5-1991

MatrixCad Version 1.0

Terry D. Hawkins

Follow this and additional works at: http://opensiuc.lib.siu.edu/uhp_theses

Recommended Citation
MatrixCad Version 1.0

by
Terry D. Hawkins

for
Honors Program Independent Study

Academic Advisor:  Dr. M. S. Wainer
Department of Computer Science
Southern Illinois University at Carbondale

May 10, 1991
My intent and purpose for this project was to explore the much-valued techniques of object oriented programming (OOP). In particular, I wanted to apply these techniques to the graphical user interface Windows 3.0 from MicroSoft. Windows has developed a reputation for being as difficult and tedious for the programmer as it is easy to use for the typical user. The techniques of object oriented programming are reputed to improve programmer-productivity and program reliability. Windows 3.0 seemed to be a stringent test for this claim.

I initially began this project, writing a matrix manipulation program, using MicroSoft's C Optimizing Compiler, along with the MicroSoft Software Development Kit (SDK) for Windows 286. This setup did not support object-oriented programming. The SDK came with several thousand pages of documentation, covering many features of Windows programming as well as the 600+ functions in the library. Most of those functions required quite a few parameters. I found that even the simplest 'Hello World' program required several hundred lines of code, mostly for setup and initialization. The difficulty in programming in this environment was compounded by having to use separate, non-integrated tools (the compiler, the linker, resource compiler, the editor, ...). I made progress, but it was slow and tedious, with a lot of time spent combing through the documentation.

About five weeks before the end of the semester, I purchased Borland's newly released Turbo Pascal for Windows. This software provided a fully integrated development environment complete with editor, compilers, a debugger, a resource toolkit, and plenty of sample code. In addition, Turbo Pascal's OOP extensions were supported, with a well-designed library of window classes were included as well. Using this system, I was able to translate and re-design my C code into a fully object-oriented application in Turbo Pascal.

Having written the matrix application essentially twice, once without object extensions, and then with, gave me a unique opportunity to see the advantages of object orientated programming. Using the Turbo Pascal system, I was able to increase my productivity by a factor of four, with far fewer bugs and 'resets' along the way. Much of this gain was due to Turbo's integrated environment; however, the object extensions made it much easier to write well-structured, bugfree code.

Object programming seems to encourage a more global, top-down design approach without imposing any real restrictions upon the programmer's creativity. It is an especially natural fit with the message-passing, window-based
Windows 3.0, OOP also promises to approach the ideal of truly reusable code, by allowing one to extend existing code without actually modifying it.

Despite the improvements in software tools such as Turbo Pascal, though, Windows programming remains a formidable affair. Even in Turbo Pascal, the Microsoft SDK is still buried under all the pretty windows and OOP extensions, and the programmer who needs to do more than just the simplest Windows applications must still learn to deal with it.
MatrixCad is designed to provide maximum flexibility and ease of use while working with two dimensional matrices. Version 1.0 provides the ability to manipulate many different matrices on the screen at the same time with several methods of selection and display. The individual matrices are easily navigated and edited using a moving cell pointer indicated by a solid black border around the current cell. Both unary and binary mathematical operations are included; the unary operation is scalar multiplication; binary operations included are addition, subtraction, and multiplication.

MatrixCad makes use of the Microsoft Windows 3.0 Multiple Document Interface (MDI). This interface provides a number of very flexible window manipulation options, including window resizing, moving from one position to another, minimizing a window, and maximizing a window.

To resize a window (changing the actual size of the window on the screen), move the mouse cursor to any border on the window until the cursor changes to a double arrow. While holding down the left mouse button, drag the border of the window to the desired size, and then release the mouse button. The window will repaint itself to the new size.

To move a window to a new screen position, move the mouse cursor to the title bar of the window. Press the left mouse button down, and while holding it down, move the window to the desired position. Release the left mouse button and the window will repaint itself in the new position.

To minimize a window (change it to an icon), move to the right side of the window title bar and click on the button with the down arrow. The window will change to a small icon on the bottom of the parent window.

The system menu is opened by clicking on the button to the left side of the title bar. It’s options include window minimizing, maximizing, and closing.

MatrixCad’s Window menu selection provides means to:
1. create a new matrix window. New matrices are initialized to 4 rows by 4 columns, with all elements set to 0. Each matrix window is automatically named according to its creation order (i.e. the third matrix window created is named Matrix #3).

2. arrange matrix windows by cascading or tiling. Cascading of windows means to arrange the matrix windows overlapping each other, top left to bottom right, with title bars showing for easy selection. Tiling of windows resizes and repositions each matrix window so that no window overlaps another, and the parent window's client area is completely covered.

3. arranging icons of matrix windows. This option puts any existing icons into a neat row at the bottom left of the parent window's client area.

4. closing all matrix windows. Does exactly that, leaving a clean window, so make sure that's what you really want to do.

5. selecting a matrix window from a list. Any open matrix window can be selected by name from a list, with the currently active matrix window indicated with a checkmark.

The menu selections Unary Operations, Binary Operations, and Settings, use pop-up windows called dialog boxes to obtain user input. Each dialog box has an OK button, selected by either clicking with the mouse cursor or by pressing enter on the keyboard, which basically means "I like the selections I've just made, so do the operation I've selected...". A corresponding CANCEL button terminates the selected operation. The input fields can be moved from one to another using the mouse cursor or the tab key.

Only one unary matrix operation is currently supported, and that is multiplication by a scalar. Scalar multiplication only affects the currently active matrix.

The binary operations supported are addition, subtraction, and multiplication. The Binary Operations dialog box provides the ability to specify the operation (addition is the default), the matrices to do the operation with (specified as the left and right operands), and a target
matrix. The operand and target matrices default to the first matrix created, and can be changed using the mouse cursor.

The menu option Settings provides the ability to change the size of the matrix in terms of the number of rows and columns. Changing column size will "re-flow" matrix entries into next/previous rows. MatrixCad internally keeps a matrix as a single dimensional array, and simply interprets that array as a matrix in line with the current row/column settings.

Inside a matrix window, the current cell is indicated by a solid black border, called the cell pointer. The keyboard arrow keys move the cell pointer left, right, up, and down, wrapping around both columns and rows. Pressing a numeric key puts the cell pointer into editing mode, allowing data entry into the current cell. Editing mode is indicated by an 'H-beam' cursor appearing in the current cell. If editing mode is terminated by pressing the enter key, the current cell is updated, and the cell pointer moves to the next cell and re-enters edit mode. This allows rapid entry of matrix cell values. Editing mode can be exited by pressing the escape key, which will restore the contents to the current cell.
Program name : MatrixCad
Version # : 1.0
Language : Borland´s Turbo Pascal for Windows
Extensions : Borland´s ObjectWindows

Programmer : Terry D. Hawkins
Academic Advisor : Dr. M. S. Wainer
Completion Date : May 10, 1991
Course : UHON 399

Description : MatrixCad provides addition, subtraction, and multiplication of multiple matrices with emphasis on flexibility and ease of use.

Purpose : To explore the advantages of object-oriented programming techniques, in particular within the context of a graphical-user-interface.

program MatrixCad;
{$R \tpw\MDI.RES}
uses WObjects, WinTypes, WinProcs, Strings, StdDlgs;

CONSTANTS
const
  cm_CountChildren = 102;
  id_CantClose = 201;
  id_cell = 301;

  cm_specs = 1001;
  cm_scalarMult = 501;
  cm_BinaryOps = 601;

specs dialog box id´s }
  id_rows = 1101;
  id_cols = 1102;

ops dialog id´s }
  id_TimesButton = 2201;
  id_PlusButton = 2202;
  id_MinusButton = 2203;
  id_opStatic = 2301;
{ user defined message constants }
wm_CellReturned = wm_User;
wm_CellEscaped = wm_User + 1;

{************************************************************************
{ TYPE DECLARATIONS
{************************************************************************
type

{--- Application -----------------------------------------------}
TMatrixMDIApp = object(TApplication)
  procedure InitMainWindow; virtual;
end;

{--- Matrix Specifications Dialog Object ------------------------}
PSpecsDialog ='TSpecsDialog;
TSpecsDialog = object(TDialog)
  procedure ok (var msg : TMessage);
    virtual id_First + id_OK;
end;

{--- Binary Operations Dialog Object -----------------------------}
POpsDialog = 'TOpsDialog;
TOpsDialog = object(TDialog)
  procedure TrapPlusButton(var Msg:TMessage);
    virtual id_First + id_PlusButton;
  procedure TrapTimesButton(var Msg:TMessage);
    virtual id_First + id_TimesButton;
  procedure TrapMinusButton(var Msg:TMessage);
    virtual id_First + id_MinusButton;
end;

{--- Settings Dialog Box Transfer Record -------------------------}
TransferSpecsRecord = record
  NumRows, NumCols : array[0..32] of Char;
end;

TransferOpsRecord = record
  operation: array[0..5] of Char;
    LOpList : PStrCollection;
    Lindex : integer;
    ROpList : PStrCollection;
    Rindex : integer;
    TOpList : PStrCollection;
    Tindex : integer;
end;

{ Cell type }
PCell = ^TCell;
TCell = Object(TObject)
  e : integer;
  constructor init(r : integer);
  procedure print; virtual;
end;

{ Cell Window type -----------------------------------------------}
PCellWindow = ^TCellWindow;
TCellWindow = object(TEdit)
  procedure dataEntry (var Msg : TMessage);
      virtual wm_first + wm_KeyDown;
end;

{ MDI Child Window type ----------------------------------------}
PMatrixMDIChild = ^TMatrixMDIChild;
TMatrixMDIChild = object(TWindow)
  Num: Integer;
  Name : PChar;
  Description : PChar;
  TopLabel : PChar;
  SideLabel : PChar;
  Cells : PCollection;
  CellWindow : PCellWindow;
  CellPaint : boolean;
  Rows : integer;
  Cols : integer;
  CurrRow : integer;
  CurrCol : integer;
  xStart : integer;
  yStart : integer;
  cellwidth : integer;
  cellHeight : integer;
  constructor Init (AParent: PWindowsObject; ChildNum: Integer);
  procedure SetupWindow; virtual;
  procedure Paint (PaintDC: HDC; var PaintInfo: TPaintStruct); virtual;
  procedure TrapKeyboard (var Msg: TMessage);
      virtual wm_first + wm_KeyDown;
  procedure TrapReturn(var Msg: TMessage);
      virtual wm_first + wm_CellReturned;
  procedure TrapEscape(var Msg: TMessage);
      virtual wm_first + wm_CellEscaped;
  procedure MoveEditCellRel(s : PChar); virtual;
  procedure MoveEditCell (i,j : integer); virtual;
  function CanClose: Boolean; virtual;
  function CellEval(i, j: integer): integer; virtual;
function CellStr (i, j: integer): PChar; virtual;
procedure CellStore (i, j: integer; r: integer); virtual;
procedure Specs(var Msg: TMessage);
  virtual cm_first + cm_specs;
procedure ScalarMult (var Msg: TMessage);
  virtual cm_first + cm_scalarMult;
end;

{ MDI Window }--------------------------------------------

PMatrixMDIWindow = `TMatri
cMDIWindow = object(TMDIWindow)
  MatrixNames: PStrCollection;
  childCount: integer;
constructor Init(ATitle: PChar);
procedure SetupWindow; virtual;
function CreateChild: PWindowsObject; virtual;
procedure UpdateChildList(var C: PStrCollection);
procedure BinaryOpsDlg (var Msg: TMessage);
  virtual cm_first + cm_BinaryOps;
procedure BinaryOps (index1, index2, index3: integer; op: PChar); virtual;
procedure AddMatrices (pL, pR, pT: PMatrixMDIChild); virtual;
procedure TimesMatrices (pL, pR, pT: PMatrixMDIChild); virtual;
procedure MinusMatrices (pL, pR, pT: PMatrixMDIChild); virtual;
end;

{************************************************************************
{ Specs Dialog Methods
{************************************************************************
procedure TSpecsDialog.OK (var Msg: TMessage);
begin
  TDialog.ok(msg);
end;

{************************************************************************
{ Binary Operations Dialog Box
{************************************************************************
procedure TOpsDialog.TrapPlusButton(var Msg: TMessage);
var
  opstr: PChar;
begin
  Opstr := `PLUS';
  SendDlgItemMsg(id_opStatic,wm_settext,0,Longint(Opstr));
procedure TOpsDialog.TrapTimesButton(var Msg:TMessage);
var
  opstr : PChar;
begin
  Opstr := 'TIMES';
  SendDlgItemMsg(id_opStatic,WIDLsettext,O,Longint(Opstr));
end;

procedure TOpsDialog.TrapMinusButton(var Msg:TMessage);
var
  opstr : PChar;
begin
  Opstr := 'MINUS';
  SendDlgItemMsg(id_opStatic,WIDLsettext,O,Longint(Opstr));
end;

constructor TCell.init(r : integer);
begin
  e := r;
end;

procedure TCell.print;
var
  wstring : string;
begin
  write(e,' ');  
end;

procedure TCellWindow.dataEntry (var Msg :TMessage);
begin
  {--- user terminated cell edit by pressing the return key ---}
  if msg.wParam = vk_return then
begin
  SetFocus(Parent^.HWindow);
  SendMessage(Parent^.HWindow, wm_CellReturned, 0, 0);
end;

{--- user terminated cell edit by pressing the return key ---} if msg.wParam = vk_Escape then
begin
  SetFocus(Parent^.HWindow);
  SendMessage(Parent^.HWindow, wm_CellEscaped, 0, 0);
end;
end;

{***********************************************************************}
{ MDI CHILD WINDOW METHODS }
{***********************************************************************}
{----------------------------------------------------~--------------------}
constructor TMatrixMDIChild.Init(AParent: PWindowsObject; ChildNum: Integer);
var
  TitleStr: array[0 .. 12] of Char;
  ChildNumStr: array[0 .. 5] of Char;
  i: integer;
begin
  { assign a numbered default name to this new matrix instance }
  Str(ChildNum, ChildNumStr);
  StrCat(StrECopy(TitleStr, 'Matrix #'), ChildNumStr);
  TWindow.Init(AParent, TitleStr);

  CellPaint := true;
  { initialize a collection with 50 item pointers, and increase by 10 }
  Cells := New(PCollection, Init(50,10));
  { initialize 50 cells to 0 }
  for i := 1 to 50 do Cells^.Insert(New(PCell, Init(0)));
  New(CellWindow, Init(@Self, id_Cell, 50,40,60,25, 24,false));
end;

{ -----------------------------------------------------------------------}
procedure TMatrixMDIChild.SetupWindow;
begin
  TWindow.SetupWindow;

  { these fields will be displayed (and user selectable) in a later version }
  Name := 'a matrix name';
Description := 'a matrix descrip';
TopLabel := 'top label';
SideLabel := 'side label';

{ default current cell top left }
CurrRow := 1;
CurrCol := 1;

{ initialize to 4 by 4 matrix }
cols := 4;
rows := 4;

{ top left corner of matrix in pixels }
xStart := 50;
yStart := 40;

