

14

Oriel for Windows

Oriel for Windows is a graphics-based batch language for Windows 3 that gives you direct access to the Graphics Device Interface (GDI), the same tool Windows programmers use to create customized Windows programs. Oriel for Windows reads and acts on the commands contained in an ASCII file that can be created with any text editor, such as Notepad.

Oriel for Windows recognizes 33 different commands that let you perform such diverse functions as building custom menus, running executable programs, and drawing graphic objects on the screen using different shapes, colors, and patterns. Here are some examples of what you can do with these commands:

- Build Windows demo programs complete with custom screens, menus, and messages boxes.
- Create front-end shells for launching Windows and DOS applications.
- Write simple draw programs that accept keyboard and mouse input.
- Build hypertext-like programs similar to the ToolBook demo bundled with early copies of Windows 3.0, where you click on a region of the screen (a button, for example) and text pops up giving you more information.
- Experiment with the GDI interactively in a way that is simply not possible with the Windows 3.0 Software Development Kit (SDK).

HOW ORIEL FOR WINDOWS WORKS

If you've ever tried to create a simple Windows program using the Windows SDK and a C compiler, you know that it takes several pages of code just to put a simple message like "Hello world!" on the screen. With Oriel for Windows, creating such a program is easy. All you do is use a text editor (we recommend Notepad) and place the following two commands in an ASCII file:

```
DrawText(10,10,"Hello world!")  
WaitInput()
```

If you then name the ASCII file **HELLO.TXT** and save it in the Oriel directory (for example, **C:\ORIEL**), you can run Oriel from File Manager and have it execute the contents of **HELLO.TXT**. To do so, you can start File Manager, navigate to the Oriel directory, and use the File Run command with the following command line:

```
ORIEL HELLO.TXT
```

Oriel creates the window in Figure 14-1.

Notice that what you've created is a true Windows application, complete with a resizable window, a window Control menu, and Maximize and Minimize buttons. As you can with any other Windows application, you can switch away from your Oriel program, perform work in another application, and then switch back. All the while, your Oriel program continues to execute, waiting for input.

From this point, you can add more Oriel commands to the ASCII file. The most convenient way is to use Notepad to make the additions you want and save the file. You can then execute the program again to see the effect of the new commands.

If you've made a mistake in the script, Oriel terminates it and shows a message, like the one in Figure 14-2, which identifies the errant line. By keeping Notepad open on the desktop, you can easily change your script and test it again.

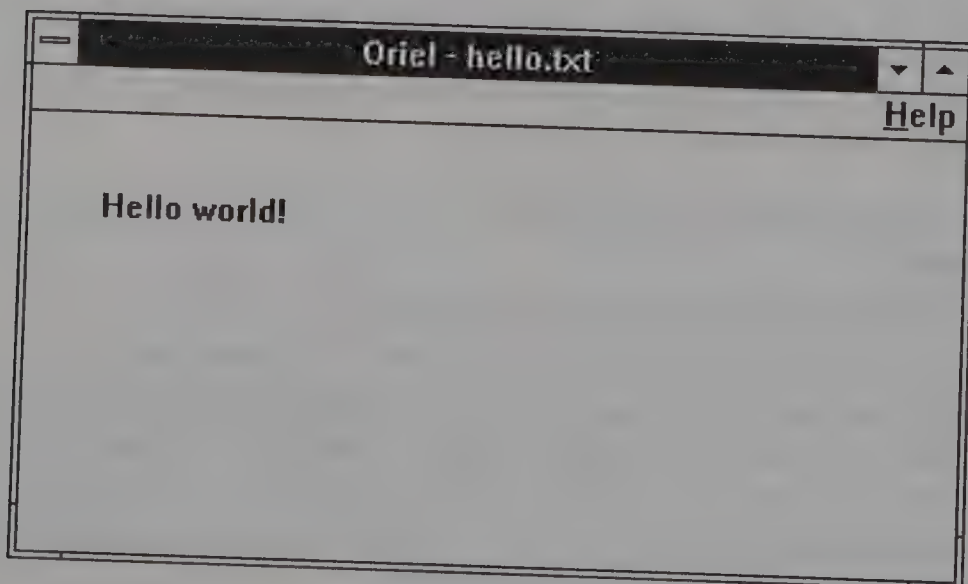


Figure 14-1 Hello world! using Oriel for Windows

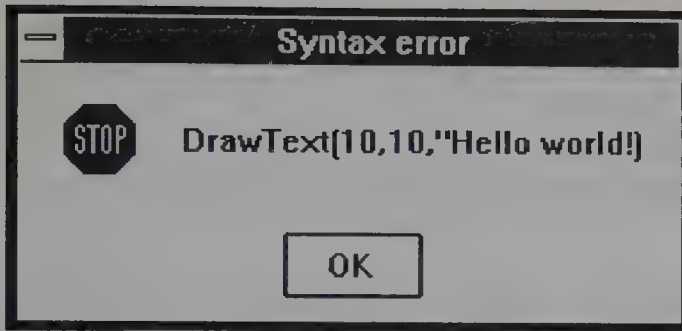


Figure 14-2 A sample syntax error message

Note: Oriel for Windows can accept ASCII files of up to 64K in size. However, Notepad can only work with ASCII files that are less than 50K. If you have a program that is larger than 50K, you'll have to use another editor besides Notepad to work with it.

MEMORY REQUIREMENTS

The Oriel for Windows executable file is quite small, only about 18K. However, the amount of memory that Oriel occupies in your system depends on the type of video display you have. Oriel for Windows uses a full-screen bitmap to store the contents of the screen so that it can restore the window when it needs to. Therefore, you can use the following equation to compute the amount of memory Oriel will require in your system:

$$\text{Amount of memory} = \text{Size of ORIEL.EXE (18K)} + \text{Size of a Full-Screen Bitmap (.BMP) File}$$

For example, on a standard VGA system, the size of a full-screen bitmap is approximately 150K. Therefore, the amount of memory that Oriel occupies is approximately 168K (18K + 150K). By contrast, on an EGA system the size of a full-screen bitmap is approximately 112K. Therefore, the amount of memory Oriel consumes on this type of system is about 130K (18K + 112K).

BASIC RULES AND SYNTAX

The basic rules and syntax for Oriel for Windows are quite simple. There are four classes of identifiers (names) in Oriel: commands, variables, tokens, and labels. These four kinds of identifiers are not case sensitive and can be any length. The sections that follow describe additional rules for the four classes of identifiers.

Commands

Commands are the basic building blocks of an Oriel script. Most Oriel commands follow the syntax shown in Figure 14-3, where a keyword is followed by parameters enclosed

Keyword(parameter1,parameter2,...,parametern)




Figure 14-3 The most common command syntax

in parentheses. The *keyword* names the action the command is to perform and can occur anywhere on a line. *Parameters* (also called arguments) provide the information necessary to execute the command and are separated by commas. Parameters can be integers, text enclosed in double quotes, variables, and tokens.

Other commands, such as If, follow the looser syntax shown in Figure 14-4. With these commands, the only rule is that the command elements must be separated by at least one space. You'll find the syntax for all Oriel commands in the command reference that follows and in the quick reference at the end of the chapter.

Variables

Oriel for Windows lets you create integer variables. To create an integer variable, all you have to do is use its name.

Variable names can be any length and can be upper or lower case. A variable name can use any of the characters A-Z, a-z, 0-9, or _ (underscore). However, it cannot start with a number. For example, the following are all valid variable names:

Mousex _012 Foxtrot y1 NEXT_LINE0

When you create a variable, Oriel automatically initializes it to zero. You can also initialize variables yourself using the Set command. For example, the following command initializes the variable named Counter to 6:

```
Set Counter=6
```

In many situations, you will not need to initialize a variable to a value. For example, some commands will set a variable for you, as in the following command, which lets you get mouse input:

```
SetMouse(1,1,10,10,Mouse_hit,x,y)
```

If <condition> Then <commands>




Figure 14-4 The syntax of the If command

In this command, the x and y variables are set automatically by Oriel; they indicate the point in the window where the mouse pointer was sitting when the user clicked the mouse button.

You can also perform simple mathematical calculations using integers and store the result in a variable. For example, the following command sets the variable `Mouse_x2` to the value in `Mouse_x1` multiplied by three:

```
Set Mouse_x2=Mouse_x1*3
```

See the Set command in the command reference for more details on variables.

Note: You can have a maximum of 500 variables in an Oriel program.

Tokens

Several Oriel for Windows commands require that you use tokens as parameters. A token is a special identifier that has been predefined by Oriel. For example, in the following command syntax, `PIXEL` and `METRIC` are tokens:

```
UseCoordinates (PIXEL/METRIC)
```

For this command, you *must* use either `PIXEL` or `METRIC` for the parameter, and you must spell the token correctly. No other parameter will be accepted.

When a command requires a token, the token appears in upper case in the command syntax. You'll find as you create your Oriel programs that it's a good idea to follow this same convention.

Labels

In Oriel for Windows, labels follow the same naming conventions as variables. For example, they can be any length and can be upper- or lower-case. Labels have the additional restrictions that they must be placed at the start of a line (in the first column of the line), and they must end with a `:` (colon). For example, here are some valid and invalid labels:

```
Next:                {A valid label}
Wait_for_input:      {An invalid label because it isn't
                      located at the start of the line}
```

Note: You can have up to 500 labels in an Oriel program.

Comments

All characters between `{` and `}` are treated as comments by Oriel for Windows. You can place comments anywhere in an Oriel text file. You can also nest comments.

For example, the following program draws the pyramid bitmap (`PYRAMID.BMP`) located in the `\WINDOWS` directory in a continuous line across the screen. The program is generously commented to make it easier to read.


```

{-----PYRAMID.TXT-----}
This program draws the pyramid bitmap across the screen a set
number of pixels apart.
-----}
{Initialize}
    UseCoordinates(PIXEL)           {Use pixels, not millimeters}
    Set x=30                        {Starting x-coordinate}
    Set y=20                        {Starting y-coordinate}
    Set Step=31                     {Step by 31 pixels at a time}

{Maximize the window}
    SetWindow(MAXIMIZE)

{Put up the pyramid bitmap for the first time}
    DrawBitmap(x,y,"C:\WINDOWS\PYRAMID.BMP")
    WaitInput(1000)                {Pause 1 second}

{Loop to draw the pyramid across the screen}
Next:  WaitInput(0)
        DrawBitmap(x,y,"C:\WINDOWS\PYRAMID.BMP")
        Set x=x+Step
        If x<600 Then Goto Next

{Leave the finished window up until the user kills it}
    WaitInput()

```

White Space

White space is a general term for the elements that Oriel ignores in a script. Oriel for Windows treats as white space all blanks, tab characters, carriage returns, line feed characters, split vertical bars (|), and comments. White space is ignored at any point in an Oriel script.

STARTING A PROGRAM

To start an Oriel program from the command line, you must provide the name of the Oriel executable file (ORIEL.EXE) followed by the name of the script file. For example, if Oriel for Windows is located in the C:\ORIEL directory and your script file is named SCRIPT.TXT and is located in C:\ORIEL\WORK, you would use the following command line:

```
C:\ORIEL\ORIEL C:\ORIEL\WORK\SCRIPT.TXT
```

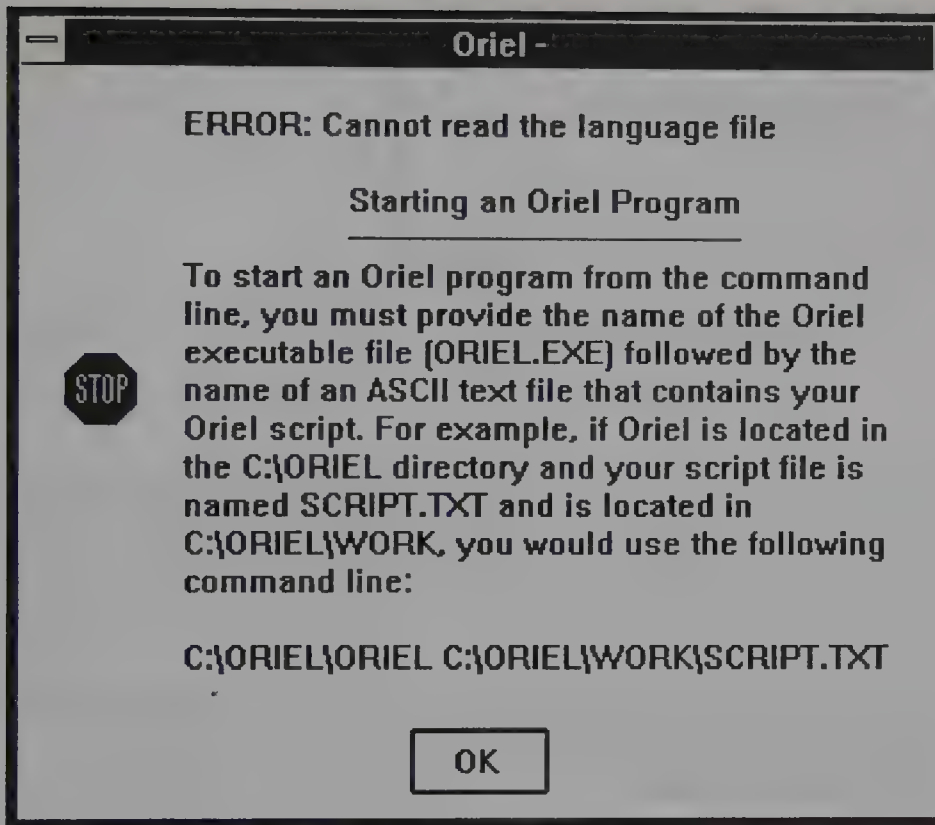


Figure 14-5 When Oriel for Windows cannot find your script file

If Oriel for Windows cannot find the script file, you'll see the message shown in Figure 14-5.

There are several ways you can simplify the Oriel command line. For example, if you have placed the Oriel directory in your DOS PATH statement (and you should), you can simplify the previous command line to read as follows:

```
ORIEL C:\ORIEL\WORK\SCRIPT.TXT
```

Another advantage of placing the Oriel directory in your DOS PATH statement is that you can invoke Oriel from another directory and have it look in that directory for the script file and any files that the script file might load, like bitmap (.BMP) files. Here's a variation of the example command line that uses this technique:

```
C:\ORIEL\WORK\ORIEL SCRIPT.TXT
```

Note: If you use this type of command line when creating an icon, Windows will issue a warning saying that the specified path is invalid. You can safely ignore this warning and accept the command line anyway. (See Chapter 2, "Program Manager Techniques," for more on this.)

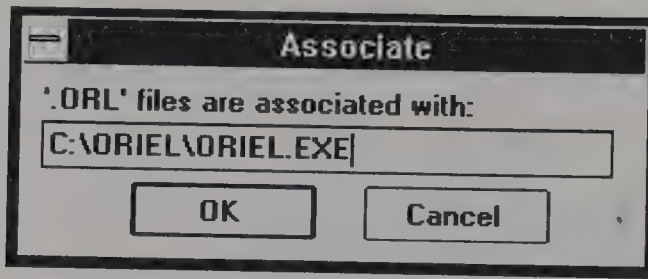


Figure 14-6 The File Associate dialog box

Yet another way to simplify the Oriel command line is to use File Manager to set up an association. For example, suppose you name all your Oriel script files using a unique extension like .ORL. You can create an association using the following steps:

1. Start File Manager.
2. Highlight a filename with an .ORL extension and select Associate from the File menu. Windows displays a dialog box like the one in Figure 14-6.
3. Type the path and name of the Oriel program file and select OK.

File Manager then updates your WIN.INI file with the association. From this point on, all you need to provide on the command line is the name of the Oriel script file, for example, SCRIPT.ORL. (If you use the Windows 3 Power Tools Setup program to install Oriel, a file association for “.ORL” is automatically set up.)

To create an icon for an Oriel program, you use the File New command from Program Manager. Chapter 2 gives a complete description of how to create icons.

Tip: *When script files don't require path information*

If a script file is located in the same directory as Oriel, you don't need to include path information for it.

STOPPING A PROGRAM

To stop an Oriel program at any point, press CTRL+BREAK. When you do so, Oriel for Windows displays the message box shown in Figure 14-7. To end the program, select Yes. To have the program resume where it left off, select No.

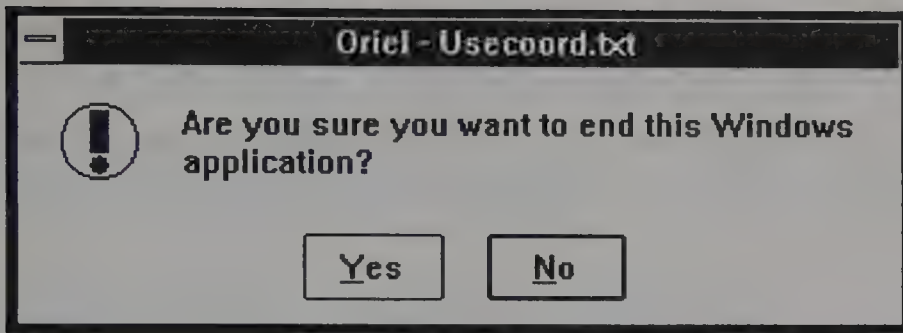


Figure 14-7 Stopping an Oriel for Windows program

THE ORIEL COORDINATE SYSTEM

In Oriel for Windows, the upper-left corner of a window is the origin, or point (0,0). Each millimeter to the right represents one unit along the positive x -axis. Each millimeter down represents one unit along the positive y -axis. Figure 14-8 shows the Oriel coordinate system.

You can modify the coordinate system to use pixels instead of millimeters (see the UseCoordinates command). Pixel coordinates are more accurate, but they are also device dependent. In other words, when you use pixel coordinates, an Oriel program you create for an EGA display will appear smaller when you run it on a system with a VGA display because there are more pixels per square inch on a VGA.

SELECTING DRAWING TOOLS

Oriel for Windows lets you use a variety of tools to draw within a window. It lets you set up pens to draw lines, brushes to fill interiors, and fonts to write text. To create tools for

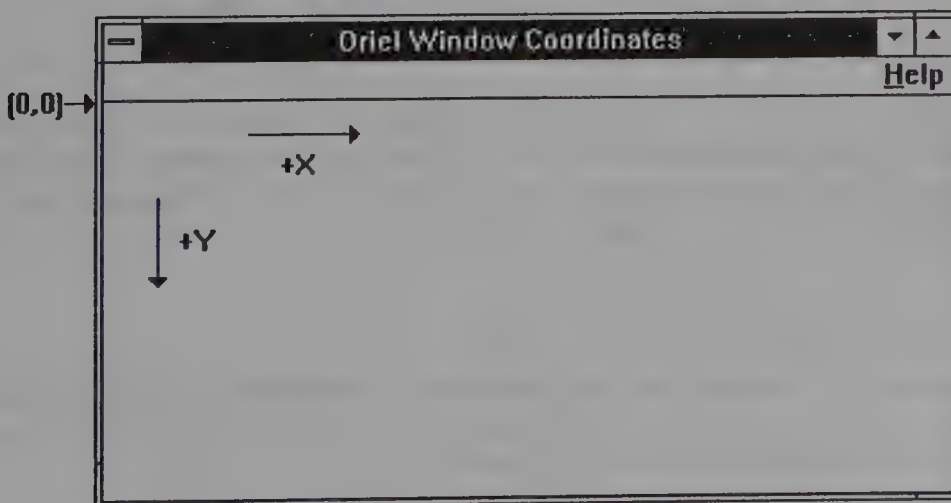


Figure 14-8 The Oriel coordinate system

drawing geometrical shapes, you use commands such as `UsePen` and `UseBrush`. To create fonts for writing text, you use the `UseFont` command.

