Wordup Graphics Toolkit Version 5.1 for OS/2 Warp

Programmers Guide

Last Revised July 1997

First Printing, May 1995 (c) Egerter Software
Seconding Printing, January 1996 (c) Egerter Software
OS/2 Warp Version, September 1996 (c) PolyEx Software

WordUp Graphics Toolkit 5.1
Adobe PDF Version June 1997

Contacting the authors:

PolyEx Software	Tech Support:
6815 14th Street West Suite 110	Send your customer ID and problem to:
Bradenton, Florida	polyex@netsrq.com
USA 34243	We can also accept uuencoded binaries!

For the latest WGT software (patches, demos, tutorials, etc):

WWW: http://www.polyex.com

Internet FTP sites and directories:

x2ftp.oulu.fi	pub/msdos/programming/wgt

What is WGT?

The WordUp Graphics Toolkit is a library of routines that are callable from C language under the OS/2 Warp 3.0 and higher operating system. This version of WGT is for Watcom C/C++ v10.0. The library is compiled for the flat memory model, 32 bit, OS/2 PM or OS/2 full screen application.

WGT contains over 250 routines for graphics, input and output devices, and more. While other graphics libraries provide you with the most basic graphics commands, WGT offers many complex routines, including a sprite system, tiled background scrolling, a fully operational file selector, an animation player, a 3D rendering library, and image shading routines. This means you can spend more time on programming your application and less time working on the low level routines required.

The WordUp Graphics Toolkit gives you more than just a library of routines. The WGT Sprite Editor and WGT Map Maker are two utilities which are an integral part of the development process.

The WGT Sprite Editor is a fully featured paint program for managing sprites, fonts, mouse cursors, and background screens. It uses a graphic user interface and a mouse, allowing you to design images quickly and effortlessly. It is the most essential tool in creating graphics applications, yet other graphics libraries fail to provide a utility such as this. This is a native OS/2 Application for FullScreen OS/2 sessions.

The WGT Map Maker is needed to create background maps for use with the tiled background scrolling library. It uses a windowing environment to keep your tiles, sprites, and maps organized. It also supports SVGA
(If your video card supports PMI) viewing modes, tile templates, sprite placement, and even generates source code for your particular maps.

1.1 Installing WGT

Insert DISK 1 into your floppy drive. At the DOS prompt, type the letter for your floppy drive (usually A or B) followed by a colon, then press ENTER. Type INSTALL, and the installation program will begin.

This program will copy all or part of the toolkit to your hard drive. It will create a directory called \POLYEX\WGT4OS2\ in the root directory. Each part of the toolkit will have a subdirectory in the WGT4OS2 directory.

Follow on screen the prompts to complete the installation.

1.2 Compiling the Examples

The examples are found in the WGT4OS2\EXAMPLES directory. A special batch file called make is in this directory which will compile any C file you pass it. The examples are not compiled so you must enter the following at the DOS prompt (in the example directory):

```
make WGTxx
```

xx ranges from 01 to 70. A similar make batch file is included. The 3D rendering example is contained in the WGT4OS2\3D directory and can be compiled with the makecam batch file.

1.3 Compiling your own programs

When compiling your own programs, you have several choices of how you can link the WGT libraries.

1. You can create a batch file similar to the examples. For every program you compile you will need a different batch file with the correct directories.

2. You can create a make file or let the OS/2 IDE create one for you.

3. You can set up the compiler so it looks in the WGT4OS2 directory for additional header files and libraries. This is the recommended method for compiling, and you should do the following immediately after installing the software:

The compiler also needs to know where the header files are. You can do this by changing the INCLUDE environment variable from your autoexec.bat file. An example of this line:

```
SET INCLUDE=e:\watcom\h;d:\wgt5\h
```

Now the compiler can find both the libraries and header files, which means you can compile and link a program using the following command from the DOS prompt:

```
wcl386 program.c wgt5_wc.lib
```

This command will work from any directory.

WGT for OS/2 Source Code purchases may be made send e-mail to
polyex@netsrq.com for more information.

1.4 Compiling the source code

If you have purchased the source code for WGT, you can build all the libraries with a single command. Change to the WGT5\MAKE directory and type:

```
wmake -f makefile
```

You will first need to modify the first 4 lines of makefile which contain the path where you installed WGT. Once this command is run, it will compile each source file in WGT4OS2\CODE and create new libraries in WGT4OS2\LIB.

2.0 Initializing and Restoring the Video Mode

WGT operates with either a virtual buffer (usually for DIVE or PM) and a video mode called mode 13h (for fullscreen OS/2 applications). Mode 13h has screen dimensions of 320x200 pixels, and has 256 colors on the screen at once. The virtual buffer has no dimension limits, but you are limited to 256 colors. Only one video page is available in mode 13h , so WGT emulates page flipping through the use of virtual screens stored in memory.

The vga256 command is used to initialize the WGT library, you use this command to assign a pointer (with virtual buffer) or get a pointer from your default screen.
It is also at this point where you tell WGT whether this application is PM/DIVE or FullScreen. If you are in fullscreen will probably want to restore the original video mode after your program ends, and WGT provides two routines to do this called wgetmode and wsetmode.

vga256	- Initializes WGT
wgetmode	- Returns the current video mode (fullscreen only)
wsetmode	- Sets the video mode (fullscreen only)

Here is a simple program to save the video mode, enter WGT's default video mode, and restore the original mode:

```
#include <wgt5.h>

short oldmode;
char *videoBuffer; /* Our Default Drawing Buffer */
void main (void)
{
 oldmode = wgetmode ();
 vga256 (videoBuffer,320,200, WGTDRAWMODE_FSVGA);

 wsetmode (oldmode)
]
```

When using a virtual buffer you would allocate your buffer
which must have a color depth of 8 bits (256 colors) then tell
WGT the buffer by passing it, as you can see from the below
code snippet;

.... earlier in the program we get a pointer to a buffer that DIVE is going to use to blit. The pointer to this buffer is called diveBuffer. Its dimensions are 640x480.

We call vga256 with:

```
vga256(diveBuffer,640,480,WGTDRAWMODE_PM);
```

You may want to refer to the DIVE and FullScreen examples for better understanding

3.0 Introduction to the VGA Hardware and FullScreen OS/2

Before any attempts can be made to write a graphics program or a game, the programmer should be familiar with the hardware for which he/she is developing software. In the case of the graphics programmer, the video system is the primary hardware to be concerned with. The VGA (Video Graphics Array) is a standard in the world of PC-compatibles (switching rapidly to Super-VGA). An attractive feature of this type of video system is the ability to display 256 colors at a time rather than the 16 or fewer which were possible on CGA and EGA systems of earlier times. The arcade-game industry quickly adopted one particular graphics mode as the new standard. This mode has a horizontal resolution of 320 pixels, a vertical resolution of 200 pixels, and a set of 256 colors to use for drawing. This is mode 13h (hex) when indexed by a programmer.

With 320*200 pixels representing a full screen, this means that at any given time, the monitor is updating 64000 pixels. It does this at about 70hz (70 cycles a second). Each pixel can be one of 256 colors. Eight bits of information are required to achieve 256 unique numbers, so each pixel uses one byte of memory. That's a total of 64000 bytes for one screen of data.

A very handy feature of the IBM's video system is that the video RAM is "mapped" into the addressable memory space. At segment A000 (the 10th segment) the VGA maps in 64k of video RAM. Using the refresh rate of the monitor (70Hz) this section of memory is scanned and drawn several times a second. In order to change what you see on the screen, simply change a few bytes in this 64k block of memory, and the VGA hardware and update the screen for you. Using real-mode, the segment starts at 0xA000, and the offset begins at 0x0000. By manipulating the offset alone you have access to the full range of pixels on a 320*200*256 screen. Data is stored sequentially, a row at a time. A suitable formula for addressing one individual pixel is:

offset of (x,y) = (y*320) + x

where the x and y values range from 0-319 and 0-199 respectively. A position of (0,0) indicates the top-left screen corner, and (319,199) indicates the lower-right corner.

Upon calling the vga256 command in a fullscreen WGT OS/2 program, a pointer to the visual page at A0000h (flat model) is created, called abuf. This pointer will always point to the location of the page we are writing to. You may use the formula above to access pixels within the video page. For example: abuf[y*320 + x] would access the pixel at (x,y) on the current video page. Wordup Graphics Toolkit does this by wrapping the vio calls that are sometimes difficult to find documentation on.

