Priority Round-Robin Scheduling for Very Large Virtual Environments and Networked Games

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

Abstract. *The problem of resource bottlenecks is encountered in almost any distributed virtual environment or networked game. In a typical client-server setup, where the virtual world is managed by a server and replicated by connected clients which visualize the scene, the server must repeatedly transmit update messages to the clients. The computational power needed to select the messages to transmit to each client, or the network bandwidth limitations often allow only a subset of the update messages to be transmitted to the clients; this leads to a performance degradation and an accumulation of errors, e.g. a visual error based on the positional displacement of moving objects.*

This paper presents a scheduling algorithm that enforces priorities based on a freely definable error metric, trying to minimize the overall error. It is able to achieve a graceful degradation of the system's performance and to minimize the risc of starvation, while retaining an output sensitive behavior. This makes it suitable not only to schedule the update messages to transmit to the various clients, but it also allows to employ filtering techniques at a constant effort.

1 Introduction

In the simulation of large virtual environments, contention for limited resources such as CPU, rendering pipeline, or network bandwidth frequently causes a degradation of the system's performance. Whenever there is such a 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. 'Filtering' 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. [Funk93]), dead reckoning ([Sing95], [Mace94]) and visibility culling ([Suda96], [Funk95]).
2. Traditional scheduling algorithms such as known from operating systems theory deal with the issue of selecting items that are granted resources ([Silb88], [Tane92]).

A popular approach to build virtual environments and networked games is to use a client-server architecture: the virtual world is managed by the server and (partially) replicated by connected clients, which visualize the scene and/or navigate an avatar through the environment. All updates from the clients are routed via the server (often also responsible for the simulation of autonomous entities), which can perform arbitrary filtering functions; some systems employ visibility information in order to decrease the network load, by transmitting to each client only updates for those objects visible to it. Timely delivery of update messages to clients is essential to avoid visual errors (e.g. different positions of the same object on server and client). However these approaches cause a substantial overhead to the server, as it is often required to examine all objects in the environments for each client. For example, to transmit only the visible object updates to a client, it is necessary for the server to keep track of the point of view for all clients, and continously select the corresponding visible objects. Assume n = number of clients = number of objects. This means examining all objects for all clients leads to an effort of $O(n^2)$, which substantially affects the scalability. Furthermore these filtering techniques do not deal with the issue of scheduling the remaining objects. If the number of messages to be transmitted still exceeds the network bandwidth, the bottleneck problem may persist.

In this case, or if no filtering is employed at all (as in many peer-to-peer systems), we face a scheduling problem similar to those found in operating systems research. However, scheduling in operating systems is not identical to scheduling 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 selecting every element in turn - cannot accommodate dynamically changing simulations. For example, if the server has to distribute updates of entities moving 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 plain 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 (e.g. visual error), which the algorithm attempts to minimize. This allows not only to schedule the entites competing for a resource, e.g. updates competing for the network, but also to fill the gap between 'filtering' and 'scheduling' techniques: it allows to perform both filtering and scheduling of the elements at a constant effort. The freely definable error metric allows to include the filtering technique - e.g. visibility culling - in the determinition of the elements' priorities, which influences the time to wait between two consecutive schedulings; the PRR in turn selects the elements upon which the filter is applied.

We will evaluate our algorithm in the aforementioned client-server system; PRR is used to schedule the update messages at a constant effort of *O(k)* per client, where *k* is the number of updates that can be transmitted by the network (and thus have to be selected); the priority of the elements is determined by the visual error, e.g. the position displacement. By applying visibility culling (determining whether an object is visible) when it is selected by PRR, the resulting effort is still *O(k)*. Hence we have an overall effort of *O(k*n)=O(n)* for *n* connected clients, an *output* sensitive behavior which is crucial for scalable environments. Other than client-server oriented virtual environments and typical current online games (such as Ultima Online, Everquest, Asheron's Call, Half-Life, Quake III Arena, Unreal Tournament etc.), PRR scheduling is also applicable to peer-to-peer systems (with the known difficulties of emplyong filtering techniques in serverless systems).

2 Related Work

Most virtual environments employ strategies to deal with the network bandwidth restrictions that limit the number of possible update messages. Many of them are filtering techniques which reduce the number of elements competing for the resource. They can roughly be categorized into the following groups:

Several systems exploit the fact that a client is typically interested only in a small subsection – an *area of interest* - of the virtual world, which has local scope. Often the clients perception is limited to what can be seen from the current viewpoint. Objects that are occluded or too far away are not considered. Updates can be propagated on a “need to know” basis, greatly reducing the amount of messages that must be considered. The regions for which communication locality is exploited can either be given by the application designer, such as in SPLINE [Barr96], based on a regular (e.g. hexagonal) subdivision such as in NPSNET-IV [Mace95], by the viewing frustum/view cone such as in AVIARY [Snow94], or by *visibility culling* [Funk95, Makb99]. Visibility culling is often carried out with potentially visible sets (PVS) first introduced by Airey [Aire90]: An environment is first decomposed into cells, for which inter-cell visibility is pre-computed and used at runtime to identify visible objects for a given viewpoint. A simple PVS algorithm [Schm96] was also used for our test system.

