

# Real-Time Structured Video Decoding and Display

by

Brett Dawson Granger

B.S., Electrical and Computer Engineering  
Rice University, Houston, Texas 1992

Submitted to the Program in Media Arts and Sciences,  
School of Architecture and Planning,  
in Partial Fulfillment of the Requirements for the Degree  
of


MASTER OF SCIENCE  
in Media Arts and Sciences  
at the  
Massachusetts Institute of Technology  
February 1995

© Massachusetts Institute of Technology, 1994  
All Rights Reserved

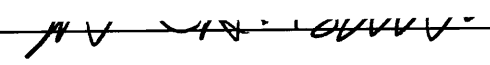
Author

  
Program in Media Arts and Sciences  
October 7, 1994

Certified by

  
V. Michael Bove, Jr.  
Associate Professor of Media Technology  
Program in Media Arts and Sciences  
Thesis Supervisor

Accepted by

  
Stephen A. Benton  
Chairperson, Departmental Committee on Graduate Students  
Program in Media Arts and Sciences

Rotch

# **Real-Time Structured Video Decoding and Display**

by

Brett Dawson Granger

Submitted to the Program in Media Arts and Sciences,  
School of Architecture and Planning,  
on October 7, 1994  
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE in Media Arts and Sciences  
at the  
Massachusetts Institute of Technology

## **Abstract**

Increasing amounts of research are being dedicated to the representation of video sequences in terms of component parts which are rendered and composited according to scripting information. Representations chosen range from two-dimensional layers all the way through full three-dimensional databases. These types of representations show great promise for compression, interactivity, and post-production flexibility and are collectively labeled "structured video" for the purposes of this thesis.

This thesis implements a flexible decoder for structured video representations. The implemented decoder supports 2D, 2-1/2D (2D with z-buffers), and 3D objects as well as explicit and parametric transformations and error signals. A simple scripting language is also created for use in testing the system. Using the environment thus created, an example structured video application, contextual resizing, is implemented and presented.

Thesis Supervisor: V. Michael Bove, Jr.  
Title: Associate Professor of Media Technology

This work was supported by the Television of Tomorrow and Movies of the Future consortia.

# **Real-Time Structured Video Decoding and Display**

by

Brett Dawson Granger

The following served as readers for this thesis:

Reader



---

Glorianna Davenport  
Associate Professor of Media Technology  
Program in Media Arts and Sciences

Reader

---

Edward H. Adelson  
Associate Professor of Vision Science  
Program in Media Arts and Sciences

## Acknowledgements

Heaven knows I haven't made this easy on myself. I truly do appreciate all that everyone has done to help me finally get this done. In particular, I must thank:

My advisor, Mike Bove, for being the source of many new ideas as well as for knowing when to push and when not to, and for being an all-around great guy.

My readers Glorianna and Ted for good advice and willingness to make time to listen.

John Watlington (Wad), for somehow knowing something relevant to every question I could ever ask, and for lots of great advice on how to make Cheops do what I wanted it to.

Shawn, for being a good friend and the font of all Cheops software knowledge. The title "cheops software czar" is definitely earned!

Katy and Araz for so much help developing software even when I had no clue where I was going and changed my mind constantly.

Everybody in the Garden in general for being fun people to be around. Special thanks to Matt, Jill, and Roger for cheerful comments and listening ears.

All my friends at the Cambridge University Ward (far too many to name) for giving me a stable background against which to place this whole experience. Thanks for encouraging me when I needed it and for helping me to get away when I absolutely had to.

My family, for never quite understanding what I do but being completely confident that I would find away to solve all problems I encountered along the way.

Good luck to all of you, my friends!

# Contents

<b>1</b>	<b>Introduction</b>	<b>8</b>
1.1	Examples of Structured Video .....	10
1.2	A Generic Structured Video Decoder.....	12
1.3	Thesis Overview.....	16
<b>2</b>	<b>The Cheops Imaging System</b>	<b>18</b>
2.1	Hardware.....	18
2.2	Software.....	19
<b>3</b>	<b>The Data Processing Pipeline</b>	<b>23</b>
3.1	Transformation Unit.....	23
3.1.1	3D Objects.....	24
3.1.2	2D & 2-1/2D Objects.....	26
3.2	Compositing Unit.....	31
3.3	Error Adders.....	32
<b>4</b>	<b>The Script Interpreter</b>	<b>33</b>
4.1	Time-Based to Frame-Based.....	33
4.2	Describing a Single Frame.....	34
4.2.1	View Parameters.....	35
4.2.2	Actors.....	40
4.2.3	Display Parameters.....	42
4.3	Creating Sequences from Frames.....	43
4.3.1	Compiled Scripts.....	43
4.3.2	Interpreted Scripts.....	44
<b>5</b>	<b>Example Applications for a Structured Video Decoder</b>	<b>49</b>
<b>6</b>	<b>Results, Ideas for Future Work, Conclusion</b>	<b>52</b>

# Appendices

<b>A</b>	<b>A Sample Scripting Language</b>	<b>60</b>
A.1	Currently Supported Script Commands.....	61
A.1.1	Sequence Control Commands.....	61
A.1.2	User Input and Macros.....	62
A.1.3	Defining and Controlling Actors .....	64
A.1.4	Defining and Controlling View Parameters.....	67
A.1.5	Defining and Controlling Display Parameters .....	68
A.1.6	Other Commands .....	68
A.2	A Simple Script: Contextual Resizing .....	70
A.3	Adding New Functions to the Scripting Language.....	75
<b>B</b>	<b>Structured Video Object File Format</b>	<b>78</b>
<b>C</b>	<b>Pipeline Implementation Specifics</b>	<b>83</b>
C.1	The Transformation Unit .....	83
C.1.1	The 3D Transformation Unit .....	84
C.1.2	The 2D Transformation Unit .....	85
C.2	The Compositing Unit .....	86
C.2.1	Stream Rearrangement .....	86
C.2.2	Compositing Objects .....	89
C.2.3	Hardware Versus Software Compositing.....	90

# List of Figures

1-1	Structured video decoder processing pipeline.....	14
1-2	A number of different configurations of the generic processing pipeline .....	15
2-1	Block diagram of Cheops P2 processor module.....	20
3-1	Simplified block diagram of data processing pipeline.....	24
4-1	A typical perspective projection.....	36
4-2	Points and vectors Defining a typical 3D view.....	37
4-3	View Structure .....	39
4-4	Actor Structure.....	41
4-5	DispStruct Structure.....	43
4-6	ParamStruct and LL_Struct Structures .....	46
5-1	Sample output from contextual resizing application .....	51
6-1	Implementation of the data processing pipeline consisting of 2 P2 modules ...	55
B-1	Tree representation of structured video object file format.....	79
C-1	Dataflow diagram for the 3D transformation unit.....	85
C-2	Dataflow diagram for the 2D transformation unit.....	86
C-3	Splitting transformed 4-vectors into xy and z streams .....	88
C-4	Interleaving intensity and z values.....	88
C-5	Compositing Objects using remap/composite module.....	89

# Chapter 1

## Introduction

Using computers to assemble and view movies is by no means a novel concept. As early as 1972, Catmull used "fast" hidden-surface removal algorithms and smooth polygon shading techniques along with a primitive motion picture description language to generate computer images. A computer controlled movie camera aimed at a high-precision scope recorded these computer-generated movies [Catmull72]. Movies which come out of Hollywood today are marvels of digital image compositing and computerized retouching of images, making previously impossible scenes commonplace.

Decreasing prices of both computer memory and mass storage devices, accompanied by increasing technology in both hardware and software are bringing real-time image generation and display ever closer to the common user. Most high end personal computers now boast video capabilities in which they claim to be able to display images of  $M \times N$  pixel resolution at  $F$  frames per second.

The Information and Entertainment Group of the MIT Media Laboratory has for several years been investigating moving away from pixel- and frame-based image representations and toward structural representations of moving scenes. This promises both greater coding efficiency and the ability (whether in post-production or in interactive viewing) to manipulate the data in a physically or semantically meaningful way [Bove94b]. As a simple example of this, consider a sequence in which a single object moves in a straight line across a static background. Using a  $1K \times 1K$ , 24-bit color representation on a



60Hz progressive-scan display will result in 180 megabytes transmitted per second of the sequence being displayed. If, on the other hand, the receiver already has a model of both the background and the object in local memory, the information transmitted at display time can be reduced to a simple command such as "Object A moves from point 1 to point 2 along a linear path in front of background B in time T." Granted, the description will probably be in some scripting language, not in English, and the receiver will need a method for synthesizing the proper sequence based on the description given, but even if the description were much more complex, transmitting it would still be a substantial savings over the 180 megabytes per second required in the frame-based representation.

More likely, the receiver will not already have a model of the background and the object in memory and those models will have to be transmitted as well. But even should this be the case, a 1Kx1K color background can be described in 3 megabytes and an object smaller than the background will obviously require less than that. The resultant maximum of 6 megabytes plus a small script will still be a tremendous savings when compared to 180 megabytes per second, as long as the models enable more than one thirtieth of a second of video to be synthesized. More complex models will require more memory to describe, but will also enable longer sequences to be created before the next models must be transmitted.

A video sequence described in the manner outlined above is an example of structured video. For the purposes of this thesis, structured video refers to the coding of an image sequence by describing it in terms of components that make up the scene. One natural breakdown is actors and backgrounds, but others are also possible (see the examples given in section 1.1).

Another benefit inherent in the structured video description of this sequence is that no mention is made in the description (or script) of resolution or of the frame rate of the output device. Thus, given the proper hardware and software, the above script combined with the data already in memory could describe similar sequences on any number of display devices with varying resolutions and/or frame rates. This makes structured video

representations ideal for implementing open architecture (OAR) television as defined in [Bender90], which is scalable both in resolution and in time, but structured video environments can be used for more than television playback, no matter how scalable.

In structured video environments, there is no requirement that the components come from digitized video sources. In combination with a scene analysis system, for example, a structured video decoder can simply reconstruct a transmitted video sequence. Using the same system, however, real actors can be inserted into a synthetic background (or vice-versa), or a completely synthetic three-dimensional world can be rendered for display.

Interactivity is also much easier in a structured video environment, since by definition the scene is broken down into component parts. Knowing the components makes it a simple thing to have any component respond to the state of the system or to user input. Similarly, structured video representations should allow for easier semantic searches through stored video. A request such as "Find the episode in which so-and-so finds a certain object in the living room" actually becomes feasible in a structured video environment.

This thesis implements a decoder as well as an environment for the decoding and synthesis of structured video sequences. The implemented system is designed to accommodate many varying representations of data, both real and synthetic, as well as to allow interactivity with the various components described based on user input. The script which describes the desired output is based on time rather than frame rates to allow even greater flexibility.

## **1.1 Examples of Structured Video**

One possible method of describing a scene is to segment it into objects that would be considered foreground and objects which would be considered background. In

[McLean91], for example, a method is explored for low-bandwidth transmission of video sequences which attempts to take advantage of the structure within the sequence. In this system, actors and other large independent objects are segmented out of the scene. From what remains of the sequence, a model of the background is constructed which is stored locally at the receiver and so does not need to be transmitted. The actors and other objects which are to be inserted into the scene are then transmitted along with parameters describing how the background itself is to be animated, and from this data the original sequence is reconstructed.

A different approach is suggested by Wang and Adelson [Wang94]. In their system, moving images are represented by a collection of 2D layers, typically ordered by depth. Distinct components of the image are then represented by separate layers. Along with the intensity maps which define the basic image, additional maps are used which add information about velocity, attenuation, or changes between frames. A depth map may also be attached which gives a distinct depth to every point in the 2D layer (depth maps are also often called "z-buffers" and 2D objects which include z-buffers are called 2-1/2D objects by many authors). Thus in this system a single object moving in front of a background may be simply described by two intensity maps, an attenuation map (or "mask"), and a velocity map. A more complete and accurate description might include a delta map ("change map") for error corrections, as well as motion blur and depth maps.

As an example of a system which incorporates structured video concepts to create an output sequence, consider the "video finger" application created by John Watlington [Watlington89]. Video finger monitors the state of a shared computer workspace and creates a synthetic sequence which displays information about the state of the system such as which users are currently logged in and to a limited extent what they are currently doing on the system (e.g. compiling, reading news, sitting idle). This is achieved without the use of a physical camera by storing a collection of digitized sequences of each user performing some basic tasks such as entering and sitting down, reading, or even falling asleep. Video

finger then selects the correct users and sequences based on information returned by the system being monitored and inserts these sequences into a background to create the final synthetic movie.

More recently, researchers have developed a virtual studio system in which digitized video of actors may be combined with other footage shot at a different time or even with 3D sets synthesized using computer graphics techniques [Kazui94]. In this system, actors are correctly positioned and scaled based upon parameters that are obtained from sensors placed on actual cameras to determine position, orientation, and focal length, or from virtual cameras controlled by the system operator. The separate components are then composited in a back-to-front manner for proper occlusion of hidden surfaces.

The above-cited examples show that structured video representations are being considered more and more for their capacity to decrease the amount of data necessary to represent a video sequence while at the same time increasing the flexibility and manipulability of the same data. All of these examples are quite different in the representation and reconstruction schemes chosen, however, and much less research appears to be going towards the creation of a flexible structured video decoder that might be called on to create sequences from a variety of different representations, particularly when the reconstruction is desired in real-time for display on an output device.

## **1.2 A Generic Structured-Video Decoder**

After examining the above examples of structured video coding, as well as thinking about how to maintain an open and flexible system, a generic structured video decoding pipeline has been proposed [Bove94a]. It is envisioned that the data may take on various representations and that the pipeline must be able to process several types of objects, including but not limited to:

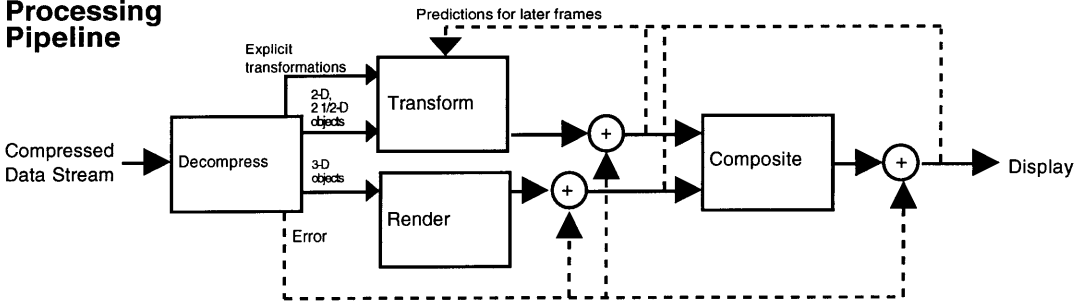
- 2D objects - These are simply two dimensional arrays of pixels, which are assumed to lie all at the same depth specified by the script. They may also exhibit transparency as defined by an additional "alpha" channel. In general, it is hoped that 2D objects will be digitized at the maximum resolution at which it will eventually be displayed, as enlarging images tends to make them blurry or blocky and thus deteriorates the quality of the output.
- 2 1/2D objects - These are 2D objects with an added depth value at every pixel, specified by a z-buffer, which is used for compositing.
- 3D objects - In the system implemented in this thesis, 3D objects are represented by particle databases. These are a collection of points, where each point has an x, y, and z position value as well as an intensity or color value. These are treated the same as three dimensional computer graphics objects which require rendering before they can be displayed. As hardware developments allow, transformed polygons may also be supported in the future.

In addition to the above-described object types, several other types of information need to be accepted by a flexible system in order to deal with representations such as Adelson and Wang's layers or even MPEG coders. These objects include:

- explicit transformations - These may be specified as full arrays of values to be applied to every point of an object or frame. They may be also defined in the script parametrically such as would be the case with affine transformations (see section A.1.6).
- error signals - 2D arrays of values which are added to processed objects or to composited frames to correct for errors in the encoding scheme.

