The Visual Computer manuscript No.
(will be inserted by the editor)

Towards a General Model for Virtual Worlds Generation

Gabriel López-García · Rafael Molina-Carmona · Javier Gallego-Sánchez

the date of receipt and acceptance should be inserted later

Abstract One of the most important problems in Virtual Reality (VR) is the diversity in displays and interaction devices. Moreover, the heterogeneity in rendering, physics and artificial intelligence engines causes the inexistence of a comprehensive and coherent model which combines all the features of a VR system. A new formal model is proposed to solve these diversity problems.

The proposed model, based on a grammar, integrates the visualization and the interaction with a complete VR system. The world is described as a sequence of ordered primitives, transformations and actors, in a broad sense. Primitives are not just drawing primitives, but actions which operate in a representation system. Transformations are used to change the behaviour of primitives. Actors represent the activities in the virtual world which are carried out through events.

The resulting model separates the activity of the system from the devices that originate it. It has several advantages: devices can be easily replaced or simulated; system components can be reused; and the graphic representation can be different depending on the visual device. In short, this paper proposes a comprehensive, adaptive, reusable and generic model. Finally, a practical example is presented to prove the effectiveness of this model.

Keywords Virtual Reality · Virtual Worlds generation · Formal model · Interaction

Grupo de Informática Industrial e Inteligencia Artificial
Universidad de Alicante, Ap.99, E-03080, Alicante, Spain
Tel.: +34 96 590 3400
Fax: +34 96 590 3902
E-mail: {glopez, rmolina, ajgallego}@dccia.ua.es

1 Introduction

In recent years, the development of Virtual Reality (VR) and graphical systems has been amazing [41]. This progress has contributed to the creation of new ways to analyze and work with information, such as graphics, animations or simulations. Computer games industry is one of the elements which has contributed most to the advance of graphical systems and to the emergence of new VR systems [23]. However, these developments have opened up new challenges.

One of the most important problems is the wide variety of visual devices (monitors, PDAs, cell phones, VR goggles), each one with different features. For that reason, it is necessary to develop adaptive systems which generate the images according to the specific characteristics of the device (resolution, size, ...).

While the processes of visualization and renderization of images have reached a certain maturity in the past decades, in the field of interaction a similar evolution has not been produced [44,13]. Keyboard and mouse still dominate the world of interactive devices. High degree-of-freedom input devices are complex to manage, and other possibilities of interaction are only used in research and graphic art [44]. However, there has been tentative progress in recent years. Wii's input device is a highlight among them [48].

It is necessary to develop a model which unifies the whole range of devices of interaction and visualization. For example, the action of moving an object in the virtual world, should not be connected with the push of a specific button on a mouse, a keyboard, or any other device. Instead, this diversity of events should be unified in one action: "move the object", removing this way all the specific characteristics of the different devices.

In the development of VR systems, there are three major modules which have had an uneven evolution. These are the Graphics, the Physics and the Artificial Intelligence (AI) engines [3]. The first one handles the display of data into visual devices. The second one simulates all the physical processes, providing more coherent actions and visual effects to the virtual world [31]. The last one tries to simulate an intelligent and independent behaviour of the actors in the scene.

Multiple tools and solutions to develop these three modules have been proposed (see Section 2). However, there are no unified criteria to deal with the different development challenges. Some common points have been used to define certain structural designs, but there is no study about the connections between the different modules.

This paper presents a new model which integrates the diversity of the VR systems and the aforementioned modules (Graphics, Physics and AI engines). The major goal is to separate the scene definition from the hardware-dependent characteristics of the system devices. It uses a grammar definition which integrates activities, visualitation and interaction with users.

The following section describes the state of the art of VR systems and the related definition languages. Section 3 shows the main objectives to achieve. Section 4 defines the proposed model, and Section 5 presents a case study. Finally, the conclusions and the lines for future work are shown.

2 Background

A VR system can be considered to be composed of three main modules: rendering system, physics engine and AI system. It is also necessary to use a definition language to model the graphical environment and its behavior in the system. Next, the background of these modules are reviewed briefly.

There are two main libraries which are used to develop components of graphics engines: Direct3D [6] and OpenGL [26]. Both libraries are defined as a layer between the application and the graphics card. Different systems which try to unify these two APIs (*Aplication Program Interfaces*) into a single interface have appeared. This is the case of OGRE [24] and VTK [45]. There are also libraries which are used to manage the interactive hardware. Two examples are SDL [42] and DirectX [6]. In general, they have similar features, but SDL is open source and cross-platform.

There are a large variety of tools which implement physics engines [40], such as Working Model [51] (used mainly in the area of education to perform simulations), Newton Game Dynamics [22], Physics Abstraction Layer (PAL) [27] (it is an open source and cross-platform API for physical simulation), Open Dynamics Engine (ODE) [25] (it is an implementation of a part of the PAL specifications), and so on. There are many other API among the proprietary software, such as PhysX [32] (developed by NVidia and used in the PlayStation 3) and Havok [10] (it belongs to the company Havok and it is implemented for Windows, MacOS, Linux, Xbox 360, Wii, Playstation3 and PlayStation Portable).

The last type are the tools used to develop the AI engine. There are very few libraries which can be used as a generic solution because they are designed specifically for particular applications [39]. One of the reasons why the AI systems have been developed more slowly than the rest, may be that the effort being done to improve the graphical aspects of games is not comparable to the one done for the improvement of the intelligence of its characters [16]. Nevertheless, some types of systems can be found, such as evolutive algorithms or agent-based systems.

There are some tools which facilitate the use of evolutive algorithms, such as EO Evolutionary Computation Framework [8] (it is implemented in C++ and it has all the necessary to work with evolutive algorithms), CILib [4] (it is a development environment implemented in Java). There are also a lot of papers about genetic algorithms and about the development of evolutive algorithms for specific cases [52,18,37].

A wide research has been done in the area of agent systems, such as in [50,49]. Currently, these systems are widely used in research for the development of AI, game characters (called *bots*, especially in first person games), social simulations and mobile robots [23,39,19,14]. There are different frameworks which simplify the task of *bots* development, but they are often oriented to specific games: QuakeBot for Quake game and FlexBot for Half-Life game [16,15], for instance. There are other libraries for the development of generic agents, such as Jade [12] (used to develop multi-agent-based applications).

Besides the modules that make up a VR system, it is necessary to use a modeling language for the definition of the graphical environment. This environment or virtual world will be composed of a roughly complex set of graphic elements that will interact with the rest of the system in a particular way: they will react to external inputs (by the user or by other elements), follow a specific physical behavior, use its own AI, or evolve over time.

In general, the modeling languages used to design virtual worlds [2] are restricted to the description of the graphic environment. Some of the most used languages are: VRML (Virtual Reality Modeling Language [47]), X3D (successor of VRML, current ISO standard [47]), O3D (developed by Google [9]), DMLW (3D Markup Language for Web [7]) or even 3D Graphics formats as 3DF, 3DS or DXF. These languages mainly differ from the proposed grammar in the fact that they are not grammars but modeling languages, they do not allow the user to specify how the system interacts with the modeled environment, they are not evolutive, they generally cannot be separated into reusable modules, and they do not describe the entire system (cannot describe physics, AI, or how to react to events). Hence, they could only be considered a part of the proposed grammar: the description of the scene.

In the aforementioned graphics engines, such as OGRE [24] and VTK [45], it is also used a modeling language for describing the scene, which internally is encoded as a tree data structure. However, these languages are also limited to the description of the scene, being affected by the same constraints discussed.

In addition to modeling languages, there are also other languages or grammars which are widely used for the description of objects using a rule set. These are known by the term Procedural Modeling, which includes techniques such as L-Systems, fractals, or generative modeling [29]. They focus on creating a model from a rule set, rather than editing the model via user input. Much of the appeal of such renderings lies in the possibility to depict the complexity of large-scale systems, which are composed of simpler elements. Most of these approaches address the appearance of natural phenomena. Some of these systems include: the simulation of erosion [21], particle based forests [36] and cloud modeling [30]. Grammar-based generation of models (especially L-systems) are employed in computer graphics mainly to create plant geometry [33,38]. L-systems have evolved into very sophisticated and powerful tools [35,28] and have been used in the modeling of plants ecosystems [34], music [17], biochemical [43], or architecture [29,20,46].

These systems have certain similarities with the proposed system because they also use a grammar and a set of rules, and they are evolutive. But there are large differences, because these grammars are used only for the visual part, and they cannot specify physics, AI, or the interaction with the system. These differences prevent the use of these grammars or modeling languages in a system that attempts to integrate the visualization and the interaction with a VR system, like the proposed system.