A related concept is that of temporal bounding volumes (TBV) [Suda96, Suda97] and update free regions (UFR) [Makb99]. A TBV is a region of space which completely contains an object for a determined period of time. For an object in a completely hidden TBV, no update must be considered during the validity interval of the TBV. UFR implement a similar concept for mutual visibility of objects. In this paper, we construct and use TBVs to enhance message scheduling.

Communication filtering can also be performed based on proximity such as in DIVE [Benf93], or by explicitely registering interest in particular objects or events such as exemplified by NPSNET-IV or AVIARY.

A scalable distributed virtual environment requires also some care in the choice of network topology, which is often considered together with a message filtering mechanism. Several systems use multicasting instead of or together with client-server schemes to achieve better scalability. Multicast groups are often associated with a particular location or message type for implicit message filtering by multicast subscriptions. Examples for such methods can be found in NPSNET-IV, DIVE, SPLINE, and RING [Funk96]. It should be noted that although it was tested in a client-server environment, the scheduling algorithm presented in this paper is independent of network topology and can be used in any network setup.

Finally, *dead reckoning* is a networking enhancement technique used in physically based simulations where the motion of the objects is computed from linear velocity vectors. Each host stores a local copy of a remote object and predicts movements of the objects based on the current velocity. An update gets sent only when the difference between actual movement of the object at the remote host and the local copy exceeds a certain treshold. Several forms of dead reckoning have been developed, including prediction based on first-order derivatives such as in NPSNET [Mace94], position history such as in PARADISE [Sing95], or group dead reckoning such as in NetEffect [Das97].

Although all these techniques may reduce the number of messages to be transmitted by a considerable amount, they usually require a separate examination of all objects for each connected client (e.g. in visibility culling each client can have a different viewpoint), leading to a considerable effort. Furthermore, if the number of remaining messages still exceeds the network bandwidth, they must be scheduled or sorted in some way.

The scheduling algorithm presented in this paper fills in this gap. While the factors used in the algorithm are limited to a few (visual error, visibility), its freely definable metric makes it principially suitable to accommodate any of the above filtering techniques, and work together with other networking techniques.

3 The Priority Round-Robin Algorithm

3.1 Overview

The inspiration of the Priority Round-Robin (PRR) algorithm can be found in the short-term process scheduling known from operating system's research, where a set of independent processes is given processor time in order to optimize determined system's parameters [Deit90, Silb88, Stal95, Tane92]. Two of the most widely used algorithms are *Round-Robin* (or *First Come-First Served*, which is the non-preemptive version of Round-Robin), and the *Multilevel Feedback Queue* (MLFQ). Round-Robin (RR) is widely used due to its simplicity, output sensitivity and starvation-free performance, but prevents the use of priorities. The MLFQ does enforce priorities (it consists of a set of levels with decreasing priorities), but has either to deal with the risk of starvation, or must constantly monitor all processes and thus renounce to a constant overhead.

The scheduling of processes in operating systems and the scheduling of objects in virtual environments bears some substantial differences: in virtual environments for example – as opposed to process scheduling - the objects usually need be scheduled repeatedly, and their high number prevents an examination or sorting of all objects (for more details refer [Fais00]). However, by combining the basic properties of RR and MLFQ, the PRR algorithm inherits the advantages of both, providing an output sensitive and starvation free performance, and being able to enforce priorities. It is therefore a valid replacement for RR in most circumstances. We will employ PRR in our client-server testbed to schedule position update messages, a task which is usually handled by a simple RR queue.

The priority management of PRR is based on the assumption that if an object is not granted the resource requested, it accumulates error, e.g. visual error. To be useful for scheduling, this error must be modeled as an appropriate error metric (such as deviation in position); the goal of the PRR algorithm is thus to *minimize the cumulative error* over all objects in the environment, called the 'overall error'.

Each object in the algorithm is assigned a so called *'Error Per Unit' (EPU)*, which is a prediction of how much the error will increase in a determined time unit[1]. If the error is a deviation in position, then the velocity of an object is a suitable EPU.

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 EPU must be scheduled more often than elements with lower EPU. 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 wait between two consecutive schedulings the r*epetition count (rc)*; it is a measure of the time an element has to wait between two selections and thus for the cumulative error generated by the element until the next scheduling. All elements in level *i* have the same repetition count rc_i, which determines the scheduling frequency and hence the priority of a level.

Let *lev* denote the number of levels and ne_i denote the number of elements in level *i*. If we repeatedly take one element from each level (we traverse all levels at an equal speed of one), the repetition count is simply

$$rc_i = ne_i * lev \qquad (1)$$

In the example shown in Figure 1, the elements of the first level (A and B) must wait 6 times between two consecutive schedulings, those of the second level have a repetition count of 12, and element G is scheduled every 3 elements.