{ default cell size in pixels }
cellwidth := 60;
cellHeight := 25;

end;

{------------------------------------------------------------------------}
procedure TMatrixMDIChild.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
var
rect : TRect;
RowPen,ColPen,OldPen : HPen;
rowY,colX,i,j,xStop,yStop : integer;
pstring : string[79];
wstring : array [0 .. 79] of char;
outstring: array [0 .. 79] of char;
cptr : PCell;
cellLen,x : integer;
begin
{message box (hWindow, 'calling child paint', 'test',mb_ok);}

{GetClientRect (HWindow, &rect);}
{ DPtoLP (hDC, (LPPOINT) &rect, 2); }

RowPen := CreatePen(ps_solid,0,RGB(0,0,128));
OldPen := SelectObject(PaintDC,RowPen);

{* draw row lines *}
xStop := xStart + (cols * cellWidth);
for i := 0 to rows do
begin
rowY := (i * cellHeight) + yStart;
MoveTo (PaintDC, xStart, rowY);
LineTo (PaintDC, xStop, rowY);
if (i < rows) then
begin
for j := 0 to (cols-1) do
begin
procedure TMatrixMDIChild.TrapKeyboard (var Msg: TMessage);
var
  CountStr: array[0..5] of Char;
  wstring, outstring : array[0..79] of char;
  akey, moveWin : boolean;
  ns : PChar;
begin
  akey := false;
  moveWin := false;

  if msg.wParam = vk_right then
    MoveEditCellRel(´right´)
  else if msg.wParam = vk_left then
    MoveEditCellRel(´left´)
  else if msg.wParam = vk_up then
    MoveEditCellRel(´up´)
  else if msg.wParam = vk_down then
    MoveEditCellRel(´down´)
end;

var
  akey : boolean;
  xStart, cellWidth, cellLen : Integer;
  yStart, rowY, colX, colY, i, j, CellEval, CellStr : integer;
  wstring, outstring, oldstring : array[0..79] of Char;
begin
  akey := false;

  if msg.wParam = vk_up then
  MoveEditCellRel(´up´)
  else if msg.wParam = vk_down then
    MoveEditCellRel(´down´)
  else if msg.wParam = vk_left then
    MoveEditCellRel(´left´)
  else if msg.wParam = vk_right then
    MoveEditCellRel(´right´)
end;
MoveEditCellRel('left')

else if msg.wParam = vk_up then
    MoveEditCellRel('up')

else if msg.wParam = vk_down then
    MoveEditCellRel('down')

{else if msg.wParam = vk_return then
    begin
        Str(CellEval(CurrRow,CurrCol):5,wstring);
        StrPCopy(outstring,wstring);
        SetFocus(CellWindow~.hWindow);
        CellWindow~.Clear;
        CellWindow~.SetText(outstring);
    end}

else if ( (msg.wParam >= ord('0')) and
           (msg.wParam <= ord('9')) ) then
    begin
        case msg.wParam of
            ord('0') : ns := '0';
            ord('1') : ns := '1';
            ord('2') : ns := '2';
            ord('3') : ns := '3';
            ord('4') : ns := '4';
            ord('5') : ns := '5';
            ord('6') : ns := '6';
            ord('7') : ns := '7';
            ord('8') : ns := '8';
            ord('9') : ns := '9';
        end; { case }
        CellWindow~.Clear;
        CellWindow~.SetText(ns);
        SetFocus(CellWindow~.hWindow);
    end

else if ( (msg.wParam >= vk_NumPad0) and
           (msg.wParam <= vk_NumPad9 ) ) then
    begin
        case msg.wParam of
            vk_NumPad0 : ns := '0';
            vk_NumPad1 : ns := '1';
            vk_NumPad2 : ns := '2';
            vk_NumPad3 : ns := '3';
            vk_NumPad4 : ns := '4';
            vk_NumPad5 : ns := '5';
            vk_NumPad6 : ns := '6';
            vk_NumPad7 : ns := '7';
            vk_NumPad8 : ns := '8';
            vk_NumPad9 : ns := '9';
        end; { case }
        {StrPCopy(outstring,ns);} CellWindow~.Clear;
CellWindow^ SetText (ns);
    SetFocus (CellWindow^ hWindow);
end;
end;

{------------------------------------------------------------------------}
{ response method for user-defined message id_CellReturned...  
  the user pressed the return key to exit editing of the current cell  }
procedure TMatrixMDIChild.TrapReturn (var Msg : TMessage);
var
    data : array[0..23] of char;
    r    : integer;
    err  : integer;
begin
    CellWindow^ . GetText(@data,23);
    Val(data,r,err);
    if err = 0 then { val reported a successful conversion }
        begin
          CellStore(Currrow,Currcol,r);
          MoveEditCellRel('right');
          SetFocus(CellWindow^ . HWindow);
          CellWindow^ . Clear;
          CellPaint := false;
        end
    else { val complained ... invalid data } 
        begin
          messagebox(HWindow, data, `invalid entry`, mb_ok);
          SetFocus(HWindow);
        end;
end;

{------------------------------------------------------------------------}
{ response method for user-defined message id_CellEscaped...  
  the user pressed the escape key to exit editing of the current cell  }
procedure TMatrixMDIChild.TrapEscape (var Msg : TMessage);
var
    wstring,outstring : array[0..79] of char;
begin
    { reject cellwindows contents...replace with cell's current contents }
    Str(CellEval(CurrRow,CurrCol),wstring);
    StrPCopy(outstrins,wstrins);
    CellWindow^ . SetText (outstring);
end;

{------------------------------------------------------------------------}
procedure TMatrixMDIChild.MoveEditCellRel (s : PChar);
var
    moveWin : boolean;
begin
    moveWin := false;
if strcomp(s, 'right') = 0 then
    begin
        moveWin := true;
        if currCol < cols then
            currCol := currCol + 1
        else
            begin
                currCol := 1;
                if currRow < rows then
                    currRow := currRow + 1
                else
                    currRow := 1;
            end;
    end;

if strcomp(s, 'left') = 0 then
    begin
        moveWin := true;
        if currCol > 1 then
            currCol := currCol - 1
        else
            begin
                if currRow > 1 then
                    currRow := currRow - 1
                else
                    currRow := rows;
                    currCol := cols;
            end;
    end;

if strcomp(s, 'up') = 0 then
    begin
        moveWin := true;
        if currRow > 1 then
            currRow := currRow - 1
        else
            begin
                if currCol > 1 then
                    currCol := currCol - 1
                else
                    currCol := cols;
                    currRow := rows;
            end;
    end;

if strcomp(s, 'down') = 0 then
    begin
        moveWin := true;
        if currRow < rows then
            currRow := currRow + 1
        else
            begin
                if currCol < cols then
                    currCol := currCol + 1
                else
                    currCol := currCol - 1
            end;
    end;
currcol := currcol + 1
else
  currcol := 1;
currRow := 1;
end;

if moveWin then
  MoveEditCell(CurrRow,CurrCol);
end;

{------------------------------------------------------------------------}
Procedure TMatrixMDIChild.MoveEditCell(i,j:integer);
var
  wstring,outstring : array[0..79] of char;
begin
  currRow := i;
currCol := j;
  MoveWindow (CellWindow^.hWindow, cellwidth * (currCol - 1) + xStart, cellHeight * (currRow - 1) + yStart, cellWidth, cellHeight, True);
  Str(CellEval(CurrRow,CurrCol),wstring);
  StrPCopy(outstring,wstring);
  CellWindow^.SetText (outstring);
end;

{------------------------------------------------------------------------}
{ MATRIX CELL REFERENCE ROUTINES }
{------------------------------------------------------------------------}
{ allows reference to the matrix in the conventional two dimensional manner...returns the integer value of the referenced cell }
function TMatrixMDIChild.CellEval(i,j : integer) : integer;
var
  index : integer;
cptr : PCell;
begin
  index := ((i-1)*cols) + (j-1);
cptr := cells^.at(index);
  CellEval := cptr^.e;
end;

{------------------------------------------------------------------------}
{ allows reference to the matrix in the conventional two dimensional manner...updates the integer value of the referenced cell }
procedure TMatrixMDIChild.CellStore(i,j : integer;r : integer);
var
  index : integer;
cptr : PCell;
begin
  index := ((i-1)*cols) + (j-1);
cptr := cells`.at(index);
cptr`.e := r;
end;

{allows reference to the matrix in the conventional two dimensional manner...returns the string value of the referenced cell}
function TMatrixMDIChild.CellStr(i, j : integer) : PChar;
  var
tempstring, retstring : array[0..79] of char;
begin
  Str(CellEval(i,j):5,tempstring);
  StrPCopy(retstring,tempstring);
  CellStr := retstring;
end;

{------------------------------------------------------------------------}
{ UNARY MATH OPERATIONS }
{------------------------------------------------------------------------}
{ multiplies a matrix by a scalar }
procedure TMatrixMDIChild.ScalarMult (var Msg : TMessage);
  var
    inputText: array[0..5] of char;
i, error, scalar : integer;
cptr : PCell;
begin
  Str(1,inputText);
  if application^.ExecDialog(new(PInputDialog,
      Init(@Self, 'Scalar Multiply', 'Enter a scalar:',
        InputText, SizeOf(InputText))) = id_OK then
  begin
    Val(InputText, scalar, error);
    if error = 0 then
      for i := 1 to (rows*cols) do
      begin
        cptr := cells`.at(i);
        cptr`.e := cptr`.e * scalar;
      end;
    InvalidateRect(HWindow,nil,true);
  end;
end;

{-----------------------------------------------------------------------}
{ SETTINGS DIALOG BOX }
{-----------------------------------------------------------------------}
{ Settings Dialog Box; currently only allows setting the number of rows and columns of a matrix }
procedure TMatrixMDIChild.Specs(var Msg : TMessage);
  var
D : PDialog;
E : PEdit;
s1,s2 : array[0..32] of char;
err,retValue : integer;
specsRecord : TransferSpecsRecord;

begin

{ setup transfer record }
str(rows:2,specsRecord.NumRows);
str(cols:2,specsRecord.NumCols);

{ initialize and execute specs dialog resource }
D:= New(PSpecsDialog,Init(@Self,"Specs"));
New(E, InitResource(D, 1101, SizeOf(specsRecord.NumRows)));
New(E, InitResource(D, 1102, SizeOf(specsRecord.NumCols)));
D^.TransferBuffer := @SpecsRecord;
retvalue := Application^.ExecDialog(D);

{ user clicked id_ok / pressed return key }
if retvalue = id_OK then
begin

{ update the matrix object fields }
Val(specsRecord.NumRows,rows,err);
Val(specsRecord.NumCols,cols,err);
CurrRow := 1;
CurrCol := 1;
{ make sure Windows repaints the matrix window }
InvalidateRect(HWindow,nil,true);
end;

end;

------------------------------
{ CanClose will be used in a later version in support of file operations }
function TMatrixMDIChild.CanClose;
begin
  CanClose := true;
end;

{************************************************************************
{ MDI CLIENT WINDOW METHODS }
************************************************************************}

{------------------------------------------------------------------------}
constructor TMatrixMDIWindow.Init(Atitle : PChar);
begin
  TMDIWindow.Init ("MatrixCad", LoadMenu(HInstance, "MDIMENU");
  {Attr.X := 0;
Attr.Y := 0;
Attr.W := 640;
Attr.H := 300;
Attr.Style := ws_Overlapped or ws_SysMenu or ws_MinimizeBox;
MatrixNames := New(PStrCollection,Init(10,5));
end;

{ SetupWindow creates the first MDI child }
procedure TMatrixMDIWindow.SetupWindow;
var
  ARect: TRect;
  NewChild: PMatrixMDIChild;
begin
  TMDIWindow.SetupWindow;
  CreateChild;
end;

{ Create a new MDI child }
function TMatrixMDIWindow.CreateChild: PWindowsObject;
var
  ChildNum: Integer;

  function NumberUsed(P: PMatrixMDIChild): Boolean; far;
  begin
    NumberUsed := ChildNum = P^.Num;
  end;

begin
  ChildNum := 1;
  while FirstThat(@NumberUsed) <> nil do Inc(ChildNum);
  CreateChild := Application^.MakeWindow(New(PMatrixMDIChild,
    Init(@Self, ChildNum)));
end;

{ returns a list of the names of all matrices }
procedure TMatrixMDIWindow.UpdateChildList(var C : PStrCollection);

  procedure GetAChild(AChild: PMatrixMDIChild); far;
  begin
    C^.Insert(StrNew(AChild^.attr.title));
  end;

begin
  C^.FreeAll; {clear out the collection for updating}
  ForEach(@GetAChild);
end;
procedure TMATRIXMDIWindow.BinaryOpsDlg (var Msg : TMessage);

var
    D : PDialog;
    S : PStatic;
    LL,RL,TL : PListBox;
    s1,s2 : array[0..32] of char;
    i,err,returnValue : integer;
    opsRecord : TransferopsRecord;

procedure UpdateChildren(p : pMatrixMDIChild); far;
begin
    InvalidateRect(p^.HWindow,nil,true);
end;

begin
    with opsRecord do
    begin
        { initialize matrix name lists }
        LOpList := New(PStrCollection,Init(10,5));
        ROpList := New(PStrCollection,Init(10,5));
        TOpList := New(PStrCollection,Init(10,5));

        UpDateChildList(LOpList);
        UpDateChildList(ROpList);
        UpDateChildList(TOpList);

        StrPCopy(operation,'PLUS');
        Lindex := 0;
        Rindex := 0;
        Tindex := 0;
    end;

    { initialize and execute dialog box }
    D:= New(POpsDialog,Init(@Self,'BINARY OPS'));
    New( S, InitResource(D, 2301, SizeOf(opsRecord.operation));
    New(LL, InitResource(D, 2101));
    New(RL, InitResource(D, 2103));
    New(TL, InitResource(D, 2105));
    D^.TransferBuffer := @opsRecord;
    returnValue := Application^.ExecDialog(D);

    if returnValue = id_OK then
    begin
        with opsrecord do
        begin
            BinaryOps(Lindex,Rindex,Tindex,operation);
            ForEach(@UpdateChildren);
        end;
    end;
end;
procedure TMatrixMDIWindow.BinaryOps(index1,index2,index3 : integer; op : PChar);
var
  i : Integer;
  pL,pR,pT : PMatrixMDIChild;
  s : PChar;

  {-- locates the left operand child window pointer --}
  function FindLOp(AChild: PMatrixMDIChild): boolean; far;
  begin
    FindLOp := (i = index1);
    i := i + 1;
  end;

  {-- locates the right operand child window pointer --}
  function FindROp(AChild: PMatrixMDIChild): boolean; far;
  begin
    FindROp := (i = index2);
    i := i + 1;
  end;

