Contents

11

Windows and Panels

The data type *window* is the base type for all visualisation and animation support in the LEDA system. It provides an interface for the graphical input and output of basic geometric objects for both the *X11* system on Unix platforms and for Microsoft *Windows* systems.

An instance W of type *window* is a rectangular window on the display screen. The width w and height h of W are measured in pixels and can be defined in the constructor. The default constructor initializes the width and height of W to default values depending on the system and screen resolution of the display. The position on the display is given by the pixel coordinates of the upper left corner of W. It can be specified in the *display* operation.

A window consists of two rectangular regions, a *panel section* in the upper part and a *drawing section* in the rest of the window. Either section may be empty. The panel section contains *panel items* such as sliders, choice fields, string items, and buttons. They have to be created by the operations described in Section 11.14 before the window is displayed for the first time. Figure 11.1 shows a typical LEDA window. If a window has no drawing section we call it a *panel*. Figure 11.2 shows the LEDA panel used for the *xlman* manual reader.

The drawing section can be used to draw geometric objects such as points, lines, segments, arrows, circles, polygons, graphs, ... and to input any of these objects using the mouse input device.

In this chapter we discuss LEDA windows and show how to use them in demo and visualization programs.

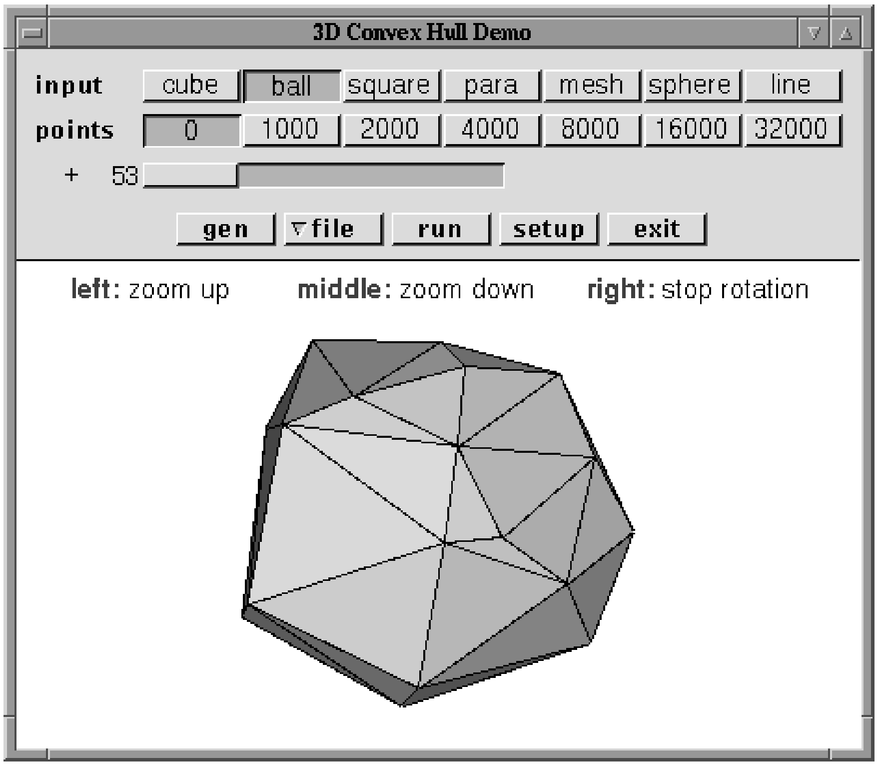


Figure 11.1 A typical LEDA window.

Figure 11.2 A typical LEDA panel: xlman.

11.1 Pixel and User Coordinates

The underlying graphics systems (X11 or Windows) maps windows to rectangular regions of the display screen using a pixel based coordinate system. In this *pixel coordinate system*, the upper left corner of the window rectangle has coordinates $(0, 0)$, x-coordinates increase from left to right, and y-coordinate increase from top to bottom. This is illustrated in Figure 11.3.

All drawing and input operations in the drawing section use the *user coordinate system* whose y-axis is oriented in the usual mathematical way, i.e., from bottom to top. The

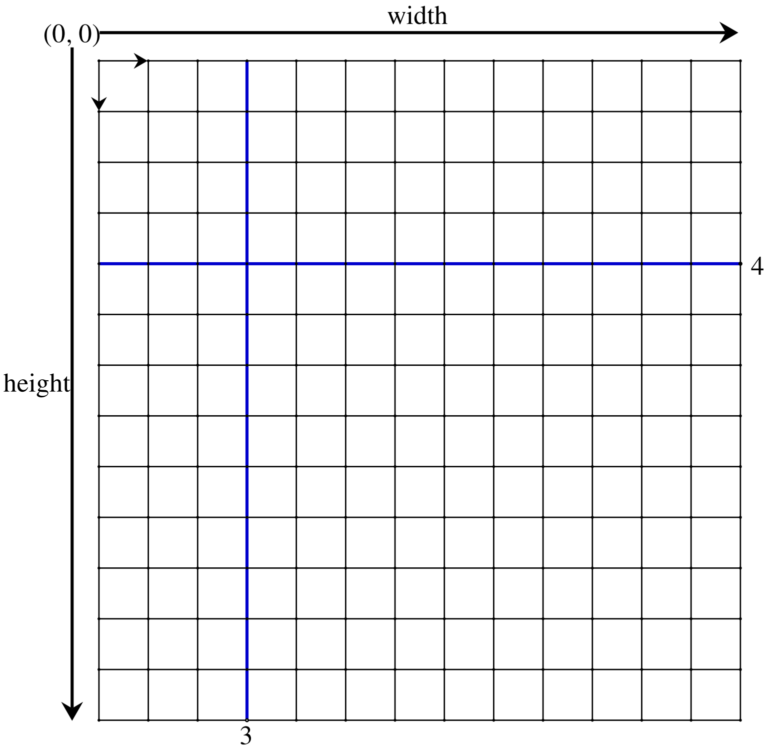


Figure 11.3 The pixel coordinate system: the pixel with coordinates $(3, 4)$.

user coordinate system is defined by three numbers of type *double*: *xmin*, the minimal x-coordinate, *xmax*, the maximal x-coordinate, and *ymin*, the minimal y-coordinate. The two parameters *xmin* and *xmax* define the scaling factor

$$scaling = w/(xmax - xmin),$$

where w is the width of the window in pixels. The maximal y-coordinate *ymax* of the drawing section is equal to $ymin + h \cdot scaling$, where h is the height of the drawing section in pixels. The user coordinates (x, y) correspond to the pixel

$$(scaling \cdot (x - xmin), scaling \cdot (y - ymin)).$$

The window type provides operations for translating user coordinates into window coordinates and vice versa.

11.2 Creation, Opening, and Closing of a Window

We describe how to create, open, and close a window.

Figure 11.4 The LEDA default icon.

```
window W;
```

creates a window of default size.

```
window W(int w, int h);
```

creates a window W of size $w \times h$ pixels.

```
void W.display();
```

opens W and displays it at the default position on the screen. Note that W.display() has to be called before all drawing operations and that all operations adding panel items to W (cf. Section 11.14) have to be called before the first call of W.display().

```
void W.display(int x, int y);
```

opens W and displays it with its left upper corner at position (x, y) in pixel coordinates. The three special constants *window*::*min*, *window*::*center*, *window*::*max* can be used for positioning W at the minimal or maximal x- or y-coordinate or centering it horizontally or vertically on the screen.

```
void W.display(window W0, int x=window::center, int y=window::center);
```

opens W and displays it at position (x, y) above window *W0* which must be displayed already.

```
void W.iconify();
```

closes W and displays it as a small icon. If no user-defined icon is specified (see the *icon pixrect* parameter) the LEDA default icon, as shown in Figure 11.4, is used.

```
void W.close();
```

closes W and removes it from the display.

11.3 Colors

The data type *color* represents all colors available in drawing operations.

Each color value corresponds to a triple of integers (r, g, b) with $0 \leq r, g, b \leq 255$, the so-called *rgb-value* of the color. The number of available colors is restricted and depends on the underlying hardware. A color can be created from rgb-values,

```
color col(int r, int g, int b);
```

from a color name in a system data base (X11 only)

```
color col(string color_name);
```

or from one of the integer color constants defined in <*LEDA/impl/x_window.h*>

```
color col(int color_const);
```

where *color_const* is one of the constants from the enumeration

```
enum { black, white, red, green, blue, yellow, violet,
       orange, cyan, brown, pink, green2, blue2,
       grey1, grey2, grey3, ivory, invisible }
```

A drawing operation with the special color *invisible* has no effect on the display.

The definition of a color may fail due to one of the following reasons:

- There is a system dependent limitation on the total number of different colors any application may use and the construction exceeds this limit.
- One of the specified (r, g, b)-values is illegal, i.e., not in the range $[0, \ldots, 255]$.
- The color name is not present in the systems color data base or the system does not support this method of specifying colors.

If the definition of a color fails, we say that the constructed color is *bad*; it is called *good* otherwise. The operation

```
bool col.is_good()
```

tests whether a color is good or bad.

It is also possible to retrieve the (r, g, b)-values of a color by

```
void col.get_rgb(int& r, int& g, int& b);
```

The following program tries to construct all 256 possible grey colors and reports how many of them are available.

⟨*greyscales.c*⟩≡

```
#include <LEDA/window.h>
#include <LEDA/array.h>
main()
{
  array<color> grey(256);
  int n = 0;
```

```
  for(int i = 0; i < 256; i++)
  { color c(i,i,i);
    if (c.is_good()) grey[n++] = c;
   }
  cout << n << " different greys available." << endl;
  return 0;
}
```

Exercises for 11.3

1 How man different versions of “red” are available on your system? Write a program to find out.

2 Write a program that displays a rainbow.

11.4 Window Parameters

Every window has a list of parameters which control its appearance and the way drawing operations are performed on the window. In this section we will first survey the available window parameters and then show how to read and to change them.

