Order Number 9304659

Placement of data on a CD-ROM for seamless multimedia applications

Fernandez, Sergio A., Ph.D.

City University of New York, 1992

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

PLACEMENT OF DATA ON A CD-ROM FOR SEAMLESS MULTIMEDIA APPLICATIONS

by

SERGIO A. FERNANDEZ

A Dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy,
The City University of New York

1992

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy

Date 7/2/92

T C Wesselkamper
Chair of Examining Committee
Professor. T. C. Wesselkamper

Date 7/2/92

Executive Officer
Professor. Stanley Habib

Supervisory Committee
Samuel D. Glazer

THE CITY UNIVERSITY OF NEW YORK

ABSTRACT

PLACEMENT OF DATA ON A CD-ROM FOR

SEAMLESS MULTIMEDIA APPLICATIONS

by

Sergio A. Fernandez

Adviser: Dr. T.C. Wesselkamper

This dissertation studies and proposes solutions to the problems that arise in performing multimedia applications in small Disk Operating System (DOS) based, personal computers using a Compact Disk Read Only Memory (CD-ROM) as the delivery mechanism.

The problems dealt with in this dissertation are principally due to the small Random Access Memory (RAM) capacity of the DOS, and the long time that a CD-ROM takes to access a particular sector (seek time).

The memory problem is addressed by an anticipatory paging Multimedia Storage Management system (MSM). The slow seek time of the CD-ROM is handled by an algorithm that places data on the CD-ROM, thus minimizing seeks. The placement of data is done in two steps: first, a compiler creates a sequential files containing sound and images. After that is done, a linker links the sequential files together. Finally, a Multimedia Performer plays multimedia seamless applications.

ACKNOWLEDGMENTS

Many thanks to Tom Wesselkamper for his invaluable advice; to my boss Sam Glatzer for his support; to my friends in Prodigy's technology department; to Ana and Cristina who had so much fun pressing the reset button and to Kimberly the protector of the reset button.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

1.1 Multimedia

Using Compact Disk Read Only Memory (CD-ROM), personal computers are able to play high quality digital sound and simultaneously display digital images. These types of applications are termed multimedia presentations. In its simplest form, a multimedia presentation resembles an audio-visual presentation with no interruption from the beginning to end. A more complex multimedia application allows the user to interact with the program by pressing the mouse button over "hot areas" of the screen.

This type of application is modeled in a natural way by a directed graph, wherein each vertex contains some amount of information (sound, images and text). The vertices are connected by directed edges, each of which is associated with some region of the screen, called an anchor in hypertext terminology (Ni90). When users activate an anchor they follow the associated link to its destination vertex, thus traversing the graph. It is desirable to make the traversal as seamless as possible. If the traversal can be done in a seamless way, the result is a customized audio-visual presentation.

The digital recording of high definition images and sound requires enormous amounts of disk space. Because of their large storage capacity and low cost, CD-ROMs are presently an excellent medium to store and distribute multimedia applications.

1. 2 Characteristics of the CD-ROM

A single CD-ROM can hold 550 Mega Bytes (MB) of data. To put this in perspective, 550 MB is the equivalent to 150,000 printed pages of text. A CD ROM is a 5.25 inch plastic disk that stores data by burning small holes into the disk. Data is retrieved by reflecting a laser beam onto the surface of the disk (Le86).

To understand the characteristics of today's CD-ROM, it is necessary to realize that most of the CD-ROM technology comes from the consumer electronics industry.

In the late 1960s and 1970s, home video disks were developed as accessories to television sets. Several of those systems were marketed, but only Laser Vision (LV) survived. LV disks are usually twelve inches in diameter, and contain thirty or sixty minutes of programming per side depending on the format. The thirty minute format is called constant angular velocity (CAV). In CAV, the disk rotates at 1800 revolutions per minute (RPM). The density of the recording is greater at the center of the disk than at the outset of the disk. A revolution on the disk stores a single frame. This allows for random access to individual frames. The sixty minute format is constant linear velocity (CLV). In this format, the density of the recording is constant throughout the disk, allowing it to store sixty minutes. In CLV, in order to maintain the linear density of the information constant, the disk rotates at different speeds, slower in the center and faster at the outset. Random access is not possible in this format. In order to find a track, first the head has to move, then the speed of the disk needs to be synchronized; only after the speed is synchronized it is possible to know the exact position of the head. Unfortunately, it may not be the desired track and the

process may need to be repeated various times. All this translates to very slow seek time for CLV.

During the 1980s, as audio Compact Disk (CD) matured, many people began to think that a modified CD could be used as a distribution medium for large quantities of data. This modified CD could take advantage of the technology developed for CD, by sharing the same laser optics, servo-electronics, optical media, and disk mastering process. CD-ROM and CD-ROM drives could be manufactured in a relatively inexpensive way. The only thing that was needed was to improve the reliability of the CD (Le86).

CD-ROM is a close relative of an audio CD. An audio CD stores audio on the disk. A CD-ROM can store not only audio but data in the disk. The data in a CD-ROM can be programs, image files, and all kinds of digital files. In audio data, small errors are not relevant. If a single bit is lost, no one notices the difference. However, if the data stored in a CD-ROM has an error, the consequences can be disastrous. It is for this reason that a very reliable error correction scheme is used. This scheme called Layered Error Correction Coding (LECC), can take a signal coming from a disk in which one in every 10,000 bits is wrong, with bursts of wrong or missing bits over 1000 bits long, and regenerate all but one bit in every 10^{12} (Ha86).

Today's CD-ROM characteristics are very similar to audio CD. Because CD-ROM uses CLV it has a seek time of up to half a second. This is very slow when compared to 1/30 of a second for a commercial hard disk. A CD-ROM drive reads data at a rate of about 150 Kilo Bytes (KB) per second. This is acceptable when compared to 100 KB to 300 KB for a magnetic drive. A single CD-ROM can store about 540,000 KB. This is a lot when compared to most

commercial hard drives. Because the technology for a CD-ROM is derived from audio CD, it is relatively cheap to build CD-ROM drives, and to press CD-ROM disks. This price advantage makes CD-ROM an ideal medium to distribute multimedia to the home (La86).

The producer of a multimedia application should be concerned with the sequence of images, the sound, and the methods for interacting with the user. The target machine (multimedia player), is a virtual machine that encapsulates all the low level functionality.

To isolate the producer of a multimedia application from low level details, it is necessary to provide a high level language to describe interactions with the user and sequences of images and sound.

Among the tools needed to produce applications for the virtual machine are a compiler that translates programs, images, and sound into data that can be performed as an uninterrupted sequence, and a program to glue together the sequences, thus allowing seamless branching between them.

1.3 Performing Multimedia in a Personal Computer.

One of the basic limitations of a digital computer is the size of its available memory. In *most cases, it is neither feasible nor economical for a user to insist that every problem program fit into memory* (Be66). A multimedia sequence (MMS) is a sequence of images over sound, with no branching from beginning to end. A MMS is similar to an audiovisual presentation. To understand why memory is still one of the major limitations of personal computers, consider that a multimedia application, performing a two minute long MMS, may require up to eighteen MB of memory.

Memory requirements can be even higher if the storyboard requires the application to jump between MMSs, based on user interaction, and with no stop between the MMSs. For example, consider the MMS *A*, *B*, *C*, *D*, of thirty seconds each. If *A* is the initial MMS, in order to perform a seamless jump to *B*, *C*, or *D*, based on user input, all the MMSs need to be in memory before playing starts. This may require eighteen MB for sixty seconds.

1.4 Memory Characteristics of a Multimedia Application.

By studying computer processes it can be observed that *processes tend to reference storage in non uniform, highly localized patterns*. This concept is termed *locality* (Be66) (Ba76). *Temporal locality* means that storage locations referenced in the recent past are likely to be referenced in the near future. Supporting this observation are looping, subroutines, stacks and variables (De90). A unique characteristic of a Multimedia Application (MMA) running in a small personal computer is the lack of temporal locality of the process.

A small personal computer that uses 320 KB as primary storage to run an MMA can only store a loop that is two or three seconds long. Most sequences in MMAs are much longer than that, making it very unlikely that a storage page is referenced before it is pushed out of memory. This is especially true when the page contains image or sound data.

About ninety percent of the memory required by a MMA is used to store the images; 9.5 percent is used for sound, and 0.5 percent is used to store the programs.

1.5 Storage Management.

Multimedia Storage Management (MSM) has to solve three problems: storage organization, storage mapping and storage allocation. Storage

organization determines the way primary storage is divided. Mapping allows a program's logical space to be separated into units that may occupy non-contiguous portions of the primary storage. The allocation algorithm determines which parts of a program or program data must reside in primary storage.

1.5.1 Storage Organization.

The primary storage of MSM consists of 320 KB of random access memory (RAM) divided into 160 pages of two KB each. 320 KBs can hold data for about two seconds of images and sound. This is enough to cover the seek time of about 1.5 seconds for a CD-ROM.

The types of objects created by the compiler can be classified in three types: images, sound, and program/data. Blocks are also classified depending on what they store: image blocks, sound blocks and program/data blocks.

The following assumptions are made in the computation of the storage utilization.

- Images can be zero to sixty four KB.
- On average the last block of an image occupies half a block.
- The average image size is thirty-two KB or sixteen blocks.
- Sound is packed in two KB segments.
- On average blocks containing program/data use only 400 Bytes.

There are two reasons to choose a two KB size for the pages. The first one is that the CD-ROM is already formatted into two KB pages. The second one has to do with memory utilization.

To compute memory utilization the formula $u = \frac{2s_s s_p}{s_p^2 + 2s_s(1+s_p)}$, where u = utilization, S_s = segment size and S_p = page size, is used (HA88).

To compute the storage utilization of MSM we need separately to compute image utilization (I_U), sound utilization (S_U) and programs/data utilization (P_U) and then multiply them for the expected percentages of images, sound and program/data blocks.

For images $S_S > S_P$, $I_U = 0.97$, for sound $S_S = S_P$, $S_U = 1$, for program/data $S_S > S_P$, $P_U = S_S / S_P = 0.2$. The MSM memory utilization is 0.97 * 90 /100 + 1.0 * 9.5 / 100 + 0.2 * 5 / 100 = 98 %.

For four KB pages, the utilization drops to 95% and for one KB pages it goes up to 99%, but the overhead of creating twice as many pages does not justify reducing the page size.

1.5.2 Storage Mapping.

Traditionally, page mapping is managed by dividing the pointers to objects in two parts: the first part contains the page number, and the second part contains the displacement within the page. When an object needs to be accessed, its real address is computed utilizing a page map table to find the page's real starting address and then adding the displacement.

Handles are unique numbers assigned to an object when the objects are created by the compiler. In MSM, handles are used to access the object. As objects are brought into RAM and instantiated, they register themselves with the object manager. The object manager maintains a table of handles to real addresses. When access to an object is needed, the object manager looks in the table and returns the address to the object. When objects are destroyed, their handles are removed from the table. Removing the handles is done to avoid supplying invalid addresses. Because the MMAs are required to run in a seamless way, no page faults are allowed. A fatal error occurs when the object

manager does not have an entry in its table for a handle. Using handles over virtual addresses results in a smaller table, since the number of objects in the system at any given time is smaller than the number of entries needed to reference the virtual memory space, considering that the virtual space of an application can be 500 MB.

In a paging/segmentation system, a virtual address comprises three parts: the segment, the page and the displacement. To find the real address of an object, a segment table is used to map the segment to a page map table for the segment. The page portion of the address is subsequently used to find the real address of the page. Finally, the displacement is added to the page's real address to find the object's real address (De90).

In MSM, segments occur in image objects. Each image object contains a segment table. Before the object is instantiated, the value for each entry in the segment table is the block number that contains the data for the image. When a block is loaded, it first sends a message to the sound object to see if the block contains sound. A sound object contains a table of the initial blocks number and after that, each block is placed at a constant interval making it possible to know if a block contains sound by doing a modulus operation with the interval and the block number. If the block does not contain sound, a message is sent to each incomplete image until the block is claimed by an image. When the message is received, the image object compares the incoming block number with the blocks numbers contained in its table. If the block belongs to the image, the segment table is updated. If the block does contains neither sound nor image then it must contain programs and data objects. Program/data blocks have at the beginning

a description of each object that they contain. This description is used to instantiate the objects.

