# Incorporating Update Rates into Today's Graphics Systems

Matthias M. Wloka

Department of Computer Science
Brown University
Providence, Rhode Island 02912

**CS-93-56**
December 1993

# Incorporating Update Rates into Today's Graphics Systems

Matthias M. Wloka
CS Department
Brown University
Providence, RI 02912

December 30, 1993

## 1 Introduction

Personal computing is expanding to include the latest developments in 3D graphics, multimedia, and even virtual reality. To cope with the resulting increased performance demands on the supporting hardware, new technologies, e.g., time-critical computing, need to be explored.

### 1.1 Time Critical Computing Systems

Our ultimate goal is to design and implement a graphics system that is time-critical, i.e., that trades quality for computation speed, thus allowing us to adhere to implicit and explicit timing information [23]. Such a system moves away from the viewpoint prevalent in the graphics community of "it takes as long as it takes to generate the next frame" towards a more user-friendly philosophy of guaranteeing certain response times [22] [5], heightening the user's perception of reality in dynamic, computer-generated scenes by making it act real [4]. Note that while this feature is merely "nice" for conventional desktop environments, it becomes essential for immersive environments to sustain the illusion of immersion and even avoid motion sickness [8] [16] due to lagging feedback.

We imagine our time-critical graphics system to also incorporate multimedia data. Most multimedia data, for example, live or prerecorded audio and video streams, is inherently dynamic and sensitive to timing and synchronization issues. Because of this sensitivity to timing and synchronization, it often is judicious to gracefully degrade the quality of the multimedia stream, rather than to delay the information. Therefore, multimedia fits well into the above outlined time-critical system.

## 1.2 Update Rates

An integral part of such a time-critical system is the specification of update rates. The update rate of a dynamic object is the rate with which its representation is recomputed. Update rate is equivalent to sampling rate, yet, the term sampling rate is preferred in connection with multimedia applications. We use the terms update rate and sampling rate interchangeably. Frame rate, on the other hand, is the update rate of the renderer, i.e., the frequency with which the renderer generates new frames.

Update rates are common to both multimedia and computer graphics. Most multimedia data is presampled and therefore has an implicit sampling or update rate associated with it. Computer graphics, while based on continuous time models [24] [3], is ultimately sampled, for example, when displaying discrete frames. 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. Within a combined multimedia and 3D graphics system, update rate thus is one possible quality control parameter. It is a very powerful parameter, since, for example, skipping the evaluation of objects or skipping a whole frame saves substantial amounts of compute power.

In order to make this power accessible, update rate specification must be an explicit concept. Both the system, as well as the end-user need to control update rate. For example, the system needs to automatically degrade update rates, if the given time constraints require it. Furthermore, presampled data types command prespecified update rates. The end-user needs to delimit the possible range of automatic degradation and also needs to explicitly specify update rates for data defined continuously over time.

For maximum flexibility in fine-tuning degradation effects update rates are specified on an per-object or even on an per-attribute level. This fine-grained specification contrasts with the currently common methodology of equating all update rates with the frame rate. Updating objects slower than frame rate is desirable, for example, if the particular object is slow-moving, in the background, or otherwise visually unimportant, thereby enabling us to save computation time. Similarly, updating objects faster than frame rate is required for objects that are controlled by error-sensitive numerical simulations or by interaction (i.e., cursor-type) objects that are rendered separately (i.e., as sprites or sprite-like objects), allowing for interactive, minimal lag user feedback.

## 1.3 An Ideal System

We concede that the need to specify update rates feels unnatural to the computer graphics user (especially when using a continuous time model). A user would rather specify update rates indirectly, for example, by stating that an object is

resampled whenever its screen representation has changed "noticeably."

Several definitions of "noticeable" change are possible. To calibrate the error measurements, we designate the correct frame to be the one that is generated by updating all objects with a rate equal to the frame rate. Hence, one possible definition of a "noticeable" change is if the produced frame differs from the correct frame when compared pixel by pixel. A less stringent definition compares the rendered object representations: only if an object's rendering differs in more than 10% of the total number of the object's pixels, then it is considered noticeably different.

While an ideal system would automatically determine the individual update rates so that no difference is noticeable, relieving the user entirely of specifying update rates poses additional requirements. Such an ideal system must then also determine how to degrade those update rates. Degradation should be based on rate of change of object and attributes, size of object, object-viewer distance, and possibly other measurements of object importance. The system has to decide all these issues in real time.

We therefore believe that such an ideal system is currently infeasible, yet a system that lets the user manually specify update rates can already address many of the fundamental issues related to update rate. In particular, in this paper we discuss how update rate itself is specified, and how this specification integrates with the rest of the graphics system. We view this work as a required, first step towards the ultimate goal of time-critical graphics systems that incorporate multimedia.

### 1.4 Overview

In the rest of the paper, we describe in detail the various parts of such an update rate model and how it is integrated with today's graphics systems. In Section 2, we analyze related work and offer a general view of today's graphics systems, on which we base the following discussion. Then, in Section 3, we describe how to integrate update rates with the generalized graphics system previously described, addressing issues such as relating and inheriting update rates when specified on an per-attribute level. In Section 4, we introduce and discuss various possible update rate specification models. Finally, Section 5 concludes the paper with a summary of the described work and suggestions of future work.

## 2 Today's Graphics Systems

### 2.1 Related Work

Today's graphics systems are just starting to address the problems associated with modeling dynamic behavior. So far, the main thrust of these efforts has

introduced successful concepts in the space domain to the time domain. Examples of these concepts include encapsulation, hierarchy, and non-linear relationships. Dynamic behaviors are encapsulated into reusable building blocks, e.g., as "time-dependent behaviors" [20] or "time graphs" [11]. Hierarchical coordinate systems, common in the space domain, are applied to time, e.g., as "time coordinate systems" [20] [21]. Finally, non-linear relationships over time link separate time coordinate systems, a procedure that is commonly called "time warping" [19] [2]. More recently, the concept of linking logical (modeling) time to real-time [1] has been rediscovered [3], allowing dynamic behavior to be synchronized to wall-time.

None of the contemporary computer graphics systems we know of [24] [3] [21] [10] [15] [18] [5] handle update rate as an explicit concept. Accordingly, they cannot benefit from the advantages mentioned in Section 1. In particular, since update rate is implicit to the system, the user cannot control it. The system decides when to render the next frame, and thus the next frame is rendered as soon as the previous frame is finished ("as fast as possible mode") — only processing power and scene complexity determine frame rate.

Furthermore, because update rate is implicit, it is automatically set equal to the frame rate. (The work described in [15] is an exception; the authors distinguish two update rates — animation update rate and rendering frame rate — which are both operated in "as fast as possible mode." We discuss [15] in more detail below.) Therefore, efficiency is lost due to the uniform update rate for all objects, i.e., degrading individual update rates is impossible. Various systems attempt to regain this lost efficiency, for example, via elaborate and costly caching schemes [24] or via complicated code analysis and compilation [3]. Worse, updating faster than frame rate is impossible [24] or done outside the usual evaluation process [3] [18].