4.0 Color

The reason for the popularity of this video mode is the wide range of colors which may be represented. From a total of 262,144 colors, 256 may be displayed at the same time. Each color is made up of 3 primary colors (red, green, and blue) in varying levels of brightness. The programmer may select each of the 256 color indices and set the RGB levels (red/green/blue) to reflect the color which is to be represented. The VGA card uses 6 bits to determine RGB levels. This gives 64 levels for each primary color, and a total of 64*64*64 (262,144) possible outcomes. A level of 0 indicates no color, a level of 63 is full brightness.

For example, an RGB value of (0,0,0) results in BLACK (no red, no green,and no blue). Increasing each level by one would produce shades of gray until we reach (63,63,63) which is WHITE. A value of (63,0,0) is pure red, a value of (0,63,0) is pure green, etc.

This range of possibilities may seem to be more than enough for most needs, but you must remember that you only have the ability to display 256 colors at any given time. Careful planning must be made to ensure that all images may be accurately displayed when onscreen. If you attempt to show two pictures with two different palettes, you will have to show them one at a time to be able to use the right colors. Some computer artists will design a palette which has enough unique colors to draw any image reasonably close to the desired result, but some quality must be sacrificed. WGT supplies routines to set palettes, load and save palettes, cycle through ranges of colors, fade palettes in and out (to black, or between each other), and redraw images using a palette which is different than the original. This last feature is known as remapping an image.

4.1 Image Remapping

Let's assume you are displaying a rainbow in one region of the screen, and would like to show a grayscale image of a person's face on another region. The problem is, the palette you are using to show the rainbow doesn't match the one used to draw the face. WGT can analyze the two palettes and find colors in the rainbow which are reasonably close to (or exactly the same as) the one used for the face. It will then change the image of the face so that it uses the colors in the rainbow palette which suit it best. The end result of this operation is the ability to show both the colorful rainbow and the grayscale face at the same time, using the same set of 256 colors. Doing this takes considerable amounts of time however, and should not be done realtime during a program. Where possible, make all images you will be loading use the same palette. Planning ahead of time will save you a lot of grief later on.

WGT also offers realtime image remapping through shade tables. This lets you perform special lighting effects such as darkness, fog, shadows, translucency, and Gouraud texture mapping. See the shade table chapter for more information.

5.0 Video Pages

The simplicity of the video system in this mode has provided a lot of software authors with the ability to write their own video routines. We have done this for you, and have specialized in this graphics mode alone to provide the absolute fastest, most-powerful and flexible set of routines available. Since only 64k of the VGA card (256k standard) is accessible in mode 13h, a technique known as page flipping is not possible (unless the VGA is "tweaked", which we will not discuss here). Page flipping involves the creation of an image on a hidden screen, which is shown immediately upon completion of the image. By drawing on the other screen again and then flipping back, smooth animation may be achieved as far as the eye is concerned. Mode 13h always displays the same 64k of video RAM, so we do not have the ability to page flip. WGT solves this problem by allowing us to use conventional memory as if it were video RAM. By allocating blocks of memory and telling WGT to write to them instead, we can "draw" on a hidden page. The problem occurs when we want to see what has been drawn. To do this, we must physically copy the memory into the 64k video buffer.

WGT contains routines which will allow you to specify the region to copy, and which screens to use as source and destination. By copying as little information as necessary to update the visual screen, enough speed may be obtained to simulate page flipping. The faster video cards (16-bit, 32-bit and 64-bit) and CPUs also help avoid flicker due to slow memory copies. An 8-bit card is extremely slow and cannot achieve very good frame rates using this technique, but a good VESA system or PCI card can easily achieve between 70 and 200 frames per second. A fast BUS or CPU makes this process even more attractive.

The OS/2 version of WGT allows for any block of memory to be used as a virtual screen. It does not have to be a 320x200 block. WGT has a general video setup structure called WGT_SYS which is used to keep track of the current virtual screen width and height, among other things.

```
struct {
        short   xres;
        short   yres;
        short   videomode;
        int     videobanksize;
        short   (*bankswitch)(short);
        short   screenwidth;
        short   screenheight;
} WGT_SYS;
```

xres and yres contain the width and height of the current virtual screen.

6.0 Graphics Primitives

Every graphics library contains a set of routines to perform the most basic drawing operations available. WGT has an extensive collection of graphics primitives to make life easy for the programmer. These range from plotting pixels to drawing texture mapped polygons. Most primitives are drawn using a color index which has been previously set. The following is a list of the fundamental graphics operations which WGT provides for you:

wbar	- Draws a solid rectangle
wbezier	- Calculates the points on a curve
wbutt	- Draws a shaded button
wcircle	- Draws a circle using center and radius
wclip	- Sets the clipping area
wcls	- Clears the entire video page with a color
wdeinitpoly	- Frees the polygon scan conversion buffers
wdeinit_triangle_renderer	- Frees the triangle renderer buffers
wdraw_scanpoly	- Draws a scan-converted concave polygon
wellipse	- Draws a hollow ellipse with center and radii
wfastputpixel	- Puts a single pixel on screen - no clipping
wfill_circle	- Draws a filled circle with center and radius
wfill_ellipse	- Draws a filled ellipse with center and radii
wfline	- Draws a fast line with no clipping
wfree_scanpoly	- Frees memory from a scan-converted polygon
wgetpixel	- Returns color of a pixel at a point
wgouraudpoly	- Draws a gouraud-shaded polygon
whline	- Draws a horizontal line
whollowpoly	- Draws a hollow polygon
winitpoly	- Allocates polygon scan conversion buffers
winit_triangle_renderer	- Allocates triangle rendering buffers
wline	- Draws a line
wputpixel	- Sets the color of a pixel at a point
wrectangle	- Draws a hollow rectangle
wregionfill	- Fills a region with the current color
wscan_convertpoly	- Creates a scan-converted polygon in memory
wsetcolor	- Set color index to use for primitives
wsolidpoly	- Draws a filled polygon
wstyleline	- Draws a line using a pattern
wtexturedpoly	- Draws a texture-mapped polygon
wtriangle_solid	- Draws a solid triangle
wtriangle_gouraud	- Draws a Gouraud shaded triangle
wtriangle_texture	- Draws a textured triangle
wtriangle_flat_shaded_texture	- Draws a flat shaded textured triangle
wtriangle_gouraud_shaded_texture	- Draws a Gouraud shaded textured triangle
wtriangle_translucent_gouraud	- Draws a translucent Gouraud shaded triangle
wtriangle_translucent_texture	- Draws a translucent textured triangle
wxorbox	- Draws a filled box using XOR mode

These functions are described in detail within the library reference.

7.0 Displaying Text

WGT provides a set of routines to produce text input/output on the graphics screen. These routines often take fonts (style of characters) as parameters. In such a low resolution video mode, text often looks quite "chunky", and detail may not be achieved. Nevertheless, a creative font can produce the desired effect if drawn correctly. The font files used in WGT are created using the WGT Sprite Editor. If no fonts are loaded, a default font is used. This font is called SYSTEM.WFN, and must reside in the same directory as your executable in all WGT for OS/2 programs.The following text routines available:

wflashcursor	- Flashes the simulated text cursor once
wfreefont	- Frees memory from a previously loaded font
wgettextheight	- Returns the height of the tallest letter in a string
wgettextwidth	- Returns the width of the entire string
wgtprintf	- Same functionality as printf for text modes
wloadfont	- Allocates memory and loads custom font
wouttextxy	- Displays a string at screen coordinates
wsetcursor	- Sets the height of the text cursor
wstring	- Inputs a string using the default font
wtextbackground	- Sets the background text color
wtextcolor	- Sets the foreground text color
wtextgrid	- Turns the text grid on or off
wtexttransparent	- Turns the text foreground or background on or off

7.1 Custom Fonts

WGT enables you to create your own fonts with the WGT Sprite Editor. Only the first 128 characters of the ASCII set are used. Each character can be any size and the characters will be proportionally spaced when displayed on the screen.

The system font has been saved in a sprite file called "system.spr". When you want to create a new font, copy "system.spr" to a new name, and modify the new file. The file has all characters in normal text, so you know which sprites represent each character.

If you don't plan on using certain characters in your program, simply leave them as they are. After you have created your font, you can convert it into WGT's font file format from within the WGT Sprite Editor. You can then load the font in your program using the wloadfont command.

To use the new font, declare a font variable such as:
```
        wgtfont sans_serif;
```

Then load it into memory using wloadfont:
```
        sans_serif = wloadfont ("newfont.wfn");
```

To display text using the new font, pass the font to the wouttextxy
or wgtprintf command:
```
        wgtprintf (10, 10, sans_serif, "This is the new font");
```

After you are finished using the custom font, release it from memory
with the wfreefont command:
```
    wfreefont (sans_serif);
```

You can have as many fonts as you like loaded into memory at once.

