AIM Technical Note #53

UVLO
Motion-Video Encoding and Decoding

Kirk Rader, Philips IMS and Stephen Tickell, PRL June 5, 1990

UVLO is a motion-video encoding tool developed at Philips Research Laboratory in England. The UVLO technique for video data compression is derived from DYUV. UVLO takes advantage of the fact that the human eye is more sensitive to differences in brightness than to variations in color. UVLO reduces the amount of stored UV information by a factor of 2 over DYUV, thus greatly compressing the image data and increasing the screen area that can be covered by the image.

UVLO: Motion-Video Encoding and Decoding

OVERVIEW

In this document, we describe the UVLO motion-video encoding tool developed at Philips Research Laboratory in England. We provide a program example of the use of the encoded data at run-time. (See Appendix 1 for a sample program). Our intention is to introduce title engineers and other production personnel to the use of this tool and the constraints on the resulting UVLO motion-video sequences. We will publish a detailed user guide for this tool in the near future.

UVLO IMAGE COMPRESSION

UVLO is a technique for compressing video data derived from DYUV data. The effectiveness of UVLO is based on the fact that human visual perception is more sensitive to differences in brightness (Y) than to variations in color (UV). By separating the Y, U, and V channels of an image and reducing the amount of data stored for the U and V channels, we reduce the amount of data required for storage of one picture. We can achieve this without seriously sacrificing image quality.

When compared to full-resolution YUV, UVLO reduces the amount of UV information in the horizontal dimension by a factor of 4; it reduces the amount of UV in the vertical dimension by a factor of 2. It leaves the Y information unchanged. Note that normal DYUV compresses UV data by a factor of 2 only in the horizontal dimension only.

UVLO DECODING

UVLO supports motion video on a base case player. UVLO data representation allows the 68000 processor in the player to decode the image to standard DYUV format data in real-time.

UVLO ENCODING

The production tools used to capture and encode UVLO images for use in still pictures or motion video sequences are not capable of running in real-time. The encoding tool runs on a Sun workstation and uses the Androx image-processing board for both image capture and encoding to YUV. (The Androx board possesses 4 DSP chips that speed up the processing required for this type of encoding.) The UVLO encoder (which runs on the Sun workstation) takes the YUV data created by the Androx software and encodes it to UVLO.

The entire process, from video input to UVLO output, takes approximately forty times real time; that is, it takes approximately forty minutes to encode one

minute of video. The UVLO tool should not become obsolete with the introduction of specialized full-motion video hardware in the consumer CD-I player. UVLO images can also be used to support multiple planes of motion video or to significantly increase the number of still images that can be stored on a disc. (The hardware in the consumer player will be based on a completely different coding scheme.) UVLO is a truly base-case technique; the specialized hardware to support higher compression ratio full-motion video is an extension of the Green Book specification.

