aAccion* es un agente generador de eventos del tipo *eAccion* y que sirve para mandar al agente del robot que ejecute la decisión tomada por el robot. Su función de evolución se define de la siguiente forma:

$$f_{evolucion}(q, aAccion) = \{aAccion \quad q.insertar(eAccion)\}$$

En este caso, al igual que los agentes anteriores la función de visualización no existe.

Agente aRobot

Símbolo:	$aRobot^{eObstaculo, eMarca, eObjetivo, eAccion}_{\langle mem, x, y, o, acc \rangle}$
Función de Evolución:	*Sí.*
Eventos Generados:	-
Eventos Aceptados:	*eObstaculo, eMarca, eObjectivo, eAccion*
Atributos:	*mem,x,y,o,acc*
Función de Visualización:	*No.*

Este agente es el que trata todos los eventos generados por los anteriores agentes y el más complejo de todos. Tiene varios atributos que se describen a continuación.

El atributo *mem* es donde se almacena el entorno explorado por el robot. Se define mediante un rejilla *NxN* siendo N redimensionable conforme el robot va explorando su entorno.

Los atributos *x,y* indican la celda de la rejilla en la que se encuentra el robot. Su posición inicial será la celda (0,0) que se encuentra situada en el

centro de la rejilla. Todas las celdas de la rejilla están inicializadas a 0. Las celdas por las que pasa el robot se incrementarán en 1 y si se encuentra con un obstáculo se asignará a esta celda el valor de infinito.

El atributo *acc* son tuplas del tipo *<dist,ang>* que representan la distancia y el giro que el robot ejecuta.

Por último, tenemos el atributo o que representa el objetivo que debemos detectar para llegar al lugar donde pararemos el robot.

Una vez explicados todos los atributos la función de evolución se puede expresar de la siguiente manera:

$$f_{evolucion}(q, aRobot^{Eventos}_{\langle mem,x,y,o,acc\rangle}) = \begin{cases} aRobot^{eventos}_{\langle mem,x,y,obj,\varepsilon\rangle} & Si\ eObjectivo_{\langle obj\rangle} \in q \\ aRobot^{eventos}_{\langle mem,x,y,\varepsilon,\varepsilon,\varepsilon\rangle} & Si\ eMarca_{\langle marca\rangle} \\ & \wedge\ marca = objetivo \\ aRobot^{eventos}_{\langle marcarObstaculo(mem,x',y'),x,y,o,acc\rangle} & Si\ eObstaculo_{\langle dist,ang\rangle} \in q \\ aRobot^{eventos}_{\langle mem,x'',y'',obj,pathFindingA(mem,x,y)\rangle} & Si\ eAccion \in q \wedge o \neq \varepsilon \\ aRobot^{eventos}_{\langle mem,x,y,o,acc\rangle} & En\ cualquier\ otro\ caso \end{cases}$$

Como podemos observar en la función de evolución, con el evento *eObjetivo* cambiamos el atributo *o* del agente asignando un nuevo objetivo al robot.

Por otro lado, cuando llega el evento *eMarca* y el atributo del evento coincide con el objetivo del robot, entonces la acción pasa a vacío representado por *ε,* indicando que el robot ha llegado a su destino y quedando a la espera de nuevas órdenes.

También podemos observar que existen dos funciones auxiliares en esta función de evolución: `marcarObstaculo` y `pathFindingA`. Estas funciones nos sirven de apoyo para determinar qué debemos hacer cuando nos llega el evento *eObstaculo* y *eAccion*.

La función `marcarObstáculo(mem,x',y')` es la encargada de marcar en la memoria del robot los obstáculos que el robot, mediante sus sensores va detectando. Es decir, cuando se recibe un evento *eObstaculo*$_{<dist,ang>}$, donde *dist* es la distancia del obstáculo y *ang* el ángulo donde se ha detectado, la función `marcaObstaculo` marcará la celda *(x',y')* con un infinito siendo x' e y' las expresiones:

```
x' = x + eObstaculo.dist · cos(eObstaculo.ang)
y' = y + eObstaculo.dist · sin(eObstaculo.ang)
```

La función `pathFindingA(mem, x, y)` es la función que decide qué

acción ejecutar dependiendo de lo que hay en el atributo *mem* registrado y la posición *x,y* del robot. Este algoritmo puede ser, por ejemplo, el típico algoritmo A* de búsqueda de caminos. Esta función analizará los valores de las celdas vecinas a la posición del robot y seleccionará la celda con menor valor que se denotará como *(xSel,ySel)*. Esta posición se devuelve como una tupla del tipo *<dist,ang>* y servirá para calcular el valor del atributo *acc*, que será:

```
dist = sqrt((x – xSel)^2 + (y – ySel)^2)
 ang = arctg( (y – ySel) / (x - xSel))
```

Además, aumentará en uno la posición actual del robot. Como ya sabemos qué acción vamos a tomar, se modificará la posición del robot según la expresión:

```
x'' = x + acc.dist · cos(acc.ang)
y'' = y + acc.dist · sin(acc.ang)
```

En la figura Figura 34, podemos ver un ejemplo de cómo se va modificando la memoria con el tiempo.

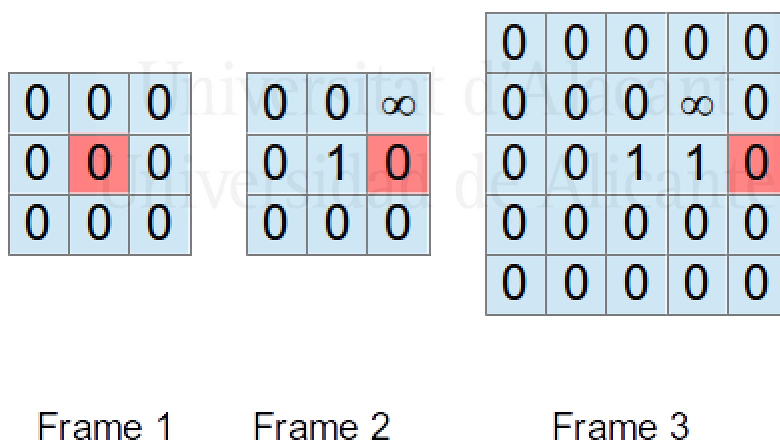


Figura 34: Actualización del atributo mem

Universitat d'Alacant
Universidad de Alicante

En este ejemplo la posición *x,y* está identificada con la celda color salmón. Se puede ver cómo en el primer frame tenemos una rejilla de 3x3 y el robot está situado en el centro y la posición es *x*=0 e *y*= 0. En este frame toda la

rejilla está inicializada a 0. En el segundo frame vemos como se ha marcado a 1 la celda donde estaba anteriormente el robot y se desplaza a la derecha, según le indica el resultado de la función `pathFinding`. Además, desde esta posición se detecta un obstáculo en la celda (1,1) y se marca a infinito con la función `marcaObstaculo`. Por último, en el frame 3 se puede ver como se redimensiona la rejilla cuando el robot rebasa los límites de la rejilla.

Una vez explicada la función de evolución y teniendo en cuenta que el atributo *acc* contiene la acción a ejecutar por el robot, la función de visualización queda como:

$$f_{visulizacion}\left(q, aRobot^{Eventos}_{\langle mem,\, x,\, y,\, o,\, acc,\, deci\rangle}\right)=\left\{\begin{matrix} tGiro_{\langle acc.\ ang\rangle}\left(tDesplaza_{\langle acc.\ dist\rangle}\left(PRobot\right)\right) & Si\, acc\neq\varepsilon \\ \varepsilon & En\ otro\, caso \end{matrix}\right\}$$

Con la función de visualización pasamos al traductor del lenguaje L(V) la acción decidida por el robot. Si no existe acción que ejecutar, la función de visualización es vacía.

4.3. Cadena inicial

Por último queda definir la cadena inicial del sistema que está representa por

$$AKeyCam(ACamara_{<0,020,0,0,0>}(ALaser(AVideo(\\ AOperador(AAccion(ARobot<mem,0,0,\mathcal{E},\mathcal{E},\mathcal{E}>))))))$$

Con esta cadena inicial se puede observar cómo los agentes de sensores están más al exterior y en la parte más interior de la cadena está el robot recibiendo todos los eventos.

Es fácil ver que para añadir nuevos sensores al modelo basta con añadir en la parte más exterior los agentes que definen los sensores para que el robot reciba los eventos. Si un sensor define un nuevo evento deberíamos modificar la función de evolución del robot para que lo tenga en consideración. Sin embargo, si el sensor genera un evento de los ya establecidos, se incorpora en el modelo de forma sencilla.

Por último, en los atributos iniciales del robot se observa que el objetivo no está definido. Esto significa que hasta que el operador no le asigne un objetivo el robot permanecerá en esta posición.

Universitat d'Alacant
Universidad de Alicante

5. Juego de carreras de coches

A continuación, se va a modelar un videojuego de coches inspirado en el famoso juego del Scalextric®. La elección de esta aplicación se ha realizado porque se consideró que era un ejemplo sencillo e ilustrativo de todas las características de MINERVA que se desea exponer.

Figura 35: Juego de carreras de coches

El juego consiste en varios coches de carreras que compiten entre ellos para llegar primero a la meta (Figura 35). En este juego los coches van en distintos raíles y pueden cambiar de raíl cada vez que lo deseen, aumentar la velocidad o frenar. Si un coche va a una velocidad inadecuada en una curva, se saldrá de la pista a causa de la fuerza centrífuga y será penalizado con un tiempo de espera. El ganador será aquel coche que haga lo antes posible el número de vueltas en que consiste la carrera.

Con la construcción de este juego tendremos la oportunidad de desarrollar los tres motores de los que se ha estado hablando durante toda la tesis: motor gráfico, motor físico y motor de inteligencia artificial. Además de los tres motores, se va a realizar la implementación, en MINERVA, de varios dispositivos de entrada para ver cómo incorporar de forma sencilla un conjunto heterogéneo de dispositivos.

Las características que se van a implementar para el motor gráfico son las siguientes:

Universitat d'Alacant
Universidad de Alicante

1. Se definirá el lenguaje L(V) para describir todos los elementos que van a formar parte de la escena.
2. Se implementará el traductor para dibujar toda la escena en 3D usando la librería OpenGL. Para ello, todos los símbolos del lenguaje L(V) serán traducidos a entidades 3D de OpenGL.
3. Se implementará el traductor para dibujar toda la escena en 2D con sprites. Igual que con el anterior traductor, todos los símbolos del lenguaje L(V) serán traducidos a sprites.
4. Se implementará el traductor para traducir los elementos de L(V) en sonidos. Con este traductor se pretende fusionar la salida visual con la del sonido.
5. Se implementará un traductor para guardar todos los estados del videojuego en un fichero de texto con el fin de poder almacenar todo el desarrollo de la partida. De esta forma, después se podrá reproducir la partida almacenada a modo de grabación de vídeo, en 3D o en 2D, sin más que aplicar los traductores de los puntos 2 o 3, respectivamente.

Con estos 4 traductores se quiere mostrar cómo MINERVA puede dibujar una escena en varios dispositivos de salida sin tocar los elementos de la escena. Además, se demuestra que los traductores no sólo se utilizan para dispositivos visuales, sino que también pueden procesar la escena traductores para dispositivos no visuales, como ficheros de texto o sonido.

Ahora, se van a describir las características del motor físico que el sistema debe modelar:

1. Se implementará la fuerza centrífuga en las curvas.
2. Se implementará el algoritmo de detección de colisiones.
3. Se implementará un algoritmo que establezca lo que cada coche puede ver a su alrededor.

Con el motor físico, se pretende mostrar la facilidad de desarrollar características físicas sin necesidad de cambiar ninguna parte más allá del motor físico, de tal manera, que cualquier agente del lenguaje L(M) puede reaccionar a los efectos físicos modelados únicamente reaccionando a los

eventos generador por dicho motor.

Por otro lado, el motor de inteligencia artificial tiene los siguientes requerimientos:

1. Se implementarán pilotos virtuales que puedan tomar sus propias decisiones como aumentar o disminuir la velocidad dependiendo de si se acercan o no a una curva, tapar al piloto que quiere adelantarlo o decidir si aceleran para adelantar al rival, entre otras posibilidades.
2. Se implementará un algoritmo con planificación basado en una máquina de estados para la evaluación de la decisiones.

Por último, para demostrar la capacidad de MINERVA para gestionar varios dispositivos de entrada se establecen los siguientes requerimientos:

1. Se implementará para el teclado.
2. Se implementará para el joystick.

En adelante, se definirán tanto los símbolos del lenguaje L(V) como los del lenguaje L(M). Además, especificaremos los eventos que van a generar los dispositivos de entrada. Por último, se desarrollarán las funciones de evolución que describen todas las características anteriormente enumeradas.

Universitat d'Alacant
Universidad de Alicante

5.1. Definición del lenguaje L(V)

Cómo se ha descrito en capítulos anteriores el lenguaje L(V) es el lenguaje que define los elementos visuales de sistema de realidad virtual. En el caso del juego de carreras se describen en la tabla 7:

Símbolo	Descripción
pCircuito	Primitiva que dibuja el circuito donde se va a desarrollar la carrera.
pPaisaje	Primitiva que dibuja un cielo y un suelo a la escena.
pCocheId	Primitiva que representa diferentes tipos de coches identificados mediante un identificador Id. Podemos poner tantos tipos de coches como se quiera, donde cada Id indica la forma de un coche.
$tTono_{<tono>}$	Transformación que cambia el tono de un sonido. El atributo *tono* identifica el nivel del tono.
$tDes_{<indPista>}$	Transformación que desplazará el coche a la pista *indPista*.
$tRot_{<ang>}$	Transformación que rotará el ángulo *ang* en el eje Z.
$tCamara_{<x,y,z,giroX,giroY,giroZ>}$	Transformación para gestionar el punto de vista de la cámara. Reutilizada del simulador.

Tabla 7: Definición del lenguaje L(V) para el juego de carreras de coches

Estos símbolos serán procesados por los diferentes traductores que transformarán las primitivas en acciones del sistema gráfico adecuado. En esta experimentación se va a trabajar con cuatro traductores: Traductor OpenGL, Traductor Sprite, Traductor de sonidos y Traductor en ficheros.

5.1.1. Traductor OpenGL

Universitat d'Alacant
Universidad de Alicante

Este traductor será el encargado de traducir las primitivas y transformaciones en operaciones de OpenGL. Ahora se enumerarán cada una de las primitivas y cómo se transforman en mallas de triángulos de OpenGL. También se explicará cómo se traducen las transformaciones en rutinas de OpenGL.

Este traductor tiene un atributo *PosPistasGL* dentro de su implementación que contiene las posiciones de cada una de las pistas en el espacio 3D que gestiona OpenGL. Las pistas se identifican con un índice que vale entre 1 y N, siendo N el número máximo de pistas disponibles. La función `getPosicionPista3D(posPistaGL, indPista)` devuelve la posición 3D de la pista *indPista*.

Primitiva pCircuito

El traductor de OpenGL traducirá esta primitiva en un conjunto de pistas

que forman el circuito. Cada pista es una malla de triángulos formados por vértices en 3D, por lo que el circuito es un conjunto de mallas que conforman un cuerpo 3D. Para dibujar estas mallas de triángulos, se usarán las funciones de OpenGL que gestionan buffers de vértices.

Primitiva pPaisaje

Esta primitiva es un cubo en cuyas caras internas se mapeará una textura con una imagen de un cielo. En el caso del terreno, se dibuja la cara inferior con una textura de hierba. Para el mapeado de texturas se utilizan las rutinas de OpenGL habilitadas para este cometido. Podemos ver el resultado de *pPaisaje* en la Figura 36.

Figura 36: Primitiva pPaisaje en OpenGL.

Primitiva pCocheId

En este caso, el traductor transformará esta primitiva en un conjunto de mallas 3D de triángulos que forman el modelo de un coche en 3D (Figura 37). Este modelo se encuentra almacenado en un archivo 3ds. Una vez cargada la malla se utilizarán las rutinas de OpenGL para dibujar los triángulos que forman la malla.

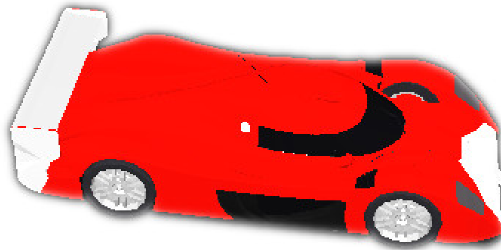


Figura 37: Primitiva pCocheId en OpenGL

Traducción de las transformaciones

Existen tres transformaciones en el lenguaje L(V) definido. La primera es $tDes_{\langle indPista \rangle}$ que utilizando la función `getPosicionPista3D(posPistaGL, indPista)`, aplicará el desplazamiento en 3D para desplazarse a la posición de la pista *indPista*. La segunda es $tRot_{\langle ang \rangle}$ que representa la transformación de rotación en el eje Z de un determinado ángulo. La tercera es $tCamara_{\langle x,y,z,giroX,giroY,giroZ \rangle}$ que representa la posición de la cámara de la escena y que es reutilizada del simulador de incendios forestales.