The fonts used with the commands described above are single color fonts only. Their color is set with the wtextcolor command. To create a font with more than one color, store your letters in a sprite file in a similar manner. Instead of using the text commands in WGT, you could use the wputblock command to show each letter individually.

The code below is an example of this technique:

```
void displaystring (char *string, int x, int y, int mode)
{
 short len;
 short i;
 char ch;

 len = strlen (string);

 for (i = 0; i < len; i++)
  {
   ch = string[i];
   wputblock (x,y, sprites[ch], mode);
   x += wgetblockwidth (sprites[ch]);
  }
}
```

This code will start displaying a string at (x,y) by using wputblock to display each sprite. The code is based on a previously loaded global array called sprites. The sprite is based on the corresponding ASCII character in the string. The x coordinate is increased by the width of the sprite in pixels. Another twist to this routine would be to center the string on the screen by scanning and finding the total width in pixels before drawing the sprites. Divide this total by 2 and subtract it from the passed x position to calculate the new x position. The FOR loop (in the above code) can then be performed exactly the same way to display the string which will now be centered about (x,y).

8.0 Images in WGT

The most commonly used features of WGT are the bitmap functions. These functions manipulate rectangular regions of image data to perform various effects. Bitmaps in WGT are referred to as BLOCKS.

8.1 What is a BLOCK?

A block is a sequential chunk of data which represents an image. It is merely a series of numbers ranging from 0-255 indicating the color of the pixels at each point. If, for example, you have a circle drawn on the screen, and you wish to preserve a copy of the image in memory, you can call a routine which grabs the image from the screen and stores a copy in a pointer. The pointer is given a name (by you, the programmer) to be used to reference that image at a later time. The pointer points to the image data and the block header which is as follows:

16 bits (1 word)	- width of image in pixels
16 bits (1 word)	- height of image in pixels
width*height bytes	- image data, row by row

This format is not required knowledge for most users, but is handy to know for those who want to create their own bitmap-manipulation routines.

To show the simplicity of the bitmap system, here is a short WGT example which will draw a box on the screen and save a copy in a block. The block will then be used to create a duplicate of the image in the upper-left corner of the screen.

```
#include <wgt5.h>

block my_image;

void main (void)
{
        short oldmode;
          char *videoBuffer;
        oldmode = wgetmode ();                    /* preserve initial video mode */
        vga256 (320,200,videoBuffer,WGT_DRAWMODEFSVGA,);/* initialize WGT system and
graphics mode */
        wsetcolor (15);                           /* draw with the 16th color entry */
        wbox (100, 100, 115, 130);                        /* draw a box from (100,100) to (115,130)
*/
        my_image = wnewblock (100, 100, 115, 130);        /* Grab the image from the screen
                                                    and store it in a block */
        wputblock (10, 10, my_image, NORMAL);             /* Paste a copy of the block on
                                                   the screen at (10,10) */
        wfreeblock (my_image);                    /* Free memory used by block */
        getch();                                  /* Wait for a keypress */
        wsetmode (oldmode);                       /* Restore old video mode */
}
```

8.2 Block Copy Modes

There are two basic ways to display a block. The fastest way is to select the NORMAL technique (see programming example for 8.1). This technique simply copies out the rectangular region saved in the pointer exactly the way it is stored. Quite often, however, programmers do not desire a perfectly rectangular image, and would prefer to display images with some colors treated as transparent. WGT allows you to select XRAY mode to indicate that some parts of the image are not to be displayed. In WGT, any pixel which is color index 0 will be treated as transparent in this block mode. If you were to display an image of a human figure, and all parts of the rectangular region which aren't included as the figure are drawn using index 0, they will be skipped when drawing the image. This technique is slower because WGT must check each pixel before drawing it, but it provides a very essential mode for displaying images, primarily used for sprites.

It should be noted that quite often an image which is primarily composed of pixels using index 0 can be drawn FASTER than the same image drawn in NORMAL mode. If the image is large and the majority of "transparent" pixels are contained in sequential data, the image will be blasted to the screen extremely quickly. Optimized logic within the assembly code has been used to perform this action very efficiently. Since this is true, a game consisting of a large number of sprites and moving objects can achieve very good frame rates and take advantage of the extra processing time for more complex routines. The scrolling library uses this same approach to obtain fast frame rates when displaying parallax layers for 3D effects. The more "empty" space in an image, the faster it will be reproduced on the screen.

9.0 Graphic Files

WGT offers support for several different graphics file formats. WGT can load and/or save PCX, CEL, BMP, LBM/IFF, BLK, and PAK file formats. The last two are custom formats only used by WGT.

PCX files are created with many shareware and commercial drawing programs. The image can be any size. A palette is also stored in the file. Most PC users have access to a paint program which supports this format if they are running MS Windows. PC Paintbrush uses this format as well as BMP to create 16 color images. Since this is a compressed image file format, data can be saved to disk using very little capacity, and the image can be decompressed very quickly to memory when loaded.

CEL files must be in Autodesk Animator 1.0 format. This is an uncompressed format, which includes a palette.

BMP files are commonly used in Microsoft Windows, for wallpaper and icons. Any resolution is supported, and WGT will convert the image into 256 color mode when loading. A palette is also stored in the file. Image compression is not used in the standard BMP format, so images require a lot of space on disk. It is not recommended that this format be used unless you require compatibility with Windows.

LBM / IFF files are created by Deluxe Paint. This was originally an Amiga picture format, and a great number of artists still use the machines to produce images and animation sequences. Support is provided for loading, but not for saving these formats.

BLK files are the simplest format, and store the image in an uncompressed form. This is the format used internally by WGT. Any image format discussed here is converted to BLK format when loaded. It makes displaying and manipulating data very fast and efficient compared to a compressed format, with the expense of a loss of memory.

PAK files use a run-length encoding compression scheme similar to the PCX format. It is a custom format provided only by WGT. This format is good for storing images when you do not wish the general public to be able to load your files. The specifications for this format have not been published and therefore provide a fair amount of image security.

10.0 Input Devices
WGT provides many functions for handling the keyboard, mouse and Joystick in OS/2

10.1 WGT Keyboard Handler

The WGT keyboard handler installs a device monitor thread which intercepts all keypresses. This is used to enable multiple keypresses at once and eliminates the problems with the keyboard buffer filling up, and the keyboard repeat rate. A global array called kbdon contains 128 short integers, each representing the state of a single key on the keyboard. The integer contains either 0 or 1. 1 means the key is being pressed. It will return to 0 as soon as the key is released.

There are two commands which control the custom monitor thread. installkbd replaces the current keyboard interrupt with its own. uninstallkbd shuts down the monitor, freeing resources from the thread.

When the handler is installed, normal commands that read the keyboard such as getch or scanf will not function because keypresses are not placed in the keyboard buffer. To determine if a key is pressed, you must look at the kbdon array. Each element in the array corresponds with the scan code returned from the keyboard when that key is pressed. To find these scan codes, use the "scancode.exe" program. This program will display the scan code of the key you are pressing.

In addition, a flag is increased every time a key is pressed. This flag is called keypressed. To check if any key was pressed, set the flag to 0 and wait until it increases.

10.3 Mouse Routines

Before you use any mouse routines, you must first install WGT's custom mouse monitor. This thread will be called whenever the mouse is moved or a button is pressed. It updates three global variables:

mouse.mx	- X coordinate of mouse cursor
mouse.my	- Y coordinate of mouse cursor
mouse.but	- mouse button state

Since these value may change at any time, it is wise to store their values into some temporary variables. If you do not store the value, your program may act incorrectly as the coordinates of the mouse may change halfway through a procedure which expected them to be constant.