3 Objectives

Our objectives are primarily focused on achieving a model which unifies all the modules of a VR system. This system must be independent of hardware devices, both visual and interaction devices. It means that one device can be replaced by another one, or even by a simulation, without affecting the internal mechanisms of the system. To achieve this, the following objectives are proposed:

1. Define a graphics engine which is independent of the diversity of visual devices. That is, only one description is needed to process the scene for any graphical device with a level of detail according to its features.
2. Provide flexibility to the graphics system in order to be able to change the elements of representation. This way, the system can provide different types of information without the need to modify the scene description.
3. Design a graphics system which allow the extraction and use of the elements that are defined in the scene. In addition, it is intended to handle other non-visual elements like sound.
4. Define a physics engine which models the whole system activity and can be adapted to different hardware devices. If there are hardware components which implement physical algorithms, the system must exploit them, but if not, it should be implemented with software.
5. Design a physics engine which can provide the information needed by the graphics system at any time. It is intended to perform functions of simulation and analysis of the elements that compose the scene.
6. Provide to the physics engine the ability to modify the scene elements through simulations, but without the need to know the architecture of these elements in detail.
7. Unify the AI system with the physics engine. The limitations imposed by the physics engine must be considered by the AI system.
8. Make the interaction independent from the hardware. The system has to abstract the origin of the action and process the orders directly from the user. Therefore, the interaction could be handled as an internal process of the system (such as the calculation of collisions).
9. Design and implement the elements so that they can be reused. If, for example, an item is designed for a specific virtual world, it should be able to be reused in another virtual world or VR application.

10. Control the activity of the system through events. However, the origin of these events could be unknown.

For the fulfillment of these objectives, mathematical models are used to formalize the different components of the system, abstracting the characteristics of the three major modules of a VR system.

4 Model for Virtual Worlds Generation

A virtual world can be described as an ordered sequence of primitives, transformations and actors. The concept of a primitive is considered as an action executed in a certain representation system. It is not, for instance, just a primitive to draw a sphere or a cube. Instead, it could implement any action which could be carried out in a representation device, such as a sound, for example. Transformations modify the behavior of primitives. Actors are the components which define the activities of the system in the virtual world. They will be displayed through primitives and transformations. To model the different actor's activities, the concept of an event is used. Events are not necessarily generated by an input device. They are considered as an action that represents the activation of a certain activity and they can be run by one or more actors.

Each element in the scene is represented by a symbol from the so called *set of symbols of the scene*. They are used to build strings to describe scenes. These strings are formed using a language syntax, which is presented as a grammar [5]. This grammar (hereafter denoted by M) is defined by the tuple $M = < \Sigma, N, R, s >$, where Σ is a finite set of terminal symbols used to build strings, N is a finite set of non-terminal symbols or variables which represents the substrings of this language, R is a finite set of syntactic rules used to describe how a non-terminal symbol can be defined as a function of terminal and non-terminal symbols (a syntactic rule is an application r: $N \rightarrow W^*$, where $W = \Sigma \cup N$) and $s \in N$ is the initial symbol or axiom of the grammar.

Considering the previous definition, M can be defined, in our case, as the following grammar:

1. Let $\Sigma = P \cup T \cup O \cup A^D_{ATTR}$ be the set of terminal symbols, where:
   - P is the set of symbols which defines primitives.
   - T is the set of symbols which defines transformations.
   - $O = \{\cdot\ ()\}$ is the set of symbols of separation and operation. () indicates the scope of the previous symbol, and $\cdot$ the concatenation of symbols.
   - A^D_{ATTR} is the set of symbols which represents actors where D is the set of all the types of events which can be generated by the system and $ATTR$ is the set of all the attributes of actor which define all the possible states. For example, the actor a^H_{attr} will carry out its activity when it receives an event e^h, where $h \in H$, $H \subseteq D$ and $attr \in ATTR$ is its current state.
2. Let N = {WORLD, OBJECTS, OBJECT, ACTOR, TRANSFORMATION, FIGURE} be the set of non-terminal symbols.
3. Let s = WORLD be the initial symbol of the grammar.
4. Grammar rules R are defined in Table 1.

Rule 1.	**WORLD** $\rightarrow$ OBJECTS
Rule 2.	**OBJECTS** $\rightarrow$ OBJECT | OBJECT $\cdot$ OBJECTS
Rule 3.	**OBJECT** $\rightarrow$ FIGURE | TRANSFORMATION | | ACTOR
Rule 4.	**ACTOR** $\rightarrow a^H_{attr}$, $a^H_{attr} \in \mathbf{A}^D_{ATTR}, H \subseteq D$
Rule 5.	**TRANSFORMATION** $\rightarrow t$(OBJECTS), $t \in T$
Rule 6.	**FIGURE** $\rightarrow p^+$, $p \in P$

Table 1 Grammar rules R

A string $w \in \Sigma^*$ is generated by the grammar M if it can be obtained starting with the initial symbol WORLD and using the rules of the grammar. The language generated by the grammar M is the set of all the strings which can be generated by this method. This language is called $L(M)$ and is defined as:

$$L(M) = \{w \in \Sigma^* \mid \text{WORLD} \overset{*}{\rightarrow} w\} \qquad (1)$$

The grammar M is a context-free grammar (or a type-2 grammar, according to the Chomsky hierarchy). Therefore, there is a parsing procedure which verifies if a scene is correctly described or, in other words, if a string belongs to the language $L(M)$ or not. In general, context-free grammars are parsed in quasilinear time tipically, and in cubic time in worst case [1,11]. In our case, as the proposed grammar is not ambiguous, strings are efficiently parsed in linear time.

Let us define a planetary system as an example to illustrate the grammar definition. Let us suppose that the system is made up of planets, a sun and a spaceship. The planets rotate around the sun an the spaceship can travel from a planet to another one. The sets of terminal symbols for primitives and transformations are shown in Table 2.

From Tables 1 and 2, it can be determined that the string $Translate_{x,y,z}(Spaceship)\ \cdot$

$P = \{\mathit{Planet},\ \mathit{Sun},\ \mathit{Spaceship}\}$
$T = \{\mathit{Rotate}_{\gamma,N_x,N_y,N_z},\ \mathit{Translate}_{x,y,z},\ \mathit{Scale}_s\}$

Table 2 Example of primitives and transformations for a planetary system

$\mathit{Rotate}_{\gamma,N_x,N_y,N_z}(\mathit{Planet}) \in L(M)$, but on the contrary, $\mathit{Translate}_{x,y,z}(\mathit{Rotate}_{\alpha,x,y,z}) \cdot \mathit{Planet} \notin L(M)$.

Apart from the language syntax, it is necessary to define the functionality of the strings, that is, the semantics of the language. It can be denoted through three methods: operational, denotational and axiomatic. The first one uses abstract machine language operations. The second one uses mathematical functions to describe the meaning of the strings. The third one defines the meaning of the strings through mathematical logic. In our case, the denotational method is used.

4.1 Semantics of the Language $L(M)$

In this section, a mathematical function is assigned to each rule of Table 1.

4.1.1 Semantic Function for Primitives (Rule 6)

Rule 6 defines the syntax of a figure as a sequence of primitives. Primitive's semantics is defined as a function α. Each symbol in the set P carries out a primitive on a given geometric system G. Therefore, the function α is an application defined as:

$$\alpha : P \to G \tag{2}$$

So, depending on the definition of the function α and on the geometry of G, the result of the system may be different. G represents the actions to be run on a specific visual or non-visual geometric system.

A usual example of geometric system are graphical libraries, such as OpenGL or Direct3D. In this case, each symbol of P would have associated an action of the rendering system.

In the planetary system example, the symbols $P = \{\mathit{Planet},\ \mathit{Sun},\ \mathit{Spaceship}\}$ of Table 2 are primitives. Therefore, functions α, that implement the drawing of the objects in the corresponding graphics system (e.g. Direct3D or OpenGL), must be defined. In Table 3 some simplified functions to draw with OpenGL and Direct3D are presented to illustrate the procedure.

These functions are examples of visual geometric systems. But this definition can also be extended to

$\alpha_{gl}(\mathit{Planet}) = \mathit{drawPlanetGL}$
$\alpha_{gl}(\mathit{Sun}) = \mathit{drawSunGL}$
$\alpha_{gl}(\mathit{Spaceship}) = \mathit{drawSpaceshipGL}$
$\alpha_{d3d}(\mathit{Planet}) = \mathit{drawPlanetD3D}$
$\alpha_{d3d}(\mathit{Sun}) = \mathit{drawSunD3D}$
$\alpha_{d3d}(\mathit{Spaceship}) = \mathit{drawSpaceshipD3D}$

Table 3 Example of functions α for two simplified graphics systems

non-visual geometric systems. For example, a non-visual system could be the sound system. So, a new function $\alpha_{sound}(\mathit{Spaceship}) = \mathit{soundOfSpaceship}$ can be defined for the same alphabet P, where $\mathit{soundOfSpaceship}$ is a function which plays the sound of the spaceship while it is moving in the scene.

