

# Using a discrete event simulator as real time graphic applications kernel

Inmaculada García \*, Ramón Mollá

*Computer Graphics Section, Department of Computation and Computer Systems, Polytechnic University of Valencia,  
Camino de Vera, s/n E-46022 Valencia, Spain*

Received 28 July 2005; received in revised form 9 June 2006; accepted 19 July 2006  
Available online 22 September 2006

---

## Abstract

Simulation is often used to solve problems in many areas in the form of problem analysis. Real time graphic applications such as videogames typically use a continuous simulation scheme. This operating scheme has disadvantages that can be avoided by using a discrete event simulator as the application kernel. This paper proposes the integration of a discrete event simulator into a real time graphic application to control the kernel simulation. Using a discrete methodology avoids disorderly event execution or the execution of cancelled events. The use of this methodology involves using events as the method of modeling the system dynamics and the interaction and behavior of the objects.

© 2006 Elsevier B.V. All rights reserved.

*Keywords:* Discrete events; Simulation; Computer graphics; Videogames

---

## 1. Introduction

Since the beginning of simulation [13], discrete event simulators [1] have been used to solve problems involving systems analysis or system design. Simulators are in a state of continual evolution with new characteristics appearing from time to time. Improvements have been made to simulation speed, result analysis methods, graphic interfaces and graphic model description. Simulation is an appropriate tool for solving problems in areas such as [10] military applications, science and engineering, training programs and management. Simulation is frequently used in the field of computer graphics [11,24,17,21].

Both computer simulation and computer graphics employ numerical modeling methods to simulate the behavior of world entities and mechanisms. The integration of computer simulation and computer graphics presents many benefits [11]:

- Real time graphic applications (RTGA) use simulation techniques for animation (high level modeling methods). The use of simulation techniques in computer graphics is restricted to modeling methods, such as Petri nets or queues.

---

\* Corresponding author. Tel.: +34 963877007; fax: +34 963877359.

*E-mail addresses:* [ingarcia@dsic.upv.es](mailto:ingarcia@dsic.upv.es) (I. García), [rmolla@dsic.upv.es](mailto:rmolla@dsic.upv.es) (R. Mollá).

- Simulators use graphic techniques for the visualization of the simulation process output. Some simulators allow an animated graphic output (WITNESS [12], AutoMod [20], CINEMA [16], Simgraphics [22] or PROOF ANIMATION [4]).

Although the dispute between discrete event simulators and continuous simulators has been going on for some time, discrete RTGA techniques are not very widely used. RTGA use discrete simulation techniques to manage the events belonging to a specific sequence. For instance, the use of discrete events to establish the system sampling period in a character's movement [15], or to define the character's behavior as a state machine using events [19] or to define a scheduler to manage game events belonging to different game technologies, such as simulation of physical phenomena, character animation, collision detection, game AI or rendering [8]. However, the whole system still uses continuous simulation because it defines a sampling period and all the RTGA active objects are sampled at each main loop step.

### 1.1. Real time graphic application simulation paradigm

RTGA has a wide range of applications, including virtual reality or videogames. We have selected videogames as the study objective because:

- There are videogames with free source code available.
- The code complexity of these videogames is diverse.
- They usually run on a single PC.
- They are representative of the RTGA area.

Although the study was carried out on videogames, the conclusions can be generalized to RTGA. In order to verify the use of simulation techniques, an events management study was performed on a variety of videogames with free source codes. Some of these videogames were non-commercial games made by enthusiasts [28]. These videogames lack internal organization and employ quite rudimentary simulation techniques and so were not included explicitly in the present study. Only a few commercial videogames have published their source code. Among these we considered DOOM v1.1 [9,3] or QUAKE v2.3 [18] and the Fly3D kernel [26,27] because of their importance in the videogames area. Their working model may be seen in Fig. 1.

**Algorithm 1.** Real time graphic applications main loop [14].

```

WHILE (true)
  Get information from input devices
  Compute a tick of the simulation
  Update graphics to the user
ENDWHILE

```

The main loop of the preceding videogames has three main phases (Algorithm 1, Fig. 1):

1. The videogame gets the user events and process them.
2. Simulate (evolve the world). During the execution of a step of simulation (tick) all the videogame objects (such as avatar, characters, stones or cars) are asked for pending simulation events (such as movement or shooting) taking charge of the time elapsed since the last simulation step. Videogame objects are usually included in the scene graph. The simulation phase travels through the scene graph and asks each object if it has something to do. All the active objects are sampled in a world evolution. This videogame object-sampling produces a continuous simulation scheme in the system. The simulation cycle (time slot) is defined as the time elapsed in a run of the program main loop (continuous simulator sampling period). All events are obliged to evolve at the highest speed the computer system can supply.

