GDI Reference Manual

Mark Salem, Sue Skowronski, Beat Bruderlin

UUCS-92-031

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

September 16, 1992

Abstract

GDI is a dialog interface tool library for C++ applications. It facilitates the design and implementation of graphical, object-oriented user interfaces for workstations equipped with a graphical display, a mouse and a keyboard. GDI's design allows for its portability onto a multitude of platforms. This is achieved by separating the user interface from the application program, as well as the orderly detachment of system independent user- interface tools from the system dependent, low level, window operations.

GDI Reference Manual

An Object-Oriented, Graphical User Interface Toolkit
(Includes Motif Version)

Technical Report UUCS-92-031

Mark Salem
Sue Skowronski
Beat Bruderlin

Department of Computer Science
University of Utah
1992

Table of Contents

1. Introduction to GDI

What is GDI?

GDI is a dialog interface library for C++ applications. It facilitates the design and implementation of graphical, object-oriented user interfaces for workstations equipped with a graphical display, a mouse and a keyboard. GDI's design allows for its portability onto a multitude of platforms. This is achieved by separating the user interface from the application program, as well as the orderly detachment of system independent, user interface tools from the system dependent, low level, window operations.

Introduction

Graphical user interfaces are quickly becoming the norm in computing environments. Such interfaces are normally very difficult to build. Cardelli {2, p. 152} enumerates three major factors contributing to this difficulty. The first is the artistic burden: the artistic insight needed in preparing attractive user interfaces; i.e., the choice of shapes, proportions, arrangements and colors. The second factor is the polishing burden: the constant redesigning and polishing of interfaces to achieve "professional smoothness". By "smoothness," Cardelli describes the sense of being intuitive and not causing surprises to the user. The final factor is the programming burden: the knowledge of graphics and windowing techniques required to build such interfaces. Application programmers should not need to deal with the low-level quirks of the particular window system for which they are programming. Such details of technique often make the difficulty of "getting things to work" an unnecessarily demanding undertaking.

GDI (Graphical Dialog Interface) mitigates all three of the above burdens. Used in conjunction with a GDI application, called HyperGDI, GDI attacks the artistic burden. HyperGDI is a user interface editor which allows the application programmer to construct a layout of the user interface as well as test it before the application is ever begun. GDI also simplifies the remaining two burdens. It allows the application programmer to define user interfaces at a high level. This approach not only saves the programmer from a lot of work spent in building primitive interactors, but also makes the application easily portable. The library provides graphical units of interaction called interaction objects {1, p. 2}. These objects have a visual appearance on the screen, and react to user interactions such as clicking, dragging, or releasing the mouse. The objects' behavior in response to user interaction is specified by methods which are particular to that interaction object, and defined within the application. GDI also provides a number of objects with predefined methods; e.g., buttons, sliders, textfields. Unlike with conventional interactive program design, GDI applications do not dictate an interaction dialog. In other words, the user can directly interact with all active and visible objects in any order; each object, in turn, passes messages back to the application describing the user's interaction with it.

Developments of GDI

There are several projects concerning GDI (and consequently MGDI) worth mentioning: The GDI Editor and GDI-3D. The GDI Editor aids in the design of screen layouts for graphical, interactive applications using MGDI (or GDI). It allows the application programmer to specify windows, text, graphics, interaction objects and their locations graphically. The Editor can then produce compile-ready code. This eases the application programmer's job substantially since the user-interface is automatically written. GDI-3D is another extension of the GDI library providing 3D transformations, projection, etc.

2. DIGraphicWindows

DIGraphicWindows is the part of the MGDI toolkit that is concerned with graphic operations and window objects. The routines are grouped into logical sections depending on their purpose. All routines in DIGraphicWindows must be prepended with 'DIGraphicWindows.'; e.g., 'DIGraphicWindows.GetBorderMode'. This is because DIGraphicWindows is an instance of a C++ object (DIGraphicWindowsMod) that has these routines as its methods. Only this one instance (DIGraphicWindows) should be used; no others need be created.

Class *DIWindowObj*

Class DIWindowObj

```
DIWindowObj( Rectangle r, SET attribs, char *name
DIWindowObj();
```

Constructor for the DIWindowObj class--the GDI window object. 'r' specifies the dimensions of the window, and 'name' its title. 'attribs' is an enumerated type that can be:

hasBorder	hasTitle
mayBeClosed	mayBeDeleted
mayBeMoved	mayBeChanged

To get a combination of attributes, use OR; e.g., hasBorder | hasTitle.

Screen & Windows

ClearWindow

```
void ClearWindow( Object *w );
```

Erases the inner rectangle of object 'w' irrespective of the paint mode.

CursorIsVisible

```
BOOLEAN CursorIsVisible( void );
```

Returns TRUE if cursor is visible, FALSE otherwise.
See also ShowCursor, HideCursor.

GetClipRectangle

```
void GetClipRectangle( BOOLEAN *flag, Rectangle *r );
```

Returns in 'r' the clipping rectangle for the window and sets 'flag' to TRUE if there is clipping in the current window, FALSE otherwise.
See also SetClipRectangle, UnsetClipRectangle.

GetNumColors

```
int GetNumColors();
```

Returns the size of the color map (the # of possible colors).
See also NewColor, GetColor.

GetPort

```
void GetPort( Object **o );
```

Returns in 'o' the active port.
See also SetPort.

GetWindowOf

```
void GetWindowOf( Object *o, Object **w );
```

Sets 'w' to the window of the object 'o'.
See also SetPort, GetPort.

HideCursor `void HideCursor( void );`
Hides the cursor on the screen in the current window.
See also ShowCursor.

ScreenRectangle `Rectangle ScreenRectangle( void );`
Returns the rectangle of the largest usable screen area.
See also ScreenSize.

ScreenSize `void ScreenSize( CARDINAL *width, CARDINAL *height );`
Returns the width and height of the screen in the variables 'width' and 'height'.
See also ScreenRectangle.

SetClipRectangle `void SetClipRectangle( Rectangle r );`
Sets the clipping rectangle for the current window.
See also UnsetClipRectangle, GetClipRectangle.

SetPort `void SetPort( Object *o );`
Sets the active port to the window or sub-window of object 'o'.
See also GetPort.

ShowCursor `void ShowCursor( void );`
Shows the cursor in the current window.
See also HideCursor.

UnsetClipRectangle `void UnsetClipRectangle( void );`
Removes the clipping rectangle for the current window.
See also SetClipRectangle, GetClipRectangle.

Bitmaps

DefineBitmap `Bitmap DefineBitmap( int w, int h, char *bits );`
Creates a bitmap structure from an array of bits. 'w' & 'h' are the number of pixel columns and rows, respectively. The entire array should be: (0..(h*w DIV C)), where C is the byte size of the computer.
See also EmptyPattern, FullPattern, HalfPattern, LoadPattern.

EmptyPattern `Bitmap EmptyPattern( void );`
Returns a empty bitmap (i.e., all pixels set to background color) for filling geometric shapes.
See also EmptyPattern, FullPattern, LoadPattern.

FullPattern `Bitmap FullPattern( void );`
Returns a full bitmap (i.e., all pixels set) for filling geometric shapes.
See also EmptyPattern, HalfPattern, LoadPattern.

HalfPattern

```
Bitmap HalfPattern( void );
```

Returns a grey bitmap (i.e., every alternate pixel set with fill color, others set to background color) for filling geometric shapes.
See also EmptyPattern, FullPattern, LoadPattern.

LoadPattern

```
Bitmap LoadPattern( char *filename );
```

Returns an MGDI bitmap (pattern) object which it creates from an X bitmap application file ('filename').
See also EmptyPattern, FullPattern, HalfPattern, DefinePattern.

Color

GetColor

```
void GetColor( Color clr,
                        double *r, double *g, double *b );
```

Given the color map index value 'clr', GetColor will return the red, green, and blue values associated with it in r, g, & b, respectively.
See also NewColor, GetNumColors.

NewColor

```
Color NewColor( double redValue, double greenValue,
                         double blueValue );
```

Defines a new color cell in the color map with specific RGB components and returns its index value. The range for the RGB components is (0..1.0). Note that this does not set the current color.
See also GetColor, GetNumColors.

Predefined Colors

GDI also has a set of predefined colors:

White	Blue
Black	Yellow
Red	Cyan
Green	Magenta
Fg	Bg

These predefined colors, as members of a class, need to be prepended with "DIGraphicWindows."; e.g., DIGraphicWindows.Yellow. 'Fg' and 'Bg' hold the configuration-dependent foreground and background colors, respectively. Using 'Fg' and 'Bg' instead of explicit White and Black allows GDI applications to run independently of the configuration foreground and background colors.

Setting Graphic Parameters

The following functions set the graphic parameters of the currently active window (See SetPort).

SetBorderMode

```
void SetBorderMode( BorderMode bm );
```

Sets the border mode of filled shapes in the currently active port.
'bm' can be:

noBorder:	no explicit border is drawn. The border line is filled with the pattern.
solidBorder:	the border is drawn as a solid line with the window's default paint mode.
backgroundBorder:	no border is drawn.

Default value is 'noBorder'.
See also SetPaintMode, SetFillPattern.

SetFillColor

```
void SetFillColor( Color c );
```

Sets the color of filled shapes in the currently active port. The color is defined by the colormap indeces as retrieved by *NewColor*.
Default value is 'Fg' (foreground color).
See also SetPaintMode, SetFillPattern.

SetFillPattern

```
void SetFillPattern( Bitmap pattern );
```

Sets the fill pattern in the currently active port.
Default value is 'FullPattern'.
See also SetPaintMode, SetFillColor.

