2. Graphics primitives and environments

Learning objectives:

- turtle graphics
- QuickDraw: A graphics toolbox
- frame program
- interactive graphics input/output
- example: polyline input

Turtle graphics: a basic environment

Seymour Papert [Pap80] introduced the term *turtle graphics* to denote a set of primitives for line drawing. Originally implemented in the programming language Logo, turtle graphics primitives are now available for several computer systems and languages. They come in different versions, but the essential point is the same as that introduced in the example of the robot car: The pen (or "turtle") is a device that has a state (position, direction) and is driven by incremental operations “move” and “turn” that transform the turtle to a new state depending on its current state:

```
move(s)        { take s unit steps in the direction you are facing }
turn(d)        { turn counterclockwise d degrees }
```

The turtle's initial state is set by the following operations:

```
moveto(x,y)    { move to the position (x,y) in absolute coordinates }
turnto(d)      { face d degrees from due east }
```

In addition, we can specify the color of the trail drawn by the moving pen:

```
pencolor(c) { where c = white, black, none, etc. }
```

Example

The following program fragment approximates a circle tangential to the x-axis at the origin by drawing a 36-sided polygon:

```
moveto(0, 0); { position pen at origin }
turnto(0);    { face east }
step := 7;    { arbitrarily chosen step length }
do 36 times   { 36 sides · 10° = 360° }
   { move(step);  turn(10) } { 10 degrees counterclockwise }
```

In graphics programming we are likely to use basic figures, such as circles, over and over again, each time with a different size and position. Thus we wish to turn a program fragment such as the circle approximation above into a reusable procedure.

Procedures as building blocks

A program is built from components at many different levels of complexity. At the lowest level we have the constructs provided by the language we use: constants, variables, operators, expressions, and simple (unstructured) statements. At the next higher level we have *procedures*: they let us refer to a program fragment of arbitrary size and complexity as a single entity, and build hierarchically nested structures. Modern programming languages provide yet another level of packaging: *modules*, or *packages*, useful for grouping related data and procedures. We limit our discussion to the use of procedures.

Programmers accumulate their own collection of useful program fragments. Programming languages provide the concept of a *procedure* as the major tool for turning fragments into *reusable* building blocks. A procedure consists of two parts with distinct purposes:

1. The *heading* specifies an important part of the procedure's external behavior through the list of *formal parameters*: namely, what type of data moves in and out of the procedure.
2. The *body* implements the action performed by the procedure, processing the input data and generating the output data.

A program fragment that embodies a single coherent concept is best written as a procedure. This is particularly true if we expect to use this fragment again in a different context. The question of how general we want a procedure to be deserves careful thought. If the procedure is too specific, it will rarely be useful. If it is too general, it may be unwieldy: too large, too slow, or just too difficult to understand. The generality of a procedure depends primarily on the choice of formal parameters.

Example: the long road toward a procedure “circle”

Let us illustrate these issues by discussing design considerations for a procedure that draws a circle on the screen. The program fragment above for drawing a regular polygon is easily turned into

```
procedure ngon(n,s: integer);  { n = number of sides, s = step
                                  size }
var  i,j: integer;
begin
  j := 360 div n;
  for i := 1 to n do  { move(s);  turn(j) }
end;
```

But, a useful procedure to draw a circle requires additional arguments. Let us start with the following:

```
procedure circle(x, y, r, n: integer);
  { centered at (x, y);  r = radius;  n = number of sides }
var  a, s, i: integer;  { angle, step, counter }
begin
  moveto(x, y – r);  { bottom of circle }
  turnto(0);  { east }
  a := 360 div n;
  s := r · sin(a);  { between inscribed and circumscribed polygons }
  for  i := 1  to  n  do  { move(s);  turn(a) }
end;
```

