WordUp Graphics Toolkit 5.1

For Watcom C/C++ 10.x

Revised: May 20, 1996

First Printing, May 1995

WordUp Graphics Toolkit 5.1

Contacting the authors:

EGERTER SOFTWARE
94 Andover Drive
London, Ontario
CANADA N6J 3X2

Telephone / FAX : 1-519-641-7542
Barry or Chris Egerter
egerter@wwdc.com
75242.2411@compuserve.com (Chris)

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

ONLINE SYSTEMS OF CANADA BBS
1-519-642-0700

Last Name: word
First Name: up
Password: graphics

Internet FTP sites and directories:
x2ftp.oulu.fi pub/msdos/programming/wgt

Internet WEB Address:
http://www.wwdc.com/egerter

Table of contents:

PROGRAMMER'S GUIDE

1.0 What is WGT?
1.1 Installing WGT
1.2 Compiling the Examples
1.3 Compiling your own programs
1.4 Compiling the source code
2.0 Initializing and Restoring the Video Mode
3.0 Introduction to the VGA Hardware
4.0 Color
4.1 Image Remapping
5.0 Video Pages
6.0 Graphics Primitives
7.0 Displaying Text
7.1 Custom Fonts
8.0 Images in WGT
8.1 What is a BLOCK?
8.2 Block Copy Modes
9.0 Graphic Files
10.0 Input Devices
10.1 WGT Keyboard Handler
10.2 Keyboard Lockout
10.3 Mouse Routines
10.4 Joystick
11.0 Sprite Library
11.1 What is a SPRITE?
11.2 The Sprite Library
12.0 Multidirectional Scrolling
12.1 Tiling
12.2 Tile Types
12.3 Coordinate Systems
12.4 Scrolling Windows
12.5 Scrollsprites
12.6 Global Variables in the Scrolling Library
12.7 Saving Map Files
12.8 Optimizing SOVERLAP
13.0 Program Timing
13.1 Calculating Frames Rates
14.0 The WGT Menu Library
15.0 Using the WGT File Selector
16.0 WGTLIB: Data File Libraries
17.0 FLI/FLC Animation Library
18.0 Fixed Point Math
19.0 Sine and Cosine Tables

20.0 Color Lookup Tables.....
20.1 Creating Lookup Tables.....

WGT SPRITE EDITOR V5.1.....

1.0 System Requirements.....
2.0 Introduction.....
3.0 System Statistics.....
4.0 Program Configuration.....
5.0 Screen Layout.....
6.0 Edit Window.....
7.0 Color Selector.....
8.0 Drawing Colors.....
9.0 Swap Colors.....
10.0 Color Edit.....
11.0 Toolbank.....
PENCIL.....
LINE.....
BOX.....
BAR.....
CIRCLE.....
FILLED CIRCLE.....
ELLIPSE.....
FILLED ELLIPSE.....
POLYGON.....
FILLED POLYGON.....
SPRAY CAN.....
FILL REGION.....
COLOR CHANGE.....
WHICH COLOR.....
GET REGION.....
PUT REGION.....
HORIZONTAL FLIP.....
VERTICAL FLIP.....
CLEAR SPRITE.....
DELETE SPRITE.....
INSERT SPRITE.....
TEXT.....
LOAD/SAVE PICS.....
TRASH SPRITES.....
CLEAR CLIPBOARD.....
UNDO.....
LOAD SPRITES.....
SAVE SPRITES.....
QUIT.....
OTHER TOOLBANK.....
ROTATE 90.....
ROTATE.....

RESIZE SPRITE
RESIZE CLIPBOARD
GET SPRITE
PUT SPRITE
COPY SPRITES
MOUSE CURSOR
GRID GRAB
REMAP SPRITES
GET BIG SPRITE
PUT LAST
LOAD ANIMATION
SAVE ANIMATION
ANIMATE
SHRINK SPRITE
TRIM SPRITE
TRIM ALL
MACRO ON
MACRO OFF
MACRO PLAY
LOAD MACRO
SAVE MACRO
CREATE FONT
REGISTRATION
SETUP OPTIONS
DOS SHELL
FREE MEMORY
OTHER TOOLBANK
12.0 Quick Pick Colors
13.0 Put Storage
14.0 Get Storage
15.0 Storage Area
16.0 Storage Position
17.0 Zoom Window
18.0 Zoom Selectors
19.0 Scrollers
20.0 Help System
21.0 Status
22.0 The File Selector

WGT MAP MAKER V5.1

1.0 System Requirements
2.0 Introduction
3.0 System Statistics
4.0 User Interface
5.0 File Menu
5.1 Project Files
5.2 Load Project

5.3 Save Project.....
5.4 DOS Shell.....
5.5 Load Tiles.....
5.6 Load Objects.....
5.7 Load Map.....
5.8 Save Map.....
5.9 Quit.....
6.0 TEMPLATE MENU.....
6.1 Load.....
6.2 Create.....
6.3 Reset.....
6.4 Use.....
6.5 Exit Paste.....
6.6 Save.....
6.7 Copy Map.....
6.8 Grab Region.....
7.0 WINDOW MENU.....
7.1 Tile Window.....
7.2 Map Window.....
7.3 Object Window.....
7.4 Template Window.....
9.0 VIEW MENU.....
9.1 Supported View Modes.....
9.0 OPTIONS MENU.....
9.1 Reduced Map.....
9.2 Tile Types.....
9.3 Object Menu.....
9.4 Map Size.....
9.5 Export PCX.....
10.0 MISCELLANEOUS MENU.....
10.1 Memory Status.....
10.2 Disk Space.....
10.3 Clear All.....
10.4 Create Source.....
11.0 QUICK PICKS.....
12.0 PROGRAMMER'S INFO.....
12.1 Map File Format.....
12.2 Project File Format.....

WGT REFERENCE MANUALS.....

WGT5_WC.LIB.....

getmemfree.....
installkbd.....
lib2buf.....
mdeinit.....
minit.....

moff
mon
mouseshape
msetbounds
msetspeed
msetthreshhold
msetxy
noclick
setlib
setpassword
uninstallkbd
vga256
vgadetected
wallocblock
wbar
wbezier
wbutt
wcircle
wclip
wcls
wcolrotate
wcopyscreen
wdeinitpoly
wdeinit_triangle_renderer
wdissolve
wdonetimer
wdraw_scanpoly
wellipse
wfade_between
wfade_between_once
wfade_in
wfade_in_once
wfade_out
wfade_out_once
wfastputpixel
wfill_circle
wfill_ellipse
wflashcursor
wfline
wflipblock
wfree_scanpoly
wfreeblock
wfreefont
wfreesprites
wget_capslock
wget_imagebit
wget_imagebyte

wget_lalt.....
wget_lctrl.....
wget_lshift.....
wget_numlock.....
wget_ralt.....
wget_rctrl.....
wget_rshift.....
wget_scrlock.....
wgetblockheight.....
wgetblockwidth.....
wgetmode.....
wgetpixel.....
wgettextheight.....
wgettextwidth.....
wgouraudpoly.....
wgtprintf.....
whline.....
whollowpoly.....
winitpoly.....
winittimer.....
winit_triangle_renderer.....
wline.....
wloadblock.....
wloadbmp.....
wloadcel.....
wloadfont.....
wloadiff.....
wloadpak.....
wloadpalette.....
wloadpcx.....
wloadsprites.....
wnewblock.....
wnormscreen.....
wouttextxy.....
wpan.....
wputblock.....
wputblock_shade.....
wputpixel.....
wreadpalette.....
wrectangle.....
wregionfill.....
wremap.....
wresize.....
wresize_column.....
wresize_shade.....
wretrace.....
wsaveblock.....

wsavebmp
wsavecel
wsavepak
wsavepalette
wsavepcx
wsavesprites
wscan_convertpoly
wset_capslock
wset_imagebyte
wset_numlock
wset_scrlock
wsetcolor
wsetcursor
wsetmode
wsetpalette
wsetrgb
wsetscreen
wsettimerspeed
wskew
wsline
wsolidpoly
wstarttimer
wstoptimer
wstring
wstyleline
wtextbackground
wtextcolor
wtextgrid
wtexttransparent
wtexturedpoly
wtriangle_flat_shaded_texture
wtriangle_gouraud
wtriangle_gouraud_shaded_texture
wtriangle_solid
wtriangle_texture
wtriangle_translucent_gouraud
wtriangle_translucent_texture
wupdate_imagebytes
wvertres
wwarp
wwipe
wxorbox

WJOY_WC.LIB

wcalibratejoystick
wcheckjoystick
winitjoystick

wreadjoystick.....

WGT3D_WC.LIB.....

winit3d.....
wrotatepoints.....
wsetrotation.....

WFILE_WC.LIB.....

wfilesel.....