While Inventor [18] does not address dynamic behavior per se, it is interesting that its builtin support for timers is potentially used for simulating dynamic behavior and even sampling. Timers are autonomous objects that are directly linked to wall-time. Therefore, they can be set up such that they change objects and attributes over time. Updating objects and attributes in this discrete fashion is the equivalent of sampling. However, since all of the dynamic behavior support has to reside within the timer objects, the user alone is responsible for modeling dynamic behavior, without support from Inventor.

The Alice and DIVER system [15], driven by the exacting processing needs of virtual reality, divorces animation frame rate from rendering frame rate. While a step in the right direction, the authors fail to recognize the full potential of being able to specify and control a multitude of update rates.

Most recently, several researchers [13] [12] [5] [7] have independently started to address the problem of time-critical rendering. A time-critical renderer produces frames in prespecified amounts of time, degrading various renderer-specific attributes to achieve its goal. In effect, the user specifies the update rate for the renderer in relation to wall-time. Since the user controls exactly one update

rate, i.e., the rendering frame rate, time-critical rendering is a subproblem of the more general area of time-critical graphics we are addressing. It is an especially important subproblem, since it addresses the most visible and the most traditional component in a graphics system, namely, the renderer. In addition, because for certain applications rendering constitutes the computational bottleneck, it is susceptible to be improved by new techniques such as time-critical computing. However, it is therefore also susceptible to hardware assistance, which complicates the time-critical rendering problem considerably [12].

In contrast, multimedia systems, being based on presampled data types that are inherently wall-time based, are aware of update rates. Recent work [6] [9] also addresses issues in controlling jitter and degrading the data stream in order to stay synchronized. However, since the involved data types are presampled, thus prespecified and often sampled at periodic intervals, there is no need for complex update rate models. Jitter control and degradation strategies are accordingly ad hoc; relating update (sampling) rates to influence each other is inconceivable.

Since we want to integrate synthetic and presampled data types, our update rate model is more complex to allow for general, dynamic, and degradable update rates. In our model, the user explicitly specifies update rates and associated, allowed jitter.

## 2.2 Generalized View of Graphics Systems

In this section, we outline a generalized view of present graphics systems. This general description provides us with a framework that points out assumptions we rely on and within which we discuss our update rate model. We prefer this general discussion to a system-specific one, because we hope that our update rate model is general enough to be applied to a variety of graphics systems; if a graphics system fits the generalized view, it can incorporate the update rate model. The following partitioning is derived from Robert Zeleznik's contribution to [14].

In our view, a graphics system consists of several key modules. Each module encapsulates basic functionality and is self-contained to the extent of being interchangeable with another module that provides similar functionality. The functionality and therefore complexity of each module is chosen depending on the application specific problem a system wants to solve. For example, a system geared towards producing static, high-quality renderings of synthetic scenes only needs a very simple (or even non-existing) dynamics module. We believe a modern, interactive real-time graphics system needs to contain the following modules: objects, sharing, dynamics, and glue.

The *objects module* provides the functionality that has always been associated with graphics systems. It allows a user to describe synthetic scenes, often consisting of geometric shapes, lights, and cameras and all their associated attributes and operators. Modern systems are often designed to easily expand

this module to include new shapes or attributes [24] [3], effectively allowing the interchange of the present module with one that has a richer set of primitives.

As pointed out in [14], *sharing* is practiced on many different levels. Sharing simplifies and automates the user's task of designing and modeling a coherent, synthetic scene. It alleviates the user of making mechanical changes to the scene, e.g., all instances of an object are automatically updated when the master instance is changed, and allows the user to incorporate structural, higher level information, e.g., the user expresses that two objects always share the same color. The most common forms of sharing in graphics systems are inheritance and global data propagation networks. To simplify the graphics system's structure, it is advantageous to support all types of sharing via the same underlying mechanism, as done in [24], where both hierarchical inheritance and global data propagation are implemented via the same one-way dependencies. It is conceivable to interchange such a one-way dependency module with a more powerful sharing module, e.g., a constraint solver such as SkyBlue [17]. We foresee that future graphics systems employ sharing modules that are based on optimizing constraint solvers and possibly more sophisticated mechanisms.

The *dynamics module* allows the user to specify how the described synthetic scene changes over time, i.e., how objects, attributes, their relations, and their constraints vary dynamicly. This module encapsulates the model of time used, how the model relates to wall-time, and how update rates are specified. The update rate specification we propose in this paper assumes a continuous time model: the synthetic scene can be sampled at arbitrary points in time. Note that computer graphics systems prefer continuous time models even when concentrating on modeling discontinuous, event-based behaviors [11], due to the synthetic nature of computer graphics. Note also that we wish to maintain the continuous view of time; the introduction of temporal sampling is a necessary post-process that is executed on top of the continuous time model.

The *glue module* has little to do with computer graphics per se and is only mentioned to complete the description of general graphics systems. The glue module supplies the language that lets the user access all the functionality of the other modules consistently and coherently. To some extent, it is also responsible for integrating the functionality of the various modules. Graphics system's implementations for this module range from home-brew languages, e.g. FLESH [24], to commercially available, high-level languages, e.g. Scheme [3]. Often, these languages are interpreted, and not compiled, to improve the user's ability to quickly prototype and view new geometric models and their behavior.

All these modules must be integrated as equals to obtain maximum functionality. For example, constraints between objects should be allowed to vary over time, just as constraints should be allowed to operate on object behaviors [3]. Similarly, module functionality should be applicable to all (and not only some) primitives of other modules, e.g., dynamics should be applicable to all object attributes, including object type [24].

# 3 How to Incorporate Update Rates in Today's Graphics Systems

In this section we discuss how to incorporate update rates into graphics systems, namely, for which parts of the system update rates are specified (i.e., which incorporation model to choose), what operations on update rates are useful to ease specification and sharing of update rates, and possible update rate defaults derived from the structure of a scene. Naturally, we base our discussion on the general graphics system structure described in Section 2.2.

## 3.1 Incorporation Models

Before we specify update rates in a graphics system, we have to decide with which component to associate the update rate. We associate update rate with each object.[1] This decision is justified, because the objects in a graphics system are the entities that perform the actual work of transforming user-specified parameters into, ultimately, the desired sequence of 2D bitmaps (see also Figure 1).