En el caso de la transformación $tTono_{\langle tono \rangle}$ el traductor de OpenGL no hará nada.

Universitat d'Alacant
Universidad de Alicante

5.1.2. Traductor de sprites

El traductor de sprites transforma nuestra cadena del lenguaje V, definido anteriormente, en un conjunto de imágenes 2D que se irán componiendo hasta representar toda la escena. Este conjunto de imágenes se añadirá en sus posiciones correspondientes dentro de otra imagen donde dibujaremos toda la escena. Por último, sólo tendremos que dibujar la imagen final. La escena se dibujará en 2D y con la cámara posicionada desde arriba.

Este traductor tiene un atributo *posPistasSprite* que es un vector de N posiciones donde N es el número de pistas del circuito. En cada una de las posiciones se almacenan las posiciones de las pistas del circuito dentro del espacio 2D. Para obtener la posición de una pista, se ejecutará la rutina `getPosPistasSprite(posPistasSprite, indPista)` que devolverá la posición 2D de la pista *indPista*.

Primitivas pCircuito

En este caso, el *pCircuito* se traducirá en una imagen que representa todo el circuito donde va a transcurrir la carrera.

Primitivas pCocheId

pCocheId se traducirá a una imagen que identifica cada *pCocheId* con una imagen diferente. Podemos utilizar la misma imagen para representar varios coches con el mismo chasis.

Como el giro distorsiona las imágenes (problema de pixelado), se va a definir un conjunto de imágenes que definirán diferentes ángulos del coche para después, dependiendo del ángulo de la transformación activa, seleccionar la imagen más cercana al ángulo activo.

Esto significa que en el dibujo del coche es necesario saber cuál es el ángulo de rotación de la transformación actual.

Primitivas pPaisaje

En este caso, el paisaje sólo está representado por el suelo donde se ubica el circuito. Este suelo está representado por una imagen única.

Traducción de las transformaciones

En el caso de las transformaciones, se construye una pila donde acumularemos las diferentes transformaciones. Para la aplicación de las transformaciones se procederá de forma similar a con el traductor de OpenGL. Es decir, los dos símbolos '(' y ')' limitarán la aplicación de las transformaciones. Estos dos símbolos indicarán cuando debemos apilar y desapilar, respectivamente, la matriz actual de transformación.

En este traductor se aplicarán las transformaciones $tDes_{<indPista>}$ y $tRot_{<ang>}$ a la matriz de transformación. En la transformación $tDes_{<indPista>}$ se utilizará la función `getPosPistasSprite(posPistasSprite, indPista)` para desplazarse a la pista *indPista*. Con la matriz de transformación se podrá obtener la posición donde se deben dibujar las primitivas y el giro que permitirá seleccionar la imagen más adecuada para ese ángulo. En el caso de la transformación $tCamara_{<x,y,z,giroX,giroY,giroZ>}$ y de $tTono_{<tono>}$, este traductor no hará nada.

5.1.3. Traductor de sonido

Este traductor, transformará en sonido aquellos símbolos que tengan sonido. En particular, el único símbolo que tiene sonido es la primitiva del *pCocheId*. Esta primitiva, modificará un sonido del motor dependiendo de la transformación $tTono_{<tono>}$. Esto hará que el coche tenga un sonido más grave o más agudo.

5.1.4. Traductor de fichero

El traductor de fichero realizará una traducción de la cadena del lenguaje L(V) en cada frame a una cadena de caracteres que representa los símbolos de lenguaje L(V). En la tabla 8 se muestra la traducción de cada símbolo del lenguaje L(V):

Símbolo	Traducción
pCircuito	*pCircuito*
pCocheId	*pCocheId*
pPaisaje	*pPaisaje*
$tTono_{<tono>}$	*tTono<tono>*
$tDes_{<dx,dy,dz>}$	*tDes<dx,dy,dz>*
$tRot_{<ang>}$	*tRot<ang>*
$tCamara_{<x,y,z,giroX,giroY,giroZ>}$	*tCamara<x,y,z,giroX,giroY,giroZ>*
(	(
)	)

Tabla 8: Traducción de los símbolo de L(V) en cadena de caracteres

Universitat d'Alacant
Universidad de Alicante

Una vez traducida en una cadena de caracteres, lo escribiremos en el fichero de texto de salida.

5.2. Definición del lenguaje L(M)

Una vez establecido el lenguaje L(V) y sus traductores, se explicará cuáles son los eventos que se van a necesitar y qué símbolos del lenguaje L(M) son necesarios para hacer evolucionar el sistema.

5.2.1. Definición de los eventos

En este sistema existen varios eventos que tienen distintos cometidos. Como ya se dijo en anteriores apartados, los eventos en Minerva son eventos que no están relacionados con los dispositivos de entrada, sino que son eventos con significado de alto nivel. Los eventos definidos para este sistema se describen en la tabla 9:

Evento	Descripción
$eActualiza_{<regPosCoches,pistas>}$	Evento que se emite cada cierto tiempo. Es un evento para actualizar los datos de la posiciones de los coches.
$eAcel_{<idCoche,incrVel>}$	Evento que informa de la aceleración que se debe aplicar a un coche.
$eCambiaRail_{<idCoche>}$	Este evento se genera cuando se quiere cambiar de raíl en la pista.
$eCamDesplaza_{<dx,dy,dz>}$	Evento reutilizado del simulador de incendios forestales.
$eCamGira_{<dx,dy,dz>}$	Evento reutilizado del simulador de incendios forestales.
$eDespFuerza_{<idCoche,dx,dy>}$	Evento que define el desplazamiento producido por la fuerza centrífuga o a una colisión.
$eCurva_{<idCoche, distancia>}$	Evento que informa de si existe una curva a cierta distancia del coche.
$eVision_{<idCoche,vista,distancia,rail>}$	Evento que informa de la existencia de otro coche en el campo de visión del piloto.

Tabla 9: Definición de los eventos para el juego de carreras de coches

Universitat d'Alacant
Universidad de Alicante

Para cada uno de los eventos existen atributos que dan más información acerca del evento. Con estos atributos cada agente realizará la acción pertinente según su función de evolución.

En primer lugar está el evento $eActualiza_{<regPosCoches,pistas>}$. Este evento se genera a cada frame del traductor de evolución y sirve para actualizar las posiciones de los coches en una base de datos donde están almacenados estas posiciones. Las posiciones están referidas a las pistas del circuito. Estas pistas están numeradas de 0 a N-1 (N número total de pistas del circuito) y cada una de ellas tienen una forma (recta, giro izquierda, giro derecha, etc) que nos define una curvatura.

En este evento está el atributo *pistas*. Este atributo tiene las posiciones en el espacio 3D de las pistas. Con este atributo podemos pasar las

posiciones antes explicadas a una posición en el espacio 3D que será necesaria para el dibujado de los coches y el cálculo de colisiones.

Por otro lado, el atributo *regPosCoches* es un registro formado por tuplas con los siguientes datos *<idCoche, vel, pos, rail, dim, ang>*. En esta tupla, *idCoche* identifica el coche. Con el atributo *vel* fijamos la velocidad actual del coche. Además, *pos* define la posición del coche en el circuito. Esta posición es un número real con un decimal, donde la parte entera identifica el índice de la pista del circuito y la parte decimal señala donde está el coche dentro de la pista. Por ejemplo, una posición de 6.5 nos indicará que estamos a la mitad de la pista 6. La posición del coche queda totalmente definida con el atributo raíl que puede tener el valor izquierda o derecha identificando cada uno de los raíles. Por otro lado, el atributo *dim* identifica la caja 3D que envuelve el coche. Este atributo es utilizado para calcular las intersecciones entre las cajas envolventes para saber si dos coches están colisionando. Hay que tener en cuenta que a esta caja debemos aplicarle la posición y el ángulo para calcular correctamente la colisión.

Con el evento $eAcel_{<idCoche,incrVel>}$ se define la aceleración, donde el atributo *incrVel* define el incremento de velocidad que se va aplicar al coche con identificador *idCoche*. Si este valor es negativo estamos aplicando una desaceleración.

$eCambiaRail_{<idCoche>}$ define el evento que ejecuta el cambio de raíl del coche *idCoche*. Si este coche está en el raíl de la izquierda, cambiará a la de la derecha y si está en la derecha, cambiará al raíl de la izquierda.

Los eventos *eCamDesplaza y eCamGira* son eventos reutilizados del simulador de incendios forestales.

$eDespFuerza_{<idCoche,dx,dy>}$ representa el desplazamiento originado por una fuerza y se generará cuando se produzca una colisión en el coche *idCoche.* Este coche se va a desplazar *dx* en la dirección longitudinal del coche y *dy* en la dirección transversal del coche. También se genera este evento cuando un coche entra en una curva y está bajo el influjo de la fuerza centrífuga entonces se generará un evento que representará el desplazamiento generado por dicha fuerza.

Para ayudar en la toma de decisiones de cada uno de los agentes de inteligencia artificial, se definen dos eventos más. $eCurva_{<idCoche,distancia>}$ es el evento que se genera cuando el coche puede ver una curva a cierta distancia. En este evento *idCoche* identifica el coche que tiene a cierta distancia la curva y la *distancia* a la curva. Por último, está el evento

Universitat d'Alacant
Universidad de Alicante

$eVision_{<idCoche,vista,distancia,rail>}$ que identifica la existencia de otro coche. En este evento se tiene el atributo *idCoche* que identifica el coche al que se refiere el evento de visión, *vista* que puede ser uno de los siguientes valores *{atrás, delante, aLado}* identificando la posición del coche contrario, *distancia* que define la distancia a la que está el otro coche y *rail* que identifica en que raíl puede estar *{izquierda, derecha}.*

En estos dos últimos eventos el atributo distancia no hace referencia a la distancia exacta sino que es una distancia aproximada y puede tener uno de los siguientes valores *{lejos, cerca, muyCerca}*. Aunque se podría calcular la distancia exacta entre los diferentes elementos, no se hace así para hacerlo más real, ya que en la realidad no se tiene una distancia exacta sino más bien una distancia estimada.

Una vez determinados todos los eventos se pueden definir los agentes que van a formar parte del lenguaje M.

5.2.2. Definición de los agentes

A continuación, se hace una primera descripción resumida de cada uno de los agentes (tabla 10) que componen el lenguaje L(M) para después pasar a una más amplia descripción donde se mostrarán las funciones de evolución, sus atributos y los eventos a los que reacciona.

Universitat d'Alacant
Universidad de Alicante

Símbolo	Descripción
aPaisaje	Agente que define el paisaje donde está la pista de carreras. Gestiona el cielo y el terreno.
aCircuito	Agente que define el circuito.
aKeybUser1	Agente que gestiona el teclado del usuario 1.
aKeybUser2	Agente que gestiona el teclado del usuario 2.
aKeybCam	Reutilizado del simulador de incendios forestales.
aJoystUser1	Agente que gestiona el joystick del usuario 1.
aJoystUser2	Agente que gestiona el joystick del usuario 2.
aCamara	Reutilizado del simulador de incendios forestales.
aFisica	Agente que define el entorno y donde se implementará el motor físico.
aIAPiloto	Agente que define el comportamiento de un piloto usando un algoritmo de inteligencia artificial.
aCoche	Agente que define el comportamiento de un coche.

Tabla 10: Definición de los agentes para el juego de carreras de coches

A continuación, se describirán todos los agentes que forman parte del juego. Los agentes reutilizados no se volverán a describir porque comparten la misma función de evolución y visualización.

Agente aPaisaje

Universitat d'Alacant
Universidad de Alicante

Símbolo:	*aPaisaje*
Función de Evolución:	*Sí.*
Eventos Generados:	-
Eventos Aceptados:	-
Atributos:	-
Función de Visualización:	*Sí.*

Este agente representa el paisaje donde se aloja el circuito. Tanto su función de evolución como su función de visualización son muy sencillas ya que no hace nada más que dibujar el paisaje en el sistema gráfico.

$$f_{evolución}(q, aPaisaje) = \{aPaisaje \quad Para\ todos\ los\ casos\}$$

$$f_{visualizacion}(q, aPaisaje) = \{pPaisaje\}$$

Agente aCircuito

Símbolo:	$aCircuito$
Función de Evolución:	*Sí.*
Eventos Generados:	-
Eventos Aceptados:	-
Atributos:	-
Función de Visualización:	*Sí.*

Este agente representa el circuito donde se desarrollará la carrera. Al igual que el paisaje es un agente que sólo tiene como función representar el circuito. Por tanto, su función de evolución y visualización son de complejidad similar a *aPaisaje*:

$$f_{evolución}(q, ACircuito) = \{ACircuito \quad Para\ todos\ los\ casos\}$$

$$f_{visualizacion}(q, ACircuito) = \{PCircuito\}$$

Universitat d'Alacant
Universidad de Alicante

Agente aKeybUser1 y aKeybUser2

Símbolo:	$aKeybUser1, aKeybUser2$
Función de Evolución:	*Sí.*
Eventos Generados:	*eAcel, eCambiaRail*
Eventos Aceptados:	-
Atributos:	-
Función de Visualización:	*No.*

Estos dos agentes son los encargados de transformar los eventos del teclado en eventos de alto nivel que pueden entender los demás agentes. aKeybUser1 transformará las acciones del teclado del usuario 1 y

aKeybUser2 transformará las acciones del teclado del usuario 2. No tienen ningún atributo ni función de visualización. En este caso sólo tendrán función de evolución y es la encargada de generar los eventos pertinentes. Su función de evolución es la siguiente:

$$f_{evolucion}(q, aKeybUser1) = \begin{cases} aKeybUser1 & Si\ Tecla = 'a' \rightarrow q.insertar(eAcel_{\langle +1, Coche1 \rangle}) \\ aKeybUser1 & Si\ Tecla = 'z' \rightarrow q.insertar(eAcel_{\langle -1, Coche1 \rangle}) \\ aKeybUser1 & Si\ Tecla = 'x' \rightarrow q.insertar(eCambiaRail_{\langle Coche1 \rangle}) \\ aKeybUser1 & En\ cualquier\ otro\ caso \end{cases}$$

$$f_{evolucion}(q, aKeybUser2) = \begin{cases} aKeybUser2 & Si\ Tecla = 'j' \rightarrow q.insertar(eAcel_{\langle +1, Coche2 \rangle}) \\ aKeybUser2 & Si\ Tecla = 'm' \rightarrow q.insertar(eAcel_{\langle -1, Coche2 \rangle}) \\ aKeybUser2 & Si\ Tecla = 'n' \rightarrow q.insertar(eCambiaRail_{\langle Coche2 \rangle}) \\ aKeybUser2 & En\ cualquier\ otro\ caso \end{cases}$$

En estas funciones de evolución se puede ver qué eventos son generados para dos coches identificados con 1 y 2. Las teclas 'a', 'z', 'x' gestionan el coche1 y las teclas 'j', 'm' y 'n' gestionan el coche2 que corresponden a los coches que van a manejar dos jugadores. Se puede observar que las teclas 'a' y 'j' son teclas que aceleran el coche, mientras que las teclas 'z' y 'm' desaceleran el coche. Por otro lado, la tecla 'x' cambia de raíl el coche1, mientras que la tecla 'n' cambia de raíl el coche2.

Si quisiéramos introducir un nuevo jugador sólo tendríamos que seleccionar las teclas que manejaría este jugador y proceder como en el caso del coche1 o coche2 con un nuevo identificador de coche como por ejemplo coche3.

Otro tipo de diseño para la gestión del teclado puede ser el definir un único agente *aKeyboard* para gestionar todo el teclado unificando la gestión en un único agente. Sin embargo, esta solución tiene como desventaja el hecho de que añadir un nuevo usuario significaría la modificación del agente y consideraría a los n usuarios, jueguen o no jueguen. Sin embargo, la solución aquí aportada y separada por usuario tiene la ventaja de que sólo añadiremos a la cadena del sistema aquellos jugadores que realmente estén jugando.