WSCR_WC.LIB.....

is_in_window.....
soverlap.....
wcopymap.....
wendscroll.....
wfreemap.....
wgetworldblock.....
wgetworldpixel.....
winitscroll.....
wloadmap.....
wputworldblock.....
wsavemap.....
wscreen_coordx.....
wscreen_coordy.....
wscrollwindow.....
wshowobjects.....
wshowwindow.....

WMENU_WC.LIB.....

checkmenu.....
initdropdowns.....
removemenubar.....
showmenubar.....

WSPR_WC.LIB.....

animate.....
animoff.....
animon.....
deinitialize_sprites.....
draw_sprites.....
erase_sprites.....
initialize_sprites.....
movex.....
movexoff.....
movexon.....
movey.....
moveyoff.....

moveyon
overlap
spriteoff
spriteon

WFLIC_WC.LIB

openflic
nextframe
copyflic
closeflic

WVESA_WC.LIB

wvesa_bank
wvesa_detected
wvesa_getmodeinfo
wvesa_init
wvesa_setlogical
wvesa_setphysical
wvesa_setwidth
wvesa_supported

WLINK_WC.LIB

wlink_answer
wlink_dial
wlink_flush_incoming
wlink_flush_outgoing
wlink_getuart
wlink_getvector
wlink_hangup_modem
wlink_initmodem
wlink_initport
wlink_modemcommand
wlink_modemresponse
wlink_read_byte
wlink_shutdownport
wlink_write_buffer

The WordUp Graphics Toolkit
Version 5.0

Programmer's Guide

1.0 What is WGT?

The WordUp Graphics Toolkit is a library of routines that are callable from C language under the MS-DOS operating system. This version of WGT is for Watcom C/C++ v10.0. The library is compiled for the flat memory model, which means you have to use a DOS extender such as the Rational Systems DOS/4GW which is included with Watcom.

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, image shading routines, and a communications library. 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.

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 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 WGT5 in the root directory. Each part of the toolkit will have a subdirectory in the WGT5 directory.

Follow on screen the prompts to complete the installation.

1.2 Compiling the Examples

The examples are found in the WGT5\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. As well, example 68 has a subdirectory under examples. A similar make batch file is included. The 3D rendering example is contained in the WGT5\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 windows IDE create one for you.

3. You can set up the compiler so it looks in the WGT5 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:

Edit wlsystem.lnk from the watcom/binb directory. You will see a line which says "system begin dos4g", and below a couple of lines which contain "libpath". Add another line which tells Watcom which directory to find the WGT libraries. An example of

this might be:

libpath d:\wgt5\lib

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.

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 WGT5\CODE and create new libraries in WGT5\LIB.

To compile the sprite editor or map maker, run wmake on the appropriate make file in the WGT5\CODE\SPR5 or WGT5\CODE\MAP5 directory. This will create an executable in the current directory. You should then copy the executable to WGT5\SPR5 or WGT5\MAP5 and run it from there.

2.0 Initializing and Restoring the Video Mode

WGT operates in a single video mode (with the exception of the SVGA library) called mode 13h. This mode has screen dimensions of 320x200 pixels, and has 256 colors on the screen at once. Only one video page is available in this mode, so WGT emulates page flipping through the use of virtual screens stored in memory.

The vga256 command is used to initialize the WGT library and enter this graphics mode. You 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 and sets the video mode to 320x200x256
wgetmode	- Returns the current video mode
wsetmode	- Sets the video mode

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;

void main (void)
{
 oldmode = wgetmode ();
 vga256 ();
 wsetmode (oldmode);
}
```

3.0 Introduction to the VGA Hardware

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, 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.

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 Watcom 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. 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;

        oldmode = wgetmode ();                          /* preserve initial video mode */
        vga256 ();                                      /* 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

10.1 WGT Keyboard Handler

The WGT keyboard handler installs a custom interrupt handler 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 interrupt handler. installkbd replaces the current keyboard interrupt with its own. uninstallkbd restores the original handler.

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.

10.2 Keyboard Lockout

Some keyboards have a limit of how many keys can be pressed at once. Each brand of keyboard is different in the way it behaves. After a number of keys are held down, the keyboard will report a value of 255 regardless of which keys are actually being pressed. This is called a keyboard lockout. Lockouts occur when you hold down a number of keys in the same area on the keyboard. In general, you should find keys which do not interfere with each other. If your program requires multiple keys to be pressed at once, you should include a utility to alter which keys should be hit, in order to resolve any keyboard lockouts.

10.3 Mouse Routines

The mouse routines support any Microsoft compatible mouse driver. Before you use any mouse routines, you must first install WGT's custom mouse interrupt. This interrupt 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 joystick must be calibrated. This determines the range of values the joystick will return.

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

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 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 interrupt, use the winittimer command followed by wstarttimer. The winittimer command tells the system that you will be using a different routine for the timer interrupt. The wstarttimer command allows you to specify which routine and the rate at which it will be called. A special macro called TICKS has been defined to calculate this frequency.

A simple timer routine would be:

```
void timer_proc (void)
{
timer++;
}
```

This simply increases a counter by 1 each tick. The user can install this routine by the following:

```
winittimer ();
wstarttimer (timer_proc, TICKS(70));
```

This will cause timer_proc to be called 70 times per second.

In order to stop calling this procedure, use the wstoptimer command. You can then install a new routine or change the speed with wstarttimer. To remove the custom timer interrupt completely, call wdonetimer.

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 counter into a second counter variable, then reset the counter to 0. 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 timer_proc (void)
{
 timer++;
}

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)
   {
    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

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.

The file selector looks like this:

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 5.1
                        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).
                                                        WATCOM C++ VERSION
=============================================================================
*/

#include <dos.h>
#include <stdlib.h>
#include <wgt5.h>
#include <wgtflic.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;

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 ();                              /* Go to graphics mode */
        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 */
   }
}
```

WGT Sprite Editor v5.1

User's Manual

Written by Chris Egerter

1.0 System Requirements

386 or better IBM Compatible computer
VGA or better display card and monitor
Microsoft Compatible Mouse with at least two buttons
At least 2 megs of memory

2.0 Introduction

The WGT Sprite Editor is a paint program for drawing images which can later be used in your own programs.

This sprite editor is part of the WordUp Graphics Toolkit 5.1 for Watcom C. It is meant specifically for this package and therefore saves and loads files which are currently only supported by WGT.

The WGT Sprite Editor is designed to give the programmer many tools for designing small images. These images can be saved in a common file to be loaded and displayed in your own programs. While the main function of the Sprite Editor is to design sprites, it has a variety of other built in tools which allow you to design fonts, custom mouse cursors, animation sequences, and palettes. It is a very important piece of software that is used a lot during the development of any game or graphical program.

3.0 System Statistics

When the editor is run, a number of statistics about your computer are shown. They are:

Video: This always displays "VGA compatible card detected" if a VGA or better video card is found.

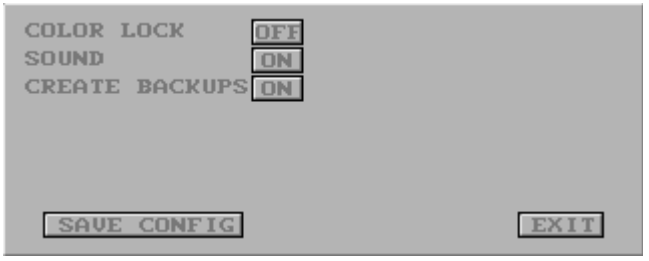
Mouse: This will display "Microsoft compatible" and the number of buttons the mouse has, if a mouse was found. It then reports what kind of mouse it is, out of the following:

- Bus mouse
- Serial mouse
- InPort mouse
- PS/2 mouse
- Hewlett-Packard mouse

The IRQ of the mouse driver is also shown.

4.0 Program Configuration

The first time you run the editor, you will be required to configure some options. A setup screen will appear which contains the following:



Turning the color lock off will prevent the editor from reassigning the menu colors when loading in a picture. This will make sure the picture is shown correctly, however the editor's menus will change color.

Turning the sound off may be desired when you are running a long macro, or you are working in a quiet area.

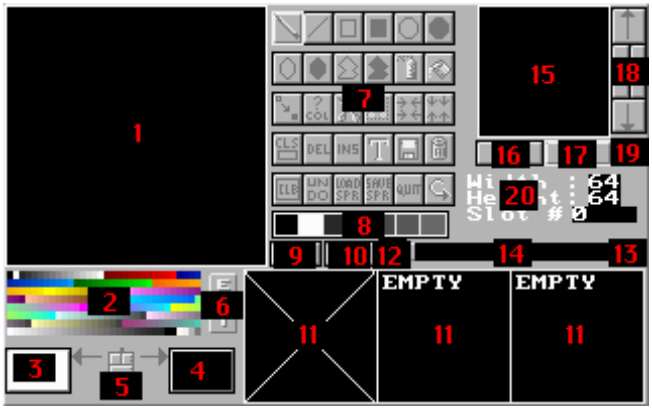
When Create Backups is turned on, the editor will rename old sprite files to an extension of ".spb" when you save over an existing file. This will protect your data against corrupt files and accidentally saving over the wrong file.