  {-- locates the target operand child window pointer --}
  function FindTOp(AChild: PMatrixMDIChild): boolean; far;
  begin
    FindTOp := (i = index3);
    i := i + 1;
  end;

begin
  i := 0;
  pL := PMatrixMDIChild(firstthat(@FindLOp));

  i := 0;
  pR := PMatrixMDIChild(firstthat(@FindROp));

  i := 0;
  pT := PMatrixMDIChild(firstthat(@FindTOp));

  if ((pL<>nil) and (pR<>nil) and (pT<> nil)) then
    if strcomp(op, 'PLUS') = 0 then
      AddMatrices(pL,pR,pT)
    else if strcomp(op, 'TIMES') = 0 then
      TimesMatrices(pL,pR,pT)
    else if strcomp(op, 'MINUS') = 0 then
      MinusMatrices(pL,pR,pT);
end;
procedure TMatrixMDIWindow.TimesMatrices(pL,pR,pT : PMatrixMDIChild);
var
  pLx,pRx,pTx : pcell;
  i,j,k : integer;
  sum : integer;

  { get the value of the i,j referenced cell in a collection of cells }
function gcv (p : pCollection; i,j,cols : integer) : integer;
var
  index : integer;
  cptr : pcell;
begin
  index := ((i-1)*cols) + (j-1);
  cptr := p^.at(index);
  gcv := cptr^.e;
end;

  { put the value of the i,j referenced cell in a collection of cells }
procedure pcv (p : pCollection; i,j,cols : integer; putval : integer);
var
  index : integer;
  cptr : pcell;
begin
  index := ((i-1)*cols) + (j-1);
  cptr := p^.at(index);
  cptr^.e := putval;
end;
begin
  for i := 1 to pL^.rows do
    for j := 1 to pR^.cols do
      begin
        sum := 0;
        for k := 1 to pL^.cols do
          begin
            sum := sum + (gcv(pL^.cells,i,k,pL^.cols) *
                            gcv(pR^.cells,k,j,pR^.cols));
            pcv(pT^.cells,i,j,pR^.cols,sum);
          end;
        pT^.rows := pL^.rows;
        pT^.cols := pR^.cols;
      end;
end;

{ subtract two matrices...put the result into the specified target; }
{ NOTE: the left operand controls the row/cols extent of the subtraction }
procedure TMatrixMDIWindow.MinusMatrices(pL,pR,pT : PMatrixMDIChild);
var
  pLx,pRx,pTx : pcell;
  i : integer;
begin
  for i := 0 to ((pL^-rows * pL^-cols) - 1) do
    begin
      pLx := pL^-cells^-at(i);
      pRx := pR^-cells^-at(i);
      pTx := pT^-cells^-at(i);
      pTx^-e := pLx^-e - pRx^-e;
    end;
  end;
end;

{ add two matrices...put the result into the specified target;
  NOTE: the left operand controls the row/cols extent of the addition }procedure TMatrixMDIWindow.AddMatrices (pL,pR,pT : PMatrixMDIChild);
var
  pLx,pRx,pTx : pcell;
  i : integer;
begin
  for i := 0 to ((pL^-rows * pL^-cols) - 1) do
    begin
      pLx := pL^-cells^-at(i);
      pRx := pR^-cells^-at(i);
      pTx := pT^-cells^-at(i);
      pTx^-e := pLx^-e + pRx^-e;
    end;
end;

{************************************************************************}{ APPLICATION METHODS }{************************************************************************}{ Construct a main window object }procedure TMatrixMDIApp.InitMainWindow;
begin
  MainWindow := New(PMatrixMDIWindow, Init(´MatrixCad´));
end;

{************************************************************************}{ MAIN MODULE }{************************************************************************}var
  MatrixMDIApp: TMatrixMDIApp;
begin
  MatrixMDIApp.Init(´MatrixCad´);
  MatrixMDIApp.Run;
  MatrixMDIApp.Done;
end.
SOUTHERN ILLINOIS UNIVERSITY

MATHEMATICAL GRAPHICS OBJECTS IN MTXCAD 1.1 FOR WINDOWS 3.0

UNIVERSITY HONORS 499

ACADEMIC ADVISOR: DR. MARK BEINTEMA

TERRY D. HAWKINS

DECEMBER 11, 1991
For the last several decades, structured computer programming has been an important concept in software engineering. However, in the last few years a new paradigm called Object-Oriented Programming (OOP) has quickly become accepted by many as a better way to productively handle the complexity of modern computer programs.

Because of this trend, along with the explosion of interest in graphical user interfaces (GUI), my objectives for this project were to further explore the Object-Oriented Programming (OOP) paradigm under MicroSoft's GUI Windows 3.0. The vehicle for this exploration was a mathematical matrix graphics program, called MtxCad 1.1. This project was based on a similar previous project which provided simple matrix operations. Noting the synergy between matrix theory and graph theory, this project extended that by creating a new mathematical graphics object type.

MtxCad 1.1 provides tools that allow a user to easily draw a graph using a mouse within a Multiple Document Interface (MDI) window. Several ideas in MtxCad's user interface were borrowed from current PC Computer-Aided-Design (CAD) technology. In 'point mode', new graph nodes can be set to snap to a displayable rectangular grid, which can be toggled on and off. In 'edge mode', new graph edges can be stretched like rubber-bands between any two nodes, with automatic capture by the nearest node.

Several mathematical tests can be made on a graph. The
existence of a Euler path can be quickly determined. MtxCad can also find the shortest path between the first and last nodes in the graph, displaying the length of the shortest path as well as the path itself.

The software tools used to create MtxCad 1.1 were Borland’s Turbo Pascal For Windows 3.0 Version 1.0 (TPW), along with Borland’s Resource Workshop Version 1.0 (RW). TPW provides a fully integrated environment under Windows which compares favorably with the DOS version of Turbo Pascal, featuring seamless editing, compiling, linking, and testing. RW allows easy resource development with comprehensive project management. When considering the high complexity of the Windows programming environment, integrated tools such as these are essential to achieving a reasonable level of productivity.

II. OBJECT-ORIENTED PROGRAMMING: CONCEPTS

Besides its inherent complexity, development in MicroSoft Windows is synergistic with the OOP paradigm because Windows also implements process multitasking by using inter-process message passing. Similarly, the OOP paradigm says that objects are never ‘called’ like in conventional programs, but that they respond to ‘messages’. For example, instead of a program initializing it’s objects, it sends messages telling them to initialize themselves. The program therefore doesn’t need to know anything at all about the details of initialization. This characteristic is very useful in promoting the modularity and re-usability of code.
The primary component of the OOP paradigm is, appropriately enough, the object type. Also commonly referred to as a class, it defines an object's data fields and methods. Declared much like a traditional Pascal record, an object type is a definition for an object, not the object itself. A program therefore declares 'instances' of object types.

The object differs from a record in that, in addition to data fields, operations are defined on those fields which describe the actions that the object 'knows' how to do. A method, in other words, is a procedure or function definition in an object class. An object performs these operations upon itself in response to the appropriate messages. This binding of both the data fields and the method definitions is called 'Encapsulation'.

Another hallmark of OOP is called inheritance. The objects in a program are related in a hierarchical fashion. Objects can inherit the properties (ie. the data field and methods) of objects higher in the hierarchical tree. An 'ancestor' is an object from which another object is descended. Turbo Pascal allows a descendent to have only one ancestor, although ancestor objects may have any number of descendants. A descendant has access to all the data fields and methods of it's ancestors, and can redefine those definitions as well as add new ones of it's own. It is this property of inheritance that allows the reusability of code.

III. IMPLICATIONS OF USING BORLAND'S OBJECTWINDOWS ON IMPLEMENTATION OF GRAPH OBJECT DATA STRUCTURES.
With TPW, Borland provides a class library called ObjectWindows, from which all of MtxCad’s objects were descended from. The class TCollection had the most direct impact on the design of the program, having been used to keep track of nodes as well as simulate an adjacency matrix for edges along with weights. The manipulation of matrices normally requires the declaration and use of multi-dimensional arrays. MtxCad used collections almost exclusively instead of arrays. Specifically, a collection is an object that stores a group of pointers and provides a host of methods for manipulating them, such as item insertion, deletion, and searching. Collection pointers are untyped so that, unlike arrays, they can point to any type of data structure.

Collections have two unique features compared to traditional Pascal arrays: dynamic allocation, and polymorphism. Dynamic allocation allows collections, even though initialized to a specific size, to grow at run time to accommodate new data stored into them. Memory then only needs be allocated when it is actually needed, which is important in a multitasking environment like Windows.

Collections are also useful in using polymorphism; they can contain objects of different types which may be unknown at compile time. Since each object knows how to perform operations on itself, collection behavior depends wholly on the type of objects it contains. In normal Pascal arrays, all array elements must be of the same type, and each type must be determine at compile time. Collections accomplish this because they are essentially dynamic
arrays of untyped pointers that can not only point to atomic elements and record structures, but any kind of defined object instances as well.

This kind of flexibility provides a power that should be carefully used. It's a good idea to have all objects in a given collection to have at least one abstract ancestor object in common. Some of TCollection's methods expect to work on TObject-derived instances; so it's advisable to use only those type of objects in TCollections. In practice, this is not a significant limitation. Another caution; if you put mixed types in a collection, be careful, or you can create some very hard to find bugs.

IV. SHORTEST PATH ALGORITHM

MtxCad currently computes the shortest path between the first and last nodes defined in the graph. This algorithm assumes that we have a simple, weighted, connected graph where the all the weights are positive.

We start out with a set of nodes which initially only contains the starting node, implemented here by setting a boolean value.

InNodes[startNode] := true;

We scan all nodes other than the starting node to determine the edge weights from the starting node.

for i := 1 to NodeCount do
  begin
    if i <> startnode then

begin

Scan the edge adjacency matrix for the element associated with the current pair of nodes.

\[ \text{cptr} := \text{AdjMtxFind} (\text{startNode}, i); \]

If the number of edges is greater than 0 between the two current edges, then record the weight for this edge, and set the corresponding s element, since this edge is considered the shortest distance between the two current nodes.

\[ \text{if } \text{cptr}^e > 0 \text{ then} \]

\[ \text{begin} \]

\[ \text{weights}[i] := \text{cptr}^\text{weight}; \]

\[ s[i] := \text{startNode}; \]

\[ \text{end}; \]

\[ \text{end}; \]

The following while loop is where the algorithm does its real work. Note that InNodes grows as the algorithm proceeds. At any given time InNodes contains every node whose shortest path from startNode, using only nodes in InNodes, has so far been determined. For every node \( z \) outside of InNodes, we keep track of the shortest distance \( d(z) \) from startNode to that node, using a path whose only non-InNodes node is \( z \). In addition, we keep track of the node adjacent to \( z \) on this path, \( s(z) \).

To determine which node should be next moved into InNode, we pick the non-InNode node with the smallest weight(i), or distance; then we have to recompute the weights for all the remaining nodes...
not in InNodes, because there might be a shorter path from \( x \) going through \( p \) than there was before \( p \) belonged to InNodes. And if so, then \( s(z) \) must be updated so that \( p \) is now shown to be the node adjacent to \( z \) on the current shortest path. Note that the algorithm terminates when \( y \) is put into InNodes, even though there may be other nodes in the graph not yet in InNodes.

\[
\text{while (not InNodes[endNode]) do}
\begin{align*}
p &:= \text{smallestWeight;} \\
\text{InNodes}[p] &:= \text{true;} \\
\text{for } z := 1 \text{ to NodeCount do} \\
&\begin{align*}
\text{if not InNodes}[i] &\text{ then} \\
&\begin{align*}
\text{oldWeight} &:= \text{weights}[z]; \\
cptr &:= \text{AdjMtxFind}(p,z); \\
\text{weights}[z] &:= \\
\text{minweight}(\text{weights}[z],\text{weights}[p],cptr^.weight) \\
\text{if weights}[z] \neq \text{oldWeight then} \\
\text{s}[z] &:= p;
\end{align*}
\end{align*}
&\end{align*}
\end{align*}
\]

The length of the shortest path from between the two nodes will be found as the value of the endnode'\text{th} element of the weights array.

\[
\text{Int2PChar(weights[endNode]);}
\]
MessageBox(HWND, pcharBuffer, 'The length of the shortest path is:', mb_OK);

The nodes of the shortest path are found by looking at y, s(y), s(s(y)), etc until we have traced the path back to x. These nodes are stored within the object collection PathNodes for display purposes. Note the use of the methods FreeAll and Insert; these methods were inherited from TCollection.

PathNodes^.FreeAll;
i := endnode;
errortrap := 0;
PathNodes^.Insert(Nodes^.At(i-1));
repeat
begin
  i := s[i];
  PathNodes^.Insert(Nodes^.At(i-1));
  Inc(errortrap);
end;
until ((i = startnode) or (errortrap > NodeCount));

It can be proven that no shorter path exists.

IV. EULER PATH

It is known that a Euler path exists in a connected graph if and only if there are no odd degree nodes or there are two odd
degree nodes. The Euler Path algorithm uses this fact by counting the number of nodes adjacent to each node and determining whether that number is odd or even. If the number of odd degree nodes is zero or two, then an Euler path must exist.

procedure MtxGraph.IsEulerPath (var Msg:TMessage);

var
  odd,i,degree,j : integer;
  cptr : PCell;
  connected : boolean;

begin
  odd := 0;
  i := 0;

  Check that this graph is connected.
  connected := (Cells^.Count > 0);

  Loop through all the nodes until path determination can be made.
  while ((odd<=2) and (i<nodeCount) and connected) do
    begin
      degree := 0;
      
      Scan each row of the adjacency matrix (which represents all the edges from a given node to all the others), adding up all the edges.
      
      for j := 0 to (nodeCount-1) do
        begin
          cptr := Cells^.At((nodeCount*i)+j);
          degree := degree + cptr^.e;
if degree=0 then

connected := false;

Check for odd degree.

if (degree mod 2 = 1) then odd := odd + 1;

i := i + 1;

end; {while}

V. CONCLUSIONS

MicroSoft Windows graphical user interface provides some very useful tools in developing mathematical graphics software. Of special use is the Multiple Document Interface, which allows a user to simultaneously work with several graphs and matrices. Also, a fair assortment of graphics primitives is available for drawing lines, curves, etc.

Borland's ObjectWindows allows a structured, building-block approach to building Windows applications. Collections are very powerful, but since most published algorithms use standard arrays, substantial rewriting is required to use them. This was probably the most difficult problem in creating the new graphics object.

