GDESK: Game Discrete Event Simulation Kernel

Article · January 2004

Source: CiteSeer

CITATIONS
8

READS
228

4 authors, including:

Inmaculada García
Universitat Politècnica de València
33 PUBLICATIONS **840** CITATIONS
SEE PROFILE

Ramón Mollá Vayá
Universitat Politècnica de València
95 PUBLICATIONS **237** CITATIONS
SEE PROFILE

GDESK: Game Discrete Event Simulation Kernel

Inmaculada García
Computer Graphics Section
Technical University of Valencia
Camino de Vera S/N
Spain (46022), Valencia, Valencia
ingarcia@dsic.upv.es

Ramón Mollá
Computer Graphics Section
Technical University of Valencia
Camino de Vera S/N
Spain (46022), Valencia, Valencia
rmolla@dsic.upv.es

Toni Barella
Computer Graphics Section
Technical University of Valencia
Camino de Vera S/N
Spain (46022), Valencia, Valencia
tbarella@dsic.upv.esl

ABSTRACT

Simulation has been used traditionally to solve other areas problems. Real time applications like videogames use typically a continuous simulation scheme. That way of operation has disadvantages that can be avoided using a discrete event simulator as a game kernel. This paper proposes the integration of a discrete event simulator into real time applications to control the applications simulation. The use of a discrete methodology avoids disorderly events execution or the execution of cancelled events. This implies to use events in order to model the system dynamics, the objects interaction and the objects behavior. GDESK is a discrete event simulator prepared to be used as a videogame kernel.

Keywords

Videogames, simulation, discrete events, kernel

1. INTRODUCTION

Videogames Simulation Model

Videogames follows a scheme of continuous simulation that couples rendering phase and simulation phase [Pau95]. A study of different videogames has been made, from simple videogames [Ziron] to complex ones. Among the videogames studied are Doom v1.1 [Ids][Doom], Quake v2.3 [Quake] and Fly3D [Fly3D][Wat01][Wat03]. They have been selected because of their importance in the videogames history or the availability of their source code.

The videogames main loop has typically three phases [Pau95]:

1. Test the user interaction.
2. Simulate. Videogame objects use to be included in a scene graph. The scene graph is used both to simulate and to render the scene. Simulation is done traveling the scene graph and asking to each object if it has something to do. Compute a time step (tick) of simulation supposes to ask the scene graph objects for pending simulation events: movement, shooting,...
3. Render the current scene.

This main loop supposes:

- Videogames use a continuous simulation scheme. Because the entire scene graph objects are sampled in a world evolution. Simulation cycle (time slot) is defined as the time elapsed in a run of the program main loop (continuous simulator sampling period).
- Simulation and rendering are highly coupled (each world evolution always requires a full world rendering).

That mechanism has disadvantages. The rendering and simulation coupling disadvantages are:

- The system is not sensitive to times lower than the sampling period.
- The simulation events are artificially synchronized to match the sampling period. They are not executed in the very moment when they happen.
- The sampling frequency depends on topics that can change during the game, such as available computer power, world complexity, other active tasks in system, network overload or current simulation and rendering load. So, the sampling frequency is variable and not predefined.
- A new simulation cycle requires always an entire world rendering, although the frame can be shown on the screen or not.

Journal of WSCG, Vol.12, No.1-3, ISSN 1213-6972
WSCG'2004, February 2-6, 2004, Plzen, Czech Republic.

- Time generating renderings is wasted since many frames will never be appreciated on screen.

The continuous simulation model disadvantages are:

- All objects in the scene graph are accessed, although many objects will never generate events. Some videogames allow to access only to the active objects. Access through the scene graph when many objects will never generate events, is quite inefficient.
- Events are not time ordered. Events are executed in the order in which the objects management structure is accessed.
- The objects priority for simulation depends on the objects situation in the scene graph.
- It is just possible that the simulation be erroneous because of: the disorderly events execution and the execution of cancelled events.
- The sampling frequency is the same for all objects, independently of their requirements: If objects behaviors do not match Nyquist-Shannon theorem, they will not be simulated properly, loosing events, not detecting collisions,... That is to say, objects will be undersampled. On the other hand, objects with a very slow behavior may be oversampled.