SetLineColor

```
void SetLineColor( Color color );
```

Sets the color for lines and border-lines of filled shapes in the currently active port. The color is defined by the colormap indeces as retrieved by procedure *NewColor*.
Default value is 'Fg'.
See also SetPaintMode, SetFillPattern.

SetLinePattern

```
void SetLinePattern( Bitmap pattern );
```

Sets the pattern for lines and border-lines of filled shapes in the currently active port.
Default value is 'FullPattern'.
See also SetPaintMode, SetFillPattern.

SetLineThickness

```
void SetLineThickness( int thickness );
```

Sets the line thickness in pixels.
Default value is 1.
See also SetLineColor, SetLinePattern.

SetPaintMode

```
void SetPaintMode( PaintMode mode );
```

Sets the paint mode for the currently active port.
'mode' can be one of the following:

- paint --
- replace --
- invert --
- erase --
- mask --

Default value is 'replace'.
See also SetLinePattern, SetFillPattern.

Getting Graphic Parameters

GetBorderMode	`void GetBorderMode( BorderMode *bm );` Returns the border mode of filled shapes in the currently active port. 'bm' will be: noBorder: no explicit border is drawn. The border line is filled with the pattern. solidBorder: the border is drawn as a solid line with the window's default paint mode. backgroundBorder: no border is drawn.. See also GetPaintMode, GetFillPattern.
GetFillColor	`void GetFillColor( Color *color );` Returns the color of filled shapes in the currently active port. The color is defined by the colormap indeces as retrieved by *NewColor*. See also GetPaintMode, GetFillPattern.
GetFillPattern	`void GetFillPattern( Bitmap *pattern );` Returns the fill pattern in the currently active port. See also GetPaintMode, GetFillColor.
GetLineColor	`void GetLineColor( Color *color );` Returns the color for lines and border-lines of filled shapes in the currently active port. The color is defined by the colormap indeces as retrieved by procedure *NewColor*. See also GetPaintMode, GetFillPattern.
GetLinePattern	`void SetLinePattern( Bitmap *pattern );` Returns the pattern for lines and border-lines of filled shapes in the currently active port. See also GetPaintMode, GetFillPattern.
GetLineThickness	`void GetLineThickness( int *thickness );` Returns the line thickness in pixels. See also GetLineColor, GetLinePattern.
GetPaintMode	`void GetPaintMode( PaintMode *mode );` Returns the paint mode for the currently active port. See also GetLinePattern, GetFillPattern.

Graphic Operations

ClearRectangle	`void ClearRectangle( Rectangle r );` Clears the area specified by rectangle 'r'. See also ClearWindow.

MoveTo

```
void MoveTo( int x, int y );
```

Sets the current pixel position to 'x', 'y' in window coordinates.
See also LineTo, DrawChar, DrawString.

PixelIsSet

```
BOOLEAN PixelIsSet( int h, int v );
```

Returns TRUE if pixel('h', 'v') in window coordinates is set. FALSE is returned for points outside the rectangle of the window.
See also DrawPixel.

Drawing Tools

DrawArc

```
void DrawArc( Point center, int h, int v,
              double fromAngle, double toAngle );
```

Draws and arc; i.e., part of an ellipse defned by 'center', 'h', and 'v' (see DrawEllipse). 'fromAngle' and 'toAngle' are given in radians measured clockwise from the half axis and delimit the arc to be drawn.
See also DrawEllipse.

DrawCircle

```
void DrawCircle( Point center, int radius );
```

Draws a circle with 'radius' and 'center' using the current paint mode.
See also DrawEllipse.

DrawEllipse

```
void DrawEllipse( Point center, int h, int v );
```

Draws an ellipse with 'h' and 'v' as the lengths of the half axes, and 'center' as the ellipse's center.
See also DrawCircle.

DrawLine

```
void DrawLine( Point from, Point to );
```

Draws a line from point 'from' to point 'to'.
See also LineTo.

DrawPixel

```
void DrawPixel( int h, int v );
```

Sets pixel('h', 'v') in window coordinates if the paint mode is *ppa''* resets the pixel if paint mode is *erase*. The other paint modes behave as expected from source set to TRUE.
See also PixelIsSet.

DrawPolygon

```
void DrawPolygon( Point *polygon, int numPoints );
```

Draws a closed polygon consisting of 'numPoints' points using the current paint mode. If HIGH(polygon) < 'numPoints' - 1 the result is unspecified. The last point is connected to the first point.
See also DrawPolyline, DrawRectangle.

DrawPolyline

```
void DrawPolyline( Point *polygon, int numPoints );
```

Draws an open polyline consisting of 'numPoints' points using the current paint mode. If HIGH(polygon) < 'numPoints' - 1 the result is unspecified.
See also DrawPolygon, DrawRectangle.

DrawRoundRectangle

```
void DrawRoundRectangle( Rectangle r, int rad );
```

Draws a rectangle with rounded corners: 'rad' is the radius of the corner circle arcs; 'r' is the rectangle.
See also DrawRectangle.

DrawRectangle

```
void DrawRectangle( Rectangle r );
```

Draws rectangle border given by the 'left', 'right', 'top', and 'bottom' components of rectangle r (in window coordinates) using the current paint mode.
See also DrawRoundRectangle.

LineTo

```
void LineTo( int x, int y );
```

Draws a line from current pixel position to pixel('x', 'y') and sets the current pixel position to be 'x', 'y'.
See also MoveTo.

Filling Tools

FillCircle

```
void FillCircle( Point center, int radius );
```

Fills the circle whose center is 'center' and with radius 'radius'. The fill is done dependent on the current fill pattern, border color, line pattern, fill color, and paint mode.
See also DrawCircle.

FillEllipse

```
void FillEllipse( Point center, int h, int v );
```

Fills the ellipse whose center is 'center' and with half axes of lengths 'h' and 'v'. The fill is done dependent on the current fill pattern, border color, line pattern, fill color, and paint mode.
See also DrawEllipse.

FillPolygon

```
void FillPolygon( Point *polygon, int numPoints );
```

Fills the polygon consisting of 'numPoints' points in 'polyon'. The fill is done dependent on the current fill pattern, border color, line pattern, fill color, and paint mode.
See also DrawPolygon.

FillRectangle

```
void FillRectangle( Rectangle r );
```

Fills the rectangle r. The fill is done dependent on the current fill pattern, border color, line pattern, fill color, and paint mode.
See also DrawRectangle.

FillRoundRectangle

```
void FillRoundRectangle( Rectangle r, int radius );
```

Fills the rounded-corner rectangle r. The fill is done dependent on the current fill pattern, border color, line pattern, fill color, and paint mode.
See also DrawRoundRectangle.

FillSector

```
void FillSector( Point center, int h, int v,
                    double fromAngle,double toAngle );
```

Fills the arc defined by 'center', 'h', 'v', 'fromAngle', 'toAngle' (see DrawArc). The fill is done dependent on the current fill pattern, border color, line pattern, fill color, and paint mode.
See also DrawArc.

---

3. DIObjects

Class *Object*: General Functions

Objects are defined as a class Object. Each object is defined with a rectangle containing its bounds. Each also has a set of flags indicating its status on the screen, and a set of virtual methods for event handlers which are automatically activated by the system. Each object has a parent, and the highest-level parent is the window which the object is in. The following describes these methods of the class Object.

Class Object	`Object( Rectangle rect );`
	`Object();`
	Constructor for the 'Object' class. The first allows specification of the object's rectangle.

GetParent	`Object *GetParent();` Returns the parent of the object; e.g., window of object.
GetRectangle	`void GetRectangle( Rectangle &rect );` Upon its return, 'rect' will hold the rectangle of the object. See also SetRectangle.
IsActive	`BOOLEAN IsActive();` Returns TRUE if the object is currently active, FALSE otherwise. See also IsVisible.
IsVisible	`BOOLEAN IsVisible();` Returns TRUE if the object is currently visible, FALSE otherwise. See also IsActive.
IsWindow	`BOOLEAN IsWindow();` Returns TRUE if the object is a window, FALSE otherwise.
SetRectangle	`void SetRectangle( Rectangle rect );` Sets the object's rectangle to 'rect'. See also GetRectangle.

Class *Object*: Event-Handlers

HandleNextEvent	`void HandleNextEvent();` Handles the next event stored in the event-queue (or returns immediately if the queue is empty). Depending on cursor position at the time of the event, HandleNextEvent calls the appropriate object event-handlers (described below).

See also Sample Code for a sample usage.

The object event--handlers are called when the object is indicated by the user. For instance, when the user presses the left mouse button down on an object, that object's down() handler is called. The draw() handler is called when the MGDI system draws/redraws the display list. The following are the base-class event-handlers.

