Dissertation Proposal: Time-Critical Graphics

Matthias M. Wloka

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-50
November 1993

Dissertation Proposal: Time-Critical Graphics

Matthias M. Wloka
CS Department
Brown University
Providence, R.I. 02912-1910
E-mail: mmw@cs.brown.edu

November 1, 1993

1 Introduction

Personal computing is expanding to include the latest developments in interactive 3D graphics, multimedia, and virtual reality. These three fields all combine the need for high throughput and low-latency response times with degradable performance demands. The batch-job computing paradigm prevalent in today's applications is inadequate for addressing these requirements.

In the batch-job computing paradigm, a user submits a computation request and then waits for the result. The computation time for the result depends on the complexity of the request. Ever-faster hardware and better algorithms make response times bearable. However, faster hardware is not the ultimate answer to achieve fast response times, since computer users are always able to use — and exceed — whatever processing power is available.

Instead, I propose to apply a time-critical computing paradigm to computer-graphics computations, but not rendering. In this paradigm, the user steers the performed computations by trading various quality parameters — and computation time is one quality parameter — for one another. In order to achieve faster computation times, other aspects of the computation, for example the accuracy of the computed result, are degraded.

A time-critical computing paradigm is well suited for highly interactive environments, such as are found in interactive 3D computer graphics and virtual reality, and processes inherently linked to real time, such as those in multimedia and virtual-reality applications. It guarantees low-latency response times and allows synchronization of the application to real time.

Timely delivery of computation results is the most important attribute in these fields; presentation quality is only a secondary concern. For instance,

virtual-reality applications require fixed, relatively high frame rates with little lag, since otherwise the user becomes disoriented, or worse, motion-sick [21] [32]. It is of comparatively less relevance whether the rendered objects are Gourad-shaded, flat-shaded, or wire-frame. Presentation quality is expendable: to be wrong and on time is more valuable than to be right and late.

Flight simulators and video games are more traditional examples of applications that value speed over presentation quality. However, designers of such systems achieve the real time computation times by assessing the worst case computation requirements a priori, and choosing software algorithms and hardware to satisfy them. Since they base their choices on a worst-case analysis, the resulting system is rarely fully utilized. Applying a time-critical computing paradigm instead would better exploit the capabilities of such a system.

Computation speed is only one limiting hardware factor. The time-critical computing paradigm can be generalized to take other factors into account, for example, amount of memory or available network bandwidth. (Note that, for example, compression algorithms are already to balance storage space with processing power.) However, I concentrate on time-criticality because of its paramount importance to the above mentioned fields of interactive 3D graphics, multimedia, and virtual reality.

This time-critical approach differs from the traditional computer graphics outlook, displayed, for example, in the field of photorealistic rendering [24] [9]. The objective there is to create realistic, static images, regardless of computation cost. Presentation quality takes absolute priority over computation time. Performance/presentation-quality tradeoffs are neglected, even though the advantages of iterative computing, i.e., progressive refinement rendering methods [8], are valued for their faster feedback time. The emerging field of virtual reality — an inherently real time bound, interactive domain — has made researchers aware of the need for explicitly balancing computation speed with quality.

1.1 Outline

I intend to design and build a time-critical graphics system. In addition to the usual 3D graphics primitives, the system will incorporate multimedia, for example, live or prerecorded video streams. Multimedia presentation is similar to 3D graphics in that it is inherently dynamic and sensitive to timing and synchronization issues. In order for multimedia to affect virtual-reality applications, it must to be integrated with 3D graphics on a low level; synchronizing separate multimedia displays and synthetic graphics display windows is insufficient. Instead, I will integrate and synchronize multimedia data with the 3D graphics primitives.

The system is intended to demonstrate the advantages of the time-critical approach: first, guaranteed, fixed response times heighten the user's perception of reality in dynamic, computer-generated scenes [5]. Second, low-latency feedback becomes possible, improving application-specific task performances.

Third and last, incorporating time into the system eases the inclusion of real time dependent data, such as multimedia.

Such a time-critical graphics system integrates three main characteristics that are not addressed by traditional graphics systems. First, in order to synchronize objects to real time, the system needs to be aware of time. Consequently, the system incorporates *wall-time*. Wall-time is an example of an *external signal*. Other external signals, for example, user input or sensors, are equally useful in an interactive graphics system. The first characteristic therefore is the explicit incorporation of general, external signals.

Second, I introduce *update rates* to computer graphics, integrating them with the continuous-time model prevalent in traditional graphics systems. Since dynamic computer graphics is ultimately sampled, e.g., when displaying discrete frames, and since multimedia is traditionally presampled, it is essential to include update rates in the system to control discretization of the time domain. Just as polygon resolution is a parameter that controls rendering quality of abstract graphics primitives, update rate is a parameter that controls rendered behavior quality. Update rates do not replace the continuous-time model, just as polygons do not replace the abstract graphics primitives. The second characteristic therefore addresses update rates of the various graphics and multimedia primitives.

Third and last, the previous two characteristics necessitate attending to *time-critical computing* concerns. Since the first characteristic specifies object behavior in relation to external signals such as time, and the second characteristic specifies how often that behavior is computed, the amount of time available to compute an object's behavior is constrained. The third characteristic therefore attends to time-critical computing issues, such as compute-time predictions, quality degradation, and scheduling.

The structure of the remainder of this proposal is as follows. In Section 2 I describe three example application scenarios that will benefit from or are made possible by this work; these scenarios justify the research. I expand and detail the outline presented above of the three characteristics in Section 3; in particular, I describe the exact requirements of each. Prior and related work is discussed in Section 4 in a format that stresses the history and evolution of graphics system models. Finally, in Section 5 I summarize the presented approach.

2 Application Scenarios

In this section I describe three different graphics applications taken from the areas of virtual environments and software engineering. Each application highlights various features introduced by the proposed system. They all share the need for a time-critical computing paradigm.

2.1 The Virtual Office

The virtual office is an augmented-reality application that combines reality with computer-generated images [3] [13]. Users wear head-mounted, see-through displays. In the view of the user, the actual office is augmented with computer-generated icons that provide 3D interfaces to computer applications. The immediate advantage of the virtual office over the desktop metaphor is the vastly increased area that is available for laying out icons and other application-related imagery. Since distinctive, stationary, and — most important — real navigational cues, such as desks and shelves, populate the office space, navigation through the virtual information space is easy.

An example application for the virtual office environment is a teleconferencing telephone. Teleconferences use video streams to represent remote participants. The head-mounted display treats these video streams as computer graphics objects, thereby integrating them into the environment and eliminating the need for additional, physical monitors in the office.

A user interface metaphor works only if the objects within the interface behave as the user expects. In the virtual office, where computer-generated objects overlay the real world on see-through displays, the user expects the computer-generated objects to update as fast as the real world. Worse, the lag between a user action and the system's response has to be perceptually instantaneous. To achieve such response times, new strategies, for example, predicting user actions and initiating computations before they are needed, are required.

Thus, to make the virtual office real, a number of open research problems must be solved. I focus on only a few of these problems: the tight integration of video data with computer graphics, time-critical computation of object behavior, and incorporation of external signals, such as user actions, into a larger framework.

2.2 Walk-Throughs of Dynamic Environments

Interactive, architectural walk-throughs [6] [15] place a user, who wears a head-mounted display, inside a virtual environment, typically an architectural model. The user interactively explores the environment by moving through the data set.

So far, 3D interactive walk-throughs are preeminently static: only the user changes position and orientation; all other objects in the environment, except for individual objects the user interacts with, are fixed. These limitations result from the tight performance bounds of the application. If only the user changes position and orientation, then the application need only rerender the data set. In contrast, highly dynamic environments require regular recomputations of the data set itself. These recomputations are costly, delay the display of frames to non-interactive rates, and therefore destroy the user's illusion of immersion.

In order to produce a realistic image stream for highly dynamic and interactive environments, a constant, fast frame rate has to be maintained. To do this, the regular recomputations of the data set have to complete in prespecified time intervals. Recomputing the data set is therefore time-critical, and the various computation tasks must be degradable to fit within the prespecified time-slots.

