archive copy

Picture and Sound Editing on Optical Media with a Graphical User Interface: Progress Report for MacEdit

Submitted in partial fulfillment of the requirements for the S.B. degree in Computer Science

by

J. Michael Tindell

August 10, 1986

Signature of Author:
J. Michael Tindell on Aug. 10, 1986

Thesis Advisor: Richard Leacock
[illegible] on Sep 8 1986

Thesis Supervisor: Glorianna Davenport
[illegible] on Sept 8 1986

Introduction

The optical disk medium has made random access to audio-visual information a practical reality. There are no low-cost systems that allow users to impose an abstract structure upon the material stored on a disk or disks. The organization might be a traditional edited program, a hierarchical database, or something else. The advantage to a software approach is the ability to impose an arbitrary number of structures upon the same optically stored information.

Background

In the information tradition, user interfaces for the artist were a natural outgrowth of the technical and physical characteristics of the media. This is true for all media, the inks and paper, paint and canvas, and the modern picture and sound media, film and videotape.

Film has always been a very physical medium and so have the techniques for editing it. In a very real sense film editors "sculpt" raw footage into a finished product in an evolutionary, non-linear fashion. The celuloid gives physical reality to the notions of space and time and the editor has total control over the flow. The techniques of film editing are much the same today as those used in the infancy of filmmaking. The picture and sound tracks are generally edited on separate lengths of celuloid. These tracks are played back in synchronization on some sort of mechanical viewing system; two of the most widely used today are the flatbed editor and the moviola. The editor watches and listens to the sequence he is editing and periodically stops the viewer to make a change. This is accomplished by physically cutting the celuloid and taping the cut ends. In this manner a given shot may be lengthened, shortened, or substituted. The process is often tedious and time-consuming; however, editors become quite adroit at the mechanics of timing and splicing edit points. Most important to the film editor is his confidence that, given time, he can achieve a desired effect with his medium.

Magnetic videotape was first introduced into television broadcasting as a means to record programs for later rebroadcast. These programs were either live video or film that was scanned into video. Soon engineers began to do some rudimentary editing by splicing the videotape. At first this was accomplished by painting various ferromagnetic solutions on the tape in order to make the scanning lines and control track pulses visible. Later they devised electronic means of locating these splice points and eventually built circuitry that would identify the edit point and insert a special pusle on the control track at that point (the control track is a sequence of 60 hertz pulses recorded on part of the tape that are used by feedback circuitry to maintain the correct tape speed). Eventually editing began to be done by building a master tape by succesively recording segments. This was done by rolling the machines in sync and then turning on the record heads of the master deck at the desired edit point. This was the genesis of electronic videotape editing. Eventually computer control was integrated into the process (companies and systems) and the process was made precise to the frame and repeatable to allow rehearsals of edits.

Modern videotape editing is limited by the fact that a program or sequence is built linearly by recording on to a master tape. Changing the duration of one previously recorded shot requires altering the duration of another, or may require the rerecording of every subsequent shot if the change cannot be compensated for elsewhere without creative sacrifice. This has shaped the way that people have used tape; it has also, to many observers, limited the creative options in tape editorial.

The optical disk is a new animal: virtually instant random access to audiovisual material. There is, as of yet, no system that takes full advantage of the new medium in a way that is not a paradigm of a traditional (ie. video or film) editing methodology. These editing methodologies are not necessarily the "best" ways of organizing moving pictures and sound. They were simply *ad hoc* responses to the available technology; perhaps we can do better with the optical disk.

Some novel editing systems

Until very recently there were not any concerted efforts toward applying state-of-the-art computer technology to the editorial process. Computerized videotape editing systems used (and still use, for the most part) hardware and software that is antiquated by today's standards; however, this situation is changing. In the last couple of years there have been two interesting developments in this area.

About six years ago George Lucas got together with several other filmmakers to discuss how computer technology could be used to enhance and speed up the production and post-production processes. He came away with a strong commitment to developing new editing systems utilizing the latest hardware. The fruit of this commitment is the EditDroid (manufactured by The Droid Works, San Rafael CA). The EditDroid is based on a Sun Microsystems workstation, complete with an advanced bit-mapped graphic display. The high-quality graphics allow for the development of a highly visual and interactive user interface. The system, as currently conceived, is essentially an electronic flatbed film editing table. The EditDroid was designed with videodisk media in mind and works best with them.

More recently, Montage Computer Corporation developed and introduced the Montage Picture Processor. This system attempts to make video editing more of a visual than numeric process. The in and out points of various shots are identified not by timecode numbers but by digitized video frames. These head and tail frames are displayed together for an effect somewhat akin to looking at a collection of film clips. This system is unique in its approach to random access media. Rather than use videodisks, which were viewed as impractical for the near term, the system uses a stack of fourteen Beta Hi-Fi VCRs to record source material and then is able to assemble a real-time preview of a finished program by sequencing the playback of the Beta decks.

Both of these systems were widely acclaimed as "breakthroughs", yet neither has truly radicalized the editorial process. They are important in the sense that they have broken the ice and left open the way for further development.

Design goals for this system

The scope of design will be limited here to discussion of a system that will be an alternative to current editorial processes in the realm of post-production. The ideas here might easily be expanded to the emergent technologies of CD-ROM and CD-Interactive. The design will emerge as an authoring system that could be generalized for any information stored in optical form.

The best designs are ones that follow the flow of the particular task at hand. The mission for a videodisk editing system is to provide an easily referenced organizational framework that can incorporate the material on to be edited. Once this is accomp[lished the editing process may begin. In this stage the editor will want to be able to arrange shots into sequences, play them back, and make changes rapidly. This will require the abilty to shuttle the videodisk players so that frames may be identified and easily referenced.

One of the simplest database organizations is one that has a hierarchical structure. In this system the organizational unit is called a **segment.** A segment contains either additional segments or a representation of an interval on the disk. Since the system has no restrictions on the physical content of the disk, it is up to the user to define these segments by playing the disk and identifying beginning and end points for each of his segments. The system needs controls on the screen for shuttling the disk and marking the beginning and ending points of a segment. There should also be controls for playing back a segment or segments and adjusting the in and out points of each.

Once the user has defined a segment he needs some identifier for it. He may identify each segment with a textual or visual mnemonic so that he can reference them without having to play a segment from the disk. These mnemonics would be displayed as icons on the Macintosh screen. In this way the user can build up a database of the information on the disk.

After the user has organized some material he will want to do something with it. This will be accomplished by icon-dragging and the other natural Macintosh screen functions. For the purposes of discussion let us assume that he wants to edit the material into a program. The material may be organized in the desired order and played back from beginning to end. The windows that contain segments will all have the some characteristics. In this application, the user may chose to organize his segments according to location, scene, shot, etc. Then he would open an empty segment window and copy the desired segments for a particular sequence into it and edit there, leaving the original organization intact for future reference.

Software implementation

The system software is written in the C language, using the Mac-C development environment (copyright Consulair Corporation, 1983, 1984) . The system itself is broken down into modules that correspond to a particular functional area: for example, all of the procedures that draw, move, and do i/o with the Macintosh windows are contained in the **window** module.

The main control loop for the system is in the **macedit** module. It initializes the system data areas and dispatches events. The basic system data types are **segment** s and **macedit_window** s. The former is the basic unit of the hierarchical database while the latter is used to keep track of information about windows that the Macintosh window data type does not encompass. The segment objects are stored in doubly-linked lists. The links in the list are full longword pointers; this is not the most efficent use of available memory resources and would have to be changed to a scheme using array indicies for a finished implementation.

Videodisk control

The control of videodisk players is relatively straightforward. The individual frames of information may be treated as addresses on the disk and a player may be instructed to search for a frame, play for a given number of frames, then stop or repeat as desired. The control information is exchanged over an RS232 serial communication line to a Macintosh.