Handler Method	**Event/Description**
virtual void **down**(int x, int y);	left mouse button depressed
virtual void **up**(int x, int y);	left mouse button up after depression
virtual void **move**(int x, int y);	mouse moving with button unpressed
virtual void **drag**(int x, int y);	mouse moving with button depressed
virtual void **dclick**();	mouse double-clicked
virtual void **bpress**();	mouse button continuously held down
virtual void **char_ev**(char c);	single key pressed
virtual void **fkey**(CARDINAL fk);	function key pressed
virtual BOOLEAN **ppick**(Point p);	determine if 'p' is within the object's rectangle
virtual BOOLEAN **rpick**(Rectangle r);	determine if 'r' overlaps the object's rectangle
virtual void **draw**();	called when object needs to be redrawn
virtual void **enable**();	called when the object is enabled
virtual void **disable**();	called when the object is disabled
virtual void **delete_ev**();	called when the object is deleted
virtual void **copy**(Object *o);	called when the object is copied

When an event-handler method is redefined, a corresponding flag must be set in the constructor of the inheriting object. The following are these flags:

```
BOOLEAN down_method_replaced;
BOOLEAN up_method_replaced;
BOOLEAN move_method_replaced;
BOOLEAN drag_method_replaced;
BOOLEAN char_method_replaced;
BOOLEAN fkey_method_replaced;
BOOLEAN dclick_method_replaced;
BOOLEAN bpress_method_replaced;
BOOLEAN ppick_method_replaced;
BOOLEAN rpick_method_replaced;
BOOLEAN draw_method_replaced;
BOOLEAN enable_method_replaced;
BOOLEAN disable_method_replaced;
BOOLEAN delete_method_replaced;
BOOLEAN copy_method_replaced;
```

DIObjects Functions

In addition to descriptions of individual objects, a list of objects is maintained for each window. An object is not added to this list until "DIObjects.AddObject" is called for it. To make it visible and allow its event handler methods to be activated, "DIObjects.Activate" must be called for that object.

Below are operations on the list and on objects as they affect the list. The following routines need to be prepended with "DIObjects.".

Activate

```
void Activate( Object *o );
```

Activate is simply the equivalent of Enable() and Show() on an object.
See also Enable, Show.

AddObject

```
void AddObject( Object *o, Object *w );
void AddObject( Object *o, Object *w, Point p );
```

Adds the object 'o' to the display list as a child of window 'w'. Concerning the second declaration: every graphic object has a rectangular extension. This rectangle is moved such that its upper left corner is at 'p'. It must be given in window coordinates, its size in pizels. The position and size of an object relative to its window can be changed by Move and Resize.

Copy

```
void Copy( Object *o, Object **o2, Object *toWin );
```

Copies the object 'o' to the object 'o2' in window 'toWin'.
See also Move.

Deactivate

```
void Deactivate( Object *o );
```

Activate is simply the equivalent of Disable() and Hide() on an object.
See also Disable, Hide.

Disable

```
void Disable( Object *o );
```

Makes the object 'o' inactive; i.e., no more events can be directed to it. The object is not disposed of or made invisible. Disabling an already disabled object has no effect.
See also Enable, Show.

Dispose

```
void Dispose( Object *o );
```

Deletes the object 'o' and deallocates its resources. 'o' is set to an invalid value. Disposing an already disposed object has no effect.

Draw

```
void Draw( Object *o );
```

Redraws object 'o'.
See also Copy.

Enable

```
void Enable( Object *o );
```

Announces the object to the event-handler. This means that an event, which occurs inside the object's rectangle, is directed to that object's methods. Enabling an already enabled object has no effect.
See also Disable, Hide.

Hide

```
void Hide( Object *o );
```

Hides the object 'o' on the screen. No more events can be directed towards that object. Hiding an already hidden object has no effect.
See also Show, Deactivate.

Inside

```
BOOLEAN Inside( Rectangle r, int x, int y );
```

Returns TRUE if the point ('x','y') is within rectangle 'r'.
See also Intersecting.

Intersecting

```
BOOLEAN Intersecting( Rectangle r1, Rectangle r2 );
```

Returns TRUE if rectangle 'r1' intersects rectangle 'r2'.
See also Intersecting.

Move

`void Move( Object *o, int dx, int dy );`

Changes the coordinates of object o's rectangle and thereby moving it to a new position within the window. Displacement is specified by 'dx', 'dy'.
See also Resize.

Resize

`void Resize( Object *o, Rectangle r );`

Changes object o's size. If the object is active, it will be redrawn with the new shape. The new rectangular size is specified in 'r'.
See also Move.

PointPick

`void PointPick( Object *o, Point p );`

If point 'p' is within the rectangle of 'o' the point pick method is called.
See also RectanglePick, SetPointPickProc.

RectanglePick

`void RectanglePick( Object *o, Rectangle r );`

If rectangle 'r' overlaps with the rectangle of 'o' the rectanlge pick method is called.
See also RectanglePick, SetPointPickProc.

SetPriority

`void SetPriority( Object *o, Priority p );`

Sets the priority of object 'o'. It has no effect on inactive objects. The priority of objects indicates the order in which they are checked for selection if an event occurs. The priority is set automatically upon activation of any object. The object that is activated last has the hightes priority. Priority only matters for overlapping objects. 'p' can either be "last", or "first".

Show

`void Show( Object *o );`

Displays the object 'o' on the screen by calling its draw() method. However, it will not be activated until Activate() is used. Showing an already visible object has no effect.
See also Hide, Activate.

---

4. DIButtons

Buttons are derived from the class Object. Three types of buttons exist: checkboxes (signifying on/off), radiobuttons (signify which of a set of buttons is set), and command buttons (have text identification and are "loaded" with routines which exectue when the buttons are clicked).

Class *DICheckBox*

Class DICheckBox

```
DICheckBox( int xleft = 0, int ytop = 0 );
DICheckBox( Point upperleft );
DICheckBox( Point upperleft, DIWindowObj win );
```

Constructor for the checkbox object. New checkboxes will be turned off. The first definition takes and 'x,y' pair, while the second definition takes a Point describing an 'x,y' pair. The third definition allows for an option specification of the window to which the checkbox is added (i.e., AddObject is not required).

SwitchCheckBox

```
void SwitchCheckBox( BOOLEAN onoff );
```

Switches the status of the checkbox according to 'onoff'.
See also IsOn.

IsOn

```
BOOLEAN IsOn();
```

Returns TRUE if the checkbox is ON, FALSE otherwise.
See also SwitchCheckBox.

Class *DIRadioButtonSet*

Because of the length of the string "RadioButton", the two characters "RB" will replace it in the marginal function names (but will remain in the elongated form within the description section).

Class DIRBSet

```
DIRadioButtonSet( int xleft=0, int ytop=0,
        int numOfButtons=2, DIDirection dir=vertical,
        int bDist=10 );
DIRadioButtonSet( int xleft, int ytop,
        int numOfButtons, DIDirection dir, int bDist,
        DIWindowObj win );
```

Constructor for the RadioButtonSet object. 'xleft' and 'ytop' define the upper left position of the set in screen coordinates. 'numOfButtons' is the number of buttons in the radiobutton set. 'dir' can either be 'vertical' or 'horizontal' representing the direction of the set. 'bDist' is the distance between the buttons in the set. The second form of the constructor is similar to the first except for allowing the specification of 'win', the window to which the radiobutton set is to be added (i.e., AddObject is not required).

GetButtonDist

```
int GetButtonDist();
```

Returns the distance between the buttons in the set.
See also SetButtonDist.

GetDirection

```
DIDirection GetDirection();
```

Returns the direction of the radiobutton set. It will return either 'horizontal' or 'vertical'.
See also SetDirection/

GetNumButtons

```
int GetNumButtons();
```

Returns the number of buttons within the set.

SetButtonDist

```
void SetButtonDist( int bDist );
```

Sets the distance between the buttons.
See also GetButtonDist.

SetDirection

```
void SetDirection( DIDirection dir );
```

Sets the direction of the buttonset; 'dir' can be either 'horizontal' or 'vertical'.
See also GetDirection

SwitchRBSet

```
void SwitchCheckBox( int newButton );
```

Switches the radiobutton set to 'newButton'. If 'newButton' is either 0 or greater than the number of buttons in the set, then all the buttons will be switched off.
See also WhichRadioButtonOn.

WhichRBOn

```
int WhichRBOn();
```

Returns the index of the radiobutton that is switched on within the set.
See also SwitchRadioButtonSet.

Class *DICMDButton*

Class DICMDButton

```
DICMDButton( Rectangle rect, char *text=NULL,
      ButtonAction funcptr=NULL );
DICMDButton( Rectangle rect, char *text,
      ButtonAction funcptr, DIWindowObj win );
```

Constructor for the DICMDButton object. Creates a new button that executes the function 'funcptr' when clicked. 'rect' defines the size of the button, and 'text' entitles the button. The second form of the constructor allows the specification of 'win', the window to which the command button is to be added (i.e., AddObject is not required).

GetText

```
char *GetText();
```

Returns the text within the button to 'text'.
See also SetText.

GetTextButton

```
ButtonAction GetTextButton();
```

Returns a pointer to the action routine attached to the button.
See also SetButtonAction.

SetButtonAction

```
void SetButtonAction( ButtonAction funcptr );
```

Sets the action routine attached to the button.
See also GetTextButton.

SetText

```
void SetText( char *text );
```

Sets the text within the button to 'text'.
See also GetText.

---

5. DITextField

Textfields are derived from Object. The user may freely insert text anywhere within the existing string by clicking into a position and typing. Characters may also be deleted with the delete key. Areasy may be highlighted by dragging the mouse cursor over the desired area, then may be replaced or deleted by typing. The following operations, Cut, Copy, and Paste are not methods within the TextField class, but are rather operators on the class.

Cut

```
void Cut();
```

Cuts the highlighted portion of the currently highlighted text and copies it into a temporary buffer.
See also Copy, Paste.

Copy

```
void Copy();
```

Copies the highlighted portion of the currently highlighted text into a temporary buffer.
See also Paste, Cut.

Paste

```
void Paste();
```

Inserts the contents of the temporary buffer at the current cursor position.
See also Copy, Cut.

The following are the methods within the DITextField class (i.e., for DITextField objects).