UVLO Production Pathway

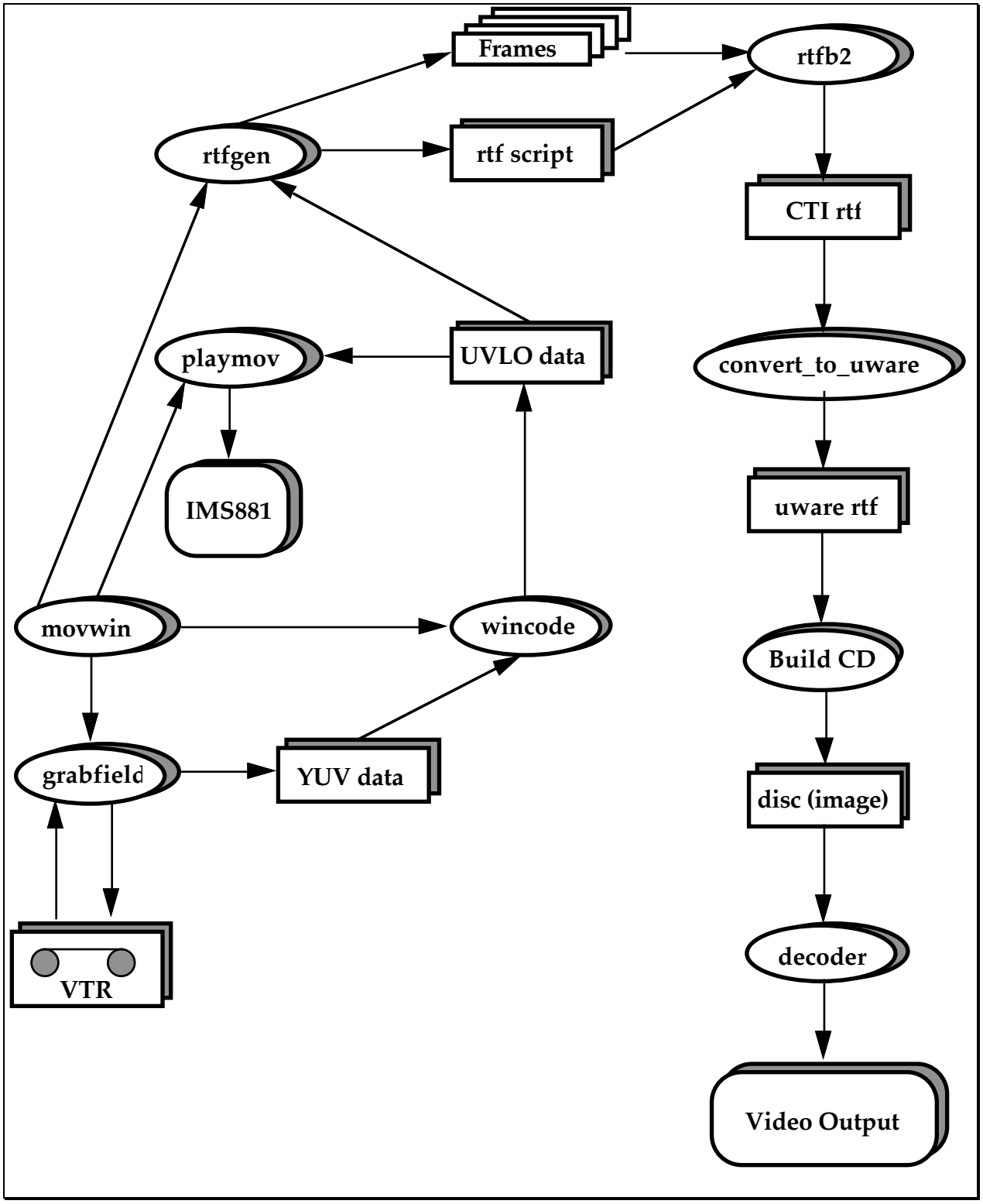
The production pathway for UVLO-based images consists of:

- Capturing images from Betacam videotape
- Encoding images to UVLO representation
- Including run-time decoding functions in the title (application) code

Capturing images from videotape and encoding those images into UVLO representation are completed with two different parts of a Sun/Androx tool. Your run-time library must include facilities for decoding in real time to support motion video applications. The data flow diagram on the following page shows how UVLO data is captured, encoded, and then displayed on the player.

Note that the UVLO tool supports only frame rates of 15 or 30 frames per second (FPS). It also provides precompression data reduction parameter that allows you to scale the source image used for motion video. However, some image fidelity is lost through the precompression process. Precompression is implemented as a subsampling filter that simply lowers the resolution and, therefore, the bandwidth requirements of the grabbed YUV image.

The exact calculation of the amount of screen area that can be covered by motion video using the UVLO real-time files produced by this tool is complex. (See Appendix 2 for some measurements of screen utilization.) It depends on the frame rate and audio level. In addition, the precompression factor creates an additional constraint on the percentage of the source image that can be scaled to



UVLO Data Flow Diagram

fit in the UVLO window. In practice, it is probably necessary to determine these parameters by trial and error—especially since the resulting perceptual image fidelity is partly determined by the nature of the particular image grabbed. Since the optical disc's bandwidth (75 sectors per second) must allow the delivery of 2 frames per 5 sectors to support 30 FPS (5 sectors represents 1/15 of a second), the maximum practical size of a frame can be only 2 sectors (4648 bytes). This is not a large percentage of the screen (120^2 pixels, or about 16% of the screen). To increase the screen percentage, you must reduce the frame rate. At 15 FPS, it is possible to double the area of the screen covered by motion video; then, the maximum frame size is 4 sectors (assuming that you leave some free sectors for audio or other data).