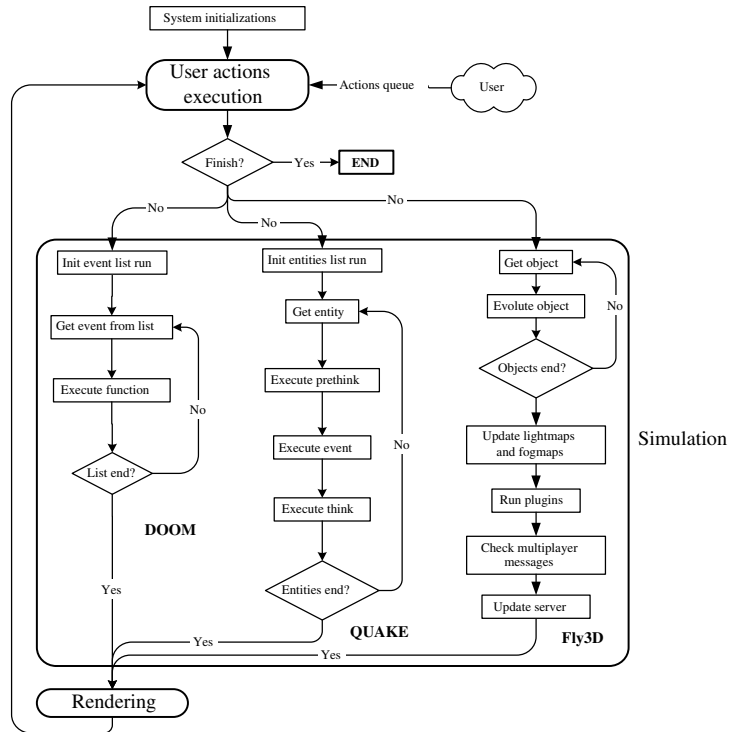


Fig. 1. DOOM, QUAKE and Fly3D working model.

3. Render the current scene. The scene graph is used both to simulate and to render the scene. Some videogames use the scene graph pass to select the objects that must be rendered, including them in some data structure. In this way, the rendering only requires to travel the secondary data structure, diminishing the cost of the rendering process.

The conclusions obtained from the videogames main loop are:

- The simulation phase makes a continuous sampling of the videogame objects because all objects are sampled at each main loop step. Therefore, the simulation techniques used in videogames in many cases involve considering the videogames as continuous systems (in accordance with the systems classification based on the way the system evolves in time [25]).
- The working model of videogames follows the typical scheme of RTGA (Algorithm 1) coupling the rendering phase and simulation phase (each world evolution always requires a full world rendering). The scene could be shown on the screen or not depending on the next screen refresh. If the scene is not shown on the screen, the time spent rendering the scene is wasted.

1.2. Criticism

A continuous simulation model has several disadvantages for RTGA:

- All the objects in the scene graph are checked, although many objects will never generate events. Some RTGA only sample the active objects during the simulation phase. Objects that change from activity to inactivity must always be checked in order to verify if they change their state. Access through the scene graph when many objects will never generate events, is quite inefficient.

- The simulation may be erroneous because the events execution could be disorderly. The objects events are not ordered by time. The events priority for simulation depends on their situation in the scene graph. Furthermore, some cancelled events could be executed. In a particular simulation cycle, an event could decide if other events must be executed or not. If the object that has the associated event is situated further down the scene graph the cancelled events are executed. For example, a missile must destroy a wall in the current simulation cycle. In this cycle, the player pushes a button and deactivates the missile. When the missile is checked its state is “destroy the wall”. The wall is destroyed because the new missile state is not checked until the next simulation cycle.
- The sampling frequency is common to all the RTGA objects, independently of their requirements (the system sampling frequency must be high enough to simulate properly every object to match the Nyquist-Shannon theorem). An undersampled object could not be simulated properly (losing events or not detecting collisions). Objects with very slow behavior may be oversampled, uselessly wasting CPU power.
- RTGA objects could have both continuous and discrete behaviors (the continuous system evolution in time may be altered by events not associated to the sampling period). RTGA are therefore hybrid systems. Since they are usually implemented as continuous coupled systems, they lose performance and may produce erroneous behaviors.

The system coupling also has disadvantages. The scene has to be rendered for every simulation cycle, regardless of whether it is going to be shown on the screen or not. So, the system sampling frequency is highly dependent on computer power and the simulation and rendering workload. Events are synchronized with the sampling period, so the system is not sensitive to time intervals lower than the system sampling period.

## 2. Objectives

Videogames follow a scheme of continuous simulation although they could be considered as hybrid systems. Discrete event simulation allows the modeling of both continuous, discrete or hybrid systems [1]. The aim of this paper is to show how the use of discrete event simulation techniques can improve RTGA simulation. The proposal is to change the continuous simulation RTGA paradigm for the discrete paradigm to allow a more accurate simulation than using the continuous paradigm. The objective is to integrate a discrete event simulator into an RTGA in order to change the simulation paradigm.

The advantages of using a discrete event simulator as RTGA kernel are:

- Game quality is improved because the events are executed ordered in time and there is no disorderly events execution. Each object defines its own sampling frequency. It is defined according to the behavior of each object. The system is sensitive to times lower than the sampling period, because every object has its own sampling period. The level of detail of simulation increases: unnoticed events are simulated. The sampling period does not change depending on topics such as the system load or the scene complexity.

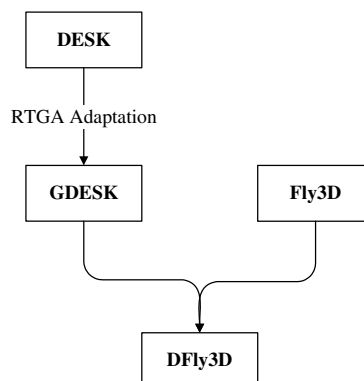


Fig. 2. Steps in the DFly3D creation.

- Game efficiency increases. Discrete systems make better use of computer power. The power thus released can be used to better process other videogame parts (AI or kinematics). Only the objects that generate events will be checked, avoiding access to the remaining objects. Videogames can thus be executed in machines with lower power. Distributed RTGA can be run on machines with different computing power showing the same behavior.

The discrete event simulator DESK is presented in the following sections. Section 3.2, contains a description of a modification for integration into a videogames kernel (GDESK). Section 3.3 describes how GDESK was integrated into the Fly3D videogames kernel, while the resulting discrete videogames kernel was DFLy3D, which is analyzed in Section 3.4. Fly3D uses a scheme of continuous simulation and DFLy3D a discrete. Both kernels are compared. The discrete event simulator GDESK manages the DFLy3D events. A scheme of the relationships between all these kernels may be seen in Fig. 2.