Class DITextField

```
DITextField( Rectangle rect,
     BOOLEAN hasBorder = TRUE );
DITextField( Rectangle rect,  BOOLEAN hasBorder,
     DIWindowObj win );
```

Constructor for the textfield object. 'rect' specifies the rectangle of the textfield. 'hasBorder' specifies whether or not the textfield has its rectangle outlined; this only works in GDI; MGDI textfields always have borders. The second declaration allows the specification of the window to which the textfield is added (i.e., AddObject not necessary).

DeleteString

```
void DeleteString( CARDINAL start, CARDINAL end );
```

Deletes all characters from 'start' to 'end'.
See also WriteString, WriteLn, InsertString.

Erase

```
void Erase();
```

Erases the contents of the textfield.

GetFont

```
void GetFont( DIFont *font );
```

Returns (in 'font') the font in the textfield.
See also SetFont.

HasBorder

```
BOOLEAN HasBorder();
```

Returns TRUE if the textfield has a border, FALSE otherwise.

Highlight

```
void Highlight( CARDINAL start, CARDINAL end );
```

Highlights the characters from 'start' to 'end'.
See also HighlightAll.

HighlightAll

```
void HighlightAll();
```

Highlights the entire textfield.
See also Highlight.

InsertString

```
void InsertString( CARDINAL at, char *str );
```

Inserts the string 'str' at position 'at'. The current position is reset after the operation is completed to the following character.
See also WriteString, WriteLn, DeleteString.

Read

```
void Read( char *text );
```

Returns (in 'text') the current contents of the textfield.
See also Write, WriteString, WriteLn, InsertString, DeleteString.

SetFont

```
void SetFont( DIFont *font );
```

Sets the font in the textfield to 'font'.
See also GetFont.

Write

```
void Write( char ch );
```

Writes the character 'ch' at the current position in the textfield.
See also WriteString, WriteLn, InsertString, DeleteString.

WriteLn

```
void WriteLn();
```

Causes a carriage return and line field in the textfield.

WriteString

```
void WriteString( char *str );
```

Writes the text 'str' into the textfield starting at the current position.
See also Write, WriteLn, InsertString, DeleteString.

6. DISlider

A slider is a another graphical interactor. Values can be selected by moving a lider up and down the scale. A slider has two action-procedures assigned to it: one, called ValueChangeAction, is executed every time the value indicated by the slider changes (i.e., mouse dragged); the other, called FinalValueAction, is only called when the slider's value is stable (i.e., mouse is released). The following are the methods of the DISlider object.

Class DISlider

```
DISlider( char *title, double from, double to,
        double value, double step, Rectangle rect,
        DIDirection dir,
        SliderActionProcedure dragAction,
        SliderActionProcedure upAction );
DISlider( char *title, double from, double to,
        double value, double step, Rectangle rect,
        DIDirection dir,
        SliderActionProcedure dragAction,
        SliderActionProcedure upAction,
        DIWindowObj win );
```

Constructor for the DISlider object. 'title' is written next to the slider object using the font set by SetSliderTitleFont. 'from' and 'to' indicate the smallest and largest values represented by the slider. 'step' is the minimal difference between two numbers on the slider. 'rect' represents the dimensions of the slider. 'dir' is either 'vertical' or 'horizontal' representing the direction of the slider. 'dragAction' and 'upAction' represent pointers to the ValueChangeAction and FinalValueAction functions, respectively. The second definition allows for specifying which window to add the slider to (i.e., no AddObject necessary).

AssignValueChangeAction

```
void AssignValueChangeAction(
        SliderActionProcedure dragAction );
```

Sets the ValueChangeAction function for the slider. This is, by default, a no-op operation.
See also AssignFinalValueAction.

AssignFinalValueAction

```
void AssignFinalValueAction(
        SliderActionProcedure upAction );
```

Sets the FinalValueAction function for the slider. This is, by default, a no-op operation.
See also AssignValueChangeAction.

GetFont

```
void GetFont( DIFont *font );
```

Returns the font used for the title in 'font'.
See also SetFont.

GetSliderValue

```
double GetSliderValue();
```

Returns the actual current value of the slider.
See also SetSliderValue.

SetFont	`void SetFont( DIFont *font );` Sets the font used for the title. See also GetFont.
SetSliderValue	`void SetSliderValue( double val );` Sets the actual current value of the slider to 'val'. IMPORTANT NOTE: this function must never be used within the FinalValueAction procedure assigned to the slider since they would call each other infinitely. See also GetSliderValue.

7. DIUtilities

DIUtilities contains some hi-level routines based on the graphics operations in other modules. Prepend the following routines with "DIUtilities." to call them.

CFade

`void CFade( Object *o );`

Fades out the rectangular area of an object. CFade is called internally when an object is disabled.
See also Deactivate, Disable.

CText

`void CText( Object *o, char *text );`

Writes centered text onto the object's rectangle.

Frame

`void Frame( Object *o );`

Draws a rectangular frame around the object 'o'.
See also Highlight.

Highlight

`void Highlight( Object *o );`

Inverts the rectangular area of object 'o'.
See also Frame.

8. DIText

DIText deals with text manipulation within graphic windows. Text can be placed with various colors and fonts. Individual fonts can be queried for character heights and widths. To set a new font, LoadFont must first be called, (using the ascii name of the font in the X system), then SetFont to set it within the current window. Text placement position is determined by DIGraphicWindows.MoveTo. Prepennd the following operations with "DIText.".

CharHeight

`int CharHeight( char c );`

Returns the height in pixels of the character 'c' in the current font.
See also CharWidth.

CharWidth

`int CharWidth( char c );`

Returns the width in pixels of the character 'c' in the current font.
See also CharHeight.

DrawChar

`void DrawChar( char c );`

Write the character 'c' in the current window, with the current font, and current text color.
See also DrawString.

DrawString

`void DrawString( char *s );`

Write the string 's' in the current window, with the current font, and current text color.
See also DrawString.

GetFont

`void GetFont( DIFont *font );`

Sets 'font' to the current font.
See also SetFont, LoadFont.

GetTextColor

`Color GetTextColor( void );`

returns the text color in the current window.
See also SetTextColor.

LoadDefaultFont

`void LoadDefaultFont( DIFont *font );`

Sets 'font' to the default font for the X screen.
See also GetFont, SetFont, LoadFont.

LoadFont

`void LoadFont( char *fname, DIFont *font, int *res );`

Given 'fname' as an X font name, LoadFont loads it in and returns a pointer to the DIFont structure in 'font'. 'res' is 0 if an error (e.g., font not found) occurs. Note: this does not set the current font.
See also GetFont, SetFont.

SetFont

`void SetFont( DIFont *font );`

Sets the current font to 'font'.
See also GetFont, LoadFont.

SetTextColor

```
void SetTextColor( Color color );
```

Sets the text color in the current window to 'color'.

See also GetTextColor.

9. DIMenus

Prepend the following operations with "DIMenus.".

ActivateMenu	`void ActivateMenu( Menu *menu, Object *w );` Makes 'menu' available to the interactive user attached to window 'w' if it is an active window. See also DeactivateMenu.
AddItem	`void AddItem( MenuItem *item, char *itemName, MenuAction action, Menu menu );` Adds the item with name 'itemName' to 'menu' and assigns the procedure 'action' to it as the action to be taken when this item is selected. 'item' is used to reference this item in RemoveItem. See also RemoveItem.
DeactivateMenu	`void DeactivateMenu( Menu *menu, Object *w );` Makes 'menu' unavailable by deactivating it See also ActivateMenu.
DisposeMenu	`void DisposeMenu( Menu *menu );` Prevents 'menu' from being selectable any more. If 'menu' is still active, it will be deactivated. 'menu' is set to an invalid value. See also NewMenu.
NewMenu	`void NewMenu( Menu *menu, char *title );` Creates a menu with the given 'title'. 'menu' is an identifier which can be used to reference the menu later on. See also DisposeMenu.
RemoveItem	`void ActivateMenu( MenuItem *item, Menu menu );` Removes 'item' from 'menu'. See also AddItem.

10. Sample Code

The following is a simple sample program (For more, see the Appendix). It creates four command buttons ("Cut", "Copy", "Paste", and "Exit") as well as two textfields. The text in the MGDI textfields can be edited and copied from one textfield to another (or to any other X-window). Although a very simple example, this program demonstrates the organization of a typical GDI (or MGDI) program. The first section of the code includes all the necessary GDI function and object declarations (".h" files). Next come the definitions of the GDI objects (rectangle, window, commandbuttons, textfields, font) and their layout. These objects are then activated (DIObjects.Activate) in the body of main(). Activating an object displays it within its window and announces it to the event-handler so as events lying within that object's area are directed to it.

