Hacia un modelo integral para la generación de Mundos Virtuales

Gabriel López García, Rafael Molina Carmona y
Antonio Javier Gallego Sánchez
Grupo de Informática Industrial e Inteligencia Artificial
Universidad de Alicante, Ap.99, E-03080, Alicante, Spain
{glopez, rmolina, ajgallego}@dccia.ua.es

18 de julio de 2014

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 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 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.

1. Introducción

El desarrollo en las últimas décadas de los sistemas de Realidad Virtual (RV) y de los sistemas gráficos en general ha sido espectacular. Este progreso ha contribuido a que la información de nuestro alrededor se presente mediante gráficas, animaciones o simulaciones, dando a conocer nuevas formas de análisis. Los videojuegos constituyen hoy en día una de las industrias más pujantes y cada día nuevos desarrollos de Realidad Virtual aparecen a nuestro alrededor. Sin embargo, esto no significa que todo esté hecho ni mucho menos, sino que la situación actual hace que los retos que se presentan sean aún más estimulantes.

Uno de los problemas más acuciantes es la diversidad de los dispositivos visuales. Se tiene toda una variedad de dispositivos hardware que generan imágenes (monitores, teléfonos móviles, PDAs, gafas de realidad virtual) y la visualización debe adaptarse a cada uno de ellos según sus características particulares (resolución, tamaño, etc.).

Mientras que todo lo referente a la visualización ha alcanzado una cierta madurez en las pasadas décadas, en el campo de la interacción no se ha producido la evolución deseada [JS04, DJK05]. Sigue dominando el teclado y el ratón en los interfaces de usuario, los mandos con libertad de movimiento son complejos de manejar, y otras posibilidades de interacción se quedan en el mundo de la investigación y el arte gráfico [JS04]. Si que es cierto, que en los últimos años ha habido tímidos avances. Entre ellos se puede destacar el mando de la Wii.

Es necesario el desarrollo de algún tipo de modelo que unifique de alguna

forma toda la diversidad de dispositivos para la interacción y la visualización. Por ejemplo, si se desea mover un objeto, sería interesante que la acción no se corresponda con una pulsación de ratón o un determinado botón de un mando, sino que toda esta diversidad de eventos se unifique en una sola acción, mover el objeto, y filtre todos los detalles concretos de los dispositivos de entrada.

En el desarrollo de un sistema de RV existen tres módulos importantes que han tenido desigual evolución. Estos tres módulos son: Motor Gráfico, Motor Físico e Inteligencia Artificial. El primero se hace cargo de todo lo referente a la visualización de los datos en dispositivos visuales, el segundo tiene como objetivo simular todos los aspectos físicos que hacen que la acción que se desarrolla en la escena sea coherente y el tercero busca que los actores de la escena se comporten de una forma independiente mostrando en el mundo virtual capacidades inteligentes.

Las diferentes herramientas aportadas en la actualidad para el desarrollo de los tres módulos anteriores son muy variadas. Existen múltiples soluciones a diferentes problemas. Sin embargo, generalmente no se observa ninguna unificación de criterios a la hora de abordar los diferentes retos que tiene su desarrollo. Si que es cierto, por otra parte, que se vislumbran ciertos puntos en común a la hora de definir determinados diseños estructurales, aunque no se aprecia un estudio de cómo los tres módulos se relacionan entre sí.

En los próximos capítulos se presentará un modelo integral que unifica de forma sencilla y flexible la diversidad que existe en los sistemas de RV y en los tres módulos anteriormente descritos. Para ello se utilizará la definición de un lenguaje que integra la actividad necesaria, su visualización y su interacción con el usuario. En el apartado 2 se presentan algunos antecedentes de sistemas de RV. Los objetivos que pretendemos alcanzar centran el contenido del apartado 3. En el apartado 4 se define el modelo propuesto, mientras que el 5 presenta un caso práctico. Por último, las conclusiones y los trabajos futuros se exponen en el apartado 6.

2. Antecedentes

Un sistema de RV está formado por tres grandes módulos: un motor gráfico, un motor físico y un módulo de inteligencia artificial. A continuación se hace un breve repaso de los antecedentes referidos a estos módulos.

En el desarrollo de los componentes que se utilizan para los motores

gráficos existen dos librerías: Direct3D [DirX] y OpenGL [OpGL]. Ambas librerías se definen como una capa entre la aplicación y la tarjeta gráfica. Han aparecido sistemas que unifican las dos API (*Aplication Program Interfaces*) en un único interfaz. Este es el caso de OGRE [OGRE], o de VTK [VTK].

Existen librerías que se utilizan para gestionar el hardware interactivo. Dos ejemplos son SDL [SDLay] y DirectX [DirX]. En general, tienen las mismas características salvo que SDL es software libre y está implementada tanto para Windows, como para Linux, mientras que la segunda no.

Hay una gran variedad de herramientas que implementan motores físicos Como ejemplos se tiene Working Model [WMod], que realizan simulaciones que se utilizan principalmente en el área educativa. Después existe Newton Game Dynamics [NGDy] Además existe Physics Abstraction Layer (PAL) [PALay] que es una capa abstracta para el desarrollo de MTR siendo Open Dynamics Engine (ODE) [ODEn] una implementación de parte de las especificaciones de PAL. Dentro del software propietario existen otras tantas API, destancando por un lado PhysX [PhyX], cuyo propietario es NVidia y es utilizado en la PlayStation 3, y Havok [Havok], perteneciente a la compañía Havok y que está implementado para Windows, Xbox 360, Wii, PlayStation3 y PlayStation Portable, Mac OS X y Linux.

Los últimos tipos de herramientas son los que desarrollan el módulo de IA. No existen librerías que puedan ser utilizadas como soluciones genéricas, salvo en algún caso. Su número es escaso y cada sistema de IA está diseñado específicamente para una aplicación concreta. Uno de los motivos por los que los sistemas de IA se han desarrollado con mayor lentitud que el resto de sistemas podría ser el esfuerzo que se realiza por mejorar el aspecto gráfico de los juegos, y no tanto por la inteligencia de sus personajes [Lai01]. Aún así, podemos especificar varios tipos de sistemas: Sistemas evolutivos y Sistemas basados en agentes.