1.5.3 Storage Allocation.

Deitel classifies storage management strategies into:

1. Fetch strategies
    a) Demand fetch strategy
    b) Anticipatory fetch strategies
2. Placement strategies
3. Replacement strategies (De90)

Fetch strategies are concerned with loading the next piece of program data from secondary storage to primary storage.

Demand fetch strategies load the program data when it is referenced by a running program. Deitel presents several reasons for this strategy (De90). Demand paging guarantees that the only pages brought to main storage are those actually needed by the process. The overhead involved in deciding which pages to bring to main storage is minimal.

Anticipatory page fetch strategies might require substantial execution overhead. Computability results, specifically the halting problem (Le81), tell us that the path of execution of a program cannot be predicted. Therefore, any attempt to preload pages in anticipation of their use might result in the wrong pages being loaded. In anticipatory fetching, also called prepaging, the operating system attempts to predict the pages a process needs, and then preloads the page when the space is available. If the correct decisions are made, the total run time for the process is reduced considerably. Triverdi defines several practical prepaging algorithms that reduce the paging problem of array

algorithms operating in large arrays (Tr76). He also points out that, if the Central Processing Unit (CPU) utilization needs to be maximized, then the number of page faults is the most relevant performance measurement. This measurement is critical to the MSM since not a single page fault can occur while performing a seamless MMA. Smith takes advantage of the sequentially of a program to fetch into primary memory the pages that are likely to be accessed in the near future, before they are actually needed, thus improving system performance (Sm78). Multiple buffering has been very successful in accessing sequential files and data bases (Sm76). Sequential prefetching has also been applied to cached and other high speed memories. Look-ahead buffering of instruction fetching is used in many pipeline computers (Ha88). MSM is based on information produced by running a simulation of a multimedia application. The information produced by running simulations of programs has been used to find optimal replacement algorithms by Belady's MIN algorithm (Be66) and by Budzinski and Davidson's VMIN algorithm (Bu81). MSM uses the information to decide which pages should be prefetched and when to discard each page.

Placement strategies determine where in memory an incoming fragment or program of data should be placed. The goal of the strategy is to minimize fragmentation when loading segments of different sizes (Ra69). When primary storage is organized into pages, no fragmentation occurs, making the problem of placement trivial. MSM uses a page and segment storage organization.

Replacement strategies determine which portion of the program should be pushed out of memory to make space to the incoming programs. Knowing which pages can be deallocated is critical to the success of MSM.

1.5.4. Storage Allocation in MSM.

MSM uses an anticipatory fetching strategy based on information produced by the multimedia storage management production tools (MSMPT). There are two components to MSMPT; a compiler, and a linker. In the next sections, the fetch strategy, the compiler, and the linker are discussed.

1.5.4.1 MSM Fetch Strategy.

A Storage Extent (SE) is a sequential file formatted in two KB pages that contains the data for a MMS. All the fetch strategies that have been mentioned previously try to minimize either the number of page pulls, the number of page faults, or the memory utilization. The implicit assumption is that the secondary storage is fast enough to make it acceptable for the system to incur page faults. Because a CD-ROM driver may take up to 1.5 seconds to seek for a page, for MSM to perform a multimedia application, using a CD-ROM as a secondary storage, it is not acceptable to incur any page fault. Instead of deciding which pages to bring into memory at run time, it is necessary for the pages that are going to be pulled into storage to be placed sequentially in the CD-ROM. Having the pages in the right sequence reduces the time of a page pull to the time it takes to transfer two KB from the CD-ROM driver to memory $\frac{2KB}{150KB/Sec} = \frac{1}{75} Sec.$, eliminating the need to seek for different pages, and the need to guess which pages to pull. MSM tries to pull as many pages as it can fit into primary storage, and it is the responsibility of MSMPT to make sure that there is enough data in the pages to perform the application.

MSMPT comprises two programs; compiler and linker. The compiler reads a source file that describes a MMS (MMS.SRC) and creates a SE and a file

containing a list of branch out points (SEBP). The SE can be played and if seamless branching is not required, the SE can be recorded onto the CD-ROM. If seamless branching is required, the linker is used. The linker reads a SE and its SEBP file. For each branch in SEBP the linker places enough data in the SE to cover for the seek time of the CD-ROM. The output is a seamless storage extent (SSE). A SSE can branch to other SEs without delay.

1.5.4.2 The Compiler

The compiler first parses the MMS.SRC file and creates all the objects that are needed to perform a MMS. After all the objects are created, it runs a simulation and tells each object when it is needed and when it is no longer needed. After the simulation is run, the time of the MMS is used to create a SE with enough blocks to cover the entire time of the MMS (75 blocks per second). Once the SE is created, each object places itself as close to the point where it is needed as possible. To avoid having to keep in storage all the sound for a sequence image, data and sound block are interleaved. Sound blocks are placed at a constant interval. The size of the interval is determined by the amount of data per second needed to keep the sound playing.

1.4.3 The Linker.

Although it is not possible to predict the path of execution of a program, it is possible to create its *program schema*. A *program schema* can be created by observing the relationship between the program and the processes that the program generates. This should be done not from the point of view of what the process generates or computes, but from the point of view of where the process goes, or of the possible paths that it takes during execution.

If E = { e_1, e_2,...,e_n } is a finite set, called a set of *edges*, and V = { v_1, v_2,..., v_n } is a finite set, called a set of *vertices* or *states*, and *f* is a total function *f*: E -> V X V, then the triplet < E, V, *f* > is called a *graph*. If G = < E, V, f > is a graph with the additional property that for each element *v* of V, there is a *v'* in V such that f^{-1} (*v*, *v'*) in E exists, then G is called a *program schema*. Throughout this paper, we denote the cardinality of the set V as |V| = n and label the elements of V with the natural numbers 1,2,...,*n*. (We82).

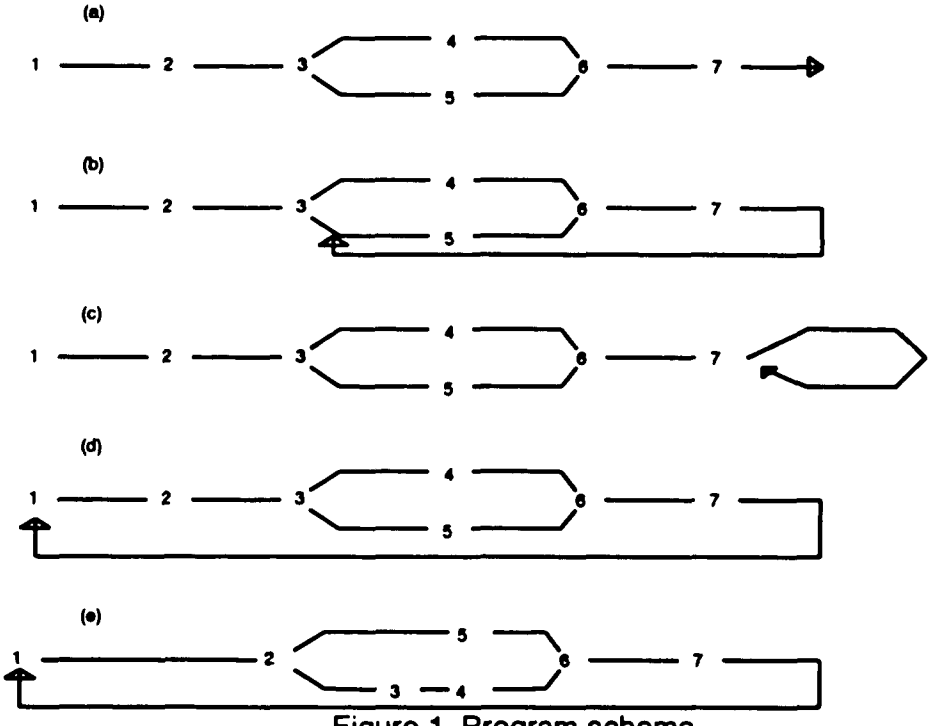

Figure 1. Program schema

The graph (a) is not a *program schema* since vertex 7 has no successor. The graph (b) is a program schema. Note that the set {1,2}, once left is never reentered (a transient set), and the set {3,4,5,6,7}, once entered, is never left (an absorbing set). The graph (c) is a program schema in which the singleton set {7} acts as a halt state. The graph (d) is strongly connected. Each vertex of this

graph has a cycle of length six, and each cycle of the graph is congruent to zero modulus six. The graph (e) is also strongly connected, but now there are cycles of lengths five (125671) and six (1234671), and so there are cycles of lengths $5a + 6b$ where a and b are non-negative integers.

A graph G = < E, V, f > is *strongly connected* if for each pair of vertices i, j, there exist a sequence of vertices $v_1, v_2,..v_m$ such that $i = v_1$, $j = v_m$, and for each k in ($1 <= k <= m$-1), the edge $f^{-1}(v_k, v_{k-1})$ exist, that is, there is a path from i to j along the edges of E. If $i = j$, the path is called a *cycle*, and a cycle of length one is called a *loop*. If all the vertices of a cycle are distinct, the cycle is called a *circuit*. There is no requirement that the vertices which make up a path be distinct. If for each (not necessarily distinct) pair of vertices i, j, the edge from i to j exists, then the graph is called *totally connected*. A totally connected graph is, of course, strongly connected. It is clear that every strongly connected graph is a program schema, but the converse is false (We82).

The *degree* of a vertex is the number of edge's incident to that vertex. In the case of a directed graph, the *in-degree* of a vertex v, Id(v), is the number of edges for which v is the head. The *out-degree*, Od(v), is defined to be the number of edges for which v is the tail. A *path* from vertex v_p to vertex v_q in a graph G is a sequence of vertices $v_p, v_{i1}, v_{i2},..., v_{in}, v_q$ such that $(v_p, v_{i1})(v_{i1}, v_{i2}),...,(v_{in}, v_q)$ are edges in E(G). A *simple path* is a path in which all vertices except possibly the first and the last are distinct. (Ho86).

A vertex v_i is a first vertex if Id(v) different from one, or if V_{i-1}, is a last vertex. A vertex v_i is a last vertex if Od(v) is different from one, or if V_{i+1}, is a first vertex. These definitions can be implemented by traversing the graph twice; in the first pass mark as first all the vertices that have an Id(v) $\neq$ 1, and mark as

last the sets that have Od(v) ≠ 1. In the second pass, mark as first the vertex whose V_{i-1} is a last vertex, and mark as last the vertex whose V_{i+1} is a first vertex.

A *sequence* (SEQ) is a simple path in which there is a first and a last vertex. A SEQ contains the vertices and all the edges between the first and last vertices. A vertex can belong to more than one SEQ. An edge cannot be part of more than one SEQ.

After a program schema has been built, the set of sequences {S} that have the following properties must be determined:

If G = < E, V, f >, there is a set of sequences $S = \{ s_1, s_2, ..s_n \}$ such that the intersection of *the edges in* s_i and the edges in s_j where i, j exist in n, is always null, and the union of the set of vertices in each element of S is equal to V.

A branch is an edge that connects independent sequences. The set of branches B is determined by $B = E - (ES_1 + ES_2 + ... + ES_n)$ for example to determine the branches in the next graph $E = \{ a,b,c,d,e,f,g,h,i \}$, $ES_1 = \{ a \}$, $ES_2 = \{ d \}$, $ES_3 = \{ e \}$ $ES_4 = \{ h \}$,

$B = \{ a, b, c, d, e, f, g, h, i \} - (\{ a \} + \{ d \} + \{ e \} + \{ h \}) = \{ b, c, f, g, i \}$.