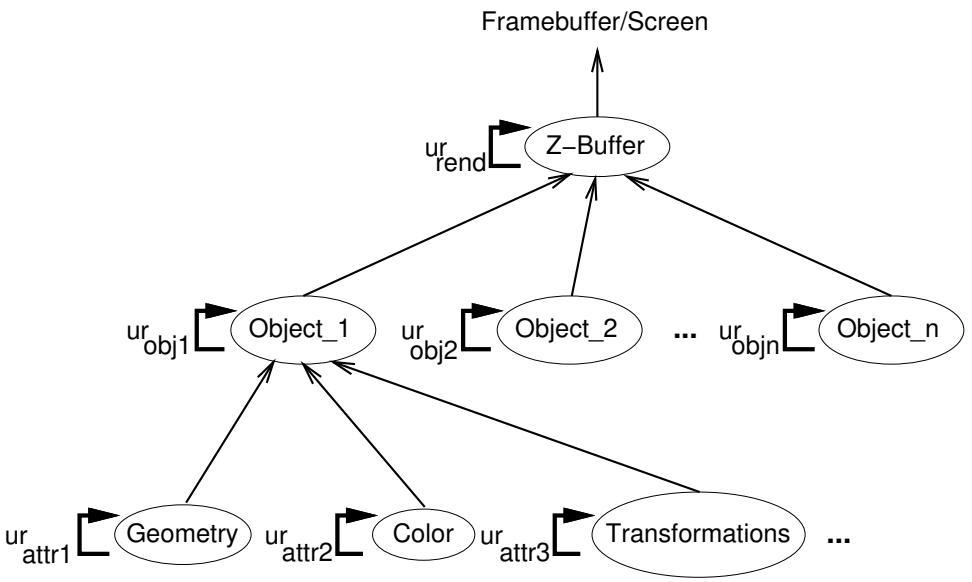


Figure 1: Objects Transform Abstract Specifications into 2D Images

Instead of specifying update rate per-object (i.e., via the object module), it

[1] Here and throughout, we use object as defined in the description of the object module in Section 2.2, i.e., graphics primitives, for example, graphical objects, attributes, and operations.

is possible to associate update rate with each sharing link (i.e., via the sharing module). Whenever two objects share information, the resulting link between these two objects is subject to an update rate. This incorporation model is useful, since it reflects that update rates are determined by interobject relations, and not only by internal object parameters. The model acknowledges that objects are updated for a purpose, i.e., to provide data to its dependent objects. (An object that has no dependents does not have to be updated.)

A simple example illustrates the usefulness of the per-link incorporation model. Assume that two users view a complex scene that is viewed by two users. Each user is independent of the other, and controls their own camera and renderer. Because the scene is complex, various parameters are degraded, in particular, update rates differ for various objects. A specific object, call it object $A$, is visible to both users. Object $A$ is far away from the first user, but relatively close to the second user. Therefore, object $A$ is unimportant to the first viewer, but important to the second user. Accordingly, the required update rate for object $A$ is low for the first user, but high for the second user. The link between object $A$ and the first user's renderer should associate a low update rate with it, while the link between object $A$ and the second user's renderer should have a high update rate assigned to it.

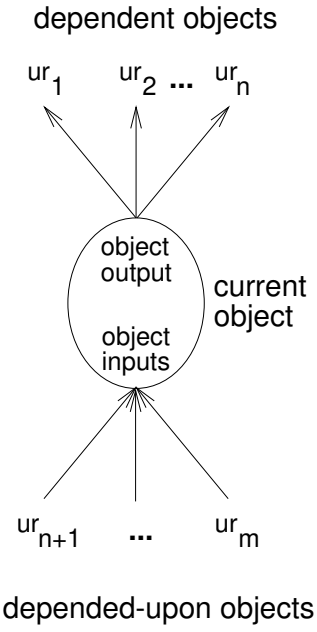


Figure 2: The Various Update Rates Associated with an Object When Transforming the Per-Sharing-Link Model To the Per-Object Model

Both incorporation models — associating update rates per-object and per-sharing-link — are valid. While the per-sharing-link model is probably easier to grasp for users, it eventually has to be converted to the per-object model: after all, update rates control and steer the overall amount of computation, yet, all computations are performed by the individual objects. Therefore, update rates are ultimately associated with each object. It follows that all incorporation models need to convert their specification to a per-object update rate. For example, the per-sharing-link model has to transform the update rates specified on each sharing link to update rates for each object.

To transform the per-sharing-link model to the per-object model, we assign an update rate to each object that is a combination of the update rates specified on the links attached to the object. Refer to Figure 2 for the various update rates associated with an object. In particular, one possible method of converting the per-sharing-link model to the per-object model consists of assigning an object's update rate to be the union of all link update rates:

$$ur_{obj} = ur_1 \cup ur_2 \cup \cdots \cup ur_m$$

## 3.2 Operators

In general, to transform any incorporation model into the per-object model we require a multitude of operators that transform and combine update rates. Furthermore, the user also needs such operators when specifying the update rates, i.e., to specify update rates in relation to each other. For example, it makes sense to specify that two objects share the same update rate. To enrich the possible relations between update rates — beyond simple equality relations — we list a number of useful operators in Table 1. The exact semantics of these operators depend on the underlying specification model, e.g., simple, jitter-interval, or weighted jitter-interval (see Section 4). To keep the discussion generally applicable, we therefore describe the associated semantics only informally.

## 3.3 Defaults

While these operators ease the user's task of specifying meaningful update rates, they do not help to reduce the sheer quantity of update rates to be specified in an average scene; depending on the incorporation model, every object or every dependency link requires the assignment of an update rate. The usability of an incorporation model is inversely related to the number of update rates to be specified. The less update rates the user has to specify, while maintaining maximum generality and control, the better the incorporation model. Sensible defaults can greatly reduce the number of update rates to be specified.

We can divide objects directly related to the current object into dependent objects and depended-upon objects. Dependent objects are objects that make use of the data computed by the current object. Depended-upon objects are

| **Operator** | **Notation** | **Semantics** |
|---|---|---|
| Union of $ur_a$ and $ur_b$ | $ur_a \cup ur_b$ | *Update when $ur_a$ or $ur_b$ updates; if $ur_a$ and $ur_b$ update simultaneously, then update only once.* |
| Intersection of $ur_a$ and $ur_b$ | $ur_a \cap ur_b$ | *Update only when $ur_a$ and $ur_b$ update simultaneously.* |
| Difference of $ur_a$ and $ur_b$ | $ur_a - ur_b$ | *Update only when $ur_a$ updates, but not when $ur_b$ updates simultaneously with $ur_a$.* |
| Scalar multiply $ur_a$ by $c$ | $c \cdot ur_a$ | *Multiply the time associated with each update of $ur_a$ by $c$.* |
| Degrade $ur_a$ by $\frac{d}{f}$ | $ur_a\lfloor_{\frac{d}{f}}$ | *Drop $d$ samples over a period of $f$ samples from $ur_a$ (see Section 4.4).* |
| Restrict $ur_a$ to $ur_b$ | $\lceil ur_a \rceil^{ur_b}$ | *Update when $ur_a$ updates; if $ur_a$ updates more than once in-between two adjacent updates of $ur_b$, then only the first of these updates is used (see Section 3.3).* |
| Assist $ur_a$ with $ur_b$ | $\lfloor ur_a \rfloor_{ur_b}$ | *Update when $ur_a$ updates; if $ur_a$ updates less than once in-between two adjacent updates of $ur_b$, then the first update of this pair of $ur_b$'s updates is also used (see Section 3.3).* |

Table 1: List of Possible Update Rate Operators and Their Informal Semantics

objects whose output is needed to compute the current object. If we interpret the update rates $ur_1, \ldots, ur_m$ in Figure 2 to be associated with the dependent and depended-upon objects and not with the links, then we see that $ur_1, \ldots, ur_n$ are the update rates of the dependent objects and that $ur_{n+1}, \ldots, ur_m$ are the update rates of the depended-upon objects.