Discrete Event Simulation

A real-time graphic application, like a computer game may be considered a system [Ban01]. As a system, it can be represented using modeling and simulation techniques [Ban01][Wai96]. Attending to the systems classification, based on the way the system evolves in time [Wai96], a real-time graphic application could be considered as a hybrid system. In that, the continuous system evolution in time may be altered by events not associated to the sampling period.

Discrete event simulation [Fis78][Ban01][Sch99] have been used to solve problems consisting on the system analysis using modeling or to design systems. Simulation is a skill used to solve other areas problems such as [Kul03]: military applications, science and engineering, learning and training and management. Attempts to integrate simulation techniques in computer graphics applications have been made [Lee99][Ter88][Ree83]. But, the employing of simulation techniques in computer graphics is restricted to the use of modeling methods as Petri Nets or queues [Lee99].

A discrete event simulator copes with discrete systems, continuous system and hybrid systems [Ban01].

2. OBJECTIVES

The proposal is to use a discrete events simulator as a computer game kernel in order to achieve the following objectives:

- Increases the videogame quality:
  - Only those objects that generate events will be checked, avoiding to access the remainder objects.
  - Events are executed ordered in time. There is not disorderly events execution.
  - Each object defines its own sampling frequency. It is defined according to the objects behavior.
  - The level of detail of simulation increases: unnoticed events are now simulated.
  - The system is sensible to times lower than the sampling period, since every object has its own sampling period.
  - Sampling period does not change depending on topics as the system load or the model complexity.
- Increase the videogame efficiency:
  - They make a better use of the computer power. The released power can be used to get better other game parts (such as game artificial intelligence or kinematics).
  - Games can be executed in machines with lower power.
  - Real-time distributed applications can be run in machines with different computing power.

3. GDESK ORIGINS: DESK

GDESK is the adaptation of DESK [Gar00] to the videogame kernel requirements.

DESK is a discrete event simulator kernel. It is a universal object oriented package developed using ANSI C++. It may be used both as a fast prototyper and as a final model descriptor simultaneously. DESK can simulate whatever model. Although it uses dynamic memory to avoid computer or compiler lacks, no penalty in performance is noticed due to a client pool that avoids unnecessary calls to the Dynamic Memory Manager (DMM). The main DESK characteristics are:

- General purpose simulator.
- Powerful enough to manage whatever system.
- Flexible enough to allow fanciful behaviors.
- No restrictions to the simulation model.
- Support to real-time simulation.

- Support to external functions.
- Easy model definition and implementation. The system model definition must be done defining the simulation components and their characteristics.
- Easy debugging and changing.
- Implemented as a C++ library to allow using in the simulation other C++ libraries.

DESK Structure

The DESK basic entities are:

- Events: model the change of state of a client in the system.
- Clients: are passive entities that support the events data structure. They can be created, destroyed or modified as convenience during the simulation.
- Service stations: are the elements that give service to clients during simulation.

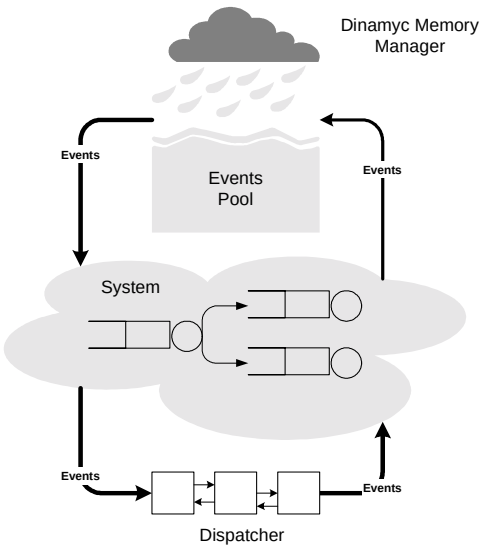

Figure 1. DESK structure

The DESK key structures are (figure 1):

- Dispatcher: contains the events that are going to happen in system, ordered by time.
- Events Pool: each time a new event is necessary in system, the DMM must be called. When the events finish 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 the DMM calls, an events pool is used. The pool is a way to maintain the events that are not actually in the system. Events are being created during simulation. In a given moment, they finish and leave the system, and then they are inserted in the pool instead of being 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 not available events in the pool.