This procedure places the burden of choosing n on the programmer. A more sophisticated, "adaptive" version might choose the number of sides on its own as a function of the radius of the circle to be drawn. We assume that lengths are measured in terms of pixels (picture elements) on the screen. We observe that a circle of radius r is of length 2πr. We approximate it by drawing short-line segments, about 3 pixels long, thus needing about 2·r line segments.

```
   procedure circle(x, y, r: integer);  { centered at (x, y);  radius
 r}
   var  a, s, i: integer;  { angle, step, counter }
   begin
     moveto(x, y – r);  { bottom of circle }
     turnto(0);  { east }
     a := 180 div r;  { 360 / (# of line segments) }
     s := r · sin(a);  { between inscribed and circumscribed polygons }
     for  i := 1  to  2 · r  do  { move(s);  turn(a) }
   end;
```

This circle procedure still suffers from severe shortcomings:

1. If we discretize a circle by a set of pixels, it is an unnecessary detour to do this in two steps as done above: first, discretize the circle by a polygon; second, discretize the polygon by pixels. This two-step process is a source of unnecessary work and errors.
2. The approximation of the circle by a polygon computed from vertex to vertex leads to *rounding error*s that accumulate. Thus the polygon may fail to close, in particular when using integer computation with its inherent large rounding error.
3. The procedure attempts to draw its circle on an infinite screen. Computer screens are finite, and attempted drawing beyond the screen boundary may or may not cause an error. Thus the circle ought to be clipped at the boundaries of an arbitrarily specified rectangle.

Writing a good circle procedure is a demanding task for professionals. We started this discussion of desiderata and difficulties of a simple library procedure so that the reader may appreciate the thought and effort that go into building a useful programming environment. In chapter 14 we return to this problem and present one possible goal of "the long road toward a procedure 'circle'". We now make a huge jump from the artificially small environments discussed so far to one of today's realistic programming environments for graphics

QuickDraw: a graphics toolbox

For the sake of concreteness, the next few sections show programs written for a specific programming environment: MacPascal using the QuickDraw library of graphics routines [App 85]. It is not our purpose to duplicate a manual, but only to convey the flavor of a realistic graphics package and to explain enough about QuickDraw for the reader to understand the few programs that follow. So our treatment is highly selective and biased.

Concerning the circle that we attempted to program above, QuickDraw offers five procedures for drawing circles and related figures:

```
procedure FrameOval(r: Rect);
procedure PaintOval(r: Rect);
procedure EraseOval(r: Rect);
procedure InvertOval(r: Rect);
procedure FillOval(r: Rect; pat: Pattern);
```

Each one inscribes an oval in an aligned rectangle r (sides parallel to the axes) so as to touch the four sides of r. If r is a square, the oval becomes a circle. We quote from [App 85]:

FrameOval draws an outline just inside the oval that fits inside the specified rectangle, using the current grafPort's pen pattern, mode, and size. The outline is as wide as the pen width and as tall as the pen height. It's drawn with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

Right away we notice a trade-off when comparing QuickDraw to the simple turtle graphics environment we introduced earlier. At one stroke, "FrameOval" appears to be able to produce many different pictures, but before we can exploit this power, we have to learn about grafPorts, pen width, pen height, pen patterns, and pattern transfer modes. 'FrameOval' draws the perimeter of an oval, 'PaintOval' paints the interior as well, 'EraseOval' paints an oval with the current grafPort's background pattern, 'InvertOval' complements the pixels: 'white' becomes 'black', and vice versa. 'FillOval' has an additional argument that specifies a pen pattern used for painting the interior.