### 3. Discrete application development

#### 3.1. The discrete event simulator DESK

DESK [6,2] is a universal object oriented library developed using ANSI C++. It may be used both as a fast prototyper and as a final model descriptor simultaneously. DESK can simulate any model without restrictions. Although it uses dynamic memory to avoid computer or compiler lags, no penalty in performance is noticed due to a client pool that avoids unnecessary calls to the dynamic memory manager (DMM). It is powerful enough to manage any system and flexible enough to allow fanciful behaviors. It allows easy model definition and implementation. The system model definition must be carried out defining the simulation components (service stations and clients) and their characteristics. DESK is based on two discrete event simulators: SMPL [7] and QNAP [23]. Both philosophies were combined: the speed and freedom of old C based packages like SMPL and the organization and simplicity of an object-oriented model description like QNAP (fast and easy prototyping).

The DESK key structures are:

- Dispatcher: contains the events that are going to happen in the system ordered by time.
- Events pool: each time a new event is necessary in the system, the DMM must be called. When the event finishes its function and leaves the system, the DMM must be called again to destroy it. This is inefficient, because the DMM is constantly being called. In order to minimize DMM calls, an events pool is used. The pool is a way to maintain the events that are not currently in the system. Events are being created during simulation. In a given moment, they leave the system and they are inserted in the pool instead of being

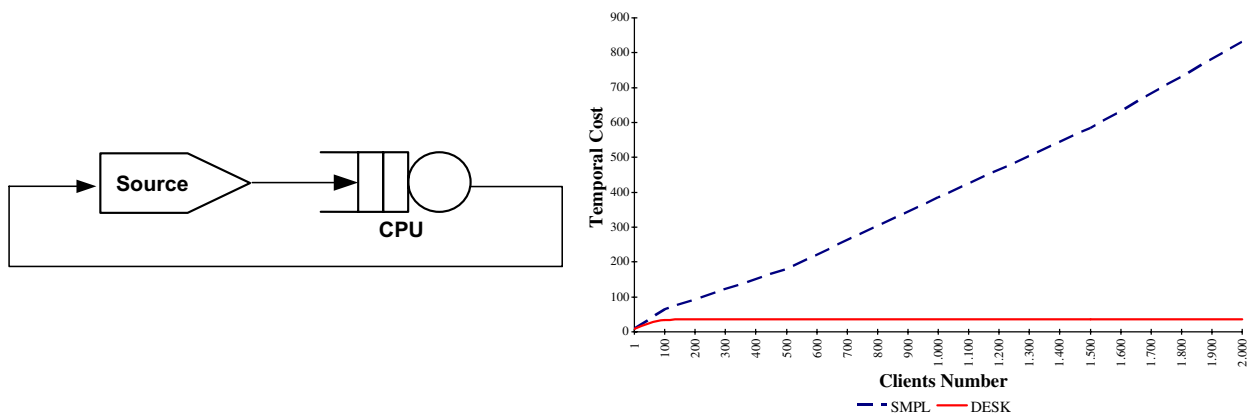


Fig. 3. SMPL and DESK simulation cost (closed system) increasing the number of clients.

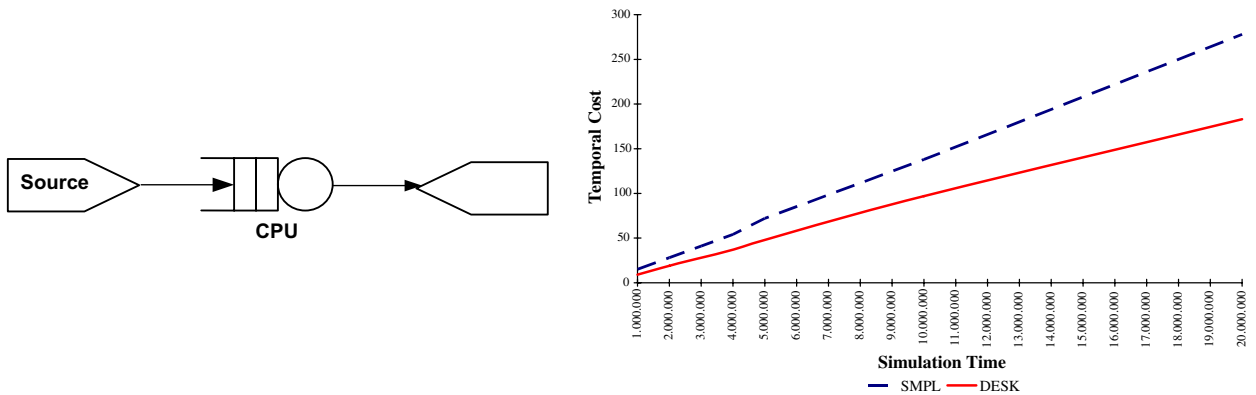


Fig. 4. SMPL and DESK simulation cost (open system) increasing the simulation time.

destroyed. When a new event is necessary in the system, the simulator verifies if there are events in the pool and an event will be extracted from the pool if the pool is not empty. The DMM will be called if and only if there are no available events in the pool.

In general, DESK is faster than SMPL for almost all simulated models in the laboratory (Figs. 3 and 4). DESK allows simulation models that are impossible to simulate using SMPL due to the amount of clients or service stations in the system or the model complexity. Figs. 3 and 4 show the simulation temporal cost comparison between SMPL and DESK. Fig. 3 is obtained simulating a closed system while changing the number of clients. Fig. 4 shows an open system that is simulated increasing the simulation total time.