Overall, even though the programmer's learning curve is very steep, OOP with Windows has the potential to create very maintainable and powerful interactive mathematical software.
VII. BIBLIOGRAPHY

MtxCad Graphics Object Unit

Programmer: Terry D. Hawkins

uses Utilis, mtxmsgs, mtxids, WinCrt, WObjects, WinTypes, WinProcs, Strings, StdDlgs

begin

Node = object(TObject)
    x, y : integer;
    no : integer;   {node number}
    constructor init(px, py, n: integer);
    procedure Paint(hw: HWND); virtual;
end;

Cell = object(TObject)
    e : integer;
    weight : integer;
    constructor Init(i: integer);
end;

MtxGraph = object(TWindow)
    GridOn : boolean;
    ButtonDown : boolean;
    DisplayPath : boolean;
    DC : HDC;
    OldBrush, NewBrush;
    X1, X2, Y1, Y2 : integer;
    EdgedMode : integer; {0: Point; 1: Edge}
    WeightMode : integer; {1:ON; 0:OFF}
    Nodes : PCollection;
    PathNodes : PCollection;
    Cells : PCollection;
    NodeCount : integer;
    GridMesh : integer;
    EdgeV1 : integer;
    EdgeV2 : integer;
    Vertices : integer;
    constructor Init(AParent: PWindowsObject; ATitle: PChar);
    procedure Paint(PaintDC: HDC; var PaintInfo: TPaintStruct); virtual;
    procedure PaintGrid (PaintDC: HDC; var PaintInfo: TPaintStruct); virtual;
    procedure PaintEllipse(PaintDC: HDC; var PaintInfo: TPaintStruct);
end;
procedure PaintNodes (PaintDC: HDC; var PaintInfo: TPaintStruct);  
procedure PaintEdges (PaintDC : HDC; PaintInfo : TPaintStruct);

procedure GetN(var Msg: TMessage); virtual cm_first + GetN;
procedure ReSet; virtual;

procedure GridToggle (var Msg: TMessage); virtual cm_first + Grid_Toggle;
procedure PointToggle (var Msg: TMessage); virtual cm_first + PointMode;
procedure EdgeToggle (var Msg: TMessage); virtual cm_first + EdgeMode;
procedure WeightToggle (var Msg: TMessage); virtual cm_first + ShowWeightMode;

procedure ClearEdges (var Msg: TMessage); virtual cm_first + cm_ClearEdges;
procedure ClearGraph (var Msg: TMessage); virtual cm_first + cm_ClearGraph;
procedure JoinAllNodes (var Msg: TMessage); virtual cm_first + cm_JoinAllNodes;
procedure IsEulerPath (var Msg: TMessage); virtual cm_first + cm_IsEulerPath;
procedure ShortestPath (var Msg: TMessage); virtual cm_first + cm_ShortestPath;
procedure MinSpanningTree (var Msg: TMessage); virtual cm_first + cm_MinSpanTree;

procedure WMLButtonDown (var Msg: TMessage); virtual wm_first + wm_LButtonDown;
procedure WMRButtonDown (var Msg: TMessage); virtual wm_first + wm_RButtonDown;
procedure WMMouseMove(var Msg: TMessage); virtual wm_first + wm_MouseMove;
procedure WMLButtonUp(var Msg: TMessage); virtual wm_first + wm_LButtonUp;

procedure Snap2Grid(var x,y: integer);
function NearestNode(px,py:integer): PNode;
procedure NewNode(m,n:integer);
procedure DrawRubberband;
procedure AdjMtxInit;
procedure AdjMtxInc;
function AdjMtxFind(m,n:integer): PCell;
procedure AdjMtxSet(row,acol,edges,weight:integer);

function NodeExist(px,py: integer): pNode;
procedure DeleteNode (ANode : PNode); virtual;
procedure DeleteEdges(rowcol: integer);

procedure TestInit;

end;

var
errortrap : integer; {error var used for various error trap functions}
NODE METHODS

constructor Node.init(px,py,n:integer);
begin
  x := px;
  y := py;
  no := n;
end;

procedure Node.Paint(hw :HWnd);
var
  DC : HDC;
  s : PChar;
  Radius : integer;
begin
  s := 'x';
  DC := GetDC(hw);
  Radius := 3;
  Ellipse(DC,x - Radius, y - Radius, x + Radius,y + Radius);
  ReleaseDC(hw, DC);
end;

CELL METHODS

constructor TCell.Init(i : integer);
begin
  e := i;
  weight := 0;
end;
MTXGRAPH UTILITY METHODS

Find out if a node exists at a given xy position

```pascal
function MtxGraph.NodeExist(px, py : integer) : pNode;
function Matches(ANode : PNode) : Boolean; far;
begin
  Matches := ((ANode^.x = px) and (ANode^.y = py));
end;
begin
  NodeExist := Nodes^.FirstThat(@(Matches);
end;
```

Called whenever a node has been deleted from graph. The edges adjacency matrix is adjusted to reflect the deletion.

```pascal
procedure MtxGraph.DeleteEdges(rowcol : integer);
var
  r, s : integer;
begin
  if Cells^.Count > 0 then
  begin
    { get rid of row }
    s := (NodeCount+1)*(rowcol-1);
    for n := 1 to (NodeCount+1) do
      Cells^.AtDelete(s);

    { get rid of column }
    s := rowcol-1;
    for n := 0 to (NodeCount-1) do
      Cells^.AtDelete(s + (NodeCount*n));
  end;
end;
```

Delete a specified node

```pascal
procedure MtxGraph.DeleteNode (ANode : PNode);
var
  i : integer;
begin
  if NodeCount > 0 then
  begin
    i := Nodes^.IndexOf(ANode);
```
if i > -1 then
begin
NodeCount := NodeCount - 1;
DeleteEdges(i+1);
Nodes+A.AtFree(i);
end;
end;

Find the nearest node to a given xy screen position
-----------------------------------------------
function MtxGraph.NearestNode(px,py:integer) : PNode;
var
n : PNode;
end, dist : real;

procedure NextNode(ANode: PNode); far;
begin
r := Sqrt(abs(px-Anode^x)+abs(py-Anode^y));
if n=nil then
begin
n := Anode;
dist := r;
end
else
if r<dist then
begin
dist := r;
end
n := Anode;
end;

begin
n := nil;
dist := 10000.0;
Nodes^_Foreach(@NextNode);
NearestNode := n;
end;

************************************************************************
************************************************************************
MTXGRAPH INITIALIZATION METHODS
************************************************************************
************************************************************************
}

constructor MtxGraph.Init(AParent: PWindowsObject; ATitle: PChar);
var
I: Integer;
RanAngle: Integer;
Ran: Real;

Initialize a window of type MtxGraph
-------------------------------------


begin
TWindow.Init(AParent, ATitle);

GridOn := True;
ButtonDown := False;
DisplayPath := False;

E Mode := 0;
WeightMode := 0;
NodeCount := 0;
GridMesh := 20;

Nodes := New(PCollection, Init(20, 10));
PathNodes := New(PCollection, Init(20, 10));
Cells := New(PCollection, Init(200, 10));

ReSet;
TestInit; }

Create a new node in the NODES Collection
------------------------------------------------------------------------
procedure MtxGraph.NewNode(m, n:integer);
var
  nptr : PNode;
begin
  NodeCount := NodeCount + 1;
  nptr := new(PNode, Init(m, n, nodeCount));
  Nodes^.Insert(nptr);
end;

Reset radial grid variables
------------------------------------------------------------------------
procedure MtxGraph.ReSet;
var
  I: Integer;
  StepAngle: Integer;
  Radians: Real;
begin
  StepAngle := 360 div Vertices;
  for I := 0 to Vertices - 1 do
  begin
    Radians := (StepAngle * I) * PI / 180;
    Points[I].x := Cos(Radians);
    Points[I].y := Sin(Radians); }
end;

*************************************************************************
*************************************************************************
MTXGRAPH MENU RESPONSE METHODS
*************************************************************************
*************************************************************************
Clear all edges from the graph

```pascal
procedure MtxGraph.ClearEdges(var Msg:TMessage);

procedure NextCell(ACell: PCell); far;
begin
    ACell^.e := 0;
end;

begin
    Cells^.ForEach(@NextCell);
    InvalidateRect(HWindow, nil, True);
end;
```

Clear all nodes and edges from the graph

```pascal
procedure MtxGraph.ClearGraph(var Msg:TMessage);

begin
    Cells^.DeleteAll;
    Nodes^.DeleteAll;
    NodeCount := 0;
    EditMode := 0;
    InvalidateRect(HWindow, nil, True);
end;
```

Create edges between all nodes in the graph

```pascal
procedure MtxGraph.JoinAllNodes(var Msg:TMessage);

fix this later to not set diagonal entries

procedure EachCell(aCell: pCell); far;
begin
    aCell^.e := 1;
end;

begin
    AdjMtxInit;
    Cells^.ForEach(@EachCell);
    InvalidateRect(HWindow, nil, True);
end;
```

Grid Toggle

```pascal
procedure MtxGraph.GridToggle(var Msg:TMessage);
begin
    if GridOn=TRUE then
        begin
            GridOn:=FALSE;
            CheckMenuItem(Attr.Menu, Grid_Toggle, mf_Unchecked);
```

end
else

begin
GridOn:=TRUE;
CheckMenuIItem(Attr.Menu,Grid_Toggle,mF_Checked);
end;
ToggleCheck(Attr.Menu,Grid_Toggle);
InvalidateRect(HWindow,nil,True);
end;

------------------------------------------------------------------------}
procedure MtxGraph.PointToggle (var Msg:TMessage);
begin
 EditMode := 0;
end;

------------------------------------------------------------------------}
procedure MtxGraph.WeightToggle (var Msg : TMessage);
begin
  if WeightMode = 1 then
    WeightMode := 0
  else
    WeightMode := 1;
  InvalidateRect(HWindow,nil,True);
end;

************************************************************************
MTXGRAPH TESTING AND DEBUGGING METHODS
************************************************************************

************************************************************************

Pre-load graph for testing of methods
************************************************************************
procedure MtxGraph.testInit;
var
  'i : integer;
begin

( insert nodes )
NewNode(150,20);
NewNode(100,100);
NewNode(200,100);
NewNode(100,200);
NewNode(200,200);
NewNode(150,250);
{ insert cells }
AdjMtxInit;
AdjMtxSet(1,2,1,3);
AdjMtxSet(1,3,1,8);
AdjMtxSet(1,4,1,4);
AdjMtxSet(1,6,1,10);
AdjMtxSet(2,1,1,3);
AdjMtxSet(2,4,1,6);
AdjMtxSet(3,1,1,8);
AdjMtxSet(3,5,1,7);
AdjMtxSet(4,1,1,4);
AdjMtxSet(4,2,1,6);
AdjMtxSet(4,5,1,1);
AdjMtxSet(4,6,1,3);
AdjMtxSet(5,3,1,7);
AdjMtxSet(5,4,1,1);
AdjMtxSet(5,6,1,1);
AdjMtxSet(6,1,1,10);
AdjMtxSet(6,4,1,3);
AdjMtxSet(6,5,1,1);
end;

************************************************************************
************************************************************************

MTXGRAPH MOUSE MESSAGE METHODS

procedure MtxGraph.WMLButtonUp(var Msg: TMessage);
procedure MtxGraph.WMButtonDown(var Msg: TMessage);
procedure MtxGraph.WMMove(var Msg: TMessage);
procedure MtxGraph.WMButtonDown(var Msg: TMessage);
procedure MtxGraph.DrawRubberband;
procedure MtxGraph.Snap2Grid(var x,y : integer);

************************************************************************
************************************************************************

procedure MtxGraph.Snap2Grid(var x,y : integer);
begin
  x := Integer(Round(x/GridMesh)*GridMesh);
  y := Integer(Round(y/GridMesh)*GridMesh);
end;

procedure MtxGraph.DrawRubberband;
begin
  MoveTo(DC,X1,Y1);
  LineTo(DC,X2,Y2);
end;

procedure MtxGraph.WMMove(var Msg: TMessage);
begin
  In

if ButtonDown then with Msg do
  begin
    DrawRubberband; { erase old line}
    with Msg do
      begin
        X2 := LParamLo;
        Y2 := LParamHi;
        DrawRubberband; { draw new line }
      end;
  end;
end;

--------------------

procedure MtXGraph.WMLButtonDown(var Msg: TMessage);
var
  x,y: integer;
  nptr: PNode;
begin
  x := Msg.LParamLo;
  y := Msg.LParamHi;
  if EditMode=0 then { point mode }
    begin
      if GridOn then
        Snap2Grid(x,y);
      nptr := NodeExist(x,y);
      if nptr=nil then begin
        NodeCount := NodeCount + 1;
        nptr := new(PNode,Init(x,y,nodeCount));
        Nodes^.Insert(nptr);
        nptr^.paint(HWindow);
      end
    end
  else
    begin
      DeleteNode(nptr);
      InvalidateRect(HWindow, nil, True);
    end;
end;

if EditMode=1 then { edge mode }
if not ButtonDown then with Msg do
  begin
    nptr := NearestNode(x,y);
    if nptr<>nil then begin
      edgeV1:= nptr^.no;
      DC := GetDC(HWindow);
      X1 := nptr^.x;
      Y1 := nptr^.y;
      X2 := LParamLo;
      Y2 := LParamHi;
      OldBrush := SelectObject(DC,GetStockObject(hollow Brush));
      SetROP2(DC,r2_Not);
      DrawRubberband;
      ButtonDown := True;
      SetCursor(LoadCursor(0,idc_Cross));
      SetCapture(HWindow);
    end;
procedure MtxGraph.WMButtonUp(var Msg: TMessage);
var
  n : PNode;
begin
  if ButtonDown then with Msg do
  begin
    DrawRubberband;
    ButtonDown := False;
    SetROP2(DC,r2_Black);
    n := nearestnode(lparamlo,lparamhi);
    if n<>nil then
    begin
      X2 := n^.x;
      Y2 := n^.y;
      EdgeV2 := n^.no;
      AdjMtxInc;
      DrawRubberband;
    end;
    SelectObject(DC,OldBrush);
    SetCursor(LoadCursor(0,idc_Arrow));
    ReleaseCapture;
    ReleaseDC(HWindow,DC);
  end;
end;

procedure MtxGraph.WMButtonDown(var Msg: TMessage);
begin
  InvalidateRect(HWindow, nil, True);
end;

************************************************************************
************************************************************************
MTXGRAPH EDGES ADJACENCY MATRIX METHODS
function MtxGraph.AdjMtxFind(m,n:integer) : PCell;
procedure MtxGraph.AdjMtxInit;
procedure MtxGraph.AdjMtxSet(arrow,acol,edges,weight:integer);
procedure MtxGraph.AdjMtxInc;
procedure MtxGraph.EdgeToggle (var Msg:TMessage);

************************************************************************
************************************************************************
procedure MtxGraph.AdjMtxInit;
i : integer;
begin
  for i := (Cells^.Count-1) to NodeCount*NodeCount do
    Cells^.Insert(New(PCell,Init(0)));
end;

union MtxGraph.AdjMtxFind(m,n:integer) : PCell;
var
  index : integer;
begin
  index := ((m-1)*NodeCount) + (n-1);
  AdjMtxFind := Cells^.at(index);
end;

procedure MtxGraph.AdjMtxSet(arrow,acol,edges,weight:integer);
var
  cptr : PCell;
begin
  cptr := AdjMtxFind(arrow,acol);
  if cptr <> nil then
    begin
      cptr^.e := edges;
      cptr^.weight := weight;
    end;
end;

procedure MtxGraph.AdjMtxInc;
var
  index : integer;
  cptr : pcell;
begin
  { assume, for now, an undirected graph }

  cptr := AdjMtxFind(EdgeV1,EdgeV2);
  cptr^.e := cptr^.e + 1;
  cptr^.weight := cptr^.e;

  cptr := AdjMtxFind(EdgeV2,EdgeV1);
  cptr^.e := cptr^.e + 1;
  cptr^.weight := cptr^.e;
end;

procedure MtxGraph.EdgeToggle (var Msg:TMessaqel);
begin
  EditMode := 1;
  {if adjacency matrix has not been initialized before, then init}
  AdjMtxInit;
end;
MTXGRAPH PAINT METHODS

procedure MtxGraph.PaintEllipse(PaintDC: HDC;
procedure MtxGraph.Paint(Paint(PaintDC: HDC;
procedure MtxGraph.PaintGrid(PaintDC: HDC;
procedure MtxGraph.PaintNodes(PaintDC: HDC;
procedure MtxGraph.PaintEdges(PaintDC : HDC; PaintInfo:TPaintStruct);