The mechanics of videodisk player control are well defined. The more difficult problem is the design of a generalized user interface to pictures and sound that happen to be on videodisk, one that is amiable to disparate user groups.

Results

The progress so far has been limited by a thorough understanding of Macintosh graph primitives. I have written a skeleton program that does basic interfacing to the Macintosh operating system (ie. windows and menus), as well as the basic data types and their operators to be used in the application (ie. segments). The problem has been in implementing the icon notion with respect to segments: saving them, drawing them, and dragging them around the screen. This is a problem that will hopefully be solved soon; I have writted code to do this that *should* work, but still must be debugged. Once this is resolved I should be able to get a simple system up fairly soon. With this accomplished, I will add more features and sophistication.

Conclusions and directions

The future of picture and sound editorial clearly lies in the direction of random and, as nearly as possible, instantaneous access to the material at hand. I have attempted to exp;lore new possiblities for the human interface to these media, using contemporary, widely-available, low-cost computer technology. While my success was limited by a combination of technical and time limitations, it is my conviction that the next generation of media editors will be working in an environment similar to the one I have described.

The problems of accessing large databases of picture and sound information will be compounded with the advent of multimedia storage systems such as Interactive Compact Disk. With this vartiety of elements at his disposal, the editor (or, perhaps as appropriately, the author) will require an interface that transcends the traditional film and videotape paradigms. The challenge will be to allow access to and organization of these elements in spatial and temporal dimensions. These are the tasks at hand for media system designers.

