Priority Round-Robin Scheduling for Very Large Virtual Environments

Chris Faisstnauer, Dieter Schmalstieg, Werner Purgathofer
Vienna University of Technology, Austria
Email: faisst@cg.tuwien.ac.at

Abstract

In virtual environments containing a very large number of objects, the limited amount of available resources often proves to be a bottleneck, causing a competition for those resources – for example network bandwidth, processing power or the rendering pipeline. This leads to a degradation of the system's performance, as only a small number of elements can be granted the resource required. In this paper we present a generic scheduling algorithm that allows to achieve a graceful degradation; it is output sensitive, minimizes the risk of starvation and enforces priorities based on a freely definable error metric. Hence it can be employed in virtual environments of almost any size, to schedule elements which are competing for a determined resource, because of a bottleneck.

1. Introduction

One of the biggest challenges in implementing large and complex virtual environments (VE) is the overcoming of bottlenecks that arise when elements of the VE compete for an insufficient number of resources. This resource can be the graphics pipeline for the rendering of graphical objects, the processing power of the CPU for physical simulation or network bandwidth needed to synchronize the state of the VE in a distributed system. Whenever there is such competition and not all elements that require the resource can be serviced, an approximation must be made in order not to compromise interactive performance. The techniques employed to deal with such a situation can be divided in two groups:

1. Techniques which reduce the absolute number of elements that require the resource and are thus competing for it. Examples include level of detail rendering (see e.g. [Hopp98], [Heck97], [Funk93]), visibility culling ([Zhan97], [Suda96]) and dead reckoning ([Sing95]).
2. Traditional scheduling algorithms such as known from operating systems theory deal with the issue of selecting items that are granted resources ([Silb88], [Tane92]).

However, the problems found in operating systems are not identical to the those in VEs. In particular, VEs can host a very large number of elements, so that the examination of every element in every turn is too computationally expensive. Instead, application in a VE requires an *output sensitive* algorithm that operates with constant effort independent of the number of elements. The simple *round robin* (RR) approach to scheduling has this property and is therefore often used for such scheduling problems.

But the RR strategy - simply scheduling every element in turn - cannot accommodate dynamically changing simulations. For example, a networked virtual environment may require position updates of moving entities with variable speed. For increased realism in the behavior, fast entities will require more frequent updates than slower ones. Such priorities cannot be achieved with simple RR.

In this paper, we propose an enhancement to RR called *priority round robin scheduling* (PRR). This algorithm enforces priorities, while retaining the output sensitivity and starvation-free performance of RR. Priorities are set by a user-defined error metric, which the algorithm attempts to minimize.

We will show the efficiency of our algorithm in an often encountered VE setup, namely the aforementioned networked virtual environment where position updates of a large number of entities must be scheduled for transmission. PRR is demonstrated to lead to significant improvements over a simple RR scheduling.

A technique often used in simulations based on physical motion is *dead reckoning* (DR) ([Sing95], [Mace94]), i. e. prediction based on motion extrapolation. However, DR is tied to movements based on a velocity/acceleration model; furthermore it does not consider the fact that the number of elements selected by the DR algorithm may exceed the bandwidth of the network. We demonstrate how PRR can be employed to enhance the DR algorithm.

2. Related Work

Many of the techniques developed to achieve a graceful degradation in bottleneck situations do not schedule the elements competing for the resource, but rather try to reduce their absolute number. Geometric Levels of Detail (LOD) for example reduce the number of geometric primitives submitted to a rendering pipeline; dead reckoning allows to reduce the number of updates that are necessary to syncronize the movement between a simulator and connected clients. Other approaches exploit the hierarchical structure of a scene to limit the number of tree-branches to traverse (e.g. when determining which objects to simulate or render).

