From Continuous to Discrete Games[1]

Inmaculada García[2], Ramón Mollá[2], Pedro Morillo[3]
[2]*Technical University of Valencia*
{ingarcia,rmolla}@dsic.upv.es
[3]*University of Valencia*
{Pedro.Morillo}@uv.es

Abstract

Computer games follow a scheme of continuous simulation, coupling the rendering phase and the simulation phase. That way of operation has disadvantages that can be avoided using a discrete event simulator as a game kernel. This paper proposes to integrate a discrete event simulator (DESK) to manage the videogames events. The videogame kernel used is Fly3D. The new kernel is called DFly3D. It allows a discrete event simulation scheme and the rendering and simulation phase independence. The integration objective has been to maintain the Fly3D main structure and functionality, changing only the events management. The videogames objects behavior and interconnection is modeled by message passing. Maintaining the videogame quality, the videogames created using DFly3D allow to reduce the computer power used to execute it. That allows to execute the videogame in computers with less computing power or to improve the game quality.

1. Introduction

Only a few commercial computer games have their source code published. Among them, we have considered the games DOOM v1.1 [6], QUAKE v2.3 [9] and the Fly3D kernel [11][12] because of their importance in computer games. The study has considered too videogame kernels as: 3D GameStudio [2] or Crystal Space [3]. But, they have not been selected because most of them are only rendering kernels and the conclusions obtained for some videogames created using those engines follows the same simulation scheme. We have selected in the present study Fly3D as a videogame kernel to make the integration with a discrete event simulator. The reasons are: freeware license, open source code fully developed using C++, documented code, object oriented, modularized and highly structured, and plug-ins oriented.

A Fly3D real-time application is a collection of Fly3D objects. Fly3D main loop follows a typical scheme of continuous simulation that couples the simulation phase and rendering phase [8]. The simulation process is executed for each main loop evolution. The simulation takes care of the time elapsed from last simulation until now. The simulation process calls each active object simulation function. For each main loop step a complete simulation and rendering is done.

Simulation techniques [1] used in videogames suppose in many cases consider computer games as continuous systems. Videogames objects could have both continuous and discrete behaviors. Computer games are usually implemented as continuous coupled systems. A continuous coupled simulation model has disadvantages [5], as:

- All objects in the scene graph are accessed, although many objects will never generate events
- The objects priority for simulation depends on the objects situation in the scene graph.

A discrete events simulator can be implemented in three different ways [5]: programming languages, general purpose programming language libraries (DESK [4] or SMPL [7]) or toolkits. The simulator selected for the integration into the videogame is DESK because it is open source code, it is implemented as a library and in a general purpose language commonly used in the videogames implementation and widely used by the scientific community.

[1] Supported by the Spanish MCYT under grant TIC2003-08154-C06-04

2. Objectives

The main objective is to change the videogame simulation paradigm from discrete to continuous. To achieve that objective, the discrete event simulator DESK is included into the real-time kernel Fly3D. The discrete event simulator manages the application events. The paper objectives are:

- Adapt the discrete event simulator DESK to work as videogame kernel (GDESK).
- Integrate GDESK into Fly3D SDK. GDESK, used as videogame kernel, allows: to change the videogame simulation paradigm from continuous to discrete and to decouple the simulation phase and the rendering phase.

3. GDESK

GDESK (**G**ame **D**iscrete **E**vent **S**imulation **K**ernel) [5] is a real time applications simulation kernel that copes with the videogame events (messages) handling. GDESK controls the objects communication by message passing. GDESK maintain the messages ordered by time until their time stamp is exceeded. GDESK uses two basic entities to model the message passing mechanism: objects and messages. **Objects** are the dynamic system entities. Any videogame element must be a GDESK object. GDESK objects include both game functional components objects (console or render) and game objects (walls, players, missiles or balls). GDESK treats any object in the same way. The GDESK basic object contains the functions to send and receive messages. **Messages** are the passive elements to communicate objects. The system dynamic is modeled by message passing.

4. DFly3D: Discrete Fly3D

DFly3D (**D**iscrete **Fly3D**) is the modified Fly3D kernel result of using GDESK to manage Fly3D events.

Once the integration is done, the videogame is an objects collection interchanging messages.

The application main loop must be changed to cope with discrete events and to decouple the system. The DFly3D main loop supposes to change the Fly3D simulation function by the GDESK simulation function and remove the rendering function from main loop. The rendering is integrated in the simulation phase to decouple the system [5].

There are two object types in the DFly3D kernel. **System objects**: they are the objects explicitly created in the GDESK integration to develop some Fly3D kernel tasks. System objects do not exist in Fly3D. **Videogame objects**: they are the objects created by the videogame programmer (as walls, characters or weapons). They join Fly3D and GDESK basic objects characteristics. The object behavior and the object interaction with other objects must be modeled by message passing. Both object types generate messages. GDESK does not make any difference between the system objects messages and the videogame objects messages. An object only works when it must to do it. It is not necessary to ask the console periodically for pending tasks. The object defines its behavior as response to a message arrival. It decides how to act when a message is received. Depending on the kind of message and the message parameters, the object response will be different.