A sensible default is context-sensitive. For update rates, a default is computed from the update rates of the dependent objects, the depended-upon objects, or a combination of the two. The following discussion is based on the per-object incorporation model, but is equally applicable to the per-sharing-link model.

If none of the dependent update rates are known, a default $ur_{obj}$ needs to be a combination of the known depended-upon update rates. Choosing $ur_{obj}$ to default to the union of all known depended-upon update rates amounts to an implementation of a data-flow model: the update of any object is immediately percolated to the top, i.e., the renderer, of a scene. This default behavior is highly inefficient, yet it avoids *time disparity*.

We define time disparity as follows. Since various objects and attributes have potentially different update rates, the provided information may represent different times. Therefore, a rendered frame shows several objects at disparate points in time. We call the concurrent display of time-disparate objects time disparity.

We achieve a more efficient default, if we are willing to allow for time disparity. If the known depended-upon update rates are $ur_1, ur_2, \ldots, ur_n$, then we combine these update rates with the assist operator:

$$ur_{obj} = \lfloor \ldots \lfloor ur_1 \rfloor_{ur_2} \ldots \rfloor_{ur_n}$$

The resulting update rate $ur_{obj}$ samples as often as the fastest update rate, but does not do so independently for each depended-upon object update rate.

Analogously, if none of the depended-upon update rates are known, a default $ur_{obj}$ needs to be a combination of the known dependent update rates. As before, if we choose union as the combining operator, the resulting behavior corresponds to a non-time-disparate, inefficient evaluation model. Data flows as if demand-driven: objects update whenever dependent objects demand new data. As before, we use the assist operator to combine the known update rates and therefore derive a default $ur_{obj}$ that is time-disparate, but more efficient.

If both dependent and depended-upon update rates are known, then the default for $ur_{obj}$ needs to reflect the added amount of available information. First, using any of the above described methods, the dependent update rates are combined into a single output update rate, $ur_{out}$. Second, using any of the above described methods, all depended-upon update rates are combined into a single output update rate, $ur_{in}$. Third and last, these two update rates are combined to produce $ur_{obj}$. Choosing union as the combining operator yields an accurate, but unnecessarily inefficient update rate $ur_{obj}$. We observe that updating the current object faster then $ur_{out}$ has no visible effect, since the output of the current object is not distributed faster than $ur_{out}$. Similarly, updating the current object faster than $ur_{in}$ is also wasteful, because recomputing the current object when none of the depended-upon objects have changed generates the

same object. Therefore, we use the restrict operator to assign a value to $ur_{obj}$:

$$ur_{obj} = \lceil ur_{out} \rceil^{ur_{in}}$$

or possibly

$$ur_{obj} = \lceil ur_{in} \rceil^{ur_{out}}$$

The restrict operator ensures that the current object is not unnecessarily recomputed.

Finally, we examine the case that non of the directly related objects have associated update rates. We assume that the user has specified the update rate for at least one object within the scene. Starting from that object we compute defaults for all the related objects. We iterate the process to derive update rates for all objects in the scene. Since it is conceivable that two objects are linked via several dependent/depended-upon relation paths, the resulting default update rate for each object depends not only on the chosen operators to combine defaults, but also on the execution order of the iteration.

# 4 Update Rate Specification

Update rates are specified in a variety of formats, various formats satisfying different needs. We are interested in update rates in the context of sampled, time-critical graphics systems such as described in Section 1. At a minimum, the specification of update rates therefore needs to allow for update rates that are arbitrary, dynamic, and degradable by the controlling system.

We describe three such update rate specification models in this section. We begin by introducing a simple model in Section 4.1. Then in Section 4.2 we extend this simple model to allow inexactness in the specification of update rates. The resulting jitter-interval model thereby introduces jitter to the specified update rate. The weighted jitter-interval model, described in Section 4.3, enhances the jitter-interval model by weighting the importance when to sample within a jitter interval. We close with Section 4.4, where we discuss the various models and possible degradation schemes.

## 4.1 Simple Model

The most common method to specify update rates is to indicate the number of samples to be taken over a unit of time, for example, 4 samples per second. This specification is commonly interpreted to sample the time domain $\mathcal{T}$ periodicly and in regular intervals, for example, sampling at times 0s, 0.25s, 0.5s, and 0.75s for the specification of 4 samples/sec. Figure 3 depicts this example. While this specification model is effective, it is also unnecessarily limiting.

The simple model we propose is not limited to periodicly spaced samples. Update rate is specified as a function

$$ur : \mathcal{T} \rightarrow \{0, 1\}, \text{ s.t.}$$

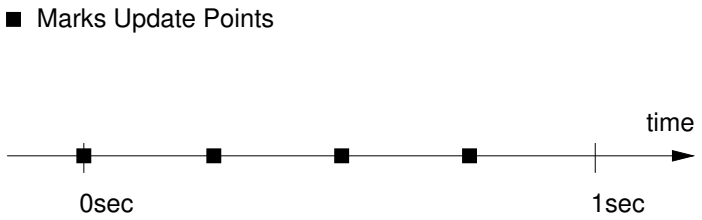


Figure 3: Resulting Update Points for Specification of 4 Samples/Sec

$$\text{if } ur(t) = 1, \text{ it follows that } ur(1 \pm \epsilon) = 0 \text{ for } 0 < \epsilon < \delta,$$

for sufficiently small $\delta$. The update function $ur$ is interpreted to mean that sampling is performed whenever $ur(t) = 1$.

The definition of $ur$ specifies that samples are taken whenever $ur(t) = 1$ and that, while samples can be arbitrarily close together, they are always singular points. Figure 4 depicts the effect of an example update function. This simple

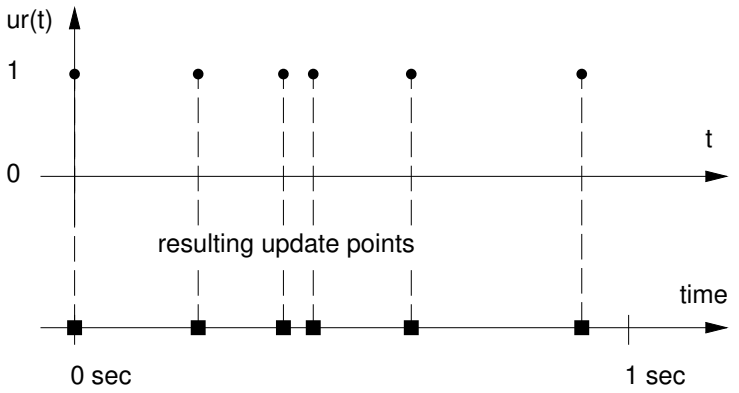


Figure 4: Resulting Update Points for a Simple Update Function

specification model allows update rates that are arbitrarily complex, yet includes periodic update rates as a special case, e.g., by specifying periodic functions.