Previous examples show that the function α provides the abstraction needed to homogenize the different implementations of a rendering system. Therefore, only a descriptive string is needed to run the same scene on different graphics and non-graphics systems.

So far, only Euclidean geometric systems have been considered. But, the function α has no restrictions on the geometric system that can be applied as long as there is a definition for this primitive. Thus, it could also implement the movement of a robot, the reflection of a material and so on. Moreover, the definition of the function α could also describe systems to write different file formats (VRML, DWG, DXF, XML, etc.), to send strings in a network, etc.

4.1.2 Semantic Function for Transformations (Rule 5)

Rule 5 defines the syntax for transformations. The scope of a transformation is limited by the symbols "()". Two functions are used to describe the semantics of a transformation. These functions are:

$$\begin{aligned} \beta &: T \to G \\ \delta &: T \to G \end{aligned} \tag{3}$$

β represents the beginning of the transformation. It is carried out when the symbol "(" is processed and it has to take into account the previous symbol of the set T. The function δ defines the end of the transformation which has previously been activated by the function β. It is carried out when the symbol ")" is found.

These two functions have the same features that the function α, but they are applied to the set of transformations T, using the same geometric system G.

In the planetary system example (Table 2) there are three terminal symbols which represent transformations of rotation, translation and scaling. $Rotate_{\gamma,N_x,N_y,N_z}$ rotates the object around the axis given by the vector (N_x, N_y, N_z) an angle given by γ. $Translate_{x,y,z}$ translates the object (x, y, z) units. $Scale_s$ performs a scaling of the three coordinates a factor given by s. Table 4 shows how to implement some of the functions β and δ for three examples of visual and non-visual systems. A very simplified notation is used to represent the common functions of OpenGL, Direct3D and a sound system to manage the transformations.

$\beta_{gl}(Translate_{x,y,z}) = pushTransfGL; translateGL(x, y, z)$
$\delta_{gl}(Translate_{x,y,z}) = popTransfGL$

$\beta_{d3d}(Rotate_{\gamma,x,y,z}) = pushTransfD3D; rotateD3D(\gamma, x, y, z)$
$\delta_{d3d}(Rotate_{\gamma,x,y,z}) = popTransfD3D$

$\beta_{sound}(Scale_s) = pushTransfSound; changeVolume(s)$
$\delta_{sound}(Scale_s) = popTransfSound$

Table 4 Example of functions β and δ for simplified OpenGL, Direct3D and a sound system

A new function φ is defined using the set of primitives P, the set of transformations T and the functions α, β and δ. Given a string $w \in L(M)$ and using only symbols of P and T, this function φ runs the sequence of primitives and transformations in the geometric system G. It is defined as:

$$\varphi(w) = \left\{ \begin{array}{ll} \alpha(w) & \textit{if } w \in P \\ \beta(t); \varphi(v); \delta(t) & \textit{if } w = t(v) \wedge v \in L(M) \wedge \\ & \wedge\, t \in T \\ \varphi(u); \varphi(v) & \textit{if } w = u \cdot v \wedge u, v \in L(M) \end{array} \right\} \tag{4}$$

One of the most important features of this system is the independence from a specific graphics system. The definition of the functions α, β and δ provides the differences in behaviour (as it can be seen in Tables 3 and 4). These functions encapsulate the implementation details which may differ for different systems, such as a rendering system, a sound system, a geometric calculation system, and so on. Therefore, in the development of strings to define virtual worlds is not necessary to consider the special features of the geometric system. In addition, these strings may be used on all the systems which implement these functions.

4.1.3 Semantic Function for Actors (Rule 4)

Rule 4 of the grammar M refers to actors, which are the dynamic part of the system. The semantics of the actor is a function which defines its evolution in time. For this reason, the semantic function is called *evolution function* λ and it is defined as:

$$\lambda : A^D_{ATTR} \times E^D \to L(M) \tag{5}$$

where E^D is the set of events for the set of all the event types D. Some deeper aspects about events will be discussed in Sections 4.2 and 4.3.

The function λ has a different expression depending on its evolution over time. However, a general expression can be defined. Let $H = \{h_0, h_1, \ldots, h_n\} \subseteq D$ be the subset of event types which the actor a^H_{ATTR} is prepared to respond to. The general expression for λ is:

$$\lambda(a^H_{ATTR}, e^h) = \left\{ \begin{array}{lll} u_0 \in L(M) & \textit{if} & h = h_0 \\ u_1 \in L(M) & \textit{if} & h = h_1 \\ \vdots & & \\ u_n \in L(M) & \textit{if} & h = h_n \\ a^H_{ATTR} & \textit{if} & h \notin H \end{array} \right\} \tag{6}$$

where $u_0, u_1, \ldots, u_n$ are strings of $L(M)$. This equation means that an actor a^H_{ATTR} can evolve, that is, it is transformed into another string u_i when it responds to an event e^h which the actor is prepared to respond to. However, the actor remains unchanged when it is not prepared to respond to this event type.

Let us suppose a new terminal symbol for the planetary system example. This new symbol, *SpaceshipActor*, is an actor that implements the movement of the spaceship to travel from a planet to another one. This actor is prepared to react to two event types: a new frame event (named fr) and a travel event (named tr). As it will be explained in Section 4.2, the events can include data, so e^{fr} includes the current frame number and e^{tr} includes the destiny position of the spaceship. The *SpaceshipActor* has also two attributes stored as tuples: $\langle cpos, dpos \rangle$ which store, respectively, the current and the destiny position as (x, y, z) coordinates. The evolution function λ in this case could be defined as in Table 5.

If the system receives a frame event e^{fr} and $cpos \neq dpos$, the actor *SpaceshipActor* evolves changing its current position *cpos* in accordance with the destiny position *dpos* and the frame number stored in the event (function ρ in Table 5 would perform this calculation by interpolation, for instance). If the frame event is

$$\lambda(\mathit{SpaceshipActor}^{fr,tr}_{\langle cpos,dpos\rangle},\ e^h) =$$

$$\left\{\begin{array}{ll} \mathit{SpaceshipActor}^{fr,tr}_{\langle \rho(cpos,dpos,e^{fr}),dpos\rangle} & \\ & \mathit{if}\quad h = fr \wedge cpos \neq dpos \\ \mathit{SpaceshipActor}^{fr,tr}_{\langle cpos,dpos\rangle} & \\ & \mathit{if}\quad h = fr \wedge cpos = dpos \\ \mathit{SpaceshipActor}^{fr,tr}_{\langle cpos,e^{tr}\rangle} & \\ & \mathit{if}\quad h = tr \\ \mathit{SpaceshipActor}^{fr,tr}_{\langle cpos,dpos\rangle} & \\ & \mathit{otherwise} \end{array}\right\}$$

Table 5 Example of function λ for a planetary system

received and $cpos = dpos$, the spaceship is not flying, so there is no change in the actor description. If the actor receives a travel event e^{tr} its destiny position is set to the destiny position stored in the event. Finally, if other events are received the actor must remain unchanged.

The result of the function λ may or not contain the own actor or it may even generate new actors. For example, let us suppose that the spaceship can be destroyed. In this case, when a *destruction event* is received, the evolution function must generate a new string where the *SpaceshipActor* does not appear anymore. Moreover, a spaceship could also launch a missile as a response to a *launch missile event*. In this case, the new string would include a new actor to represent a moving missile.

The function λ can define a recursive algorithm which, given a string of $L(M)$ and a sequence of events, describes the evolution of the system at a given frame. This function is called *function of the system evolution* and it is represented by η. Its definition requires a sequence of sorted events $S = e^1 \cdot e^2 \cdot e^3 \ldots e^n$, where every $e^i \in E^D$. The creation and management of the events sequence will be discussed in Section 4.3. The system evolution function is defined as:

$$\eta(w,S) = \left\{\begin{array}{lll} w & \mathit{if} & w \in P \\ t(\eta(v,S)) & \mathit{if} & w = t(v) \\ \prod_{e^i\in S} \lambda(a^H_{attr}, e^i) & \mathit{if} & w = a^H_{attr} \\ \eta(u,S)\cdot\eta(v,S) & \mathit{if} & w = u\cdot v \end{array}\right\} \qquad (7)$$

The operator $\prod_{e^i\in S} \lambda(a^H_{attr}, e^i)$ concatenates the strings generated by the function λ. It is important to note that the strings which are only composed of transformations and primitives are not processed. These strings are not part of the system evolution. This fact is important for the system optimization, because the evolution is carried out only for the strings which contain actors, that is, the dynamic part of the system.

