# Chapter 2

# Objects and Graphics Devices

In this chapter, we will give a general conceptualization for graphics objects and we will show how they relate to graphics devices. From these concepts, we will present the two-dimensional graphics library that will be used in the implementation of interactive progamas discussed in the book.

## 2.1 Graphics Objects

To study the computer graphics processes, ideally, we would like to have an inclusive conceptualization to allow an understanding of the area as a whole. This conceptualization should be based on a mathematical model including the relevant objects in the area, such as geometric models and images.

In this sense, we will formulate the concept of *graphics object* that will be the starting point for constructing our analysis. Therefore, we establish computer graphics as the area where graphics objects are studied. The processes in the area thus correspond to operations with graphics objects of a certain type, as well as conversions between different types of graphics objects.

A graphics object, $\mathcal{O} = (S, f)$, consists of a subset $S \subset \mathbb{R}^m$, and a function $f\colon S \rightarrow \mathbb{R}^n$. $S$ is called *geometric support* of $\mathcal{O}$, and it determines the geometry and the topology of the graphics object. Function $f$ specifies the properties of $\mathcal{O}$ at each point $p \in S$, and it is called *attribute function* of the object (see Figure 2.1).

The dimension of the object $\mathcal{O}$ is given by the dimension of its geometric support $S$. The several attributes of $\mathcal{O}$ correspond to $l$-dimensional sub-spaces in the Euclidean space $\mathbb{R}^n$. For more information on graphics objects, we refer the reader to (**?**).

This definition is sufficiently general to include all of the relevant objects for computer graphics, such as points, curves, surfaces, solids, images and volumes.

A family of graphics objects of great importance is constituted by the "planar graphics objects", for which $m = 2$, that is, the geometric support is contained on the Eu-

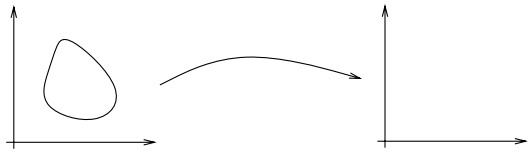


**Figure 2.1** Generic Graphics Object.

clidean plane $\mathbb{R}^2$. Its relevance is due to the fact that the class of objects can be mapped directly into the usual graphics devices. These objects have dimension $\mathrm{Dim}(S) \leq 2$, and they correspond to points, curves and planar regions (we exclude the set of fractals).

Two important examples of planar graphics objects are curves and polygonal regions. In general, these objects are used to represent, in an approximate way, curves and arbitrary regions on the plane. Another important example of graphics object is a digital image. For more details on these graphics objects, the reader should consult (**?**).

### 2.1.1 Description of Graphics Objects

Two general forms exist to mathematically describe the geometric support of a graphics object: the parametric and the implicit forms.

In the *parametric* description, the set of points $p \in S$ is directly specified by a function $g : \mathbb{R}^k \to \mathbb{R}^m$, where $k = \mathrm{Dim}(S)$

$$S = \{(x_1, \ldots, x_m) \quad | \quad (x_1, \ldots, x_m) = g(u_1, \ldots, u_k)\}$$

In the *implicit* description, the points of $S$ are indirectly determined by a function $h : \mathbb{R}^m \to \mathbb{R}^{m-k}$

$$S = h^{-1}(c) = \{(x_1, \ldots, x_m) \quad | \quad h(x_1, \ldots, x_m) = c\}$$

**Exemplo 2.1** (Círcle)**.** To compare these two descriptions, we will use as example the unit circle (see Figure 2.2):

- Parametric Description:
  $(x, y) = (\sin(u), \cos(u))$, where $u \in [0, 2\pi]$.
- Implicit Description:
  $h^{-1}(1)$, with $h(x, y) = x^2 + y^2 = 1$.

Notice the above descriptions constitute a continuous mathematical model of the geometry of a graphics object. Therefore, we have to obtain a finite representation of these models to work with them in the computer, which is a discreet machine.

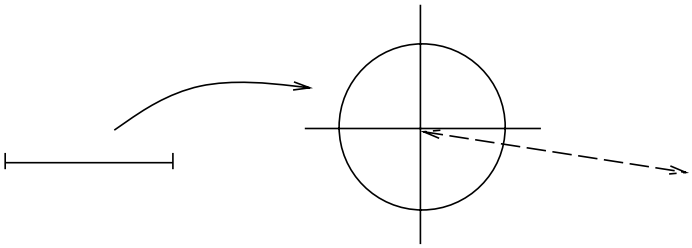


**Figure 2.2** Parametric and Implicit Descriptions of a Circle.

### 2.1.2 Discretization and Reconstruction of Graphics Objects

The passage from a continuous to a discreet object is called *discretization* or *representation* of the object. The inverse process, of recovering the continuous model from its discreet representation is called *reconstruction*. The reconstruction can be exact or approximate, depending completely on the characteristics of the process as a whole.

For this end, a simple form, quite used in practice, consists on the discretization by point sampling and the reconstruction by linear interpolation, as represented in Figure 2.3.