Figure 1-1 shows a block diagram of the data processing portion of the decoder pipeline. Before the final sequence can be output, whether it be to display or file or another

## Processing Pipeline

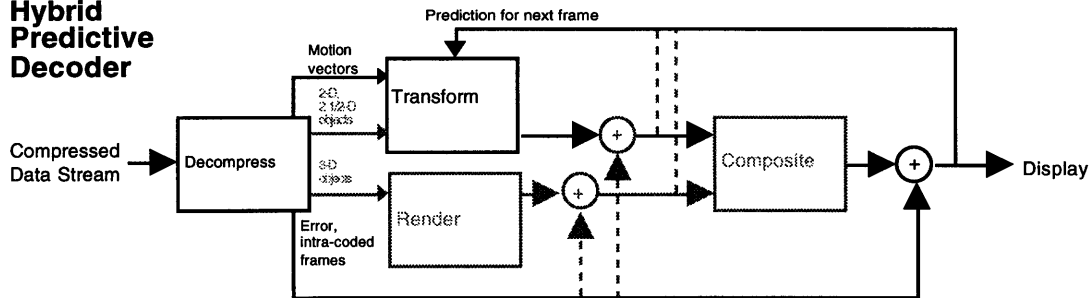


**Figure 1-1:** Data processing pipeline of a generic structured video decoder. Not shown is the script interpreter which controls the operations of each stage of this pipeline.

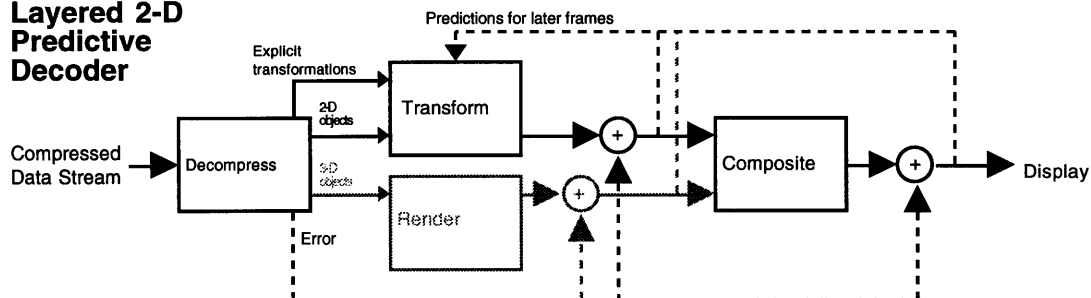
output device, it is assumed that the various types of objects will need to be processed in some way. All 3D objects will need to be rendered, for example. This includes geometry transformations on each point in the object as well as compositing the results into an output buffer. 2D and 2-1/2D objects might need to be warped or scaled. If the final image consists of more than one object, which will be the case in most sequences, those objects will need to be composited together. The compositing stage uses a z-buffer algorithm to display only the visible surfaces of each object in the final image. Error signals might also need to be added into the final output to complete the image.

Depending on which data paths are enabled in this generic pipeline, various types of decoders can be modeled, as is shown in Figure 1-2, ranging from an ordinary hybrid predictive coder up through a full 3D synthetic computer graphic scene renderer.

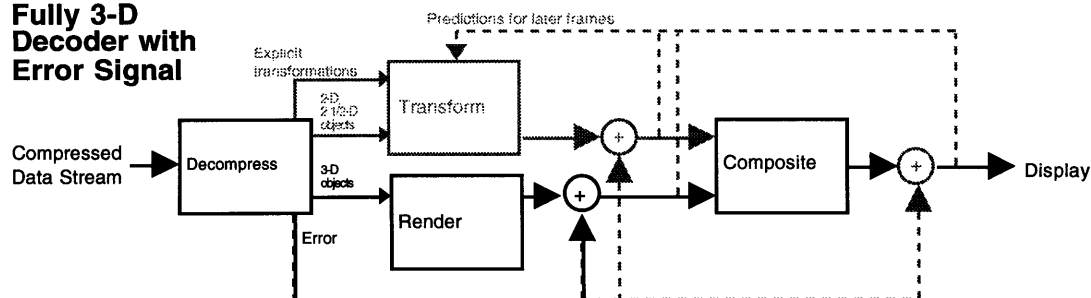
### Hybrid Predictive Decoder



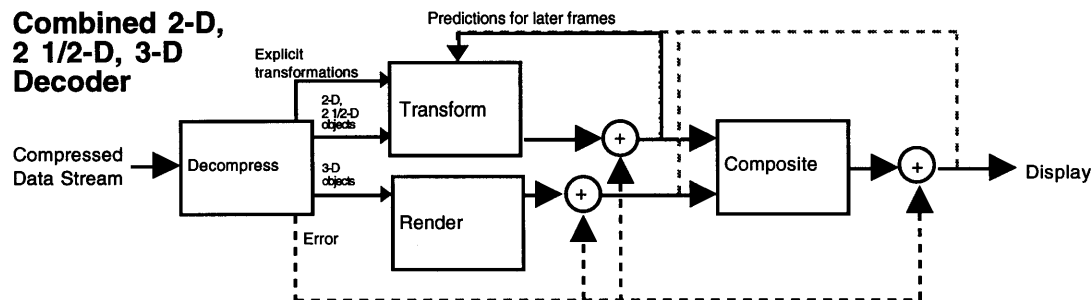
### Layered 2-D Predictive Decoder



### Fully 3-D Decoder with Error Signal



### Combined 2-D, 2 1/2-D, 3-D Decoder



**Figure 1-2:** A number of different configurations of the generic pipeline shown in figure 1-1 are possible, enabling the decoding of a variety of different coding methods. In this figure, gray datapaths are inactive and dashed datapaths may be enabled by the algorithm if needed.

## 1.3 Thesis Overview

This thesis implements the data processing pipeline described in the previous section as well as a script interpreter which is not shown, with one notable exception: no data decompression scheme is implemented. It is assumed that decoders already exist for the various ways in which the incoming data may be encoded, such as transform coding or run-length coding, and that they can easily be incorporated into the pipeline at a later time. The resulting structured video decoding environment resembles a simple computer graphics animation system in that a mechanism is provided by which objects digitized at different times or with differing parameters can be converted to common units and made part of a three-dimensional world where they can be manipulated with a simple scripting language. This world is then viewed by defining a virtual camera. Extensions to the world are made which allow for the description of explicit transformations and error signals as objects as well as for controlling how those transformations and errors are used in the pipeline.

Chapter 2 provides a brief introduction to the Cheops system hardware and software on which the structured video environment presented in this thesis is implemented. The Cheops system is chosen for several reasons. First, the Cheops system architecture is designed with high-bandwidth video transfers in mind. In addition, it is hoped that the resulting environment will be friendlier and more conducive to testing new algorithms and developing applications on the Cheops system than the environment which is currently in place.

A description of how the data processing pipeline presented in Section 1.2 is implemented on the Cheops system is given in Chapter 3, as well as a discussion of how the various types of objects (2D, 2-1/2D, 3D) are processed by the system. Wherever possible, specialized Cheops system hardware is used in an attempt to achieve real-time performance with the system.



The script interpreting system is presented in Chapter 4. The basic structures used to describe a single frame are discussed along with the mechanism used to describe the creation of sequences from consecutive frames. It is expected that various different user interfaces to this system will be explored, including the one presented in Appendix A. As long as all those interfaces create output in the form of the structures described in Chapter 4, the system should have no difficulty creating the desired output.

Several examples which have already been implemented using this structured video system are described in Chapter 5. A few additional ideas for applications which have not yet been tried are also proposed.

Finally, Chapter 6 summarizes some of the results obtained using the structured video system, analyzing both the performance of the system as well as the ease of its use. Recommendations are made as to how system performance might be improved, and ideas are presented for using this system in combination with other recent research in video sequence creation, both real and synthetic.

Three appendices are also provided. Appendix A describes a simple scripting language which has been developed in conjunction with the structured video decoder. A sample script is provided along with the output it creates. How to add commands to the script is also explained in this appendix. Appendix B describes the file format which has been developed to describe 2D, 2-1/2D, and 3D objects. Appendix C discusses the specifics of how the data processing pipeline was implemented in the current Cheops environment.

# Chapter 2

## The Cheops Imaging System

The Cheops Imaging System is a compact, modular platform for acquisition, real-time processing, and display of digital video sequences and model-based representations of moving scenes, and is intended as both a laboratory tool and a prototype architecture for future programmable video decoders [Bove94b].

### 2.1 Hardware

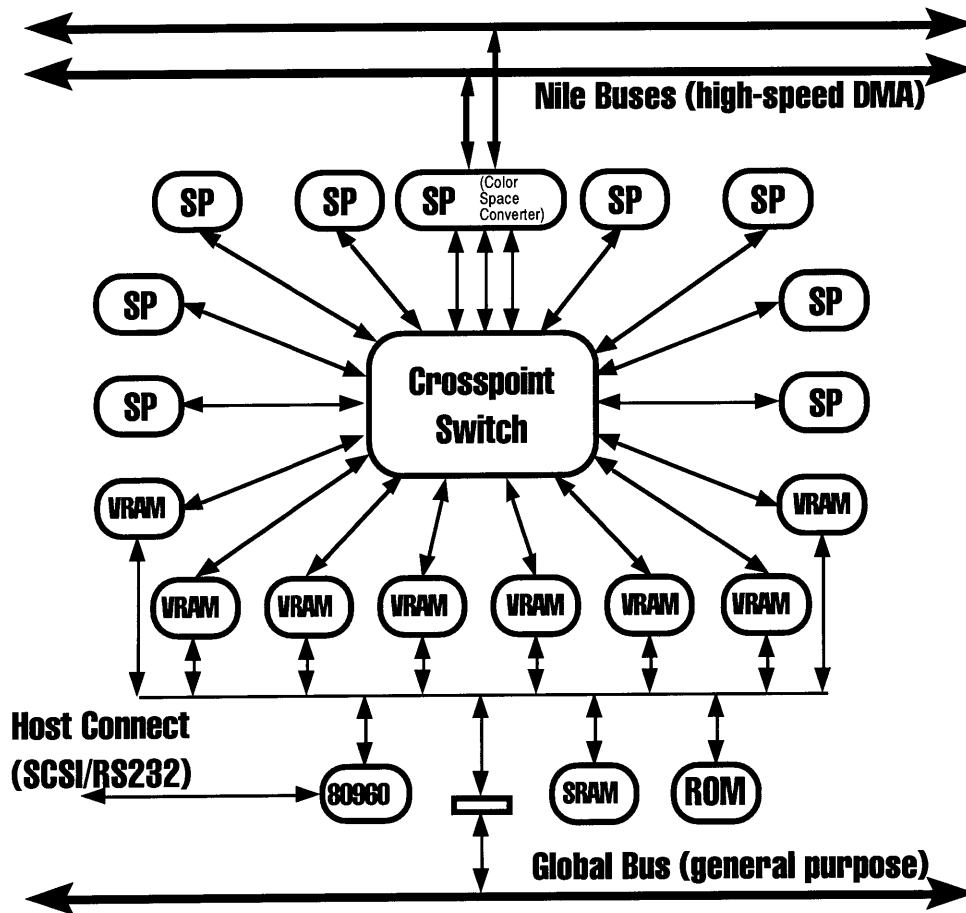
The Cheops system hardware is divided into modules based on the stages of video processing: input/memory modules (M1), processing modules (P2), and output/display modules (O1/O2). Up to four of each type of module may be present in the system at one time, allowing for a broad range of system configurations. These modules are interconnected via three linear buses, including two capable of sustaining high-bandwidth transfers on the order of 120 Mbytes/sec. These buses (called "Nile" buses) can be used to transfer data from a P2 module to be displayed on an output module, from an M1 memory module to a P2 module where it can be processed, or even in between P2 modules where different processing elements may reside.

In addition to the inherent modularity of the system, Cheops is designed to be both hardware and software configurable. Rather than using a large number of general-purpose processors and dividing up image processing tasks spatially, Cheops abstracts out a set of basic, computationally intensive stream operations that may be performed in parallel and embodies them in specialized hardware [Bove94b]. On each P2 processor board, eight memory units are connected through a full crosspoint switch to up to eight stream processing units. A single memory unit consists of dual-ported dynamic memory and a two-dimensional direct memory access (DMA) controller called a "flood controller" which can transfer a stream of data through the crosspoint switch at up to 40 Msample/sec to the specialized stream processors. Six of these stream processors reside on removable sub-modules (two per sub-module). Thus the hardware configuration is easily changed by appropriately selecting sub-modules to include in the system and hardware is easily upgraded by creating new sub-modules as opposed to completely redesigning the processor module. Figure 2-1 shows a highly simplified block diagram of the P2 processor module.

A number of stream processors have been developed for the Cheops system, but two prove to be of particular use in the development of a structured video decoder. The first is a flexible filter processor which can be used to perform both one- and two-dimensional filtering, multiplication of vectors by matrices, and multiplication of one stream by another. The second is a remap/composite unit which performs image warping and also incorporates a 16-bit hardware z-buffer for graphics compositing and hidden surface removal.

## **2.2 Software**

As mentioned previously, the Cheops system abstracts out a set of basic stream operations that may be performed in parallel. This parallelism is very important to the



**Figure 2-1:** Block diagram of the Cheops P2 processor module. Memory units are denoted with the label VRAM, while stream processor sub-modules are labeled SP.

performance obtained in the Cheops system. The hardware is set up with enough control signals such that up to three stream transfers may occur simultaneously contingent that they utilize separate memory banks and different stream processors. Thus, for example, the three color components of a color image can all be filtered at the same time, provided there are at least three filter units in the system configuration. A programmer who plans on doing much filtering of color images would therefore probably like to ensure that there are

at least three filters in the system. In general, however, the user will not want to and should not have to be bothered with the details of which stream processors are in which daughter card slots in the system, or even whether certain stream processors are currently in the system at all. For this reason, as well as to simplify management of the parallel transfers, the NORMAN resource management daemon and the RMAN interface library were developed for the Cheops system.

The NORMAN daemon manages the stream processors and flood controllers by maintaining scoreboards and wait queues for each [Shen92]. A transfer through a stream processor is started as soon as that processor along with the source and destination memory banks requested by the transfer are free, and all other transfers upon which the current one has been said to be dependent have completed. Thus Cheops is programmed as a data flow computer in which algorithms are described as data flow graphs specifying operations to be performed on streams of data by the stream processors and the dependencies among those operations. NORMAN itself handles the setting up and enabling of the hardware so that the user does not need to worry about that. The user, however, must still provide the parameters to be used in setting up the hardware.

Since the data structures which describe the configuration of the various processors are rather complicated and tedious to initialize manually each time, the RMAN (Resource Management) interface library to NORMAN was also developed. The RMAN library provides routines for the creation of the structures that define many basic stream operation in which a large number of the parameters which the hardware requires have been set to common default values. The user can then specify a filter operation, for example, with only the essential parameters, such as x and y dimension of the buffer, source and destination addresses, and the specific filter taps to be used. Additional functions provide access to the other hardware parameters so that the user can customize the operation as needed. The RMAN library also provides mechanisms for specifying dependencies

between operations and the linking of operations into a complete pipeline which can then be passed in to NORMAN where it will be executed as the resources become available.

# Chapter 3

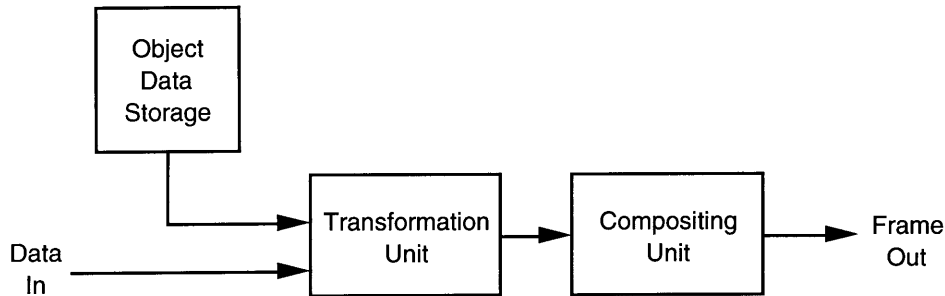
## The Data Processing Pipeline

Section 1.2 introduces the generic structured video data-processing pipeline that is implemented in this thesis. This Chapter addresses the specifics of how this pipeline is implemented on the Cheops system.

Figure 3-1 shows a slightly more simplified diagram of the data-processing pipeline, indicating that aside from the script interpreter which is described later, there are only two main functional blocks that must be implemented, regardless of the type of object being processed. The transformation unit includes geometry transformations and projections of 3D objects, and positioning, scaling, and even explicit transformations defined by motion vectors on 2D and 2-1/2D objects. The compositing unit takes each object as it is output by the transformation unit (by now in a common representation) and composites it into the final frame using a z-buffer algorithm.

