Design of a Delayed File Loading Module

Nakhoon BAEK[†]*, Suwan PARK[†], Seong Won RYU[‡], Chang Jun PARK[‡]

[†] School of EECS, Kyungpook National University, Daegu 702-701, Korea
[‡] Digital Content Research Division, Advanced Game Technology Development Center,
Electronics and Telecommunications Research Institute, Daejeon 305-700, Korea
* corresponding author, oceancru@gmail.com

Abstract: Including three-dimensional interactive graphics applications, recent application programs often perform the file loading operations to read texture and data files from hard disks. In most cases, these file loading operations are implemented as serialized operations, with which the overall programs are suspended to wait the completion of the file loading operation. In this paper, we propose a new file loading approach whose operations are not easy to be recognized explicitly, through perform the file loading operations with satisfying the time constraints on the major work flow. We represent the basic ideas and implementation methods for our proposed approach. Interface designs for the C++ classes are also presented. Finally, we found that the file loading operations are well executed as we originally designed.

Key-Words: - file loading, delayed action, time constraint.

1 Introduction[*]

File loading operations are frequently used, and required by so many programs. Generally, they implement this operation as serialized forms. In other words, given a file name, system calls such as open(…) or fopen(…) read the complete contents of the corresponding file, and stores the data into a buffer to finally return to the caller function. From the viewpoint of caller functions, it is suspended to allow the called function entirely perform the file loading operations[Glas03].

In contrast, mainly due to the wide spread use of windows systems, event-driven approaches are also required. Since user inputs may be entered in any time, these programs should avoid the suspended cases of the overall programs[Petz02].

File loading operations are also not an exception. Especially, three-dimensional interactive graphics programs usually show rapid drops in their frame rates, when the overall programs are suspended. In contrast, recent three-dimensional graphics programs are easy to require lots of texture files for high-quality screen outputs. And thus, they need lots of file loading operations.

As a specific example, game application programs require loading of various files such as sound files, map files, and others, in addition to the texture files. In contrast, from the user's point of view, it is very annoying if the system frequently suspended mainly due to the file loading operations. Thus, in these days, some computer games use more complicated file loading mechanisms, in which users are hard to recognize the file loading operations. [Bila01]

In this paper, we present a new file loading approach, which performs file loading operations silently, to finally avoid user's explicit notice to the file loading operations. In section 2, the formal definitions of the problem would be represented. We also show C++ implementation methods in section 3, and our final conclusions are shown in section 4.

2 Problem Formulation

We should consider some constraints on the file loading operations. First, we assume that the file loading module should be executed on a single CPU core, with other time-critical or time-limited operations. Reversely, if the file loading module can be exclusively executed on a single CPU core, a brute-forth approach, or immediate execution of requests would be the best strategy. In contrast, according to the overall architecture of real-world applications for PCs or game consoles, the file loading operations have relatively low priorities. Thus, they

[*] This work was supported by the IT R&D program of MIC/IITA. [2007-S001-01, Development of Multi-Core CPU & MPU-Based Cross-Platform Game Technology]

tend to share a CPU core with other, more important operations.

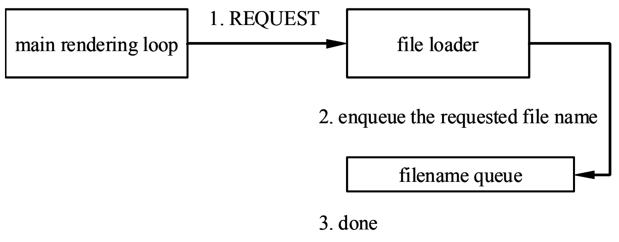
As an example, for a typical three-dimensional graphics program compiled for a single-core CPU, the most important constraint is generating 60 frames per second from the rendering module. In this case, the file loading operations would be performed only for the times remained after the completion of this important constraint. In this configuration, we can meet a failure situation of too-late file loading operations, due to the CPU over-usage of the rendering part. However, typical graphics applications use the texture files mainly to improve visual effects, and file loading failure does not make any critical damage on visual information. Thus, in this paper, we assume that file loading operations are not time-limited jobs, and aim to design a file loading scheme to smoothly loading the files without making the overall system suspended.

For the ease of explanation, we assume that the main rendering part performs all the works except the file loading ones, with some time limits. So, the main rendering part and the file loader do their own works in shifts, while the main rendering part should be repeatedly invoked for every time slice T and thus the execution time of the file loader may vary depending on that of the main rendering part. Now, the file loader will work only for ($T - t$) time slices, to satisfy the time constraint of the main rendering part.

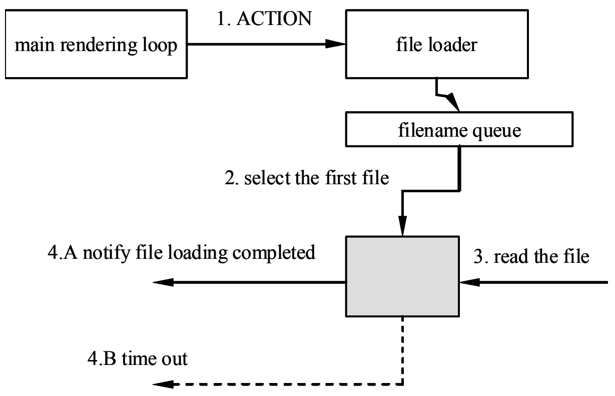
As shown in Figure 1.(a), the overall scenario starts from instructing the file loader to read some files which would be used for late rendering or calculations, during the execution of some time-limited jobs in the main rendering part. Then, the file loader will enter the file name and its related information into a FIFO queue. From the viewpoint of the file loader, since the current time slice is not its turn, it only records the file name and returns the CPU control to the main rendering loop immediately.

