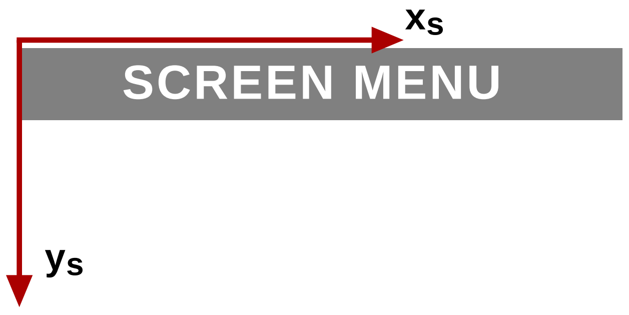
APPENDIX B - GRAPHICS

Most computer simulation work produces lots of numerical data. The analysis of that data is a challenge in itself, but displaying the data in a way that allows easy interpretation may be useful, if not essential. This is particularly true for simulations that involve very long codes in which algorithmic errors are difficult to detect. Not only may an effective visual display expose coding errors, it may also point out important behavior not seen in simple numerical analysis. One can think of many examples from fluid dynamics to molecular structure where the display of data plays a pivotal role in the understanding of a system. *Visualization* in scientific computing is an important subfield in its own right.

In this section, we describe some elements of computer graphics that are appropriate to the Apple PowerPCs of the Computational Physics Lab. Further editions of these notes will include Windows versions of the graphics. Our purpose is to introduce some elements of computer graphics, both in a somewhat generic form in Sec. B.1, and in a Mac OS specific way in Sec. B.2. The project of Section 1 involves the simple tasks of drawing on a screen, reading the position of the mouse, and using the mouse to control the actions of a code.

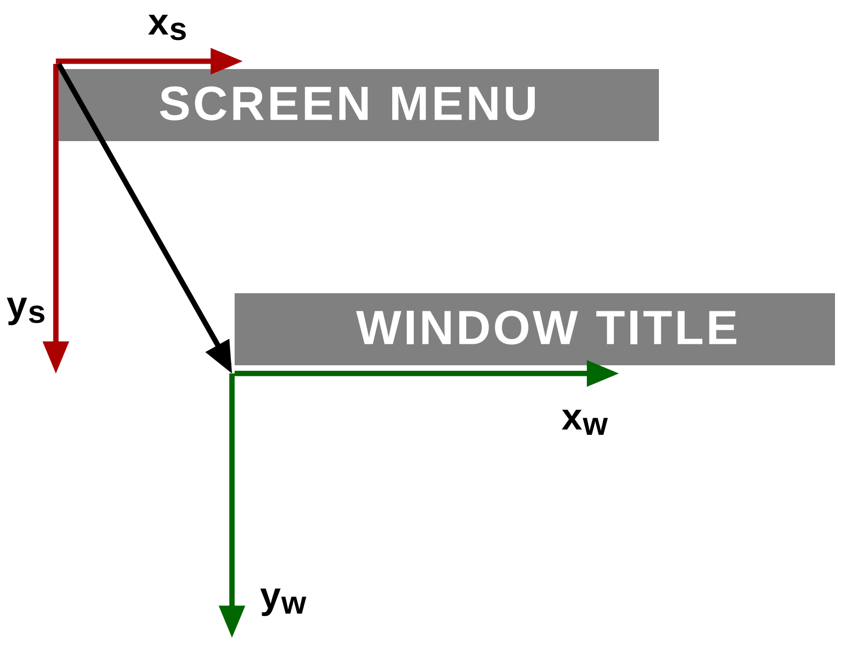
B.1 Coordinates

A two-dimensional computer screen can be described by a conventional *xy* coordinate system. But the Mac OS has an unconventional origin: the *x*-axis runs from left to right, as usual, but the *y*-axis runs from top to bottom:



We denote coordinates in the screen system by (x_s , y_s). In general, this coordinate system is only used to describe the position of the window within which the actual drawing takes place.