A possible implementation of the simple model is based on a global, sorted queue. This queue stores sampling events. Each update function $ur$ generates a sampling event for every $t$ such that $ur(t) = 1$. The queue therefore stores all sampling events of all update functions, sorted by sampling event time $t$.[2] A reader associated with the queue pops the top event, i.e., the event at the

[2]We assume that all events exist in the same time coordinate system; if they they do not, we first have to transform all events into one logical, global time coordinate system.

beginning of the queue, and performs the actual sampling or updating of the associated behavior at the event-specified time.

The reader operates in logical time, not in wall-time [1], i.e., behaviors are always sampled in logical time. A scheduler component elsewhere in the system is responsible for linking the logical time to wall-time. While linking these two time coordinate systems and thereby synchronizing logical time to wall-time is a non-trivial problem, it is irrelevant for the time specification model we discuss here.

This implementation, if interpreted literally, requires an event queue of infinite length to store all sampling events of all update functions. To make the implementation practical, we propose to specify the update functions iteratively: instead of specifying an update function in its entirety at once, e.g., by providing infinite lists or periodic descriptions of when to sample, we allow the user to only specify the next sampling event. In particular, an update function $ur$ is implemented as a function of the form

$$iur : \mathcal{T} \rightarrow \mathcal{T} \times \mathcal{B}.$$

The semantics of $iur$ are inferred from the original $ur$ as follows:

$$t_{next} := \min_{t > t_{inq} \wedge ur(t)=1}(t)$$

$$iur(t_{inq}) = \begin{cases} (t_{next}, T) & \text{if } ur(t_{inq}) = 1 \\ (t_{next}, F) & \text{otherwise} \end{cases}$$

The update function $ur$ is restorable from the iterative implementation $iur$. Evaluating $iur$ at an initial, arbitrary time $t_0$ results in the closest $t_1$ at which $iur(t_1) = (t_2, T)$ and therefore $ur(t_1) = 1$. Consequently, applying $iur$ iteratively to the returned $t_i$ results in all other sampling points, i.e., all the points where $ur(t) = 1$. We illustrate this process in the following.

$$\begin{aligned} iur(t_0) &= (t_1, T/F) \\ iur(t_1) &= (t_2, T) \\ &\vdots \\ iur(t_i) &= (t_{i+1}, T) \\ &\vdots \end{aligned}$$

$$\Rightarrow ur(t) = \begin{cases} 1 & \text{if } t = t_i \text{ for } i = 1, \ldots \\ 0 & \text{otherwise} \end{cases}$$

The function $iur$ also returns a boolean to confirm that the passed in time is indeed a sampling event. The inclusion of the boolean stabilizes the definition of $iur$, since applying $iur$ to a non-sampling point is recognized and corrected by returning the next actual sampling event. We actually take advantage of this

stable behavior by applying *iur* to an arbitrary value to initialize the iteration as described above.

We take this thought further and change the definition of *iur* slightly. Let

$$t_{next} := \min_{t > t_{inq} \wedge ur(t)=1} (t)$$

$$t_{inq} < t_{new} \leq t_{next}$$

$$iur(t_{inq}) = \begin{cases} (t_{new}, T) & \text{if } ur(t_{inq}) = 1 \\ (t_{new}, F) & \text{otherwise} \end{cases}$$

In effect, we allow *iur* to return arbitrary points in time that are or are not actual behavior sampling events. Doing so allows *iur* to effectively supersample the function *ur*. We require this feature, for example, if it is uncertain what the actual value of $t_{next}$ is, i.e., when the function *ur* is highly dynamic.

The implementation model for this iterative definition of the simple model still operates on a global, sorted queue. Initially the queue is empty. The system starts by applying all iterative update functions to a common start time. Each iterative update function returns a time $t_{new}$ which constitutes a possible behavior sampling event. We extend the term sampling event to also include these possible sampling points $t_{new}$. All sampling events are then placed in the queue ordered by their associated $t_{new}$. When the reader reads the top event of the queue, it first has to evaluate the event by applying the associated iterative update function to it. The iterative update function returns a boolean and a new event. If the boolean is true, indicating that the evaluated event is indeed a behavior sampling point, the reader samples the associated behavior at the indicated time. If the boolean is false, the reader does nothing. Before reading the next event in the queue, the reader places the new event generated by the previous evaluation of the iterative update function in the global queue.

This iterative implementation model has several advantages. The iterative definition of the update functions allows us to restrict the length of the global queue. Since the queue has to store only as many events as there are update functions — a finite number — it is possible to implement the model efficiently. Efficiency is helped further by storing only one event per update function in the global queue. If more than one event is stored, dynamicly changing update functions require the system to update events already in the queue, followed by resorting the queue itself.

The iterative specification methodology is more flexible than fully specifying functions at system initialization. Especially update functions that are not periodic, e.g., dynamicly changing update functions, benefit from the ability to delay specification as long as possible. In effect, explicitly allowing events in the queue that are not actual behavior sampling points further supports this notion of delay. If at the time of invocation an update function cannot yet decide when the next sample is due, it delays the decision by returning a new event that ensures that the function is called again before the actual behavior sample is due.

## 4.2 Jitter-Interval Model

In the simple model described in section 4.1 all update rates are exact in the time domain $\mathcal{T}$. However, a user might not want to specify exact update rates. For example, a typical animation incorporates many different update rates. While the user can link these update rates to make them interdependent (see Section 3), it is of advantage to only specify guidelines for the update rates of various objects. The system is then free to use these guidelines to decide on the exact update rates, while automatically optimizing other global properties, e.g., sampling a maximum number of objects at the same point in time.

The jitter-interval model we introduce in this section allows users to specify update rates with an associated jitter interval. We derive the jitter-interval model from the simple model by relaxing the singular point restriction on the update function $ur$:

$$ur : \mathcal{T} \rightarrow \{0,1\}, \text{ s.t.}$$

$$\forall t : ur(t) = 1 \; \exists t_0, t_1 \text{ s.t.} \quad \forall s \in [t_0, t_1] : ur(s) = 1 \text{ and}$$
$$\forall \epsilon : 0 < \epsilon < \delta, ur(t_0 - \epsilon) = 0 \text{ and } ur(t_1 + \epsilon) = 0$$

for sufficiently small $\delta$. Updating or sampling is performed exactly once in each of the above defined maximum intervals $[t_0, t_1]$. The system is free to choose the exact sampling point $t$ anywhere within $[t_0, t_1]$.