The area that bears the greatest similarities to our requirements for an output sensitive scheduling algorithm is the short-term scheduling problematic well known from the operating systems research: independent processes, competing for the CPU power, have to be allocated processor time in such a way as to optimize one or more aspects of system behavior (see e.g. [Stal95], [Deit90], [Silb88]). The simplest scheduling policy is executing the processes one after another in the order they are submitted (that is, using their age or time of arrival as priority), called *First Come-First Served (FCFS).* FCFS is a so called “non-preemptive” algorithm, where a selected process can terminate before a new process is scheduled. In “preemptive” algorithms the currently running process may be interrupted to schedule another process. This happens for example in *Round Robin (RR*) scheduling, the preemptive version of FCFS, where each process is removed from the resource and re-inserted at the end of the queue after exceeding a determined time-slice. As it is output sensitive and immune to starvation, it is often employed; however its performance is limited as it does not enforce priorities.

A scheduling algorithm which employs priorites and also includes a feedback from the system is the so called *Multilevel Feedback Queue:* it consists of levels with decreasing priorities, and the algorithm, starting with the highest level, picks all objects present in that level in a round-robin fashion. After a determined timeslice the actually selected process is preempted and moved to the next lower level; thus the priority of the processes is decreased with increasing execution time. When a level is empty, the algorithm selects the processes of the next lower level. As new processes are inserted in the highest level, it may lead to starvation of processes in the lower levels; to overcome this problem, processes waiting to be scheduled may be raised to a higher level after a determined amount of time. However, it is not apt to employed as generic-purpose scheduling algorithm for virtual environments, as there are substancial differences between the scheduling of processes and elements in a virtual environments:

- Processes are usually scheduled only once (except rescheduling because of preemption), after which they are terminated and removed from the scheduling queue. For further scheduling, the processes have to be re-submitted to the algorithm, where they are treated as new processes. In virtual environments, we often have to schedule the same element repeatedly (e.g. recurringly sending the position updates for a moving car).
- If priorities are employed by process scheduling algorithms, then to determine which process is to be selected next: they always select the process with the highest priority, which may lead to starvation of lower priority processes (e.g. in the MultiLevel Feedback Queue). Techniques to avoid starvation employ a constant monitoring of all processes to treat lower level processes (or penalize high level processes); this leads to an overhead depending from the number of processes.
- Process scheduling algorithms usually treat with a reasonable number of processes, allowing them to continuously examine all processes to determine their characteristics. As the overhead of our algorithm should not depend on the number of elements, it is prevented from sorting or comparing all elements against each other.
- The “amount” of resources required by processes may the vary substantially, so that it is often necessary to preempt the execution of a process to be resumed later. Furthermore, it is necessary to distinguish between CPU and I/O-bound processes. For our scheduling algorithm we assume that all elements just need a small, constant amount of non-blocking resource, so that they can be serviced completely when scheduled (e.g. transmitting an update over the network, simulating the position of a car, or calculating the orientation of a skeleton joint). Hence we can neglect the notion of preemption.

What process scheduling techniques have in common with our attempt to schedule elements in virtual environments is the attempt to minimize determined system paramteres, to enforce priorities and to minimze the risk of starvation (every element should be guaranteed to be serviced at least once within a determined amount of time).

3. Basic Priority Round Robin scheduling

RR scheduling is usually implemented as a FIFO (first in first out) queue. The first element is removed from the head of the queue, scheduled, and then re-inserted at the tail of the queue. Every element is treated equally, no priorities are assigned.

However, in a VE with dynamic simulation, the state of each element is constantly changing. If the element cannot get access to a resource (e. g. the CPU for an update of a physical simulation, or the network for a positional update), it will accumulate error. To be useful for scheduling, this error must be modeled as an appropriate error metric, e. g. deviation in position. The goal of selecting a number of elements for scheduling is to minimize the cumulative error over all elements in the environment.

However, keeping all elements sorted all the time requires a significant amount of computational effort, in particular if all elements constantly change state, and thus error and sorting order must be re-evaluated for every frame. Instead, our algorithm relies on approximate sorting in multiple *levels* (FIFO queues), which combines the advantages of RR and full sorting. As the number of levels is predetermined, the effort to schedule a single element is constant, which makes the algorithm output sensitive.

The elements are assigned to one of the levels according their so called *error per unit* (EPU). The error per unit of an element is a prediction - based on past performance - of how much the error metric (e. g., the visual error) will increase in a determined time unit[1]; it is the contribution of the element to the overall error per time interval. If the total error is a deviation in position, the error per unit is equivalent to velocity.