Two of my proposed system's main characteristics help solve the problems associated with dynamic, interactive walk-throughs. First, the time-critical computing mechanisms allow me to schedule the various recomputations. Second, the update and sample rates associated with all primitives within the system act as a degradable quality parameter for dynamic processes. High sampling rates result in smooth motion; low sampling rates save computation time, but produce low-quality behavior. The system therefore inherently supports degradation for dynamic or interactive objects. In contrast to the work described in [14], I apply prediction, scheduling, and degradation to the computation of dynamic environments, not to the rendering of static environments.

2.3 Architecture-Independent Applications

One goal of software engineering principles is to make an application architecture-independent. Ideally, applications are compiled and run on a variety of architectures without having to change the associated source code. This goal is achieved by relying on software libraries, which encode a general, conceptual model of the underlying hardware functionality.

None of the existing conceptual models take into account the amount of time it takes to execute compute tasks. (This is not surprising, since these conceptual models are based on the batch-job computing paradigm.) These conceptual models therefore do not address time-sensitive computations, i.e., computations for which the delivery time of the result is more important than the result's quality. Time-sensitive computations are hence architecture-dependent: they depend on the architecture's processor speed.

My research stresses an application's dependence on time. In particular, application data and computations are linked to real time. Therefore, computations deliver results according to the associated, user-specified time constraints and are independent of processor speed. To achieve this new level of architecture independence, other application-specific quality parameters are degraded.

3 Requirements

In this section I motivate and describe the requirements of a time-critical graphics system. These requirements are grouped into three sections, corresponding to the three main characteristics of my proposed time-critical graphics system: external signals, update rates, and time-critical computing.

3.1 External Signals

External signals describe aspects of the real world to the computer graphics system. Examples of external signals are wall-time, i.e., the time that passes in the real world, user input devices, and sensors, i.e., position or temperature sensors, etc. An external signal is generally modeled as a function whose domain is wall-time in the interval $(-\infty, now]$, where now is the current wall-time. Typically, an external signal function is highly irregular and therefore hard to approximate or to predict. A special case is wall-time itself, which is interpreted as the identity function in the above general definition.

My system will include external signals as a basic notion. Graphics objects can depend on an external signal, thereby coupling their behavior to events in the real world. These dependencies are either direct or functionally mapped. In a direct dependency the graphics object directly uses the raw, untransformed external signal values; for example, a graphics object spatially displaying temperature incorporates the output of a temperature sensor in its position computations. More often, the dependencies are functionally mapped, i.e., the graphical objects use offset-based, scaled, or warped versions of the external signal values. Systems already exist that functionally map wall-time to objects [38] [12].

Since the system explicitly incorporates an external signal model, the resulting relationships between objects, i.e., temporal and interactive relations, are explicit as well. Multimedia and other time-dependent data rely on this feature to stay in synchronization with the real world. I illustrate these benefits for time-dependent and interactive applications throughout Section 2.

Due to memory limitations, an actual implementation cannot store the external signals over their whole domain $(-\infty, now]$. Since external signals are irregular, the implementation must approximate them by storing samples over a much smaller interval, i.e., $[now - c_t, now]$, where c_t is an implementation-specific constant, e.g., one second. The interval $[now - c_t, now]$ shifts as wall-time progresses and the value of now changes. The external signal is interpolated between sample points within $[now - c_t, now]$ and extrapolated in the interval $(-\infty, now - c_t]$. As mentioned above, these approximations are inaccurate to a degree depending on the external signal and the number of stored samples.

Often applications find it useful to inquire an external signal in the future, i.e., at some point $now + f_t$, for some value of f_t (see Section 2.1 for an example). I therefore extend the domain of the external signal functions to be wall-time in the interval $(-\infty, +\infty)$. Since, in general, it is impossible to predict an external signal with accuracy, all inquiries in the interval $[now, +\infty)$ are only a guess (obtained, for example, by extrapolation).

Note that it is therefore possible to inquire an external signal anywhere in time, even if the associated sensor or user input device supplies samples only at discrete time points. This functionality contrasts with traditional user input models [20], which are event-based and therefore discrete in time. However,

a continuous model of external signals is of advantage for the continuous-time models prevalent in computer graphics. The continuous-external-signal model has the disadvantage of returning inaccurate results for a signal in the future, in the distant past, and in-between samples — but the event-based user input models return no result for these cases.

The above implementation model for external signals (i.e., storing samples for the continuously changing interval $[now - c_t, now]$ and inter- and extrapolating all other inquiries), while a good first approximation, is severely flawed: when inquiring about an external signal at an absolute time abs_t, such that $abs_t > now$, the returned value $ext\_sig(abs_t)$ is a prediction. Reinquiring the same value slightly later, i.e., inquiring at abs_t when $abs_t < now$, results in a value that is computed via inter- or extrapolation from different samples, yielding a different $ext\_sig(abs_t)$, i.e., $ext\_sig(abs_t) \neq ext\_sig(abs_t)$ — a mathematical paradox.

I therefore propose the following model for external signals: an external signal is a function whose domain is the cross-product of wall-time in the interval $(-\infty, +\infty)$ and transaction-time in the interval $[now - hist, now]$:

$$ext\_sig : \text{wall-time} \times \text{transaction-time} \rightarrow \text{Ext\_Signal\_Values}$$

The constants c_t and $hist$ are implementation-specific; they describe the actual stored time interval for wall-time and transaction-time, respectively. Since these time intervals are finite, I also require conditions to ensure consistency for similar inquiries. Inquiries at different (transaction-) times for the same value of an external signal must return the same result, if both inquiries fall into the stored wall-time intervals (see Figure 1):

$$\forall wt, now_1 - c_t \leq wt \leq now_0 : ext\_sig(wt, now_0) = ext\_sig(wt, now_1).$$

Similarly, the extrapolation mechanisms for prediction and distant history (i.e., the shaded areas in Figure 1) should be reasonably accurate, at least for values of wt close to now_0 and $now_1 - c_t$:

(Prediction)	$\forall wt \approx now_0 :$	$ext\_sig(wt, now_0) \approx ext\_sig(wt, now_1)$
(Dist. History)	$\forall wt \approx now_1 - c_t :$	$ext\_sig(wt, now_0) \approx ext\_sig(wt, now_1)$

Transaction-time, a term borrowed from temporal databases [26], describes which version of the external signal description is accessed. A new version is created every time the newest sample value overwrites an old sample. For example, hard-wiring transaction-time to the current time, i.e., using $ext\_sig(., now)$, returns the most up-to-date version of the external signal, effectly simulating the previous external signal definition, while avoiding the mathematical paradox described earlier.

Removing this restriction on transaction-time and allowing inquiries of the form $ext\_sig(., trans_t)$, such that $trans_t \in [now - hist, now]$, permits the selection of various versions of the external signal. While such a change makes the

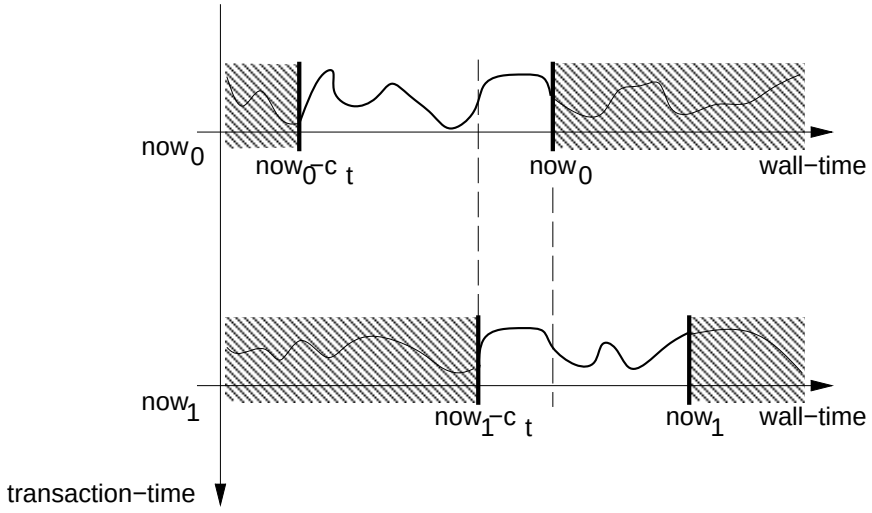

Figure 1: Consistency Conditions

model more powerful, it also requires additional memory to store the additional information needed to support this model.