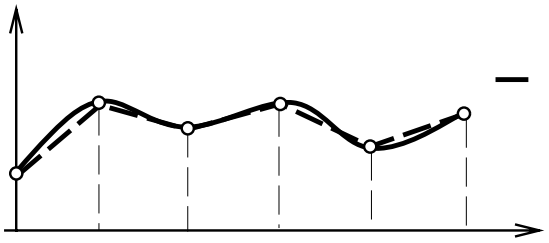


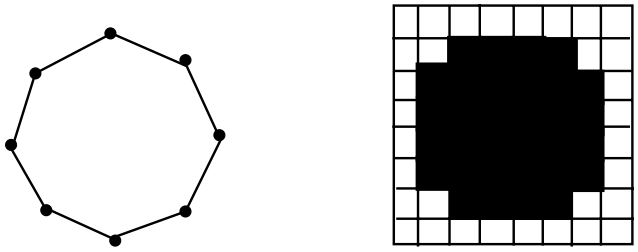
**Figure 2.3** Sampling and Reconstruction.

Consider a continuous function $f : \mathbb{R} \to \mathbb{R}$. The representation by uniform point sampling is given by the sequence of samples $(y_i)_{i\in\mathbb{Z}}$, where $y_i = f(x_i)$ corresponds to the value of $f$ at the sampled points $x_i = x + i\Delta x$. The reconstruction is obtained from the samples $(y_i)$, by linear interpolation $\overline{f}(x) = ty_i + (1-t)y_{i+1}$, where $t = x \bmod \Delta x$ e $i = \lfloor x/\Delta x \rfloor$. Notice in this case, the representation only provides an approximate reconstruction, that is, $\overline{f} \approx f$ (see Figure 2.3).

**Exemplo 2.2** (Representation of Implicit and Parametric Objects)**.** To construct a

discreet representation of a circle, starting from its parametric description, we discretize the parameter $u \in [0, 2\pi]$, making $u_i = i/2\pi$, $i = 0, \ldots, N-1$, and evaluate $(x_i, y_i) = g(u_i)$, obtaining the coordinates of these $N$ points on the circle. The circle representation is therefore given by this list of points. (see Figure 2.1.2)

To construct a discreet representation of a unit disk, starting from its implicit description, we discretize the environment space $\mathbb{R}^2$ and evaluate the implicit function $f(x_i, y_j)$ from a given a regular grid $N \times M$. The representation will be given by the matrix $A$ of dimensions $N \times M$. If $f(x_i, y_j) < 1$, we then make $a_{ij} = 1$, otherwise $a_{ij} = 0$. This representation corresponds to a discretization of the characteristic function of the disk (see Figure 2.1.2).



(a) (b)

**Figure 2.4** Formats of vector (a) and raster (matrix) (b) data.

## 2.2 Graphic Devices and Representation

A graphics device has a representation space in which we should map the object to be manipulated by the device. This way, to visualize a certain graphics object $\mathcal{O} = (U, f)$, we should obtain a representation of the object so that the discretized object can be mapped in the representation space of the device. Once mapped in this space, the device performs the reconstruction of the object, allowing its visualization.

### 2.2.1 Vector Devices

In vector devices, the representation space consists of points and straight line segments. More precisely, the representation space is a subset of the plane where we can assign coordinates to points; besides, given two points $A$ and $B$, the device performs the reconstruction of the segment $AB$. These devices can be used to visualize polygonal curves and surfaces or polyhedral regions. In this case, we only draw the polygon edges of the representation, as shown in Figure 2.5(a).

### 2.2.2 Raster (Matrix) Devices

The representation space of these devices allow to visualize a $m \times n$ matrix, in which each point has a color attribute. Therefore, to visualize a graphics object in these devices, we should obtain a matrix representation of the object. For more details on raster representation of planar graphics objects, we refer the reader to (**?**). Raster devices are appropriate for visualizing digital images (see Figure 2.5(b)).

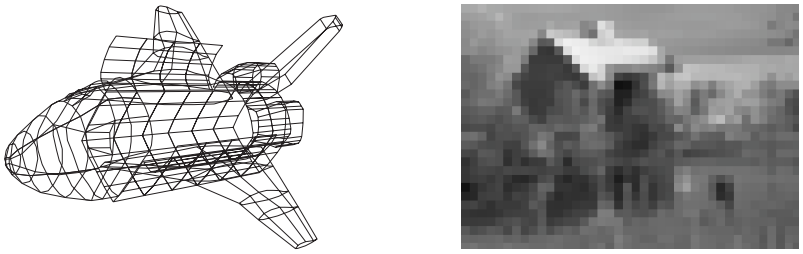
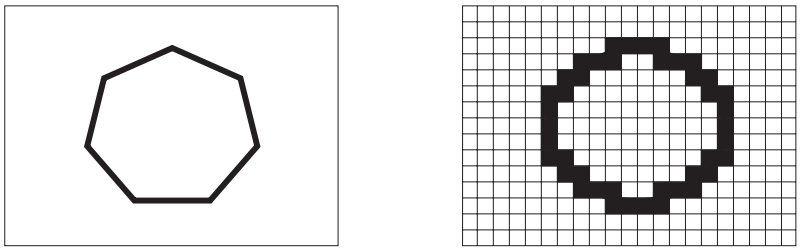


**Figure 2.5** Visualization on a vector (a) and raster (matrix) (b) device.

**Exemplo 2.3** (Rendering a Circle)**.** We should have the appropriate representation to visualize (render) a graphics object in either a vector or raster device. For example, the visualization of a circle can be performed by representing the circle by a polygonal curve (see Figure 2.6(a)). To visualize the circle in a raster device, it must be *rasterized* (i.e. scan converted) to obtain its matrix representation (see Figure 2.6(b)). Notice the polygonal approximation of a circle can be displayed in a raster device; for this, the straight line segments constituting the sides of the polygon must be rasterized.



