MITSUBISHI ELECTRIC RESEARCH LABORATORIES
CAMBRIDGE RESEARCH CENTER

Time Synchronization In Spline

Richard C. Waters

TR-96-09 April, 1996

Abstract

The Spline scalable platform for interactive environments makes it easy to build virtual worlds where multiple people interact with each other and with computer simulations in a 3D visual and audio environment. A key problem shared by Spline and many other systems is achieving accurate time synchronization of events and data streams.

In Spline, synchronization is achieved through the use of timestamps. Given the nature of the audio and visual data supported by Spline, timestamps on the order of 64 bits are required to accurately specify the absolute position of individual pieces of data in time. Nevertheless, synchronization is achieved by using timestamps only 32 bits long, thereby saving bandwidth, storage and computation time.

The use of 32-bit timestamps is made possible by reducing the precision and range of the timestamps required by the data. The precision is limited by grouping data that is sampled at high frequency into lower frequency chunks, and giving each chunk a low precision timestamp. The range is limited by using an efficient form of modular arithmetic for expressing timestamps.

For more information on Spline visit `http://www.merl.com`.

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of Mitsubishi Electric Research Laboratories of Cambridge, Massachusetts; an acknowledgment of the authors and individual contributions to the work; and all applicable portions of the copyright notice. Copying, reproduction, or republishing for any other purpose shall require a license with payment of fee to Mitsubishi Electric Research Laboratories. All rights reserved.

Copyright © Mitsubishi Electric Research Laboratories, 1996
201 Broadway, Cambridge, Massachusetts 02139

Contents

1. Introduction 1
- 1.1 The Synchronization Problem 1
- 1.2 An Efficient Solution 2
- 1.3 Generality 3

2. Spline 3

3. High Precision Absolute Timestamps 6

4. Reducing Precision Via Data Chunking 7

5. Reducing Range Via Modular Arithmetic 7
- 5.1 The Computational Complexity of Modular Arithmetic . 8
- 5.2 Quotient-Normalized Modular Timestamps 10
- 5.3 Periodic Renormalization 10
- 5.4 Adjusting For Clock Differences Between Machines 11

6. Efficient Buffering of Chunked Data 12

7. Conclusion 14

Acknowledgments 15

References 15

1 Introduction

Our long term goal is creating social virtual reality systems where people can interact in real time for learning, work and play. In particular, we seek to create virtual worlds featuring: multiple, simultaneous, geographically separated users; multiple computer simulations interacting with the users; spoken interaction between the users; immersion in a 3D visual and audio environment; and comprehensive run-time modifiability and extendability.

To explore the ways that social virtual reality systems can best support work and education as well as entertainment, we believe that it is essential to create and rapidly evolve a number of actual systems. To make this kind of exploration possible, we have focussed much of our effort over the past two years on the creation of a flexible platform called Spline (Scalable PLatform for INteracive Environments) for building social virtual reality systems.

The goal of Spline is to allow an application writer to focus on creating the content of a particular world without worrying about communication and synchronization between multiple users. In comparison to other tools for supporting 3D virtual worlds, Spline is notable in that it supports both visual and audio interaction between people, allows runtime extensions of all kinds of data, and is designed to support large numbers of users.

The first use of Spline has been to support a multi-user virtual world called Diamond Park [1]. Diamond Park is a social virtual reality system in which multiple geographically separated users can speak to each other and participate in joint activities. The most important part of a visit to the park is meeting and talking with other visitors. We have chosen to focus primarily on social interaction in Diamond Park, because we believe that good support for social interaction will be the most important feature of social virtual reality applied to any purpose.

The central theme of Diamond Park is bicycling. Visitors to the park can explore a square mile of 3D terrain. In addition to the 3D animated avatars of human users, the park contains a number of computer simulations including a tour bus. Diamond Park is notable for providing a relatively high level of visual and auditory detail while maintaining interactive frame rates. We consider it a major validation of Spline that while Diamond Park required the construction of many large graphical models and large amounts of recorded sounds, very little code had to be written over and above what is contained in Spline itself.

1.1 The Synchronization Problem

In a typical use of Spline, a number of computers running Spline communicate a variety of different kinds of information over a computer network. This information includes live audio (e.g., the spoken interaction of users), changes in the appearance of the virtual world (e.g., altered positions of objects), and events (e.g., commands to play prerecorded sound effects). (Although it could readily be incorporated into the architecture, Spline does not currently support video.)

To support a quality immersive experience, it is important that the various streams of data coming to a particular Spline node be accurately synchronized before presentation to the user. For example, this is needed to ensure that the sound of a door slamming in the virtual world will be heard at the same moment that the door is seen to close, rather than earlier or later.

In Spline, synchronization must be achieved in the face of an inherently messy communication environment in which UDP multicast messages are used to communicate data over an asynchronous wide area network such as the internet. In particular, the messages from a single

source are not guaranteed to arrive in order and may have latencies that differ by 100 msecs or more. The latencies experienced by messages from different sources may differ even more.

As a result, it is not possible for the Spline processes sending data to ensure that the data will arrive at a given receiver in a synchronous fashion. Rather, explicit timestamps are used as the foundation of synchronization in Spline. Whenever a piece of data is sent in a message, it is accompanied by a timestamp specifying when it should be used. Synchronization occurs at the receiver where pieces of data are stored in buffers and removed for use at the times specified.