Dentro de los sistemas evolutivos hay alguna herramienta que facilita el uso de dichos algoritmos. Podemos destacar EO Evolutionary Computation Framework [EOECF] desarrollado en C++ y que implementa lo necesario para el desarrollo de algoritmos evolutivos. También existe CILib [CILib], entorno de desarrollo implementado en Java. Por el contrario, sí que se puede encontrar una gran variedad de artículos que describen el desarrollo de algoritmos genéticos y artículos que definen sistemas evolutivos para casos concretos [GNY04, CM07, RGR05].

En cuanto a los sistemas basados en agentes, se puede obtener una extensa literatura que detalla los aspectos de este tipo de elementos, como por ejemplo

en [Woo97, WD00]. En algunos casos, estos agentes se denominan *bots*, sobre todo en el desarrollo de juegos en primera persona. Actualmente, este tipo de sistemas están muy extendidos en grupos de investigación para el desarrollo de IA, tanto para juegos como para simulaciones sociales y desarrollo de robots móviles [JHM07,Ken06]. Existen varios entornos que nos simplifican la tarea para desarrollar *bots*, aunque estos son sistemas para juegos concretos: por ejemplo se tiene QuakeBot para el juego Quake y FlexBot para el juego Half-Life [Lai01, AK02]. Existen librerías más genéricas para el desarrollo de Agentes, por ejemplo Jade [Jade], que modela sistemas multiagentes basados en Java.

3. Objetivos

Los objetivos se orientan principalmente a conseguir un modelo que integre los diferentes módulos que forman parte de un sistema de RV y que tenga una total independencia de los 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. Para lograr esto se pretende:

1. Definir un motor gráfico que elimine la diversidad de los diferentes dispositivos visuales. Es decir, se desea que con una única descripción de la escena se procese la misma, de tal manera que se muestre en cualquier dispositivo gráfico y con un nivel de detalle acorde con las características gráficas del dispositivo.

2. Definir un motor físico que modele toda la actividad del sistema, adaptándose a los diferentes componentes hardware donde se va a ejecutar. Si se dispone de componentes hardware que implementan algoritmos físicos, el sistema debe aprovecharlos, pero si por el contrario no existen, los debe implementar mediante software.

3. El motor de IA se debe integrar con el motor físico considerando las limitaciones impuestas por este. Esto supone que se debe realizar un trabajo importante en la integración de un motor sobre el otro.

4. Se pretende que la interacción con el sistema no dependa del hardware de entrada, si no que se abstraiga del origen de la interacción, y procese directamente las órdenes del usuario.

5. La reutilización de los diferentes elementos de forma casi inmediata. Si, por ejemplo, se diseña un elemento para un determinado mundo virtual, utilizando los mecanismos proporcionados por el sistema, se podrá reutilizar dicho componente en cualquier otro mundo virtual o aplicación que gestione un sistema de RV.

Para la realización de todos estos objetivos se utilizarán modelos matemáticos que formalizarán los diferentes componentes del sistema, abstrayendo las características que definen los tres módulos importantes de un sistema de RV.

4. Modelo para la generación de Mundos Virtuales

Un mundo virtual se caracteriza por un conjunto de actores o elementos, con una descripción geométrica y que realizan una actividad en un escenario. La actividad puede o no modificar objetos que componen el escenario, así como el aspecto del actor o de otros actores.

La descripción de un mundo se puede considerar como una secuencia ordenada de primitivas, transformaciones que modifican el comportamiento de las primitivas y actores que definen la actividad dentro del sistema. El concepto de primitiva se debe considerar, no como una primitiva de dibujo tal como una esfera, un cubo, etc. sino más bien como una acción que se debe ejecutar en un determinado sistema de visualización. Por ejemplo, dentro del concepto de primitiva cabe considerar la ejecución de un sonido. Los actores, por su parte, serán los componentes que desarrollen una o varias actividades en el mundo virtual y que se visualizarán mediante primitivas y transformaciones. Para modelar las diferentes actividades de los actores se va a usar el concepto de evento, eso sí, considerando un evento, no como una acción generada por un dispositivo de entrada, sino más bien como una acción que representa la activación de una determinada actividad que puede ser ejecutada por uno o varios actores.

A cada elemento que compone una escena se le puede asignar un símbolo, formando un conjunto de símbolos. Con este conjunto se puede construir diferentes cadenas que describen una escena. Estas cadenas deben construirse con una sintaxis definiendo un lenguaje. Habitualmente una sintaxis se presenta como una gramática [DW94], en adelante denotada por M, que queda

definida por la tupla $M = < \Sigma, N, P, S >$. Con la gramática M se determinará el lenguaje $L(M)$. La definición de la tupla gramatical se expresa como:

Sea P el conjunto de símbolos que define un conjunto de primitivas.

Sea T el conjunto de símbolos que define las transformaciones.

Sea $O = \{ \cdot ( ) \}$ el conjunto de símbolos de separadores y operaciones.

Sea D el conjunto de tipos de eventos generados para el sistema.

Sea A^D el conjunto de símbolos que representan actores que van a activarse para los eventos enumerados en el superíndice. Por ejemplo, el actor a^d será ejecutado cuando se produzca un evento e^d.

Sea $\Sigma = P \cup T \cup O \cup A^D$ el conjunto de símbolos terminales.

Sea $N = \{mundo, variosObjetos, objeto, actor, transformación, figura\}$ el conjunto de símbolos no terminales.

Sea $S = \{mundo\}$ el símbolo inicial de la gramática.