The following is a list of the mouse commands in WGT:

mdeinit	- Removes the custom mouse interrupt
minit	- Installs the mouse interrupt and initializes mouse
moff	- Hides the mouse cursor
mon	- Shows the mouse cursor
mouseshape	- Change the shape and hotspot of the mouse
msetbounds	- Sets the boundaries of the mouse cursor
msetspeed	- Changes the mouse sensitivity
msetthreshhold	- Changes the mouse doubling threshhold
msetxy	- Sets the coordinates of the mouse cursor
noclick	- Loops until all mouse buttons are released

10.4 Joystick

The joystick routines allow up to two joysticks to be connected at once. Before using the routines, the IBM joystick driver must be installed

wcalibratejoystick	- Determines the joysticks range
wcheckjoystick	- Sees if the joystick is connected
winitjoystick	- Initializes the joystick
wreadjoystick	- Reads values from the joystick

Here's a short demonstration of the joystick routines and their proper use:

```
#include <stdlib.h>
#include <conio.h>
#include <wgt5.h>
#include <wgtjoy.h>

void main (void)
{
        joystick joya, joyb;            /* Structure for joysticks A & B */
        int display_row;        /* Row for status display */
        int result;                     /* Result of joystick detection */
        int joya_found = 0;             /* 1 if joystick is found */
        int joyb_found = 0;             /* 1 if joystick is found */

        if (!( result = wcheckjoystick () ))
        {
          printf ("Joysticks not found. Program aborted.\n");
          exit (0);
        }
        if (result & 1)
        {
          joya_found = 1;                          /* It's there */
          printf ("Joystick A detected.\n\n");
          winitjoystick (&joya, 0);                /* Initialize it */
          printf ("Calibrate joystick A by swirling it and then pressing ENTER.\n");
          wcalibratejoystick (&joya);
          printf ("Joystick A calibrated.\n\n\n");
        }
        if (result & 2)
        {
          joyb_found = 1;                          /* It's there */
          printf ("Joystick B detected.\n\n");
          winitjoystick (&joyb, 1);                /* Initialize it */
          printf ("Calibrate joystick B by swirling it and then pressing ENTER.\n");
          wcalibratejoystick (&joyb);
          printf ("Joystick B calibrated.\n\n\n");
        }
        printf ("Now reading joystick values. Press any key to end program.\n\n");
        joya.scale = 2000;
        joyb.scale = 2000;
        while (!kbhit ())
        {
          if (joya_found) {
           wreadjoystick (&joya);                  /* Read values into structures */
           printf ("Joystick A      x: %5d    y: %5d    Buttons: %5d\n", joya.x, joya.y, joya.buttons);
          }
          if (joyb_found) {
           wreadjoystick (&joyb);
           printf ("Joystick B      x: %5d    y: %5d    Buttons: %5d\n", joyb.x, joyb.y, joyb.buttons);
          }
        }
        getch ();
}
```

11.0 Sprite Library

The sprite Library can be an integral part of any games developers toolkit

11.1 What is a SPRITE?

A 'sprite' is a term used when dealing with moving objects that do not destroy the background image behind them. They are used in video games with backgrounds that do not move. WGT includes a command called wloadsprites, which loads a sprite file created with the WGT Sprite Editor. The file is actually just a series of pictures, which is loaded into an array of blocks. These pictures are usually for sprites, but can also be for fonts, pictures, or any other image data. WGT contains two sprite engines. One for stationary backgrounds, and one for scrolling backgrounds. A non-scrolling background can be faster because only the parts of the screen that contain sprites have to be updated.

To produce animation, a series of blocks are displayed and moved on the visual screen, much like the frames of a cartoon sequence. The artist draws each block as an individual frame of the animation, and then plays them back in a specific order to simulate some sort of action. WGT provides functions to describe this animation sequence, including the blocks used, the timing between frames, and the movement of the object. To simplify the naming of the sprites, an array of blocks is used to reference them. This array is passed to the sprite functions, and the function uses the index to access each image.

For example, if the user declares block my_sprites[1000]; in their program, they have created an array of pointers to the blocks which will be used for animation. A single function will load the images from a sprite file (designed with our Sprite Editor utility) and store them in the array. Let's assume you want to simulate a human running. Each block in the sprite file will contain one frame of the animation sequence. If 15 images are used, they will all be stored in the same file. One call to wloadsprites will load the image data in the indices 0-14 of the array.

11.2 The Sprite Library

The sprite library allows you to load sprites created with the WGT Sprite Editor, and display them on the screen. The best part about the sprites is you can set up movement and animation sequences and the library will handle everything for you. This library is used for animations over a static background only. You may NOT combine this library with the WGT scrolling library. The scrolling library has its own routines for displaying sprites overtop a moving background. You should decide which library your program will need depending on what type of game it is.

The sprite library uses the following variables:

```
block backgroundscreen;        - Holds the constant background
block spritescreen;                    - Work buffer (same image as backgroundscreen)
short maxsprite;               - The highest sprite number used in your program
                                         Initialized to 100 by initialize_sprites
```

A large structure contains all data about the sprites:

```
typedef struct
{
  unsigned char num;                    /* Sprite number shown */
  short x, y;                           /* Coordinates on screen */
  unsigned char on;                     /* On/Off, for visibility */

  int ox, oy, ox2, oy2;
  signed char animon;                                    /* Animation on/off */
  short animation_images[MAX_ANIMATION];        /* Animation numbers */
  unsigned char animation_speeds[MAX_ANIMATION];        /* Animation speeds */
  signed char current_animation;                 /* Current animation counter */
  unsigned char animation_count;                 /* Delay count for animation */

  signed char movex_on;                          /* X movement on/off */
  short movex_distance[MAX_MOVE];                        /* X distance per frame */
  short movex_number[MAX_MOVE];                          /* Number of times to move */
  unsigned char movex_speed[MAX_MOVE];                   /* Delay between each movement */
  signed char current_movex;                             /* Movement index */
  short current_movex_number;                            /* Number of times moved */
  unsigned char movex_count;                             /* Delay count for X movement */

  signed char movey_on;                          /* Y movement on/off */
  short movey_distance[MAX_MOVE];                        /* Y distance per frame */
  short movey_number[MAX_MOVE];                          /* Number of times to move */
  unsigned char movey_speed[MAX_MOVE];                   /* Delay between each movement */
  signed char current_movey;                             /* Movement index */
  short current_movey_number;                            /* Number of times moved */
  unsigned char movey_count;                             /* Delay count for Y movement */
 } sprite_object;

sprite_object s[MAX_SPRITES];
```

The following defines should be altered to suit your program needs:

```
#define MAX_SPRITES             100
#define MAX_ANIMATION           40
#define MAX_MOVE                15
```

The source code for this library is included, so you can adjust the default values to suit your program.

If you need to store the movements or animations as integers, it will be easier to put them directly into the arrays yourself.

If you draw something other than a sprite on the virtual screen, you will need to copy more of the screen to the visual page. Therefore change the minx,miny,maxx,maxy variables of a sprite that is on to the coordinates of the area you need to copy.

The following commands are available in the sprite library:

animate	- Define the animation of a sprite
animoff	- Turn off the animation of a sprite
animon	- Turn on the animation of a sprite
deinitialize_sprites	- Shut down the sprite engine
draw_sprites	- Update animation and movement, and draw sprites
erase_sprites	- Erase the sprites from the work screen
initialize_sprites	- Start the sprite engine
movex	- Define horizontal movement of a sprite
movexoff	- Turn off horizontal movement of a sprite
movexon	- Turn on horizontal movement of a sprite
movey	- Define vertical movement of a sprite
moveyoff	- Turn off vertical movement of a sprite
moveyon	- Turn on vertical movement of a sprite
overlap	- Return 1 if two sprites are touching
spriteon	- Turn on a sprite
spriteoff	- Turn off a sprite

12.0 Multidirectional Scrolling

A large part of the WGT toolkit deals with creating and using a scrolling background for games. When dealing with a large world like this, it is obvious we cannot use large bitmaps due to the memory limitations of the PC. To make a very large background picture while keep the memory used to a minimum, we need to use a tiling approach.

12.1 Tiling

Tiling is done by creating a number of small reusable images that can be placed together to form a larger background picture. Some simple examples are a picture of some grass, dirt, or rocks.
With these tiles, you can place them together to form a map. These maps can be created with the WGT Map Maker. Each map can be up to 320x200 tiles, and use a maximum of 65536 tiles. If you're wondering why, 65536 is the number of unique numbers an unsigned short integer can hold, and 320x200 tiles requires 128k of memory. Each number indicates a tile to display at that position in the map. In most graphics libraries, the job of drawing the tiles from a starting position in the map is left to the programmer. However, WGT has a very powerful scrolling library which controls all the aspects of scrolling the screen, and drawing sprites in your world. Therefore we will focus on how to use the scrolling system rather than how it works. The code is also provided so you meet each program's requirements.