The functions λ can be classified in accordance with the strings which are obtained during the evolution of the system. One important type of λ functions are those which return strings with only primitives and transformations, because they are essential to define the visualization of actors. As it was explained before, the functions α, β and δ do not admit strings with actors. Therefore, actors must be first converted into strings made up only of primitives and transformations. This conversion is carried out by a type of function λ called *visualization function*, or function θ, which is defined as:

$$\theta : A^D_{ATTR} \times E^V \rightarrow L(M') \qquad (8)$$

where $V \subseteq D$, $E^V \subseteq E^D$ are events created in the visualization process, and $L(M')$ is a subset of the language $L(M)$, made up of the strings with no actors. Let $H \cap V = \{v_0, v_1, \ldots, v_n\} \subseteq D$ be the subset of visual event types which the actor a^H_{ATTR} is prepared to respond to. The expression of the visualization function is defined as:

$$\theta(a^H_{ATTR}, e^v) = \left\{\begin{array}{lll} z_0 \in L(M') & \mathit{if} & v = v_0 \\ z_1 \in L(M') & \mathit{if} & v = v_1 \\ \vdots & & \\ z_n \in L(M') & \mathit{if} & v = v_n \\ \epsilon & \mathit{if} & v \notin H \cap V \end{array}\right\} \qquad (9)$$

There are small differences between λ and θ. The first one is that θ returns strings belonging to $L(M')$. The second is that θ returns an empty string if the event does not match. So, the actor will not have representation for that event.

In the planetary system example, the actor *SpaceshipActor* can respond to a new event type *draw*: draw the spaceship. The function θ for this actor could be defined as in Table 6

$$\theta(\mathit{SpaceshipActor}^{fr,tr,draw}_{\langle cpos,dpos\rangle},\ e^v) =$$

$$\left\{\begin{array}{ll} \mathit{Translate}_{cpos}(\mathit{Spaceship}) & \mathit{if}\quad v = \mathit{draw} \\ \epsilon & \mathit{if}\quad v \neq \mathit{draw} \end{array}\right\}$$

Table 6 Example of function θ for a planetary system

In this example, when a e^{draw} event is received, a new string containing only primitives and transformations is generated. Specifically, a *Spaceship* primitive is created to draw the spaceship. This primitive is translated for the spaceship to be drawn on its current position.

The set of event types V (visualization events) may correspond to different output devices or views of the system. For instance, to design a system to filter out certain elements in order that they become invisible, it is only necessary to avoid reacting to those events. Moreover, they can also be used to change the visualization type, depending on the window type, or on the output device and its features.

As with the function λ, an algorithm is defined for θ. It returns a string of the language $L(M')$ given a string $w \in L(M)$ and a sequence of ordered visualization events $S' = e^1 \cdot e^2 \cdot e^3 \ldots e^n$, where every $e^i \in E^V$ and $S' \subseteq S$. This function is called *function of system visualization* π and it is defined as:

$$\pi(w, S') = \left\{ \begin{array}{lll} w & \textit{if} & w \in P \\ t(\pi(v, S')) & \textit{if} & w = t(v) \\ \prod_{e^i \in S} \theta(a^H_{ATTR}, e^i) & \textit{if} & w = a^H_{ATTR} \\ \pi(u, S') \cdot \pi(v, S') & \textit{if} & w = u \cdot v \end{array} \right\} \quad (10)$$

This function is the same as the function η except that λ is replaced by θ, and that events can only belong to V. If the event does not belong to V, the function π returns an empty string.

4.1.4 Semantic Functions for OBJECT, OBJECTS and WORLD (Rules 1, 2 and 3)

The semantic function of these rules breaks down the strings and converts them into substrings. The functions specified above are carried out depending on whether the symbol is an actor, a transformation or a primitive. Therefore, the semantic function of WORLD is a recursive function which breaks down the string of the WORLD and converts it into substrings of OBJECTS. Then, these substrings are in turn broken down into substrings of OBJECT. And for each substring of OBJECT, depending on the type of the object, the semantic function of ACTOR, PRIMITIVE or TRANSFORMATION is run.

4.2 Activity and Events

In VR systems some mechanisms must be established to model the activity of the graphics system. This activity can appear during the action of an actor, between several actors with a certain relationship (for example, an actor which is composed of other actors —which are their children— and there is some activity between them), and between input devices and scene's strings. Each type of activity is different, but they have some features in common, and they are going to be established.

As said before, the actors activity is carried out when a certain type of event is produced with some specific data. But not all activities are directly run when an event is received. There are events that only run their activity when certain conditions are satisfied, depending on the actor state. This condition is not defined at the object that runs the activity, but at the event. The following event definition is established: e^d_c *is defined as an event of type* $d \in D$ *with data* e*, which is carried out only when the condition* c *is fulfilled.*

For simplicity, when there is not any condition, that is, c=*TRUE*, the event is represented by e^d.

Let us notice that the origin of events is not identified. In fact, the origin is not important, but the event type and its data. An event is related with the activity of an actor only by the type of event. Therefore, the event e^d carries out the activity of type d defined in the evolution function of the actor a^d_{attr}.

4.3 Input Devices and Event Generators

It is necessary to establish the independence between the system and the input devices. So, the events needed to make the system respond to a set of input devices must be defined. It means that there may be actions on input devices (buttons, mouse wheel, movements and so on) which generate one or more events, and others which do not generate any. The input devices need not be only physical devices, they may be software devices which, depending on the system state, may or may not generate events. For example, a mouse could be used as input device to move the objects in the scene. However, there has not to be a mouse event associated to the object. Instead, there should be a generic event of movement associated to the object and the mouse has an event generator which creates events of movement. On the other hand, an example of software device may be an algorithm for detecting collisions. In this case, an event can be generated when a collision is detected.

A new function called *event generator* is defined as: *Let* $C^d(t)$ *be a function which creates a sequence of ordered events of type* d *at the time instant* t*, where* $d \in D$ *and* D *is the set of event types which can be generated by the system.* This function is:

$$C^d : Time \rightarrow (E^D)^* \quad (11)$$

In the previous definition, it should be noticed that events are generated in the time instant t. It is due to synchronization purpose. The event generator can generate several or no events at a given moment.

This definition makes the system independent from the input devices. Thus, event generators can be created for a mouse, a VR glove, a keyboard or any other input device, and all of them can generate the same type of events. The most interesting fact about the implementation of the event generator is that the device-dependent code is encapsulated and separated from the rest.

One problem is that different event generators can create the same type of events. So, a priority order among event generators must be established to avoid ambiguities. Given two generators C_i and C_j which create the same event, if $i < j$, then the events generated by C_i will have a higher priority. For example, the following priority order could be established among input devices: $1st$ joystick, $2nd$ mouse and $3rd$ keyboard. This ambiguity is due to the fact that the system must be implemented for all the possible devices. If there are several devices, only one prevails. The priority order can be established using different criteria: the level of immersion into the system, the ease of use and so on.

The process which obtains the sequence of events produced by input devices and their associated event generators is defined as follows:

Let $C^* = \bigcup_{i=0}^{n} C_i$ be the set of all the event generators.

The function $add(S, e^d)$ concatenates an event e^d in a sequence S only if this event is not already in the sequence. It is defined as:

$$add(S, e^d) = \left\{ \begin{matrix} S \cdot e^d & if & e^d \notin S \\ S & if & e^d \in S \end{matrix} \right\} \qquad (12)$$

The function $addSeq(S, R)$ concatenates all the events of sequence Z into sequence S only if those events are not already in the sequence. It is defined as:

$$addSeq(S, Z) = \prod_{e^d \in Z} add(S, e^d) \qquad (13)$$

The function $collect(S, C^*, t)$ collects all the events from all the generators in a time instant t and concatenates them in a sequence S. If an event already exists, then it is not inserted in the sequence considering the generator priority. This function is defined as:

$$collect(S, C^*, t) = \prod_{i=1}^{n} addSeq(S, C_i(t)) \qquad (14)$$

The sequence S obtained with this function is the sequence of sorted events referred in Section 4.1.3.

4.4 System Algorithm

Once all the elements involved in the model have been defined, the algorithm which carries out the entire system can be established. It defines the system evolution and its visualization at every time instant 't' or frame. The algorithm of virtual worlds generation is defined as:

```
1   $w = w_0$
2   $t = 0$
3   $while\ w \neq \epsilon\ do$
4      $S = \emptyset$
5      $collect\ (S, C^*, t)$
6      $Z =\ extract\ visual\ events\ from\ S$
7      $w_{next} = \eta(w, S)$
8      $v = \pi(w, Z)$
9      $g = \varphi(v)$
10     $w = w_{next}$
11     $t = t + 1$
12  $end\ while$
```

Where:

- w_0 is the initial string of the system.
- $D = \{$ Set of all the types of possible events in the system $\}$.
- $V = \{$ Set of all the types of visual events $\}$ where $V \subseteq D$.
- $C^* = \{$ All the event generators which generate events of type D $\}$.
- g is the output device.
- S is a sequence of all the events generated by the system at instant t.
- Z is a subsequence of S, and it includes all the events from visual devices. These events are the input of the visual algorithm π.