Definimos las reglas gramaticales P como:

mundo :- variosObjetos
variosObjetos :- objeto | objeto · variosObjetos
objeto :- figura | transformación | actor
actor :- a^d(variosObjetos), $a^d \in M^D, d \in D$
transformacion :- t(variosObjetos), $t \in T$
figura :- p, $p \in P$

La gramática M es una gramática independiente del contexto, o de tipo 2, y por tanto se asegura que existe un procedimiento que verifica si una escena está bien descrita o no, o dicho de otro modo, que la cadena pertenece o no al lenguaje $L(M)$. Es decir, se podrá determinar que, por ejemplo, la cadena $a_1^d(p_1 \cdot p_2) \cdot p_3 \cdot t_1(p_1 \cdot p_2) \in L(M)$ donde $p_i \in P$, $t_i \in T$, $a_i^d \in A^D, d \in D$, pero que la cadena $a_1^d \cdot p_1 \notin L(M)$.

Sin embargo, se necesita saber cuál va a ser la funcionalidad de las diferentes cadenas. Se puede considerar que la semántica de una lenguaje explica esta funcionalidad. La semántica tiene varias formas de definición: operacional, denotacional y axiomática. En el caso presente se va a utilizar el método denotacional que describe funciones matemáticas para cada una de las cadenas.

La semántica del actor se describe como la ejecución del mismo cuando recibe o no un evento que debe manejar. Esta ejecución se representará como una función que define la evolución del actor a través del tiempo. Esto significa que va a cambiar la cadena que describe su actual estado. La función que

define el comportamiento de un actor para un evento producido se denomina *función evolutiva del actor* y se denotará con λ. Su expresión viene dada por:

$$\lambda(a^d(v), e^f) = \left\{ \begin{array}{lll} u \in L(M) & Si & f = d \\ a^d(v) & Si & f \neq d \end{array} \right\} \tag{1}$$

El resultado de la función λ puede contener o no al propio actor o puede generar otro evento para la siguiente etapa. Si se desea acumular un evento para la siguiente etapa se utilizará la notación Δe^f. Esto permitirá que los actores puedan generar eventos, y pasen a la siguiente etapa acciones que puedan recoger otros actores implementando un proceso de retroalimentación del sistema. En el caso de que el evento no coincida con el evento activador entonces la función devuelve el mismo actor. Más adelante se verá que las funciones λ pueden clasificarse según las cadenas que se definan en su evolución. Sobre todo, van a ser esenciales para la visualización del sistemas determinadas funciones λ, que solo devuelven cadenas con primitivas y transformaciones.

Con la función λ se puede definir un algoritmo que, dado un conjunto de eventos y una cadena, describe como el sistema evoluciona. A esta función se la llamará función de evolución α del sistema. Para su definición se necesita un conjunto de eventos $e^i, e^j, e^k, ..., e^n$ que abreviando se denotará como e^v siendo $v \in D^+$. La función se define como:

$$\alpha(w, e^v) = \left\{ \begin{array}{lll} w & Si & w \in P \\ t(\alpha(v, e^v)) & Si & w = t(v) \\ \prod\limits_{\forall f \in v} (\lambda(a^d(\alpha(y, e^v)), e^f)) & Si & w = a^d(y) \\ \alpha(s, e^v) \cdot \alpha(t, e^v) & Si & w = s \cdot t \end{array} \right\} \tag{2}$$

El operador $\prod\limits_{\forall f \in v} (\lambda(a^d(x), e^v))$ realiza la concatenación de la cadenas generadas por λ. Se debe observar que para las cadenas que solo son transformaciones y primitivas, el sistema se queda como está y no forman parte de su evolución. Este tipo de cadenas van a ser importantes a la hora de optimizar el sistema, ya que solo ejecutaremos α para aquellas cadenas que sean actores.

Por otro lado, están las primitivas cuya semántica será definida mediante la función γ. Dado un símbolo del alfabeto P ejecuta una primitiva sobre lo que se denominará una geometría aplicada G. Por tanto, la función será una aplicación definida como:

$$\gamma : P \rightarrow G \tag{3}$$

Esto significa que dependiendo de la función γ, el resultado variará según la definición que se haga de G. Esta G se puede definir como las acciones que se ejecutan en una librería gráfica concreta, como OpenGL o Direct3D. Se relacionarán los símbolos de P con la ejecución de funciones de la librería y si, por ejemplo, $esfera \in P$ se podrá ejecutar la función que dibuja una esfera en OpenGL o Direct3D:

$$\gamma_{opengl}(esfera) = glutSolidSphere$$
$$\gamma_{direc3d}(esfera) = drawSphereDirect3D$$

Sin embargo, esta definición no solo se puede utilizar para librerías gráficas, sino que se puede ampliar a otras definiciones. Por ejemplo, se puede definir para el mismo alfabeto P otra función γ_{bounds} que calcule los límites de la figura que define una primitiva. Es decir, si '*boundSphere*' es una función que implementa el cálculo de los límites de una esfera, entonces:

$$\gamma_{bounds}(esfera) = boundSphere$$

Los ejemplos anteriores demuestran que la función γ aporta toda la abstracción necesaria para homogeneizar las diferentes implementaciones que existen para realizar la definición de un mundo concreto.

Sin embargo, solo se han tratado elementos geométricos o de dibujo. No hay ninguna restricción al respecto siempre y cuando exista la definición para esta primitiva en la función γ. Así, también se pueden considerar como primitivas la ejecución de un sonido, el reflejo de un material, el color de un personaje, el movimiento de un robot, la manipulación de una máquina, etc. Además, la definición de la función γ también describe sistemas que escriban diferentes formatos de ficheros (VRML, DWG, DXF, XML, etc.), transporten cadenas por la red, ejecuten instrucciones de una máquina que realiza esculturas 3D, etc.

Por último, la gramática define lo que son las transformaciones que modifican el comportamiento de las primitivas. Estas transformaciones tienen un ámbito de aplicación y deberán ser aplicadas a determinado conjunto de primitivas. Es decir, se desea que una transformación $t \in T$ se aplique sobre una subcadena w que se simboliza en el lenguaje como $t(w)$. Para definir el ámbito de aplicación de una trasformación se definen las siguientes funciones semánticas:

I: T $\rightarrow$ G (Inicia transformación)
F: T $\rightarrow$ G (Finaliza transformación)

Estas dos funciones tienen las mismas características que la función γ pero aplicadas al conjunto de transformaciones anteriormente descritas.