### 3.1 The transformation unit

The transformation unit is the primary means by which this pipeline simulates a real camera looking at a three-dimensional world model. It is the transformation unit that projects 3D objects into 2-1/2D objects so that they can be displayed on 2D displays. The



**Figure 3-1:** Simplified block diagram of the data processing pipeline shown in Figure 1-1. Again, the script interpreter which controls the operation of this pipeline is not shown.

transformation unit also scales 2D and 2-1/2D objects to the proper dimensions they would assume if they were in fact 3D objects placed at their respective positions in a three-dimensional world being viewed by the specified camera. The projection and transformations to which a photographic system subject objects in the world can be modeled quite realistically by type of planar geometric projection known as perspective projection, thus the transformation unit attempts to implement perspective projection for all objects. For a description of perspective projections and view parameters, see section 4.2.

As previously shown in Figure 1-1, the transformation unit actually subdivides into two different functional blocks based on whether the object to be processed is 3D or 2-1/2 or 2D.

### 3.1.1 3D Objects

In the current implementation of the pipeline, 3D objects are specified as particle databases. This means that the data composing a 3D object consists of a collection of points each of which has an associated x, y, and z position as well as a single intensity or three color components. Since the entire collection of points defines the object, no particular ordering of the data points is necessary. In general, only the points on the surface ("surfels") of the object need to be included in the data, since internal points will never be displayed and there is no need to process them.



One of the possible configurations of the Cheops filter processor is a render mode. In this mode, the stream processor is capable of premultiplying a stream of 4x1 column vectors with a 4x4 matrix which has been preloaded into the processor. [Foley90] describes how 3D planar geometric transformations may be specified as 4x4 matrices. Following this model, 3D object position data is stored in memory as 4-vectors of the form  $[x\ y\ z\ w]$  where  $x$ ,  $y$ , and  $z$  are provided in the object data transmitted to the decoder, and  $w$  is an implementation-dependent value. The  $x$ ,  $y$ , and  $z$  values provided in the object data are assumed to be relative to the 3D object's own origin. Each object to be processed then has a transformation matrix associated with it that specifies the position and orientation of the object in the world. The 4x4 transformation matrix which is loaded into stream processor is obtained by composing each object's individual transform matrix with the matrix which is obtained from the view parameters. Every object is therefore multiplied by a different total transformation matrix, but each point within a single object is multiplied by the same matrix.

The final step necessary to complete a perspective projection on 3D objects is to scale the projected  $x$  and  $y$  coordinates of each point correctly to correspond with the effect caused by perspective foreshortening. This effect is achieved by multiplying each point by the scale factor  $f/z$ , where  $f$  is the focal length of the camera and  $z$  is the transformed  $z$  coordinate of the point to be scaled. This proves to be troublesome to implement in the Cheops system hardware. The filter processor can be configured to perform stream multiplication, but no hardware currently exists which can perform stream division. Focal length is fixed in a single frame, so if all points were at the same  $z$  position it would be trivial to perform the single division in software and then use the hardware to multiply the  $xy$  stream by the single scaling factor. In 3D objects, however, the transformed  $z$  coordinate is potentially different at every point. In computer graphics systems, the  $w$  of the  $[x\ y\ z\ w]$  vector contains the  $f/z$  factor and so perspective foreshortening is applied by dividing the  $x$ ,  $y$ , and  $z$  by the  $w$ . Stream multiplication of  $1/w$  could be performed by the

Cheops filter hardware using a lookup table to find the value of  $1/w$ . However, no lookup table hardware exists in the Cheops system which makes a hardware implementation currently impossible. Since one of the driving factors in the implementation of this pipeline is an attempt to achieve real-time performance, it was decided that perspective projection of 3D objects would therefore not be implemented because a software implementation would be much too slow to make the system useful.

Once a 3D object has been passed through the transformation unit, it is basically a 2-1/2D object and is ready to be passed to the compositing unit, which will be described in section 3.2.

### **3.1.2 2D & 2-1/2D Objects**

One obvious method of incorporating a 2D or 2-1/2D object into a three dimensional world is to turn it into a 3D object by creating x, y, and z position values for every point in the object. After all, the x and y coordinates for each of the points in the 2D array of pixels is implied by the x and y offset of the pixel from the start of the two-dimensional array making up the object, and the z value is either uniform and specified by the script (2D objects), or is contained in a z-buffer (2-1/2D objects). The resultant 3D object could then be passed through the 3D transformation unit as described previously.

Several problems arise with this treatment of 2D objects, however, making it undesirable for the purposes of this system. One of the major problems is that 2D objects are flat so that every point in them lies in a single plane. Subjecting these objects to generic 3D rotations will in general not produce usable views, and on occasion will even produce single vertical lines, if the 2D object is viewed directly from the side. 2-1/2D objects may fare slightly better, since z values do exist for them. However, the z-buffers are discrete valued, and will thus not create continuous surfaces when viewed from certain angles. In general, it will be desirable to keep 2D and 2-1/2D objects always directly facing the camera. Another problem with creating 3D objects from 2D objects is that arbitrary origin

points cannot be specified for each object. Even within objects that are composed of several views (for example, a set of views of an actor which will be used in sequence to simulate walking), each view might have a different point which would be considered the origin, upon which it is easiest to base all positioning and path planning. A third problem with 2D objects being used as 3D objects is that it is very difficult if not impossible to determine the real-world dimensions of a 2D object from the resolution at which it is digitized without knowing a great deal about the equipment which performed the digitization. When attempting to realistically combine objects which have been digitized at different times or which have been filmed using different camera parameters, this information will be important and will not always be readily available to the user.

In this pipeline, 2D objects are not simply treated as 3D objects, and a separate transformation unit is created for them. In this unit, 2-1/2D objects are treated identically to 2D objects, thus in the description that follows, wherever 2D objects are mentioned, unless explicitly stated otherwise it should be assumed that 2-1/2D objects receive the same treatment.

When a 2D object is encountered, it will be transformed in one of two ways. If an explicit transformation expressed as a vector field or affine parameters is provided, the object will be transformed according to that transformation and then passed on to the compositing unit. If no explicit transformation is provided, then it is assumed that the 2D object is intended to be treated similarly to a 3D object in that it will be placed at a location in the world and transformed according to the view parameters which have been established. In such a case, the only transformations which will be applied to the object are scaling and/or translation.

The case of explicit transformation is implemented using the remap/composite unit in the Cheops system. In remapping mode, the unit takes two two-dimensional streams as input: a stream of intensity values, and a stream of x,y offset vectors which are applied to the intensity values. These vectors can be applied as write vectors (the offset is applied

before the intensity is written into memory), or as read vectors (the offset is used to calculate the address from which the intensity will be read). In this implementation the vector fields are used as read vectors, since read vectors will ensure that every output pixel gets an intensity, whereas write vectors may not write intensities to all pixels, leaving holes in the output. If the vector field is provided along with the 2D object, then both streams are merely passed to the remap processor and the transformation is performed. If the transformation is specified in terms of affine parameters in the script, then those parameters are used to create a vector field of the appropriate dimensions to match the 2D object data, and then the transformation is performed using the remap unit.

If the 2D object is to be treated as an object in a world and transformed as if it were a 3D object, several more calculations must be performed, and several assumptions are made. The first assumption is that 2D objects will always directly face the camera; that is, they will always be viewed at the same angle from which they were filmed. This means that rotation of a single 2D object is not allowed, only scaling and translation. However, rotation is often simulated with 2D objects by having a series of views taken at various angles around the object. A normal vector indicating the orientation of the 2D object can then be compared to the vector which defines the camera angle to choose the appropriate view of the 2D object to be shown, based on the angle from which it is being viewed. This automated view selection based on angle is not currently implemented in the pipeline, but is fairly straightforward.

In order for a 2D object to be placed into a three dimensional world, several things must happen. First, the raw data in the 2D object must be scaled into world units. If the object were 3D, it would then be transformed based on its position and the view parameters. However, since all the points lie at the same depth, and since 2D objects will not be processed in the same way as 3D objects, only the origin point of the object is actually transformed. A mechanism is provided in the object data definition to define the scale factor which converts from pixels into world units as well as the to define the origin

point of the object (see Appendix B). If this information is not provided, the origin of the object is assumed to be its pixelwise midpoint in x and y dimensions and the resolution in pixels is assumed to correspond exactly to the units of the world into which it is being placed. The second scale factor that is applied to the object is the effect of perspective foreshortening based on the distance from the eye to the origin point of the object. Since all points in the 2D object lie at the same depth from the eye, the  $f/z$  scale factor can be applied uniformly to all points in the object. And finally, a scale factor will need to be applied which takes the object from world units back to pixel units for display based on the view and display parameters specified in the script.

Since only one point in the object actually needs to be transformed to determine all three scale factors, as well as the position, and since the compositing unit composites in pixels to produce an image which is ready for display, all three factors are multiplied together to get one overall scale factor which will be applied to all the data in the 2D object. And from the transformations performed on the origin of the object, a "paste point" is calculated in pixels which indicates to the compositing unit where the upper left corner of the scaled 2D data is to be placed in the final image. The calculation of the scale factors, and the transformation of the single point are all performed in floating point arithmetic in software. The scaling of the 2D data is implemented in hardware.

There are several possible methods of implementing a scaling pipeline in the Cheops hardware. One method, which is not implemented, again involves the use of the remap/composite unit. In this method, a contracting vector field is created which would cause the appropriate scaling on the object, and then the data is appropriately prefiltered and passed through the remap processor along with the contracting field to obtain the correctly scaled output. This possibility was not chosen because creating the contracting vectors each time is a slow process, and creating the vector fields ahead of time and storing them requires a great deal of memory.

The 2D scaling pipeline is instead implemented using scaling filters and the built-in capability of the Cheops flood controllers to replicate, zero-pad, and decimate streams of data by integral factors up to sixteen. This of course limits scaling of 2D objects to rational factors whose numerators and denominators are each less than or equal to sixteen, but this was deemed sufficient for this system. The Cheops filter unit and separable two-dimensional filters are used in this pipeline, since the filter unit can perform one-dimensional filters with a reasonable number of taps (up to sixteen at the full system clock rate). The current implementation uses triangular filters, as this was deemed sufficient, but gaussian filters are another easy-to-implement possibility. Using separable 2D filters involves two passes through the filter unit with a transposition of the data in between passes, so the complete 2D scaling pipeline consists of transposing the data, filtering it in the y dimension, transposing it again, then filtering it in the x dimension. Since a transfer through the filter unit involves two flood controllers in addition to the filter unit, the data can be upsampled, filtered, and downsampled all in the same pass. Thus scaling occurs at the same time as the filtering. The filter coefficients to be used are chosen appropriately based on the maximum upsampling or downsampling rate, as described in [Oppenheim89] and many other digital signal processing texts.

In 2D objects, up to five channels might need to be passed through this scaling pipeline: three color channels, a z-buffer, and an alpha buffer. None of these channels depends on any of the others, and so they are passed in to NORMAN as parallel operations. The Cheops system does not have the necessary control signals nor does it have enough flood controllers to perform all five scaling operations in parallel, but NORMAN takes care of performing the operations when it can and parallelizing as much as possible.

One large discrepancy in treating 2-1/2D objects identically to 2D objects comes in the application of the perspective scaling. By definition, all points in a 2-1/2D object do not lie at the same depth, and so while the perspective scale calculated may be correct for all

points which lie in the same plane as the defined origin point, it will not be correct for the other points in the object. It is therefore preferable to place 2-1/2D objects into the world in such a way that they will not need to have perspective scaling applied to them. 2-1/2D objects are generally obtained by pre-rendering 3D scenes with certain parameters or by using special equipment such as Bove's range-finding camera [Bove89]. It is therefore envisioned that 2-1/2D objects will be used primarily as backgrounds or sets or other large non-moving items, and it will not pose a problem to position them such that perspective scaling is not a factor.

## 3.2 Compositing Unit

The compositing unit is implemented in the Cheops system using the remap/composite stream processor in z-buffer mode. In z-buffer mode, the remap unit accepts a stream of data which consists of intensity and z values interleaved. A second stream may also be included which contains the interleaved x and y coordinates which correspond to the intensity and z in the first stream. If the second stream is not present, the z-buffer assumes that the data is being presented in 2D raster order, and expects an x,y coordinate pair which specifies the upper left corner of the 2D pixel array which is to be composited into the image. As might be expected, 2D and 2-1/2D objects are presented to the z-buffer in raster order, and the x,y offset coordinates are taken from the "paste point" that was calculated in the transformation unit. The z component of the paste point is a constant offset that is added to every z value in the 2D object. Since 3D object data consists of a collection of points in any order with specific x, y, and z components, the output from the 3D transformation unit is rearranged to form an xy stream and an intensity-z stream which are then presented to the z-buffer for compositing. The intensity-z stream input allows for the interleaving of only one intensity channel with the z channel, therefore

compositing color objects consists of making three separate transfers, interleaving each color component separately with the same z values.

### **3.3 Error Adders**

One of the modes of the Cheops filter unit is a stream addition mode, and it is this mode which is used to implement the error addition units. Errors are expected to be decoded as explicit 2D arrays of values which can be added to individual objects, or more commonly to complete frames. It is not yet clear how error signals might be used in conjunction with 3D objects, but due to the simplicity of its implementation, that unit is implemented and left as a hook for future use.

Combining the transformation units, the compositing unit, and the error addition units creates the low-level hardware and software base that can accept and process object data based on the verbose description which will be provided by the script interpreter. This interpreter is described in the sections that follow. For more specific descriptions of the specifics of the implementation of the data processing pipeline, see Appendix C.



# Chapter 4

## The Script Interpreter

A low-level hardware and software system now exists which is ready to accept, transform, and composite together data of many differing representations. The level on top of this one must of course describe the data that is to be fed into the rendering system. And despite the fact that the representation of the sequence is preferably time-based so as to achieve frame-rate independence, any display device on which the final sequence is to be presented will have an inherent frame rate and so the basic unit of any synthetic sequence is of necessity a single frame. It is therefore important to make the conversion from the time-based representation of the script to the frame-based requirements of the output.

### 4.1 Time-based to Frame-based

In the Cheops system each output card has associated with it an output card descriptor which contains a field specifying the frame rate of that particular card in frames per second. In addition to this, the NORMAN resource management daemon keeps a frame counter for each output module in the system. The combination of these two numbers creates a real-time clock against which times in the script may be compared. The system maintains four global variables to assist with this: `display_time`, `seq_time`,

`display_period`, and `sample_period`. `Display_time` tracks the time that has elapsed since the sequence started playing, and `seq_time` keeps track of what time within the script is currently being displayed. These two numbers will differ if the sequence is repeating or playing palindromically; the `seq_time` will always be a value that falls within the limits defined in the script, whereas the `display_time` is monotonically increasing.

`Display_time` is compared with the current time on the output module to determine when the next frame should be displayed. After each frame is displayed, `display_time` and `seq_time` are incremented by `display_period` and `sample_period`, respectively.

`Display_period` and `sample_period` are by default set to the inverse of the frame rate of the output module to which the sequence will be displayed, but are changeable by the user. As already described, `display_period` controls the frame rate of the output sequence.

`Sample_period` can be thought of as the "shutter speed" of the virtual camera -- it defines how much time elapses in the world model in between displayed frames. When `display_period` and `sample_period` are equal, the effect is that of a sequence filmed in real time with a camera of the defined shutter speed. If `sample_period` is less than `display_period`, however, there is a slow-motion effect, and if `sample_period` is greater than `display_period` then the output sequence appears to be running in a "fast-forward" mode.