Note: If you do not establish drawing tools before drawing in a window, Oriel uses these default drawing tools: a black pen, a white brush, and the System font.

Using Pens

The `UsePen` command lets you assign a pen for drawing lines and borders. Pens can be solid, dashed, dotted, and more. For example, the following example creates a solid black pen, two pixels wide:

```
Set Width=2  
UsePen(SOLID,Width,0,0,0)
```

The `Width` argument controls the width of the pen in pixels. The three arguments following the `Width` argument specify the color of the pen. They control the intensity of the colors red, green, and blue, respectively. In this example, all the colors have 0 intensity, so the pen will be black. Conversely, the following line would create a white pen:

```
UsePen(SOLID,Width,255,255,255)
```

The pen you specify with `UsePen` will be used in all subsequent drawing operations, or until you use `UsePen` again to change the pen.

Note: The default pen is solid, black, and has a width of 1 pixel. If you use a command that draws a shape, but you haven't yet set up a pen with the `UsePen` command, Oriel for Windows uses the default pen.

Using Brushes

The `UseBrush` command lets you establish brushes for drawing and filling areas in rectangles, ellipses, pies, and the like. You can create brushes that are solid or hatched, have diagonal lines, horizontal lines, vertical lines, and more. For example, here's the command to create a solid blue brush:

```
UseBrush(SOLID,0,255,0)
```

As you might have guessed, the last three arguments control the color of the brush. The brush you specify with `UseBrush` will be used in all subsequent drawing operations, or until you use `UseBrush` again to change the brush.

Note: The default brush is solid white.

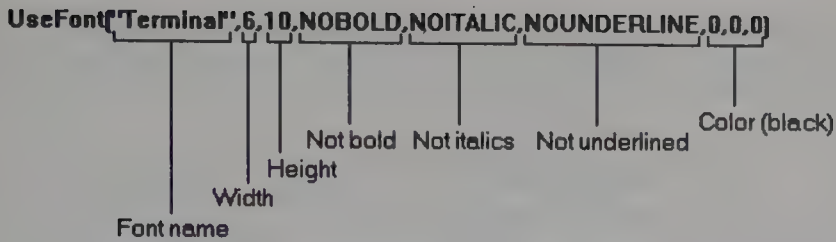


Figure 14-9 Command to select the Terminal font

Using Fonts

You establish a font in Oriel for Windows by using the `UseFont` command and giving it a series of font attributes, including the font name, width, height, style (bold, italics, or underline), and the color. Oriel for Windows uses the font you've established the next time you draw text or numbers on the screen. For example, Figure 14-9 shows a command that selects the Terminal font using different font attributes.

Note: Oriel for Windows lets you use fonts that are installed by third-party font packages, such as the Adobe Type Manager or Bitstream's Facelift. (See Chapter 4, "Of Fonts and Printing," for a description of these fonts.)

Note: By default, Oriel for Windows uses a black System font.

DRAWING AND WRITING

Oriel for Windows provides a variety of commands for drawing and writing output to the screen. The following sections give a brief overview of these commands. For more detailed descriptions of these commands, see the command reference.

Drawing Text

To draw text in a window, you use the `DrawText` command. For example, the following command displays the text "The ABC Company Shell" using the current font. The starting position of the text is at the point (20,10).

```
DrawText(20,10,"The ABC Company Shell")
```

If you want to display a number in a window, you can use the `DrawNumber` command. For example, the following command displays the number 200 starting at the point (50,60):

```
DrawNumber(50,60,200)
```

`DrawNumber` is also convenient for displaying the contents of variables in a window.

<i>Command</i>	<i>Purpose</i>
<code>DrawArc</code>	Draws an arc.
<code>DrawChord</code>	Draws a chord.
<code>DrawEllipse</code>	Draws an ellipse (or circle).
<code>DrawFlood</code>	Floods an area with color using the current brush.
<code>DrawLine</code>	Draws a line.
<code>DrawPie</code>	Draws a pie wedge.
<code>DrawRectangle</code>	Draws a rectangle.
<code>DrawRoundRectangle</code>	Draws a rectangle with rounded edges.

Table 14-1 Commands for Drawing Lines and Shapes

Drawing Lines and Shapes

Oriel provides several commands for drawing lines and shapes, as shown in Table 14-1. All of these commands use the current pen to draw borders and the current brush to fill interiors.

The following example shows how to use the `DrawLine` command to draw a line from the point (10,80) to the point (100,20):

```
DrawLine(10,80,100,20)
```

To draw a rectangle, you use the `DrawRectangle` command. The following command draws a rectangle that has its upper-left and lower-right corners at the points (15,25) and (75,110), respectively:

```
DrawRectangle(15,25,75,110)
```

This function uses the current pen to draw the border of the rectangle and the current brush to fill the interior.

The `DrawEllipse` command lets you draw a circle or an ellipse. The following example draws an ellipse that is bounded by the rectangle specified by the points (100,30) and (250,90):

```
DrawEllipse(100,30,250,90)
```

As with the `DrawRectangle` command, the `DrawEllipse` command uses the current pen to draw the border of the rectangle and the current brush to fill the interior.

Drawing Bitmaps

Oriel has two commands for placing the contents of bitmap (.BMP) files on the screen: `DrawBitmap` and `DrawSizedBitmap`. The `DrawBitmap` command is the simpler of the

two. It lets you locate a bitmap starting at a specified point in a window. For example, the following program places the party bitmap (PARTY.BMP) starting at the point (10,10):

```
DrawBitmap(10,10,"C:\WINDOWS\PARTY.BMP")
WaitInput()
```

The DrawSizedBitmap command lets you stretch or compress a bitmap to fit within a specified rectangle. You indicate the upper-left corner of the rectangle using the first two parameters and the lower-right corner using the second two. For example, the following program places the party bitmap within the smaller rectangle specified by the points (10,10) and (50,60):

```
DrawSizedBitmap(10,10,50,60,"C:\WINDOWS\PARTY.BMP")
WaitInput()
```

You can also use the DrawSizedBitmap command to invert a bitmap as you place it on the screen (see the DrawSizedBitmap command for more details).

FLOW OF CONTROL COMMANDS

Oriel for Windows has three commands to control the flow of programs: If, Goto, and Gosub. The If command lets you make a decision when there are two alternative outcomes. It tests the value of a condition, and if that condition is true, the program continues executing commands on the same line following the Then. However, if the condition is false, the program begins executing commands on the next line following the If.

For example, the following If command tests the value of the variable Green to see if it is greater than 255. If it is, Oriel executes the Goto command on the same line. Otherwise, it executes the WaitInput() command on the next line.

```
If Green>255 Then Goto Exit
WaitInput()
Exit:
```

The Goto command transfers control unconditionally to a label. In the previous example, the Goto command causes the program to branch to the label Exit.

The Gosub command lets you execute a block of code as a subroutine. When the subroutine is completed, Oriel executes the next command following the Gosub.

MESSAGE BOXES

The MessageBox command lets you create your own customized message boxes. For example, the following command creates a message box with OK and Cancel buttons (the second parameter, 1, causes the OK button to be highlighted) and a question-mark icon. In addition, the message box displays the text "Do you want to exit?" and uses the caption "Exit box."


```

MessageBox (OKCANCEL, 1, QUESTION,
            "Do you want to exit?", "Exit box", Button)

```

The button you select is returned in the Button variable.

CONTROLLING THE WINDOW SIZE

The SetWindow command lets you maximize, minimize, or restore the Oriel window. For example, the following commands maximize the Oriel window, pause the program for 2 seconds, then restore the window to its original size:

```

SetWindow (MAXIMIZE)
WaitInput (2000)
SetWindow (RESTORE)
WaitInput ()

```

PAUSING A PROGRAM

In Oriel, you can pause a program a specified number of seconds, or you can pause it indefinitely. Both require the WaitInput command.

Pausing a Specified Number of Seconds

By using the WaitInput command with an argument, you can pause an Oriel program a specified number of seconds. For example, the following program displays the message "Waiting...". Next, it pauses the program for three seconds then erases the window's contents.

```

DrawText (10, 10, "Waiting...")
WaitInput (3000)    {Pause the program for 3 seconds}
DrawBackground     {Erase the window}
WaitInput ()

```

As you may have guessed, the parameter for the WaitInput command is in milliseconds.

Pausing Indefinitely

By using the WaitInput command without an argument, you can pause a program indefinitely. In general, you should use WaitInput() whenever you are not performing any work in your Oriel window. This makes more system resources available to other Windows programs.

If you fail to use WaitInput() at some point in your program, one of two things will happen: either the window will disappear immediately after executing your script or, if you are stuck in a continuous loop, the hour-glass icon will always be present, and you may not be able to switch away.

The `WaitInput()` command is also important for building your own custom menus and for getting keyboard or mouse input. The next two sections will give you more information.

BUILDING MENUS

To build your own custom menus, you must use the `SetMenu` command to define a menu template. Then, when the program is pausing for input (a `WaitInput()` command is in effect) and the user selects a menu item, the program branches to the label associated with that menu item, as defined in the template.

For example the following program creates a simple menu with only two menu items: `Write` and `Exit`. If you select the `Write` option, the program branches to the `Run_write` label where `Windows Write` is launched. If you select the `Exit` option, the program ends.

```
{Define the menu template}
  SetMenu("Write",Run_write,
          ENDPOPUP,
          "Exit",Leave,
          ENDPOPUP)
```

```
Wait_for_input:
  WaitInput()
```

```
Run_write:
  Run("WRITE.EXE")
  Goto Wait_for_input
```

```
Leave:
  End
```

GETTING MOUSE INPUT

To get mouse input in an Oriel program, you use the `SetMouse` command and define rectangular regions on the screen as mouse hit-testing regions. Then, when the program is pausing for input and you click the left mouse button within a mouse hit-testing region, the program branches to the label associated with that region, as defined by the `SetMouse` command.

For example, suppose you want to modify the previous program to place some "buttons" on the screen and provide mouse support for when you click on a button. You could change the program as follows:

```
{Define the menu template}
  SetMenu("Write",Run_write,
          ENDPOPUP,
```

```

        "Exit",Leave,
        ENDPOPUP)

{Draw buttons with text}
    DrawRectangle(10,10,30,20)
    DrawText(14,12,"Write")
    DrawRectangle(10,25,30,35)
    DrawText(15,27,"Exit")

{Set up the mouse}
    SetMouse(10,10,30,20,Run_write,x,y,
            10,25,30,35,Leave,x,y)

Wait_for_input:
    WaitInput()

Run_write:
    Run("WRITE.EXE")
    Goto Wait_for_input

Leave:
    End

```

Figure 14-10 shows how the window appears.

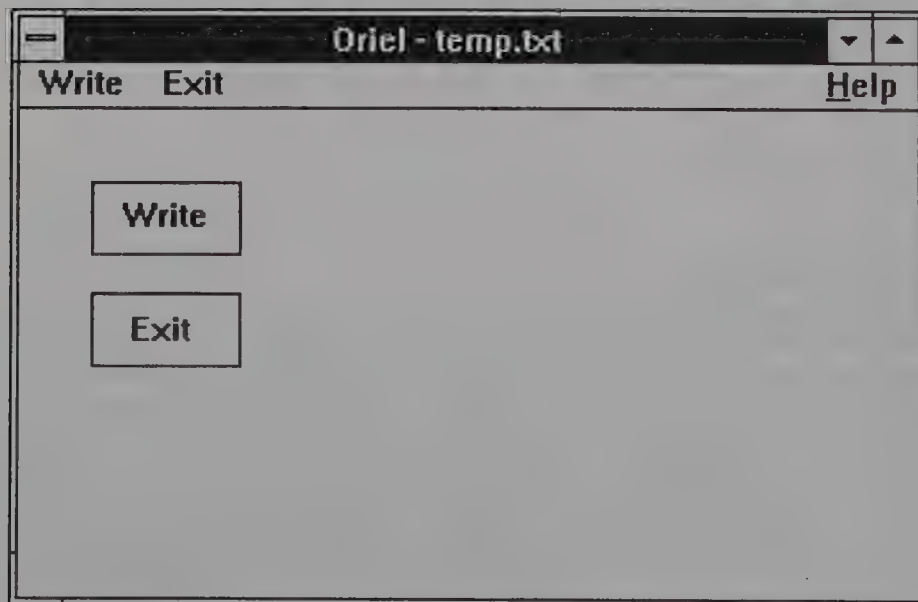


Figure 14-10 A sample program with a menu and “buttons”

In this example, two hit-testing regions are defined by the `SetMouse` command, both corresponding to the rectangles that were drawn with the `DrawRectangle` commands. The first hit-testing region is defined by the upper-left corner (10,10) and the lower-right corner (30,20)—the same area as the `Write` button. If you click the mouse within this region, the program branches to the `Run_write` label. In the same way, if you click the mouse button within the region defined by the points (10,25) and (30,35)—the same area as the `Exit` button—the program branches to `Leave`.

GETTING KEYBOARD INPUT

To read single keystrokes in an Oriol program, you use the `SetKeyboard` command to define the keys that you will accept. Then, when the program is pausing for input and you press a specified key, the program branches to the label associated with that key.

Building on the previous example, suppose you want to add keyboard support so that when you press `w` or `W` the program runs `Write` and when you press `e` or `E`, the program exits. Here's how you would modify the program:

```
{Define the menu template}
    SetMenu("Write",Run_write,
            ENDPOPUP,
            "Exit",Leave,
            ENDPOPUP)

{Draw buttons with text}
    DrawRectangle(10,10,30,20)
    DrawText(14,12,"Write")
    DrawRectangle(10,25,30,35)
    DrawText(15,27,"Exit")

{Set up the mouse}
    SetMouse(10,10,30,20,Run_write,x,y,
            10,25,30,35,Leave,x,y)

{Put up a message regarding keyboard support}
    DrawText(10,40,"Press W for Write or E to exit")

{Set up the keyboard}
    SetKeyboard("W",Run_write,
               "w",Run_write,
               "E",Leave,
               "e",Leave)
```

```

Wait_for_input:
    WaitInput()

Run_write:
    Run("WRITE.EXE")
    Goto Wait_for_input

Leave:
    End

```

Note: By using virtual keys, you can read keys that are not on the typewriter portion of the keyboard (see SetKeyboard for more details).

RUNNING OTHER PROGRAMS

By using the Run command, you can execute Windows and non-Windows applications from within an Oriel program. For example, the following program starts Notepad and then launches a copy of COMMAND.COM:

```

Run("NOTEPAD.EXE")
Run("C:\COMMAND.COM")
WaitInput()

```

Each program you start takes on a life of its own independently of the Oriel script that invoked it. This means that, under normal circumstances, Oriel for Windows does not pause after executing a Run command, but continues with the next command in the script. For example, in the previous program, Oriel for Windows starts Notepad then immediately starts COMMAND.COM.

If you want Oriel to pause after starting Notepad, you would use the following script:

```

SetWaitMode(FOCUS)
Run("NOTEPAD.EXE")
WaitInput(1)
Run("C:\COMMAND.COM")
WaitInput()

```

By placing the SetWaitMode(FOCUS) command before and WaitInput(1) after Run("NOTEPAD.EXE"), you can have Oriel pause until it gets the focus back. That is, only when you close Notepad or switch back to your Oriel program does it execute the next Run command to launch a copy of COMMAND.COM. (See the WaitInput and SetWaitMode commands later for more information.)

COMMAND REFERENCE

The following is a list of commands that are supported by Oriel for Windows. The quick reference at the end of the chapter is handy if all you need to know is the syntax for an Oriel command.

Beep

This command sounds the bell.

Syntax: Beep

Example: This program puts up a message box with Yes, No, and Cancel buttons. It then beeps once if you select Yes, twice for No, and three times for Cancel.

```
{Put up a message box}
  MessageBox(YESNOCANCEL,1,QUESTION,
             "Did you vote?","Question",Button)
{Test for which button was selected}
  If Button=1 Then Goto Beep1
  If Button=2 Then Goto Beep2
  {Else} Beep
  WaitInput(500){Pause for 1/2 second}
Beep2:
  Beep
  WaitInput(500)
Beep1:
  Beep
```

DrawArc

This command draws an elliptical arc using the current pen. To draw the arc, you use the points (x1,y1) and (x2,y2) to define a rectangle that bounds the ellipse containing the arc, as shown in Figure 14-11. You then use the parameters (x3,y3) to specify the point on the ellipse where the arc starts, and the parameters (x4,y4) to specify where it ends. Note that when the GDI sweeps the arc, it begins at (x3,y3) and moves in a counterclockwise direction towards (x4,y4).

Syntax: DrawArc(x1,y1,x2,y2,x3,y3,x4,y4)

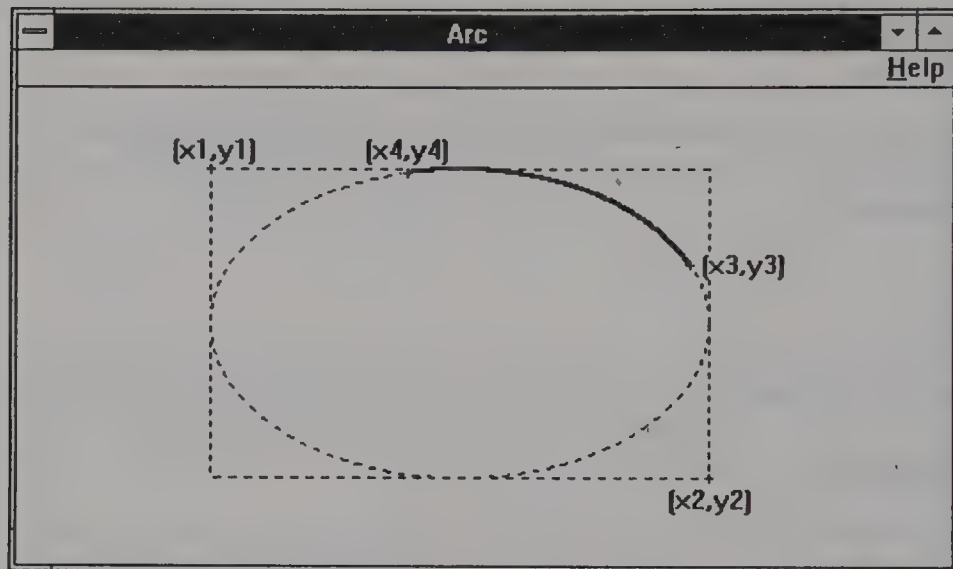


Figure 14-11 The DrawArc coordinates

Parameters:

$x1, y1$	The upper-left corner of the rectangle bounding the ellipse containing the arc.
$x2, y2$	The lower-right corner of the rectangle bounding the ellipse containing the arc.
$x3, y3$	The starting point of the arc on the ellipse.
$x4, y4$	The ending point of the arc on the ellipse.

Remark: The points $(x3, y3)$ and $(x4, y4)$ do not have to lie precisely on the ellipse. If they do not, however, the GDI uses points on the ellipse that are the shortest distance from $(x3, y3)$ and $(x4, y4)$.