For example in:

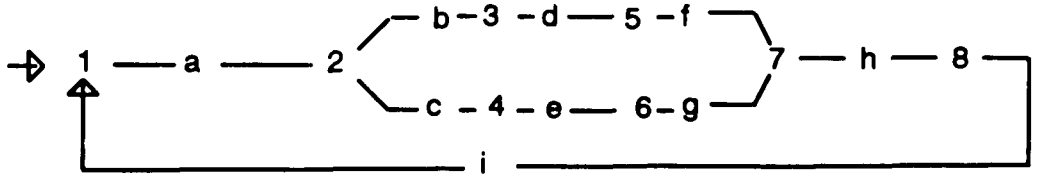

Figure 2. Sequence

The sequences S = {s_1,..s_4} are:

	vertex	Edge	Branch
s_1 =	1, 2	a	b, c
s_2 =	3,5	d	f
s_3 =	4, 6	e	g
s_4 =	7,8	h	i

When a branch is selected, the drive moves the head to the new position in the CD-ROM. In order to provide seamless branching, it is necessary to prefetch some data to cover the time when the drive is not reading (seek time). For example, in the previous graph, s_1 needs to prefetch the resources to cover the start time for s_4 and s_2.

The function R(v,t) determines the amount of resources needed by the linear sequence s_n to perform for time t. The variable t is the time that the sequence needs to start playing, including seek time and setup time.

The amount of prefetching needed by a sequence (SP) is determined by adding the prefetching needed for each branch in the set *Bv*.

Sequence load (*SL*) is determined by the amount of data that is needed for a MMS. Sequence Time (*ST*) is the time it takes to play a MMS. Sequence Capacity (*SC*) if the amount of data that can be transferred from the CD-ROM in

the time it takes to perform MMS, and is determined by the equation *SC* = *ST* x 150MB. Sequence free space (*SF*) is determined by *SF* = *SC* - *SL*.

The success of MSM depends on two factors, the first one has to do with the amount of data that can be transfer between the CD-ROM and main memory, the second has to do with the amount of memory available to store the prefetched information.

For MSM to work, *SF* must be at least large enough to cover for *SP* when *v* in *Bv* is One. This allows seamless branching to at least one target. If seamless branching is required for all targets *SF* must be greater than the sum of all the *SP* for each branch. If *SF* is less than one page, the algorithm is not implementable since MSM works in pages. *SF* refers to data transfer between the CD-ROM drive and memory. The other limiting factor is blocks of memory available to MSM to hold the prefetched blocks at run time. The number of memory blocks used at run time is determined by traversing the blocks in sequence, adding one to a memory counter every time that a block containing program data or image data is encountered, and decreasing the counter by the size of an image every time that an image is used. Images are used at node activation. The node activation time is kept in milliseconds and it is converted into blocks by multiplying it 150, dividing it by 1000, and adding the number of blocks to time zero. The number blocks needed for sound is constant since the SS releases a block every time a block is loaded. Synchronization is not an issue because the if the CD-ROM drive is too fast and memory is exhausted, the CD-ROM handler pauses. A report generated by the compiler (Appendix B) prints the number of blocks in use at every block. The linker generates a report (Appendix C) based on the one generated by the compiler, adding to the

memory needed at each block of the sequence the memory needed to store the blocks that contain prefetched information. If the number of memory blocks needed exceeds the number of blocks available to the performer the linker will print an error. The application designer can modify the application to make it possible to branch in a seamless way.

The primary storage efficiency (*SEf*) for MSM allocation for a sequence can be determined by $SEf = \frac{SL + S^{Sel}}{\sum_{1}^{nb} SP + SL}$, where ***nb*** = number of branches, and ***Sel*** = selected branch. For a run of ten seconds with two branches and an average load of sixty percent. *SL* = 10 Sec x 60% x 150 KB/Sec = 900 KB., *SP*= 2 Branches x 1.5 Sec x 60% x 150 KB/Sec = 270 KB, $SEf = \frac{900KB + 135KB}{270KB + 900B} = 0.88$ This means that twelve percent of the pages pulled into memory are the wrong pages.

Seamless branching can be successful in applications that maintain a sequence for at least five seconds and have a small number of branches. If the number of branches increases, more memory is needed to keep the prefetched sequences. If the sequence is smaller, not enough space will be available to store the prefetched sequences, since the maximum number of interactions that a human being may engage in during a given time, is much greater than the number of seamless branches that can packaged by the algorithm. In application like video games, prefetching can be used to determine the next sequence of the game. Depending on a score, a sequence determines the background and sound that plays while the user interacts with the game. This allows the user to have many interactions, minimizing at the same time the number of branches.

The main difference between the program schema of an MMA and that of a more common computer program is the size of the sequences. In an MMA, the time to run a sequence is very long, from four seconds to several minutes. In a computer program, a *ST* is measured in nanoseconds. Since *SP* is determined ultimately by *ST*, in a more common computer program is always less than a page of memory. If this algorithm is to be used in a paged memory, the *SP* is one, since it is not possible to bring in to memory less than a page. For a normal computer program, *SP* is equal to one page, SL is always just a few bytes since it is determined by *ST*. Assuming that there are 100 bytes between jumps, *SL* is 100 bytes, and that the page size is 2000 bytes.

$SE = \frac{100 + 100}{2 \times 2000 + 100} = 0.048$. It is likely that when the selected branch is executed, it also has a *SL* of 100 bytes. Therefore, the useful part of the prefetched sequence is SP_{Sel} = 100 bytes. . That means that out of every 100 pages brought into primary storage, only 5 are of any use. This makes it clear that this algorithm would not work for most computer programs.

1.6. Optimal Placement of Data.

A *run* is defined as a path in which all vertices are distinct. A run includes all the vertices and the edges between the first and the last vertex. Looking at the figure on page 16, it is possible create many different runs.

For example:

r1 = vertices *1,2,3,5,7,8* edges *a,b,d,f,h,i.*

r2 = vertex *4,6* edges c, *e* and *g.*

or

r1 = vertices *1,2,4,6,7,8* edges *a,c,e,g,h,i.*

r2 = vertex *3,6* edges *b,d* and *f.*

There is an optimal set of runs, R, that minimizes the amount of prefetching needed for each vertex. To find the optimal set, consider the entry point of a MMA, (specified by the author of the application), as the root of the graph. To build the optimal set of runs, R, start with the vertex at the root of the program schema. At each vertex, select the branch that requires the most prefetching as part of the run. Process all free vertices to create additional runs until no free vertices exist.

For example, in the graph on page 16 at vertex 2, the decision needs to be made whether to select 3 or 4 as the next vertex in the run. If R(*3,f*) (see page 16) is greater than R(*4,f*), vertex 3 should be selected because the total resources needed by edge *c* are smaller. The resources needed for *c* need to be preloaded together with the resources needed for vertex 2 before branching can occur.

The determination of which branch should be selected as part of the run is independent of probability. If, in the graph on page 15 , it is more likely that at

vertex 2, edge *c* is selected over edge *b*, and the resources needed for edge *b* are greater than the resources for edge *c* , *c* is selected as part of the run. However, this decision does not create the optimal set of runs R.

The compiler, linker and prototyper are implemented for sequences with branches at the end of the sequence.

CHAPTER 2

MULTIMEDIA EXPERIENCE

Vannevar Bush in an article published in *The Atlantic Monthly*, July of 1945, describes the operation of the human mind: "It operates by association. With one item in its grasp, it snaps instantly to the next, that is, suggested by the association of thoughts, in accordance with some intricate web of trails carried by the cells of the brain. It has other characteristics, of course; trails that are not frequently followed are prone to fade, items are not fully permanent, memory is transitory. Yet the speed of action, the intricacy of the trails, the details of mental pictures, is awe-inspiring beyond all else in nature." He was describing a device called memex that acts as "an enlarged intimate supplement to his memory". Multimedia has the capabilities of becoming the memex machine. Memex would have allowed users to follow an idea. MMA in general and seamless multimedia in particular allows the user to follow ideas by customizing a presentation every time the presentation is run. This customizing is based on the user input.

2.1 What the User Sees.

An example of a MMA can be a fashion show. In the show there is a band of music playing and models are showing dresses. In the live show, a model can see if there is interest in a dress by observing the audience. If there is interest in the dress, she may show the dress for a little longer. If the model takes longer showing a dress than expected, the band can improvise by playing a little longer. In the multimedia show there is music playing and pictures of the models are

being displayed on the screen. On the computer, the user shows interest by clicking the mouse.

On the computer as in real life, there is a limited window of opportunity to show interest. The window is defined by the time it takes the model to walk the stage. To be able to imitate the model, the MMA needs a script with both a default short scene and an optional scene for each dress. When the multimedia show proceeds in the default mode, it shows one dress after another. When there is interest in a the dress, the optional scene is shown.

In a non-seamless MMA, it is impossible to imitate the behavior of the model. There are two ways of packing the data for the show. The first one packs the default scenes one after another, and has to pause every time there is some interest in a dress. The second way packs the default scene followed by the optional scene. When packing is done in the second way, the computer has to pause every time a new dress is shown. Every pause stops the music and waits for the head to reposition before continuing with music and images. If the MMA is seamless, it imitates the behavior of the real fashion show and it does not pause whether a user shows any interest in a dress or not.

The types of applications that can be built using interactive multimedia are very similar to Interactive Video (IV). IV involves the combination of digital graphics and analog video/audio graphic signals (multimedia is all digital) stored on laser disk or captured live with a closed circuit camera. It is used in some consumer applications, such as interactive storytellers. These computer-controlled games let the user direct the outcome of the story by making choices at specific decision points. "The Gap" store uses this technology to introduce new clothing products. The consumer touches an article of clothing worn by an

actor on the screen. The touch screen entry retrieves more information (like color options) about that particular article of clothing. Training is a popular non-consumer application. Large corporations and schools use IV to teach everything from foreign languages to the repair of machinery (Er91).

2.2 Multimedia Resources.

MMAs are made of sound, images, and programs.

2.2.1 Sound.

An important characteristic of a MMA is the sound. Sound is provided using a sound card that provides sampled audio. The application sends digitized sound to the card, which plays the sound through an amplifier and speakers. The sound is prepared during the application development and stored in the CD-ROM. The sound is digitized using a technique called pulse code modulation (PCM). To digitize sound, an analog audio source is connected to the analog-to-digital converter (ADC). The ADC is a digital measurement of the instantaneous amplitude of the audio wave form. This digital amplitude is periodically sampled at a particular rate and stored in memory or in a file. On the output side, the samples are sent to a digital to analog converter (DAC) to become an audio wave form again. This wave form is sent to an amplifier and speakers. The fidelity of the sampled audio is determined by the rate at which the wave form is sampled and the number of bits used for each sample.

The sampling rate sets an upper bound to the high audio frequency that can be captured and reproduced by the system. In 1928 Hartley proved that the amount of information that can sent over a channel in a given amount of time is directly proportional to the channel's bandwidth. Nyquist proved that a channel whose bandwidth W can carry $2W$ separate voltages per second. Shannon

proved that if signals are sent with a signal power *S* over a channel perturbed by random noise of power *N* then the capacity *C* of the channel is $C = W Log_2(1+S/N)$ (Ma88). In order to record without distortion the sampling rate must be at least twice as high as the highest audio frequency. If an attempt is made to record a frequency higher than half the sample frequency, it is recorded as an alias. An alias is an audio frequency that is below half the sample frequency. To avoid this problem, a filter to limit the sound passing to the ADC to half the sampling frequency is used. Because humans beings can generally hear sounds up to twenty KHz, a sampling rate of forty KHz is needed to capture and reproduce sound over the full range of human hearing. Music stored on a CD is recorded with a sampling rate of 44.1 KHz. (Pe91)

The number of bits per sample determines the dynamic range of the sound. The dynamic range is the difference between the loudest and the softest sound. The dynamic range is measured in decibels (dB), which is a logarithmic scale that mimics the response of the human ear.

A bel is defined as a tenfold increase in sound intensity. A decibel is one tenth of a bel in equal multiplicative increments. Hence, one decibel is equivalent to an increase in sound intensity by a factor of the tenth root of ten, or approximately 1.26. One decibel is about the lowest increment in sound intensity that a human being can discern.