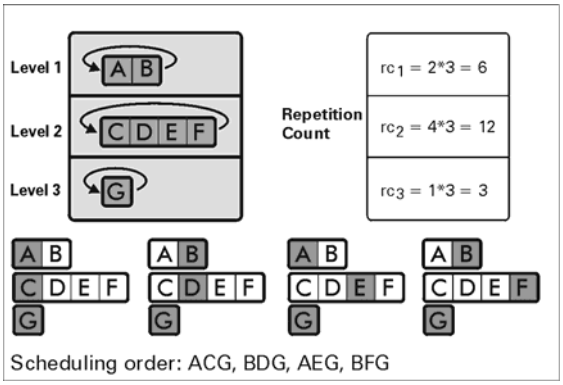

Figure 1: Scheduling order of the elements if all levels are traversed at an equal speed of one.

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 the 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 thel*Level frequency* lf_i of level *i* as

$$lf_i = \frac{ne_m}{ne_i} \qquad (2)$$

[1] The time unit chosen for the Error Per Unit does not affect the performance of the algorithm.

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

$$tt = ne_m * lev \quad (1)$$

If the EPU of an element can be assumed to be constant (such as for entities travelling at constant speed), a predicted error *pe* for that element can be calculated from

$$pe = epu * rc \quad (2)$$

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

3.2 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 level 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 a 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 the 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.

3.3 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 algorithm then selects that level which leads to the 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 EPU.
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 moving 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.

3.4 Optimum traversal rate

In Section 3.1 we have introduced a scheduling strategy that consists of repeatedly picking one element from each level. In this case the average contribution of an element to the overall error is determined by the number of elements in each level, other that the EPU of the element (Equations 1 and 2).

But by assigning an element to a level according to the EPU of both element and level, the length of the level is fixed by the error distribution of the elements. This retains from minimizing the overall error produced by the elements by determining an appropriate repetition count for each level. This would in turn require all levels to have a given length (in order to achieve a determined repetition count, if all levels are traversed at an equal speed of one).

Therefore we need to detemine for each level a different 'speed' with which it is traversed (calculated from the error generated by the level), rather than constantly picking one element from each level. This *traversal rate* tr_i describes for each level *i* the number of elements that are selected from that level each time it is visited (all levels are accessed in turn, as in Figure 1). This makes the repetition count not depend only on the number of elements in each level (other than the number of levels), but allows it to be varied by modifying the tr_i.

Using a determined traversal rate tr_i, the repetition count rc_i for level *i* is now given by

$$rc_i = \frac{ne_i}{tr_i} * \sum_{k=1}^{lev} tr_k \qquad (4)$$

where ne_i is the number of elements in a level and *lev* the total number of levels. If av_i is the average[2] EPU of level *i*, we can furthermore calculate a *predicted error* pl_i for level *i*.

$$pl_i = ne_i * av_i * rc_i$$

$$pl_i = \frac{ne_i^2 * av_i}{tr_i} * \sum_{k=1}^{lev} tr_k \qquad (5)$$

By summing up the errors predicted for each level, we can derive a formula for the *overall error* (*err*).

$$err = \sum_{i=1}^{lev} pl_i$$

$$err = \sum_{i=1}^{lev} \frac{ne_i^2 * av_i}{tr_i} * \sum_{i=1}^{lev} tr_i \qquad (6)$$

Our goal is to minimize the overall error *err* by selecting the optimum traversal rate tr_i for each level *i*. Hence we can build a cost-function *err* to minimize, with tr_i being the variables of the function. The number of elements ne_i and the average error av_i of each level *i* can be treated as constant; hence we can ignore the sum of the tr_i in Equation 6, and use su_i to substitute for

$$su_i = ne_i^2 * av_i \qquad (7)$$

This allows us to construct the following cost-function *err* from Equation 6:

$$err(tr_1,...,tr_{lev}) = \sum_{i=1}^{lev} su_i * \frac{1}{tr_i} \qquad (8)$$

This optimization problem is best solved with the help of Lagrange Multipliers: Equation 9 allows us to find the extrema of function *f*, with *g* being a constraint function for the variables of *f*:

$$grad\ \mathrm{f} = \lambda * grad\ \mathrm{g} \qquad (9)$$

Hence we use *err* as function to minimize (substitute for *f*) and introduce a constraint function *cons* (Equation 10) given by the sum of all traversal rates, which is assumed to be equal to one[3].

$$cons = \sum_{i=1}^{lev} tr_i = 1 \qquad (10)$$

Substituting for function *f* and *g* in Equation 9 yields Equation 11, which allows us to find the values for the variables tr_i where *err* is a minimum:

$$grad(\sum_{i=1}^{lev} su_i * \frac{1}{tr_i}) = \lambda * grad(\sum_{i=1}^{lev} tr_i) \qquad (11)$$

To do so, we have to build for all variables tr_i the partial derivatives F_{tr_i}

$$F_{tr_i} : \frac{\partial(\sum_{k=1}^{lev} su_k * \frac{1}{tr_k})}{\partial tr_i} = \lambda * \frac{\partial(\sum_{k=1}^{lev} tr_k)}{\partial tr_i} \qquad (12)$$