Androx Bug

The UVLO tool depends on the Androx image processing board. The Androx board has a hardware bug in its video circuitry. All signals that pass through the board, whether for capture or display, have a forced "pedestal." That is, the black levels are raised several units above normal. Furthermore, any signal that drops below nominal black level is clipped. This can result in washed out, color distorted images. The extent to which the image is noticeably affected is dependent on the nature of the image. Androx has been informed of the problem but has announced no plan to correct it.

USING THE ENCODER

The interface to the UVLO encoder package is a program called **movwin** that runs on the Sun workstation. When you invoke **movwin**, it opens a blank window and waits for you to specify the video source data. To do so, you use the window frame's menu operated by the right button on your mouse. A dialogue box with the option **Source select** appears to prompt you for the name of the source material. At this point, you can select material that has already been grabbed or which needs to be grabbed from the video tape recorder.

If you specify that your source is to be grabbed, **movwin** prompts you for the starting and ending time codes of the sequence to be encoded. The SMPTE time codes are entered in hours, minutes, seconds, and frames. You must make sure that the VTR is connected, loaded with the desired tape, and ready for remote control. Then, the program **grabfield** automatically searches for the specified starting frame (using the time code) and scans through the tape grabbing each frame without your intervention. In addition to grabbing the RGB image, the Androx board also converts the data to YUV format. Thus, the data file produced by **grabfield** has been preprocessed for input to the UVLO encoder.

The grabbing and preprocessing of data does not occur in real time. However, it is I/O bound by the VTR transport and the SUN disk rather than by the processor. Therefore, it is reasonably fast.

The UVLO encoder, **wincode**, runs entirely on the Sun. After the data is processed by **grabfield**, **wincode** converts each frame of the data file into UVLO.

The UVLO data contains image size and position information for each frame. Once the source is selected and has been grabbed, the previously blank **movwin** window on the Sun contains a black and white, low-resolution representation of the image of the first frame of the YUV data. To scan the images, you simply use the left and right mouse buttons to step through the sequence.

You can also define a specific area of the image for conversion to UVLO by defining a rectangular window using the shift key and the middle mouse button. The maximum size of the source window is controlled using the **Coding** option in the **movwin** frame menu. The **Coding** option pops up a dialog box that allows you to specify the coding method, precompression (an extra "squeeze" factor that allows larger areas of the screen to be encoded), frame rate, and audio level. After you set the coding parameters and define the image area for each frame, you invoke the **wincode** program by returning to the **Coding** dialog box. In the coding dialog box, you specify that a real-time file be produced and the name of an audio file (which must be of the audio level specified earlier) to interleave with the UVLO data. This creates a shell script that is automatically executed when you select the **Start coder button**.

When the process is complete, you will have the following new files:

- The grabbed YUV data file
- The UVLO data file (the same name as the YUV data file, but followed by a **.uvl** extension
- A shell script that was run to generate these files
- A subdirectory with the same name and the YUV data file, but followed by an **_rtf** extension

The **_rtf** subdirectory has one file per frame of the UVLO encoded sequence, a real-time file builder script, a CTI format real-time file, and a Microware format real-time file.

Encoder Problems and Bugs

There are a number of problems and bugs in the encoder production pathway. First, the YUV output of the **grabfield** program is enormous—over 100k per frame. Thus, you must have either sufficient free space for hundreds or thousands of megabytes (for a realistically long video sequence) of intermediate files, or you must pipe the output of the grabber directly to **wincode**, the encoder. This results in direct creation of the **.uvl** file without storage of a huge YUV file on disk. The problem with this technique is that you must be certain in advance of the correct arguments for **wincode**.

To accomplish this in the most practical way, use **movwin** as usual to grab and generate a script to use a specified pipe for I/O and to use the actual ending time code. Deciding later that you would like to change coding parameters, such as audio level, precompression ratio, or window size requires regrabbing; otherwise, you will not be able to modify the YUV file. In addition, if a precompression scaling factor is specified, the encoder program, **wincode**, uses more processor time and lags behind the grabber program, **grabfield**.

There is another problem during the real-time file generation phase of this process. There are bugs in **movwin** and in **rtfgen** that create errors in the scripts generated. The last step in the encoder script is to invoke **rtfgen**. This program then creates individual UVLO frame files and a real-time file builder script. It also invokes **rtfb2** to generate the CTI format real-time file. (It also invokes **convert_to_uware** to generate a Microware format real-time file, but we are not concerned with that format here.)