We may not need to know all of this in order to use one of these procedures, but we do need to know how to specify a rectangle. QuickDraw has predefined a type 'Rect' that, somewhat ambiguously at the programmer's choice, has either of the following two interpretations:

```
    type Rect  = record  top, left, bottom, right: integer  end;
    type Rect  = record  topLeft, botRight: Point  end;
  with one of the interpretations of type 'Point' being
    type Point = record  v, h: integer  end;
```

Exhibit 2.1 illustrates and provides more information about these concepts. It shows a plane with first coordinate v that runs from top to bottom, and a second coordinate h that runs from left to right. (The reason for v running from top to bottom, rather than vice versa as used in math books, is compatibility with text coordinates where lines are naturally numbered from top to bottom.) The domain of v and h are the integers from $-2^{15} = -32768$ to $2^{15} - 1 = 32767$. The points thus addressed on the screen are shown as intersections of grid lines. These lines and grid points are infinitely thin - they have no extension. The pixels are the unit squares between them. Each pixel is paired with its top left grid point. This may be enough information to let us draw a slightly fat point of radius 3 pixels at the grid point with integer coordinates (v, h) by calling

```
    PaintOval(v - 3, h - 3, v + 3, h + 3);
```

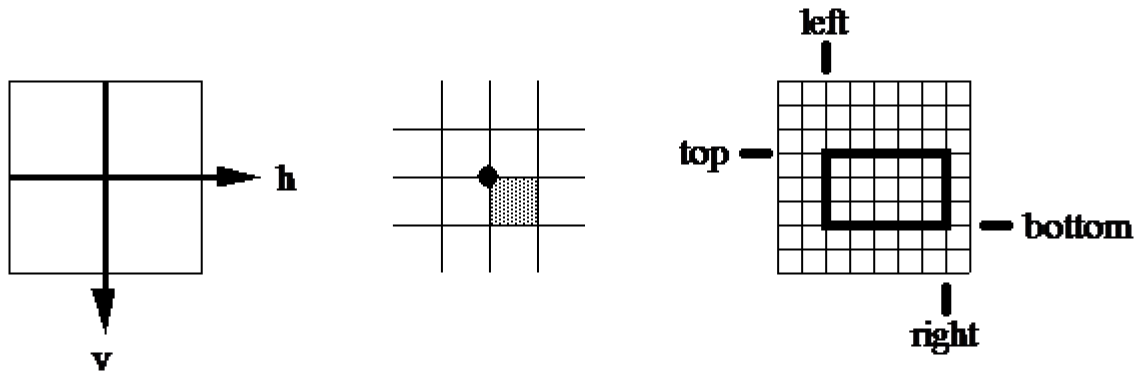

Exhibit 2.1: Screen coordinates define the location of pixels.

To understand the procedures of this section, the reader has to understand a few details about two key aspects of interactive graphics:

- timing and synchronization of devices and program execution
- how screen pictures are controlled at the pixel level

Synchronization

In interactive applications we often wish to specify a grid point by letting the user point the mouse-driven cursor to some spot on the screen. The 'procedure GetMouse(v, h)' returns the coordinates of the grid point where the cursor is located at the moment 'GetMouse' is executed. Thus we can track and paint the path of the mouse by a loop such as

```
  repeat  GetMouse(v, h);  PaintOval(v – 3, h – 3, v + 3, h + 3)
until stop;
```

This does not give the user any timing control over when he or she wants the computer to read the coordinates of the mouse cursor. Clicking the mouse button is the usual way to tell the computer "Now!". A predefined boolean function 'Button' returns 'true' when the mouse button is depressed, 'false' when not. We often synchronize program execution with the user's clicks by programming *busy waiting loops*:

```
  repeat until Button; { waits for the button to be pressed }
  while Button do; { waits for the button to be released }
The following procedure waits for the next click:
  procedure waitForClick;
  begin  repeat until Button;  while Button do  end;
```

Pixel acrobatics

The QuickDraw pen has four parameters that can be set to draw lines or paint textures of great visual variety: pen location 'pnLoc', pen size 'pnSize' (a rectangle of given height and width), a pen pattern 'pnPat', and a drawing mode 'pnMode'. The pixels affected by a motion of the pen are shown in Exhibit 2.2.