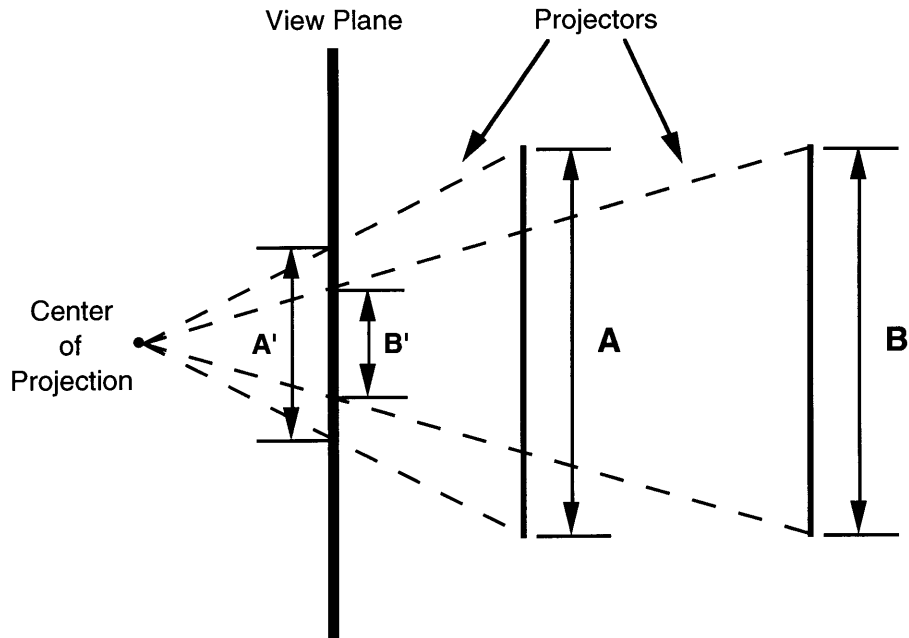
## 4.2 Describing a Single Frame

The frame is a snapshot of the world (real or synthetic) at a particular time, capturing actors and background at that moment, based on the parameters of the camera itself. A frame can therefore be described entirely by describing the state of each of the actors (the background is just a special case of an actor) along with the viewing parameters, which define the transformations to which the actors will be subjected. In addition, if the

final destination of the frame is a display device, as is the case in this system, the physical window parameters must also be defined. In the most general case, all three of these items -- actors, view parameters, display parameters -- can change each frame. Consider a scene in which an actor walks across a set, the camera pans to follow and zooms in on the actor while at the same time the display window shrinks or grows, for example. Thus a complete frame description must include a description of the view parameters, a description of each of the actors that are presently in the world, and a description of the physical display to which the frame will be output.

#### **4.2.1 View Parameters**

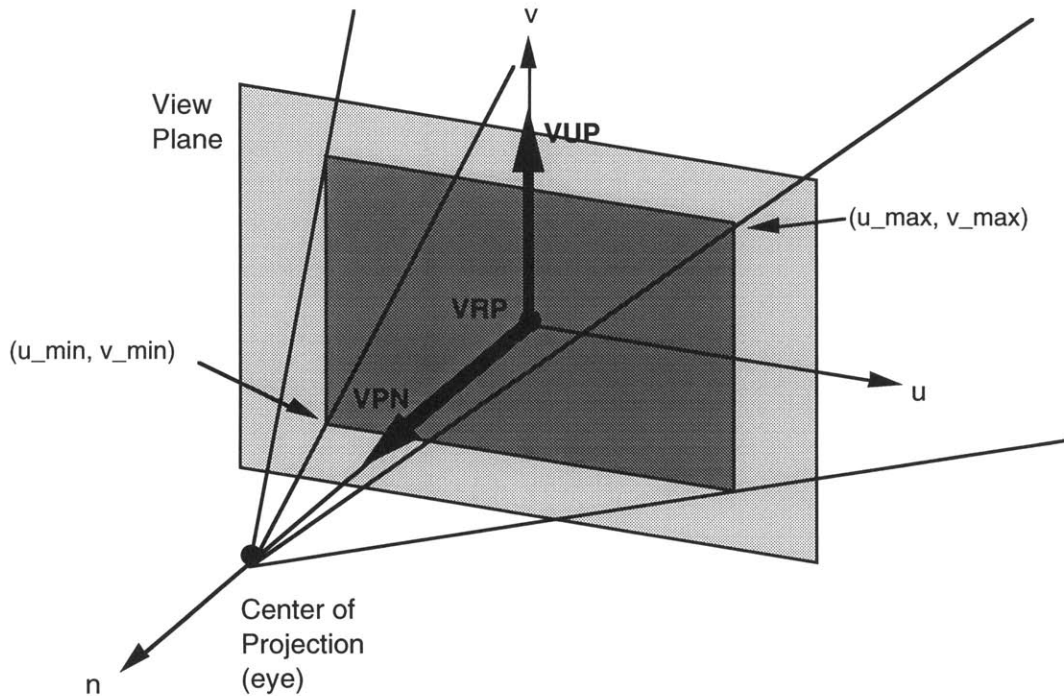
The view parameters define how much of the world will be displayed on the final display window as well as the transformations to which the objects being viewed will be subjected as they are projected from a three-dimensional world model onto a two-dimensional display. The projections most commonly used in computer graphics are called planar geometric projections. These projections are defined by specifying a point called a center of projection from which straight rays called projectors are extended to every point on the surface of the object being projected which is visible from the center of projection. These projectors intersect a plane called the projection plane, and it is the collection of intersection points on the projection plane which are called the projection of an object. [Foley90] describes the two basic families -- perspective and parallel -- into which planar geometric projections can be divided. In parallel projections, the center of projection is infinitely distant from the plane of projection and so the projectors are parallel, whereas in perspective projections the center of projection is a finite distance from the projection plane. The non-parallel projectors in perspective projections lead to a phenomenon known as perspective foreshortening. Figure 4-1 shows a typical perspective projection. Note that line B will appear to be shorter than line A in the projection even though the two lines are the same height, because line B is farther away from the center of projection than line A.



**Figure 4-1:** A typical perspective projection illustrating the effect of perspective foreshortening. A' and B' are the projected lengths of lines A and B, respectively.

The visual effect of a perspective projection is similar to that of the human eye as well as to that of photographic systems. As the desired result of this system is to simulate a sequence that was filmed by a camera, only perspective projections will be discussed in the paragraphs that follow even though it was explained in section 3.1.1 that perspective foreshortening is not currently implemented for 3D objects.

To specify an arbitrary 3D view requires not only a projection but also a 3D view volume which defines the portion of the world that will be seen in the final output. As explained previously, a projection is described by a point (center of projection) and a projection plane, also called a view plane. There are many different ways of specifying a plane, and the view plane is typically defined by a point on the plane called the view reference point (VRP), and a normal to the plane called the view plane normal (VPN). An additional vector, the view up vector (VUP), fixes the orientation of the view. If put in terms of a virtual camera, the vector from the center of projection to the VRP can be said to specify the pan and tilt of the camera, while the VUP specifies the roll.



**Figure 4-2:** Points and vectors defining a typical 3D view. All points in the world which fall within the pyramid created by the four rays extending from the Center of Projection will be seen in the final projection.

Next, since planes have infinite extent and most output devices do not, a window on the view plane needs to be defined. The VRP, VPN, and VUP together specify a new coordinate system called the view-reference coordinates (VRC) with the VRP as the origin and an axis along the VPN called the  $n$ -axis. The parallel projection of VUP onto the view plane specifies the  $v$ -axis, and the  $u$ -axis is then defined such that the  $u$ ,  $v$ , and  $n$ -axes form a right-handed coordinate system. The window is specified in terms of the VRC by giving minimum and maximum  $u$  and  $v$  coordinates. The view volume is then defined by extending an infinite ray from the center of projection through each of the four corners of the window to form a pyramid. Figure 4-2 shows a typical 3D view. In many computer graphics view specifications, a front and back clipping plane are also defined. However, since there is no easy method in hardware to cull points that would ordinarily be clipped in software by a rendering system and since the fact that the hardware  $z$ -buffer on the spatial

remapping card in the Cheops system is limited to 16 bits imposes inherent clipping planes, there is no need for them to be specified in this representation.

There are many ways of thinking about the view specification more intuitively. One possibility is to consider the window on the view plane to be a sheet of glass that has the same dimensions in world units as the window itself. Any object that falls on the view plane will not be affected at all by perspective foreshortening. This is useful in cases where the actual dimensions of the digitized objects are not known and only the pixel dimensions after sampling are known because the origin point of the background can be placed at the VRP so no scaling will occur, and then the positions of all other objects to be composited into the scene can be calculated using the formula for perspective foreshortening to achieve the proper proportions in the final output. A representation which more closely models the real world would be to consider the view window as the virtual film plane in a virtual photographic system and give it corresponding dimensions such as 24mm x 36mm. In this case, the eye-distance (the center of projection is often called the eye location) from the view plane is equivalent to focal length. The resultant projection is then multiplied by aspect ratios in each dimension to get the final size used for output or display.

The C declaration of the `View` structure is shown in figure 4-3. This structure is used by the script interpreter to define the various view parameters. Unless otherwise specified, values in the `View` structure are floating point because of the ease with which floating point numbers allow modeling of a three-dimensional world space. [Foley90] describes in detail how to go from the view parameters to the 4x4 matrix that can be used to project every point in the world into the 2D display coordinates, so that process is not described here. The fields used to specify 3D views in the `View` structure are:

- `state` - the `CurrState` structure and its associated fields are described in section 4.3.
- `view_ref_point` - 3D specification of the view reference point in world coordinates.

```

typedef struct ViewStruct {
    CurrState state;
    /* Specifies VRP */
    Point3 view_ref_point;
    Vector3 view_plane_normal;
    Vector3 view_up_vector;
    /* Eye */
    float eye_distance;
    /* The window */
    float u_min;
    float v_min;
    float u_max;
    float v_max;
    /* aspect ratios between window and viewport */
    Point3 aspect_ratio;
    /* Matrices */
    Matrix4 view_orientation_matrix;
    Matrix4 view_mapping_matrix;
    Matrix4 view_total_mapping_matrix;
} View;

```

**Figure 4-3:** The C declaration of the View structure.

- `view_plane_normal` - normalized 3D vector specifying the view plane normal in world coordinates.
- `view_up_vector` - normalized 3D vector giving the up orientation vector for this view in world coordinates.
- `eye_distance` - distance of the eye (center of projection) from the projection plane. In this implementation, the eye is assumed to be located `eye_distance` units away from the view reference point in the direction of the view plane normal.
- `u_min`, `v_min`, `u_max`, `v_max` - four points specifying the boundaries of the viewport (visible portion of the world) on the projection plane. If the projector from the eye to a point in the world does not lie within these boundaries, then that point will not be seen from this view.
- `aspect_ratio` - three values specifying the aspect ratio in the x, y, and z directions. The aspect ratio is the conversion factor that takes a point from world coordinates to integral pixel coordinates. In the x and y dimensions, these

numbers are based on the ratio of the screen dimensions to the viewport dimensions. In the z dimension, it is not yet entirely clear what is the best method of defining the aspect ratio. In the current implementation, the z aspect ratio merely counteracts the pixel-to-world conversion factor which is applied when the raw data is taken into world coordinates. Although the aspect ratios depend on the display parameters, they are included in the view structure because the final result of the rendering process will be a two-dimensional pixel array, regardless of whether that pixel array is displayed to a screen or merely saved to file for later viewing.

- `view_orientation_matrix`, `view_mapping_matrix` - steps along the way to creation of the `view_total_mapping_matrix`.
- `view_total_mapping_matrix` - the 4x4 matrix created from view parameters. For 3D objects, this matrix is composed with the individual transform matrix for each actor to give the final matrix with which each point in the actor will be multiplied. For 2D and 2-1/2D objects this matrix is used to transform the origin point of the actor which then determines the scale to be applied to the actor due to perspective foreshortening.

### **4.2.2 Actors**

Actors are instances of objects. Thus it is possible to use the same object data more than once in a single frame by having several actors which are instances of the same object and placing them at different locations in the world. One cow or a whole herd of cows could be placed in a meadow, or Michael Jordan could be made to play basketball against himself with only one object but several actors.

The primary parameters which are used to place an actor in the world are position of the actor, rotation of the actor around its own origin, and which view of the object is being presented if there is more than one view included in the object data. Depending on whether



```

typedef struct Actor {
    CurrState state;
    int interp;
    Object *obj;
    int obj_view;
    long    scale;    /* 2, 2 1/2 D objects */
    float  pos[3];
    float  rot[3];
    int    cache_line;
    Matrix4 transform_matrix;
    int    paste_point[3];
} Actor;

```

**Figure 4-4:** C declaration of the Actor struct.

the object of which an actor is an instance is two-dimensional (including 2-1/2D) or three-dimensional, these parameters may be used in different ways. The `Actor` structure which is used to represent an actor is shown in figure 4-4. The fields are used as follows:

- `state` - the `CurrState` structure and its associated fields are described in section 4.3.
- `obj` - pointer to the object description of which this actor is an instance.
- `obj_view` - which view of the object is being shown by this actor.
- `scale` - for 2D and 2-1/.2D objects only, the index into the array of structures which contain the information about which scaling factors and filter taps are to be used to obtain the calculated scale for this actor.
- `pos` - the three-dimensional position of this actor in world coordinates. For three-dimensional objects, this information is not accessed directly, but is used in combination with the `rot` field (described below) to create the transform matrix for this actor, which is then composed with the view matrix to create the final transformation matrix with which each point in the actor will be multiplied.
- `rot` - rotation (specified in degrees) of this actor around each of its three principal axes (x, y, and z) relative to the its own origin. For three-dimensional objects, this information is not accessed directly, but is used in combination with

the `pos` field (described above) to create the transform matrix for this actor, which is then composed with the view matrix to create the final transformation matrix with which each point in the actor will be multiplied. For 2D and 2-1/2D objects, this rotation can be used to find a normal vector for the orientation of the actor, which can then be used to decide which view of the object should be shown.

- `cache_line` - not currently implemented. A hook should object caching be added to the system to increase performance.
- `transform_matrix` - for 3D objects, the 4x4 transformation matrix created from the `pos` and `rot` fields described above. NULL for 2D and 2-1/2D objects.
- `paste_point` - for 2D and 2-1/2D objects only, the offset in pixels in each of the three dimensions indicating where the upper left corner of the actor will be composited into the frame.

### 4.2.3 Display Parameters

The display is the simplest of the frame elements and is described by a data structure called a `DispStruct`. In this system, a display is defined by four parameters: the x and y position of the upper left corner of the display window (offset from the upper left corner of the screen), and the x and y dimensions of the window. All of these values are integers, since they are all in terms of pixels. Figure 4-5 shows the C declaration of the `DispStruct`. The `CurrState` structure will be described in section 4.3.2.

In truth, the display parameters could be considered (and in many implementations of graphics systems are considered) part of the view parameters. Note, for example, that the view parameters described in section 4.2.1 include aspect ratios, which are dependent on the final display parameters. It was decided for this implementation, however, that separate control of the display window and the view parameters is a desirable feature. Another benefit of this separation is the reduction of the number of parameters kept track of

```

typedef struct DispStruct {
    CurrState state;
    int x_pos;
    int y_pos;
    int x_dim;
    int y_dim;
} DispStruct;

```

**Figure 4-5:** C declaration of DispStruct which defines display parameters.

by the view structure from seventeen to fourteen. Thus, to cause the display window to change position on the screen it is not necessary to enter seventeen values into a script command when fifteen or sixteen of them will not change.

## 4.3 Creating Sequences from Frames

Using the data structures described in section 4.2, it is possible to specify the view and display parameters as well as the position, orientation, and view to be used for every actor in a single frame. The next step is to put frames in order to make a video sequence. Two different methods exist for creating the frame descriptor for each frame in the sequence: create them all prior to play time or create the frame descriptor just prior to processing the data for the frame in which it is needed. Both approaches have been tried in this system.

### 4.3.1 Compiled Scripts

In the first implementation of this system, the decision was made to have the parser also act as a compiler and create a frame descriptor for each frame that would be displayed in the sequence prior to displaying any of the frames. The parser determined from the script how long the sequence would be, and then based on the frame rate of the output device created enough frame descriptors for a sequence of that length. The descriptors

were held in an array, and one by one were passed to the rendering and compositing system at the correct time to create the desired output sequence. The main motivation behind this decision was an attempt to minimize the number of calculations that would need to be performed on frame descriptors during the actual playing of the sequence in order to come closer to achieving real-time performance.