The 'while' statement near the end of the program is the "event-handling loop." That is, the call DIObjects.HandleNextEvent() handles the next event to any of the GDI objects (e.g., mouse click, mouse drag, etc). That call is continually made until a boolean variable (exitt) is set (by the function ExitProc() which is called when the command button "Exit" is pressed).

```
#include "DIObjects.h"
#include "DIGraphicWindows.h"
#include "DIButtons.h"
#include "DITextFields.h"
#include "DITerminate.h"

void ExitProc();
/* Define the objects & their layout */
Rectangle r;
DIWindowObj win( r.R(100, 100, 345, 255 ), hasBorder|hasTitle|mayBeMoved, "MDemo" );
DICMDButton cut_button( r.R( 25, 130, 70, 30 ), "Cut", &Cut, &win ),
         copy_button( r.R( 130, 130, 70, 30 ), "Copy", &Copy, &win ),
         paste_button(r.R( 235, 130, 70, 30 ), "Paste", &Paste, &win ),
         exit_button( r.R( 245, 205, 70, 30 ), "Exit", &ExitProc, &win );
DITextField tf1( r.R( 30, 30, 265, 25 ), &win );
DITextField tf2( r.R( 30, 85, 265, 25 ), &win );
BOOLEAN exitt = FALSE;
DIFont font1;

void ExitProc()
{   exitt = TRUE; }                    /* Sets 'exitt' to TRUE so as event-handling loop exits */

/* main program */
void main()
{
   int res;
    DIObjects.Show( &win );            /* Display main window */

   /* Activate the buttons */
   DIObjects.Activate( &cut_button );
   DIObjects.Activate( &copy_button );
   DIObjects.Activate( &paste_button );
   DIObjects.Activate( &exit_button );

   /* Activate the textfields */
   DIText.LoadFont( "couro14", &font1, &res );
   tf1.SetFont( &font11 );
   DIObjects.Activate( &tf1 );
   tf1.WriteString( "Edit " );
   tf2.SetFont( &font1 );
   DIObjects.Activate( &tf2 );
   tf2.WriteString( "Here" );

   while( !exitt )                     /* Start the event-handling loop */
      DIObjects.HandleNextEvent();

   DITerminate.Terminate();            /* Shut down */
}
```

11. Acknowledgements

The members of the MGDI project team were: Mark Salem, Paul Moosman, and Rob Nelson. The GDI Editor was written by Kenin Page. Finally, and most importantly, Sue Skowronski ported GDI from Modula-2 on Suntools to C++ on X-windows. The GDI Toolbox has developed dramatically since its inception at ETH Zurich by Dr. Beat Bruderlin. It has been used as a teaching tool in various computer science classes at the University of Utah including CS202 (Modula-2) and CS431-433 (computer graphics & geometric modeling sequence). GDI, although stable, is developing to span more and more platforms currently including X-windows, Motif, Suntools, and Macintosh {1, p. 35}.

The following people contributed to the development of GDI (and its predecessor in Modula-2, DI): Scott McGee, Loren Larsen, Michael Moore, and Mark Salem. If you are interested in receiving a copy of GDI, please contact one of the following:

Dr. Beat Bruderlin	bdb@cs.utah.edu	(801) 581-4944
Mark Salem	msalem@peruvian.utah.edu	(801) 581-3753

12. Appendix

The MGDI Project

The MGDI (Motif GDI) project consisted of the transparent porting of GDI onto Motif. GDI had already been ported onto the X-windows window system. However, using such a low-level graphics system, the interaction objects did not look very attractive. Motif [illegible]
"widgets" (Motif interaction objects) with three-dimensional appearance which are also built on top of X-windows. Porting GDI onto Motif made it feasible to produce more professional-looking applications than formerly possible. Because of GDI's already-existing clean separation between the tools and the underlying window system, the port would be concentrated in the low-level, system dependent code.

GDI had been used for the length of the academic year in the instruction of Computer Graphics (CS431-CS433) at the University of Utah. For this reason, one of the main project goals was making MGDI absolutely compatible with GDI. Therefore, MGDI would work with GDI code with no changes at all needed.

The Structure of MGDI

MGDI, like GDI, consists of four system-dependent modules: DIObjects, DIGraphicWindows, DIText, and DIMenus. These modules are almost identical in both GDI and MGDI. The main difference is DIWindowObj which is the interaction object for a graphical window. In GDI, this object used a low-level X-window call to open a window. In contrast, MGDI uses a higher-level call to Motif to create a similar X-window that has a motif structure associated with it. This change was necessary since the Motif widgets may only be added to Motif windows (which are really X-windows at the low-level).

The high level objects reside in three modules: DIButtons, DISliders, and DITextFields. These modules were redesigned since the objects they contained were replaced by Motif widgets. No low-level calls to create such widgets existed in GDI; therefore, these functions had to be implemented from scratch using low-level graphics and event-handling functions.

In MGDI, each of the GDI standard objects is mapped onto a Motif widget. However, the window object is a little more complicated. Each DIWindowObj object requires an application shell (an Xt Intrinsics structure) that enables each window to have its own independent tree of widgets attached to it. Below the application shell, a MainWindowWidget is created as a child. Within each MainWindowWidget is a BulletinBoardWidget and a MenuBarWidget. The BulletinBoardWidget is the equivalent of the GDI DIWindowObj -- a plain drawing area. The MenuBarWidget is for future attachment of pulldown menus to MGDI. All MGDI standard and user-objects are attached as children of the BulletinBoardWidget.

Another necessary change between GDI and MGDI is the initialization of the Xt Intrinsics library. This code was added in the X-window initialization module of GDI. Furthermore, since the Xt initialization procedure can parse command-line arguments, this capability is now inherently a part of MGDI. The format of the command-line arguments accepted follows the standard X-windows style (See the X-windows Reference Manual).

Event-Handling in MGDI

Event-handling was a major issue in this project. The chief problem centered around the fact that both Motif and GDI had their own event-handlers [illegible]

[illegible]

implemented above the Xt Toolkit.

the objects in the particular toolkit interacted with the user. The first approach we tried with this problem was to capture each event, decide if it was directed to a Motif widget or a user object, and then send the event to the appropriate event-handler. Motif's event-handler could return a TRUE if it handled the widget (i.e., there's a widget where the event happened) and FALSE otherwise. This method was used to determine whether or not the event was directed to a Motif widget.

Thinking that this was a successful approach, each event was sent to the Motif event-handler; if a TRUE was returned, it was assumed that the event had been sent to the appropriate method within a Motif widget; otherwise, the event was not directed to a Motif widget and would be sent to the GDI event-handler. After implementing this algorithm, it was discovered that the Motif Event handler never returned FALSE to MGDI, implying that all events were "valid" to Motif. After investigation, it was discovered that since the window itself was a Motif widget, any events within the bounds of the window were considered "valid" by the Motif event-handler even if the event did not occur within the bounds of a widget within that window. The final solution to this problem involved capturing each event and sending a copy to each event-handler. This solved the problem very easily since each handler, upon receiving an "invalid" event, merely discarded it.

The following are a few extra sample programs that demonstrate some of the capabilities of (M)GDI. Moreover, a few diagrams present a graphical layout of both GDI and MGDI.

Sample Program: *ButtonTest*

```
/* This illustrates the following in the (M)GDI system:
    1- Check boxes
    2- Radio buttons
    3- Command buttons
    4- Drawing background graphics via window draw()
*/

#include "DIObjects.h"
#include "DIText.h"
#include "DIGraphicWindows.h"
#include "DIUtilities.h"
#include "buttons.h"
#include <stdio.h>


Point cpt( 20, 50 );

main_window::main_window( Rectangle r, SET attributes, char *title )
: DIWindowObj( r, attributes, title )
{
   int stat;

   draw_method_replaced = TRUE;
   DIText.LoadFont( "*-courier-bold-r-*-240-*", &big_font, &stat );
   DIText.LoadFont( "*-times-medium-i-*-140-*", &small_font, &stat );
}

void main_window::draw()
{
   Rectangle r( 0, 0, rect.right-rect.left, cpt.v );
   Object o( r );

   DIGraphicWindows.SetPaintMode( replace );

   /* Draw Title */
   DIText.SetFont( &big_font );
   DIText.SetTextColor( DIGraphicWindows.Yellow );
   DIUtilities.CText( &o, "Button Test" );

   /* Label CheckBox. */
   DIText.SetFont( &small_font );
   DIText.SetTextColor( DIGraphicWindows.Red );
   DIGraphicWindows.MoveTo( cpt.h + 20, cpt.v );
   DIText.DrawString( "This is a CheckBox" );
   /* Label Radio Buttons. */
   DIText.SetTextColor( DIGraphicWindows.Cyan );
   DIGraphicWindows.MoveTo( cpt.h + 20, cpt.v + 50 );
   DIText.DrawString( "RadioButton #1" );
   DIGraphicWindows.MoveTo( cpt.h + 20, cpt.v + 75 );
   DIText.DrawString( "RadioButton #2" );
}

void exitproc();
void check_buttons();
void btestproc( double d );
void btestproc2();

BOOLEAN exitt = FALSE;
int button_num = 1;

Rectangle rect( 10, 10, 300, 500 );
SET wattrs = hasBorder| hasTitle | mayBeClosed | mayBeChanged;
main_window mwin( rect, wattrs, "Test" );

DICheckBox cb1( cpt );
DIRadioButtonSet rb1( cpt.h, cpt.v + 50, 7 );
Rectangle r1( cpt.h, cpt.v + 100, cpt.h + 50, cpt.v+120 );
DICMDButton cmdb1( r1.R( cpt.h+55, cpt.v+100, 50, 20 ), "Enable", &check_buttons );
DICMDButton cmdb2( r1.R( cpt.h+105, cpt.v+100, 50, 20 ), "*QUIT*", &exitproc );
DICMDButton cmdb3( r1.R( cpt.h+155, cpt.v+100, 50, 20 ), "Disable", &btestproc2 );

Rectangle rect1( 160, 250, 170, 350 );


main( int, char** )
{
   /* Make window show up, then activate event handlers */
   /* DIObjects.Show( &mwin ); */
   DIObjects.Activate( &mwin );

   /* Check Box. */
   /* Make CheckBox */
   DIObjects.AddObject( &cb1, &mwin );
   cb1.SwitchCheckBox( TRUE );
   DIObjects.Activate( &cb1 );

  /* Make Radio buttons */
   DIObjects.AddObject( &rb1, &mwin );
   DIObjects.Activate( &rb1 );

   /* Make Command Button */
   DIObjects.AddObject( &cmdb1, &mwin );
   DIObjects.Activate( &cmdb1 );
   DIObjects.AddObject( &cmdb2, &mwin );
   DIObjects.Activate( &cmdb2 );
   DIObjects.AddObject( &cmdb3, &mwin );
   DIObjects.Activate( &cmdb3 );
   do
   {
           DIObjects.HandleNextEvent();
   } while ( !exitt );
}


void btestproc2()
{
  DIObjects.Disable( &cb1 );
  DIObjects.Disable( &rb1 );
  DIObjects.Hide( &cmdb2 );
}


void btestproc( double d )
{
  int numberOfButtons;
  numberOfButtons = int( d ) / 10;
  rb1.SetNumButtons( numberOfButtons );
}

void exitproc()
```

```
{
   exitt = TRUE;
}

void check_buttons()
{
   fprintf( stderr, "\nButton Status:\n" );
   fprintf( stderr, "\tCheckBox: " );
   if ( cb1.IsOn() )
          fprintf( stderr, " ON\n" );
   else
          fprintf( stderr, " OFF\n" );

   fprintf( stderr, "\tRadio Button: %d\n", rb1.WhichRadioButtonOn() );

   DIObjects.Enable( &cb1 );
   DIObjects.Enable( &rb1 );
   DIObjects.Show( &cmdb2 );
}
```