When run for the first time, either the Save Config button or Exit Config button will automatically save your settings. If you have chosen setup from within the editor, only Save Config will save to the configuration file.

The screen is divided into several sections, which will be covered in detail below. The main areas are the edit window, color selector, toolbank, storage, and size window.

5.0 Screen Layout

Below is a diagram of all the important areas on the screen.



1 - Edit Window
2 - Color Selector
3 - Left Mouse Button Color
4 - Right Mouse Button Color
5 - Swap Colors
6 - Color Edit Menu
7 - Toolbank
8 - Quick Pick Color
9 - Retrieve from Storage
10 - Put into Storage
11 - Storage Area
12 - Decrease Storage Position
13 - Increase Storage Position
14 - Storage Movement Slider
15 - Drawing Buffer / Zoom Window
16 - Upper Left Zoom Box Selector
17- Lower Right Zoom Box Selector
18 - Scrollers
19 - Help System
20 - Sprite Statistics

6.0 Edit Window

This is where all sprite editing is done. Clicking in the edit window will cause different actions depending on which tool is currently in use. It shows a magnified area of the sprite being edited. It may be magnified from 2 to 32 times the original size. Higher magnifications make it easier to draw since the pixels are larger.

7.0 Color Selector

The Color Selector contains all of the colors available for drawing your sprites. Each sprite file contains a single palette, which is used by all the sprites in the file. Each mouse button may have a different color assigned to it, by clicking with that button on one of the colors. If you change colors 253-255, the WGT Sprite Editor's menu colors will be changed.

8.0 Drawing Colors

These boxes show which colors have been assigned to each mouse button. When a tool is used in the edit window, one of these colors will be used depending on which button is pressed.

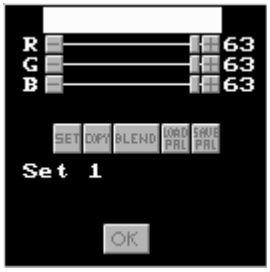
9.0 Swap Colors

The mouse icon will swap the colors of the left and right mouse buttons. This is useful when using the color change tool and choosing a right mouse button color from the quick pick boxes.

10.0 Color Edit

This will open a color control panel which allows you to modify the current palette. You can set new red, green, and blue components for each of the 256 available colors, copy one color to another, blend between two colors, and load or save palettes.

Once you have clicked on the edit button beside the color selector, a submenu will appear allowing you to change the red, green, and blue values for the color. It looks like this:



Red, green, and blue values are shown on the right, and a graphical slider shows the amount of each in the middle. Clicking on the plus and minus will change the values or you can click and drag the sliders to new positions. Below are some control buttons for performing various actions on the palette. The lines underneath tell which colors you will be changing. To finish changing the colors, click on the OK button.

Choosing SET, COPY, or BLEND will activate the color selector. Press the left button to choose a color or the right button to cancel the operation. Blend and copy require you to select two colors.

SET simply selects the color to change the RGB values of.

COPY will copy the first color to the second color.

BLEND will smoothly blend the colors between the two selected. For example, to create a gray scale, set one color to black, and another to white. Blend the two together by clicking on the black, and then the white. The more colors between the two selected, the smoother the transition is between them.

LOAD and SAVE will bring up a file selector for selecting the palette. The default palette may be changed by modifying the "default.pal" file.

11.0 Toolbank

This is located in the top middle part of the screen. It contains most of the tools you will need to control the WGT Sprite Editor. Each tool in the toolbank is described in detail below.

The toolbank is split into two sets, and you can switch between them with the bottom right tool in the toolbank.

First Toolbank:

PENCIL

The pencil tool lets you draw individual pixels where you click the mouse. It is the simplest of the tools.

LINE

The line tool will draw a line between two points. Click on the first point, and hold the button. Move the mouse to the second point and release the button to draw the line.

BOX

The box tool will draw a rectangle between two corners. Click on one corner of the rectangle. Hold the button down while you move to the second corner and then release the button to draw the rectangle.

BAR

The bar tool will draw a filled rectangle between two corners. Click on one corner of the rectangle. Hold the button down while you move to the second corner and then release the button to draw the bar.

CIRCLE

The circle tool will draw a hollow circle where you click the mouse. Click on the center of the circle and hold the button while you move the mouse away from the center point. This will change the radius of the circle. Release the button to draw the circle.

FILLED CIRCLE

The filled circle tool lets you draw a filled circle where you click the mouse. Click on the center of the circle and hold the button while you move the mouse away from the center point. This will change the radius of the circle. Release the button to draw the filled circle.

ELLIPSE

The ellipse tool lets you draw a hollow ellipse where you click the mouse. Click on the center of the ellipse and hold the button while you move the mouse away from the center point. This will change the horizontal and vertical radius of the ellipse depending on the distance away from center in each direction. Release the mouse button to draw the ellipse.

FILLED ELLIPSE

The ellipse tool lets you draw a filled ellipse where you click the mouse. Click on the center of the ellipse. Hold the button while you move the mouse away from the center point. This will change the horizontal and vertical radius of the ellipse depending on the distance away from center in each direction. Release the mouse button to draw the filled ellipse.

POLYGON

The polygon tool lets you draw a hollow polygon from a set of points you select. Click on the first point with either the left or right mouse button. This will start the polygon using the appropriate button color. You may continue to add new points using the left button. The right button will finish the polygon and connect the first point with the last.

FILLED POLYGON

The filled polygon tool lets you draw a filled polygon from a set of points you select. Click on the first point with either the left or right mouse button. This will start the polygon using the appropriate button color. You may continue to add new points using the left button. The right button will finish the polygon by connecting the first point with the last, and filling the shape.

SPRAY CAN

The spray can lets you draw random pixels around the place you click with the mouse, simulating a spray paint effect. Left and right buttons have different drawing colors when clicking in the edit window.

If you click on the spray can tool in the toolbox with the right hand button, a small menu appears which allows you to change the spray settings. Click on the plus and minus buttons to change the values. From this menu, you can modify the delay time, and size of the spray.

FILL REGION

The fill region tool lets you fill an area of the sprite with a color. The area is bounded by any color other than the one at the pixel you clicked on. Left and right buttons have different drawing colors.

COLOR CHANGE

The color change tool lets you change pixels of one color to a new color, given a rectangular area. Click on the first corner of the rectangle. Hold the button and move to the second corner. Release the mouse button and all pixels of the right mouse button color will be changed to the left mouse button color.

WHICH COLOR

The which color tool lets you change the left or right mouse button color by choosing a color from the sprite. Click on the pixel anywhere on the edit window and the drawing color for the button you pressed will be changed.

GET REGION

The get region lets you cut and paste portions of a sprite. Click on the first corner of the area to grab. Hold the button and move to the second corner. Release the mouse button and the area selected will be stored in a paste buffer. You then use the paste region tool to paste the area onto the sprite in a new location. The area selected is not cleared to color 0.

PUT REGION

The put region lets you cut and paste portions of a sprite. Click and hold the mouse button while in the edit window. You can move the region that was grabbed by the get region tool to the new location. Releasing the button will paste the region onto the sprite. If you change your mind, and do not want to paste the region onto the sprite, move the region off the bottom right side so the sprite will not be modified.

HORIZONTAL FLIP

The horizontal flip tool flips the entire 64x64 drawing buffer horizontally.

VERTICAL FLIP

The vertical flip tool flips the entire 64x64 drawing buffer vertically.

CLEAR SPRITE

The clear sprite tool erases the drawing buffer for working on a new sprite. The current zoom window area is not reset to the full window. Use the UNDO tool to recover the sprite if you click on this by mistake since it doesn't confirm the action.

DELETE SPRITE

The delete sprite tool removes a sprite from the storage, and moves any sprites after it back one storage slot. Make sure you insert a blank sprite if program already uses the file or your sprites will have the wrong slot numbers.

INSERT SPRITE

The insert sprite tool moves all sprites between the current slot location and the end of the file up one slot number. It then puts a blank sprite (which takes no memory) into the current slot. In other words, it makes a space for a new sprite between the current slot and the slot before. If a sprite exists at the last slot (2000) the operation is aborted.

TEXT

The text tool allows you to print text onto the clipboard. You can use and test custom fonts that have been created with the sprite editor. A text menu will appear when you select this tool. It has the following options:

Load font will bring up the file selector. Font files have the extension ".wfn" and can be created using the Create Font tool.

Default font returns to the default 8x8 font, if you have chosen a custom font previously.