The value of the wave form sample determines the amplitude of the wave form. The sound intensity is the square of the wave form amplitude. Therefore, one decibel is equivalent to an increase in wave form amplitude by a factor of the twentieth root of ten, or approximately 1.12 (the square root of 1.26). The difference in decibels between two wave forms can thus be calculated as twenty

times the base ten logarithm of the ratio of the amplitudes of the wave forms. For example if a wave form has an amplitude that is 1.12 times another wave form, the difference in decibels is twenty times the base ten logarithm of 1.12 or one decibel. Audio compact disks use sixteen bit samples. With sixteen bit samples, the dynamic range is 96 dB. This is twenty times the base ten logarithm of 2^{16}. A range of 96 dB is a little less than the difference between threshold of hearing (twenty dB) and the threshold of pain (120 dB). With eight bit, the dynamic range drops in half to forty-eight dB. What this means is that the softest sound in an eight bit system must be twice as loud as the softest sound in a sixteen bit system in order not to degrade into noise. Even at a constant volume an eight bit sampling system has some noise (Pe91).

For the dissertation demonstration, a *soundblaster* board will be used. This board provides a PCM sampling rate of 8,000 samples per second. This results in a band width of four KHz. Four KHz is the band width of commercial amplitude modulated (AM) radio (St85).

At a sampling rate of eight KHz, one second of sound requires eight KB, one minute requires 480 KB. In order to avoid gaps, it is necessary to send data continuously to the digital sound processor (DSP). Direct Memory Access (DMA) is used in order to free the CPU from the task of sending data to the DSP.

2.2.1 Images.

There are only two ways to create an image in a computer. The first involves the description of the color of each dot or pixel on a given image, as in the Venetian mosaics, where images are represented by a large wall filled with small tiles of different colors. In the computer world these are known as raster or

pixel images. The second way is vector graphics. In this way, the image is encoded as a set of graphic objects that are decoded and painted every time the image is displayed.

Pixel images are very large. For a Virtual Graphic Adapter (VGA) screen using 320 x 200 pixels and 256 colors, sixty-four KB are required. This means that loading two full screen images per second requires 128 KB. Eight KB are required for sound. That leaves about fourteen KB per second for other data.

Pixel images look like photographs. For the dissertation demonstration pixel images are used. It is expected that future multimedia platforms would incorporate decompression boards. This makes it possible to have full motion video (by displaying twenty or thirty pixel images per second) utilizing about the same band width (150K per second) that is used in the dissertation demonstration. The *Communications of the ACM*, April 1991, has an extensive discussion of the algorithms and hardware used to compress and decompress images.

CHAPTER 3

THE RUN TIME PERFORMER

3.1 Introduction

The performer is the program that executes an MMA. The performer receives as input an SE created by the compiler or the linker, plays sound, displays images and interacts with the user. The performer can be studied from two different points of view: top down and bottom up. The top down approach, views the performer as a virtual machine that presents a simple high level abstraction to the application programmer. The virtual machine is much easier to program than the real machine. The virtual machine hides the details of the hardware, concealing interrupt handlers, timers, mouse handlers, and all low level functions of the system. The bottom up view presents the performer as a resource manager; handling interrupts, timers, memory, and other low level functions.

3.2 The Virtual Machine

An MMA is experienced by the user as a continuous display of visual images and sound. The screen is divided into regions. Some of the regions may exist inside other regions. Some of the regions change very often while others are almost static. In order to explain some of the concepts, a trivial example of an MMA is presented. The MMA example consists of a sound track and four different screens.

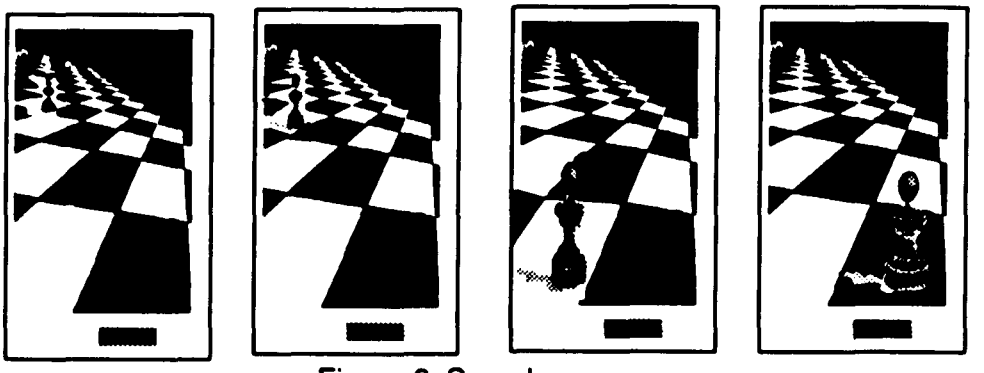

Figure 3. Sample sequence

In the example, each screen has three regions. The largest region is the background region. The other two regions are the command region and the picture region. In the first screen, the command region contains the word START. When the user left clicks with the mouse, on the command region, sound starts playing. After two seconds, the picture changes and the command changes to STOP. If the user does not click on the command region (in gray), the picture changes again after one second. After three seconds, the sound stops and the command region changes to RESTART. The four pictures displayed are picture 1, picture 2, picture 3 and picture 4. The commands are called START, STOP, and RESTART. The application and all its resources can be viewed in a static way and in a dynamic way. The static representation is an inverted tree with the resources that change the least at the root of the tree.

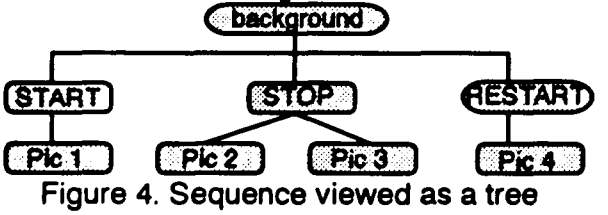

Figure 4. Sequence viewed as a tree

The dynamic view of the application can be represented by a time line.

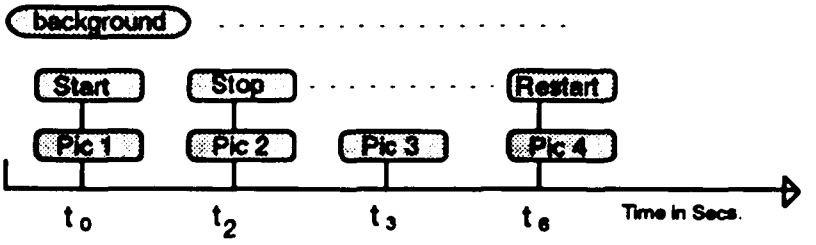

Figure 5. Sequence time line

The time diagram shows the resources that need to be loaded at any particular time. The time diagram also shows the active path of the tree. At t_0 the active path is *Picture 1*, *Start* and *Background*. At t_2 the active path is *Picture 2*, *Stop* and *Background*.

Each one of the vertices in the previous tree is called a *context*. A context contains resources. A region on the screen is a resource of a context. Other resources of the context can be text strings, event handlers, etc. A context has access to all the resources of its ancestors. The application programmer can modify the behavior of a context by writing event handlers. The topmost context, called the *system* context, contains event handlers for all the events. In the previous example, when the user clicks on START, the event handler for mouse click of the *start* context, has the necessary instructions to start the sound, and to set a two second timer. The other event handler of the *start* context is the timer event handler. This event handler executes two seconds after the beginning of the sound. The time out event handler contains instructions to activate the next context.

A special notation was developed to specify MMAs.

```
/* Ex1.src Sample for a simple sequence with four images */

context SYSTEM
{
  region  /*The Background region*/
     {
```

```
    color red;
    size   0, 0, 319, 199;
    }
  sound "demo.snd";

context Page4   /*New Screen */
{
     region
    {
    size   10, 01, 300, 180;
    picture "Pic4.m13";
    }
      context RESTART_BUTTON
    {
       region
         {
         size   50, 70, 80, 20;
         text   "RESTART";
         }
       event M_L_DOWN
         {
         goto_sequence"ex1";  /*This will restart
                              the extent*/
         }
    }
     }  /* End of Page 4*/
context Page3 /*New Screen*/
{
  event MyTimeOut
    {
    activate Page4;
    /*The sound will stop by starvation,
    when there is no more data to play*/
    }
  event INIT
    {
    start_timer;
    set_timer MyTimeOut, 3000;
    }
  region
    {
    size   10, 01, 300, 180;
    picture "Pic3.m13";
    }
   context STOP_BUTTON
    {
       region
         {
         size   50, 70, 80, 20;
         text   "STOP";
         }
```

```
        event M_L_DOWN
          {
          exit 1;
          }
      }
  } /* End of Page3*/
  context Page2   /* New Screen*/
  {
    event MyTimeOut
      {
      activate Page3;
      }
    event INIT
      {
           start_timer;
           set_timer MyTimeOut, 1000;
       }
    region
      {
           size   10, 01, 300, 180;
           picture "Pic2.m13"
      }
     context STOP_BUTTON
      {
         region
           {
           size   50, 70, 80, 20;
           text   "STOP";
           }
         event M_L_DOWN
           {
           exit 1;
           }
      }
 }/* End of Page2*/
context Page1   /*First Screen*/
 {
    sound "demo.snd";
    region
      {
      size   10, 01, 300,199;
      picture "Pic1.m13";
      }
      context START_BUTTON
      {
      region
     {
     color blue;
     size   50, 100, 80, 20;
     text   "START";
     }
```

```
        event MyTimeOut
             {
             activate Page2;
             }
        event M_L_DOWN
       {
       start sound;
       set_timer MyTimeOut 2000;
       }
}
```

3.3 The Resource Manager.

When the performer is studied as a resource manager, it appears as a small operating system. The performer manages devices, memory, and handles input and output. The performer is a message driven system. All the actions of the performer are executed by sending messages to the different subsystems. This message passing system enforces a very clean interface among the different components. The performer was designed using Object Oriented (OO) concepts and was implemented in C++. One of the subsystems that illustrates the advantages gained by the use of OO design is the message management.

3.3 1 Message Management.

The Message Manager object (MsgM) is the center of all activity. The MsgM contains a free queue, a timer queue, a high priority queue, and a low priority queue. In the constructor function of the MsgM, all the messages are linked together in the free queue. By using a queue for the free messages, the operation of receiving and dispatching messages does not involve any memory allocation. When a message needs to be posted or sent, a message is removed from the free queue by updating pointers; the data in the message is set, and finally, the message is placed in another queue depending on its priority.

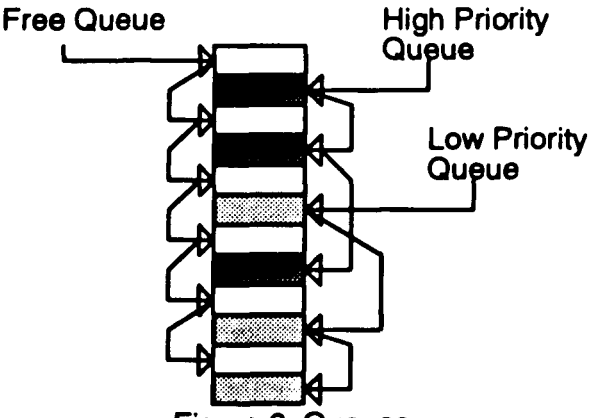

Figure 6. Queues

When a message gets dispatched, it is processed by the destination object and then it is placed back in the free queue.

Many different types of objects can receive messages. For example, when the mouse is pressed, the mouse interrupt handler posts a message for the screen object. When the screen object receives a mouse message, it locates the context that has a region below the mouse pointer, and sends the message to it. By deriving all the objects that can receive messages from the same parent class and declaring the *Receive Message* function of the parent as a virtual function, the message dispatcher does not need to know what class of objects receives the message. It only needs to call the *Receive Message* function using a pointer to the parent class.

Derived classes inherit all the characteristics of their parents (Ma87). This allows a subclass to be specified as a variation of a parent class. The subclass can then override the inherited operations or functions, add new functions or add new data members (De89). In the performer, each class of object that receives a message has a different implementation for the *Receive Message* function.

All the classes that receive messages are derived from the same parent class. A derived class can be pointed to using a pointer to a parent class. In the

previous example, the dispatcher uses a pointer for the parent class. This feature is called *polymorphism.* Polymorphism is the capacity for objects of more than one type to be assigned to a particular variable (Ko90). Complete polymorphism is easily accomplished by leaving the language typeless (e. g. smalltalk). C++ integrates classes and their instances into the language's strong typing. In C++, the degree of polymorphism is left to the designer of the class. Polymorphism in C++ can be defined as the capacity for objects of more than one type to be assigned to a particular variable as long as the variable is declared as a pointer to an ancestor of the objects to be assigned to the variable.