Sample Program: *Slider*

```
#include "DIObjects.h"
#include "DIGraphicWindows.h"
#include "DIMenus.h"
#include "DIText.h"
#include "DISliders.h"
#include "DITerminate.h"
#include "DIButtons.h"

/* Sample object to change with slider bar,
 * should really be in separate file.
 */

class myRect : public Object
{
 public:
   myRect( Rectangle );
   Point centre;
 private:
   virtual void draw();
};

void noop2();
Point cpt( 20, 50 );
BOOLEAN exitt = FALSE;
void slidertest();
void exitproc();
void rectChange( double d );

int sliderL = 10;
int sliderT = 50;
int sliderR = 35;
int sliderB = 200;

Rectangle rect1( sliderL, sliderT, sliderR, sliderB );
DISlider slider( "Percent Size", 0.0, 100.0, 50.0, 15.0, 5, 0, 100, 50, rect1);

Rectangle r1( cpt.h, cpt.v + 100, cpt.h + 50, cpt.v+120 );
DICMDButton cmdb1( r1.R( cpt.h+65, cpt.v+115, 60, 45 ), "Test", &slidertest );
DICMDButton cmdb2( r1.R( cpt.h+135, cpt.v+115, 60, 45 ), "Quit", &exitproc );

Rectangle rect2( 140, 50, 240, 150 );
myRect mr( rect2 );

main ( int, char** )
{
   Rectangle r( 100, 10, 410, 300 );
   SET wattrs = hasBorder| hasTitle | mayBeClosed | mayBeChanged;
   DIWindowObj win( r, wattrs, "Slider Demo" );

   /* Make window show up. */
   DIObjects.Activate( &win );
   DIObjects.Show( &win );
```

```
    DIObjects.AddObject( &slider, &win );
    DIObjects.Activate( &slider );

/* Set action routines for slider bar. (optional) */
    slider.AssignValueChangeAction( &rectChange );
    slider.AssignFinalValueAction( &rectChange );

    DIObjects.AddObject( &cmdb1, &win );
    DIObjects.Activate( &cmdb1 );

    DIObjects.AddObject( &cmdb2, &win );
    DIObjects.Activate( &cmdb2 );

    DIObjects.AddObject( &mr, &win );
    DIObjects.Activate( &mr );

    do
    {
      DIObjects.HandleNextEvent();
    } while ( !exitt );
    DITerminate.Terminate();
}


void
noop2()
{
}


void slidertest()
{
   int value;
   printf(" Current slider value is %f \n", slider.GetSliderValue());
   printf(" Please input a value to set the slider to : ");
   scanf( "%d", &value );
   slider.SetSliderValue( (double) value );
}


void
exitproc()
{
    exitt = TRUE;
}

/* This routine causes the rectangle to grow & shrink when the
 * slider bar is moved.
 */
void
rectChange( double d )
{
    Rectangle r;
    int radius;

    /* Undraw old rectangle */
    mr.GetRectangle( r );
    DIGraphicWindows.SetPaintMode( invert );
    DIGraphicWindows.DrawRectangle( r );


    /* reconstruct rectangle with slider value d as radius. */
    radius = (int) d/2;
    r.left =   mr.centre.h - radius;
    r.right =  mr.centre.h + radius;
    r.top =    mr.centre.v - radius;
    r.bottom = mr.centre.v + radius;
    mr.SetRectangle( r );
    DIGraphicWindows.SetPaintMode( invert );
    DIGraphicWindows.DrawRectangle( r );
}

myRect :: myRect( Rectangle r )
{
    SetRectangle( r );
    centre.h = (r.left + r.right) / 2;
   centre.v = (r.top + r.bottom) / 2;
   draw_method_replaced = TRUE;
}

void
   myRect :: draw()
{
   Rectangle r;

   GetRectangle( r );
   DIGraphicWindows.SetPaintMode( invert );
   DIGraphicWindows.DrawRectangle( r );
}
```

Sample Program: *MovingRectangle*

```
/* This sample program illustrates 2 main points:
    1- How to create one's own object and attach event handlers
    2- How to create a simple menu
*/

#include "DIObjects.h"
#include "DIGraphicWindows.h"
#include "DIMenus.h"
#include "DITerminate.h"
#include "DIButtons.h"
#include <stdio.h>

/* Includes pattern for rectangle. */
#include "xxxs.bmp"
Bitmap xxxs_pattern;


class moving_obj : public Object
{
 public:
   int x0, y0, x1, y1;
   Rectangle r;
   void down( int x, int y );
   void drag( int x, int y );
   void up( int x, int y );
   void draw();
   moving_obj( Rectangle );
};

void exitproc();
void noopproc();
BOOLEAN exitt;

Rectangle r;
DICMDButton button( r.R( 10,10, 80,50 ), "QUIT", &exitproc);

main( int*, char** )
{
   DIGraphicWindows.Fg = DIGraphicWindows.Red;
   Rectangle r( 245, 245, 265, 265 );
   moving_obj mo( r );
   SET wattrs = hasBorder; hasTitle | mayBeClosed | mayBeChanged;
   DIWindowObj win( r.R( 10, 10, 300, 300 ), wattrs, "Moving Parts" );
   Menu main;
   MenuItem mitem1, mitem2;

   /* Make window show up. */
   DIObjects.Show( &win );

   /* Initialize oattern for moving rectangle. */
   xxxs_pattern =
         DIGraphicWindows.DefineBitmap( xxxs_width, xxxs_height, xxxs_bits );

   /* Make rectangle which "moves" */
   DIObjects.AddObject( &mo, &win );
   DIObjects.Activate( &mo );
```

```
    DIObjects.AddObject( &button, &win );
    DIObjects.Activate( &button );



    /* Set up Menus */
    DIMenus.NewMenu( &main, "EXIT" );
    DIMenus.AddItem( &mitem1, "Exit", &exitproc, main );
    DIMenus.AddItem( &mitem2, "No Op", &noopproc, main );
    DIMenus.ActivateMenu( main, &win );

    do
    {
       DIObjects.HandleNextEvent();
    } while ( !exitt );
    DITerminate.Terminate();
}

void exitproc()
{
    exitt = TRUE;
}

void noopproc()
{
    fprintf( stderr, "Boo!\n" );
}


moving_obj::moving_obj( Rectangle r )
{
    rect = r;
    x0 = y0 = x1 = y1 = 0;
    up_method_replaced = TRUE;
    drag_method_replaced = TRUE;
    down_method_replaced = TRUE;
}

void moving_obj::draw()
{
    Rectangle r;

    DIGraphicWindows.SetPaintMode( paint );
    DIGraphicWindows.SetBorderMode( noBorder );
    DIGraphicWindows.SetFillPattern( xxxs_pattern );
    GetRectangle( r );
    DIGraphicWindows.FillRectangle( r );
}

void moving_obj::down( int x, int y )
{
    GetRectangle( r );
    DIGraphicWindows.SetPaintMode( erase );
    DIGraphicWindows.FillRectangle( r );
    x0 = x1 = x;
    y0 = y1 = y;
    DIGraphicWindows.SetPaintMode( invert );
    DIGraphicWindows.FillRectangle( r );
}

void moving_obj::drag( int x, int y )
{
    Rectangle r2;

    DIGraphicWindows.FillRectangle( r2.R( r.left+x1-x0, r.top+y1-y0,
                                            r.right-r.left, r.bottom-r.top) );

    x1 = x;  /* redraw at new place */
    y1 = y;
    DIGraphicWindows.FillRectangle( r2.R( r.left+x1-x0, r.top+y1-y0,
                                            r.right-r.left, r.bottom-r.top) );
}

void moving_obj::up( int x, int y )
{
    Rectangle r2;
```

```
   DIGraphicWindows.FillRectangle( r2.R( r.left+x1-x0, r.top+y1-y0,
                                              r.right-r.left, r.bottom-r.top) );
   DIObjects.Move( this, x-x0, y-y0 );
}
```