Since Spline supports 48 kHz audio and non-stop operation over extended periods of time, timestamps with a precision of 0.02 msec ranging over years would be required to specify the exact absolute positions in time of individual data samples. Such timestamps would be straightforward to use, but would occupy 64 bits of storage. Using 64-bit timestamps would be costly because they use a lot of space and, on 32-bit machines, require the use of multiple precision arithmetic.

Because the typical Spline session involves the communication of a large amount of data, synchronization and the manipulation of timestamps is a significant part of the activity of Spline. As a result, using 64-bit timestamps would introduce a significant amount of overhead. To avoid this, Spline has been designed to operate with compact timestamps that can be much more efficiently manipulated.

1.2 An Efficient Solution

Two key observations make it possible to use 32-bit or even 16-bit timestamps instead of 64-bit timestamps. First, audio samples are used only in groups rather than individually. By appropriately breaking the audio into chunks, timestamps with a precision of only 1 msec can be used.

Second, while non-stop operation is important, synchronization never involves pieces of data that are far apart in time, but rather only pieces of data that are quite close to each other in time. In particular, while it is not possible to guarantee that data will arrive at the correct millisecond, it is in general trivial to guarantee that it will arrive within at most a second of the right time. This observation suggests the use of modular arithmetic to reduce the range of timestamps that have to be accommodated at any one time.

Using standard forms of modular arithmetic would reduce the amount of memory needed to represent timestamps; however, it would lead to increased computation. The reason for this is that the standard algorithms for manipulating modular numbers are more complex than the corresponding algorithms for manipulating absolute numbers. Fortunately, the fact that the modular numbers to be manipulated are known to correspond to absolute numbers that are near to each other makes it possible for Spline to use a special form of modular arithmetic in which timestamps are represented in terms of a periodically changing base plus an offset. In this representation, many kinds of manipulation are just as fast as the equivalent operations applied to absolute numbers (see Section 5.2).

In Spline, we use modular, millisecond timestamps with a modulus of one week. These timestamps can be accommodated in 32 bits, thereby saving storage, communication bandwidth, and computation time in comparison with 64-bit timestamps. We chose to use 32-bit timestamps modulo one week, because it allows Spline to tolerate clock-synchronization errors between machines of a day or more. In addition, this choice allows Spline to conveniently refer to times several days in the future and durations of up to one week.

1.3 Generality

The approach to timestamps presented here was developed in the context of Spline. However, it could be used to support synchronization in a wide range of contexts. Specifically, the approach should be valuable whenever high precision synchronization is required in a non-stop system, but the events that have to be simultaneously considered for synchronization are known in advance to be near to each other in time.

If the maximum interval between events whose timestamps have to be compared can be sufficiently limited, it is possible to go beyond the approach used in Spline and use timestamps with even fewer than 32 bits. For example, in some situations one could choose to use modular, millisecond timestamps with a modulus of only 15 seconds. Such timestamps could be accommodated in only 16 bits, providing an even greater savings in storage and communication bandwidth.

2 Spline

Before discussing the timestamps used in Spline in greater detail, it is useful to consider the basic way that Spline operates. Spline provides a convenient architecture for implementing multi-user interactive environments that is based on a shared world model. The world model is a distributed object-oriented database containing information about everything in a virtual world—where things are, what they look like, what sounds they are making, etc. Applications interact with each other by making changes in the world model and observing changes made by other applications.

The world model simultaneously supports four different kinds of data: large slowly changing data (e.g., graphic models, recorded sounds), small rapidly changing data (e.g., the positions of objects), real time data streams (e.g., user speech), and actions (programs that run remotely in Spline processes). A key feature of Spline is that it includes an efficient scheme for synchronizing these different kinds of data.

To allow rapid interaction between individual applications and the world model, Spline distributes the world model, maintaining a partial copy of the model locally in each Spline process. This copy contains the parts of the model that are relevant to what the process is doing—i.e., are sufficiently near to the process's focus of attention and consisting of the kinds of objects the process is interested in. Spline provides all the processing necessary to maintain consistency between the world model copies associated with a group of communicating Splines, sending update messages when necessary. To minimize computation and communication, the Spline world model is broken up into compact regions called 'locales' that are communicated using separate multicast communication channels (see [2]).

To allow applications to modify and extend all aspects of a virtual world, mechanisms are provided so that all four of the types of data in the world model can be transmitted between Spline processes at run time. Standard formats are used for graphic models and recorded sounds so that standard tools can be used to create graphic models and recorded sounds.

The structure of a single Spline process is shown in Figure 1. The inter-process communication module sends out multicast messages describing changes in the local world model copy made by the local application and receives messages from other Spline processes about changes made remotely.

Spline's Application Program Interface (API) consists primarily of operations for creat-

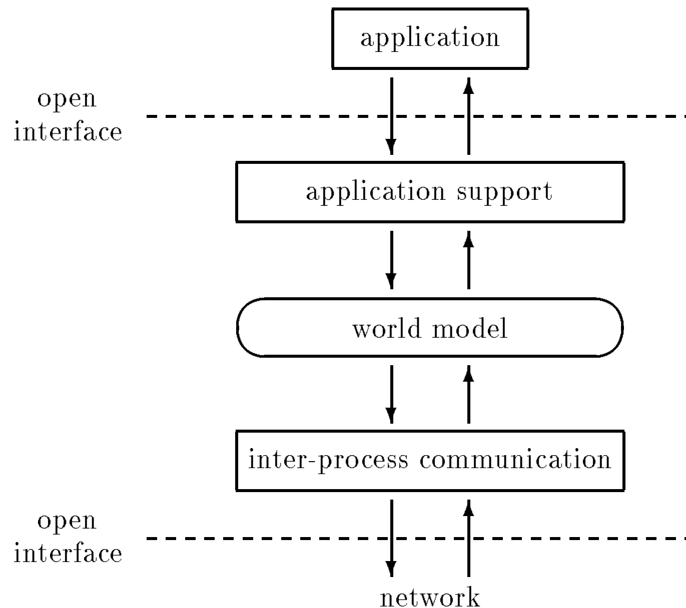

Figure 1: A Spline process.