Existe una restricción sobre las tres últimas funciones, y es que no pueden ser aplicadas sobre actores. Es decir, no se puede ejecutar $\gamma(a_1^d(w))$, ni $I(a_1^d(w))$, ni $F(a_1^d(w))$. Se debe primero transformar el actor en una cadena de primitivas y transformaciones, que será su representación gráfica. Por ejemplo, si se define un actor que describe la actividad de una nave espacial, se debe transformar el actor en una secuencia de primitivas y transformaciones que van a representar la imagen de la nave espacial.

Como se ha dicho con anterioridad, las funciones λ se pueden clasificar según la cadena de resultados que devuelvan. Uno de los tipos de funciones λ son aquellas que devuelven solo cadenas que sean primitivas y transformaciones. A estas funciones se les va a denominar función de aspecto τ y su expresión es:

$$\tau(a^d(v), e^f) = \begin{cases} z \in L(E) & Si \quad f = d \\ \epsilon & Si \quad f \neq d \end{cases} \tag{4}$$

Se puede observar que existen pequeñas diferencias entre λ y τ. La primera es que τ devuelve cadenas que pertenecen a $L(E)$, siendo este el lenguaje de la gramática E que es igual que M pero eliminado los actores. Además si no coincide con el evento devuelve una cadena vacía. Esto quiere decir que para ese evento el actor no tiene representación en la vista. El tipo de evento es importante y no se corresponde con ningún dispositivo de entrada en concreto, si no más bien es un evento que se encargará de establecer los diferentes tipos de vista que tiene el sistema. Así, si se desea un sistema de dibujo que filtre determinados elementos para que queden invisibles, solo tienen que no reaccionar al evento. Estos eventos también sirven para decidir qué tipo de visualización se desea, dependiendo de la ventana o dispositivo de salida.

Al igual que con λ, se define un algoritmo para τ que dado una cadena $w \in L(M)$ y un conjunto de eventos e^v con $v \in V^+$ y $V \subset D$, siendo V los tipos de eventos para visualizar el sistema, se devuelva una cadena $v \in L(E)$. A esta función le denominamos función de visualización Ω y se define como:

$$\Omega(w, e^v) = \left\{ \begin{array}{cc} w & Si \quad w \in P \\ t(\Omega(y, e^v)) & Si \quad w = t(y) \\ \prod\limits_{\forall f \in v} \tau(a^d(\Omega(y, e^v)), e^f) & Si \quad w = a^d(y) \\ \Omega(s, e^v) \cdot \Omega(t, e^v) & Si \quad w = s{\cdot}t \end{array} \right\} \tag{5}$$

Se puede comprobar que es igual que la función α salvo que sustituimos λ por τ y que los eventos que pueden ser pasados a la función solo pueden pertenecer a V. En el caso de que a la función Ω se le pase un evento que no pertenece a V lo único que producirá será una cadena vacía.

4.1. Actividad y Eventos

La actividad en un sistema de RV la representan los actores. Esta actividad suele realizarse entre actores del propio sistema y entre dispositivos de entrada y cadenas de la escena.

Toda actividad consiste en un proceso que se produce en un sistema como reacción a un determinado evento. Este evento puede ser producido por otra actividad o por un dispositivo de entrada. Como resultado de una actividad se puede producir una modificación de los estados de los actores que forman el sistema. Así, se puede ver que lo que define a una actividad es, por un lado, el proceso de ejecución de la actividad que realiza el actor, y por otro, el evento o el conjunto de eventos que van a activar dicha actividad.

El concepto de evento dentro de un actor es importante, ya que define cuándo se va a ejecutar la actividad independientemente del origen del evento. Esta independencia es necesaria para la generalización de los dispositivos de entrada, ya que independiza el dispositivo de la actividad que se debe procesar.

La generalización de los eventos facilita la realización de simulaciones. Por ejemplo, si se quiere hacer una simulación en la que un usuario mueve un determinado elemento, solo se deben generar tantos eventos de 'movimiento' como se desee, simulando la presencia del usuario. También, se puede simular la existencia de dispositivos. Por ejemplo, simular la existencia de un guante de realidad virtual con un ratón. Para ello, se debe generar aquellos eventos producidos por el guante de realidad virtual.

Ya se ha visto que la identificación del evento es uno de los atributos dentro de la actividad de un sistema de RV. Pero además, en un evento se pueden definir unos datos que caracterizan a una instancia del evento. Por

ejemplo, un evento de movimiento puede contener los datos del movimiento. Por supuesto, pueden existir eventos que no tengan datos asociados.

No toda actividad tiene porqué ejecutarse cuando se envía un evento. Pueden existir determinados eventos que ejecutan la actividad si se cumplen ciertas condiciones, dependiendo del estado del actor. Esta comprobación va a estar definida en el evento y no en el objeto que tiene definida la actividad. Por tanto, se va a establecer una definición de evento:

Se define e_c^d como el evento de tipo 'd', que tiene como datos 'e' y que se ejecuta de forma opcional bajo la condición 'c'.

Se puede observar que el origen de la creación de dichos eventos puede ser cualquiera, y que no es importante el origen, sino más bien el tipo de evento que se quiere enviar y su definición.

4.2. Dispositivos de entrada

Se desea establecer una independencia entre el sistema y los dispositivos de entrada. Por tanto, se necesita definir, para un conjunto de dispositivos de entrada, los eventos necesarios para hacer reaccionar al sistema. Por ejemplo, si tenemos un ratón y se quiere mover un personaje de un juego, no se tiene que definir para el personaje el evento de ratón, sino que el ratón va a definir un evento que genera el movimiento del personaje.

Para ello, se va a especificar una función que, dependiendo del dispositivo, va a generar los eventos que se necesitan para hacer reaccionar al sistema. A este elemento se va denominar *generador de eventos* y su definición es:

Se define como $G^D(t)$ al generador que crea los eventos del tipo d en el instante t, siendo $d \in D$ donde D es el conjunto de tipos de eventos que pueden ser generados.