Ordinarily in C++ the member function called depends on the type of the pointer used to access it, not the actual type of the object to which the pointer refers. In order for the determination of the function called to be based in the actual object type, and not the type of the pointer used to access the object, the function should be a virtual function (Jo90).

Virtual functions provide the functionality that is needed for the *Receive Message* function.

If the type of a variable cannot be determined in advance and there is a virtual function being called, the compiler cannot determine which function to invoke. Polymorphism requires the ability to resolve function calls dynamically (i.e. during execution) rather than during compilation (Wi88). Languages that allow complete polymorphism normally use dynamic binding, though they may provide the ability to specify static binding where efficiency is required (as in Objective-C). C++ is a strongly typed language that uses static binding unless the function is declared a virtual function. To implement late binding, the C++

compiler generates a table of functions. At run time, the table of functions is dereferenced depending on the subclass of object to which the pointer is pointing.

The combination of inheritance, polymorphism and the dynamic binding of virtual functions facilitated the design and implementation of MsgM.

3.4 Performer Overview

All the components of the performer are objects. Some of them are part of the executable program and others are objects that are instantiated at run time utilizing the data from the SE. When a new object is created in C++, storage for the object gets allocated before the constructor executes. To implement this storage mechanism, the compiler generates code to call the default allocation function. This mechanism is not appropriate for the performer, because after allocating memory for an object, it is necessary to copy the data for the object into it, creating fragmentation since objects may be of different sizes. Fortunately, an alternate allocation mechanism exists. By assigning the C++ reserved variable *this* (St87) to a value in the constructor for the object, the C++ compiler is informed that the programmer has taken control over storage allocation and does not generate code to call the default allocation function. When objects inside blocks are instantiated, the constructor assigns *this* to the location where the data for the object was loaded, making it unnecessary to copy the data. A possible alternative to executing a constructor to assign *this* to the location of the data, is to assign the pointer to the location of the object. This only works for classes that do not have virtual functions because the pointer to the table of functions will not be initialized. The next graphic represents the main

components of the performer.

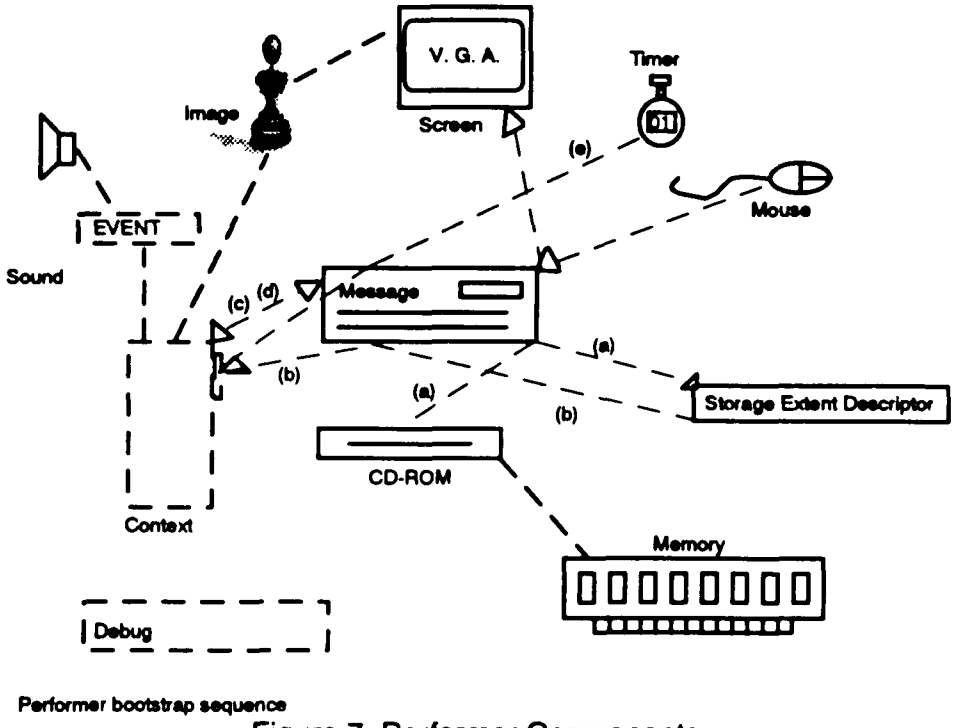

Figure 7. Performer Components

The process of bootstrap illustrates the interactions between the different objects that made the performer. When the performers start its execution, the first thing that happens, is initialization. At initialization, the following objects are created: Branch Manager (BM), CD-ROM, Debug Object (DO), Message Manager (MsgM), Memory Manager (MemM), Mouse Handler (MH), Object Manager (OM), Sound Manager (SM), Screen, System Context, Timer and VGA. After all the objects are created, the open function of the CD-ROM object opens and starts reading the SE. SEs, are placed in sequential files in the CD-ROM. If there is no Storage Extent Descriptor (SED) object, the CD-ROM object creates one. Since the CD-ROM drive may take up to 1.5 seconds to seek for the file location, opening a file is not a synchronous operation. Upon returning from the

open function, the kernel is entered. The kernel has an endless loop that calls the dispatcher function of the MsgM. The dispatcher's job is to send the highest priority message to the appropriate object.

When the CD-ROM drive is ready to load a block of data (seventy-two times per second), it raises an interrupt. Interrupt handlers have to be very fast since no other device can communicate with the central processing unit (CPU) while the interrupt is being serviced (Ta87). The CD-ROM interrupt handler gets memory from the MemM, loads a block of data and sends a *load* message to the SED (*(a)* on page 37).

It is up to the SED to process the in-coming blocks. The SED has the block numbers that belong to it, pointers to the sound descriptors (SD), images descriptor (ID) objects, and zero, one or more SEDs. When SED receives a *load* message it finds out if the message belongs to itself. If it does, a block object is instantiated using the memory address of the block. The constructor of the block object instantiates the objects inside the block. When objects are instantiated, they register with OM. OM maps the handle object to its real address. If the block does not belong to SED, SED tries to find out if the block belongs to the SD or to any imgage object by executing their processing methods. If the block does not contain sound or image data, then SED executes the processing method of BM to see if the block belongs to another SED. If the block does not belong to anybody, MemM frees the storage occupied by the block. If the block just received has a number equal to the *activate at block number variable*, an *activate* message is sent to the *begin* context (*(b)* on page 37).

Applications in the performer move by activating contexts. When a context gets activated, it executes its parent *activate up* method to place itself in the parent active list, then it executes its children *activate down* method. Activate up and down methods tests for an *activation flag* to avoid initializing a context multiple times. When a context is activated, it posts a *paint* and an *initialize* message to itself (*(c,d)* on page 37).

When the context receives a *paint message* (*(c)* in page 37). it finds any resource that needs to be painted and paints it. Paint messages are low priority to insure better user response.

Upon receiving an *Initialize message* (*(d)* in page 37), the context looks for an initialize processor. If it exists, the context executes it.

Execution of events is done by the *interpreter.* The interpreter decodes event instructions. Events are capable of starting sound, activating contexts, setting timers, etc. To perform a sequence of images over sound, the initialize event of the *begin* context usually starts the sound and sets a timer, the *time out* event activates the next context.

At every clock tick, the timer interrupt handler tests to see if the message at the front of the time queue has a time greater or equal to the current time. If this is the case, the message at the front of the queue is moved to the highest priority queue. The next time the MsgM dispatches a message, it sends this message to its destination.

At any point, the user can interact with the performer by clicking in the hot areas of the screen. When this happens, the mouse handler posts a *mouse down* message to the screen object. The screen object uses the mouse position to determine which context should receive the *mouse down* message.

3.5 Sound Subsystem

There are two major problems that have to be solved by the sound subsystem (SS): synchronization between images and sound, and interleaving of sound. The synchronization problem occurs because after a branch the time it takes to start reading the data is variable. After a branch the time it takes to start reading data is determined by the time it takes to move the head to its destination, the time it takes for the drive to get to the correct speed (CLV), and the latency or the time it takes for the data to appear under the head. Interleaving consists of placing blocks of sound data in the middle of other types of blocks. Interleaving is necessary because it is not possible (due to memory constraints) to load all the sound in advance.

The SS is implemented as a C++ object. The interface with the sound object consists of functions to receive sound data and to play sound. The *Add Block* function is used by SED to send sound blocks to the sound object. Because some CDs are faster than others, the application should be programmed to work with the slowest CD. This makes some fast CDs get ahead of the application. The SS uses a queue to store sound blocks, thus allowing the performer to load data until all blocks are used. When this happens, the CD-ROM stops moving the head until some blocks become available. This acts as a throttle mechanism for fast CD-ROM drives.

The SS contains the code to interface with the *soundblaster* card. This card has the capability to perform a loop of sound data formatted in blocks. SS and soundblaster share a circular queue of blocks. SS places data at least one block ahead of the block that soundblaster is playing. Soundblaster raises an

interrupt every time a new block starts, and the interrupt code increments a counter that is used to synchronize with SS.

3.6 Debug Object

The DO is a C++ object that uses the monochrome screen adapter MCA to display performer's debug messages and to interact with the developer.

3.7 Storage Management

As mentioned in chapter 1, most storage management decisions are taken by MSMPT. The only thing the performer needs to do is get rid of blocks as soon as possible. Image objects destroy themselves as soon as the image is painted, making all the blocks that contain image data available to the MemM. The SS gets rid of the blocks in its queue as soon as they are copied into the circular buffer of the soundblaster. Destroying images and sound frees most of the blocks. To reduce overhead, the remainder of the objects are not destroyed until the SED gets destroyed. A SED is destroyed when a new SE is opened. Since a single image may need up to sixty-four KB, (a fifth of the available storage,) if an image needs to be used more than once, it is more efficient to load it again from the CD-ROM than to keep it in memory.

CHAPTER 4

PLACEMENT OF DATA

4.1 Introduction

The input to the compiler is a source code file (SRC); files containing images and a sound file. The compiler processes the files to produce an SE. An SE is a single sequential file containing interleaved sound, images, event processors, and all the resources needed to run the SE. The performer, compiler and linker share all the headers that contain description of classes. Sharing headers facilitates the maintenance of the programs when a class description is modified.

When the SE is run by the performer, it appears to the user as a sequence of images and sound. There may be some areas of the screen where the user can move the mouse icon and click. That may cause the performer to jump to another SE. The jump to another SE requires opening a second file. This may take up to 1.5 seconds. During the time it takes to open the new SE, no sound is played and no new images are displayed.

After a SE has been compiled, it can be linked to a second SE. Linking an SE to a second or destination SE places some data of the destination SE extent into the first SE, allowing the performer to cover the time it takes to open the second SE by playing the initial sound and displaying the initial images of the destination SE.

4.2 The Compiler.

The compiler creates and packages objects into a SE. The compiler phases are: parsing and object creation, sequence creation, object placement, and output. Each of the phases takes as input the output produced by the previous phase.

Parsing and object creation takes as input source, images and sound files and are divided into:

1. Initialization. Initializes the system by creating the necessary system objects;

2. Lexical Analyzer. Implemented in LEX; transforms words into tokens;

3. Syntax Analyzer. Implemented in YACC, checks the syntax of the language. Annotations in the YACC source code call the semantics function to check the semantics and functions to create objects;

4. Semantic Analyzer. Checks the semantics rules;

5. Object Creator. Creates instances of the objects used by the performer;

6. Sequence Creation. Takes as input, objects and produces a sequence;

7. Object Placement. Takes as input, a sequence and creates a SE;

8. Output. The input to this phase is a SE and it produces a file containing a SE and some other files used by the CD-ROM simulator in the performer, and the linker.