---------------------------------------------------------------------------------------------

procedure MtxGraph.Paint(PaintDC: HDC;
    var PaintInfo: TPaintStruct);
begin
    if GridOn then
        PaintGrid(PaintDC,PaintInfo);
        (PaintEllipse(PaintDC,PaintInfo));
        PaintNodes(PaintDC,PaintInfo);
        PaintEdges(PaintDC,PaintInfo);
end;

// Paint snap-to grid for the current graph
---------------------------------------------------------------------------------------------

procedure MtxGraph.PaintGrid(PaintDC: HDC;
    var PaintInfo: TPaintStruct);
    var
        TheRect: TRect;
        NewPen, OldPen: HPen;
        i,j,r,x,y : Integer;
        IncX: Integer;
    begin
        NewPen := CreatePen(ps_DOT,1,RGB(125,0,0));
        OldPen := SelectObject(PaintDC,NewPen);
        GetClientRect(HWindow,TheRect);
        IncX := TheRect.Right div GridMesh-1;
        for i := 1 to ((TheRect.Bottom div GridMesh)-1) do
            for j := 1 to ((TheRect.Right div GridMesh)-1) do
                begin
                    x := Round(GridMesh*j);
                    y := Round(GridMesh*i);
                    r := 1;
                    Ellipse(PaintDC,x - r, y - r, x + r, y + r);
                end;
        for i := 0 to (TheRect.Bottom div GridMesh) do
            begin
                MoveTo(PaintDC,0,Round(GridMesh*i));
                LineTo(PaintDC,TheRect.Right,Round(GridMesh*i));
            end;
        for i := 0 to (TheRect.Right div GridMesh) do
            begin
                MoveTo(PaintDC,Round(GridMesh*i),0);
            end;
end;
Paint all nodes for the current graph

```
procedure MtxGraph.PaintNodes(PaintDC: HDC; var PaintInfo: TPaintStruct);
  procedure PaintNextNode(ANode: PNode); far;
  begin
    ANode^:Paint(HWindow);
  end;

  begin
    Nodes^:ForEach(@PaintNextNode);
  end;
```

Paint edges for the current graph

```
procedure MtxGraph.PaintEdges(PaintDC: HDC; PaintInfo: TPaintStruct);
  var
    i,m,n,o,tx,ty : integer;
    cptr : PCell;
    mptr,nptr : PNode;
    Sstr : string[11];
    tstr : array[0..11] of char;
  begin
    {assume undirected graph, with no loops, for right now...}
    if DisplayPath then
      begin
        for i:=1 to PathNodes^:Count-1 do
          begin
            mptr := PathNodes^:at(i-1);
            nptr := PathNodes^:at(i);
            MoveTo(PaintDC,mptr^:x,mptr^:y);
            LineTo(PaintDC,nptr^:x,nptr^:y);
          end;
        DisplayPath := false;
      end
    else if Cells^:Count > 0 then
      begin
        for m := 2 to NodeCount do
          for n := 1 to (m-1) do
            begin
              cptr := AdjMtxFind(m,n);
              if cptr^:e >0 then
                begin
                  mptr := Nodes^:at(m-1);
                  nptr := Nodes^:at(n-1);
                  MoveTo(PaintDC,mptr^:x,mptr^:y);
                  LineTo(PaintDC,nptr^:x,nptr^:y);
                  if WeightMode = 1 then
                    {do something with weight}
                end;
            end;
      end
```
begin
  tx := (mpt^x + npt^x) div 2;
  ty := (mpt^y + npt^y) div 2;
  int2PChar(cptr^weight);
  TextOut(PaintDC,tx,ty,pCharbuffer,strlen(pCharbuffer));
end;

{-----------------------------------------------
Create an elliptical graph with the current settings
-----------------------------------------------}
procedure MtxGraph.PaintEllipse(PaintDC: HDC;
  var PaintInfo: TPaintStruct);

var
  CenterX,Centery,i,j : integer;
  TheRect: TRect;
  Radius,
  StepAngle: Word;
  Radians: real;

begin
  GetClientRect(HWindow,TheRect);
  CenterX := TheRect.Right div 2;
  CenterY := TheRect.Bottom div 2;
  Radius := Min(CenterX, CenterY);
  Ellipse(PaintDC,Centery - Radius, CenterY - Radius, CenterX + Radius,
          CenterY + Radius);
  for I := 0 to Vertices - 1 do
    begin
      for J := I + 1 to Vertices - 1 do begin
        MoveTo(PaintDC, CenterX + Round(Points[I].X * Radius),
               CenterY + Round(Points[I].Y * Radius));
        LineTo(PaintDC, CenterX + Round(Points[J].X * Radius),
               CenterY + Round(Points[J].Y * Radius));
      end;
    end;
end;

{-----------------------------------------------}
function GetNumber (h : pWindow; n : integer;a,b:PChar) : integer;

var
  inputText: array[0..5] of char;
  error : integer;

begin
  Strn(inputText);
  if application^ExecDialog(new(PInputDialog,
      Init(h, a,b,inputText, SizeOf(inputText))) = id_OK then
    Val(inputText,n,error);
  GetNumber := n;
end;
procedure MtxGraph.GetN (var Msg : TMessage);
begin
  Vertices := GetNumber(@Self, Vertices, '# of Vertices', 'Enter # of Vertices');
  if Vertices > 50 then
    Vertices := 50;
  Reset;
  InvalidateRect(HWindow, nil, true);
end;

procedure MtxGraph.MinSpanningTree (var Msg : TMessage);

procedure MtxGraph.IsEulerPath (var Msg: TMessage);

procedure MtxGraph.ShortestPath (var Msg : TMessage);

Determine if a Euler path exists
for the current graph

procedure MtxGraph.IsEulerPath (var Msg: TMessage);

odd, i, degree, j : integer;
cptr : PCell;
connected : boolean;
begin
  odd := 0;
i := 0;
  connected := (Cells^, Count > 0);
  while ((odd<>2) and (i<nodeCount) and connected) do
    begin
      degree := 0;
      for j := 0 to (nodeCount-1) do
        begin
          cptr := Cells^.At((nodeCount*i)+j);
          degree := degree + cptr^.e;
        end;
      if degree=0 then
        connected := false;
      if (degree mod 2 = 1) then odd := odd + 1;
i := i + 1;
    end; {while}
  if not connected then
    MessageBox(HWindow, 'Graph not connected...No Euler path exists', 'Euler Path Check', mb_OK)
  else
    if (odd <= 2) then
      MessageBox(HWindow, 'Yes, Euler path exists', 'Euler Path Check', mb_OK)
else
MessageBox(HWindow, 'No Euler path exists', 'Euler Path Check', mb_OK);
end;

Compute the shortest path for the current graph

procedure MtxGraph.ShortestPath (var Msg : TMessage);
var
  weights,s : array[1..200] of integer;
  InNodes : array [1..200] of boolean;
  i,x,p,z,oldWeight,startNode,endNode : integer;
cptr : PCell;

procedure shows;
var i : integer;
begin
  for i := 1 to 6 do
    writeln(s[i]);
    writeln;
end;

procedure showweights;
var i : integer;
begin
  for i := 1 to 6 do
    write(weights[i]);
    writeln;
end;

procedure DisplayAdjMtx;
var i,j : integer;
cptr : PCell;
begin
  for i := 1 to NodeCount do
    begin
      for j := 1 to NodeCount do
        begin
          cptr := AdjMtxFind(i,j);
          write(cptr^.weight, ' ');
        end;
      writeln;
    end;
end;

function smallestWeight : integer;
var
  i,n : integer;
begin
  n := 32767;
  smallestWeight := 1;
  for i:=1 to NodeCount do
    if ( (not INnodes[i]) and (n > weights[i]) and (weights[i] > 0)) then
      begin
        n := weights[i];
        smallestWeight := i;
      end;

function minWeight(i1,i2,i3:integer):integer;
begin
  if ((i2=0) or (i3=0)) then
    minWeight := i1
  else
    if ((i1=0) or ((i2+i3)<i1)) then
      minWeight := i2+i3
    else
      minWeight := i1;
end;

begin
  startNode := 1;
  endNode := NodeCount;
  for i := 1 to NodeCount do
    begin
      weights[i] := 0;
      s[i] := startnode;
      InNodes[i] := false;
    end;
  InNodes[startNode] := true;
  for i := 1 to NodeCount do
    begin
      if i <> startnode then
        begin
          cp: = AdjMtxFind(startNode,i);
          if cp^e > 0 then
            begin
              weights[i] := cp^weight;
              s[i] := startNode;
            end;
        end;
    end;
  while (not InNodes[endNode]) do
    begin
      p := smallestWeight;
      InNodes[p] := true;
      for z := 1 to NodeCount do
        begin
          if not InNodes[i] then
            begin
              oldWeight := weights[z];
              cp := AdjMtxFind(p,z);
              weights[z] := minWeight(weights[z],weights[p],cp^weight);
              if weights[z] <> oldWeight then
                s[z] := p;
            end;
        end;
    end;
Int2PChar(weights[endNode]);
MessageBox(HWindow,pcharBuffer, 'The length of the shortest path is: ', mb_OK);
Compute the Minimum Spanning Tree for the current graph

procedure MtxGraph.MinSpanningTree (var Msg : TMessage);
const
  unseen = maxint - 2;
var
  k, min, t, V : integer;
  val, dad : array[0..200] of integer;
  cptr : PCell;
begin
  V := NodeCount;
  for k := 1 to V do
    begin
      val[k] := -unseen;
      dad[k] := 0;
    end;
  val[0] := -(unseen+1);
  min := 1;
  repeat
    k := min;
    val[k] := -val[k];
    min := 0;
    if val[k] = unseen then
      val[k] := 0;
    for t := 1 to V do
      if val[t] < 0 then
        begin
          cptr := AdjMtxFind(k, t);
          if (cptr^.<0) and (val[t] < -cptr^.) then
            begin
              val[t] := -cptr^.;
              dad[t] := k;
            end;
          if val[t] > val[min] then min := t;
        end;
    end;
  until (i = startnode) or (errortrap > NodeCount));
  displayPath := true;
  invalidateRect(HWindow, nil, True);
  InNodeArray[1] := endNode;
  i := 2;
  while false do
    begin
      InNodeArray[i] := s[i];
      x := InNodeArray[endNode];
    end;
end;
end;
    until min = 0;
end;

nd.
MtxCad 1.1 provides addition, subtraction, and multiplication of multiple matrices with emphasis on flexibility and ease of use. It also provides a graphics window with a variety of graph creation and analysis tools.

**Purpose:** To explore the advantages of object-oriented programming techniques, in particular within the context of a graphical-user-interface, along with the effect on the development of mathematical software.

---

**File Management Notes:**

**FRAME WINDOW**
- Select a file as the current matrix file
  - ...place into title bar of frame window if successful
  - ...initializes the internal matrix file header object
  - ...uses the fileOpen dialog box

**FRAME WINDOW**
- Load a matrix from the current file
  - ...if current file, calls the loadmatrices dialog box
    - which contains a list box of matrix names (from the matrix file header object), plus the name of the current file.
    - ...if no current file, opens fileopen dialog box first
      - then if successful, calls the loadmatrices dialog box

**LOAD MATRICES DIALOG BOX**
- ...selection of matrix file
- ...selection of any or all matrices in the selected file

**CHILD WINDOW**
- Save the current matrix into the current file
  - ...uses the name of the current matrix
  - ...if the name is already in the file, that object is replaced, else the matrix is appended to the file as a new object, and the header record is updated

**CHILD WINDOW**
- Save matrix into the current file
...use the filesave dialog template
...allows changing the name of the matrix

ren MatrixCad;

{$R MTXCAD.RES}

uses MtxGrfx,MtxMsgs,MtxIds,Utils,
   WObjects,WinProcs,WinTypes,WinDos,StdDlgts,Strings;

************************************************************************}

CONSTANTS
************************************************************************}

const
  cm_CountChildren = 102;
  id_CantClose = 201;
  id_cell = 301;
{
  menu command identifiers
} cm_specs = 1001;
  cm_scalarMult = 301;
  cm_matrixPower = 302;
  cm_scalarAdd = 303;
  cm_BinaryOps = 601;
{
  file menu command identifiers
} cm_Open = 701;
  cm_New = 702;
  cm_Save = 703;
  cm_SaveAs = 704;
  cm_About = 750;
{
  help command identifiers
} cm_help = 2000;
{
  specs dialog box id's
} id_rows = 1101;
  id_cols = 1102;
{
  ops dialog id's
} id_TimesButton = 2201;
  id_PlusButton = 2202;
  id_MinusButton = 2203;
  id_PolyXButton = 2204;
  id_opStatic = 2301;

************************************************************************}

TYPE DECLARATIONS
************************************************************************}

type
{
  Application -------------------------------}
  MatrixMDIApp = object(TApplication)


procedure InitMainWindow; virtual;
end;

(*--- About Dialog Box ----------------------------------------------------------*)
PAboutDialog = ^TAboutDialog;
TAboutDialog = object(TDialog)

(*--- Matrix Specifications Dialog Object -------------------------------------*)
PSpecsDialog = ^TSpecsDialog;
TSpecsDialog = object(TDialog)
  procedure ok (var msg : TMessage);
    virtual id_First + id_OK;
end;

(*--- File Open and Matrix Selection Dialog Box --------------------------------*)
PLoadMatrixDialog = ^TLoadMatrixDialog;
TLoadMatrixDialog = object(TFileDialog)

(*--- Binary Operations Dialog Object -----------------------------------------*)
POpsDialog = ^TOpsDialog;
TOpsDialog = object(TDialog)
  procedure TrapPlusButton(var msg:TMessagel);
    virtual id_First + id_PlusButton;
  procedure TrapTimesButton(var msg:TMessagel);
    virtual id_First + id_TimesButton;
  procedure TrapMinusButton(var msg:TMessagel);
    virtual id_First + id_MinusButton;
  procedure TrapPolyXButton(var msg:TMessagel);
    virtual id_First + id_PolyXButton;
end;

(*--- Settings Dialog Box Transfer Record -------------------------------------*)
TransferSpecsRecord = record
  NumRows, NumCols : array[0..32] of Char;
end;