Sample Program: *DrawLines*

```
#include "DIObjects.h"
#include "DIGraphicWindows.h"
#include "DIMenus.h"
#include "line.h"


/***********************************************************
 * TAG( my_window_obj::my_window_obj )
 *
 * Create a default construction of my_window_obj.  To change
 * some of the attributes later, use objects::SetRectangle,
 * DIWindowObj::SetAttributes & DIWindowObj::SetTitle.  These
 * must be changed BEFORE adding window to display list.
 */
my_window_obj::my_window_obj()
: DIWindowObj()
{
   p1.h = p1.v = 0;
   p2.h = p2.v = 0;

   /* Set the proper event-handler flags. */
   down_method_replaced = TRUE;
   drag_method_replaced = TRUE;
   up_method_replaced = TRUE;
}

/***********************************************************
 * TAG( my_window_obj::my_window_obj )
 *
 * Constructor for my_window_obj.  Arguments are the same as listed
 * in DIGraphicWindows.h for DIWindowObj.
 */
 my_window_obj::my_window_obj( Rectangle rect, SET attributes, char *title )
: DIWindowObj( rect, attributes, title )
{
   p1.h = p1.v = 0;
   p2.h = p2.v = 0;

   /* Set the proper event-handler flags. */
   down_method_replaced = TRUE;
   drag_method_replaced = TRUE;
   up_method_replaced = TRUE;
}

/***********************************************************
 * TAG( my_window_obj::down )
 *
 * This sets the event handler which is activated when the mouse
 * button is pressed down within this window. This draws an initial
 * line at the given point, which will be changed as the line is
 * "rubberbanded".
 */
void my_window_obj::down( int h, int v )
{
   p1.h = p2.h = h;
   p1.v = p2.v = v;
   DIGraphicWindows.SetPaintMode( invert );
   DIGraphicWindows.DrawLine( p1, p2 );
}

/***********************************************************
 * TAG( my_window_obj::drag )
 *
 * This is the event-handler which is activated when the mouse
 * button is pressed and the mouse is moved, or dragged within
 * the window. Rubberbanding takes place here; the old line
 * is erased and a new one is drawn based on the last point.
 */
void my_window_obj::drag( int x, int y )
{
   DIGraphicWindows.DrawLine( p1, p2 );
   p2.h = x;
   p2.v = y;
   DIGraphicWindows.DrawLine( p1, p2 );
}

/*************************************************************
 * TAG( my_window_obj::up )
 *
 * This event-handler is activated when the left mouse button
 * is released after being pressed within the window.  The previous
 * line is erased, and a permanent one is drawn.  Also,
 * a permanent line object is created based on this final line.
 */
void my_window_obj::up( int x, int y )
{
   line_obj *line;

   DIGraphicWindows.DrawLine( p1, p2 );
   p2.h = x;
   p2.v = y;
   DIGraphicWindows.SetPaintMode( paint );
   DIGraphicWindows.DrawLine( p1, p2 );

   /* Create line object. */
   line = new line_obj( p1, p2 );
   DIObjects.AddObject( line, this );
   DIObjects.Activate( line );
}


/*************************************************************
 * TAG( line_obj::line_obj )
 *
 * Default function for making line_obj. Use access functions
 * to change.
 */
 line_obj::line_obj()
{
   p1.h = p2.h = 0;
   p1.v = p2.v = 0;
   move_method_replaced = TRUE;
}

/*************************************************************
 * TAG( line_obj::line_obj )
 *
 * Another line_obj constructor.
 */
 line_obj::line_obj( Point pt1, Point pt2 )
{
  Rectangle r;

  p1 = pt1;
  p2 = pt2;
  set_rect();
  move_method_replaced = TRUE;
}

/*************************************************************
 * TAG( line_obj::draw )
 *
 * The draw event handler.  Is called when the area of the
 * window that line is on is redrawn.
 */
void line_obj::draw()
{
   DIGraphicWindows.SetPaintMode( paint );
   DIGraphicWindows.DrawLine( p1, p2 );
}

/*************************************************************
 * TAG( set_rect )
 *
 * This sets/resets the bounding rectangle of this line.
 */
void line_obj::set_rect()
{
   Rectangle r;

   r.left   = p1.h;
   r.top    = p1.v;
   r.right  = p2.h;
   r.bottom = p2.v;
   SetRectangle( r );
}

/*************************************************************
 * TAG( set_p1 )
 *
 * Set point 1 in line.
 */
void line_obj::set_p1( Point p )
{
    p1 = p;
    set_rect();
}
/*************************************************************
 * TAG( set_p2 )
 *
 * Set point 2 in line.
 */
void line obj::set_p2( Point p )
{
    p2 = p;
    set rect();
}

/*************************************************************
 * TAG( line_obj::move )
 *
 * Move event handler. This is just a test to see how "move" works.
 */
void line obj::move( int, int )
{
   fprintf( stderr, "Line move event.\n" );
}
BOOLEAN quit = FALSE;
void exitproc();
void front_proc();
void back_proc();

Rectangle r( 25, 25, 425, 325 );
SET attributes = hasBorder | hasTitle ;
my_window_obj my_window( r, attributes, "Line Sketcher" );

main( int, char** )
{
   Menu main, prio;
   MenuItem mitem1, mitem2;
   MenuItem prio_item1, prio_item2;

   /* Make window show up. Since this window has event handlers,
    * we also need to "activate" them. */
   DIObjects.Show( &my window );
   DIObjects.Activate( &my window );

   /* Set up Menus */
   DIMenus.NewMenu( &main, "EXIT" );
   DIMenus.AddItem( &mitem1, "Exit", &exitproc, main );
   DIMenus.AddItem( &mitem2, "Nothing Done Here.", NULL, main );
   DIMenus.ActivateMenu( main, &my_window );

   DIMenus.NewMenu( &prio, "WINDOW PRIORITY" );
   DIMenus.AddItem( &prio_item1, "Window in front", &front_proc, prio );
   DIMenus.AddItem( &prio_item2, "Window in back", &back_proc, prio );
   DIMenus.ActivateMenu( prio, &my_window );

   /* Let system handle events. */
   do
   {
          DIObjects.HandleNextEvent();
   } while ( !quit );
}

void exitproc()
{
   quit = TRUE;
}

void front_proc()
{
   DIObjects.SetPriority( &my_window, first );
}

void back_proc()
{
   DIObjects.SetPriority( &my_window, last );
}
```

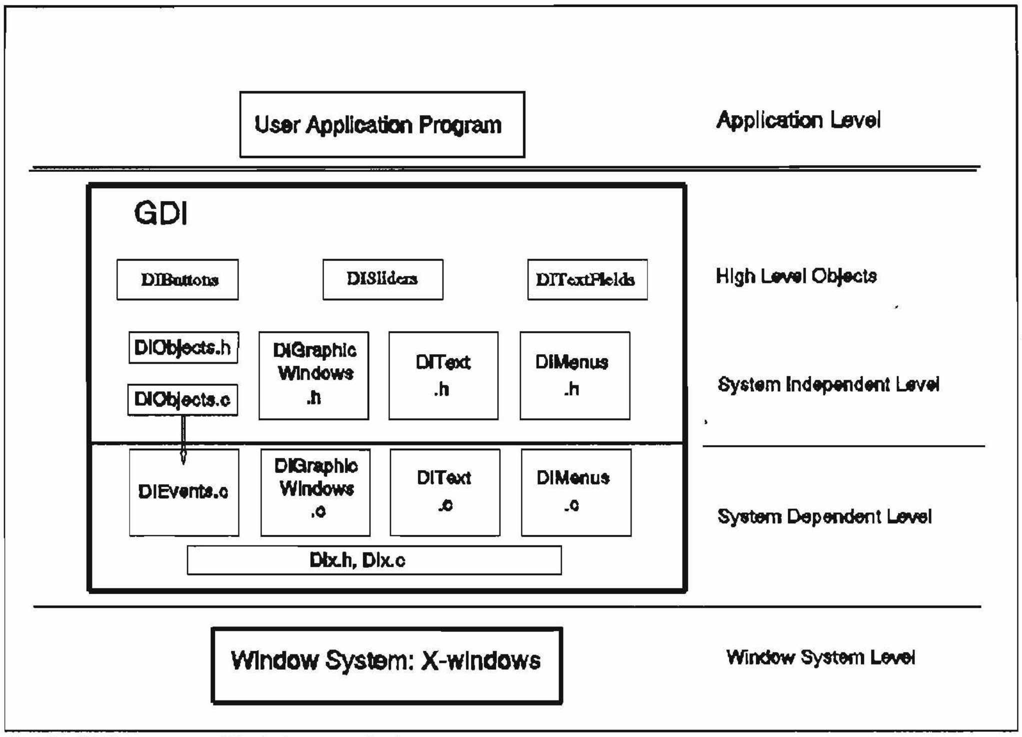

Fig. 1. GDI structure (X-windows version).