Type text switches views to the clipboard. Click on the place you want the text to appear. Type in your text with the keyboard. Hitting enter will move the cursor to the next row. Also, you can use the backspace key to correct any mistakes. The left mouse button color is used for the text color. Once the text is placed on the clipboard, you can use the Get Sprite or Get Big Sprite tools to place it into a slot.

Exit will take you back to the main menu.

LOAD/SAVE PICS

This tool brings up a menu for loading and saving pictures which are stored in another format. You can load or save BLK, PAK, PCX, CEL, BMP, or LBM pictures.

After clicking on the button, you will be asked if you want to load or save a picture. The first two selections operate on the picture in the clipboard. The second two selections (load/save single) operate on the sprite in the current storage slot. Using these, you can directly load an image into a single sprite slot, or export individual sprites.

After selecting one, another menu appears which lets you choose which kind of picture to load or save. If you choose a type, a file selector appears and you may select your picture. If you are loading a picture, it will be placed onto the clipboard. You will have to use the Get Sprite or Get Big Sprite tool to use the picture.

If you are loading a picture which contains a palette, you will be asked if you want to use the palette of the picture,or remap the picture to the current colors. Remapping will attempt to recolor the picture with the current palette. When loading a picture, the Sprite Creator modifies the palette for its own use. It makes sure color 1 is white, and colors 253-255 are grays. If the picture uses these colors, it will appear different. To get around this, load the picture in twice, the first time using the palette,and the second time remapping it. This will ensure the picture is as close as possible to the original. Alternately, you can turn off the color lock option from the setup screen. This will force the editor's menu colors to be the ones in the picture.

If you are loading a single picture into a storage location, the palette information is ignored, and no color remapping is performed.

TRASH SPRITES

The trash sprites tool deletes all of the sprites from the storage. You will be asked to confirm this action.

CLEAR CLIPBOARD

The clear clipboard tool will clear the clipboard with a color, depending on which mouse button you click.

UNDO

The undo tool restores the sprite with what is in the current undo buffer. The sprite is copied into an undo buffer every time you click on a tool button. For example, if you click on the circle tool and draw a circle, then click on the undo tool, the circle will disappear, and the sprite returned to how it looked before. Undo only works for tools in the edit window. Actions such as CLB, or getting/putting with the storage cannot be undone.

LOAD SPRITES

The load sprite file tool prompts for a filename with the file selector. The current storage area is erased and the new sprite file is loaded. All sprites in the current storage area will be lost.

SAVE SPRITES

The save sprite file tool prompts for a filename with the file selector. If the sprite file you choose already exists, a backup will be made with the extension ".spb". You may turn this option off in the setup screen. The current storage is saved into a sprite file for loading into your own programs.

QUIT

The quit tool will exit the sprite editor and your unsaved sprites will be lost. You will be confirmed on this action.

OTHER TOOLBANK

The other toolbank allows you to switch between two sets of tools. The first tools you see when running the sprite editor are only half of them. Click on this to use the second half.

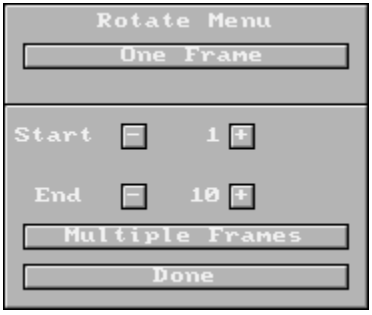
Second Toolbank:

ROTATE 90

The rotate 90 degrees tool quickly rotates the image by 90 degrees counterclockwise. For other angles, see the freehand rotate tool.

ROTATE

The freehand rotate tool brings up a rotation menu which looks like this:



One frame will allow you to choose the angle of rotation and visually see the sprite when it is rotated by that angle. After choosing your angle, click on the done button. The sprite will be grabbed into a 64x64 buffer. If the rotated sprite is larger than 64x64, it will be cut off around the edges to fit into the zoom window. The start and end buttons select a range of sprites which will be used when the multiple frames button is selected. The sprite will be rotated 360 degrees within the start and end range. This makes it extremely easy to rotate an image. To rotate an image 360 degrees by 10 degree increments, choose a range with 36 sprites and click on multiple frames. The sprite editor will first find the smallest sprite size needed to fit all of the rotated images. It will then rotate each image and place it in the storage area. Sprites may need to be adjusted manually for centering, and pixel touch-ups.

RESIZE SPRITE

The resize sprite tool takes the current sprite in the drawing buffer and resizes it over the clipboard. While on the clipboard screen, pressing the left button will resize the sprite between the top left corner and the mouse cursor, and the right button will exit.

RESIZE CLIPBOARD

The resize clipboard tool is the same as the resize sprite tool only it resizes the entire clipboard instead of the current sprite.

GET SPRITE

The get sprite tool grabs a portion of the clipboard into the edit window. This lets you grab sprites from PCX, BMP, LBM, and other pictures. Click on the top left corner of the sprite. Move to the bottom right corner and click again. The maximum size for this tool is a 64x64 sprite. To grab large sprites, see the get big sprite tool.

PUT SPRITE

The put sprite tool takes the sprite in the drawing buffer and allows you to paste it onto the clipboard. You can create a full screen picture by pasting various sprites onto the clipboard, and save it as a PCX, CEL or WGT's custom image formats. Clicking the left button while viewing the clipboard will paste and the right button will exit. If the right button is clicked on this tool, the sprite in the current storage slot will be pasted instead of the sprite in the drawing buffer. This allows you to paste sprites which are larger than 64x64 onto the clipboard.

COPY SPRITES

The copy sprites tool lets you copy a range of sprites to a new location. A menu will be displayed where you can select the range you wish to copy. After setting the source and destination slot numbers, click on the copy button. If the source and destination ranges overlap, the sprites will not be copied, and a tone will sound. Clicking on the abort button will return to the main drawing screen.

MOUSE CURSOR

This tool allows you to design a new shape for the mouse cursor. Two 16x16 pixel sprites are needed to use this tool. Draw all pixels that will appear white in the first sprite. Any color other than 0 will be considered white. To create the second sprite, choose a different color and make an outline around the first sprite. Any pixels set in this sprite will appear black. To test the mouse cursor, position the current storage area on the first sprite, and choose the Mouse Cursor tool. If the sprites are not 16x16 or the storage is in the incorrect place a tone will sound and you will return to the main screen. Otherwise, the mouse will change into your cursor so you can see what it will look like. Pressing S will prompt for a filename with the file selector. It then saves a portion of C language code which you can later use with the mouseshape command in WGT.

GRID GRAB

The grid grab tool lets you quickly grab a number of images which have been arranged in a grid on a picture. The most common use for this tool is for grabbing tiles, since they are all equal in size and fit into a grid perfectly. A menu will appear which looks like this:

If you click on the number beside each option, you may edit the value. Horizontal and vertical spacing is the number of pixels between each tile in the grid. The offset values let you move the grid around the screen. Click on view to see what the grid looks like with the current values. Clicking on grab will go through each tile in the grid and grab the next sprite. Sprites will be grabbed starting at the current storage location, but only if the grid box is completely on the screen.

REMAP SPRITES

The remap sprites tool prompts you for a filename of a palette file. All the sprites in the storage will then be remapped to use the palette chosen.

GET BIG SPRITE

The Get Big Sprite tool allows you to grab a sprite off the clipboard that is larger than 64x64 pixels. Note that you cannot edit these sprites due to the limitations of the sprite editor. You can paste the large sprites back onto the clipboard by pressing the right hand button on the put sprite tool. Other WGT commands that use sprites will be able to use these large sprites without any modifications to your program. A sprite larger than 64x64 will be reduced to fit within the storage location at the bottom of the screen, and the sprite's actual size is displayed at the top of the reduced image.

PUT LAST

The Put Last tool is the same as the Put Sprite tool but you do not get to select where it will go on the clipboard. Instead, the sprite is pasted where you last grabbed a sprite with either the Get Sprite or Get Big Sprite tools. You can edit a full screen picture easier with this tool. First grab an area of the picture using the Get Sprite tool. Edit the area as you would a normal sprite. Now click on this tool and your changes will be copied to the clipboard.

LOAD ANIMATION

This tool loads in an animation sequence that was previously saved with the Save Animation Sequence tool. Animation sequences are created and used in the animation menu.

SAVE ANIMATION

This tool saves an animation sequence that you have designed in the animation menu. This is useful if your animation is fairly long and it is difficult to recreate. You should keep animation sequences of important images so you can easily see what the final animation will look like if you make some changes.

ANIMATE

The animate tool lets you preview your sprites in animation. This is useful for seeing if the sprites will animate correctly within your program. A menu will appear in the edit window. It has the following options:

CLEAR ALL will wipe out the whole animation sequence. You will be asked to confirm this choice. The last three options have push buttons for increasing or decreasing values. Clicking the left button will change it by one. Clicking the right button will keep changing the value while you hold down the button. This is useful for quickly setting the number.