TransferOpsRecord = record
  operation: array[0..5] of Char;
  LOpList : PStrCollection;
  Lindex : integer;
  ROpList : PStrCollection;
  Rindex : integer;
  TOpList : PStrCollection;
  Tindex : integer;
end;

(* Cell type ---------------------------------------------------------------*)
PCell = ^TCell;
TCell = object(TObject)
  e : real;
  constructor Init(r : real);
  procedure Print; virtual;
  procedure Store(var $ : TStream); virtual;

procedure Load (var S: TStream); virtual;
end;

{-----------------------------------------------------------------------}
PScratch = ^TScratch;
TScratch = Object(TObject)
Cells : PCollection;
rows, cols : integer;
constructor Init;
procedure InsertCell(r : real); virtual;
end;

{ Cell Window type -----------------------------------------------------}
PCellWindow = ^TCellWindow;
TCellWindow = object(TEdit)
  procedure dataEntry (var Msg : TMessage);
    virtual wm_first + wm_KeyDown;
  procedure Store (var S: TStream); virtual;
  procedure Load (var S: TStream); virtual;
end;

{ MDI Child Window type --------------------------------------------------}
PMatrixMDIChild = ^TMatrixMDIChild;
TMatrixMDIChild = object(TWindow)
  ChildMsg: PStatic;
  Num: Integer;
  Name : PChar;
  Description : PChar;
  TopLabel : PChar;
  deLabel : PChar;
  Cells : PCollection;
  CellWindow : PCellWindow;
  CellPaint : boolean;
  Rows : integer;
  Cols : integer;
  CurrRow : integer;
  CurrCol : integer;
  xStart : integer;
  yStart : integer;
  cellwidth : integer;
  cellHeight : integer;

{FileName: array[0..fsPathName] of Char;}
FileName : PChar;
IsDirty, IsNewFile: Boolean;
constructor Init (AParent: PWindowsObject; ChildNum: Integer);
procedure SetupWindow; virtual;
procedure Paint (PaintDC: HDC; var PaintInfo: TPaintStruct); virtual;
procedure TrapKeyboard (var Msg: TMessage);
  virtual wm_First + wm_keydown;
procedure TrapReturn(var Msg: TMessage);
procedure MoveEditCellRel(s : PChar); virtual;
procedure MoveEditCell (i,j : integer); virtual;

function CanClose: Boolean; virtual;
function CellEval(i, j: integer) : real; virtual;
function CellStr (i, j: integer) : PChar; virtual;
procedure CellStore (i,j : integer; r : real); virtual;
procedure Specs(var Msg : TMessage);
  virtual cm_first + cm_specs;
procedure ScalarMult (var Msg : TMessage);
  virtual cm_first + cm_scalarMult;
procedure MatrixPower (var Msg : TMessage);
  virtual cm_first + cm_MatrixPower;
procedure ScalarAdd (var Msg : TMessage);
  virtual cm_first + cm_ScalarAdd;

procedure MatrixSave (var Msg : TMessage);
  virtual cm_first + cm_save;

procedure FileNew(var Msg: TMessage);
  virtual cm_first + cm_New;
procedure FileOpen (var Msg: TMessage);
  virtual cm_first + cm_Open;
procedure FileSave(var Msg: TMessage);
  virtual cm_first + cm_Save;
procedure FileSaveAs(var Msg: TMessage);
  virtual cm_first + cm_SaveAs;

procedure LoadFile;
procedure SaveFile;
procedure Load (var S: TStream); virtual;
procedure Store (var S: TStream); virtual;

{ MDI Window ----------------------------------------------------------}
PMatrixMDIWindow = ^TMatrixMDIWindow;
TMatrixMDIWindow = object (TMDIWindow)
  MatrixNames : PStrCollection;
  childCount  : integer;

constructor Init(ATitle: PChar);
procedure SetupWindow; virtual;

procedure NewGraphWin(var Msg: TMessage);
  virtual cm_first + NewGraphWin;

function CanClose : boolean; virtual;
function CreateChild: PWindowsObject; virtual;
function LoadChild: PWindowsObject; virtual;
procedure UpdateChildList(var C : PStrCollection);
procedure BinaryOpsDlg (var Msg: TMessage);
  virtual cm_first + cm_BinaryOps;
procedure BinaryOps (index1,index2,index3:integer; op : PChar); virtual;
procedure AddMatrices (pL, pR, pT : PMatrixMDIChild); virtual;
procedure TimesMatrices (pL, pR, pT : PMatrixMDIChild); virtual;
procedure MinusMatrices (pL, pR, pT : PMatrixMDIChild); virtual;
procedure PolyXMatrices (pL, pR, pT : PMatrixMDIChild); virtual;

{procedure FileOpen (var Msg: TMessage);
virtual cm_first + cm_fileOpen;}

procedure About (var Msg: TMessage);
virtual cm_first + cm_About;
end;

***********************************************************************
Stream Registration Records
***********************************************************************

const

RMatrixMDIChild: TStreamRec = (  
   ObjectType : 210;
   VmtLink : Ofs(TypeOf(TMMatrixMDIChild)^);
   Load : @TMMatrixMDIChild.Load;
   Store : @TMMatrixMDIChild.Store);

RCellWindow: TStreamRec = (  
   ObjectType : 220;
   VmtLink : Ofs(TypeOf(TCell)^);
   Load : @TCell.Load;
   Store : @TCell.Store);

TCell: TStreamRec = (  
   ObjectType : 230;
   VmtLink : Ofs(TypeOf(TCell)^);
   Load : @TCell.Load;
   Store : @TCell.Store);

************************************************************************
. Global Procedures
.************************************************************************
procedure StreamRegistration;
begin
   RegisterType(RCollection);
   RegisterType(RMatrixMDIChild);
   RegisterType(RCell);
   RegisterType(RCellWindow);
end;

************************************************************************
Specs Dialog Methods
************************************************************************

trap id_ok from specs dialog...not currently used

procedure TSpecsDialog.OK (var Msg : TMessage);
begin
   TDialog.ok(msg);
end;
Binary Operations Dialog Box

procedure TOpsDialog.TrapPlusButton(var Msg: TMessage);
var
  Opstr : PChar;
begin
  Opstr := 'PLUS';
  SendDlgItemMsg(id_opStatic,wm_settext,0,Longint(Opstr));
end;

procedure TOpsDialog.TrapTimesButton(var Msg: TMessage);
var
  Opstr : PChar;
begin
  Opstr := 'TIMES';
  SendDlgItemMsg(id_opStatic,wm_settext,0,Longint(Opstr));
end;

procedure TOpsDialog.TrapMinusButton(var Msg: TMessage);
var
  Opstr : PChar;
begin
  Opstr := 'MINUS';
  SendDlgItemMsg(id_opStatic,wm_settext,0,Longint(Opstr));
end;

procedure TOpsDialog.TrapPolyXButton(var Msg: TMessage);
var
  Opstr : PChar;
begin
  Opstr := 'POLY';
  SendDlgItemMsg(id_opStatic,wm_settext,0,Longint(Opstr));
end;

CELL METHODS

constructor TCell.init(r : real);
begin
  e := r;
end;

procedure TCell.print;
var
  wstring : string;
begin
  write(e, ' ');
end;

procedure TCell.Store(var S: TStream);
begin
  Write(e, SizeOf(e));
end;
procedure TCell.Load(var S: TStream);
begin
  S.Read(e, SizeOf(e));
end;

******************************************************************************
TSCRATCH METHODS
******************************************************************************
constructor TScratch.Init;
var
  i : integer;
begin
  Cells := New(PCollection, Init(50,10));
  for i := 1 to 50 do Cells^.Insert(New(PCell,Init(0,0)));
  rows := 0;
  cols := 0;
end;

procedure TScratch.InsertCell(r : real);
begin
  Cells^.Insert(New(PCell,Init(r)));
  messagebeep(0);
end;

******************************************************************************
CELLWINDOW METHODS
******************************************************************************
procedure TCellWindow.dataEntry (var Msg :TMessage);
begin
  {--- user terminated cell edit by pressing the return key ---}
  if msg.wParam = vk_return then
    begin
      SetFocus(Parent^.HWindow);
      SendMessage(Parent^.HWindow,wm_CellReturned,0,0);
    end;

  {--- user terminated cell edit by pressing the return key ---}
  if msg.wParam = vk_Escape then
    begin
      SetFocus(Parent^.HWindow);
      SendMessage(Parent^.HWindow,wm_CellEscaped,0,0);
    end;
end;

procedure TCellWindow.Store (var S: TStream);
begin
  TEdit.Store(S);
end;

procedure TCellWindow.Load (var S: TStream);
begin
  TEdit.Load(S);
end;

}
constructor TMatrixMDIChild.Init(AParent: PWindowsObject; ChildNum: Integer):

var
  TitleStr: array[0..12] of Char;
  ChildNumStr: array[0..5] of Char;
  i: integer;

begin

  // assign a numbered default name to this new matrix instance
  Num := ChildNum;
  Str(ChildNum, ChildNumStr);
  StrCat(StrECopy(TitleStr, 'Matrix'), ChildNumStr);
  TWindow.Init(AParent, TitleStr);

  CellPaint := true;
  { initialize a collection with 50 item pointers, and increase by 10 upon demand }
  Cells := New(PCollection, Init(50, 10));
  { initialize 50 cells to 0 }
  for i := 1 to 50 do Cells^.Insert(New(PCell, Init(0)));

  New(CellWindow, Init(@Self, id_Cell, '', 50, 40, 60, 25, 24, false));
end;

procedure TMatrixMDIChild.SetupWindow:
begin
  TWindow.SetupWindow;

  { these fields will be displayed (and user selectable) in a later version }
  Name := 'a matrix name';
  Description := 'a matrix descript';
  TopLabel := 'top label';
  SideLabel := 'side label';

  { default current cell top left }
  CurrRow := 1;
  CurrCol := 1;

  { initialize to 4 by 4 matrix }
  cols := 4;
  rows := 4;

  { top left corner of matrix in pixels }
  xStart := 50;
  yStart := 40;

  default cell size in pixels }
cellWidth := 60;
cellHeight := 25;

end;

procedure TMatrixMDIChild.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);

rect : TRect;
RowPen,ColPen,OldPen : HPen;
rowY,colX,i,j,xStop,yStop : integer;
pstring : string[79];
wstring : array [0 .. 79] of char;
outstring : array [0 .. 79] of char;
cptr : PCell;
cellLen,x : integer;

begin

{messagebox(hWindow, 'calling child paint', 'test', mb_ok);}  

{GetClientRect (hWindow, &rect);}  
{ DPtoLP lhDC, (LPPOINT) &rect, 2); }  

RowPen := CreatePen(ps_solid,0,RGB(0,0,128));  
OldPen := SelectObject(PaintDC,RowPen);  

(* draw row lines *)  
xStop := xStart + (cols * cellWidth);  
for i := 0 to rows do  
begin  
rowY := (i * cellHeight) + yStart;  
MoveTo (PaintDC, xStart, rowY);  
LineTo (PaintDC, xStop, rowY);  

if (i < rows) then  
begin  
for j := 0 to (cols-1) do  
begin  
{ cellLen := CellStr(i,j,wstring);}  
Str(CellEval(i+1,j+1):6:4,wstring);  
StrPCopy(outstring,wstring);  
{outstring := CellStr(i+1,j+1);}  
TextOut(PaintDC,(j*cellWidth)+xStart+3,rowY+3,outstring,strlen);  
end;  
end;  

ColPen := CreatePen(ps_dot,0,RGB(128,128,128));  
SelectObject(PaintDC,ColPen);  

(* draw col lines *)  
for i := 0 to cols do  
begin  
colX := (i * cellWidth) + xStart;  
MoveTo (PaintDC, colX, yStart);  
LineTo (PaintDC, colX, rowY);  
end;  

SelectObject(PaintDC,OldPen);  

}
DeleteObject(RowPen);
DeleteObject(ColPen);

{ repaint cell edit window }
if CellPaint then
begin
  Str(CellEval(CurrRow,CurrCol),wstring);
  StrPCopy(outstring,wstring);
  CellWindowA.SetText(outstring);
end
else
  CellPaint := true;
end;

------------------------------------------------------------------------
{ MATRIX CELL MOVEMENT AND EDITING ROUTINES }
------------------------------------------------------------------------
procedure TMatrixMDIChild.TrapKeyboard(var Msg: TMessage);

var
  CountStr: array[0..5] of Char;
  wstring,outstring : array[0..79] of char;
  akey, moveWin : boolean;
  ns : PChar;
begin
  akey := false;
  moveWin := false;

  if msg.wParam = vk_right then
    MoveEditCellRel('right')
  else if msg.wParam = vk_left then
    MoveEditCellRel('left')
  else if msg.wParam = vk_up then
    MoveEditCellRel('up')
  else if msg.wParam = vk_down then
    MoveEditCellRel('down')
  (else if msg.wParam = vk_return then
    begin
      Str(CellEval(CurrRow,CurrCol),5,wstring);
      StrPCopy(outstring,wstring);
      SetFocus(CellWindowA.hWindow);
      CellWindowA.Clear;
      CellWindowA.SetText(outstring);
    end)
  else if ( (msg.wParam >= ord('0')) and
    (msg.wParam <= ord('9'))) then
  begin
    case msg.wParam of
    ord('0') : ns := '0';
    ord('1') : ns := '1';
    ord('2') : ns := '2';
    ord('3') : ns := '3';
    ord('4') : ns := '4';
    ord('5') : ns := '5';
    case msg.wParam of
    ord('0') : ns := '0';
    ord('1') : ns := '1';
    ord('2') : ns := '2';
    ord('3') : ns := '3';
    ord('4') : ns := '4';
    ord('5') : ns := '5';
    ord('9') : ns := '9';
end;
end;
ord ( '6' ) := '6';
ord ( '7' ) := '7';
ord ( '8' ) := '8';
ord ( '9' ) := '9';

end; { case }
CellWindow^Clear;
CellWindow^SetText (ns);
SetFocus (CellWindow^hWindow);

else if ((msg.wParam >= vk_NumPad0) and
          (msg.wParam <= vk_NumPad9)) then
  begin
    case msg.wParam of
      vk_NumPad0 : ns := '0';
      vk_NumPad1 : ns := '1';
      vk_NumPad2 : ns := '2';
      vk_NumPad3 : ns := '3';
      vk_NumPad4 : ns := '4';
      vk_NumPad5 : ns := '5';
      vk_NumPad6 : ns := '6';
      vk_NumPad7 : ns := '7';
      vk_NumPad8 : ns := '8';
      vk_NumPad9 : ns := '9';
    end; { case }
    {StrPCopy(outstring,ns);}
    CellWindow^Clear;
    CellWindow^SetText (ns);
    SetFocus (CellWindow^hWindow);
  end;
end;