(a) (b)

**Figure 2.6** Vector (a) and raster (matrix) (b) representations.

Some graphics objects are difficult, or even impossible, to be appropriately visualized in vector devices. The visualization of a polygonal region can be obtained by

placing hatching marks in the reconstruction, while in a raster device its visualization is immediate. The visualization of a digital image is very difficult in vector devices. In this way, in general, we use the vector format to represent geometric models and the raster (matrix) format to represent digital images.

Despite that, both geometric models and digital images can be represented in any of these two formats. In fact, the concept of graphics object allows a unified treatment of these two elements. On one side, we can consider an image as a Monk's surface, and use Differential Geometry techniques in its processing. On the other hand, we can consider the coordinates $(x, y, z)$ in the parametric space $(u, v)$ of a surface as values of an image, and in this way use image processing techniques for modeling purposes (see Figure 2.7).

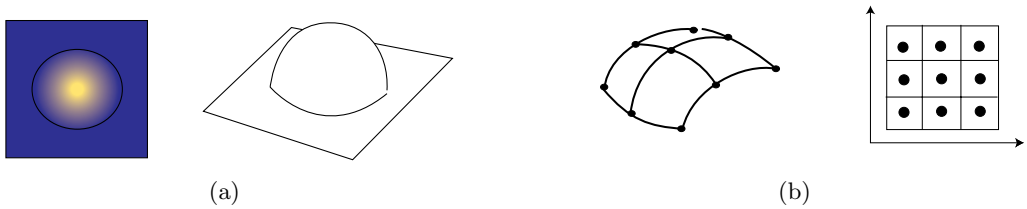



**Figure 2.7** An image as a model (a) and a model as an image (b)

## 2.3 Classification of Graphics Devices

The user-computer interaction with graphics objects happens through graphics devices.

### 2.3.1 Conceptualization

To obtain a classification of graphics devices, we should use abstraction levels. This approach, known as paradigm of the 4 universes, implicates on studying a problem in the context of the following four universes: physical, mathematical models, their representations and implementations. This way, graphics devices can be analyzed according to their use, functionality, graphics format and implementation structure (see Figure 2.8).

#### Usage Mode

The *usage mode* relates to the application to which the graphics device is aimed at being used. According to this criterion, graphics devices can be interactive and non-interactive.

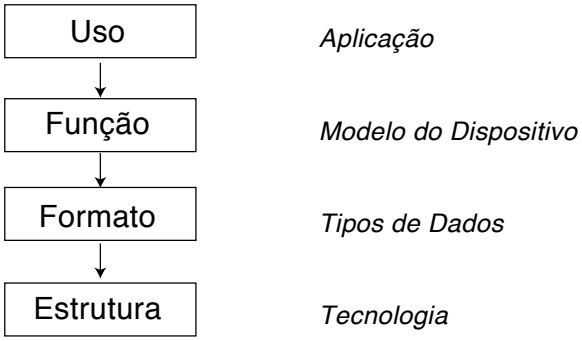



**Figure 2.8** Abstraction levels of graphics devices.

#### Functional Characteristic

The *functional characteristic* relates to the role of the device in the computational model. According to this criterion, graphics devices can be for: input, processing, and output.

#### Data Format

Devices can be classified as *vector* and *raster (matrix)*, according to the geometric nature of their representation space.

#### Implementation Structure

The *implementation structure* is determined by the technology used in the construction of the graphics device. This structure also depends on the usage, functionality and format modes of the device. An example of different implementation structures for a device with the same function can be given by the calligraphic and matrix display devices; both non-interactive graphics output devices, however, the former adopts the vector format while the latter adopts the raster format.

### 2.3.2 Classification

By using the above conceptualization, we can classify graphics devices according to their functionality, and for each type, according to the graphics data format. In this way, we have the graphics devices for input, processing and output, in the vector and raster (matrix) formats.

Examples of vector input devices include: mouse, Trackball and Joystick, operating with relative coordinates; and tablet, touch screen, and data glove, operating with

absolute coordinates. Notice that all of them are two-dimensional, except the date glove, which has six degrees of freedom. Examples of raster input devices include: the frame grabber, scanners, and range devices.

Examples of vector graphics processing devices are the graphics pipelines, such as the Geometry Subsystem of SGI. Examples of raster graphics processing devices are the parallel machines from Pixar and the Pixel Machine.

Examples of vector graphics output devices are: plotter and vector displays. Examples raster graphics output devices are: laser or inkjet printers, and monitors of the type CRT or LCD.