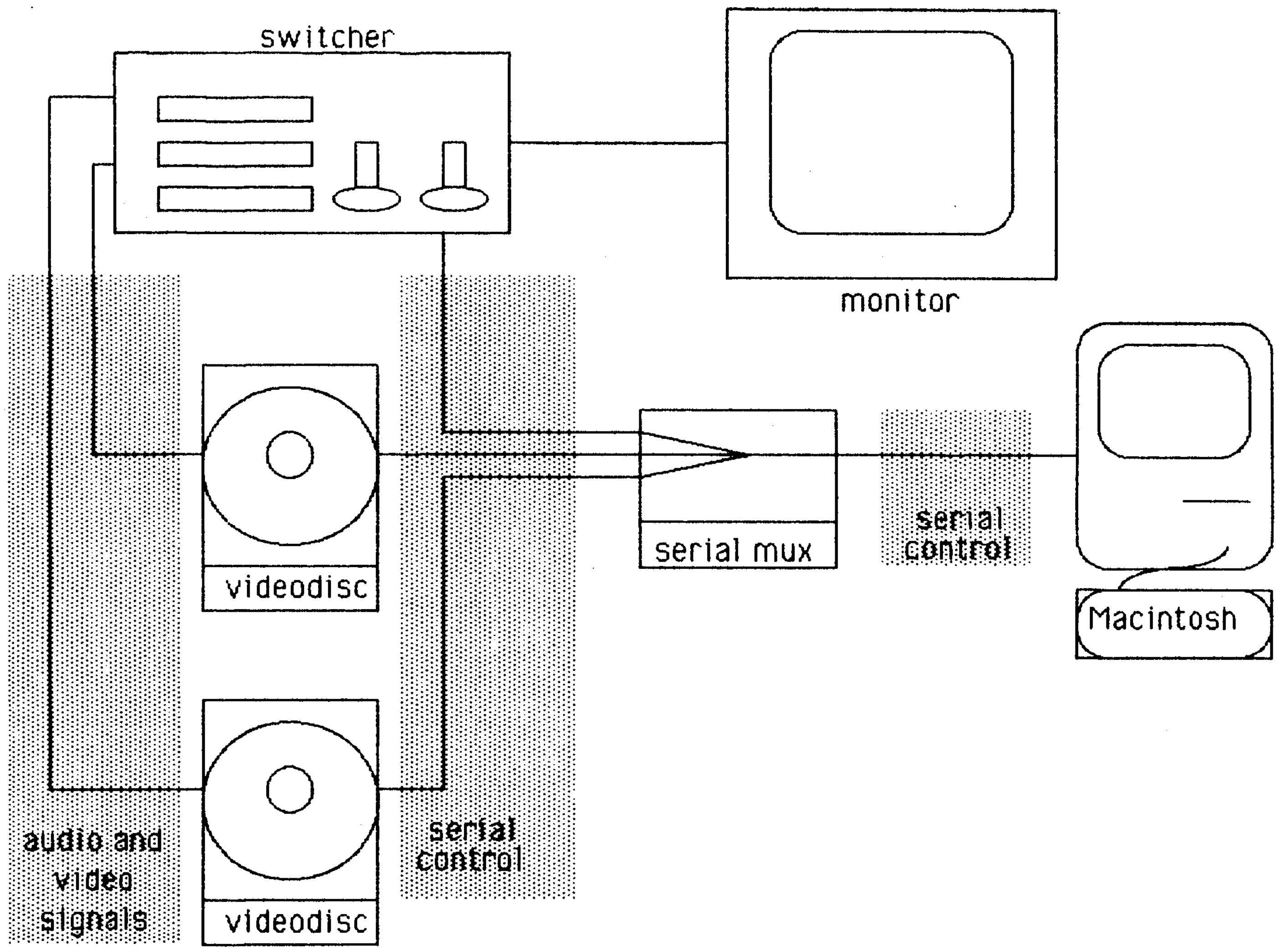

Figure 1: diagram of system hardware configuration

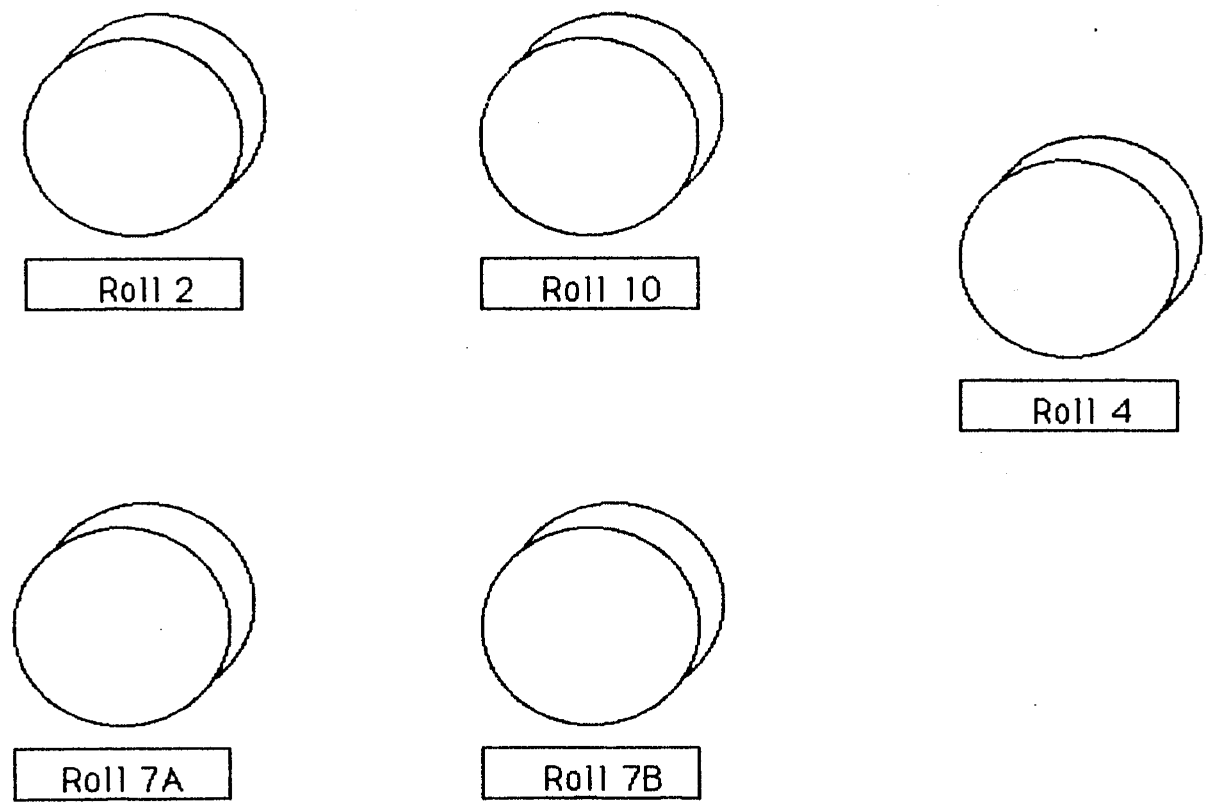

Figure 2: Sample screen display at a high level of the hierarchy

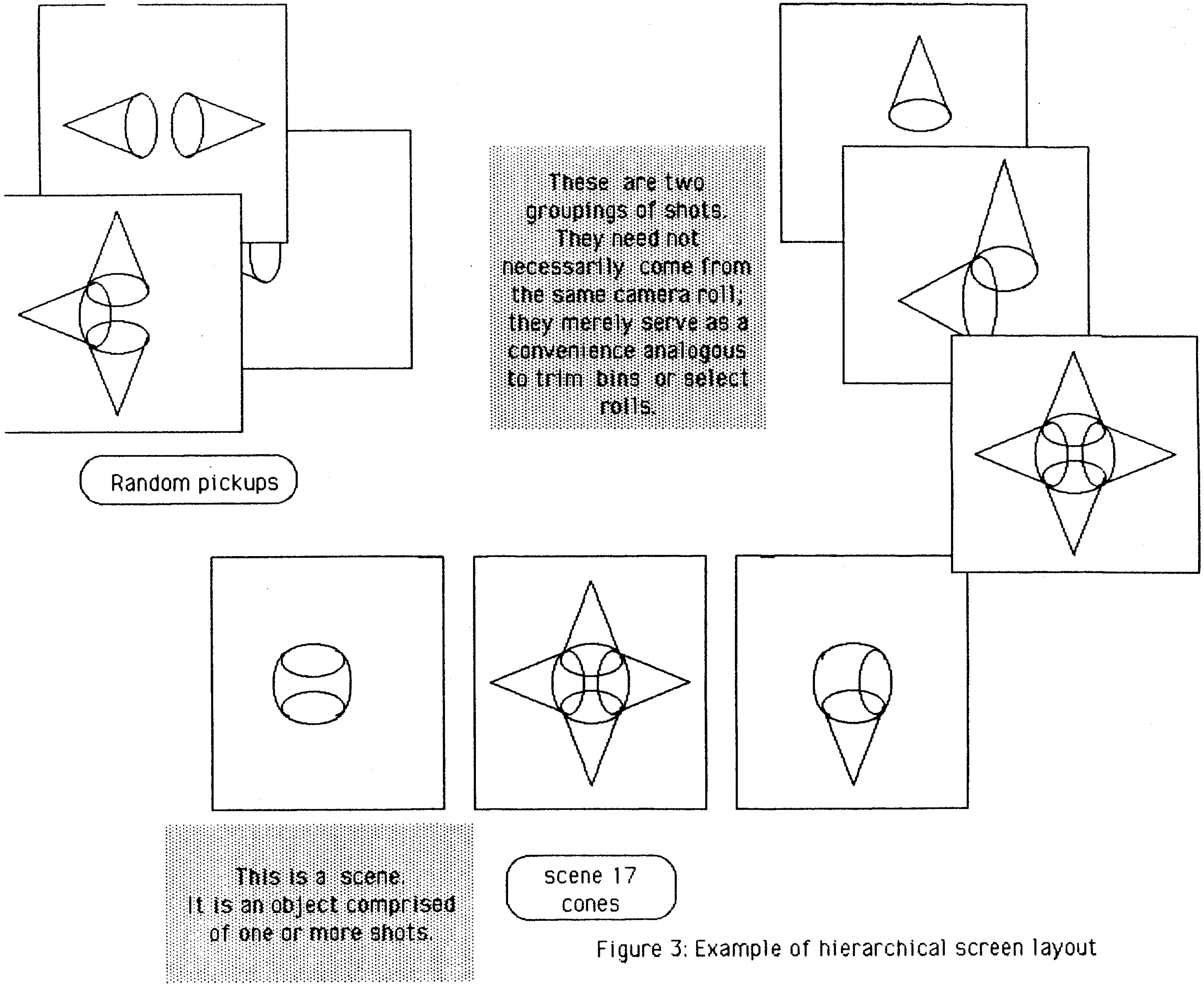
These are two groupings of shots. They need not necessarily come from the same camera roll; they merely serve as a convenience analogous to trim bins or select rolls.
Random pickups
This is a scene. It is an object comprised of one or more shots.
scene 17 cones