The Available Parameters: We list the parameters together with their type, default value, and a short description of their meaning.

background color: A parameter of type *color* (default value *white*) defining the default background color (e.g., used by $W.clear(\,)$ to erase the drawing area).

background pixrect: A parameter of type *char*∗ (default value: *NULL*) defining a pixrect (see Section 11.8) that is used to tile the background of the window. If it is different from *NULL* the background color parameter is ignored.

foreground color: A parameter of type *color* (default value: $black$) defining the default color to be used in all drawing operations. All drawing operations have an optional color argument that can be used to override the default foreground color temporarily.

mouse cursor: A parameter of type *int* (default value: -1) defining the shape of the mouse cursor. Its value must be either the default value or one of the values listed in <*LEDA/X11/cursorfont.h*>.

text font: A parameter of type string (default value: system dependent) defining the name of the font to be used in text drawing operations. Possible values are strings of the form: *T*<*num*>, *F*<*num*>, *I*<*num*>, and *B*<*num*>. Here *T* stands for (normal) text, *F* for fixed size, *I* for italic, and *B* for bold, and num gives the font size in points. These special names are used by the window class to provide a platform independent way of specifying fonts. For example, ”*B14*” specifies a “usual” 14pt bold font of the underlying operating system. Note, however, that, in general, a font specified in this way will look different for different platforms. On Unix systems fonts can also be specified by an X11 font name as for instance `-adobe-helvetica-medium-r-*-*-14-*-*-*-*-*-*-*`.

window coordinates (xmin, xmax, ymin): Parameters of type *double* (default values: $(0, 100, 0)$) defining the user coordinate space of the window, i.e., *xmin* is the minimal x-coordinate, *xmax* the maximal x-coordinate, and *ymin* the minimal y-coordinate of the drawing area. The maximal y-coordinate *ymax* depends on the shape and size of the drawing area.

grid width: A parameter of type *int* (default value: 0) defining the width of the grid used in the drawing area. A grid width of 0 indicates that no grid is to be used.

grid style: A parameter of type *grid_style* (default value: *point_grid*) defining how a grid is represented in the window. Possible values are *invisible_grid*, *point_grid*, and *line_grid*.

frame label: A parameter of type *string* (default value: LEDA header) defining the frame label of the window that is used by the graphics system or window manager.

icon label: A parameter of type *string* (default value: empty) defining the icon label of the window.

icon pixrect: A parameter of type *char∗* (default value: *NULL*) defining a pixrect (see Section 11.8) that is used as the icon of the window. If it has value *NULL* the default icon is used.

show coordinates: A parameter of type *bool* (default value: *false*) determining whether the current coordinates of the mouse pointer are displayed in the upper right corner of the window.

line width: A parameter of type *int* (default value: 1) defining the width of all kinds of lines (segments, arrows, edges, circles, polygons) in pixels.

line style: A parameter of type *line_style* (default value: *solid*) defining the style of all kinds of lines. Possible styles are *solid*, *dashed*, *dotted*, and *dashed_dotted*.

node width: A parameter of type *int* (default value: 10) defining the diameter of nodes created by the *draw_node* and *draw_filled_node* operations.

text mode: A parameter of type *text_mode* (default value: *transparent*) defining how text is inserted into the window. Possible values are *transparent* and *opaque*.

drawing mode: A parameter of type *drawing_mode* (default value: *src_mode* defining the logical operation that is used for setting pixels in all drawing operations. Possible values are *src_mode* and *xor_mode*. In *src_mode* pixels are set to the respective color value, in *xor_mode* the value is bitwise added to the current pixel value.

clip region: A parameter defining the clipping region of the window, i.e., the region of the window to which drawing operations are applied (default value: the entire drawing area). In the current implementation clip regions are restricted to rectangles (defined by *set_clip_rectangle*) and ellipses (defined by *set_clip_ellipse*).

redraw function: A parameter of type *void* (∗*func*)(*window*∗) (default value: NULL).

Its value is a pointer to a function that is called with a pointer to the corresponding window, whenever a redrawing of the window is necessary, e.g., if the shape of the window is changed or previously hidden parts of the window become visible.

client data: A parameter of type *void*$*$ (default value: NULL). Its value is an arbitrary pointer value that can be set or read by client applications. In most cases it is used to associate user-defined data with a window for use in *redraw* or other call-back functions.

buttons per line: A parameter of type *int* (default value: ∞) defining the maximal number of buttons in one line of the panel section.

Reading and Changing Parameters: Most parameters may be retrieved or changed by *get* and *set* functions. We use *param* to denote any of the window parameters and *param_t* to denote its type.

```
param_t W.get_param()
```

returns the current value of parameter *param*, and

```
param_t W.set_param(param_t val)
```

sets the value of parameter *param* of type *param_t* to the new value *val* and returns the former value of the parameter.

Here are some simple examples:

```
line_style = W.get_line_style();
int lw = W.get_line_width();
W.set_cursor(XC_dotbox);
W.set_bg_pixrect(leda_pixmap);
W.set_grid_dist(10);
W.set_grid_style(line_grid);
W.set_line_width(1)
W.set_bg_color(ivory)
W.set_color(blue)
W.set_redraw(redraw_func);
```

The fact that the *set*-operation returns the old value of the parameter is very convenient when a parameter is to be changed only temporarily. For instance, in order to change the mouse cursor to a “watch symbol” during the execution of a time consuming operation, one writes:

```
int old_cursor = W.set_cursor(XC_watch);
// some time consuming computation
W.set_cursor(old_cursor);
```

There are a few operations for changing parameters that do not follow the scheme described above, e.g., the *init* operation for changing the user coordinate system that is explained in the next section.

11.5 Window Coordinates and Scaling

We discuss the connection between coordinates and pixels. We use w and h for the width and the height of the drawing section in pixels. Both values are determined by the appearance of a window on the screen. The coordinate system underlying the drawing area is defined by the *init* operation.

```
void W.init(double x0, double x1, double y0, int grid_dist=0);
```

defines the coordinate system underlying the drawing area of W by setting *xmin* to x_0, *xmax* to x_1, and *ymin* to y_0. It also defines implicitly a scaling factor *scaling* and the maximal y-coordinate *ymax* of the drawing area.

$$scaling = w/(xmax - xmin) \quad \text{and} \quad ymax = ymin + h \cdot scaling.$$

If, in addition, a *grid_dist* argument is supplied, it is used to initialize the grid distance of the window. The following function give information about the window coordinates and the scaling factor:

```
double W.xmin()
```

returns *xmin*, the minimal x-coordinate of the drawing area of W, i.e., the coordinate of the left window border in user space. The analogous functions $W.xmax(\,)$, $W.ymin(\,)$, and $W.ymax(\,)$ are also available.

```
double W.scale()
```

returns the scaling factor of the drawing area of W, i.e. the number of pixels of a unit length line segment in user space.

```
double W.pix_to_real(int p)
```

translates pixel distances into user space distances, more precisely, returns the length of a p pixel horizontal or vertical line segment in the user coordinate system.

```
double W.real_to_pix(double d)
```

translates user space distances into pixel distances, more precisely, returns the number of pixels contained in a horizontal or vertical line segment of length d.

11.6 The Input and Output Operators $\ll$ and $\gg$

For the input and output of basic two-dimensional geometric objects of the floating point kernel (*point*, *segment*, *ray*, *line*, *circle*, *polygon*) the $\ll$ and $\gg$ operators can be used. In analogy to C++ input streams, windows have an internal state indicating whether there was more input to read or not. The state is true initially and is turned to false if an input sequence is terminated by clicking the right mouse button (similar to ending stream input by the *eof*-character). In conditional statements, objects of type *window* are automatically converted to boolean by returning this internal state. Thus, window-objects can be used in conditional statements in the same way as C++ input streams. For example, to read a sequence of points terminated by a right button click, use

```
while (W >> p) { .... }
```

The following program uses the $\gg$ operator to read points defined by mouse clicks and draws each point using the $\ll$ operator until input is terminated by clicking the right mouse button.

⟨*draw_points.c*⟩≡

```
#include <LEDA/window.h>
main()
{
  window W(400,400);
  W.display(window::center,window::center);
  point p;
  while (W >> p) W << p;
  W.screenshot("draw_points.ps");
}
```

Graphical input and output for LEDA windows can be extended to user-defined types by overloading the $\ll$ and $\gg$ operators. This is in analogy to C++ stream input and output. For example, <*LEDA/rat_window.h*> contains input and output operators for the objects of the rational kernel.

```
window& operator<<(window& W, const rat_point& p)
{ return W << p.to_point(); }
window& operator>>(window& W, rat_point& p)
{ point q;
  W >> q;
  p = rat_point(q);
  return W;
 }
```

Exercises for 11.6

1 Modify the program *draw_points.c* such that segments (circles, line, or polygons) are echoed. The modified program is supposed to work for only one of the mentioned objects.

2 Write operators $\ll$ and $\gg$ for *rat_polygon*s.

11.7 Drawing Operations

The $W \ll$ *object* output operators apply to the basic objects of the floating point kernel. The windows class also provides a large number of additional drawing operations that give more flexibility. In this book we can only give a few examples. For the complete list of operations we refer the reader to the LEDA User Manual.

There are two kinds of drawing operations

```
void W.draw_object(coords, color col=window::fg_color);
void W.draw_object(object, color col=window::fg_color);
```

For the first variant, a geometric object is given by its coordinates in the user coordinate system of the window, and for the second variant, the object is given as an object of the floating point kernel. For example,

```
W.draw_circle(double x, double y, double r, color col);
```

draws a circle with center (x, y) and radius r,

```
W.draw_polygon(list<point> P, color col);
```

draws a polygon with vertex sequence P,

```
W.draw_circle(circle C, color col);
```

draws the circle C, and

```
W.draw_polygon(polygon P, color col);
```

draws the polygon P.

The allowed objects are points, pixels, segments, lines rays, ellipses, circles and disks, triangles (unfilled and filled), polygons (unfilled and filled), rectangles and boxes, arcs, Bezier curves, splines, arrows, text, nodes, and edges. The window data type can draw many more types of objects than are available in the geometry kernel. For these types only the first variant exists that takes an explicit coordinate representation as input.

The optional color argument at the end of the parameter list can be used to specify a color that is to be used as foreground color by the operation. If it is omitted the current value of the foreground color parameter (cf. Section 11.4) is used.

The clear operation erases the window by painting it with the background color or tiling it using the background pixrect (if defined).

```
void W.clear();
void W.clear(double x0, double y0, double x1, double y1);
```

The second variant only clears rectangle (x_0, y_0, x_1, y_1).

Exercises for 11.7

1 Write a program that draws a red circle, a green line segment, and a blue filled polygon.

2 Write a program that draws a filled box for each available shading of grey.

11.8 Pixrects and Bitmaps

Pixrects and bitmaps are rectangular regions of pixels and bits, respectively.

11.8.1 *Pixrects*

Pixrects (often called pixmaps) are rectangles of pixels of a certain width and height. Each pixel has a color value from the possible set of colors available in the underlying graphics system. In this way pixrects represent rectangular pictures.

There are operations to copy a pixrect into a rectangle of the drawing area of a displayed window of the appropriate size and to construct a pixrect from a rectangle of the drawing area. Pixrects can also be constructed from external representations of pictures stored in *xpm* files or *xpm* data strings. xpm data strings are of type $char**$, i.e., they are represented by arrays of C++_ strings. An xpm file contains the (C++) definition of an xpm data string, see Figure 11.5 for an example. For the exact definition of the xpm format we refer the reader to one of the *X11* handbooks or manuals [Nye93]. LEDA provides a small collection of icon pictures stored in xpm files in the <*LEDA/pixmaps/button32*> directory. A typical X11 system provides tools for the construction and manipulation of xpm files.

In the current implementation of LEDA pixrects and bitmaps are not realized by real data types but by pointers (of type $char*$). In particular, there is no constructor and destructor, i.e., the user must explicitly create and destroy pixrects or bitmaps by calling *create* and *destroy* operations.