Vector graphics output devices were very common in the 60's and 70's. Raster devices start being more used in the 80's. Nowadays, a good combination consists on using input vector graphics devices (mouse and tablet for instance) and raster output devices (CRT or LCD monitors and laser printers or inkjet).

## 2.4 Graphics Workstations

Above we saw examples of individual graphics devices. In practice, graphics devices are used in conjunction. Mainly for interactive applications, we combine input, processing and output graphics devices into a complete graphics system.

Interactive graphics workstations are the most common class of graphics system. In fact, most current computers can be considered a graphics system.

In this book, we will assume that the graphics implementation is aimed towards a standard graphics system, formed by raster output device, a vector input device and a general purpose processor. (see Figure 2.9). Vector graphics input descriptions are converted to raster descriptions by the windowing system of the graphics workstation.

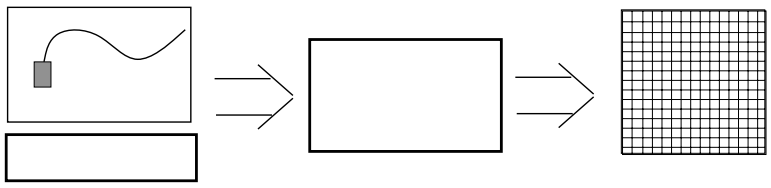


**Figure 2.9** Interactive Graphics Workstation.

### 2.4.1 Windowing System

A modern interactive graphics workstation is controlled by a graphics sub-system known as a "Windowing System". This sub-system is usually incorporated into the operational

system of the machine and it controls the graphics input, processing and output functions. Windowing systems are based in the paradigm of the "desktop", in other words, they implement the view of a work table with multiple documents. In this type of system, each window corresponds to a separate computational process. Examples of windowing systems are: X-Windows for UNIX platforms, MS-Windows for PC platform, and the Desktop for Macintosh platforms.

### 2.4.2 Viewing Transformations

To visualize a planar graphics object, we define a window in the coordinate system of the object ("world coordinate system", WC). This window should be mapped into a viewport in the display space of the device. To increase the degree of device independence, a system of normalized coordinates is used ("normalized device coordinates", NDC). This system is defined in the rectangle $[0,1] \times [0,1]$ (see Figure 2.10).

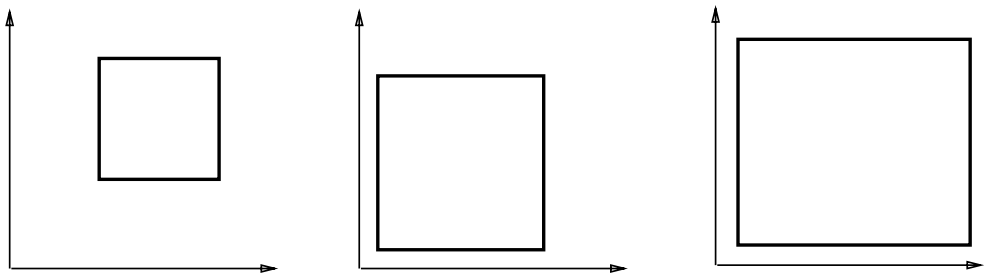


**Figure 2.10** Viewing Transformations.

In this case, the viewport is defined in normalized coordinates, and it is in this viewport that we should map the window defined in the object space. The transformation mapping the points of the window into points of the viewport in normalized coordinates is called *2D viewing transformations*. If the window is defined by the coordinates $(xmin, ymin)$ and $(xmax, ymax)$ and the viewport is defined by $(umin, vmin)$ and $(umax, vmax)$, the viewing transformation is given by

$$u = \frac{umax - umin}{xmax - xmin}(x - xmin) + umin \tag{2.1}$$

$$v = \frac{vmax - vmin}{ymax - ymin}(y - ymin) + vmin. \tag{2.2}$$

For the final stage of the viewing process, the viewport (in normalized coordinates) is mapped into the graphics device.

The transformations between the window (in space coordinates), the viewport (in normalized coordinates) and the graphics device are obtained by a simple change of scaling in the coordinates, which alters the window dimensions (see (**?**)).

## 2.5 The GP Graphics Package

One key problem related to the implementation of interactive graphics programs is the *portability*. Ideally, graphics programs would indistinctively work in any platform. At least, it would be desirable that the same code could be used for graphics devices of each basic type.

The solution to this problem is in the concept of *device independence*, which implicates on creating a programming layer to isolate implementation differences from the several devices. This layer is the *graphics package*. As in our conceptual schema of the 4 universes, we can therefore have a common representation for different implementations.

### 2.5.1 GP Characteristics

In this book, we will adopt the GP (Graphics Package), originally developed by Luiz Henrique Figueiredo.

GP uses a representation space only allowing vector specifications. Starting from the vector specifications, GP performs the appropriate conversion to reconstruct the graphics object in the device being used. Due to this fact, we say that GP uses a vector metaphor to manipulate the graphics objects in both the input and output.

In general, GP assumes the existence of a graphics workstation composed of a two-dimensional raster output device and a vector input device (besides the keyboard). The graphics sub-system of the workstation should be based on the paradigm of windows.