Messages have two utilities:

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

The programmer defines the object behavior using the receive message function.

Let suppose two videogame characters (*A* and *B*) interacting:

- *A* wants to interact with *B*. *A* fills the videogame message parameters and invoke the send message function.
- The dispatcher catches the message and calculates the absolute simulation message time using the message time stamp and the simulation clock. The message is stored ordered by time. The dispatcher goes on with the system simulation. It executes the events with absolute time lower than the *A* message. When the *A* message absolute time exceeds the simulation clock the message is removed from the dispatcher. The dispatcher sends the message to the destiny object (character *B*).
- The object *B* receives the message. The character *B* does not realize the message comes from the dispatcher. The character *B* seems the message comes from *A*. *B* acts as the message arrival consequence. It can change its state or/and it sends messages addressed to other objects or a message addressed to itself. *B* releases the message.

5. Results

The results have been obtained creating two videogame versions for both Fly3D and DFly3D kernels.

We say that the computer system is *collapsed* when a system is not able to show the number of frames per second specified and render the scene properly. A system could be collapsed due to the videogame scene or simulation complexity, because of the low computing power. If the system is collapsed both kernels do not allow running he videogame properly. The given moment of system collapse depends on the videogame complexity and the kernel used.

Be:

- T_V: videogame total time.
- T_R: scene rendering time.
- T_S: simulation time.
- T_F: released time. Time when the videogame is idle if there is enough computer power.
- SRR: (screen refresh rate) number of frames per second (fps) that the computer shows in a second.
- B: number of balls.

Both kernels accomplish:

$$T_V = T_R + T_S + T_F \quad (1)$$

The system load depends on the number of objects in the system. The scene rendering time and the simulation time grow as the objects number does:

$$B\uparrow \rightarrow T_R \uparrow , T_S \uparrow \quad (2)$$

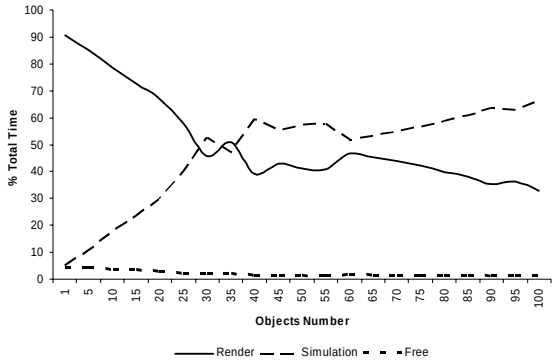

Figure 1: Fly3D times

Fly3D kernel follows a coupled scheme of simulation and rendering. The system is continuously simulating and rendering at its maximum speed (figure 1). The system uses nearly the 100% of application time rendering and simulating:

$$T_V \approx T_R + T_S, \quad T_F \rightarrow 0 \quad (3)$$

An increase in the simulation load supposes to decrease the rendering time (equation 4). The SRR decreases too. If SRR is lower than 25fps the videogame is not showed properly.

$$T_S \uparrow \rightarrow T_R\downarrow \rightarrow SRR\downarrow \quad (4)$$

An increase in the rendering load supposes to decrease the simulation time (equation 5) because the time T_R+T_S remains although a change of T_R or T_S.

$$T_R \uparrow \rightarrow T_S\downarrow \quad (5)$$

This behavior is efficient when the computer power is low or the videogame complexity is high; because it allows to simulate and to render at the maximum speed.

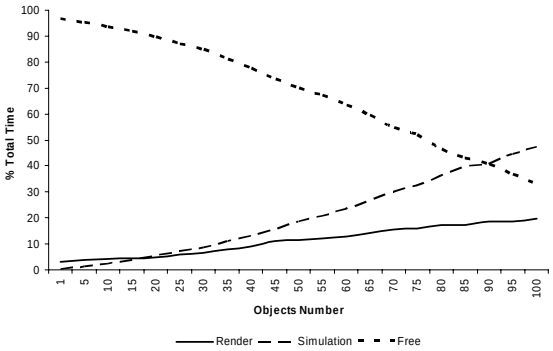

Figure 2: DFly3D times

DFly3D kernel allows the simulation and rendering processes independence (decoupling) (figure 2). So, the videogame time is not shared by rendering and simulation processes (equation 6). If the computer power is enough there is released time.

$$T_V > T_R + T_S \quad (6)$$

DFly3D kernel defines the SRR generated by the videogame. The SRR is maintained while the system is not collapsed. The time spent rendering a frame depends directly on the scene complexity. The whole rendering process consumes the time necessary to show the scene SRR times per second. The rendering time does not depend on the simulation load while the system is not collapsed (equation 7). In that situation, DFly3D releases computing time to be used by other videogame tasks or by other applications. The simulation time depends on the world simulation complexity. If the number of objects grows, both the simulation process and the rendering process consume a bigger amount of processing time.