Agente aJoystUser1 y aJoystUser2

Símbolo:	$aJoystUser\,1, aJoystUser\,2$
Función de Evolución:	*Sí.*
Eventos Generados:	*eAcel, eCambiaRail*
Eventos Aceptados:	-
Atributos:	-
Función de Visualización:	*No.*

Estos agentes son los encargados de transformar los eventos del joystick en eventos que pueden entender los demás agentes. No tienen ningún atributo y no tienen función de visualización. En este caso, sólo tendrán función de evolución que es la encargada de generar los eventos pertinentes. Su función de evolución es la siguiente:

$$f_{evolucion}(q, aJoystUser\,1) = \begin{cases} aJoystUser\,1 & Si\ Joyst_1 = up & \rightarrow & q.insertar(eAcel_{\langle +1, Coche1 \rangle}) \\ aJoystUser\,1 & Si\ Joyst_1 = down & \rightarrow & q.insertar(eAcel_{\langle -1, Coche1 \rangle}) \\ aJoystUser\,1 & Si\ Joyst_1 = bto_1 & \rightarrow & q.insertar(eCambiaRail_{\langle Coche1 \rangle}) \\ aJoystUser\,1 & En\ cualquier\ otro\ caso & & \end{cases}$$

$$f_{evolucion}(q, aJoystUser\,2) = \begin{cases} aJoystUser\,2 & Si\ Joyst_2 = up & \rightarrow & q.insertar(eAcel_{\langle +1, Coche2 \rangle}) \\ aJoystUser\,2 & Si\ Joyst_2 = down & \rightarrow & q.insertar(eAcel_{\langle -1, Coche2 \rangle}) \\ aJoystUser\,2 & Si\ Joyst_2 = Bto_1 & \rightarrow & q.insertar(eCambiaRail_{\langle Coche2 \rangle}) \\ aJoystUser\,2 & En\ cualquier\ otro\ caso & & \end{cases}$$

En este caso, se supone que pueden existir dos joysticks y que cada uno es manejado por un usuario.

Observando con detenimiento los agentes *aJoystUser* y *aKeybUser*, podemos apreciar que generan los mismos eventos, relacionados con el manejo de los coches por parte de los usuarios.

En el caso de que el joystick no esté conectado, este agente no añadirá ningún evento.

Agente aFisica

Símbolo:	$aFisica_{\langle registroPosCoches, pistas \rangle}$
Función de Evolución:	*Sí.*
Eventos Generados:	*eDespFuerza, eCurva, eVision, eActualiza*
Eventos Aceptados:	-
Atributos:	*regPosCoches,pistas*
Función de Visualización:	*No.*

El agente *aFisica* es el que se va a encargar de gestionar todo lo referente al motor físico. En este caso, se ha optado por concentrar todo el motor físico en un mismo agente. Sin embargo, Minerva no restringe a que esto sea así.

El otro atributo de *aFisica* es *pistas* que identifica cada una de las pistas del circuito. Estas pistas irán de 0 a N-1 donde N es el total de las pistas. El dato de cada pista nos dará el ángulo de giro de la pista.

Con estos atributos se podrá calcular las colisiones entre los coches. Para ello, se utilizará un algoritmo que testee si existen colisiones entre todos los coches del circuito. Para saber si dos coches tienen colisiones simplemente comprobamos si las cajas que los envuelven, que pueden calcularse a partir de los datos almacenados en las tuplas *<idCoche, rail, pos, dim, ang>,* intersectan entre ellas o no. Para todos los bloques que intersectan, se generará un evento $eDespFuerza_{<idCoche, dx, dy>}$, en cuyos atributos se identifica el coche que colisiona y los desplazamiento *dx, dy* que debe tener el coche para que la intersección se elimine. Al final, se obtiene una lista de eventos de *eDespFuerza* que añadiremos a los eventos del traductor de evolución. A esta función lo denominaremos `testColision(regPosCoches)`.

Además de calcular las colisiones, *aFisica* se encarga de calcular la fuerza centrífuga que afecta a los coches a los que se les debe aplicar. Esta fuerza centrífuga se podría tratar con un evento aparte, pero puede servir el evento $eDespFuerza_{<idCoche, dx, dy>}$ como evento que gestione esta fuerza. Se define la función `fzaCentr(registroPosCoche, pistas)` que devolverá una lista de eventos del tipo $eDespFuerza_{<idCoche, dx, dy>}$ que informa del desplazamiento del coche como resultado de aplicar la fuerza centrífuga. Este algoritmo a partir de la posición del coche, deducirá en

Universitat d'Alacant
Universidad de Alicante

qué pista está y le aplicará la fuerza centrífuga dependiendo de la velocidad del coche y del radio de curvatura de la curva. En caso de que no haya que aplicar fuerza centrífuga no se generará tal evento.

Para registrar en la base de datos las posiciones de los coches, el agente *aFisica* genera el evento $eActualiza_{<regPosCoches,\ pistas>}$. Este evento se genera de forma cíclica varias veces por segundo y sirve para actualizar la posición de los coches y calcular las posiciones de los coches aplicando los desplazamientos resultados de la velocidad. Esta actualización se realizará en el agente *aCoche* que es el agente donde están los datos del coche.

aFisica también se hace cargo de informar a todos los coches de qué es lo que tienen a su alrededor. Esto significa que generará los eventos *eCurva* para aquellos coches que estén a cierta distancia de una curva con su respectiva distancia aproximada y *eVision* para aquellos coches que tengan a otros coches cerca. Para generar el primer tipo de evento se implementa la función `testCurvas(regPosCoches, Pistas)` que calcula una lista de eventos del tipo *eCurva* para cada coche si hay un curva cerca de él. En ese caso, calculará la distancia aproximada y el atributo tomará uno de los valores *{lejos, cerca, muyCerca}*. Para el segundo tipo de eventos se implementa la función `testVision(regPosCoches)` que calcula una lista de eventos de tipo *eVision* para cada uno de las posibles posiciones de otros coches *{atrás, delante, aLado}* y para cada coche, indicando los coches que tiene a su alrededor. Si existe un coche cercano generará el evento con la distancia aproximada y el raíl donde se detectó el coche.

Una vez descritos todos los eventos que va a generar este agente, se puede definir la función de evolución como sigue:

$$f_{evolucion}(q, aFisica_{\langle regPosCoche, pistas\rangle}) = \left\{ aFisica_{\langle regPosCoche, pistas\rangle} \quad \begin{array}{l} q.insertar(testColision(regPosCoches)) \\ \wedge\, q.insertar(fzaCentr(regPosCoche, pistas)) \\ \wedge\, q.insertar(testCurvas(regPosCoches, pistas)) \\ \wedge\, q.insertar(testVision(regPosCoches)) \\ \wedge\, q.insertar(eActualiza_{\langle regPosCoche,\ pistas\rangle}) \end{array} \right\}$$

La función de evolución en este caso, se resume en insertar todos los eventos generados por las funciones que resuelven cada una de las funcionalidades del motor físico.

Como ya se comentó anteriormente, este agente podría haberse dividido en varios agentes especializados: uno para las colisiones, uno para la fuerza de la gravedad y otro para la actualización de posiciones. Esta es una de las libertades que nos brinda MINERVA a la hora de diseñar el motor físico.

Agente aIAPiloto

Símbolo:	$aIAPiloto^{eVision,\, eCurva}_{\langle idCoche,\, rail \rangle}$
Función de Evolución:	*Sí.*
Eventos Generados:	*eAcel, eCambiaRail*
Eventos Aceptados:	*eVision,eCurva*
Atributos:	*idCoche,rail*
Función de Visualización:	*No.*

El agente *aIAPiloto* implementa el comportamiento inteligente de los coches autónomos, basándose en los eventos de *eVision* y *eCurva* que como ya se ha visto, son generados por el agente *aFisica*. La función de evolución es la encargada de expresar las acciones que el agente va a ejecutar dependiendo de lo que acontece a su alrededor a partir de los eventos. Se trata de un simple algoritmo cognitivo del tipo *Si condición entonces acción,* donde la acción va a ser generada mediante un evento de tipo *eAcel* para acelerar el coche o un evento de tipo *eCambiaRail* para cambiar de raíl si el algoritmo piensa que es oportuno.

Como atributos, este agente tiene *idCoche* que identifica el coche que está manejando y que sobre todo sirve para identificar los eventos a los que tiene que responder, y *rail* que es el atributo que nos dice en qué raíl está el coche. Este último atributo nos condicionará a la hora de tomar las decisiones.

Como veremos a continuación, el orden en el que se evalúan las condiciones de la función permite determinar la prioridad de cada acción. Dicho esto, la expresión de la función de evolución queda de la siguiente forma:

$$f_{evolucion}\left(q, aIAPiloto^{eCurva,eVision}_{\langle idCoche,rail\rangle}\right)=\left|\begin{array}{ll}
aIAPiloto^{eCurva,eVision}_{\langle idCoche,rail\rangle} & \begin{array}{l} Si\ eCurva_{\langle idCoche,distancia\rangle}\in q \\ \wedge\ eCurva.idCoche=aIAPiloto.idCoche \\ \wedge\ eCurva.distancia=cerca \\ \rightarrow q.insertar\left(eAcel_{\langle aIAPiloto.idCoche,-1\rangle}\right)\end{array} \\
aIAPiloto^{eCurva,eVision}_{\langle idCoche,Derecha\rangle} & \begin{array}{l} Si\ eVision_{\langle idCoche,vista,distancia,rail\rangle}\in q \\ \wedge\ eVision.idCoche=aIAPiloto.idCoche \\ \wedge\ eVision.distancia=cerca\wedge eVision.vista=detras \\ \wedge\ eVision.rail\neq aIAPiloto.rail\wedge aIAPiloto.rail=Izquierda \\ \rightarrow q.insertar\left(eCambiaRail_{\langle aIAPiloto.idCoche,Derecha\rangle}\right)\end{array} \\
aAPiloto^{eCurva,eVision}_{\langle idCoche,Derecha\rangle} & \begin{array}{l} Si\ eVision_{\langle idCoche,vista,distancia,rail\rangle}\in q \\ \wedge\ eVision.idCoche=aIAPiloto.idCoche \\ \wedge\ eVision.distancia=cerca\wedge eVision.vista=Delante \\ \wedge\ eVision.rail=aIAPiloto.rail\wedge aIAPiloto.rail=Izquierda \\ \rightarrow q.insertar\left(eCambiaRail_{\langle aIAPiloto.idCoche,Derecha\rangle}\right)\end{array} \\
aIAPiloto^{eCurva,eVision}_{\langle idCoche,Izquierda\rangle} & \begin{array}{l} Si\ eVision_{\langle idCoche,vista,distancia,rail\rangle}\in q \\ \wedge\ eVision.idCoche=aIAPiloto.idCoche \\ \wedge\ eVision.distancia=cerca\wedge eVision.vista=detras \\ \wedge\ eVision.rail\neq aIAPiloto.rail\wedge aIAPiloto.rail=Derecha \\ \rightarrow q.insertar\left(eCambiaRail_{\langle aIAPiloto.idCoche,Izquierda\rangle}\right)\end{array} \\
aIAPiloto^{eCurva,eVision}_{\langle idCoche,Izquierda\rangle} & \begin{array}{l} Si\ eVision_{\langle idCoche,vista,distancia,rail\rangle}\in q \\ \wedge\ eVision.idCoche=aIAPiloto.idCoche \\ \wedge\ eVision.distancia=cerca\wedge eVision.vista=Delante \\ \wedge\ eVision.rail=aIAPiloto.rail\wedge aIAPiloto.rail=Derecha \\ \rightarrow q.insertar\left(eCambiaRail_{\langle aIAPiloto.idCoche,Izquierda\rangle}\right)\end{array} \\
aIAPiloto^{eCurva,eVision}_{\langle idCoche,rail\rangle} & \begin{array}{l} Si\ eVision_{\langle idCoche,vista,distancia,rail\rangle}\in q \\ \wedge\ eVision.idCoche=aIAPiloto.idCoche \\ \rightarrow q.insertar\left(eAcel_{\langle aIAPiloto.idCoche,+1\rangle}\right)\end{array} \\
aIAPiloto^{eCurva,eVision}_{\langle idCoche,rail\rangle} & En\ otro\ caso
\end{array}\right|$$

En la expresión de la función de evolución se puede observar que el piloto cambiará de raíl cuando tenga un coche en el raíl contrario y esté por detrás para tapar el paso del coche rival o si el rival está delante cambiará de raíl cuando esté en el mismo. Todo esto se ejecutará si no hay una curva cercana con lo que significará que frenará para poder tomar la curva que es la acción prioritaria y por eso es la primera expresión dentro de la función de evolución. En el caso que no tenga rivales a su alrededor y no tenga curvas cerca entonces la acción por defecto será acelerar el coche.

En realidad este algoritmo implementa un máquina de estados, típica forma en que se resuelve el comportamiento inteligente de los bots en muchos juegos. En particular, la máquina de estados asociada al agente *aIAPiloto* es la que se muestra en la Figura 38.

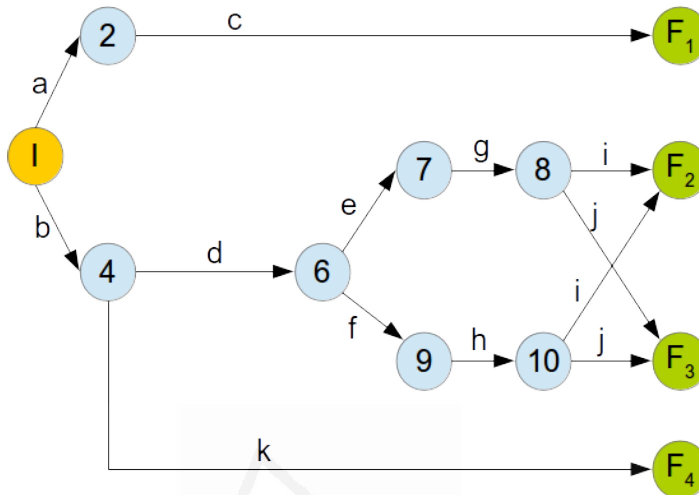


Figura 38: Máquina de estado del agente aIPiloto

En esta figura los estados intermedios de la máquina de estados están etiquetados por números, el estado inicial por el estado I y los estados finales etiquetados por F_i. Si la máquina de estado no llega a ningún estado final, el agente no cambia de estado correspondiéndose con la última linea de la función de evolución y si llega a algún estado final entonces hace los siguiente:

Universitat d'Alacant
Universidad de Alicante

- F1: Añade a la pila de eventos $eAcel_{<aIAPiloto.idCoche,-1>}$ ejecutando la acción de frenar el coche.
- F2: Añade a la pila de eventos $eCambiaRail_{<aIAPiloto.idCoche,Derecha>}$ ejecutando la acción de cambio de raíl derecho.
- F3: Añade a la pila de eventos $eCambiaRail_{<aIAPiloto.idCoche, Izquierda>}$ ejecutando la acción de cambio de raíl izquierdo.
- F4: Añade a la pila de eventos $eAcel_{<aIAPiloto.idCoche,+1>}$ ejecutando la acción de acelerar el coche.

Ahora sólo falta enumerar las condiciones que pasan de un estado a otro y que en la figura de la máquina de estado (Figura 38) están etiquetadas con letras:

a: $eCurva_{\langle idCoche,distancia \rangle} \in q$ *y eCurva.idCoche=aIAPiloto.idCoche.*
b: $eVision_{\langle idCoche,distancia \rangle} \in q$ *y eVision.idCoche=aIAPiloto.idCoche.*
c: *eCurva.distancia = cerca*
d: *eVision.distancia = cerca*
e: *eVision.vista = detras*
f: *eVision.vista = delante*
g: *eVision.rail = AIAPiloto.rail*
h: *eVision.rail = AIAPiloto.rail*
i: *aIAPiloto.rail = izquierda*
j: *aIAPiloto.rail = Derecha*

También se puede observar que los eventos que se consideran sólo son los que tienen el *idCoche* igual al del piloto de inteligencia artificial.

En cuanto a la función de visualización no existe ya que este agente no tiene ninguna visualización.

Agente ACoche

Símbolo:	$aCoche^{eAcel,\ eCambioRail,\ eColision,\ eFrame,\ eActualiza}_{\langle idCoche,\ estado,\ vel,\ pos,\ rail,\ despCol \rangle}$
Función de Evolución:	*Sí.*
Eventos Generados:	*No.*
Eventos Aceptados:	*eAcel, eCambioRail, eDespFuerza, eActualiza*
Atributos:	*idCoche, estado, vel, pos, rail, despCol*
Función de Visualización:	*Sí.*

aCoche es el agente que gestiona el coche conducido tanto por el usuario como por el agente *aIAPiloto*. El identificador *IdCoche* permite distinguir cada coche. Cada evento de entrada (generado por el agente de teclado *aKeyb* o por el agente de joystick *aJoyst*) y cada evento de la IA (generado por el agente *aIAPiloto*) incorpora el identificador del coche al que se aplica (*IdCoche*), por lo que cada agente *aCoche* está vinculado a un jugador, real o virtual.