Examples: This example sets the coordinate system to pixel, then draws an arc within the rectangle defined by the points (30,20) and (200,180). It sweeps the arc starting from the point (200,20) and moving in a counterclockwise direction to the point (30,20).

```
UseCoordinates (PIXEL)
DrawArc (30, 20, 200, 180, 200, 20, 30, 20)
WaitInput ()
```

The next program draws an arc by asking you to click on each of the four points needed to define an arc. The first point it asks for is the upper-left corner of the rectangle bounding the ellipse containing the arc, and the second is the lower-right corner of that same rectangle. The program then draws a temporary

ellipse using the points you've defined so that you can select the third and fourth points needed for the arc—its starting and ending points. After you've clicked on these points on the ellipse, the program clears the screen and draws the arc using the DrawArc command. It then loops back up for you to specify another arc.

```
{Set up the environment}
  SetWindow(MAXIMIZE)
  UseCoordinates(PIXEL)
  UseFont("Terminal",10,10,NOBOLD,NOITALIC,NOUNDERLINE,0,0,0)

Arc1:
  UsePen(DOT,1,0,0,0)      {Use a dotted pen for the temporary ellipse}
  SetMouse(0,0,700,600,Arc2,x1,y1)
  DrawText(10,300,
    "Click on the upper-left corner of the rectangle bounding the arc ")
  Goto Get_Input

Arc2:
  SetMouse(0,0,700,600,Arc3,x2,y2)
  DrawText(10,300,
    "Click on the lower-right corner of the rectangle bounding the arc")
  Goto Get_Input

Arc3:
  DrawEllipse(x1,y1,x2,y2)      {Draw a temporary ellipse}
  SetMouse(0,0,700,600,Arc4,x3,y3)
  DrawText(10,300,
    "Click on the arc's starting point                                ")
  Goto Get_Input

Arc4:
  SetMouse(0,0,700,600,Arc_End,x4,y4)
  DrawText(10,300,
    "Click on the arc's ending point                                ")
  Goto Get_Input

Arc_End:
  DrawBackground              {Clear the temporary ellipse}
  UsePen(SOLID,1,0,0,0)      {Use a solid black pen}
  DrawArc(x1,y1,x2,y2,x3,y3,x4,y4)
  Goto Arc1

Get_Input:
  WaitInput()
```

Related Commands: UsePen

DrawBackground

Draws a window's background using the current background color. If the window has any contents, they are overwritten in the process.

Syntax: DrawBackground

Remark: You control the current background color using the UseBackground command. (The current background mode, TRANSPARENT or OPAQUE, has no effect on the DrawBackground command's behavior.)

Example: This example sets the background color to green using the UseBackground command and then draws the background. Next, it draws some text in the window, waits for 2 seconds, and erases the text by issuing the DrawBackground command again.

```
UseBackground(TRANSPARENT,0,255,0)
DrawBackground
DrawText(10,10,"The effect of text")
WaitInput(2000)
DrawBackground
WaitInput()
```

Related Commands: UseBackground

DrawBitmap

This command places the contents of a bitmap (.BMP) file at a specified location on the screen.

Syntax: DrawBitmap(*x*,*y*,"*Filename*")

Parameters:

<i>x</i>	The x-coordinate of the upper-left corner of the bitmap.
<i>y</i>	The y-coordinate of the upper-left corner of the bitmap.
" <i>Filename</i> "	The name of the bitmap file, enclosed in quotation marks. Be sure to include the path if the bitmap file is not in the current directory.

Remark: You can use Paintbrush to get the size of a bitmap in pixels. To do so, begin by reading the bitmap into Paintbrush and choose Cursor Position from the View menu. When you move the cursor to the bottom right corner of the drawing area, Paintbrush will show the size of the bitmap as an x,y coordinate.

Example: This example reads the chess bitmap file (CHESS.BMP) on the \WINDOWS directory and draws its contents beginning in the upper-left corner of the window.

```
DrawBitmap(0,0,"C:\WINDOWS\CHESS.BMP")
WaitInput()
```

This next example places two bitmaps on the screen: the ribbons bitmap (RIBBONS.BMP) starting at (5,5), and the pyramid bitmap (PYRAMID.BMP) starting at (130,30):

```
DrawBitmap(5,5,"C:\WINDOWS\RIBBONS.BMP")
DrawBitmap(130,30,"C:\WINDOWS\PYRAMID.BMP")
WaitInput()
```

Related Commands: DrawSizedBitmap

DrawChord

This command draws a chord—a region bounded by the intersection of a line and an ellipse. To draw a chord, you use the points (x1,y1) and (x2,y2) to define a rectangle that bounds the ellipse that is part of the chord, as shown in Figure 14-12. You then specify the line that intersects the ellipse using the points (x3,y3) and (x4,y4). The command draws the chord's border using the current pen and fills its interior using the current brush.

Syntax: DrawChord(x1,y1,x2,y2,x3,y3,x4,y4)

Parameters:

x1,y1	The upper-left corner of the rectangle bounding the ellipse that is part of the chord.
x2,y2	The lower-right corner of the rectangle bounding the ellipse that is part of the chord.
x3,y3	A point on the line that intersects the ellipse.
x4,y4	A second point on the line that intersects the ellipse.

Remark: The points (x3,y3) and (x4,y4) do not have to lie precisely on the ellipse. If they do not, however, the GDI uses points on the ellipse that are the shortest distance from (x3,y3) and (x4,y4).

Examples: This example draws a chord by using the points (40,30) and (210,190) to define the rectangle that bounds the ellipse that is part of the chord. The points used for the line that intersects the ellipse are (210,30) and (40,30). The chord is drawn with the default black pen and a solid aqua brush.

```
UseCoordinates(PIXEL)
UseBrush(SOLID,0,255,255)
DrawChord(40,30,210,190,210,30,40,30)
WaitInput()
```

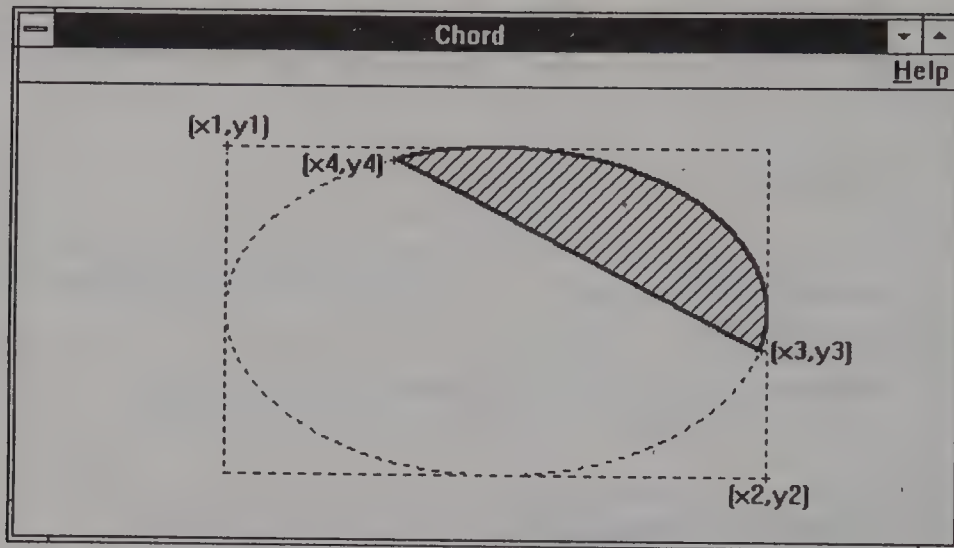



Figure 14-12 The DrawChord coordinates

This next example is a variation of one shown for the DrawArc command. It draws a chord by prompting you to click on the four points needed to define a chord. The first point is the upper-left corner of the rectangle bounding the ellipse that is part of the chord; the second is the lower-right corner of that same rectangle; the third is a point on the line bounding the chord; and the fourth is another point on that same line. After you've clicked on all four points, the program draws the chord using the DrawChord command, then loops back up for you to specify another chord.

```
{Set up the environment}
    SetWindow(MAXIMIZE)
    UseCoordinates(PIXEL)
    UseFont("Terminal",10,10,NOBOLD,NOITALIC,NOUNDERLINE,0,0,0)

Chord1:
    UsePen(DOT,1,0,0,0)      {Use dotted pen for temporary ellipse}
    UseBrush(NULL,0,0,0)    {Use hollow brush for temporary ellipse}
    SetMouse(0,0,700,600,Chord2,x1,y1)
    DrawText(10,300,
        "Click on upper-left corner of rectangle bounding chord ")
    Goto Get_Input

Chord2:
    SetMouse(0,0,700,600,Chord3,x2,y2)
    DrawText(10,300,
        "Click on lower-right corner of rectangle bounding chord")
    Goto Get_Input

Chord3:
    DrawEllipse(x1,y1,x2,y2) {Draw temporary ellipse}
```

```

SetMouse(0,0,700,600,Chord4,x3,y3)
DrawText(10,300,
    "Click on one point of line defining chord",
Goto Get_Input

Chord4:
    SetMouse(0,0,700,600,Chord_End,x4,y4)
    DrawText(10,300,
        "Click on second point of line defining chord",
    Goto Get_Input

Chord_End:
    DrawBackground                {Clear the temporary ellipse}
    UsePen(SOLID,1,0,0,0)          {Use a solid black pen}
    UseBrush(SOLID,255,0,0)        {Use a solid red brush}
    DrawChord(x1,y1,x2,y2,x3,y3,x4,y4)
    Goto Chord1

Get_Input:
    WaitInput()

```

Related Commands: UseBrush, UsePen

DrawEllipse

This command draws an ellipse (or circle). To specify the ellipse, you use the points (*x1,y1*) and (*x2,y2*) to define a rectangle that bounds the ellipse, as shown in Figure 14-13. The command uses the current pen to draw the border and the current brush to fill the interior.

Syntax: DrawEllipse(*x1,y1,x2,y2*)

Parameters:

<i>x1,y1</i>	The upper-left corner of the rectangle bounding the ellipse.
<i>x2,y2</i>	The lower-right corner of the rectangle bounding the ellipse.

Examples: The following example draws an ellipse that is bounded by the rectangle specified by the points (20,30) and (80,60). It draws the ellipse using a solid black pen (the default) and a solid yellow brush.

```

UseBrush(SOLID,255,255,0)
DrawEllipse(20,30,80,60)
WaitInput()

```

This second example prompts you to click on the points of the rectangle bounding an ellipse. It then draws the ellipse using the default pen and a solid green brush.

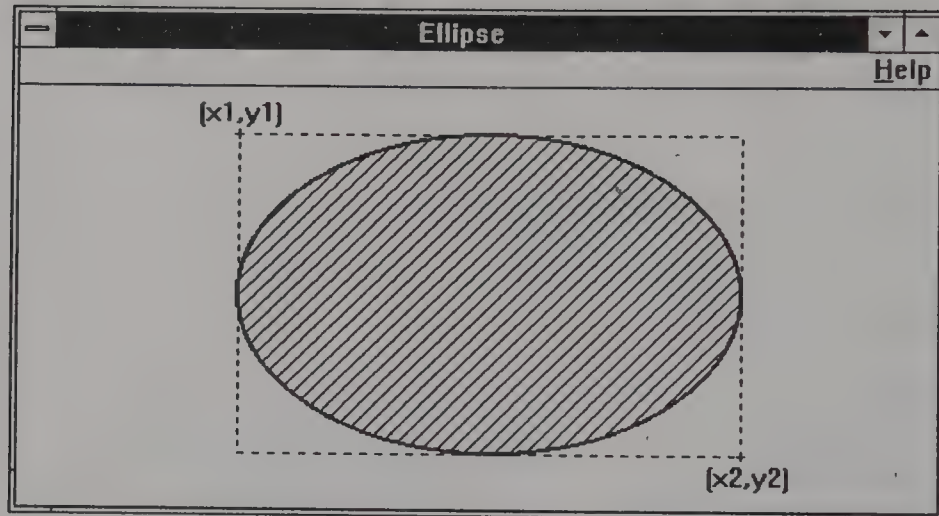


Figure 14-13 The DrawEllipse coordinates

```
{Set up the environment}
    SetWindow(MAXIMIZE)
    UseCoordinates(PIXEL)
    UseFont("Terminal",10,10,NOBOLD,NOITALIC,NOUNDERLINE,0,0,0)
    UseBrush(SOLID,0,255,0)    {Use a solid green brush}

Ellipse1:
    SetMouse(0,0,700,600,Ellipse2,x1,y1)
    DrawText(10,300,
        "Click on upper-left corner of rectangle bounding ellipse ")
    Goto Get_Input
Ellipse2:
    SetMouse(0,0,700,600,Ellipse3,x2,y2)
    DrawText(10,300,
        "Click on lower-right corner of rectangle bounding ellipse")
    Goto Get_Input
Ellipse3:
    DrawEllipse(x1,y1,x2,y2)
    Goto Ellipse1

Get_Input:
    WaitInput()
```

Related Commands: UsePen, UseBrush

DrawFlood

This command fills in any enclosed shape or area using the current brush. It begins at the point specified by the *x* and *y* parameters and fills in all directions until it reaches the color boundary specified by the *r*, *g*, *b* parameters.

Syntax: DrawFlood(*x*,*y*,*r*,*g*,*b*)

Parameters:

<i>x</i>	The x-coordinate of a point inside the area you want to fill.
<i>y</i>	The y-coordinate of a point inside the area you want to fill.
<i>r</i> , <i>g</i> , <i>b</i>	Specifies the color of the boundary up to which the command is to fill.

Example: This example draws a series of shapes on the screen using a blue pen and a hollow brush. It then sets the brush to solid red and asks you to click within an area to flood it with red. When you click on an area, the program branches to Flood_Mouse, where the DrawFlood command fills in the selected area using the current brush. (Notice that the *r*, *g*, *b* parameters are 0,0,255, causing the command to fill an area until it encounters a blue border.) The program stays in a loop, allowing you to flood as many areas as you like.

```
{Set up the mouse, pen, and brush}
    SetWindow(MAXIMIZE)
    SetMouse(0,0,700,600,Flood_Mouse,Flood_x,Flood_y)
    UsePen(SOLID,3,0,0,255) {Use a 3-pixel wide blue pen}
    UseBrush(NULL,0,0,0)    {Use hollow brush for random shapes}

{Fill the window with random shapes}
    DrawRectangle(5,10,200,150)
    DrawRectangle(55,31,82,50)
    DrawRectangle(105,41,120,140)
    DrawRectangle(50,50,100,150)
    DrawRectangle(105,50,120,133)
    DrawRectangle(5,120,180,125)
    DrawEllipse(30,30,140,140)
    DrawEllipse(10,30,150,100)
    DrawEllipse(25,10,50,100)
    DrawEllipse(33,51,123,143)
```

```

        DrawText(5,3,"Click within an area to flood it with red")
        UseBrush(SOLID,255,0,0)      {Use a red brush for flooding}
        UseCoordinates(PIXEL)        {More accurate than metric}

Flood_Wait:
        WaitInput()

Flood_Mouse:
        DrawFlood(Flood_x,Flood_y,0,0,255) {Flood till it meets blue}
        Goto Flood_Wait

```

Related Commands: UseBrush

DrawLine

This command draws a line using the current pen. The line begins at the point specified by (x1,y1) and extends up to, but does not include, the point specified by (x2,y2).

Syntax: DrawLine(x1,y1,x2,y2)

Parameters:

x1,y1	The coordinates of the first point on the line.
x2,y2	The coordinates of the ending point on the line.

Example: This program draws a series of five lines, each one using a different pen width. Figure 14-14 shows the results.

```

UsePen(SOLID,1,0,0,0)
DrawLine(10,10,40,10)

UsePen(SOLID,2,0,0,0)
DrawLine(10,15,40,15)

UsePen(SOLID,3,0,0,0)
DrawLine(10,20,40,20)

UsePen(SOLID,4,0,0,0)
DrawLine(10,25,40,25)

```

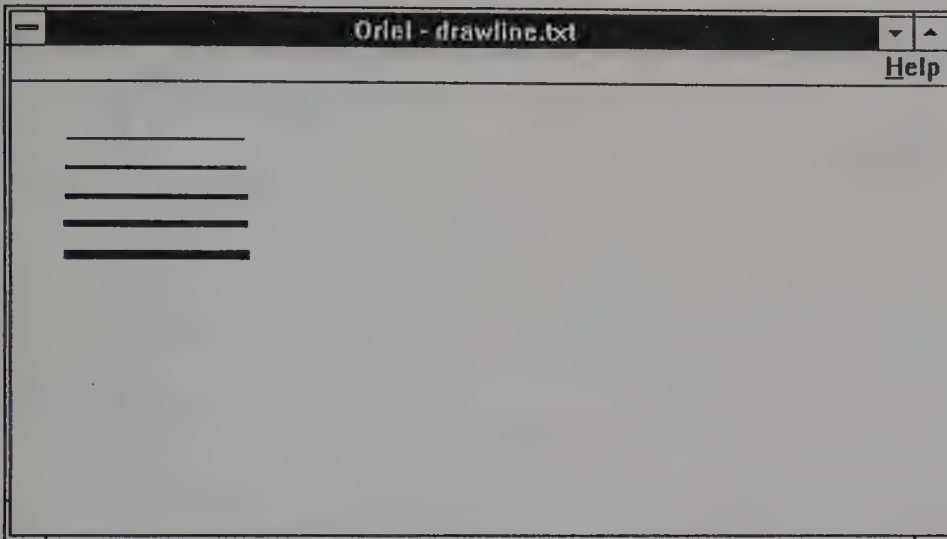



Figure 14-14 The effect of the DrawLine command with different pen widths

```
UsePen(SOLID,5,0,0,0)
DrawLine(10,30,40,30)

WaitInput()
```

See the UsePen command for another example of DrawLine.

Related Commands: UsePen

DrawNumber

This command displays an integer, n , using the current font. The starting position of the integer is given by the x and y parameters.

Syntax: DrawNumber(x, y, n)

Parameters:

x	Specifies the x-coordinate of the starting point of the integer.
y	Specifies the y-coordinate of the starting point of the integer.
n	The integer or the value of the integer variable you want to display.

Examples: This example changes Oriel's coordinate system from metric (the default) to pixel and then displays the integer 1000 starting 20 pixels to the right and 10 pixels below the upper-left corner of the window.