SLOT NUMBER may range from 1 to 80. This means you can have up to 80 sprites in the animation sequence. Each slot has a unique sprite number and delay value.

SPRITE NUMBER contains the sprite from the storage for this slot. The sprite will be shown just below the delay option. A sprite number of -1 means the sequence is finished.

DELAY contains the delay value for the current slot. To the right of the slot number values is a button called ADD. Clicking on it will increase the slot number, and increase the sprite number at the same time. This is very useful if you have arranged your sprites in sequence. You can quickly set up the animation sequence by holding the right mouse button. If you go too far, set the first unwanted sprite to -1, to stop the animation at that point.

At the bottom of the menu is the animate button. Press and hold this button to animate the sprite. For each slot, the sprite is shown for a period of time, depending on how long the delay value is. The sequence is repeated when a sprite value of -1 is found. See the Save and Load animation tools for managing your animation sequences.

SHRINK SPRITE

The Shrink Sprite tool removes all black pixels (color 0) on the top left edges of the current sprite in the storage. This is meant to chop off any extra space that is not needed in the sprite. After the sprite is shrunk, it is moved into the edit window. If you have aligned your sprites for animation purposes to a certain x and y offset, this tool will cause you problems, and you should only use the Trim Sprite tool.

TRIM SPRITE

The Trim Sprite tool removes all black pixels (color 0) on the bottom right edges of the current sprite in the storage. This is meant to chop off any extra space that is not needed in the sprite. After the sprite is trimmed, it is moved into the edit window. If you have aligned your sprites for animation purposes to a certain x and y offset, do not worry. Trim will not move your sprite to the top left corner of the window. Trim only changes the bottom right corner if necessary.

TRIM ALL

The Trim All Sprites tool removes all black pixels (color 0) on the bottom right edges of all of the sprites in the storage. This is meant to chop off any extra space that is not needed in the sprite. It operates the same as the Trim Sprite tool. This tool is useful before saving a sprite file. It will decrease the size of the sprite file, and save memory when you load the sprites from your program.

MACRO ON

This tool starts recording a macro which stores all of your mouse selections. You can later execute these actions again on other sprites. To make a set of actions perform on a range of sprites, you can click on one of the storage slider buttons as the last action. When you play the macro back a number of times, it will move to a new sprite and perform those actions. Only mouse movements/clicks are recorded. If a keypress is detected, the macro will stop playing. Mouse actions in the file selector are NOT recorded, so macros must not involve file operations. Macros are very useful for repetitive operations such as changing colors in sprites, and alignment adjustments.

MACRO OFF

This tool stops recording a macro, if you have previously started recording using the macro on tool.

MACRO PLAY

This tool will play the current macro the specified number of times. If you press the right button on this tool, you can set the number of times a macro will play. This counter will reset to 1 every time you execute the macro. A macro can be stopped in mid-execution by pressing a key.

LOAD MACRO

This tool lets you recall a macro you have saved. This is useful for keeping macros that you use often.

SAVE MACRO

This tool saves the current macro under a new name. A file selector will appear and allow you to choose a new name for the macro. Macros have the extension ".smc".

CREATE FONT

Custom fonts may be created using this tool. A filename is requested for the new font file. Font files have the extension ".wfn". These fonts can be used with the text tool and within the WGT programming library. To make a font file, place the images of the letters in storage locations 0-127. The storage location corresponds to the number of the character in the ASCII table. Several sample sprite files containing fonts have been included for you to begin with. You can load in one of these files and simply replace the current images with your own, and save the file under a new name. Once your characters are ready, click on this tool and type in a new font name. Each sprite will be scanned and converted into a font. Only monochrome fonts with on/off states are allowed, and any pixel other than color 0 will be considered to be turned on. Font characters must not be greater than 64x64 in size. To test your new font, load it into from the text menu and try typing some text.

REGISTRATION

This button will display the initial title screen and allows you to print an order form.

SETUP OPTIONS

This button will take you into a setup screen where you can toggle specific options of the WGT Sprite Editor on or off. This setup screen is described above in the installation section of the Sprite Editor Manual.

DOS SHELL

This button will attempt to enter a DOS shell. If you do not have enough memory, it will return to the editor Immediately. Typing "exit" will return to the editor as you left it.

FREE MEMORY

This button will show how much free memory is left.

OTHER TOOLBANK

The other toolbank allows you to switch between two sets of tools. The first tools you see when running the sprite editor are only half of them. Click on this to use the first half.

12.0 Quick Pick Colors

The quick pick boxes are designed to store several commonly used colors which can be recalled without hunting through the color selector. Click on the box with the right button to assign a new color to the box. Move the mouse over the color you wish and click the left button to assign the new color.

Clicking the left button on the quick pick will set the left mouse button color to the color in the quick pick box.

13.0 Put Storage

This button will save what is inside the zoom box into the current storage slot. It does NOT save what is in the zoom window. It only saves what is displayed in the edit window. If a sprite already exists in the current storage slot, it is erased, and replaced with the new sprite.

14.0 Get Storage

This button will delete whatever is currently in the drawing buffer, and replace it with the sprite in the current storage slot. If no sprite exists in the storage slot, nothing happens.

15.0 Storage Area

The storage area holds all of your sprites. It can hold up to 2000 sprites at once. The middle box shows the current storage slot, while the ones on either sides show the previous and next slots. When a sprite file is saved, all the storage slots are saved into the file.

The storage section is basically a group of sprites, which can be moved around in the storage, copied from one to another and of course saved and loaded back in. Instead of having many files containing separate sprites and loading them in one at a time, you can put all the graphics in one file, and load them into your programs with one command. The palette is saved as well. This is much more convenient than having lots of small graphic files because it allows you to edit them for animation quickly. You will see three boxes in the lower right corner of the screen. The middle box shows the sprite you are working with. The boxes on either side show what is next in the storage area. The sprite number indicator shows the number of the middle box.

At the ends of the storage, a box marked with a large X will be shown. You cannot store sprites in these storage slots. If a storage slot does not contain a sprite, the word "EMPTY" will be shown at the top of the storage box. This helps distinguish between sprites filled with black, and truly empty sprites which take up no memory.

16.0 Storage Position

The current storage slot may be changed by clicking on the icons on the left and right, and using the slider in the middle. Clicking the right mouse button on the side icons will quickly move through the storage. To use the slider, hold the left button and move it to a new location. The slider moves through the storage area in multiples of 20 sprites.

17.0 Zoom Window

This window shows the sprite with the actual size it will appear. Also, a zoom box is shown over the sprite. This box is the region which is shown in the edit window. The zoom box controls which area of the sprite is shown in the edit window, and it also defines what will be put into the storage when you click on the Put Storage icon.

18.0 Zoom Selectors

These icons will allow you to change either corner of the zoom box. The first icon lets you change the upper left corner, and the second changes the bottom right. An outline is shown around one of these boxes, to show which corner can be moved. As you change the size of the zoom box, the size indicators change accordingly.

19.0 Scrollers

These allow you to scroll the sprite around within the sprite buffer. They are used to view another area of the sprite while maintaining the same zoom magnitude, and also to move the sprite to a new location within the drawing buffer.

20.0 Help System

The WGT Sprite Editor contains a help system that can be examined while within the program. To activate this system, click on the question mark icon on the right of the screen. The mouse cursor will change into a question mark. This is the point and click help mode. If you click with the left mouse button on a tool or other area on the screen, the help information on that item will be shown. A right mouse button click will exit the help mode. If you click on the help button twice, a table of contents is shown. You can use the mouse to browse through all the available help topics and read the ones you click on. Again, a right mouse button click will exit the help mode.

When a help topic is shown, it may contain more than one page of information. Click a button or press a key to advance to the next page.

21.0 Status

The status indicators show where the sprite storage slot is selected, and how large the current zoom box is. You can modify these values by clicking on the value, and typing in a new one. This way you can jump to a specific storage slot, or change the dimensions of the sprite to an exact size.

22.0 The File Selector

Any time you need to save or load a file, a file selector window will appear. The file selector is used to select a file, and move to different disk drives and directories. The file selector looks like this:

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.

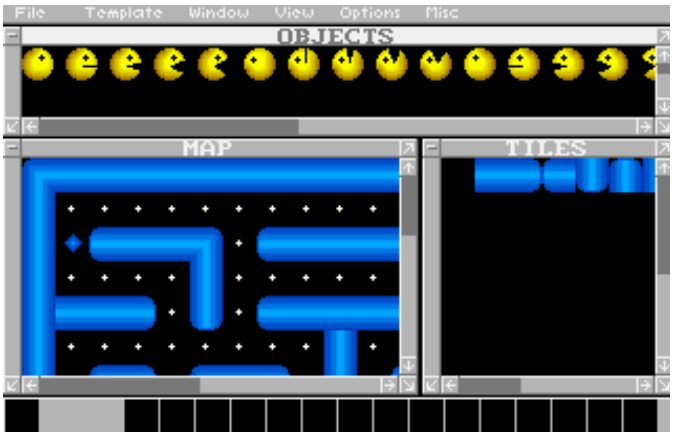
Below the text entry box are three file mask buttons. The first contains the current file mask. All files with this extension will be listed in the directory listing below. If you want to change the file mask, click on the first button and type in the new mask. Hit enter when you are finished and the 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 perform the action on the file.