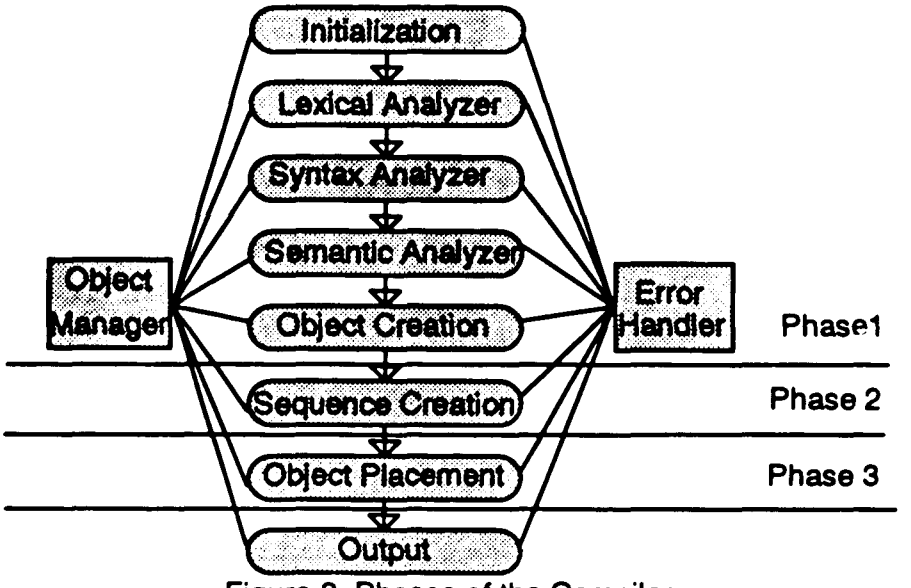

Figure 8. Phases of the Compiler

4.2.1 Initialization.

When the compiler is initialized, it creates DO and OM. The DO in the compiler uses the VGA instead of the MCA. The OM is exactly the same as the one in the performer. MemM is not created; instead, memory for the object is allocated out of the heap. The last job of the initializer is to open the source file.

The Lexical analyzer (Ah86) (Sc85) (Unix) is implemented by a finite automata recognizer created by the LEX compiler. The LEX compiler's input is a file containing the regular expressions that describe the tokens of the language.

The syntax analyzer is implemented in Yet Another Compiler Compiler (YACC) (Ah86) (Sc85) (Unix). YACC is a powerful utility of the Unix system that accepts a grammar specified in Backus-Naur Form (BNF). YACC checks the grammar and indicates problems that make the grammar unsuitable for recognition or ambiguous. If the grammar is accepted, YACC output is a 'C' file that contains a table driven look-ahead left to right (LALR) parser, and the tables that represent the grammar. The BNF grammar accepted by YACC contains

annotations. The annotations consist of calls to 'C' functions. The functions are called when a production is reduced. This group of functions comprises the semantic analyzer and the object creator.

4.2.2 Semantic Analyzer.

The semantic analyzers check the validity of the instruction. For example, context activation is only valid for context but it is grammatically correct to activate an integer.

4.2.3 Object Creator.

The object creator creates instances of the objects described by the source code. When an object is created, it registers itself with the OM. Some objects also register themselves with their owner. For example, a region places itself in a context and an image places itself in a region.

An event specifies the behavior of the system. The events implemented are: Left Mouse Down, Time Out, Initialize.

Event processors contain instructions. The instructions implemented are: Activate context, Operators +, -, =,x, ==, /, >, <, >=, <=, Set Timer, Go to sequence, and Start sound.

Picture objects contain the name of the file containing the digitized picture. Picture objects are transformed by the compiler into image descriptor objects (ID).

4.2.4 Creating a Sequence.

A SEQ (see page 15) when run by the performer is perceived by the user as sequences of images over sound. A SEQ consists of a link list of *nodes*. A node occurs at a particular time and contains all the resources that are needed to perform the sequence between one node and the next.

After all the objects are created, the context named BEGIN is found and it is added to the SEQ's first node. Adding a context to a SEQ is done by the node's *add node* function.

Add node receives two parameters; a pointer to the context to add, and the time at which the node's resources are expected to be needed. For the first node, the pointer points to the context named BEGIN and the time is zero.

Add node creates a node and places the node at the end of the link list of nodes. After the node is created, it sends to the context a message *add resources*. *Add resources* receives one parameter; a pointer to the node to which the resources of the context are added. Adding the resources of a context to a node is similar to the run time context activation by the performer. The goal for each node is to create a list that contains all the resources needed for the context, minus the resources loaded by previous nodes. A flag is used to avoid adding resources of a previously processed context. When a context is added, the tree of contexts is traversed towards the root until a context with a flag set is found. The tree is also traversed towards the leaves, adding the resources for each context to the node and setting the activation flag. The class *Context* is derived from the class *Container*. To add the resources of a container to a node, the function *process* is used.

Process receives as a parameter, a pointer to the node to which the resources of the context are added. *Process* adds the resources of the container to the node. When an event processor is added to a node, the event's *emulate* function is executed. *Emulate* is a modified event interpreter. When *emulate* encounters a *set timer* instruction, it creates a new node using the node's *add node* function. *Add node* is an overloaded function (De89b). The *add*

node function called by the event emulator receives two parameters; a pointer to the event that is executed when the timer expires, and the time at which the timer expires. *Add node* creates a node at the end of the linked list and call the event's *emulate* function for the event that executes at time out. When *emulate* encounters an *activate context* instruction, it executes the activated context's *add resources* function, passing to it the new node as a parameter. These recursive calls create all the nodes of the SEQ.

4.2.5 Object Placement

The size of SE is determined by the time it takes to play SEQ plus the number of blocks that need to be loaded before sound can start playing. SEQ's *find time* function is used to find SEQ time and SD's *offset to time zero* is used to determine the number of blocks that need to be loaded in the performer before sound can start.

The constructor for the sound descriptor determines the interval between sound blocks called interleave factor (IF), the number of sound blocks needed for sound in SE (NSB), and the offset to time zero (TZ). IF and NSB are based on the KB per second needed to play sound. The SED cannot process any sound data blocks until an SD is instantiated. The performer takes some time to instantiate an object after it is loaded. The number of blocks that go by the CD-ROM header during the time it takes to instantiate an object is termed blocks from load to instantiation (BLI). Time Zero is the time at which a sequence starts playing. Before a sequence starts playing some blocks need to be loaded and instantiated. The offset to time zero (OTZ) is the number of blocks that need to be loaded and instantiated before the sequences starts playing. OTZ is equal to BLI plus to the SD's queue size.

The SE constructor creates a SED and places the SED as the first object of the first block.

To guarantee that sound blocks are placed at fixed intervals, the first thing done after all the objects that form a SEQ are loaded, is to place the sound blocks. After SE is created, a SD's *place in SE* function is executed to place the sound in the SE. SD is placed in the first block of SE. The sound blocks needed for SD queue are placed immediately before the block at time zero. All the other sound blocks are placed at a constant IF. For example, if five sound blocks are needed for the queue, OTZ is eighty, and if IF is seven, sound blocks one to five will occupy positions seventy-five to seventy-nine; sound block six will occupy position eighty; block seven, position eighty-seven, etc.

Once the sound is in place, the SEQ first node's *place* function is executed to place the resources needed by the node in SE. For each entry in the node, the node's *place* function executes the node entry's *place* function. Finally the node entry's *place* function executes the SE *place right to left* function passing as parameters the time for when the node are needed, and a pointer to the object in the entry.

Place right to left looks for a block from right to left, starting at the block that is expected to be processed when the node is needed (BL). If no block exists, a block is created and the object is placed in it. If a block exists, the block's *place* function is executed. If the object fits in the block, the function accepts it. If the object is not accepted, BL is decreased by one and the process are repeated. The size of an object is obtained by a virtual *get size* function. This allows the blocks to know the size of an object without having to know the object type.

If the object is a picture, an ID is created by the block object. After the ID object is placed in the block, its *place in SE* function is executed. The ID's *place in SE* function reads the file containing the picture, formats the picture data into two KB blocks, places the blocks in SE, and places the ID in at least BLI blocks before the first image data block for the image.

4.2.6. Wiring the SE

Before the SE can be written, it needs to update the last object number in the SE, the object number for the BEGIN context, and the OTZ. Three files are written. The name in all three files is the same as the name of the source file. The extensions are *ext*, *map* and *seq*. The *ext* file contains the storage extent. The *map* file used by the CD-ROM simulator in the performer, maps the blocks in the *ext* file to the blocks as they would be layered out in the CD-ROM. The use of a *map* file minimizes the size of the *ext* file by avoiding writing blocks with no data. The *seq* file is used to tell the linker in which blocks the branches to other SE are located and the name of the target's SE.

The SE's *write* function executes the block's *write* function for each block. The block's *write* function checks to see if there is any event processor inside the block. If an event exists, the event is scanned searching for the *go to SE* instruction. If there is a *go to SE* instruction, the block number and SE target name are recorded in the *seq* file. As blocks are written a report is printed. The first column in the report is the block number. The block number tells when the block is loaded. The next column is the block number noting when the block is expected to be used. Since blocks for images and sound are discarded as soon as they are used, the total amount of storage needed by the performer to hold the blocks at run time can be computed using these numbers.

4.3 The Linker

The purpose of the linker is to take data from a target SE and place it in the source SE in order to cover for the time it takes to open a file by the CD-ROM drive.

The input to the linker is the name of the source SE.

The linker opens the source SE *seq* file and loads from it the SE targets in a table. Next, it opens the SE *map* file and creates a linked list of blocks. A node in the linked list is a block object (BO). A BO contains the block number, the source SE, and the physical address of the block in the source SE *ext* file.

Once this is done for each target SE, it places the target's blocks before OTZ in the source SE. To find out the target's OTZ, the first block of the target SE is loaded. The first object of that block of SE is the SED and contains OTZ.

Target blocks (TB) are placed from right to left starting with the block at OTZ. To place a TB, the source block (SB) before the branch occurs in the source SE is located. The TB is placed in the first space available to the left of it.

The next figure shows the linked list after a TB from SE No 2 is placed in the source SE No 1.

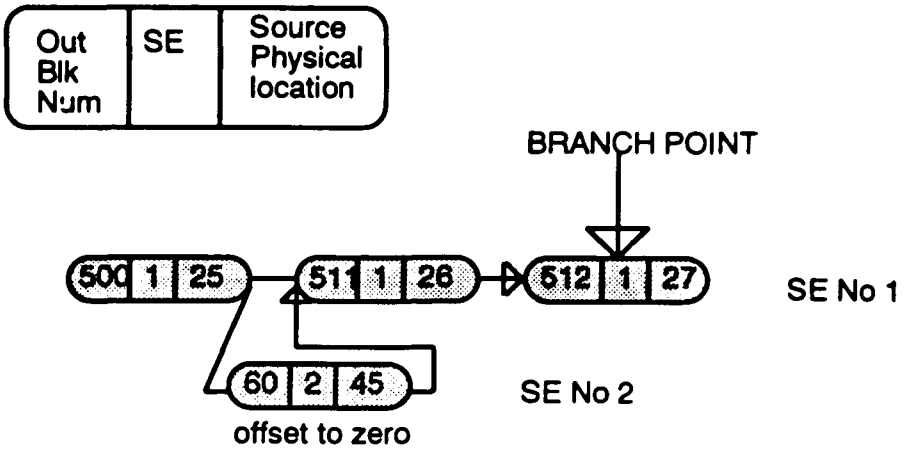

Figure 9. Block Insertion

After all the TBs are placed in the linked list, the linked list contains the blocks needed to create the new SE.

The objects in the blocks of target SEs have references to other blocks and need to be back-patched. For example, an ID has the block numbers of the image data. All its block numbers are invalid when the blocks from the target SE are moved to the source SE. Back-patching references to blocks is done when blocks are written. If a SED, ID, or SD is in the block, the particular object needs to be back-patched. To back-patch an object, for each block referenced by it, look at the reference original map file to find the position of the block in its original SE *ext* file; find the BO that represented that position in the linked list; and patch the reference with the location of the block in the new SE.

If it is not possible to place all the TBs in the source sequence, the application designer can place less or smaller images in the source sequence to decrease its load, it can increase the source sequence length, or place less or smaller images in the target sequences to decrease their load.

The other possibility of failure is due to the number of memory blocks available to the performer at run time. To guarantee that the SE fits, the linker generates a report (appendix B), in the report, for every block, the number of prefetched blocks is added to the number blocks needed by the source sequence. If the total number of loaded blocks is greater than number of blocks available at run time, the SE will not fit.