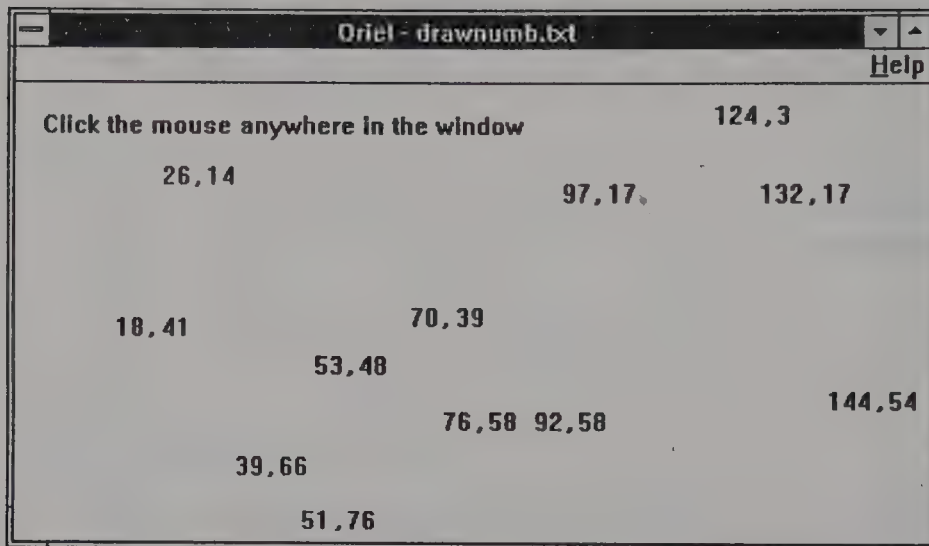


Figure 14-15 Using the DrawNumber command

```
UseCoordinates (PIXEL)
DrawNumber (20,10,1000)
WaitInput ()
```

This next example reads a mouse click and uses the DrawNumber command to display the coordinates of where the mouse pointer was when the click took place. Figure 14-15 shows an example of how the window appears after you have clicked the mouse several times.

```
{Set up the mouse hit testing area and where to branch on a click}
SetMouse(0,0,1000,1000,Draw_Coord,Mouse_x,Mouse_y)
DrawText(5,5,"Click the mouse anywhere in the window")

Wait_Mouse:
    WaitInput()

{Draw the x-coordinate}
Draw_Coord:
    DrawNumber(Mouse_x,Mouse_y,Mouse_x)

{Increase the x-coordinate by 3 millimeters for each digit in Mouse_x}
    If Mouse_x>=100 Then Set x=Mouse_x+9 | Goto Draw_Comma {3 digits}
    If Mouse_x>=10  Then Set x=Mouse_x+6 | Goto Draw_Comma {2 digits}
    {Else}                Set x=Mouse_x+3                {1 digit only}

{Draw the comma}
Draw_Comma:
    DrawText(x,Mouse_y,",")
```

```
{Increase x by 2 and draw the y-coordinate}
Set x=x+2
DrawNumber (x,Mouse_y,Mouse_y)
Goto Wait_Mouse
```

Related Commands: UseFont

DrawPie

This command lets you draw a pie wedge using the current pen. A pie wedge consists of an arc whose center and end points are connected by lines. To draw the arc, you use the points $(x1,y1)$ and $(x2,y2)$ to define a rectangle that bounds the ellipse containing the arc, as shown in Figure 14-16. You then use the parameters $(x3,y3)$ and $(x4,y4)$ to specify the starting and ending points of the arc. Note that when the GDI sweeps the arc, it begins at $(x3,y3)$ and moves in a counterclockwise direction toward $(x4,y4)$. The current brush is used to fill the resulting pie-shaped area.

Syntax: DrawPie($x1,y1,x2,y2,x3,y3,x4,y4$)

Parameters:

- | | |
|---------|--|
| $x1,y1$ | The upper-left corner of the rectangle bounding the ellipse containing the arc. |
| $x2,y2$ | The lower-right corner of the rectangle bounding the ellipse containing the arc. |

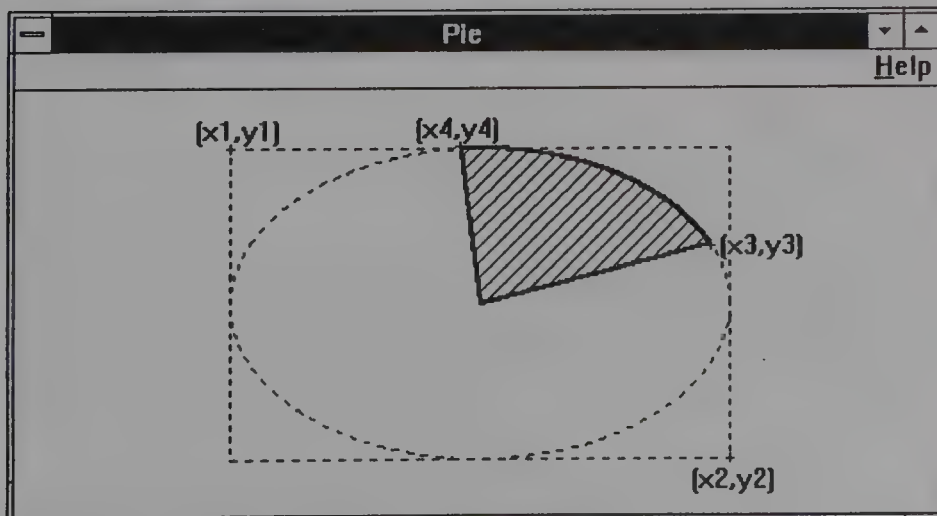


Figure 14-16 The DrawPie coordinates

x_3, y_3	The starting point of the arc. This point does not have to lie on the arc.
x_4, y_4	The ending point of the arc. This point does not have to lie on the arc.

Remark: The points (x_3, y_3) and (x_4, y_4) do not have to lie precisely on the ellipse. If they do not, however, the GDI uses points on the ellipse that are the shortest distance from (x_3, y_3) and (x_4, y_4) .

Examples: This example draws a pie wedge using the default black pen to draw the border and an aqua brush to fill the interior. The pie wedge is bounded by the rectangle specified by the points (110,30) and (160,80). The pie's arc starts at the point (160,30) and ends at the point (160,80).

```
UseBrush(SOLID, 0, 255, 255)
UseCoordinates(PIXEL)
DrawPie(110, 30, 160, 80, 160, 30, 160, 80)
WaitInput()
```

This next program draws a pie wedge based on your mouse clicks. The first two clicks define the rectangle bounding the arc, and the second two define the lines connecting the points of the arc with its center.

```
{Set up the environment}
  SetWindow(MAXIMIZE)
  UseCoordinates(PIXEL)
  UseFont("Terminal", 10, 10, NOBOLD, NOITALIC, NOUNDERLINE, 0, 0, 0)

Pie1:
  UsePen(DOT, 1, 0, 0, 0)      {Dotted pen for temporary ellipse}
  UseBrush(NULL, 0, 0, 0)     {Hollow brush for temporary ellipse}
  SetMouse(0, 0, 700, 600, Pie2, x1, y1)
  DrawText(10, 300,
    "Click on upper-left corner of rectangle bounding pie ")
  Goto Get_Input

Pie2:
  SetMouse(0, 0, 700, 600, Pie3, x2, y2)
  DrawText(10, 300,
    "Click on lower-right corner of rectangle bounding pie")
  Goto Get_Input

Pie3:
  DrawEllipse(x1, y1, x2, y2)
  SetMouse(0, 0, 700, 600, Pie4, x3, y3)
  DrawText(10, 300,
```

```

        "Click on one end of arc defining pie          ")
    Goto Get_Input

Pie4:
    SetMouse(0,0,700,600,Pie_End,x4,y4)
    DrawText(10,300,
        "Click on other end of arc defining pie      ")
    Goto Get_Input

Pie_End:
    DrawBackground
    UsePen(SOLID,1,0,0,0)
    UseBrush(SOLID,255,0,0)
    DrawPie(x1,y1,x2,y2,x3,y3,x4,y4)
    Goto Pie1

Get_Input:
    WaitInput()

```

Related Commands: UsePen, UseBrush

DrawRectangle

This command draws a rectangle using the current pen and fills its interior using the current brush. You specify the upper-left corner of the rectangle using *(x1,y1)* and the lower-right corner using *(x2,y2)*, as shown in Figure 14-17.

Syntax: DrawRectangle(*x1,y1,x2,y2*)

Parameters:

<i>x1,y1</i>	The upper-left corner of the rectangle.
<i>x2,y2</i>	The lower-right corner of the rectangle.

Examples: The following example draws a rectangle whose upper-left corner is located at (10,10) and lower-right corner is at (200,100). It uses a black pen that is three pixels wide and fills the interior using an orange brush.

```

UsePen(SOLID,3,0,0,0)
UseBrush(SOLID,255,128,0)
UseCoordinates(PIXEL)
DrawRectangle(10,10,200,100)
WaitInput()

```

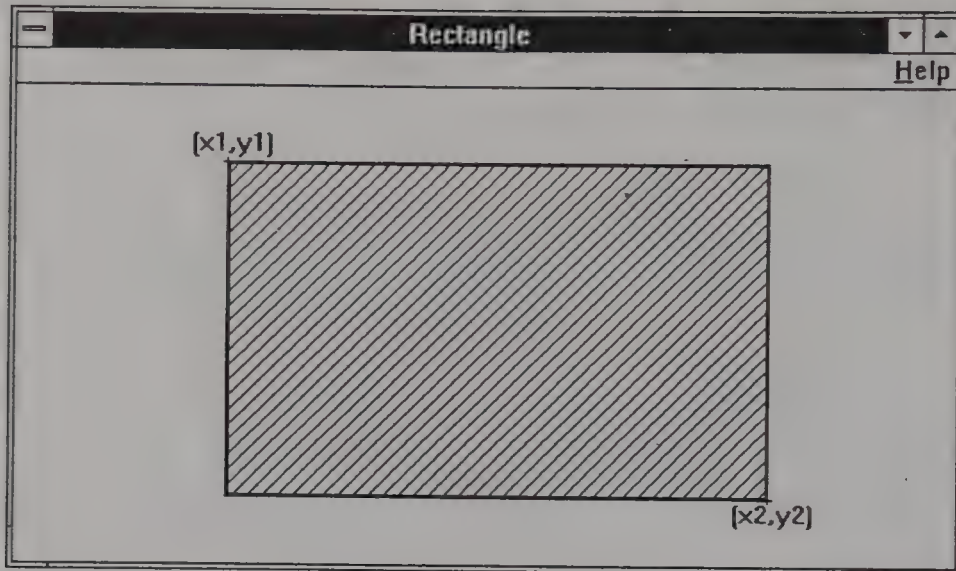



Figure 14-17 The DrawRectangle coordinates

This next example draws a rectangle based on two mouse clicks. The first mouse click defines the upper-left corner of the rectangle and the second one defines the lower-right. The rectangle is filled using a light cream-colored brush.

```
{Set up the environment}
  SetWindow(MAXIMIZE)
  UseCoordinates(PIXEL)
  UsePen(SOLID,3,0,0,0)
  UseBrush(SOLID,255,255,230)

Rect1:
  SetMouse(0,0,700,600,Rect2,x1,y1)
  DrawText(10,300,
    "Click on upper-left corner of the rectangle")
  Goto Get_Input
Rect2:
  SetMouse(0,0,700,600,Rect3,x2,y2)
  DrawText(10,300,
    "Click on lower-right corner of the rectangle")
  Goto Get_Input
Rect3:
  DrawRectangle(x1,y1,x2,y2)
  Goto Rect1

Get_Input:
  WaitInput()
```

Related Commands: UsePen, UseBrush

DrawRoundRectangle

This command draws a rectangle with rounded corners. It draws the border of the rectangle using the current pen and fills its interior using the current brush. You specify the upper-left corner of the rectangle using $(x1,y1)$ and the lower-right corner using $(x2,y2)$, as shown in Figure 14-18. You control the width and height of the ellipse used to draw the rounded corners using $x3$ and $y3$.

Syntax: `DrawRoundRectangle(x1,y1,x2,y2,x3,y3)`

Parameters:

$x1,y1$	The upper-left corner of the rectangle.
$x2,y2$	The lower-right corner of the rectangle.
$x3$	The width of the ellipse used to draw the rounded corners.
$y3$	The height of the ellipse used to draw the rounded corners.

Examples: This example draws a rounded rectangle that resembles an OK button. The upper-left corner of the button is at $(10,10)$ and the lower-right is at $(20,17)$. The height and width of the ellipse used to draw the rounded corners are both 2.

```
UseBrush(SOLID,192,192,192)      {Use a light gray brush}
DrawRoundRectangle(10,10,20,17,2,2)
UseBackground(TRANSPARENT,0,0,0) {Make text background transparent}
DrawText(12,11,"OK")
WaitInput()
```

Related Commands: `UsePen`, `UseBrush`

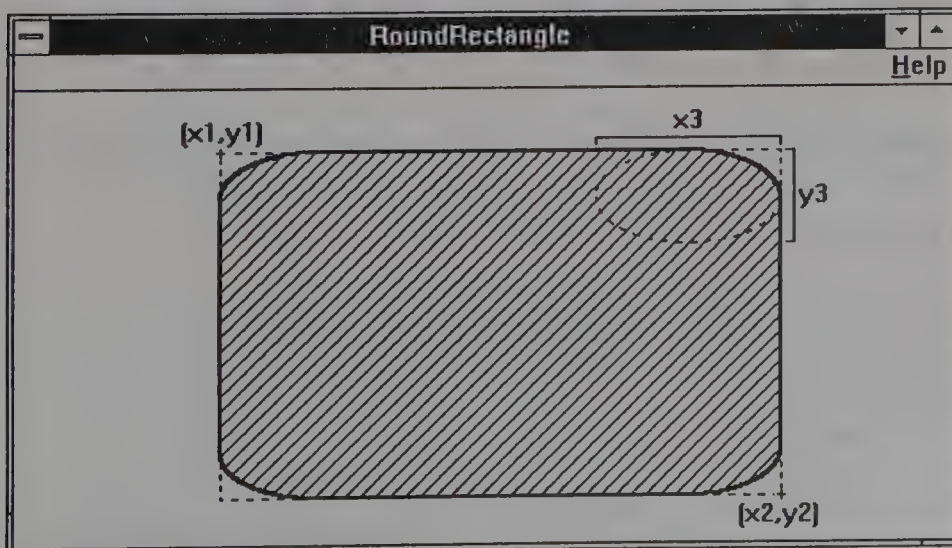


Figure 14-18 The `DrawRoundRectangle` coordinates

DrawSizedBitmap

This command reads the contents of a bitmap (.BMP) file and either stretches or compresses it to fit within a specified rectangle. You indicate the upper-left corner of the rectangle using (*x1,y1*), and the lower-right corner using (*x2,y2*). This command will create a mirror image of the bitmap if (*x1,y1*) is the lower-right instead of the upper-left corner.

Syntax: `DrawSizedBitmap(x1,y1,x2,y2, "Filename")`

Parameters:

<i>x1,y1</i>	The upper-left corner of the rectangle.
<i>x2,y2</i>	The lower-right corner of the rectangle.
"Filename"	The name of the bitmap file in quotes. Be sure to include the path if the bitmap file is not in the current directory.

Examples: The following program uses the `DrawBitmap` command to read the ribbons bitmap (RIBBONS.BMP) and place it on the screen. (This bitmap is located in the \WINDOWS directory.) Next, it uses the `DrawSizedBitmap` command to draw another copy of the bitmap next to the first one and reduce its size in the process.

```
UseCoordinates(PIXEL)
DrawBitmap(10,10,"C:\WINDOWS\RIBBONS.BMP")
DrawSizedBitmap(350,10,514,174,"C:\WINDOWS\RIBBONS.BMP")
WaitInput()
```

This next example is the same as the previous one, except that it flips the second copy of of the ribbons bitmap as it places it on the screen, as shown in Figure 14-19.

```
UseCoordinates(PIXEL)
DrawBitmap(10,10,"C:\WINDOWS\RIBBONS.BMP")
DrawSizedBitmap(514,174,350,10,"C:\WINDOWS\RIBBONS.BMP")
WaitInput()
```

Related Commands: `DrawBitmap`

DrawText

This command draws a character string in the window, using the current font. The starting position of the string is given by the *x* and *y* parameters.

Syntax: `DrawText(x,y, "Text")`

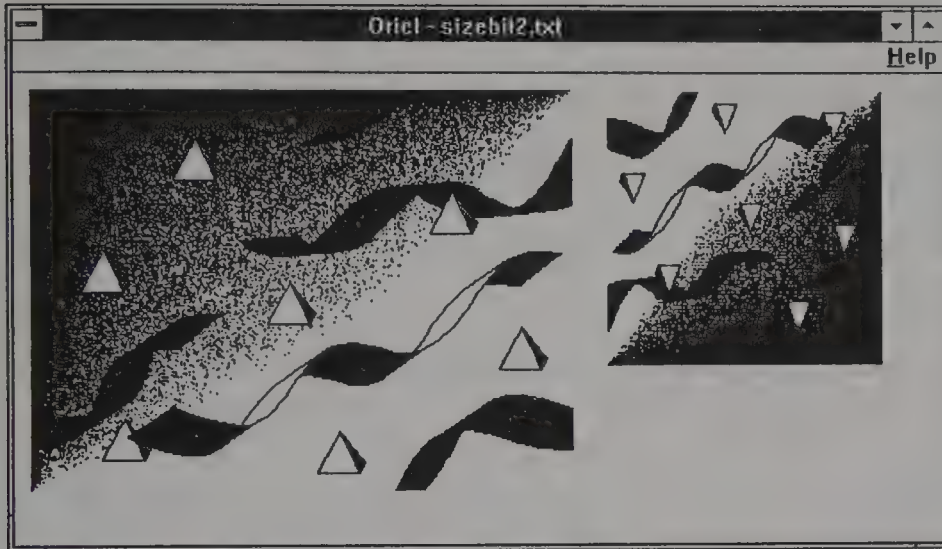


Figure 14-19 Flipping a bitmap with DrawSizedBitmap

Parameters:

<i>x</i>	The x-coordinate of the starting point of the string.
<i>y</i>	The y-coordinate of the starting point of the string.
"Text"	The character string to be drawn, in quotes.

Remark: By default, Oriel for Windows uses the System font in black when drawing text. You can change the font with the UseFont command.