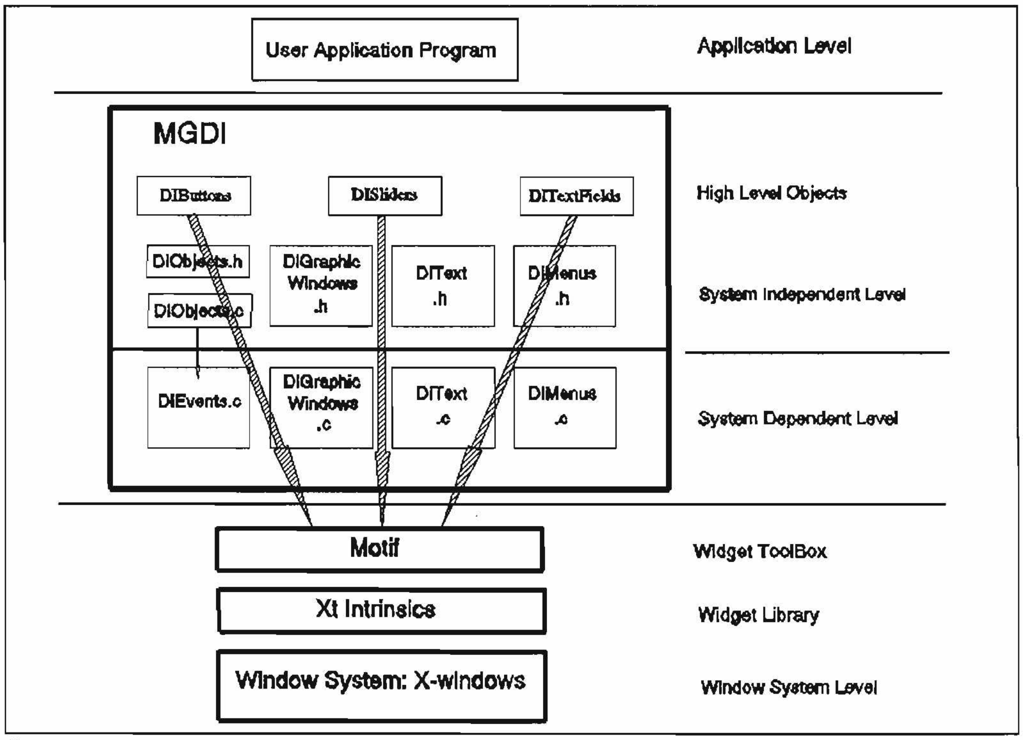

Fig. 2. MGDI structure (Motif version).

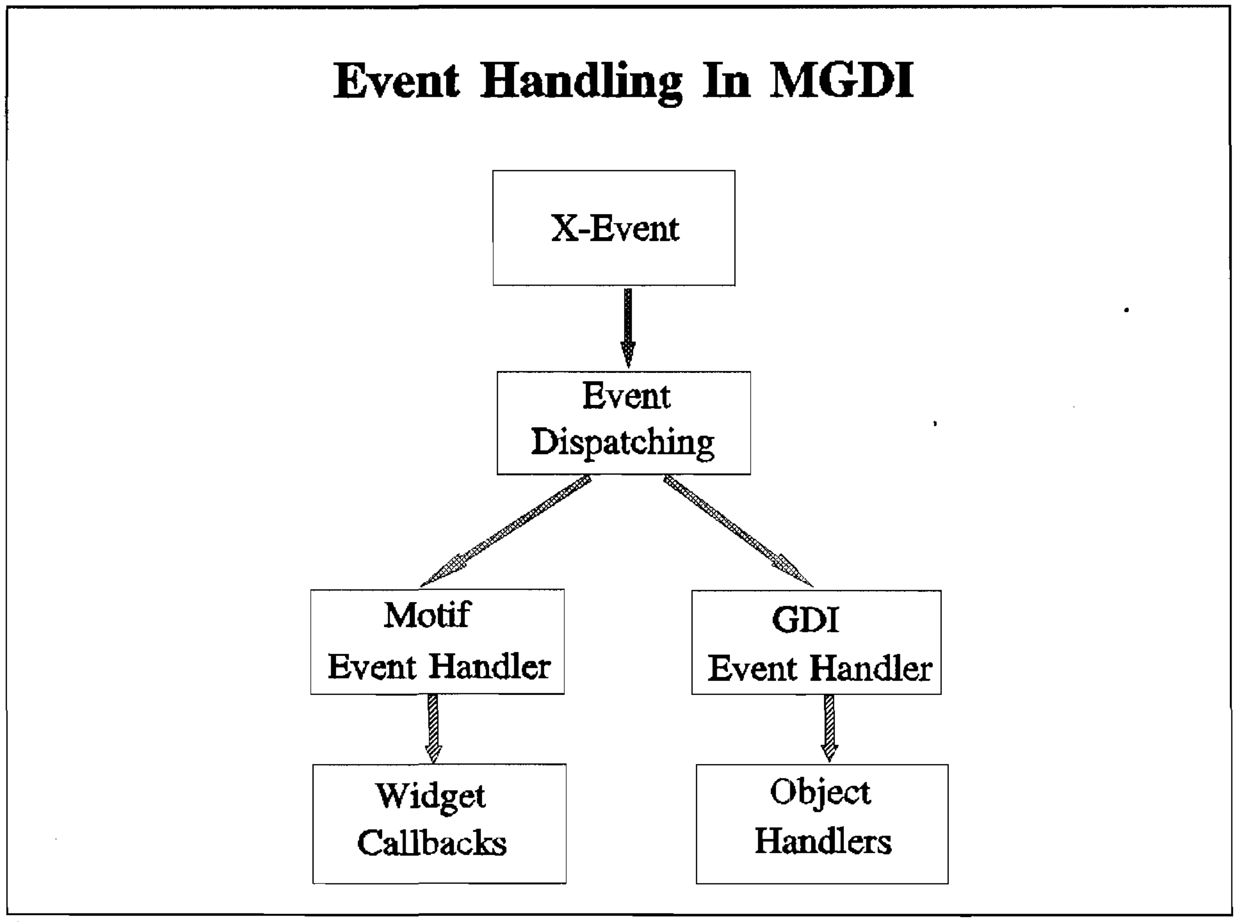

Fig. 3. Event-handling in MGDI.

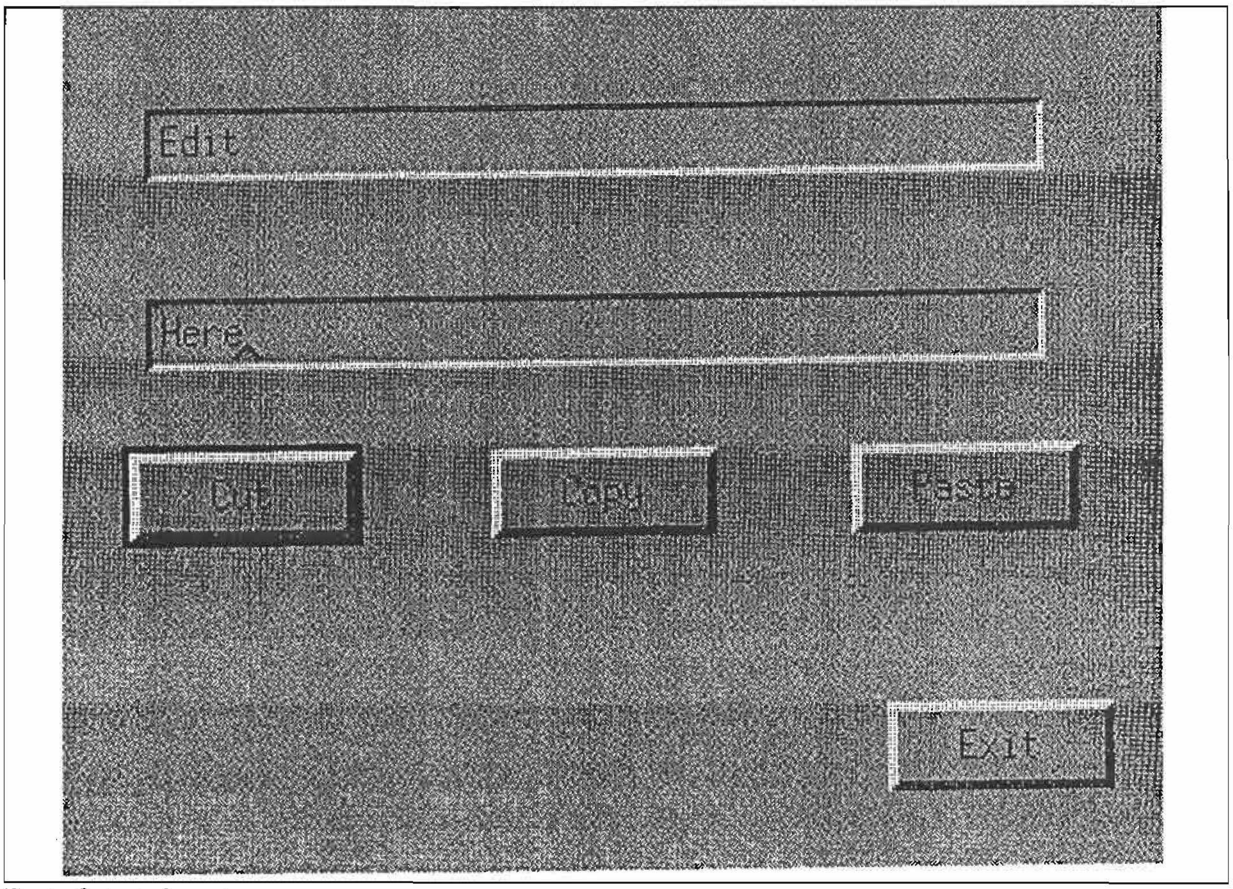

Fig. 4. Output of sample program.

13. References

1. Bruderlin, B. and McGee, S. DI: an object-oriented user interface toolbox for Modula-2 applications. Tech. Rep. UUCS-90-005, Computer Science Dept., University of Utah, 1990.

2. Cardelli, L. Building user interfaces by direct manipulation. In Proceedings of the ACM SIGGRAPH Symposium on User Interface Software, ACM Press, 1988, 152-166.

3. OSF/Motif programmer's reference, revision 1.0. The Open Software Foundation, Inc. Prentice-Hall, Inc. 1988.

4. Salem, M., and Bruderlin B, "GDI: A Portable, Object-Oriented User Interface Toolbox", Jounral of Undergraduate Research, vol. III. University of Utah, 1992.

14. Index