This definition means that samples are specified as closed intervals over which $ur(t) = 1$. Each interval results in one sample, to be taken anywhere within the specified interval. These jitter-intervals are arbitrarily close together and arbitrarily long. The jitter-interval model contains the simple model as a special case, e.g., when restricting the length of the jitter-intervals to single points. Figure 5 shows an example of such an update function, as well as a valid choice of update rates resulting from the specified function.

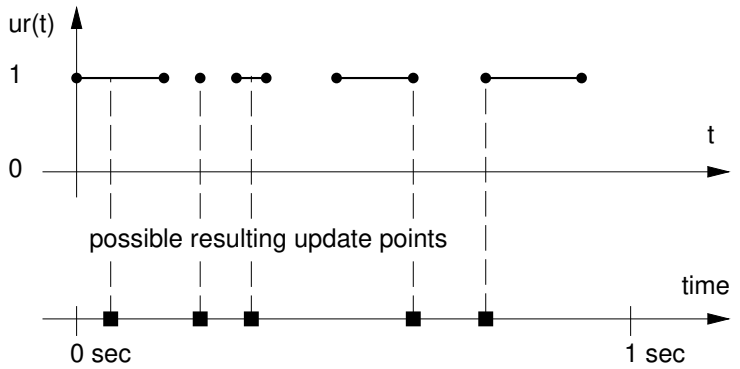


Figure 5: Jitter-Interval Update Function and One Possible Resulting Update Rate

An implementation of the jitter-interval model which we sketch below uses the freedom provided by the jitter-intervals to maximize the number of samples that are taken at the same point in time. Since various objects and attributes have potentially different update rates, the information they provide possibly represents different times. Therefore, a rendered frame shows several objects at a variety of disparate points in time. Maximizing the number of samples that are taken at the same point in time, and therefore minimizing time disparity, is useful to reduce temporal artifacts in a rendered scene.

Even more important, time disparity directly affects efficiency of the system. For example, if the frame rate depends on the update rates of the rendered objects such that a new frame is requested as soon as any of the objects update (thereby keeping the rendered frame as up-to-date as possible), a high time disparity causes a high frame rate. A high frame rate constitutes a large system load. Lowering the time disparity within the limits of the jitter-intervals minimizes the number of rendered frames while staying completely within the user specifications. It therefore also reduces system load.

The jitter-interval model implementation is based on the same principles as the simple model implementation described in Section 4.1. Therefore, the formulas are analogous to Section 4.1.

In particular, update rate functions are specified iteratively, i.e.,

$$\mathcal{I} := \{[s,t] \mid s,t \in \mathcal{T}, s \leq t\} \text{ (defines intervals of time)}$$

$$iur : \mathcal{I} \rightarrow \mathcal{I} \times \mathcal{B}.$$

Iterative update rate functions return time intervals that are analogous to behavior sampling events, specifically, they may or may not include the actual sampling point. These time intervals are called behavior sampling intervals.

The definition of $iur$, i.e.,

$$iur(I_{inq}) = \begin{cases} (I_{true}, T) & \text{if} \quad \forall t \in I_{inq} : ur(t) = 1 \text{ and} \\ & \quad \exists \delta \forall \epsilon, 0 < \epsilon < \delta : ur(t_{inq_r} + \epsilon) = 0 \\ (I_{false}, F) & \text{else} \end{cases}$$

is designed such that the function $iur$ returns true exactly once for each jitter-interval of $ur$.

In particular, the function $iur$ reports true at least once per jitter-interval, because $I_{true}$ and $I_{false}$ cannot skip jitter-intervals:

$$\begin{aligned} I_{true} := \quad & [t_{true_l}, t_{true_r}] \text{ with} \\ & t_{inq_r} < t_{true_l} \leq t_{next_l} \\ & t_{true_l} \leq t_{true_r} \end{aligned}$$

$$
\begin{array}{lll}
I_{false} := & [t_{false_l}, t_{false_r}] \text{ with} & \\
& \text{if } (ur(t_{inq_l}) = 1) & \text{then} \\
& & t_{false_l} = t_{inq_l} \\
& & t_{false_r} = t_{next_r} \\
& & \text{else} \\
& & t_{inq_l} < t_{false_l} \leq t_{next_l} \\
& & t_{false_l} \leq t_{false_r}
\end{array}
$$

and because *iur* returns false if several jitter-intervals are enclosed in a single $I_{inq}$ (see the definition of $iur(I_{inq})$, i.e., $\forall t \in I_{new} : ur(t) = 1$).

On the other hand, the function *iur* reports true at most once per jitter-interval, since a positive result requires the passed-in $I_{inq}$, e.g.,

$$I_{inq} := [t_{inq_l}, t_{inq_r}]$$

to have the same right interval-endpoint as an actual jitter-interval (i.e., the line $\exists\delta\forall\epsilon, 0 < \epsilon < \delta : ur(t_{inq_r} + \epsilon) = 0$ in the definition of *iur*). In addition, the definition of $I_{next}$ disallows repeated inquiries with the same $I_{inq}$ (i.e., $I_{next} \neq I_{inq}$).

$$I_{next} := [t_{next_l}, t_{next_r}] \text{ s.t. } I_{next} \neq I_{inq}$$

$$t_{next_l} = \min_{t \geq t_{inq_l} \wedge ur(t)=1}(t)$$

$$t_{next_r} = \inf_{t > t_{next_l} \wedge ur(t)=0}(t)$$

Note that *iur* does not require the left endpoint of $I_{inq}$ to match the left interval-endpoint of a jitter-interval to return true, making the definition of *iur* stable.

We do not specify $t_{next_l} < t_{inq_l}$ as in the simple model, because it is conceivable that $I_{inq}$ includes $I_{next}$. For example, $I_{inq} = [t_{inq_l}, t_{inq_r}]$ is an inaccurate estimate of the actual jitter-interval $I_{next} = [t_{next_l}, t_{next_r}]$ such that $t_{inq_l} = t_{next_l}$ and $t_{inq_r} > t_{next_r}$. In this case $\min_{t > t_{inq_l} \wedge ur(t)=1}(t)$ does not exist and has to be replaced with $\inf_{t > t_{inq_l} \wedge ur(t)=1}(t)$ which is equivalent to our definition, since the $ur(t) = 1$ only in closed intervals.

An actual implementation of the jitter-interval model is similar to the simple model implementation described in Section 4.1. *Events* are redefined once more to consist of time intervals that are associated with their respective iterative update functions. Applying all iterative update functions to a common start interval (e.g., $[0, 0]$) fills an initially empty global queue. The global queue sorts the events by their left interval-endpoints.

A reader evaluates the top event of the queue by applying the associated iterative update function to the interval represented by the event. If the event corresponds to an actual jitter-interval (i.e., the iterative update function returns true), we store this event in a cache to be processed later. The new event (i.e., the next possible jitter-interval $I_{true/false}$) resulting from the evaluation is inserted into the queue.