Example: This example uses the Terminal font to draw the string "Windows 3 Power Tools" in a window in two different sizes. Figure 14-20 shows the results.

```
UseCoordinates (PIXEL)
UseFont ("Terminal", 10, 20, NOBOLD, NOITALIC, NOUNDERLINE, 0, 0, 0)
DrawText (10, 10, "Windows 3 Power Tools")
UseFont ("Terminal", 25, 40, NOBOLD, NOITALIC, NOUNDERLINE, 0, 0, 0)
DrawText (10, 30, "Windows 3 Power Tools")
WaitInput ()
```

This next program shows the effect of the current background setting when you draw text. In this example, the first character string is drawn using the default background, opaque white. The second string is drawn after the background has been set to light grey. Before the third character string is drawn, the font is changed to white, creating a reverse effect. Figure 14-21 shows the results. (See UseFont and UseBackground for more on this.)

```
UseCoordinates (PIXEL)
UseFont ("System", 10, 20, NOBOLD, NOITALIC, NOUNDERLINE, 0, 0, 0)
DrawText (10, 10, "Windows 3 Power Tools")
```

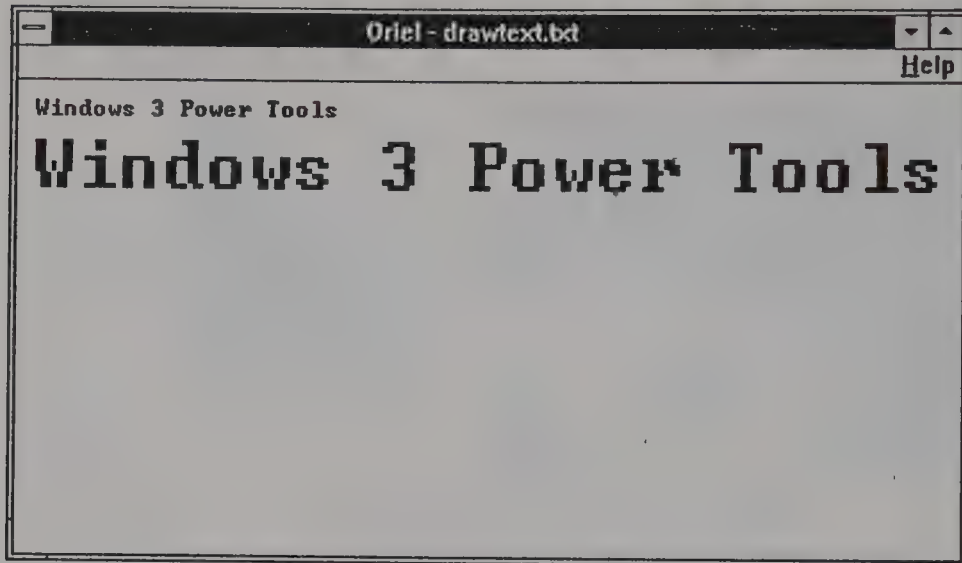



Figure 14-20 The effect of the DrawText command

```
UseBackground(OPAQUE,128,128,128){Light grey background}  
DrawText(10,30,"Windows 3 Power Tools")  
UseFont("System",10,20,NOBOLD,NOITALIC,NOUNDERLINE,255,255,255)  
DrawText(10,50,"Windows 3 Power Tools")  
WaitInput()
```

Related Commands: UseFont, UseBackground

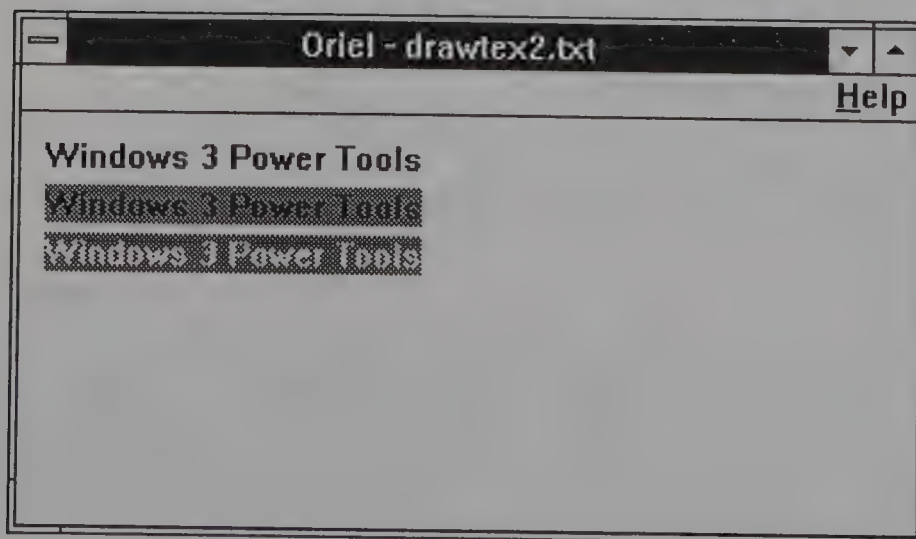


Figure 14-21 The current background affects your text

End

This command terminates an Oriel program. You can use End anywhere in a program, either in the main routine or in subroutine. If Oriel encounters an End in a subroutine, it will terminate execution not only for the subroutine but for the entire program as well.

Syntax: End

Remark: Oriel also ends a program when it encounters an end of file or CTRL-Z character (1AH).

Gosub

This command executes a block of code as a subroutine. When the subroutine is completed—that is, a Return command is encountered—control returns to the command following the Gosub.

Syntax: Gosub label / Return

Parameter:

label The label in the current text file where the subroutine begins.

Remark: You can have up to 50 Gosub/Return nesting levels.

Example: This example uses a simple subroutine to draw text in different point sizes. The subroutine uses the Count variable to set the width of the font as well as the y-coordinate for each line that is drawn. Figure 14-22 shows the results. Notice that the text does not increase in size in smooth increments. Rather, it finds the closest match it can for the height and width you've chosen (see UseFont for more details).

```
{Initialize counter}
    Set Count=1

Next:
    If Count > 10 Then Goto Wait_for_Input
    Gosub Put_Line
    Set Count=Count+1
    Goto Next

Put_Line:
    UseFont("System",Count,1,NOBOLD,NOITALIC,NOUNDERLINE,0,0,0)
```

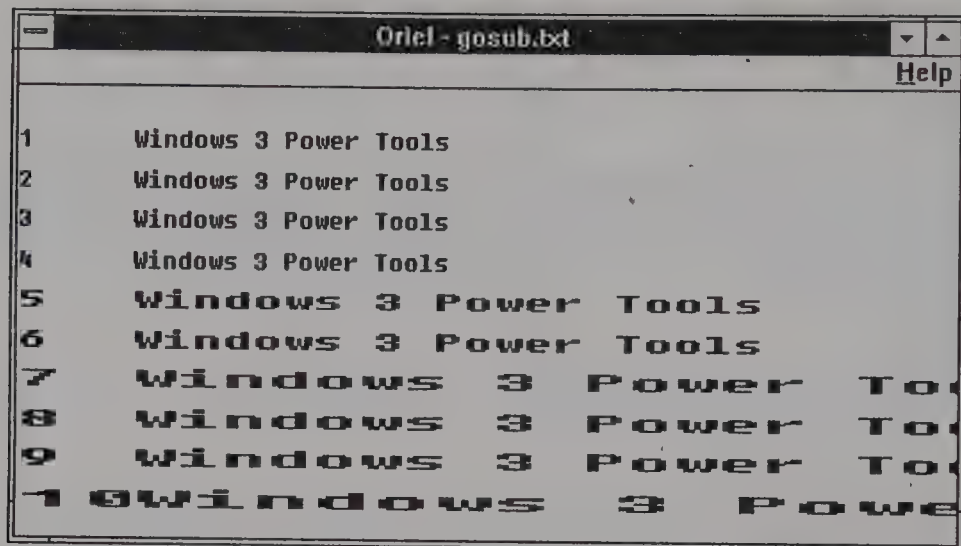


Figure 14-22 Using a subroutine to draw different font sizes

```
Set y=Count*7
DrawNumber(0,y,Count)
DrawText(20,y,"Windows 3 Power Tools")
Return
```

```
Wait_for_Input:
WaitInput()
```

Related Commands: Goto, Run

Goto

This command transfers control unconditionally to a label.

Syntax: Goto *label*

Parameters:

label A label in the current text file.

Remark: You can use a : (colon) following *label* if you want. For example, Oriel treats the following two commands identically:

```
Goto Next
Goto Next:
```

Example: The following example launches Notepad and/or Calculator based on your responses to message boxes. The first message box asks whether you want to

run Notepad and displays Yes and No buttons. If you select Yes (Button is set to 1), the program issues the Run command to launch Notepad. If you select No, the Button variable is set to 2 and the program uses a Goto to transfer control to the label Run_Calc; a similar message box then asks whether you want to run Calculator.

```
{Program to launch Notepad and/or Calculator}
  MessageBox(YESNO,1,QUESTION,"Run Notepad?",
             "Notepad?",Button)
  If Button=2 Then Goto Run_Calc
  Run("NOTEPAD.EXE")
Run_Calc:
  MessageBox(YESNO,1,QUESTION,"Run Calculator?",
             "Calculator?",Button)
  If Button=1 Then Run("CALC.EXE")
End
```

Related Commands: Gosub

If

This command lets you execute commands conditionally. If the condition tested is true, execution continues on the same line, following the Then. If the condition tested is false, execution continues on the next line following the If; all commands on the same line as the If are ignored.

Syntax: If <condition> Then <commands>

Parameters:

<condition> A conditional expression used to compare two integers taking the form

Integer1 logical_operator Integer2

where *Integer1* and *Integer2* are integers (or integer variables) and *logical_operator* is one of the logical operators in Table 14-2. For example, the following are all valid conditional expressions:

Red <= 255

Mouse_x1 <> Mouse_x2

10 > Counter

<commands> One or more Oriel commands.

Remark: Because Oriel treats the split vertical bar (|) as white space, it serves as a nice way to separate commands following the Then. Here's an example:

```
If x<10 Then Set x=x+1 | Set y=20 | Goto Next
```

<i>Operator</i>	<i>Meaning</i>
=	equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
<>	not equal to

Table 14-2 Logical Operators

Without the split vertical bars, this line would be more difficult to read because the commands following the Then would be all jumbled together.

Example: This program draws an 8-by-8 matrix of colored boxes starting at (5,5). Figure 14-23 shows the positioning of the boxes in the window. The program uses several If commands to test the value of different variables and branch accordingly. For example, when you first start the program, it draws the first box in the window, increments the Color_Red variable by 64, and tests the value of that variable using the following If command:

```
If Color_Red<=255 Then Goto Color_Ready
```

In this case, if Color_Red is less than or equal to 255, the program branches to the Color_Ready: label. Otherwise, the program continues execution on the next line following the If.

```
{Initialize variables}
    Set Color_Red=63
    Set Color_Green=63
    Set Color_Blue=63
    Set Color_X1=5           {Start the matrix at (5,5)}
    Set Color_Y1=5

Draw_Color:
    UseBrush(SOLID,Color_Red,Color_Green,Color_Blue)
    Set Color_X2=Color_X1+8
    Set Color_Y2=Color_Y1+8
    DrawRectangle(Color_X1,Color_Y1,Color_X2,Color_Y2)

    Set Color_Red=Color_Red+64
    If Color_Red<=255 Then Goto Color_Ready
    Set Color_Red=63
    Set Color_Green=Color_Green+64
```

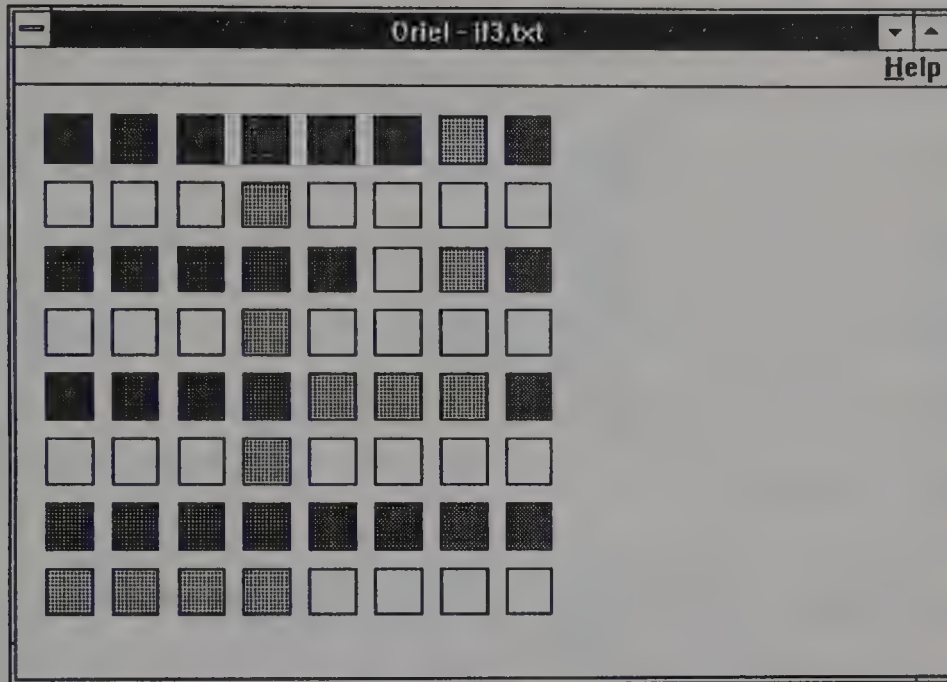


Figure 14-23 Using If to test variables and branch accordingly

```

    If Color_Green<=255 Then Goto Color_Ready
    Set Color_Green=63
    Set Color_Blue=Color_Blue+64
Color_Ready:

    Set Color_X1=Color_X1+11
    If Color_X1<=82 Then Goto Draw_Color
    Set Color_X1=5
    Set Color_Y1=Color_Y1+11
    If Color_Y1<=82 Then Goto Draw_Color

    WaitInput()
    
```

See MessageBox and Set for another example of the If command.

Related Commands: Goto, Gosub

MessageBox

This command lets you create a custom message box complete with your own prompt, caption, and push buttons.

<i>Token</i>	<i>Meaning</i>
OK	Causes the message box to display one push button: OK.
OKCANCEL	Causes the message box to display two push buttons: OK and Cancel.
YESNO	Causes the message box to display two push buttons: Yes and No.
YESNOCANCEL	Causes the message box to display three push buttons: Yes, No, and Cancel.

Table 14-3 Push Button Types

Syntax: `MessageBox (Type, Default_button, Icon, "Text", "Caption", Button_pushed)`

Parameters:

<i>Type</i>	Specifies the type of push buttons that appear within the message box. It must be one of the tokens in Table 14-3.
<i>Default_button</i>	An integer indicating the button you want to appear as the default. The buttons are numbered left to right starting with 1.
<i>Icon</i>	Controls the type of icon that appears in front of "Text" in the message box. You must use one of the tokens in Table 14-4.
<i>"Text"</i>	The message to be displayed within the message box. "Text" can have multiple lines in the script (see the second example below). You can have as many lines as you like, but be aware that if you have too many, the message box may not fit on the screen.
<i>"Caption"</i>	The text you want to place at the top of the message box.
<i>Button_pushed</i>	A variable that returns a number corresponding to the button that the user pushed. The buttons are numbered left to right starting with 1.

Examples: The following program displays a message box that asks whether you want to change the background of the window to blue. The message box has Yes and No command buttons and a question-mark icon, as shown in Figure 14-24.

```

MessageBox (YESNO, 1, QUESTION, "Change the background to blue?",
            "Background box", Button)
If Button=1 Then UseBackground (OPAQUE, 0, 0, 255) | DrawBackground
WaitInput ()

```

This next example displays three message boxes, each with more lines of text than the previous one.

<i>Token</i>	<i>Meaning</i>
INFORMATION	Causes the message box to display an icon consisting of a lowercase i in a circle.
EXCLAMATION	Causes the message box to display an exclamation-point icon.
QUESTION	Causes the message box to display a question-mark icon.
STOP	Causes the message box to display a stop sign icon.
NOICON	Causes the message box to display no icon.

Table 14-4 Icon Types

```

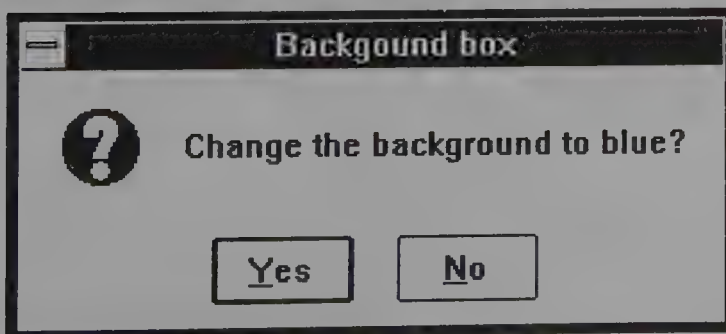
{Show a single-line message}
    MessageBox(OKCANCEL,1,QUESTION,
        "See a message box with a two-line message?",
        "One-line message",Button_mashed)
    If Button_mashed=2 Then Goto Wait_for_input

{Show a two-line message}
    MessageBox(YESNO,1,QUESTION,
        "This box has two lines of text.
        Do you want to see one with three?",
        "Two-line message",Button_mashed)
    If Button_mashed=2 Then Goto Wait_for_input

{Show a three-line message}
    MessageBox(OK,1,INFORMATION,
        "This box has three lines of text.
        You've now seen message boxes with,
        one, two, and three lines of text.",
        "Three-line message",Button_mashed)

Wait_for_input:
    WaitInput()

```

**Figure 14-24** A sample message box

Run

This command lets you run a Windows or non-Windows application from within an Oriel program.

Syntax: `Run("Command_line")`

Parameter:

`"Command_line"` A string containing the command line (filename plus optional parameters) for the application to be executed.

Remarks: When Oriel for Windows encounters a Run command, it launches the program specified in `"Command_line"`, and then immediately executes the next command in the script without pausing. If you prefer that your Oriel program pause after executing a Run command, you must use a `SetWaitMode(FOCUS)` command before the Run command and a `WaitInput(1)` command immediately after (see the example below). This technique is particularly useful when you use the Run command to execute macros you've recorded with Recorder.

If `"Command_line"` does not contain a directory path, Windows searches for the executable file in the following order:

1. The current directory.
2. The Windows directory.
3. The Windows system directory (typically, `C:\WINDOWS\SYSTEM`).
4. The directories specified in the PATH variable.
5. The list of directories mapped in a network.

If you do not include an extension with the application filename, Oriel assumes an .EXE extension.

Examples: This example launches Notepad and has it automatically load the WIN.INI file on startup. Without pausing, the program then launches COMMAND.COM.

```
Run("NOTEPAD WIN.INI")
Run("C:\COMMAND.COM")
WaitInput()
```

This next example is a variation of the previous one. Rather than run the two programs in quick succession, however, it pauses after loading Notepad. Only after you leave Notepad (or switch back to Oriel for Windows) does the program pick up execution following `WaitInput(1)` and execute the second Run command to launch COMMAND.COM.

```

SetWaitMode(FOCUS)
Run("NOTEPAD WIN.INI")
WaitInput(1)    {Wait until the focus returns}
Run("C:\COMMAND.COM")
WaitInput()

```

Related Commands: WaitInput

Set

This command lets you assign an integer to a variable, or perform simple mathematical calculations using integers and store the result to a variable.

Syntax: Set Variable = 0,1,2,3,...,65535

or

Set Variable = <math_expression>

Parameters:

<i>Variable</i>	A valid variable name (see "Variables" earlier).
<math_expression>	<p>A mathematical expression of the form</p> <p><i>Integer1 math_operator Integer2</i></p> <p>where <i>Integer1</i> and <i>Integer2</i> are integers (or integer variables) and <i>math_operator</i> is one of the mathematical operators in Table 14-5. For example, the following are all valid mathematical expressions:</p> <p>Counter+3</p> <p>X2/18</p> <p>10*Box_size</p>

Remarks: In all cases, the results of a mathematical expressions are rounded down and stored as integers. For example, the command Set Green=2/3 results in the

<i>Operator</i>	<i>Meaning</i>
+	Addition
-	Subtraction
*	Multiplication
/	Division

Table 14-5 Mathematical Operators

variable Green being assigned a value of zero. Likewise, the command Set Green=3/2 results in Green being assigned a value of 1.