Figure 3: Example of hierarchical screen layout

```
/*
   Cursor handling
*/

#include <Mike.H>

/*
 * Cursor Tables
 */

static Cursor vs_curs =                    // Vertical scroller cursor
   {0x0000, 0x0000, 0x0100, 0x0380, 0x07C0, 0x0FE0, 0x0380, 0x0380,
    0x0380, 0x0380, 0x0FE0, 0x07C0, 0x0380, 0x0100, 0x0000, 0x0000,

    0x0000, 0x0000, 0x0100, 0x0380, 0x07C0, 0x0FE0, 0x0380, 0x0380,
    0x0380, 0x0380, 0x0FE0, 0x07C0, 0x0380, 0x0100, 0x0000, 0x0000,

    8, 7};

static Cursor hs_curs =                    // Horiz. scroller cursor
   {0x0000, 0x0000, 0x0000, 0x0000, 0x0420, 0x0C30, 0x1FF8, 0x3FFC,
    0x1FF8, 0x0C30, 0x0420, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,

    0x0000, 0x0000, 0x0000, 0x0000, 0x0420, 0x0C30, 0x1FF8, 0x3FFC,
    0x1FF8, 0x0C30, 0x0420, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,

    7, 8};

static Cursor grow_curs =
   {0x0000, 0x0000, 0x3F00, 0x3E00, 0x3E00, 0x3F00, 0x3F80, 0x27C0,
    0x03E4, 0x01FC, 0x00FC, 0x007C, 0x007C, 0x00FC, 0x0000, 0x0000,

    0x0000, 0x0000, 0x3F00, 0x3E00, 0x3E00, 0x3F00, 0x3F80, 0x27C0,
    0x03E4, 0x01FC, 0x00FC, 0x007C, 0x007C, 0x00FC, 0x0000, 0x0000,

    8, 8};

static Cursor scrawl_curs =
   {0x000C, 0x0012, 0x0021, 0x0059, 0x00BA, 0x0114, 0x0248, 0x0490,
    0x0920, 0x1240, 0x2480, 0xC900, 0x9200, 0x8400, 0x8800, 0xF800,

    0x000C, 0x001E, 0x003F, 0x007F, 0x00FE, 0x01FC, 0x03F8, 0x07F0,
    0x0FE0, 0x1FC0, 0x3F80, 0xFF00, 0xFE00, 0xFC00, 0xF800, 0xF800,

    14, 0};

static CursHandle edit_crs_hand;
static CursHandle wait_crs_hand;

/*
 * INIT_CURS() - Initialize Cursors
 *
 * Sets the cursor initially to the arrow, and loads the I-beam
 * and watch cursors from the system resource.
 */
init_curs()
   {
    *InitCursor();                          // Start off with the arrow
    edit_crs_hand = (CursHandle)*GetCursor(1);      // Load I-beam
    *HNoPurge(&edit_crs_hand);        // Make not purgable
    wait_crs_hand = (CursHandle)*GetCursor(4);      // Load watch
    *HNoPurge(&wait_crs_hand);        // Make not purgable
    }

/*
 * UPD_CURS() - Change cursor shape depending on where it is
 *
 * This routine changes the cursor as it is moved around the
 * screen.  Changes occur only if the cursor in in the frontmost
 * (active) window.  If it's a scrawl window, the do_scrawl routine
 * is also called.
 */
upd_curs()
    {
    Point cpt;
    WindowPtr cwp;
    short w_part, c_part;
    ControlHandle ch;

    *GetMouse(&cpt);                    // Mouse loc. (local coord's)
    *LocalToGlobal(&cpt);               // Convert to global coord's;
    w_part = (short)*FindWindow(&cpt, &cwp);  // Where in what window?

    if(cwp == NULL || cwp != dwp->wp)     // If not in live front window
       {
       *InitCursor();                   // Cursor is the arrow
       return;
       }

    //
    // Cursor in live front window.
    //
    switch(w_part)                      // Dispatch on what part of desk
       {
       case inDesk:                     // Out on the Desktop
       case inMenuBar:                  // Menu Bar
       case inSysWindow:                // Desk Accessory
       case inGoAway:                   // Go away box
       case inDrag:                     // Drag region
          *InitCursor();                // ... all use the Arrow
          break;

       case inGrow:                     // Grow region
          *SetCursor(&grow_curs);       // Use grow cursor
          break;

       case inContent:                  // Content region ...
          *GlobalToLocal(&cpt);         // Convert back to local coord's

          if(dwp->te_handle == NULL)    // If scrawl window
             {
             *SetCursor(&scrawl_curs);  // Show scrawl pencil
             do_scrawl();               // DO SCRAWLING HERE!!
             break;                     // and that's it.
```

```
            }

         if(*PtInRect(&cpt, &((*(dwp->te_handle))->viewRect)))
            *SetCursor(*edit_crs_hand);   // I-Beam if in editable text
         else                             // Must be in a scroller
            {
            *FindControl(&cpt, cwp, &ch); // Which control?
            if(ch == dwp->vs_handle)      // If vertical scroller
               *SetCursor(&vs_curs);      // Use vertical arrows
            else if(ch == dwp->hs_handle) // If horiz scroller
               *SetCursor(&hs_curs);      // Use horiz arrows
            else                          // No man's land ...
               *InitCursor();             // Use the arrow
            }
         break;

      default:
         *InitCursor();
      }
   }
```

```
/*
   Menu functions
*/

#include <Mike.H>

/*
 * SETUP_MENU() - Initialize menu system for this program
 *
 * Fills in an array of menu handles.  Not used in this version
 * of the program, but may be useful for controlling the appearance
 * of menus at run-time.
 */
setup_menu()
    {
   #InitMenus();
   #InsertMenu(menus[APPLE] = (MenuHandle)#GetMenu(APPLE_ID), 0);
   #AddResMenu(menus[APPLE], 'DRVR');
   #InsertMenu(menus[FILE] = (MenuHandle)#GetMenu(FILE_ID), 0);
   #InsertMenu(menus[EDIT] = (MenuHandle)#GetMenu(EDIT_ID), 0);
   #InsertMenu(menus[MISC] = (MenuHandle)#GetMenu(MISC_ID), 0);
   #DrawMenuBar();
    }

/*
 * DO_MENU() - Handle menu selection
 *
 * Input:
 *    Result longword from MenuSelect or MenuKey
 */
do_menu(result)
unsigned result;
    {
   unsigned short menu_id;               // Resource ID of selected menu
   unsigned short item_no;               // Item number selected
   char item_name[64];                   // Item name (for desk acc.)
   Ptr dp;                               // Dialog pointer for "about ..."
   unsigned short item_hit;              // Dialog item that was hit

   if(result == 0)                       // Just for safety with MenuKey
      return;                            // Ignore zero results

   menu_id = (short)#HiWord(result);     // Use toolbox for example
   item_no = (short)#LoWord(result);

   switch(menu_id)
      {
      case APPLE_ID:                     // "Apple" menu
         if(item_no > 1)                 // If desk accessory
            {
            #GetItem(menus[APPLE], item_no, item_name);
            #OpenDeskAcc(item_name);
            }
         else                            // Our About ... dialog
            {
            dp = #GetNewDialog(ABOUT_ID, 0, -1); // Bring in dialog to front
            #SetPort(dp);                // Hook QuickDraw up to dialog
            *ModalDialog(0, &item_hit);   // Do the dialog return item * hit
            *DisposeDialog(dp);           // Close & free heap space
            }
         break;

      case FILE_ID:       // File menu
         switch (item_no)
            {
            case FM_NEW:
               curr_edit = macedit_window ();
               break;
            case FM_OPEN:
               break;
            case FM_CLOSE:
         printf ("%s", "Now in CLOSE section\r\n");
               make_picture ();
               break;
            case FM_EXIT:
               *ExitToShell();            // Ugly exit
               break;
            default:
            }
         break;

      case EDIT_ID:
         //
         // First we must hand off any desk-accessory edit commands.
         // Note the comments in Demo.H regarding implicit assumptions
         // in numbering items in the Edit menu (boo).
         //
         if(*SystemEdit(item_no - 1))     // Relies on item numbering!!
            break;

         //
         // Do the requested edit function only if this is an editing
         // window and there is a window open.
         //
         // NOTE:  In reality, the edit options should be dimmed until
         // a live editing window is in front and active.
         //
         if(dwp == NULL || (dwp->te_handle) == NULL)  // Ignore if not editing window
            break;

         *SetPort(dwp->wp);               // Arm QuickDraw for this window

         switch(item_no)
            {
            case EM_CUT:
               *TECut(dwp->te_handle);
               break;

            case EM_COPY:
               *TECopy(dwp->te_handle);
               break;

            case EM_PASTE:
               *TEPaste(dwp->te_handle);
            break;

         case EM_CLEAR:
            *TEDelete(dwp->te_handle);
            break;

         default:
         }
      break;

   case MISC_ID:
      switch(item_no)
         {
         case MM_PLAY:
            break;

         case MM_PICTURE:
            scrawl_window ();
            break;

         case MM_TEXT:
            edit_window ();
            break;

         default:
         }
      break;

   default:
   }
*HiliteMenu(0);                                   // Turn off highlighted title
}
```

```
/*
  Common definitions
*/

// Turn off implicit toolbox trap name recognition

# Options -N

# include   <MacDefs.H>
# include   <QuickDraw.H>
# include   <Control.H>
# include   <Events.H>
# include   <Menu.H>
# include   <TextEdit.H>
# include   <MacCDefs.H>

# define MAX_WINDOWS 8
# define MAX_MENUS   8

# define TRUE     1
# define FALSE    0
# define NULL     0


// segments - basic units of hierarchy