Constructing and Destroying Pixrects: We discuss functions for constructing and destroying pixrects.

```
char* W.create_pixrect(double x0, double y0, double x1, double y1)
```

constructs a pixrect of all pixels contained in the rectangle (x_0, y_0, x_1, y_1) of the drawing area of W and returns it.

```
char* W.get_window_pixrect()
```

constructs a pixrect of all pixels in the drawing area of W and returns it.

```
char* W.create_pixrect(char** xpm)
```

constructs a pixrect from the xpm pixmap data string xpm.

```
char* W.create_pixrect(string xpm_file)
```

constructs a pixrect from the xpm pixmap data in file *xpm_file*.

```
void W.del_pixrect(char* prect)
```

destroys pixrect *prect*.

Drawing Pixrects: We discuss the functions for drawing picrects.

```
void W.put_pixrect(double x, double y, char* prect)
void W.put_pixrect(point p, char* prect)
```

copies the pixels of pixrect *prect* into a rectangle of the drawing area of W which is placed with its left lower corner at the specified position of the drawing area.

~~~~
<<xpm_example_file.h>>=

/* XPM */
static char *example_xpm[] = {
/* width height ncolors chars_per_pixel */
"32 32 6 1",
/* colors */
"` c #000000",
"a c #F5DEB3",
"b c #E6E6FA",
"c c #DBDBDB",
"d c #CC9933",
"e c #FFFFCC",
/* pixels */
"cccccccccccccccccccccccccccccccc",
"cccccccccccccccccccccccccccccccc",
"cccccccccccc````cccccccccccccccc",
"ccccccccc```bebe``c```cccccccccc",
"ccccccc``bebebebeb`beb`c`ccccccc",
"cccccc`ebebebebebebebeb`b`cccccc",
"cccccc`bebebebebeb`bebebeb`ccccc",
"ccccccc`bebe``b```bebebebe`ccccc",
"cccccccc```b```dd``beb`bebe`cccc",
"ccccccccc```ddddddd````ebebe`ccc",
"ccccc``````dddddddddddd`ebeb`ccc",
"cccc``aaa``d`ddd`ddd`dd```be`ccc",
"ccc``aaaa``dd`ddd`ddd`d```eb`ccc",
"ccc`aaa````dd`ddd`ddd`d``eb`cccc",
"ccc`aa`cc``dd`ddd`ddd`d``be`cccc",
"ccc`aa`cc``dd`ddd`ddd`d``eb`cccc",
"ccc`aa`cc``dd`ddd`ddd`d``b`ccccc",
"ccc`aa`cc``dd`ddd`ddd`d``e`ccccc",
"ccc`aa`cc``dd`ddd`ddd`d``b`ccccc",
"ccc`aa`cc``dd`ddd`ddd`d``e`ccccc",
"ccc`aa`cc``dd`ddd`ddd`d``b`ccccc",
"ccc`aa`cc``dd`ddd`ddd`d``e`ccccc",
"ccc`aaa````dd`ddd`ddd`d``b`ccccc",
"ccc``aaaa``dd`ddd`ddd`d``e`ccccc",
"cccc``aaa``dd`ddd`ddd`d```cccccc",
"ccccc``````dd`ddd`ddd`d``ccccccc",
"ccccccccc``d`dd``dd``dd``ccccccc",
"ccccccccc``dddddddddddd``ccccccc",
"ccccccccc````````````````ccccccc",
"cccccccccc``````````````cccccccc",
"cccccccccccccccccccccccccccccccc",
"cccccccccccccccccccccccccccccccc"
};
~~~~

Figure 11.5 A pixrect stored in *xpm* format.

```
void W.center_pixrect(double x, double y, char* prect)
void W.center_pixrect(point p, char* prect)
```

copies the pixels of pixrect *prect* into a rectangle of the drawing area of *W* that is placed with its center at the specified position of the drawing area.

In the following example we construct a pixrect representing the LEDA icon and put it (with its lower left corner) at positions defined by mouse clicks. Figure 11.6 shows a screenshot.

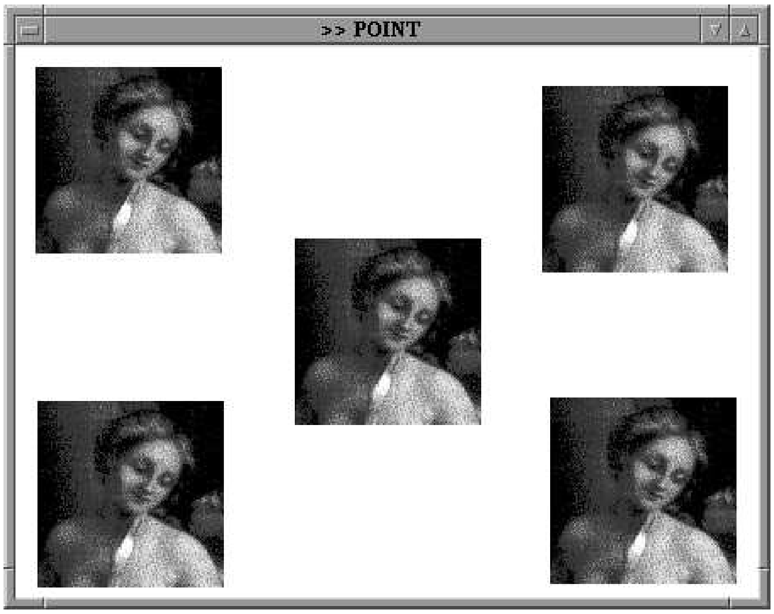


Figure 11.6 A screenshot of the put_pixrect program.

⟨*put_pixrect.c*⟩≡

```
#include <LEDA/window.h>
#include <LEDA/pixmaps/leda_icon.xpm>
main()
{
  window W(400,400);
  W.display();
  char* pr = W.create_pixrect(leda_icon);
  point p;
  while (W >> p) W.put_pixrect(p,pr);
  W.del_pixrect(pr);
  W.screenshot("put_pixrect.ps");
  return 0;
}
```

11.8.2 ***Bitmaps***

Bitmaps are pixrects containing pixels of only two possible colors: black and white. The name indicates that each pixels in a bitmap can be represented by a single bit and that is exactly the way bitmaps are usually represented: by a triple (w, h, s), where w and h give the width and height of the bitmap and s is a string of bits (of type *char∗*). A file that contains the (C++) definition of such a string is called a bitmap file. Usually the suffix *xbm* (x bit map) is used for such a file. LEDA provides a small collection of bitmap pictures stored in *xbm* files in the <*LEDA/bitmaps/button32.h*> directory. As for pixmaps there are many programs for constructing and manipulating *xbm* files.

Bitmap Operations:

```
char* W.create_bitmap(int w, int h, char* xbm)
```

creates a bitmap of width w and height h from the bits in the xbm string xbm. The length of xbm must be at least $w \cdot h$ bits, i.e., $\lceil (w \cdot h)/8 \rceil$ characters.

```
void W.put_bitmap(double x, double y, char* bmap, color c)
void W.put_bitmap(point p, char* bmap, color c)
```

places the bitmap *bmap* with its left lower corner at the specified position of the drawing area and draws with color c all pixels in the drawing area that correspond to a pixel of *bmap* with value one.

```
void W.del_bitmap(char* bmap)
```

destroys bitmap *bmap*.

The following program is very similar to the last example program but uses a bitmap instead of a pixrect. First, we construct a bitmap representing the LEDA icon and put it (with its lower left corner) at positions defined by mouse clicks.

⟨*bitmap.c*⟩≡

```
#include <LEDA/window.h>
#include <LEDA/bitmaps/leda_icon.xbm>
main()
{
  window W(400,400);
  W.set_bg_color(yellow);
  W.display();
  // construct bitmap from the bitmap data in
  // <LEDA/bitmaps/leda_icon.xbm>
  char* bm = W.create_bitmap(leda_icon_width, leda_icon_height,
                             (char*)leda_icon_bits);
  // copy copies of bm into the window
  point p;
  while (W >> p) W.put_bitmap(p.xcoord(),p.ycoord(),bm,blue);
  W.del_bitmap(bm);
  W.screenshot("bitmap.ps");
  return 0;
}
```

Exercises for 11.8

1 Write a program that converts a bitmap into a pixrect.
2 Construct a pixrect containing your picture.
3 What is shown in the pixrect of Figure 11.5

11.9 Clip Regions

Sometimes it is necessary to limit the effect of a drawing operation to some restricted area, a so-called *clipping region* of the window. The following operations allow us to define clipping regions.

```
void W.set_clip_rectangle(double x0, double y0, double x1, double y1);
```

sets the clipping region to rectangle (x_0, y_0, x_1, y_1).

```
void W.set_clip_ellipse(double x0, double y0, double r1, double r2);
```

sets the clipping region to the ellipse with center (x_0, y_0), horizontal radius r_1 and vertical radius r_2.

```
void W.reset_clipping();
```

resets the clipping region to the entire drawing area of the window.

We give an example for the usefulness of clipping. We show how to fill a circle with a pixrect picture. In this situation, we have to restrict the effect of a *put_pixrect* operation to the interior of this circle. This can be done by defining a corresponding clip-ellipse. Here is the program and the resulting picture (Figure 11.7).

$\langle$*clip_pixrect.c*$\rangle \equiv$

```
#include <LEDA/window.h>
#include <LEDA/pixmaps/leda_icon.xpm>

void draw_pix_circle(window& W, const circle& C, char* prect)
{
  point  p = C.center();
  double x = p.xcoord();
  double y = p.ycoord();

  double r = C.radius();

  W.draw_disc(C,black);

  W.set_clip_ellipse(x,y,r,r);
  W.center_pixrect(x,y,prect);
  W.reset_clipping();
}

main()
{
   window W(400,400, "Clipping a Pixmap");
   W.display();

   // create a pixrect using LEDA's xpm icon

   char* leda_pix = W.create_pixrect(leda_icon);

   circle c;
   while (W >> c) draw_pix_circle(W,c,leda_pix);

   W.del_pixrect(leda_pix);

   W.screenshot("clip_pixrect.ps");
   return 0;
}
```

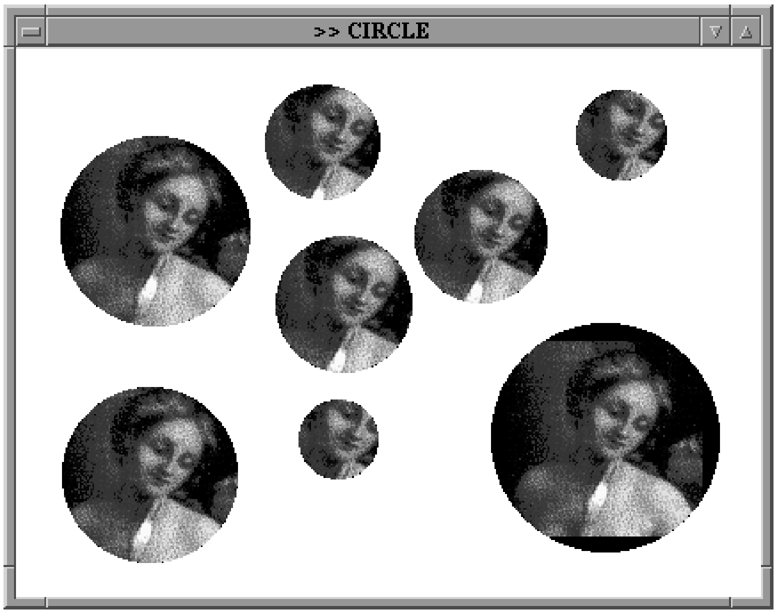


Figure 11.7 A screenshot demonstrating the effect of clip regions.