Tiles are created using the WGT Sprite Editor. To create a group of tiles, use the sprite slots 0-2000 and save them into a sprite file. If you have more than 2000 tiles loaded at once, you will need to use multiple sprite files. Sprite files are not for sprites only. They can hold any group of images that will be used for any purpose. The important point you should remember is that sprite files are just an array of images, or blocks. You will see this in the example programs declared as

```
        block mytiles[256];  or  block mysprites[500];
```

Tiles are usually square, but they do not have to be. Rectangular tiles are allowed in version 5.1 of WGT. Tiles have a maximum size of 64x64 pixels.

12.2 Tile Types

Tile types are a simple way of organizing tiles into similar groups. The most common types of tiles in a game are hollow and solid. By assigning each tile a number, we can categorize each tile. For example, we can define a couple of tile types as such:

```
        #define HOLLOWTILE 0
        #define SOLIDTILE  1
```

An array of 256 short integers is used to hold the types for each of the first 256 tiles loaded. All tiles containing a 1 in this array would be considered to be solid and your program can react accordingly. In this example, tile contains a tile image number number from 0 to 255.

```
  example:   if (tiletypes[tile] == SOLID)
                  printf("You hit a wall");
```

Tile types are not required by the WGT scrolling system, but they do save you time, and make your source code easier to read if you use defines like above. Map files only save the first 256 tile types. If you use more than 256 tiles you will have to define them in a different file or array within your program.

It will also help if you put all the solid tiles together so you can access them with an IF statement.

```
   example:
       if ((tile >= 128) && (tile <= 200))
         {
          if ((tiletypes[tile] == DOOR) && (keys > 0))
            open_the_door ();
          else you_hit_a_wall ();
         }
```

The tiles 128-200 in this example would contain solid tiles. A further comparison would narrow the kind of solid tile down. If the type of tile is a DOOR and you have at least 1 key, then it would open the door. This way you can detect doors, but it won't open them unless you also have a key.

Planning ahead will save you some programming time. Once you draw the tiles with the sprite creator and design the map files, it will be hard to switch the tiles around. Think about this BEFORE you start drawing your tiles.

12.3 Coordinate Systems

Two different coordinate systems are used with the WGT scrolling system. They are:

Map coordinates: Tile increments (tile dimensions)
To convert a map coordinate to a world coordinate, multiply by the tile dimensions. These are the same coordinates used in the Map Maker.

World coordinates: Pixel increments (1 pixel)
To convert a world coordinate to a map coordinate, divide by the tile dimensions. These are used by scrollsprites because they can have any orientation on the map.

12.4 Scrolling Windows

A scrolling window is defined with the winitscroll routine. The window contains a single scrolling background which can be combined with other windows to create parallax scrolling effects.
 Windows that contain the background that is farthest away are called NORMAL windows. All tiles in these windows are shown normally, with color 0 being drawn. Windows that contain the layers overtop the NORMAL window are called PARALLAX windows. Every pixel containing the color 0 will be considered to be see-through. This allows you to place PARALLAX windows over the NORMAL window, giving different layers that move at different speeds. This is called parallax scrolling.

When constructing each frame of the animation, you must draw the windows in order, from back to front. You may also draw sprites between each layer if desired. Once the frame has been completed, it is copied to the visual screen with the wcopyscroll routine. To place objects onto the scrolling window, we use a sprite structure called scrollsprites.

12.5 Scrollsprites

Unlike the commands in wspr_wc.lib, movement and animation is left to the programmer in the scrolling system. This is due to the size of the animation and movement structure required and the number of sprites on the map at once. Large maps may have over 1000 sprites on at once, so we must keep the memory used to a minimum. Sprites used in the scrolling windows are called scrollsprites. We will use this term to make a distinction from the sprites used in the wspr_wc.lib library.

The scrollsprite structure is defined as the following:

```
typedef struct {
        char on;                /* sprite is turned on=1 */
        short x;                /* world x coordinate  */
        short y;                /* world y coordinate  */
        unsigned short num;     /* image # from sprites array to show  */
        } scrollsprite;
```

on can be either 0 or 1. 1 means the scrollsprite will be drawn. Scrollsprites can be turned off and still retain their coordinates and sprite number.

x and y are the world coordinates of the scrollsprite. World coordinates are based on the number of pixels in the world, and depend on the size of the tiles used.

To make the scrollsprites animate and move, you must change the values in the scrollsprite array yourself. For animation, it is wise to place your sprites in sequence so you can animate them with a simple counter loop. Once the counter reaches the last sprite image, reset it to the first image to repeat the sequence.

Moving scrollsprites can be accomplished in many ways. You may want them to follow a predetermined path, a random path, or have some kind of artificial intelligence to control their movements and actions.

The wshowobjects command is used to display a range of objects on the map. It has the following prototype:

```
void wshowobjects (short currentwindow, short start, short end, block *sprites,
                        scrollsprite *wobjects);
```

This routine simply changes the current drawing buffer to scrollblock[currentwindow], which contains the frame being contructed, and draws each sprite on the screen with wputblock. It only draws the sprite if it is turned on, the image is not NULL, and it lies within the current viewing position of the map.
 By changing this routine you can add sprites that are skewed, texture mapped, have shadows, or any other special effect. Any WGT function that operates with a block can be used to manipulate the sprite image.

The positions of the sprites on the screen are based on the world viewing coordinates of the current window, and the world coordinates of the objects. The speed and object will scroll across the screen depends on which window it is associated with.

12.6 Global Variables in the Scrolling Library

Several variables contain information about the map and windows being used. They are:

```
#define MAXWINDOWS 50          /* Maximum number of windows available

block *scrolltiles[MAXWINDOWS];           /* Pointer to array of tile images for each window
wgtmap scrollmaps[MAXWINDOWS];       /* Pointer to map for each window */

wgtmap scrollblock[MAXWINDOWS];       /* Holds the image for the scrolling window */
```

Note that PARALLAX layers have the same scrollblock pointer as the NORMAL window they are linked to. This lets you draw multiple layers on a single drawing buffer.

```
short mapwidth[MAXWINDOWS];            /* The width (in tiles) of the map loaded */
short mapheight[MAXWINDOWS];           /* The height (in tiles) of the map loaded */
short tilewidth[MAXWINDOWS]; /* The width (in pixels) of tiles */
short tileheight[MAXWINDOWS];/* The height (in pixels) of tiles */
short windowmaxx[MAXWINDOWS];       /* The width of the window (in pixels) minus 1.*/
short windowmaxy[MAXWINDOWS];       /* The height of the window (in pixels) minus 1.*/
short worldx[MAXWINDOWS];    /* World x coordinate of the top left corner
short worldy[MAXWINDOWS];    /* World y coordinate of the top left corner
short worldmaxx[MAXWINDOWS];           /* Maximum possible x coordinate each the window
short worldmaxy[MAXWINDOWS];           /* Maximum possible y coordinate each the window
short windowwidth[MAXWINDOWS];       /* Width of window (in tiles) */
short windowheight[MAXWINDOWS];      /* Height of window (in tiles) */
```

Example:
If you are working with window 0 defined as MAINWIN and object 0 is moving to the right, it is necessary to keep the object within the map boundaries. Let's say you want the object to wrap around to the other side of the map.

```
   wobject[0].x+=8;   /* Move to the right at 8 pixels at a time */
   if (wobject[0].x > mapwidth[MAINWIN] * tilewidth[MAINWIN])
       wobject[0].x = -64;  /* Move it a little off the screen so it will
                 slowly come onto the screen instead of suddenly
                 appearing. */
```

12.7 Saving Map Files

You will need to save a map file when using a "Save Game" feature in your program. wsavemap is provided for this purpose. It will save the map data, positions of objects, and tiletypes. It does not save the current viewing coordinates of the scrolling windows. You will need to save this and any other data in a different file.

When using parallax scrolling, it is only necessary to save the layers that could have been modified with wputblock. If you do not use this command, it is better off to just save the scrollsprite arrays in your own file if the parallax layers contain scrollsprites.