For some combinations of coding parameters, the scripts generated by **movwin** and **rtfgen** are incorrect. This causes the controlling script to hang and/or **rftb2** to fail with a channel conflict error between the UVLO data and the audio data. If you experience one of these problems, you must either correct the problem yourself and re-invoke **rtfb2** or **convert_to_uware** or create a script (such as a DBL script) for another real-time file builder.

USING THE ENCODED DATA

On a Sun workstation with a "green" video display board (that is, the IMS881 board), you can play back the encoded **.uvl** file without leaving **movwin**. This allows you to check the quality of the encoded image and to verify that the sequence matches the title's requirements.

After you have built the disc image from the real-time file created by the encoder, you can use a UVLO decoder program on the player to display it. The remainder of this document is a sample program that demonstrates how to decode UVLO at run time and how to display the resulting DYUV image. The sample program accepts the name of a real-time file and an optional integer number of sectors per UVLO frame (the default is 2). Then it plays a "movie" of the decoded UVLO data. The functions **uvl_decode** and **uvl_lutgen** are defined

in a file **uvl_decode.r** provided to AIM by PRL. The primary feature of this program is to demonstrate that you must first prepare for DYUV output by opening the video path and creating and initializing the appropriate DYUV draw maps. You must also allocate the UVLO look-up table (with **uvl_lutgen**). Then, you call **uvl_decode** and pass it the following:

- Buffer into which the UVLO data was read
- Line-start array from the draw map
- Look-up table
- Pixel coordinates on the screen where the image is to be displayed