ing/deleting objects in the world model and reading/writing data fields in these objects. The application support module contains various tools that facilitate interaction between an application and the local world model copy. Foremost among these is an interactive interface created by interfacing GNU Scheme to Spline. This greatly speeds the implementation of applications by supporting interactive experimentation.

Figure 2 shows Spline being used to support an application that interacts with a human user. The figure shows three Spline processes. The application itself presents an interface to the user and interacts with the Spline process in the middle of the figure. Visual and audio rendering modules that are supplied as part of Spline interact with separate Spline processes on the left and right of the figure. It is likely, but not necessary, that the three Spline processes in Figure 2 run on a single machine.

Spline's visual renderer is implemented using the Performer toolkit from SGI. In particular, it was implemented by modifying SGI's Perfly example so that the scene graph to be displayed is obtained from Spline's world model. Each object in the world model that has a visual appearance has an associated graphical model. The 'camera' specifying the point of view (POV) of a given user is represented as an object in the world model called a visual POV. (Using an explicit object for this allows Spline to support a wide variety of interaction models including a through-the-eyes view and more distant views.) The graphical models for the world model objects that are sufficiently near the visual POV to be potentially visible are combined together into a scene graph, which is then rendered by Performer. It is easy for Spline to support any graphical format that can be loaded into Performer.

Spline's audio renderer uses standard algorithms to capture what the user says as data in

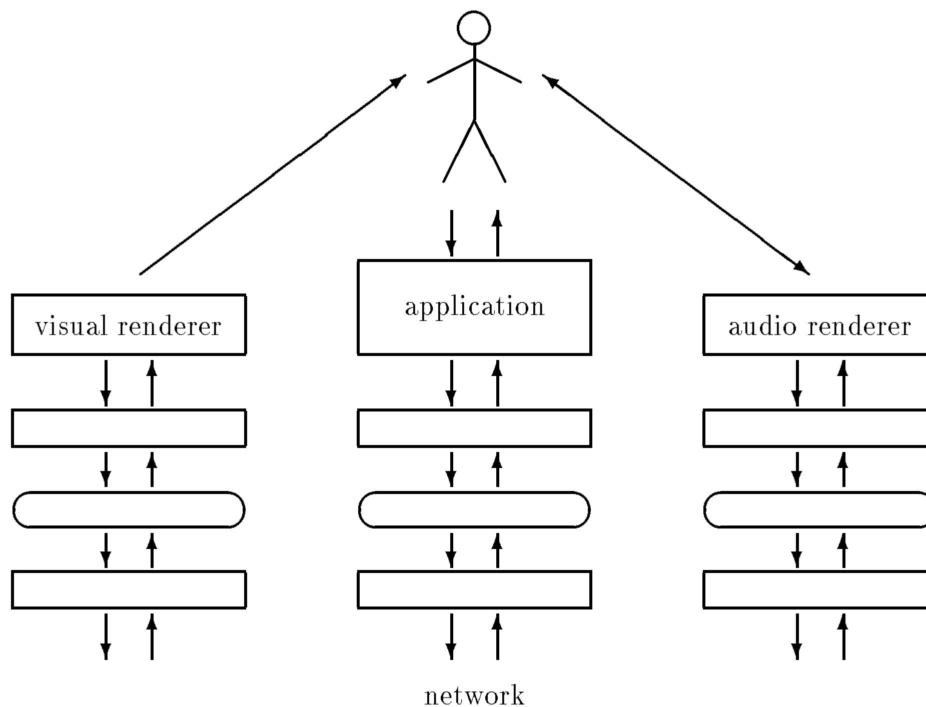

Figure 2: Typical configuration supporting a user.

the world model and create stereo sound for the user to hear. Spline supports both prerecorded sounds and live sound streams between users. Speaker objects in the world model represent the places from which sound emanates. They can be either point sources or diffuse ambient sources. Sound streams are typically transmitted over the network using either an 8-bit Ulaw encoding at 64 kbits/second or ADPCM encoded at 32 kbits/second. The point of view from which a given user perceives sound is represented by an aural POV object. Volume attenuation is used to indicate distance and differential attenuation of the left and right channels is used to indicate direction. In the future, we anticipate using better audio rendering algorithms to create a more detailed auditory environment.

At the current time, the parts of Spline shown in Figure 1 run on both SGI machines and HP machines; however, the visual and audio renderers in Figure 2 run only on SGI machines. We anticipate porting Spline to PCs shortly, using plug-in boards to support visual and audio rendering.

The prime point of comparison between Spline and other platforms for supporting multi-user interactive environments is the fact that Spline is scalability to large numbers of users and more comprehensive, supporting: both 3D graphics and live conversation.