If the linker reports that there are not enough blocks of memory to run the application, the designer can decrease the number of MMS targets, or place fewer or smaller images at the beginning of the target MMSs to decrease their initial load.

If all the TBs are successfully placed in the source, and the number of blocks needed at run time is less than the number of blocks available to the performer, it is possible to do a seamless branching at run time.

CHAPTER 5

FUTURE RESEARCH

5. Future Research

To make the implementation of the performer easier, the branching was only allowed at the end of a sequence. A future implementation may allow branching in the middle of a sequence thus facilitating the construction of applications. This dissertation describes the use of CD-ROM to deliver interactive MMAs. As the cost of telecommunications decreases, there will be opportunities to deliver data streams through some data communication equipment with throughput similar to or greater than the data streams that can be obtained from a CD-ROM. Delivering these high data rates to multiple seamless interactive users will require very sophisticated algorithms to determine the placement of objects in direct access data storage devices.

There are other types of applications that do not need to be seamless, but where quick response is required. These can be optimized by analyzing the probabilities of a branch in the program schema and prefetching the objects needed by the most probable branches. The use of data compression boards will make it possible to have full motion seamless interactive MMAs delivered on CD-ROM. Finally another area of great interest is the applicability of the algorithms described in this dissertation to *windows* base operating environments like MS windows, OS/2 PM and XWindows.

LIST OF ABBREVIATIONS

ADC	Analog to Digital Converter
BL	Block
BO	Block Object
BLI	Blocks from Load to Instantiation
BM	Branch Manager
BNF	Backus-Naur Form
CAV	Constant Angular Velocity
CD	Compact Disk
CD-ROM	Compact Disk Read Only Memory
CLV	Constant Linear Velocity
CPU	Central Processing Unit
DAC	Digital to Analog Converter
DMA	Direct Memory Access
DO	Debug Object
DOS	Disk Operating System
DSP	Digital Sound Processor
ID	Image Descriptor
IF	Interleave Factor
IV	Interactive Video
KB	Kilo Bytes
KHz	Kilo Hertz
LECC	Layered Error Correction Coding
LEX	Lexical Analyzer
LV	Laser Vision
MB	Mega Bytes
MCA	Monochrome Adapter
MemM	Memory Manager
MH	Mouse Handler
MIN	Belady's Optimal Page Replacement Algorithm
MMA	Multimedia Application
MMS	Multimedia Sequence
MsgM	Message Manager
MSM	Multimedia Storage Management System
MSMPT	Multimedia Storage Management Production Tools
NSB	Number of Sound Blocks
OM	Object Manager
OO	Object Oriented
OTZ	Offset to Time Zero

PCM	Pulse Code Modulation
RAM	Random Access Memory
SB	Source Blocks
SC	Sequence Capacity
SD	Sound Descriptor
SE	Storage Extent
SEBL	Storage Extent Branch List
SED	Storage Extent Descriptor Object
SEf	Storage Efficiency
SEQ	Sequence
SF	Sequence Free space
SL	Sequence Load
SM	Sound Manager
SP	Amount of Prefetching Needed by a Sequence
SRC	Source File
SS	Sound Sub-System
SSE	Seamless Storage Extent
ST	Time to Play a Sequence
TB	Target Blocks
VGA	Virtual Graphics Adapter
VMIN	Optimal Variable Space Page Replacement Algorithm
YACC	Yet Another Compiler Compiler

APPENDIX A

Storage Extent
Node 0 time 0
Node 1 time 1000
Node 2 time 2000
Node 3 time 2600
Node 4 time 3300
Node 5 time 4100
Node 6 time 10100
Column one contains the block number
The next column is the number of memory blocks needed at run time
The Last column contains the data type D for programs or data, i for image data and s for sound data.
The indentation indicates the node that will use the data

0001 1 D
0036 2 i
0037 3 i
0038 4 i
0039 5 i
0040 6 i
0041 7 i
0042 8 i
0043 9 D
0044 10 i
0045 11 i
0046 12 i
0047 13 i
0048 14 i
0049 15 i
0050 16 i
0051 17 i
0052 18 i
0053 19 i
0054 20 i
0055 21 i
0056 22 i
0057 23 i
0058 24 i
0059 25 i
0060 26 i
0061 27 i
0062 28 i
0063 29 i
0064 30 i
0065 31 i

```
0066 32 i
0067 33 i
0068 34 i
0071          s
0072          s
0073          s
0074          s
0075          s
0076          s
0077          s
0078          s          Time Zero, Node zero is active
0089          s
0100          s
0111          s
0113   3 D
0122          s
0133          s
0138    4 D
0144          s
0155          s
0166          s
0172     5 D
0173  6   i
0174  7   i
0175  8   i
0176  9   i
0177          s
0178 10   i
0179 11   i
0180 12   i
0181 13   i
0182 14   i
0183    15 D
0184 16   i
0185 17   i
0186 18   i
0187 19   i
0188            s
0189 20   i
0190 21   i
0191 22   i
0192 23   i
0193 24   i
0194 25   i
0195 26   i
0196 27   i
0197 28   i
0198 29   i
0199            s
0200 30   i
0201 31   i
```

```
0202 32   i
0203 33   i
0204 34   i
0205 35   i
0206 36   i
0207 37   i
0208 38   i
0210        s
0215 39    i
0216 40    i
0217 41    i
0218 42    i
0219 43    i
0220 44    i
0221        s
0222 45    i
0223 46    i
0224 47    i
0225     48 D
0226 49    i
0227 50    i
0228 51    i            Node 1 is active
0229 20    i
0230 21    i
0231 22    i
0232        s
0233 23    i
0234 24    i
0235 25    i
0236 26    i
0237 27    i
0238 28    i
0239 29    i
0240 30    i
0241 31    i
0242 32    i
0243        s
0244 33    i
0245 34    i
0246 35    i
0247 36    i
0248 37    i
0249 38    i
0250 39    i
0254        s
0263 40     i
0264 41     i
0265        s
0266 42     i
0267 43     i
0268 44     i
```

```
0269 45      i
0270 46      i
0271 47      i
0272 48      i
0273 49      i
0274       18 D
0275 19      i
0276         s
0277 20      i                    Node 2 is Active
0278 21      i
0279 22      i
0280 23      i
0281 24      i
0282 25      i
0283 26      i
0284 27      i
0285 28      i
0286 29      i
0287         s
0288 30      i
0289 31      i
0290 32      i
0291 33      i
0292 34      i
0293 35      i
0294 36      i
0295 37      i
0296 38      i
0297 39      i
0298         s
0299 40      i
0309         s
0319 41       i
0320         s
0321 42       i
0322 43       i
0323 44       i
0324 45       i
0325 46       i
0326 15       i                   Node 3 is active
0327 16       i
0328 17       i
0329 18       i
0330       19 D
0331         s
0332 20       i
0333 21       i
0334 22       i
0335 23       i
0336 24       i
0337 25       i
```

```
0338 26       i
0339 27       i
0340 28       i
0341 29       i
0342         s
0343 30       i
0344 31       i
0345 32       i
0346 33       i
0347 34       i
0348 35       i
0349 36       i
0350 37       i
0351 38       i
0352 39       i
0353         s
0354 40       i
0355 41       i
0364         s
```

APPENDIX B

Block In Disk	Block In Extent	Original Position	Original Sequence	Original Required	Mem Prefetch Required	Mem Total Memory
0	1	1	0	1	0	1
1	36	36	0	2	0	2
2	37	37	0	3	0	3
3	38	38	0	4	0	4
4	39	39	0	5	0	5
5	40	40	0	6	0	6
6	41	41	0	7	0	7
7	42	42	0	8	0	8
8	43	43	0	9	0	9
9	44	44	0	10	0	10
10	45	45	0	11	0	11
11	46	46	0	12	0	12
12	47	47	0	13	0	13
13	48	48	0	14	0	14
14	49	49	0	15	0	15
15	50	50	0	16	0	16
16	51	51	0	17	0	17
17	52	52	0	18	0	18
18	53	53	0	19	0	19
19	54	54	0	20	0	20
20	55	55	0	21	0	21
21	56	56	0	22	0	22
22	57	57	0	23	0	23
23	58	58	0	24	0	24
24	59	59	0	25	0	25
25	60	60	0	26	0	26
26	61	61	0	27	0	27
27	62	62	0	28	0	28
28	63	63	0	29	0	29
29	64	64	0	30	0	30
30	65	65	0	31	0	31
31	66	66	0	32	0	32
32	67	67	0	33	0	33
33	68	68	0	34	0	34
34	71	71	0	34	0	34
35	72	72	0	34	0	34
36	73	73	0	34	0	34
37	74	74	0	34	0	34
38	75	75	0	34	0	34

39	76	76	0	34	0	34
40	77	77	0	34	0	34
41	78	78	0	34	0	34
42	89	89	0	34	0	34
43	100	100	0	34	0	34
44	111	111	0	34	0	34
45	113	113	0	3	0	3
46	122	122	0	3	0	3
47	133	133	0	3	0	3
48	138	138	0	4	0	4
49	144	144	0	4	0	4
50	155	155	0	4	0	4
51	166	166	0	4	0	4
52	172	172	0	5	0	5
53	173	173	0	6	0	6
54	174	174	0	7	0	7
55	175	175	0	8	0	8
56	176	176	0	9	0	9
57	177	177	0	9	0	9
58	178	178	0	10	0	10
59	179	179	0	11	0	11
60	180	180	0	12	0	12
61	181	181	0	13	0	13
62	182	182	0	14	0	14
63	183	183	0	15	0	15
64	184	184	0	16	0	16
65	185	185	0	17	0	17
66	186	186	0	18	0	18
67	187	187	0	19	0	19
68	188	188	0	19	0	19
69	189	189	0	20	0	20
70	190	190	0	21	0	21
71	191	191	0	22	0	22
72	192	192	0	23	0	23
73	193	193	0	24	0	24
74	194	194	0	25	0	25
75	195	195	0	26	0	26
76	196	196	0	27	0	27
77	197	197	0	28	0	28
78	198	198	0	29	0	29
79	199	199	0	29	0	29
80	200	200	0	30	0	30
81	201	201	0	31	0	31
82	202	202	0	32	0	32
83	203	203	0	33	0	33
84	204	204	0	34	0	34
85	205	205	0	35	0	35
86	206	206	0	36	0	36
87	207	207	0	37	0	37
88	208	208	0	38	0	38
89	210	210	0	38	0	38

90	215	215	0	39	0	39
91	216	216	0	40	0	40
92	217	217	0	41	0	41
93	218	218	0	42	0	42
94	219	219	0	43	0	43
95	220	220	0	44	0	44
96	221	221	0	44	0	44
97	222	222	0	45	0	45
98	223	223	0	46	0	46
99	224	224	0	47	0	47
100	225	225	0	48	0	48
101	226	226	0	49	0	49
102	227	227	0	50	0	50
103	228	228	0	51	0	51
104	229	229	0	20	0	20
105	230	230	0	21	0	21
106	231	231	0	22	0	22
107	232	232	0	22	0	22
108	233	233	0	23	0	23
109	234	234	0	24	0	24
110	235	235	0	25	0	25
111	236	236	0	26	0	26
112	237	237	0	27	0	27
113	238	238	0	28	0	28
114	239	239	0	29	0	29
115	240	240	0	30	0	30
116	241	241	0	31	0	31
117	242	242	0	32	0	32
118	243	243	0	32	0	32
119	244	244	0	33	0	33
120	245	245	0	34	0	34
121	246	246	0	35	0	35
122	247	247	0	36	0	36
123	248	248	0	37	0	37
124	249	249	0	38	0	38
125	250	250	0	39	0	39
126	254	254	0	39	0	39
127	263	263	0	40	0	40
128	264	264	0	41	0	41
129	265	265	0	41	0	41
130	266	266	0	42	0	42
131	267	267	0	43	0	43
132	268	268	0	44	0	44
133	269	269	0	45	0	45
134	270	270	0	46	0	46
135	271	271	0	47	0	47
136	272	272	0	48	0	48
137	273	273	0	49	0	49
138	274	274	0	18	0	18
139	275	275	0	19	0	19
140	276	276	0	19	0	19