Además del atributo *idCoche*, este agente tiene un atributo estado que puede tener los valores *{OK, Fuera}*. Si el coche está en el estado *Fuera*, se queda parado hasta que el estado vuelve a *OK*. De esta forma, podemos implementar una penalización cuando los coches se salen del circuito.

Universitat d'Alacant
Universidad de Alicante

Los dos siguientes atributos *pos* y *rail* identifican la posición del coche dentro del circuito, de la misma manera que fue descrita en el agente *aFisica*. El atributo *pos* es un número real con un decimal que determina la pista donde está y su posición dentro de esta, y *raíl* identifica si está en el raíl *izquierdo* o *derecho*.

Por último, está el atributo *desplCol* que es un punto 2D que identifica lo que se desplaza el coche en los ejes X y Z después de una colisión o debido a la fuerza centrífuga. El módulo de este punto 2D nos dirá si el coche debe cambiar al estado de *Fuera* o si se mantiene en el estado *OK*.

Los eventos que acepta *aCoche* son *eAcel*, *eCambioRail*, *eDespFuerza*, y *eActualiza*. El primero especifica el incremento de la velocidad. Si este incremento es negativo la velocidad del coche disminuirá, si es positivo aumentará. El segundo indica el cambio del raíl del coche. El tercero hará efectivos los desplazamientos debidos a la colisión almacenados en el atributo *desplCol*. Por último, *eActualiza* será el responsable de desplazar el coche y aplicar los desplazamientos según la velocidad.

Considerando lo anterior la función de evolución se expresa de la siguiente manera:

Universitat d'Alacant
Universidad de Alicante

$$f_{evolucion}(q, aCoche^{Eventos}_{\langle attr \rangle}) = \left| \begin{array}{lll} aCoche^{Eventos}_{\langle attr\,1 \rangle} & Si\, eAcel_{\langle idCoche,\, incrAcc \rangle} \in q & (1) \\ & \wedge aCoche.estado = Ok & \\ & \wedge aCoche.idCoche = eAcel.idCoche & \\ aCoche^{Eventos}_{\langle attr\,2 \rangle} & Si\, eUpdate_{\langle BDPosCoches,Pistas \rangle} \in q & (2) \\ & \wedge ACoche.estado = Ok & \\ & \wedge ACoche.pos + Step * vel < N & \\ ACoche^{Eventos}_{\langle attr\,3 \rangle} & Si\, EFrame_{\langle BDPosCoches,Pistas \rangle} \in q & (3) \\ & \wedge ACoche.estado = Ok & \\ & \wedge ACoche.pos + Step * vel > N & \\ aCoche^{Eventos}_{\langle attr\,4 \rangle} & Si\, eCambiaRail_{\langle idCoche,\, rail \rangle} \in q & (4) \\ aCoche^{Eventos}_{\langle attr\,5 \rangle} & Si\, eCambiaRail_{\langle idCoche,\, rail \rangle} \in q & (5) \\ & \wedge aCoche.estado = Ok & \\ & \wedge aCoche.idCoche = eCambiaRail.idCoche & \\ & \wedge aCoche.rail = derecha & \\ aCoche^{Eventos}_{\langle attr\,6 \rangle} & Si\, eColision_{\langle idCoche,dx,\,dy \rangle} \in q & (6) \\ & \wedge aCoche.idCoche = eDespFuerza.idCoche & \\ aCoche^{Eventos}_{\langle attr\,7 \rangle} & Si\, eUpdate_{\langle BDPosCoches,Pistas \rangle} \in q & (7) \\ & \wedge |aCoche.desplCol| > 1 & \\ aCoche^{Eventos}_{\langle attr\,8 \rangle} & Si\, eUpdate_{\langle BDPosCoches,Pistas \rangle} \in q & (8) \\ & \wedge aCoche.estado = fuera & \\ aCoche^{Eventos}_{\langle attr\,9 \rangle} & Si\, eUpdate_{\langle BDPosCoches,Pistas \rangle} \in q & (9) \\ & \wedge aCoche.despCol = 0 & \\ aCoche^{Eventos}_{\langle attr \rangle} & En\, otro\, caso & (10) \end{array} \right|$$

Para simplificar la notación en la función de evolución, todos los eventos se han agrupado bajo el nombre Eventos que son *eAcel, eCambioRail, eDespFuerza y eActualiza*. Además, se han etiquetado los atributos como *attrN*, para explicar a continuación cómo se actualizan los atributos en cada caso. Partimos de que

<attr> = <idCoche, estado, velocidad, pos, rail, despCol>

es el estado inicial de los atributos del agente *aCoche*.

Además se han numerado las expresiones de la función de evolución para poder hacer referencia a dichas expresiones. Dicho esto, a continuación explicamos la funcionalidad de todas las expresiones de la función de evolución del coche:

1. Esta línea corresponde a cuando se produce un evento de aceleración. En este caso, los atributos se modifican de la siguiente manera:

<attr1> = <idCoche, estado, velocidad+eAcel.incrVel, pos, rail, despCol>

es decir, incrementamos la velocidad del coche. Para acelerar el coche en existen dos condiciones necesarias y es que el estado del coche debe ser *OK* y que el evento de aceleración tenga el mismo *idCoche* que el agente.

2. Esta línea corresponde con el comportamiento del agente cuando recibe un evento de sincronización *eActualiza* para las posiciones menores que N, siendo N el número máximo de pistas. Este evento es el responsable de mover el coche en la pista. En este caso, los atributos deben actualizarse de la siguiente manera:

   *<attr2> = <idCoche, estado, pos + vel * Step, rail, MAX(0,despCol-StepCol)>*

3. Esta línea es similar a la anterior salvo que trata el caso para posiciones mayores que N. Entonces vuelve a la pista 0 para hacer el circuito circular. En este caso, los atributos deben actualizarse de la siguiente manera:

   *<attr3> = <idCoche, estado, vel * Step, rail, MAX(0,despCol-StepCol)>*

4. Esta línea corresponde al cambio de raíl de izquierda a derecha y modificamos los atributos de la siguiente manera:

   <attr4> = <idCoche, estado, pos, derecho, despCol>

5. Esta línea corresponde al cambio de raíl de derecha a izquierda y la modificación de atributos es de la siguiente manera:

   <attr5> = <idCoche, estado, pos, izquierdo, despCol>

   Hasta ahora, las líneas explicadas se realizaban cuando el estado del coche estaba en *OK*. A partir de este momento el estado del coche no tiene porque ser *OK*.

6. Esta línea corresponde a la respuesta de un evento de desplazamiento por una fuerza (puede ser por colisión o por fuerza centrífuga) aumentando el desplazamiento de colisión del coche expresado como:

Universitat d'Alacant
Universidad de Alicante

<attr6> = <idCoche, estado, vel, pos, rail, despCol + EdespFuerza.despCol>

7. Esta línea corresponde cuando el coche se sale fuera de pista, expresado por la condición de que el módulo del vector *despCol* sea mayor que uno. En este caso, el estado del coche pasa a estado *Fuera* de la siguiente manera:

   <attr7> = <idCoche, Fuera, vel, pos, rail, despCol>

8. Esta línea corresponde a la aplicación de la penalización del tiempo cuando el coche está en el estado *Fuera*. Esta penalización se establece en el *<attr8>* que va descontando cada cierto tiempo una constante, *StepCol*, al vector *despCol* expresado como:

   <attr8> = <idCoche, estado, vel, pos, rail, MAX(0,despCol-StepCol)>

9. *E*sta línea corresponde a la condición de que el módulo del vector *despCol* es igual a 0. Entonces el coche vuelve otra vez al estado *OK* con velocidad 0. Esto está expresado en *<attr9>* como sigue:

   <attr9> = <idCoche, estado, 0, pos, rail, 0>

10. Por último, existe el caso por defecto, es decir, cuando no se cumple ninguno de los otros casos. Entonces el agente queda exactamente como está.

Universitat d'Alacant
Universidad de Alicante

Así queda completamente definida la función de evolución. Hay que tener en cuenta que la función de evolución sólo ejecuta uno de los casos definidos cada ciclo de evolución.

Una vez definida la función de evolución, la función de visualización queda como sigue:

$$f_{visualizacion}\left(q, aCoche^{Eventos}_{\langle idCoche, estado, vel, tramoPista, despCol\rangle}\right) = \left\{tDesplaza_{\langle tramoPista\rangle}\left(tTono_{\langle vel\rangle}\left(pCocheIdCoche\right)\right)\right\}$$

5.3. Cadena inicial

Una vez establecido todos los elementos del lenguaje L(M) se puede definir la cadena inicial con el que el sistema va a arrancar. Hay que considerar

que la cadena inicial determinará el comportamiento del sistema. Por ejemplo, una cadena inicial del sistema podría ser la siguiente:

$$\begin{gathered} \mathit{aCircuito} \cdot \mathit{aPaisaje} \cdot \mathit{aFisica}(\mathit{aJoytickUser1}(\mathit{aKeybUser1}(\mathit{aCoche}_{\langle 1,OK,0,0,Izquierdo,0\rangle})) \\ \cdot\, \mathit{aJoytickUser2}(\mathit{aKeybUser2}(\mathit{aCoche}_{\langle 2,OK,0,0,Derecho,0\rangle})) \\ \cdot\, \mathit{aIAPiloto}_{\langle 3,Izquierdo\rangle}(\mathit{aCoche}_{\langle 3,OK,0,0.5,Izquierdo,0\rangle}))\cdot \\ \cdot\, \mathit{aIAPiloto}_{\langle 4,Derecho\rangle}(\mathit{aCoche}_{\langle 4,OK,0,0.5,Derecho,0\rangle}))) \end{gathered}$$

En este caso el coche con identificador 1 está manejado por el usuario 1 y el coche con identificador 2 está manejado por el usuario 2. Sin embargo, tanto el coche 3 como el 4 están manejados por el algoritmo de inteligencia artificial.

Existen cadenas que tienen el mismo resultado que la anterior. Por ejemplo, la siguiente cadena

$$\begin{gathered} \mathit{aFisica}(\ \mathit{aCircuito} \cdot \mathit{aPaisaje} \cdot \mathit{aJoytickUser1}(\mathit{aKeybUser1}(\mathit{aCoche}_{\langle 1,OK,0,0,Izquierdo,0\rangle})) \\ \cdot\, \mathit{aJoytickUser2}(\mathit{aKeybUser2}(\mathit{aCoche}_{\langle 2,OK,0,0,Derecho,0\rangle})) \\ \cdot\, \mathit{aIAPiloto}_{\langle 3,Izquierdo\rangle}(\mathit{aCoche}_{\langle 3,OK,0,0.5,Izquierdo,0\rangle})) \\ \cdot\, \mathit{aIAPiloto}_{\langle 4,Derecho\rangle}(\mathit{aCoche}_{\langle 4,OK,0,0.5,Derecho,0\rangle}))) \end{gathered}$$

obtendrá el mismo resultado que la anterior cadena entre otras cosas porque tanto *aCircuito* como *aPaisaje* no reaccionan ante ningún evento que genera *aFisica*. Es decir, estas dos son cadenas semánticamente iguales.

Por otro lado, pueden existir cadenas que aunque son correctas sintácticamente no tienen ningún interés para el juego planteado. La siguiente cadena

$$\begin{gathered} \mathit{aCircuito} \cdot \mathit{aPaisaje} \cdot \mathit{aJoytickUser1}(\mathit{aKeybUser1}(\mathit{aCoche}_{\langle 1,OK,0,0,Izquierdo,0\rangle})) \\ \cdot\, \mathit{aJoytickUser2}(\mathit{aKeybUser2}(\mathit{aCoche}_{\langle 2,OK,0,0,Derecho,0\rangle})) \\ \cdot\, \mathit{aIAPiloto}_{\langle 3,Izquierdo\rangle}(\mathit{pCoche}_{\langle 3,OK,0,0.5,Izquierdo,0\rangle})) \\ \cdot\, \mathit{aIAPiloto}_{\langle 4,Derecho\rangle}(\mathit{aCoche}_{\langle 4,OK,0,0.5,Derecho,0\rangle})) \end{gathered}$$

elimina de la cadena el agente *aFisica* con lo que no se generarán nunca los eventos de colisión. Es decir, los coches pueden acelerar todo lo que quieran que nunca serán penalizados cuando se salgan de una curva o cuando choquen con otro coche. Es más, esos choques ni siquiera se detectarán.

Existen otras cadenas que configuran distintos juegos. Por ejemplo, si eliminamos de la primera cadena comentada los agentes de inteligencia artificial, el juego se convierte en un juego entre dos jugadores sin más rivales que ellos dos.

Universitat d'Alacant
Universidad de Alicante

$$aCircuito \cdot aPaisaje \cdot aFisica(aJoytickUser1(aKeybUser1(aCoche_{<1,OK,0,0,Izquierdo,0>}))$$
$$\cdot\, aJoytickUser2(AKeybUser2(aCoche_{<2,OK,0,0,Derecho,0>}))$$

Uno de los aspectos más interesantes del lenguaje L(M) es que cualquier agente definido puede ser reutilizado en otros juegos. Por ejemplo, se puede suponer que en vez de una carrera de coches se desea implementar un juego de carreras de naves espaciales en un espacio 3D. El algoritmo de inteligencia artificial podría funcionar si las naves aceptaran los eventos de *eAcel* y *eCambioRail* transformando estos eventos a un desplazamiento lateral en el espacio 3D y la aceleración como una aceleración de la nave espacial pudiendo quedar una cadena como la que sigue:

$$aIAPiloto(aNave_{<attr>})$$

De esta manera se pueden reutilizar algoritmos de inteligencia artificial para otros sistemas.

Como se puede observar una vez definidos los diferentes elementos del lenguaje L(M) las posibilidades que se nos abren son muy amplias teniendo un lenguaje común para el desarrollo de sistemas.

Universitat d'Alacant
Universidad de Alicante

VII. Discusión del modelo

Para una mejor exposición de las ventajas e inconvenientes que tiene el modelo se va a comparar con los sistemas de OGRE y Unity *que tienen algunos objetivos similares a los de MINERVA, como la gestión de los motores físicos y de inteligencia artificial o la desvinculación de los detalles de la API de renderizado.*

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

1. Discusión del modelo

Recopilando lo dicho hasta ahora, un sistema de realidad virtual se ha dividido en cuatro componentes principales: el motor gráfico, los dispositivos de entrada, el motor físico y el motor de inteligencia artificial.

El motor gráfico tiene como principal objetivo visualizar en los dispositivos de salida de una gran variedad de tamaños y capacidades, el entorno y los personajes que componen el mundo virtual. Es una de las fases más estudiadas y está cableada por casi todas las tarjetas gráficas de hoy en día. Además, ya existen APIs maduras independientes del dispositivo, como OpenGL y Direct3D y que se han convertido en un estándar dentro de los sistemas gráficos. Sin embargo, sólo son una mera API de comunicación con el dispositivo gráfico. Una forma de poder adaptar la información a visualizar de forma dinámica es una característica importante en el proceso de renderizado. Esta adaptación es posible en MINERVA gracias a que podemos tener varios traductores de renderizado (TR) ajustados a la salida gráfica.

Los traductores de renderizado, como ya hemos visto, se pueden utilizar para otros cometidos. Por ejemplo, en los videojuegos que se juegan en red, transportar por la misma todos los elementos de la escena eficientemente es vital. Esto supone que la escena que se está dibujando pueda ser fielmente reproducida en otros medios que no sean los visuales. Para ello, el proceso de visualización y de envío por la red debe ser el mismo. El traductor TR es un sistema que describe la escena en un modelo que puede ser tanto dibujado, como transportado por red, como guardado en cualquier formato de dibujo. Este tipo de renderizado resulta necesario para el procesamiento de escenas de realidad virtual, máxime si la realidad a procesar es cada vez más compleja.

El traductor TR se encarga de evaluar las características de la tarjeta gráfica de salida y según las potencialidades de la tarjeta dibujar con una técnica u otra los elementos visibles.

Dentro de este tratamiento unificado se puede incorporar la sonorización. Efectivamente, mientras que tanto el renderizado como la sonorización han tenido una evolución considerable, hasta ahora no se ha tratado de juntar en un mismo proceso tanto el sonido como los

Universitat d'Alacant
Universidad de Alicante

componentes gráficos. Esto es importante porque los sonidos, generalmente, están asociados a objetos gráficos que se mueven en la escena. Este proceso está sincronizado por el traductor TR teniendo como resultado una coherencia entre sonido y visualización producida por el procesamiento conjunto.