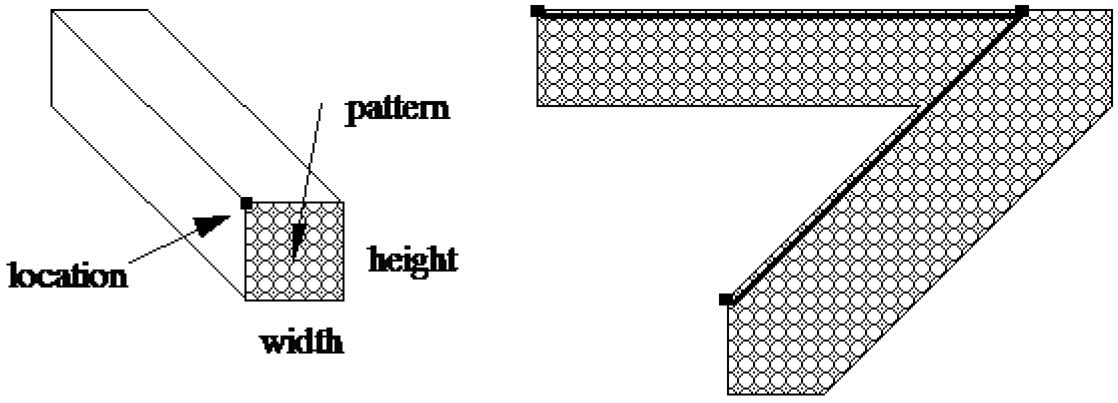

Exhibit 2.2: Footprint of the pen.

Predefined values of 'pnPat' include 'black', 'gray', and 'white'. 'pnPat' is set by calling the predefined 'procedure PenPat(pat: Pattern)' [e.g. 'PenPat(gray)']. As 'white' is the default background, drawing in 'white' usually serves for erasing.

The result of drawing also depends critically on the transfer mode 'pnMode', whose values include 'patCopy', 'patOr', and 'patXor'. A transfer mode is a boolean operation executed in parallel on each pair of pixels in corresponding positions, one on the screen and one in the pen pattern.

- 'patCopy' uses the pattern pixel to overwrite the screen pixel, ignoring the latter's previous value; it is the default and most frequently used transfer mode.
- 'patOr' paints a black pixel if either or both the screen pixel or the pattern pixel were black; it progressively blackens the screen.

- 'patXor' (*exclusive-or*, also known as "odd parity") sets the result to black iff exactly one of (screen pixel, pattern pixel) is black. A white pixel in the pen leaves the underlying screen pixel unchanged; a black pixel complements it. Thus a black pen inverts the screen.

'pnMode' is set by calling the predefined 'procedure PenMode(mode: integer)' [e.g. 'PenMode(patXor)'].

The meaning of the remaining predefined procedures our examples use, such as 'MoveTo' and 'LineTo', is easily guessed. So we terminate our peep into some key details of a powerful graphics package, and turn to examples of its use.

A graphics frame program

Reusable software is a time saving concept that can be practiced profitably in the small. We keep a program that contains nothing but a few of the most useful input/output procedures, displays samples of their results, and conducts a minimal dialog so that the user can step through its execution. We call this a *frame program* because its real purpose is to facilitate development and testing of new procedures by embedding them in a ready-made, tested environment. A simple frame program like the one below makes it very easy for a novice to write his first interactive graphics program.

This particular frame program contains procedures 'GetPoint', 'DrawPoint', 'ClickPoint', 'DrawLine', 'DragLine', 'DrawCircle', and 'DragCircle' for input and display of points, lines, and circles on a screen idealized as a part of a Euclidean plane, disregarding the discretization due to the raster screen. Some of these procedures are so short that one asks why they are introduced at all. 'GetPoint', for example, only converts integer mouse coordinates v, h into a point p with real coordinates. It enables us to refer to a point p without mentioning its coordinates explicitly. Thus, by bringing us closer to standard geometric notation, 'GetPoint' makes programs more readable.