12.8 Optimizing SOVERLAP

Checking for overlapping scrollsprites can be time consuming. Below is the source code for the soverlap routine:

```
short soverlap (short s1, scrollsprite *wobjects1, block *sprites1,
                short s2, scrollsprite *wobjects2, block *sprites2)
/* Sees if two objects overlap, by comparing the rectangles each are
   contained in.  Pixel precise detection is not possible with WGT routines. */
{
   short n1,n2;
   short sw1,sh1,sw2,sh2;  /* width/height of both sprites */
   scrollsprite *obj1;
   scrollsprite *obj2;
   block image;

   obj1=&wobjects1[s1];
   obj2=&wobjects2[s2];


   if ((obj1->on & obj2->on)) /* Are they both on? */
   {
      n1 = obj1->num;  /* for easier reading */
      n2 = obj2->num;

      image = sprites1[n1];
      sw1= *(short *)image;    /* Width is first 2 bytes of data */
      image+=2;
      sh1= *(short *)image;    /* Height is next 2 bytes of data */

      image=sprites2[n2];
      sw2= *(short *)image;
      image+=2;
      sh2= *(short *)image;

      if (( obj2->x >= obj1->x - sw2 ) &&
          ( obj2->x <= obj1->x + sw1 ) &&  /* check all four corners */
          ( obj2->y >= obj1->y - sh2 ) &&
          ( obj2->y <= obj1->y + sh1 )) return 1;
   }
   return 0; /* not colliding */
```

You can see that for every sprite, the width and height of the image must be extracted from the image. It can be optimized by storing the widths and heights into an array after you load the images. If you only use one array for all scrollsprites, you can optimize the routine further. Here is an example of an optimized version:

```
int collide (int s1, int s2)
/* Sees if two objects overlap, by comparing the rectangles each are
   contained in.  Assumes we are using the same image and scrollsprite
   arrays for both sprites, and they are called sprites and wobject. */

{
    int n1,n2;
    int sw1,sh1,sw2,sh2;  /* width/height of both sprites */
    scrollsprite *obj1;
    scrollsprite *obj2;
    block image;

    obj1=&wobject[s1];
    obj2=&wobject[s2];

    if ((obj1->on & obj2->on)) /* Are they both on? */
    {
            n1 = obj1->num;  /* for easier reading */
            n2 = obj2->num;
            sw1 = spritewidth[n1];
            sh1 = spriteheight[n1];
            sw2 = spritewidth[n2];
            sh2 = spriteheight[n2];
            if (( obj2->x >= obj1->x - sw2 ) &&
                ( obj2->x <= obj1->x + sw1 ) &&  /* check all four corners */
                ( obj2->y >= obj1->y - sh2 ) &&
                ( obj2->y <= obj1->y + sh1 )) return 1;
    }
    return 0; /* not colliding */
}
```

13.0 Program Timing

WGT provides extra timing control through a set of routines that work with the IBM Hi-Resolution Timer which operate at a custom frequency specified by the user. The standard PC timer runs at 18.2 ticks per second. This is not very accurate when graphics and frame rates are involved. Most games programmers require millisecond accuracy.

In order to initialize a custom timer counter, use the winittimer command followed by wstarttimer. The winittimer command Initializes a timer event semaphore. The wstarttimer command allows you to specify which rate the Semaphore will be posted.

```
winittimer ();
wstarttimer (70);
```

This will to be called 70 times per second.
by calling wchecktimerticks, you can have a ULONG returned which tells you how many ticks have occured. The maximum resolution of the timer is 1000 ticks per second, although accuracy may suffer if you attempt for > 500 hz. Most games only need 60-100hz.

In order to stop calling this procedure, use the wstoptimer command.

To create a program which runs at the same speed on any computer, you can use the simple timer routine shown above. First you need to divide the program into two sections. The first section moves sprites and objects. The second section updates the screen and draws graphics. At the beginning of each frame you draw, store the value from wchecktimerticks into a second counter variable, then reset the counter to 0 (wresetticks). You should then perform the movement section in a loop from 0 to the second counter's value. The drawing section is then called once to update the screen. This will enable the objects to move at a constant rate, while the screen is updated as fast as the computer can handle. The next frame will take the total time to draw the current frame into consideration.

13.1 Calculating Frames Rates

Suppose a program had the following variables and routines:

```
int timer;
int updates;          /* Number of times the screen is updated per second */
int framerate;


void draw_screen (void)
/* Constructs a frame of animation and displays it on the screen */
{
 updates++;

 /* Construct the image */

 wgtprintf (0, 0, NULL, "%I", framerate);

 /* Display the image */
}
```

```
void main_loop (void)
{
 do {
  draw_screen ();
  if (timer > TIMER_FREQUENCY)
   {
    timer = wchecktimerticks();
    framerate = updates;
    timer = 0;
    updates = 0;
   }
 } while (!kbhit ());

}
```

This code will keep track of how many times the procedure draw_screen is called in 1 second. You can then print this value while the program is running to show how many frames per second it is achieving.

14.0 The WGT Menu Library

The WGT Menu Library provides a quick and easy method of selecting options available to the end user. If you have a mouse, you can simply move to the item you want, and click the mouse button. If do not have a mouse, use the arrow keys to move around in the drop down menus.

Creating a drop down menu:

First you need to define what the names of the drop down menus will be. This is done by entering them into an array at the beginning of your program:

```
char *menubar[10] = {" QUIT  "," FILES "," Menu 1 "," Menu 2 ", NULL, NULL,
                                        NULL, NULL, NULL, NULL};
```

These names will appear on the menu bar at the top of the screen. Notice how some are NULL. These will not show anything on the menu bar. You should put spaces on either side of the names, to keep the choices apart. When a user moves the mouse over these names (or presses F10 when there is no mouse installed), a sub-menu will appear.

To define the sub-menus, you must enter each choice as follows:

```
strcpy (dropdown[0].choice[0], " Help ");
strcpy (dropdown[0].choice[1], " Quit ");
strcpy (dropdown[1].choice[0], " Load ");
strcpy (dropdown[1].choice[1], " Save ");
```

This would make sub-menus under the first and second drop-down choices.

The format for sub-menus is:
strcpy (dropdown[m].choice[n], "Your choice")

with m and n ranging from 0 to 9.

This means you can have up to 10 drop down menus, with up to 10 choices in each, for a total of 100 choices available to the user at a click of the mouse button! The source code for the menu library has been included so you can modify these limits if needed.

To determine which menu choice was clicked on, use the checkmenu command. This command will remain in a loop until the mouse button is clicked, or the user selects a menu choice with the keyboard by hitting enter on the drop down menu.   If the mouse is clicked below the menu bar, checkmenu will return -1. This allows you to have other buttons on the screen that can be selected, by looking at where the mouse was clicked.

Menus can support different fonts as well. To change the font, load it in using the wloadfont command and change the variable menufont:

```
        menufont = sans_serif;
```

The menu will be shown using your font when it is displayed. Make sure the font is not too large however, or the all menus will not fit on the screen.

Each drop down menu can have different colors. To change them, set the following variables:

```
dropdown[0].color = ?
dropdown[0].bordercolor = ?
dropdown[0].textcolor = ?
```

The menu bar can have different colors as well by setting:

```
menubarcolor = 254;
menubartextcolor = 1;
bordercolor = 255;
highlightcolor = 144;
```

To see if the mouse is installed, check the variable mouseinstalled. It will equal 1 if the mouse is installed, or 0 if no mouse is found when initdropdowns is called.

At default, the hotkey which brings the drop down menus from the keyboard is F10. You can change this key by changing the value in menuhotkey.

15.0 Using the WGT File Selector (FULL SCREEN SESSIONS ONLY)

WGT includes a routines which displays a fully functional window for selecting filename.   It can list files with common extensions, and change to different drives and directories. To activate the file selector, use the wfilesel command.

At the top of the window, a title will be shown which tells you what file operation will be performed on the chosen file. The most common file operations are loading and saving. Below the title is a black box. This is the text entry box. If you prefer typing in a filename instead of using the mouse, you can simply begin typing the filename. After the first letter is pressed, a cursor will be shown in the text entry box. You may also click on the box to begin entering your text. The backspace, delete, and arrow keys are all functional when in the text entry box.

Below the text entry box are three file mask buttons. The first contains the current file mask. All files with this extension will be listed in the directory listing below. If you want to change the file mask, click on the first button and type in the new mask. Hit enter when you are finished and the file listing will change accordingly. The second file mask button is always "*.*", which will list every file in the current directory. The third button sets the file mask to the default mask which was first used in the file selector.

In the middle of the file selector, the file listing is shown. There are three kinds of elements in the listing, each distinguished by the text to the right of them. Disk drives are shown with "<DRIVE>" beside the drive name. Clicking on a drive will attempt to change the current directory to that drive. Directories are shown with "<DIR>" beside the directory name. Use the ".." directory name to move back a level in the directory tree. Files are shown with their file size to the right of them. Clicking on one of these will close the file selector, and return the name of the file you selected.

To the right of the directory listing is a slider and two buttons. The buttons move up and down through the directory listing. The slider shows a box in relation to which portion of the listing is being shown. If you hold the mouse button while dragging the box, you can quickly move to a different location in the listing. As well, moving up or down a page at a time can be achieved by clicking above or below the box.