If you want to perform more than one mathematical operation at a time, the only way to do so is to break them up into separate operations. For example, the following command is invalid:

```
Set y=3*x+4
```

But you can accomplish the same thing using these two commands:

```
Set y=3*x
```

```
Set y=y+4
```

Example: The following program draws the 3-D ball in Figure 14-25 in red. By modifying the Set commands at the beginning of the program, you can change the size of the ball and its position on the screen.

```
{Initialize variables}
    UseCoordinates(PIXEL)
    UsePen(NULL,0,0,0,0)      {Use NULL pen for shading}
    Set Red=0
    Set Blue=0
    Set Green=0
    Set x1=150                {Starting x position}
    Set Final_x1=x1
    Set y1=160                {Starting y position}
    Set Final_y1=y1
    Set Ball_size=90          {Ball size}
    Set Count=1
Next_shade:
    If Count>10 Then Goto Flood
    Set x2=x1+Ball_size
    Set y2=y1-Ball_size
    UseBrush(SOLID,Red,Green,Blue)
    DrawEllipse(x1,y1,x2,y2)
    Set x1=x1+2
    Set y1=y1-2
    Set Red=Red+25
    Set Count=Count+1
    Goto Next_shade

Flood:
    UsePen(SOLID,2,0,0,0)
    UseBrush(NULL,0,0,0)
    Set x2=Final_x1+Ball_size
    Set y2=Final_y1-Ball_size
    DrawEllipse(Final_x1,Final_y1,x2,y2)
```

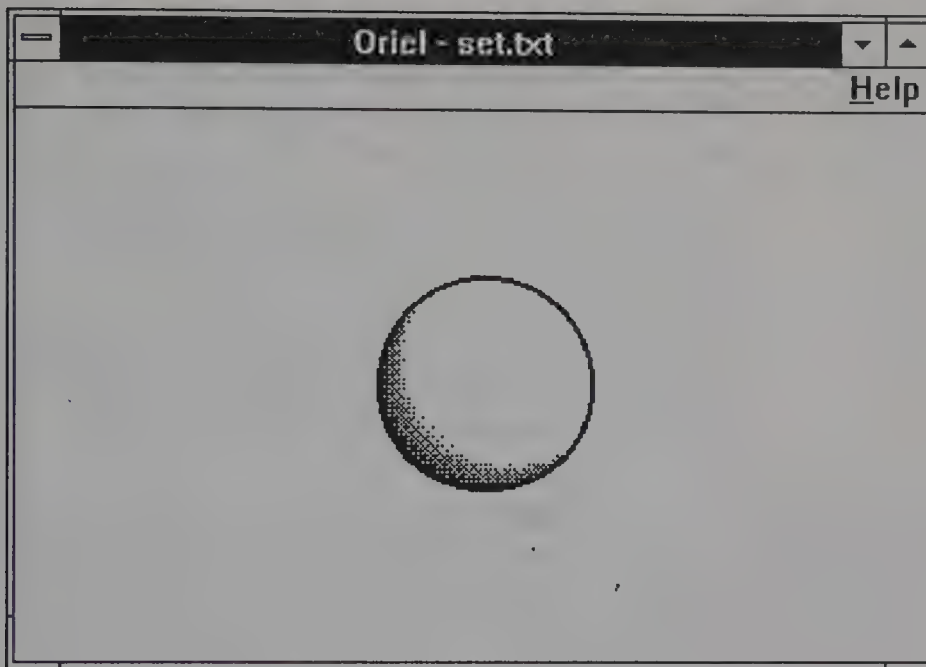



Figure 14-25 A 3-D ball

```
UseBrush(SOLID,255,255,255) ' {White brush}
DrawFlood(1,1,0,0,0)         {Flood with white}
WaitInput()
```

Related Commands: DrawNumber

SetKeyboard

This command lets you get keyboard input from the user. When the program is pausing for input and the user presses a specified key, the program branches to the label associated with that key.

Syntax: SetKeyboard()

or

```
SetKeyboard("a",label,
            "^a",label,
            vkey,label)
```

Parameters:

"a"	Any white key on the keyboard (except function keys and certain keys on the numeric keypad) enclosed in quotation marks. For example, "a" represents lowercase <i>a</i> and "R" represents uppercase <i>R</i> .
-----	---

<code>" ^a"</code>	Any white key on the keyboard (except function keys and certain keys on the numeric keypad) in combination with CTRL. For example, <code>"^b"</code> represents CTRL+b and <code>"^V"</code> represents CTRL+V.
<code>vkey</code>	A virtual key number taken from Table 14-6. For example, the virtual key number for the F1, function key is 112. Using a virtual key number is the only way to test for certain keys, including function keys and several keys on the numeric keypad.
<code>label</code>	A label you want Oriel to branch to when the user presses the preceding key. For example, the command <code>SetKeyboard ("C",Run_calc)</code> causes the program to branch to the label <code>Run_calc</code> when the user presses <code>C</code> .

Remarks: When Oriel encounters a `SetKeyboard` command in your program, it does not immediately branch anywhere. Rather, it waits until it encounters a `WaitInput()` command (which causes the program to pause indefinitely for user input) and the user presses a specified key. Only then does control transfer to the *label* associated with that key.

The syntax above shows only three keys in the `SetKeyboard` list. However, you can actually include as many keys as you want in the list.

A `SetKeyboard` command remains in effect until any of the following occurs:

- You use another `SetKeyboard` command.
- You use `SetKeyboard` without any parameters to reset the keyboard.
- The program ends.

Examples: The following example puts the message "Press E to end the program" on the screen, and then pauses the program indefinitely until you press *e* or *E*. When you press either key, the program ends.

```
DrawText(1,1,"Press E to end the program")
SetKeyboard("E",End_it,"e",End_it)
WaitInput()
End_it:
End
```

This next example places two buttons on the screen, as shown in Figure 14-26. When you click on a button with the mouse, or press the key associated with the button, the program runs the named Windows application. On the other hand, if you press F1, the program displays a message box with some simple help text.

```
{Draw Buttons}
    UseBrush(Solid,192,192,192)           {Grey brush}
    UseBackground(TRANSPARENT,0,0,0) {Prevent white text background}
    DrawRoundRectangle(10,10,40,25,5,5)
```

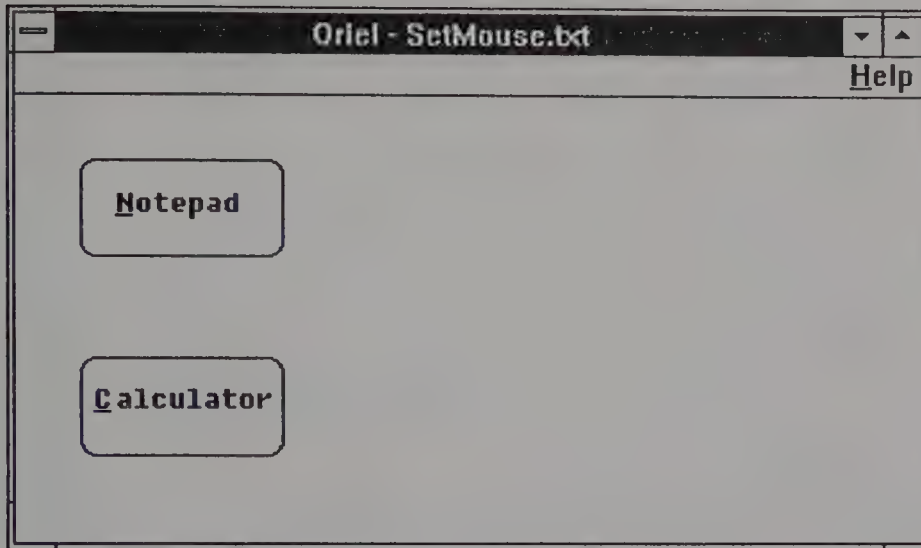


Figure 14-26 Getting keyboard input with SetKeyboard

```

UseFont("System",1,3,NOBOLD,NOITALIC,UNDERLINE,0,0,0)
DrawText(15,14,"N")

UseFont("System",1,3,NOBOLD,NOITALIC,NOUNDERLINE,0,0,0)
DrawText(18,14,"otepad")
DrawRoundRectangle(10,40,40,55,5,5)

UseFont("System",1,3,NOBOLD,NOITALIC,UNDERLINE,0,0,0)
DrawText(12,44,"C")

UseFont("System",1,3,NOBOLD,NOITALIC,NOUNDERLINE,0,0,0)
DrawText(15,44,"alculator")

{Set up the mouse}
SetMouse(10,10,40,30,Run_Notepad,Temp,Temp,
        10,40,40,60,Run_Calc,Temp,Temp)

{Set up the keyboard}
SetKeyboard("N",Run_Notepad,
           "n",Run_Notepad,
           "C",Run_Calc,
           "c",Run_Calc,
           112,Help_Box)      {112=virtual key for F1}

Wait_for_Input:
WaitInput()

```

```

Run_Notepad:
    Run("NOTEPAD.EXE")
    Goto Wait_for_Input

Run_Calc:
    Run("CALC.EXE")
    Goto Wait_for_Input

Help_Box:
    MessageBox(OK,1,NOICON,
    "Pick a button to run the Windows application",
    "Help box",TEMP)
    Goto Wait_for_Input

```

Related Commands: SetMenu, SetMouse, WaitInput

<i>Value</i>	<i>Description</i>
8	BACKSPACE
9	TAB
12	5 on numeric keypad with NUMLOCK off
13	ENTER
16	SHIFT
17	CTRL
18	ALT
19	PAUSE (or CTRL+NUMLOCK)
20	CAPS LOCK
27	ESCAPE
32	SPACEBAR
33	PGUP
34	PGDN
35	END
36	HOME
37	LEFTARROW
38	UPARROW
39	RIGHTARROW
40	DOWNARROW
44	PRINTSCREEN
45	INSERT
46	DELETE
48	0
49	1

Table 14-6 Virtual Key Numbers

<i>Value</i>	<i>Description</i>
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
96	Numeric key pad 0 (NUMLOCK must be on)
97	Numeric key pad 1 (NUMLOCK must be on)
98	Numeric key pad 2 (NUMLOCK must be on)
99	Numeric key pad 3 (NUMLOCK must be on)
100	Numeric key pad 4 (NUMLOCK must be on)

Table 14-6 Virtual Key Numbers (*continued*)

<i>Value</i>	<i>Description</i>
101	Numeric key pad 5 (NUMLOCK must be on)
102	Numeric key pad 6 (NUMLOCK must be on)
103	Numeric key pad 7 (NUMLOCK must be on)
104	Numeric key pad 8 (NUMLOCK must be on)
105	Numeric key pad 9 (NUMLOCK must be on)
106	Numeric key pad *
107	Numeric key pad +
109	Numeric key pad -
110	Numeric key pad . (NUMLOCK must be on)
111	Numeric key pad /
112	F1
113	F2
114	F3
115	F4
116	F5
117	F6
118	F7
119	F8
120	F9
121	F10
122	F11
123	F12
124	F13
125	F14
126	F15
127	F16
144	NUM LOCK
145	SCROLL LOCK

The following key codes apply to US keyboards only:

186	Colon/semi-colon
187	Plus/equal
188	Less than/comma
189	Underscore/hyphen
190	Greater than/period
191	Question/slash
192	Tilde/backwards single quote

Table 14-6 Virtual Key Numbers (*continued*)

<i>Value</i>	<i>Description</i>
219	Left curly brace/left square brace
220	Pipe symbol/backslash
221	Right curly brace/right square bracket
222	Double quote/single quote

Table 14-6 Virtual Key Numbers

SetMenu

This command lets you create your own custom menus. When the program is pausing for input and the user selects a menu item, the program branches to the label associated with that menu item.

Syntax: `SetMenu()`

or

```
SetMenu("Top1", IGNORE/label,
        "ItemA", IGNORE/label,
        "ItemB", IGNORE/label,
        SEPARATOR,
        "ItemC", IGNORE/label,
        ENDPOPUP,
        "Top2", IGNORE/label,
        .
        .
        .
        ENDPOPUP)
```

Parameters:

"Top1", "Top2"

The items that are to appear on the main or top-level menu bar, also called the *action bar*. You can have as many items as you like on the main menu bar. (If you have more than a single line's worth, Windows will extend the menu bar to a second line and beyond.)

"ItemA", "ItemB",
"ItemC"

The items that are to appear within a popup menu, also called a *pull-down menu*. You can have as many items as you like within a popup menu.

<i>label</i>	A label you want Oriel to branch to when the user selects a menu item.
IGNORE	Tells Oriel not to do anything when the menu item is selected. When IGNORE follows an item on the main menu bar, Oriel displays the item's popup menu, provided you've defined one.
SEPARATOR	Divides the items in a popup menu into groups.
ENDPOPUP	Ends a popup menu. In addition, this token is always the last argument in a SetMenu command.

Remarks: A top-level menu consists of one or more items—“*Top1*”, “*Top2*”, “*Top3*”, and so on. Below each top-level menu item is a popup menu. You can have one or more items within a popup menu—“*ItemA*”, “*ItemB*”, “*ItemC*”, and so on.

When you define a menu in Oriel for Windows, you define it sequentially. You begin by setting up the first top-level menu item and its popup menu. You then set up the second top-level menu item and its popup, and so on.

Following each top-level menu item (“*Top1*”) and popup menu item (“*ItemA*”) is a label you want the program to branch to when a menu item is selected. If you don't want the program to branch anywhere, use the IGNORE token instead; the IGNORE token is most often used following a top-level menu item when all you want Oriel to do is show the item's popup menu. (The IGNORE token is recognized even if you've created an IGNORE label.)

When Oriel for Windows encounters a SetMenu command in your program, it does not immediately branch anywhere. Rather, it waits until it encounters a WaitInput() command (which causes the program to pause indefinitely for user input) and the user selects a menu item. Only then does control transfer to the *label* associated with that item.

Here are some conventions you may want to follow when creating Oriel menus in order to give the user additional information about the items within the menu:

- An *underlined letter* in a menu item indicates that the letter can be used to select the menu item. You create underlined letters by placing an & in front of the letter you want underlined. For example, the parameter “&Notepad” underlines the letter *N* in the Notepad menu item. When a letter is underlined, it is known as a *mnemonic*. To select an item on the top-level menu using its mnemonic, you press ALT+mnemonic, for example ALT+N. Once a popup menu appears, you can select an item within it by pressing that item's mnemonic alone.
- By placing an *exclamation point* at the end of a top-level menu item, you can indicate that no pop-up menu will appear when the user selects the item.
- By using a *separator*, you can divide items within a popup menu into groups. You can have as many separators as you like within a popup menu.

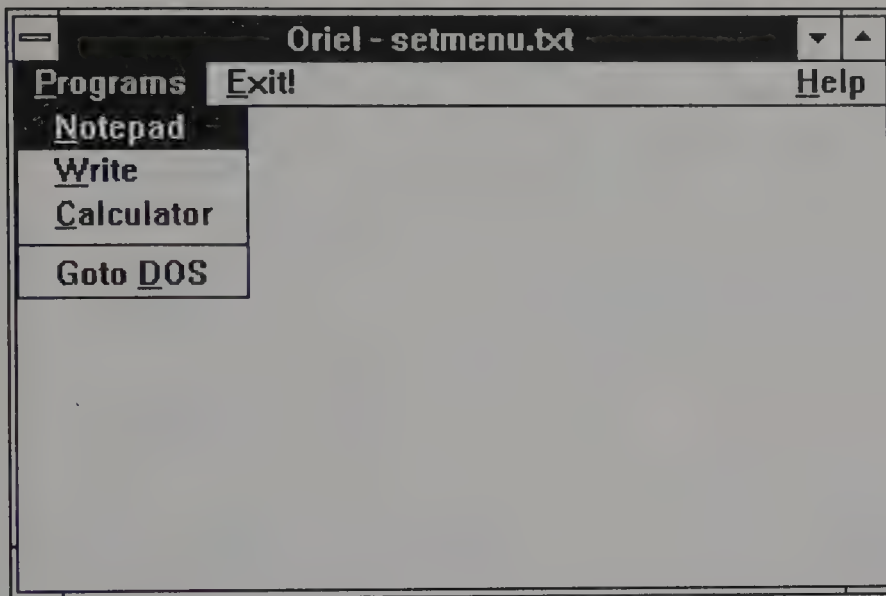


Figure 14-27 A menu created with SetMenu

Example: This example creates the simple menu show in Figure 14-27. As you can see, the menu has two top-level menu items, Programs and Exit!. (Because Exit! does not lead to a popup menu, it includes an exclamation point.) When you select an item from Programs' popup menu, the program branches to the associated label. For example, when you select Goto DOS, the program branches to the label DOS:, where it launches a copy of COMMAND.COM.

```
{Set up the menu}
SetMenu("&Programs", IGNORE,
        "&Notepad", Run_Notepad,
        "&Write", Run_Write,
        "&Calculator", Run_Calculator,
        SEPARATOR,
        "Goto &DOS", DOS,
        ENDPOPUP,
        "&Exit!", Shut_Down,
        ENDPOPUP)
```

```
Wait_for_Input:
    WaitInput()
```

```
Run_Notepad:
    Run("NOTEPAD.EXE")
    Goto Wait_for_Input
```

```
Run_Write:
    Run("WRITE.EXE")
```

```

        Goto Wait_for_Input

Run_Calculator:
    Run("CALC.EXE")
    Goto Wait_for_Input

DOS:
    Run("C:\COMMAND.COM")
    Goto Wait_for_Input

Shut_Down:
    End

```

Related Commands: SetKeyboard, SetMenu, WaitInput

SetMouse

This command lets you get mouse input. When the program is pausing for input and the user clicks the left mouse button within a specified rectangular region, the program branches to the label associated with that region.

Syntax: SetMouse()

or

```

SetMouse(region1_x1,region1_y1,region1_x2,region1_y2,label,x,y,
        region2_x1,region2_y1,region2_x2,region2_y2,label,x,y,
        .
        .
        regionn_x1,regionn_y1,regionn_x2,regionn_y2,label,x,y)

```

Parameters:

<i>regionx_x1,regionx_y1</i>	The upper-left corner of a rectangular mouse hit-testing region.
<i>regionx_x2,regionx_y2</i>	The lower-right corner of a rectangular mouse hit-testing region.
<i>label</i>	The label you want to branch to when the user clicks the mouse within the mouse hit-testing region (as defined by the previous four arguments).
<i>x,y</i>	The coordinates of the mouse pointer, as measured from the upper-left corner of the window.

Remarks: To get mouse input, you must set up rectangular areas in the window known as *mouse hit-testing regions*. When the user clicks the mouse within a hit-testing region, the program branches to the label associated with that region and saves the mouse pointer coordinates in the two variables that follow, *x* and *y*.

When Oriel encounters a `SetMouse` command in your program, it does not immediately branch anywhere. Rather, it waits until it encounters a `WaitInput()` command (which causes the program to pause indefinitely for user input) and the user clicks the mouse within a hit-testing region. Only then does control transfer to the *label* associated with that region.

You can have as many hit-testing regions as you like for a given `SetMouse` command. If two hit-testing regions overlap, Oriel will branch to the label for the first one in the list.