In summary, including external signals in the system has several advantages. First, objects depend on wall-time via user-specified mappings. Second, objects specify interactive behavior explicitly, as opposed to outside entities editing objects to achieve interactive behavior. Third, temporal and interactive relations are therefore explicit. Fourth and last, the time model for external signals is continuous, allowing the sampling of user input anywhere in time.

3.2 Update Rates

The *update rate* of an object is the rate at which its state is recomputed. The update rate of individual objects therefore influences how often the object's displayed representation is refreshed. Multimedia-related applications also substitute the term *sampling rate* for update rate. I use the term update rate and not sampling rate. *Frame rate*, on the other hand, is the update rate of the renderer, i.e., the frequency with which it generates new frames.[1]

Graphics systems commonly equate an object's update rate with the overall frame rate. Notable exceptions are the Alice & DIVER system [31], the MR Toolkit [33], and VPL's RB2 [4]; all these systems distinguish frame rate from the overall scene update rate. Nonetheless, all these systems update all objects

[1] I am assuming a unique frame rate associated with a single display. Nonetheless, several displays with various, independent frame rates are possible.

in the scene conceptually simultaneously to produce the next frame in an animation or simulation; they inquire the most up-to-date information from every object in the scene. This inquired information is then sent to the renderer, which incorporates it into the next displayed frame.

In general, update rates are inherent to graphics systems, since all objects are dynamic entities that need to be inquired at particular points in time to obtain their specific value. In this work, I envision that every object potentially has its own update rate, indeed all objects are updated possibly independent of each other. Many objects, for example slow-moving objects, relatively unimportant background objects, or static objects, do not require recomputation at every displayed frame; it suffices to only occasionally inquire and recompute these objects. Thus, the update rate of these objects is slower than the frame rate. The system can redistribute the saved computation time to objects that require high update rates. I describe examples of such scenarios throughout Section 2.

Conversely, if the underlying hardware provides sprite or sprite-like rendering capabilities,[2] it is possible to render user input device representations faster than frame rate, thus improving interaction latency dramatically. Current workstations apply this principle to the mouse cursor to achieve acceptable, interactive performance. The principle can be generalized: user-interface objects and objects with which the user is interacting can also be updated and rendered at a rate faster than frame rate. Faster-than-frame-rate update rates are also useful for physical simulations, which typically proceed with small time steps, i.e., with an update rate higher than frame rate, to produce more accurate numerical results.

Allowing the user to control the update rates inherent to an animation is a new concept. Traditional graphics systems determine the update rates in an ad hoc way, transparent to the user. They operate in an as-fast-as-possible mode: a new frame is computed as soon as the computation of the previous frame finishes. This mode of operation is not optimal. I therefore propose to let the user control the update frequencies. Control is more valuable than global as-fast-as-possible sampling, as the following scenarios show.

Previewing an animation at the same frame rate it is to be recorded on tape is desirable: displaying an animation at 40 frames per second makes it look smooth while previewing, yet, when recording it at 20 frames per second, time-aliasing artifacts make the resulting video unacceptable. Had it been possible to preview the animation at 20 frames per second, the problem could have been identified and solved, for example, by applying motion blur techniques.

In addition, when integrating multimedia data types with computer graph-

[2] A sprite is a frame-buffer-independent graphic that blends with the frame-buffer graphics at every screen refresh. Typically, a sprite overlays the frame-buffer, but other, simple foreground/background blends are possible. Since these blends are performed 60 times a second and memory access times are finite, only a limited number of memory accesses can be performed, thereby restricting the number of possible sprites. Typically, hardware platforms provide less than sixteen sprites of limited extent.

ics, multimedia must adopt a continuous-time model. However, multimedia data types are presampled, in contrast to computer graphics objects, which are synthetic. *Synthetic* data types are defined ("continuously") over the whole time-space and can be inquired at any point in that space. *Presampled* data types, on the other hand, are defined only at certain points within the time-space. While it is possible to cast a presampled data type into a synthetic one by discretely interpolating it, it eradicates the essential feature of the presampled data type: new data is only supplied at predetermined points within the sampling space.

Presampled data types seem to resemble user input devices or external signals in general — both are sampled at discrete points in time. Yet I advocate external signals as a synthetic data type in Section 3.1. However, presampled data types and external signals differ critically in two respects.

First, a presampled data type is conceptually discontinuous. For example, a stream of video frames is likely to include "cuts" — a succession of discontinuous changes in scenes or camera positions and angles. Interpolating such a data stream non-discretely therefore corrupts rather than improves it.

Second, a presampled data type is typically more complex than an external signal; for example, a stream of video frames is more complex than a stream of pointer-device positions. Interpolating a presampled data type is therefore more expensive — too expensive in the context of the interactive, real time systems I am interested in.

Even though the distinction between presampled data types and external signals is not sharp, presampled data types are justified. In order to integrate multimedia it must therefore be possible to acknowledge and honor predetermined sampling rates.

Controlling the update frequencies of all objects also avoids unnecessary computations. For example, a simulated animation of a bee flying around a growing flower does not need to update the flower growth as frequently as the bee, since the (simulated) plant growth progresses at a slower rate than the movements of the bee. Extending the concept further and allowing separate update rates for the various parts of a hierarchical object, or even the individual attributes of an object, increases possible performance savings.

At a minimum, these update rates apply to each computer graphics object and, at a maximum, to every specifiable attribute within the system. An object's (or an attribute's) update rate is influenced by the user, the system (trading quality for performance automatically), the object itself (presampled data types), or a combination of the three. This variety of influences reflects the required types of update rates: regular (i.e., updating an object every n milliseconds), irregular, dynamic, or dependent on external events.

Summarizing, my system introduces pervasive update rates to computer graphics. As already mentioned, the specification of update rates does not replace the already existing continuous-time models; it complements them. The user, the system, or the objects themselves control update rates, which can be

regular, irregular, dynamic, or external-signal-dependent. Explicitly controlling sampling rates per object or even per attribute is advantageous, because it allows fine control over computation performance and produced quality. Such control is necessary in a system that tries automatically to weigh, adjust, and trade various quality parameters, degrading one parameter to achieve higher quality in other aspects.

3.3 Time-Critical Computing

The combination of the above two requirements, dependency on external signals and user-specifiable update rates, makes it necessary to address *time-critical computing* concepts: If the user specifies only update rates without linking them to wall-time (or, in general, external signals), i.e., if update rates are specified only in relation to each other, then the underlying system need not be concerned with how long it takes to generate a frame. In effect, all traditional (and in particular batch-) animation systems [34] [42] operate on this principle: an animation scripting language defines behaviors by linking actions or operations to time lines or time variables. These abstract time lines or time variables, called *animation time*, are independent of external signals. The user specifies the update rate by choosing points in animation time for which the system is to generate a frame. However, the computation time for generating a frame or the time at which the frame is displayed is unrelated to the animation time the frame represents.

Similarly, if only synchronization to external signals is specified, without specifying update rates, then the underlying system need not control the time it takes to generate a frame, either. To synchronize animation time to the external signal, it is only necessary to parameterize a frame-generation request with the current value of the external signal. As long as frame-computation times are all at least roughly of the same length Δt, the displayed stream of frames is synchronized to the external signal with a lag of Δt. Both Cardman's system [7] and TBAG [12] [11] work in this manner. To eliminate the lag, or to handle situations in which frame-computation times vary substantially, the underlying system needs to be able to predict how long the computation of a frame takes. The frame-generation request is parameterized not by the current value of the external signal, but instead by the value the external signal is predicted to have at the time the frame-generation is completed. I therefore allow the user to trade the error introduced by lag for the error generated by the prediction mechanisms. The user can revert to the lag-error by using a prediction mechanism that always returns the latest known value.

In contrast, letting the animator control both specification of synchronization to external signals, as well as per object update rates, places demands on the underlying system to compute various aspects of the scene in a prespecified amount of time. If the prespecified amount of time is adequate, nothing further needs to be done. The typical case, however, is that too little time is available;

the demands can only be met if the system degrades various quality parameters of the objects. My system therefore needs to be able to predict computation times, degrade quality parameters to improve computation speed, and schedule the execution of computation tasks. I discuss each of these three issues in their own subsections.

3.3.1 Prediction

Accurate prediction of computation times in general is impossible, since it is undecidable whether a given program terminates or not. However, I do not strive for exact predictions or predictions of general programs. For example, a simple feedback loop is often adequate. Furthermore, the application programmer can supply the prediction mechanism with hints on expected execution times.