DESK Suitability for Integration

DESK accomplishes the necessary conditions in a discrete event simulator to the integration into a videogame:

- It must have open source code:
  - It is necessary to know how the simulator is implemented in order to know if it is efficient enough.
  - It is necessary to modify the simulator implementation.
- The discrete event simulator must be implemented:
  - As a library.
  - In a general purpose language and commonly used in the videogames implementation and widely used by the scientific community.
  - It must support:
    - Dynamic data structures.
    - OpenGL, DirectX, …
    - Software engineering new technologies.

4. GDESK

GDESK is the adaptation of DESK to be used as videogames kernel.

Using a discrete event simulator as a simulation engine for computer games does not imply to change topics as the structure of the scenes description files or characters. It does not modify the file parser, the scene graph, the rendering techniques applied or videogame style. It only modifies the system events management and introduces a discrete event scheme. It only focuses on the videogame events management. Therefore, it can be applied to any kind of video game and rendering format (2, 2.5 y 3D).

Objects Interaction

GDESK treats any element that appears in a computer game as an object, animated or unanimated. A hierarchy of objects can be established whenever the programmer defines it. The whole system is a set of objects generating events. There are two objects types in a videogame:

- Game components objects: all videogame functional components must be modeled as objects generating events. They are the objects that control the game. Examples of system objects are console, render, multi-user control or server control. That category includes the object that translates the user events into GDESK events.
- Rendered game objects: avatars, missiles, walls,...

All objects in nature interact by means of particles exchanging. So, objects in GDESK, interact by means of events exchange. Objects in videogames interact and evolve using the events generation mechanism. Both objects types use the same mechanism. An event in GDESK is modeled by means of a client that contains the necessary information to develop an adequate interaction. The events mechanism in GDESK is similar to the message passing mechanism.

Events

The system dynamic is controlled by events. Events model the objects behavior. An object only acts when an event is produced and sent to it. When the object receives the event, it can change its behavior and it can generate other events to other objects or to itself. The change in the object behavior depends on topics as the object that generates the event, the event kind or the event content.

Events uses are:

- Objects communication: when the object *A* needs a communication with the object *B*, *A* generates an event addressed to *B*. The object *B* could act or change its behavior as a consequence of the event arrival. Only the object that receives an event may generate more events.
- Model the object behavior: an object only acts as an event arrival consequence. When an object *A* must change its own behavior, *A* generates an event addressed to itself. As a consequence of an event arrival from itself, the object *A* modifies its state as convenience.

When a projectile collides with a wall, the projectile generates an event to the wall indicating the collision in that very moment. This event includes also the necessary parameters (mass, point of impact, velocity, projectile kind,...). The wall reacts to the event, establishing how to change its state (totally or partially knocked down, to be performed, do nothing...,). After that, the wall returns an event to the projectile indicating how the event has modified the projectile attributes. The projectile will determine what should do with that new event: to be deformed, fall to floor, change velocity (module and direction),… The projectile will not accept this kind of event if it does not comes from the wall the projectile has collided with.

As a consequence of event, the behavior of the receiving object depends on the object transmitter and the kind of event sent. The receiving object will determine if it is sensitive to that kind of event coming from that specific object. If is it so, the object will determine what kind of behavior is to be presented. For example, if a small stone collides with a wall, perhaps the wall attributes will not be affected with that event (depending on the weight of the stone). If the projectile is a missile, the wall can disappear. The same event with different transmitting object causes different behaviors in the same receiving object; but also in the transmitter (the bullet rebounds if the wall is made of concrete and it is incrusted if the wall is made of clay). Different kinds of objects have different behaviors.