Example: The following program sets up two mouse hit-testing regions, a rectangle on the left-hand side of the screen and an Exit button on the right, as shown in Figure 14-28. When you click the mouse within the rectangle—the region (1,1) to (300,200)—the program saves the mouse pointer coordinates in `Mouse_x` and `Mouse_y` and branches to the `Mouse_hit` label, where it draws a small black rectangle at the location of the mouse pointer. If you click on the Exit button—the region (340,80) to (400,120)—the program branches to the `Goodbye` label where an `End` command ends the program.

```
{Set coordinate system to pixels}
    UseCoordinates(PIXEL)

{Draw the rectangle that will later become a hit-testing region}
    DrawRectangle(1,1,300,200)
    DrawText(30,210,"Click the mouse within the rectangle")

{Draw the Exit button using a grey brush}
    UseBrush(SOLID,192,192,192)
    DrawRoundRectangle(340,80,400,120,10,10)
    UseBackground(TRANSPARENT,0,0,0)
    DrawText(357,90,"Exit")

{Use a black brush}
    UseBrush(SOLID,0,0,0)

{Set up the mouse}
    SetMouse(1,1,300,200,Mouse_hit,Mouse_x,Mouse_y, {Rectangle}
            340,80,400,120,Goodbye,Temp,Temp)      {Exit button}

Wait_for_Input:
    WaitInput()
```

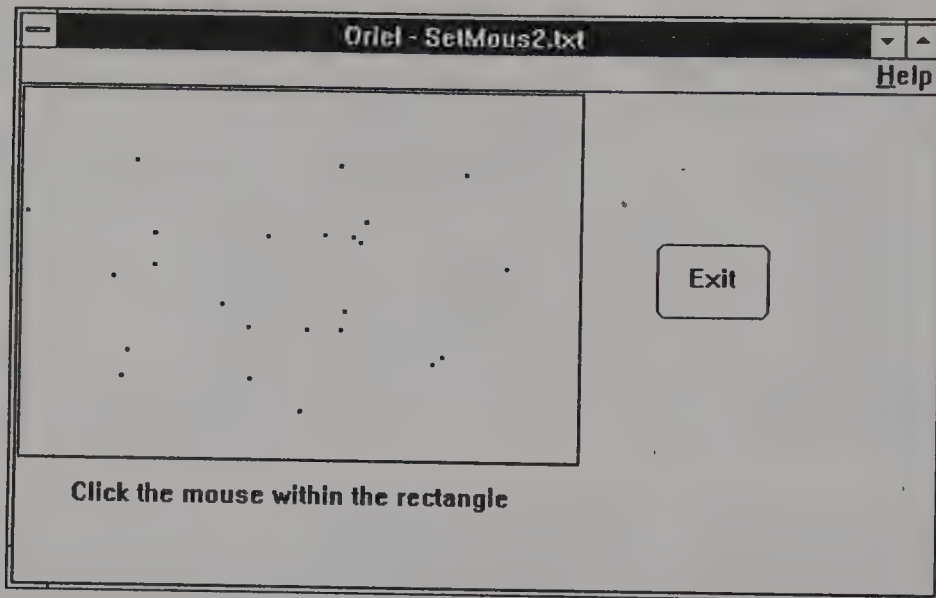


Figure 14-28 Using SetMouse to get mouse input

```

Mouse_hit:
    Set x2=Mouse_x+2
    Set y2=Mouse_y+2
    DrawRectangle(Mouse_x,Mouse_y,x2,y2)
    Goto Wait_for_Input

```

```

Goodbye:
    End

```

See the DrawNumber and SetKeyboard commands for other examples of SetMouse.

Related Commands: WaitInput

SetWaitMode

This command controls how Oriel for Windows behaves after a Run command starts another application and Oriel encounters a WaitInput command in your script.

Syntax: SetWaitMode(NULL/FOCUS)

Parameters:

NULL	Causes WaitInput to behave in the normal way. That is, WaitInput with an argument pauses Oriel a specified number of milliseconds, and WaitInput without an argument pauses Oriel indefinitely while it waits for user input. SetWaitMode(NULL) is the default.
FOCUS	Causes Oriel for Windows to pause until the focus returns when it encounters a WaitInput(1) command.

Remarks: By default, when you use a Run command to start another application, Oriel for Windows does not pause after executing the Run, but immediately executes the next command in the script. By using SetWaitMode(FOCUS) before the Run command, and WaitInput(1) immediately after, you can have Oriel pause until it gets the focus back—that is, until you switch back to Oriel. It will then pick up execution after the WaitInput(1). (See the example below.)

If you want Oriel for Windows to pause until it gets the focus back, you *must* use a WaitInput(1) command following the SetWaitMode(FOCUS) command. Note that using *1* as the argument for WaitInput is critical. If you use WaitInput without an argument, Oriel will pause indefinitely and show no signs of continuing after it gets the focus back.

In most cases, you do not need to use the SetWait Mode command. The only time you will want to consider using it is when your program includes a Run command and the commands that follow it in the script create a confusing display—for example, an Oriel message box appears in front of another application's window.

Using SetWaitMode(FOCUS) and WaitInput(1) is particularly important when you use the Run command to start a macro that launches several applications. Without these commands bracketing the Run, your Oriel program will not pause and may create a confusing display as it horns in on the applications that are being launched by the macro.

Examples: The following program uses the Run command to start Notepad. Without pausing, the program then executes the MessageBox command, which causes it to display a message box in front of the Notepad window, as shown in Figure 14-29.

```
SetWaitMode(NULL)
Run("NOTEPAD.EXE")
MessageBox(OK,1,INFORMATION,"Comes up in front of Notepad",
           "Oriel message box",Temp)
```

This next example is a variation of the previous one. It uses SetWaitMode(FOCUS) to change the behavior of WaitInput so that when Oriel

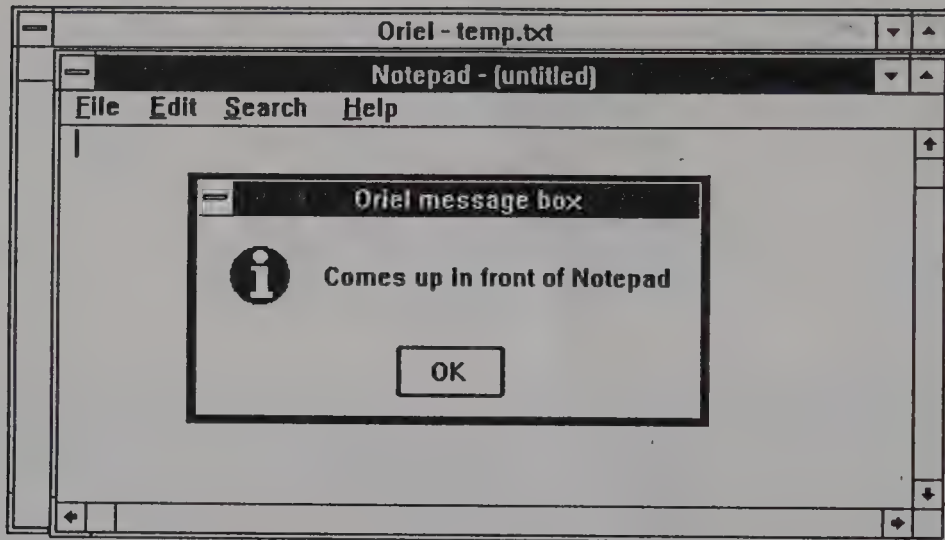


Figure 14-29 An Oriel message box appearing in front of another application's window

encounters `WaitInput(1)`, it pauses until it gets the focus back. By doing so, the message box only appears after you return to Oriel for Windows.

```
SetWaitMode(FOCUS)
Run("NOTEPAD.EXE")
WaitInput(1)                      {The 1 is necessary here}
MessageBox(OK,1,INFORMATION,"The focus is back","",Temp)
```

Related Command: `WaitInput`

SetWindow

This command maximizes, minimizes, or restores the Oriel window.

Syntax: `SetWindow(MAXIMIZE/MINIMIZE/RESTORE)`

Parameters:

MAXIMIZE	Maximizes the Oriel window.
MINIMIZE	Minimizes the Oriel window.
RESTORE	Restores the Oriel window.

Example: This program uses the `SetWindow` command to maximize, minimize, and restore the Oriel window in various ways, pausing for one second in between `SetWindow` commands.

```

SetWindow(MAXIMIZE)
WaitInput(1000)
SetWindow(RESTORE)
WaitInput(1000)
SetWindow(MINIMIZE)
WaitInput(1000)
SetWindow(RESTORE)
WaitInput()

```

UseBackground

This command has the dual role of controlling the background mode and the background color. The background mode establishes whether the GDI removes existing background colors before drawing any of the following:

- Text
- Shapes with a hatched brush
- Lines with a dotted pen

The GDI uses the background color to fill the small rectangles behind characters (called character cells), the gaps in dotted pens, and hatched lines in brushes.

Syntax: `UseBackground(OPAQUE/TRANSPARENT, r, g, b)`

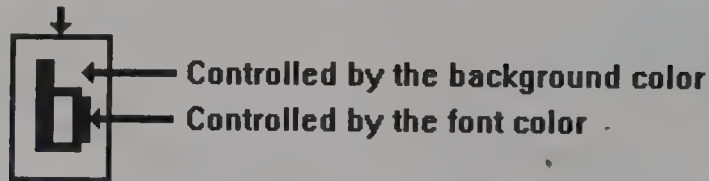
Parameters:

OPAQUE	Causes the background to be filled with the current color specified by <i>r, g, b</i> .
TRANSPARENT	The background is left unchanged.
<i>r, g, b</i>	Specifies the color of the background using a combination of red, green, and blue. The default background color is white (255,255,255).

Remarks: When you draw a character on the screen, the GDI does more than draw the squiggles that make up the character. It actually draws a rectangular area enclosing the character, called the *character cell*, as shown in Figure 14-30. The current font color determines the color of the characters, but the current background color controls the color in the character cells.

When the background mode is set to OPAQUE, the GDI fills the character cells with the RGB value you've set with the *r, g, b* parameters. On the other hand, when the background mode is set to TRANSPARENT, the GDI does not change the color in the character cells.

Character cell

**Figure 14-30** The character cell

These same principles apply to hatched brushes and dotted pens. That is, when the background mode is set to OPAQUE, the GDI fills the gaps in dotted pens and hatched brushes with the RGB value you've set with the *r*, *g*, *b* parameters. When the background mode is set to TRANSPARENT, the GDI does not change the color in the gaps.

You can change the color of the entire window's background using the DrawBackground command. This command also erases any window contents. (The background mode setting—OPAQUE or TRANSPARENT—has no effect on the DrawBackground command.)

Example: The following example shows the effect of the background mode and color settings when you draw text, draw lines with a dotted pen, and draw rectangles with a hatched brush (Figure 14-31). Notice that the white background only appears for character cells and the gaps in dotted pens and hatched brushes when the background mode is set to OPAQUE.

```
{Change window's background to light grey}
```

```
    UseBackground(TRANSPARENT,192,192,192)
```

```
    DrawBackground
```

```
{Change background mode to TRANSPARENT and color to white}
```

```
    UseBackground(TRANSPARENT,255,255,255)
```

```
{Draw text, dotted line, and hatched rectangle}
```

```
    DrawText(10,10,"Text with a TRANSPARENT background")
```

```
    UsePen(DOT,1,0,0,0)           {Dotted pen}
```

```
    DrawLine(10,21,40,21)
```

```
    UsePen(SOLID,1,0,0,0)         {Reset the pen to solid}
```

```
    UseBrush(CROSS,0,0,0)         {Cross hatched brush}
```

```
    DrawRectangle(10,30,40,40)
```

```
{Change back to default background mode and color (white)}
```

```
    UseBackground(OPAQUE,255,255,255)
```

```
{Draw more text and another dotted line and hatched rectangle}
```

```
    DrawText(10,60,"Text with an OPAQUE background")
```

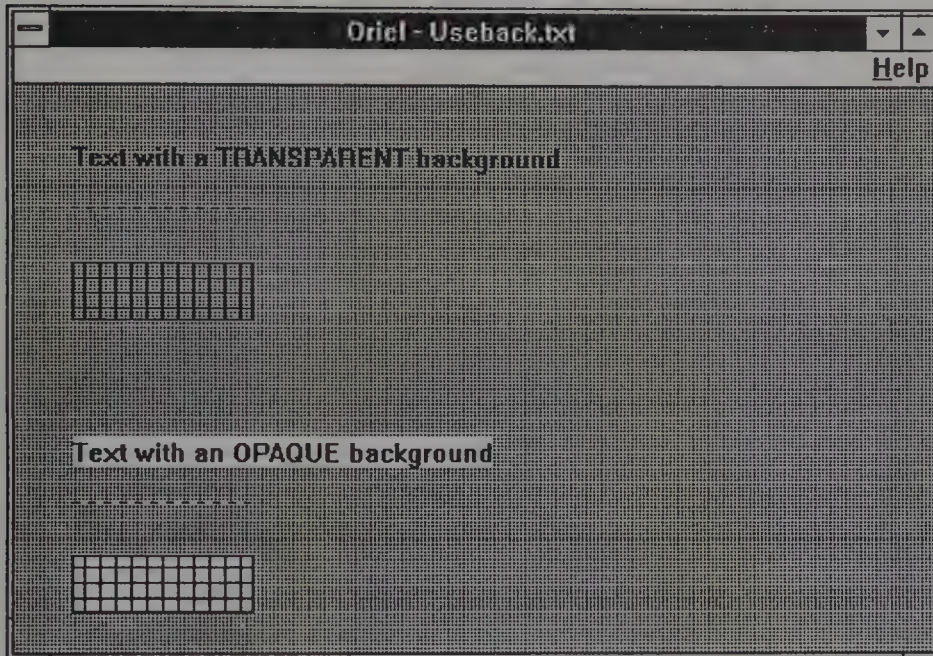


Figure 14-31 The effect of the background mode and color settings

```

UsePen(DOT,1,0,0,0)      {Dotted pen}
DrawLine(10,71,40,71)
UsePen(SOLID,1,0,0,0)    {Reset the pen to solid}
UseBrush(CROSS,0,0,0)    {Cross hatched brush}
DrawRectangle(10,80,40,90)

{Wait for input}
WaitInput()

```

See `DrawRoundRectangle` for another example of `UseBackground`.

Related Commands: `DrawBackground`, `DrawText`, `UsePen`, `UseBrush`

UseBrush

This command defines the style and color of the brush that will be used in subsequent drawing operations.

Syntax: `UseBrush(SOLID/DIAGONALUP/DIAGONALDOWN/DIAGONALCROSS/
HORIZONTAL/VERTICAL/CROSS/NULL, r, g, b)`

Parameters:

SOLID	A solid brush
DIAGONALUP	45-degree upward hatch (left to right)
DIAGONALDOWN	45-degree downward hatch (left to right)
DIAGONALCROSS	45-degree crosshatch
HORIZONTAL	Horizontal hatch
VERTICAL	Vertical hatch
CROSS	Horizontal and vertical crosshatch
NULL	A null or "hollow" brush (no color is drawn)
<i>r, g, b</i>	Specifies the color of the brush using a combination of red, green, and blue.

Examples: This program defines a crosshatched red brush and draws a round rectangle with it.

```
UseBrush(CROSS, 255, 0, 0)
DrawRoundRectangle(10, 10, 40, 40, 10, 10)
WaitInput()
```

This next program draws a series of circles using the eight available brush styles. Figure 14-32 shows the window that the program produces.

```
UseBrush(SOLID, 0, 0, 255)
DrawEllipse(10, 10, 30, 30)
DrawText(13, 32, "SOLID")

UseBrush(DIAGONALUP, 0, 0, 255)
DrawEllipse(49, 10, 69, 30)
DrawText(45, 32, "DIAGONALUP")

UseBrush(DIAGONALDOWN, 0, 0, 255)
DrawEllipse(88, 10, 108, 30)
DrawText(80, 32, "DIAGONALDOWN")

UseBrush(DIAGONALCROSS, 0, 0, 255)
DrawEllipse(127, 10, 147, 30)
DrawText(119, 32, "DIAGONALCROSS")

UseBrush(HORIZONTAL, 0, 0, 255)
DrawEllipse(10, 50, 30, 70)
DrawText(6, 72, "HORIZONTAL")

UseBrush(VERTICAL, 0, 0, 255)
DrawEllipse(49, 50, 69, 70)
DrawText(49, 72, "VERTICAL")
```

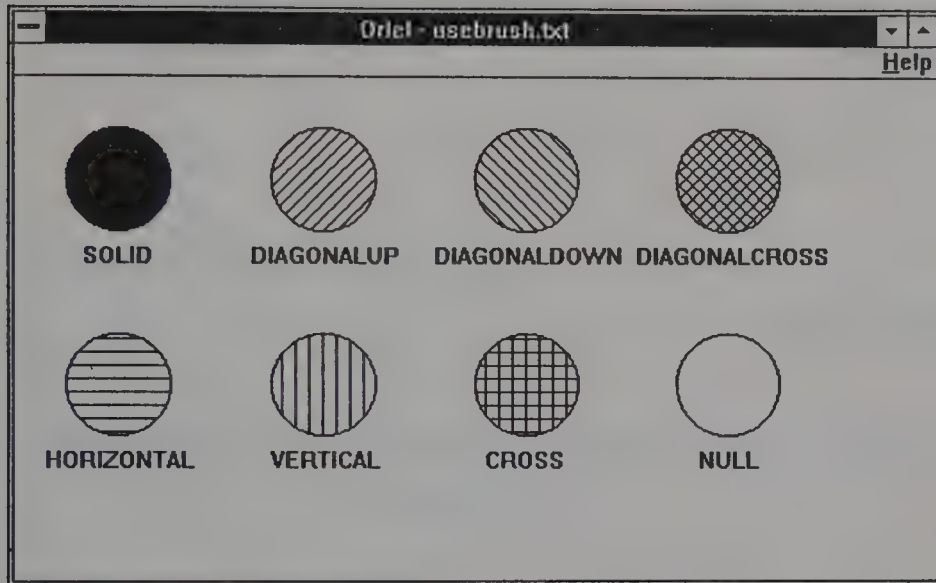


Figure 14-32 The eight different brush styles

```
UseBrush(CROSS,0,0,255)
DrawEllipse(88,50,108,70)
DrawText(90,72,"CROSS")
```

```
UseBrush(NULL,0,0,255)
DrawEllipse(127,50,147,70)
DrawText(131,72,"NULL")
```

```
WaitInput()
```

Related Comments: DrawArc, DrawChord, DrawEllipse, DrawFlood, DrawPie, DrawRectangle, DrawRoundRectangle

UseCaption

This command lets you place your own caption at the top of the Oriel window.

Syntax: UseCaption("Text")

Parameter:

"Text"	The text you want to use for the caption, enclosed in double quotes.
--------	--

Example: This command places the caption "All that Jazz" at the top of the Oriel window.

```
UseCaption("All that Jazz")
WaitInput()
```

UseCoordinates

Specifies Oriel's coordinate system as either pixel or metric and controls the unit of measure that Oriel uses for all subsequent drawing operations.

Syntax: UseCoordinates (PIXEL/METRIC)

Parameters:

PIXEL	Causes Oriel to use pixels as the unit of measure.
METRIC	Causes Oriel to use millimeters as the unit of measure. METRIC is the default.

Remarks: By using metric coordinates, you can guarantee that your programs will be device independent. That is, your programs will appear the same regardless of the video display type.

On the other hand, programs written using pixel coordinates are device dependent, which may or may not be a problem for you. For example, suppose you have an EGA system and you write a program that draws shapes in a window. When you run that program on a VGA system, the shapes will appear much smaller. By the same token, pixel coordinates are much more accurate than metric. Therefore, if you need to address the screen with the highest precision, you'll want to use pixel coordinates rather than metric.

Example: The following program draws the same shapes using pixel and metric coordinates. The different results are shown in Figure 14-33.

```
UseCoordinates (PIXEL)
DrawEllipse (50,10,70,30)
DrawRectangle (50,30,70,50)
DrawText (50,55,"Pixel")
UseCoordinates (METRIC)
DrawEllipse (50,10,70,30)
DrawRectangle (50,30,70,50)
DrawText (50,55,"Metric")
WaitInput()
```

Related Commands: All Draw commands.