After **uvl_decode** returns, the draw map is filled with the decoded DYUV data

APPENDIX 1: SAMPLE PROGRAM

```
/*****************************************************************************
*****
*
* Filename:  uvlomove.c
* Project:   UVLO decoding prototype
* Purpose:   Decode UVLO real-time files
*
*
* Author:    Kirk Rader
* Date:      March 10, 1990
* Revisions:
* Tests:
* Dependencies:
* Notes:
*
* Copyright 1990 Philips Interactive Media Systems, all rights reserved.
*
*****************************************************************************
****/

#include <stdio.h>
#include <modes.h>
#include <cdfm.h>
#include <ucm.h>
#include <stddef.h>
#include <csd.h>
#include <motion.h>

extern int errno;

#define MAX(a,b) (((a) > (b)) ? (a) : (b))

#define FCTA (sesptr->fctid0)             /* Plane A's FCT. */
#define FCTB (sesptr->fctid1)             /* Plane B's FCT. */
#define LCTA (sesptr->lctid0)             /* Plane A's LCT. */
#define LCTB (sesptr->lctid1)             /* Plane B's LCT. */

#define PCBERR (1<<15)                    /* Bit in PCB_Stat field which indicates
                                             error in play. */

#define PCLERR (1<<7)                     /* Bit in PCL_Ctrl feild which indicates
error. */

#define VIDEOCHAN 0                       /* Channel number of UVLO data. */
#define VIDEOCHANF (1<<VIDEOCHAN)         /* Channel mask of UVLO data. */

#define AUDIOCHAN 0                       /* Channel number of audio data. */
#define AUDIOCHANF (1<<AUDIOCHAN)         /* Channel mask of audio data. */

#define SIGBUFULL 1000                    /* Buffer full signal. */
#define SIGEOF 1001                       /* End of play signal. */

#define NUM_PCL 4                         /* Number of PCL's in
                                             circular list.  */
#define NUM_DM 2                          /* Number of drawmaps. */

static int x_off = 50;                    /* X offset for uvl_decode. */
static int y_off = 10;                    /* Y offset for uvl_decode. */
```

```
static int width = 0;                    /* Min width for matte. */
static int height = 0;                   /* Min height for matte. */

static int sect_per_frame = 2;           /* Sectors per frame of UVLO data. */

static int os_version;                   /* Version of CDRTOS. */

static int bufull = 0;                   /* Buffer full flag (set by signal handler
                                            on buffer full signal). */

static int matte_set = 0;                /* Flag which indicates need to set matte
                                            LCT instructions. */

static int quit = 0;                     /* Quit flag (set by signal handler on end
of
                                            play). */

static int framecnt = 0;                 /* Count of decoded frames. */

static int missedcnt = 0;                /* Count of missed frames. */

static int curdm = 0;                    /* Index of current drawmap. */

static unsigned short lut[2048];         /* UVLO lookup table */

static PCL pcl_video[NUM_PCL];           /* Array of PCL's from which to build
circular list
                                            [see init_pcl(), below]. */

static int queue_empty = 1;              /* PCL queue-empty flag. */
static PCL* read_pcl = &(pcl_video[0]);  /* Current PCL into which data was read. */
static PCL* decode_pcl = &(pcl_video[0]); /* Current PCL out of which to decode. */

static PCL* cil_video[32] = {
  &(pcl_video[0]), (PCL*)NULL, (PCL*)NULL, (PCL*)NULL,
  (PCL*)NULL, (PCL*)NULL, (PCL*)NULL, (PCL*)NULL,
  (PCL*)NULL, (PCL*)NULL, (PCL*)NULL, (PCL*)NULL,
  (PCL*)NULL, (PCL*)NULL, (PCL*)NULL, (PCL*)NULL,
  (PCL*)NULL, (PCL*)NULL, (PCL*)NULL, (PCL*)NULL,
  (PCL*)NULL, (PCL*)NULL, (PCL*)NULL, (PCL*)NULL,
  (PCL*)NULL, (PCL*)NULL, (PCL*)NULL, (PCL*)NULL,
  (PCL*)NULL, (PCL*)NULL, (PCL*)NULL, (PCL*)NULL
};

static PCB pcb = {
 0, SIGEOF, 1, VIDEOCHANF | AUDIOCHANF, AUDIOCHANF,
 cil_video, (PCL*)NULL, (PCL*)NULL
};

static DrawmapDesc *dm[NUM_DM] = { NULL, NULL }; /* Drawmap array. */
static DrawmapDesc *bgdm = NULL;         /* Background drawmap. */
static UCMSES *sesptr = NULL;            /* pointer to screen */
static int vpath = -1;                   /* video path */
static int fd = -1;                      /* Real-time file descriptor. */

static int die(string)
char* string;
{
  int i;
  if (string) _errmsg(errno, "%s\n", string);
  (void)printf("%d Frames decoded, %d missed.\n", framecnt, missedcnt);
  for (i = 0; i < NUM_DM; ++i)
       if (dm[i]) dm_close( vpath, dm[i]);
  if (bgdm) dm_close(vpath, bgdm);
  if (vpath != -1) close_vid( vpath, sesptr);
```

```
  if (fd != -1) {
       ss_abort(fd);
       close(fd);
  }
  exit(0);
}

static int init_pcl(num_pcl, pcl)        /* Init num_pcl PCL's as a circular list. */
int num_pcl;
PCL* pcl;
{
  register int i;
  register int dsize;
  register int error;
  dsize = sect_per_frame * DSECT;
  error = 0;
  for (i = 0; i < num_pcl; ++i) {