To achieve scalability to large numbers of users, Spline adopts the basic approach pioneered by SIMNET [3] and the distributed interactive simulation protocol (DIS) that grew out of SIMNET. However, Spline goes beyond DIS in five important ways: (1) Spline is a complete platform providing visual and audio rendering modules, not just a communication standard, (2) much greater emphasis is placed on communicating audio information, (3) every kind of data in the world model can be transmitted between Spline processes at run time, (4) the ownership of objects can be transferred from one process to another, and (5) the world model is broken

up into locales that are communicated separately [2].

There is currently a great deal of activity surrounding VRML and Java and the developers of these tools are quite interested in supporting multi-user interaction. However, these tools do not yet support interaction between users. VRML 1.0 is strictly a graphical modeling language and Java is a scripting language that supports the easy and safe transport of programs between different machines. VRML and Java are both very valuable tools and we intend to combine them with Spline. This should be straightforward to do, since there is actually very little overlap in the capabilities of the three systems—they complement each other rather than competing.

3 High Precision Absolute Timestamps

As noted in the introduction, Spline uses timestamps as the basis for synchronization. Each piece of data sent over the network is tagged with a timestamp. Synchronization is achieved by buffering data on receipt so as to convert inherently variable transmission delays into a constant delay corresponding to the worst case transmission delay. In order to reduce the overhead introduced by synchronization, it is important that timestamps be very efficiently manipulated and compared.

A central question regarding timestamps is exactly how much computer memory is needed to represent a single timestamp. This is determined by two key factors: the precision needed in the timestamps and the total range of timestamps required.

The level of precision needed for timestamps is determined by the properties of the human perceptual system and the data being transmitted. The precision needed when synchronizing visual changes is limited by the time sensitivity of the human visual system and the frame rates of various visual media. Except for certain special situations, the human eye cannot respond to differences in time less than several tens of milliseconds. Accordingly, visual media typically have frame rates no greater than 100 Hz. All in all, a precision of 10 msecs or so should be adequate for visual synchronization.

The precision needed when synchronizing audio changes is limited by the time sensitivity of the human hearing system and the sample rates of various audio formats. The ear is much more sensitive to time differences than the eye. Differences of only a few milliseconds can be heard. Further, audio media have a wide range of sample rates, including some that are very high. For example, CDs use a sample rate of 44,100 Hz and a sample rate of 48,000 Hz is used in some other digital media. All in all, audio synchronization requires much higher precision that visual synchronization. A precision approaching 0.01 msec is needed if individual sound samples are to be exactly positioned in time.

Most basically, the range of timestamps required depends on how long the system is intended to operate. In the case of Spline, we desire non-stop operation over months or even years. It is expected that while a typical user will probably participate in a Spline session for at most a few hours at a time, the session itself could in principle continue indefinitely with users coming and going. To support this, the timestamps used must be able to represent times ranging over long periods. To be on the safe side, one should pick a generous range such as 100 years.

The analysis above leads to the conclusion that synchronization in Spline requires timestamps with a precision of 0.01 msec ranging over 100 years or so. Unfortunately, a large number of bits are required to represent timestamps of this precision and range. In particular, $\log_2(100,000 \times 60 \times 60 \times 24 \times 365 \times 100) = 49$ bits. Further, given current computer architec-

tures, there is no practical way to use only 49 bits; rather one has to step up to 64 bits. (Note that 32 bits suffices for a range of only about 11 hours at a precision of 0.01 msec.)

Using 64-bit timestamps, synchronization would be straightforward, but costly. To start with, a lot of space is required because 64 bits are required for every timestamp in network messages and elsewhere. In addition, a lot of computation time is required because most current computers do not support computation with integers of more than 32 bits. Manipulating 64-bit numbers requires multiple precision calculation routines rather than simple instructions.

Fortunately, as discussed in the following two sections, it is possible to achieve effective synchronization with timestamps that have significantly less precision and greatly reduced range.

4 Reducing Precision Via Data Chunking

The dominant consideration when determining the level of precision needed for timestamps is the requirements of audio. This is true in part because the human audio system is more sensitive to synchronization, or the lack thereof, than the visual system. However, the human audio system only requires a precision of 1 msec or so.

The force behind the need for a precision as small as .01 msec comes not from a need to synchronize sounds as a whole with each other or visual phenomena to anywhere near this accuracy, but rather by the need to position individual sound samples with respect to each other within a single sound very accurately. This additional need for synchronization can be eliminated by appropriately chunking the data.

Since audio samples never occur in isolation, but only as part of long sequences of samples, nothing is lost by requiring that they be grouped together into chunks. If the length in time of the chunks is required to be exactly representable using the precision of timestamps chosen, then the chunking makes it possible to use timestamps with much lower precision.

Specifically, Spline uses millisecond timestamps and requires that audio data be broken up into chunks, where each chunk is an exact integer number of milliseconds in length. For example, at a sound sample rate of 8,000 Hz, a chunk can contain 8 samples (1 msec), 16 (2 msecs), 32 (4 msecs), etc., but not 33 samples (4.125 msecs). At the music CD sound rate of 44,100 Hz, a chunk can contain 441 samples (10 msecs), 882 (20 msecs), etc., but not 100 samples (2.268 msecs). If a sound as a whole is not an integral number of milliseconds in length, the last chunk is padded with silence.

Using chunks that correspond to an exact integer number of msecs insures that when using msec timestamps successive chunks will line up exactly one right after the other, without a gap or overlap of even a single sample. However, the first chunk will be forced to line up exactly at a millisecond time boundary. This represents a small loss of precision in comparison with using more precise timestamps; however, the error is not detectable by the human ear.