Solving the partial derivatives F_{tr_i} we get

$$F_{tr_i} : -\frac{su_i}{tr_i^2} = \lambda * 1$$

and from this we can solve for tr_i:

$$tr_i = -\frac{\sqrt{su_i}}{\sqrt{\lambda}} \qquad (13)$$

Now, by plugging Equation 13 into the constraint function (Equation 10, Equation 14)

$$tr_1 + ... + tr_{lev} = 1 \qquad (14)$$

we can solve for λ

$$\sqrt{\lambda} = \sum_{i=1}^{lev} \sqrt{-su_i}$$

After substituting for $\sqrt{\lambda}$ into Equation 13 we get a formula for tr_i:

$$tr_i = \frac{\sqrt{su_i}}{\sum_{k=1}^{lev} \sqrt{su_k}} \qquad (15)$$

Resolving for su_i (Equation 7) we finally get the optimum value for the traversal rate tr_i, in order to minimze the overall error *err*.

$$tr_i = \frac{\sqrt{ne_i^2 * av_i}}{\sum_{k=1}^{lev} \sqrt{ne_k^2 * av_k}} \qquad (16)$$

[2] As an element is assigned to a level according to its EPU, the average EPU of a level is the moving average of the elements' EPU contained in that level. Simplifying, we can assume that all elements in a level have the same (average) EPU.

[3] The value of one is arbitrary and only chosen for convenience

The main loop of the PRR algorithm hence consists in simultaneously traversing all levels according to their 'speed' tr_i. Every time an object is selected, it is granted the resource requested (e.g. transmitting a position update), after which the object is re-evaluated: first a new EPU is determined (we base it on the actual velocity), then the object is reassigned to one of the levels according to its EPU. Assigning the object to the level whose average EPU is most close to the EPU of the object yields an simple yet effective adaptation to even rapidly changing error distributions. Afterwards the traversal rate of the levels is modified so to account for the new error distribution.

By assuming a fixed number of levels, the effort needed to schedule an object is constant; hence the PRR algorithm can achieve an output-sensitive behavior. The freely definable EPU allows us to include visibility information in the determination of an object's priority.

4 Using visibility information

4.1 Overview

Visibility information is already available in many existing virtual environments and networked games, usually employed to limit the amount of data transmitted over the network. In indoor scenes, rooms and building occlude most parts of the environment; in outdoor scenes the visibility is often limited by a radius around the user, e.g. the so called 'fog of war' in strategy games.

Visibility culling of objects in a virtual environment can be accomplished by first determining the visible area that can be seen from the viewpoint, and then checking which objects are inside and outside that area. Figure 2 depicts the visible area for a client, with object A and B being visible, and object C being invisible.

Usually visibility culling is first used to reduce the number of objects, then a plain FIFO or Round-Robin (RR) queue is used to schedule the remaining objects; hence the visibility information is employed to insert or remove objects from the queue.

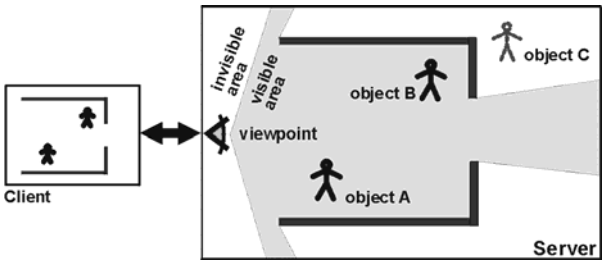

Figure 2: Visible area and visible objects for a given viewpoint of the client.

In contrast, we replace RR with a Priority Round-Robin (PRR) scheduler and include visibility information in the priority of the objects. This allows us to reduce the effort for the server to determine which updates should be sent to each client. As each client has its own field of view, the server must usually examine all objects for each client. Assuming the number of clients approaching the number of objects, it is an effort of $O(n^2)$.

By employing the PRR algorithm it is possible to shift part of this effort the scheduler; we let PRR repeatedly schedule as many objects (k) as the network permits, achieving an overall effort of $O(k*n)=O(n)$ for n connected clients. Whenever an object is selected, PRR checks whether it is visible or not. For a visible object the update is transmitted, otherwise the algorithm continues its selection, looking for visible objects, with the highest speed permitted by the computing power and the network bandwidth. The visibility information affects how the objects' priority is determined: visible objects get a priority equal to their velocity (their EPU); if an object is invisible, the priority is chosen such as to let the object be rescheduled when it is expected to become visible again. In our implementation we base the prediction of when an object will be visible again on the shortest path from the actual position to the next visible area (and the actual velocity of the object).

4.2 Temporal bounding volumes

The determination of the time interval an object is supposed to remain invisible is based on a technique called 'temporal bounding volumes' (TBV). A TBV is a region of space (for simplicity often a circle or sphere) which completely contains an object for a specific period of time (called the validity interval). The TBV becomes invalid if the object leaves the volume. Hence its 'expiration date' is determined by the movement of the object (e.g. rotating around a fixed point, traveling along a track, or translating freely in space) and by the size the TBV can have. In the extreme, a TBV encompassing the whole area of movement of the object will always be valid.