The number of events generated in an interaction depends on how programmer models the interaction.

Figure 2 shows another example of objects interaction using events.

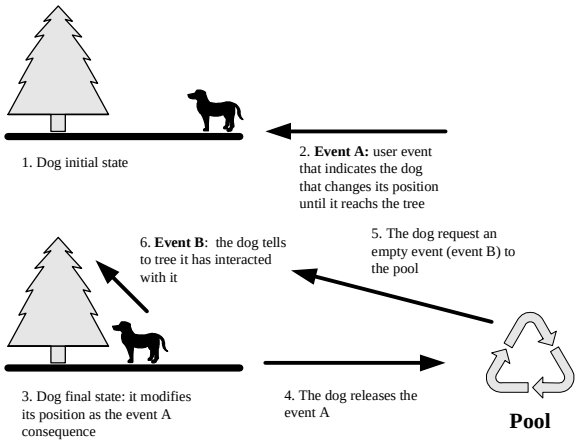

Figure 2. Events generation example

An event needs to include the following information:

- Source object: the object that generate the event.
- Destination object: the object receiving the event action.
- Event time stamp. It is the given time when the event must happen. It is a relative time. It defines the time that must pass until the event will be executed. A value cero in the time stamp indicates the event is instantaneous, so the event must happen in the actual time. The object time does not depend on a local objects clock. The object works with time intervals. The event time

stamp will be converted to a global simulation time by the dispatcher.

- Videogame information: the interaction between objects or the object behavior modeling needs some parameters to model that interaction. A destination object *A* may behave into different ways depending on the information associated to the event sent by the source object *B* and the object *B* itself. Videogame attributes are defined by the programmer (kind and number), because the videogames attributes are highly dependent on the specific videogame. The programmer has the responsibility to synchronize the data type with the information required.

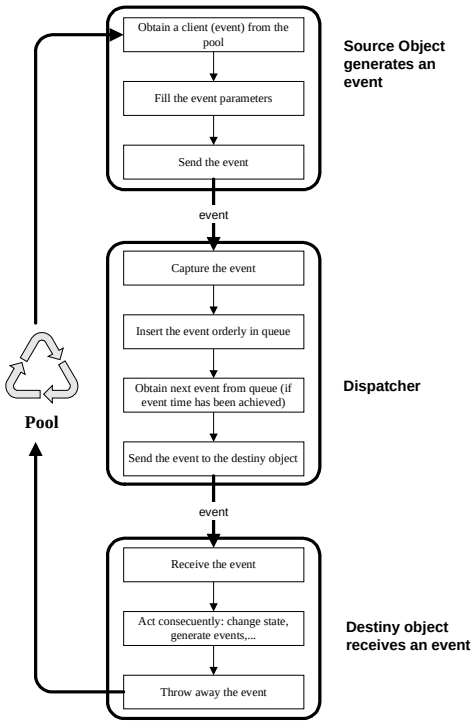

Figure 3. Event life cycle

Figure 3 show the full event life cycle. An object wants to interact with other object. So, it generates an event directed to other object. The event is really caught by the discrete event simulator dispatcher (figure 4). The dispatcher stores the event ordered by time until the event time stamp is reached. At this moment, the dispatcher processes the preceding events.

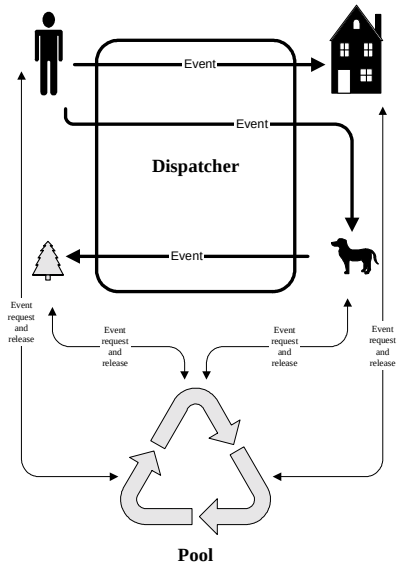

Figure 4. Events communication mechanism

GDESK Structure

The main GDESK components are the dispatcher and the events pool. They are similar to the DESK structures. The differences are the queue structure to maintain events ordered and the way the dispatcher manages the simulation clock to match it with the real time.