This compiler-like system did indeed work, particularly with sequences which involved only one story line and no interactivity. Simple reconstruction of a predefined video sequence lends itself easily to a representation which can be completely described and calculated ahead of time. However, interactivity by definition requires some calculations during the play time of the sequence and the ability to evaluate the state of certain parameters on the fly and respond to them. Also, Cheops system hardware is designed such that all transfers through the stream processors are handled by DMA controllers leaving the main CPU free for other duties. Since all values in the first implementation were precalculated, a great deal of CPU time was wasted in tight loops simply waiting for data to be processed or output to screen that could easily have been used to precalculate the parameters for the next frame descriptor or respond to user input. It was therefore decided that it would be preferable for this system to assemble frame descriptors at display-time, which would take advantage of previously wasted CPU cycles as well as give the system the ability to respond to user input or the current state of the system allowing for the implementation of interactivity and conditionals. In this second implementation, the script is now represented as key instances of views, actors, and displays which have key times associated with them, and algorithms are implemented for getting from one instance to another.

### **4.3.2 Interpreted Scripts**

The largest concern in moving to display-time creation of frame descriptors is that real-time functionality might be lost with the number of calculations that must be performed

each frame to determine view, actor and display parameters. As described in section 4.2.1 a view is defined by the VRP, VPN, and VUP, each of which contains 3 floating point values, as well as the window specification (4 values) and an eye distance for a total of 14 floating point values. Actors are specified by position, rotation, and view (6 floating point and 1 integer value), and display parameters by position and dimensions (4 integer values). In the case of views and 3D objects, the floating point values must be manipulated even further to create transformation matrices. It is obvious that the script cannot be reparsed before every frame to create a new frame descriptor, therefore an intermediate level must be created which can be quickly evaluated to obtain the necessary values for the next frame, but which also allows for user interaction and conditionals in the script.

The intermediate level needed is obtained through the creation of three additional data structures: the `preView`, `preActor`, and `preDisp`. These structures are themselves very straightforward, consisting of two substructures: the `ParamStruct` which is used to represent different types of data which are evaluated to obtain the values used in the frame descriptors, and the `LL_Struct` which contains fields necessary to create doubly linked lists of the "pre" structures for forward and backward traversing of the script.

As currently defined, the `ParamStruct` supports four different kinds of parameters labeled as `PARAM_TYPE_CONST`, `PARAM_TYPE_PARAM`, `PARAM_TYPE_KNOB`, and `PARAM_TYPE_FUNC`. These four types refer to constants, other parameters, knob/user inputs, and functions, respectively. The `ParamStruct` is set up generically to allow it to be used for representing other types of data as well, should the need arise. Figure 4-6 shows the C declaration of the `ParamStruct`. The fields are used in different ways depending on the type of data the structure represents:

- `in_use` - not used at run-time. Used in scripts to define macros (see appendix A for a description of a sample scripting language and sample script).
- `p_type` - parameter type, of the four so far defined.

```

typedef struct ParamStruct{
    int in_use;
    param_type ptype;
    int pnum;
    int (*func)();
    struct ParamStruct *args;
    int changed;
    float val;
} ParamStruct;

```

```

typedef struct LL {
    struct LL *next;
    struct LL *prev;
    int interp;
    float time;
} LL_Struct;

```

**Figure 4-6:** C declarations of ParamStruct and LL\_Struct. These structures are used to link individual instances of views, actors, and displays into a script.

- pnum - parameter number of the other parameter to which PARAM\_TYPE\_PARAM refers; knob number for PARAM\_TYPE\_KNOB.
- func - pointer to the function which will be called if PARAM\_TYPE\_FUNC.
- args - array of arguments which will be passed into the function if PARAM\_TYPE\_FUNC. The arguments themselves consist of ParamStructs, so that nested function calls and user input to the functions are supported.
- changed - set to true if the value to which this parameter evaluates has changed since the last time it was evaluated.
- val - floating point value to which this parameter evaluates. This field holds the constant value if PARAM\_TYPE\_CONST.

The LL\_Struct declaration is also shown in figure 4-6. It is what allows the connection of the "pre" structures into doubly linked lists to form a coherent script and describes how to get from one instance of a view, actor, or display to the next. It contains the following fields:

- `next`, `prev` - standard doubly-linked list pointers to the next and previous elements in the list.
- `interp` - specifies how the data represented by this instance of a view, actor, or display is to be treated until the next instance in the list. Three possible actions are currently supported:
  - i) `NO_INTERP` - no interpolation is performed between this instance and the next instance. The data is used exactly as it is represented in the current instance.
  - ii) `INTERP` - the data actually used is interpolated between the current instance and the next instance in the list. Currently linear interpolation is used in all cases, except possibly in the object view field of the actor structure. See appendix A for the reasoning behind this exception as well as how to use it.
  - iii) `REMOVE` - used only for actors. Indicates that the actor is not displayed in any frame until a later instance changes this status. Used primarily as a place holder if an actor is to make several appearances over the course of a script and not be displayed in between appearances.
- `time` - the time at which this instance of a view, actor, or display takes effect.
- `changeable` - a boolean which indicates whether or not this instance of a view, actor, or display is capable of changing. This is so far simplistically defined in that an instance is said to be not changeable if the "pre" structure for that instance represents only constants. Other parameters, user inputs (i.e. knobs), and functions are assumed to be able to change. A more robust system might check whether the other parameters themselves represented constants, or whether all the arguments to a function were constant and thus not changeable as well.

The doubly linked lists of "pre" structures correspond one-to-one with the `View`, `Actor`, and `DispStruct` structures used in the current frame descriptor. This means that there is one list describing the displays, one list for the views (a single-camera model is assumed), and one list for each of the actors that will be used. At the time the frame descriptor is compiled, each of the `ParamStructs` in the correct instance from each of these lists is evaluated and the resulting values are placed into the corresponding frame descriptor structures. Evaluation can be as simple as immediately returning a value in the case of a constant or another parameter, or it can involve a function call, or even polling an input device.

In order not to have to start at the beginning of each doubly linked list and locate the correct instance every frame, each `View`, `Actor`, and `DispStruct` structure contains a `CurrState` structure within it. The `CurrState` structure contains a pointer to the instance of the "pre" structure that is currently relevant, as well as a pointer to the next instance and the time at which that instance will take effect. The next pointer is also used for quick access to data if interpolation is to be performed. Along with these three fields the `CurrState` also contains other fields which are used to determine whether or not the view, actor, or display needs to be updated, thus gaining some speed by avoiding unnecessary calculations.

This second implementation does not preclude describing sequences in a frame-by-frame fashion as was done in the first implementation. Each frame could be considered a key instance with its associated complement of "pre" structures. A script this verbose might be expected from a scene analysis system, for example, or an automatic script generator. As long as the script is described in this linked-list format, however, it makes absolutely no difference what kind of system created the script. For a sample scripting language and some scripts written using it, see Appendix A.



# Chapter 5

## Example Applications for a Structured Video Decoder

Various applications have now been written using the structured video decoding system described in the previous chapters in conjunction with the scripting language described in Appendix A. Obvious first applications included resizing of a standard video sequence based on user input as well as allowing a user to explore a simple 2-1/2D or 3D space by manipulating a 2D actor.

A more recent application which is just beginning to be explored is the concept of "contextual resizing." In contextual resizing, the output sequence that is produced by the structured video decoder can be made to change based on the parameters of display to which the output is being sent, such as size or aspect ratio. If the display window is large or has a wide aspect ratio, for example, then perhaps the presentation can be almost theatrical in that the whole set can be seen and the viewpoint changes only rarely. On a smaller display window, perhaps the viewpoint moves to follow the main action in the scene or cuts to a close-up of the actor. The parameter-based changes in presentation are completely specified in the script, giving the director the control necessary to compose the sequence differently for display on a movie theater screen or a television screen, for example. Beyond differing viewpoints, however, changes in display could engender completely different story lines, such as a scene taking place set against a panoramic

backdrop (e.g. the rim of the Grand Canyon) on a large display, or in a much tighter space (e.g. a narrow forest path) on a smaller display. A large number of creative possibilities which were formerly not available for storytelling open up in a structured video environment.

Figure 5-1 shows one specific example which has been implemented of contextual resizing. In the sequence represented, an actress starts at the right side of the scene, walks across a room in a synthetic art gallery and out the door on the left side of the room, passing behind one statue and in front of another along the way. The gallery is prerendered into a 2-1/2D object, and the actress is represented as a series of views of a 2D object which simulate a walking motion. When the sequence is displayed in a large window, the viewpoint is stationary and remains centered on the entire room as the actress walks across the screen. In a smaller window, the view cuts in closer to the actress and tracks her as she walks amidst the statues and paintings of the gallery. Both sequences shown in the figure occur in the same amount of time. For a complete description of how this example was implemented including the script used, see Appendix A.

Other possible applications that have not yet been attempted on this system are to create stories in which the user determines the storyline. At certain key times throughout the script, the next set of actions portrayed by the sequence could depend on user input. A structured video system is also ideal for implementing Dolan's synthetic transition shot [Dolan94] in which a model of a bobsled and bobsled track are used in conjunction with a virtual camera to create a smooth transition in between two real camera shots that might otherwise cause disorientation to the viewer.



Figure 5-1: Screen dump of selected frames output by the contextual resizing application at two different window sizes.

# Chapter 6

## Results, Ideas for Future Work, Conclusion

The structured video decoding system described in this thesis has been implemented on the Cheops system and to varying degrees has been functioning regularly for several months. The low-level data processing pipeline hardware and software have been running basically unchanged for the past seven months while the script interpreter has undergone various revisions. As stated in section 1.3, decompression of the data as shown in figure 1-1 has not been implemented and remains a task for the future.

Timing tests performed on the system indicate that the hardware/software combination on a Cheops system configured with only one filter card and one spatial remapping card (so that none of the data transfers can be performed in parallel) is capable of compositing a single 2D color actor of about 400x200 pixels into a 512x512 pixel 2-1/2D color background at a rate of between six and seven frames per second. Adding two more actors to the scene slowed the system to just over three frames per second. Rendering and display of a color 3D object consisting of approximately 250,000 points can be performed at almost three frames per second. Adding more filtering and remapping stream processors to the Cheops configuration will certainly increase the frame rate as operations can then be performed in parallel. If the system configurability is limited to the three stream processor cards that can be placed on one P2 processor module, it appears that

the best configuration will be to have one filter card and two remap cards, as the remap card accepts data at a significantly slower clock rate than the filter card. This configuration has not been verified with actual tests as of yet, however.

A pleasant surprise came when testing the system after changing the script interpreter to create the frame descriptors at display time instead of ahead of time. Rather than a decrease in performance which was expected due to the number of floating point operations which are now being performed before each frame is processed, no change in performance occurred. Closer inspection of the system revealed that the time it takes for the Cheops system DMA controllers to transfer the composited frame to the output card is more than sufficient for the CPU to perform all necessary calculations to prepare the frame descriptor for the next frame, even if the operations are primarily in floating point arithmetic, as is currently the case. This becomes untrue if the display window happens to be very small (i.e. less than 128x128), but even in these cases performance degradation is minimal.

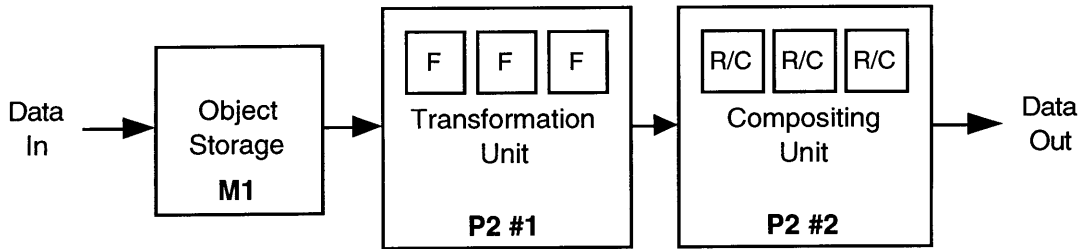
Additionally, the introduction of the scripting language and the structured video environment has considerably eased the task of using the Cheops system to test algorithms and create demonstration applications. Once the scripting language was debugged, it took the author approximately fifteen minutes to implement the contextual resizing application described in Chapter 5 and Appendix A. One half of the demonstration applications which are currently used on the Cheops system are also trivially implementable using the newly created environment with the caveat that the generality of the structured video system leads to slightly slower performance than can be achieved by creating the applications as optimized stand-alone demonstrations.

The system as it currently stands is far from perfect, however. One thing the system has definitely accomplished is to show some of the strengths and weaknesses of the Cheops system itself in terms of its use as a real-time video decoder. For example, a single filter card is capable of performing 4x4 matrix multiplications on 4x1 vectors at the rate of

6.7 million 4-vectors per second. But the current implementation of the hardware z-buffer and all of the operations that must be performed on the 3D data to create acceptable input streams to the z-buffer cause the overall system to be able to render and display only 1.1 million points per second. A color 3D object consisting of 250,000 points will require that 750,000 points be rendered which makes the system very slow as a 3D rendering unit. It is therefore evident that the hardware which will be used in future implementations of this system will have to undergo some revision. The hardware, however, is not immediately changeable.

On the software and system configuration side optimization can also be done to increase performance and flexibility. The current implementation of conditionals in the script representation and more particularly in the scripting language is not as flexible as might be desired. It is not easy at present, for example, to specify that the effects of a user input should result in two completely separate future courses for the storyline.

The possible output frame rate of the system can also be significantly increased by increasing the concurrency of operations on the data. As was mentioned previously, adding more stream processors to the system should dramatically increase the output possibilities. Ideally, in order to achieve maximum parallelism, it would be nice to have a configuration which includes three filter cards and three remap cards, so that all three channels of color objects can always be processed in parallel. This is particularly relevant to the transformation of 2D objects which may have up to five channels which theoretically can be processed in parallel. In the current implementation of the data processing pipeline, objects are transformed and then composited one by one because the amount of local memory on a P2 processor board limits the number and size of temporary work buffers which can be allocated and used. Both filter and remapping cards spend a significant amount of time idle while waiting for other processing to be performed on the data. One possible solution to this bottleneck would be to implement a system which includes two P2 processor modules, as shown in Figure 6-1. Each of the P2 modules could then be made



**Figure 6-1:** Implementation of the data processing pipeline consisting of two P2 processor modules. P2 #1 is configured with three filter processors, and P2 #2 with three remap/composite processors.

to correspond to one of the main functional units in the data processing pipeline and each could have three stream processor cards to perform operations in parallel. At present, the NORMAN resource management system does not support communication in between P2 modules. Along with splitting the processing in between two boards which increases the availability of both stream processors and local memory, the transformation unit and compositing unit could each be made into servers which buffer up requests for transforming and compositing objects and then perform them when possible so that the CPU can have more time to spend calculating complex frame descriptors instead of waiting in tight loops to control data flow.

Performance considerations aside, other implementations and/or uses for this system would also be interesting to explore. One possibility would be to make a system more similar to the animation system described in [Reynolds82] in which the actors are themselves pieces of code or objects which are not evaluated but are rather "invoked." Each actor would know how to evaluate itself and do the appropriate thing based on system parameters as well as the predefined script.

Interesting work has also been done recently with intelligent camera control [Drucker94] and 3D scene pre-visualization [Higgins94]. It would certainly be possible to create an intermediate application which could take a description output from Higgins' system, for example, and create a script which could be passed in to this structured video environment. Then, given a good model of the set (such as a detailed CAD/CAM model) as well as the necessary digitized actors, it would be possible to create a completely

synthetic movie from previsualization through output without ever even building the set or filming the actors directly performing the script.

This is not to say that creating such a system is as easy as it sounds. Research has just begun to scratch the surface of previsualization as well as structured video description and reconstruction. It can already be seen that the script representation chosen in this implementation will be inadequate to fully represent the broad new options for flexibility and variability possible in structured video environments. Many questions still need to be looked into. How do we describe this variability in a script? How should data be thought of and prepared when a structured video representation is the known final objective? Recent experience at a video shoot with this final purpose indicates that it is perhaps not as straightforward as simply filming a script.

On the whole, though, the outlook for the future of real-time structured video decoding and display is very promising. Scene segmentation and modelling of real-world objects (especially people) is far from perfect but continues to be an active area of research. Investigations into the requirements for real-time structured sequence synthesis are just beginning. The system implemented in this thesis is a useful start, but greater performance and flexibility can be envisioned and will even be required by future storytellers and sequence creators. Future refinements in both hardware and in algorithms will make structured video decoders very useful tools both technically and creatively.