The discussion above focuses on audio; however, the same chunking approach could be used for any kind of data where the precision required to position individual samples is significantly greater than the precision required by the human perceptual system.

5 Reducing Range Via Modular Arithmetic

In Section 3, it was pointed out that the timestamps being used in Spline have to be able to express times throughout the entire lifetime of a Spline session, which can be very long.

However, when used for synchronization, timestamps are not used one at a time, but rather in pairs. The key question really is what is the greatest difference in time between two timestamps that have to be compared? A Spline session can last for months without the timestamps on two pieces of data to be synchronized ever differing by more than a few seconds. Spline takes advantage of this fact by using modular timestamps.

In particular, instead of using the absolute time in milliseconds as a timestamp, Spline uses the time in milliseconds modulo one week. Since one week in milliseconds is $1000 \times 60 \times 60 \times 24 \times 7 = 604,800,000$, which is less than $2^{32} = 4,294,967,296$, these timestamps can be expressed using only 32 bits. (Following the Unix convention, time is calculated relative to January 1, 1970. The detailed reasons for choosing one week in milliseconds as opposed to some other number less than 2^{32} are discussed in later sections of this paper.)

Using modular timestamps, one can precisely and unambiguously synchronize any two events that are separated in absolute time by less than one half the modulus. However, with regard to longer time intervals, one has no useful information.

In the case of Spline, the modulus is one week and synchronization is supported for events differing by no more than 3.5 days. The fact that longer intervals cannot be handled is unimportant, because Spline never has to simultaneously consider events separated by such large intervals. In particular, using asynchronous networks, it is not possible to guarantee that data will arrive at the correct millisecond, or perhaps even in the right second, but it is trivial to guarantee that it will arrive without a few seconds of the right time, and therefore without question much less than 3.5 days from the arrival time of any data it should be synchronized with.

5.1 The Computational Complexity of Modular Arithmetic

Using modular arithmetic to change from 64- to 32- bit timestamps saves memory space and communication bandwidth by reducing the size of timestamps, but does not necessarily save computation time. The reason for this is that, in general, arithmetic operations on 32-bit modular numbers are much more complex than operations on 32-bit absolute numbers. To understand this, one must consider a few facts about modular arithmetic.

The fundamental relationship between an absolute number X and the corresponding modular number x with respect to a modulus m is the following:

$$X = X' \times m + x$$

In general, modular numbers are required to be what we will call 'remainder-normalized'. This means that it is required that $0 \leq x < m$. In this situation, x is the remainder of dividing X by m and X' is the quotient of X divided by m. For example, since

$$\begin{aligned} 57 &= 6 \times 10 - 3; \quad \text{and} \\ 57 &= 5 \times 10 + 7; \quad \text{and} \\ 57 &= 4 \times 10 + 17; \end{aligned}$$

the remainder-normalized representation of 57 modulo 10 is 7, while -3 and 17 are modular representations that are not remainder-normalized.

To compute the sum Z of two absolute numbers X and Y, one can use a single machine instruction. However to add the corresponding remainder-normalized modular numbers x and

y one has to add the numbers and then remainder-normalize the result, using a computation like the following, which requires several machine instructions instead of just one.

```
z = x + y;
if (z >= m) z = z - m;
return z;
```

For example, if 57 modulo 10 is added to 58 modulo 10, the normalized result is $5 = 7 + 8 - 10$.

An even greater problem arises when using modular numbers as a basis for comparing the absolute numbers they correspond to. To start with, one must assume that the absolute numbers differ by no more than half the modulus, because without such an assumption, the comparison makes no sense. For example, suppose one is asked to use the two base 10 modular numbers 4 and 7 as a basis for comparing the corresponding absolute numbers. The modular number 4 corresponds to the absolute numbers 4, 14, 24, 34, etc. The modular number 7 corresponds to the absolute numbers 7, 17, 27, 37, etc. Without some assumption about which absolute numbers the modular numbers can correspond to, there is no way to compare the modular numbers. If 4 corresponds to 34 while 7 corresponds to 17, then 4 corresponds to the larger absolute number. Alternatively, if 4 corresponds to 14 and 7 corresponds to 17, then 7 corresponds to the larger absolute number.

However, if one assumes that the absolute numbers corresponding to 4 and 7 differ by no more than 5 (which is one half of the modulus 10) then one can definitively state that the absolute number 4 corresponds to is less than the absolute number 7 corresponds to. The reason for this is that the corresponding absolute numbers might be 14 and 17 or 24 and 27, but pairs like 24 and 17, that differ by more than 5 are ruled out. In every permitted pair, the absolute number corresponding to 4 is less than the absolute number corresponding to 7.

Even with the assumption above, determining which of two remainder-normalized modular numbers corresponds to the larger absolute number is complex. The basic problem is that one must compensate for the remainder-normalization when making the comparison. In particular, due to remainder-normalization, it is possible for a modular number to be numerically smaller than another modular number and yet correspond to a larger absolute number. For example, while the base 10 modular number 4 corresponds to a smaller absolute number than than the base 10 modular number 7, the base 10 modular number 1 corresponds to a larger absolute number. The reason for this that under the assumption above, it must be the case that if 1 corresponds to 11 then 7 corresponds to 7, not 17.