After the rendering job of the main rendering part, there is the remained time ($T - t$) until the next invoke of the rendering part. Now, as shown in Figure 1.(b), the main rendering part passes over the CPU control to the file loader, for the remaining time ($T - t$). The file loader fetches a file name from the FIFO queue, and starts to read its content. To keep its own time limit ($T - t$), the file loader read the file in a segment-by-segment manner, rather than reading the whole content in a single action. When the time limit ($T - t$) is over before completely reading the file content, the file loader stops its operations and passes over the CPU control to the main rendering part, to finally keep the time constraint of the main rendering part.

In the next time slice, the file loader resumes to the suspended file reading operation. When a file reading operation is completed, the file loader reports the end of the file loading operation and its resulting data to the main rendering loop. It fetches the next file name and starts the next file read operation. The overall time transition diagram is shown in Figure 2.



(a) request method



(b) action method

Figure 1. Method design.

3 Class Implementation

We have implemented the file loading schemes explained in section 2 as a C++ class. Major operations are implemented into two methods, as shown in followings:

void request(**char*** *filename*, **bool*** *completed*, **void*** *buf*);

It requests the loading of the file whose name is “filename”. The parameter “completed” is the flag to indicate the status of the file loading operation. This function resets the “completed” to be false, while it becomes true after loading the file contents to the buffer area "buf", by the action(…) method shown below. Then, the caller function checks the "completed" flag and use the data in the "buf" area, when the "complete" flag is true.

void action(**float** *remained*);

Use the remained time slice ($T - t$) for the file loading operation. First, it registers an ALARM signal after "remained" time, and does its file loading operation. When the file loading is completed before the ALARM signal, the "completed" flag is set to true, and the loaded data into the buffer "buf". For the time out situations, the current operation is suspended and it returns to the caller function.

Using the above interfaces, we implemented the file loader and its corresponding rendering part. The testing result shows that the delayed file loading works well, as our original goals.

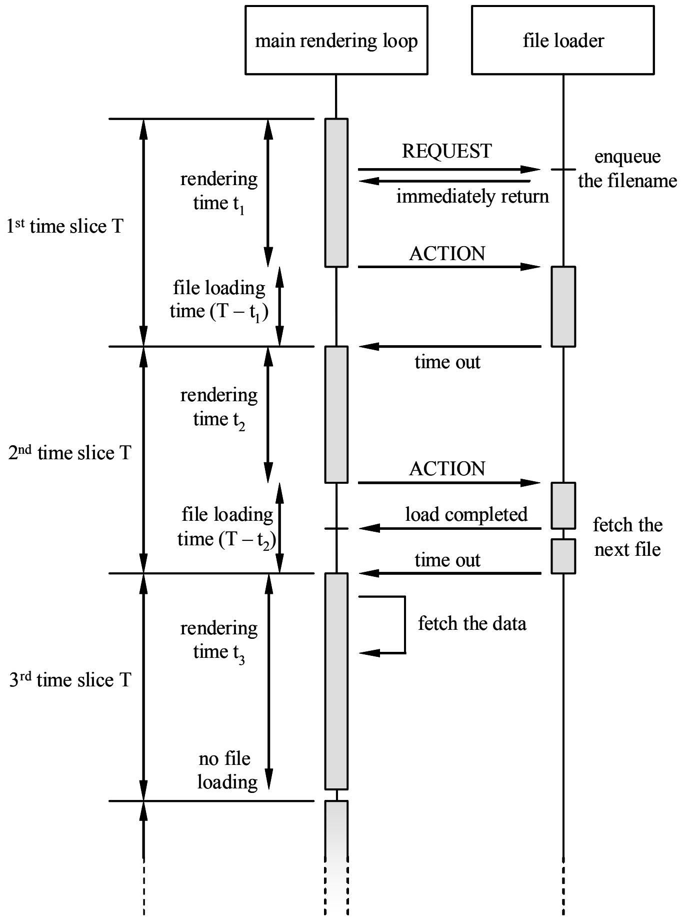


Figure 2. Overall time-transition diagram.

4 Conclusion

Many application programs frequently use the file loading operations, and in most cases, they are implemented as serialized operations. In contrast, they need to perform the file loading operations in a parallelized manner. In this paper, we propose a new file loading approach whose operations are not easy to be recognized explicitly, through perform the file loading operations with satisfying the time constraints on the major work flow. Basic ideas and its corresponding implementations are represented. We also show the interface designs for the C++ classes. We found that the file loading module works well as its original design. In the future, more improvements are needed for more efficient and intuitive implementations.

Acknowledgement

We would like to thank Ms. Eun-mi Lee for help with the partial implementation of API's. We also thank Ms. Jung Hwa Lee and Ms. Myung Lan Jo for their support.

References:

[Glas03] G. Glass and K. Ables, *Unix for Programmers and Users, 3rd Ed.*, Pearson/Prentice-Hall, 2003.

[Petz02] C. Petzold, *Programming Windows, 5th Edition*, 2002.

[Bila01] S. Bilas, "The continuous world of dungeon siege", *Game Developer's Conference 2003*, 2003.