Prediction of computation times is necessary for two reasons. First, a system that synchronizes to wall-time, or external signals in general, needs to assess whether it is possible to finish all required computations in the allocated time. Simply starting up all necessary computations and hoping to finish on time is not a solution. Instead, the system needs to predict the complexity of the computation: if the allocated time is sufficient, the system initiates the required computations. If the computations cannot be completed before the deadline, the system has to signal failure, or, if possible, try to simplify the computations (see Section 3.3.2).

Second, if computations are correlated to wall-time, for example, to display motion accurately over time, then a frame displayed at wall-time t_{frame} needs to show the objects in the frame at that time, i.e., as of time t_{frame}. The problem is that t_{frame} is unknown at the beginning of the computation of the objects. Conventional systems disregard this problem and inquire all objects at the time the computation begins, i.e., at time $t_{begin\_computation}$. In effect, they therefore assume $t_{begin\_computation} = t_{frame}$, i.e., they assume that inquiry and rendering of the objects is instantaneous.

A slightly more accurate method is to compute the dynamic objects by passing the current value of wall-time as the time parameter. While this method minimizes lag, it produces frames that show each object as of a slightly different time.

More precisely, the first object is inquired at time $t_{begin\_computation}$. I call the amount of time needed to compute the inquiry of the first object Δt_1, or, in general, for the kth object, Δt_k. Note that the value of Δt_k is unknown until the computation actually finishes. The second object is then inquired at time $t_{begin\_computation} + \Delta t_1$. In general, the kth object is inquired at time

$$t_{begin\_computation} + \sum_{i=1}^{k-1} \Delta t_i$$

When all object inquiries and computations are complete, the frame is dis-

played. Using the same notation and assuming there are a total of n objects, the time the frame is displayed is therefore

$$t_{frame} = t_{begin\_computation} + \sum_{i=1}^{n} \Delta t_i$$

Thus, each object in the frame represents a different time, and none of the objects represents t_{frame}. To avoid generating frames that display objects that do not correspond to t_{frame} therefore requires that I inquire all objects at time t_{frame}. However, since t_{frame} depends on the Δt_k, t_{frame} is unknown as of $t_{begin\_computation}$. Predicting each of the Δt_k enables me to predict t_{frame}.

Two approaches to predict compute times exist: static or dynamic. A static prediction mechanism associates approximate compute times with the various computation tasks at compile time. The static prediction mechanism derives these approximate compute times, for example, by timing a test execution once, or by analyzing the task. A dynamic prediction mechanism, on the other hand, times computation tasks on the fly. The result of the timing influences the prediction of the length of future tasks, effectively implementing a feedback loop.

Static and dynamic mechanisms complement each other. For example, a static prediction algorithm cannot account for the current CPU-load, yet the CPU-load strongly influences a dynamic prediction algorithm. Ideally, the prediction mechanism used is a combination of the two concepts.

The prediction mechanism, regardless of whether it is static or dynamic, needs to be fast. If predicting the compute time of a task takes any significant fraction of the time of actually executing the task, then it is useless. It is more efficient to execute the task, instead of predicting its compute time and then computing it. To a first approximation, the prediction mechanism needs to be an order of magnitude faster than the predicted compute time to allow for sufficient time to actually execute the task.

I do not expect predictions to be accurate — too many variables influence the computation time of any given computation. I plan to define a simple cost heuristic that, based on only a few key variables, for example, processor speed and previous execution times, approximates execution times.

3.3.2 Degradation

A system that incorporates user-specifiable update rates and synchronization to external signals limits the execution time for inquiring objects. Often such a system requests computations that cannot be completed in the allocated time. The computation requests must be simplified to fit them into the given time-frame. To simplify computations it therefore needs to be possible to degrade various quality parameters.

Several issues need investigation. First, I need to identify the various possible quality parameters in a graphics system. Some examples of quality parameter

are update rate and rendering attributes. Update rate is a quality parameter that influences the observed temporal behavior of an object and that is applicable to all types of graphics primitives. Rendering attributes, such as polygon resolution, lighting model, and approximate representations, effect not only rendering performance, but also the data computation time.

Second, each quality parameter degrades in a different way. Some change continuously (e.g., update rate), while others have only discrete values (e.g., polygon resolution). Therefore, different quality parameters warrant different degradation methods, for example, parameterized computation routines or explicit, alternative versions. Regardless of how degradation is implemented, the loss of quality for each degradation step has to be qualified and quantified.

Third and last, after quality parameters and their degradation methods are identified, I need to determine their cost (see Section 3.3.1). A cost-benefit model needs to be formulated. Benefit of an object can be measured, for example, by relative size, direct user specification, or relative position to the camera (foreground/background, center/off-center, etc.). Such a cost-benefit model should penalize the rapid flip-flopping of attribute settings to avoid visual ringing [14].

Once these questions are answered, a system has a variety of ways to improve performance by degrading the various quality parameters. The objects with the highest cost-benefit values are degraded least, whereas low cost-benefit objects are degraded most.

3.3.3 Scheduling

The execution of the various computing tasks needs to be coordinated by a scheduler. A scheduler needs to know about computation requests in advance to divide the available computation time appropriately. Simply starting to compute the requests as they arrive is insufficient, because it is likely that the later requests are then not able to finish in time.

The scheduling task in a time-critical graphics system is comparable to the scheduling tasks found in hard real time operating systems [35]. It is simpler, because the deadlines imposed by the graphics system are not as critical, yet it is also more complex, because of the added degrees of freedom. A given computation task in a time-critical graphics system is not of a fixed length: computation times vary depending on the available and chosen degradation schemes for that task. Therefore, the scheduler not only has to decide on the order and importance of the various computation tasks, but it also has to decide how much to degrade each task. Furthermore, time-critical graphics systems do not perform the same computations repeatedly. In fact, even the importance of each task (task priorities in real time operating system jargon) varies dynamically. Therefore, all scheduling decisions need to be dynamic and cannot be precomputed, imposing time-limits on the scheduling process itself.

The general problem of finding the optimal schedule for a given set of degrad-

able computations in a predetermined amount of time is hard. Since it is equivalent to the Continuous Multiple Choice Knapsack Problem, it is NP-complete [14]. Like the prediction mechanism described in Section 3.3.1, the scheduler runs in real time. Therefore, I cannot hope to find the optimal solution; I need to make fast approximations [16].

The scheduler bases its decisions on a broad collection of global information. First, it knows about all current computation requests. Second, to avoid bottlenecks, the scheduler should also be aware of future compute requests, allowing the scheduling over ranges of time. Third and last, it needs to know about available degradation options; it therefore acts over ranges of computation times.

4 Prior Work

In this section I survey high-level, conceptualized graphics models for interactive, real time graphics systems. Each of the models described here reflects key advancements made by various graphics systems when they were first introduced. I discuss each particular model and its contributions in a separate subsection. The last subsection (Section 4.4) then shows how the requirements proposed in Section 3 fit into the context of these high-level graphics models and how the proposed new model advances the state-of-the-art.

4.1 Retained-Mode Model

Figure 2 illustrates a basic graphics system. I term it the *retained-mode model*, since it represents the retained mode in PHIGS [2] [17]. As shown in the figure, three key components are coordinated: the graphics database, the renderer, and the application.

The graphics database in the retained-mode model is conceptually simple. It is a static database, i.e., it is only capable of storing a "snapshot" of the data, even if the data is time-varying. Furthermore, it only stores data that is directly relevant to the rendering process of the scene, e.g., polygons, appearance attributes, and transformations.

The simplicity of this database model allows a straightforward traversal of the whole database, sending accumulated geometry and attributes to the renderer. If the renderer is a one-pass renderer, e.g., a z-buffer, it can render the data directly. There is no need to substantially process the contents of the graphics database before rendering it.

