BROWN UNIVERSITY
Department of Computer Science
Master's Project

CS-96-M11

"Scheduling Time-Critical Graphics on Multiple Processors for Virtual Environments"

by

Thomas Meyer

Scheduling Time-Critical Graphics on Multiple Processors for Virtual Environments

Thomas Meyer

Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science at Brown University

May 1996

5/20/96

Professor Andries van Dam
Advisor

Scheduling Time-Critical Graphics on Multiple Processors for Virtual Environments

Tom Meyer

May 17, 1996

Chapter 1

Introduction

This paper describes an algorithm for the scheduling of time-critical computation and (possibly dependent) rendering tasks on single- and multiple-processor architectures, with minimal pipelining delay. The target application of this algorithm is to manage virtual environment-based scientific visualization scenes consisting of hundreds of objects, each of which can be computed and displayed at large numbers of resolution levels. The algorithm progressively refines the time-critical schedule; it always returns a feasible schedule and, when it runs to completion, produces a good schedule, taking advantage of much of the multiple-processor system (and using all of a single-processor system, when that is possible).

Complex datasets (e.g., large time-varying fluid-dynamics simulations, high-resolution MRI scans, and structural simulations) require sophisticated interaction and visualization techniques. For example, a scientist may want to interactively manipulate and examine complex visualizations, such as a probe with dozens or hundreds of emitted streamlines in a time-varying flow. For such complex scenes, maintaining a fast interaction rate can be quite difficult, especially in immersive, computationally demanding environments such as the Virtual Wind Tunnel at NASA Ames [BL91].

Two of the requirements for achieving the feeling of immersion in a virtual environment are to maintain a constant, fast frame rate, and to reduce end-to-end lag time from a user action to the simulated response. Experiments with virtual environment systems have shown that it is necessary to attain a frame rate of at least 8-10 frames per second,

and preferably much faster [DM95]. The manipulation of objects in the simulated environment also becomes difficult when there is more than 100 milliseconds of delay between the manipulation and the system's display of a response to that change.

Graphical scenes with highly demanding computations may have a reduced frame rate if there is too much graphical information to be rendered in a single frame, or if the computations require more than one frame to complete. Deep pipelining of these operations, though it increases frame rates, also increases the lag inherent in the graphics system.

One technique for maintaining constant frame rates with increasing graphical complexity is to simplify portions of the scene which are not as perceptually important to the viewer. Objects which are far away from the viewer or not at the center of attention could be simplified or computed less frequently, and objects which are not visible at all can be completely ignored. By not computing objects which we will not render, we can also lower the amount of computational demand on the system, and thus reduce the frame time. Deciding which objects to show and in what order to compute and render them involves dynamically scheduling them, since the number of objects in an interesting world is unlikely to be fixed (and the user may even be able to dynamically add large numbers of new objects).

In order to support the task of exploratory visualization in these complex datasets, we develop a time-aware scheduling algorithm to provide importance-based time-critical computation and rendering of a common scientific-visualization technique (streamlines in a vector field), as shown in Figure 1.1. This algorithm takes advantage of a dedicated graphics workstation with a single-threaded graphics pipeline and from one to several dozen processors, communicating using a shared-memory model.

The techniques described in this paper, though developed to support scientific visualization tasks, can be easily extended to general graphics scheduling. Many objects with a large number of possible representations – for example, curves and surfaces can be computed at nearly arbitrary intervals – are particularly well-suited to be scheduled using this algorithm.

This scheduling algorithm has the following advantages:

- After an initial startup phase, it can terminate at any time during its incremental refinement phase, and will always return a feasible schedule. This allows the scheduling algorithm itself to be time-critical, so that adding more tasks to be scheduled will not result in increased scheduling time. (Such algorithms are also known as

"anytime algorithms.")

- It results in the good usage of single- and multiple-processor machines if it runs to the completion of the refinement phase.
- It schedules all computations using very fine-grain pipelining, reducing lag times by rendering objects in the order computed.
- It balances the benefit of spending time computing new data against the time required to redisplay existing data at its already-computed resolution.

Its disadvantages are as follows:

- Running a scheduler can reduce the amount of time available for computation and rendering, and could severely reduce the performance of some systems (e.g., if the demands on the system are demanding but fairly static, it might be better to optimize for the worst case, and not use a scheduler at all).
- It is optimized for managing computation/rendering tradeoffs for relatively smooth functions on a multiple-processor machine. A simpler algorithm may be much faster in a more restricted problem domain, such as a single-processor static architectural walk-through.

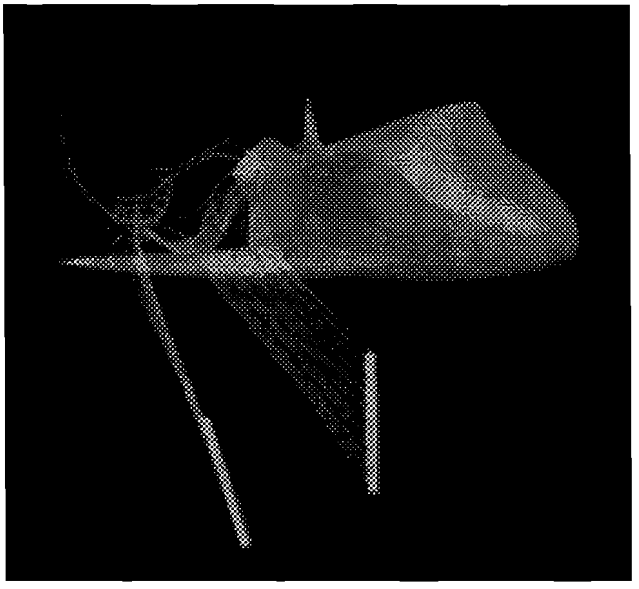

Figure 1.1: The target application, scientific visualization of complex computational fluid dynamics scenes in an immersive environment. This picture shows a visualization of a precomputed dataset of air flow past the descending Space Shuttle. There are two "rakes" (the cylinders) visible in the scene, each of which has several "streamlines" (the lines which follow the flow of air through the dataset) attached to it.