typedef struct seg
   {
   struct seg *sons, *previous, *next; // pointers to other Segments
   short type;                         // Segment type
   int frame;                          // beginning frame
   int duration;                       // length of segment
   PicHandle picture;                  // identifying picture
   RgnHandle region;                   // region associated with the segment
   } Segment;


// wcb - Window Control Block

struct wcb
   {
   WindowPtr      wp;             // Pointer to window record in heap
   Rect           drag_rect;      // Dragging limits
   Rect           grow_rect;      // Window size limits
   TEHandle       te_handle;      // Handle to textEdit record in heap
   Point          te_origin;      // Text origin
   ControlHandle  vs_handle;      // Vertical scroller's handle
   ControlHandle  hs_handle;      // Horizontal scroller's handle
   short          type;           // Window type
   Segment        *segments;      // Pointer to list of segments
   };

/*
   Data allocation definitions
*/

#  define MAX_SEGMENTS 1000
```

```
// global data

# ifndef GLOB_DATA

extern struct wcb dw[MAX_WINDOWS];  // Our window contexts
extern struct wcb *dwp;             // --> Current doc-window
extern unsigned n_windows;          // Number of open windows
extern struct wcb *curr_edit;       // pointer to the current edit window

extern EventRecord Event;           // Our event record (duh)
extern short EvType;                // Event result type
extern WindowPtr EvWindow;          // Event window

extern Segment segments[MAX_SEGMENTS]; // segment allocation
extern Segment *free_segments;         // list of free segments
extern Segment *temp_segment;          // temporary segment pointer
extern Segment *curr_segment;          // current segment
extern Handle curr_picture;            // current picture to save

extern WindowPtr tty_window;

extern MenuHandle menus[];          // Our menu contexts
# define APPLE 1
# define FILE  2
# define EDIT  3
# define MISC  4

# else

struct wcb dw[MAX_WINDOWS];         // Our window contexts
struct wcb *dwp = NULL;             // --> Current doc-window
unsigned n_windows = 0;             // Number of open windows
struct wcb *curr_edit = NULL;       // pointer to the current edit window

EventRecord Event;                  // Our event record (duh)
short EvType;                       // Event result type
WindowPtr EvWindow;                 // Event window

MenuHandle menus[MAX_MENUS];        // Our menu contexts

Segment segments[MAX_SEGMENTS];  // segment allocation
Segment *free_segments;          // list of free segments
Segment *temp_segment;           // temporary segment pointer
Segment *curr_segment;
Handle curr_picture;

WindowPtr tty_window;

# endif

/*
   Resource ID's of menus in resource file.
*/

# define APPLE_ID 1
# define FILE_ID  256
# define EDIT_ID  257
# define MISC_ID  258

/*
   File menu item numbers
*/

# define FM_NEW      1
# define FM_OPEN     2
# define FM_CLOSE    3
# define FM_EXIT     4

/*
   Window type definitions
*/

 # define EDIT_WINDOW      0
 # define SCRAWL_WINDOW    1
 # define TEXT_WINDOW      2

/*
   Edit menu item numbers
*/

# define EM_UNDO     1
# define EM_CUT      3
# define EM_COPY     4
# define EM_PASTE    5
# define EM_CLEAR    6

/*
   Misc menu assignments
*/

# define MM_PLAY     1
# define MM_PICTURE  2
# define MM_TEXT     3


/*
   Other resource ID's
*/

# define ABOUT_ID    256
# define WINDOW_ID   256
# define EDIT_WINDOW_ID 257


/*
   External function declarations
*/

extern Segment *alloc_segment ();
extern struct wcb *macedit_window ();
```

```
/*
   Basic data types and their operators
*/


# include <mike.h>


Segment *cut_segment (segment)

register Segment *segment;
   {
   if (segment)
      {
      if (segment->previous)
         segment->previous->next = segment->next;
      if (segment->next)
         segment->next->previous = segment->previous;
      return (segment);
      }
   else
      return (-1);
   }


cut_segments (start, end)

register Segment *start, *end;
   {
   if (start && end)
      {
      if (start->previous)
         start->previous->next = end->next;
      if (end->next)
         end->next->previous = start->previous;
      return (start);
      }
   else
      return (-1);
   }


Segment *paste_segment (segment, where)

register Segment *segment, **where;
   {
   if (segment && *where)
      {
      segment->previous = (*where)->previous;
      segment->next = *where;
      (*where)->previous = segment;
      }
   else if (segment)
      *where = segment;
   else
      return (-1);

   return (0);
   }


Segment *paste_segments (start, end, where)

register Segment *start, *end, *where;
   {
   if (start && end && where)
      {
      start->previous = where->previous;
      end->next = where;
      where->previous = end;
      return (start);
      }
   else
      return (-1);
   }



init_segments ()
   {
   register int index;

   fill (0, segments, sizeof (Segment) * MAX_SEGMENTS);

   for (index = MAX_SEGMENTS; index > 0; index--)
      segments[index].previous = &segments[index - 1];

   free_segments = &segments[MAX_SEGMENTS];
   }


Segment *alloc_segment ()
   {
   register Segment *free;

   if (free_segments)
      {
      free = free_segments;

# ifdef NEED_ERROR_HANDLER
      if (!free->picture)
         {
         free = 0;
         break;
         }
# endif

      free_segments = free->previous;
      }
   else
      free = 0;

   return (free);
   }


free_segment (segment)

register Segment *segment;
   {
   if (segment)
      {
      *DisposeRgn (segment->region);
      *KillPicture (segment->picture);
      fill (0, segment, sizeof (Segment));
      segment->previous = free_segments;
      free_segments = segment;
      }
   }



fill (value, address, bytes)

char value, *address;
int bytes;
   {
   while (bytes-- > 0)
      *address++ = value;
   }
```

```
*
*

MikeRsrc
RSRCXXXX

TYPE WIND
  ,256

50 50 300 450
Visible GoAway
0
0

  ,257

100 20 250 470
Visible GoAway
0
0

TYPE MENU
  ,1
\14
About Mike ...
(-

  ,256
Segment
New
Open
Close
Quit

  ,257
Edit
(Undo /Z
(-
Cut/X
Copy/C
Paste/V
Clear/D

  ,258
Options
Play
Picture
Text

TYPE DLOG
  ,256

54 145 203 376
Visible NoGoAway
1
0
256
```

```
TYPE DITL
  ,256
2

button
116 58 142 174
RESUME MIKE

staticText Disabled
9 9 105 228
MacEdit \0D ++
Mike Tindell
```

```
/*
        Wundow functions
*/

#include <Mike.H>

static int pen_down = FALSE;
static Point pold = {0,0};

/*
 * EDIT_WINDOW() - Create an "edit window" with all of the usual junk
 *
 * Allocates a WCB, creates the window from a resource template,
 * complete with scrollers, grow box, go-away box, etc.  Sets
 * the window as the current one and activates it.
 */
edit_window()
        {
        Rect view_rect;                                                                 // TextEdit view r
        Rect bounds_rect;                                                                       // Bounds
        Rect dest_rect
        Rect *pr;                                                                                       //

        dest_rect.top = 4;                                                              // temp
        dest_rect.left = 4;
        dest_rect.bottom = 1000;
        dest_rect.right = 500;

        if((dwp = (struct wcb *)get_wcb()) == NULL)
                return;                                                                                 //

        //
        // The following sets up the window's basic dimensions and
        // rect's.  The rest of the code drives off of the basic
        // window dimensions.
        //
        dwp->type = TEXT_WINDOW;
        dwp->wp = (WindowPtr)*GetNewWindow(WINDOW_ID, 0, -1);   // Get a copy of the window
        *SetWTitle(dwp->wp, "\013Text\0");                  // Hack the title
        *SetRect(&dwp->drag_rect, 4, 24, 508, 338);         // Fixed values in example
        *SetRect(&dwp->grow_rect, 100, 60, 512, 302);
        *SetPort(dwp->wp);                                                                              //

        //
        // Set up TextEdit in the window
        //
        // For this program, the destRect is an arbitrary size
        // large enough to demonstrate horizontal and vertical
        // scrolling.  The viewRect is set to the window's portRect
        // less 4 pixels "bleed" on the left side and 15 pixels on the
        // right and bottom for the scroll bars (thanks to calc_vrect()).
        //
        pr = &((dwp->wp)->portRect);
        calc_vrect(pr, &view_rect);                         // Calculate viewRect
        dwp->te_handle = (TEHandle)*TENew(&dest_rect, &view_rect);
        dwp->te_origin.v = dwp->te_origin.h = 4;
        //
        // Add the scroll bars. Dynamically calculated from window
        // dimensions.  Note that the controls must overlap the
        // window boundaries.  Start controls out hidden, activate will
        // draw them.
        //
        bounds_rect.top = pr->top - 1;