The application in this model is responsible for coordinating all the components. In particular, it controls when the graphics database is to be rendered, as indicated by the flow of control arrow from the application to the renderer in Figure 2. (The application might base its decision of when to render a frame on supplied external signals, also shown in Figure 2.) More importantly, the application is also responsible for the contents of the graphics database. It executes

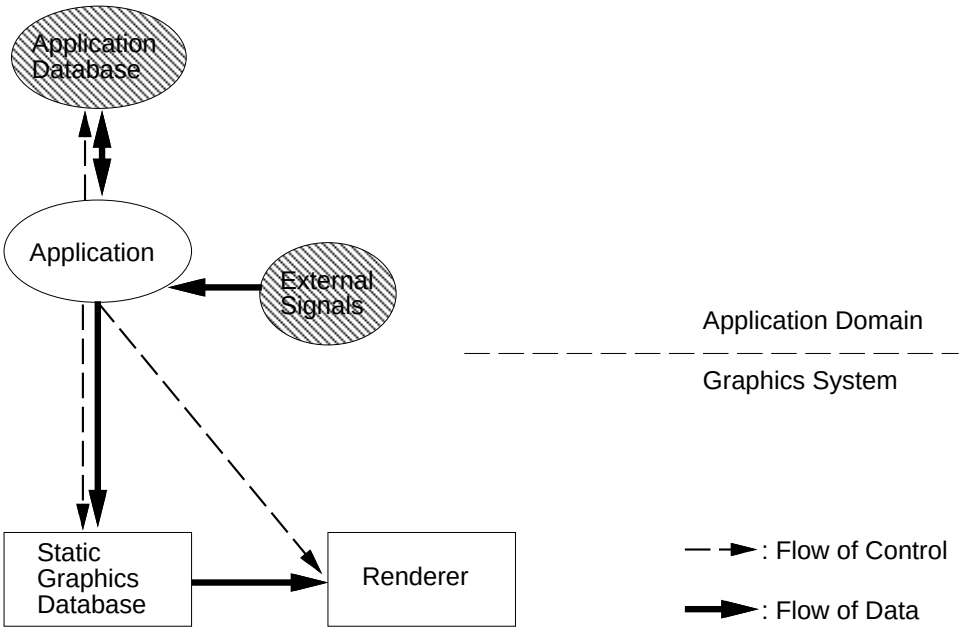

Figure 2: Schematic of the Retained-Mode Model

all changes to the simple, static graphics database; it is "pumping" data into the graphics database.

Thus, the retained-mode model is at a disadvantage, when, for example, generating an animation sequence. First, since the graphics database is static, it is impossible to store the time-varying attributes (i.e., the behavior) of an object in it. Only the application stores the full temporal data for all animated objects in its application database. Therefore, to render the next frame the application needs to transmit a new temporal snapshot to the graphics database. Typically, large parts of the static graphics database are therefore reedited before the next frame is rendered. The application can avoid these successive edit/rerender cycles, if it implements special purpose mechanisms that detect and take advantage of interframe coherency. However, such mechanisms are hard to implement, expensive to execute, and of limited use, since interframe coherency is uncommon in highly dynamic animations. In short, the data-path from the application to the static graphics database is heavily used, resulting in high overhead for the application.

Second, since the simple, static graphics database cannot store all object-related information, the application itself has to maintain and store all objects, their characteristics, and behaviors, in its own application database. Maintaining a separate application database, while in general necessary to accommodate

general applications, also means that the application is responsible for inquiring and extracting information from it, as well as keeping the two databases in synchronization.

Third, after the application downloads all information into the static graphics database, it issues the request to render a frame, causing the graphics database to transfer its contents to the renderer. This transferral is in effect a duplication of the prior data transportation from the application to the static graphics database.

Fourth and last, since the application coordinates all key components, it is coupled closely to the graphics system. This close coupling influences the application programmer, who therefore has to know details about the graphics subsystem when designing the application. The ideal goal of isolating the application programmer from the graphics subsystem is not achieved with the retained-mode model.

4.2 Extending and Abstracting the Graphics Database

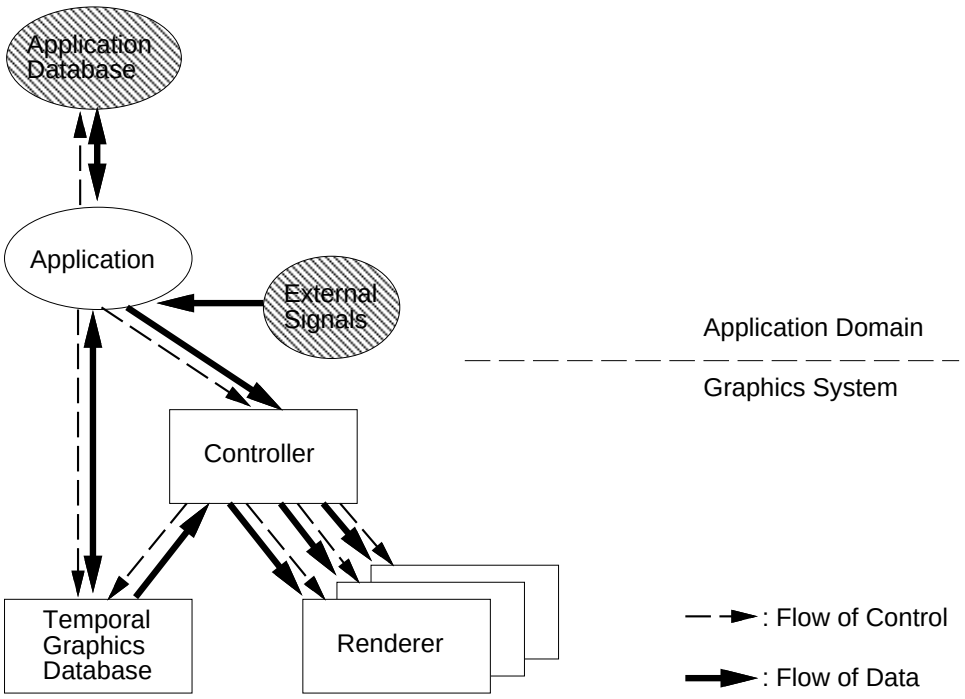

Figure 3: Schematic of the Abstract Database Model

The *abstract database model*, which I introduce here, is more complex than the retained-mode model. An additional component, the controller shown in Figure 3, attests to this complexity. Since this model successfully addresses the shortcomings of the retained-mode model, most of today's graphics systems [34] [42] [22] employ it as their graphics subsystem. As in the previous section, I describe the abstract database model by explaining its key components, i.e., the database, the controller, the renderer, and the application, and their relationship to each other.

The most dramatic change, which necessitates the addition of the controller component, occurs in the structure of the graphics database. The graphics database is extended to be temporal, non-graphical, and abstract. A temporal database stores temporal behaviors of objects. For example, it stores how an object moves over time by recording key-positions and the interpolation methods to be used in between these key-positions. A non-graphical database is a database that stores attributes and data which are not necessarily graphics-related. For example, it might store the mass, salary, or birthday of each object. Finally, an abstract database stores its data in symbolic form, subject to later evaluation or interpretation. An abstract database stores, for example, object shapes as type codes and not as polygon lists.

Extending the graphics database in this way prohibits the use of a simple traversal mechanism as in the retained-mode model. A separate, more intelligent entity, the controller, replaces the traversal mechanism. The controller transforms the temporal, abstract database contents into data that is suitable for renderers. It does so by extracting snapshots from the temporal description, ignoring the non-graphical data or transforming it into graphical information, and evaluating the abstract information into a renderer-specific form. As illustrated in Figure 3, the controller requests information from the graphics database, which returns the inquired data to the controller. Typically, the per inquiry amount of data transmitted to the controller is small (resulting in fast transactions), since the controller only requests particular objects or object attributes as of a particular time. The controller then evaluates the returned data and collects it into a frame data structure to be sent to a renderer.

Note that since the controller has to evaluate the abstract data before it is sent to a renderer, it is possible to support several different renderers without overhead. For example, to render an object via a z-buffer renderer, a ray-tracer (for pick detections), and a symbolic tree drawing (for hierarchy visualizations), the controller first inquires the various relevant attributes from the database.[3] Then, each attribute is evaluated as required by the renderer type: for example, the abstract description of the object "blue sphere," is transformed into triangle strips for the z-buffer renderer, into a mathematical equation for the ray-tracer, and into the text string "Sphere" for the symbolic tree drawing.

[3]The controller can be selective as to what is considered relevant for a given type of renderer, increasing performance because only relevant data is transmitted.

As shown in Figure 3, the renderers receive their data, as well as the command that specifies when to start rendering, from the controller. Since the specifics of interpreting the abstract data are application dependent, the controller component therefore needs to be customizable.