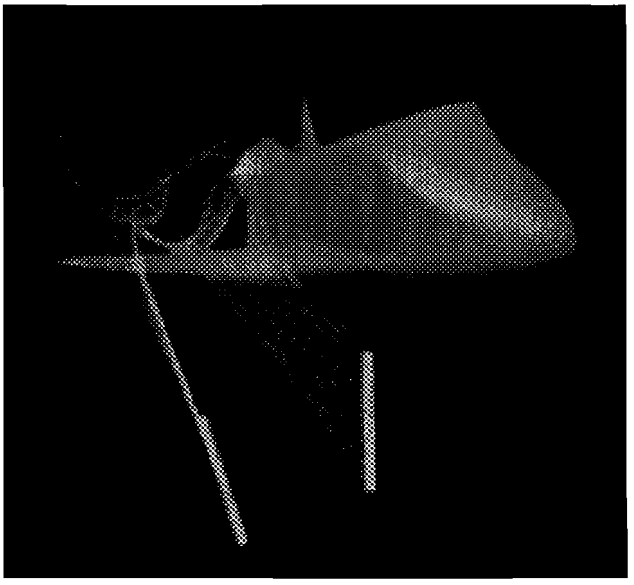

Figure 1.1: The target application, scientific visualization of complex computational fluid dynamics scenes in an immersive environment. This picture shows a visualization of a precomputed dataset of air flow past the descending Space Shuttle. There are two "rakes" (the cylinders) visible in the scene, each of which has several "streamlines" (the lines which follow the flow of air through the dataset) attached to it.

Chapter 2

Previous Work

A large body of research on real-time scheduling exists, dating from the 1950's. A good introduction to the relevant issues is [SSNB95].

Classical real-time theories mainly deal with static scheduling problems in which the algorithm has complete knowledge of the demands placed on it, and where there are generally hard constraints which, if violated, could result in catastrophic failure (airplanes crashing, factories blowing up, etc). These types of problems, known as "hard real-time", are fairly well understood, and many algorithms exist for solving them.

Although dynamic multiple-processor scheduling is becoming an active area, little prior work has been done on it. Almost all multiprocessing scheduling is NP-complete, and good approximation algorithms are only beginning to emerge [SSNB95]. One interesting result is that, in many cases, no dynamic scheduling algorithm for multiprocessors can be optimal [Mok83]. There is wide variety of possible heuristics used to generate schedules for real-time tasks; some are described in [RSZ89].

Soft real-time scheduling, also known as "imprecise computation" scheduling, addresses the scheduling of problems where there may be some fixed startup time, but where incremental results are stored as the computation progresses. Here, the deadlines are not as strict as they are for hard real-time, since there is some benefit from performing a partial computation (it may be less accurate, or based on out-of-date knowledge, but it is still useful). Since scientific-visualization graphical objects may be computed at many levels of detail, and can also be computed incrementally, we

will be trying to optimize a very similar problem. The paper [LLS+91] describes various scheduling techniques for such problems.

Another important part of the time-critical scheduling process is to balance the time spent planning against the amount of spent executing the schedule. This has become an important area for artificial intelligence research, as described in [GD95].

Unfortunately, problems of particular interest to scientific visualization have not been much studied. The narrowness of the problem – scheduling multiple independent pairs of tasks (computation and rendering) with the two requirements that the "rendering" portions all take place in a single thread (since modern graphics pipelines are not commonly multithread-safe), and that each rendering portion start only after the completion of its computation portion – makes it too specialized to warrant much attention as a general systems problem.

Because all virtual environments require near-constant, high frame rates, several systems which address time-critical issues, particularly in static or walk-through environments, have been developed:

Richard Holloway's Viper system [Hol92] uses objects with predefined levels of resolution, and renders objects at a global level of resolution sufficient to display all of them in the allotted time. It does not provide for individual levels of importance for the objects.

Funkhouser and Sequin describe a real-time scheduling algorithm for complex virtual walk-throughs in [FS93]. However, their algorithm does not scale well for objects with many different levels of representation and their faster algorithm only works well on objects with a globally convex-downward benefit function. Nonetheless, the ideas in that paper provide the starting point for our algorithm. Also, their analysis of the relative benefits of rendering various levels of detail is an extremely useful model for describing how to trade-off various rendering elements when creating time-critical models.

The multi-processing scheduling algorithm described by Rohlf and Helman in [RH94] schedules computation, culling and rendering of geometric data by using pipelining, which results in the addition of at least one frame's worth of lag to the system (lag is as bad as low frame rates in virtual environments). They do not model the computational requirements of generating the scene (since it was originally designed for static walk-throughs), but require the application designer to schedule computations appropriately. Additionally, they use a feedback-based

model for managing scene complexity, so cannot bound frame times when the scene changes rapidly.

Little work has been done on combining computation and rendering, and on managing tradeoffs among computing expensive but useful information.

2.1 Novel Aspects of This Research

This research extends the previous work (primarily the Funkhouser-Sequin algorithm) in the following ways:

- We provide a characterization of the general qualities of cost-benefit functions in continuous domains, for tasks which combine computational and rendering costs. This is not intended to quantify the perceptual or semantic benefits of a particular task, but describes a model which can be parameterized to suit experimental results determining such benefits (where such results indicate separable influences on the benefit function).

- We iteratively schedule tasks in the *continuous* domain, on a single processor, and provide bounded estimates of the quality of such a schedule.

- We iteratively schedule tasks on a multiple-processor system. This includes developing a heuristic for good ordering of the compute and render task allocations in such a system.

Chapter 3

Benefit Function

For a set of rendering tasks, we need to determine the most useful amount of time to spend computing and rendering each one. We define a function *Benefit*(*time*), which reflects the approximate value of spending an amount of time *time* computing and rendering a graphics task. For any set of tasks, we want to maximize

$$B = \sum_i Benefit_i(time_i)$$

subject to the constraint that

$$\sum_i time_i \leq frametime$$

We base our benefit model on that developed in [FS93]. However, to simplify the analysis, we assume that the various terms of the benefit function are linearly separable (the true benefit function would most likely be extremely subtle, and would probably vary from person to person, depending on the time of day, visual acuity, attention span, and the task at hand).