        bounds_rect.left = pr->right - 15;
        bounds_rect.bottom = pr->bottom - 14;
        bounds_rect.right = pr->right + 1;
        dwp->vs_handle = (ControlHandle)#NewControl(dwp->wp, &bounds_rect,
                                                     "", TRUE,
                                                     dest_rect.top, dest_rect.top, dest
                                                     scrollBarProc, 1);
        #ValidRect(&bounds_rect);

        bounds_rect.top = pr->bottom - 15;
        bounds_rect.left = pr->left - 1;
        bounds_rect.bottom = pr->bottom + 1;
        bounds_rect.right = pr->right - 14;
        dwp->hs_handle = (ControlHandle)#NewControl(dwp->wp, &bounds_rect,
                                                     "", TRUE,
                                                     dest_rect.left, dest_rect.left, de
                                                     scrollBarProc, 1);
        #ValidRect(&bounds_rect);

        //
        // Finally, draw in the "grow icon"
        //
        #DrawGrowIcon(dwp->wp);
        n_windows++;
        }

/*
 * SCRAWL_WINDOW - Create a "scrawl" window
 */
scrawl_window()
        {
        if((dwp = (struct wcb *)get_wcb()) == NULL)
              return;                                                              //
        //
        // See comments above ...
        //
        dwp->type = SCRAWL_WINDOW;
        dwp->wp = (WindowPtr)#GetNewWindow(WINDOW_ID, 0, -1);   // Get a copy of the window
        #SetWTitle(dwp->wp, "\015Picture\0");            // Hack the title
        #SetRect(&dwp->drag_rect, 4, 24, 508, 338);      // Fixed values in example
        #SetRect(&dwp->grow_rect, 100, 60, 512, 302);
        #SetPort(dwp->wp);                                                 // hook back up tc
        #BackPat(&(QD->white));                                    // Background is black
        #PenPat(&(QD->black));                                     // Pen is white (redundant
        #PenSize(2,2);                                                           // 2x2 pix
        #EraseRect(&(dwp->wp->portRect));                // Establish new background
        n_windows++;

        temp_segment = alloc_segment ();
        temp_segment->picture =
                        (PicHandle)*OpenPicture (&((WindowPeek)(dwp->wp))->contRgn);
        *ShowPen ();                                                              // OP call hides p
        curr_picture = dwp->wp->picSave;
        dwp->wp->picSave = 0;                                    // stop saving calls as picture

        return(0);
        }



/*
        Make an edit window
*/

struct wcb *macedit_window ()
        {
        if((dwp = (struct wcb *)get_wcb()) == NULL)
                return;                                                                              //

        dwp->type = EDIT_WINDOW;
        dwp->wp = (WindowPtr)*GetNewWindow(EDIT_WINDOW_ID, 0, -1);      // Get a copy of the windc
        *SetWTitle(dwp->wp, "\015Edit");                        // Hack the title
        *SetRect(&dwp->drag_rect, 4, 24, 508, 338);     // Fixed values in example
        *SetRect(&dwp->grow_rect, 100, 60, 512, 302);
        *SetPort(dwp->wp);                                                        // hook back up tc
        *BackPat(&(QD->white));                                         // Background is white
        *PenPat(&(QD->black));                                          // Pen is black (redundant
        *PenSize(2,2);                                                                    // 2x2 pix
        *EraseRect(&(dwp->wp->portRect));               // Establish new background

        n_windows++;

        return (dwp);
        }



/*
 * DELETE_WINDOW() - Remove window from screen & dispose of all structures
 *
 * Inputs:
 *                      dp      --> WCB of the window to delete
 *
 *      Outputs:
 *                              none
 *
 * Calls the *DisposeWindow service to remove the window from the screen
 * and release all associated data structures, including the window
 * record. NOTE - assumes window was created with record on heap.
 * Then frees up the WCB associated with the window.
 */
delete_window(dp)
struct wcb *dp;
        {

        if(dp->te_handle != NULL)                                                 // If edit window
                {
```

```
                *TEDispose(dp->te_handle);                                      // Return TE recor
                dp->te_handle = NULL;                                           // Mark TE inactiv
                }
        *DisposeWindow(dp->wp);                                                 // Dispose of the
        dp->wp = NULL;                                                                          //

        n_windows--;                                                                            //
        }

/*
 *      CONTENT_CLICK() - Handle mouse down in window content region
 *
 * Inputs (global variables only)
 *              dwp             -->     WCB of window in which mouse was clicked (verified)
 *              Event.where             Mouse-down location (global coordinates)
 *
 * Outputs:
 *              No explicit outputs
 *
 */
content_click()
        {
        unsigned short ext;
        ControlHandle ch;
        unsigned short part;
        long foo;
        Rect *_vr;
        int scroll_up();
        int scroll_down();

        //
        // Mouse-down in scrawl window changes the "old" position
        // to the current position before allowing drawing.  Then
        // it turns on the pen-down flag.
        //
        if(dwp->type == SCRAWL_WINDOW)                  // If this is a "scrawl" window
                {
                *GetMouse(&pold);
                pen_down = TRUE;
                return(0);
                }
        else if (dwp->type == TEXT_WINDOW)
                {
                //
                // We have a mouse-down in content of an edit window.
                //
                // If it's in the viewRect, do the TEClick.  Otherwise, handle
                // the click in a control.
                //

                *GlobalToLocal(&Event.where);                           // Convert coordinates to
                foo = *PtInRect(&Event.where, &((*(dwp->te_handle))->viewRect));
                if(foo)
                        *TEClick(&Event.where, 0, dwp->te_handle); // 0 should be "ext"
                else
                        {
                        part = (short)*FindControl(&Event.where, EvWindow, &ch);
        switch(part)
            {
            case inUpButton:
                *TrackControl(ch, &Event.where, scroll_up);
                break;

            case inDownButton:
                *TrackControl(ch, &Event.where, scroll_down);
                break;

            case inPageUp:
                page_scroll(part, ch, -1);
                break;

            case inPageDown:
                page_scroll(part, ch, 1);
                break;

            case inThumb:
                *TrackControl(ch, &Event.where, 0);
                edit_scroll();
            }
        }
    }
else if (dwp->type == EDIT_WINDOW)
    {
    Segment *segments;
    unsigned long location;

    segments = dwp->segments;
    location = 0;

    while (segments)
        {
        printf ("\r\nSegments region %x", segments->region, "\r\n");
        if (*PtInRgn (&Event.where, segments->region))
            {
            location = (unsigned long)*DragGrayRgn
                                            (segments->region, &Event.where,
                                            &(*((WindowPeek)(dwp->wp))->contRg
                                            &(*((WindowPeek)(dwp->wp))->contRg
                                            0, 0);
            break;
            }
        else
            segments = segments->previous;
        printf ("\r\nSegments previous %x", segments->previous, "\r\n");
        }
    printf ("\r\n%s", "Out of first while\r\n");

    if (!location)
        {
        segments = dwp->segments->next;

        printf ("\r\nSegments  %x", segments, "\r\n");
        while (segments)
            {
```

```
                                        printf ("\r\nSegments region %x", segments->region, "\r\n");
                                        if (*PtInRgn (&Event.where, segments->region))
                                                {
                                                location = (unsigned long)*DragGrayRgn
                                                                                (segments->region, &Event.
                                                                                &(*((WindowPeek)(dwp->wp):
                                                                                &(*((WindowPeek)(dwp->wp):
                                                                                0, 0);
                                                break;
                                                }
                                        else
                                                segments = segments->next;