Due to the more complex structure of the graphics database, the application is less coupled to the graphics subsystem when rendering an animation sequence than in the retained-mode model. Once the graphics database is set up, the application merely has to issue a stream of frame requests to the controller. Each of these frame requests also contains information about which temporal snapshot the frame is to represent. The controller then independently requests and computes the data to be send to the renderers.

The abstract database model addresses the disadvantages of the retained-mode model as follows. First, data traffic from the application to the graphics database is reduced substantially. Because the graphics database stores temporal information, the application does not have to reedit the graphics database prior to rendering each frame. Second, database duplication in the application is not needed to the same extent, because the graphics database is capable of storing the application's, graphics related data. The graphics database cannot store all of the application-specific data, unless it is an application-encompassing, general database. (Unfortunately, interactive behavior cannot be stored in the graphics database; I discuss this issue in more detail below.) Third, data transportation is not duplicated, since the controller is inquiring only specific data from the graphics database, evaluates it, and sends renderer-specific data to each renderer. Fourth and last, since the controller is responsible for data-inquiry, -evaluation, and -transportation, the application is less coupled to the graphics subsystem than in the retained-mode model; it is sufficient for the application to request frames from the controller to generate an animation sequence.

However, some problems persist in the abstract database model. For one, the timing of the generated animation is unrelated to wall-time. Frames are usually produced as-fast-as-possible; the timing of an animation therefore relies purely on the performance of the frame-generation process.

Another problem is that interactive behavior is not addressed adequately. Interactive applications are responsible for handling user input themselves, causing the same problems that temporal data caused in the retained-mode model: the need for significant duplication of the database in the application in order to relate interactive application data to data in the graphics database; high data traffic from the application to the graphics database as user interaction necessitates the modification of data; duplication of data transport; and relating the application closely to the graphics subsystem.

4.3 External Signal Awareness

To remedy the problems of the abstract database model, the *external signal model* described here adds the explicit dependency of data on external signals.

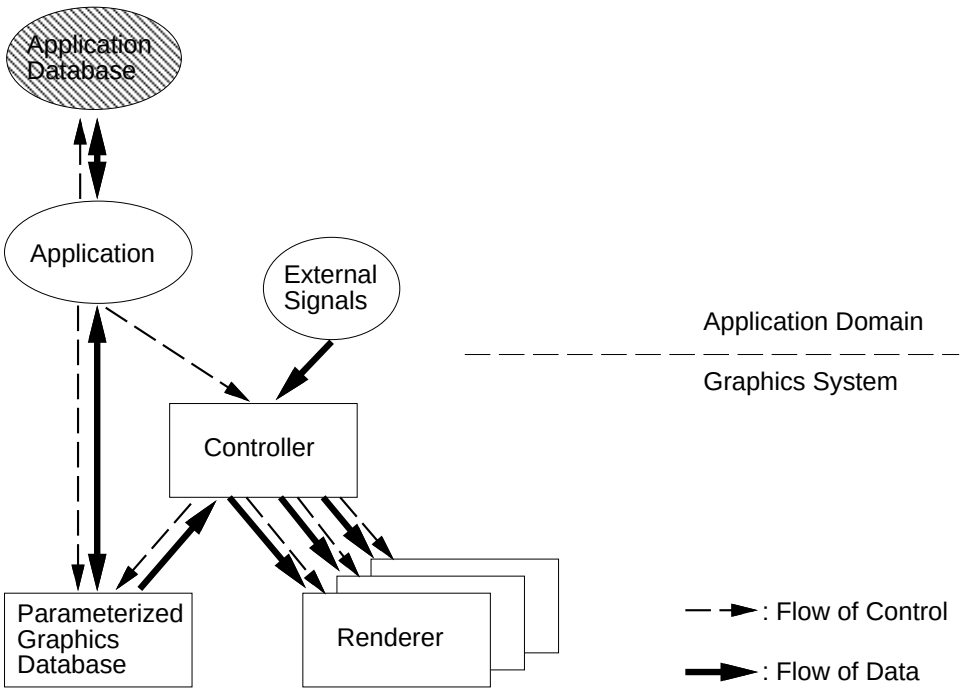

Figure 4: Schematic of the External Signal Model

External signals include wall-time and user input devices, as explained in Section 3.1. Since this model represents the current state-of-the-art, only a few systems currently exist that use it [12] [38] [37] [39] [10][4] [36]. The availability of external signals to the controller, as shown in Figure 4, causes conceptual changes in the roles of the database, the controller, and the application, which I describe in the following. The renderers are unaffected and since they continue to operate as described in the abstract database model, I do not repeat their description here.

The external signal model modifies the graphics database structure to store generally parameterized descriptions of the objects and their attributes, as opposed to descriptions that are parameterized only in the temporal dimension as in the abstract database model. This parameterization allows objects to be explicitly dependent not only on time, but also on other external signals, such as user input devices. Therefore, input parameters need to be specified explicitly. Furthermore, an object can depend on multiple parameters simultaneously.

The controller's main function remains the inquiry and evaluation of the

[4]The system described in [10] evolved from the previous implementation [42] to fit the external signal model.

parameterized, abstract data stored in the graphics database. However, the application no longer supplies the controller with parameter values. Instead, the controller receives these values directly from the external devices (e.g., clock, user input devices), as depicted in Figure 4. The controller therefore not only evaluates the parameterized, abstract data, but is also responsible for supplying the right parameter values to the data. Since the application now only supplies the controller with compute requests, the controller has become more independent.

Overall, the application is further detached from the graphics subsystem than in previous models. Not only is the application's connection to the controller reduced to carry only rendering requests, but the application's connection to the graphics database is simplified as well, since the database is able to store generally parameterized data.

The external signal model addresses interactive behavior by storing behaviors of objects in the database which are explicitly parameterized to depend on user input devices. When the controller evaluates these behaviors, it supplies them with the current, appropriate values as provided by the external signal module [12] [29]. Since the application is not required to intervene during this process, interactive behaviors are efficient.

In general, applications running under the external signal model are farther removed from the specifics of the graphics subsystem, yet, the produced animations are intimately linked to the outer world via the provided external signals. The application-to-graphics-database connection is only used to communicate editing changes (e.g., when the behavior of an existing object changes or objects are added or deleted); in particular, it is not misused to simulate dynamic or interactive behavior.

Two problems remain. First, the external signal model does not explicitly address sampling. To minimize the application-controller connection sampling is therefore uniform, i.e., all objects are updated when a new frame is requested. Uniform sampling and updating for all objects is overly restrictive, as discussed in Section 3.2. Second, as described in Section 3.3, the explicit specification of sampling or update frequencies in addition to the dependence on external signals requires the incorporation of time-critical computing concepts.

4.4 Incorporating Update Rates and Time-Critical Computing

In this section I introduce the *time-critical model* illustrated in Figure 5. The time-critical model results from altering the external signal model so that the previously mentioned problems, uniform sampling and time-critical computing, are addressed. Uniform sampling is replaced by allowing the explicit specification of update rates on an per-object or per-attribute basis. Even though I believe these capabilities to be essential in next generation graphics systems, no graphics architectures I know of incorporate them.