The first two steps initialize the system. The initial state w_0 is introduced to the system and the first frame is set to 0.

Steps 4, 5 and 6 manage the system events. In Step 4, an empty sequence of events is created. Then, through the *collect* function, all the generators are called to insert all the events in the list S. In Step 6, the visual events are added to a new sequence Z to be the input of the visual algorithm π. However, all the events S are the input of the evolution algorithm η.

In Step 7, the evolution algorithm η is called with the current string w and the list of events (S). The output is the string for the next frame or instant.

In Steps 8 and 9, the visualization of the system is performed. First, actors are transformed into primitive and transformations through the use of the visualization algorithm π which is called with the visual events (Z). Finally, the function φ is called to display the current state of the system into the rendering engine g.

In Steps 10 and 11, the next iteration is prepared. w is assigned to w_{next} and the time instant (or current frame) is increased in 1.

This process is done while the current string does not satisfy the condition of completion, that is, while the string at this moment, w, is not empty (Step 3). Therefore, to finish the main algorithm the function η must just return an empty string. This empty string can be generated by a special event which is created when the system must stop.

It must be noticed that Step 7 can be exchanged with Steps 8 and 9, because they do not share any data. This feature is very important for the parallel implementation of the algorithm. So, Step 7 and Steps 8 and 9 can be divided into two parallel tasks. This type of optimization has a significant impact on the final system, which leads to a faster system performance.

The diagram of the virtual world generation algorithm is shown in Figure 1.

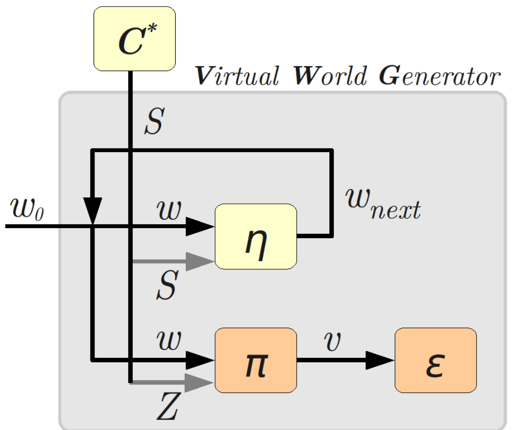

Fig. 1 Virtual worlds generation algorithm

This formalization of the system has two main consequences. First, the scene definition is separated from the hardware-dependent characteristics of components. The functions α, β and δ provide the independence from the visualization system, and the event generators provide the independence from the hardware input devices. Secondly, due to the fact that there is an specific scheme to define the features of a system, the different system elements can be reused easily in other areas of application.

5 Case Study

This example is an application that simulates fires in forests with animals caused by lightning. It could be used to analize the best distribution of trees to reduce the number of burnt trees [19]. It is important to notice that we are not actually interested in the problem of fire simulation, but rather in defining a graphics system using the proposed model and all the concepts presented in this paper. Therefore, it studies the ease of system modeling, expansion, portability and system independence from hardware and other aspects of graphical representation.

5.1 Problem Description

Let (i, j) be the cells of a 2D grid representing a squared world. At each cell, a tree can grow with a given probability g. Bolts of lightning can also fall on a (i, j) position with a probability f. In this case, if there is a tree, it will burn as well as the trees around it, in a chain reaction. Moreover, there are rabbits that can move through the forest. Theses rabbits will be dead if they occupie the same cell as a burning tree. An example can be seen in Figure 2.

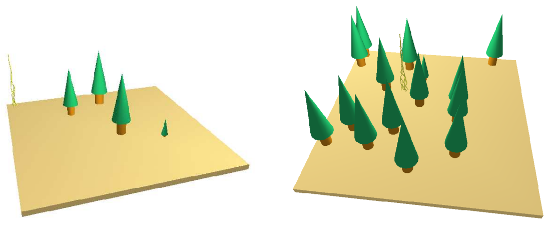

Fig. 2 Examples of different simulation states

To model this example with our proposal, four main elements need to be defined. First, the graphic primitives must be defined to show the visual elements on the rendering system. Second, it is necessary to define events to run actor activities. These events include the necessary information to develop their activity. After that, it is necessary to design the event generators that separate the interactive system from the origin of the events (mouse, keyboard, physics engine and so on). Finally, it is necessary to define the actors included in the scene. These elements are used by the system to evolve over time.

5.2 System Formalization

Following the steps described above, the different elements of the system are defined.

5.2.1 Primitives and Transformations

The graphical elements that can appear in the scene are: the grid (that is, the ground), the trees (they can have three different graphical representation depending on their state), the bolts of lightning and the rabbits. Therefore, six primitives are needed to represent these elements. They are shown in Table 7.

Primitive	*Description*
PrimTree	Draw a tree
PrimBurningTree	Draw a burning tree
PrimGrowningTree	Draw a growing tree
PrimBoltLightning	Draw a bolt of lightning
PrimRabbit	Draw a rabbit
PrimGridNN	Draw a grid of NxN

Table 7 Definition of primitives

To manipulate the primitives two transformations are needed: translation (used to move trees, bolts and rabbits), and scaling (used to simulate the growth of a tree). The transformations are shown in Table 8:

Transformations	*Description*
$Translate_{(i,j)}$	Translate to (i, j) position of grid
$Scale_s$	Scale a s factor

Table 8 Definition of transformations

The functions α, β and δ define the semantics for these primitives and transformations. They are implemented as functions containing calls to a graphics library (in this case OpenGL). This way, the number of primitives and transformations can be easily extended just implementing the corresponding functions α, β and δ.

5.2.2 Events

Events are used to produce the necessary activity of the system. They are described by an identifier and a set of data. Their aim is to produce certain activities of the actors that compose the scene. The events defined for this example are shown in Table 9.

Type	*Meaning*	*Associated data*
frame	Event generated every frame	Increasing time since previous frame event
tree	Create a tree at a given position	Grid position (i, j) where the tree is created
bolt	Create a bolt of lightning at a given position	Grid position (i, j) where the bolt of lightning is created
rabbit	Create a rabbit at a given position	Rabbit identifier and the grid position (i, j) where the rabbit is created
kill	Kill a rabbit with a specific identifier	Identifier of the rabbit to kill
draw	Draw using the graphics system	Void

Table 9 Events definition

5.2.3 Event Generators

The next step is to define the event generators. They create the events described in the previous section. All generators share a table that has information about the elements in the forest. This table, called *Positions Table (PT)*, has three fields: the element unique identifier (id), the element position (pos) and the element type (tree, burning tree, bolt of lightning or rabbit).

1. Frame event generator: This generator has the responsibility of animating the system. Every time instant, it generates an event e^{frame} that is usually associated with some type of transformation of primitives. Thus, it changes the appearance of an actor over time. In this example, the transformation will be the scaling of an element.

$$C_{frame} = \{e^{frame} \text{ } \textit{each frame}\}$$

2. Collision event generator: They analyze if a rabbit is in the same cell as a burning tree or if a bolt of lightning strike up a tree. In this case, the generator sends a kill event to the rabbit or an event to the tree to burn. The collision event generator uses the *isBurning* function to calculate whether a rabbit should die. This function is true if there is a burning tree in the same position as a rabbit. The collision event generator also calculates if a tree should be burned. In this case, it uses the *IsBolt* function. Theses function uses the information stored in the *PT* table, and returns true if there is a bolt of lightning at the same position as a tree.

$$C_{collision} = \left\{ \begin{array}{ll} e^{kill} & \forall x \in PT \\ & \wedge\ x.type = rabbit \\ & \wedge\ IsBurning(x.pos) = true \\ e^{burn} & \forall x \in PT \\ & \wedge\ x.type = tree \\ & \wedge\ IsBolt(x.pos) = true \end{array} \right\}$$

e^{kill} has the rabbit identifier that must be killed and e^{burn} has the tree identifier that must be burned.

3. Forest event generator: This generator produces events to create trees, bolts of lightning and rabbits. Taking into account the previous specification, trees are created with a probability g, lightning with a probability f and rabbits with a probability r. This generator is defined as:

$$C_{forest} = \left\{ \begin{array}{ll} e^{tree} & with\ probability\ g \\ e^{bolt} & with\ probability\ f \\ e^{rabbit} & with\ probability\ r \end{array} \right\}$$

All the events created by this generator have an associated unique element identifier. They also add a new entry to the PT table.

4. Visualization event generator: This generator is necessary to visualize all the elements in the scene. This means that every time a drawing order is received, the generator captures this order and produces an event. At the same time, it sends the elements of the scene to draw in the graphics system. It is defined as:

$$C_{visualization} = \{e^{draw}\ each\ drawing\ cycle\}$$

5.2.4 Actors

The last step is to specify the actors which compose the dynamic part of the system. As explained in section 4.1.3, an actor is defined by its evolution function λ. If it has a graphical representation, it also needs a function θ to generate the primitives and the transformations.