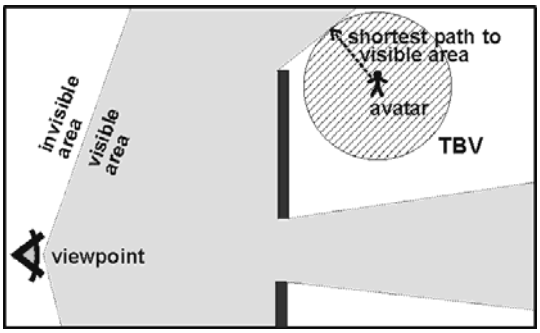

Figure 3: Temporal bounding volume for an invisible object based on the shortest path to the next visible area.

For objects with unconstrained translational movement, the expiration date of the TBV is directly related to its size. The validity interval of a TBV could be calculated by dividing the size of the TBV by the velocity of the object. However, in large virtual environments the entities are usually avatars with an unpredictable behavior.

Our application of the TBV consists in using them to determine the priority of objects in the PRR algorithm: every time an object is scheduled, PRR determines whether it is visible or not. In the latter case, a TBV is constructed, based on the time the object is supposed to become visible again (thus, the size of the TBV determines its validity interval). Given the fact that the scheduling frequency of an object is reflected by its priority, we assign the object a priority such as to become scheduled again at the same moment the TBV expires (and the object is supposed to become visible again); this provides kind of an automated 'wake-up' function. Figure 3 shows the TBV for an object with unbound translation, calculated from the shortest path to the next visible area (hatched area).

4.3 Integrating visibility information in PRR

In order to be usable by the PRR algorithm, we express the time interval an object has to wait (given by the TBV) in number of scheduling actions[4]. Hence we can directly compare the waiting time of an object - given by a number of scheduling actions - to the scheduling frequency of each level (given by the repetition count, as calculated using Equation 1). An object is then assigned to that level whose scheduling frequency best matches its required waiting time.

This causes a difference in how an object is assigned to a level, depending whether it is visible or not: if an object is visible, it is assigned to that level whose average EPU best matches the EPU of the object (given by its velocity). If it is invisible, that level is chosen whose scheduling frequency best matches the waiting time determined by the TBV. In the latter case, the EPU of the object is not determined by its velocity; rather it temporarily assumes the average EPU of the assigned level. This allows the PRR algorithm to simultaneously process visible and invisible objects.

5 Testbed implementation

Our testbed consists of a server which moves a determined number of objects through an environment, generated from a floor plan. A client visualizes the whole scene from a determined viewpoint and receives position updates of the various objects, in order to remain consistent with the server. Due to limited network bandwidth, only part of all pending updates can be transmitted to the client.

The visual error, given by the difference in position of the objects on server and client, will be minimized by the enhanced PRR algorithm. While the simulator continuously moves all objects, we let the PRR algorithm select as many objects as the network permits (e.g. 10 %), using the velocity of the objects as Error Per Unit (EPU). As reference we use the same setup, but we use Round-Robin (RR) instead of the PRR algorithm.

In the first part of the evaluation we will run the testbed without visibility culling (all updates are transmitted to the clients), focussing on the scheduling capabilites of PRR. The second part includes visibility culling as filtering technique, where both PRR and RR send only updates of the visible objects to the clients (but RR does not enforce priorities).

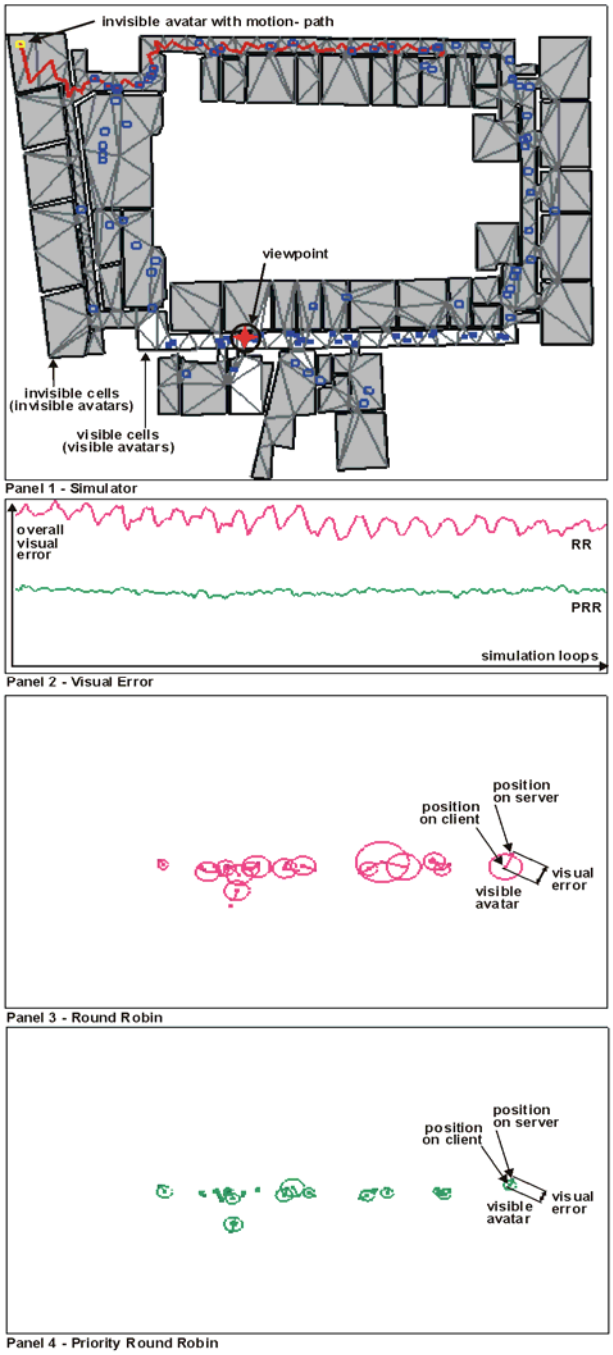

Figure 4a-d: Screenshots of the testbed employed to evaluate the enhanced PRR algorithm.