       pcl[i].PCL_Sig = SIGBUFULL;
       pcl[i].PCL_Nxt = &(pcl[i + 1]);
       pcl[i].PCL_Buf = malloc(dsize);
       if (pcl[i].PCL_Buf == NULL) {
         error = -1;
         break;
       }
       pcl[i].PCL_BufSz = sect_per_frame;
       pcl[i].PCL_Err = NULL;
       pcl[i].PCL_Ctrl = 0;
       pcl[i].PCL_Cnt = 0;
  }
  pcl[num_pcl - 1].PCL_Nxt = &(pcl[0]);
  return error;
}

static void set_matte(left, top, right, bottom)
int left, top, right, bottom;
{
  (void)printf("matte: %d, %d, %d, %d\n",
              left,
              top,
              right,
              bottom);
  if (dc_wrli(vpath,
             LCTB,
             top,
             5,
             cp_matte(0, MO_SET, MF_MF0, 0, left)))
    die("matte left");
  if (dc_wrli(vpath,
             LCTB,
             top,
             6,
             cp_matte(1, MO_RES, MF_MF0, 0, right)))
    die("matte right");
  if (dc_wrli(vpath,
             LCTB,
             bottom,
             5,
             cp_matte(0, MO_END, MF_MF0, 0, left)))
    die("end matte");
}

static int sighandler(sig)
int sig;
{
  static int curpcl = 0;
```

```
  switch (sig) {
  case SIGEOF:
    quit = 1;
    break;
  case SIGBUFULL:
    if (pcl_video[curpcl].PCL_Ctrl & PCLERR)
      die("data error");
    /* Re-init PCL. */
    pcl_video[curpcl].PCL_Cnt = 0;
    pcl_video[curpcl].PCL_Ctrl = 0;
    pcl_video[curpcl].PCL_BufSz = sect_per_frame;
    /* Advance to next PCL. */
    read_pcl = &(pcl_video[curpcl]);
    if (++curpcl >= NUM_PCL) curpcl = 0;
    if (!queue_empty && (read_pcl == decode_pcl)) {
      decode_pcl = read_pcl;
      missedcnt += 1;
    }
    else {
      bufull = 1;
      queue_empty = 0;
    }
    break;
  default:
    die("sighandler");
    break;
  }
}

static void handle_bufull()
{
  /* Reset buffer full flag. */
  bufull = 0;
  /* Create the UVLO window matte, if necessary. */
  if (!matte_set) {
    struct frame_hd *hd = (struct frame_hd *)decode_pcl->PCL_Buf;
    int left, top, right, bottom;
    left = x_off * 2;
    top = y_off * 2;
    right = left + MAX(width, hd->xmax);
    bottom = top + MAX(height, hd->ymax);
    set_matte(left, top, right, bottom);
    matte_set = 1;
  }
  /* Decode and display frame, if able. */
  while (!queue_empty && (decode_pcl != read_pcl)) {
    uvl_decode(decode_pcl->PCL_Buf,
               dm[curdm]->dm_lnatbl1,
               lut,
               x_off,
               y_off);
    if (dc_wrli(vpath,
              LCTA,
              0,
              0,
              cp_dadr( (int)dm[curdm]->dm_map1)))
      die("updating LCT");
    /* Advance to next drawmap. */
    if (++curdm >= NUM_DM) curdm = 0;
    framecnt += 1;
    if ((decode_pcl = decode_pcl->PCL_Nxt) == read_pcl) queue_empty = 1;
  }
}

/*********************************************************************
```

```
*
* Function:         main()
* Purpose:          Initializes the video display and shows the slideshow.
* Globals:          extern int errno
*               static unsigned short lut[]
*
*************************************************************************/

main( argc, argv)
       int argc;
       char *argv[];
{
       char* filename = "", *progname;
       int n;

       progname = argv[0];

       (void)printf("UVLO move\n");

       while (--argc > 0) {
         char* arg;
         arg = *++argv;
         if (*arg == '-') {
           switch (*++arg) {
           case 'x':
             argc -= 1;
             arg = *++argv;
             sscanf(arg, "%d", &x_off);
             break;
           case 'y':
             argc -= 1;
             arg = *++argv;
             sscanf(arg, "%d", &y_off);
             break;
           case 'w':
             argc -= 1;
             arg = *++argv;
             sscanf(arg, "%d", &width);
             break;
           case 'h':
             argc -= 1;
             arg = *++argv;
             sscanf(arg, "%d", &height);
             break;
           case 'n':
             argc -= 1;
             arg = *++argv;
             sscanf(arg, "%d", &sect_per_frame);
             break;
           default:
             die("usage: 'uvlomove' ['-n' bufsize] filename");
             break;
           }
         }
         else
           filename = arg;
       }

       (void)printf("file='%s', buffer=%d\n",
                  filename,
                  n);

       uvl_lutgen(lut);

       get_os_version();
```

```
if (init_pcl(NUM_PCL, pcl_video)) die("init_pcl");

intercept(sighandler);

/* Open the vpath and the special effects screen pointer */
if (init_vid( &vpath, &sesptr, STD_DM_HEIGHT) == SYSERR)
  die("init_vid");

setCompMode( vpath);

/* Create DYUV drawmap in plane A */
for (n = 0; n < NUM_DM; ++n)
  if ((dm[n] = dm_create( vpath, PA, D_DYUV, 768, 480, 92960, 0))
      == NULL)
    die("dm_create");

/* Create background drawmap. */
if ((bgdm = dm_create( vpath, PB, D_DYUV, 768, 480, 92960, 0))
    == NULL)
  die("dm_create");

/* Setup the generic display control program */
if ( setup_dcp( vpath, sesptr, ICM_DYUV, ICM_DYUV) == SYSERR)