The procedure 'DragLine', on the other hand, is a very useful routine for interactive input of line segments. It uses the *rubber-band technique*, which is familiar to users of graphics editors. The user presses the mouse button to fix the first endpoint of a line segment, and keeps it depressed while moving the mouse to the desired second endpoint. At all times during this motion the program keeps displaying the line segment as it would look if the button were released at that moment. This rubber band keeps getting drawn and erased as it moves across other objects on the screen. The user should study a key detail in the procedure 'DragLine' that prevents other objects from being erased or modified as they collide with the ever-refreshed rubber band: We temporarily set 'PenMode(patXor)'. We encourage you to experiment by modifying this procedure in two ways:

1. Change the first call of the 'procedure DrawLine(L.p_1, L.p_2, black)' to 'DrawLine(L.p_1, L.p_2, white)'. You will have turned the procedure 'DragLine' into an artful, if somewhat random, painting brush.
2. Remove the call 'PenMode(patXor)' (thus reestablishing the default 'pnMode = patCopy'), but leave the first 'DrawLine(L.p_1, L.p_2, white)', followed by the second 'DrawLine(L.p_1, L.p_2, black)'. You now have a naive rubber-band routine: It alternates erasing (draw 'white') and drawing (draw 'black') the current rubber band, but in so doing it modifies other objects that share pixels with the rubber band. This is our first example of the use of the versatile *exclusive-or*; others will follow later in the book.

```
 program Frame;
   { provides mouse input and drawing of points, line segments,
circles }

 type point = record  x, y: real  end;
   lineSegment = record  p1, p2: point  { endpoints }  end;
```

```
  var  c, p: point;
    r: real;  { radius of a circle }
    L: lineSegment;

  procedure WaitForClick;
  begin  repeat until Button;  while Button do  end;

  procedure GetPoint (var p: point);
  var  v, h: integer;
  begin
    GetMouse(v, h);
    p.x := v;  p.y := h  { convert integer to real }
  end;

  procedure DrawPoint(p: point; pat: Pattern);
  const  t = 3;  { radius of a point }
  begin
    PenPat(pat);
    PaintOval(round(p.y) - t, round(p.x) - t, round(p.y) + t,
round(p.x) + t)
  end;

  procedure ClickPoint(var p: point);
  begin  WaitForClick;  GetPoint(p);  DrawPoint(p, Black)  end;

  function Dist(p, q: point): real;
  begin  Dist := sqrt(sqr(p.x - q.x) + sqr(p.y - q.y))  end;

  procedure DrawLine(p1, p2: point; pat: Pattern);
  begin
    PenPat(pat);
    MoveTo(round(p1.x), round(p1.y));
    LineTo(round(p2.x), round(p2.y))
  end;

  procedure DragLine(var L: lineSegment);
  begin
    repeat until Button;  GetPoint(L.p1);  L.p2 := L.p1;
PenMode(patXor);
    while Button do  begin
      DrawLine(L.p1, L.p2, black);
      { replace 'black' by 'white' above to get an artistic drawing
tool }
      GetPoint(L.p2);
      DrawLine(L.p1, L.p2, black)
    end;
    PenMode(patCopy)
  end;  { DragLine }

  procedure DrawCircle(c: point; r: real; pat: Pattern);
  begin
    PenPat(pat);
    FrameOval(round(c.y - r), round(c.x - r), round(c.y + r),
round(c.x + r))
  end;

  procedure DragCircle(var c: point; var r: real);
  var  p: point;
  begin
    repeat until Button;  GetPoint(c);  r := 0.0;  PenMode(patXor);
    while Button do  begin
      DrawCircle(c, r, black);
      GetPoint(p);
      r := Dist(c, p);
      DrawCircle(c, r, black);
    end;
    PenMode(patCopy)
  end;  { DragCircle }

  procedure Title;
  begin
    ShowText;  { make sure the text window and … }
    ShowDrawing;  { … the graphics window show on the screen }
    WriteLn('Frame program');
    WriteLn('with simple graphics and interaction routines.');
    WriteLn('Click to proceed.');
    WaitForClick
  end;  { Title }

  procedure What;
  begin
    WriteLn('Click a point in the drawing window.');
    ClickPoint(p);
    WriteLn('Drag mouse to enter a line segment.');
    DragLine(L);
    WriteLn('Click center of a circle and drag its radius');
    DragCircle(c, r)
  end;  { What }

  procedure Epilog;
  begin  WriteLn('Bye.')  end;

  begin  { Frame }
    Title;  What;  Epilog
  end.  { Frame }
```