At the bottom of the file selector is a cancel button. This is used to abort the file operation.

If a filename is chosen, the routine will return a pointer to the filename. If the cancel button was chosen, it returns NULL.

The file selector can be moved around on the screen if you pass it the name of a block containing the current screen contents. It will then be able to restore the background when it is moved to a different location. This block is also used to erase the file selector when the routine ends. If the block is null, it is up to the programmer to restore the screen, and the selector will be fixed in one location.

16.0 WGTLIB: Data File Libraries

In order to protect your graphic files from other users, WGT provides a way to collect all of your files into one large data file and even protect it with a password. This will prevent other programmers to look at your graphics. Data files are created using the "wgtlib.exe" utility. It operates similar to the wlib utility included with the C compiler.

It can drastically reduce the number of files sitting around in a directory, and reduces the chances that one of the necessary files is stored in an improper directory.

WGTLIB is a DOS parameter based program which combines the given files into a LIBRARY. You begin by specifying a library filename, password (if desired), and then specify individual filenames.

Options:

+	Add file to library
-	Remove file from library
*	Extract file without removing it
-* or *-	Extract file and remove it from library
+$	Add password to library
-$	Remove password from library
*$	Modify existing password
@	Execute commands in listfile

Examples:

To update a library file called TEST.LIB and add LIBFILE.DOC,
```
TYPE    WGTLIB TEST.LIB +LIBFILE.DOC
```

To update TEST.LIB and add two files and a password,
```
TYPE    WGTLIB TEST.LIB +WGTLIB.EXE +$MYPASSWORD +LIBFILE.TPU
```

To view the contents of TEST.LIB (with password),
```
TYPE    WGTLIB TEST.LIB #MYPASSWORD
```

To remove WGTLIB.EXE from TEST.LIB,
```
TYPE    WGTLIB TEST.LIB #MYPASSWORD -WGTLIB.EXE
```

To remove the password from TEST.LIB,
```
TYPE    WGTLIB TEST.LIB #MYPASSWORD -$
```

To execute all commands within a listfile,
```
TYPE    WGTLIB TEST.LIB @MAKEFILE.DAT
        Where MAKEFILE.DAT contains +,-,*,-* or *-,+$,-$,*$ commands
        (one per line)
```

NOTES:
-any time you specify a library filename which does not exist, WGTLIB creates a new file
-once a password is added, you must specify it by typing # and the password as the second parameter for all subsequent commands
-removing a password only requires a -$, no other parameters
-full pathnames are accepted anywhere a filename is used
-passwords are a maximum of 15 characters

Using a library file is very simple:

Initialize the library file by calling

```
        setlib("libfile.wlb");
```

If you set a password on the file, call

```
        setpassword("secret");
```

All other commands in WGT which load something off the disk will look in the library file. To test it, make a new directory and copy your program and the library file. Do not copy the separate files you put inside the library file. Try running the program. It should work with no problems unless you forgot to put a file in the library, or you have pathnames in your load commands.

Suggestions for files to store in a library:

1) Anything which you would like to remain hidden from the user.
2) Graphics images to be used in a program.
3) Sound files (VOC,CMF,MOD)
4) EXE'S and COM's that you rarely use (ZIP the library and store on a floppy). This prevents separation of related files when backing up information.
5) Drivers, etc. which a program requires to run. A neat trick is to store multiple drivers within the library. If you use different video, sound, or printer drivers according to the user's setup, you can store all the possibilities in one file. Then when the program runs, it loads the appropriate driver into a buffer and outputs it to disk. Now the user has only the necessary files in the directory. It can be done in memory instead of disk as well.

17.0 FLI/FLC Animation Library

This small library is used to display animations stored in FLI or FLC format. These formats are used by the commercial programs Animator(Pro) and 3D Studio, both by Autodesk. The files store information in a compressed format which reduces the amount of data that needs to be stored, especially in large graphic-intensive animation sequences. Even with the compression scheme, some files will exceed the memory capacity of the computer. To handle these situations, the library supports playing animations straight from disk or from memory.

If you choose to play from disk, any size file may be loaded and played, but continued disk access will slow down the animation. If you have a slow computer or a slow disk drive, this may cause significant delay.

From memory, files may be played as long as the entire file fits into memory at once. Since every machine can have a different memory limitation, you must be careful to provide enough error-detection as possible when loading and playing from memory.

Animation files may change the palette during execution, and some programs do not want their palette altered. To prevent the palette from being changed (if you do this, colors probably won't look right) you can pass a 0 to the openflic command for the color-update variable. Normally this value would be a 1.

At this point, animation is extremely simple. First, call the openflic command and pass it a filename to load, a mode to run the animation with (FLIC_DISK or FLIC_MEM), and a status value indicating how to update the palette. Once this is completed you may perform the nextframe command in a while loop. This loop should check the return value of the nextframe command and watch for a result of FLIC_DONE. You may choose to break out of the loop at this point, or you may continue (the animation will cycle through indefinitely). Once the animation loop is done, you should call closeflic to close the file and tidy up any loose ends.

```
/*
```

```
==============================================================================
==
                    WordUp Graphics Toolkit Version 5.0
                          Demonstration Program 52

 A full FLI/FLC animation player. Allows filenames with wildcards and a
 selection of memory or disk playback. Resolution of the animation is
 assumed to be 320x200 max (although you could assign flicscreen to a large
 virtual screen buffer to handle bigger resolutions).

 *** PROJECT ***
 This program requires the WGT5_WC.LIB and WFLIC_WC.LIB files to be linked.

 *** DATA FILES ***
 Any FLI or FLC file (or files).
                                                  IBM OS/2 Warp Version
==============================================================================
==
*/

#include <stdlib.h>
#include <wgt5.h>
#include <wgtflic.h>
#include <os2.h>

#define ESC        27        /* ESCAPE KEY */

char       ch;              /* Keyboard input */
int        playmode;          /* Memory or disk playback */
int        filefound;         /* 1 if the animation file was found */
int        oldmode;           /* Previous video mode */
int        status;          /* Status of FLI/FLC */
int        flic_mode;

char *videoBuffer;
void main (int argc, char *argv[])
{
 unsigned totl;

 if ((argc < 2) || (argc > 3))       /* Display how to use this program */
 {
   printf ("\nWGT52 - Plays FLI and FLC files from either memory or disk\n");
   printf ("USAGE:   WGT52 filename [play_mode]\n");
   printf ("playmode can be:\n");
   printf (" 0 - Play from disk (default)\n");
   printf (" 1 - Play from memory\n");
   printf ("\nPress any key\n");
   getch ();
   exit (1);
 }

 if (argc > 2)
   flic_mode = atoi (argv[2]);       /* Get playmode from command line */
 else flic_mode = FLIC_DISK;         /* Or default to disk */
 if (flic_mode > 1)
   flic_mode = FLIC_DISK;


 oldmode = wgetmode ();              /* Preserve initial video mode */
 vga256 (videoBuffer,320,200, WGTDRAWMODE_FSVGA);              /* Go to graphics mode */
 wcls(0);
 flicscreen = abuf;                  /* Set to visual screen */
                                                          /* You must set this AFTER vga256(); */

 if (openflic (argv[1], flic_mode, 1) == FLIC_OK) /* See if we opened file ok */
   do {
     status = nextframe ();               /* Show frame of animation */
     if ((status != FLIC_OK) && (status != FLIC_DONE))
         break;                          /* Abort if error */
     delay (flichdr.speed);               /* Delay proper amount */
   } while (!kbhit ());                 /* Continue until keypress */

 /* NOTE: If you don't want the flic to loop, remove the check for
        FLIC_DONE. This will abort playback when the animation is done */

 while (kbhit())                        /* Get key from buffer */
   ch = getch ();

  closeflic ();                         /* Close current animation */

 wsetmode (oldmode);                    /* and video mode */
}
```

18.0 Fixed Point Math

As you have probably noticed, using floating point numbers in graphics routines is pretty slow. If you need fractions you need floating point right? Wrong! Whenever you don't need extreme accuracy, you can use fixed point math. Fixed point means exactly what it is called. The decimal point remains at a fixed position within the number. It is an abstract concept, because you have to imagine the decimal point is really there. Let's start with a simple example. Suppose you want to represent the number 1234.56 using fixed point. First you store all the digits into an integer, ignoring the decimal place for now. The integer would be 123456, and have an imaginary decimal between the 4 and the 5. If you wanted to access the whole part and truncate the decimals, you would divide the integer by 100, and store the result into another integer. If you wanted to keep only the decimals, you would take the remainder of this division.