  die("setup_dcp");

/* Set the DYUV start values to the standard Thomson values */
if (dc_wrfi(vpath,
          FCTA,
          FCT_YUV,
          cp_yuv( PA, 16, 128, 128)))
  die("DYUV start values (plane A)");
if (dc_wrfi(vpath,
          FCTB,
          FCT_YUV,
          cp_yuv( PB, 16, 128, 128)))
  die("DYUV start values (plane B)");

/* Set plane order. */
if (dc_wrfi(vpath, FCTA, FCT_PO, cp_po(PR_BA)))
  die("plane order");

/* Set transparency control information. */
if (dc_wrfi(vpath,
          FCTA,
          FCT_TCI,
          cp_tci(MIX_OFF, TR_OFF, TR_MAT0_T)))
  die("transparency control");

/* Link drawmaps to LCT's. */
if (dc_wrli(vpath,
          LCTA,
          0,
          0,
          cp_dadr( (int)dm[NUM_DM - 1]->dm_map1)))
  die("linking plane A drawmap");

if (dc_wrli(vpath,
          LCTB,
          0,
          0,
          cp_dadr( (int)bgdm->dm_map1)))
  die("linking plane B drawmap");

/* Execute DCP's. */
```

```
        if (dc_exec(vpath, FCTA, FCTB))
          die("dc_exec");

        /* Open real-time file. */
        if ((fd = open(filename, S_IREAD)) == -1) die("open");

        /* Begin play of real-time data. */
        if (ss_play(fd, &pcb)) die("ss_play");

        /* Loop waiting for buffer-full or end-of-play signals. */
        while (!quit) {
          while (bufull)
            handle_bufull();
          pause();
        }

        /* Check for error condition on end-of-play. */
        if (pcb.PCB_Stat & PCBERR)
          die("play err");

        /* Exit normally. */
        die(NULL);

}



/************************************************************************
* setCompMode()
* Purpose:   Set the compatability mode on the CD-I system assuming
*                         that the images used are 384 actual bytes wide
* Passed:    the video device number
* Output:    The compatability mode will be set according to the info
*                         found in the CSD
* Returned:  OK or SYSERR
************************************************************************/
INT setCompMode( vpath)

        int           vpath;

{
        CHAR *disp_dev, *disp_param, *csd_devname(), *csd_devparam();
        REG CHAR *dp;
        int device;
        int    mode;

        device = (os_version == 1) ? DT_VIDEO : DT_DISPLY;

        if ( (disp_dev = csd_devname( device, 1)) == NULL )
        {
                fprintf( stderr,
                                "Can't get display devname, errno = 0x%x\n",
                                errno );
                return ( SYSERR );
        }

        if ( (disp_param = csd_devparam( disp_dev )) == NULL )
        {
                fprintf( stderr,
                                "Can't get display parameter, errno = 0x%x\n",
                                errno );
                return ( SYSERR );
        }
```

```
	mode = 1;                                   /* Mode 1 is for 384 width on Monitor
*/
	dp = disp_param;
	while ( *dp != '\0' )
	{                                           /* If "TV" is in the parameter, its a
TV */
		if ( *dp == 'T' )                   /* otherwise it's a Monitor */
			if ( *(dp + 1) == 'V' )
			{
				mode = 0;           /* TVs need mode 0 for 384 wide pix */
				break;
			}

		dp++;
	}

	free( disp_dev );                           /* Give back memory for csd stuff */
	free( disp_param );
                                                    /* Set appropriate compatability mode
*/
	if ( dc_setcmp( vpath, mode) == SYSERR )
		return ( error( errno, "Can't reset compatibility mode" ) );

	return ( OK );
}


/*********************************************************************
*
* Function:		get_os_version()
* Purpose:		Determine the version of CD-RTOS that we are running under.
* Passed:
* Globals:		os_version
* Assumes:
* Outputs:
* Returns:
* Author:		Ken Ellinwood, August 25, 1989
*
*********************************************************************/
get_os_version()
{

	char	*dev_name;
	char	*dev_params;
	char	*tmpptr;
	int		param_len;
	char	*rindex();

	/* If anything goes wrong, os_version will be 99 */
	os_version = 99;

	/* Check for device #0, returns NULL under 0.99 */
	if ((dev_name = csd_devname( 0, 1)) != NULL)
	{
		/* Get device parameters for #0 */
		if ((dev_params = csd_devparam( dev_name)) == NULL)
		{
			printf("Can't get csd params for %s\n", dev_name);
			free( dev_name);
			exit( 0);
		}

		/* Search for the string "1.0" within the device parameters */
		param_len = strlen( dev_params);
```

```
                tmpptr = dev_params;

                while ( tmpptr != (dev_params + param_len - 2))
                {
                        if (strncmp( tmpptr++, "1.0", 3) == 0)
                        {

                                /* When "1.0" is found, set os_version to one */
                                os_version = 1;
                                break;
                        }
                }

                free( dev_name);
                free( dev_params);
        }
}
```