# Bibliography

- [Bender90] Bender, W., Bove, V.M. Jr., Lippman, A., Liu, L., and Watlington, J., "HDTV in the 1990s: Open Architecture and Computational Video," *HDTV World Review, the journal for high definition and advanced television technology*, Volume 1, Number 3, pp. 11-15, Summer 1990.
- [Bove89] Bove, V.M. Jr., "Synthetic Movies Derived from Multi-Dimensional Image Sensors," PhD Dissertation, Massachusetts Institute of Technology, April 1989.
- [Bove94a] Bove, V.M. Jr., Granger, B.D., and Watlington, J.A., "Real-Time Decoding and Display of Structured Video," *Proc. IEEE ICMCS '94*, pp. 456-462, May 1994.
- [Bove94b] Bove, V.M. Jr. and Watlington, J.A., "Cheops: A Reconfigurable Data-Flow System for Video Processing," submitted to *IEEE Transactions on Circuits and Systems for Video Technology*, Revised August 1994.
- [Catmull72] Catmull, E., "A System for Computer Generated Movies," *Proc. ACM Annual Conference*, pp. 422-431, 1972.
- [Dolan94] Dolan, M.C. Jr., "...Not That They Win or Lose - But How You Watch The Game: Personalized Sports Viewing," SM Thesis, Massachusetts Institute of Technology, June 1994.

- [Drucker94] Drucker, S.M., "Intelligent Camera Control for Graphical Environments," PhD Dissertation, Massachusetts Institute of Technology, June 1994.
- [Foley90] Foley, J.D., van Dam, A., Feiner, S.K., and Hughes, J.F., *Computer Graphics, Principles and Practice*, Second Edition, Addison-Wesley Publishing Company, Reading, MA, 1990.
- [Higgins94] Higgins, S.C., "The Moviemaker's Workspace: Towards a 3D Environment for Pre-Visualization," SM Thesis, Massachusetts Institute of Technology, September 1994.
- [Kazui94] Kazui, F., Hayashi, M., and Yamanouchi, Y., "A Virtual Studio System for TV Program Production," *SMPTE Journal*, 103(6), pp. 386-390, June 1994.
- [McLean91] McLean, P.C., "Structured Video Coding," SM Thesis, Massachusetts Institute of Technology, June 1991.
- [Oppenheim89] Oppenheim, A.V. and Schaffer, R.W., *Discrete-Time Signal Processing*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1989.
- [Reynolds82] Reynolds, C.W., "Computer Animation with Scripts and Actors", *ACM Computer Graphics*, Vol. 16, No. 3, July 1982, pp. 289-296.
- [Shen92] Shen, I.J., "Real-Time Resource Management for Cheops: A Configurable, Multi-Tasking Image Processing System," SM Thesis, Massachusetts Institute of Technology, September 1992.
- [Various93] Various authors, *Cheops Documentation*, MIT Media Laboratory internal memo, February 1993.

[Wang94] Wang, J.Y.A. and Adelson, E.H., "Representing Moving Images with Layers," *IEEE Transactions on Image Processing*, Vol. 3, No. 5, September 1994, pp. 625-638.

[Watlington89] Watlington, J.A., "Synthetic Movies," SM Thesis, Massachusetts Institute of Technology, September 1989.

# Appendix A

## A Sample Scripting Language

Seeing as how such a large part of the structured video decoder functionality is determined by the frame descriptor that is created by the script interpreter, and also given how difficult and tedious it is to correctly initialize the hundreds of structures that will be required by any script of a decent length manually, it seemed not only appropriate but necessary to create a simple scripting language to test the functionality of the structured video decoder implemented in this thesis. The scripting language was never the focus of this thesis, however, and so the language here presented is doubtless incomplete. It is meant to be easily expandable, though, and hopefully gives a good start for the structured video environment.

The first section of this appendix presents the scripting language as it has been developed so far, describing each of the supported commands in detail. Subsequent sections show sample scripts using the current scripting language, then describe how to add a new command to the script as well as the required format for functions called in the script.

## A.1 Currently Supported Script Commands

The scripting language presented here resembles a simple computer graphics animation language. The supported commands can be broken down into five basic groups: general sequence control, user input and macros, object control, view parameter control, and display window control. In general, commands which begin with "SET\_" or "PLACE\_" involve no interpolation, and commands which start with "CHANGE\_" or "MOVE\_" involve interpolation from the previous instance of the actor, view, or display being changed or moved. In the following presentation, portions of commands which are optional are enclosed in square brackets ("[]"). Note also that the scripting language is not case sensitive as far as the commands are concerned, but for clarity all fixed portions of each command are capitalized.

### A.1.1 Sequence Control Commands

The default behavior of a script is to run once until the last instance of a view, actor, or display is reached and then quit. Both the display period and sample period default to the inverse of the frame rate of the destination display device in an attempt to display data as fast as is allowable. The following commands can alter the default behavior. All are optional.

**RUN\_TIME** <sequence length in seconds>

Defines the running time of the sequence. This command is not required nor is it necessarily heeded. The actual running time of the sequence is actually the greater of the time specified by the run\_time command and the maximum key time specified for any view, actor, or display instance in the script.

**DISPLAY\_PERIOD** <display period in seconds>

Defines the display period used by the decoder. Internally this number is actually converted to a frame count based on the frame rate of the destination display device and used as the number of frames on the display device which are allowed to pass before new data is displayed.

**SAMPLE\_PERIOD** <sample period in seconds>

Sets the sample period which is used as described in section 4.1.

**REPEATING**

Indicates that the sequence should be played repeatedly. When the script interpreter reaches the end of the script, it begins again at time 0.

**PALINDROMIC**

The sequence will be played palindromically.

### **A.1.2 User Input and Macros:**

Currently knobs are the only supported input device, this is easily changeable in the future as other input devices such as mice become available on the Cheops system. A certain number of parameters are made available to use as macros. These are labeled "param0" through "paramN", where N is the defined maximum number allowed. The parameters are just an array of `ParamStructs` as defined in section 4.3, and thus can be made to be constants, refer to other parameters, poll input devices, or evaluate functions. The parameters defined by the user are updated before every frame descriptor is created. By default the input devices the user acquires are also polled before the creation of every frame descriptor, but the device polling rate is user controllable. An input device should be

acquired as a param before it is used, and should only be referred to through that parameter.

**ACQUIRE\_KNOB** <knob number> <knob type> AS PARAM <param number>

Acquire the given knob (supported numbers are 0 through 7) as the desired parameter. Knob types are defined by the Cheops knob interface library and can currently be one of three types. Knobs that require defining parameters include them in parentheses:

- `linear(<min>, <max>, <gain>, <bias>)` -- The value returned by the knob is given by the equation:  $\text{knob\_value} * \text{gain} + \text{bias}$ , and clipped to be within the defined min and max values.
- `exponential(<min>, <max>, <base>, <gain>, <bias>)` -- The value returned by the knob is given by the function:  $\text{pow}(\text{base}, \text{knob\_value} * \text{gain} + \text{bias})$ , and clipped to be within the defined min and max values.
- `boolean` -- This knob type requires no parameters. The value returned by the knob is either 0 or 1.

example: `ACQUIRE_KNOB 0 linear(0, 360, 1.5, 0) AS PARAM 3` acquires knob 0 as a linear knob running from 0 to 360 with a gain of 1.5 and a bias of 0, assuming that knob 0 has not been acquired earlier or by another process and that parameter 3 has not been previously defined to some other value.

**DEFINE\_PARAM** <param number> <parameter definition>

Defines a parameter as a macro. Parameters can be defined to be constant numbers, other parameters, or function calls.

examples: DEFINE\_PARAM 0 3.14159  
DEFINE\_PARAM 1 PARAM0  
DEFINE\_PARAM 2 MIN(PARAM0, 10.3)

**SET\_PARAM** <param number> <value>

Sets the value of a parameter. Only constant parameters and parameters which refer to input devices can be set with this command. This command is primarily used to set a previously acquired knob to a desired initial value.

**KNOB\_READ\_PERIOD** <period in seconds>

Sets the input device polling period. Input devices (knobs so far) are polled every knob read period and the values of acquired devices are stored in an array.

Parameters which access input device values actually read the values out of this array and do not poll the input device directly. If knob\_read\_period is not set, the input devices will be polled before the creation of every frame descriptor.

### **A.1.3 Defining and Controlling Actors**

This set of commands allows for the loading of object data and defining actors to be instances of those objects. Commands adding instances of actors do not need to be placed in the script in ascending order of their key times. However, an actor must be defined by either `LOAD_OBJECT` or `REUSE_OBJECT` before it can be referred to anywhere else in the script. Backgrounds are just large, immovable actors and so the same commands are used to refer to them.

Several defaults apply to actors. If position is not specified, it defaults to the world origin (0, 0, 0). Rotation defaults to (0, 0, 0) as well. Likewise, if view or time are not specified, they also default to 0. Object filenames and actor names *are* case sensitive and must be typed exactly the same throughout the script.



In `PLACE_OBJECT` and `MOVE_OBJECT`, position, rotation and object view arguments are parsed as `ParamStructs` and so can be constants, parameters, or function calls. Times are parsed as constant numbers in the current implementation, although there is no particular reason that they also are not parsed as `ParamStructs`. Position and rotation arguments are required to be 3-vectors. If one value of the triple is specified, then the other two must also be, even if they are 0.

**LOAD\_OBJECT** "<filename>" **AS** <actor name>

Loads object data from the given filename (which must be enclosed in quotes) and creates an actor with the given actor name as an instance of that object. Files are accessed in the usual manner for datfiles (see Appendix B for the object file format) and so submatrixing can be included in the filename.

example: `LOAD_OBJECT "museum[d 0 0+480 0+512]" AS museum`  
loads the submatrixed data from the datfile `museum` and creates an actor called "museum" as an instance of that object data.

**APPEND\_OBJECT** "<filename>" **TO** <actor name>

Appends data from the specified file to the object of which actor name is an instance (this is to avoid having to type out very long filenames again). The data in the file being appended must have the same dimensions (whether originally or by submatrixing) as the object to which the data is being appended and must also be of the same type (2D, 2-1/2D, or 3D). This can be useful when trying to assemble a series of views of an object into a coherent action, such as walking and turning, which use raw data from different files. All actors which are instances of the same object data will also see all data which is appended into an object.

**REUSE\_OBJECT** "<defined actor name>" **AS** <new actor name>

Creates an actor with the new name which is an instance of the same object as the actor with the already-defined name (again, to avoid retyping filenames). In this manner, the data does not have to be reloaded for the new actor. The two actors access the same data, but can access it with completely different parameters, so that the actors can be performing entirely separate actions.

**PLACE\_ACTOR** <actor name> **AT** [pos=(<x>,<y>,<z>)]

[rot=(<x>,<y>,<z>)] [view= <object view>] [AT time= <time>]

Places the specified view of the named actor at the specified point with the given orientation at the given key time. If no arguments are given, they default to the values described at the beginning of the section on actor commands. When an actor is placed, no interpolation from any previous instance of the actor is performed. Also, if the actor was not being displayed previously, this command turns the actor on for display until it is removed again (see below).

**MOVE\_ACTOR** <actor name> **TO** [pos=(<x>,<y>,<z>)]

[rot=(<x>,<y>,<z>)] [view= <object view>] [AT time= <time>]

Moves the actor to the specified position, orientation, and view at the specified time. Linear interpolation is used from the immediately preceding instance of the actor in the script, if one exists. The one possible exception to the linear interpolation comes in the determination of the object view. It is very likely that some objects will have multiple views which form a sequence simulating an action such as walking, which should be played as a sequence and not necessarily interpolated between. Thus, if the destination view in the **MOVE\_ACTOR** command is set to be greater than or equal to the number of views the object contains, the script interpreter will cycle through the object views in ascending

order to give the appearance of the desired action. Likewise, if the destination view is set to be less than 0, the script interpreter will cycle through the object views in descending order.

**REMOVE\_ACTOR** <actor name> [AT time= <time>]

Indicates that the named actor is not to be displayed from the time specified until it is turned on again by a **PLACE\_ACTOR** or **MOVE\_ACTOR** command. Actors are by default "removed" until they are explicitly placed.

#### **A.1.4 Defining and Controlling View Parameters**

Views are defined by a large number of parameters. What these parameters are and what they each define is explained in section 4.2.1. As with actors, all of the parameters are completely optional, and all but the time are parsed as ParamStructs so they can be constants, parameters (by which input devices are accessed), or function calls. The defaults for each of the parameters is:  $VRP=(0, 0, 0)$ ,  $VPN=(0, 0, 1)$ ,  $VUP=(0, 1, 0)$ ,  $VP= \min(-1, -1) \max(1, 1)$ ,  $eye\_distance = 1$ , and  $time = 0$ .

```
SET_VIEW TO [VRP=(<x>, <y>, <z>)] [VPN=(<x>, <y>, <z>)]  
[VUP=(<x>, <y>, <z>)] [VP= min(<umin>, <vmin>)  
max(<umax>, <vmax>)] [ed = <eye_distance>] [AT time =  
<time>]
```

Set the current view to the specified parameters at the given key time. The view is not interpolated from any previous instance of the view which may have occurred in the script.

```
CHANGE_VIEW TO [VRP=(<x>, <y>, <z>)] [VPN=(<x>, <y>, <z>)]  
[VUP=(<x>, <y>, <z>)] [VP= min(<umin>, <vmin>)
```

```
max(<umax>,<vmax>)] [ed = <eye_distance>] [AT time =  
<time>]
```

Interpolate from the previous instance of the view parameters in the script if they exist to arrive at the specified view parameters at the given key time.

### **A.1.5 Defining and Changing the Display Window**

The only controllable aspects of the display window are its dimensions and the position of its top left corner. As with actors and views, all parameters are optional and all but time are parsed as ParamStructs. Defaults are: dims=(512, 512), pos=(-1,-1), time=0. Negative values in either position axis (x or y) signify that the window is to be centered on the screen along that axis.

```
SET_DISPLAY TO [dims = (<xdim>,<ydim>)] [pos =  
(<xpos>,<ypos>)] [AT time = <key time>]
```

Set the display parameters as defined at the given key time. No interpolation with previous display parameters is performed.

```
CHANGE_DISPLAY TO [dims = (<xdim>,<ydim>)] [pos =  
(<xpos>,<ypos>)] [AT time = <key time>]
```

Interpolate from the previous instance of the display parameters in the script (if they exist) to arrive at the specified display parameters at the given key time.

### **A.1.6 Other Commands**

Other commands which have not yet been fully implemented or tested deal largely with using the decoder as a two-dimensional transform decoder such as would be used with layers or MPEG-like sequences. These include:

**ADD\_ERROR** <error object name> [TO <actor name>]

Error signals can easily be loaded as objects and assigned to a name, just as actors are. This command specifies that an error adder is to be enabled. If an actor name is specified, the error is assumed to be added to that actor after it has been transformed and before it is composited. Otherwise, the error is assumed to apply to the whole frame after compositing has occurred. Error signals are generally created by encoders which already know what transformations will be applied to an object, and so the error signal is expected to be "synchronized" with the data to which it will be added in the sense that there will be an error signal for every frame created by the script interpreter. Thus, an "AT <time>" extension does not seem necessary for this command.

**APPLY\_TRANSFORMATION** <explicit transformation name>

[zoom = <zoom factor>] TO <actor name> [AT time = <time>]

As with errors, explicit vector field transformations can be loaded as objects and given names. This command specifies that the named transformation should be applied to the given actor. If the transformation consists of only one vector field, it is applied to the actor every frame. If multiple vector fields comprise the transformation (analogous to multiple object views), then they are applied in order. If a time is specified, the transformation is applied to the actor starting at the specified time. The optional zoom factor allows vectors to operate on blocks of pixels such as would be the case with MPEG encoded sequences.