The reader continues to evaluate and generate events until the intersection of jitter-intervals stored in the cache no longer forms a non-empty interval (see Figure 6). The latest event (which is responsible for producing an empty intersection) is temporally removed from the cache. The remaining jitter-intervals are intersected to form a new jitter-interval, called *elect interval* for short. Any sampling point within this elect interval is a valid choice for sampling all the behaviors associated with the events in the cache (see also Figure 6). After we

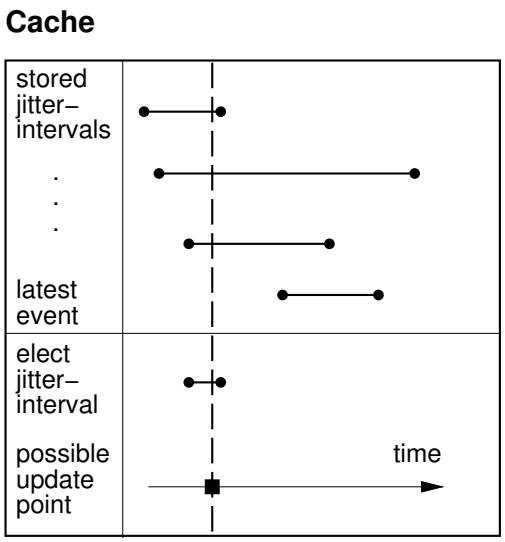


Figure 6: Minimizing Time Disparity for Behaviors with Jitter-Intervals

choose such a sampling point and all the event-associated behaviors are sampled at that sampling point, the previously removed, latest event is restored to the cache. We restart the reader to continue to evaluate events in the queue.

This implementation of the jitter-interval model maximizes the number of samples taken at the same point in time, thereby minimizing time disparity. Rather than choosing a sample point at random within the elect interval, it is conceivable to choose a sample point based on other optimizing strategies.

## 4.3 Weighted Jitter-Interval Model

The model described in Section 4.2 allows the user to specify update rates that have associated jitter-intervals. Every point within a jitter-interval is equally eligible to be the sampling point; the user cannot express any preferences. However, there are cases when such preferences exist. For example, if objects are updated in response to user actions, then it is of advantage to update as early as possible to minimize lag.

The weighted jitter-interval model extends the jitter-interval model. It allows the specification of preferred sampling locations within the jitter-interval. Update rate becomes a function defined as

$$ur : \mathcal{T} \rightarrow \{x \mid x \in \mathcal{R}, 0 \leq x \leq 1\}, \text{ s.t.}$$

$$\forall t, ur(t) \neq 0 \; \exists t_0, t_1 \text{ s.t.} \quad \forall s \in [t_0, t_1] : ur(s) \neq 0 \text{ and}$$
$$\forall \epsilon, 0 < \epsilon < \delta : ur(t_0 - \epsilon) = 0 \text{ and } ur(t_1 + \epsilon) = 0$$

for sufficiently small $\delta$. As in the jitter-interval model, sampling is performed exactly once in each of the maximum intervals $[t_0, t_1]$.

Each closed interval $[t_0, t_1]$ specifies exactly one sampling point to be taken anywhere within the interval, but preferably at $\max_{t_0 \leq t \leq t_1} ur(t)$. An example of such an update function and a preferred update rate are shown in Figure 7. The shown update rate is not unique: other update rates are equally good, since

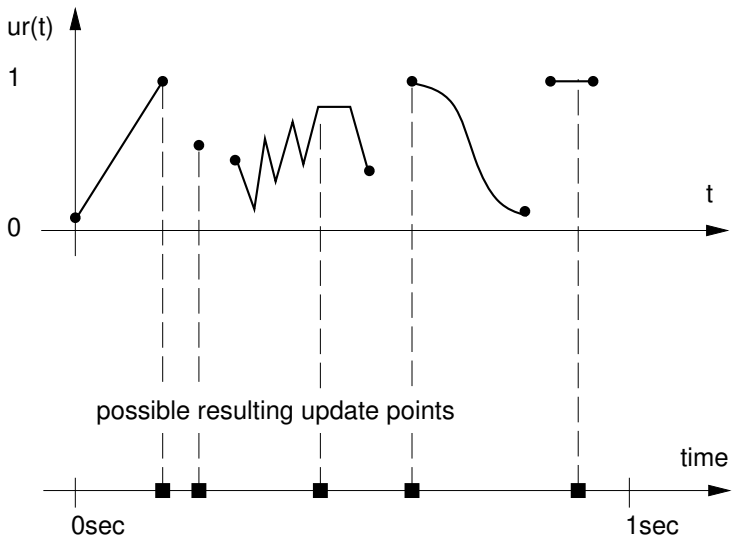


Figure 7: Weighted Jitter-Interval Update Function and One Possible Prefered Update Rate

some of the jitter-interval maxima are not unique.

An implementation of the weighted jitter-interval model is analogous to the jitter-interval model implementation outlined in Section 4.2. We, therefore, do not repeat the definitions for the iterative update rate functions, the algorithm for setting up and operating the global event queue, etc., here.

However, the two implementations differ considerably in how they examine events in the cache (see Section 4.2). The jitter-interval model computes the optimum sampling point by forming the elect interval from consecutive sampling

intervals. In the weighted jitter-interval model, the problem can no longer be broken down into maximal groups of events that form non-empty intersections. As Figure 8 demonstrates, the optimal solution when to sample within the

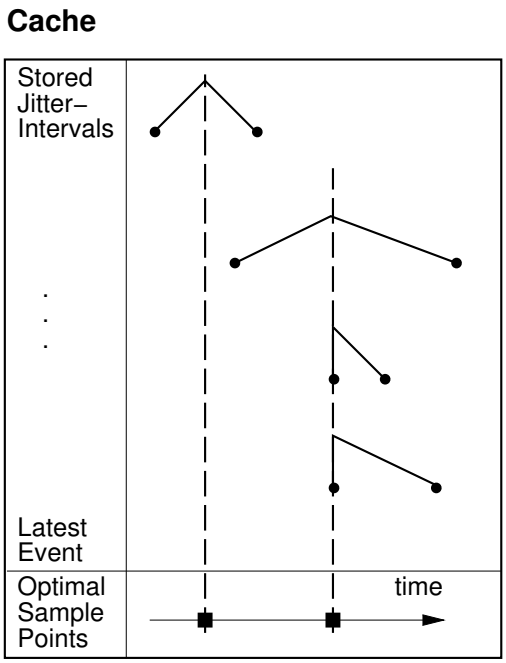


Figure 8: Minimizing Time Disparity for Behaviors with Weighted Jitter-Intervals

weighted jitter-interval requires a non-trivial grouping of events. Hence, the problem of finding the optimal sampling point cannot be divided into more easily solvable problems, and therefore becomes a global optimization problem. Furthermore, since the weighted jitter-intervals are general functions, analyzing them to find the maxima is difficult.