The drawing window itself has a separate coordinate system. The advantage to having two coordinate systems is that the coordinates of objects within the window can be made local, and do not change with the position of the window. We denote the coordinates of an object within a window as (x_w , y_w). The origin of the window coordinate system (x_w , y_w) is:



Thus, to establish a drawing window, one first determines where the window is to be placed on the screen, by defining the upper left corner of the window and the lower right corner of the window, in terms of the screen coordinate system (x_s , y_s) *as measured from the upper left corner of the screen, which may include the screen menu.* The window coordinate origin is the upper left position of the drawing screen*, and does not include the Window Title bar.*

To make life easier, there are many pre-defined drawing functions within most operating systems, so each pixel does not have to be individually labelled by the user. For example, in the QuickDraw graphics library for the Mac OS, there are functions to define polygons, fill polygons, outline polygons *etc*. There also are functions to reverse the color of a region, oval or rectangle (a quick way to do the black-and-white inversion one sees when scrolling down a menu). A small sample of graphics functions for QuickDraw is given in Sec. B.2, and it is straightforward to translate these functions to their Windows or unix equivalents.

Frequently, there are two variations of a given routine, allowing one to work in absolute coordinates as well as relative coordinates. We have already come across this with (x_s , y_s) and (x_w , y_w), introduced so that the contents of an entire window can be moved simply by changing (x_s , y_s). Even working within a window, the "pen" can be moved to a new position either by specifying a new absolute coordinate within the window (x_w , y_w), or by specifying the coordinates of the new pen position *relative* to the current pen position. The names of the absolute and relative functions are similar, so they are easy to remember; however, a drawing may take on a certain abstract appearance if the functions are used incorrectly. For example, MoveTo(x_w , y_w) uses absolute (window) coordinates, while Move(x_r , y_r) uses relative coordinates.

The author, like many mature (*i.e.* older) research scientists, remembers the awkward operating systems of the sixties and seventies, in which the instructions for simple operations filled several manuals and notebooks. Many of the operating system commands had few mnemonics to remind the user of their purpose. Some numerical libraries still have not left this tradition behind. The great advance of the Mac OS in the early eighties was the introduction of icons, and the use of the cursor to generate operating system commands. Included in Sec. B.2 are the simplest mouse commands, that will allow us to control a window in the project of Sec. B.3.

B.2 QuickDraw

The PowerPCs use the QuickDraw graphics library for two-dimensional drawings. There are a very large number of graphics routines in the QuickDraw library, and we present only a fraction of them here. Further information on the routines, including examples, can be found using the **THINK** reference utility. We group the routines according to function.

Note: The same symbol **rec0** is used throughout Sec. B.2 to represent a rectangle (see Sec. 1B.6). Most graphics codes contain many different rectangles, and it is generally more convenient, and often necessary, to use different symbols for each: rec0, rec1, rec2, ...

B.2.1 Declarations

As usual with C, the graphics elements must be declared. Examples are:

WindowPtr myWind;	pointer for a window named **myWind**
Rect rec0;	declares a rectangle **rec0**
PolyHandle myPoly;	declares a polygon named **myPoly**
PicHandle myPict;	declares a picture named **myPict**
Point pt0;	declares a variable **pt0** for the cursor coordinates
Pattern pat0;	declares a pattern **pat0** for the drawing; **pat0** may be used to fill a region, or draw a shaded line

B.2.2 Initializations

Not all of these statements are needed in most codes, but we include them all for completeness. They are self-explanatory, and should be placed at the beginning of the routine before the window is set up:

```
InitGraf(&qd.thePort);
InitFonts();
InitWindows();
InitMenus();
TEInit();
InitDialogs(nil);
InitCursor();
```

B.2.3 Definitions

There are several functions that define patterns for the drawing. Some predefined patterns can be obtained using the command

```
GetIndPattern(&pat0, sysPatListID, nn);
```

which assigns a particular pattern **nn** to the previously-declared **pat0**. The patterns in **sysPatListID** include simple grayscales as well as tilings such as regularly and irregularly placed bricks *etc* (the complete list can be found in **THINK**). Some personal favorites of the author for doing grayscale images are (substitute these for **nn**):

1	black
3	very dark gray
4	dark gray
23	medium gray
22	light gray
21	very light gray
20	white

Depending on the current color setting of the code, invoking a nominally "black" pattern, from **GetIndPattern(&pat0, sysPatListID, 1);** may not produce black. Rather, pattern number **1** corresponds to a solid color, which may be red or something else defined by the user.