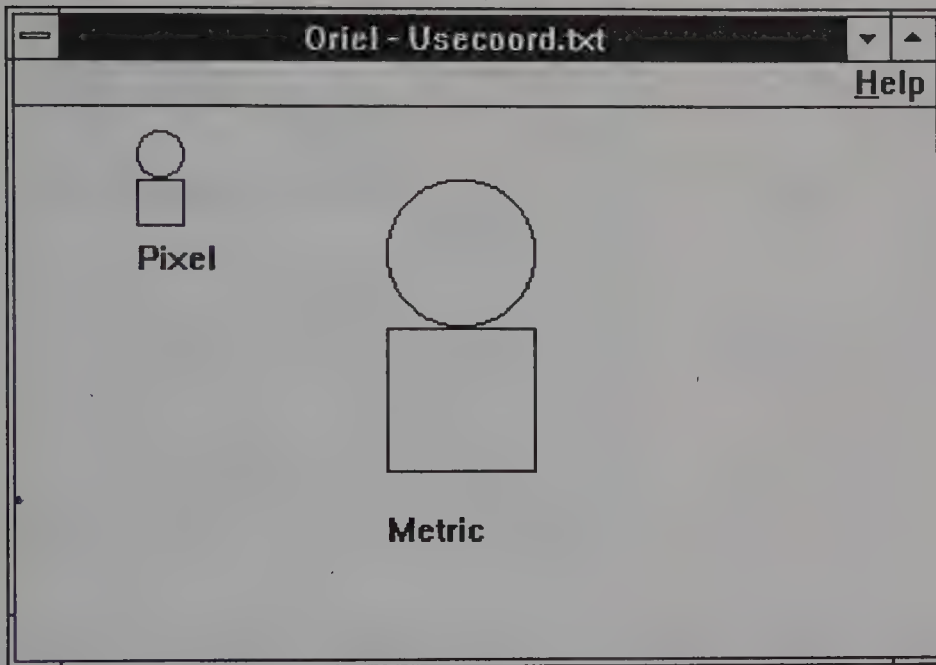


Figure 14-33 Pixel versus metric coordinates

UseFont

This command establishes the font that will be used for subsequent text-drawing operations. It lets you specify the width, height, style (bold, italics, and underlining), and color of the font.

Syntax: `UseFont("Name",Width,Height,Set_bold,Set_italic,
Set_underline,r,g,b)`

Parameters:

<i>"Name"</i>	The name of the font, enclosed in double quotes. The standard Windows screen fonts are Terminal, Roman, Script, Modern, Helv, Courier, Tms Rmn, Symbol, Digital, and System.
<i>Width</i>	The width of the font using the current coordinate mode (either pixels or millimeters). If you specify a width of zero, then the default width is used for the given font.
<i>Height</i>	The height of the font using the current coordinate mode (either pixels or millimeters). If you specify a height of zero, then the default height is used for the given font.

<i>Set_bold</i>	Specifies whether the font is bold and must be either of the following tokens: BOLD Font is bold NOBOLD Font is not bold
<i>Set_italic</i>	Specifies whether the font is italic and must be either of the following tokens: ITALIC Font is italic NOITALIC Font is not italic
<i>Set_underline</i>	Specifies whether the font is underlined and must be either of the following tokens: UNDERLINE Font is underlined NOUNDERLINE Font is not underlined
<i>r, g, b</i>	Specifies the color of the font using a combination of red, green, and blue.

Remarks: The GDI maintains all the fonts that are available in the system and their sizes in a font table. When you describe a font with the `UseFont` command, that font may or may not exist in the GDI's table. The GDI compares the font parameters you've supplied to the fonts it has in its table and returns the font with the closest match. This is a process known as font mapping.

The process by which the GDI chooses a font from the table that matches the font you've specified is based on a handicapping system. In short, it assigns certain penalties to the fonts in the table when they do not match the font you've specified. The font that has the fewest penalties is the font that the GDI selects, and is the one that is used the next time you draw text on the screen in Oriol.

In some cases, the GDI will synthesize a font to match the parameters you've supplied. For example, the GDI may determine that it can double the height and width of a font in its table to produce a font that is closest to the one you've asked for. In fact, the GDI is capable of quite complicated synthesis, including the ability to change the aspect ratio (the height to width ratio) of a font and to produce an italic or bold font from a standard font.

To see a list of names of the fonts available in your system, select the **Fonts** menu item in Paintbrush, as shown in Figure 14-34. You may also find the names of additional fonts by selecting the **Fonts** icon in Control Panel. To use one of the fonts you've identified, simply supply its name as the "*Name*" parameter in `UseFont`. Be sure to spell the name exactly as you see it on the screen.

Oriol for Windows also lets you access fonts you've added with third-party font packages. The names of these fonts may or may not be listed in Paintbrush or the Control Panel. See Chapter 4, "Of Fonts and Printing," for more on where to find font names.

The default font is the System font in black.

If you notice that the color behind each character is not what you intended, you may need to change the background mode or color. See the `UseBackground` command for more details.

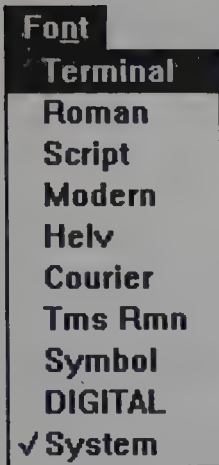


Figure 14-34 Font names in Paintbrush

Examples: The following program shows the effect of explicitly controlling the width and height of a font versus using the default width and height. Figure 14-35 shows the results.

```
{Set the coordinate mode to pixel}
    UseCoordinates (PIXEL)

{Set System font's width to 6 and height to 10 and draw text}
    UseFont ("System", 6, 10, NOBOLD, NOITALIC, NOUNDERLINE, 0, 0, 0)
    DrawText (10, 10, "System font with specified height and width")

{Use the default width and height to draw text}
    UseFont ("System", 0, 0, NOBOLD, NOITALIC, NOUNDERLINE, 0, 0, 0)
    DrawText (10, 30, "System font with default height and width")

    WaitInput ()
```

This next program shows some samples of the standard fonts found on any system. Figure 14-36 shows the window the program produces.

```
UseFont ("Terminal", 0, 0, NOBOLD, NOITALIC, NOUNDERLINE, 0, 0, 0)
DrawText (10, 10, "Terminal")
UseFont ("Roman", 0, 0, NOBOLD, NOITALIC, NOUNDERLINE, 0, 0, 0)
DrawText (10, 18, "Roman")
UseFont ("Script", 0, 0, NOBOLD, NOITALIC, NOUNDERLINE, 0, 0, 0)
DrawText (10, 26, "Script")
UseFont ("Modern", 0, 0, NOBOLD, NOITALIC, NOUNDERLINE, 0, 0, 0)
DrawText (10, 34, "Modern")
UseFont ("Helv", 0, 0, NOBOLD, NOITALIC, NOUNDERLINE, 0, 0, 0)
DrawText (10, 42, "Helv")
UseFont ("Courier", 0, 0, NOBOLD, NOITALIC, NOUNDERLINE, 0, 0, 0)
DrawText (10, 50, "Courier")
```

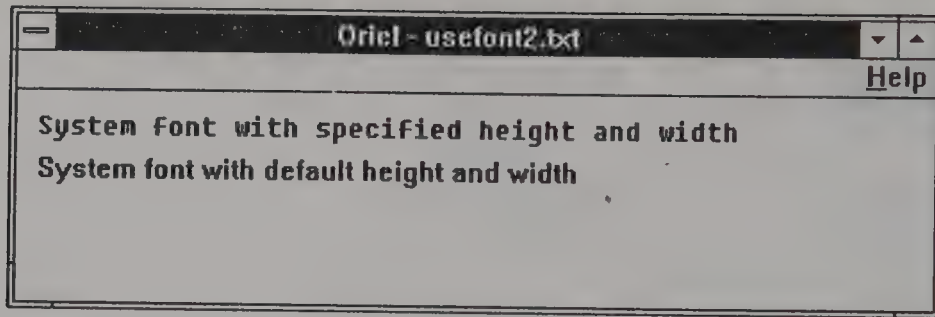


Figure 14-35 Controlling a font's width and height versus using the defaults

```
UseFont("Tms Rmn",0,0,NOBOLD,NOITALIC,NOUNDERLINE,0,0,0)
DrawText(10,58,"Tms Rmn")
UseFont("Symbol",0,0,NOBOLD,NOITALIC,NOUNDERLINE,0,0,0)
DrawText(10,66,"Symbol")
UseFont("DIGITAL",0,0,NOBOLD,NOITALIC,NOUNDERLINE,0,0,0)
DrawText(10,74,"DIGITAL")
UseFont("System",0,0,NOBOLD,NOITALIC,NOUNDERLINE,0,0,0)
DrawText(10,82,"System")
WaitInput()
```

Related Commands: DrawText, UseBackground

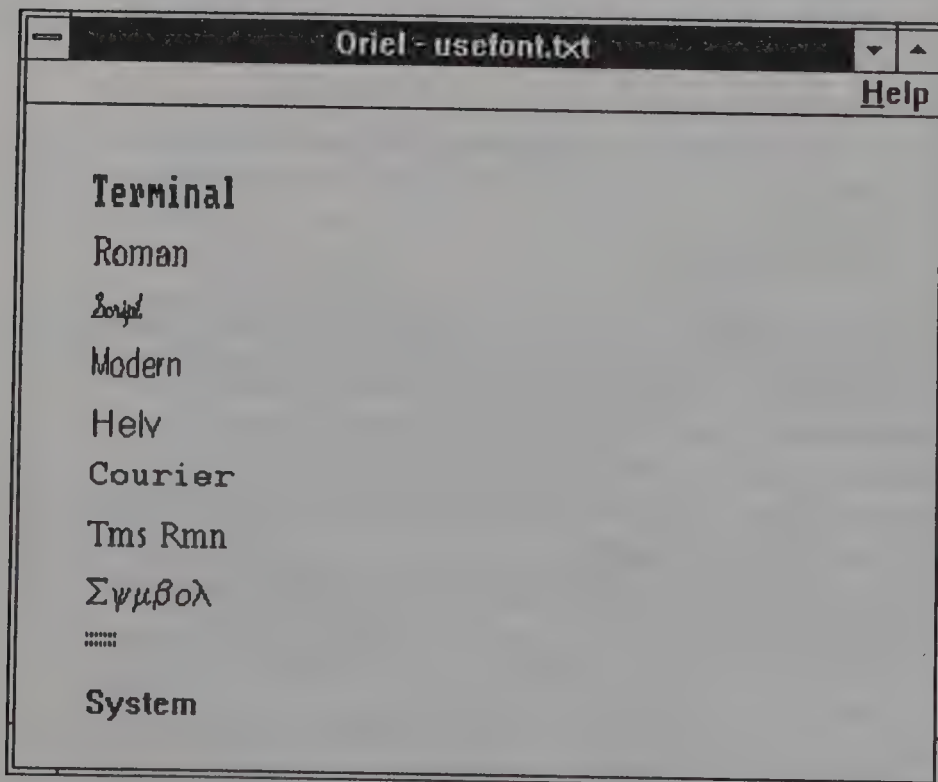


Figure 14-36 The standard fonts found on any system

<i>Token</i>	<i>Description</i>	<i>Result</i>
SOLID	Solid line	_____
DASH	Dashed line	-----
DOT	Dotted line
DASHDOT	Dash-dot line	-. - . - . - . -
DASHDOTDOT	Dash-dot-dot line	-. - . . - . . -
NULL	No line	

Table 14-7 Pen Styles

UsePen

This command establishes the pen that will be used for subsequent drawing operations. It lets you specify the style, width, and color for the pen.

Syntax: `UsePen(Style,Width,r,g,b)`

Parameters:

<i>Style</i>	The style of the pen. It must be one of the tokens in Table 14-7. The default is SOLID.
<i>Width</i>	The width of the line in pixels. It must be 1, unless the pen style is SOLID or NULL. The default width is 1.
<i>r,g,b</i>	Specifies the color of the pen using a combination of red, green, and blue. The default is black (0,0,0).

Example: The following example draws five lines, each using a different pen style and color. Figure 14-37 shows the window that the program produces.

```
{Solid line in black}
UsePen(SOLID,1,0,0,0)
DrawLine(10,10,90,10)

{Dashed line in red}
UsePen(DASH,1,255,0,0)
DrawLine(10,20,90,20)

{Dotted line in green}
UsePen(DOT,1,0,255,0)
DrawLine(10,30,90,30)
```

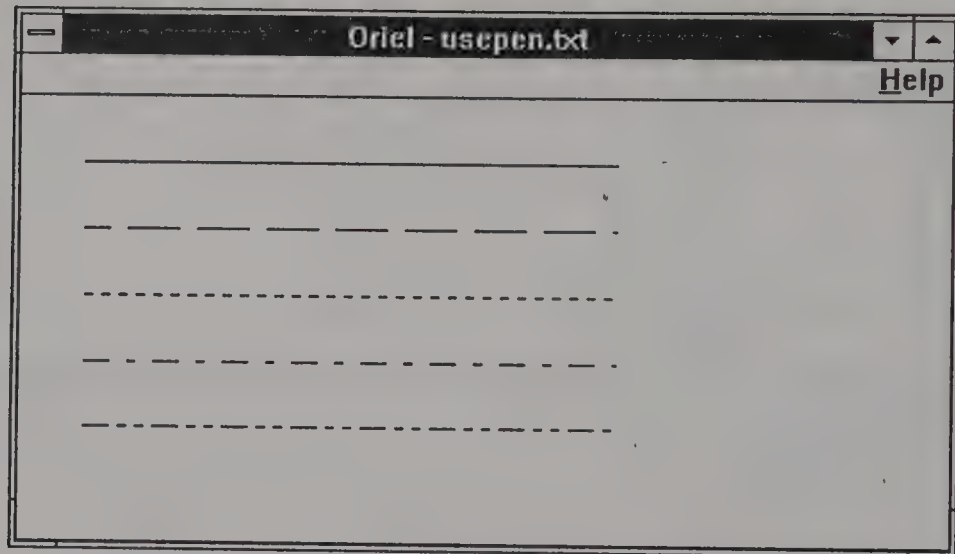



Figure 14-37 The effect of different pens

```
{Dash-dot line in blue}
UsePen(DASHDOT,1,0,0,255)
DrawLine(10,40,90,40)

{Dash-dot-dot line in pink}
UsePen(DASHDOTDOT,1,255,0,255)
DrawLine(10,50,90,50)
WaitInput()
```

WaitInput

This command lets you pause a program a specified number of milliseconds or pause it indefinitely to wait for user input.

Syntax: `WaitInput()`
 or
`WaitInput(milliseconds)`

Parameter:

milliseconds The number of milliseconds you want to pause a program.

Remarks: `WaitInput` *without* an argument pauses a program indefinitely to wait for user input. When the user presses a key, clicks the mouse, or makes a menu selection and a `SetKeyboard`, `SetMouse`, or `SetMenu` command is in effect, Oriol for Windows immediately transfers control to the appropriate label specified in one

of these commands. For example, if a SetMouse command is in effect and the user clicks the mouse within a specified hit-testing region, control transfers to the label associated with that hit-testing region.

WaitInput *with* an argument pauses a program a specified number of milliseconds. For example, WaitInput(1000) pauses Oriel for 1 second, WaitInput(3000) for three seconds, and WaitInput(250) for a quarter of a second.

Because the PC's clock ticks only 16 times a second, the granularity of the WaitInput command is not as fine as a *milliseconds* argument might have you believe. For example, WaitInput(1) has the same effect as WaitInput(62); they both pause your program for approximately 1/16 of a second.

If you use SetWaitMode(FOCUS) intending to pause an Oriel program until it gets the focus back, you must use a *milliseconds* argument with WaitInput, for example, WaitInput(1). If you do not use a *milliseconds* argument, your program will not pause (see the SetWaitMode command for more details).

WaitInput(0) is treated the same as WaitInput(1).

As a general rule, you should always use WaitInput() when you are not performing work in the Oriel window. This gives more of the system's resources to other Windows programs at a time when your Oriel program does not need them.

Example: The following program displays a message on the screen, pauses for 2 seconds, places another message on the top of the previous one, and then pauses indefinitely until you close the window.

```
DrawText(10,10,"Oriel will pause for 2 seconds")
WaitInput(2000)
DrawText(10,10,
    "Oriel will now pause indefinitely until you close the window")
WaitInput()
```

See the SetMenu, SetMouse, and SetKeyboard commands for other examples of WaitInput.

Related Commands: SetWaitMode, SetMenu, SetKeyboard, SetMouse

QUICK REFERENCE

Beep

DrawArc(x1,y1,x2,y2,x3,y3,x4,y4)

DrawBackground

DrawBitmap(x,y,"Filename")

DrawChord(x1,y1,x2,y2,x3,y3,x4,y4)

```

DrawEllipse(x1,y1,x2,y2)

DrawFlood(x,y,r,g,b)

DrawLine(x1,y1,x2,y2)

DrawNumber(x,y,n)

DrawPie(x1,y1,x2,y2,x3,y3,x4,y4)

DrawRectangle(x1,y1,x2,y2)

DrawRoundRectangle(x1,y1,x2,y2,x3,y3)

DrawSizedBitmap(x1,y1,x2,y2,"Filename"),

DrawText(x,y,"Text")

End

Gosub label / Return

Goto label

If <condition> Then <commands>
    where <condition> uses <, >, <=, >=, <>, or =

MessageBox(OK/OKCANCEL/YESNO/YESNOCANCEL,Default_button,
            INFORMATION/EXCLAMATION/QUESTION/STOP/NOICON,
            "Text","Caption",Button_pushed)

Run("Command_line")

Set Variable = 0,1,2,3,...,65535
Set Variable = <math_expression>
    where <math_expression> uses +, -, *, or /

SetKeyboard()
SetKeyboard("a",label,
            "^a",label,
            vkey,label)

SetMenu()
SetMenu("Top1",IGNORE/label,
        "ItemA",IGNORE/label,
        "ItemB",IGNORE/label,
        SEPARATOR,
        "ItemC",IGNORE/label,
        ENDPOPUP,
        "Top2",IGNORE/label,
        .
        .
        ENDPOPUP)

```

```

SetMouse()
SetMouse(region1_x1,region1_y1,region1_x2,region1_y2,label,x,y,
        region2_x1,region2_y1,region2_x2,region2_y2,label,x,y,
        .
        .
        regionn_x1,regionn_y1,regionn_x2,regionn_y2,label,x,y)

SetWaitMode(NULL/FOCUS)

SetWindow(MAXIMIZE/MINIMIZE/RESTORE)

UseBackground(OPAQUE/TRANSPARENT,r,g,b)

UseBrush(SOLID/DIAGONALUP/DIAGONALDOWN/DIAGONALCROSS/
        HORIZONTAL/VERTICAL/CROSS/NULL,r,g,b)

UseCaption("Text")

UseCoordinates(PIXEL/METRIC)

UseFont("Name",Width,Height,BOLD/NOBOLD,ITALIC/NOITALIC,
        UNDERLINE/NOUNDERLINE,r,g,b)

UsePen(SOLID/NULL/DASH/DOT/DASHDOT/DASHDOTDOT,Width,r,g,b)

WaitInput()
WaitInput(milliseconds)

```