There are four actors in the scene: the forest, which represents the ground and manages all the other actors, the trees, the bolts of lightning and the rabbits.

The actor $Forest^{tree,bolt,rabbit,draw}$ can respond to four events. Three of them (create a tree, create a bolt of lightning and create a rabbit) are involved in its evolution function and the other one (draw the forest) in the visualization function. So, the evolution function for $Forest^{tree,bolt,rabbit,draw}$ includes the creation of a tree when it receives the event e^{tree}, the creation of a bolt of lightning when it receives the event e^{bolt} and the creation of a rabbit when it receives the event e^{rabbit}. These events also contain the identifiers of the trees, the bolts of lightning or the rabbits to be created. This evolution function is defined in Table 10. The e^{draw} event has been eliminated in the table for simplicity and $Forest^{tree,bolt,rabbit}$ is represented as $Forest^{t,b,r}$. The drawing function θ will be defined later.

Actor	$Forest^{t,b,r}$
Description	Represents the forest
$\lambda(Forest^{t,b,r}, e^h) = \left\{ \begin{array}{ll} Tree^{frame,burn}_{\langle gr, e^{tree}.id, 1\rangle} \cdot Forest^{t,b,r} & if\ h = tree \\ Lightning^{frame}_{\langle e^{bolt}.id, 1\rangle} \cdot Forest^{t,b,r} & if\ h = bolt \\ Rabbit^{frame,kill}_{\langle \cdot, e^{rabbit}.id\rangle} \cdot Forest^{t,b,r} & if\ h = rabbit \\ Forest^{t,b,r} & if\ h \notin \{tree, bolt, rabbit\} \end{array} \right\}$	

Table 10 Evolution function of the actor *Forest*

The actor $Tree^{frame,bolt,draw}_{\langle st,id,nframe\rangle}$ (see Table 11) can also respond to three events (the frame event, the creation of a bolt of lightning and the draw event) and has three attributes. The first attribute, the state (st), defines whether the tree is *growing*, *burning* or *adult*. The second attribute corresponds to the (i, j) position of the tree within the grid. The last attribute is used to store the current tree animation frame. The tree evolves depending on the received event and on its current attributes.

For instance, when a tree is growing and receives a *frame* event, the growing animation advances, unless the final animation frame (N_{frames}) is reached. On the contrary, when the tree state is *burning*, the animation is reversed by decreasing the $nframe$ attribute until 0 is reached (some other details about this animations will be discussed later). Finally, if a bolt of lightning is created at the same position as a tree, the collision event generator sends a *burn* event to change the tree state to *burning*.

Table 12 defines the evolution function of a bolt of lightning. In this case, it can respond to the $frame$ and the $draw$ events. It has two attributes: the bolt of lightning identifier and the animation frame. The lightning animation is defined similarly to that of the trees.

The last actor to consider is the rabbit. Its evolution function is defined in Table 13. This actor decides the direction to take whenever he receives a *frame* event. This behaviour is controlled by a function called $DecideDesire(e^{frame})$, which form part of the artificial intelligence engine. In this case, this function implements a random behaviour which could be left,

Actor	$Tree^{frame,burn}_{\langle st,pos,nframe\rangle}$
Description	Represents a tree

$$\lambda(Tree^{frame,burn}_{\langle st,pos,nframe\rangle}, e^h) =$$

$$\left\{\begin{array}{ll} Tree^{frame,burn}_{\langle growing,id,nframe+1\rangle} & \textit{if } h = frame \\ & \wedge st = growing \\ & \wedge nframe < N_{frames} \\ Tree^{frame,burn}_{\langle adult,id,N_{frames}\rangle} & \textit{if } h = frame \\ & \wedge st = growing \\ & \wedge nframe \geqslant N_{frames} \\ Tree^{frame,burn}_{\langle burning,id,nframe-1\rangle} & \textit{if } h = frame \\ & \wedge st = burning \\ & \wedge nframe > 0 \\ \epsilon & \textit{if } h = frame \\ & \wedge st = burning \\ & \wedge nframe = 0 \\ Tree^{frame,burn}_{\langle burning,id,nframe\rangle} & \textit{if } h = burn \\ & \wedge e^{burn}.id = id \\ Tree^{frame,burn}_{\langle st,pos,nframe\rangle} & \textit{if } h \notin frame, burn \end{array}\right\}$$

Table 11 Evolution function of the actor *Tree*

Actor	$Lightning^{frame}_{\langle id,nframe\rangle}$
Description	Represents the bolt of lightning

$$\lambda(Lightning^{frame}_{\langle id,nframe\rangle}, e^h) =$$

$$\left\{\begin{array}{ll} Lightning^{frame}_{\langle id,nframe+1\rangle} & \textit{if } h = frame \\ & \wedge\ nframe < N_{frames} \\ \epsilon & \textit{if } h = frame \\ & \wedge\ nframe \geqslant N_{frames} \\ Lightning^{frame}_{\langle id,nframe\rangle} & \textit{if } h \neq frame \end{array}\right\}$$

Table 12 Evolution function of the actor *Lightning*

right, front, back or stop. This function could be replaced by a more advanced behavior. For example, run in the opposite direction when the actor is next to a burning tree.

Actor	$Rabbit^{frame,kill}_{\langle desire,id\rangle}$
Description	Represents a rabbit

$$\lambda(Rabbit^{frame}_{\langle desire,id\rangle}, e^h) =$$

$$\left\{\begin{array}{ll} Rabbit^{frame,kill}_{\langle DecideDesire(e^{frame}),id\rangle} & \textit{if } h = frame \\ \epsilon & \textit{if } h = kill \end{array}\right\}$$

Table 13 Evolution function of the actor *Rabbit*

Finally, Table 14 shows the definition of the drawing functions θ. These functions are used to obtain the set of primitives and transformations representing the visual aspect of the actors. They are helped by *PosPT(id)* (which returns the position of an element) and *PosWithDesirePT(desire,id)* (which returns the position of an element and updates its position using the last decision). For example, if the decision is going up, the element will increment its Y position; if the decision is going down, it will decrement the Y position; and so on.

$$\theta(Forest^{draw}, e^h) = \left\{\begin{array}{ll} PrimGridNN & \textit{if } h = draw \\ \epsilon & \textit{if } h \neq draw \end{array}\right\}$$

$$\theta(Tree^{draw}_{\langle st,id,nframe\rangle}, e^h) =$$

$$\left\{\begin{array}{ll} Translate_{PosPT(id)}(S_s(PGTree)) & \wedge\ h = draw \\ & \textit{if } st = growing \\ Translate_{PosPT(id)}(S_s(PBTree)) & \wedge\ h = draw \\ & \textit{if } st = burning \\ Translate_{PosPT(id)}(PrimTree) & \wedge\ h = draw \\ & \textit{if } st = adult \\ \epsilon & \textit{if } h \neq draw \end{array}\right\}$$

Where:

$PGTree = PrimGrowingTree$

$PBTree = PrimBurningTree$

$$\theta(Ligthning^{draw}_{\langle id,nframe\rangle}, e^h) =$$

$$\left\{\begin{array}{ll} Translate_{PosPT(id)}(S_s(PrimBoltLightning)) & \textit{if } h = draw \\ \epsilon & \textit{if } h \neq draw \end{array}\right\}$$

$$\theta(Rabbit^{draw}_{\langle desire,id\rangle}, e^h) =$$

$$\left\{\begin{array}{ll} Translate_{PosWithDesirePT(desire,id)}(PrimRabbit) & \textit{if } h = draw \\ \epsilon & \textit{if } h \neq draw \end{array}\right\}$$

Table 14 Definition of the drawing function π

In Table 14, the actor *Forest* is drawn as a *PrimGridNN* primitive. In the case of trees, the used primitives are *PrimGrowingTree*, *PrimBurningTree* or *PrimTree* depending on the tree state. In the case of *growing* or *burning* states, a scaling transformation is also introduced. The growing scale factor s is defined by the expression $s = \frac{nframe}{N_{frame}}$, so the higher the $nframe$ value, the higher the factor s. From this expresion and from the function in Table 11, it can be deduced that the *Tree* size is increased when it receives the event e^{frame} and it is growing, and when the *Tree* is in burning state, this effect is inverse. In the case of *Lightning* the animation is done in the same way. The rabbit is drawn using the *PrimRabbit* primitive and his drawing process is similar to those already explained. This example shows how the attributes of an actor are used to change its behaviour.

In general, actors animations always depend on *frame*. It modifies the current state of the actor to change its representation and thus, it performs the animation.