Our approximation to the actual benefit function consists of a product of several other benefit values, computed on a per-item basis. For any item i, we decompose the benefit of alloting time t rendering that item into a product of three

parts:

$$Benefit_i(time) = Importance_i \cdot Processor_i(time) \cdot Hysteresis_i(time)$$

Importance is a per-frame constant importance value for the item, expressing the object's inherent value, closeness to the viewer, current interactions with the user and the rest of the scene, and the visual focus of the viewer. Any number of perceptually-based metrics could be weighted into this. In the current implementation we assign high importance to streamlines with which the user is interacting, but do not use any additional heuristic; defining useful metrics for determining an object's importance, both in perceptual and semantic terms, is beyond the scope of this paper.

Processor expresses the amount of value to be gained by spending that amount of the system's computation, and rendering time on that object. We assume that this is a nondecreasing function, convex to the right of some "startup time," where there is no initial benefit. The monotonic, convex-downward aspect reflects the idea that for most visualizations, "something is better than nothing, but fine detail is worth only a little more than coarse detail." Such a function is illustrated in Figure 3.1. Without performing detailed user studies, it is difficult to determine what exactly this function should be, except that it fulfill the above constraints. Since such a study is beyond the scope of our research, we use `sqrt` as a suitable function, and a placeholder until more detailed understanding becomes available. Our implementation uses 0 for $time \leq time_s$. where $time_s$ is the initial startup time, and $\sqrt{time - time_s}$ for $T > T_s$. (Many convex downward functions would have the proper behavior, but the square root is relatively fast to calculate and differentiate.)

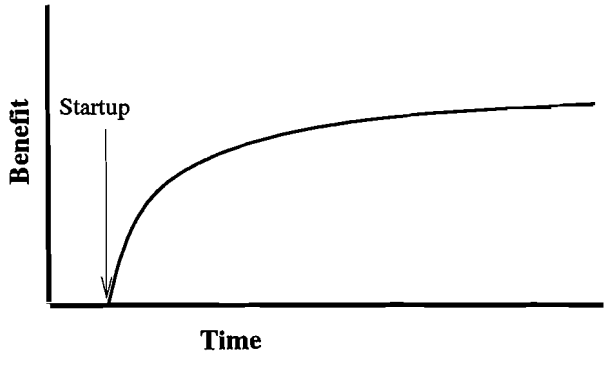

Figure 3.1: This is an example of a suitable function for determining the computational benefit of performing computations. Note that there is a startup cost, followed by a convex-downward monotonic function, as additional computation time results in reduced additional benefit.

The *Hysteresis* term, as shown in Figure 3.2, is a sigmoid function designed to encourage inter-frame continuity. It varies smoothly from a value of 1 at some point $time < T_{\text{prev}}$ up to a value of $1 + \delta$ at T_{prev}, where T_{prev} is the time allocated to the task in the previous frame.

We use a sigmoid function rather than the other obvious choice, a "hat" (gaussian-shaped) function, because our scheduler requires that the benefit function be monotonic. We also do not mind if an object is rendered at a *higher* level of detail than in the previous frame; the situation to avoid is when the object oscillates up and down between levels of detail. Additionally, since all the optimization techniques we describe here operate using the derivatives, the actual optimization takes place using the derivative of the sigmoid function, which is a hat function.

As in the *Processor* term, we have not performed the necessary studies to determine what the hysteresis function should look like. We use a piecewise-cubic function to approximate the sigmoid function, since that is also fast to compute and differentiate.

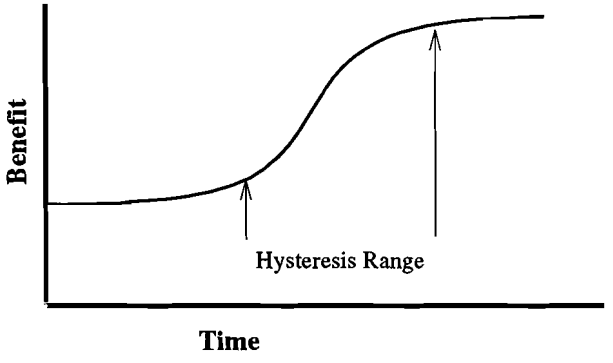

Figure 3.2: The hysteresis function has a constant benefit until reaching the hysteresis range. It then increases smoothly until reaching the point at which the task was scheduled in the previous frame.

The resulting function, the product of these three components, is shown in Figure 3.3.

Since we want to maximize the sum of all the benefits subject to the constraint that the sum of the times scheduled for all tasks is less than the frame time, we first examine the benefit per time unit for the tasks. If this benefit per time $Benefit_i(time)/time$, which Funkhouser and Sequin called the *value*, is increasing at some *time*, we would ideally like to allocate more time to task i: doing so would reduce the average cost of the benefits derived from executing task i. But if the benefit per time is decreasing, then allocating more time to task i will increase the average cost of the benefit, and should be done only if other tasks cannot benefit more from being given the additional time instead. We

therefore consider the points where the value $Benefit_i(time)/time$ is at a maximum as good starting points in the search for optimal time allocations.

As can be seen in Figure 3.3, due to the fact that each of the two functions it is composed of has only one convex region, the benefit/time function will have at most two local maxima – near the extreme points of $Processor_i(time)/time$ and $(Hysteresis_i(time) - 1)/time$. If these functions are expressed analytically, it is simple to compute these two points. As long as both functions are differentiable and have few local maxima, we can perform a similar analysis to obtain the starting points.

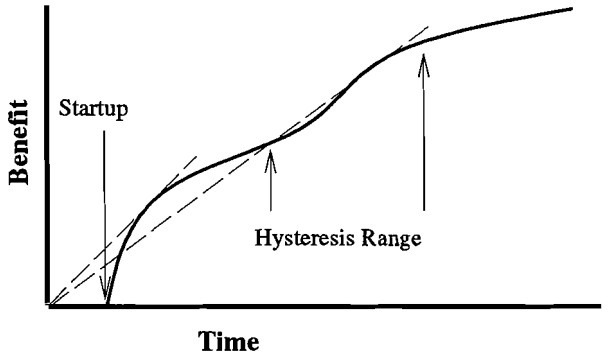

Figure 3.3: The benefit function has a fixed startup cost, rises quickly after that, and gradually falls off until the hysteresis point. The two dashed lines indicated the local maxima of the function $Benefit(time)/time$.