11.10 **Buffering**

The default behavior of all drawing operations discussed in the preceding sections is to draw immediately into the drawing area of the displayed window. There are, however, situations where this behavior is not desired, and where it is very useful to construct an entire drawing in a memory buffer before copying it (or parts of it) into the drawing area.

Buffering allows us to draw complex objects, which require several primitive drawing operations, in a single blow. One draws the complex object into a buffer and then copies the buffer to the drawing area. In this way, the illusion is created that the entire object is drawn by a single drawing operation. The ability to draw complex objects in a single operation is frequently needed in *animations*, where one wants to display a sequence of snapshots of a scene that changes over time. Another application of buffering is to create a pixrect copy of a drawing without displaying it in the drawing area. At the end of this section we will give example programs for both applications.

These are the most important buffering operations:

```
void W.start_buffering()
```

starts buffering of window W, i.e, all subsequent drawing operations have no effect in the drawing area of the displayed window, but draw into an internal buffer with the same size and coordinates as the drawing area of W.

```
void W.flush_buffer()
```

copies the contents of the internal buffer into W.

```
void W.flush_buffer(double x0, double y0, double x1, double y1)
```

copies all pixels in the rectangle (x_0, y_0, x_1, y_1) of the buffer into the corresponding rectangle of W. This can be much faster if the rectangle is significantly smaller than the entire drawing area of W and is often used in animations when the drawing changes only locally in a small rectangular area.

```
void W.stop_buffering()
```

stops buffering and deletes the internal buffer; all subsequent drawing operations again draw into the drawing area of W. The alternative

```
void W.stop_buffering(char*& pr)
```

stops buffering and converts the internal buffer into a picrect that is assigned to *pr*.

The following program uses buffering to move the LEDA pixrect ball that was drawn by the previous example program smoothly across the window and to let it bounce at the window border lines.

⟨*buffering1.c*⟩≡

```
#include <LEDA/window.h>
#include <LEDA/pixmaps/leda_icon.xpm>

void move_ball(window& W, circle& ball, double& dx, double& dy,
                                                   char* prect)
{
  ball = ball.translate(dx,dy);

  point  c = ball.center();
  double r = ball.radius();

  if (c.xcoord()-r < W.xmin() || c.xcoord()+r > W.xmax()) dx = -dx;
  if (c.ycoord()-r < W.ymin() || c.ycoord()+r > W.ymax()) dy = -dy;

  W.clear();
  W.set_clip_ellipse(c.xcoord(),c.ycoord(),r,r);
  W.center_pixrect(c.xcoord(),c.ycoord(),prect);
  W.reset_clipping();
  W.draw_circle(ball,black);
}

main()
{
  window W(300,300, "Bouncing Leda");

  W.set_bg_color(grey1);
  W.display(window::center,window::center);

  circle ball(50,50,16);

  double dx = W.pix_to_real(2);
  double dy = W.pix_to_real(1);

  char* leda = W.create_pixrect(leda_icon);

  W.start_buffering();
  for(;;)
  { move_ball(W,ball,dx,dy,leda);
```

```
    W.flush_buffer();
   }
  W.stop_buffering();
  W.del_pixrect(leda);
  W.screenshot("buffering1.ps");
  return 0;
}
```

We next show how to use buffering to construct a pixrect copy of a drawing. The following program uses an auxiliary window *W1* in buffering mode to create a pixrect picture that is used as an icon for the primary window W.

⟨*buffering2.c*⟩≡

```
#include <LEDA/window.h>
main()
{
  window W1(100,100);
  W1.set_bg_color(grey3);
  W1.init(-1,+1,-1);
  W1.start_buffering();
  W1.draw_disc(0,0,0.8,blue);  W1.draw_circle(0,0,0.8,black);
  W1.draw_disc(0,0,0.6,yellow);W1.draw_circle(0,0,0.6,black);
  W1.draw_disc(0,0,0.4,green); W1.draw_circle(0,0,0.4,black);
  W1.draw_disc(0,0,0.2,red);   W1.draw_circle(0,0,0.2,black);
  char* pr;
  W1.stop_buffering(pr);
  window W(400,400);
  W.set_icon_pixrect(pr);
  W.display(window::center,window::center);
  point p;
  while (W >> p) W.put_pixrect(p,pr);
  W.del_pixrect(pr);
  W.screenshot("buffering2.ps");
  return 0;
}
```

Exercises for 11.10

1 Draw ten random line segments, once without buffering and once with buffering.

2 Extend the "Bouncing LEDA" program, such that the ball is compressed when it hits the boundary of the window.

11.11 **Mouse Input**

The main input operation for reading positions, mouse clicks, and buttons from a window W is the operation $W.read\_mouse(\,)$. This operation is blocking, i.e., waits for a button to be pressed which is either a “real” button on the mouse device or a button in the panel section of W. In both cases, the number of the selected button is returned. Mouse buttons have predefined numbers *MOUSE_BUTTON*(1) for the left button, *MOUSE_BUTTON*(2) for the middle button, and *MOUSE_BUTTON*(3) for the right button. The numbers of the panel buttons can be defined by the user. If the selected button has an associated action function or sub-window, this function/window is executed/opened (cf. Section 11.14 for details).

There is also a non-blocking input operation *W.get_mouse*( ), it returns the constant *NO_BUTTON* if no button was pressed since the last call of *get_mouse* or *read_mouse*, and there are even more general input operations for reading window events. Both will be discussed at the end of this section.

Read Mouse: The function

```
int W.read_mouse();
```

waits for a mouse button to be pressed inside the drawing area or for a panel button of the panel section to be selected. In both cases, the number n of the button is returned. The number is one of the predefined constants *MOUSE_BUTTON*(i) with $i \in \{1, 2, 3\}$ for mouse buttons and a user defined value (defined when adding the button with *W.button*( )) for panel buttons. If the button has an associated action function, this function is called with parameter n. If the button has an associated window M, M is opened and *M.read_mouse*( ) is returned.

The functions

```
int W.read_mouse(double& x, double& y)
int W.read_mouse(point& p)
```

wait for a button to be pressed. If the button is pressed inside the drawing area, the position of the mouse cursor (in user space) is assigned to (x, y) or p, respectively. If a panel button is selected, no assignment takes place. In either case the operation returns the number of the pressed button.

The following program shows a trivial but frequent application of *read_mouse*. We exploit the fact that *read_mouse* is blocking to stop the program at the statement *W.read_mouse*( ). The user may then leisurely view the scene drawn. Any click of a mouse button resumes execution (and terminates the program).

⟨*read_mouse1.c*⟩≡

```
#include <LEDA/window.h>
main()
{
  window W;
  W.init(-1,+1,-1);
  W.display();
  W.draw_disc(0,0,0.5,red);
  W.read_mouse();
```

```
  W.screenshot("read_mouse1.ps");
  return 0;
}
```

The next program prints the different return values of *read_mouse* for clicks on mouse and panel buttons.

⟨*read_mouse2.c*⟩≡

```
#include <LEDA/window.h>
main()
{
  window W;
  W.button("button 0"); W.button("button 1");
  W.button("button 2"); W.button("button 3");
  int exit_but = W.button("exit");
  W.display();
  for(;;)
  { int but =  W.read_mouse();
    if (but == exit_but) break;
    switch (but) {
     case MOUSE_BUTTON(1): cout << "left button click" << endl;   break;
     case MOUSE_BUTTON(2): cout << "middle button click" << endl; break;
     case MOUSE_BUTTON(3): cout << "right button click" << endl;  break;
     default:  cout << string("panel button: %d",but) << endl;    break;
    }
  }
  W.screenshot("read_mouse2.ps");
  return 0;
}
```

Get Mouse: The functions

```
int W.get_mouse()
int W.get_mouse(double& x, double& y)
int W.get_mouse(point& p)
```

are non-blocking variants of *read_mouse*, i.e., they do not wait for a mouse click, but check whether there is an unprocessed click in the input queue of the window. If a click is available, it will be processed in the same way as by the corresponding *read_mouse* operation. If there is no click, the special button value *NO_BUTTON* is returned.

The following program draws random points. It uses *get_mouse* at the beginning of every execution of the main loop to check whether a mouse button has been clicked or not. If the right button has been clicked the loop is terminated, if the left button has been clicked the drawing area is erased.

⟨*get_mouse.c*⟩≡

```
#include <LEDA/window.h>
random_source& operator>>(random_source& ran, point& p)
{ int x,y;
  ran >> x >> y;
  p = point(x,y);
  return ran;
}
main()
{
  window W(400,400);
  W.display(window::center,window::center);
  W.message("left button: clear    right button: stop");
  random_source ran(0,100);
  int but;
  while ( (but = W.get_mouse()) != MOUSE_BUTTON(3) )
  {
    if (but == MOUSE_BUTTON(1)) W.clear();
    point p;
    ran >> p;
    W.draw_point(p,blue);
  }
  W.screenshot("get_mouse.ps");
  return 0;
}
```

Exercises for 11.11

1 The following lines of code wait for a mouse click.

```
int but;
do but = W.get_mouse(); while (but == NO_BUTTON);
```

What is the difference to $but = W.read\_mouse(\,)$?

2 Write a program that implements the input operator $\ll$ for polygons.

11.12 **Events**

In window systems like the *X11* or *Windows* system, the communication between input devices such as the mouse or the keyboard and application programs is realized by so-called *events*. For example, if the mouse pointer is moved across a window, the system generates motion events that can be handled by an application program to keep track of the current position of the mouse pointer, or, if a mouse button is clicked, an event is generated that carries the information which button was pressed at what position of the mouse pointer, or, if a key is pressed, a keyboard event is triggered that tells application programs which key was pressed and what window had the input focus, i.e., should receive this character input.

Events are buffered in an *event queue* such that applications can access them in a similar way as character input of a C++ input stream. It is possible to read and remove the next event from this queue, to test whether the queue is empty, and to push events back into the queue.

LEDA supports only a restricted set of events. Each event is represented by a five-tuple with the fields type, window, value, position, and time stamp.