**APPLY\_AFFINE** <a b c d e f> TO <actor name> [AT time = <time>]

Applies a transformation to a 2D object similar to applying an explicit vector field. In this case the affine parameters are used to generate the vector field of appropriate

dimensions which is then applied to the named actor, continuously or at the specified time. The vector field is created according to the affine equations:

$$x' = ax + by + c,$$

$$y' = dx + ey + f$$

**FEEDBACK** [<actor name>] [AT time = <time>]

This command enables a feedback path. If an actor name is specified, then that specific actor is fed back to be used next frame, otherwise the entire composited frame is fed back to the transformation unit. If a time is specified, then the feedback is "one shot" and happens only at the specified time. If no time is specified, then the feedback is assumed to be continuous. If feedback is not desired every frame, a feedback period can be specified (see below).

**FEEDBACK\_PERIOD** <feedback period>

Specifies how often feedback should occur along the path which has been enabled by the FEEDBACK command. Internally this is converted to a frame rate and based on the frame counter.

## A.2 A Simple Script: Contextual Resizing

Using the commands described in the previous sections, it is fairly straightforward to design scripts. For example, a script which will cause an actor to walk across a background while the viewpoint does not move can be straightforwardly written as:

```
# script to make an actress walk across a room
LOAD_OBJECT "data/mtemp[d 0 0+480 0+512]" AS museum
LOAD_OBJECT "data/btmp90" AS audrey_walk

DISPLAY_PERIOD 0.2 # slow it down some...
```

```

PALINDROMIC

SET_VIEW TO ed= 14.2 VP= min (-7.2, -7.2) max (7.2,
7.2) AT time= 0.

PLACE_ACTOR museum AT time= 0.

PLACE_ACTOR audrey_walk AT pos= (5.3, -1.2, -5.48)
view= 0 AT time= 0.

MOVE_ACTOR audrey_walk TO pos= (-9.5, -1.2, -5.48)
view= 8 AT time= 5.6

```

In the script above, the first line exhibits how submatrixing can be used with the object files. One data set of the file data/mtemp is cropped to 512x480 pixels and loaded as an actor named "museum." As it happens, museum is a 2-1/2D object which was prerendered from a 3D computer database and is a room in a virtual art gallery which will be used as the background. The file data/btmp90 is loaded as an actor called "audrey\_walk," which actually consists of eight 2D views of an actress that together form a walking sequence. The display period is set to 0.2 seconds (5 frames per second) to keep the sequence reasonably slow, and the sequence is set to be shown palindromically.

The SET\_VIEW command sets the view parameters, and they will not change throughout the duration of the script. Since the display parameters are not mentioned at all in the script, they become the default which is a 512x512 window centered on the screen. The museum is simply placed at time 0. All position and rotation defaults are applied in this case, which means that the origin of the museum is placed at (0, 0, 0) and there is no additional rotation applied to the object. The placement of the origin to coincide with the VRP of the view means that perspective foreshortening will not be applied to the museum, which is the desired treatment of 2-1/2D objects.

Finally, the actor audrey\_walk is placed at its initial location at time 0, and then told that it will move to a second position at 5.6 seconds. Linear interpolation will be used to go from the initial state to the final state, except that since the destination view is set to be

equal to the number of views in the object of which `audrey_walk` is an instance, the script interpreter will actually cycle through all the views in the object in ascending order to create the impression that the actor is indeed walking.

A slightly more complex script (but just barely) can be written to have the camera track the actor and display the output on a smaller window:

```
# script to track an actor on a small window
LOAD_OBJECT "data/mtemp[d 0 0+480 0+512]" AS museum
LOAD_OBJECT "data/btmp90" AS audrey_walk

DISPLAY_PERIOD .2
PALINDROMIC

# make the window smaller
SET_DISPLAY TO dims = (128, 128) AT time = 0.

SET_VIEW TO ed= 14.2 VP= min (-1.8, -1.8) max (1.8,
1.8) VRP = (5, 0, 0) AT time= 0.

CHANGE_VIEW TO ed= 14.2 VP= min (-1.8, -1.8) max
(1.8, 1.8) VRP = (-5.3, 0, 0) AT time= 4.

# set the background:
PLACE_OBJECT museum AT time= 0.

PLACE_OBJECT audrey_walk AT pos= (5, -1.2, -5.48)
view= 0 AT time= 0.

MOVE_OBJECT audrey_walk TO pos= (-9.5, -1.2, -5.48)
view= 8 AT time= 5.6
```

The differences between this script and the script presented previously consist of changes in the display parameters and the view parameters only -- the actors are unaffected. In this script, the window is set to 128x128 (but it remains centered, since it was not explicitly set otherwise), and the initial view is centered on the `audrey_walk` actor. The parameters to the `CHANGE_VIEW` command are calculated such that the walking actor remains centered in the window. The view stops panning before the actor stops walking so



that it does not pan beyond the limits of the background. Once the left wall is reached (at 4.0 seconds), the view stops panning but the actor continues walking to the left and out the door.

As a final example, the two previously described scripts can be combined, one more intermediate size can be added along with knob control of window dimensions and position to create the contextual resizing application which is described in chapter 5:

```
LOAD_OBJECT "data/mtemp[d 0 0+480 0+512]" AS museum
LOAD_OBJECT "data/btmp90" AS audrey_walk

DISPLAY_PERIOD .2
PALINDROMIC

#knob 0 is display size. xdim = ydim. Must be a
#multiple of 8 for Nile to work correctly.
ACQUIRE_KNOB 0 linear(64, 512, 8, 0) AS param0

#start display at 512x512
SET_PARAM 0 512.

#knob 2 is x position, must be even, so set gain=2
ACQUIRE_KNOB 2 linear(-2, 768, 2, 0) AS param1

#negative initial value means centered
SET_PARAM 1 -2.

#knob 3 is y position
ACQUIRE_KNOB 3 linear(-2, 512, 2, 0) AS param2

#center to begin with
SET_PARAM 2 -2.

# set up a bunch of params to be used like macros:
DEFINE_PARAM 3 split3(param0, 256, 128, -7.2, -3.6,
-1.8)
DEFINE_PARAM 4 split3(param0, 256, 128, 7.2, 3.6,
1.8)
DEFINE_PARAM 5 split3(param0, 256, 128, 0, 3.6, 5)
DEFINE_PARAM 6 split3(param0, 256, 128, 0, 3.6,
3.6)
DEFINE_PARAM 7 split3(param0, 256, 128, 0, -3.6,
-5.3)

#display size and position controlled by knobs:
```

```

SET_DISPLAY TO dims = (param0,param0) pos =
(param1, param2) AT time = 0.

#ok, set the view to act appropriately based on
#display size
SET_VIEW TO ed= 14.2 VP= min (param3, param3) max
(param4, param4) VRP = (param5, 0, 0) AT time= 0.

CHANGE_VIEW TO ed= 14.2 VP= min (param3, param3)
max (param4, param4) VRP = (param6, 0, 0) AT time=
0.5

CHANGE_VIEW TO ed= 14.2 VP= min (param3, param3)
max (param4, param4) VRP = (param7, 0, 0) AT time=
4.

#actors still do the same old thing...
PLACE_ACTOR museum AT time= 0.

PLACE_ACTOR audrey_walk AT pos= (5., -1.2, -5.48)
view= 0 AT time= 0.

MOVE_ACTOR audrey_walk TO pos= (-9.5, -1.2, -5.48)
view= 8 AT time= 5.6

```

While this script looks a lot more complicated than the previous two, it really is not. It does, however, show more of the functionality of the scripting language. For example, knobs 0, 2, and 3 are acquired as linear knobs which will control display window dimensions and position, and are assigned to parameters 0 through 2. Parameters 3 through 7 are set up as macros to reduce the typing that will need to be done later in the script. The function `split3(curr_val, level1, level2, val0, val1, val2)` returns `val0` if `curr_val` is greater than `level1`, `val1` if `curr_val` is greater than `level2` but less than `level1`, and `val2` if `curr_val` is less than or equal to `level2`.

The actors are still animated as they were before -- the museum is stationary and the actor walks from left to right. The display is put under knob control. Only the view is complicated as it has been defined to react in various ways based on different display dimensions.

Figure 5-1 shows the output over time created by this script on two different-sized display windows. In the larger window, the viewpoint is stationary and the actor walks across the room, while in the smaller window the view pans to follow the actor and the background can be seen to be passing by.

The scripting language has been very useful for defining scripts which test the system as well as for creating more sophisticated scripts like the one just presented. It is very likely, however, that additions to the script will need to be made in future as more applications are attempted. Therefore, the next section describes how to add functionality to the scripting language.

### **A.3 Adding New Functions to the Scripting Language**

As described in section 4.3.2, one of the supported parameter types is `PARAM_TYPE_FUNC` so that functions can be implemented in the scripting language. At the level of the scripting language, functions resemble C function calls -- the function name cannot begin with a number, and all arguments to the function immediately follow the function name (spaces are allowed) and are enclosed in parentheses and separated by commas. No inherent functions exist in the scripting language, and so any function the user wishes to have must be implemented and added as described in this section, no matter how simple it may be.

Three separate arrays comprise the function call interface in the scripting language created here, and to be supported a function must have an entry in each of the three. The first is an array of supported function names, stored as strings. The second is an array of pointers to the functions to which those names correspond. Note that this means that the function name in the scripting language and the name of the C function used to implement it

do not have to be the same. Finally, the third is an integer array specifying the number of arguments that are expected by the functions. The function name, function pointer, and number of arguments are expected to occupy the same position in each of their respective arrays. Three arrays are used as opposed to a single array of a new structure containing three fields to allow for static initialization of all the arrays involved.

When the script parser encounters a non-numeric argument to one of the supported script commands, it assumes that it has encountered a function call and searches through the string for a left parenthesis, which confirms that it is a function call. Objects which require no arguments should still have a set of parentheses enclosing no arguments, as in C. The function name (up to the left parenthesis) is compared to the array of supported command names. If a match is found, the corresponding function pointer is stored in the `func` field of the `ParamStruct`, and an array of the appropriate number of `ParamStructs` is allocated to be arguments to the function, and pointed to by the `args` field. Arguments to script functions are parsed identically to arguments to script commands, and so function arguments can be constants, references to other parameters (macros and input devices), or function calls themselves which allows for nested function calls. All function calls are expected to eventually evaluate to floating point values.

The C functions which implement script functions must therefore adhere to a common pattern and a new type, called a `script_func` is defined for this purpose. `Script_funcs` are defined to be functions which take two arguments -- a pointer to an array of `ParamStructs` and a pointer to a float -- and return an integer. So far the returned integer has not been used, but it is left as a hook which might be used in a more intelligent script interpreter to determine whether the value returned by the function call is capable of changing. An example of implementing a function which returns the minimum of two numbers would be:

```
int sc_min(ParamStruct *args, float *ret)
{
    float val1, val2;
```

```
    eval_arg(&args[0], &val1);
    eval_arg(&args[1], &val2);
    *ret = (val1 < val2) ? val1 : val2;

    return (TRUE);
}
```

Therefore, to add a new function to the scripting language it is necessary only to implement it in C following the definition of the `script_func` type and then add the name, function pointer, and number of arguments to the appropriate arrays.

As it happens, script commands (such as `PLACE_ACTOR`, etc.) are implemented very similarly to script functions. However, as the process of adding new script commands involves actually changing the parser, it is not described in this paper.

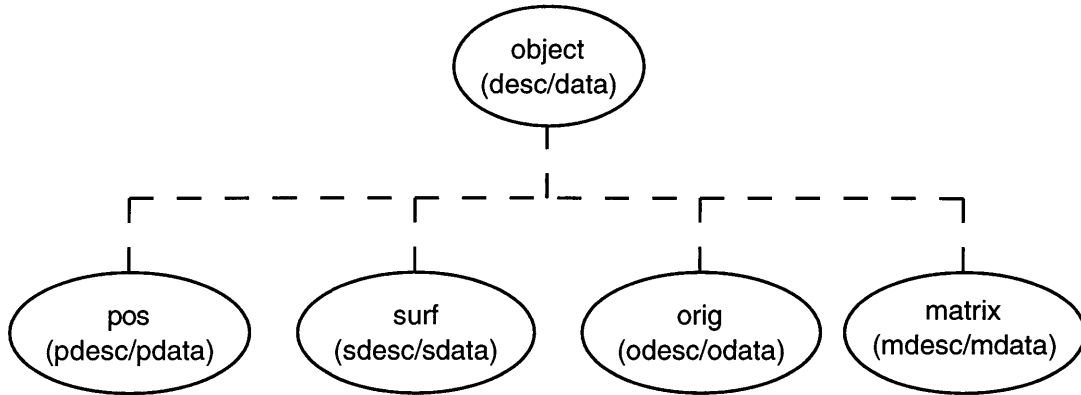
# Appendix B

## Structured Video Object File Format

In a structured video decoder system, it is quite possible that the objects may be transmitted along with the script and decoded as they are received or transmitted ahead of time and stored in local memory at the receiver. In the system implemented in this thesis, however, structured video objects are stored in files and loaded as called upon from the script.

A common file format for representing multi-dimensional data, such as 2D image sequences or 3D particle databases, which is used at the MIT Media Lab is the "datfile" format. The structured video object file format is an extension of the datfile format. For a complete description of this format, see the dat man pages. A dat"file" is actually a directory which contains two files: a descriptor file and a data file. The descriptor file describes the format of the data file in terms of the data type (byte, integer, etc.), number of data sets, channels of data per data set, and dimensions of the channels. Provisions are also made for user-defined "keys" in the descriptor file which can be used to describe extensions to the file format.

One of the default behaviors of the decoder implemented in this thesis is to revert to a simple 2D movie player if a standard 2D datfile is received with no script describing any other action to be performed on the data. Thus it seemed appropriate that the object file format be made an extension to the already-in-use datfile format. Figure B-1 shows a tree



**Figure B-1:** Tree representation of structured video object datfile format. Dashed lines indicate that all subdatfiles are optional .

representation of the structured video object file format. Ellipses indicate datfiles or "sub"datfiles, each of which has its own descriptor and data files. A description follows of the data format of each of the datfiles in the tree along with any special keys which have been defined in their respective descriptors.

### **Main datfile:**

**data ---** RGBA, 1-4 channels. Byte format.

1 channel: assumed luma only

2 channel: luma with alpha

3 channel: color

4 channel: color with alpha

If the object is 2D or 2-1/2D, the data is two-dimensional and assumed to be presented in raster order. Since 3D object data points can be presented in any order, each channel of a 3D object is one-dimensional with a number of points corresponding to the total number in the object in each channel.

**desc ---** special keys:

`_default_z`: Base depth for 2D object data. Changeable in the script, using `PLACE_ACTOR` command.

`_pos_file`: Name of position subdatfile.

`_orig_file`: Name of origin points/scale subfile.

`_surf_file`: Name of surface orientation subfile.

`_matrix_file`: Name of matrix subfile.

**NOTE:** All subdatfiles are optional. The current implementation will not check for the existence of subdatfiles if the corresponding key is not defined in the descriptor file. A datfile with no defined special keys will be treated as a 2D object and will be defaulted to the background depth.

### **Subdatfiles:**

- **pos**: Contains the depth or position data for 2-1/2D and 3D objects.

**pdata** --- [xy]z. 1 or 3 channels. Signed 16-bit ('d' or Integer2) format.

1 channel: Z. Object assumed to be 2-1/2D.

3 channel: XYZ. 3D particle database assumed.

As in the main datfile, 2-1/2D data is presented in two-dimensional raster order. Each of the 3D channels is one-dimensional.

**pdesc** --- special keys:

`_gain`: Gain which is to be uniformly applied to all position data channels.

- **orig**: Defines origin points and pixel to world scaling factors for 2D and 2-1/2D objects.



**odata** --- One channel, one-dimensional, containing 3 floating point values: `<origin_x>` `<origin_y>` `<scale factor>`, for each view of the object.

`scale factor`: Floating point value which converts from pixel resolution to world units. Scaling is assumed to be uniform in x and y directions.

`origin_x`, `origin_y`: Offsets from the upper left corner of the 2D data to the point which should be used as the origin of the object and upon which all placement and path planning will be based. The pixel offset values are pre-multiplied with the scale factor to get the origin offset in world coordinates.

**odesc** --- special keys: None currently defined.

- **surf:**

**sdata** --- PQ surface orientation. 2 channels. Float representation.

**sdesc** --- special keys: None currently defined.

- **matrix:**

**mdata** --- 1 channel 4x4 orientation matrix for object. Float representation. A hook for possible future support of texture mapped polygons.