DESK is a discrete event simulator and it is not suitable for integration into an RTGA kernel in its current form. Certain changes are therefore necessary for the integration. These changes are:

- Time management. The time evolution does not only depend on the events occurrence but must also follow real time. An event happens when its time stamp is reached by the real time clock. The time management in a discrete event simulator is not usually in agreement with the real time. The simulator has an internal clock that evolves each time an event occurs. Event times in RTGA must match the real time. An event only happens if the event time stamp is reached by the real system time. At this moment, the simulator clock is modified with the event time.
- Events are simulation passive entities that travel from one service station to another. In a discrete RTGA, the event must communicate objects (or model the object behavior). The event is produced by a videogame object and it always has an object as destination. So, an event travels through the system objects. The event must contain the information to change the destination object (event parameters). The destination object changes its behavior or state depending on the event parameters. The events parameters must be defined by the RTGA programmer, so the events management structure must be accessible by the programmer.
- Discrete events simulators usually finish the simulation defining the maximum simulation time or if the events queue is empty. In an RTGA the user controls the end of the simulation. If the events queue is empty, the RTGA execution is paused.
- DESK must be integrated into an RTGA and so must share the system dynamics with the RTGA.

### 3.2. The discrete event simulator DESK as RTGA kernel: GDESK

GDESK (Game Discrete Event Simulation Kernel) [5] is the adaptation of DESK to an RTGA kernel. It is a simulation kernel that copes with the RTGA events handling. GDESK uses two basic entities to model the events passing mechanism: objects and events. Events are messages sent from one object to another, GDESK therefore controls the RTGA objects communication by message passing. GDESK maintains the events

ordered by time until their time stamp is exceeded. The dispatcher is the GDESK basic structure. It supports the messages passing process, it maintains the messages ordered by time until the message time stamp is reached by the real time clock and it sends the messages to the target objects at the time specified by the time stamp.

Objects are the dynamic system entities. Any videogame element must be a GDESK object. Events are the passive elements that allow the objects communication. The system dynamic is modeled by message passing. All RTGA objects must inherit from the GDESK basic object. The functions for receiving and sending messages are defined in this basic object. The message data structure contains two kinds of information:

- GDESK internal information: This contains the necessary parameters to control the messages sending and receiving process: source object, destination object and message reception time.
- RTGA information: This part of the message structure is accessible to the RTGA programmer in order to define the application parameters. So the programmer has freedom to define the messages structure avoiding unnecessary data structures overloads. These parameters are managed directly by the programmer. GDESK does not take these parameter structures into account.

### 3.3. The RTGA kernel FLY3D

Fly3D SDK is a software development kit for videogames and 3D RTGA. The Fly3D main loop follows a typical scheme of continuous simulation that couples the simulation phase and rendering phase. Each RTGA component is defined in Fly3D as an object. The object defines a rendering function and a simulation function. The global simulation process calls each simulation object function following their order in the scene graph. The simulation takes care of the time elapsed from the last simulation until now. The rendering process is similar. For each main loop step a complete simulation and rendering is performed. Fly3D is plugins-oriented. An RTGA created using Fly3D is a set of plugins. The plugins are completely separated from the Fly3D kernel. All the applications have a common front-end that supports the user interface and the application main loop.

GDESK integration into Fly3D involves two levels of changes in Fly3D:

- Changes in the Fly3D kernel in order to isolate its functionality as objects. The new objects behavior is defined by message passing. These objects are the DFly3D system objects. One of these objects is the render object. That means the rendering process starts and continues by events.
- Changes in the games created using Fly3D to integrate the GDESK functionality into the videogame objects. These objects are the DFly3D world objects.

Using a discrete event simulator as a simulation engine for RTGA does not involve changing application topics like the structure of the scenes description files or characters. It does not modify the file parser, the scene graph, the rendering techniques applied or the RTGA style. It only modifies the system events management and introduces a discrete event simulation scheme and only focuses on the events management.

The RTGA main loop shown in the [Algorithm 1](#) must be changed. The new main loop is controlled by the simulator.

### 3.4. The discrete RTGA kernel DFLY3D

DFly3D (Discrete Fly3D) is an RTGA kernel resulting from the integration of GDESK into Fly3D. DFly3D is a toolkit to create discrete RTGA (discrete simulation paradigm). An RTGA created using DFly3D is a set of objects generating events. Any DFly3D element must be a GDESK object. GDESK treats all objects in the same way. There are two kinds of objects in DFly3D ([Fig. 5](#)):

- System objects: all application functional components must be modeled as objects generating events. They are the objects that control the application. Examples of system objects are console, render unit, multi-user control or server control.



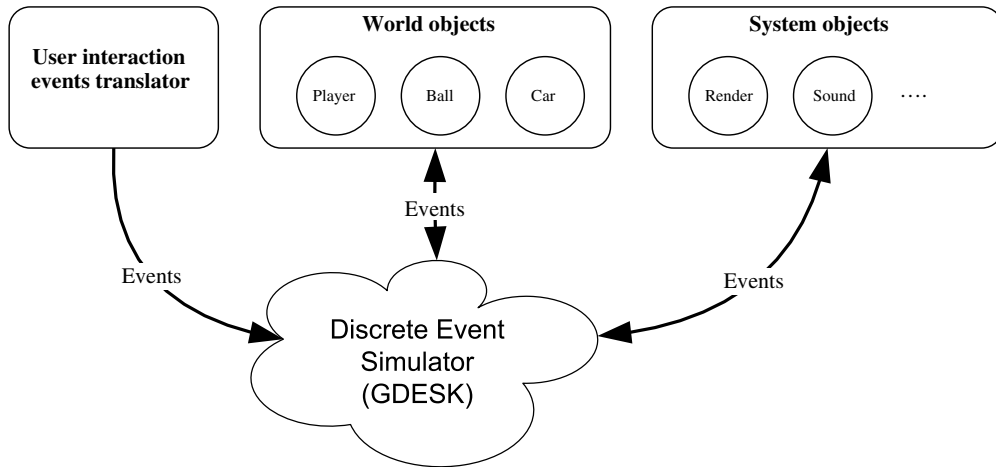


Fig. 5. Discrete event simulator as RTGA kernel.