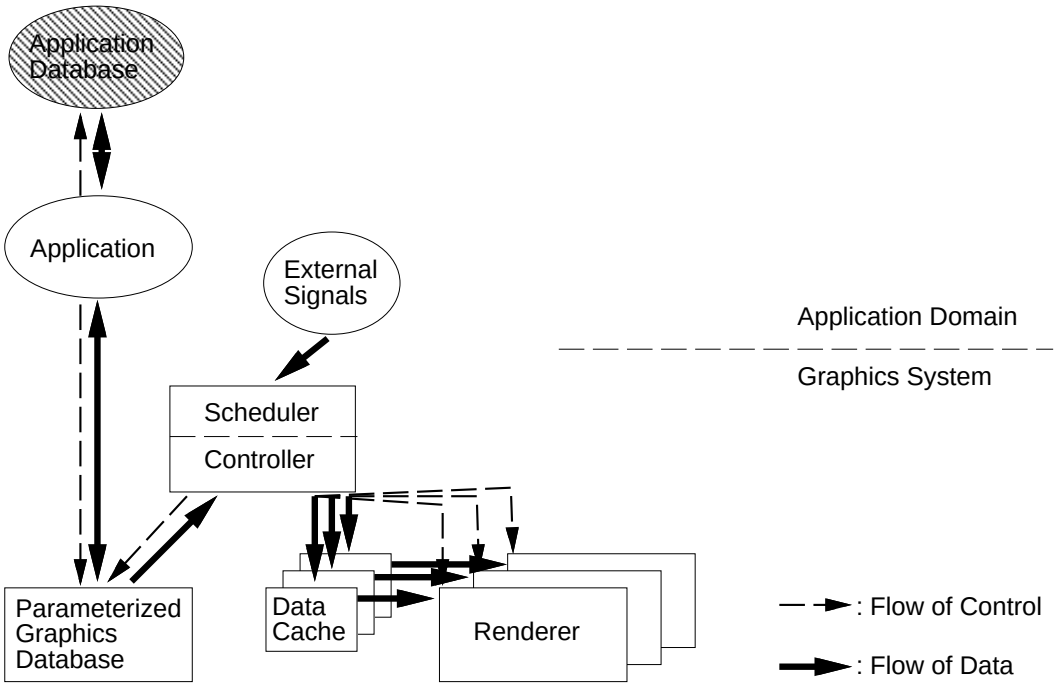

Figure 5: Schematic of the Time-Critical Model

However, such efforts exist in related fields. Several researchers [30] [25] [19] [31] [14] have independently started to address the problem of time-critical rendering. A time-critical renderer is supposed to produce frames in prespecified amounts of time, degrading various render-specific attributes to achieve its goal. Time-critical rendering is a subproblem of the more general area of time-critical computer graphics, since only one component, i.e., the renderer, is time-critical. It is an especially important subproblem, because the renderer is the most visible and the most traditional component in a graphics system. In addition, because for certain applications rendering constitutes the computational bottleneck, it is susceptible to being improved by new techniques such as time-critical computing. It is therefore also susceptible to hardware assistance, which complicates the time-critical rendering problem considerably [25]. I do not address time-critical rendering, since I assume that rendering can be contained to be fast [27] [1] compared to the data inquiry and computation time, e.g., by restraining myself to wire-frame representations.

In contrast, multimedia systems, being based on presampled data types that are inherently wall-time based, are aware of sampling rates and real time display of results [40]. Recent work [18] [23] also addresses issues in degrading

the data stream in order to stay synchronized to wall-time. However, since the involved data types are presampled, thus prespecified and often sampled at regular intervals, there is no need for complex sampling rate models or complex degradation schemes. Compared to 3D graphics systems, such presampled multimedia presentation systems are therefore relatively ad hoc.

The time-critical model for 3D graphics systems requires several changes in the key components (as compared to the external signal model). In the following paragraphs I describe these changes in the graphics database, the controller, the renderer components, and in the application.

The graphics database is extended once more to store each object's update rate in addition to the already stored information. Even though the controller component uses the update rate to decide when to sample objects, there are several reasons to store the update rate in the graphics database. First, the update rate is object-specific (attribute-specific), i.e., it is associated with an object (attribute) and therefore is stored with all the other object-specific (attribute-specific) information. Second, since the application, the graphics subsystem, and the objects themselves control the update rate (see Section 3.2), it is stored where it is accessible to all of the above — the database qualifies, as indicated by Figure 5. Third and last, because the update rate is dynamic, specified abstractly, or in general parameterized (in the sense of Section 4.3), it possibly needs to be evaluated via the same mechanisms as other object data.

The graphics database also needs to be able to store alternative data representations. In effect, data stored in the graphics database is additionally parameterized by one or more "quality" variables. These alternative representations allow for varying degrees of inquiry performance, thereby implementing degradation techniques.

The time-critical model significantly expands the controller. As shown in Figure 5, the controller includes a scheduler that is responsible for deciding when and with what parameter values to sample which objects. The scheduler bases its computation on the update rate of an object, when the sample is due, how complex the inquiry operation is predicted to be, how many other inquiry operations need to be performed in the same time span, and the different possible degradation alternatives. To gather this information, the controller relies on several new internal facilities: the controller inquires the update rate of an object from the database; it computes when the sample is due as a function of the update rate and the object's synchronization to external signals; the scheduler uses a prediction algorithm to assess performance complexity; and it uses a general scheduling algorithm to select degradation techniques and when to start computations.

Note that by incorporating update rates and time-critical computing the controller component has become autonomous in deciding when to inquire samples. Figure 5 expresses the autonomy of the controller by omitting any connections between it and the application. The application has only indirect control over when objects are updated, i.e., only an object's update rate and relation to

external signals govern when the controller inquires data.

The renderer interface is more complex than in the external signal model. Since objects have individual update rates that are potentially different from the frame rate of the renderer, the controller needs to collect and buffer the graphical data of the objects before that data is transmitted to the renderer. Rendering frames and computing object data have become asynchronous operations, requiring a buffer, or data cache in between the controller and the renderer, as shown in Figure 5.[5]

Compared to the previous graphics models, the application in the time-critical model is furthest removed from the graphics subsystem. Specifically, the application is relieved of generating sampling requests. The application describes whole interactive animations to the graphics subsystem, which is able to interpret and execute these descriptions without the application's intervention. The only communication required between the application and the graphics subsystem consists of the application editing the graphics database. The editing operations are true in the sense that they are not misused to achieve other means, e.g., dynamic behavior as in the retained-mode model, or interactive behavior as in the abstract database model. Note that the limited communication between the two components does not constrain the expressive power of the application, since all parameters are specified within the graphics database.

The time-critical model addresses update rates and time-critical computing. Individual update rates as described in Section 3.2 are introduced by allowing their storage in the graphics database. The controller inquires and evaluates the update rates just as any other object data and uses the gathered information to schedule object (or attribute) inquiry operations, honoring the various update rates.[6] The decoupling of frame-rendering rate from the other update rates requires the addition of a data cache to the renderer interface to store the sampled data before it is rendered.

Time-critical computing is incorporated by expanding the controller significantly to include a prediction algorithm, a degradation technique selector, and a scheduler. Aided by additional information stored in the database, these components work together to allow the timely delivery of data as specified by update rates and external signal dependence. The graphics subsystem is powerful enough to be independent of the application.

However, the proposed model has disadvantages. First, the application has to synchronize the application database with the graphics database, resulting in overhead. Second, due to the more general structure of the controller and sched-

[5] Note the similarity between the renderering component in this model and the subconstruct consisting of the controller, the data cache, and the renderer in the retained-mode model (Section 4.1). The "wheel of reincarnation," common in graphics hardware [28], is introduced to the graphics subsystem, a software architecture.

[6] In case that the application modifies the update rate of an object, it is necessary to notify the controller of the change as well. Otherwise, if the new update frequency is higher than the old one, the controller, basing its schedule on old information, misses this change and consequently does not inquire with the required frequency.

uler, these components are more complex to implement. Third, computation time is spent predicting and scheduling. It follows that, since the prediction and scheduling always produce overhead, the proposed graphics system is not the fastest possible. Fourth and last, the design of the controller component needs to be application-customizable to be able to interpret arbitrary application specific data.

5 Conclusion

5.1 Summary

Applications in or related to interactive 3D graphics, multimedia, and virtual reality are becoming increasingly widespread. These applications share common requirements: they need to synchronize data to real world signals, most notably wall-time and user input; they require low-latency feedback, and they are highly interactive. While these applications often request complex computations, the produced quality is not as important as the on-time delivery of results. The widely used batch-job computing paradigm is ill-suited to attend to those demands.

Instead, I propose to implement a time-critical computing paradigm to address the requirements of interactive 3D graphics, multimedia, and virtual reality. My approach is to balance the various quality parameters of a computation, paying particular attention to the execution times of the various tasks.

In particular, I concentrate on the following three issues. First, I address the requirement of synchronizing data to the real-world. I generalize wall-time, user input, and other signals into an external signal module that is easily integrated into graphics systems. Second, I allow the explicit specification of update rates for dynamic behaviors. Various behaviors require different update rates; objects are sampled when needed, not every time a frame is generated. The saved computation time is spent to update critical objects more frequently. Furthermore, update rate is a generally applicable degradation parameter that controls overall computation cost. Third and finally, I integrate time-critical computing concepts to guarantee that computations finish on time, thereby eliminating excessive lag. Predicting, degrading, and scheduling computations are the main building blocks to ensure interactive, low-latency feedback.