The *type* of an event defines the kind of input reported by this event, e.g., a click on a mouse button or pressing a key on the keyboard. Event types are specified by integers from the enumeration

```
enum {button_press_event, button_release_event, key_press_event,
      key_release_event, motion_event, configure_event, no_event}
```

The *window* of an event specifies the window to which the event refers. This is usually the window under the mouse cursor.

The *value* of an event is an integer whose interpretation depends on the type of the event, e.g., the number of a mouse button for a button press event. See below for a description of the possible values for each event type.

The *position* of an event gives the position of the mouse pointer in the user coordinate system of the window at the time the event occurred.

The *time stamp* of an event is the time of a global system clock at which the event occurred. It is measured in milliseconds.

The following event types are recognized by LEDA and can be handled in application programs:

button_press_event indicates that a mouse button has been pressed. The value of the event is the number of the pressed button. The mouse buttons are numbered *MOUSE_BUTTON*(1), *MOUSE_BUTTON*(2), and *MOUSE_BUTTON*(3).

button_release_event indicates that a mouse button has been released. The value of the event is the number of the released button.

key_press_event indicates that a keyboard key has been pressed down. The value of the event is the character associated with the key or in the case of a special key (such as a cursor or function key) a special key code.

key_release_event indicates that a keyboard key has been released, value as above.

motion_event indicates that the mouse pointer has been moved inside the drawing area. The value of this event is unspecified.

configure_event indicates that the window size has changed.

Blocking Event Input: Similar to the *read_mouse* input operation, there is a *read_event* operation that removes the first event of the system's event queue. This operation is blocking, i.e., if the event queue is empty, the program waits until a new event occurs.

```
int W.read_event(int& val, double& x, double& y, unsigned long& t)
```

waits for an event with window W (discarding all events with a different window field) and returns its type, assigns the value of the event to *val*, its position to (x, y), and the time stamp of the event to t.

```
int  W.read_event(int& val, double& x, double& y,
                            unsigned long& t, int timeout)
```

is similar, but waits (if no event for W is available) for at most *timeout* milliseconds; if no event occurs during this period of time, the special event *no_event* is returned.

The next program implements a click and drag input routine for the definition of rectangles. In its main loop the program waits for a mouse click and stores the corresponding position in a variable p by calling $W.read\_mouse(p)$. If the right button was clicked, the program terminates. Otherwise, we take p as the first endpoint of the diagonal of the rectangle to be defined, wait until the mouse button is released, say at some position q, and take q as the other endpoint of the diagonal of the rectangle. Waiting for the release of the button is implemented by the inner loop

```
while (W.read_event(val,x,y) != button_release_event) { ... }
```

This loop handles all events of window W and terminates as soon as a *button_release* event occurs. For every event processed the value of the event is assigned to *val* and the position is assigned to (x, y), in particular for motion events, the pair (x, y) keeps track of the position of the mouse pointer in the drawing area of W. In the body of the inner loop we draw the (intermediate) rectangle with diagonal from p to (x, y) as a yellow box with a black border on top of the current drawing. The current drawing is kept as a pixrect *win_buf* and is constructed by a call to *W.get_window_pixrect*( ) before the execution of the inner loop. This allows us to restore the picture without the intermediate rectangles by copying the pixels of *win_buf* into the drawing area (*W.put_pixrect*(*win_buf*)). Of course, *win_buf* has to be destroyed after the inner loop has terminated.

In addition, we use buffering as discussed in Section 11.10, to prevent any flickering effects. Figure 11.8 shows a screenshot.

⟨*event.c*⟩≡

```
#include <LEDA/window.h>
#include <math.h>
int main()
{
  window W(450,500,"Event Demo");
  W.display();
  W.start_buffering();
  for(;;)
  {
    // read the first corner p of the rectangle
    // terminate if the right button was clicked
    point p;
```

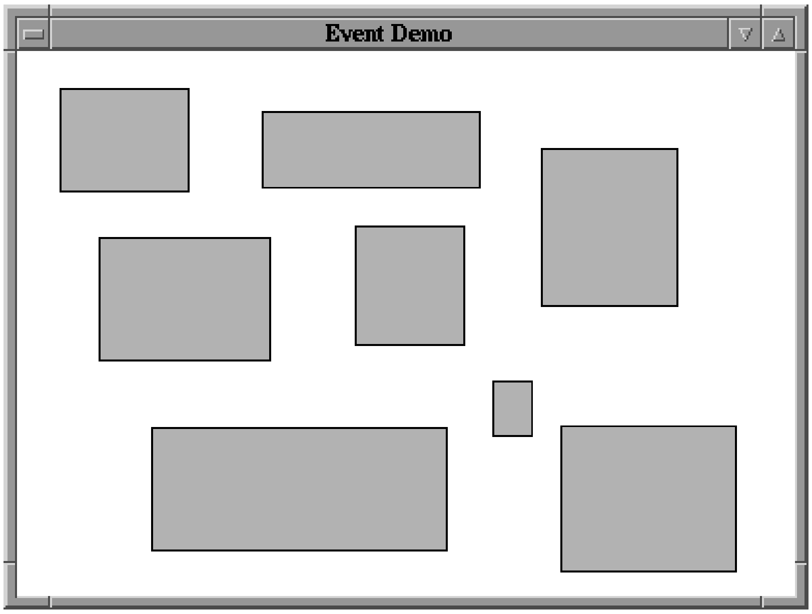


Figure 11.8 A screenshot of the Event Demo.

```
    if (W.read_mouse(p) == MOUSE_BUTTON(3)) break;
    // draw rectangle from p to current position
    // while button down
    int  val;
    double x,y;
    char* win_buf = W.get_window_pixrect();
    while (W.read_event(val,x,y) != button_release_event)
    { point q(x,y);
      W.put_pixrect(win_buf);
      W.draw_box(p,q,yellow);
      W.draw_rectangle(p,q,black);
      W.flush_buffer();
     }
    W.del_pixrect(win_buf);
  }
  W.stop_buffering();
  W.screenshot("event.ps");
  return 0;
}
```

The next example program uses the timeout-variant of *read_event* to implement a function that recognizes *double clicks*. But what is a double click?

A double click is a sequence of three button events, a button press event followed by button release event followed by a second button press event, with the property that the time interval between the two button press events is shorter than a given time limit. Usually, the time limit is given in milliseconds by a *timeout* parameter that can be adjusted by the user. In our example we fix it at 500 milliseconds.

In the program we first wait for a button press event and store the corresponding time stamp in a variable *t_press*. If the pressed button was the right button the program is terminated, otherwise, we wait for the next button release event and store the corresponding time stamp in a variable *t_release*. Now $t\_release - t\_press$ gives the time that has passed between the pressing and releasing of the button. If this time is larger than our timeout parameter we know that the next click cannot complete a double click. Otherwise, we wait for the next click but no longer than $timeout - (t\_release - t\_press)$ milliseconds. If and only if a click occurs within this time interval, we have a double click.

The program indicates double clicks by drawing a red ball and simple clicks by drawing a yellow ball. The middle button can be used to erase the window. Figure 11.9 shows a screenshot of the program.

⟨*dblclick.c*⟩≡

```
#include <LEDA/window.h>
int main()
{
  unsigned long timeout = 500;
  window W(400,400,"Double Click Demo");
  W.set_grid_dist(6);
  W.set_grid_style(line_grid);
  W.display(window::center,window::center);
  for(;;)
  {
    int b;
    double x0,y0,x,y;
    unsigned long t, t_press, t_release;
    while (W.read_event(b,x0,y0,t_press) != button_press_event);
    // a button was pressed at (x0,y0) at time t_press
    // the middle button erases the window
    if (b == MOUSE_BUTTON(2) ) { W.clear(); continue; }
    // the right button terminates the program
    if (b == MOUSE_BUTTON(3) ) break;
    while (W.read_event(b,x,y,t_release) != button_release_event);
    // the button was released at time t_release
    color col = yellow;
    // If the button was held down no longer than timeout msecs
    // we wait for the remaining msecs for a second press, if the
    // the button is pressed again within this period of time we
    // have a double click and we change the color to red.
    if (t_release - t_press < timeout)
    { unsigned long timeout2 = timeout - (t_release - t_press);
      if (W.read_event(b,x,y,t,timeout2) == button_press_event)
        col = red;
```

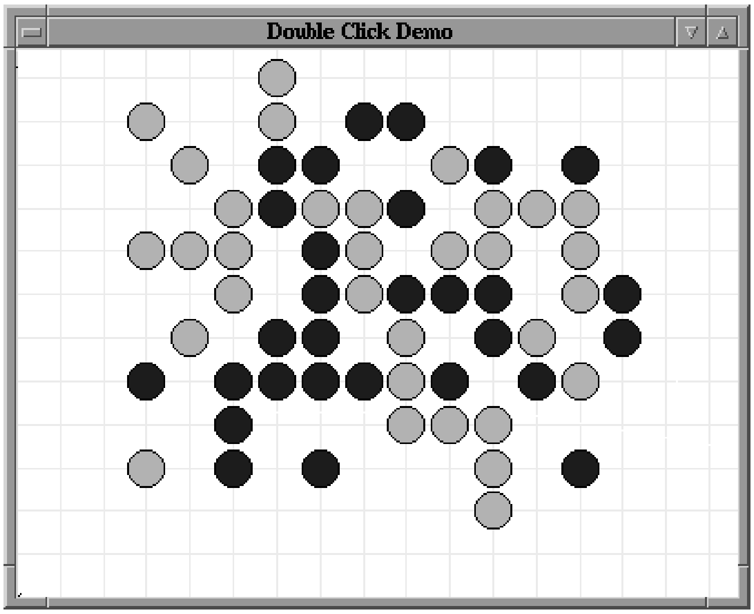


Figure 11.9 A screenshot of the double click program.

```
      }
      W.draw_disc(x0,y0,2.5,col);
      W.draw_circle(x0,y0,2.5,black);
    }
    W.screenshot("dblclick.ps");
    return 0;
  }
```

Putting Back Events: The function[1]

```
void put_back_event();
```

puts the event handled last back to the system's event queue, such that it will be processed again by the next *read_event* or *read_mouse* or basic input operation.

The function is very useful in programs that have to handle different types of input objects using the basic input operators. We give an example. We partition the drawing area of a window into four quadrants and want to draw points in the first, segments in the second, circles in the third, and polygons in the fourth quadrant. The kind of object to be drawn is defined by the position of the first mouse click. The main loop of the program waits for a mouse click and performs, depending on the quadrant that contains the position of this click, the corresponding input and output operation. The difficulty is that already the first

[1] Observe that this function is a global function and not a member function of class *window*.

click that we use to distinguish between the different input objects is part of the definition of the object.

We use the *put_back_event*( ) function to push the first mouse click back into the event queue and to make it available as the first event for the following basic input operator. The details are given in the following code. Figure 11.10 shows a screenshot.