En la definición anterior se debe destacar que los eventos se generan en un instante t. Esto se debe a motivos de sincronización. Existe además la posibilidad de que G no cree ningún evento.

Con esta definición se independiza los dispositivos de entrada del sistema de RV. Así, se puede crear un generador para un ratón, para un guante, para un teclado o para cualquier dispositivo de entrada y que puedan generar los mismos tipos de eventos. Lo interesante de la implementación del generador es que solo se tendrá en un sitio la parte dependiente del dispositivo de entrada.

La actual definición de generador de eventos no tiene porqué estar asociada a un dispositivo de entrada hardware, sino que también se pueden asociar

a procesos software. Estos procesos software pueden ser de cualquier índole: de colisión entre elementos, de contabilización, de acumulación de información para que en los instantes siguientes se produzcan otros eventos, etc. En definitiva, cualquier proceso que interactúe con el sistema.

Existe el problema de que varios generadores pueden crear eventos del mismo tipo y con distintos datos. Para que esto no cree conflictos se va a establecer un orden de prioridad entre los distintos generadores, de forma que, dados dos generadores G_i y G_j, se creen dos eventos del mismo tipo, por lo que prevalecerá el evento creado por G_i sobre el de G_j. Por ejemplo, si existe un teclado, un ratón y un *joystick*, se puede establecer el siguiente orden de prioridad: $1^{\underline{o}}$ *Joystick*, $2^{\underline{o}}$ Ratón, $3^{\underline{o}}$ Teclado. Estos procesos de redundancia se producen porque el sistema debe ser implementado para todos estos dispositivos. Si existen varios, solo prevalecerá uno de ellos. El orden de prioridad se puede establecer, por ejemplo, por orden de inmersión en el sistema o facilidad de uso. En el caso de que no exista uno de los dispositivos, actuaría el de siguiente prioridad.

El proceso para la obtención de los eventos producidos por todos los dispositivos de entrada y sus generadores asociados se define como:

Se define G^* como el conjunto de todos los generadores de eventos del sistema asociados a los dispositivos de entrada.

Se define $e(z, e^i)$ siendo $G_i(t) = e^i$, como:

$$e(z, e^i) = \begin{Bmatrix} z \cdot e^i & si & e^i \notin z \\ z & si & e^i \in z \end{Bmatrix}$$

La función $E(G^*, t)$ que recolecta todos los eventos de todos los generadores se define como:

$$E(G^*, t) = \begin{Bmatrix} e(z, G_i(t)) & z = E(G^* - G_i, t) \\ \epsilon & si \quad G^* = \emptyset \end{Bmatrix}$$

4.3. Algoritmo del sistema

Una vez definidos todos los elementos implicados en el modelo que gestionará un sistema de RV se puede establecer el algoritmo que ejecutará todo el sistema, tanto la evolución como la visualización del mismo, para cada instante t o frame. El algoritmo es:

Sea $D = \{$ Conjunto de todos los tipos de eventos posibles en el sistema$\}$.

Sea $V = \{$ Conjunto de todos los tipos de eventos de dibujo$\}$ con $V \subset D$.

Sea $G^* = \{$ *Todos los generadores de eventos que genera eventos de tipo* $D\}$

Sea g *el dispostivivo de salida.*

Sea e^* *todos los eventos generados por el sistema.*

Sea e^v *todos los eventos de los dispositivos visuales. Estos eventos será una de las entradas de la función de visualización del sistema* Φ*.*

Sea e^u *todos los demás eventos que no son visuales. Estos eventos será la entrada de la función de evolución del sistema* α*.*

Se define el algoritmo Generador de Mundos Virtuales como:

Paso 1. Sea $w = w_o$ *donde* w_o *es la cadena inicial del sistema.*

Paso 2. Sea $t = 0$*.*

Paso 3. $e^* = E(G^*, t)$

Paso 4. $e^v =$ *Evento de* e^* *donde* $v \in V^+$

Paso 5. $e^u = e^* - e^v$

Paso 6. $w_{sig} = \alpha(w, e^u)$

Paso 7. $v = \Omega(w, e^v)$

Paso 8.$g = \Phi(v)$

Paso 9. $w = w_{sig}$*;* $t = t + 1$

Paso 10. Si $w = \varepsilon$ *entonces ir a Paso 12.*

Paso 11. Ir a paso 3.

Paso 12. Fin.

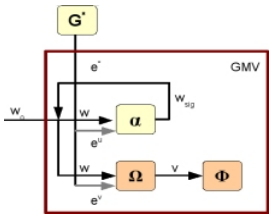

Figura 1: Diagrama del Algoritmo Generador de Mundos Virtuales

Los dos primeros pasos del algoritmo conforman la parte de la inicialización del sistema. Esto es, al sistema se le introduce el estado inicial de la escena w_o y el primer frame va a ser el 0.

Los pasos 3, 4 y 5 gestionan los eventos del sistema. Primero se llama a los generadores y se asigna a una lista e^* todos los eventos generados. En los pasos 4 y 5 se dividen los eventos en eventos para la visualización e^v y los demás eventos e^u. La primera será una de las entradas a la función Ω y la segunda a la función α.

En el paso 6, con la cadena actual del sistema w y los eventos que no son de visualización (e^u), se llama a la función de evolución α para calcular la cadena del siguiente frame o instante t.

En el paso 7 y 8 se realiza la visualización del sistema. En primer lugar se transforman los actores en primitivas y transformaciones llamando a la función Ω y con los eventos de visualización e^v. Una vez se tiene la cadena que representa la visualización del sistema en este instante t se llama a la función $\Phi(v)$ para visualizar el sistema en el motor gráfico.

En el paso 9 se prepara para la siguiente iteración y se asigna w a w_{sig} y se aumenta en 1 el número de frames o instantes.

En el paso 10 se mira si se ha llegado a la condición de finalización, esto es, si la cadena siguiente es vacía y si es cierto se termina el algoritmo, paso 12. En caso contrario, se vuelve al paso 3 para el siguiente frame.