Note that in these discussions of the benefit function we are taking advantage of the fact that our tasks are fine-grain enough that they can be treated like smooth functions. In contrast to our continuous approximations, the actual functions that constitute $Benefit(time)$ are defined only at a fixed, sparse set of values for *time*, since computation time cannot be allocated in quanta smaller than the clock cycle. Furthermore, the functions are likely to be step functions, constant over some intervals in the domain. To the extent that this is true, time spent in making small adjustments to the allocation of processor time could be wasted. On the other hand, by bounding the smallest step size we will take in adjusting processor allocations to be of the same scale as the smallest interval on which the true benefit functions are constant, we can substantially avoid such waste, while still deriving the benefit of being able to use differentiable functions.

Chapter 4

Scheduling Algorithm

The order of events in a typical frame of the system is as follows:

- user input is gathered.
- any pending database updates (such as reactions to user input) take place in single-threaded mode.
- we execute the schedule (computed in the previous frame), which determines when and where tasks are executed.

  at some point during this execution, the schedule of tasks to be performed during the *next* frame is computed.

This section of the paper discusses the scheduling portion of this loop. The other portions of the event loop are not central to this research, so will not be discussed in detail.

On a multiple-processor machine, we compute the schedule in the previous frame, so that the other processors do not have to idle while the schedule is being created. This is not a pipelining of the data, but a pipelining of the scheduling process. We may lose some reactivity if, for example, the schedule did not anticipate that the user would click the mouse, but this minimizes lag in the useful case where a user is manipulating an object, and the computational costs can be predicted in advance.

4.1 Single-Processor Case

We use a two-phase incremental-refinement algorithm, drawing on ideas presented in Funkhouser and Sequin's system [FS93]. The first phase, in particular, uses their algorithm to create an approximate, coarse schedule that is later refined using other techniques.

Greedy Phase. The first phase is essential; it generates a feasible but not necessarily good schedule, and requires $O(n \log n)$ time, where n is the number of independent tasks. This makes it possible to bound the worst-case time of the scheduler and consider that amount of time as part of the frame time. Because of this predictability, the scheduler can place itself into the generated schedule for the next frame. Of course, it is possible to have an extremely complex scene for which it would be impossible to execute even this phase during the frame time. In this case the schedule could be recomputed only once every few frames, at some loss of responsiveness.

The greedy initial phase generates for each task i a pair $(time_{ij}, Benefit_i(time_{ij}))$ at each of the local maxima of the $Benefit_i(time_ij)/time_ij$ function. We sort these pairs by the value $Benefit_i(time_ij)/time_ij$, and repeatedly take from the list the task whose value is greatest. If the task has not yet been scheduled and there is still available time, we add it to the work list; if it has been scheduled and the new value of *time* is greater than the previously scheduled one, we reschedule it at the new time (if there is space). Tasks which do not fit in during this phase are scheduled at time 0. This produces an initial packing which is at least half as good as the result from doing the NP-complete optimal packing of these cost/benefit pairs[FS93].

We can consider this initial phase as choosing a subset of the domain of the function, over which the function is convex downward. A section which is not convex downward will never be chosen during this phase, since the value of $Benefit(time)/time$ will still be increasing over such a region. The second, incremental phase then optimizes the functions over the chosen convex sub-domain.

Incremental Phase. As an extension to the Funkhouser-Sequin algorithm, we then refine this descrete estimate further in the continuous domain. During the second phase of the algorithm the scheduler iteratively refines its generated schedule, as time allows. It can terminate at any time, since the feasibility of the schedule is never violated. We can also search through the subspace more quickly than before, since it is convex and thus any local maximum is

also a global maximum of the sub-domain.

The maximum over this continuous domain will have the following characteristics:

- $\sum_i time_i = frametime$ (all time will be used up)
- the derivatives of $Benefit(time_i)$ will all be equal

The second point is true since, for any two convex-downward functions f_a and f_b, if $f'_a(x_a) < f'_b(x_b)$, then we can increase $f_a + f_b$ by scheduling f_a for less time and f_b for more time. If the same amount of time is subtracted from f_a and added to the f_b, the first condition will remain true, and we will converge onto the second condition. This process terminates when all of the tasks have equal derivatives.

When all tasks have the same derivative, it is no longer possible to increase the benefit of the functions, so we must be at a local maximum, and since a local maximum of a convex function is the global maximum, we have reached the global maximum of the sub-domain. (This is equivalent to a gradient-descent search of an i-dimensional space, where we are able to move in two dimensions at once.)

The algorithm is as follows:

- while $\sum time_i < frametime$, set the time for the function with the greatest current $Benefit'_i(time_i)$ to the maximum possible
- then, while all the $Benefit'_i(time_i)$ are not equal, take $B_a(time_a)$, the function with the minimum $Benefit'_i(time_i)$, and $B_b(time_b)$, the function with the maximum $Benefit'_i(time_i)$. Find δ such that $B_a(time_a - \delta) = B_b(time_b + \delta)$, and reschedule the functions at the new time.

This phase also takes $O(n \log n)$ time to come within some constant epsilon of the maximum, and may be terminated before the time allotted, at some reduction in the total benefit of the resulting solution.

To determine if we should spend additional time refining the schedule before executing it, we compare the margin of change in the total benefit per iteration with the amount of time required to perform that iteration. If the scheduler should run for less time, we decrease its alloted time slightly, bounded by the worst-case time. Otherwise, we can increase its alloted time.

4.2 Multiple-Processor Case

Dedicated graphics multiple-processor workstations are becoming common, especially for high-end scientific-visualization applications. These machines allow light-weight processes which communicate using low-overhead shared memory and synchronization primitives. However, because graphics pipelines on available machines are not multi-thread-safe, the rendering pipeline can be fed from only one thread at a time.

Most multiple-processor scheduling algorithms are NP-complete (even the fairly simple case of 2 processors, no precedence constraints, and arbitrary computation times is NP-complete) [GJ75]. Since we want to schedule a set of tasks on several processors with precedence constraints, our problem is at least this hard.

We extend the single-processor greedy algorithm to generate a feasible schedule for multiple processors in $O(n^2)$ time for a guaranteed schedule or $O(n \log n)$ time for an optimistic, probably feasible one (we will explain what we mean by an "optimistic, probably feasible" schedule later in this paper).