⟨*putback.c*⟩≡

```
#include <LEDA/window.h>
int main()
{
  window W(400,400, "Putback Event Demo");
  W.init(-100,+100,-100);
  W.display(window::center,window::center);
  // partition the drawing area in four quadrants
  W.draw_hline(0);
  W.draw_vline(0);
  for(;;)
  {
    double x,y;
    // wait for first click
    int but = W.read_mouse(x,y);
    // middle button erases the window
    if (but == MOUSE_BUTTON(2))
    { W.clear();
      W.draw_hline(0);
      W.draw_vline(0);
      continue;
     }
    // right button terminates the program
    if (but == MOUSE_BUTTON(3)) break;
    // now we put the mouse click back to the event queue
    put_back_event();
    // and distinguish cases according to its position
    if (x < 0)
      if (y > 0)
        { point p;
          if (W >> p) W.draw_point(p,red);
        }
      else
        { segment s;
          if (W >> s) W.draw_segment(s,green);
        }
    else
      if (y > 0)
        { polygon pol;
          if (W >> pol) W.draw_polygon(pol,blue);
        }
      else
```

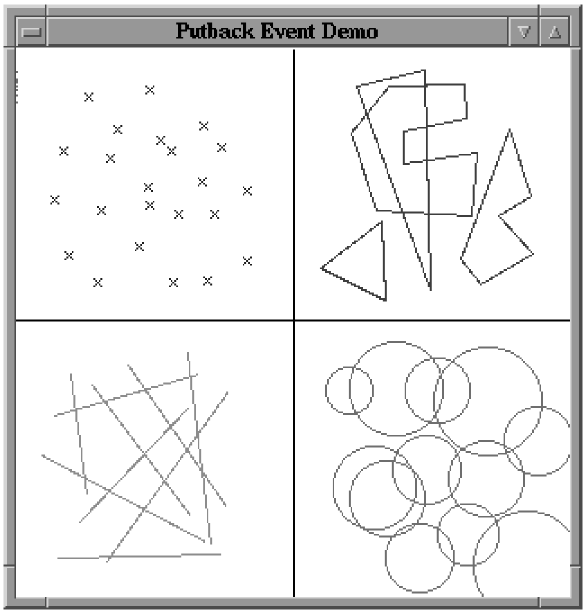


Figure 11.10 A screenshot of the putback program.

```
        { circle c;
          if (W >> c) W.draw_circle(c,orange);
        }
  }
  W.screenshot("putback.ps");
  return 0;
}
```

Non-Blocking Event Input: Similar to the non-blocking versions of the *read_mouse* operation, there are non-blocking variants of the *read_event* operation.

```
int  W.get_event(int& val, double& x, double& y)
```

looks for an event for W. More precisely, if there is an event for window W in the event queue, a *W.read_event* operation is performed, otherwise the integer constant *no_event* is returned.

There is also a more general non-member variant that allows us to read events of arbitrary windows.

```
int read_event(window*& wp, int& val, double& x, double& y)
```

waits for an event. When an event occurs, it returns its type, assigns a pointer to the corresponding window to *wp*, the value to *val*, and the position to (x, y).

This version of *read_event* can be used to write programs that can handle events for several windows simultaneously. The following program opens two windows *W1* and *W2*. The main loop reads all events, determines for each event in which of the two windows it occurred, and puts the event back to the systems event queue. If the event occurred in W_1, it reads and draws a point in W_1, if the event occurred in W_2, it reads and draws a segment in *W2* using the basic input and output operators discussed in Section 11.6.

⟨*two_windows.c*⟩≡

```
#include <LEDA/window.h>
main()
{
   window W1(500,500,"Window 1: points");
   W1.display(window::min,window::min);
   window W2(500,500,"Window 2: segments");
   W2.display(window::max,window::min);
   for(;;)
   { window* wp;
     double x,y;
     int val;
     if (read_event(wp,val,x,y) != button_press_event) continue;
     if (val == MOUSE_BUTTON(3)) break;
     put_back_event();
     if (wp == &W1) { point p;   W1 >> p; W1 << p; }
     if (wp == &W2) { segment s; W2 >> s; W2 << s; }
    }
  return 0;
}
```

Exercises for 11.12

1 Write a "click and drag" program for drawing circles.

2 Write a program that displays text written on the keyboard of your computer in a LEDA window.

3 Implement a simple graph editor that can be used to draw the nodes and edges of a graph. Your program should allow you to move a node by clicking on it and dragging it with the mouse to a new position.

11.13 **Timers**

Each LEDA window has a *timer* clock that can be used to execute periodically a user-defined function. The function and the time interval between two consecutive calls of the function are specified in the start operation

```
void W.start_timer(int msec,void (*func)(window*);
```

A call of this operation starts the timer of W and makes it call the function *func* with a pointer to W as the actual parameter (*func*$(\&W)$) every *msec* milliseconds.

Figure 11.11 A screenshot of the dclock program.

```
void W.stop_timer();
```

stops the timer.

We show the usefulness of timers by writing a simple digital clock demo program. Figure 11.11 shows a screenshot of the clock.

⟨*dclock.c*⟩≡

```
#include <LEDA/window.h>
#include <time.h>
void display_time(window* wp)
{
  window& W = *wp;
  // get the current time
  time_t clock;
  time(&clock);
  tm* T = localtime(&clock);
  // and display it (centered in W)
  double  x = (W.xmax() - W.xmin())/2;
  double  y = (W.ymax() - W.ymin())/2;
  W.clear();
  W.draw_ctext(x,y,string("%2d:%02d:%02d",
                         T->tm_hour,T->tm_min,T->tm_sec));
}
int main()
{
  window W(150,50, "dclock");
  W.set_bg_color(grey1);
  W.set_font("T32");
  W.set_redraw(display_time);
  W.display(window::center,window::center);
  W.start_timer(1000,display_time);
  W.read_mouse();
  W.screenshot("dclock.ps");
  return 0;
}
```

Exercises for 11.13

1 Implement an analog clock.

2 Write a program that draws randomly colored balls are random times.

11.14 The Panel Section of a Window

The panel section of a window is used for displaying text messages and for updating the values of variables. It consists of a list of panel items and a list of panel buttons. We discuss panel items and panel buttons in seperate subsections.

11.14.1 *Panel Items*

A panel item consists of a string label and an associated variable of a certain type. The value of this variable is visualized by the appearance of the item in the window (e.g. by the position of a slider) and can be manipulated through the item (e.g. by dragging the slider with the mouse) during a *read_mouse* or *get_mouse* operation.

There are five types of items. Figure 11.12 shows the representation of the items in a panel. It also shows some menu buttons at the bottom of the panel. The program that generates this panel can be found in *LEDAROOT/demo/win/panel_demo.c*.

Text items have only an associated string, but no variable. The string is formatted and displayed in the panel section of the window.

Simple items have an associated variable of type *int*, *double*, and *string*. The item displays the value of the variable as a string. The value can be updated in a small sub-window by typing text and using the cursor keys. For string items there exists a variant called *string menu item* that in addition displays a menu from which strings can be selected.

Choice items have an associated variable of type *int* whose possible values are from an interval of integers $[0\,..\,k]$. With every value i of this range there is a choice string s_i associated. These strings are arranged in a horizontal array of buttons and the current value of the variable is displayed by drawing the corresponding button as pressed down and drawing all other buttons as non-pressed (if the value of the variable is out of the range $[0\,..\,k]$ no button is pressed). The value of the variable is set to i by pressing the button with label s_i. Pressing a button will release the previously pressed button. It is tempting to confuse the semantics of the string s_i with the integer i. LEDA will not hinder you to use the string "seven" for the third button. Pressing the button with name "seven" will assign 3 to the variable assigned with the button.

For *multiple choice items* the state (pressed or unpressed) of the button with label s_i indicates the value of the i-th bit in the binary representation of the integer value of the associated variable. Multiple choice buttons allow several buttons to be pressed at the same time. For example, the value of the variable associated with the item named "multiple choice" in Figure 11.12 is $1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 = 13$.

In both cases there exist variants that use bitmaps $b_0, \ldots, b_k$ instead of strings to label the choice buttons. Furthermore, there are special choice items for choosing colors (*color_item*) and line styles (*line_style_item*).

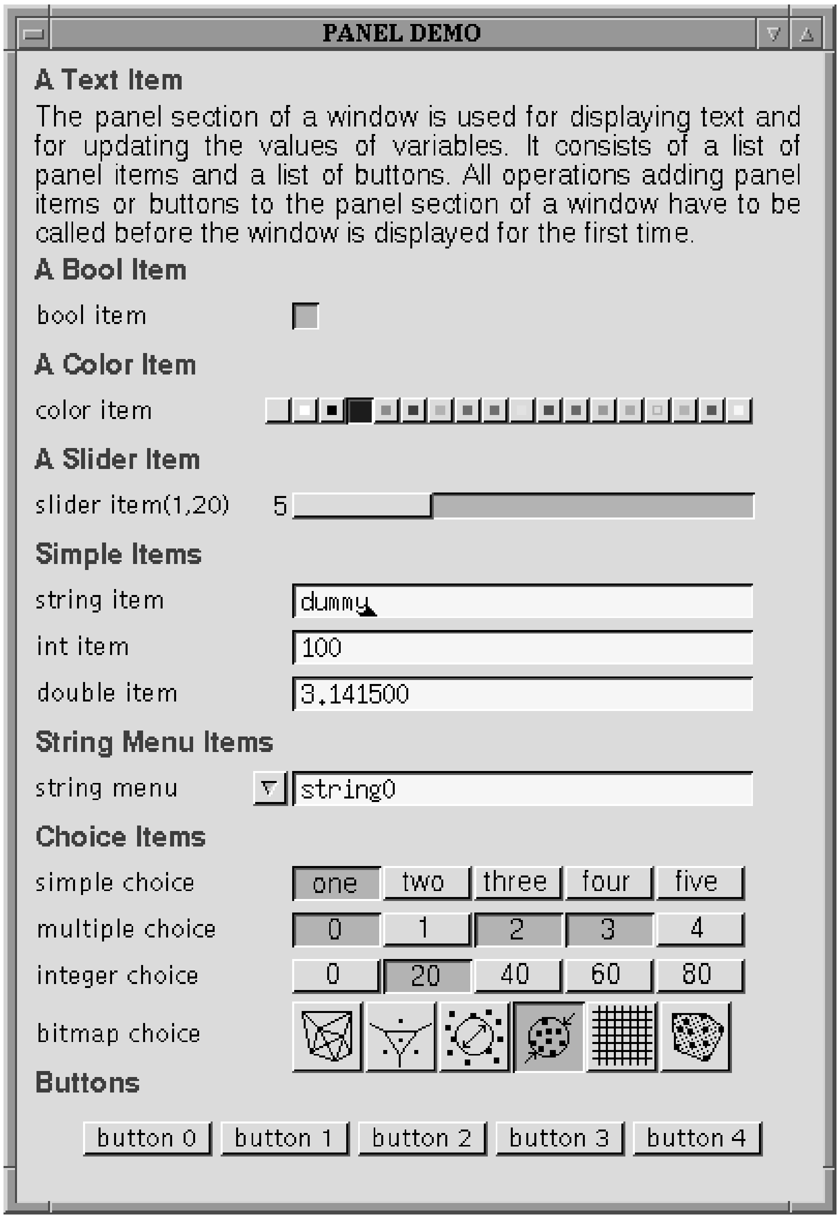


Figure 11.12 Panel items and buttons.