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.

WGT Map Maker v5.1



User's Manual

Written by Barry Egerter

1.0 System Requirements

IBM PC-compatible, 386SX or better
(S)VGA video card and monitor
Microsoft compatible mouse and drivers

2.0 Introduction

The WGT Map Maker is a utility which will provide the tools for creating a tile-based background for a game or animation. In recent years, a number of these games have set the standard which players and developers now expect from any new release. The idea is centered around the fact that the resolution of video displays can no longer meet the expected dimensions of a video game world. Players want to be placed into a world which is much larger than the visual screen, and therefore must be scrolled in various directions to get to a new location. WGT was the first publicly available toolkit to provide a full library of routines and a utility program for the creation of such games.

Creating an image which is larger than the visual screen would require a great deal of memory, so the technique used to reduce this requirement is called tiling. Basically the user will create a world using a set of basic images which are repeated throughout the scene. A large area of sky could be created by pasting a series of solid blue squares on the screen, with only one of the blue squares stored in memory. The WGT system will simply store a single number indicating which square is used to draw the background at each point. Each number requires 2 bytes, so there can be 65536 unique tiles on a map. Therefore, a 20*20 pixel image would require 400 bytes to store, and a 15*10 section of these squares requires another 300 bytes (one "tile" number each, indicating the sky piece). This totals 700 bytes for a resulting image which is 20*15+20*10 or 300 by 200 pixels. This would require 60,000 bytes of memory to store without a tiling system. The tile system has reduced our example to 1.2% of our original storage requirements.

3.0 System Statistics

When the program is run, a number of statistics about your computer are shown. They are:

Video: This always displays "VGA compatible card detected" if a VGA or better video card is found.

Mouse: This will display "Microsoft compatible" and the number of buttons the mouse has, if a mouse was found. It then reports what kind of mouse it is, out of the following:

- Bus mouse
- Serial mouse
- InPort mouse
- PS/2 mouse
- Hewlett-Packard mouse

The IRQ of the mouse driver is also shown.

Memory Status: This shows how much total memory is free, and a breakdown of the memory usage.

4.0 User Interface

The Map Maker provides you with a set of drop-down menus and windows to ease the development process. All program features are accessible through the menus, and most of them may be controlled entirely by the mouse (little keyboard control required). Some menus are disabled if your hardware is not sufficient to support the features (example: SVGA support for VIEW mode). By hiding the options unavailable to you, the program customizes itself to meet your situation.

Up to four windows are available, depending on memory status and loaded files. These windows contain simple close and resize buttons, along with horizontal and vertical sliders. Windows may be moved by clicking the mouse on the titlebar and holding the button down while moving the window to a new location. Two window activation modes are available. If the setup program has defined the windows as auto-raise, the windows will become active as the mouse passes over them. With auto-raise disabled, the user must click on the window itself in order to activate it. Be careful when using auto-raise mode because the larger windows will tend to override the smaller ones and make selection difficult.

Certain restrictions are placed on the files and images loaded in this release of the WGT Map Maker:

1) Tiles are to be created with the WGT Sprite Editor. The dimensions of each tile in a given file should be the same. Up to 65536 tiles (in slots 0-65535 of a sprite file) are permitted. Tile #0 must not be NULL (it must contain an image).

2) Objects are to be created with the WGT Sprite Editor. Objects may be any dimension, and are expected to use the same palette as the tiles. Up to 2000 objects may be placed with the Map Maker, and any objects exceeding this amount must be placed by the program itself.

3) One map file may be loaded at a time. Parallax editing is not supported. Create parallax maps one at a time and use your program to test the end result. All maps which are to be used as parallax levels should share the same palette.

5.0 File Menu

5.1 Project Files

C programmers know the benefits of a project file. Organization of a large group of source files and related data is much simpler when they may all be kept in a project file. The Map Maker provides just such a feature for its environment. When you save a project file, the system will save all current window positions, the names and paths of tile/object/map files loaded, window priorities, quick-pick banks, and the currently active bank number.

5.2 Load Project

Using the file selector, a project file may be loaded. All currently active files will be unloaded from memory and replaced with the new selections. If you have not saved any changes made to the current files, these changes will be lost. The project file contains full path information for all data files, so the program may crash if you have moved the files since the project was created or if you have deleted any of the files. After a successful load, the Map Maker will be in the same state as it was when the project was last saved. It is recommended that you save a project file before leaving the Map Maker (if you loaded one during your session). This will make sure that all changes (quick-pick selections, etc) are recorded properly.

5.3 Save Project

A file selector will prompt you for the filename to save. This filename should have the extension MPF (Map Project File). See the comments for LOAD PROJECT to find out more about the attributes which will be saved.

5.4 DOS Shell

A feature lacking in the previous Map Maker releases was a DOS SHELL option. It is now implemented to provide the user with the ability to escape to DOS to perform a few tasks, then return to the Map Maker by typing EXIT.

The program will run the command-line processor described in the system COMSPEC environment variable. Most users will simply have this set to the standard COMMAND.COM, but you may change this variable to reflect any program you wish. Simply type

set COMSPEC=c:\dos\command.com

or the full pathname to the processor you wish to run.

5.5 Load Tiles

Before any map may be created, a tile file must be loaded. Tiles are defined as a series of images all having the same dimensions. In WGT, tiles are allowed to be any size up to 64 pixels wide by 64 pixels tall. Tiles DO NOT need to be square in version 5.1 of WGT. 1 by 1 pixel tiles are permitted! Create a tile file in the WGT Sprite Editor and save it as a normal SPR file. Make sure that ALL tiles have the same dimensions. The Map Maker permits up to 65536 tiles per file.

If your tile file contains a few tiles which do not match the dimensions of the rest, a file called "BADTILES.DAT" will be created. This file lists the tile numbers which do not meet the expected size. Use it as a reference and fix up the file in the Sprite Editor before attempting to use it in the Map Maker.

The program will warn you if you have mistakenly loaded an object file. This assumption is made if more than half of the "tiles" do not share a common size. Do not proceed further if you get this warning. Leave the Map Maker and correct the tile dimensions or re-load a different tile file.

Once loaded, tiles will be placed in the TILE WINDOW. A window will prompt you for the dimensions of the map to be created. Please refer to MAP SIZE for more details on this window. Tile #0 is used to fill the background on any new map. The scrolling library in WGT does not like NULL pointers, so you must make sure that tile #0 actually contains some sort of data. If you want the tile to appear empty, simply use a solid color for the entire tile (such as black). When making parallax maps, it is assumed that tile #0 will not be shown for parallax layers, regardless of what image it contains.

5.6 Load Objects

The Map Maker provides an easy system for initial object placement on your maps. If your program had several hundred (or thousand) objects to place within the "world" once a map was loaded, the code would be immense. To avoid this, the Map Maker allows you to place up to 2000 objects on the map in their starting positions. Objects may be ANY size up to 320*200 pixels. Create an object file by drawing your objects in the Sprite Editor and saving the file as a normal SPR file.

Once a file has been selected, the Map Maker will load the images and place them in the OBJECT WINDOW. To see how much memory is available at any time, use the MEMORY STATUS menu item in the MISC dropdown.

5.7 Load Map

A previously saved map file can be reloaded using this option. Keep in mind that each map file is designed for a specific set of tiles and objects. If you load a map file which does not belong with the currently loaded images you may get undesirable results.

Once loaded, the MAP window will display the contents of the map itself. If the OBJECT bank is active, objects will be superimposed on the map tiles. See OBJECT MODE for further information on this feature.

All existing tile types, map data, and object positions will be overwritten with the information in the map file that has been loaded. Be sure to save any existing data before loading a new map. For file specifications, see the PROGRAMMER'S INFO section.

5.8 Save Map

When selected, this menu item will present the user with a file selector. Files with the extension WMP will be listed. You may either select one of the existing files to save over, or you may enter an entirely new filename in the text field at the top of the selector. WMP stands for W(GT) MaP. You do not have to use this extension, but the selector will always default to WMP files.

All information regarding the map itself will be stored within the file. This includes dimensions, tile data, object placement and tile types. See the PROGRAMMER'S INFO section for file specifications.

5.9 Quit

Once selected, the Map Maker will end execution and return the user to DOS. All information which has not been saved at this point will be lost. This menu item must be used to end the program and return the computer to its original state (the state in which the Map Maker was started).