APPENDIX 2: BASE CASE FULL-MOTION VIDEO SCREEN UTLIZATION

In the AIM evaluation laboratory, we attempted to calculate the maximum values for screen utilization with base case full-motion video using DYUV, UVLO, and interpolated data. Because these values are based on theory and not on actual screen measurements under a variety of conditions, they should be used as "rule of thumb" values. Your actual screen utilization will probably approach these values, but will never reach the maximum values stated. Variables, such as the one-sector granularity of real-time files and the mixing of audio into your program, create constraints on screen usage. However, we feel that the values resulting from our calculations will be very useful to you in planning screen utilization.

The following charts show the percentage of the screen that can be covered by full-motion video given various coding techniques, frame rates, and bandwidth utilization. Each chart shows the frame rates on the X axis, bandwidth threshold values (corresponding to the labeled audio levels) on the Y axis, and the percentage of the screen that can be covered with full motion video using those parameters.

DYUV Images

The first chart contains data for normal DYUV images. In the Green Book, this is the standard image-encoding technique for naturalistic images. It gives the best image quality available for full-motion video on base-case players. It compresses the data by throwing out half of the color (UV) information, while retaining all of the brightness (Y) information, and then it delta codes the result. Since the human eye is relatively insensitive to color compared to brightness, the resulting images are perceptually comparable to the quality of RGB images, but with a substantial reduction in data size.

The second chart contains data for UVLO images. UVLO is a technique related to DYUV, except that it compresses data further by throwing away even more of the color information in the horizontal dimension. The disadvantage of UVLO is the player's lack of real-time decoding hardware; UVLO is not defined in the Green Book. Thus, the 680x0 processor must be employed to decode UVLO data into DYUV at run time. Because this is not a trivial operation, the processor throughput is almost entirely consumed by UVLO decoding.

The third chart contains data for UVLO combined with a line-interpolation technique that doubles screen coverage vertically by halving vertical resolution. At its simplest, such a technique duplicates every line. If there is sufficient processor throughput after UVLO is decoded, some filtering might be applied to smooth out the interpolated lines.

Bandwidth Utilization

		50% 8/16 A Stereo	75% 12/16 A Mono B Stereo	88% 14/16 B Mono C Stereo	94% 15/16 C Mono	100% 16/16 No Audio
	30	3	5	6	6	6
	25	4	6	7	7	8
	24	4	6	7	7	8
	15	6	9	11	12	13
Frames per	12.5	8	11	13	14	15
Second	12	8	12	14	15	16
	10	9	14	17	18	19
	8	12	18	21	22	24
	6	16	24	28	30	32

DYUV SCREEN UTILIZATION
(in percent)

Bandwidth Utilization

		50% 8/16 A Stereo	75% 12/16 A Mono B Stereo	88% 14/16 B Mono C Stereo	94% 15/16 C Mono	100% 16/16 No Audio
	30	5	7	8	9	10
	25	6	9	10	11	11
	24	6	9	10	11	12
	15	10	14	17	18	19
Frames per	12.5	11	17	20	21	23
Second	12	12	18	21	22	24
	10	14	21	25	27	29
	8	18	27	31	34	36
	6	24	36	42	45	48

UVLO SCREEN UTILIZATION
(in percent)

Bandwidth Utilization

		50% 8/16 A Stereo	**75% 12/16 A Mono B Stereo**	**88% 14/16 B Mono C Stereo**	**94% 15/16 C Mono**	**100% 16/16 No Audio**
	30	10	14	17	18	19
	25	11	17	20	21	23
	24	12	18	21	22	24
	15	19	29	33	36	38
Frames per	12.5	23	34	40	43	46
Second	12	24	36	42	45	48
	10	29	43	50	54	57
	8	36	54	63	67	72
	6	48	72	84	90	96

DYUV SCREEN UTILIZATION
(in percent)