Slider items have associated variables of type *int* with values from an interval [*low* .. *high*]. The current value is shown by the position of a slider in a horizontal box. It can be changed by moving the slider with the mouse.

Boolean items are used for variables of type *bool*. They consist of a single small button whose state (pressed or unpressed) represents the two possible values (*true* or *false*).

We discuss the operations for adding panel items to a panel in Section 11.14.4. It is possible to associate a so-called *call-back* or *action* function with a panel item. This is a function of type

```
void (*action)(T x)
```

where T is the type of the variable of the item. The action function is called after each item manipulation (e.g. dragging a slider or pressing down a choice button) with the *new* value of the item as its argument. However, the value of the variable associated with the item is only changed *after* the return of the action function. In this way, the old and the new value of the item variable is available in the action function. This is very useful as the following program shows.

⟨*callback.c*⟩≡

```
#include <LEDA/window.h>

static int i_slider = 0;
static int i_choice = 0;
static int i_multi = 0;

void f_slider(int i_new)
{ cout << "slider: old = " << i_slider << ", new = " << i_new << endl; }

void f_choice(int i_new)
{ cout << "choice: old = " << i_choice << ", new = " << i_new << endl; }

void f_multi(int i_new)
{ cout << "multi:  old = " << i_multi  << ", new = " << i_new << endl; }

main()
{
  list<string> L;
  for(int i = 0; i < 8; i++) L.append(string("%d",i));

  window W(300,300);

  W.int_item("slider", i_slider, 0, 100, f_slider);
  W.int_item("choice", i_choice, 1, 8, 1, f_choice);
  W.choice_mult_item("multi", i_multi, L, f_multi);

  W.display();
  W.read_mouse();

  W.screenshot("callback.ps");
  return 0;
}
```

In the main program we define three panel items, each with an associated action function. In each case the action function prints the old value and the new value of the variable. The slider item has a range [0 .. 100], the choice item has eight buttons with associated values 1 to 8 (the smallest value is one, values are increased by one, and the largest value is no larger than eight), and the multiple choice item has eight buttons labeled with strings "0", "1", ..., "7". The button with label i represents the i-th bit of variable *i_multi*.

An action function associated with a panel item of a window W may obtain a pointer to W by calling the static member function *window::get_call_window*( ).

The program below implements a simple color definition panel. It uses three slider items for adjusting the (r, g, b)-values of the color. With each slider a call-back function is associated that paints the window background with the current color. A screenshot is shown in Figure 11.13.

⟨*defcolor.c*⟩≡

```
#include <LEDA/window.h>
static int r,g,b;
void slider_red(int x){window::get_call_window()->clear(color(r,g,b));}
void slider_green(int x){window::get_call_window()->clear(color(r,g,b));}
void slider_blue(int x)window::get_call_window()->clear(color(r,g,b));}
int main()
{
  window W(320,300,"define color");
  color col = green2;
  col.get_rgb(r,g,b);
  W.int_item("red  ",r,0,255,slider_red);
  W.int_item("green",g,0,255,slider_green);
  W.int_item("blue ",b,0,255,slider_blue);
  W.set_bg_color(col);
  W.display(window::center, window::center);
  W.read_mouse();
  W.screenshot("defcolor.ps");
  return 0;
}
```

The values of item variables may also be changed in the program. This has *no* effect on the display until the panel is redrawn for the next time. The *redraw_panel* operation redraws the panel area.

We use a simple progress indicator as an example. It uses a slider item to visualize the increasing value of a counter. Figure 11.14 shows a screenshot.

⟨*progress.c*⟩≡

```
#include <LEDA/window.h>
main()
{
  int count = 0;
  window W(400,100);
  W.set_item_width(300);
  W.int_item("progress",count,0,1000);
  W.display(window::center, window::center);
  for(;;)
  { count = 0;
    while (count < 1000)
    { W.redraw_panel();
      W.flush();
```

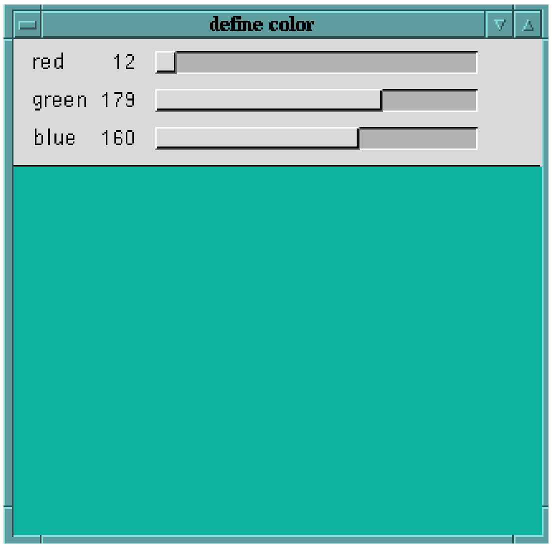


Figure 11.13 A screenshot of the defcolor program.

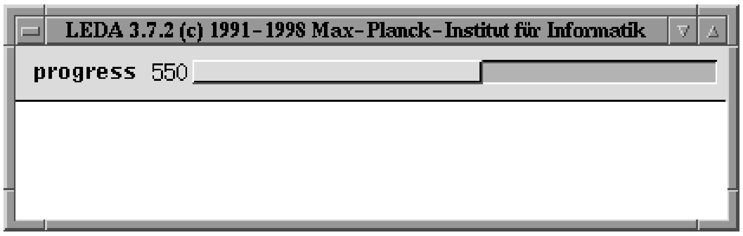


Figure 11.14 A screenshot of the progress program.

```
        leda_wait(0.05);
        count++;
      }
    if (W.read_mouse() == MOUSE_BUTTON(3)) break;
   }
  W.screenshot("progress.ps");
  return 0;
}
```

11.14.2 ***Panel Buttons***

Panel buttons are special panel items. They can be pressed by clicking a mouse button when the mouse pointer is positioned inside their area. Pressing a panel button during a *read_mouse* or *get_mouse* call has the same effect as pressing a mouse button in the drawing area: the operation terminates and the number of the pressed button is returned.

Each panel button has a label or a pixrect image (displayed on the button) and an associated number. The number of a button is either defined by the user or is the rank of the button in the list of all buttons. If a button is pressed (i.e. selected by a mouse click) during a *read_mouse* operation its number is returned. Buttons can have *action functions* of type

```
void (*action)(int but)
```

Whenever a button with an associated action function is pressed this function is called with the number of the button as its actual parameter.

Instead of an action function, a button may have an attached sub-window, in which case we call it a *menu button* (since in most cases such a sub-window is used to realize a menu). Whenever a menu button is pressed the attached sub-window (or menu) M will open and the result of M.*read_mouse*( ) will be returned by the currently active *read_mouse* operation. Of course, M again can have menu buttons, ...

11.14.3 *Panels and Menus*

The data types *panel* and *menu* are two special types representing windows that have no drawing area. Panels (windows of type *panel*) support all panel operations of the general *window* type described in the following section. In addition, they have a special P.*open*( ) operation that displays a panel P, executes P.*read_mouse*( ), closes P, and returns the result of the *read_mouse* operation. There are variants of the *open* operation allowing us to pass parameters for the (initial) positioning of the panel (see the *display* operations for windows for an explanation).

```
int P.open(int xpos=window::center, int ypos=window::center);
int P.open(window& W, int xpos=window::center, int ypos=window::center);
```

Menus (windows of type *menu*) are special panels that only consist of a vertical array of buttons. They support only one kind of panel operation, the addition of buttons, and can be used as sub-windows attached to (menu) buttons only.

11.14.4 *Adding Panel Items*

The operations in this section add panel items or buttons to the panel section of W. Note that they have to be called before the window is displayed the first time.

All operations return a pointer to the corresponding panel item (type *panel_item*)

The generic interface of an operation for adding a panel item (of kind *XXX_item*) for a variable x of type T is as follows:

```
panel_item W.XXX_item(string label, T x&, void (*action)(T) );
```

The last parameter is optional. We give some examples. In all examples we use ... to indicate the optional action function argument.

Simple Items: The following functions add simple items with name s and associated variable x.

```
panel_item W.bool_item(string s, bool& x, ...);
panel_item W.double_item(string s, double& x, ...);
panel_item W.int_item(string s, int& x, ...);
panel_item W.string_item(string s, string& x, ...);
panel_item W.color_item(string s, color& x, ...);
```

String Menu Items: The functions

```
panel_item W.string_item(string s, string& x, list<string> L, ...);

panel_item W.string_item(string s, string& x, list<string> L,int h,...);
```

add string menu items with name s, associated variable x, and a menu list L of candidate values for x. The first version displays the strings of L in a rectangular table of appropriate size. The second version uses a scroll box of height h with a vertical slider that can be used to scroll through the list.

Choice Items: The functions

```
panel_item W.int_item(string s, int& x, int l, int h, int step);
panel_item W.choice_item(string s, int& x, const list<string>& L, ...);
panel_item W.choice_item(string s, int& x, int n, int w, int h,
                                                      char** bm, ...);
panel_item W.choice_mult_item(string s, int& x,
                                              const list<string>& L, ...);
panel_item W.choice_mult_item(string s, int& x, int n, int w,
                                                  int h, char** bm, ...);
```

define choice and multi-choice items with name s and associated variable x. The first variant defines a choice item with buttons l, $l + step$, $\ldots$, the second variant defines a choice item whose buttons are labeled by the strings in L, the third variant defines a choice item with n buttons each of which is labeled by a bitmap of width w and height h (bm is the array that contains the bitmaps). The fourth and fifth variant are analogous to the second and third variant, but define multi-choice items instead of choice items.

Slider Items: The function

```
panel_item W.int_item(string s, int& x, int l, int h);
```

adds a slider item with name s, associated variable s, and range $[l \,..\, h]$.

11.14.5 *Adding Buttons*

The following operations add buttons to the panel section of a window. Note that buttons are always positioned at the bottom of the panel area. There are three basic kinds of buttons: buttons with string labels, buttons with bitmaps, and buttons with pixrects.

String Buttons:

```
int W.button(string label, int n);
```

adds a new button to W with label s and number n.

```
int W.button(string label);
```

adds a new button to W with label s and number equal to its rank in the list of all buttons.

```
int W.button(string s, int n, void *(F)(int));
```

adds a button with label s, number n, and action function F to W. Function F is called with actual parameter n whenever the button is pressed.

```
int W.button(string s, void (*F)(int));
```

adds a button with label s, number equal to its rank, and action function F to W. Function F is called with the value of the button as argument whenever the button is pressed.

```
int W.button(string s, int n, window& M);
```

adds a button with label s, number n, and attached sub-window (menu) M to W. Window M is opened whenever the button is pressed.

```
int W.button(string s, window& M);
```

adds a button with label s and attached sub-window M to W. The number returned by *read_mouse* is the number of the button selected in sub-window M.