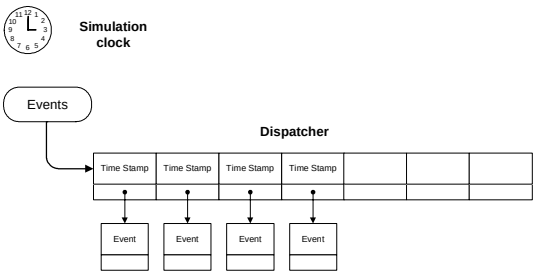

Figure 5. GDESK dispatcher

The dispatcher (figure 5) manages the videogame events. It catches the events sent from one object to another and stores them ordered by time.

The dispatcher synchronizes the videogame objects events. The videogame object event time is relative. It defines an interval of time that will pass when the event will be executed by the dispatcher. The relative event time is converted to an absolute videogame time using the global simulator clock. The dispatcher stores the event ordered by that absolute simulation time.

Once an event is stored, the dispatcher goes on working, testing if the first stored event time stamp

has reached the real time. If it is, the dispatcher sends the event to the destiny object.

The dispatcher main structures are:

- The global simulation absolute clock.
- The structure to store ordered events.

The main difference between a dispatcher in a traditional discrete event simulator and the dispatcher in the discrete event simulator integrated into the videogame kernel is the time management. The time management in a discrete event simulator does not take care of real time. The simulator has an internal clock that evolves each time an event happens. Event time in videogames must match the real system time. An event only happens if the event time stamp is reached and exceeded by the real system time. At this moment, the simulator clock is modified with the event time.

The dispatcher uses two clocks:

- The simulation global clock: it is updated each time an event is processed.
- The real time clock or the system clock: it is used to test if the events time is reached or overlapped.

The structure used to maintain events ordered by time may be a heap, a queue,...

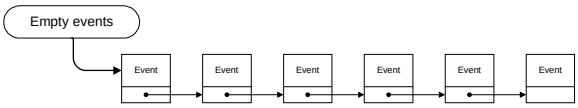

Figure 6. GDESK events pool

The events pool (figure 6) is used to store the events that are not currently in the system. Its function is to minimize the DMM calls. The objects ask the pool for empty events. So, there is a direct communication between the videogame objects and the simulator pool.

System Dynamics

The system dynamic is shared by the objects and the dispatcher. Events are passive entities that support the objects communication. The whole process can be seen in the figure 7.

Simulation Clock

Continuous system works using a global clock that defines steps in the simulation process. For each step the whole system is evolved and rendered (coupling). The global clock in a continuous system has the function to define a simulation step to synchronize objects. But, discrete systems lack from steps for the system evolution.

In a discrete system, objects act in response to events received by other objects or by themselves. If an object is stopped, it has no motion events, although it can have other pending events (for instance, a counting down bomb). A moving ball shows a continuous behavior that has to be sampled matching the Niquist-Shannon theorem. The ball implements the sampling frequency (*SF*) by sending an event to itself every *1/SF* seconds.

Each videogame object step could be constant or variable depending on the object behavior and other objects interaction. In this case, the step may be different for each object. To have a different step for each object does not suppose necessarily to have a clock for each object. The dispatcher synchronizes the objects event time to match with the global simulation clock. That global clock is modified each time an event is processed by the dispatcher.

A continuous system is a discrete system where the steps of all objects are constant and identical (including the render object). The step value is highly dependent on the system load and power.