141	277	277	0	20	0	20
142	278	278	0	21	0	21
143	279	279	0	22	0	22
144	280	280	0	23	0	23
145	281	281	0	24	0	24
146	282	282	0	25	0	25
147	283	283	0	26	0	26
148	284	284	0	27	0	27
149	285	285	0	28	0	28
150	286	286	0	29	0	29
151	287	287	0	29	0	29
152	288	288	0	30	0	30
153	289	289	0	31	0	31
154	290	290	0	32	0	32
155	291	291	0	33	0	33
156	292	292	0	34	0	34
157	293	293	0	35	0	35
158	294	294	0	36	0	36
159	295	295	0	37	0	37
160	296	296	0	38	0	38
161	297	297	0	39	0	39
162	298	298	0	39	0	39
163	299	299	0	40	0	40
164	309	309	0	40	0	40
165	319	319	0	41	0	41
166	320	320	0	41	0	41
167	321	321	0	42	0	42
168	322	322	0	43	0	43
169	323	323	0	44	0	44
170	324	324	0	45	0	45
171	325	325	0	46	0	46
172	326	326	0	15	0	15
173	327	327	0	16	0	16
174	328	328	0	17	0	17
175	329	329	0	18	0	18
176	330	330	0	19	0	19
177	331	331	0	19	0	19
178	332	332	0	20	0	20
179	333	333	0	21	0	21
180	334	334	0	22	0	22
181	335	335	0	23	0	23
182	336	336	0	24	0	24
183	337	337	0	25	0	25
184	338	338	0	26	0	26
185	339	339	0	27	0	27
186	340	340	0	28	0	28
187	341	341	0	29	0	29
188	342	342	0	29	0	29
189	343	343	0	30	0	30
190	344	344	0	31	0	31
191	345	345	0	32	0	32

192	346	346	0	33	0	33
193	347	347	0	34	0	34
194	348	348	0	35	0	35
195	349	349	0	36	0	36
196	350	350	0	37	0	37
197	351	351	0	38	0	38
198	352	352	0	39	0	39
199	353	353	0	39	0	39
200	354	354	0	40	0	40
201	355	355	0	41	0	41
202	364	364	0	41	0	41
203	375	375	0	41	0	41
204	386	386	0	41	0	41
205	397	397	0	41	0	41
206	408	408	0	41	0	41
207	419	419	0	41	0	41
208	430	430	0	41	0	41
209	441	441	0	41	0	41
210	452	452	0	41	0	41
211	463	463	0	41	0	41
212	474	474	0	41	0	41
213	485	485	0	41	0	41
214	496	496	0	41	0	41
215	507	507	0	41	0	41
216	518	518	0	41	0	41
217	529	529	0	41	0	41
218	540	540	0	41	0	41
219	551	551	0	41	0	41
220	562	562	0	41	0	41
221	573	573	0	41	0	41
222	584	584	0	41	0	41
223	595	595	0	41	0	41
224	606	606	0	41	0	41
225	617	617	0	41	0	41
226	628	628	0	41	0	41
227	639	639	0	41	0	41
228	650	650	0	41	0	41
0	656	1	2	41	1	42
1	657	36	2	41	2	43
2	658	37	2	41	3	44
3	659	38	2	41	4	45
4	660	39	2	41	5	46
229	661	661	0	41	5	46
5	662	40	2	41	6	47
6	663	41	2	41	7	48
7	664	42	2	41	8	49
8	665	43	2	41	9	50
9	666	44	2	41	10	51
10	667	45	2	41	11	52
11	668	46	2	41	12	53
12	669	47	2	41	13	54

13	670	48	2	41	14	55
14	671	49	2	41	15	56
230	672	672	0	41	15	56
15	673	50	2	41	16	57
16	674	51	2	41	17	58
17	675	52	2	41	18	59
18	676	53	2	41	19	60
19	677	54	2	41	20	61
20	678	55	2	41	21	62
21	679	56	2	41	22	63
22	680	57	2	41	23	64
23	681	58	2	41	24	65
24	682	59	2	41	25	66
231	683	683	0	41	25	66
25	684	60	2	41	26	67
26	685	61	2	41	27	68
27	686	62	2	41	28	69
28	687	63	2	41	29	70
29	688	64	2	41	30	71
30	689	65	2	41	31	72
31	690	66	2	41	32	73
32	691	67	2	41	33	74
33	692	68	2	41	34	75
34	693	71	2	41	35	76
232	694	694	0	41	35	76
35	695	72	2	41	36	77
36	696	73	2	41	37	78
37	697	74	2	41	38	79
38	698	75	2	41	39	80
39	699	76	2	41	40	81
40	700	77	2	41	41	82
41	701	78	2	41	42	83
42	702	89	2	41	43	84
43	703	100	2	41	44	85
44	704	111	2	41	45	86
233	705	705	0	41	45	86
45	706	113	2	41	46	87
46	707	122	2	41	47	88
0	708	1	1	41	48	89
1	709	36	1	41	49	90
2	710	37	1	41	50	91
3	711	38	1	41	51	92
4	712	39	1	41	52	93
5	713	40	1	41	53	94
6	714	41	1	41	54	95
7	715	42	1	41	55	96
234	716	716	0	41	55	96
8	717	43	1	41	56	97
9	718	44	1	41	57	98
10	719	45	1	41	58	99
11	720	46	1	41	59	100

12	721	47	1	41	60	101
13	722	48	1	41	61	102
14	723	49	1	41	62	103
15	724	50	1	41	63	104
16	725	51	1	41	64	105
17	726	52	1	41	65	106
235	727	727	0	41	65	106
18	728	53	1	41	66	107
19	729	54	1	41	67	108
20	730	55	1	41	68	109
21	731	56	1	41	69	110
22	732	57	1	41	70	111
23	733	58	1	41	71	112
24	734	59	1	41	72	113
25	735	60	1	41	73	114
26	736	61	1	41	74	115
27	737	62	1	41	75	116
236	738	738	0	41	75	116
28	739	63	1	41	76	117
29	740	64	1	41	77	118
30	741	65	1	41	78	119
31	742	66	1	41	79	120
32	743	67	1	41	80	121
33	744	68	1	41	81	122
34	745	71	1	41	82	123
35	746	72	1	41	83	124
36	747	73	1	41	84	125
37	748	74	1	41	85	126
237	749	749	0	41	85	126
238	750	750	0	10	85	95
38	751	75	1	10	86	96
39	752	76	1	10	87	97
40	753	77	1	10	88	98
41	754	78	1	10	89	99
42	755	89	1	10	90	100
43	756	100	1	10	91	101
44	757	111	1	10	92	102
45	758	113	1	10	93	103
46	759	122	1	10	94	104
239	760	760	0	10	94	104
240	771	771	0	10	94	104
241	782	782	0	10	94	104

BIBLIOGRAPHY

(Ah86) Aho A. V., Sethi R. Ullman J. D. Compilers Principles, Techniques and Tools. ch. 3. Adison Wesley, 1986. Reading, Ma.

(Ba76) Baer, J., and G. R. Sager, "Dynamic Improvement of Locality in Virtual Memory Systems," *IEEE Transactions on Software Engineering,"* Vol SE-1, March 1976, pp. 54-62.

(Be66) Belady, L. A. " A Study of Replacement Algorithms for Virtual-Storage Computer," *IBM Sys. J.*, vol. 5, pp. 78-101, 1966.

(Bu81) Budzinski, R.; E. Davidson; W. Mayeda; and H. Stone, "DMIN; An Algorithm for Computing the Optimal Dynamic Allocation in Virtual Memory Computer," I*EEE Transactions on Software Engineering*, VolSE-7, No 1, January 1981, pp. 113-121.

(De89) Dewhurst S. C. and Stark K. T. *Programming in C++*. pp. 101. Prentice-Hall 1989. Englewood Cliffs, NJ.

(De89b) Dewhurst S. C. and Stark K. T. *Programming in C++*. pp. 40-44. Prentice-Hall 1989. Englewood Cliffs, NJ.

(De90) Deitel, H. M. *An Introduction to Operating Systems*, Second Edition, pp. 189. Addison Wesley. Reading, MA.

(Er91) Erickson, D. J. "Multimedia the New Wave" *Windows and OS/2 magazine* Vol. 12, No 1. Feb/Mar, 1991. pp. 40-46.

(Ha86) Hardwick A. "Error Correction Codes: Key to Perfect Data," *CD-ROM the New Papyrus*. pp. 73-83. Microsoft Press. 1986 Redmond, WA.

(Ha88) Hayes J. F., *Computer Architecture and Organization.* pp. 429. Second Edition Mc Graw Hill, 1988. New York, NY.

(He77) Heinnie, F., *Introduction to Computability.* Addison-Wesley, 1977. Reading, MA.

(Ho86) Horowitz E. and Sahni S., *Fundamentals of Data Structures in Pascal.* ch. 6. Computer Science Press. Rockville, MR.

(Jo90) Jordan D. "Implementation Benefits of C++ Language Mechanism," *Communications of the ACM* Vol 33, No 9. September 1990. pp 61-64.

(Ko90) Korson Tim and McGregor J. D. "Understanding Object-Oriented: a Unifying Paradigm," *Communications of the ACM* Vol 33, No 9, *September* 1990. pp 41-60.

(La86) Lavander T. "CD-ROM Servo Systems," *CD-ROM the New Papyrus.* pp. 91-99. Microsoft press. Redmon, WA.

(Le81) Lewis H. R. and C. H. Papadimitriou. *Elements of the Theory of Computation*, Prentice Hall, 1981. Englewood Cliffs, NJ.

(Le86) Leonard L. "What is CD-ROM," *CD-ROM the New Papyrus.* pp. 47-71. Microsoft press. 1986. Redmon, WA.

(Ma87) MacLennan Bruce J., *Principles of Programming Languages.* pp. 273. Second Edition HRW 1989. New York, NY.

(Ma88) Martin James, *Data Communication Technology,* pp. 112-113. Prentice Hall. 1988. Englewood Cliffs, NJ.

(Ni90) Nielsen J. "The Art of Navigating Through Hypertext," *Communications to the ACM* Vol 33, No 3. March 1990.

(Pe91) Petzold, Charles, "The Multimedia Extensions for Windows-Enhanced Sound and Video for the PC". *Microsoft System Journal.* Vol6 No 2 March, 1991.

(Ra69) Randell, B., " A Note on Storage Fragmentation and Program Segmentation," *Communications of the ACM*, Vol 17, No7 July 1974, pp. 365-375.

(Sc85) Schreiner, Alex T. and Friedman H. George. Introduction to Compiler Construction with Unix, Prentice-Hall, 1985. Englewood Cliffs, NJ.

(St85) Stallings, William, *Data and Computer Communications* 2nd edition. page 48. Macmilan Publishing Co. New York, NY.

(St87) Stroustrup, Bjarne, the C++ Programming Language, AT&T Laboratories Murray Hill pp. 162. N.J., Addison-Wesley 1989. Reading, MA.

(Sm76) Smith A. J. "Sequentiality and Prefetching in Data Base Systems," *IBM Research report RJ* 1743, March 1976. Also in *ACM Transactions on Data Base Systems*, Sept. 1978, pp 223-247.

(Sm78) Smith, A. J. "Sequential Program Prefetching in Memory Hierarchies," *Computer*, Vol 11, No 12 December 1978 pp. 7-12.

(Ta87) Tanenbaum, A. S. *Operating Systems Design and Implementation* pp 369. Prentice-Hall 1987. Englewood Cliffs, NJ.

(Tr76) Trivedi, K. S., "Prepaging and Applications to Array Algorithms." *IEEE Transactions on Computers*," Vol. C-25, September 1976. pp. 915-921.

(Unix) Unix Programmers Manual, Volume 2, by Bell Laboratories. Murray Hill, NJ.

(We82) Wesselkamper T. C., Computer Program Schemata and the Process They Generate *IEEE Transactions on Software Engineering* July 1982, pp. 412-418.

(Wi88) Wiener S. and Pinson L. J. An Introduction to Object Oriented Programming and C++, University of Colorado at Colorado Springs. 1988. ch 6. pp 145-146. Addison-Wesley. Reading, MA.