GP was chosen by matching and fitting quite well with the proposal of the book. More specifically, this package has the following characteristics:

- **Minimality**: GP implements an API (Application Programmer Interface) which is minimal but enough for simple graphics programs.
- **Portability**: GP is an device independent package, based on the paradigm of windows.
- **Separability**: the architecture of GP isolates the implementation details of the graphics API.
- **Availability** : the package supports most of the existing platforms.

To accomplish these objectives, the architecture of GP was divided in two separate layers: gp and dv.

The *gp* layer is responsible for the two-dimensional viewing transformations. The routines at this level are device independent and they have the purpose of mapping application coordinates into device coordinates.

The *dv* layer is responsible for controlling the graphics devices. The routines of this layer are called by the routines of the gp layer. The routines at this level perform the conversion from vector description to a raster (matrix) representation of the device (rasterization). In other words, this layer implements the vector metaphor in the which GP is based. The dv should be implemented for each platform supported by GP. The implementation of this layer will not be discussed in the book.

The current implementation of the dv layer uses OpenGL for 2D vector graphics output and SDL for window creation and event handling. The option for this solution is due to the fact that OpenGL and SDL are two mature standards that are platform independent and fit well with the API modelo of GP. OpenGL uses the same 2D viewing paradigm of GP and it is supported in hardware in most modern workstations. SDL (Simple DirectMedia Layer) is a cross-platform multimedia library designed to provide low level access to keyboard, mouse, 3D hardware via OpenGL. It is very popular in the game community and it has an event model very similar to GP.

### 2.5.2 Attributing Color in GP

Color is an important attribute in any graphics object. In GP, the attributes of graphics objects consist basically on the color of the vector primitives.

The color attribution process adopted in GP is based on the concept of color map. This allows an indirect definition of colors in the GP representation space. A *color map* is the discretization of a curve in the color space. This discretization is represented by a table associating a numerical index $i \in \{0, \ldots 255\}$ to the color values of the device $c = (r, g, b)$, with $r, g, b \in [0, 1]$. This table is called *look up table* (LUT) (see Figure 2.11).

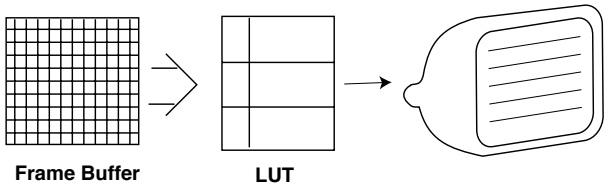



**Figure 2.11** Color map.

This model is implemented in the majority of the raster display devices. In these devices, the matrix representation is stored in the graphics board (frame-buffer), and

the color of each representation cell (pixel) is obtained by performing an addressing in the LUT.

The routine `gprgb` allows to associate a color to an index of the color map. The color is specified by the intensity values of the R, G and B components. This routine return the value 1, if the attribution can be performed, and 0 otherwise. In case the color index has the value $-1$ the color attibute is set for immediate usage. To support a full color device, it is convenient to use the immediate mode of `gprgb` that is implemented in the display with 24 bit RGB values.

```
int gprgb(int c, Real r, Real g, Real b);
#define gprgb dvrgb
```

Color attribution in gp is performed through the current color. This color is set in the immediate mode of `gprgb` or selected through the routine `gpcolor`. The parameter of this routine is an integer, indicating the table input containing the target color. This routine returns the index of the previous current color. When the specified index is not valid, it returns a negative number indicating the size of the color map.

```
int gpcolor(int c);
#define gpcolor dvcolor
```

### 2.5.3 Data Structure and Objects in GP

Given that GP is based on the windows model, the fundamental graphics object in GP is a *box*. This object consists of a rectangle on the plane whose sides are parallel to the coordinate axes. The geometry of this rectangle is given by the coordinates of its main diagonal $(xmin, ymin)$, $(xmax, ymax)$. Besides, we associate to this object a scaling attribute defined by the linear transformation $T(x, y) = (x \cdot xu, y \cdot yu)$. These scaling attributes are used to allow a change on the aspect ratio of the box, without requiring to redefine the entire box.

The basic data structure of this object is the structure `Box`, given by

20 ⟨*Box data structure* 20⟩≡

```
typedef struct {
 Real xmin, xmax;
 Real ymin, ymax;
 Real xu, yu;
} Box;
```

Defines:
`Box`, used in chunk 21.

The box has therefore the dimensions `xu (xmax - xmin)` and `yu (ymax - ymin)`. Notice that `xu` and `yu` are separate scale factors for each one of the directions.

The internal state of GP is stored in the following data structure:

21 ⟨*internal state* 21⟩≡ (? 0—1)

```
static struct {
 Box w, v, d;
 real ax ,bx, ay, by;
} gp = {
 {0.0, 1.0, 0.0, 1.0, 1.0, 1.0},
 {0.0, 1.0, 0.0, 1.0, 1.0, 1.0},
 {0.0, 1.0, 0.0, 1.0, 1.0, 1.0},
  1.0, 0.0, 1.0, 0.0,
};
```

Defines:
`gp`, used in chunks 22–25.
Uses `Box` 20 and `real` 60 60.

This structure consists of three boxes `w`, `v`, `d`, representing, respectively, the window in the two-dimensional space of the scene to be visualized, the viewport in normalized coordinates, and the window of the graphics device.