Por ejemplo, si en la escena se dibuja un coche que viene hacia el usuario, es muy importante la posición del coche con respecto al usuario para realizar el efecto Doppler dependiente de la posición. Esto evidencia que un sistema que aúne visualización y sonorización es importante para realizar escenas con coherencia sonora y visual.

Considerando el siguiente componente, los dispositivos de entrada son importantes, en cuanto que son parte fundamental para la sensación de inmersión en el mundo virtual. Aunque durante muchos años el desarrollo de estos dispositivos ha estado estancado, actualmente están teniendo un auge considerable con el desarrollo de nuevas formas de interacción hombre-máquina gracias a la industria del entretenimiento.

Uno de los problemas de los dispositivos de entrada es que no existe una forma estándar para modelar la interacción hombre-máquina. El sistema de eventos modelado por MINERVA generaliza las partes comunes de las acciones con los dispositivos de entrada y oculta las partes diferenciadoras. De esta manera, se puede utilizar varios tipos de dispositivos de entrada para realizar las mismas acciones en el entorno virtual, facilitando la integración entre el sistema de realidad virtual y los dispositivos.

El motor físico, por otro lado, es el encargado de simular con menor o mayor precisión la realidad que simula el sistema de realidad virtual, y en la actualidad está en un proceso parecido al de los motores gráficos: trasladar a procesadores especializados todos los cálculos necesarios para simular la física del mundo virtual.

Actualmente los motores físicos están centrados en el cálculo de los modelos físicos que pretenden simular. Sin embargo, el motor físico está íntimamente relacionado con el motor gráfico.

El motor gráfico tiene la propiedad de albergar la representación geométrica, así como la posición de todos los objetos que existen en el mundo virtual. Esta descripción geométrica es necesaria para el cálculo de colisiones que realiza el motor físico, y debe existir una comunicación estrecha entre ambos motores. Además, el motor físico puede modificar el número de elementos que existen en la escena, por ejemplo, realizando una

destrucción y descomponiendo en pequeñas partes un determinado objeto. El aumento de elementos a dibujar incidirá notablemente sobre el rendimiento del motor gráfico.

Además, el motor gráfico dibuja justo después de realizar los cálculos del motor físico y antes del siguiente instante de tiempo. Por tanto, tanto el sistema gráfico como la simulación física inciden sobre el rendimiento del sistema y una coexistencia ágil, hace que se puedan optimizar las formas en que se relacionan ambos motores. Pero, tal vez, sea más importante, si cabe, tener en cuenta que el motor gráfico va a dibujar los efectos generados por el motor físico y que debe, por tanto, estar al tanto de las modificaciones que el motor físico está realizando sobre los objetos de la escena. En MINERVA, esta relación está modelada mediante los traductores de evolución (TE) y de visualización (TV). El traductor TE tiene como salida una cadena que luego es introducido en el traductor TV. De esta manera, lo generado por el motor gráfico, gracias a los agentes que modelan tal motor, pasa directamente al traductor de visualización TV.

Esta íntima relación se puede explicar con la simulación de una explosión. El motor físico sería el encargado que modelar todos los fenómenos que intervienen en una explosión y esto produce una descomposición en trozos más pequeños del objeto a explosionar. Esta descomposición es realizado por el traductor TE mediante la función de evolución. Esto significa que el motor gráfico deberá estar informado de los nuevos trozos e ir dibujando todos los trozos descompuestos a medida que el motor físico vaya haciendo que la explosión evolucione. Como la cadena evolucionada por el traductor TE pasa directamente a al traductor TV, esta comunicación queda modelada en MINERVA. La descomposición del elemento aumentará la carga del motor físico como consecuencia del aumento de los elementos involucrados en la realidad virtual, pero también supone un incremento en los elementos que el motor gráfico debe mostrar. Este ejemplo explica la existencia de una relación muy estrecha entre el motor físico y el motor gráfico.

En cuanto al motor de inteligencia artificial, los sistemas multiagentes están dando resultados importantes para estos motores, aunque sigue existiendo un diversidad demasiado grande en las arquitecturas diseñadas. Los motores de inteligencia artificial tienen la peculiaridad de estar diseñados a medida del mundo de realidad virtual para los que se implementan. Es decir, no existe una estrategia común para el diseño de estos motores y cada videojuego tiene el suyo propio. Esto produce que el

coste de implementación sea, posiblemente, el más caro de desarrollar de todos los anteriores componentes.

MINERVA modela un patrón de diseño que pueda orientar la arquitectura de los motores de inteligencia artificial gracias a la función de evolución. Se podrían generalizar y explotar motores de inteligencia artificial con procesadores dedicados al igual que ya existen para los motores físicos y los motores gráficos sabiendo cuál es la patrón general de la función de evolución.

Por otro lado, el motor de inteligencia artificial y los sistemas multiagentes en particular, tienen una estrecha relación con el entorno que les rodea. Entorno que se describe por la geometría almacenada en el motor gráfico y que cambia con el tiempo gracias a las leyes físicas simuladas por el motor físico. Así, tanto el motor físico como el motor gráfico se relacionan con el motor de inteligencia artificial. Si se tiene en cuenta que el motor de inteligencia artificial puede modificar el entorno mediante los agentes inteligentes, sus acciones modifican tanto los elementos del motor físico como los del motor gráfico. Así, por ejemplo, si un avatar realiza la acción de lanzar una granada ésta realizará un movimiento parabólico simulado por el motor físico y visualizado por el motor gráfico, realizará una explosión y este producirá múltiples fragmentos, todo ello en la parte del motor físico que a su vez afectará al motor gráfico.

En el lenguaje L(M) se modelan tanto agentes inteligentes como agentes que modelan el motor físico y por tanto la relación que existe entre el motor físico y el motor de inteligencia artificial es modelada en MINERVA gracias a que un agente (que modela el motor físico) crea un ámbito de aplicabilidad sobre subagentes limitado por los paréntesis del lenguaje L(M). Así, viendo cómo es la cadena podemos ver la relación que existe entre el motor físico y el motor de inteligencia artificial. Como, además, el traductor TE y el traductor TV ya están relacionados como vimos con anterioridad, podemos ver como MINERVA realiza la conexión entre los tres motores de una forma natural.

De todo lo anterior no se puede obviar que todos los motores tienen una relación entre ellos a la que no se está prestando la suficiente atención. La Figura 39 ilustra estas relaciones.

Universitat d'Alacant
Universidad de Alicante

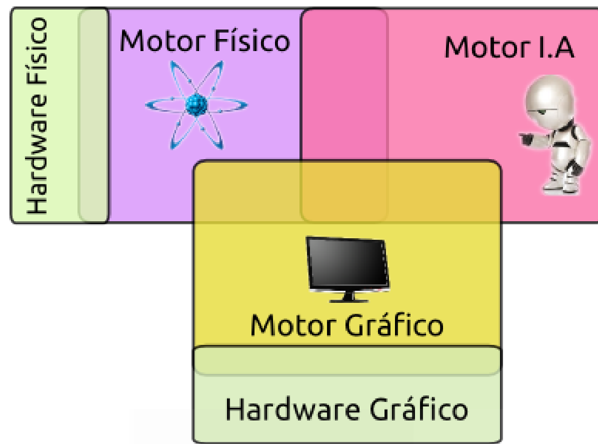


Figura 39: Relación entre los motores de un sistema de realidad virtual

Por otro lado, los dispositivos de entrada ejecutan acciones del usuario que modifican el mundo virtual que será ejecutado por el motor físico y visualizado por el motor gráfico. Además, estas modificaciones actuarán de alguna manera sobre el comportamiento de los agentes inteligentes que están en el mundo virtual y por tanto, otra vez, todos los componentes del sistema de realidad virtual están afectándose mutuamente. En este caso los eventos forman parte del lenguaje L(M) y esta relación entre dispositivos de entrada y motores físico y de inteligencia artificial también queda modelado en MINERVA.

En los entornos con características geográficas, se evidencia la relación que existe entre el motor físico y el motor de inteligencia artificial. Hay varias soluciones para resolver la forma en que se relacionan los agentes con el motor físico que define el entorno donde están los agentes. Por ejemplo, en [Maes and Darrell 1995] se describe la implementación del sistema ALIVE, un sistema de realidad virtual que relaciona agentes autónomos con usuarios, teniendo como uno de sus requerimientos, relacionar el motor físico del mundo virtual con las acciones de los agentes. En el sistema MASON desarrollado en [Luke and Cioffi-Revilla 2004] también se implementa una cierta relación entre el motor físico y el motor gráfico, donde los agentes tiene una estrecha relación con el motor físico. Sin embargo, esta relación es muy específica en los dos sistemas y no tienen elementos que puedan ser generalizables. No existen un estudio

formal sobre la relación entre el entorno, el motor físico y los agentes, como lo hace MINERVA y además deja de lado al motor gráfico y a los dispositivos de entrada.

Por otro lado, se puede comprobar la pretensión por parte de los sistemas multiagentes de la posibilidad de poder estandarizar los mecanismos de comunicación para que dos agentes puedan entenderse. Sin embargo, no existen un estándar preponderante que simplifique el proceso de comunicación entre agentes y al final se opta por un sistema de comunicación particular del entorno de realidad virtual que se está implementando. Con el modelo de eventos de MINERVA todo agente que sea gestionado con el modelo, podrá aprovecharse del sistema de comunicación implementado en MINERVA.

Como se puede hacer notar, todos los componentes hasta ahora analizados se han desarrollado por separado y se ha solucionado la problemática de cada una de sus partes específicas. Ahora, los modelos están lo suficientemente maduros como para abordar un estudio de la integración de todos los componentes.

MINERVA es un modelo matemático donde se incluye la descripción del mundo virtual, se modelan las leyes físicas del motor físico y donde pueden desarrollar los agentes sus acciones inteligentes que ayuda de forma eficaz a la implementación de los sistemas de realidad virtual. Efectivamente, la capacidad que tienen los modelos formales como MINERVA para extraer las características abstractas de los elementos a estudiar, suponen un objetivo interesante para que la implementación de estos sistemas sea más rápida y libre de errores ya que la implementación se basa en un patrón de diseño especificado mediante este sistema formal MINERVA.

Además, la utilización de MINERVA tiene otras ventajas. Por ejemplo, el uso de un lenguaje común, el lenguaje del modelo formal, tiene como virtud el poder pasar un elemento implementado de un entorno a otro entorno totalmente diferente para poder comparar su eficacia. Esto es especialmente importante en la simulación. Uno de los problemas significativos que tiene la simulación es el de la reproducción de los resultados por diferentes equipos de investigación como se estudia con profundidad en [Axelrod 2003]. Si estos equipos usan el mismo lenguaje formal para desarrollar sus investigaciones, podrán ser verificadas fácilmente por otros equipos.

Por otro lado, los lenguajes formales como MINERVA son la pieza

fundamental dentro de los sistemas formales. Es la forma de expresión que se utiliza en los sistemas formales libres de ambigüedades y nos ayudan a construir sistemas comprobando su validez. Además, gracias a que los sistemas formales están muy estudiados por las matemáticas, se pueden aprovechar todos los avances realizados hasta ahora para ayudar a desarrollar sistemas de realidad virtual libres de ambigüedades y correctos desde el punto de vista de los criterios del diseño.

Una forma de especificar un lenguaje formal es mediante gramáticas que son representaciones estructurales que desempeñan un papel importante en los sistemas formales. Son modelos muy útiles para el desarrollo de software destinado a procesar datos que poseen una estructura recursiva, como los árboles de escena. El uso de gramáticas, como las gramáticas M y V, para describir modelos tanto en motores gráficos, como motores físicos, como motores de inteligencia artificial y que de alguna forma modelen también sus relaciones ayuda a expresar estos sistemas en un lenguaje común, libre de ambigüedades y con una abstracción suficiente como para poder realizar diferentes experimentos en todos aquellos entornos implementados con la arquitectura definida a través de estas gramáticas definidas en MINERVA.

Esta gramática puede ser usada también para los dispositivos de entrada. Estas estructuras matemáticas tienen la capacidad de separar la parte fundamental de las acciones de un dispositivo de entrada de la parte concreta del dispositivo. Esta separación es importante si se quiere una homogeneidad en las acciones de los dispositivos de entrada como hace MINERVA con su sistema de eventos.

Universitat d'Alacant
Universidad de Alicante

2. Comparación con otros sistemas

En el capítulo de experimentación se han modelado varios entornos virtuales con el fin de demostrar que MINERVA puede utilizarse en varios tipos de sistemas pero con la ventaja de tener un modelo formal común.

Para una mejor exposición de las ventajas e inconvenientes que tiene el modelo se va a comparar con otros sistemas que tienen algunos objetivos similares a los de MINERVA, como la gestión de los motores físicos y de inteligencia artificial o la desvinculación de los detalles de la API de renderizado.

Por ser muy populares se han elegido el sistema de creación de videojuegos Unity y el motor de renderizado 3D orientado a escenas OGRE. Ambos sistemas tienen algunos objetivos similares a los que desarrolla MINERVA, a saber:

- Tener un sistema de renderizado que lo aísle de los detalles de API de renderizado de más bajo nivel como OpenGL y Direct3D.
- Tener herramientas para modelar motores físicos y motores de inteligencia artificial.
- Ser multiplataforma.
- Reutilizar objetos ya desarrollados.

A continuación, se va a hacer un análisis de los diferentes modelos, OGRE, Unity y MINERVA y se identificarán las ventajas y los inconvenientes de cada uno de ellos. Para un análisis estructurado se verá cómo cada sistema modela los tres motores de los que se ha estado hablando durante toda la tesis: motor gráfico, motor físico y motor de inteligencia artificial.

3. Comparación de los motores gráficos

Como ya se ha dicho anteriormente, el motor gráfico es el encargado de visualizar la escena del sistema en un dispositivo visual.

En el caso de OGRE, el motor gráfico es un motor para generar visualizaciones en 3D. De hecho, es posible hacer visualizaciones en 2D, pero dibujando el sistema en 3D aplicando todo sobre un plano. El sistema no se adapta a modelos 2D de forma natural. Esto supone que se va a usar un sistema con ciertas características que no se van a utilizar y que por tanto estará sobredimensionado para juegos 2D.

Además, el motor gráfico de OGRE está implementado para Direct3D y OpenGL y aunque hay una capa que aísla estas dos librerías, no es posible utilizar otra API de dibujado que no sean estos dos sistemas. Este es el motivo de que no se puedan utilizar sistema de dibujado 2D, porque está excesivamente orientado a estos dos modelos de API con características muy similares.

En el caso de Unity, el motor gráfico también está muy orientado a la

Universitat d'Alacant
Universidad de Alicante

representación 3D. Al igual que OGRE, utiliza un árbol de escena para almacenar los elementos del sistema y tiene todas las características necesarias para un dibujado excelente en 3D. Pero a diferencia de OGRE, Unity sí que tiene una versión para modelados en 2D. El problema estriba en que o se hace un sistema en 2D o se hace un sistema 3D, pero no se puede diseñar un entorno virtual para ambos tipos de espacios con un mismo modelo del sistema.

En el caso de MINERVA, y gracias a sus traductores, la descripción de un modelo se puede trasladar a 2D o 3D indistintamente ya que sólo depende del traductor TR que se utiliza para visualizar la escena. Con MINERVA y por su nivel de abstracción, no se pierde el modelado en árbol de la escena ya que como se ha indicado anteriormente, el árbol de la escena, similar a los de OGRE o Unity, es equivalente al árbol sintáctico de la cadena perteneciente al lenguaje L(V).

Además con MINERVA se pueden realizar renderizados no visuales, tales como sonidos, etc y los sistemas de Unity y OGRE no tienen en cuenta esta peculiaridad. Esto no significa que no se pueda introducir el sonido en implementaciones de Unity y OGRE, pero se modela como un elemento aparte del sistema de renderizado y no como una traducción de la misma descripción gráfica. La ventaja de que el sonido se derive de la propia descripción de la escena es que se asegura la coherencia entre el sonido y los elementos visuales de la escena.

Por otro lado, existen otros elementos no visuales que no se implementan en OGRE o en Unity, tales como la salida del sistema en un fichero para la grabación de los acontecimientos del sistema. Como en el caso del sonido, todos estos sistemas deben ser tratados como elementos añadidos al sistema de renderizado y no derivados de la propia descripción del sistema. MINERVA, sin embargo, sí contempla la posibilidad de definir traductores que generen cualquier tipo de salida no visual.