To fix the current pattern for drawing lines, use:

PenPat(&pat0);

once **pat0** has been declared and defined. Definitions of text size and font are given in Sec. B.2.8.

B.2.4 Define the color

All of the demonstration codes provided for this course are drawn using primary colors. An older routine for defining colors is

ForeColor(theColor);

where, in obvious notation, **theColor** is one of
whiteColor
yellowColor
greenColor
cyanColor
blueColor
magentaColor
redColor
blackColor.
Specific functions can be used for defining RGB colors (see **THINK**).

B.2.5 Define the geometry

Routines are available both to define the width of the pen stroke, and to specify the regions which will be filled or outlined. The function

```
PenSize(xx,yy);
```

defines the current line to be xx pixels wide and yy pixels high. The function which defines the size of rectangles is

```
SetRect(&rec0, x1, y1, x2, y2);
```

where **(x1,y1)** are the upper left coordinates of the rectangle and **(x2,y2)** are the lower right coordinates. Rectangles are used for:
•fixing the position and size of the drawing window, in which case **(x1,y1)** are screen coordinates (x_S,y_S).
•fixing the position and size of a drawing rectangle within the window, in which case **(x1,y1)** are window coordinates (x_W,y_W).

Just because a region has been defined as a rectangle does not mean that it can be used just to draw rectangles. Drawing routines which produce rectangles with rounded edges (**FrameRoundRect** and **FillRoundRect**) or ovals (**FrameOval** and **FillOval**) use the rectangular boundaries set by **SetRect**. See Sec. 1B.8 or **THINK** for details of the calls to these functions.

Of course, there are many situations in which it is more convenient to define a polygon, than make repeated calls to a rectangle. The sequence to define a polygon starts with **OpenPoly** and ends with **ClosePoly**. Assuming that a structure named **myPoly** has been declared in the code (as in Sec. 1B.2), then the sequence is

```
myPoly=OpenPoly();
MoveTo(x1,y1);
LineTo(x2,y2);
LineTo(x3,y3);
...
LineTo(x1,y1);
ClosePoly();
```

The commands **MoveTo** and **LineTo** are described in the next section. Note that the polygon begins and ends at the same position.

B.2.6 Moving the pen

There are two commonly-used pairs of functions for moving the drawing pen. The only difference between members of a pair is that one function uses absolute coordinates **(x_w,y_w)** with respect to the window origin, while the other function uses relative coordinates with respect to the current position of the pen **(x_r,y_r)**. An example of relative coordinates would be 2 pixels over and 1 up from the current pen position. The following pair of functions move the position of the pen:

MoveTo(x_w,y_w);
Move(x_r,y_r);

in which the pen is moved to a new location without drawing a line. The following pair of functions draw a line while the pen moves

LineTo(x_w,y_w);
Line(x_r,y_r);

in which case a straight line is drawn from the current coordinates to the new coordinates given in the argument **(x,y)**. The width of the line is set by **PenSize**, and the pattern is set by **PenPat**.

B.2.7 Framing and filling

Rectangles and polygons define two-dimensional regions of the drawing. Routines are available to frame the region (*i.e.*, draw a border around the region) or to fill it. For example, if the region is **rec0**, then the form of the frame drawing is:

FrameRect(&rec0);
FrameOval(&rec0);

The frame is drawn with the current pen setting. Examples of functions that fill the region **rec0** with the pattern is **pat0** are:

FillRect(&rec0, &pat0);
FillPoly(&rec0, &pat0);

B.2.8 Text

Two functions used to define text size and font are:

TextSize(nn); where **nn** is the size of the font in points (this text is 12 pt). A point is 1/72 of an inch, about the size of one pixel.