The coefficients `ax, bx, ay, by`, will be used as scale factors to implement the 2D viewing transformations. Notice the initial state corresponds to the standard configuretion where all of the boxes are unitary (and consequently, the viewing mapping is the identity function).

A window in GP has a color attribute called *background color*. However, this color is not stored in the `Box` data structure. The background color is given by the color at the index $i = 0$ of the look up table. This color is attributed using one of the routines gppallete or gprgb (as previously seen).

### The API of GP

The API of GP can be divided in 4 classes of routines according to their function:

- Control;
- Viewing;
- Drawing and Text;
- Graphics Input and Interaction;

Next, we will study in detail the routines of each one of these classes.

### 2.5.4 Control Routines

We can deduce from GP's internal state that it only supports one window. The control routines in GP are for manipulating this window on the screen of the graphics workstation. The routine `gpopen` initializes GP and opens a window with its name passed as a parameter.

22a ⟨*initialization* 22a⟩≡ (? 0—1)

```
real gpopen(char* name, int width, int height)
{
 real aspect;
 gp.d=*dvopen(name, width, height);
 calculate_aspect();
 gpwindow(0.0,1.0,0.0,1.0);
 gpviewport(0.0,1.0,0.0,1.0);
 gprgb(0,1.,1.,1.);
 gprgb(1,0.,0.,0.);
 gpcolor(1);
 return (gp.d.xu/gp.d.yu);
}
```

Uses `calculate_aspect` 22b, `dvopen`, `gp` 21, `gpcolor`, `gprgb`, and `real` 60 60.

This routine calls the dv layer to initialize the device. In this call, the box parameters of device `d`, structure `gp` are defined. The routine also creates a standardized window and viewport $[0,1] \times [0,1]$ by calling the routines `gpwindow` and `gpviewport`, respectively. These two routines will be studied next in the section on viewing routines. The routine `calculate_aspect` calculates the box scaling parameters of the device, so we can map a square window of maximum dimensions in the device:

22b ⟨*window aspect* 22b⟩≡ (? 0—1)

```
static void calculate_aspect (void)
{
  if (gp.d.xu > gp.d.yu) {
    gp.d.xu /= gp.d.yu;
    gp.d.yu =  1.0;
  } else {
    gp.d.yu /= gp.d.xu;
    gp.d.xu  = 1.0;
  }
}
```

Defines:
`calculate_aspect`, used in chunk 22a.
Uses `gp` 21.

Notice the routine *gpopen* initializes the background color of the window as being the white color. The routine also attributes the black color to the index 1 of the color map, and the call to the routine `gpcolor(1)` attributes this color to the current color of the package.

The routine *gpclose* shutdowns GP, eventually waiting for a certain time in case the parameter `wait` is positive, or for an action from the user in case `wait` is negative. This routine is implemented in the layer *dv* and this is the reason it is defined as a macro.

```
void gpclose(int wait);
#define gpclose dvclose
```

The routine `gpclear` clears the window by painting the background color. The parameter `wait` follows the convention described above.

```
void gpclear(int wait);
#define gpclear dvclear
```

The routine `gpflush` immediately executes all the pending operations for any graphics output. The routine `gpwait` pauses according to the parameter value `t`: $t > 0$, waits for $t$ miliseconds; $t < 0$, and waits for the user's input.

```
void gpflush(void);
#define gpflush dvflush

void  gpwait(int t);
#define gpwait          dvwait
```

### 2.5.5 Viewing Routines

The routines `gpwindow` and `gpviewport` are used to specify the 2D viewing transformation, as we saw in the previous section.

23 ⟨*window* 23⟩≡ (? 0—1)

```
real gpwindow(real xmin, real xmax, real ymin, real ymax)
{
  gp.w.xmin=xmin;
  gp.w.xmax=xmax;
  gp.w.ymin=ymin;
  gp.w.ymax=ymax;
  gpmake();
  dvwindow(xmin, xmax, ymin, ymax);
  return (xmax-xmin)/(ymax-ymin);
}
```

Uses `gp` 21, `gpmake` 24b, and `real` 60 60.

24a ⟨*viewport* 24a⟩≡ (? 0—1)

```
real gpviewport(real xmin, real xmax, real ymin, real ymax)
{
  gp.v.xmin=xmin;
  gp.v.xmax=xmax;
  gp.v.ymin=ymin;
  gp.v.ymax=ymax;
  gpmake();
  dvviewport(xmin, xmax, ymin, ymax);
  return (xmax-xmin)/(ymax-ymin);
}
```

Uses gp 21, gpmake 24b, and real 60 60.

To calculate the viewing transformation coefficients between the window (in the space of the scene) and the viewport (in normalized coordinates), the routines gpwindow and gpviewport call the routine gpmake.