To determine whether an absolute number X is less than another absolute number Y one can use a single comparison instruction; however, comparing the corresponding remainder-normalized modular numbers requires a computation like the following, which again requires several machine instructions instead of just one.

```
z = y - x;
if (z < -m/2 || (0 < z && z < m/2)) return TRUE;
return FALSE;
```

Note that in both of the code fragments above, the number of bits used to represent the intermediate result `z` must be sufficient to represent the sum of two modular numbers and to represent negative numbers corresponding to differences between modular numbers. It is therefore important that one week is not only less than 2^{32}, but also less than $2^{30} = 1,073,741,824$, because this allows 32-bit numbers to be used to represent both timestamps and intermediate calculations involving timestamps without fear of overflow or underflow.

5.2 Quotient-Normalized Modular Timestamps

To avoid the computational complexities presented in the last section, Spline does not use remainder-normalized modular numbers. Rather it uses modular numbers that are what we will call 'quotient-normalized'. Specifically, given two modular numbers x and y

$$X = X' \times m + x;$$
$$Y = Y' \times m + y;$$

instead of requiring that $0 \leq x < m$ and $0 \leq y < m$, quotient-normalization requires that $X' = Y' = Q$ for some particular Q while requiring only that $-m < x < 2 * m$ and $-m < y < 2 * m$. At any given moment, all the timestamps in existence in a given Spline process are normalized to correspond to the same quotient Q. One can think of quotient normalized modular numbers as representing numbers by an offset from an agreed common base. For example, 17, 24, and 31 might be represented by the base 10 quotient normalized modular numbers -3, 4, and 11 respectively using the common quotient 2.

(Note that quotient normalized modular numbers require $2 + \log_2(m)$ bits to represent them. However, as noted above, at least this many bits is also required if remainder-normalized numbers are to be manipulated efficiently).

Quotient-normalization is appropriate for synchronization because, since all the pieces of data under consideration for synchronization at any one moment are known to be relatively close to each other in time, it is not unreasonable to require that $X' = Y'$. In situations where modular numbers corresponding to widely different absolute numbers have to be simultaneously considered, this restriction would be untenable.

The value of quotient-normalization for synchronization is that the normalization used does not complicate the process of comparing two timestamps. In particular, one can determine whether a quotient-normalized modular number corresponds to a smaller absolute number than another quotient-normalized modular number by simply comparing the modular numbers themselves using a single machine instruction. The reason for this is that since $X' = Y'$, $X < Y$ if and only if $x < y$.

Further, if one maintains a group of quotient-normalized numbers with $-\delta < x < m + \delta$ where $0 < \delta \ll m$, then the quotient-normalized numbers can be incremented (or decremented) by small amounts without fear of overflow or having to consider renormalizing the group of numbers.

Because quotient-normalized modular numbers can be manipulated so efficiently, quotient-normalization makes it possible for Spline to gain the space efficiencies of a modular representation without paying a price of greater computation every time timestamps are compared and incremented.

5.3 Periodic Renormalization

One aspect of quotient-normalized modular numbers that is more complex than remainder normalized modular numbers is that the notion of normalization is a group phenomenon. If it is decided that the quotient Q has to be changed, then every modular number in use must be simultaneously changed. However, the amortized cost of this need not be great, because renormalization need not be frequent.

Specifically, with a modulus of 1 week, quotient-normalized timestamps only have to be renormalized once per week—a negligible expense. (One reason Spline's timestamp modulus is

chosen to be large is to guarantee that renormalization will be a very infrequent event.) The only complexity involved is that Spline must keep track of precisely where every timestamp is stored so that it is prepared to renormalize them when necessary.

At the start of each cycle of operation of the system, Spline calculates the remainder normalized time modulo one week. Whenever the result is less than the modular time corresponding to the last operational cycle, this indicates that time has moved from the end of one week to the beginning of the next week. To compensate for this, Spline renormalizes all the timestamps currently stored by subtracting one week from them. This guarantees not only that the timestamps are always quotient normalized, but also that they are almost remainder-normalized in the sense that $-\delta < x < m + \delta$ for a δ on the order of no more than a second or so.

Note that the check for the need to renormalize happens very seldom in comparison to the number of times that individual timestamps are manipulated and that the number of times renormalization actually has to be performed is extremely small.

5.4 Adjusting For Clock Differences Between Machines

Spline neither assumes nor enforces a rigid synchronization between the clocks on different machines. As a result, it is entirely possible for two machines running Spline to disagree on what week the current time is in and therefore on the appropriate value of Q. In particular, when one machine shifts to a new week and moves all its timestamps into that week, it may continue to receive timestamps in the previous week from other machines for some time. This has to be detected so that when these timestamps are read into the machine they can be converted to the same week as the other timestamps in the machine.

As a basis for detecting when timestamps in incoming messages have to be renormalized, Spline assumes that while the clocks on the various machines in a Spline session need not be rigidly synchronized, they must be approximately synchronized. Specifically, Spline assumes that the clocks on two communicating machines differ by no more than 24 hours. In comparison to a modulus of 1 week this is a relatively small difference, and makes it possible to unambiguously determine the relative values of Q on two machines. (Allowing for large differences between the clocks on communicating machines is a second reason why a large modulus is used by Spline.)

In particular, the values of Q on two machines cannot differ by more than 1. If they differ, then it must be the case that one machine is near the end of a week and the other machine is at the beginning of the next week.