[4] This value depends on the number of objects the PRR can schedule per unit time.

Figure 4 shows snapshots of the evaluation testbed when visibility culling is used: the server (simulator) in Panel 4a depicts the floor plan and the moving objects; the client's viewpoint is depicted by a star, and the invisible areas are shown shaded dark. The Panels 4c and 4d show the visual error of the connected client; Panel 4d depicts the visual error for RR, while Panel 4c shows the visual error for the enhanced PRR. Please note that only the visible objects are shown in Panels 4c and 4d. The visual error is depicted by a circle; its center coincides with the last updated position of an object on the client, while the actual position on the server determines the radius of the circle (both positions are also connected with a line). Thus the bigger the radius, the higher is the visual error. The graph in Panel 4b permits to monitor the overall error for RR and PRR scheduling.

The motion of the objects through the environment was implemented by first digitizing and triangulating a floor plan, and then generating a connection-graph of the triangles. The simulator generates for each object a path from the current position to a random destination position, and then moves the object along this path with a given velocity (used as EPU).

The area visible from the viewpoint chosen by the client is also easily computed with the help of the triangulated floor plan. Starting from the triangle which contains the viewpoint, the 2.5D-visibility algorithm presented by Schmalstieg in [Schm96] generates the set of potentially visible triangles from the viewpoint.

If the object is invisible, then the algorithm determines the shortest path from the actual to the nearest visible triangle. From the length of the path and the object's velocity the PRR makes a safe guess of the moment the object will become visible in the worst case (if it immediately starts heading for the visible area), and gives the object an according priority.

6 Evaluation and results

The enhanced PRR algorithm is evaluated by comparing it to plain Round-Robin (RR) scheduling; this allows us to evaluate the performance increase that can be gained by substituting RR with PRR.

In the examples given below, the server hosts a simulator and moves 10000 objects (simulating avatars) at a predetermined velocity through the environment; the velocity is used as Error Per Unit (EPU). To simulate the network bottleneck, although the simulator can move all objects in every simulation step, only 10% of the position updates (1000 in numbers) can be transmitted to the client. The main loop of the testbed consists thus in first simulating all 10000 objects; afterwards a PRR-scheduler, as well as a plain RR queue can select and update 1000 objects. The actual overall error is computed and evaluated for both RR and PRR scheduling after each loop. Examples 1 and 2 (Section 6.1 and 6.2) compare PRR to RR without employing any filtering technique, Examples 3 and 4 (Section 6.3) include visibility culling; the use of dead reckoning is evaluated in Section 6.4.

6.1 Example 1: clustered error distribution

This example shows a case apt for the PRR algorithm, as we have three different clusters of EPUs which can be serviced by PRR at different priorities. We let 10000 avatars walk along random paths in the environment at different velocities (used as EPU):

- 500 avatars get a velocity between 9 and 10 units
- 2000 avatars get a velocity between 3 and 4 units
- 7500 avatars get a velocity between 0.1 and 0.5 units

Figure 5 shows a comparison of the overall visual error caused by a RR and a PRR algorithm (for the same client), if only 10% of the 10000 objects are scheduled after each simulation loop.

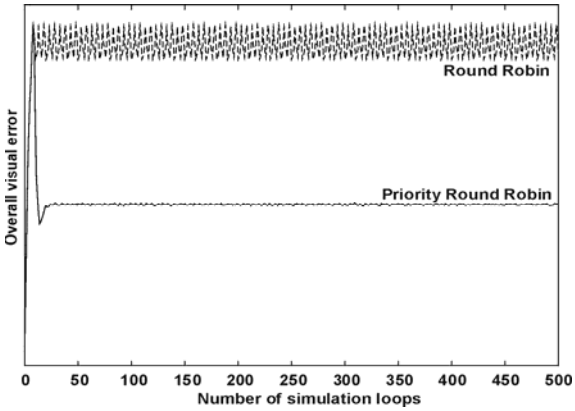

Figure 5: Due to the clustered error distribution, the visual error of the enhanced PRR is 72% lower compared to RR.

6.2 Example 2: uniform error distribution

As PRR relies on servicing the objects at different priorities, according to their EPU (velocity), a uniform error distribution prevents PRR from constructing clearly distinct error groups. In contrast to the previous example, each avatar gets a random velocity between 1 and 10 units. Hence the reduction of the overall visual error, as compared to RR scheduling, is 'only' 10%.