24b ⟨*transformation* 24b⟩≡ (? 0—1)

```
void gpmake(void)
{
 real Ax=(gp.d.xmax-gp.d.xmin);
 real Ay=(gp.d.ymax-gp.d.ymin);
 gp.ax = (gp.v.xmax-gp.v.xmin)/(gp.w.xmax-gp.w.xmin);   /* map wc to ndc */
 gp.ay = (gp.v.ymax-gp.v.ymin)/(gp.w.ymax-gp.w.ymin);
 gp.bx =  gp.v.xmin-gp.ax*gp.w.xmin;
 gp.by =  gp.v.ymin-gp.ay*gp.w.ymin;
 gp.ax = Ax*gp.ax;                                      /* map ndc to dc */
 gp.ay = Ay*gp.ay;
 gp.bx = Ax*gp.bx+gp.d.xmin;
 gp.by = Ay*gp.by+gp.d.ymin;
}
```

Defines:
gpmake, used in chunks 23 and 24a.
Uses gp 21 and real 60 60.

The viewing transformations are effectively realized by the routines `gpview` and `gpunview`, which map points from the application space to the graphics device space and vice-versa.

25a ⟨*view* 25a⟩≡ (? 0—1)

```
void gpview(real* x, real* y)
{
 *x=gp.ax*(*x)+gp.bx;
 *y=gp.ay*(*y)+gp.by;
}
```

Defines:
`gpview`, used in chunk 29.
Uses `gp` 21 and `real` 60 60.

25b ⟨*unview* 25b⟩≡ (? 0—1)

```
void gpunview(real* x, real* y)
{
 *x=(*x-gp.bx)/gp.ax;
 *y=(*y-gp.by)/gp.ay;
}
```

Defines:
`gpunview`, used in chunks 28 and 29.
Uses `gp` 21 and `real` 60 60.

### 2.5.6 Drawing Routines

The drawing routines in GP specify the objects displayed in the device.

In GP, polygonal curves (open or closed) and polygonal regions are denominated *polygonal primitive*. These primitive can be drawn using a combination of the routines `gpbegin`, `gppoint` and `gpend`. The primitive is defined by the sequence of coordinates given by calls to `gppoint`, delimited by `gpbegin` and `gpend`. Notice this schema is similar to the one in OpenGL.

```
void gpbegin(int c);
#define gpbegin dvbegin

void gpend(void);
#define gpend dvend

int gppoint(Real x, Real y)
{
 gpview(&x,&y);
 return dvpoint(x,y);
}
```

The type of primitive polygonal it is specified by the parameter `c` of the routine `gpbegin`.

- `l` - open polygonal curve
- `p` - closed polygonal curve
- `f` - filled polygon

As example on using this schema is in the implementation of routine `gptri`, which draws a triangular region given by

26 ⟨*triangle example* 26⟩≡

```
void draw_triangle(real x1, real y1, real x2, real y2, real x3, real y3)
{
 gpbegin('f');
  gppoint(x1,y1);
  gppoint(x2,y2);
  gppoint(x3,y3);
 gpend();
}
```

Defines:
`draw_triangle`, never used.
Uses `gpbegin`, `gpend`, `gppoint`, and `real` 60 60.

#### Text Routines

A text is a sequence of alpha-numerical characters. The most common attributes of a text are the color of the characters, the family type of the fonts (Helvética, times, etc.) and the variations of the font in each family (bold, italic, etc.). GP uses a fixed size vector font.

The routine `gptext` draws a sequence of characters `s` at the position $(x, y)$

```
void gptext(Real x, Real y, char* s, char* mode)
#define gptext dvtext
```

### 2.5.7 Routines for Graphics Input and Interaction

In general, several input devices exist in a workstation. The most common devices are the keyboard and the mouse. The keyboard is used for numerical alpha data entry, and the mouse is used as a locator, that is, a device allowing the user to specify positions on the screen. The mouse also has buttons allowing the user to define different states of the device.

The user's actions in the devices are captured by the system in a process called *pooling*: the devices are continuosuly verified by the system, and a queue is created, where each queue input contains the identification of the device and the data related to the user interaction with the device. This queue is called *event queue* of the system.

The gp, in general, supports the mouse and keyboard as input devices. In this way, it allows access to a queue of events where we have actions from the keyboard, buttons and relative to the mouse position (locator). There are also other events allowing to verify the state of the device (e.g. an event to inform there was a change in the size of a window).

The access to events queue is performed by a single data input routine `gpevent`, allowing the user to interact with the system. This routine is used to retrieve the first event from the event queue associated to the window in GP. The parameter `wait` determines the behavior of the routine.

`wait!=0`, waits until the next event;

`wait == 0`, returns if the queue is empty.

28 ⟨*event* 28⟩≡ (? 0—1)

```
char* gpevent(int wait, real* x, real* y)
{
  int ix,iy;
  char* r=dvevent(wait,&ix,&iy);
  *x=ix; *y=iy;
  gpunview(x,y);
  return r;
}
```

Defines:
`gpevent`, never used.
Uses `dvevent`, `gpunview` 25b, and `real` 60 60.

The routine returns events according to the code below:

`bi+` - button $i$ is pressed;

`bi-` - button $i$ is released;

`kt+` - key $t$ is pressed;

`ii+` - cursor not moving with button $i$ pressed;

`mi+` - cursor moving with button $i$ pressed;

`q+` - window closed by the windowing manager.

`r+` - request for re-drawing;

`s+` - window has new size $(x, y)$.