While the levels are processed in RR order, each level is assigned a priority, which must reflect the frequency with which the elements in the different levels are selected. Basically, elements with higher error must be scheduled more often than elements with lower error. The combination of traversing each level using RR, but with a different priority, gives our algorithm its name - *Priority Round-Robin* scheduling.

We call the waiting time between two consecutive schedulings of an element the *Repetition Count* (RC). Let NL denote the number of levels and NE(i) denote the number of elements in level *i*. Then for an element in level *i*, the repetition count RC is simply

$$RC = NE(i) \cdot NL \qquad (1)$$

The repetition count is the number of scheduling actions an element has to wait between two consecutive selections; it is a measure for the time an element has to wait and thus for the cumulative error generated by the element until the next scheduling. The way how the levels are traversed by this selection strategy is depicted in Figure 1. From the order in which the elements are scheduled, we see that the elements of the first level (A and B) must wait 6 times between two consecutive schedulings, those of the second level have an RC of 12, and element G is scheduled every 3 elements.

[1] Note that the time unit used for the error per unit of the elements is irrelevant for the performance of the algorithm.

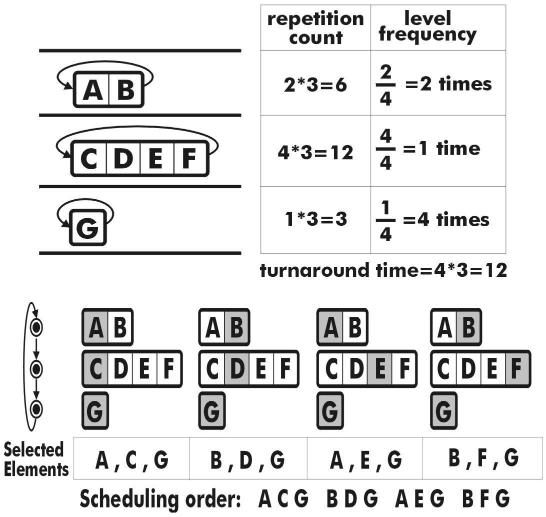


Figure 1: An example showing three levels, containing two, four and one element respectiveyl.

The more elements in a level, the longer they must wait between two consecutive schedulings. If all levels are of equal length, the repetition count of the elements is the same as if they were scheduled by RR algorithm (RC = number of elements).

We also see that in the time interval the largest level *m* is traversed exactly once, the other levels i (of equal or smaller size) are traversed at least once. We thus define the level frequency LF(i) of level *i* as

$$LF(i) = NE(m) / NE(i) \qquad (2)$$

Whenever the largest level is traversed exactly once, all elements have been scheduled at least once; those of the largest level one time, and those of the other levels one or more times. The *turnaround time* TT in which all elements have been scheduled at least once is simply

$$TT= NE(m) \cdot NL \qquad (3)$$

If the EPU of an element can be used to be constant (such as for entities travelling at constant speed), a predicted error PE for that element can be made by extrapolation and RC associated with that level.

$$PE = EPU \cdot RC \qquad (4)$$

Furthermore, an estimate of the total error per level and the total error of the environment can be computed from the error per unit for each element and the RC of the levels. Keeping score of these total error measures is done incrementally with negligible overhead.

4. Scheduling for static error distributions

So far the issue of how elements are assigned to levels has not been discussed. Also, it has not been mentioned whether the number of elements in a levels is constant or variable. Assuming a constant number of elements for each level, elements in smaller levels get scheduled more often. To fulfill the requirement mentioned in the introduction that elements with a large EPU should get scheduled more often in order to minimize overall error, these elements should be inserted into smaller levels.

If the error distribution of the objects is known a priori, it is possible to fix the number and size of the levels a priori. If we define a set of levels with increasing size and insert the elements with decreasing EPU into the levels, then the larger the EPU of the element, the smaller is the level (and thus the repetition count) the element is assigned to. After having determined the optimal number and size of the levels from the error distribution of the elements based on prediction of the error, we can determine the level to which to assign an element. Each successive level covers a range of possible errors - an error interval - corresponding to the elements it contains. If the error values associated with elements are completely static, elements will stay in the level they are assigned to.