{-------------------------------------------------------------------------}
{ response method for user-defined message id_CellReturned...
  the user pressed the return key to exit editing of the current cell }
procedure TMatrixMDIChild.TrapReturn (var Msg : TMessage);
var
  data : array[0 .. 23] of char;
  r    : integer;
  err  : integer;
begin
  CellWindow^getText (@data,23);
  Val (data,r,err);
  if err = 0 then { val reported a successful conversion }
    begin
      CellStore(Currrow,Currcol,r);
      MoveEditCellRel('right');
      SetFocus (CellWindow^hWindow);
      CellWindow^Clear;
      CellPaint := false;
    end
  else { val complained ... invalid data }
    begin
      messagebox(HWindow, data, 'invalid entry', mb_ok);
      SetFocus(HWindow);
    end;
end;
response method for user-defined message id_CellEscaped...
the user pressed the escape key to exit editing of the current cell

procedure TMatrixMDIChild.TrapEscape (var Msg : TMessage);

var
      string,outstring : array[0..79] of char;

begin
   { reject cellwindows contents...replace with cell's current contents }
   Str (CellEval (CurrRow,CurrCol),wstring);
   StrPCopy(outstring,wstring);
   CellWindow^.SetText (outstring);
end;

------------------------------------------------------------------------}

procedure TMatrixMDIChild.MoveEditCellRel (s : PChar);
var
   moveWin : boolean;

begin
   moveWin := false;

   if strcomp(s, 'right') = 0 then
      begin
         moveWin := true;
         if currCol < cols then
            currCol := currCol + 1
         else
            begin
               currCol := 1;
               if currRow < rows then
                  currRow := currRow + 1
               else
                  currRow := 1;
            end;
      end;

   if strcomp(s, 'left') = 0 then
      begin
         moveWin := true;
         if currCol > 1 then
            currCol := currCol - 1
         else
            begin
               if currRow > 1 then
                  currRow := currRow - 1
               else
                  currRow := rows;
               currCol := cols;
            end;
      end;

   if strcomp(s, 'up') = 0 then
      begin
         moveWin := true;
         if currRow > 1 then
            currRow := currRow - 1
         else
            currRow := 1;
      end;

end;
begin
  if currCol > 1 then
    currCol := currCol - 1
  else
    currCol := cols;
  currRow := rows;
end;

if strcmp(s,'down') = 0 then
begin
  moveWin := true;
  if currRow < rows then
    currRow := currRow + 1
  else
    begin
      if currCol < cols then
        currCol := currCol + 1
      else
        currCol := 1;
      currRow := 1;
    end;
end;
if moveWin then
  MoveEditCell(currRow,currCol);
end;

procedure TMatrixMDIChild.MoveEditCell(i,j:integer);

var
  wstring,outstring : array[0..79] of char;
begin
  (messagebox(hWindow,‘calling moveeditcell’,‘test’,mb_ok));
  currRow := i;
  currCol := j;
  MoveWindow (CellWindow^.hWindow,
    (cellWidth, * (currCol - 1)) + xStart,
    cellHeight * (currRow - 1) + yStart,
    cellWidth, cellHeight, True);
  Str (CellEval(currRow,currCol),wstring);
  StrPCopy(outstring,wstring);
  CellWindow^.SetText(outstring);
  (messagebox(hWindow,‘called moveeditcell’,‘test’,mb_ok));
end;

function TMatrixMDIChild.CellEval(i,j:integer) : real;
var
  index : integer;
  PCell : PCell;
begin
  index := i * cols + j;
  PCell := CellWindow;
  if PCell^[index] <> nil then
    begin
      Str (PCell^[index],wstring);
      StrPCopy(outstring,wstring);
      if outstring[0] = ' ' then
        outstring[0] := ";
      CellWindow^.SetText(outstring);
      StrPCopy(outstring,wstring);
    end;
end;
allows reference to the matrix in the conventional two dimensional manner... updates the integer value of the referenced cell

```pascal
procedure TMatrixMDIChild.CellStore(i,j: integer; r: real); var
    index : integer;
    cptr : PCell;
begin
    index := ((i-1)*cols) + (j-1);
    cptr := cells^ .at(index);
    cptr^ .e := r;
end;
```

allows reference to the matrix in the conventional two dimensional manner... returns the string value of the referenced cell

```pascal
function TMatrixMDIChild.CellStr(i,j: integer) : PChar;
var
    tempstring,retstring : array[0..79] of char;
begin
    StrICellEval(i,j),5,tempstring);
    StrPCopy(retstring,tempstring);
    CellStr := retstring;
end;
```

I. MATH OPERATIONS

multiply a matrix by a scalar

```pascal
procedure TMatrixMDIChild.ScalarAdd(var Msg : TMessage);
var
    inputText: array[0..5] of char;
    i,error : integer;
    scalar : real;
    cptr : PCell;
begin
    Str1,inputText);
    if application^ .ExecDialog(new(PInputDialog,
        Init(@Self, 'Scalar Add', ' Enter a scalar:',
        InputText, SizeOf(InputText))) = id_OK then
    begin
        Val(IInputText,scalar,error);
        if error = 0 then
        for i := 0 to (rows*cols-1) do
        begin
            cptr := cells^ .at(i);
            cptr^ .e := cptr^ .e + scalar;
        end;
        InvalidateRect(HWindow,nil,true);
    end;
end;
```
multplies a matrix by a scalar

procedure TMatrixMDIChild.ScalarMult (var Msg : TMessage);
var
  inputText: array[0..5] of char;
i, error: integer;
scalar: real;

begin
  Str(1,inputText);
  if application^$.ExecDialog(new(PInputDialog, Init(@Self, ‘Scalar Multiply’, ‘Enter a scalar:’, InputText, SizeOf(InputText))) = id_OK then
    Val(InputText, scalar, error);
    if error = 0 then
      for i := 0 to (rows*cols-1) do
        begin
          cptr := cellsA.at(i);
          cptr^e := cptr^e * scalar;
        end;
    InvalidateRect(HWindow,nil,true);
  end;
end;

procedure TMatrixMDIChild.MatrixPower (var Msg : TMessage);
var
  original, scratch : PScratch;
  inputText: array[0..5] of char;
i,j,k,n,square, error, exponent: integer;
sum: real;

get the value of the i,j referenced cell in a collection of cells

function gcv (p: pCollection; i,j,cols: integer): real;
var
  index: integer;
cptr: pcell;
begin
  index := ((i-1)*cols) + (j-1);
cptr := p.at(index);
gcv := cptr^e;
end;

{ put the value of the i,j referenced cell in a collection of cells }

procedure pcv (p: pCollection; i,j,cols: integer; putval: real);
var
  index: integer;
cptr: pcell;
begin
  index := ((i-1)*cols) + (j-1);
cptr := p.at(index);
cptr^e := putval;
end;
begin
  Str(2,inputText);
  if application^$.ExecDialog(new(PInputDialog, Init(@Self, ‘Matrix Power!’, ‘Enter a scalar:’,
InputText, SizeOf(InputText))) = id_OK then
Val(InputText, exponent, error);

if error = 0 then
begin
original := New(PScratch, Init);
scratch := New(PScratch, Init);

if cols < rows then
square := cols
else
square := rows;
rows := square;
cols := square;

for i := 1 to square do
for j := 1 to square do
pcv(original^cells,i,j,cols,gcv(Cells,i,j,cols));

for n := 1 to (exponent-1) do
begin
for i := 1 to square do
for j := 1 to square do
begin
sum := 0;
for k := 1 to square do
sum := sum + (gcv(original^cells,i,k,cols) * gcv(Cells,k,j,
          pcv(scratch^cells,i,j,cols,sum));
end;
for i := 1 to square do
for j := 1 to square do
pcv(Cells,i,j,cols,gcv(scratch^cells,i,j,cols));
end;

InvalidateRect(HWindow,nil,true);
end;
end;

-----------------------------------------------------------------------}

SETTINGS DIALOG BOX
------------------------------------------------------------------------

Settings Dialog Box; currently only allows setting the number of rows
and columns of a matrix
}

procedure TMatrixMDIChild.Specs(var Msg : TMessage);
var
  D : FDialog;
  E : FEdit;
s1,s2 : array[0..32] of char;
  err,retValue : integer;
specsRecord : TransferSpecsRecord;

begin

{ setup transfer record }
  str(rows:2,specsRecord.NumRows);
  str(cols:2,specsRecord.NumCols);

{ initialize and execute specs dialog resource }
  := New(FSpecsDialog, Init(@Self,'Specs'));
  New(E, InitResource(D, 1101, SizeOf(specsRecord.NumRows)));

end;
New(E, InitResource(D, 1102, SizeOf(specsRecord.NumCols)));  
D^.TransferBuffer := @SpecsRecord;  
retvalue := Application^.ExecDialog(D);  

{ user clicked id_ok / pressed return key }  
if retvalue = id_OK then  
  begin  
    { update the matrix object fields }  
    Val(specsRecord.NumRows, rows, err);  
    Val(specsRecord.NumCols, cols, err);  
    CurrRow := 1;  
    CurrCol := 1;  
    { make sure Windows repa...
procedure TMatrixMDIChild.FileNew(var Msg: TMessage);
begin
  Points^.FreeAll;
  InvalidateRect(HWindow, nil, True);
  IsDirty := False;
  IsNewFile := True;
end;

procedure TMatrixMDIChild.FileOpen(var Msg: TMessage);
begin
  if CanClose then
    if Application^.ExecDialog(New(PFileDialog,
      Init(@Self, PChar(sd_FileOpen),
        StrCopy(FileName, '*.MTX'))) = id_Ok then LoadFile;
  end;
end;

procedure TMatrixMDIChild.FileSave(var Msg: TMessage);
begin
  if IsNewFile then FileSaveAs(Msg) else SaveFile;
end;

procedure TMatrixMDIChild.FileSaveAs(var Msg: TMessage);
var
  FileDlg: PFileDialog;
begin
  if IsNewFile then StrCopy(FileName, '');
  if Application^.ExecDialog(New(PFileDialog,
      Init(@Self, PChar(sd_FileSave), FileName))) = id_Ok then SaveFile;
end;

procedure TMatrixMDIChild.LoadFile;
var
  TempColl: PCollection;
  TheFile: TDosStream;
begin
  TheFile.Init(fileName, stOpen);
  TempColl := PCollection(TheFile.Get);
  TheFile.Done;
  if TempColl <> nil then
    begin
      Dispose(Points, Done);
      Points := TempColl;
      InvalidateRect(HWindow, nil, True);
    end;
  IsDirty := False;
  IsNewFile := False;
end;

procedure TMatrixMDIChild.SaveFile;
var
  TheFile: TDosStream;
begin
  TheFile.Init(fileName, stCreate);
TheFile.Put(@Self);
TheFile.Done;
isNewFile := False;
IsDirty := False;
end;

---

procedure TMatrixMDIChild.Load (var S: TStream);
begin
    TWindow.Load(S);
    GetChildPtr (S, ChildMsg);
    S.Read (Num, SizeOf(Num));
    S.Read (Name, SizeOf(Name));
    S.Read (Description, SizeOf(Description));
    S.Read (TopLabel, SizeOf(TopLabel));
    S.Read (SideLabel, SizeOf(SideLabel));
    S.Read (Cells, SizeOf(Cells));
    S.Read (CellWindow, SizeOf(CellWindow));
    S.Read (CellPaint, SizeOf(CellPaint));
    S.Read (Rows, SizeOf(Rows));
    S.Read (Cols, SizeOf(Cols));
    S.Read (CurrRow, SizeOf(CurrRow));
    S.Read (CurrCol, SizeOf(CurrCol));
    S.Read (xStart, SizeOf(xStart));
    S.Read (yStart, SizeOf(yStart));
    S.Read (cellWidth, SizeOf(cellWidth));
    S.Read (cellHeight, SizeOf(cellHeight));
    S.Read (FileName, SizeOf(FileName));
    S.Read (IsDirty, SizeOf(IsDirty));
    S.Read (isNewFile, SizeOf(isNewFile));
end;

---

procedure TMatrixMDIChild.Store (var S: TStream);
begin
    TWindow.Store(S);
    PutChildPtr (S, ChildMsg);
    S.Write (Num, SizeOf(Num));
    S.Write (Name, SizeOf(Name));
    S.Write (Description, SizeOf(Description));
    S.Write (TopLabel, SizeOf(TopLabel));
    S.Write (SideLabel, SizeOf(SideLabel));
    S.Write (Cells, SizeOf(Cells));
    S.Write (CellWindow, SizeOf(CellWindow));
    S.Write (CellPaint, SizeOf(CellPaint));
    S.Write (Rows, SizeOf(Rows));
    S.Write (Cols, SizeOf(Cols));
    S.Write (CurrRow, SizeOf(CurrRow));
    S.Write (CurrCol, SizeOf(CurrCol));
    S.Write (FileName, SizeOf(FileName));
    S.Write (IsDirty, SizeOf(IsDirty));
    S.Write (isNewFile, SizeOf(isNewFile));
end;
constructor TMatrixMDIWindow.Init(Atitle: PChar); begin TMDIWindow.Init ('MatrixCad', LoadMenu(HInstance, 'MDIMENU')); (Attr.X := 0; Attr.Y := 0; Attr.W := 640; Attr.H := 300; Attr.Style := ws_Overlapped or ws_SysMenu or ws_MinimizeBox;}; ChildMenuPos := 1; MatrixNames := New(PStrCollection, Init(1, 5)); end;

---.------------_._--------------_._----------------------------------------}

SetupWindow creates the first MDI child procedure TMatrixMDIWindow.SetupWindow; var ARect: TRect; NewChild: PMatrixMDIChild; begin TMDIWindow.SetupWindow; CreateChild; {LoadChild;} end;

---.------------------------------------------

Create a new MDI child } function TMatrixMDIWindow.CreateChild: PWindowsObject; var ChildNum: Integer;

function NumberUsed(P: PMatrixMDIChild): Boolean; far; begin NumberUsed := (ChildNum = P^.Num); end;

begin ChildNum := 1; while FirstThat(@NumberUsed) <> nil do Inc(ChildNum);
CreateChild := Application^ MakeWindow(New(PMatrixMDIClild, Init(Self, ChildNum)));
procedure GetAChild(ACHild: PMatrixMDIChild); far;
begin
  C^.Insert(StrNew(achild^.attr.title));
end;

begin
  FreeAll; {clear out the collection for updating}
  Each(@GetAChild);
end;

---

procedure TMatri}MDIWindow.About(var Msg: TMessage);
var
  D : PDialog;
  ReturnValue : integer;
begin
  D:= New(PAboutDialog,Init(@Self,'ABOUT'));
  ReturnValue := Application^ExecDialog(D);
end;

---

procedure TMatri}MDIWindow.BinaryOpsDlg (var Msg : TMessage);
var
  D : PDialog;
  S : PStatic;
  LL,RL,TL : PlistBox;
  s1,s2 : array[0..32] of char;
  i,err,retValue : integer;
  opsRecord : TransferopsRecord;
begin
  with opsRecord do
  begin
    { initialize matrix name lists }
    LOpList := New(PStrCollection,Init(10,5));
    ROpList := New(PStrCollection,Init(10,5));
    TOpList := New(PStrCollection,Init(10,5));
    UpDateChildList(LOpList);
    UpDateChildList(ROpList);
    UpDateChildList(TOpList);
    StrPCopy(operation,'PLUS');
    Lindex := 0;
    Rindex := 0;
    Tindex := 0;
  end;