First, let us consider how one might build a good multiple-processor schedule. Generally we have two portions of the visualization task: a compute task taking time c and a render task taking time r (possibly with a cull task inserted between them). The compute task can run on any processor, but all render tasks must stay together, and must be limited to a single processor.

Also, any data must be computed before it can be rendered, so a good schedule would have to make sure that rendering tasks would not sit idle while waiting for data. Let us consider two tasks which are being computed on a single processor and rendered on another. If we order them so that the tasks with large values of $c - r$ (which we call the *excess compute time*) are last, we have the most room possible for additions, and minimize the startup differences and ending differences between the processors, as shown in Figure 4.1.

The multiple-processor algorithm works like the single-processor algorithm, except that we modify the insertion routine to verify that adding work to the schedule doesn't produce an infeasible schedule. We initially try to add each task to the rendering processor; if there is not enough room on a processor, we push tasks onto the next processor, starting with the task with the least excess compute on the current processor. In this way, we always minimize the total amount of computation time required before rendering can begin. Note that often we will have a completely

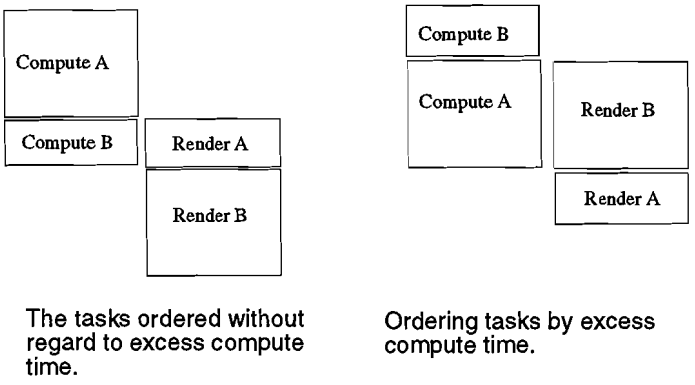

Figure 4.1: Ordering tasks by increasing excess compute time minimizes the makespan.

render-bound scene, where the render processor is completely full and there is still plenty of room for additional computation on the other processors. In this case, only purely computational tasks can be added.

Pushing a single task may cause a cascade of pushes, as shown in Figure 4.2, but we do not attempt to push a task again if a previous push on that task has failed (there is probably still not enough room for it, since pushes only go in one direction). The scheduler may attempt to push a task multiple times if tasks are repeatedly inserted before that task, so this heuristic will generally work; however, since some tasks may be shortened during the scheduling process, there may actually be room for a task which wasn't able to be pushed earlier. With this heuristic, we perform at most $O(p \cdot n)$ pushes (p is the number of processors), with each attempted push requiring an additional verification pass.

For a small number of processors, this is an acceptable algorithm. In massively parallel systems (beyond the scope of this research), it will probably be most useful to use an algorithm similar to [BL94], which scales logarithmically with the number of processors.

This is a modification of the first-fit bin-packing algorithm. Although we build up our packing dynamically, the eventual result is one which is still produceable by the first-fit algorithm, which produces a packing guaranteed to be no more than about 22 percent worse than optimal [GJ79] (this correspondence is valid in cases where our repeated-pushing heuristic is valid).

In order to guarantee feasibility, we need to look at the two possible ways in which an insertion could violate it:

- We must make sure that the sum of the work is less than the frametime, for each processor. Verifying schedule-size feasibility takes a total of linear time with the number of tasks if done as each task is added to the list.

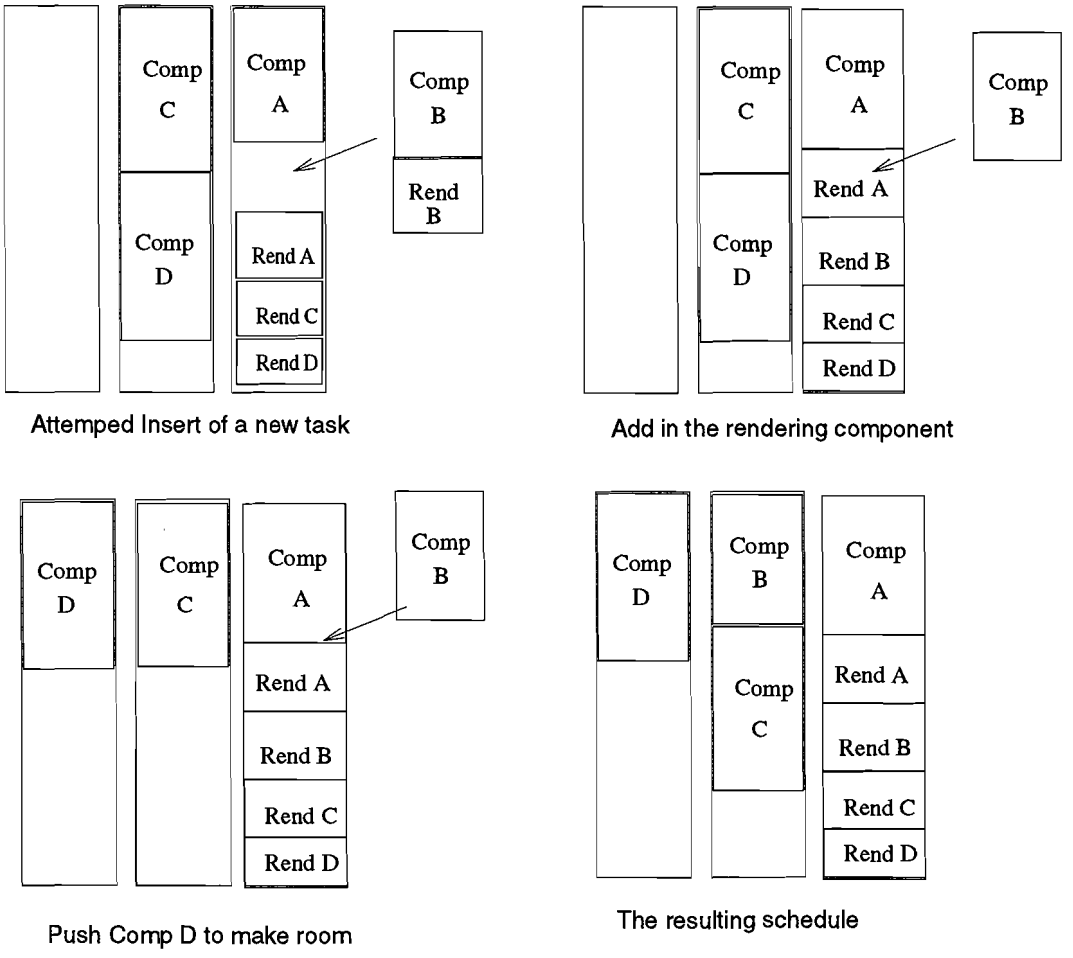

Figure 4.2: Pushing a task may cause a cascade of pushes. Here, inserting task B causes task D to move to another processor.