Se puede destacar que el paso 6 es intercambiable con los pasos 7 y 8 porque no comparten datos que tengan que modificar. Esta característica es importante para la implementación en paralelo del algoritmo. Efectivamente, si este algoritmo se desea paralelizar se puede poner en tareas diferentes el paso 6 por un lado, y los pasos 7 y 8 por otro.

También se puede observar que para que termine el algoritmo solo se debe devolver en α una cadena vacía. Esta cadena vacía se puede obtener mediante un evento especial que se genere cuando se desea terminar el sistema.

5. Caso práctico

Para exponer con mayor claridad cómo funciona y se define un sistema de RV con el modelo anteriormente definido, se va a describir un caso práctico consistente en la implementación de un modelo para la simulación de incendios forestales producidos por tormentas [JHM07]. El modelo consiste en la representación de un mundo, donde cada cierto tiempo crece un árbol con una cierta probabilidad g. El crecimiento de un árbol hace que este se sitúe en un determinado lugar (i, j) de un tablero 2D. Por otro lado, con una probabilidad f puede caer un rayo en una determinada casilla (i, j). Si un relámpago cae en una casilla sin árbol, entonces no sucederá nada. Si por el contrario, hay un árbol, entonces esté arderá y quemará todos los árboles que estén a su alrededor, y estos a su vez quemarán los que estén al suyo, produciendo una reacción en cadena. Se puede ver un ejemplo de la implementación de este modelo en la figura 2.

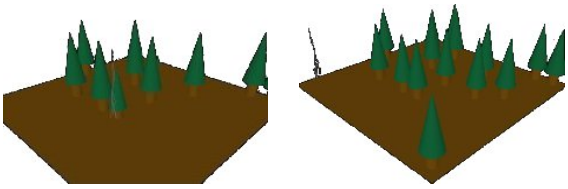

Figura 2: Ejemplos de diferentes estados del tablero de juego

Se debe resaltar que no se está interesado en el resultado propio del modelo anterior, sino más bien en definir un sistema de RV para este modelo con el sistema propuesto en el capítulo anterior.

Son cuatro los elementos principales que se deben definir. En primer lugar se definirán los eventos necesarios para que el sistema pueda ejecutar las actividades. En segundo lugar, se diseñarán los generadores de eventos. En tercer lugar, se definirán los actores que forman parte de la escena. Por último, se introducirán las primitivas que podrán hacer visible las diferentes partes que se deseen mostrar.

Los eventos que se pueden producir en el sistema se muestran en la tabla 1.

Los siguientes elementos que se deben definir son los generadores de eventos. Estos son tres:

$$G_{tiempo} = \{e^t \ cada \ cierto \ tiempo \ t\}$$
$$G_{bosque} = \begin{Bmatrix} e^c & con \ probabilidad \ g \\ e^r & con \ probabilidad \ f \end{Bmatrix}$$
$$G_{opengl} = \{e^d \ cada \ redibujado\}$$

Por supuesto, podríamos definir más generadores. Por ejemplo, cuando pulsamos un botón se podría generar un evento e^r y lanzar un relámpago en la posicion (i,j). Sin embargo, para este caso práctico no es necesario.

En la table 2 se definen cada cada una de las primitivas y las transformaciones.

Las funciones γ, I, F son implementadas para Opengl y dibujadas en una ventana.

Por último, se especifican los actores que componen nuestro sistema. Para la definición de un actor se debe definir la función de evolución λ. La tabla 3 muestra los diferentes actores y su función de evolución.

Aclarar que en el caso del actor B^{crv} y el del evento e^v, solo cambiará el estado interno del actor al asignar la posición (i, j) del evento como vacía.

Tipo de evento	*Significado*	*Datos asociados*
t	Evento que se genera cada cierto tiempo t	Incremento del tiempo con respecto al anterior evento
c	Crear un árbol en una determinada posición	Posición (i,j) donde se crea el árbol
r	Creación de un relámpago en una posición	Posición (i,j) donde se crea el relámpago
v	Elimina el árbol creado en una posición	Posición (i,j) donde se elimina el árbol
q	Quemar un árbol en una posición	Posición (i,j) donde se quema el árbol
d	Dibuja en OpenGL	Nada

Cuadro 1: Definición de eventos

Ahora todos los actores deben tener una representación que se verá en el dispositivo visual. Para ello, se define la función τ que se muestra en la tabla 4.

Se debe aclarar que en el dibujado de AC, el escalado S va a corresponder a un escalado creciente dependiendo del estado actual del actor, que será modificado dependiendo del evento e^t. El efecto es el de un árbol que crece de menos a más. La escala para cada uno de los t vendrá definida por la expresión $s = \frac{t}{N}$, donde N es el número de frames totales de la animación.

En el caso de AQ, el escalado se va a aplicar en sentido inverso, representado por s^{-1}. Esto significa que el efecto es el de un árbol quemándose y menguando.

Por último decir, que el dibujado de un relámpago dependerá del instante t en que se encuentre R, y que poco a poco se dibujará un relámpago con un

Primitiva	*Dibujo*
A	Árbol.
A_q	Árbol quemado
R	Relámpago.
B_{NxN}	Tablero de NxN que representa el bosque
Transformaciones	*Transformación*
$T_{i,j}$	Traslación (i, j)
S_s	Escalado (s)

Cuadro 2: Definición de primitivas y transformaciones

algoritmo dependiente de t.

Como se puede comprobar, en las animaciones el dibujo de los actores siempre va a depender de t, que modificará el estado del actor para cambiar su representación según vaya desarrollándose la animación.