4. Comparación de los motores físicos

Existen implementaciones de motores físicos tanto en OGRE como en Unity. En OGRE existen tres motores principalmente OGRENewt, NxOGRE y OGREODE. Por otro lado, en Unity existen los módulos adicionales Jitter y Unity Physics.

En ambas plataformas, los motores físicos son excluyentes, es decir, se

trabaja con uno u otro y su implementación exige que todos los elementos de la escena sepan las características individuales del motor físico.

Para la gestión del motor físico se utilizan *callbacks*, donde se implementa la forma en que los elementos del entorno reaccionan ante los sucesos físicos. El problema es que los parámetros de las funciones *callbacks* tienen distintos tipos siendo la implementación parecida en funcionalidad pero distinta en forma.

Que tanto los motores de OGRE, como los de Unity sean tan distintos hace que se tenga que elegir entre uno de los motores y si se quiere cambiar a otro motor tendríamos que reescribir todo el sistema. Desde el punto de vista estratégico la elección de un motor u otro puede ser crucial; pensemos que, por ejemplo, algunos motores físicos para OGRE han sido abandonados. Sin embargo, con MINERVA esto no sucede al tener totalmente aislado al motor físico del resto de agentes del sistema.

MINERVA, por otro lado, utiliza un sistema de eventos para comunicar los sucesos relacionados con la física, como las colisiones, al resto de los elementos. Esto tiene varias ventajas. En primer lugar, la función que implementa la reacción tiene sólo como parámetro la lista de eventos a diferencia de los *callbacks* que dependiendo de a qué evento reaccionan pueden tener distintos parámetros. Con esta lista, el agente sólo tiene que buscar los eventos a los que quiere reaccionar. Además, en MINERVA estos eventos son de alto nivel y por tanto pueden ser tan complejos como se quiera. En segundo lugar, el agente no tiene porqué saber quien genera el evento y el motor físico no tiene porqué saber a quién está dirigido el evento. El agente reaccionará ante un evento si está preparado para ello, mientras que el motor físico sólo tiene que añadir a la lista de eventos las características físicas que desea comunicar a los demás elementos. Sin embargo, en los sistemas de *callbacks* implementados en OGRE y en Unity el motor físico sí que sabe a qué elementos debe llamar cuando hay un acontecimiento físico, existiendo una relación entre el motor y los elementos a los que afecta. El uso del sistema de eventos hace que la reutilización del agente que implementa el motor físico en otros sistemas sea muy sencilla, simplemente añadiendo el elemento correctamente desde el punto de vista de la sintaxis del lenguaje L(M).

Los tres motores físicos de OGRE son totalmente diferentes en implementación, pero similares en concepto (igual les pasa a los motores físicos de Unity). Están muy focalizados para espacios en 3D. Esto significa que tienen implementado un amplio conjunto de variables como la masa,

la inercia, la gravedad etc, que no se tienen porqué usar en todos los sistemas. Esto supone que pueden realizarse cálculos físicos que en muchos casos no tienen porque se ser usados.

MINERVA, por otro lado, se puede ajustar más a las necesidades del sistema que se está implementando sin necesidad de tener atributos como la masa, inercia etc, que no vamos a usar y que sí existirían aunque no la usásemos en los sistemas de OGRE.

Los motores físicos de OGRE son muy dependientes de las estructuras de OGRE, como bien indica su nombre. Esto supone que el sistema gráfico y el motor físicos están fuertemente acoplados. Sin embargo, en MINERVA esta relación existe pero su acoplamiento es débil. Por un lado, existe el lenguaje L(M) que define la evolución del sistema y donde se tratará la parte física y por otro lado, el motor gráfico se ejecuta gracias al lenguaje L(V). Es decir, sólo existe la función de visualización como elemento vinculante. Existe una relación pero no están entrelazados.

Otra característica interesante de MINERVA es que existe la posibilidad de poder incorporar motores físicos ya existentes. Es decir, los motores de OGRE podrían calcular las colisiones y en los *callbacks* añadir a las listas de eventos de MINERVA aquellos eventos que sean sinónimos. Para introducirlos en MINERVA, sólo se debe implementar un agente generador de eventos que trate el motor de OGRE y que luego pase los eventos generados a los demás agentes del modelo. Esto, también se puede realizar con Unity usando la misma técnica. El único problema que tiene Unity es que es un sistema muy cerrado y usa el lenguaje Mono como lenguaje de programación. Implementando MINERVA para este lenguaje podremos embeber Unity dentro del modelo MINERVA.

Otra de las características interesantes de MINERVA es que pueden coexistir varios motores físicos en una misma descripción del sistema. De esta forma se puede activar un motor u otro dependiendo del estado del sistema en un momento dado. Si, por ejemplo, ejecutamos el sistema con un motor físico implementado en una GPU, se puede activar el motor físico que utiliza esta característica, pero si no se tiene una GPU instalada con un motor físico se puede activar un motor software pero con menos precisión.

Tal vez, la principal desventaja que tenga MINERVA es que hay que hacer un esfuerzo importante a la hora de plantear el motor físico ya que además de implementar las leyes físicas se ha de hacer un esfuerzo en saber qué eventos va a generar el motor físico. Por otro lado, para que los agentes

reaccionen bien ante los eventos del motor se debe saber a priori qué eventos van a ser implementados por MINERVA.

5. Comparación de los motores de inteligencia artificial

Cada uno de los sistemas analizados tienen sus propios motores de inteligencia artificial. Ya se ha explicado que el motor de inteligencia artificial está íntimamente relacionado con el motor físico y éste con el motor gráfico.

En el caso de OGRE esta relación debe ser implementada y no existe ningún recurso propio de OGRE para poder modelar esta relación. Los motores de inteligencia artificial son librerías independientes al sistema que se diseñan como un añadido más y cuya integración con el motor físico se debe implementar en cada caso.

En el caso de Unity la inteligencia artificial de un componente se implementa mediante una máquina de estados. Para ello se definen gráficamente, mediante el editor de Unity, los distintos estados del elemento y cómo se pasa de un estado a otro. Después, en cada uno de los estados se pueden implementar diversas acciones. Sin embargo, tampoco es sencilla la integración entre los diferentes motores y su relación debe ser modelada en cada uno de los sistemas.

Sin embargo, en el caso de MINERVA, los motores de inteligencia artificial se especifican en la función de evolución de cada uno de los agentes. En esta función se puede implementar tanto una máquina de estados, como un sistema de lógica bayesiana o o cualquier otra técnica de inteligencia artificial y además esta especificación está muy localizada en las funciones de evolución.

En cuanto a la integración entre los distintos motores se puede decir que el propio lenguaje L(M) es un elemento integrador entre los diversos agentes del sistema. En concreto, la definición del ámbito de aplicación mediante los paréntesis y los eventos que se mueven en el sistema son elementos que nos ayudan a definir desde un punto de vista formal las relaciones existentes entre los distintos motores. Esto se puede ver en la definición de la cadena inicial del juego de carreras y en la cadena del sistema robótico. En estos ejemplos, se ve claramente que tanto los motores físicos en un ejemplo y los sensores en el otro, mandan los eventos

Universitat d'Alacant
Universidad de Alicante

oportunos a los agentes inteligentes del sistema para que ellos actúen dependiendo de su función de evolución.

6. Otros elementos

Además de la comparación de los motores anteriormente expuestos, existen otros elementos que es necesario comparar.

En el caso de Unity, por ejemplo, se dispone de un potente editor en el que se pueden diseñar todos los elementos de la escena de forma gráfica. Este editor es capaz de editar todos los aspectos de las escenas, pero al tener tantos parámetros que especificar su línea de aprendizaje es muy pronunciada y requiere, sobre todo al principio, un esfuerzo muy importante para poder implementar un sistema.

Tanto en el caso de OGRE como en el de MINERVA dicho editor no existe, pero hay que considerar que la fuerte formalización que tiene MINERVA hace que el desarrollo de un editor gráfico sea muy sencillo y fácil de implementar.

Por otro lado, MINERVA tiene una dificultad y es que su fuerte formalización hace que el cambio de paradigma sea importante para desarrolladores que ya estén familiarizados con sistemas de realidad virtual y por tanto ciertos conceptos o soluciones implementados con el lenguaje L(M) sean poco familiares.

Por otro lado, con MINERVA hay que hacer un esfuerzo importante en el diseño de los eventos, de las funciones de evolución y de los agentes que van a formar parte del sistema. Sin embargo, tener el sistema formalizado proporciona un idioma común que hace que cualquier persona que conozca la sencilla sintaxis de los lenguajes L(M) y L(V) pueda entender la solución modelada.

La popularidad de OGRE y Unity también ha hecho que los desarrollos de estos dos sistemas estén muy avanzados y con muchas funcionalidades que todavía no se han implementado con MINERVA. Aunque esta claro que esto depende del tiempo de desarrollo que se le dedique a MINERVA y no tanto del diseño propio de los lenguaje L(M) y (V).

En definitiva, se puede decir que OGRE y Unity tienen como ventaja que son sistemas conocidos y maduros que todo el mundo conoce, aunque no tengan una formalización clara de todos los elementos que forman parte de un sistema de realidad virtual. Sin embargo, MINERVA tiene esta

ventaja pero la desventaja de encontrarse en fase experimental y de requerir un cambio de paradigma a la hora de modelar los mundos virtuales.

7. Tabla resumen comparativa

A continuación, se resume en las tablas 11, 12 y 13 todas las características de los modelos analizados dividido en características del motor gráfico, del motor físico, del motor de inteligencia artificial y otras características adicionales.

Motor Gráfico			
Característica	**OGRE**	**Unity**	**MINERVA**
Espacio de representación	3D, 2D[1] VDE[2]	3D, 2D[3] VDE	3D, 2D VUE[4]
Librerías gráficas	OpenGL Direct3D	OpenGL Direct3D	Cualquiera[5]
Estructura de datos	Árbol de escena	Árbol de escena	Cadena de escena[6]
Sonido	NIM[7]	NIM	IM[8]
Otros renderers	NIM	NIM	IM

Tabla 11: Comparativa de los motores gráficos

Universitat d'Alacant
Universidad de Alicante

1 Como textura sobre el 3D.
2 Versión distinta del entorno. Doble desarrollo.
3 Versiones distintas del entorno.
4 Versión única del entorno con varias visualizaciones.
5 Requiere definir el TR correspondiente
6 El árbol semántico de la escena corresponde con el árbol de escena.
7 No integrado en el motor.
8 Integrado en el motor. Requiere definir el TR correspondiente.

Motor Físico			
Característica	**OGRE**	**Unity**	**MINERVA**
Motores disponibles	OGRENewt NxOGRE OGREODE (no integrados, no ampliable)	Jitter Unity Physics (no integrados, no ampliable)	Integrado en el modelo (requiere definir eventos y agentes) o uso de motores de terceros
Utilización conjunta	Excluyentes (sólo uno se puede utilizar a la vez)	Excluyentes (sólo uno se puede utilizar a la vez)	No excluyentes (se pueden utilizar varios con diferentes cometidos)
Tipo de comunicación	*Callbacks*	*Callbacks*	Eventos
Funciones implementadas	Muchas funciones genéricas, no ampliables	Muchas funciones genéricas, no ampliables	Funciones específicas para cada caso (requiere su implementación)
Acoplamiento con el motor gráfico	Alto	Alto	Bajo

Tabla 12: Comparativa de los motores físicos

Motor de inteligencia artificial			
Característica	**OGRE**	**Unity**	**MINERVA**
Motor integrado	No (posible utilizar motores de terceros, no integrados)	Sí (sólo incluye máquinas de estados)	Sí (integrado en el modelo, cualquier técnica de IA, requiere su implementación)

Tabla 13: Comparativa motores de inteligencia artificial

Universitat d'Alacant
Universidad de Alicante

Otros aspectos			
Característica	**OGRE**	**Unity**	**MINERVA**
Editor gráfico	No	Si	No (a desarrollar en el futuro)
Curva de aprendizaje	Pendiente media	Pendiente baja	Pendiente alta
Formalización	Baja	Baja	Alta

Tabla 14: Comparativa de otros aspectos

Universitat d'Alacant
Universidad de Alicante

VIII. Conclusiones y futuras líneas de investigación

En este capítulo analizamos si los objetivos establecidos al principio de la tesis se han cumplido con MINERVA. También sugerimos posibles líneas de investigaciones futuras que aporta el hecho de que MINERVA se haya diseñando como un modelo formal.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

1. Conclusiones

Con los tres ejemplos que se han desarrollado en la experimentación, hemos tratado de demostrar que MINERVA cumple con el principal objetivo de proporcionar un marco común para el desarrollo de diversos tipos de mundos virtuales. Es un modelo que tiene la virtud de relacionar áreas de investigación tan dispares como los sistemas de realidad virtual, los sistemas multiagentes, los videojuegos y las simulaciones, entre otras, a través del uso de cadenas de descripción con una sintaxis común determinada. Es cierto que existen varios trabajos que muestran ciertos puntos de contacto entre las áreas antes mencionadas, pero lo que MINERVA propone es un modelo formal matemático donde se demuestra de forma exhaustiva la relación entre dichas áreas.

Se ha demostrado como el traductor de renderizado es capaz de manejar un conjunto muy diverso de dispositivos de salida, con diferentes características, adaptándose a cada uno de ellos. Por ejemplo, podemos renderizar en 3D con el traductor de OpenGL y en 2D con el traductor de *sprites* con la misma cadena descriptiva. También es capaz de implementar traductores de renderizados no visuales tales como el del sonido, exportación a fichero, etc.

También estudiamos cómo podemos modelar un motor físico mediante agentes y cómo utiliza la descripción del mundo para ver el ámbito de aplicación de la simulación física mediante los paréntesis. Mediante eventos se especifican leyes físicas simuladas. El utilizar eventos para este fin hace que el origen del evento no sea importante y que podamos modelar un motor físico que utiliza hardware específico o, si este hardware no existe, podamos implementar un motor físico con software siempre y cuando genere los mismos eventos. Aquellos agentes que estén en el ámbito de aplicación del motor físico, especificado por el lenguaje formal con paréntesis y que reaccionen al evento serán los agentes que deberán actuar ante el fenómeno físico según lo que dicta su función de evolución.

Gracias a la función de evolución los agentes pueden tener comportamientos heterogéneos y además pueden ser implementados de forma totalmente diferente. Por otro lado, gracias a los eventos del motor

Universitat d'Alacant
Universidad de Alicante

físico recibidos se puede reaccionar de forma inteligente a los sucesos acontecidos en el mundo virtual.

La diversidad de los dispositivos de entrada queda resuelta en MINERVA gracias a que los eventos tienen un significado de alto nivel traduciendo las acciones de los dispositivos de entrada en otra clase de eventos con un significado más abstracto. Por ejemplo, en el caso del juego de carreras, en el agente de teclado, un evento de pulsado de tecla se traduce a un evento de aceleración del coche, mientras que en el agente que gestiona el joystick, la acción de empujar el mando hacia delante se traduce en el mismo evento de aceleración.

La reutilización es otro de los objetivos alcanzados gracias a que en MINERVA se utiliza un lenguaje común igual en todos los casos. Esto hace que un agente que se ha diseñado para un determinado sistema puede ser reutilizado en otro directamente y sin ningún cambio. El agente de la cámara que se diseñó en la experimentación es un ejemplo claro de cómo reutilizar un agente en MINERVA.

El uso de la Teoría de Lenguajes Formales para el diseño del modelo tiene como principal ventaja el considerar los anteriores sistemas desde otro punto de vista, estratificando en niveles de detalle, donde cada estrato tiene su propia función. Así, está la parte descriptiva representada por la cadena del sistema donde se puede observar de un vistazo los elementos que forman parte del sistema, así como la forma que tiene cada elemento de relacionarse con los demás. También, se puede ver la estructura jerárquica que existe entre los diferentes elementos del sistema. Por otro lado, están las funciones de evolución ejecutándose gracias al traductor de evolución del sistema. En este caso, lo que se detalla es cómo cada elemento evoluciona en el tiempo y a qué eventos del sistema están reaccionando según un patrón de diseño muy característico. Y en un último estrato, se han de tener en cuenta las funciones de traducción de visualización que representan la parte del sistema que muestra en dispositivos visuales o no visuales los estados del sistema en cada instante.