6.0 Template Menu

6.1 Load

Templates which have been saved using the SAVE menu option can be reloaded at any time by using this option. Template files do not store information regarding the necessary tile file, so be sure to have the proper tiles loaded before selecting LOAD.

The normal extension for template files is TPL. This is optional, and you may choose to name your files with any extension you wish. Once loaded, the template will be placed in the template window with the appropriate regions selected. The user is automatically placed in PASTE mode. All previous template data in memory will be lost.

6.2 Create

Before a template can be used, you must create one by highlighting the desired tiles in the template window. After placing the tiles in the template window (just as you would in the map window), select CREATE to start identifying the tiles to be used in template construction. Clicking the left mouse button will highlight (select) a tile. Clicking the right mouse button will deselect a tile. Once you have selected all the tiles for a given template, you are ready to try the USE menu option.

If you want a large area highlighted, it may be easier to select the tiles with the GRAB REGION menu option.

Selecting CREATE will clear out existing highlights. Be sure not to choose this menu option if you are already building a template.

6.3 Reset

If you are finished with an existing template and wish to create a new one, this menu option will clear out the previous template and prepare for a new one. The tiles in the template window are not cleared, but the highlights are cleared and any template being used is freed from memory.

6.4 Use

After highlighting the desired tiles with the CREATE option, this command will build the template in memory and place the user in PASTE mode. The lower toolbar will display a text notice indicating that you are in PASTE mode, and the quick-pick banks will be hidden. To leave the USE mode at any point, select the EXIT PASTE menu option. Paste the template on the map just as you would a regular tile (by clicking the left mouse button).

6.5 Exit Paste

When in template USE mode, the screen will contain a notice along the bottom toolbar. Selecting this menu option will return the user to either object mode or edit mode (depending on the mode in use before templates were activated). Return to PASTE mode by selecting USE again.

6.6 Save

Save both the template data and the highlighted template tiles by selecting this menu option. Files normally use the TPL extension, but you may change this to any extension you want. Information regarding the tile file is not saved, so you must remember which template files go with which tile files. It's a good idea to name the template files similar to the tile files to keep things simple.

6.7 Copy Map

This option will copy the entire map into the template window. You may then alter the tiles or highlight them to build a template. This makes it easy to build templates from existing areas of the map.

6.8 Grab Region

Highlighting tiles one at a time can be very time-consuming. This menu option will put the user into a reduced-template mode. The screen will look exactly like a reduced map except that it is using the data from the template window. Click the left mouse button to select a corner of a region to highlight. Click the left button again to select the second corner. The area selected will become white, indicating that those tiles have been highlighted. Click the right mouse button to return to the main program.

This option allows quick selection of large template regions. Each time you choose this menu item, the previous selections are kept, allowing you to build on the existing template highlights.

7.0 Window Menu

7.1 Tile Window

Once you have loaded a tile file (in the FILE dropdown), the TILE window will now contain all the tiles loaded. There are a maximum of 65536 tiles permitted in any given tile file. You should note that the Sprite Editor only allows for 2000 images to be stored in a sprite file. If you need more, you can combine multiple sprite files into one by writing your own program which uses the wloadsprites and wsavesprites commands. When the mouse is positioned over a tile, you may click on it with either the left or right mouse button. This will assign that tile to the button you pressed, and the tile is now displayed in one of two boxes along the control bar at the bottom of the screen. Selecting the tile from the control bar will allow you to store it in the quick-pick bank (see Quick Picks later in this document). Use the window scrollbars to access the tiles which are out of view.

7.2 Map Window

The MAP window will contain all tiles and objects which have been placed by the user. Initially, this window is filled with tile 0, but it may contain info once a map file is loaded or the user "pastes" tiles or objects onto it.

When the control bar is displaying the active tiles and the tile quick-pick bank, the user is placed in map edit mode. When the mouse button is clicked within the window, the appropriate tile (for that button) is copied into the map. A highlighted box appears to show you exactly where the tile will be placed. There is no ERASE feature built into the program because all 65536 tiles may be used, so it is recommended that you have a tile which is a solid color (usually black) which may be used to erase mistakes.

If the control bar is displaying the active object and the object quick-pick bank, the user is placed in object mode. When the mouse button is clicked within the window, the active object is "pasted" onto the map at that location. If you wish to "lock" the positions into a grid which matches the tile size, press 'G' to activate grid-lock (or again to de-activate). Up to 2000 objects may be stored on a single map, and it can get confusing trying to locate or modify existing objects when there are a lot. To avoid confusion, you may want to see the object numbers superimposed on the screen (do so by pressing 'N', or 'N' again to de-activate). If you want to manually store/locate objects, use the OBJECT MENU.

Pressing 'O' will activate the OBJECT MENU, or you may also find the option in the MISC dropdown. This menu displays the entire structure which is stored for each object, and features several buttons which make life easier for the developer. See OBJECT MENU later in this documentation for more info.

7.3 Object Window

After loading an object file (in the FILE dropdown), the OBJECT window will appear with all the objects displayed in the window's client area. To select an object from the window, position the mouse over the object and click with either button. Only one object may be selected at any given time (unlike TILE mode which allows two selected tiles). Once again, you may use the window scrollbars to access items which are out of view. Selecting the object displayed on the control bar will allow you to place the object in the quick-pick bank (see Quick Picks later in this document).

7.4 Template Window

The template window and the map window behave in a similar fashion. Tiles may be placed in the template window just like you do in the map window. The window resolution is 320*200 tiles, allowing for large templates to be designed. All template operations are performed in this window. See TEMPLATE MENU for more information.

9.0 View Menu

9.1 Supported View Modes

When you have a map created in memory, it might be hard to visualize what that map looks like in a larger view area than the window. To provide a better viewing mode, you may select one of several supported video modes from the VIEW menu.

Since WGT uses 320*200*256 (all VGA cards support this mode), this option is always available from the VIEW menu. When selected, you will see the map as a full screen presentation with actual tile sizes. The map will look exactly like this when used in your

programs. If you want to see a larger area of the map at once, you can view it in a higher resolution.

Higher resolutions are supported through VESA. If your monitor and video card support Super-VGA (SVGA) modes, you may install a VESA driver in your AUTOEXEC file. Most video cards come with software drivers for this support. Supported video modes will depend on the card manufacturer, model, and memory size. The Map Maker will detect which modes are available and will present these along with the default 320*200 mode in the menu. Simply select the mode you want and follow the same process from here.

Once in a viewing mode, the map will be displayed (along with all currently placed objects) and the mouse cursor is hidden. Use the grey cursor keys to maneuver your way around the map. If you reach the edge of the map (in any direction), the key will no longer respond.

Supported keys in view mode:

LEFT ARROW	-	Move viewpoint left one tile.
RIGHT ARROW	-	Move viewpoint right one tile.
UP ARROW	-	Move viewpoint up one tile.
DOWN ARROW	-	Move viewpoint down one tile.
PAGE UP	-	Move viewpoint up half a screen.
PAGE DOWN	-	Move viewpoint down half a screen.
CTRL LEFT ARROW	-	Move viewpoint left half a screen.
CTRL RIGHT ARROW	-	Move viewpoint right half a screen.
O	-	Toggle object display.
ENTER or mouse button	-	Exit map view mode.

9.0 Options Menu

9.1 Reduced Map

A reduced map option has been implemented just as in the previous version of the Map Maker (v4.2). This feature will allow you to display the map using 1 pixel to represent each tile. The system will make its best guess as to which color to use to represent the tiles. The mouse cursor will become a rectangular shape with the same dimensions as the MAP window (reduced).

Simply move the cursor over the area of the map you wish to edit and click with either button. The cursor MAY be extremely small if only a few tiles can be displayed within the MAP window. This happens when the window is small or the tiles are large. Do not be alarmed if you can't see the cursor at first, because it is there!

9.2 Tile Types

Each of the first 256 tiles may be assigned a number from 0-255 which will indicate a property or "type". For example, all tiles which are assigned a value of 1 may indicate a solid tile (like ground or walls), while all tiles which are assigned a 0 will indicate sky pieces or transparent tiles. This option presents you with a dialog box in which you may change the assigned values.

Click on the current tile number to change to another tile. Click on the NEW TILE value to set the type for the current tile. Choose QUIT to leave this dialog when you are done. Tile types are saved with the MAP file, so you must remember to save the current map after changing tile types. Applications which require more than 256 tile types (or type settings for tiles beyond 256) should store them in an array using their own code instead of relying on the Map Maker to do so.