Suppose that machine M has just switched from one week to the next. In this situation, the timestamp representing the current time on M will be small. If M receives a message with a timestamp referring to the same week, the timestamp will also be small and all is well. However, if M receives a message with a timestamp corresponding to the prior week, then the timestamp will be large because it must refer to the end of the prior week. Observing the large timestamp, M knows that this message comes from a machine that has not yet made the switch to the current week. M adjusts the timestamp in the message forward to the current week, by subtracting the modulus from it.

Alternatively, suppose that M is nearing the time when it will switch to the next week. In this situation, the timestamp representing the current time on M will be large. If M receives a message with a timestamp referring to the same week, the timestamp will also be large and all is well. However, if M receives a message with a timestamp corresponding to the next week, then

the timestamp will be small because it must refer to the beginning of the next week. Observing the small timestamp, M knows that this message comes from a machine that has already made the switch to the next week. M switches the timestamp back to the current week, by adding the modulus to it.

Note that testing whether the times have to be renormalized in a message is a relatively simple operation and that this operation only has to be applied when messages enter a Spline process, not when manipulating individual timestamps in the process. Further, actual renormalization of times in a message very seldom has to occur, because communicating pairs of machines almost always agree on what week it is.

It should also be noted that adjusting for clock differences between machines consists of more than just insuring that the timestamps in use are appropriately quotient-normalized. In particular, a given Spline process maintains estimates of the differences between its clock and the clocks used by each of the Spline processes sending data to it. These estimates are used to adjust the timestamps in the messages received so as to cancel out clock differences. For example, if machine M estimates that the clock on another machine M' is running 150 seconds behind the clock on machine M, then M will add 150 seconds to every time specified by machine M'.

The clock difference estimates are determined based on an additional timestamp field in each message. This field contains the exact time the message was sent (according to the sending machine). A simple difference between this field and the time of receipt (according to the receiving machine) is used as the basis for estimating the clock difference between the sending and receiving machines. This method of estimation deliberately conflates the actual clock difference with the transit time of the message. This is done because it is just as essential to adjust for transit time as for clock differences. Because the transit times of individual messages between two machines can vary significantly, time difference estimates derived from many different messages are statistically combined to yield a reliable estimate of the true time difference between a given pair of machines.

For example, suppose that the clock on machine M' is running 200 msecs behind the clock on machine M. Suppose further that messages from M' to M take an average of 11 msecs to go from one machine to the other. This would result in M having a clock difference estimate for M' of 211 msecs, which in turn would cause 211 msecs to be added to every timestamp sent from M' to M. This guarantees that messages sent from M' that (from the perspective of M') contain times immediately in the future will appear to M to contain times that are immediately in the future as opposed to in the past.

In addition to a single timestamp on a message indicating when it should be used, a message can contain additional timestamps used for other purposes such as specifying when some activity should stop. These additional timestamps are adjusted and renormalized in exactly the same way as the main timestamp.

6 Efficient Buffering of Chunked Data

Once the timestamps in a message have been normalized, all that remains for Spline to do is ensure that the data in the message is used when specified. For most kinds of data, this is simple to do. The messages are simply stored in a buffer until the time specified by their timestamps, at which time they are retrieved for processing.

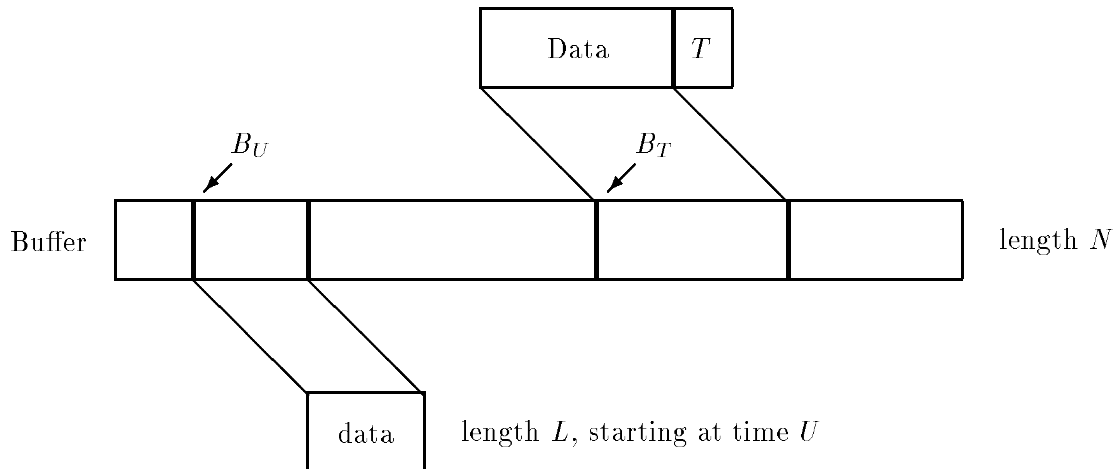

Figure 3: The buffering of sound data.

However, to efficiently handle chunks of sound data, more complex processing is required as depicted in Figure 3. The central difficulty with sound data is that there is a lot of it. A key thing that needs to be avoided is repeatedly moving the data around from place to place in memory. When a message with a chunk of data in it is received, this data must be moved to some buffer for storage until such time as it can be used. It is highly desirable to insure that the data is moved only once before it is used, rather than moving it several times, because each time the data is copied, a significant amount of computation is used.

The top of Figure 3 depicts a chunk of data with a renormalized timestamp T. The data is copied into a portion of a buffer whose total length is N msecs. Typical values of N are from 1000 to 2000 msecs, i.e., 1 to 2 seconds.