Unfortunately, dynamic virtual environments do not have a static error distribution. An element's EPU will almost certainly change each time it is inspected. Not only must the element then be inserted into another level, but also the error intervals associated with the levels must be adjusted, if the size of the levels is to be kept constant. However, we have found that for large numbers of elements, the systems response to these adjustments is slow, and the overall error is often larger than plain RR when element behavior is dynamically changing.

5. Scheduling for dynamic error distributions

In order to overcome the aforementioned problem, the size of the levels must be variable. Therefore, the error interval covered by a level is no longer an indicator of where an element should be inserted. We considered two alternative variants of how to assign an element to a level:

1. **Minimization of overall error**: The most suitable level for each element is chosen by estimating for the element's current EPU of how the overall error is affected if the element is inserted into each level. The algorithms then selects that level which leads to lowest overall error. Unfortunately, while this strategy automatically finds the best number of elements for each level, it is not superior to RR: as the assignment to a level is only dependent on global error minimization rather than directly on the EPU of the element, this algorithm tends to distribute elements with high and low errors equally on all levels. This leads to levels of equal length and equal average error, with a performance equivalent to RR scheduling. Therefore the size of the levels must be variable, and the assignment of an element to a level must directly depend on its error per unit.
2. **Average error per unit**: This approach uses an average EPU associated with each level to determine the most suitable level for an element. The average EPU is computed using a sliding average. An element is then assigned (according to its EPU) to the level with the „closest" EPU average. This does not produce a perfect grouping of the elements according to their error, but does quickly adapt to changing error distributions with no additional overhead.

After some experimentation, the second approach - based on average EPU - was chosen as the most efficient strategy.

6. Heuristic for traversal rate

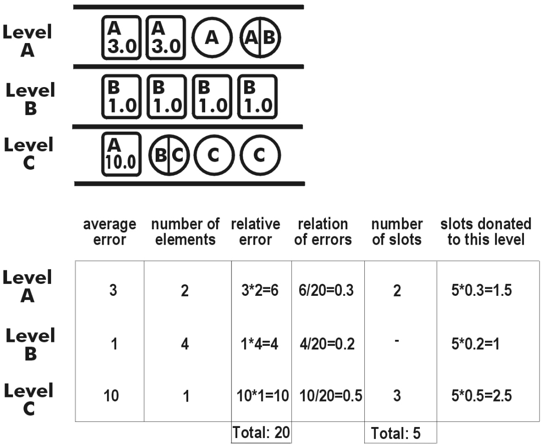
Basing our approach on an average error per unit requires us to determine for each level a different „speed" with which it is traversed, calculated from the error generated by the level. This speed, based on prediction, is a measure of how many elements should be taken from a level in each scheduling step in order to minimize the overall error. Although this allows us to find an optimal speed for each level, it has some disadvantages for practical use. It requires us either to employ statistical methods, or to solve mathematical equations with only few parameters at disposition (the number of elements for each level, and the average error per unit – all other values are predicted from these).

In practice, the different speeds of the levels may vary by several orders of magnitude. The elements in levels which produce a very low error will be visited only in very large time intervals. As an element can only be re-evaluated when it is scheduled, and elements in levels with very low speeds must wait long for the next scheduling, their error can deviate significantly from the predicted error using the last known EPU, effectively rendering the prediction useless. This leads to the situation that the PRR reacts too slow. If we do not set a limit that denotes the minimal speed a level should have, it may happen that the performance of the PRR may be temporarely worse than traditional RR, if some very slow objects start to rapidly increase their activity.

Unfortunately the optimal minimal speed of a level cannot easily be described with the information the PRR has at disposition. Instead a simple heuristic approach is proposed which automatically determines a minimal speed for the levels, which leads to encouraging results in our evaluations (see Section 7).

The heuristical approach starts from levels that are traversed at equal speeds, one element at a time. This leads to a repetition count depending on the number of elements contained in the level (see Section 3). If all levels are of equal length, then repeatedly taking one element from each level leads to a performance equal to RR. So we first fill up all levels with many "empty" elements (so called "*slots*") to achieve equal length. These slots are place holders for elements to be scheduled. Unlike normal elements, a slot can be used to schedule an element from any level. When a slot is processed, rather than scheduling an element from its own level, an element from another level is selected, leading to a relative speed up of that level.