Example of a graphics routine: polyline input

Let us illustrate the use of the frame program above in developing a new graphics procedure. We choose interactive polyline input as an example. A *polyline* is a chain of directed straight-line segments—the starting point of the next segment coincides with the endpoint of the previous one. 'Polyline' is the most useful tool for interactive input of most drawings made up of straight lines. The user clicks a starting point, and each subsequent click extends the polyline by another line segment. A double click terminates the polyline.

We developed 'PolyLine' starting from the frame program above, in particular the procedure 'DragLine', modifying and adding a few procedures. Once 'Polyline' worked, we simplified the frame program a bit. For example, the original frame program uses reals to represent coordinates of points, because most geometric computation is done that way. A polyline on a graphics screen only needs integers, so we changed the type 'point' to integer coordinates. At the moment, the code for polyline input is partly in the procedure 'NextLineSegment' and in the procedure 'What'. In the next iteration, it would probably be combined into a single self-contained procedure, with all the subprocedures it needs, and the frame program would be tossed out—it has served its purpose as a development tool.

```
  program PolyLine;
    { enter a chain of line segments and compute total length }
    { stop on double click }

  type point = record  x, y: integer;  end;
  var  stop: boolean;
    length: real;
    p, q: point;

  function EqPoints (p, q: point): boolean;
  begin  EqPoints := (p.x = q.x) and (p.y = q.y)  end;

  function Dist (p, q: point): real;
  begin  Dist := sqrt(sqr(p.x – q.x) + sqr(p.y – q.y))  end;

  procedure DrawLine (p, q: point; c: Pattern);
  begin  PenPat(c);  MoveTo(p.x, p.y);  LineTo(q.x, q.y)  end;

  procedure WaitForClick;
  begin  repeat until Button;  while Button do  end;

  procedure NextLineSegment (var stp, endp: point);
  begin
    endp := stp;
    repeat
      DrawLine(stp, endp, black);  { Try 'white' to generate artful
pictures! }
      GetMouse(endp.x, endp.y);
      DrawLine(stp, endp, black)
    until Button;
    while Button do
  end;  { NextLineSegment }

  procedure Title;
  begin
    ShowText;  ShowDrawing;
    WriteLn('Click to start a polyline.');
    WriteLn('Click to end each segment.');
    WriteLn('Double click to stop.')
  end;  { Title }

  procedure What;
  begin
    WaitForClick;  GetMouse(p.x, p.y);
    stop := false;  length := 0.0;
    PenMode(patXor);
    while  not stop  do  begin
      NextLineSegment(p, q);
      stop := EqPoints(p, q);  length := length + Dist(p, q);  p := q
    end
  end;  { What }

  procedure Epilog;
  begin  WriteLn('Length of polyline = ', length);  WriteLn('Bye.')
end;

  begin  { PolyLine }
    Title;  What;  Epilog
  end.  { PolyLine }
```

Programming projects

1. Implement a simple package of turtle graphics operations on top of the graphics environment available on your computer.
2. Use this package to implement and test a procedure 'circle' that meets the requirements listed at the end of the section “Turtle graphics: a basic environment”.

3. Implement your personal graphics frame program as described in “A graphics frame program”. Your effort will pay off in time saved later, as you will be using this program throughout the entire course.