- Any task cannot be rendered in time less than the sum of the startup, compute, and rendering times for that task. Consider a fine-grained compute task that generates small pieces of geometry (meshes, lines, or even individual polygons and line segments) at regular intervals c during computation, after the startup time s. Rendering of any of that task's data cannot begin until time $s + c$. If the time required to render a piece is r, the total time required to render x primitives is $s + c + (x - 1) \cdot \max(c, r) + r$. Any generated schedule which violates this requirement is infeasible.

Verifying precedence relations takes a total of $O(n^2)$ time, since it is necessary to check every rendering task which is scheduled before an inserted rendering task, as well as every computational task which is scheduled

after an inserted computational task.

As described previously, when scheduling several tasks, we can lower the possibility of feasibility conflicts by ordering them from low to high excess computation. Where this value is equal, we define a consistent ordering (based on creation time) of tasks so that the partial order of tasks is identical across both the compute and rendering phases. In actual practice this heuristic, when applied to scenes containing diverse types of objects, results in schedules which rarely violate the precedence relations and which achieve high processor usage. If one can tolerate the occasional slow frame, removing the precedence checking results in the previously described $O(n \log n)$ optimistic, probably feasible algorithm.

Of course, in pathological cases, any render-dominated schedule may have only the same benefit as the single-processor schedule, but in most computationally-demanding, complex, and diverse scenes, the scheduling algorithm is able to take advantage of the different computational and rendering demands of these tasks to generate a feasible, good schedule which takes advantage of the entire multi-processor system.

The amount of time required to execute a schedule on n processors, T_n, illustrates how the expense of additional computing time varies with the number of processors available. For the single-processor case, T_1, the amount of time required is obviously $\sum c_i + \sum r_i$. At T_∞, the total time required is determined almost completely by the rendering time required, and is at most $\sum r_i + \max(c_i + s_i)$ (since, in the worst case, we may have to wait for the longest compute task to finish before beginning). Although the actual packing is highly dependent on the actual input data, a loose bound for the worst-case performance (T_n) of a schedule with n processors is at most $\sum r_i + \max(c_i + s_i) * \lceil i_{total}/n \rceil$.

It may be possible to generate a tighter bound to the worst- and average-case performance, but such an analysis is beyond the scope of this research. Although the individual parts of the system are not difficult to bound analytically, understanding the interactions among these heterogeneous scheduling techniques is a very difficult task.

Chapter 5

Implementation Description

The algorithm described here was implemented using the TRIM-lite system, a set of C++ class libraries which provides for a platform-independent abstraction of drawable objects and their associated relationships. This set of class libraries is part of the UGA system [ZCW+91], developed at Brown University.

There were two large software tasks involved in writing this system:

- Porting the code used in `fflow` (our scientific-visualization testbed, written in TRIM) to C++, with appropriate generalizations.
- Developing the time-critical scheduling algorithm described in the previous sections.

There were also a few small, but important pieces:

- Writing a platform-independent multi-threading layer.
- Instrumenting the system so that it would work with the thread-monitoring package developed by Tom Doeppner and Brian Cantrill.
- Working with Kostadis Roussos to integrate his OS-level real-time threads package, written for Solaris, into the time-critical system developed in this project.

All of the C++ code currently resides in the package `flick`.

The curvilinear grid and vector-field code from `flick` can be used independently of the time-critical scheduler portions of the project, and are currently used in three other projects: `mpflow`, used as a SIGGRAPH demo by Sun Microsystems, `flow`, a TRIM-lite system designed for the rapid prototyping of scientific visualization techniques, and `vrflow`, an experimental platform for the creation of direct-manipulation interfaces for virtual environments.

5.1 Software Architecture

Since one of the goals of this project was to create a general time-critical system for diverse types of tasks, it is important that it be easy to add new primitives to the system.

5.2 General design of an API for time-critical computing

As more people begin using 3D graphics systems (through VRML and other experiments) time-critical concerns are being discussed to a much greater extent. There are several different approaches that have been proposed as APIs for time-critical computing:

- VRML 1.0 [BPP95] allows world authors to specify a list of alternate representations of an object, each with an indicator of the distance to switch between these representations. This is extremely good because of its simplicity; however, interpreting this literally means that models must always switch based on distance, rather than the overall load of the system, which means that the world will only look "right" on the system it was authored for. Most current VRML systems allow for the user to switch between a strictly distance-based interpretation and the more resilient, load-based scheduling. Another problem with this type of description is that it is limited to a few, discrete alternate versions of the model.

- SGI's Performer [RH94] uses a system similar to that used in VRML 1.0, except that it incorporates nested sets of LOD specifications. This allows the render engine to specify different ranges of alternate representations, and (possibly) morph smoothly among them.

- One model being proposed for VRML 2.0, as part of the Moving Worlds proposal [Bel95], is that each object in the scene should be able to query the current frame rate and and simplify/complexify its representation to try to maintain the same frame rate.

  This leads to two interesting problems, however. One problem will be that if all the objects are trying to reach the same target frame rate, they will tend to magnify any oscillations in the frame rate by all trying to move in the same direction simultaneously. Without some form of global control, time-critical scheduling won't work. Another problem that could arise would be if separately authored objects have different target frame rates. If they are both seeking in different directions, then one object will be repeatedly attempting to simplify itself at the same time the other is attempting to make itself more complex, resulting in a world which is drastically different than what was intended.