                                        printf ("\r\nSegments next %x", segments->next, "\r\n");
                                        }
                                printf ("\r\n%s", "Out of second while\r\n");
                                }

                        if (location)
                                {
                                // reposition region
                                }
                        printf ("Done with content click in edit window\r\n  Location %i", location);
                        }
                else;
                UNBUG;                                                  // hook back up to dwp
                }

/*
 * CONTENT_RELEASE() - Release of mouse in content region
 *
 * Only if scrawl window, toggle the draw/move flag
 */
content_release()
        {
        if(dwp->type == SCRAWL_WINDOW)
                pen_down = FALSE;
        return(0);
        }

/*
 * DO_SCRAWL() - Handle scrawling
 *
 * This is actually called fro the cursor handling
 * routine, since it is needed whether or not the mouse
 * is down.
 */
do_scrawl()
        {
        Point p;

        *GetMouse(&p);
    if((pold.vh[0] == p.vh[0]) && (pold.vh[1] == p.vh[1]))
      return;
        pold.vh[0] = p.vh[0];
        pold.vh[1] = p.vh[1];

    if(pen_down)
```

```
                {
                dwp->wp->picSave = curr_picture;
                *LineTo(p.vh[1],p.vh[0]);
                dwp->wp->picSave = 0;
                }
        else
                {
                dwp->wp->picSave = curr_picture;
      *MoveTo(p.vh[1],p.vh[0]);
                dwp->wp->picSave = 0;
                }
        }

/*
 * ACT_WIND() - Do activate stuff for window
 */
act_wind(wp)
WindowPtr wp;                                                                       // Window
        {
        struct wcb *find_wcb();
        struct wcb *dp;

        if((dp = find_wcb(wp)) == NULL)                     // Make this the current
                return(0);                                                              //

        *SetPort(wp);                                                               // This is

        if(dp->type == SCRAWL_WINDOW)                               // If it's a scrawl window
                {
                // Nothing for now
                }
        else if (dp->type == TEXT_WINDOW)                   // Do textedit window
                {
                *DrawGrowIcon(wp);                                            // Draw activated
                *TEActivate(dp->te_handle);                         // Turn on text editing
                *ShowControl(dp->vs_handle);                        // Draw in scrollers
                *ShowControl(dp->hs_handle);
                }
        else if (dp->type == EDIT_WINDOW)                   // Do edit window
                {
                printf ("%s", "Now in act_wind EDIT section\r\n");
                if (dp->segments)
                        {
                        Segment *segments;

                        segments = dp->segments;

                        while (segments->previous)
                                {
                                *DrawPicture (segments->picture, segments->region);
                                segments = segments->previous;
                                }

                        segments = dp->segments;

                        while (segments->next)
                                {
```

```
                                *DrawPicture (segments->picture, segments->region);
                                segments = segments->next;
                                }
                        }
                else;
                }
        else;

        dwp = dp;                                                                                        //
        }

/*
 * DEACT_WIND() - Do deactivation stuff for window
 */
deact_wind(wp)
WindowPtr wp;
        {
        struct wcb *dp;
        struct wcb *find_wcb();

        if((dp = find_wcb(wp)) == NULL)                  // Find our WCB for this window
                return(0);                                                                       //

        *SetPort(wp);                                                                    // This is

        if(dp->type == SCRAWL_WINDOW)                            // If it's a scrawl window
                {
                return(0);
                }
        else if (dp->type == TEXT_WINDOW)                // Do textedit window
                {
                *DrawGrowIcon(wp);                                               // Draw deactivate
                *TEDeactivate(dp->te_handle);                    // Turn off text editing
                *HideControl(dp->vs_handle);                     // Hide scrollers
                *HideControl(dp->hs_handle);
                }
        else if (dp->type == EDIT_WINDOW)                // Do edit window
                {
                }
        else;

        dwp = NULL;                                                                                      //
        }

/*
 * UPD_WIND() - Update window
 *
 * Note that this may be called whether or not the window
 * is the front window and/or active.  This must not change
 * our assumptions about the front window.
 *
 * More interesting is the fact that this routine gets called
 * continuously for each window.  If BeginUpdate sets the visRgn to
 * anything but the empty region, something gets drawn.
 */
upd_wind(wp)
WindowPtr wp;
```

```
	{
	struct wcb *find_wcb();
	struct wcb *dp;


	if((dp = find_wcb(wp)) == NULL)			// Get the WCB for this window
		return(0);										//

	*BeginUpdate(wp);									// Fudge

	*SetPort(wp);										// Hook up

	if(dp->type == SCRAWL_WINDOW)				// If it's a scrawl window
		*EraseRect(&(wp->portRect));			// Just wipe it clean
	else if (dp->type == TEXT_WINDOW)		// Do edit window
		{
		*DrawGrowIcon(wp);								// Draw the grow i
		*DrawControls(wp);								// Draw all contrc
		//
		// Note the syntax for addressing the rgnBbox of the window's
		// grafPort's visRgn.  This is the recommended Rect to use
		// when calling TEUpdate.  Who says C is easy to read?
		//
		*TEUpdate(&((*(wp->visRgn))->rgnBBox), dp->te_handle);
		}
	else if (dp->type == EDIT_WINDOW)
		{
		}
	else;

	*EndUpdate(wp);									// Restore visRgn
	}

/*
 * DO_GROW() - Handle grow/resize operations on window
 *
 * Inputs:
 *		(G)	dwp -->		WCB of current window
 *		(G)	Event		Event record for click in grow box
 *
 * Outputs:
 *		none
 *
 * Much of this code is skipped for a scrawl window.  The operations
 * done here generate an update event for the window.  Keep this in mind
 * and try to avoid redundant drawing.  Above all -- avoid redrawing the
 * entire window ... it's just too crude for the Mac.
 */
do_grow()
	{
	Rect *pr;
	Rect temp_rect;
	unsigned long grow_result;
	unsigned short height, width;

	//
	// Do the grow operation while mouse is held down.  When it is
// released, the new height and width of the window are returned
// packed in a longword.  Note that a zero result indicates no
// change, and we can skip this whole thing.
//
grow_result = *GrowWindow(dwp->wp,&Event.where, &dwp->grow_rect);
if(grow_result == 0)
        return;
height = (unsigned short)*HiWord(grow_result);
width = (unsigned short)*LoWord(grow_result);
*SetPort(dwp->wp);                                                              // Hook up
pr = &((dwp->wp)->portRect);                                            // pr --> new portRect of

if(dwp->te_handle != 0)                                                         // TextEdit window
        {
        //
        // The scroll bars must be manually accumulated into the update
        // region.  See the window manager manual for a fairly lucid
        // explanation of this.  We also have to invalidate the size box.
        // Note we are making assumptions here about the size and location
        // of the scrollers.  This stuff handles the case where the window
        // enlarges.  The calls to DrawControls and DrawGrowIcon in the
        // update event handler take care of the shrink case.
        //
        temp_rect.top = pr->top;
        temp_rect.bottom = pr->bottom;
        temp_rect.right = pr->right;
        temp_rect.left = temp_rect.right - 16;  // Right hand 16 pixels
        *InvalRect(&temp_rect);                                                 // Add it to the u
        temp_rect.top = temp_rect.bottom - 16;  // Size box
        *EraseRect(&temp_rect);                                                 // Erase size box
        temp_rect.left = pr->left;                                              // Bottom 16 pixel
        *InvalRect(&temp_rect);                                                 // Add it to the u
        }

//
// Now we re-size and set update on the winbdow.  Then move and
// resize the controls.  Finally, update the text-edit "viewRect".
//
*SizeWindow(dwp->wp, width, height, TRUE); // Resize & start update rgn