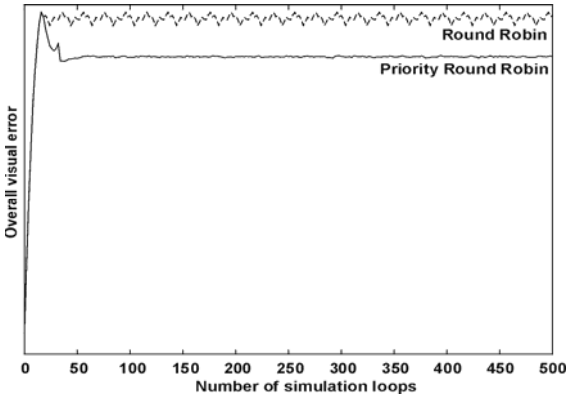

Figure 6: With a uniform error distribution, the visual error of the enhanced PRR is only 10% lower compared to RR.

6.3 Examples 3 and 4: including visibility culling

In this example we combine Priority Round-Robin with visibility culling, as described in Section 4. Both PRR and RR send only updates of avatars which are visible, as only the visible avatars contribute to the actual visual error. The position of the camera was set as shown in Figure 4a. For the avatars we use the same velocities as in Example 1 and 2. With the uniform error distribution (as in Example 2), the use of visibility culling lets PRR outperform plain RR by 92%.

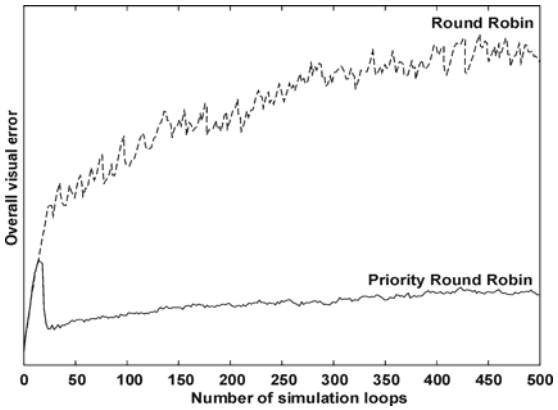

Figure 7: The combined use of visibilty culling and PRR on a uniform error distribution decreases the visual error by 92%.

With the clustered error distribution (as in Example 1), the exploitation of the visibility information allows PRR to achieve an even more substantial decrease in the visual error, outperforming RR by 307%.

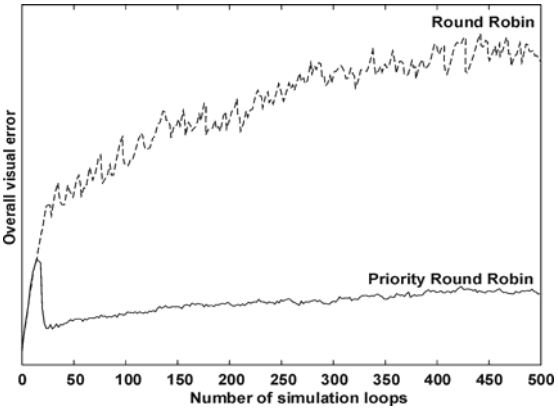

Figure 8: The use of visibility culling lets PRR outperform RR scheduling by 307%.

6.4 Example 5: including dead reckoning

Dead reckoning is often used as filtering technique in physically based simulations where the motion of objects is computed from linear velocity vectors. We compare the avatars as selected by PRR and RR to the threshold set for dead reckoning, and transmit the update only if the visual error exceeds the threshold. Thus we use the Priority Round-Robin technique to optimize the order in which the objects are examined for transmission. The priority of the elements in PRR is determined by the difference between simulated and approximated position, as compared to the threshold by dead reckoning. We interpret this difference as distance, and from its changes on each simulation step we compute an average velocity used as Error Per Unit for the avatars.

We move all 10000 avatars along random paths through the floorplan, with a given velocity:

- 1000 avatars get a velocity between 9 and 10 units
- 2000 avatars get a velocity between 2 and 3 units
- 7000 avatars get a velocity between 0.1 and 0.5 units

Both PRR and RR can schedule 10% of the 10000 avatars on each simulation loop. However, the motion through the long corridors is mostly coherent to the extent that all avatars which exceed the threshold can also be transmitted (which makes scheduling not necessary at all). Hence we let all avatars change their path every 10 simulation steps. Using e.g. a threshold distance of 5 makes dead reckoning select on average 50% of the avatars for update (which in absolute numbers is approximately 5000), compared to the 1000 objects that can at most be updated.

In this situation, by enhancing dead reckoning with PRR allows to lower the overall visual by about 42%[5], compared to employing dead reckoning with simple round-robin.

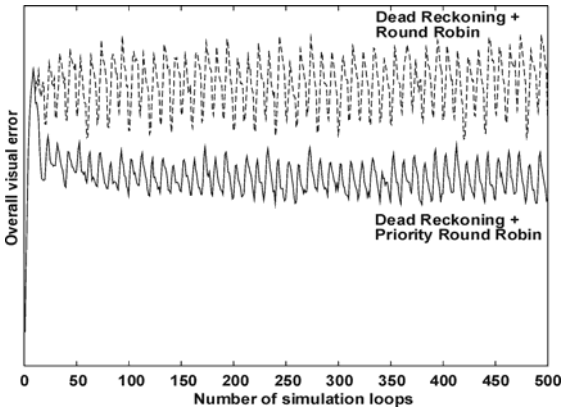

Figure 9: The overall error is decreased by 42% if dead reckoning is enhanced with PRR.