5.2 Expected Result

The overall result of this research will be a 3D graphics system that incorporates the three characteristics mentioned above. I present a rough outline of such a system in Section 4.4. My system will be the first to explicitly specify various update rates and to apply time-critical computing concepts to the computation of data within the 3D model. It will provide evidence of the validity of my

approach to the general problem area of highly interactive, real time 3D graphics systems.

5.3 Approach

So far, I completed the literature survey, and I addressed problems associated with incorporating update rates into a graphics system [41]. I have started to design the envisioned 3D graphics system; the next step consists of programming these preliminary results into a working prototype system. Concurrently, I will investigate the required prediction, degradation, and scheduling components of such a system. After all these components are integrated with the system, I will extend the system to incorporate one multimedia data type, e.g., video, to validate my claims.

Acknowledgements

I am grateful to my advisor Andy van Dam, the members of my thesis committee, John "Spike" Hughes and Randy Pausch, as well as my colleagues, most notably Robert C. Zeleznik, for the many discussions on this work. Of course, none of this could have been possible without the continuing support from our sponsors. This work was supported in part by an IBM fellowship, the NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization, by ONR Contract N00014-91-J-4052, ARPA Order 8225, and also by IBM, NCR, Sun Microsystems, Hewlett Packard, Digital Equipment Corporation, and NASA.

References

[1] Kurt Akeley. RealityEngine graphics. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 109–116, August 1993.

[2] ANSI (American National Standards Institute). American national standard for information processing systems – programmer's hierarchical interactive graphics system (PHIGS) functional description, archive file format, clear-text encoding of archive file. Technical Report ANSI, X3.144-1988, ANSI, New York, 1988.

[3] Michael Bajura, Henry Fuchs, and Ryutarou Ohbuchi. Merging virtual objects with the real world: Seeing ultrasound imagery within the patient. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):203–210, July 1992.

[4] Chuck Blanchard, Scott Burgess, Young Harvill, Jaron Lanier, Ann Lasko, Mark Oberman, and Michael Teitel. Reality built for two: A virtual reality tool. volume 24, pages 35–36, March 1990.

[5] Frederick P. Brooks, Jr. Grasping reality through illusion – interactive graphics serving science. In *Human Factors in Computing Systems*, pages 1–11, May 1988. Special Issue of the SIGCHI Bulletin.

[6] Frederick P. Brooks Jr. Walkthrough — a dynamic graphics system for simulating virtual buildings. pages 9–21, October 1986.

[7] Samuel J. Cardman. Time management in a real-time animation/graphics environment. *Computer Graphics (SIGGRAPH '75 Proceedings)*, 9(1):201–207, June 1975.

[8] Michael Cohen, Shenchang Eric Chen, John R. Wallace, and Donald P. Greenberg. A progressive refinement approach to fast radiosity image generation. 22(4):75–84, August 1988.

[9] Michael Cohen and Donald P. Greenberg. The hemi-cube: a radiosity solution for complex environments. 19(3):31–40, August 1985.

[10] D. Brookshire Conner, Scott S. Snibbe, Kenneth P. Herndon, Daniel C. Robbins, Robert C. Zeleznik, and Andries van Dam. Three-dimensional widgets. *Computer Graphics (1992 Symposium on Interactive 3D Graphics)*, 25(2):183–188, March 1992.

[11] Conal Elliott, Greg Schechter, Salim Abi-Ezzi, and Michael Deering. TBAG: Time, behavior, and geometry. Available from the authors upon request, 1991.

[12] Conal Elliott, Greg Schechter, Ricky Yeung, and Salim Abi-Ezzi. A system for interactive, animated 3D graphics based on continuous high level constraints. Available from the authors upon request, January 1993.

[13] Steven Feiner, Blair MacIntyre, and Doree Seligmann. Annotating the real world with knowledge-based graphics on a see-through head-mounted display. pages 78–85, May 1992.

[14] Thomas A. Funkhouser and Carlo H. Sequin. Adaptive display algorithm for interactive frame rates during visiualization of complex virtual environments. *Computer Graphics (SIGGRAPH '93 Proceedings)*, pages 247–254, August 1993.

[15] Thomas A. Funkhouser, Carlo H. Sequin, and Seth J. Teller. Management of large amounts of data in interactive building walkthroughs. *Computer Graphics (1992 Symposium on Interactive 3D Graphics)*, 25(2):11–20, March 1992.

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

[17] Tom Gaskins. *PHIGS Programming Manual.* O'Reilly & Associates, Inc., 1992.

[18] Simon Gibbs. Composite multimedia and active objects. In *Proceedings of OOPSLA '91*, pages 97–112, 1991.

[19] Rich Gossweiler. Time-critical rendering in an immersive virtual environment. Thesis Proposal, available from author on request, 1993.

[20] Mark Green. A survey of three dialogue models. *ACM Transactions on Graphics*, 5(3):244–275, 1986.

[21] Lawrence J. Hettinger, Kevin S. Berbaum, and Robert S. Kennedy. Vection and simulator sickness. *Military Psychology*, 2(3):171–181, 1990.

[22] Philip M. Hubbard, Matthias M. Wloka, and Robert C. Zeleznik. UGA: A unified graphics architecture. Technical Report CS-91-30, Brown University, Providence, RI, 1991.

[23] IMA Services Focus Group and Architectural Technical Working Group. Request for technology: Multimedia system services. Available via anonymous ftp from world.std.com:/pub/IMA, December 1992.

[24] J. T. Kajiya. The rendering equation. *Computer Graphics (SIGGRAPH '86 Proceedings)*, 20(4):143–150, August 1986.

[25] George Kyriazis. Time-critical rendering and the temporal behavior of graphics systems. Master's thesis, Rensselaer Polytechnic Institute, VTP Program, Rensselaer Design Research Center, May 1993.

[26] L. Edwin McKenzie, Jr. and Richard T. Snodgrass. Evaluation of relational algebras incorporating the time dimension in databases. *ACM Computing Surveys*, 23(4):501–543, December 1991.

[27] Steven Molnar, John Eyles, and John Poulton. Pixelflow: High-speed rendering using image composition. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):231–240, July 1992.

[28] T. Myer and I. Sutherland. On the design of display processors. *Communications of the ACM*, 11(6):410–414, June 1968.

[29] Brad A. Myers, Dario A. Guise, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. GARNET comprehensive support for graphical, highly interactive user interfaces. *IEEE COMPUTER magazine*, 23(11):71–85, November 1990.

[30] Robert O'Bara. Time rendering research. Draft of a thesis proposal, available from author on request, 1992.

[31] Randy Pausch, Matthew Conway, Robert DeLine, Rich Gossweiler, Steve Miale, Jonathan Ashton, and Richard Stoakley. An interdisciplinary approach to building virtual reality environments. 1992.

[32] Randy Pausch, Tom Crea, and Matthew Conway. A literature survey for virtual environments: Military flight simulator visual systems and simulator sickness. *Presence: Teleoperators and Virtual Environments*, 1(3), January 1993. In press.

[33] Chris Shaw, Jiandong Liang, Mark Green, and Yunqi Sun. The decoupled simulation model for virtual reality systems. *Proceedings of CHI'92*, pages 321–328, May 1992.

[34] SoftImage Inc. *SoftImage Manual.*

[35] John A. Stankovic and Krithi Ramamritham. *Advances in Real-Time Systems.* IEEE Computer Society Press, 1993.

[36] Paul S. Strauss and Rikk Carey. An object-oriented 3D graphics toolkit. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):341–349, July 1992.

[37] Mark A. Tarlton and P. Nong Tarlton. Visualization substrate: Animation, simulation and interaction in the user-interface. Technical Report ACT-HI-270-91(Q), Microelectronics and Computer Technology Corporation (MCC), 1991.

[38] Mark A. Tarlton and P. Nong Tarlton. A framework for dynamic visual applications. *Computer Graphics (1992 Symposium on Interactive 3D Graphics)*, 25(2):161–164, March 1992.

[39] Mark Alan Tarlton. *A Declarative Representation System for Dynamic Visualization.* PhD thesis, University of Texas at Austin, August 1993.

[40] D. Tsichritzis, editor. *Visual Objects.* Centre Universitaire d'Informatique, Université de Genève, June 1993.

[41] Matthias M. Wloka. Incorporating update rates into today's graphics systems. In preparation; available upon request from the author, November 1993.

[42] 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.