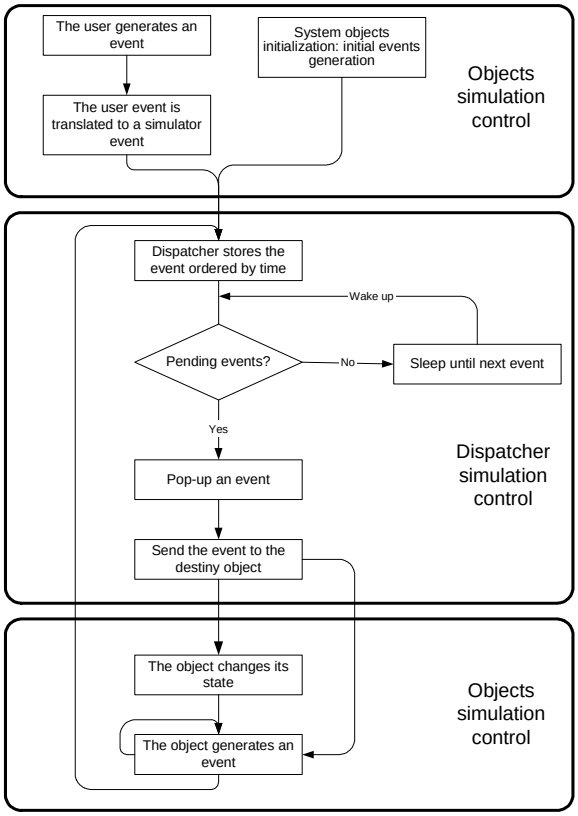

Figure 7. System dynamics

5. GDESK INTEGRATION

Any object in a videogame must inherit from the GDESK basic object entity and to provide it with the functions to generate events and to be sensitive to the event arrival:

- Send event function: it takes a previously filled event and it sends the event to another object through the dispatcher. The videogame programmer uses that function as the event receptor was the destiny object. The simulator dispatcher will send the event to the destiny object when the event stamp will reach the real time.
- Receive event function: it is a virtual function. That function starts to work when an object receives an event. As an object only acts as a consequence of an event arrival, the receive event function model the change of state and the object response to an interaction. That function implementation depends on the videogame programmer. The function implementation models the object behavior using the events mechanism. That function is used each time an objects interaction is needed.

6. THE RENDERING PROCESS USING GDESK

GDESK allows the independence of the simulation process from the rendering process (simulation phase and rendering phase decoupling). If simulation and rendering are decoupled, scenes are rendered more quickly even when the higher-level animation computations become complex [Sha92]. This decoupling increases system performance [Dar95].

The render process is controlled by the render object. The render object is similar to other videogame objects. It models its behavior using the event generation mechanism, so the rendering process must be started using an event. When a render event is generated the frame render is started.

In the system initialization, an event is sent to the render object. That event starts the render mechanism. Once that initial event is generated, the render object has an autonomous behavior. When the render object receives an event:

- The render object renders a frame.
- Once the frame *N* is rendered, the render object sends an event to itself in order to perform the next frame rendering (*N+1*). The render object generates an event addressed to itself. The event time is the time interval until the next frame render. So, the render object will receive an event from himself once the event time will pass.

The render object must decide the very moment to generate a new render event, that is, the time when a new frame is calculated. The screen refresh rate depends on the number of events generated by the render object.

The screen refresh rate can be fixed to the videogame needs. The screen refresh rate is defined by the videogame programmer:

- It can be defined by the programmer to have a screen refresh rate constant during all videogame execution. The render object events time stamp is a constant value previously defined.
- It can be an “adaptable rendering”. The programmer defines a mechanism to change the render object events time stamp depending on topics as the system load. That process is fully defined by the programmer. So, that process allows the videogame to adapt the render process dynamically to the videogame characteristics.

Although the render process is fully defined and controlled by the programmer, the render object objectives must be:

- To generate only as many renders as screen refreshes. If the number of frames generated is higher, the computer power is wasted.
- The given moment of event generation must allow to render a frame before the next screen refresh. That supposes to show always the latest frame.