Para terminar se debe definir la cadena inicial: $w_0 = B^{crv}$

Con esta última expresión queda terminada la definición de todos los elementos que componen el sistema de RV para modelar una simulación de incendios forestales producido por relámpagos. Para ver como funciona el algoritmo propuesto en el capítulo anterior, se va a suponer que los generadores definidos anteriormente construyen varios eventos. Dada la cadena inicial, el sistema evolucionaría de la siguiente manera:

Paso 3. $e^* = E(G^*, t) = \{e^t, e^c, c^d\} = \{e^{tcd}\}$
Paso 4. $e^v = e^d$
Paso 5. $e^u = e^* - e^v = e^{ct}$
Paso 6. $w_{sig} = \alpha(w, e^{ct}) = \lambda(B^{crv}, e^{ct}) = AC^0 \cdot B^{crv}$
Paso 7. $v = \Omega(w, e^v) = \Omega(B^{crv}, e^d) = \tau(B^{cr}, e^d) = B_{NxN}$
Paso 8. $\Phi(v) = \Phi(B_{NxN}) = \gamma(B_{NxN})$
Paso 9. $w = w_{sig} \quad y \quad t = t + 1$
Paso 11. Ir a paso 3
Paso 3. $e^* = E(G^*, t) = \{e^t, e^r, c^d\} = \{e^{trd}\}$
Paso 4. $e^v = e^d$
Paso 5. $e^u = e^* - e^v = e^{rt}$
Paso 6. $w_{sig} = \alpha(AC^0 \cdot B^{crv}, e^{rt}) =$
$= \lambda(AC^0, e^t) \cdot \lambda(B^{crv}, e^r) = AC^1 \cdot R^0 \cdot B^{crv}$
Paso 7. $v = \Omega(AC^0 \cdot B^{crv}, e^v) =$
$= \tau(AC^0, e^d) \cdot \tau(B^{cr}, e^d) = T_{(i,j)}(S_0(A)) \cdot B_{NxN}$

Actor	*Descripción*	*Función* λ
B^{crv}	Representa el bosque	$\lambda(B^{crv}, e^f) = \begin{cases} AC^t \cdot B^{crv} & f = c \\ R^t \cdot B^{crv} & f = r \\ B^{crv} & f = v \\ B^{crv} & f \neq c, r, v \end{cases}$
AC^t	Representa un árbol en crecimiento.	$\lambda(AC^t, e^f) = \begin{cases} AC^{t+1} & f = t \\ A^q & f = t \wedge t+1 > N_{frames} \\ AC^t & f \neq t \end{cases}$
AR^q	Árbol	$\lambda(A^q, e^f) = \begin{cases} AQ^t & f = q \\ A^q & f \neq q \end{cases}$
R^t	Representa la animación de un relámpago	$\lambda(R^t, e^f) = \begin{cases} R^{t+1} & f = t \\ \Delta e^q & f = t \wedge t+1 > N_{frames} \\ R^t & f \neq t \end{cases}$
AQ^t	Árbol quemándose	$\lambda(AQ^t, e^f) = \begin{cases} AQ^{t+1} & f = t \\ \Delta e^v & f = t \wedge t+1 > N_{frames} \\ AQ^t & f \neq t \end{cases}$

Cuadro 3: Descripción de los actores

Paso 8. $\Phi(D_{(i,j)}(S_0(A)) \cdot T) =$
$= I(T_{(i,j)}) \cdot I(S_0) \cdot \gamma(A) \cdot F(S_0) \cdot F(T_{(i,j)}) \cdot \gamma(B_{NxN})$
Paso 9. $w = w_{sig}$ *y* $t = t + 1$
Paso 11. Ir a paso 3
.....

6. Conclusiones y trabajos futuros

Se ha presentado un modelo con un objetivo prioritario, la separación de la actividad de un sistema gráfico de la de los dispositivos visuales y de interacción que componen dicho sistema.

Actor	*Función* τ
B^d	$\tau(B^d(v), e^f) = \begin{Bmatrix} B_{NxN} & f = d \\ \epsilon & f \neq d \end{Bmatrix}$
AC^d	$\tau(AC^d(v), e^f) = \begin{Bmatrix} D_{(i,j)}(S_s(A)) & f = d \\ \epsilon & f \neq d \end{Bmatrix}$
A^d	$\tau(AR^d(v), e^f) = \begin{Bmatrix} T_{(i,j)}(A) & f = d \\ \epsilon & f \neq d \end{Bmatrix}$
R^d	$\tau(R^d(v), e^f) = \begin{Bmatrix} T_{(i,j)}(R) & f = d \\ \epsilon & f \neq d \end{Bmatrix}$
AQ^d	$\tau(AQ^d(v), e^f) = \begin{Bmatrix} T_{(i,j)}(S_{s^{-1}}(A_q)) & f = d \\ \epsilon & f \neq d \end{Bmatrix}$

Cuadro 4: Definición de las funciones de dibujo

Por uno lado, se tiene un lenguaje definido mediante gramáticas independientes del contexto que detalla los elementos que componen el sistema. Se establecen determinadas funciones que representan la evolución de cada uno de los elementos a través del tiempo. Se definen funciones que extraen la representación gráfica de los elementos. Esta representación se envía a los dispositivos visuales mediante funciones que separan la implementación concreta sobre un dispositivo visual y la descripción gráfica del elemento, eliminando la dependencia del sistema de los dispositivos gráficos.

La separación de los dispositivos de entrada de la definición del sistema se ha articulado a través de los generadores de eventos, que levantan una capa entre el hardware de entrada y la representación del sistema. Para enlazar las acciones generadas por los dispositivos de entrada se utiliza el modelo de eventos.

En general, se puede comprobar que todo el modelo intenta separar las partes dependientes del tipo de dispositivo de las de la definición formal de un sistema de RV. La técnica empleada para dicha separación es utilizar modelos matemáticos que transformen las acciones de los dispositivos, tanto visuales como de entrada, en acciones más generales que puedan ser identificados por el sistema independientemente del origen de la acción, mediante abstracciones.

Esto supone varias ventajas: en primer lugar los dispositivos de entrada pueden ser sustituidos por otros dispositivos o por simulaciones de estos. Por otro lado existe la posibilidad de que los elementos del sistema sean

reutilizados. Además, la representación de los elementos puede ser visual o no visual, e incluso ser diferente dependiendo del dispositivo de visualización o de las necesidades del usuario. Así, si el dispositivo tiene unas características concretas, la representación siempre se podrá adaptar al dispositivo. Por último, varios actores pueden relacionarse entre sí enviándose mutuamente eventos que ambos reconozcan.