Bitmap Buttons: Bitmap buttons are labeled with bitmaps instead of string labels. Each bitmap button has an associated bitmap (w, h, bm) that is specified in the operation for adding the button (see below). There exist the same variants (with and without a user-defined number, with action function or with sub-window) as for string buttons.

```
int W.button(int w, int h, char* bm, string s, int n);
int W.button(int w, int h, char* bm, string s);
int W.button(int w, int h, char* bm, string s, int n, void (*F)(int));
int W.button(int w, int h, char* bm, string s, void (*F)(int));
int W.button(int w, int h, char* bm, string s, int n, window& M);
int W.button(int w, int h, char* bm, string s, window& M);
```

The following program creates the panel shown in Figure 11.15.

⟨*bm_buttons.c*⟩≡

```
#include <LEDA/window.h>
#include <LEDA/bitmaps/button32.h>
int main()
{
  panel P("Bitmap Buttons");
  P.buttons_per_line(8);
  P.set_button_space(3);
  for(int i=0; i < num_button32; i++)
    P.button(32,32,bits_button32[i],string(name_button32[i]));
```

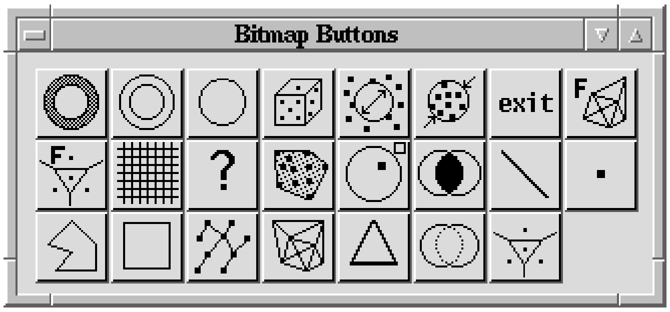


Figure 11.15 A screenshot of the bm_buttons program.

```
  int button = P.open();
  P.screenshot("bm_buttons.ps");
  return 0;
}
```

Pixrect Buttons: Pixrect buttons are labeled with pixrects instead of string labels. Each button has two pixrects, the first one (*pr1*) is used for unpressed buttons and the second (*pr2*) is used for pressed-down buttons. Again we have the same variants as for string buttons.

```
int  button(char* pr1, char* pr2, string s, int n);
int  button(char* pr1, char* pr2, string s);
int  button(char* pr1, char* pr2, string s, int n, void (*F)(int));
int  button(char* pr1, char* pr2, string s, void (*F)(int));
int  button(char* pr1, char* pr2, string s, int n, window& M);
int  button(char* pr1, char* pr2, string s, window& M);
```

The following program creates the panel shown in Figure 11.16. For simplicity, we have used the same pixrect for unpressed and pressed buttons.

⟨*pm_buttons.c*⟩≡

```
#include <LEDA/window.h>
#include <LEDA/pixmaps/button32.h>
int main()
{
  panel P("Pixrect Buttons");
  P.buttons_per_line(10);
  P.set_button_space(3);
  for(int i = 0; i < num_button32; i++)
  { char* pr = P.create_pixrect(xpm_button32[i]);
    P.button(pr,pr,name_button32[i],i);
   }
  int button = P.open();
```

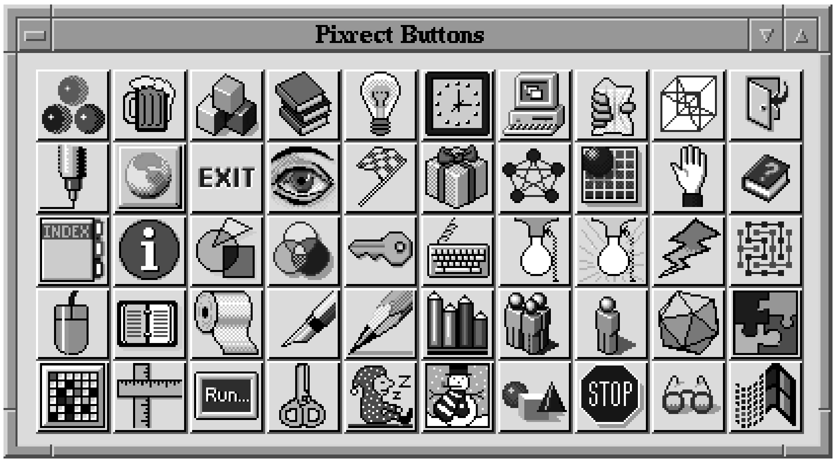


Figure 11.16 A screenshot of the pm_buttons program.

```
  P.screenshot("pm_buttons.ps");
  return 0;
}
```

Creating a Menu Bar: There are two styles for menu buttons, i.e., buttons with an attached sub-window. In the default style menu buttons are displayed as buttons with an additional menu-sign. In the second style the menu buttons are arranged into a menu bar at the top of the panel section. Figure 11.17 shows both styles. The call

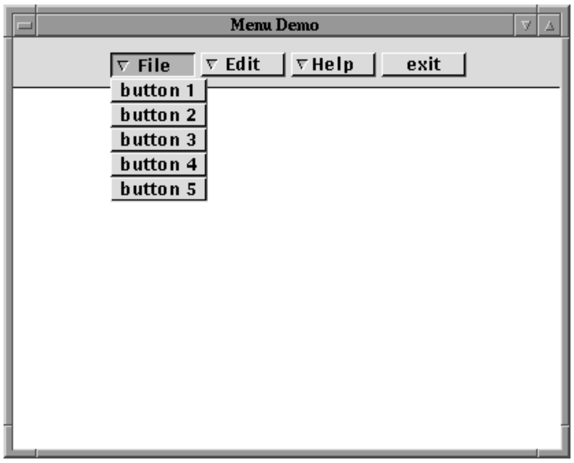
```
void W.make_menu_bar()
```

selects the menu button style.

The following program and the screenshots in Figure 11.17 demonstrate both alternatives. With the command line argument "menu bar", the menu bar version is chosen.

⟨*menu_bar.c*⟩≡

```
#include <LEDA/window.h>
int main(int argc, char** argv)
{
  menu M;
  M.button("button 1"); M.button("button 2"); M.button("button 3");
  M.button("button 4"); M.button("button 5");
  window W(400,300,"Menu Demo");
  W.button("File",M); W.button("Edit",M); W.button("Help",M);
  W.button("exit");
  if (argc > 1 && string(argv[1]) == "menu_bar") W.make_menu_bar();
  W.display();
  W.read_mouse();
```



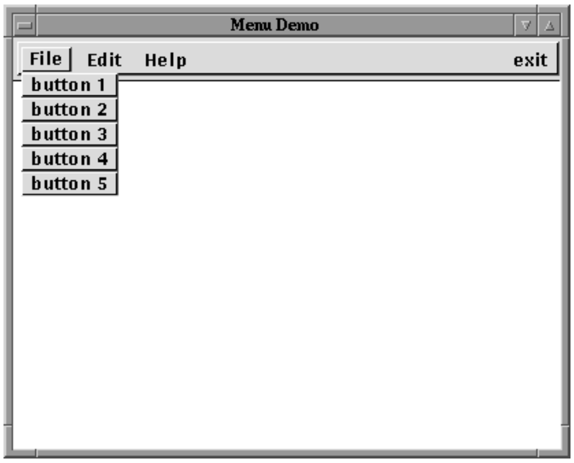


Figure 11.17 Menu buttons: The upper screenshot shows the default style and the lower screenshot shows the menu bar style.

```
  W.screenshot("menu_bar.ps");
  return 0;
}
```

Exercises for 11.14

1 Implement a simple desk calculator with a graphical input.

2 Implement quicksort and use a panel to monitor the values of all variables.

3 Implement a simple file viewer program with a menu bar containing a “File” menu with operations for loading and saving text, and an “Option” menu for defining global parameters such as the font and color of the text.

11.15 Displaying Three-Dimensional Objects: d3_window

The data type *d3_window* uses a LEDA window to visualize and animate three-dimensional drawings of graphs. If the graph to be shown is a planar map (as in the follwing application) the faces are drawn in different grey scales.

The following program uses a *d3_window* to visualize the convex hull of a set of three-dimensional points. Figure 11.1 at the beginning of this chapter shows a screenshot of the program *LEDAROOT/demo/geo/d3hull_demo.c* which expands on the program below.

The convex hull algorithm

```
CONVEX_HULL(const list<d3_rat_point>& L, GRAPH<d3_rat_point,int>& H)
```

takes a list L of three-dimensional points and constructs the surface graph H of their convex hull. H is a planar map that is embedded into three-dimensional space.

To visualize this graph we create a d3-window *d3win*, whose constructor takes a window W (that has to be displayed before), the graph H, and a node array *pos* of vectors that gives for every node v of H the position $H[v]$ of v in space as a three-dimensional vector.

Finally, we call *d3win.read_mouse*( ) that does something very similar to the *read_mouse* operation for (two-dimensional) windows. It waits for a mouse click and returns the number of the mouse button pressed. While waiting for a click, the graph H is shown in a two-dimensional projection and is, depending on the current position of the mouse pointer, rotated in space. If H is a planar map (as it is in this case), the d3-window, in addition, computes its faces and paints them in different grey scales.

There are many parameters for controlling the appearance of the graph, e.g., whether faces should be painted as described above, for the center and speed of rotation, for changing colors of nodes and edges, ... For details, we refer the reader to the user manual.

⟨*d3_hull.c*⟩≡

```
#include <LEDA/d3_hull.h>
#include <LEDA/d3_window.h>
main()
{
  // construct a random set of points L
  list<d3_rat_point> L;
  random_d3_rat_points_in_ball(50,75,L);
  // construct the convex hull H of L
  GRAPH<d3_rat_point,int> H;
  CONVEX_HULL(L,H);
  // open a window W
  window W(400,400,"d3 hull demo");
  W.init(-100,+100,-100);
  W.display(window::center,window::center);
  // extract the node positions into an array of vectors
  node_array<rat_vector> pos(H);
  node v;
  forall_nodes(v,H) pos[v] = H[v].to_vector();
  // and display H in a d3_window for window W
```

```
  d3_window d3win(W,H,pos);
  d3win.read_mouse();
  W.screenshot("d3_hull.ps");
  return 0;
}
```

Exercises for 11.15

1 Extend the 3d convex hull program by adding a panel section to the window that allows you to choose between different types of input points and to specify the size of the input point set. Your window should look like the window of Figure 11.1.

Bibliography

[Nye93] Adrian Nye. *Xlib Programming Manual for Version 11.* O'Reilly & Associates, Inc., Sebastopol CA, 3 edition, 1993.

Index