The generated slots are distributed to the levels according to their relative error (the average EPU of the level multiplied by the number of elements). Figure 2 shows an example containing three levels (called “A”, “B” and “C”). The first level contains two elements with an average error of 3.0, the second level four elements with an error of 1.0, and the third level one element with an EPU of 10.0. Thus the first level generates two slots (depicted by circles), and the third level three slots, as to achieve the same length as the longest level (level “B”, with four elements). In total, five slots were generated. According to their relative error, level “A”, “B” and “C” have a relation of 0.3 : 0.2 : 0.5. Distributing the five generated slots according to this relation leads to 1.5 slots given to level “A”, 1 slot given to level “B” and 2.5 slots given to level “C” (this is depicted by the letter inside the slot).



	average error	number of elements	relative error	relation of errors	number of slots	slots donated to this level
Level A	3	2	3*2=6	6/20=0.3	2	5*0.3=1.5
Level B	1	4	1*4=4	4/20=0.2	-	5*0.2=1
Level C	10	1	10*1=10	10/20=0.5	3	5*0.5=2.5
			Total: 20		Total: 5	

Figure 2: Example of generation and distribution of slots

This approach slows down the traversal of the levels with a lower repetition count, by bringing them to the same length as the largest level (this leads to the same performance as round robin). Then the generated slots are distributed to the levels according to their need (described by the relative error); the traversal of the levels which are given slots is accelerated. In other words, the amount by which the faster levels are slowed down (a slot slows down the level in which it is generated) is distributed to all levels, according to their relative need (speeds up the receiving level). In any case, the slower a level is (the more elements it contains), the less slots are generated in it.

This heuristic determines the lowest speed of a level, and its performance provides appealing results in our evaluations, although its overhead is only insignificantely higher than standard RR. The following section contains results of the performance of the described PRR algorithm.

7. Evaluation and results

We will evaluate the performance of the priority round robin algorithm by using a client-server system typically employed in distributed virtual environments. In our test bed, depicted in Figure 3, the server hosts a simulator which moves a large number of objects (e.g. cars) on a 2D-plane, based on velocity-vectors. The client is used to visualize the scene, and thus needs updates from the simulator whenever the position of an object changes. If the number of updates generated by the simulator exceeds the capacity of the network, only a subset of the elements‘ position can be updated, resulting in a visual error; the latter is determined by the distance between an object’s position on the simulator and on the client. The overall error is the sum of the position differences of all objects.

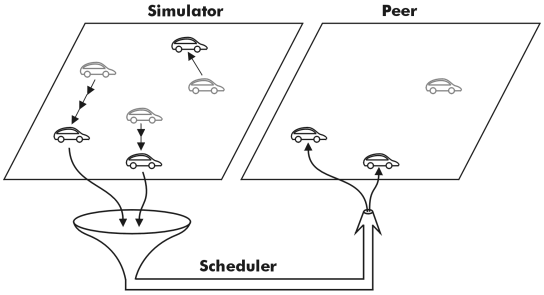
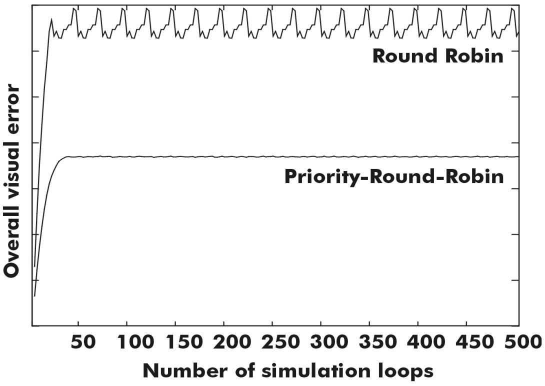


Figure 3:. Evaluation testbed

The priority round robin algorithm will be employed to minimize the (overall) visual error. In the first evaluation whose results can be seen in Graph 1 and Graph 2, it will be compared to a simple round robin scheduling. The second evaluation (Graph 3) includes the use of dead reckoning.