Con el sistema planteado se puede diseñar fácilmente un motor físico utilizando los elementos de la escena. Esto se realiza diseñando generadores de eventos de diferentes tipos, dependiendo de la característica física y activando la actividad del actor necesaria para que este reaccione ante el proceso físico. Por ejemplo, si se quiere que un actor reaccione ante una colisión, el generador de eventos de este tipo calculará las colisiones de los elementos extrayendo la geometría de la escena gracias al motor gráfico (utilizando la implementación de las funciones γ, I, F para calcular los cubos envolventes de los elementos) y generando los eventos necesarios para las colisiones. Si el sistema lo permite, se podría implementar mediante hardware este generador de eventos.

El motor de IA puede ser diseñado en las funciones de evolución de los actores. En estas funciones es donde cada actor va a tomar sus decisiones dependiendo de su estado actual. Mediante eventos se pueden ejecutar actividades que cambien el estado del actor y que modifiquen su comportamiento. Además, como el propio actor puede generar eventos es posible diseñar procesos de retroalimentación utilizados en IA. Además la integración entre motor físico y motor de IA está garantizada ya estos dos motores se relacionan entre sí mediante los eventos.

El uso de un lenguaje para la descripción de los elementos de la escena y la independencia del sistema gráfico hace que estas cadenas descriptivas puedan ser utilizadas en cualquier otro sistema, siempre y cuando están implementados los elementos básicos (todas las funciones semánticas explicadas).

El modelo presentado en este trabajo está actualmente en desarrollo y se desea seguir desarrollando los diferentes aspectos tratados aquí. Uno de los puntos a investigar y que se apuntan en el artículo es la optimización del algoritmo y su paralelización. La definición del sistema mediante cadenas facilita la posibilidad de algoritmos paralelos. Por otro lado, las cadenas representan estados del sistema y su evolución. Esta evolución puede variar mediante mutaciones y se puede investigar posibles soluciones evolutivas para el diseño del sistema. Por otro lado, se puede estudiar la posibilidad de activar eventos con una cierta probabilidad. Es decir, un actor activaría una actividad, ya no solo cuando suceda un determinado evento, sino que este

se activase con una cierta probabilidad. Así un actor podría definirse como $a^{p(e^1),p(e^2)}(w)$, siendo $p(e^i)$ la probabilidad de reaccionar al evento e^i.

En definitiva, se ha pretendido diseñar un sistema reutilizable, genérico y que se pueda incrementar y adaptar fácilmente, en el que la parte fundamental, que es la evolución en el tiempo, sea independiente de cómo se represente y con quién interactúe.

Referencias

[PhyX] PhysX by AGEIA: http://physx.ageia.com/

[DJK05] David J. Kasik William Buxton D. R. F.: Ten cad challenges. *IEEE Computer Graphics and Applications 25* (2005), 81–90.

[WMod] Working Model: Simulación de Sistemas: http://www.design-simulation.com

[DW94] Davis Martin D.; Sigal R., Weyuker E. J.: *Computability, Complexity, and Languages, Fundamentals of Theoretical Computer Science*, 2nd ed. San Diego: Elsevier Science, 1994.

[NGDy] Newton Game Dynamics: http://www.newtondynamics.com/

[ODEn] Open Dynamics Engine: http://www.ode.org

[WPEn] Wikipedia - Physics engine: http//en.wikipedia.org/wiki/physics_engine

[Havok] Havok: http://www.havok.com/

[JHM07] John H. Miller S. E. P.: *Complex Adaptative Systems*. Princeton University Press, 2007.

[JS04] Joshua Strickon J. A. P.: Emerging technologies. *Siggraph* (2004).

[PALay] PAL: Physics Abstraction Layer: http://www.adrianboeing.com/pal/

[DirX] Página oficial de DirectX: http://www.microsoft.com/windows/directx/default.mspx

[OpGL] Página oficial de OpenGL: http://www.opengl.org/

[SDLay] Simple DirectMedia Layer (SDL): http://www.libsdl.org/

[OGRE] OGRE 3D : Open source graphics engine: http://www.ogre3d.org/

[VTK] The Visualization ToolKit (VTK): http://public.kitware.com/vtk/

[EOECF] EO Evolutionary Computation Framework: http://eodev.sourceforge.net/

[CILib] CILib (Computational Intelligence Library): http://cilib.sourceforge.net/

[Jade] Jade - Java Agent DEvelopment Framework: http://jade.tilab.com/

[Lai01] Laird J. E.: Using a computer game to develop advanced ai. *Computer 34 (7)* (2001), 70–75.

[GNY04] Georgios N. Yannakakis John Levine J. H.: An evolutionary approach for interactive computer games. *In Proceedings of the Congress on Evolutionary Computation* (2004), 986–993.

[CM07] Chris Miles Juan Quiroz R. L. S. J. L.: Co-evolving influence map tree based strategy game players. *IEEE Symposium on Computational Intelligence and Games* (2007), 88–95.

[RGR05] Robert G. Reynolds Ziad Kobti T. A. K. L. Y. L. Y.: Unraveling ancient mysteries: Reimagining the past using evolutionary computation in a complex gaming environment. *IEEE transactions on evolutionary computation 9* (2005), 707–720.

[Woo97] Wooldridge M.: Agent-based software engineering. *IEEE Proceedings Software Engineering 144* (1997), 26–37.

[WD00] Wood M. F., DeLoach S.: An overview of the multiagent systems engineering methodology. *AOSE* (2000), 207–222.

[Ken06] Kenyon S. H.: Behavioral software agents for real-time games. *IEEE Potentials 25* (2006), 19–25.

[AK02] Aaron Khoo R. Z.: Applying inexpensive ai techniques to computer games. *IEEE Intelligent Systems 17(4)* (2002), 48–53.