Most commonly you will want to use the whole portion as a coordinate on the graphics screen. For this application of fixed point, you can safely ignore the decimals because you can't change the color of half a pixel. The imaginary decimal is usually placed at a certain bit within a hexadecimal number. Instead of dividing and taking the remainder, you can shift the number right and perform an AND operation. As well, the decimal is usually placed at the 8th or 16th bit. This allows you to fit the entire whole or fractional part within a register. The accuracy you need will determine how many bits you reserve for the decimal.

Placing the decimal at the:8th bit gives you 256 possible numbers, accuracy is 1/256 or 0.00390625.
Placing the decimal at the 16th bit gives you 65536 possible numbers, accuracy is 1/65536 or 0.000015259.

Using 8.8 fixed point (8 bits for the whole part, and 8 for the fraction) works the easiest in assembly language because you can fit the entire number into a register, such as DX. To access the whole portion, take DH. The only disadvantage is the maximum whole number you can have is 256.

You should base your decimal point on the highest coordinate you will use. Say you had a coordinate system that went up to 1024, or 2 to the 10th. This means you will require 10 bits for the whole number. If you choose to store the number into a 32 bit register, you have 22 bits left over for the decimal. That's a lot of precision if you need it. Another reason for fitting the value exactly into a register is the fact it will wrap around automatically.

So far we haven't done anything with our fixed point number. The most frequent application you will encounter is adding two fixed point numbers together. This simple concepts is used in scan converting polygons, gouraud shading, linear texture mapping, tweening points, image scaling, and many other graphics routines.

When adding or subtracting fixed point numbers, you just perform the operation on the integers. The decimals will carry into the whole number as if the decimal point really existed.

Let's use tweening a coordinate as an example. If you have two coordinates (50, 50) and (100, 200) and you want to move from the first point to the second point in 30 frames of animation, this is what you might do:

First you need to find out how many pixels in each direction the coordinate will move.

```
 distx = x2 - x1 + 1;
 disty = y2 - y1 + 1;
```

You want to store these values into a 16.16 fixed point number. To move the whole number to the correct position, you have to shift the number 16 bits to the left.

```
 distx <<= 16;
 disty <<= 16;
```

Now divide this by 30 frames to get the number of pixels the coordinate will move per frame.

```
 xstep = distx / 30;
 ystep = disty / 30;
```

Using the 2 coordinates given:
  (50, 50), (100, 200)

```
 xstep = ((100 - 50 + 1) << 16) / 30 = 111411
 ystep = ((200 - 50 + 1) << 16) / 30 = 329864
```

Our initial coordinates must be stored in fixed point as well, so we can add xstep and ystep to it.

```
 x1 = 50 << 16;
 y1 = 50 << 16;
```

For each frame, we add xstep to x and ystep to y. To get the actual screen coordinate, shift x and y to the right 16 bits.

The whole routine might look like this:

```
void tween_point (int x1, int y1, int x2, int y2, int frames)
{
        int xstep, ystep;
        int count;

        xstep = ((x2 - x1 + 1) << 16) / frames;
        ystep = ((y2 - y1 + 1) << 16) / frames;

        x1 <<= 16;
        y1 <<= 16;

        for (count = 0; count < frames; count++)
        {
                x1 += xstep;
                y1 += ystep;
                wputpixel (x1 >> 16, y1 >> 16);
        }
}
```

19.0 Sine and Cosine Tables

The sine and cosine functions are essential to many graphics routines. They can be used to create circles, ellipses, spirals, 2D rotations, 3D projections and more. As we discussed in the fixed point chapter, floating point numbers are slow. You should never use the sin or cos functions within a program loop. Instead, you can make a table of fixed point numbers which includes all of the possible sine and cosine values you will use.

The sine and cosine functions have some properties that you should know about. They repeat themselves every 360 degrees. They range between -1 and 1. Finally, the cosine function is the same as as the sine function but it is offsetted by 90 degrees.

The sin and cos routines operate in radians, but most people can't grasp this measurement system. If you want to use degrees instead, you have to convert from degrees to radians so the sin and cos functions will work properly. There are 360 degrees in a circle, and this is equal to 2*pi radians. This means pi radians equals 180 degrees, or 1 degree equals pi/180. Since you know what 1 degree is, you can multiply any number of degrees by pi/180 to get the number of radians.

```
angle_in_radians = angle_in_degrees * pi / 180
```

When you make a table of values, you could easily store the results as floating point numbers. We want to eliminate them however, so we will use fixed point. The maximum whole number for a sine wave is 1, so we only need 1 bit for this. The examples use 10 bits (1024) for the decimal places. To keep things simple, create two tables, one for sine and one for cosine. Since there are 360 degrees in a circle, make the tables this size. If you want, you can make 256 angles so the index into the table can be stored evenly into an unsigned char.

```
int isin[360];
int icos[360];

for (degrees = 0; degrees < 360; degrees++)
 {
  isin[degrees] = sin (degrees * 3.1415 / 180.0) * 1024;
  icos[degrees] = cos (degrees * 3.1415 / 180.0) * 1024;
 }
```

20.0 Color Lookup Tables

Being limited to 256 different colors can sometimes be a real problem. Color lookup tables can make it seem like there are more colors by performing special operations on the existing ones. A color lookup table contains 256 unsigned characters which represent a color mapping. Here is a simple example of how a lookup table can be used:

```
unsigned char table[256];
unsigned char col;
...
col = wgetpixel (x, y);
wsetcolor (table[col]);
wputpixel (x,y);
...
```

This will read a pixel off the screen at (x,y), get a new color out of the table, and change the pixel at (x,y) to the new color.

If each number in the table were equal to its index, the color would not change. However if each number were the index plus one, and the last number in the table was 0, we could make the colors on the screen cycle, by actually changing the values of the pixels instead of changing the palette. This is a very powerful concept that can be used to create shadows, display shaded pictures, fade between two pictures, or create translucent bitmaps. How you create the lookup table will depend on what effect you want to create.

20.1 Creating Lookup Tables

Making the table consists of two steps: applying a formula to the red, green, and blue components of the color, and secondly finding the closest match within the existing palette.

Let's say we wanted to create a table for shadows. First we will take the RGB values of the color and divide them in half. This will give you half the brightness of the original color. This is the formula step:

```
float lightlevel;
int col;
...
lightlevel = 0.5;
...
newred   = (float)palette[col].r * (lightlevel);
newgreen        = (float)palette[col].g * (lightlevel);
newblue         = (float)palette[col].b * (lightlevel);
```

Now that we have the new red, green, and blue values of the darkened color, we have to find which color from the palette is closest.   To do this you can use the formula:

```
distance = sqrt (r*r + g*g + b*b);
```

In this formula, r,g, and b are the differences between the RGB values of two colors.

The whole routine might like this:

```
void wcreate_shadow_table (color *palette)
{
float fr, fg, fb;                     /* New RGB values */
int ir, ig, ib;                       /* Integer versions of the above */
int absr, absg, absb;                 /* RGB Difference between two colors */

short col;                            /* Color to darken */
short findcol;                        /* Color to compare with darkened one */

unsigned long lowest;                 /* Lowest difference between two colors */
unsigned char bestfit;                /* Color with the lowest difference */
unsigned long coldif;                 /* Color difference */

float lightlevel = 0.5;               /* Half the brightness */

 for (col = 0; col < 256; col++)      /* Loop through each color in the palette */
  {
   fr = (float)palette[col].r * lightlevel;        /* Make a new color */
   fg = (float)palette[col].g * lightlevel;
   fb = (float)palette[col].b * lightlevel;

   ir = fr;
   ig = fg;
   ib = fb;

   lowest = 655350;                   /* Set the lowest to an impossible number */

   for  (findcol = 0; findcol < 256; findcol++)            /* Search through each color */
    {
     absr = abs ( (long)palette[findcol].r - ir);           /* Find absolute difference */
     absg = abs ( (long)palette[findcol].g - ig);
     absb = abs ( (long)palette[findcol].b - ib);

     coldif = sqrt (absr * absr + absg * absg + absb * absb);
     /* Calculate how close this color is */

     if  ((coldif < lowest) && (findcol != col))                  /* This color is closer */
       {                         /* optional: don't allow a color to map to itself */
        lowest = coldif;                                          /* Set the new distance */
        bestfit = findcol;                                        /* Remember this color */
       }
    }
   shadowtable[col] = bestfit;        /* Store the closest color in the table */
  }
}
```