if(dwp->te_handle != 0)                                                 // Only for TextEdit windo
        {
        *MoveControl(dwp->vs_handle, pr->right - 15, pr->top - 1);
        *SizeControl(dwp->vs_handle, 16,(pr->bottom - pr->top - 13));

        *MoveControl(dwp->hs_handle, pr->left - 1, pr->bottom - 15);
        *SizeControl(dwp->hs_handle, (pr->right - pr->left - 13), 16);

        //
        // Note the direct access to TE's view rect via the handle
        // we stored in our WCB.
        //
        calc_vrect(pr, &((*(dwp->te_handle))->viewRect));
        }
}

/*
```

```
 * CALC_VRECT() - Make a TextEdit viewRect from portRect
 *
 * Inputs:
 *              pr      -->     Window's portRect
 *              vr      -->     where to fill in viewRect
 *
 * Outputs:
 *              vr->topLeft and vr->botRight are filled in
 *
 * The TextEdit viewRect is set up to provide 15-pixel strips along
 * the right hand and bottom edges of the dditing window (for scrollers
 * and the size box. It also provides a 4-pixel "bleed" at the left edge.
 * and at the top.
 */
calc_vrect(pr, vr)
Rect *pr;
Rect *vr;
        {
        vr->top = pr->top + 4;
        vr->left = pr->left + 4;
        vr->bottom = pr->bottom - 15;
        vr->right = pr->right - 15;
        }

/*
 * PAGE_SCROLL() - Scroll a page per indicator
 *
 *      Inputs:
 *              part            Part code where first clicked down
 *              ch                      control handle
 *              dir             direction (-1 = up, +1 = down)
 *
 * Outputs:
 *              none
 */
page_scroll(part, ch, dir)
short part;
ControlHandle ch;
short dir;
        {
        Point cur_pt;
        short amount;
        Rect *vr;

        //
        // First, calculate the "page size" in pixels.  For V scroll, it is
        // the viewRect height less the line height.  For H scroll, it is
        // half the width of the viewRect.
        //
        vr = &((*(dwp->te_handle))->viewRect);  // vr -> current ViewRect
        if(ch == dwp->vs_handle)                                        // If this is the
                amount = vr->bottom - vr->top - (short)get_lh();
        else
                amount = (vr->right - vr->left) / 2;
        amount *= dir;                                                  // Change

        //
        // Now we must locally handle the mouse per the Macintosh Interface
        // Guidelines.  Look closely at this code and note what it really
        // does ...
        //
        do
                {
                #GetMouse(&cur_pt);                                         // Get current mou
                if((short)#TestControl(ch, &cur_pt) != part)    // If out of original part
                        continue;                                                   //
                #SetCtlValue(ch, #GetCtlValue(ch) + amount);    // Page control value
                edit_scroll();                                                  // Page th
                } while(#StillDown());
        }


/*
 * SCROLL_UP() - Scroll up a text edit window
 *
 * This routine is called back from the toolbox with (naturally)
 * Pascal-flavored arguments on the stack.  I've taken this
 * opportunity to demonstrate inline assembly language and how
 * to act like a Pascal procedure.
 *
 * Note that the scrolled amount is equal to the line height as
 * stored in the TextEdit record for both vertical and horizontal
 * scrolling.
 */
scroll_up()
        {
#asm
;
; Inputs:
;                       4(sp)                   Part code (int)
;                       6(sp)                   Control handle (address)
;
; NOTE: I have done the unforgivable & extracted the trap values
;                       from the trap definitions rether than include the D file.
;                       I'm impatient ...
;
                Link            a6,#0
                Move.W  8(a6),d0                        ; D0 = part code (W)
                Beq             @1                                      ; (0 means out of part reg
                Move.L  10(a6),-(sp)    ; Push control handle (for later)
                Clr.W           -(sp)                           ; Gets control value
                Move.L  10(a6),-(sp)    ; Push control handle
                DC.W            $A960                           ; _GetCtlValue (value on stack)
                Jsr             get_lh                  ; D0 = text line height, pixels (easier ir
                Sub.W           d0,(sp)                 ; (SP) = new control value
                Bge             @0                                      ; (ok, its positive)
                Clr.W           (sp)                            ; Stop control at 0 value
@0:     DC.W            $A963                   ; _SetCtlValue
                Jsr             edit_scroll             ; call C text scroller
@1:
                Unlk            a6                                      ; "standard" Pascal routir
                Move.l  (sp)+,a0
                Addq            #6,sp
                Jmp             (a0)
```

```
#endasm
        }

/*
 * SCROLL_DOWN() - Scroll down a text edit window
 *
 * This routine is called back from the toolbox with (naturally)
 * Pascal-flavored arguments on the stack. See comments above.
 */
scroll_down()
        {
#asm
;
; Inputs:
;                       4(sp)                   Part code (int)
;                       6(sp)                   Control handle (address)
;
                Link            a6,#0                           ; no local automatics
                Move.W  8(a6),d0                        ; D0 = part code (W)
                Beq             @1                                      ; (0 means out of part reg
                Move.L  10(a6),-(sp)    ; Push control handle (for later)
                Clr.W           -(sp)                           ; Gets control value
                Move.L  10(a6),-(sp)    ; Push control handle
                DC.W            $A960                           ; _GetCtlValue (value on stack)
                Jsr             get_lh                  ; D0 = text line height, pixels (easier ir
                Add.W           d0,(sp)                 ; (SP) = new control value
                DC.W            $A963                           ; _SetCtlValue
                Jsr             edit_scroll             ; call C text scroller
@1:
                Unlk            a6                                      ; "standard" Pascal routir
                Move.l  (sp)+,a0
                Addq            #6,sp
                Jmp             (a0)
#endasm
        }

/*
 * EDIT_SCROLL() - Scroll edit window per control values
 *
 * Inputs:
 *              Current value of te_origin in the WCB
 *              Current values of the scrollers
 *
 * Outputs:
 *              Origin points in the WCB are updated
 *              Enforces a low limit of 0 on the controls
 */
edit_scroll()
        {
        short int dh, dv;

        dh = dwp->te_origin.h - (short int)*GetCtlValue(dwp->hs_handle);
        if(dh <= dwp->te_origin.h)
                dwp->te_origin.h -= dh;
        else
                dwp->te_origin.h = 4;
        dv = dwp->te_origin.v - (short int) GetCtlValue(dwp->vs_handle);
        if(dv <= dwp->te_origin.v)
                dwp->te_origin.v -= dv;
        else
                dwp->te_origin.v = 4;

        *TEScroll(dh, dv, dwp->te_handle);
        }
}

/*
 * GET_LH() - Return text line height per TextEdit record
 *
 * Inputs:
 *              dwp --> current WCB
 *
 *      Outputs:
 *              returns text line height for scrolling
 */
get_lh()
        {
        return((*(dwp->te_handle))->lineHeight);  /* Easy in C!!! */
        }

/*
 * GET_WCB() - Allocate a WCB for a new window
 *
 * Inputs:
 *              none
 *
 * Outputs:
 *              Returns --> allocated WCB or NULL if none left
 */
get_wcb()
        {
        int i;

        for(i=0; i<MAX_WINDOWS; i++)
                if(dw[i].wp == NULL)
                        {
                        dw[i].te_handle = NULL;                 // Mark editing not ready this wir
                        return(&dw[i]);
                        }
        return(0);
        }

/*
 * FIND_WCB() - Find window control block for given window
 *
 * Returns WCB pointer or 0 if given window pointer does not
 * belong to one of our WCB's.
 */
find_wcb(wp)
WindowPtr wp;
        {
        int i;

        for(i=0; i<MAX_WINDOWS; i++)
```

```
				return(&dw[i]);
	return(0);
	}



/*
	Complete a picture
*/

make_picture ()
	{
	if (curr_edit && temp_segment)
		{
		dwp->wp->picSave = curr_picture;

		#ClosePicture ();
		delete_window (dwp);
		act_wind (curr_edit->wp);
		#DrawPicture (temp_segment->picture,
						((WindowPeek)(curr_edit->wp))->contRgn);
		#DrawPicture (temp_segment->picture,
						((WindowPeek)tty_window)->contRgn); // dit

		temp_segment->region = (RgnHandle)#NewRgn ();
		#SetRectRgn (temp_segment->region, 125, 200, 225, 300);
		#DrawPicture (temp_segment->picture, temp_segment->region);// for real

		paste_segment (temp_segment, &curr_edit->segments);

		temp_segment = NULL;
		}
	else;		// error

	return (0);
	}
```