**Button Events.** In a standard hardware configuretion used by GP, the button devices correspond to the mouse buttons. The sequence of data of this event begins with the character `b`, followed by a digit, 1, 2 or 3, identifying the button, and finally the `+` sign to indicate that the button was pressed or `-` to indicate it was released. In short, the sequence of button events has the format `bi+` or `bi -`.

**Locator Events.** The mouse, besides being a button device, is also the standard locator used in a workstation. Mouse motion events begin with the character `m`. In this case, the position of the mouse is stored in the parameter $(x, y)$ of the routine `gpevent`. We should observe that simultaneous mouse motion events using buttons are also preceded by the character `m`. For example, the mouse motion with the button `i` pressed is indicated by the sequence `mi+`.

**Keyboard Events.** When the keyboard key $k$ is pressed, it returns the string "kt+", indicating the event, where $t$ is the ASCII code of the key.

## 2.6 Comments and References

The reader can find more information on planar graphics objects and representation in (**?**). FUrther informatios on graphics devices can be found in (**?**).

The external API of GP is composed of the following routines:

29 ⟨*API* 29⟩≡

```
real    gpopen          (char* name);
void    gpclose         (int wait);
void    gpclear         (int wait);
void    gpflush         (void);
void    gpwait          (int t);

real    gpwindow        (real xmin, real xmax, real ymin, real ymax);
real    gpviewport      (real xmin, real xmax, real ymin, real ymax);
void    gpview          (real* x, real* y);
void    gpunview        (real* x, real* y);

int     gppalette       (int c, char* name);
int     gprgb           (int c, real r, real g, real b);
int     gpcolor         (int c);
int     gpfont          (char* name);

void    gpbegin         (int c);
int     gppoint         (real x, real y);
void    gpend           (void);

void    gptext          (real x, real y, char* s, char* mode);

char*   gpevent         (int wait, real* x, real* y);
```

Defines:
`gpevent`, never used.
Uses `gpbegin`, `gpclear`, `gpclose`, `gpcolor`, `gpend`, `gpflush`, `gpfont`, `gppalette`, `gppoint`, `gprgb`, `gptext`, `gpunview` 25b, `gpview` 25a, `gpwait`, and `real` 60 60.

### 2.6.1 Programming Layer

The GP graphic library implements the lowest layer of an interactive graphics program. It is restricted to the two-dimensional component and it corresponds to the 2D functionality of the OpenGL library.

## 2.7 Exercises

**2.1.** Compile and install the GP library.

**2.2.** Using the GP library, write an interactive program to model polygonal curves. The program should write the curve as a list of points.

**2.3.** Using the GP library, write a program to read files with polygonal curves and to draw them.

**2.4.** Design and implement a toolkit interface consisting of the following 2D widgets: button; valuator; choice; text area and canvas.

**2.5.** Using the toolkit of the previous exercise, write a program showing a menu composed of several buttons. When a button is pressed, the program should print the text on the button.

**2.6.** Using the toolkit of the previous interface, write a program showing a valuator. When the valuator is modified, the program should print the corresponding value.

**2.7.** Combine the programs of the previous exercises to implement a complete editor for polygonal curves. It should contain a menu for the different functions (to read a file, to write a file, to clean the screen, etc), valuators for window scaling and translations, and editing functions associated to the mouse buttons (to insert a vertex, to move a vertex, to delete a vertex).

**2.8.** Modify the editor for polygonal curves to also work with Bézier curves. Use a subdivision algorithm for the visualization by curve refinement. Figure 2.12 shows the example of a curve editor.

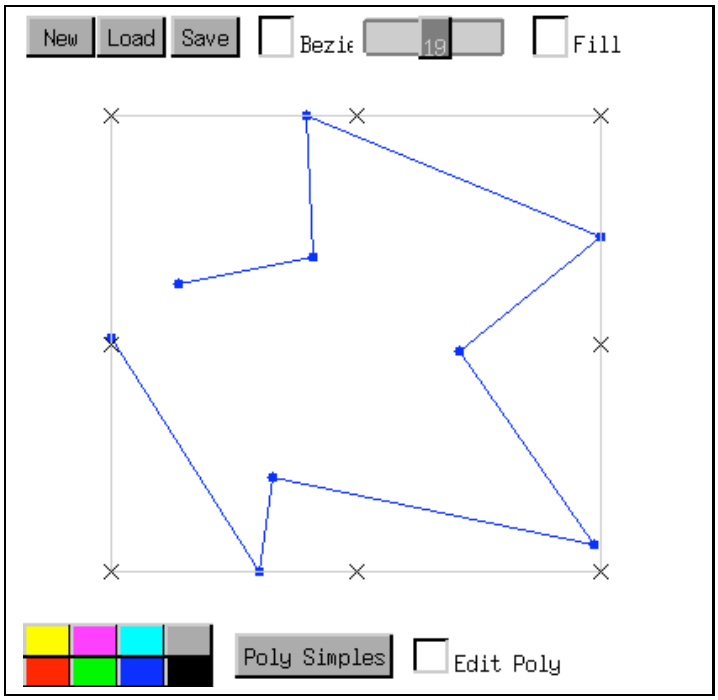



**Figure 2.12** Curve Editor.