- World objects: they are the objects simulated and/or rendered, such as avatars, cars or walls. The objects generate events that model their behavior and their interaction with other objects.

The DFly3D kernel has the following components (Fig. 5):

- User interaction events translator: the user events are converted to GDESK events. GDESK deals with all events in the same way.
- Objects: system objects and world objects.
- GDESK: collects the events generated by the objects and maintains the events ordered by time.

Objects in an RTGA interact and evolve using the events generation mechanism. The system dynamics are modeled by events. An object only acts when an event is produced and sent to it. That object may change its behavior and it may generate other events. The change in the object behavior depends on topics such as the event source object, the event type or the event parameters. Both object types use this mechanism. Events are controlled by GDESK regardless of the kind of object that generates the event. The GDESK dispatcher maintains the events ordered by time. It executes events at the moment marked by the event time stamp. Events ordered by time are executed regardless of the source event object, so the discrete event simulation avoids an erroneous objects priority. Objects priority is defined by the event time stamp.

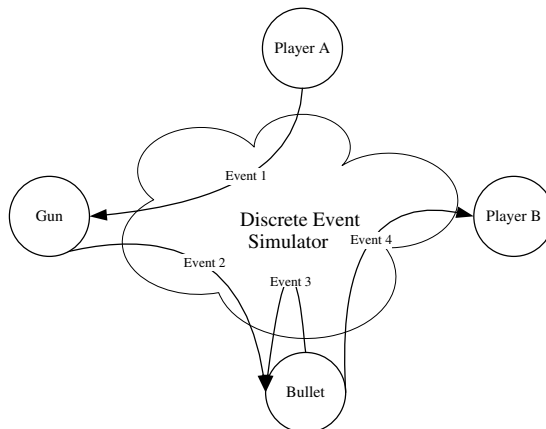


Fig. 6. Events generation example.



Events uses are:

- Objects communication: When object A needs to interact with object B, A generates an event (message  $E_{AB}$ ) addressed to B. The object B acts or changes its behavior as a consequence of the event  $E_{AB}$  arrival. Only the object that receives an event may generate more events.
- Modeling the object behavior: An object only acts as an event arrival consequence. When an object A must change its own behavior at a given moment, A generates an event addressed to itself ( $E_{AA}$ ). As a consequence of the event from itself, object A modifies its state as required.

An event needs to include the following information:

- Source object: The object that generates the event.
- Destination object: The object receiving the event.
- Event time stamp: This is the given real time when the event must happen.

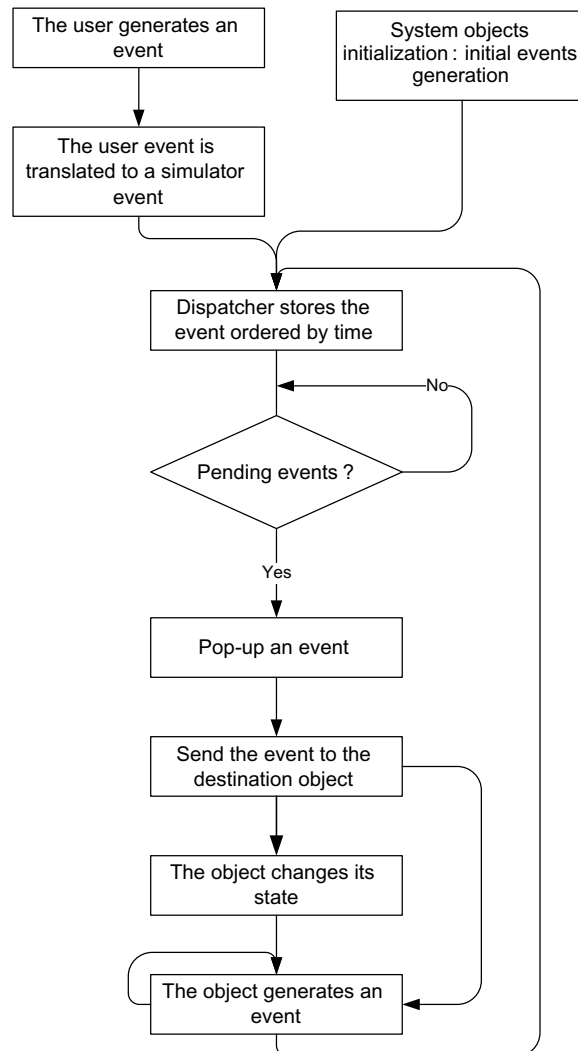


Fig. 7. Discrete system dynamics.

- Videogame information: the interaction between objects or the object behavior modeling needs certain parameters to model this interaction. A destination object A may behave in different ways depending on the information associated to the event sent by the source object B and the object B itself.

Fig. 6 shows an example of events generation. Player A decides to shoot at player B. Player A generates an event (event 1) that has the object gun as destination, to indicate the gun that shoots a bullet at player B. When the gun receives the event, it generates another event telling the bullet to go to player B (event 2). The bullet starts on its trajectory according to its characteristics (mass, weight or gravity). To model the trajectory situation during its journey, the bullet generates several events (event 3) directed to itself. When the bullet hits player B, the bullet generates an event (event 4) telling player B “I have hit you”. Depending on the state of player B, the bullet impact may have different consequences (player B dies or is only hurt).

Fig. 7 shows the DFLy3D system dynamics. The whole system is controlled by GDESK. Initially, only the user generates events (the exception is the system objects that need initial events to arrange their behavior, such as the render object).