- The underlying problem that all of these proposals are trying to address is, "How can we add new objects to the world, and have them behave in the same time-critical ways that built-in objects can?" We have proposed that Moving Worlds adopt an extremely simple API that will allow some more useful time-critical behavior. This would allow new types (e.g., written in Java) to define a `complexify` and `simplify` method. The scheduler would not be able to query an object about its actual level of detail, but would have to decide which objects to change based on general scene heuristics. This is primarily good because it is such a simple interface that there will be little excuse *not* to implement the two methods. Anything more complex would probably be ignored, as have many of the more subtle features of VRML 1.0.

The API described in the following section has not been tested with diverse applications, but is the API that this prototype system uses. We require the programmer to do more work than any of the above techniques, but believe that the gains derived from more complete knowledge outweigh the costs.

5.3 The flick class library

Adding a new type of schedulable object to the system is fairly simple. The code overhead for adding time-critical behavior to the entire visualization system is only about 30 lines of C++ code. There are four steps to adding a new

time-critical primitive to the system:

- subclass (possibly using multiple-inheritance) off an appropriate descendant of `FLICKcrit_task`.
- override any of the `execute_compute`, `execute_render`, and `execute_rerender` methods, if necessary. These methods perform, respectively, computation of new data, the rendering of data which is dependent on computation happening in that frame, and the rendering of data which is either static or has been cached from previous frames (thus with no dependencies).
- add one or two new `FLICKtiming_type`'s, if either a new class of rendering object or a new class of computation is being performed. These are used so that different classes of computations can time the amount of time they require.
- modify the cost/benefit functions for the new class (less commonly required).

Since some time-critical objects can contain other time-critical objects (e.g., the rake), it may also be necessary for the implementor to subclass some objects from `FLICKtask_holder`, a class which provides for higher-level groupings of time-critical objects.

5.3.1 FLICKcrit_task

The base time-critical class is `FLICKcrit_task`. This defines a number of important virtual methods which may be overridden by descendants, but which have good default values.

The most important of these methods are the `benefit`, `benefitd`, and `benefitdd` methods, which define a benefit function and its first and second derivatives. Efficient evaluation of these functions is extremely important for the overall performance of the scheduler, since they are evaluated extremely often during scheduling. This is the primary reason why we used `sqrt` as our benefit function – the more general function `pow` was too slow, taking up a large amount of time in system profiling. In the future, we may even use an efficient approximation to `sqrt` in order to speed up scheduling further.

The `schedule_points` method computes the local maxima of the $Benefit(x)/x$ function and schedules initial starting points on the passed-in scheduler.

The `set_importance` method allows the user to set a static, weighted importance value for the benefit function, as described previously.

There are two methods which can be called between frames, `reset` and `restart`. The `reset` method should be called at every frame interval, to indicate to a task that it should update its internal representations of the data. If the previously-computed data is invalidated (e.g., a streamline is moving and needs to be recomputed), the `restart` method should be called as well.

Methods which should be overriden by a descendant:

- feasible_prims – returns the number of primitive pieces which can be computed in a specified amount of time. This and the following time-based functions are generally computed using associated `FLICKtiming_types`.
- time_per_startup – amount of time required before any primitives can be computed.
- time_per_compute – amount of time per primitive computed.
- time_per_render – amount of time per primitive rendered.

5.3.2 FLICKrend_crit_task

This class extends `FLICKcrit_task` so that it accounts for objects with associated rendering costs but negligible compute cost (e.g., a static piece of geometry). It incorporates a piecewise-cubic hysteresis factor in the benefit function described previously (once again, we approximate the ideal behavior of the benefit function with something which can be computed quickly).

When defining a new type of rendering task, the programmer should pass in a `FLICKtiming_type`, which determines how long rendering operations take. Additionally, an `execute_rerender` method must be defined which actually performs the rendering and updates the render timing estimates.

5.3.3 FLICKcomp_crit_task

This class extends `FLICKcrit_task` so that it accounts for tasks with no rendering requirements, but with substantial computing requirements (e.g., the scheduling task itself). It keeps track of the number of cached primitive pieces, and

modifies the benefit function to account for comptuations which can be reused.

When defining a new type of compute task, the programmer must pass in a `FLICKtiming_type`, which determines how long startup and computing operations take. Additionally, an `execute_compute` method must be defined which actually performs the computing and updates the compute timing estimates.

5.3.4 FLICKcomp_rend_crit_task

This class combines the `FLICKrend_crit_task` and the `FLICKcomp_crit_task` classes. Therefore, it is necessary to write both `execute_compute` and `execute_render` methods for this class, and pass in two `FLICKtiming_types`. Also, the programmer must write an `execute_rerender` method, which renders pieces which have been computed in the current frame. Since the schedule is not guaranteed to be feasible, this method must be careful not to render any pieces before they have been computed (either by not rendering them at all, or waiting until each piece is ready). A spinlock may be useful for this purpose, but we simply wait for a shared variable to become true (shared-memory latency will guarantee that we don't end up in an inconsistent state).

5.4 Architecture-independent multithreading

One of the problems encountered during the implementation of this system was the lack of an architecture-independent multiprocessing model, similar in purpose the the architecture-independent 3D graphics library `od`. The Solaris machines use POSIX-compliant threads, SGI Irix has its own threads library, and HP/UX does not include threads as part of its standard OS. Therefore, we defined a simple multiprocessing model which transparently adapts to the number of processors available.

Additionally, on the Solaris machines, our model incorporates Tom Doeppner and Brian Cantrill's `threadmon` library, which allows for graphical performance monitoring of multithreading calls. This graphical monitor was very useful in determining where system bottlenecks were, and helped us to realize that substantial improvement was possible in several phases of our scheduler.

The `FLICKmp_manager` class determines how many processors exist on that machine, and creates that number of

`FLICK_processors`, which do not necessarily map to actual physical processors (kernel-level scheduling frustrates many attempts at closer control). Each of these virtual processors has two methods: `start_execute`, which tells that processor to begin executing a function, and `wait_for_done`, which blocks the calling process until the currently-executing function (if any) is finished.

Although this is an fairly simple multiprocessor model, it is sufficient for the types of control required in this program (building up and executing work queues). Additionally, it enables us to develop our system on the HPs, which only have a single processor and do not support multithreading (HP *does* sell multithreading libraries, but these are non-preemptive).