While the system is not collapsed; the equation 6 is accomplished and the simulation time and the rendering time are not dependent (equation 7).

$$T_S \uparrow \rightarrow T_F\downarrow , T_R \uparrow \rightarrow T_F\downarrow \quad (7)$$

If the system is collapsed, T_F is nearly 0 (equation 8). The system is not able to simulate and rendering so many times to guarantee the videogame quality.

$$T_F \approx 0 \Rightarrow T_V \approx T_R + T_S \quad (8)$$

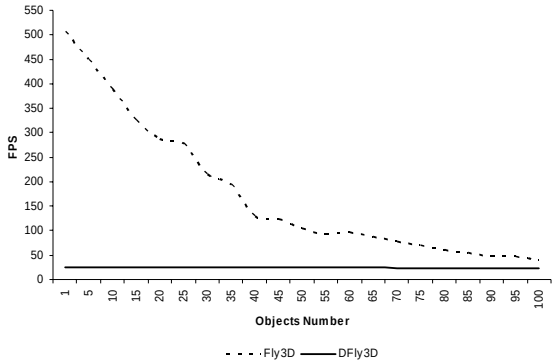

Figure 3: Frames per second generated

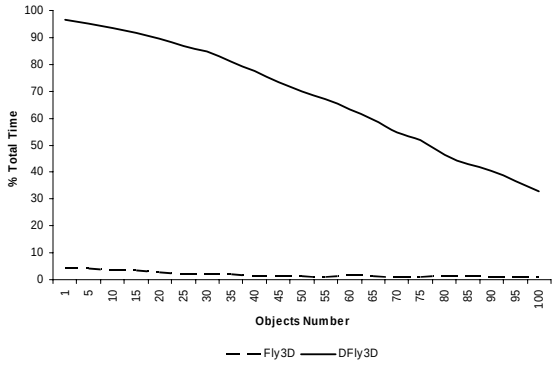

Figure 4: Released time

The DFly3D fps generation ratio is maintained during all tests (figure 3). This ratio is maintained until the system collapse. The number of frames per second generated using Fly3D is highly dependent on the computing load, so it can not be managed directly (figure 4).

The Fly3D rendering time decreases with the amount of balls because of simulation time rises. Thus, there is less time for rendering. The rendering time in DFly3D is lower than the rendering time in Fly3D. Fly3D generates unnecessary renderings. DFly3D avoids unnecessary renderings; delivering the released computer power to other tasks or improving some videogame parts, as artificial intelligence, collision detection accuracy or increasing realism.

An advantage of DFly3D is to allow an *intelligent rendering*. DFly3D kernel controls the rendering process. Intelligent behaviors can be added in order to adapt the render to the desired behavior. The SRR may change depending on the system load, avoiding renderings that will never be shown on the screen. The SRR can be adapted to the world complexity allowing a more complex simulation without collapsing the system. That allows a certain videogame independence of the computing power. A videogame can be executed in computers with different computing power maintaining the game output quality.

6. Conclusions

DFly3D is the kernel resulting from the integration of the discrete event simulator GDESK in the real time application kernel Fly3D. The integration objective is the Fly3D event handling by GDESK. Using GDESK to manage events, the kernel changes from continuous to discrete, avoiding the continuous system disadvantages (as disorderly events execution or event lost).

DFly3D decouples the simulation and the rendering phase. It allows the both processes independence. The videogames SRR can be fixed to the minimum to guarantee the videogame quality.

DFly3D releases computing power if the system is not collapsed. So, videogames quality could be improved or videogames could be executed in computers with lower computing power without quality lost.

7. References

[1] Banks, J., Carson II, J.S., Nelson, B.B., Nicol, D.M., Discrete-Event System Simulation, Prentice Hall International Series in Industrial and Systems Engineering, 2001.

[2] Conitec. http://www.conitec.net/.

[3] Crystal Space. http://crystal.sourceforge.net/drupal/.

[4] García, I., Mollá, R., Ramos, E., Fernández, M., "DESK Discrete Events Simulation Kernel", ECCOMAS, 2000.

[5] García, I., Mollá, R., Barella, A., "GDESK: Game Discrete Events Simulation Kernel", Journal of WSCG, 2004

[6] Idsoftware. www.idsoftware.com/archives/doomarc.html.

[7] MacDougal, M.H., SMPL - A Simple Portable Simulation Language, Amdahl, 1980.

[8] Pausch, R., Burnette, T., Capehart, A.C., Conway, M., Cosgrove, D., DeLine, R., Durbin, J., Gossweiler, R., Koga, S., White, J., "A Brief Architectural Overview of Alice, a Rapid Prototyping System for Virtual Environments", IEEE Computer Graphics and Applications, 1995.

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

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

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

[12] Watt, A., Policarpo, F., 3D Computer Games. Addison-Wesley. Vol.2. 2003.