Discrete decoupled system can avoid unnecessary renderings in systems with low computer power (simulation time and render time is bigger than the refresh interval). The render object can decide to generate a render event in a refresh interval if it knows that there is a possibility that the frame will be shown in the screen. Alternatively, it can decide to put off the render to the next refresh interval and go on simulating.

The render object is a common videogame object. The event generation mechanism is similar to other videogame objects. It only uses that mechanism to model its behavior. But, that mechanism could be used to interact with other videogame objects, in order to adapt the system to more complex rendering behaviors.

7. CONCLUSIONS

Current videogames follow a scheme of simulation phase and rendering phase coupling. They use a continuous simulation model. That way of operation is inefficient and may produce erroneous simulations. Using a discrete event simulation paradigm, those

problems can be avoided. That paradigm can be achieved using a discrete event simulator as a videogame kernel. GDESK is the adaptation of DESK (discrete event simulation kernel) to videogames kernel. GDESK allows the complete system to work using discrete events. The videogame is a set of objects interchanging events. The events are managed by the GDESK dispatcher. It executes the events ordered in time. The GDESK integration into a videogame produces the change of the videogame simulation paradigm. That produces a more accurate simulation and saves computing power due to the saves of unnecessary renderings.

8. ACKNOWLEDGEMENTS

This work has been funded by the Generalitat Valenciana OCYT CTIDIB/2002/344.

9. REFERENCES

[Ban01] J. Banks, J.S. Carson II, B.L. Nelson, D.M. Nicol. Discrete-Event System Simulation. (Prentice Hall, 2001).

[Dar95] R. Darken, C. Tonnesen, K. Passarella. The Bridge Between Developers and Virtual Environments: a Robust Virtual Environment System Architecture. Proceedings of SPIE 1995, No. 2409-30

[Doom] Doom World. http://doomworld.com/

[Fis78] G.S. Fishman, Conceptos y Métodos en la Simulación Digital de Eventos Discretos, (Limusa 1978).

[Fly3D] FLY3D Main Page. http://www.fly3d.com.br

[Gar00] García, I. Mollá, R. Ramos, E. Fernandez, M. D.E.S.K. Discrete Events Simulation Kernel. Eccomas conf.proc. 2000

[Ids] Idsoftware Page. www.idsoftware.com/archives/doomarc.html

[Kul03] J. Kuljis, R.J. Paul, Web-based discrete event simulation models: current status and possible futures, Simulation and gaming. v.34 no.1. (2003).

[Lee99] G.S. Lee, Towards an integration of computer simulation with computer graphics. Proceedings of the Western Computer Graphics Symposium. (1999).

[Pau95] 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.

[Quake] Quake Developers Page. www.gamers.org/dEngine/quake/

[Ree83] W. T. Reeves. Particle systems: A technique for modeling a class of fuzzy objects. In Computer Graphics, pages 59–376. ACM Siggraph, July 1983

[Sch99] T.J. Schriber, D.T. Brunner. Inside discrete-event simulation software: how it works and why it matters. Winter Simulation Conference. 1999.

[Sha92] C. Shaw, J. Liang, M. Green, Y. Sun The Decoupled Simulation Model for Virtual Reality Systems. CHI'92, May 1992, pp. 321-328.

[Ter88] D. Terzopoulos, A. Witkin, Physically Based Models with Rigid and Deformable Components. IEEE Computer Graphics and Applications, p. 41-51. (1988).

[Wai96] G.A. Wainer. Introducción a la Simulación de Sistemas de Eventos Discretos. Technical Report: 96-005. Buenos Aires University.

[Wat01] A. Watt, F. Policarpo. 3D Computer Games Technology: Real-Time Rendering and Software. Addison-Welsey. 2001.

[Wat03] A. Watt, F. Policarpo. 3D Computer Games. Addison-Welsey. 2003.

[Ziron] Ziron Page. http://www.ziron.com/links/