The simulator dispatcher stores events ordered by time. The dispatcher checks if there is a pending event (whose time has been accomplished). The pending event execution involves sending the event to the destination object. The destination object takes control of the simulation and it may change its state and/or generate events as a response.

#### 4. Results: Fly3d and DFLy3d comparison

##### 4.1. Simulation quality

Fig. 8 shows the trajectory of a ball. There are three trajectory intervals with different behaviors. The ball changes its position more quickly during interval A than in the other intervals. Fig. 8 shows the ball samples for both the continuous system and the discrete system. Traditional RTGA samples the ball with a period  $T$  that could be constant or variable depending on topics such as the current system load, the current scene complexity or the network overload. The ball simulation in a continuous system takes charge of elapsed time since last simulation step and evolves the object. Every object is sampled for each main loop step (continuous simulation), so the sampling period is not adaptable to the ball behavior. The discrete system (DFLy3D) changes the ball sampling period according to the object trajectory requirements, using the events generation mechanism (discrete simulation). Therefore, the sampling period in the first trajectory interval A must be lower than the sampling period in the other trajectory intervals B and C. The ball remains in the same position during interval B, so it does not need to be sampled. The ball knows its trajectory and adapts the next event generation to it. The ball generates an event to be executed the next time its state must change. If the ball behavior is discrete, the event must be in accordance with the next ball evolution.

If the behavior of an object in a discrete system is continuous, the events must be generated and executed at a previously defined fixed rate. The events rate depends only on the object behavior and state. Objects with a

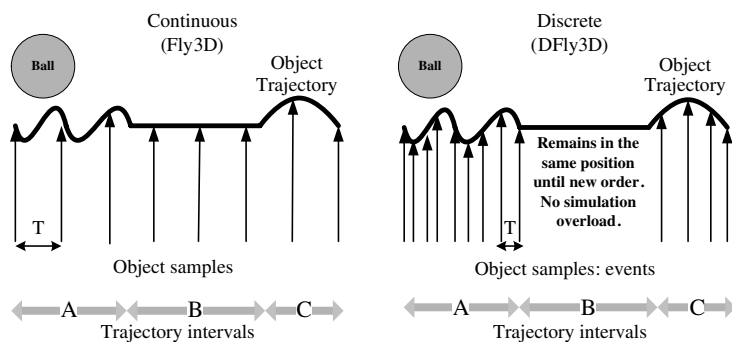


Fig. 8. Events sampling with a discrete and continuous simulator.

slow behavior generate fewer events than objects with a fast behavior. Events generation is adapted to the object behavior because the simulation is discrete. Using a discrete simulator such as the RTGA kernel adapts the objects sampling to the objects behavior, so the computer power used by the RTGA is only that necessary to simulate the objects properly. The computer power is distributed between the objects according to the object simulation requirements.

#### 4.2. Computer power

The discrete simulation paradigm not only improves the simulation because the object sampling is adapted to the object instantaneous behavior, but also saves computer power due to this adaptation.

Let us suppose a tank that is filling with water. The water tank faucet must be turned off in ten seconds. Fig. 9 shows the samples of the object faucet in a continuous and in a discrete system. The continuous system samples the faucet at each main loop step. The continuous system asks the faucet at each sample if it must be turned off now. The discrete system plans a discrete event for the given moment the faucet must be turned off. The continuous system thus wastes computer power by sampling the object unnecessarily. The discrete system uses only the minimum computer power to simulate the object properly. The saving in computer power can be used to improve other RTGA processes such as rendering. The time spent in the continuous system sampling the faucet during this interval is saved in the discrete system.

The object implementation in a discrete videogame generates as many events as necessary to model its behavior and only the number of events strictly necessary, so that computer power is adjusted to the current object behavior and waste is thus avoided.

The continuous system couples the rendering process and the simulation process, so that each time the world objects are sampled the current scene is rendered. The frame frequency is usually too high if the system load is not high. Computer power is wasted rendering scenes that will never be shown on the screen. In the discrete system the render process is controlled by the render object. Each time an event is generated by the render object, the current scene is rendered. The programmer defines the frame frequency according to the number of messages generated by the render object. This rendering rate can be set according to the RTGA needs and the rendering device performance. It may be constant during the execution or it may be dynamically adapted to the system needs. The rendering process is defined and controlled by the programmer. The programmer may define the frame frequency to render the scene only once for each screen refresh. In this way computer power is saved.

#### 4.3. Simulation correctness

Fly3D samples the objects traveling the scene graph in the order they are situated in it. This may imply an erroneous objects priority.

Let us consider the situation shown in Fig. 10. A ball and a missile are going to impact with a wall. They travel in opposite directions. At time  $t_0$ , the ball B is in position  $B_{P0}$  and the missile M in position  $M_{P0}$ . According to the velocities of both projectiles, the missile will hit the wall at time  $t_M$  and the ball at  $t_B$ . Let us assume the difference between the times  $t_B$  and  $t_M$  is small enough to simulate both impact processes in

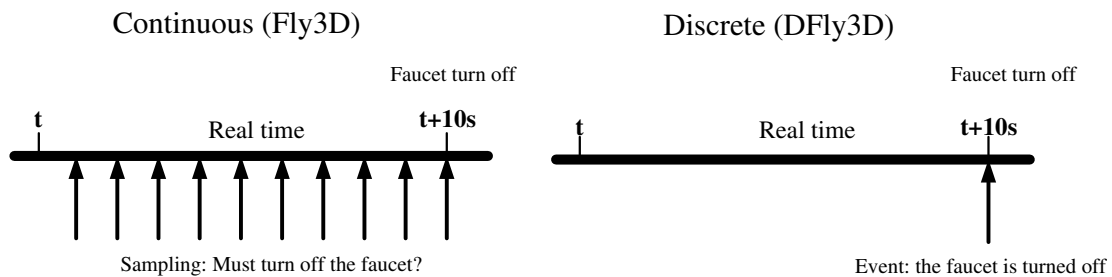


Fig. 9. Example of scheduled event.