Due to these difficulties in the weighted jitter-interval model, computing an optimal solution is complicated and therefore costly. Since we wish to use these update specification models in time-critical applications (see Section 1), we cannot afford such delays. Hence, we do not plan to investigate the weighted jitter-interval model or even more complicated models in any more detail.

## 4.4 Discussion of the Specification Models

The specification models presented allow a user varying degrees of freedom in specifying update rates. All specification models fulfill the minimum require-

ments mentioned in the beginning of the section, i.e., they allow a user to specify arbitrary, dynamic, and degradable update rates, as discussed in the following.

The above specification models do not restrict the type of update rate, i.e., update rates are arbitrary. Even though update rates therefore represent general functions, we avoid potential problems stemming from analyzing these general functions (for example, to find the next sampling interval) by specifying the update rates iteratively.

The iterative specification contributes to the general strategy of delaying computations about update rates as long as possible. This strategy aids the efficient implementation of dynamic update rates.

We have not yet discussed how we degrade update rates. Since we believe that update rate degradation is orthogonal to the actual specification model, we discuss degradation independent of the specification model. The degradation methods we present can be used with any of the introduced update rate specification models.

One method for degrading update rates is to require the user to explicitly state the alternative update rates. The system switches to the alternative update rates when needed. The user has direct control over the degraded update rates and, hence, insures that the degradation maintains essential properties of the update rate. However, since the user encodes all degradation steps manually, this method is labor-intensive; we need an automatic degradation method.

We propose a degradation method which drops $d$ samples every $f$ samples from the originally specified update rate. A list of fractions $(\frac{d_1}{f_1}, \frac{d_2}{f_2}, \ldots, \frac{d_n}{f_n})$ specifies the degradation: at first the system degrades the update rate by dropping $d_1$ samples every $f_1$ samples, then, if further degradation is required, $d_2$ samples every $f_2$ samples and so forth, up to a maximum degradation of dropping $d_n$ samples every $f_n$ samples.

While this method improves on the previous one in terms of information to be specified, it is not automatic enough. The user still has to describe every single degradation step. Therefore, we envision the user to merely specify a single fraction $\frac{d_{max}}{f}$ which specifies the maximum possible degradation of the update rate. The system degrades the update rate by initially dropping 1 sample every $f$ samples, then 2 samples every $f$ samples, up to $d_{max}$ samples every $f$ samples. This degradation method is the most compact, but also the least controllable.

The system is responsible for deciding which $d$ samples to drop. We assume it beneficial to spread the drop-outs evenly over the range of $f$ samples. A simple test computes which samples to skip: if $(\lfloor\frac{(i+1)d}{f}\rfloor - \lfloor\frac{id}{f}\rfloor) > 0$ is true for sample $i$ then that sample is dropped.

A degradation method integrates with the update rate specification models by using it as a post-process for evaluating events. We drop samples from an update rate after we determine whether an event is an actual sample, but before we use the sample to optimize global properties. We drop a sample if the following three conditions are true. First, the event is a true jitter-interval.

Second, the associated update rate is to be degraded. Third and last, the sample event is to be dropped under the used degradation scheme.

Applying degradation as a post-process to event evaluation is of advantage. If degradation is applied earlier, i.e., to the events, the actual degree of degradation is uncontrolled, because the ratio of real jitter-intervals to events is unknown a priori. If degradation is applied later, i.e., after the optimal sampling point has been found, e.g., in the jitter-interval or in the weighted jitter-interval models, the optimality of the sampling point is compromised.

# 5 Conclusion

## 5.1 Contributions

In our quest to create a time-critical graphics system that fully integrates multimedia data, we have identified update rate as an important concept. Yet, explicit update rates are currently missing from contemporary graphics systems. Explicit control, while possibly cumbersome, is more valuable than implicit, invariable update rates.

We list the paper's three main contributions in the following. First, we define a general, underlying framework for graphics systems to underline the general applicability of the concept of update rates in today's graphics systems. Second, we discuss some of the issues of how to integrate update rates into graphics systems, i.e., with which components to associate update rate, useful update rate operators, and possible default construction methods. Third and last, we introduce a number of update rate specification models to address the specific needs of dynamic, time-critical graphics systems, in particular, we allow for uncertainty in the update rate specification, how to use this uncertainty to optimize other global properties, and how to make these specification models practical.

## 5.2 Future Work

We plan to expand this work in several directions. As already indicated in Section 3, better incorporation models are needed. Furthermore, enlarging the set of update rate operators would ease update rate specification; trying to define an algebra for these operators would minimize the number of needed operators, while maintaining the same functionality.

Depending on the needs of specific applications, more sophisticated update rate specification models are required, e.g., a specification model that allows the specification of motion blurring effects. Also, trying to optimize global properties other than time disparity should prove helpful.

Finally, as pointed out in Section 1, automating the process of finding optimal update rates for individual objects (and how to optimally degrade them) is

a long term goal of this research.

## Acknowledgements

We are grateful to Andy van Dam. Robert C. Zeleznik also helped to realize this paper; he was the devil's advocate during discussions while conceiving the ideas in this paper and provided helpful suggestions while writing the paper. Of course, none of this could have been possible without the continuing support from our sponsors: this work was supported in part by 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] Samuel J. Cardman. Time management in a real-time animation/graphics environment. *Computer Graphics (SIGGRAPH '75 Proceedings)*, 9(1):201–207, June 1975.

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

[3] 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.

[4] Jr. Frederick P. Brooks. Grasping reality through illusion – interactive graphics serving science. In *Proceedings of ACM CHI'88 Conference on Human Factors in Computing Systems*, pages 1–11, 1988.

[5] 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.

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

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

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

[9] 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.

[10] Devendra Kalra. *A Unified Framework for Constraint-based Modeling.* PhD thesis, California Institute of Technology, May 1990.

[11] Devendra Kalra and Alan H. Barr. Modeling with time and events in computer animation. *Computer Graphics Forum (EUROGRAPHICS'92)*, 11(3):C45–C58, 1992.

[12] 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.

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

[14] Paul S. Strauss (organizer), Ben Trumbore (organizer), Andrew Glassner, Eben Ostby, and Robert Zeleznik. Developing large-scale graphics software toolkits. *Course Notes of Course 03 at SIGGRAPH'93*, August 1993.

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

[16] 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.

[17] Michael Sannella. The SkyBlue constraint solver. Technical Report 92-07-02, Dept. of Computer Science and Engineering, University of Washington, February 1993.

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

[19] 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.

[20] 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.

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

[22] Andries van Dam (chair), Salim Abi-Ezzi, Rick Carey, and Mark Tarlton. Graphics software architecture for the future – notes of panel discussion at SIGGRAPH'92. *Computer Graphics (SIGGRAPH'92)*, 26:389–390, July 1992.

[23] Matthias M. Wloka. Thesis proposal: Time-critical graphics. Technical Report CS-93-50, Brown University, Department of Computer Science, Providence, RI, November 1993.

[24] 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.