El hecho que estar tan estratificado hace que se pueda observar el sistema desde varios puntos de vista. En primer lugar, se puede ver el sistema desde un punto de vista puramente descriptivo realizando operaciones con las cadenas derivadas del lenguaje L(M). Simplemente manipulando cadenas se pueden describir múltiples escenarios. Además, existe un lenguaje común para la descripción del sistema. Tener este lenguaje común hace que las descripciones en forma de cadenas pueden ser

analizadas por personas no especializadas en sistemas de realidad virtual siempre y cuando conozcan el valor semántico del lenguaje, un lenguaje que por otro lado está libre de ambigüedades. En segundo lugar, el modelo se puede ver desde el punto de vista de las funciones de evolución de forma que es posible analizar la forma que tiene cada uno de los elementos de relacionarse con el entorno mediante los eventos y cómo los agentes modifican su entorno y su estado cuando reciben los eventos. En tercer lugar, se puede ver el sistema cómo una salida que representa su estado. Así, se puede ver su salida como una visualización gráfica, como una mera descripción de las cadenas o mediante un conjunto de comandos sobre una máquina real.

Por otro lado, utilizar una teoría tan establecida como la teoría de lenguajes formales y adaptarla a sistemas tan diferentes como los anteriormente enumerados, hace que se puedan trasladar muchos de los resultados de investigación de esta teoría a otras áreas como las de realidad virtual. Por ejemplo, el uso de cadenas descriptivas nos permite hacer preguntas de cómo va a funcionar el sistema simplemente manipulando las cadenas y estudiando qué significado semántico tienen estas cadenas para la ejecución del sistema.

Tal vez la característica más valiosa que tiene MINERVA es precisamente el hecho de que sea un modelo formal. Este tipo de modelos tienen ciertas propiedades que pueden ser aplicadas a los sistemas desarrollados con MINERVA. Así, se puede definir un conjunto concreto de cadenas que forman parte del modelo proporcionado una herramienta para verificar si dada una cadena ésta pertenece a las cadenas bien formadas y por tanto correctas para el sistema diseñado. Además, los modelos formales tienen un conjunto de reglas bien definidas para derivar de una cadena a otra y que por tanto se puede sistematizar la generación de cadenas. Desde el punto de vista matemático, el hecho de que se tenga un lenguaje formal hace que pueda demostrarse la corrección del sistema y la completitud del modelo definido. Esto hace que se puede establecer una demostración de que la implementación del sistema corresponde con la especificación.

Otra de las propiedades que se puede constatar con la experimentación expuesta es el hecho de que los elementos del sistema, al utilizar los mismos recursos para su creación, se pueden reutilizar en otros entornos. Es decir, todos los elementos del sistema tanto del alfabeto M como del alfabeto V son totalmente reutilizables ya que al utilizar un mismo modelo formal son por tanto elementos del mismo modelo formal. Por ejemplo,

un agente desarrollado para un cierto entorno puede ser utilizado en otro de forma automática y verificar su eficacia en este nuevo entorno, entendido aquí entorno como la distintas composiciones de cadenas de las que el agente puede formar parte. Por otro lado, los eventos al tener significado de alto nivel pueden ser también reutilizados en distintos entornos siempre y cuando sean generados por unos agentes y aceptados por otros. Todo esta reutilización se realiza gracias al hecho de que en el fondo se está utilizando el mismo modelo formal que describe de forma total, coherente y sin ambigüedades todos los elementos del sistema.

La homogeneización de la diversidad de dispositivos que los sistemas planteados despliegan es otra de las ventajas que MINERVA tiene como punto fuerte. Esta homogeneización se realiza gracias a los dos alfabetos que forman parte del sistema y que encapsulan dentro de sus funciones los detalles de los dispositivos, tanto de entrada como de salida. Esto tiene como ventaja significativa que se pueda añadir en la misma cadena descriptiva del sistema el conjunto de dispositivos de entrada y la descripción de su relación con los demás elementos. Por otro lado, los traductores TR tienen como principal función establecer una homogeneización para un conjunto cada vez más diverso de dispositivos de salida con muy distintas características tanto desde el punto de vista de potencia gráfica como desde el punto de vista de su naturaleza (es decir, dispositivos tales como los sonoros).

Universitat d'Alacant
Universidad de Alicante

Otro aspecto fundamental de MINERVA es que en cierta medida nos proporciona un paradigma distinto para el desarrollo de sistemas tales como los de realidad virtual, sistemas multiagentes o sistemas robóticos. Sistemas por otro lado con mucha tradición y con ciertos paradigmas ya muy consolidados. MINERVA es otra forma de verlo y por tanto requiere el esfuerzo adicional que representa el cambio de paradigma. Elementos tales como eventos de alto nivel, funciones de evolución y traducciones a un lenguaje descriptivo visual son elementos que ya existían en otros paradigmas pero que en MINERVA se establecen desde un punto de vista formal e integrador.

También el hecho de que MINERVA sea formal exige del diseñador que lo está aplicando, una habilidad importante al tratar con altos niveles de abstracción. Por ejemplo, diseñar eventos que no están pegados a los eventos de dispositivos requiere del diseñador un conocimiento importante de cómo se va a desarrollar el sistema ya no desde el punto de vista instrumental (pegado a los dispositivos de entrada) sino desde un punto de

vista puramente funcional. Este formalismo hace también que ciertas soluciones planteadas con el modelo resulten extrañas porque no se asemejan en apariencia a otras soluciones basadas en paradigmas ya consolidados en las áreas en que se está aplicando MINERVA. Sin embargo, se abren otras vías de desarrollo que tal vez no estén exploradas como por ejemplo el utilizar elementos de videojuegos en sistemas robóticos.

En resumen, se puede decir que MINERVA tiene un conjunto de ventajas que son muy interesantes, pero que exigirán del desarrollador que lo use de un cambio de paradigma. Los resultados parecen, a la vista de la experimentación, prometedores y por tanto parece interesante este cambio de paradigma.

2. Artículos publicados

A continuación se especifican los artículos publicados resultado del estudio del modelo MINERVA.

Título:	Construction of Intelligent Virtual Worlds Using a Grammatical Framework
Revista:	International Journal of Intelligent Systems, 29: 751–766. ISSN: 1098-111X, doi: 10.1002/int.21661, 2014.
Año:	2014
Autores:	Gabriel López García, Rafael Molina Carmona, Antonio Javier Gallego y Patricia Compañ-Rosique
Abstract	
The potential of integrating multiagent systems and virtual environments has not been exploited to its whole extent. This paper proposes a model based on grammars, called Minerva, to construct complex virtual environments that integrate the features of agents. A virtual world is described as a set of dynamic and static elements. The static part is represented by a sequence of primitives and transformations and the dynamic elements by a series of agents. Agent activation and communication is achieved using events, created by the so-called event generators. The grammar defines a descriptive language with a simple syntax and a semantics, defined by functions. The semantics functions allow the scene to be displayed in a graphics device, and the description of the activities of the agents, including artificial intelligence algorithms and reactions to physical phenomena. To illustrate the use of Minerva, a practical example is presented: a simple robot simulator that considers the basic features of a typical robot. The result is a functional simple simulator. Minerva is a reusable, integral, and generic system, which can be easily scaled, adapted, and improved. The description of the virtual scene is	

Universitat d'Alacant
Universidad de Alicante

independent from its representation and the elements that it interacts with.

Título:	A Grammatical Approach to the Modeling of an Autonomous Robot.
Revista:	International Journal of Interactive Multimedia and Artificial Intelligence, ISSN: 1989-1660, pp. 30-37, Vol. 1, 2012.
Año:	2012
Autores:	Gabriel López-García, A. Javier Gallego-Sánchez, J. Luis Dalmau-Espert, Rafael Molina-Carmona and Patricia Compañ-Rosique.
Abstract	
Virtual Worlds Generator is a grammatical model that is proposed to define virtual worlds. It integrates the diversity of sensors and interaction devices, multimodality and a virtual simulation system. Its grammar allows the definition and abstraction in symbols strings of the scenes of the virtual world, independently of the hardware that is used to represent the world or to interact with it. A case study is presented to explain how to use the proposed model to formalize a robot navigation system with multimodal perception and a hybrid control scheme of the robot. The result is an instance of the model grammar that implements the robotic system and is independent of the sensing devices used for perception and interaction. As a conclusion the Virtual Worlds Generator adds value in the simulation of virtual worlds since the definition can be done formally and independently of the peculiarities of the supporting devices.	

Título:	A Grammatical Approach to the Modeling of an Autonomous Robot.
Congreso:	Int. Symp. on Distributed Computing and Artificial Intelligence (DCAI 2012), Salamanca, Spain (2012).
Año:	2012
Autores:	Gabriel López-García, A. Javier Gallego-Sánchez, J. Luis Dalmau-Espert, Rafael Molina-Carmona and Patricia Compañ-Rosique.
Abstract	
Virtual Worlds Generator is a grammatical model that is proposed to define virtual worlds. It integrates the diversity of sensors and interaction devices, multimodality and a virtual simulation system. Its grammar allows the definition and abstraction in symbols strings of the scenes of the virtual world, independently of the hardware that is used to	

Universitat d'Alacant
Universidad de Alicante

represent the world or to interact with it. A case study is presented to explain how to use the proposed model to formalize a robot navigation system with multimodal perception and a hybrid control scheme of the robot. The result is an instance of the model grammar that implements the robotic system and is independent of the sensing devices used for perception and interaction. As a conclusion the Virtual Worlds Generator adds value in the simulation of virtual worlds since the definition can be done formally and independently of the peculiarities of the supporting devices.

Título:	Formal model to integrate Multi-Agent Systems and Interactive Graphic Systems.
Congreso:	International Conference on Agents and Artificial Intelligence, Valencia, Spain (2010). ISBN: 978-989-674-022-1, pp. 264-267,
Año:	2010
Autores:	Gabriel López García, Rafael Molina Carmona y Antonio Javier Gallego Sánchez.

Abstract

Virtual Worlds Generator is a grammatical model that is proposed to define virtual worlds. It integrates the diversity of sensors and interaction devices, multimodality and a virtual simulation system. Its grammar allows the definition and abstraction in symbols strings of the scenes of the virtual world, independently of the hardware that is used to represent the world or to interact with it.
A case study is presented to explain how to use the proposed model to formalize a robot navigation system with multimodal perception and a hybrid control scheme of the robot.

Título:	Hacia un modelo integral para la generación de Mundos Virtuales
Congreso:	Artículo CEIG
Año:	2008
Autores:	Gabriel López García, Rafael Molina Carmona y Antonio Javier Gallego

Abstract

Uno de los problemas más importantes en los sistemas de Realidad Virtual es la diversidad de los dispositivos visuales y de interacción que existen en la actualidad. Junto a esto, la heterogeneidad de los motores gráficos, los motores físicos y los módulos de Inteligencia Artificial, propicia que no exista un modelo que aúne todos estos aspectos de una forma integral y coherente. Con el objetivo de unificar toda esta diversidad, presentamos un modelo formal que afronta de forma integral el problema de la

Universitat d'Alacant
Universidad de Alicante

diversidad en los sistemas de RV, así como la definición de los módulos principales que los constituyen.

El modelo propuesto se basa en la definición de una gramática, que integra la actividad necesaria en un sistema de RV, su visualización y su interacción con el usuario. La descripción de un mundo se presenta como una secuencia ordenada de primitivas, transformaciones que modifican el comportamiento de las primitivas y actores que definen la actividad del sistema. Los conceptos de primitiva, transformación y actor son mucho más amplios de lo que es habitual en estos sistemas: Las primitivas no son simples primitivas de dibujo sino acciones que se deben ejecutar en un determinado sistema de presentación, gráfico o no; las transformaciones modifican las primitivas, dependiendo de su naturaleza; los actores desarrollan una o varias actividades en el mundo virtual, se visualizan mediante primitivas y transformaciones, y usan eventos que también se definen en sentido amplio. El modelo presentado tiene como virtud la separación de la actividad del sistema de los dispositivos visuales y de interacción concretos que lo componen. Esto supone varias ventajas: los dispositivos pueden ser sustituidos por otros dispositivos o por simulaciones de estos, los elementos del sistema pueden ser reutilizados, y la representación gráfica puede ser diferente dependiendo del dispositivo visual.

En definitiva, se ha pretendido diseñar un sistema integral, adaptativo, reutilizable y genérico.

Por último se presenta un caso práctico que permite concretar cómo se utiliza el modelo propuesto.

3. Futuras lineas de investigación

El esfuerzo por desarrollar un modelo común para los sistemas tratados hace que se queden en el tintero ciertas líneas de investigación. Al fin y al cabo lo que se pretende con la definición de MINERVA es el diseño de un marco común para que los investigadores comencemos a hacernos otro tipo de preguntas.

3.1. Paralelización de MINERVA

MINERVA está dividido en dos tipos principales de elementos. Los primeros son los elementos descriptivos con los que se puede diseñar la representación del sistema que se está implementado. Para este cometido se tienen las primitivas que definen lo que se va a representar, transformaciones que modelan la forma en que se modifican las primitivas, y los agentes y eventos que articulan toda la interacción del sistema.

La segunda parte del modelo consiste en un conjunto de traductores que transforman la descripción del modelo a lo largo del tiempo para ver primero cómo evoluciona y después cómo se puede visualizar el estado actual del sistema. Se trata de autómatas finitos que una vez diseñados no van a ser modificados y pueden ser reutilizados por todos los mundos virtuales modelados con del sistema formal.

Los traductores, por lo tanto, son un elemento común que está en continua ejecución, por lo que una optimización sobre ellos impacta directamente sobre todos los sistemas que se hayan desarrollado con el modelo formal.

En los últimos años ha habido un avance bastante importante en la tecnologías de paralelización. Desde la aparición de los procesadores con varios núcleos hasta el uso de las tarjetas gráficas para el procesamiento paralelo con librerías como CUDA [NVidia 2014] o OpenCL [Kronos 2014] ha aumentado el interés en paralelizar algoritmos y patrones de diseño en general. Por esta razón, vamos a analizar brevemente los puntos de MINERVA que es posible paralelizar.

El uso de cadenas en el modelo y de una gramática que las procesa hace que la paralelización sea más sencilla. Efectivamente, desde el punto de vista del traductor se puede dividir una cadena en partes y procesar las subcadenas en procesadores distintos y así paralelizar el proceso de traducción. El problema en la paralelización de los traductores consiste en que no podemos partir la cadena descriptiva del lenguaje por cualquier lugar.

Por ejemplo, en la Figura 40, las flechas azules son lugares donde se puede dividir la cadena inicial y pasar las partes de la izquierda y la derecha a diferentes procesadores para ejecutarlas en paralelo. Sin embargo, la flecha roja indica un punto en el que la separación hace que rompa en dos una expresión de transformación en la que se indica que todos los elementos que están entre paréntesis están afectados por la transformación.

$$p_1 \cdot t\left(p_2 \cdot p_3\right) \cdot a^d_{1\langle attr\rangle}\left(p_4 \cdot p_5\right) \cdot a^d_{2\langle attr\rangle}$$

Figura 40: Paralelización de procesamiento de cadenas

Universitat d'Alacant
Universidad de Alicante

Con este ejemplo se ve que los paréntesis de la gramática son zonas delicadas para el procesamiento paralelo. Pero esto, a su vez, es una ventaja. Efectivamente, al saber dónde están los puntos delicados para la paralelización e identificarlos mediante los paréntesis, éstos pueden servir para separar la cadena en las subcadenas contenidas entre los paréntesis y paralelizar en diferentes procesadores cada subcadena.

Por tanto, podemos separar las cadenas de la siguiente forma:

- Clasificar los elementos, diferenciando aquellos que tienen paréntesis de los que no lo tienen.
- De los elementos que tienen paréntesis se extrae la cadena dentro del paréntesis y se pasa al punto 1.
- De los elementos que no tienen paréntesis se separan en tantas partes como sean necesarias para que la paralelización sea eficiente.

De esta manera se modela un traductor cuyos elementos pueden ejecutarse en paralelo.

Pero dentro de MINERVA se utilizan tres traductores que entre ellos también puede ser ejecutados de forma paralela. Así, y viendo el algoritmo del sistema, el traductor TE puede ser ejecutado en paralelo con los traductores TV y TR. Sin embargo, estos dos traductores deben ser ejecutados en serie porque la salida de TV es la entrada de TR.

La paralelización de MINERVA es interesante entre otras cosas porque un esfuerzo en la optimización de los traductores impacta directamente en la velocidad en todos los sistemas que estén modelados por MINERVA. Es lo importante que tiene modelar un sistema de forma tan abstracta, que cualquier optimización que se haga en la parte de la abstracción generará un impacto global en la velocidad del sistema.

Sin embargo, dentro de la paralelización del sistema existen muchos puntos a investigar como por ejemplo el uso de GPU y cómo utilizar la potencia de estos procesadores para el modelo. Por otro lado, existe la posibilidad de buscar modelos de funciones de evolución que pueden usar internamente técnicas de paralelización. En definitiva, varias preguntas que podrían ser investigadas en un futuro.