  { initialize and execute dialog box }
  S:= New(POpsDialog,Init(@Self,'BINARY OPS'));
  New( S, InitResource(D, 2301, SizeOf(opsRecord.operation)));
end;
New(II, InitResource(D, 2101));
New(RL, InitResource(D, 2103));
New(TL, InitResource(D, 2105));
D^'.TransferBuffer := @opsRecord;
retvalue := Application^.ExecDialog(ID);

if retvalue = id_OK then
  with opsrecord do
  begin
    BinaryOps(Lindex, Rindex, Tindex, operation);
    ForEach(@UpdateChildren);
  end;
end;

{------------------------------------------------------------------------}
{ converts the user selected matrix index numbers into child window    }
{ pointers, then calls the appropriate operation with those pointers    }
procedure TMatrixMDIWindow.BinaryOps(index1, index2, index3 : integer; op : PChar);
var
  i : Integer;
  pL, pR, pT : PMatrixMDIChild;
  s : PChar;

  {-- locates the left operand child window pointer --}
  function FindLOp(AChild : PMatrixMDIChild); boolean; far;
  begin
    FindLOp := (i = index1);
    i := i + 1;
  end;

  {-- locates the right operand child window pointer --}
  function FindROp(AChild : PMatrixMDIChild); boolean; far;
  begin
    FindROp := (i = index2);
    i := i + 1;
  end;

  {-- locates the target operand child window pointer --}
  function FindTOp(AChild : PMatrixMDIChild); boolean; far;
  begin
    FindTOp := (i = index3);
    i := i + 1;
  end;

begin
  i := 0;
pL := PMatrixMDIChild(firstthat(@FindLOp));
  i := 0;
pR := PMatrixMDIChild(firstthat(@FindROp));
  i := 0;
pT := PMatrixMDIChild(firstthat(@FindTOp));

  MessageBox(hWnd, op, 'test', mb_OK);
if ((pL<>nil) and (pR<>nil) and (pT<> nil)) then
  if strcomp(op, 'PLUS') = 0 then
    AddMatrices(pL,pR,pT)
  else if strcomp(op, 'TIMES') = 0 then
    TimesMatrices(pL,pR,pT)
  else if strcomp(op, 'MINUS') = 0 then
    MinusMatrices(pL,pR,pT)
  else if strcomp(op, 'POLY') = 0 then
    begin
      {messagebeep(0);} 
      PolyXMatrices(pL,pR,pT);
    end;
end;

{ BINARY MATRIX OPERATIONS }

procedure TMatrixMDIWindow.TimesMatrices(pL,pR,pT : PMatrixMDIChild);
var
  pLx,pRx,pTx : pcell;
  i,j,k : integer;
  sum : real;

  { get the value of the i,j referenced cell in a collection of cells } 
function gcv (p : pCollection; i,j,cols : integer) : real;
var
  index : integer;
  cptr : pcell;
begin
  index := ((i-1)*cols) + (j-1);
  cptr := p^ .at(index);
  gcv := cptr^ .e;
end;

  { put the value of the i,j referenced cell in a collection of cells } 
procedure pcv (p : pCollection; i,j,cols : integer; putval : real);
var
  index : integer;
  cptr : pcell;
begin
  index := ((i-1)*cols) + (j-1);
  cptr := p^ .at(index);
  cptr^ .e := putval;
end;

begin

  for i := 1 to pL^.rows do
    for j := 1 to pR^.cols do
      begin
        sum := 0;
        for k := 1 to pL^.cols do
          sum := sum + (gcv(pL^.cells,i,k,pL^.cols) * gcv(pR^.cells,k,j,pR^.cols) * gcv(pT^.cells,i,j,pR^.cols,sum));
        pcv(pT^.cells,i,j,pR^.cols,sum);
      end;

  p^.rows := pL^.rows;
  p^.cols := pR^.cols;
pR matrix is considered to be a constant vector
pM matrix is the matrix variable
pT matrix is the target matrix

procedure TMatrixMDIWindow.PolyXMatrices(pL, pR, pT: PMatrixMDIChild);

var
R, accum, scratch: PScratch;
inputText: array[0..5] of char;
i, j, k, n, square, error, maxExponent: integer;
x, sum: real;

{ get the value of the i, j referenced cell in a collection of cells }
function getcv (p: pCollection; i, j, cols: integer) : real;
var
index: integer;
cptr: pcell;
begin
index := ((i-1)*cols) + (j-1);
cptr := p^.at(index);
getcv := cptr^.e;
end;

{ put the value of the i, j referenced cell in a collection of cells }
procedure putcv (p: pCollection; i, j, cols: integer; putval: real);
var
index: integer;
cptr: pcell;
begin
index := ((i-1)*cols) + (j-1);
cptr := p^.at(index);
cptr^.e := putval;
end;

begin
messagebox(hWindow, 'calling POLY!', 'test', mb_ok);

R := New(PScratch, Init);
Accum := New(PScratch, Init);
Scratch := New(PScratch, Init);

{* square up variable array to minimum dimension *}
if pR^.cols < pR^.rows then
  square := pR^.cols
else
  square := pR^.rows;

{* square target for valid result representation *}
pT^.rows := square;
pT^.cols := square;

{* copy variable array over to R array structure *}
for i := 1 to square do
  for j := 1 to square do
    putcv(R^.cells, i, j, square, getcv(pR^.Cells, i, j, square));
(* exponent determined by length of pl vector...  
\[ A^0, A^1, A^2, \ldots, A^{(\text{cols}-1)} \]  
\[ \text{maxExponent} := (\text{pl}^{\text{cols}} - 1); \]

(* aR^0  *)  
\[ \text{aaa} := \text{getcv}(\text{pl}^{\text{cells}},1,1,\text{square}); \]
\[ \text{for } i := 1 \text{ to square do} \]
\[ \text{for } j := 1 \text{ to square do} \]
\[ \text{putcv}(\text{pT}^{\text{cells}},i,j,\text{square},x); \]

(* initialize Accum *)  
\[ \text{for } i := 1 \text{ to square do} \]
\[ \text{for } j := 1 \text{ to square do} \]
\[ \text{putcv}(\text{accum}^{\text{cells}},i,j,\text{square},1); \]

\[ \text{for } n := 1 \text{ to maxExponent do} \]
  \[ \text{begin} \]
  \[ \text{for } i := 1 \text{ to square do} \]
  \[ \text{for } j := 1 \text{ to square do} \]
  \[ \text{begin} \]
  \[ \text{sum} := 0; \]
  \[ \text{for } k := 1 \text{ to square do} \]
  \[ \text{sum} := \text{sum} + (\text{getcv}(\text{R}^{\text{cells}},i,k,\text{square}) \times \text{getcv}(\text{accum}^{\text{cells}},k, \text{putcv}(\text{scratch}^{\text{cells}},i,j,\text{square},\text{sum}); \]
  \[ \text{end;} \]

(* copy powered matrix from scratch back into Accum *)  
\[ \text{for } i := 1 \text{ to square do} \]
\[ \text{for } j := 1 \text{ to square do} \]
\[ \text{putcv}(\text{accum}^{\text{cells}},i,j,\text{square},\text{getcv}(\text{scratch}^{\text{cells}},i,j,\text{square})); \]

(* multiply accum matrix by constant from pl vector  
and update result  *)  
\[ x := \text{getcv}(\text{pl}^{\text{cells}},1,n,\text{square}); \]
\[ \text{for } i := 1 \text{ to square do} \]
\[ \text{for } k := 1 \text{ to square do} \]
\[ \text{begin} \]
\[ \text{putcv}(\text{pT}^{\text{cells}},i,k,\text{square}, \text{getcv}(\text{pT}^{\text{cells}},i,k,\text{square}) + \text{x} \times \text{getcv}(\text{accum}^{\text{cells}},i,k,\text{square})); \]
\[ \text{end}; \]
\[ \text{end}; \]
\[ \text{InvalidateRect(HWindow, nil, true);} \]

---------------------------------------------------------------------}  
subtract two matrices...put the result into the specified target;  
NOTE: the left operand controls the row/cols extent of the subtraction  
procedure TMatrixMDIWindow.MinusMatrices(pl,pr,pT : PMatrixMDIChild);  
var  
plx,prx,pTx : pcell;  
\[ n := \text{integer}; \]

рав
for i := 0 to ((pL^.rows * pL^.cols) - 1) do begin
  pLx := pL^.cells^.at(i);
  pRx := pR^.cells^.at(i);
  pTx := pT^.cells^.at(i);
  pT^.<e := pL^.<e - pR^.<e;
end;

procedure TMatrixMDIWindow.AddMatrices (pL,pR,pT : PMatrixMDIChild);
begin
  var
    pLx,pRx,pTx : pcell;
    i : integer;
  begin
    for i := 0 to ((pL^.rows * pL^.cols) - 1) do begin
      pLx := pL^.cells^.at(i);
      pRx := pR^.cells^.at(i);
      pTx := pT^.cells^.at(i);
      pT^..<e := pL^.<e + pR^.<e;
    end;
  end;
end;

procedure TMatrixMDIWindow.FileOpen (var Msg: TMessage);
begin
  var
    AFile: array[0..12] of Char;
  begin
    if Application^.ExecDialog(New(PLoadMatrixDialog, Init(@Self,
      'LOADMATRICES', AFile))) = id_OK then begin
    end;
  end;
end;

APPLICATION METHODS
Construct a main window object
procedure TMatrixMDIApp.InitMainWindow;
begin
  MainWindow := New(PMatrixMDIWindow, Init('MatrixCad'));
  StreamRegistration;  { register all streamed objects }
end;

MAIN MODULE
MatrixMDIApp: TMatrixMDIApp;
begin
  MatrixMDIApp.Init('MatrixCad');
  MatrixMDIApp.Run;
MatrixMDIApp.Done;
end.
ltxCad 1.1 User-defined Messages Unit

programmer: Terry D. Hawkins

Unit MtxMsgs;

INTERFACE

Uses WinTypes;

\ { Menu bar constants }\n\const

\ { user defined message constants }\nwm_CellReturned = wm_User;
wm_CellEscaped = wm_User + 1;

\ { user defined messages processed by frame window class }\nFW_MDICHILDDESTROY = WM_USER + 0;
FW_RESIZEMDICLIENT = WM_USER + 1;
FW_GETSTATBARRECT = WM_USER + 2;
FW_SETMENUHELP = WM_USER + 3;
FW_GETMENUHELP = WM_USER + 4;
FW_DRAWSTATUSDIVIDE = WM_USER + 5;

\ { user defined messages processed by all windows classes }\nAW_PAINTMENUHELP = WM_USER + 100;

\ { user defined messages processed by all MDI Child window classes }\nAU_PAINTSTATBAR = WM_USER + 200;

IMPLEMENTATION

END.
**MtXCad 1.1 Resource Identifiers**

Programmer: Terry D. Hawkins

```
unit mtxids;

interface

const

NewGraphWin = 200;
GetN = 201;
cm_ClearGraph = 202;
cm_ClearEdges = 203;
cm_JoinAllNodes = 204;
cm_IsEulerPath = 205;
cm_ShortestPath = 206;
cm_MinSpanTree = 207;

Grid_Toggle = 300;
Square_Grid = 310;
Polar_Grid = 311;
PointSize = 320;
EdgeMode = 321;
ShowWeightMode = 322;

cm_CountChildren = 102;
CntClose = 201;
id_row = 301;

{ menu command identifiers }
cm_specs = 1001;
cm_scalarMult = 301;
cm_matrixPower = 302;
cm_scalarAdd = 303;
cm_BinaryOps = 601;

{ file menu command identifiers }
cm_Open = 701;
cm_New = 702;
cm_Save = 703;
cm_SaveAs = 704;
cm_About = 750;

{ help command identifiers }
cm_help = 2000;

{ specs dialog box id's }
id_rows = 1101;
id_cols = 1102;

{ ops dialog id's }
+TimesButton = 2201;
+PlusButton = 2202;
```
id_MinusButton = 2203;

id_PolyXButton = 2204;

id_opStatic = 2301;

implementation

nd.
programmer: Terry D. Hawkins

Ib,Cad 1.1 Misc. Utilities Unit

Init utils;

INTERFACES WinTypes, WinProcs, Strings;

function Min(X, Y: Integer): Integer;
    begin
        if X > Y then Min := Y else Min := X;
    end;

procedure Int2PChar(n : integer);
    begin
        str(n,StringBuffer);
        strcpy(pCharBuffer,StringBuffer);
    end;

procedure ToggleCheck(Menu: HMenu; MenuItemID: Word);
    var
        MAttr, WCheck : Word;
    begin
        MAttr := GetMenuState(Menu, MenuItemID, Mf_ByCommand);
        if (MAttr and Mf_Checked) = Mf_Checked then
            WCheck := Mf_ByCommand or Mf_Unchecked
        else
            WCheck := Mf_ByCommand or Mf_Checked;
        CheckMenuItem(Menu, MenuItemID, WCheck);
    end;

end.

IMPLEMENTATION

------------------------------------------------------------------
function Min(X, Y: Integer): Integer;
    begin
        if X > Y then Min := Y else Min := X;
    end;
------------------------------------------------------------------
procedure Int2PChar(n : integer);
    begin
        str(n,StringBuffer);
        strcpy(pCharBuffer,StringBuffer);
    end;
------------------------------------------------------------------
procedure ToggleCheck(Menu: HMenu; MenuItemID: Word);
    var
        MAttr, WCheck : Word;
    begin
        MAttr := GetMenuState(Menu, MenuItemID, Mf_ByCommand);
        if (MAttr and Mf_Checked) = Mf_Checked then
            WCheck := Mf_ByCommand or Mf_Unchecked
        else
            WCheck := Mf_ByCommand or Mf_Checked;
        CheckMenuItem(Menu, MenuItemID, WCheck);
    end;

end.
program test;
$R MTXCAD.RES}
uses MtxGrfx,MtxMsgs,MtxIds,Utils,
    WObjects,WinProcs,WinTypes;

--------------------------------------------------

type
    TestApp = object(TApplication)
        procedure InitMainWindow; virtual;
    end;

type
    PTestMDIWin = ^TestMDIWin;
    TestMDIWin = object(TMDIWindow)
        procedure NewGraphWin(var Msg: TMessage);
            virtual cm_First + NewGraphWin;
    end;

--------------------------------------------------

procedure TestMDIWin.NewGraphWin(var Msg: TMessage);
begin
    Application^.MakeWindow(New(PMtxGraph, Init(@Self,
            'Matrix Graph')));
end;

--------------------------------------------------

procedure TestApp.InitMainWindow;
begin
    MainWindow := New(PTestMDIWin,
        Init('Matrix Graphics', LoadMenu(HInstance,MakeIntResource('MDIMENU'))));
end;

------------------------------------------------------------------------

var
    Test1: TestApp;

begin
    Test1.Init('Matrix Graphics');
    Test1.Run;
    Test1.Done;
end.