**mdesc** --- special keys: None currently defined.

With the above outlined file format, it is assumed that 2D objects will generally consist of only a main datfile with an origin/scale subfile; 2-1/2D objects will always have a 1-channel position file and usually an origin/scale file in addition to the main data file; and 3D objects will have a 3-channel position subfile. In addition, the specified file format

allows current 2D image sequences stored as datfiles to be treated as 2D objects in the 3D world or simply be played back as video sequences with no need for data or format conversion. Also, many other types of objects which are not currently supported by the decoder should be supportable with this file format or can be easily supported by the addition of more specially defined descriptor keys.

# Appendix C

## Pipeline Implementation Specifics

This appendix is intended primarily for people who will be making future modifications to the data processing pipeline implemented in this thesis. It describes the current implementation, as well as discussing some of the reasoning behind the current design. It is assumed that the reader of this appendix is familiar with the Cheops system as well as the resource management interface library (RMAN). For additional information on these topics, see [Various93].

Discussion of the various components in the pipeline will proceed in largely the same order as the overview of the pipeline which is presented in Chapter 3. For each major component data flow diagrams are presented. In these diagrams, circles (or ellipses) will represent memory buffers and rectangles will represent operations performed using Cheops system hardware.

### C.1 The Transformation Unit

As described in section 3.1, the transformation is broken up into two units: one for processing 3D objects and one for processing 2D and 2-1/2D objects. Both units are implemented using the Cheops filter stream processor, but the format of 3D data as

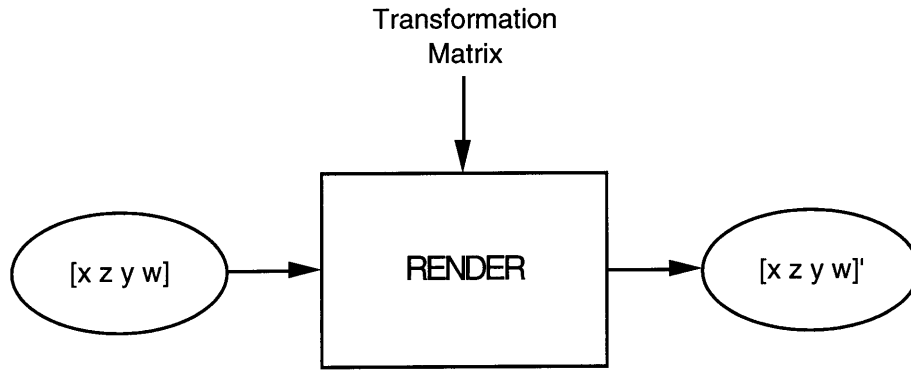
opposed to 2D data (including 2-1/2D) is so different and the manner in which the filter card is used differs so greatly in the two units that they each are described in separate sections.

### **C.1.1 The 3D Transformation Unit**

The 3D transformation unit is perhaps the easiest to implement of all the components in the pipeline. One of the possible configurations of the filter stream processor is the "render" mode in which a preloaded 4x4 matrix is premultiplied with a stream of 4x1 vectors, which just happens to be ideal for the representation of three-dimensional data points in homogeneous coordinates. Thus, when the 3D object data is read into memory the x, y, and z channels are interleaved with a w value to create a 4-vector which is ready to be passed through the transformation unit when necessary. The matrix which is preloaded into the hardware is the composition of the view matrix defined by the view parameters with the individual transform matrix for each 3D object. The reader is again referred to chapter 6 of [Foley90] for a detailed explanation of how these matrices are created.

Only several hardware details prevent the implementation of this unit from being completely straightforward. The largest is actually due to the manner in which the remap/composite processor which is used in the compositing unit requires data to be presented to it (see section C.2). In order to ease the later creation of the xy stream that will be required in the compositing unit, the 4-vector which is stored in memory is not the more standard [x y z w] but is instead [x z y w]. This means merely that in the matrix which is multiplied with the 3D data, rows 2 and 3 (with rows numbered 1 through 4) must be swapped, and columns 2 and 3 must also be swapped.

Figure C-1 shows a data flow diagram of the 3D rendering unit which consists of a single transfer through the Cheops filter hardware in render mode after the matrix has been preloaded. This makes geometry transformations of 3D objects very fast, since each filter



**Figure C-1:** Dataflow diagram for the 3D transformation unit. The transformation matrix is preloaded into the rendering hardware.

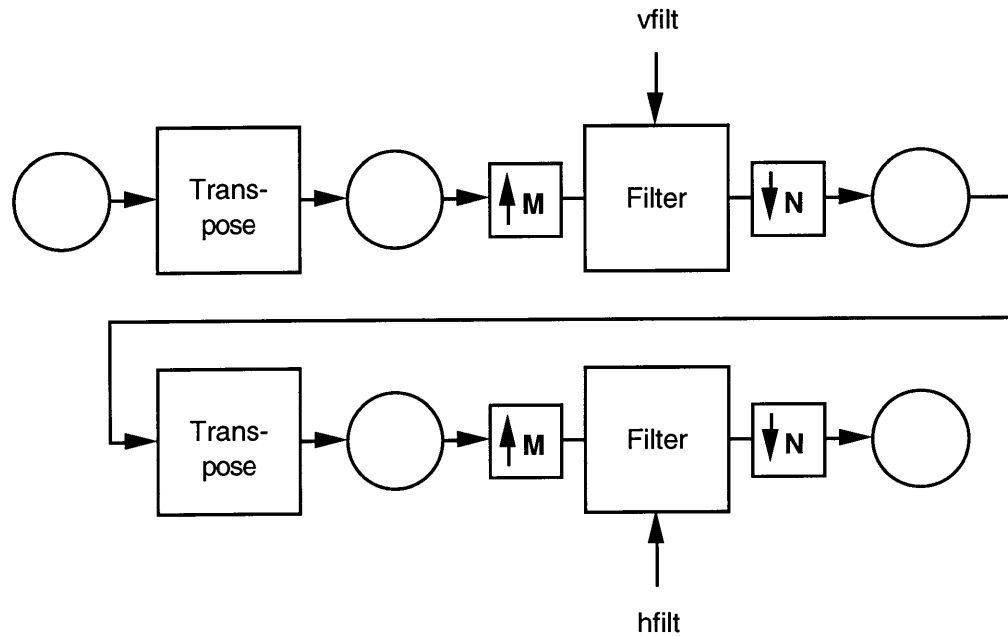
sub-module contains two renderers which can run in parallel, and each renderer can process 6.7 million points per second.

### C.1.2 2D Transformation Unit

As stated in section 3.1.2, the 2D and 2-1/2D transformation unit is implemented using separable two-dimensional scaling filters. This is a well-known and often-used operation in the Cheops system, and its data flow diagram is shown in Figure C-2. Up to five of these filtering pipelines may be requested in parallel if an object has three color channels as well as a z-buffer and an alpha buffer.

As indicated in the figure, the vertical and horizontal filters can be different. That allows for output to displays whose aspect ratios are not square even though the current implementation assumes square aspect ratio. If differing aspect ratios are desired, the change is trivial.

There really are no particular details that must be watched out for in this pipeline -- it is composed of simple `RmanTranspose` and `RmanFilter` elements.



**Figure C-2:** 2D transformation unit implemented using two-dimensional scaling filters.

## C.2 The Compositing Unit

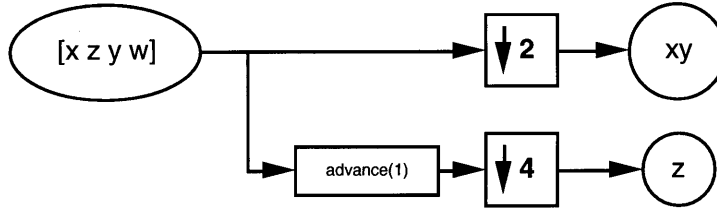
The implementation of the compositing unit itself is quite straightforward using the remap/composite sub-module in the Cheops system. However, a substantial amount of data rearrangement must occur in between the transformation unit and the compositing unit in order to create the data streams expected by the remap/composite module. There are also several possible manners of thinking about how to use the compositing hardware. These points are discussed in this section along with a discussion of a hardware versus a software implementation of the composer.

### C.2.1 Stream Rearrangement

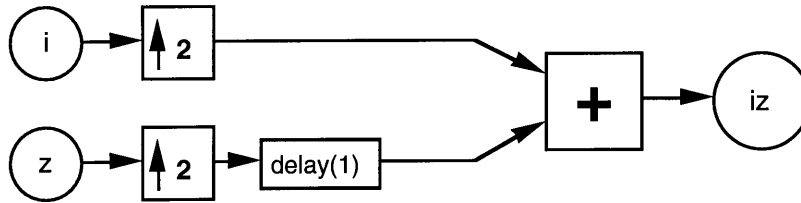
The remap/composite sub-module in the Cheops system hardware is called a "two-phase" device. This means that the device accepts data and stores it in internal memory in

one phase and then outputs data which may have been processed in another phase. In the input phase (also called the "write" phase since data is being written into the device) the composite module in z-buffer mode accepts up to two streams of data. One stream is composed of intensity and z values interleaved, and this stream is required for all object types. The second stream is present only for 3D objects and consists of the x and y position values interleaved. The transformation unit does not process nor does it output data in the same format, so these streams must be created before the hardware z-buffer can be used.

As described in section C.1.1, 3D object data is processed by the transformation unit as  $[x\ z\ y\ w]$  vectors. The built-in capability of the Cheops flood controllers to zero-pad, replicate, and decimate streams of data makes it quite easy to separate these 4-vectors to create an xy stream and separate out the z to be interleaved with the intensity using only stream transfers and no other hardware. Figure C-3 shows how this is done in a data flow diagram (down arrows signify decimation by the indicated factor). Advancing the data stream by one sample is actually accomplished by telling the destination flood controller that the delay before data is valid is one sample more than it really is. This causes the destination flood controller to accept the z value of the first vector as the first valid value and then the decimating by 4 separates out all the z values into a single buffer. In theory, the ability of the crosspoint switch to connect multiple sources and destinations into a single stream transfer should make it possible to create both the xy stream and the z stream in a single transfer. However, due to the different delays in each stream and the fact the z buffer is complete before the xy buffer, these separations are currently done as two separate transfers and the single transfer solution has not yet been tried. There is no loss in performance using the two-transfer method if the z-buffer is created first, and then the xy buffer is created in parallel with the interleaving of the z-buffer with the intensity values which is described next.



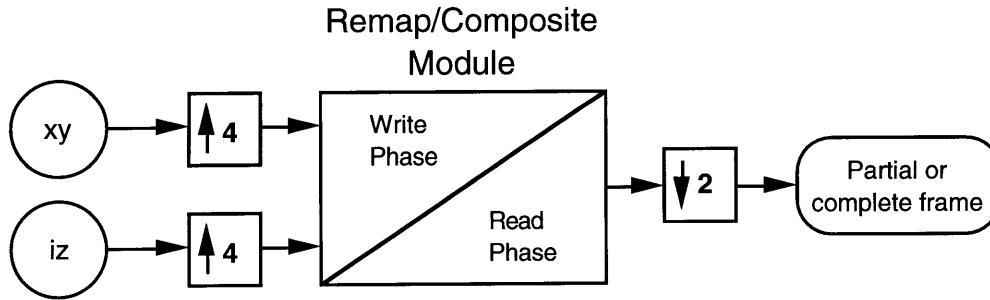
**Figure C-3:** Splitting the transformed 3D 4-vectors into an xy stream and a z stream.



**Figure C-4:** Interleaving the intensity and z values using the filter module in add mode.

While 2D and 2-1/2D objects do not require the creation of an xy stream, all object types have intensity and z values which must be interleaved in order to be composited in the hardware. 2-1/2D and 3D objects will already have complete z-buffers ready to be interleaved, but a small z-buffer containing a constant depth value needs to be created for 2D objects (the data stride passed to the source flood controller hardware can be modified to make a small buffer seem as large as necessary). The interleaving can be accomplished quite straightforwardly using the stream adder mode of the filter module as shown in Figure C-4 (up arrows signify zero-padding by the indicated factor). Each of the streams to be interleaved is zero-padded by a factor of two and then the z stream is delayed by one so that when the streams are added, interleaving is the net result. The stream delay is performed in the filter hardware itself. If the object to be composited is a color object, then there will be three color components and the same z data will have to be interleaved with





**Figure C-5:** Compositing objects using the remap/composite module in composite mode. The diagonal line indicates that the z-buffer is a two phase device. The xy stream will only be present when compositing 3D objects.

each of these three streams to form three intensity-z streams to be passed to the compositing hardware.

Preparing the data streams for the compositing hardware can therefore require as many as five stream operations on the output of the transformation unit. Once these operations are completed, however, compositing objects together is very straightforward.

### C.2.2 Compositing Objects

As shown in figure C-5, compositing objects using the hardware z-buffer is very simple. All that is required is to present the appropriate streams to the inputs of the composite module in the write phase and then read the results of the composition out of the module in the read phase. What makes compositing objects more complicated are the replication and decimation factors shown in the same figure which indicate that the remap/composite hardware is not capable of accepting or of presenting data at the full system clock rate. The slowness of the hardware leads to various attempts to minimize the number of transfers that must be made to or from the hardware per object being composited as well as per frame being assembled.

On an object by object basis the number of transfers into the hardware z-buffer is currently limited by restricting the maximum frame size to be 512x512 pixels. Since the internal memory on the composite sub-module is 1024x2048 samples divided into a

1024x1024 intensity buffer and a 1024x1024 z-buffer, this size limit allows all three components of a color frame to be stored in the hardware at one time so that in between color components the data does not need to be read out and the hardware reinitialized before the next component is composited.

In between objects there are two possible ways of thinking about the hardware z-buffer. The one that minimizes transfers to and from the hardware is to treat the hardware as a dedicated accumulation buffer so that data does not need to be read out after one object is composited and then written back in before a new object is composited. Using the hardware in this manner unfortunately violates the whole concept of the Cheops system multi-tasking in which several different processes might need to use the same resources. The NORMAN resource management system was created to minimize resource idle time and several different processes may be granted permission to use the same hardware in an interleaving fashion. Thus, there can be no guarantee that the data in the z-buffer will be the same when a process returns to it as when the process left it. The "correct" way to use the hardware is, therefore, to preload the z-buffer with any partial frame that may already have been assembled, composite the new object into that frame, and then read the results back out so they can be preloaded before the next object is composited. This is also true if alpha blending is to be implemented correctly. Currently, alpha is faked in 2D objects by creating a z-buffer that is set to the minimum possible z-value in regions where the object is supposed to be transparent so that those regions will not show. If the data is read out of the z-buffer, then it can be multiplied by the proper factor and new data added in. The ideal solution to alpha would of course be a sub-module which could perform alpha blending and compositing at the same time in hardware.

### **C.2.3 Hardware versus Software Compositing**

In the currently operational implementation of the compositing unit, all of the above discussion and acrobatics is bypassed by doing all the compositing of objects directly into

the output buffers using software routines custom written in assembly language. Using software has several advantages over hardware: none of the data stream rearrangement described in section C.2.1 needs to be performed; no multiple passes through the slow z-buffer hardware are necessary; and the data is already in the output buffer when composition is complete as opposed to having to read it back into local memory from the sub-module memory. Additionally, no temporary buffers are needed to store the intermediate streams of data or the partially assembled frames. Thus, while the hardware is much faster than software, the tremendous reduction in overhead that the software provides makes it a viable solution.

Empirically, it has been found that using the software compositing unit the system can composite a 200x400 actor into a 512x512 background at approximately six frames per second, while using the hardware with its associated overhead can only produce just under three frames per second of the same sequence. The performance of the hardware and software become about equal when compositing very large 3D objects (greater than 250,000 points) -- both hardware and software can render and display just over two frames per second -- but a 3D object consisting of only 3000 points can be rendered by the software at about 15 frames per second while the hardware is capable of only about 6 frames per second.

For these reasons, the structured video decoder currently operates with a software compositing unit by default. The hardware compositing unit described above has also been implemented, however, and a command line argument at system startup time can specify that the decoder should use the hardware instead of the software.

Undoubtedly the future will bring new and/or improved Cheops system hardware and changes will need to be made to the structured video decoder as it currently stands. It is hoped that this section has helped prepare the reader for what he or she will find when delving into the depths of the code which implement the system.