TextFont(nn); where **nn** is a number associated with a font. Examples include

0	Chicago
1	AppleFont
2	New York
21	Helvetica
23	Symbol.

Routines are also available for measuring the width of a string or character. For example,
StringWidth("\pword");
returns the length, in pixels, of the Pascal string **word** as it would be written in the current textsize and textfont.

For drawing the Pascal string **word**, the obvious function is
DrawString("\pword");

B.2.9 Opening and closing the drawing

The following set of commands are used to begin the drawing:

```
wind = NewWindow(nil, &rec0, "\p", TRUE, documentProc,
                (WindowPtr) -1, FALSE, 0);
SetPort(wind);
myPict = OpenPicture(&rec0);
ShowPen();
```

where **myPict** is declared with **PicHandle** (see Sec. 1B.2). The effect of the drawing routines on the screen is recorded once **ShowPen** is called. When the drawing is complete, the following commands close the window

```
PenNormal();
HidePen();
ClosePicture();
```

B.2.10 Cursor commands

Interactive graphics require that a code recognize the position of the cursor and recognize whether the mouse button is up or down. There are several routines for cursor operations, and we mention only three of them:

GetMouse(&pt1);	obtains the cursor coordinates, stored in **pt1**
PtInRect(pt1, &rec0);	returns true if the cursor is within the rectangle **rec0**
Button();	returns true if the mouse button is depressed.

The **Button** function can be used in a variety of situations. One way of stalling a code while it waits for input from the mouse is:

```
while( !Button() ) {};
```

which simply executes **{}** as long as the mouse button is up.

B.2.11 Printing the window

You may want to print the contents of the window that your code has drawn. The following routine is offered, without comment, as a mechanism for sending the window drawing to a printer. This printing routine is called using **PrintPicture(myPict, &rec0);** The usual printer dialog box will appear on the screen once the routine is called.

```
void PrintPicture(PicHandle whichPic, Rect *whichDestRect)
{
	GrafPtr   savePort;
	TPrStatus  prStatus;
	TPPrPort   printPort;
	OSErr     err;
	THPrint   hPrint;
	GetPort(&savePort);
	PrOpen();
	hPrint = (THPrint) NewHandle(sizeof(TPrint));
	PrintDefault(hPrint);
	ClipRect(whichDestRect);
	if(PrJobDialog(hPrint)) {
		printPort = PrOpenDoc(hPrint,nil,nil);
```

```
            SetPort(&printPort->gPort);
            PrOpenPage(printPort,nil);
            DrawPicture(whichPic, whichDestRect);
            PrClosePage(printPort);
            PrCloseDoc(printPort);
            if(((*hPrint)->prJob.bJDocLoop = bSpoolLoop) &&
                  (!PrError() ) )
                  PrPicFile(hPrint, nil, nil, nil, &prStatus);
            }
      PrClose();
      SetPort(savePort);
}
```

B.3 Project B - interactive stop sign

In this project, we perform a number of drawing and other activities typical of a graphics-oriented display. We:
•set up a window
•paint it black and draw a red "stop" sign on it
•use a mouse-click in a predefined region to stop the program.

Code

1. Place the upper left corner of the drawing window at position (50,50) of the screen coordinates.

2. Make the drawing window 400 x 400 pixels; color it solid black.

3. Place the center of a red octagonal "stop" sign at the center of the window. The octagon is 200 pixels between opposing faces and is equilateral.

4. Write STOP in white letters (50 pt Helvetica); center the writing using a function that measures the STOP string length.

5. Make a 200 x 200 area in the center of the drawing "active", such that a mouse click in this region stops the code. No other part of the screen should be active.

6. Your main routine will contain the following components:
•**declarations** for: window, picture, cursor, patterns, rectangles (B.2.1)
•**initializations** (B.2.2)
•set up and open the window (B.2.9)

•set up the picture and start to record with the pen (B.2.9)
•color the background (B.2.3), (B.2.4), (B.2.5), (B.2.7)
•set up and fill the octagon (B.2.5), (B.2.6), (B.2.7)
•letter the stop sign (B.2.8)
•read the mouse and control the code (B.2.10)
•stop drawing and close the picture (B.2.9)