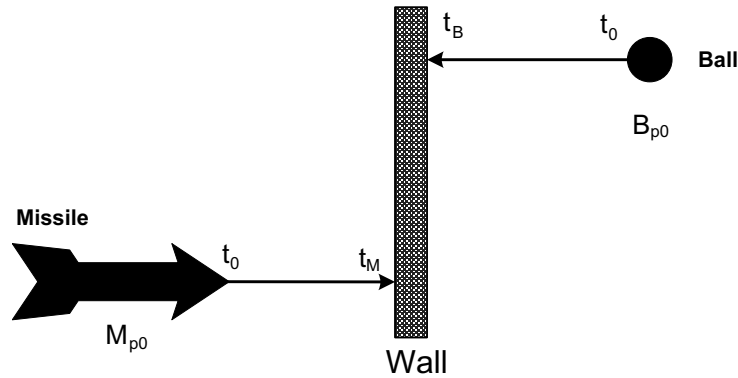


Fig. 10. Example of execution of disorderly events.

the same sample interval in Fly3D (simulation cycle). According to the values of  $t_B$  and  $t_M$  the simulation result must be different. If the ball impacts with the wall first ( $t_B < t_M$ ), the ball must rebound and the missile must destroy the wall later. If the missile impacts with the wall first ( $t_B > t_M$ ), the missile destroys the wall and the ball continues on its trajectory through the hole in the wall. In Fly3D the simulation will be right or wrong depending on the situation of the objects in the scene graph. Let  $O_B$  be the position of the object ball in the scene graph and  $O_M$  the position of the object missile. Table 1 shows the different situations of wrong or right simulations produced in Fly3D dependent on the values of  $t_B$  and  $t_M$  and the objects' situation in the scene graph. The Fly3D simulation may therefore be correct or incorrect, depending on the situation of the objects in the scene graph.

DFly3D executes the events ordered by time, so the possible situations are shown in Table 2. Both situations produce a correct simulation.

#### 4.4. Temporal cost

Fig. 11 show the percentage of simulation time used by Fly3D and DFLy3D to simulate, render and remain idle while increasing the simulation and rendering load (increasing the number of objects).

The application time in the continuous system is shared by the simulation and the rendering processes. The system uses nearly 100% of the time rendering and simulating. For each main loop step each object is simulated once and the scene is rendered. There is no free time. An increase in the simulation load means a decrease in the rendering process and an increase in the rendering load means a decrease in simulation time. In a continuous system there is no released computer power. The system is continuously simulating and rendering at the maximum speed.

Table 1  
Order of events execution in Fly3D

Arrival time	Scene graph position	Order of actions	Correct simulation
$t_B < t_M$	$O_B < O_M$	Ball rebounds missile destroys the wall	Yes
$t_B < t_M$	$O_B > O_M$	Missile destroys the wall ball rebounds	No
$t_B > t_M$	$O_B < O_M$	Ball goes through missile destroys the wall	No
$t_B > t_M$	$O_B > O_M$	Missile destroys the wall ball goes through	Yes

Table 2  
Order of events execution in Fly3D

Arrival time	Order of actions	Correct simulation
$t_B < t_M$	Stone rebounds missile destroys the wall	Yes
$t_B > t_M$	Missile destroy the wall stone goes through	Yes

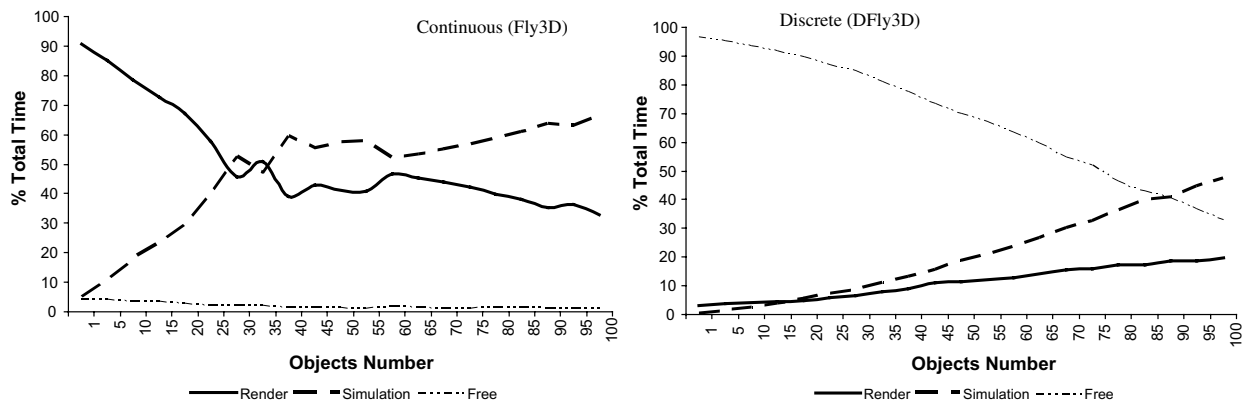


Fig. 11. Fly3D and Dfly3D simulation times.

In the discrete system the rendering process and the simulation process are not dependent (decoupled). So the system time is not shared by the rendering process and simulation process. The system uses only the minimum computer power necessary to simulate each object behavior. The system uses nearly 100% of CPU time if the system is collapsed (the system load is greater than computer power). The discrete system always freed more time than the continuous system. The resources consumed by a discrete videogame are always lower than the resources consumed by a continuous videogame. If the computer power is enough to simulate the system objects properly, the released computer power may be used by other system applications. The discrete system uses only the necessary computer power to correctly simulate the system objects. Dfly3D allows an RTGA to be executed using less computer power or in less powerful computers.