5.3 Example of the Evolution of the Algorithm

Table 15 shows the first iterations in the evolution of the algorithm for this example. The algorihtm starts with the initial string $w_0 = Forest^{tree,bolt,rabbit,draw}$. It is the first string to be processed in the algorithm. From this initial state, the system evolves over time (See algorithm scheme in Section 4.4). Two events are always generated at every step to make the system evolve and to draw it on the display: e^{frame} and e^{draw}. Besides, at some iterations, random events to create trees and bolts of lightning are also generated.

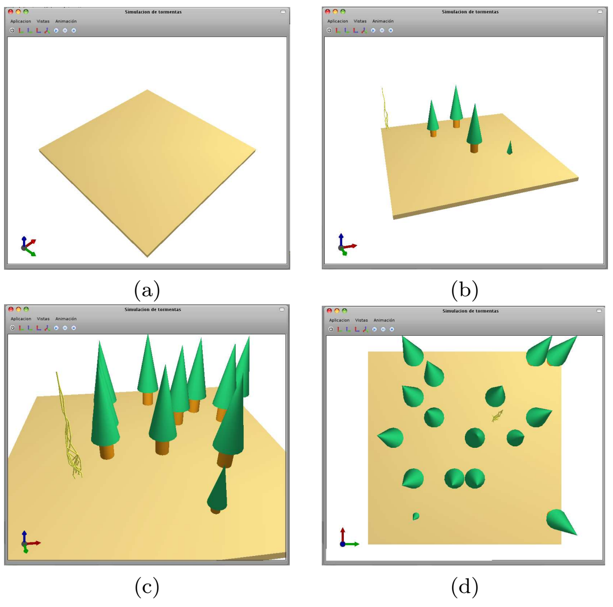

Fig. 3 Some application views where it can be seen the evolution of forrest

6 Conclusions and Future Work

A new model of Virtual Worlds generation has been presented. The major goal of this system is to separate the scene definition from the hardware-dependent characteristics of the system devices.

First, a language which details the elements of the system using context-free grammars is defined. Moreover, new functions used to represent the evolution of the elements in time and to extract their graphic representation are established. This representation is sent to visual devices using functions which separate the hardware-dependent implementation of the visual device from the graphic description of the element. This way, it removes the dependence of the system from the graphics devices.

The separation of system definition from input devices is made by event generators. These generators create a layer between the input hardware and the representation of the system. The actions generated by input devices are linked using the model of events.

In general, the whole model separates the hardware-dependent parts of devices from the formal definition of a VR system. In order to carry out this separation, different mathematical models are used to transform device actions, both visual and input devices, into more general actions. The system must be able to identify these general actions regardless of the origin of the action. It is achieved using abstraction.

This model has several advantages: firstly, input devices can be replaced by other devices or even software-simulated devices. Secondly, the different elements of the system can be easily reused. Thirdly, the representation of the elements can be visual or non-visual, or even it may be different depending on the display device or the user needs. Thus, if a device has specific characteristics, the representation can always be adapted to the device.

Using the proposed system, new physics engines which use the elements of the scene can be easily designed. It is achieved by setting up different types of event generators. Depending on the physical features of the device, it activates the activity of the actor needed to react to that physical process. For example, if an actor has to react to collisions, the event generator of this type calculates the collisions between elements by extracting the scene geometry from the graphics engine (it uses the implementation of the functions α, β and δ to calculate the bounding box of the elements). Then, it generates the events needed to react to such collisions. This event generator could be implemented with hardware if the system allows it.

The AI engine can be implemented in the evolution function of the actors. Actors use these functions to make decisions according to their current status. Moreover, events trigger activities which can change the status and the behaviour of actors. The integration between the physics engine and the AI engine is guaranteed because both engines are connected by events.

The model presented in this work is currently under development. It is pretended to continue developing several issues. One point to investigate —which is also introduced in this article— is the optimization of the algorithm and its parallelization. The definition of the system through strings facilitates the possibility of parallel algorithms. From another point of view, strings represent the states of the system and its evolution. This evolution may change through mutations, so different evolutive solutions may be conceived to design new systems. We also consider the possibility of a new type of events which are activated with a certain probability. For example, if an actor is defined as $a^{d,h}$ and it receives the event e^d, then the function associated with this event will be carried out only with a certain probability.

In conclusion, the main aim has been to design a reusable and generic system, which can be easily increased, adapted, and improved. It is also important that the core of the system (the evolution in time) is independent from its representation and the elements which it interacts with.

References

1. Aho, Alfred V. and Ullman, Jeffrey D.: The Theory of Parsing, Translation, and Compiling. Volume I: Parsing. Prentice-Hall, Inc. Englewood Cliffs, N.J. ISBN: 0-13-914556-7 (1972).
2. Bartle, Richard A.: Designing virtual worlds. New Riders Pub (2004)
3. Burdea, Grigore C. and Coiffet, Philippe.: Virtual Reality Technology, Second Edition. Wiley-IEEE Press (2003)
4. CILib (Computational Intelligence Library): http://cilib.sourceforge.net (Accessed October 2009)
5. Davis, Martin; Sigal, Ron and Weyuker, Elaine J.: Computability, Complexity, and Languages, Fundamentals of Theoretical Computer Science, 2nd ed. San Diego: Elsevier Science (1994)
6. DirectX: http://www.microsoft.com/windows/directx/default.mspx (Accessed October 2009)
7. DMLW - 3D Markup Language for Web: http://www.3dmlw.com (Accessed October 2009)
8. EO Evolutionary Computation Framework: http://eodev.sourceforge.net (Accessed October 2009)
9. Google O3D API: http://code.google.com/apis/o3d/ (Accessed October 2009)
10. Havok: http://www.havok.com (Accessed October 2009)
11. Hopcroft, John E.; Motwani, Rajeev and Ullman, Jeffrey D.: Introduction to Automata Theory, Languages, and Computation. Addison Wesley, 2nd Edition. ISBN: 0201441241 (2000)
12. Jade - Java Agent DEvelopment Framework: http://jade.tilab.com (Accessed October 2009)
13. Kasik, David J.; Buxton, William and Ferguson, David R.: Ten cad challenges. IEEE Computer Graphics and Applications 25, 81–90 (2005)
14. Kenyon, S. H.: Behavioral software agents for real-time games. IEEE Potentials, vol. 25 (4), 19–25 (2006)
15. Khoo, Aaron and Zubek, Robert: Applying inexpensive AI techniques to computer games. IEEE Intelligent Systems, vol. 17 (4), 48–53 (2002)
16. Laird, John E.: Using a computer game to develop advanced AI. Computer. IEEE Computer Society Press. Vol. 34 (7), 70–75 (2001)
17. Manousakis, Stelios: Musical L-Systems. Master's Thesis – Sonology. The Royal Conservatory, The Hague (2006)
18. Miles, C.; Quiroz, J.; Leigh, R. and Louis, S.J.: Co-evolving influence map tree based strategy game players. IEEE Symposium on Computational Intelligence and Games, 88–95 (2007)
19. Miller, John H. and Page, Scott E.: Complex Adaptive Systems: An Introduction to Computational Models of Social Life. Princeton University Press (2007)
20. Müller, Pascal; Wonka, Peter; Haegler, Simon; Ulmer, Andreas and Gool, Luc Van: Procedural Modeling of Buildings. ACM SIGGRAPH 2006. International Conference on Computer Graphics and Interactive Techniques. 614–623 (2006)
21. Musgrave, F.K.; Kolb, C.E. and Mace, R.S.: The Synthesis and Rendering of Eroded Fractal Terrains, In SIGGRAPH 89 Proceedings, 41–50 (1990)
22. Newton Game Dynamics: http://www.newtondynamics.com (Accessed October 2009)
23. Novak, Jeannie: Game Development Essentials: An Introduction. Second edition. Delmar Cengage Learning (2007)
24. OGRE 3D: Open source graphics engine: http://www.ogre3d.org (Accessed October 2009)
25. Open Dynamics Engine: http://www.ode.org (Accessed October 2009)
26. OpenGL: http://www.opengl.org/ (Accessed October 2009)
27. PAL: Physics Abstraction Layer: http://www.adrianboeing.com/pal/ (Accessed October 2009)
28. Palubicki, Wojciech; Horel, Kipp; Longay, Steven; Runions, Adam and Lane, Brendan; Mech, Radomir and Prusinkiewicz, Przemyslaw: Self-organizing tree models for image synthesis. ACM Transactions on Graphics 28(3), 58:1–10 (2009)
29. Parish, Yoav I. H. and Müller, Pascal: Procedural Modeling of Cities. ACM SIGGRAPH 2001, 301–308 (2001)
30. Perlin, K.: An Image Synthesizer. Computer Graphics (SIGGRAPH 85 Proceedings), 19(3): 287–296 (1985)
31. Physics engine description: http://en.wikipedia.org/wiki/physics_engine (Accessed October 2009)
32. PhysX by AGEIA: http://physx.ageia.com (Accessed October 2009)
33. Prusinkiewicz, P. and Lindenmayer, A.: The algorithmic beauty of plants. Springer-Verlag. ISBN: 978-0387972978. (1990)
34. Prusinkiewicz, Przemyslaw: Simulation Modeling of Plants and Plant Ecosystems. Communications of the ACM vol. 43, no. 7, pp. 84–93 (2000)
35. Prusinkiewicz, Przemyslaw; Erasmus, Yvette; Lane, Brendan: Harder, Lawrence D. and Coen, Enrico: Evolution and Development of Inflorescence Architectures. Science 316(5830), pp. 1452–1456 (2008)
36. Reeves, W.T. and Blau, R.: Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems. ACM SIGGRAPH 85 Proceedings, 19(3): 313-322 (1985)
37. Reynolds, R.G.; Kobti, Z.; Kohler, T.A. and Yap, L.Y.L.: Unraveling ancient mysteries: Reimagining the past using evolutionary computation in a complex gaming environment. IEEE Transactions on Evolutionary Computation, vol. 9 (6), 707–720 (2005)
38. Rozenberg, Grzegorz and Salomaa, Arto: Lindenmayer systems: impacts on theoretical computer science, computer graphics, and developmental biology. Springer-Verlag. ISBN: 0387553207. (1992)
39. Sánchez-Crespo Dalmau, Daniel: Core techniques and algorithms in game programming. New Riders, ISBN: 0-13-102009-9 (2004)