In this first example we let the simulator simultaneously move 10000 cars over a large 2D-plane. We keep the simulation very simple by translating the cars according to a fixed velocity vector. We compare the priority round robin algorithm to a standard round robin, by updating only a small part of the simulated elements. The overall error produced is determined by summing up the visual errors of the single cars (which is the difference between the position on the server and on the client). The velocity of an objects describes by how much its error increases per unit of time, and is thus suited to be used as error-per-update.

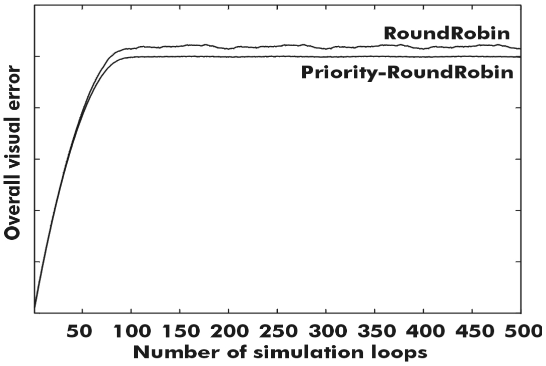


Graph 1: 76% performance increase

Graph 1 shows the overall error produced by a standard round robin scheduling compared to priority round robin algorithm when only 10% of the simulated cars are updated (in total numbers: 1000 out of 10000). The velocities of the cars allow the PRR to achieve a significant advantage advantage over simple round robin, as it can distinguish the cars into different priority groups.

100 cars get a velocity vector of a length between 90 and 100, 2000 cars get a velocity between 2 and 3, and the remaining cars gets a speed between 0.1 and 1 (direction is always random). This error distribution reflects a situation that often occurs in virtual environments: the updates of few fast-moving objects are delayed by a mass of slower moving objects. In this case the priority round robin scheduling brings significant advantage; its overall error is in average 76% lower than that produced by standard round robin.

Graph 2 also shows the performance of both algorithms when 10% of the cars are scheduled, but this time the length of the velocity vectors is assigned randomly out of an interval from 1 to 10. The uniform distribution of the errors produced by the cars allows the PRR algorithm to gain only a 3.7 % advantage over simple round robin.



Graph 2: Only 3.7 % gain

This client-server test bed is a typical example of an application than can be optimized using the dead reckoning technique. As in dead reckoning the clients move the objects simultaneously to the simulator (based on the most recent velocity-vector received), the simulator only transmits the new velocity vector (and actual position) if the visual error of an object exceeds a determined threshold. Dead reckoning already makes an efficient pre-selection of the updates to transmit to the clients. However, dead reckoning is limited if the number of updates still exceeds the network bandwidth (which might by the case if the objects frequently change direction, or perform a non-coherent motions as e.g. in video games).

Due to its freely definable error metric, the priority round robin algorithm can be employed to enhance dead reckoning, by optimizing the transmission of the remaining updates. The overall visual error is still determined by the sum of the position differences between simulator and client from all cars. But the client moves the cars simultaneously to the server, and the dead reckoning technique selects the updates to transmit according to the difference between the simulated position and an approximation of the actual position on the client's site. As on each simulation loop a different set of objects may by selected by dead reckoning, we cannot fill the priority round robin algorithm each time with a different set of objects.

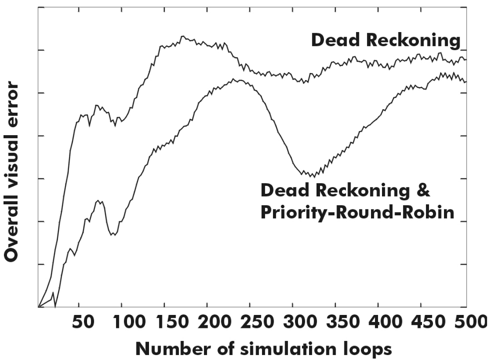
Therefore we use the priority round robin algorithm to optimize the order in which the objects are examined for transmission. We have to use the difference between simulated and approximated position, compared by the dead reckoning algorithm versus a given threshold, to determine the error-per-update used by priority round robin. If we interpret this difference as distance, and sum up its changes on each simulation step, we can compute an average velocity to use as error-per-update.