5.5 Implementation Results

We tested this algorithm on several different systems: a single-processor HP 735 TVRX, a two-processor SGI Onyx with a Reality Engine 2 graphics accellerator, a single-processor UltraSPARC Creator, and a four-processor Sun SPARC 10 ZX.

Although both the SGI and Solaris systems allow for privileged users to take advantage of hard real-time scheduling techniques, dedicating processors to specific tasks, we have initially attempted to use only user-level scheduling techniques to to run our real-time usability tests. Recently we have begun to experiment with OS-level real-time functionality, in particular in Solaris, but do not yet have any conclusive reports.

On the SGI Onyx machine, simulating the allocation of 100 milliseconds (one complete processor at 10 frames/second) for the scheduling phase, we fill 97% of the other three processors, when the task is compute-bound. With 50 milliseconds allocated for the scheduling task, the algorithm still manages to fill 92% of the available processor time. The benefit of the algorithm declines sharply after this point, however. Giving the scheduler 25 milliseconds resulted in only 65% processor usage, while 15 milliseconds dropped to 32% usage, resulting in less computational time than a single processor.

Chapter 6

Future Work

We intend to perform additional investigations to understand how the scheduling algorithm's behavior degrades under stress, and modify the search techniques which it uses to cope better with situations involving too much work or too little time to schedule.

Implementing the interleaved version of this algorithm, in which it schedules itself as a time-critical task onto the work queue, scheduling the next frame while the current frame is being executed, has has raised several interesting questions, involving feedback problems and how to deal with unexpected user input (the scheduler can't tell when a user will begin to manipulate a large, complex visualization tool, so will necessarily lag slightly behind in that case).

It is also be extremely important to understand more about human perception in complex, data-intense environments, in order to generate more realistic benefit functions. As an example, degrading the individual books in a complex walk-through of a library would be useful for an architect, but would be an extremely bad for a researcher walking down the aisle, trying to find a book based on the memory of its color. Similar but far more complicated cases arise in fluid visualization, where a scientist looking for vortices may require different degradation tactics than a scientist investigating laminar air flow.

Because the different tools may also change their relative significance to different scientists, it will also be necessary to work with scientists to determine the actual semantic benefit of each tool when performing varying types of tasks. Unfortunately, there are no easy answers in this area, and it seems that any good heuristics might require major

knowledge-modeling efforts, on a user-by-user basis.

Another interesting research area is in dealing with highly-complex time-varying environments. In such cases, even when we have good heuristics describing how beneficial the current visualization is, we are not able to predict the benefit of a visualization in the next frame (a feature may arise which did not exist before, for example).

6.1 Acknowledgments

Thanks to Steve Bryson, who provided the impetus to work on time-critical scientific visualization problems, and who provided invaluable suggestions during the course of this work. John Hughes helped formalize some of the math involved, and Andries van Dam guided the course of the research. There were many people in the Brown Graphics Lab who helped with various parts, especially Bob Zeleznik, Kenneth P. Herndon, Matthias Wloka, Sam Trychin, Tim Miller, and Currier McEwen. Discussions with Tom Doeppner, Pascal van Hentenryck, and Philip Klein also proved invaluable.

We would also like to thank the sponsors of the Graphics Lab: NASA, NSF, Sun, SGI, HP, Microsoft, and Taco.

Bibliography

[Bel95] Gavin Bell. Moving Worlds. `http://webspace.sgi.com/moving-worlds/`, 1995.

[BL91] Steve Bryson and Creon Levitt. The Virtual Windtunnel: An Environment for the Exploration of Three-Dimensional Unsteady Flows. In *Visualization '91*, pages 17–24, 1991.

[BL94] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–268, November 1994. `http://www.cs.utexas.edu/users/rdb/FOCS94.ps.gz`.

[BPP95] Gavin Bell, Anthony Parisi, and Mark Pesce. The Virtual Reality Modeling Language Version 1.0 Specification. `http://vrml.wired.com/vrml.tech/vrml10-3.html`, 1995.

[DM95] Nathaniel I Durlach and Anne S. Mavor, editors. *Virtual Reality: Scientific and Technological Challenges.* National Academy Press, 1995.

[FS93] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 247–254, August 1993.

[GD95] Loyd Greenwald and Thomas Dean. Anticipating Computational Demands when Solving Time-Critical Decision-Making Problems. *The Algorithmic Foundations of Robotics*, 1995.

[GJ75] R. Garey and D. Johnson. Complexity Results for Multiprocessor Scheduling Under Resource Constraints. *SIAM Journal of Computing*, 1975.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[Hol92] Richard L. Holloway. Viper: A Quasi-Real-Time Virtual-Environment Application. Technical Report TR92-004, University of North Carolina, Chapel Hill, 1992.

[LLS+91] Jane W. S. Liu, Kwei-Jay Lin, Wei-Kuan Shih, Albert Chuang shi Yu, Jen-Yao Chung, and Wei Zhao. Algorithms for Scheduling Imprecise Computations. *IEEE Computer*, pages 58–68, May 1991.

[Mok83] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

[RH94] John Rohlf and James Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real–Time 3D Graphics. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 381–395. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[RSZ89] Krithi Ramamithram, John A. Stankovic, and Wei Zhang. Distributed Scheduling of Tasks with Deadlines and Resource Requirements. *IEEE Transactions on Computers*, pages 1110–1123, August 1989.

[SSNB95] John A. Stankovic, Marco Spuri, Marco Di Natale, and Giorgio Buttazzo. Implications of Classical Scheduling Results for Real-Time Systems. *IEEE Computing*, pages 16–25, June 1995. `ftp.cs.umass.edu, /pub/ccs/spring/impl_sch_rts.ps`.

[ZCW+91] Robert C. Zeleznik, D. Brookshire Conner, Matthias M. Wloka, Daniel G. Aliaga, Nathan T. Huang, Philip M. Hubbard, Brian Knep, Henry Kaufman, John F. Hughes, and Andries van Dam. An object-oriented framework for the integration of interactive animation techniques. *Computer Graphics (SIGGRAPH '91 Proceedings)*, 25(4):105–112, July 1991.