40. Seugling, Axel and Rolin, Martin: Evaluation of Physics Engines and Implementation of a Physics Module in a 3d-Authoring Tool. Master's Thesis in Computing Science. Umea University, Sweden (2006)
41. Sherman, William R. and Craig, Alan: Understanding virtual reality: interface, application, and design. Morgan Kaufmann, ISBN:1-55860-353-0 (2003)
42. Simple DirectMedia Layer (SDL): http://www.libsdl.org (Accessed October 2009)
43. Spicher, Antoine; Michel, Olivier; Cieslak, Mikolaj; Giavitto, Jean-Louis and Prusinkiewicz, Przemyslaw: Stochastic P systems and the simulation of biochemical processes with dynamic compartments. Biosystems 91(3): 458–472 (2008)
44. Strickon, Joshua: Interacting with Emerging Technologies. IEEE Computer Graphics and Applications, vol. 24, no. 1, pp. C2 (2004)
45. The Visualization ToolKit (VTK): http://public.kitware.com/vtk (Accessed October 2009)
46. Wang, Wen; Ma, Xueqiang and Liu, Hong: A Computer _aided Harmonious Architecture Design Method Based on Fractals. Fourth International Conference on Natural Computation, vol. 7, 323–326 (2008)
47. Web3D Consortium. http://www.web3d.org (Accessed October 2009)
48. Wii-Nintendo: http://www.nintendo.es (Accessed October 2009)
49. Wood, Mark F. and Deloach, Scott A.: An overview of the multiagent systems engineering methodology. The First International Workshop on Agent-Oriented software Engineering (AOSE), 207–222 (2000)
50. Wooldridge, Michael: Agent-based software engineering. IEEE Proceedings Software Engineering, 26–37 (1997)
51. Working Model: Design Simulation Technologies: http://www.design-simulation.com (Accessed October 2009)
52. Yannakakis, Georgios; Levine, John and Hallam, John: An evolutionary approach for interactive computer games. In Proceedings of the Congress on Evolutionary Computation, 986–993 (2004)

It.	*Evolution Strings*	*Draw Strings*	*Generated Events*	*Observations*
1	$Forest^{tree,bolt,rabbit,draw}$		$e^{draw}, e^{frame}, e^{tree}$	Initial String. Apart from e^{draw} and e^{frame}, $e^{tree} = (tree0)$ is received (random event, create a tree with tree0 identifier at (1,2) position).
2	$Tree^{frame,bolt,draw}_{\langle gr,tree0,1\rangle}$ $\cdot Forest^{tree,bolt,rabbit,draw}$	$PrimGridNN$	$e^{draw}, e^{rabbit}, e^{frame}$	A tree is generated in the evolution string. The primitive $PrimGridNN$ is added to the draw string. e^{rabbit} is create with rabbit0 identifier at position (2,1)
3	$Tree^{frame,bolt,draw}_{\langle gr,tree0,2\rangle}$ $\cdot Rabbit^{frame,kill,draw}_{\langle \cdot,rabbit0\rangle}$ $\cdot Forest^{tree,bolt,rabbit,draw}$	$Translate_{(1,2)}(S_1(PGTree))$ $\cdot PrimGridNN$	e^{draw}, e^{frame}	The tree animation evolves. The tree is drawn, adding, scaling and translating a $PrimGrowingTree$ primitive. A rabbit is showed with rabbit0 identifier and desire stop.
4	$Tree^{frame,bolt,draw}_{\langle gr,tree0,3\rangle}$ $\cdot Rabbit^{frame,kill,draw}_{\langle left,rabbit0\rangle}$ $\cdot Forest^{tree,bolt,rabbit,draw}$	$Translate_{(1,2)}(S_2(PGTree))$ $\cdot Translate_{(2,1)}(PrimRabbit)$ $\cdot PrimGridNN$	$e^{draw}, e^{frame}, e^{bolt}$	The tree animation goes on evolving. $e^{bolt} = (bolt0)$ is received (random event, create a bolt of lightning at (2,2) position). The rabbit desires to go to left direction
5	$Tree^{frame,bolt,draw}_{\langle adult,tree0,3\rangle}$ $\cdot Rabbit^{frame,kill,draw}_{\langle \cdot,rabbit0\rangle}$ $\cdot Lightning^{frame,draw}_{\langle bolt0,1\rangle}$ $\cdot Forest^{tree,bolt,rabbit,draw}$	$Translate_{(1,2)}(S_3(PGTree))$ $\cdot Translate_{(1,1)}(PrimRabbit)$ $\cdot PrimGridNN$	e^{draw}, e^{frame}	A bolt of lightning is generated. After 3 frames, the tree changes its state to adult. Rabbit changes his position to (1,1) and his next desire is stop.
6	$Tree^{frame,bolt,draw}_{\langle adult,tree0,3\rangle}$ $\cdot Rabbit^{frame,kill,draw}_{\langle up,rabbit0\rangle}$ $\cdot Lightning^{frame,draw}_{\langle bolt0,2\rangle}$ $\cdot Forest^{tree,bolt,rabbit,draw}$	$Translate_{(1,2)}(PrimTree)$ $\cdot Translate_{(1,1)}(PrimRabbit)$ $\cdot Translate_{(2,2)}(S_1(PBLight))$ $\cdot PrimGridNN$	e^{draw}, e^{frame}	The lightning animation evolves. The lightning is drawn, adding, scaling and translating a $PrimBoltLightning$ primitive. The tree primitive is changed to draw an adult tree $PrimTree$. The rabbit desires to go to up.
7	$Tree^{frame,bolt,draw}_{\langle adult,tree0,3\rangle}$ $\cdot Rabbit^{frame,kill,draw}_{\langle eup,rabbit0\rangle}$ $\cdot Lightning^{frame,draw}_{\langle bolt0,3\rangle}$ $\cdot Forest^{tree,bolt,rabbit,draw}$	$Translate_{(1,2)}(PrimTree)$ $\cdot Translate_{(1,2)}(PrimRabbit)$ $\cdot Translate_{(2,2)}(S_2(PBLight))$ $\cdot PrimGridNN$	e^{draw}, e^{frame}	The lightning animation goes on evolving. The rabbit go up another one.
8	$Tree^{frame,bolt,draw}_{\langle adult,tree0,3\rangle}$ $\cdot Rabbit^{frame,kill,draw}_{\langle \cdot,rabbit0\rangle}$ $\cdot Forest^{tree,bolt,rabbit,draw}$	$Translate_{(1,2)}(PrimTree)$ $\cdot Translate_{(1,3)}(PrimRabbit)$ $\cdot Translate_{(2,2)}(S_3(PBLight))$ $\cdot PrimGridNN$	e^{draw}, e^{frame}	The lightning finishes its evolution and it is eliminated from the evolution string.
9	$\cdot Tree^{frame,bolt,draw}_{\langle adult,tree0,3\rangle}$ $\cdot Rabbit^{frame,kill,draw}_{\langle down,rabbit0\rangle}$ $\cdot Forest^{tree,bolt,rabbit,draw}$	$Translate_{(1,2)}(PrimTree)$ $\cdot Translate_{(1,3)}(PrimRabbit)$ $\cdot Translate_{(2,2)}(S_3(PBLight))$ $\cdot PrimGridNN$	e^{draw}, e^{frame}	Finally, the bolt of lightning is also eliminated from the draw string.
...	And so on...			

Table 15 Example of the first steps in the evolution of the algorithm