In this example we move all 10000 cars on every loop of the simulator, but the limited network bandwidth allows at most 1000 updates (10%). Thus if the number of objects that exceeds the dead reckoning threshold repeatedly exceeds 1000, the priority round robin algorithm can bring noticeable performance increases. Thus we simulate cars driving around in curves, a bad situation for dead reckoning; we let all cars change their direction every ten simulation steps.

Graph 3 shows the overall error produced by standard dead reckoning, and if enhanced with priority round robin. From all 10000 cars:

- 1000 have a of a velocity vector with a length between 9 and 10, and change their direction every ten simulation steps by an angle between 9 to 10 degrees.
- 3000 cars have a velocity between 5 and 6, and change their direction every ten simulation steps by an angle between 5 and 6 degrees.
- The remaining 6000 cars get a speed between 2 and 3; they change their direction between 2 and 3 degrees every ten simulation steps.

Given this setup, a threshold distance of 10 makes the dead reckoning select on average 25 % of the cars for update, which in absolute numbers is approximately 2500, compared to the 1000 that can at most be updated. Thus if dead reckoning is enhanced with priority round robin, the overall error produced is about 23 % lower than those produced by dead reckoning alone..



Graph 3: The overall error is decrased by 25% if dead reckoning is enhanced with PRR.

8. Conclusions

We have presented a technique to enhance standard round robin scheduling, by adding the enforcement of priorities to its advantages of being output sensitive and immune to starvation. The simplicity of PRR and its freely definable error metric make it a suitable substitution for round robin in most cases. We have shown the performance increase that can be achieved by priority round robin when scheduling simulation updates to minimize the overall visual error.

Other possible applications of PRR could be to schedule objects competing for CPU power (e.g. in order to be simulated), or be employed in the rendering pipeline. Our future work will be to evaluate the performance of the priority round robin apprach in those fields, and to determine which other system paramters can be included in the scheduling decisions without endangering the output sensitivity or immunity versus starvation of the priority dead reckoning technique.

Acknowledgements

This project was sponsored by the Austrian Science Foundation (FWF) under contract no. P-11392-MAT and by the TMR program of the European Community under contract no. FMRX-CT96-0036 (PAVR).

References

[Deit90] H. M. Deitel. An introduction to operating systems. Published by Addison-Wesley, Inc. ISBN 0-201-18038-3, 1990.

[Funk93] T. A. Funkhouser, C.H. Sequin. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. Proceedings of SIGGRAPH'93, pages 247-254, 1993.

[Heck97] P. S. Heckbert, M. Garland. Surface Simplification Using Quadric Error Metrics. Proceedings of SIGGRAPH'97, pages 43-50, 1997.

[Hopp98] H. Hoppe. Efficient Implementation of Progressive Meshes. Computers & Graphics, 1998.

[Mace94] M. R. Macedonia, M. J. Zyda, D. R. Pratt, P. T. Barham, S. Zeswitz. NPSNET: A Network Software Architecture for Large-Scale Virtual Environments. Presence, 3(4): 265-287, 1994.

[Silb88] A. Silberschatz. Operating system concepts. Published by Addison-Wesley, Inc. ISBN 0-201-18760-4, 1988.

[Sing95] S. K. Singhal, D. R. Cheriton. Exploiting Position History for Efficient Remote Rendering in Networked Virtual Reality. Presence, Vol. 4 (2), Spring 1995, pages 169-193, 1995.

[Stal95] W. Stallings. Operating systems. Published by Prentice-Hall, Inc. ISBN 0-02-415493-8, 1995.

[Suda96] O. Sudarsky, C. Gotsman. Output-Sensitive Visibility Algorithms for Dynamic Scenes with Applications to Virtual Reality. Proceedings of EUROGRAPHICS'96, Vol 15(3), pages 249-258, 1996.

[Tane92] A. S. Tanenbaum. Modern operating systems. Published by Prentice-Hall, Inc. ISBN 0-13-588187-0, 1992.

[Zhan97] H. Zhang, D. Manocha. T. Hudon, K. Hoff. Visibility Culling using Hierarchical Occlusion Maps. Proceedings of SIGGRAPH'97, pages 77-88, 1997.