7 Conclusions and future work

We have presented a technique to enhance plain Round Robin scheduling, by adding the enforcement of priorities to its advantages of being output sensitive and immune to starvation. The Priority Round-Robin (PRR) algorithm can bring a substantial contribution to the development of distributed virtual environments or networked online-games which contain a very high number of objects. The simplicity of PRR and the freely definable error metric

[5] All percentages refer to the error caused by PRR.

make it a suitable substitution for Round-Robin in most cases. In our examples we have employed PRR as substitute for the plain Round-Robin queue to transmit the update messages at a constant effort per connected client; the frequency of the updates is determined from priorites based on the behavior of the objects.

Furthermore, PRR can be efficiently combined with filtering techniques such as visibility culling. By including the visibility information in the determinition of the objects' priorities, we do not abondon output sensitivity. Hence PRR not only provides a scalable technique that leads to a graceful degradation of the system's performance caused by network bandwidth limitations. But is also helps avoid computational bottlenecks caused by a naive application of filtering techniques.

In order to furthermore optimize the PRR algorithm for distributed environments and online-games, future work will examine the scheduling of avatars (build according to a hierarchical human model) and employ Levels of Detail for the accuracy of the updates transmitted to the clients. To account for the highly unpredictable behavior of user-controlled avatars, a measure to determine the rate at which the objects change their activity is investigated. It should help to specify at which threshold the prediction becomes useless: for example, if an object has a moderate movement, it will be inserted in a 'slow' level with a low traversal speed. But if the activity of the object suddenly starts to increase rapidly, its slow traversal speed may cause the element to react too slow. Concluding, it is planned to construct an extended environment containing a large number of rooms, buildings and open landscapes, and evaluate perceptual error metrics to minimize the visual error as perceived by the user.

Acknowledgements

The work presented in this paper was sponsored by the European Community under contract no. FMRX-CT-96-0036.

References

[Aire90] J. M. Airey, J. H. Rohlf, F. Brooks Jr.: Towards Image Realism with Interactive Update Rates in Complex Virtual Building Enviroments. Computer Graphics, 24(2):41, 1990.

[Barr96] Barrus, J., Waters, R., & Anderson, R.: Locales and Beacons: Precise and Efficient Support for Large Multi-User Virtual Environments. Proceedings of VRAIS'96, pp. 204-213, Santa Clara CA, 1996.

[Benf93] S. Benford, L. Fahlen: A spatial model of interaction in large-scale virtual environments. 3rd European Conference on CSCW, pp. 109-124, 1993.

[Das97] T. Das, G. Singh, A. Mitchell, P. Kumar, K. McGhee: NetEffect: A Network Architecture for Large-scale Multiuser Virtual World. Proc. of ACM VRST'97, pp. 157-163, 1997.

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

Fais00] C. Faisstnauer, D. Schmalstieg, W. Purgathofer. Priority Round-Robin Scheduling for Very Large Virtual Enviroments. Proc. of IEEE VR'2000, pp. 135-142, 2000.

[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.

[Funk95] T. A. Funkhouser. RING - A Client-Server System for Multi-User Virtual Environments. SIGGRAPH Symposium on Interactive 3D Graphics, pp. 85-92, 1995.

[Funk96] T. Funkhouser: Network Topologies for Scaleable Multi-User Virtual Environments. Proceedings of VRAIS'96, pp. 222-229, Santa Clara CA, 1996.

[Mace94] M. R. Macedonia et al. NPSNET: A Network Software Architecture for Large-Scale Virtual Environments. Presence, Vol 3(4), pp. 265-287, 1994.

[Mace95]M. Macedonia, M. Zyda, D. Pratt, D. Brutzman, P. Barham: Exploiting Reality with Multicast Groups. IEEE Computer Graphics and Applications 15(3), pp. 38-45, 1995.

[Makb99] Y. Makbily, C. Gotsman, R. Bar-Yehuda: Geometric Algorithms for Message Filtering in Decentralized Virtual Environments. SIGGRAPH 1999 Symposium on Interactive 3D Graphics, pp. 39-46, Altanta GA, April 1999.

[Schm96] D. Schmalstieg et al.: Demand-Driven Geometry Transmission for Distributed Virtual Environments. Proceedings EUROGRAPHICS '96, 15(3), 421-433.

[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), pp. 169-193, 1995.

[Snow94] D. N. Snowdon, A. J. West. AVIARY: Design Issues for Future Large-Scale Virtual Environments. Presence, Vol 3(4), pp. 288-308, 1994.

[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), pp. 249-258, 1996.

[Suda97] O. Sudarsky , C.Gotsman: Output-Sensitive Rendering and Communication in Dynamic Virtual Environments. Proc. of ACM VRST'97, Switzerland, 1997.

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