## 5. Conclusions

Discrete events simulation has been used in RTGA to deal with specific parts of the application, but the RTGA general simulation model remains continuous. This mode of operation has disadvantages that can be avoided using a discrete event simulator as the RTGA kernel. If the RTGA simulation paradigm is changed from continuous to discrete, the system simulation improves. The simulation is more accurate because it avoids disordered execution of events. Each object is simulated using a specific sampling period to adapt its sampling to the object behavior. The object sampling may change dynamically to adapt the sampling to its current behavior. Discrete simulation avoids sampling the objects unnecessarily when a discrete event is scheduled (the continuous system samples the scheduled event until the condition is reached). Discrete RTGA uses only the computer power strictly necessary to simulate the objects' behavior, so the released computer power can be used to improve the simulation of other application elements or to run the simulation in machines with less computer power. The use of the discrete methodology in RTGA involves using events as the method of modeling the system dynamics, interaction of objects and the modeling of objects' behavior.

## References

- [1] J. Banks, J.S. Carson II, B.L. Nelson, D.M. Nicol, *Discrete-Event System Simulation*, Prentice hall, 2001.
- [2] DESK Main Page. Available from: <<http://www.sig.upv.es/proyectos/simulacion/DESK.htm>>.
- [3] Doom World. Available from: <<http://doomworld.com/>>.
- [4] N.J. Earle, J.O. Henriksen, Proof animation: reaching new heights in animation, in: *The 1994 Winter Simulation Conference*, Institute of Electrical and Electronic Engineers, 1994. pp. 509–516.
- [5] I. García, R. Mollá, A. Barella, GDESK: game discrete events simulation Kernel, *Journal of WSCG* (2004).
- [6] I. García, R. Mollá, E. Ramos, M. Fernández. D.E.S.K. Discrete Events Simulation Kernel, in: *ECCOMAS*, 2000.
- [7] Gent University. Available from: <<http://www.autoctrl.rug.ac.be/ftp/smpl/>>.
- [8] M. Harvey, C.S. Marshall, *Scheduling Game EventsGame Programming Gems 3*, Charles River Media Inc., 2002.
- [9] Idsoftware Page. Available from: <[www.idsoftware.com/archives/doomarc.html](http://www.idsoftware.com/archives/doomarc.html)>.

- [10] J. Kuljis, R.J. Paul, Web-based discrete event simulation models: current status and possible futures, *Simulation and Gaming* 34 (1) (2003).
- [11] G.S. Lee. Towards an integration of computer simulation with computer graphics, in: *Proceedings of the Western Computer Graphics Symposium*, 1999.
- [12] A. Mehta, I. Rawles. Business solutions using WITNESS, in: *Proceedings of the 1999 Winter Simulation Conference*.
- [13] R.E. Nance, Simulation programming languages: an abridged history, in: *Proceedings of Winter Simulation Conference*, 1995.
- [14] R. Pausch, T. Burnette, A.C. Capehart, M. Conway, D. Cosgrove, R. DeLine, J. Durbin, R. Gossweiler, S. Koga, J. White, A brief architectural overview of alice, a rapid prototyping system for virtual environments, *IEEE Computer Graphics and Applications* (1995).
- [15] D.C. Pottinger. Coordinated Unit. Movement. *Game Developer Magazine*, January, 1999 Available from: <[http://www.gamasutra.com/features/game\\_design/19990122/movement\\_01.htm](http://www.gamasutra.com/features/game_design/19990122/movement_01.htm)>.
- [16] D.M. Profozich, D.T. Sturrock. Introduction to SIMAN/CINEMA, in: *The 1994 Winter Simulation Conference*, Institute of Electrical and Electronic Engineers, 1994, pp. 427–430.
- [17] Przemyslaw Prusinkiewicz, Aristid. Lindenmayer, *The Algorithmic Beauty of Plants*, Springer-Verlag, 1991.
- [18] Quake Developers Page. Available from: <[www.gamers.org/dEngine/quake/](http://www.gamers.org/dEngine/quake/)>.
- [19] S. Rabin, *Designing a General Robust AI Engine* Game Programming Gems, Charles River Media, 2000.
- [20] M. Rohrer, AutoMod product suite tutorial (AutoMod, Simulator, AutoStat) by Auto Simulations, in: *Proceedings of the 1999 Winter Simulation Conference*, 1999.
- [21] W.T. Reeves, Particle systems: A technique for modeling a class of fuzzy objects *Computer Graphics*, ACM Siggraph, 1983, pp. 59–376.
- [22] E.C. Russell, Simsript II.5 and Simgraphics tutorial, in: *The 1993 Winter Simulation Conference*, Institute of Electrical and Electronic Engineers, 1993, pp. 223–227.
- [23] Simulog. Available from: <<http://www.simulog.fr/formation/qnam.htm>>.
- [24] D. Terzopoulos, A. Witkin, Physically based models with rigid and deformable components, *IEEE Computer Graphics and Applications*. (1988) 41–51.
- [25] G.A. Wainer, *Introducción a la Simulación de Sistemas de Eventos Discretos*. Technical Report: 96-005, Buenos Aires University, 1996.
- [26] A. Watt, F. Policarpo, *3D Computer Games Technology: Real-Time Rendering and Software*, Addison-Wesley, 2001.
- [27] A. Watt, F. Policarpo, *3D Computer Games*, Addison-Wesley, 2003.
- [28] Ziron Page. Available from: <<http://www.ziron.com/links/>>.