The buffer is treated as a ring with the position B_T of the data in the buffer calculated by computing the remainder of T divided by N. It will be appreciated that T itself was computed by taking the remainder of dividing an absolute time $\mathcal{T}$ by the timestamp modulus $m = 1$ week in milliseconds. Thus $B_T = (\mathcal{T}$ modulo $m)$ modulo N. In general, this kind of double use of modular arithmetic can lead to a complex and confusing relationship between B_T and $\mathcal{T}$. However, in Spline, it is required that N be a divisor of m. When this is the case, then ($\mathcal{T}$ modulo m) modulo $N = \mathcal{T}$ modulo N and thus B_T simply equals $\mathcal{T}$ modulo N.

To satisfy the requirement that N be a divisor of m, it is helpful to select a timestamp modulus that has many divisors. In this regard, a modulus of 1 week in msecs, which equals $2^{10} \times 3^3 \times 5^5 \times 7$, is particularly convenient. The wide range of divisors afforded by this number makes it possible to obtain a buffer length approximately equal to almost any desired length.

The bottom of Figure 3 shows a chunk of data of length L msecs, being read out of the buffer for use at time U. The location B_U of the data in the buffer is computed by taking the remainder of U modulo N. Two aspects of this are worthy of note. First, using modular arithmetic to compute B_U effectively retrieves the data that was most recently placed in the buffer at the specified position. For this to work, the buffer has to be large enough that the data corresponding to some particular time will have a chance to be read out before data about some future time overwrites it. For example, if a system operates in such a way that sound

data is never sent until less than 1 second before it should be used, then a buffer length of a little more than 1 second will be sufficient to ensure that future data will not overwrite older data before the older data can be used.

Second, if the length L of the data chunks to be read out is chosen so that L is a divisor of the buffer length N, then things can be arranged so that the chunk to be read out will always be a contiguous piece of memory rather than wrapping around from the end of the buffer to the beginning. This makes it possible to use the data directly out of the buffer, rather than copying it to a staging area before use.

It should be noted that the approach shown in Figure 3 handles incoming messages even if they are received out of order. The data in each message is simply put in the appropriate place in the buffer, from which it can be read out, in order, later.

7 Conclusion

By taking advantage of the fact that the human perceptual system is strictly limited in the level of synchronization that it can detect and the fact that even under an assumption of non-stop operation, synchronization of data communicated in real time is a local process only requiring the comparison of timestamps that are near each other, it is possible to achieve synchronization using 32-bit modular timestamps. By using quotient-normalization instead of remainder-normalization, the advantages of reduced computation can be obtained in addition to reduced memory and network bandwidth requirements. The use of 32-bit modular timestamps was presented in the context of the Spline scalable platform for interactive environments, but could be useful in a wide variety of situations in which multimedia data has to be communicated in real time.

In a situation where the clocks on a group of communicating machines could be quite closely synchronized, it should be possible to support synchronization with 16-bit modular timestamps. In particular, if the sum of the maximum clock difference and the maximum message transit time were on the order of only a second or so, then it should be possible to operate with a timestamp modulus of 15 seconds which is $1000 \times 15 = 15,000$ msecs, which is less than $2^{14} = 16384$ and therefore allows quotient-normalized timestamps to be represented in 16 bits. Beyond the stringent requirements on clock synchronization, the only obvious problem with this would be that the total number of timestamps being stored at any one time would have to be kept small since renormalization would have to occur four times per minute.

In Spline, it was decided to go with the much larger modulus of 1 week in milliseconds for a number of reasons. Using a large modulus reduces the frequency at which renormalizations have to occur and allows Spline to tolerate large differences among the clocks on a group of communicating machines. Beyond this, the exact choice of 1 week has many virtuous properties. To start with, $2^{29} <$ 1 week in msecs $< 2^{30}$, which means that 1 week is close to the largest modulus that allows quotient-normalized numbers to be represented in 32 bits. In addition, the fact that 1 week in milliseconds has a wide range of divisors allows for a great deal of freedom in picking the buffer length N in Figure 3 and therefore also allows a great deal of freedom in picking L. Lastly, 1 week is a unit that is easy to remember and understand. In Spline we use numbers limited to 1 week in milliseconds not only to represent timestamps (which are largely hidden from the application programmer), but also to represent durations (which are a prominent part of the application program interface).

Acknowledgments

The author and David B. Anderson are the principal architects and implementors of Spline. Additional major contributions to Spline were made by John W. Barrus, Joe Marks, David Marmor, Michael A. Casey, and William S. Yerazunis.

References

[1] Anderson D.B., Barrus J.W., Brogan D.C., Casey M.A., McKeown S.G., Sterns I.B., Waters R.C., & Yerazunis W.S. (1996) "Diamond Park and Spline: A Social Virtual Reality System with 3D Animation, Spoken Interaction, and Runtime Modifiability", MERL TR 96-02.

[2] Barrus J.W., Waters R.C., & Anderson D.B. (1996) "Locales and Beacons: Efficient and Precise Support for Large Multi-User Virtual Environments", *IEEE Virtual Reality Annual International Symposium*, (Santa Clara CA, March 1996), 204–213, IEEE Computer Society Press, Los Alamitos CA.

[3] Calvin J., et al (1993) "The SIMNET Virtual World Architecture", *Proc. IEEE Virtual Reality Annual International Symposium*, 450–455, (Seattle WA, Sept. 1993).