3.2. Nuevas líneas de investigación que abre el modelo formal

El hecho de usar un modelo formal hace que podamos cuestionarnos ciertas cosas que desde otro punto de vista no sería fácil plantear.

Tras la formulación de MINERVA, se planteó qué significaban ciertas manipulaciones de los elementos del modelo. Una de las cuestiones que podemos plantearnos es la siguiente: dado que el traductor TR define una función de traducción f_{out}, ¿qué significa su función inversa?

Recordando que el traductor TR era el encargado de traducir las cadenas del lenguaje L(V) a instrucciones de los dispositivos de salida y considerando que el dispositivo de salida sea una imagen, se puede asumir que la inversa de la función de traducción nos permitiría obtener, dada una imagen, la descripción de esta imagen en forma de elementos del lenguaje L(V). Es decir, se puede plantear desde un punto de vista formal el problema de la visión artificial.

En primer lugar, se podría demostrar que esta función inversa del traductor TR no es una función biyectiva y que por tanto para toda imagen no existe una única cadena que describa la escena. Esto parece que es así, pero gracias a este planteamiento con MINERVA se puede estudiar qué tipo de función sería el traductor TR desde el punto de vista algebraico. Sin ir más allá de este estudio se puede formular la hipótesis de que al menos la función del traductor TR no es una función biyectiva y que, por tanto, se puede deducir el porqué de las dificultades de la visión artificial.

En segundo lugar y dando como cierta la hipótesis anterior, se podría plantear algún tipo de solución algorítmica para el problema, por ejemplo, un algoritmo genético que parta de una población inicial de cadenas. En este caso, se podrían traducir las cadenas a imágenes con el traductor TR de MINERVA y comparar la imagen generada con la imagen origen visionada según algún criterio de similitud. Aquellas cadenas que tengan mejores similitudes según algún criterio pueden pasar a la siguiente generación y descartaríamos las que tuvieran peores valores. Después, como establecen los principios de los algoritmos genéticos, modificaríamos de forma aleatoria ciertas partes de las cadenas que forman parte de la población y generaríamos otras cadenas nuevas hasta encontrar alguna cadena que cumpla con un cierto nivel de similitud. El estudio de este algoritmo de

Universitat d'Alacant
Universidad de Alicante

visión artificial puede plantear una interesante línea de investigación a realizar en el futuro.

En tercer lugar, se podría estudiar qué tipo de símbolos son los más adecuados para interpretar una imagen en una cadena descriptiva considerando el algoritmo de comparación explicado anteriormente. Se podría estudiar la conveniencia de usar símbolos de tipo general como primitivas geométricas (triángulos, cuadrados, cubos) y transformaciones básicas (escalado, traslación, rotación) o se podría plantear símbolos un poco más concretos y con una carga informativa mayor (silla, mesa, pantalla).

En cuarto lugar se podrían encadenar más traducciones una vez que se realiza la traducción inversa de TR. Por ejemplo, se podrían usar en una primera instancia símbolos muy generales para luego realizar otras traducciones inversas de símbolos generales a símbolos más concretos y así sucesivamente hasta un nivel descriptivo dado.

Como se puede comprobar simplemente estudiando el modelo desde un punto de vista formal se pueden ver los problemas existentes, como la visión artificial, desde otro punto de vista. El cambio de punto de vista hace emerger nuevos caminos. El hecho de que se emplee la teoría de lenguajes formales permite trasladar resultados de esta teoría a los sistemas modelados por MINERVA. Por ejemplo, dada una cadena descriptiva de un sistema de realidad virtual podríamos preguntarnos cuál es la semántica de esta cadena o si existen simplificaciones de la misma en las que no se pierda nada de las especificaciones del sistema. Por otro lado, si existen dos cadenas que describen dos sistemas podemos preguntarnos qué significa la concatenación de estas cadenas. En definitiva, el hecho de que MINERVA sea un sistema formal basado en la teoría de lenguajes formales establece una relación entre esta teoría y los sistemas de realidad virtual, sistemas multiagentes y sistemas de simulación pudiendo trasladar resultados de dicha teoría a estos sistemas.

Es MINERVA por tanto un lenguaje común para que los investigadores puedan resolver problemas actuales desde otro punto de vista con ventajas a la hora de implementar fácilmente sistemas de realidad virtual y con una capacidad de reutilización a considerar.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

IX. Bibliografía

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

ADRIAN BOEING. 2014. Physics Abstraction Layer. http://www.adrianboeing.com/pal/index.html. Último acceso 27/08/2014

AHO, A. V., SETHI, R., AND ULLMAN, J.D. 1990. *Compiladores. Principios, técnicas y herramientas*. Addison-Wesley Iberoamerica, S.A, Wilmington, Delaware, E.U.a.

AMD. 2014. Bullet Physics Library. http://bulletphysics.org/wordpress. Último acceso 27/08/2014.

AXELROD, R. 2003. Advancing the Art of Simulation in the Social Sciences Robert Axelrod. 1–19.

BAR-ZEEV, A. 2003. Scenegraphs: Past, present and future. http://www.realityprime.com/blog/2007/06/scenegraphs-past-present-and-future. Último acceso 27/08/2014.

BATES, J. 1992. Virtual reality, art, and entertainment. *PRESENCE Teleoperators and Virtual Environments 1*, 1, 133–138.

BENKO, H., WILSON, A.D., AND BAUDISCH, P. 2006. Precise selection techniques for multi-touch screens. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, 1263–1272.

BERTALANFFY, L. VON. 1986. *Teoría General de Sistemas*.

BUCKWALD, M. AND HOLZ, D. 2014. LEAP Motion. https://www.leapmotion.com/. Último acceso 27/08/2014.

BURNS, D. AND OSFIELD, R. 2014. OpenSceneGraph. http://www.openscenegraph.org. Último acceso 27/08/2014

Universitat d'Alacant
Universidad de Alicante

CAIRE, F., POGGI, A., AND RIMASSA, G. 2003. JADE. A white paper. *3*, September, 6–19.

CAVAZZA, M., LUGRIN, J.-L., HARTLEY, S., ET AL. 2005. Intelligent virtual environments for virtual reality art. *Computers & Graphics 29*, 6, 852–861.

CILIB. 2014. CILib - Computational Intelligence Library. http://www.cilib.net. Último acceso 27/08/2014.

COOK, P.R. 2002. Sound production and modeling. *IEEE Computer Graphics and Applications 22*, 4, 23–27.

DIRAC, P.A.M. 1934. Discussion of the infinite distribution of electrons in the theory of the positron. *Mathematical Proceedings of the Cambridge Philosophical Society 30*, 02, 150–163.

DOLLNER, J. AND HINRICHS, K. 2002. A generic rendering system. *IEEE Transactions on Visualization and Computer Graphics 8*, 2, 99–118.

EBERLY, D. 2003. Game physics. .

EDWARD, L., LOURDEAUX, D., AND LENNE, D. 2008. Modelling autonomous virtual agent behaviours in a virtual environment for risk. ... *Journal of Virtual ... 7*, 3, 13–22.

EPIC GAMES. 2014. Unreal Engine Kit. https://www.unrealengine.com/. Último acceso 27/08/2014.

FÜNFZIG, C., ULLRICH, T., AND FELLNER, D.W. 2006. Hierarchical spherical distance fields for collision detection. *IEEE computer graphics and applications 26*, 1, 64–74.

FUNGE, J., TU, X., AND TERZOPOULOS, D. 1999. Cognitive Modeling: Knowledge, Reasoning and Planning for Intelligent Characters. *99*, 29–

38.

GEBHARDT, N. 2014. Irrlicht Engine. http://irrlicht.sourceforge.net/.

GEMROT, J., KADLEC, R., BÍDA, M., AND BURKERT, O. 2009. Pogamut 3 can assist developers in building AI (not only) for their videogame agents. *Agents for Games and ...*, 1–15.

GIBSON, J.J. 1979. The ecological approach to visual perception. .

GILBERT, N. 2007. Agent-based models. 1–20.

GILBERT, N. AND TERNA, P. 2000. How to build and use agent-based models in social science. *Mind & Society*.

GORMAN, B. AND THURAU, C. 2006. Bayesian imitation of human behavior in interactive computer games. ... *, 2006. ICPR 2006. ...*, 18–21.

HAVOK. 2014. Havok. http://www.havok.com. Último acceso 27/08/2014.

HIBBARD, B. 2004. The top five problems that motivated my work. *Computer Graphics and Applications, IEEE*, 9–13.

E.HOPCROFT, J., MOTWANE, R., AND ULLMAN, J.D. 2002. *Introducción a la Teoría de Autómatas, Lenguajes y Computación*. Pearson Educación, Madrid.

JADE. 2014. JADE. http://jade.tilab.com. Último acceso 27/08/2014.

JAMES, D.L. AND PAI, D.K. 2004. BD-tree: output-sensitive collision detection for reduced deformable models. *ACM SIGGRAPH 2004 Papers*, ACM, 393–398.

JULIER, S., HOCHSTRATE, J., AND KULIK, A. 2006. On 3D Input Devices.

Universitat d'Alacant
Universidad de Alicante

15–19.

JUNKER, G. 2006. *Pro OGRE 3D programming*. Apress.

KHOO, A. AND ZUBEK, R. 2002. Applying inexpensive AI techniques to computer games. *IEEE Intelligent Systems 17*, 4, 48–53.

KIM, J. AND GRACANIN, D. 2009. iPhone/iPod touch as input devices for navigation in immersive virtual environments. *Virtual Reality ...* July 2007, 261–262.

KINECT. 2014. Kinect. http://es.wikipedia.org/wiki/Kinect. Último acceso 27/08/2014.

KRONOS. 2014. OpenGL. http://www.opengl.org/. Último acceso 27/08/2014.

LAIRD, J. 2001. Using a computer game to develop advanced AI. *Computer*.

LEMERCIER, S., JELIC, A., KULPA, R., ET AL. 2012. Realistic following behaviors for crowd simulation. *Computer Graphics Forum 31*, 2pt2, 489–498.

LINDENMAYER, A., PRUSINKIEWICZ, P., AND HANAN, J. 1988. Development models of herbaceous plants for computer imagery purposes. *ACM SIGGRAPH Computer Graphics*, 141–150.

LOKKI, T., SAVIOJA, L., VAANANEN, R., HUOPANIEMI, J., AND TAKALA, T. 2002. Creating interactive virtual auditory environments. *IEEE Computer Graphics and Applications 22*, 4, 49–57.

LOMBARDO, J., CANI, M., AND NEYRET, F. 1999. Real-time collision detection for virtual surgery. *Computer Animation, 1999. ...* 1999, 82–90.

Universitat d'Alacant
Universidad de Alicante

LUKE, S. AND CIOFFI-REVILLA, C. 2004. Mason: A new multi-agent simulation toolkit. *Proceedings of the 2004 ....*

MAES, P. 1995. Artificial life meets entertainment: lifelike autonomous agents. *Communications of the ACM 38*, 11.

MAES, P. AND DARRELL, T. 1995. The ALIVE system: Full-body interaction with autonomous agents. *Computer Animation'95., ... 95*, 11–18,.

MANDELBROT, B.B. 1983. The fractal geometry of nature/Revised and enlarged edition. *New York, WH Freeman and Co., 1983, 495 p. 1.*

MEALY, G. 1955. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal 34*, 5, 1045 – 1079.

MERELO, J.J. 2014. Evolutionary Computation Framework. http://eodev.sourceforge.net. Último acceso 27/08/2014.

MICROSOFT. 2014. Direct 3D. http://es.wikipedia.org/wiki/Direct3D. Último acceso 27/08/2014.

MILES, C. AND QUIROZ, J. 2007. Co-evolving influence map tree based strategy game players. ... *Intelligence and Games, ...* Cig, 88–95.

MILLER, J.H. AND PAGE, S.E. 2009. *Complex Adaptive Systems: An Introduction to Computational Models of Social Life: An Introduction to Computational Models of Social Life*. Princeton University Press.

MOORE, M., WILHELMS, J., GRAPHICS, C., BOARD, I.S., AND CRUZ, S. 1988. Collision Detection and Response for Computer Animation. *22*, 4, 289–298.

MSC SOFTWARE. 2014. Working Model. http://en.wikipedia.org/wiki/Working_Model. Último acceso

27/08/2014.

MURPHY, J. 2009. The State of Machine Learning and Artificial Intelligence. 1–9.

NINTENDO. 2014. Wii. http://www.nintendo.com/es_LA/wiiu . Último acceso 27/08/2014.

NVIDIA. 2014. CUDA. http://www.nvidia.com/object/cuda_home_new.html. Último acceso 27/08/2014.

OGRE. 2014. OGRE. http://www.ogre3d.org. Último acceso 27/08/2014.

OKA, K., SATO, Y., AND KOIKE, H. 2002. Real-time fingertip tracking and gesture recognition. *Computer Graphics and Applications, ...* December, 64–71.

OTADUY, M. A., CHASSOT, O., STEINEMANN, D., AND GROSS, M. 2007. Balanced Hierarchies for Collision Detection between Fracturing Objects. *2007 IEEE Virtual Reality Conference*, 83–90.

PARIS, S. AND DONIKIAN, S. 2009. Activity-driven populace: a cognitive approach to crowd simulation. *IEEE computer graphics and applications 29*, 4, 34–43.

PHYXS. 2014. Phyxs NVidia. http://www.nvidia.es/object/physx_new_es.html. Último acceso 27/08/2014.

POV-RAY. 2014. POV-Ray. http://www.povray.org. Último acceso 27/08/2014.

REYNOLDS, C. 1987. Flocks, herds and schools: A distributed behavioral model. *ACM SIGGRAPH Computer Graphics 21*, 4, 25–34.

RHYNE, T. 2000. Computer games' influence on scientific and information visualization. *Computer*, 154–156.

RICHARDS, D., JACOBSON, M., AND PORTE, J. 2012. Evaluating the models and behaviour of 3D intelligent virtual animals in a predator-prey relationship. *Proceedings of the 11th ...* June, 4–8.

RUSSELL SMITH. 2014. Open Dynamics Engine. http://www.ode.org. Último acceso 27/08/2014.

RYALL, K., ESENTHER, A., AND FORLINES, C. 2006. Identity-differentiating widgets for multiuser interactive surfaces. *Computer Graphics ...* October, 56–64.

SONY. 2014. PlayStation3 Move. http://us.playstation.com/ps3/playstation-move. Último acceso 27/08/2014.

SRIBOONRUANG, Y. 2006. Visual Hand Gesture Interface for Computer Board Game Control. *... , 2006. ISCE'06. 2006 ...* 1, 1–5.

STEUER, J. 2006. Defining virtual reality: Dimensions determining telepresence. *Journal of communication*, 1–25.

STRAUSS, P. AND CAREY, R. 1992. An object-oriented 3D graphics toolkit. *ACM SIGGRAPH Computer Graphics* July 1992.

SUTHERLAND, I.E. 1963. Sketchpad: A man-machine graphical communication system. *Afips Conference Proceedings 23*, 574, 329–346.

TECHNOLOGIES, U. 2014. Unity. http://www.unity3d.com. Último acceso 27/08/2014.

TESCHNER, M., HEIDELBERGER, B., MÜLLER, M., POMERANETS, D., AND GROSS, M. 2003. Optimized spatial hashing for collision detection of

Universitat d'Alacant
Universidad de Alicante

deformable objects. *Proceedings of Vision, Modeling, Visualization VMV'03*, Citeseer, 47–54.

TISUE, S. AND WILENSKY, U. 2004. Netlogo: A simple environment for modeling complexity. *International Conference on ...*, 1–10.

TRESCAK, T., ESTEVA, M., AND RODRIGUEZ, I. 2010. A virtual world grammar for automatic generation of virtual worlds. *The Visual Computer 26*, 6-8, 521–531.

TURING, A.M. 1950. Computing machinery and intelligence. *Mind 59*, 236, 433–460.

ULLRICH, T., FUNFZIG, C., AND FELLNER, D. 2007. Two different views on collision detection. *IEEE Potentials 26*, 1, 26–30.

UNITY. 2014. Unity. http://www.unity3d.com. Último acceso 27/08/2014.

X3D. 2014. X3D. http://www.web3d.org/x3d. Último acceso 27/08/2014.

YANNAKAKIS, G. N., LEVINE, J., & HALLAM, J. (2004, June). An evolutionary approach for interactive computer games. In Evolutionary Computation, 2004. CEC2004. Congress on (Vol. 1, pp. 986-993). IEEE.

Universitat d'Alacant
Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

Universitat d’Alacant
Universidad de Alicante