9.3 Object Menu

Objects may be placed on the map by switching to the object quick-pick bank and simply pasting objects on the map window. This is known as "object mode". While in object mode, you may toggle numerical display by pressing N. If active, this will display the object numbers superimposed on the upper-left corner of the actual objects within the map window. This makes it easier to determine which object is which. You may increase or decrease the current object image used for placement by pressing the plus or minus keys. This changes the image number, not the object number. Objects may use positions up to 2000 in this version of the Map Maker. Managing such a large number of objects can be difficult, so an object menu has been implemented.

When the MAP window is active and the object quick-pick bank is being used, the object menu is available to the user. This menu provides total control over the object image numbers, positions and display status.

The object information structure for the current object is displayed in the dialog box. You may alter any of the figures by clicking on the number and typing in a new one. Object numbers range from 0-2000, on/off status must be 0 or 1, x and y coordinates may be anywhere from -32768 to -32767, and the image number is based on the object file loaded.

To move to another structure, use the PREV or NEXT buttons. To find the nearest empty structure, use one of the FIND buttons. The leftmost FIND button will search below the current object number, and the rightmost button will search above the current object number. Use the DELETE button to erase the current object structure or the DELETE ALL button to erase ALL object structures in memory. If you want to see where the current object is on the map, choose the GOTO button. The lower right-hand corner of the dialog will show the image used for the current object structure. Change the image quickly by clicking on the plus or minus buttons instead of typing in a new value.

9.4 Map Size

This option will allow you to set (or change) the dimensions of the map you are creating. Dimensions are displayed in units of tiles. With tile sizes up to 64*64, your coordinate system can go as high as (320*64) by (200*64) or 20480 by 12800 pixels. Current dimensions are displayed in the middle of the dialog box. Click on the number you wish to change with the mouse button. Type in the new number and press ENTER. When you are done (even if you didn't change anything) choose QUIT. If you have been editing a map with larger dimensions and decide to shrink the map, the extra data will not be lost until you leave the Map Maker. Later on during your session you may expand the map size again and recover the extra data.

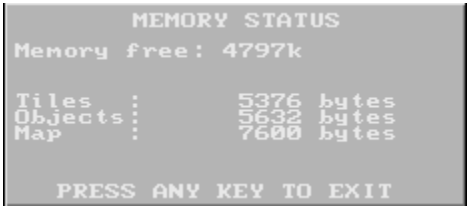
9.5 Export PCX

Sometimes it is desirable to create screen captures of your programs to use as advertisements. The PCX export feature will allow you to output the entire map (currently in memory) as a compressed image file. PCX is a standard image format which was introduced by PC Paintbrush for Windows. Even with the compression technique used, image files may be very large when created with this feature. Make sure you have enough room on your hard drive before you select this option. Once selected, the program will prompt you for a filename to save the image under, and then it will display its progress as each line of the output image is saved.

Once created, the PCX file can be altered using any of hundreds of available image processing programs or loaded using the WGT system library. Most commonly the pictures will become part of an advertisement for your game.

10.0 Miscellaneous Menu

10.1 Memory Status



This is one of the most important features of the system. It will tell you exactly how much memory is used and available.

The dialog box which appears will display the total amount of system ram free, and the total memory used by tiles, objects, and map data. When you are done checking the totals, press any key or click a mouse button to close the dialog box. If you notice that memory levels are low, it is highly recommended that you save all current data. The program may crash if it attempts to allocate memory which does not exist.

10.2 Disk Space

This option will allow you to see how much space is available on any of the valid hard drives installed on your system. The dialog displays info such as:

Click on NEXT to advance to the next logical drive available. Drive listings will loop around once you reach the last available drive. Click on QUIT to leave this dialog box at any time. Floppy drives are not available with this feature.

Do not attempt to save files on a disk which does not have enough space. Check the MEMORY STATUS dialog to see roughly how much memory is required by the map itself. You can expect up to 14k extra to be added to this total when saving (depending on the number of objects stored with the map data).

10.3 Clear All

If you wish to reset the Map Maker and start over, this is the menu item which will let you do it. This option will present you with a dialog box containing text only. It will instruct you to press "Y" to restart the Map Maker, or any other key to abort the reset (mouse clicks abort too). Any changes made to the files loaded will be lost. All items in memory are cleared and the program will start over from the very beginning. You should see the text screen again with the program statistics, followed by the menubar and an empty window area. At this point you are free to start fresh.

10.4 Create Source

Once a map has been created, this option will allow the user to save C source code for a WGT program which will instantly give life to the image. This option will not work unless the program has filenames for the tiles, objects and map. If you have created a map from scratch and haven't saved it, you must do so before attempting to use this. If you do not have any objects loaded you should do so (if you don't want objects, you still have to load in a dummy sprite file to get this option to work).

The program will present you with a screen which allows you to customize the code which is produced. Available options are window width/height, x and y location of the window, and the initial viewpoint of the window. Window (in this paragraph) refers to the scrolling window used in the WGT program, not the a window in the Map Maker. Window dimensions are based on tiles, not pixels, and the same goes for the initial viewpoint. X and Y locations are screen coordinates for the window (based in pixels).

To set the window dimensions or location, click on the button displayed and then use the mouse to select the positions on the screen. The Map Maker will allow values based on tile and map dimensions, therefore simplifying the user's input process. Clicking the mouse and dragging it to the second corner is all that's required for setting dimensions. One click is all that's needed for the window X and Y location.

An initial viewpoint is selected using a reduced map mode. The map is drawn on the screen and you simply click the mouse button when the cursor is highlighting the desired location on the map.

The EXIT button will abort code generation, and the GENERATE button will save source code based on the current settings. A file selector will prompt you for the filename to save.

11.0 Quick Picks

A new feature to the Map Maker is the addition of "Quick-Pick" banks. These banks are displayed along the bottom control bar. In one bank, the user may store the most commonly used tiles (for editing), and in the other bank objects may be stored. Instead of constantly moving to the tile window, selecting two tiles for editing, moving back to the map window and pasting tiles, it's much easier to use the quick-pick bank.

When the tile window is active, select a couple of tiles (one for each mouse button) and then move the cursor over the two leftmost boxes on the lower control bar. Clicking on one of the tiles will then place your cursor into the quick-pick bank. Move the cursor to the slot in which you want to store the tile, then click with the left mouse button. Clicking with the right button will cancel this action.

You may switch between quick-pick banks at any time by clicking the mouse button on the gray area of the lower toolbar. This area is located between the actual quick-pick slots and the active tiles (or object). This is not effective during template paste mode.

12.0 Programmer's Info

12.1 Map File Format

For those who want to attempt their own editor, or to access the map files from a different library, here's the map file format:

NAME	SIZE	Value	Meaning
MAGIC	2 bytes	8975	Magic number for v5.1 files
WIDTH	2 bytes	20-320	Width of map (in tiles)
HEIGHT	2 bytes	10-200	Height of map (in tiles)
MAPDATA	WID*HGT	0-255 each	Actual map data (stored in rows)
TILETYPES	256 bytes	0-255 each	Tile types for all 256 tiles
TOTALOBJS	2 bytes	0-2000	Number of object structures stored
OBJECTS	7 byte struct		One for each object in file

```
typedef struct {                        Object structure
        char on;                        Sprite is turned on=1
        short x;                        World x coordinate
                                         (max 320*64=20480     )
        short y;                        World y coordinate
                                         (max 200*64=12800)
        unsigned short num;             Sprite # from object file to show
} objects;
```

12.2 Project File Format

```
typedef struct {                        This defines a rectangular region
        short   x;
        short   y;
        short   x2;
        short   y2;
} coord;
```

The following structure is saved as the project file:

```
struct {
        unsigned char   title[20];Project file header string
        unsigned char   tilepath[80];     Path to TILE file
        unsigned char   mappath[80];      Path to MAP file
        unsigned char   objpath[80];      Path to OBJECT file
        coord           tile;             Window coordinates (4 windows)
        coord           map;
        coord           object;
        coord           template;
        char            winorders[10];    Window priorities
        short           picks[18][3];     Quick pick banks
        unsigned char   activebank;       Currently active quick pick bank
} PROJECT_FILE;
```

Notes: 3 quick-pick banks are saved, but only 2 are currently used
10 window priorities are saved, but only the first 4 are used

WGT Reference Manuals

WGT5_WC.LIB

This is the main WGT system library. It provides the user with everything from VGA detection routines to texture-mapped polygons. Every program using WGT must link this library file.

getmemfree

Function: Returns the number of bytes available to the program.

Declaration: **int getmemfree (void)**

Remarks: Use this function to determine if there is enough free memory for a malloc call or any other dynamic requirements during program execution. Each machine will have a different upper limit based on the amount of memory installed. Values returned may be as high as 2 gigabytes.

Parameters: None

Return Value: The number of bytes free.

Examples: 34, 63

installkbd

Function: Installs a keyboard handler which permits multiple keypresses at the same time.

Declaration: **void installkbd (void)**

Remarks: Once installed, this keyboard handler will store the status of all keypresses in an array of 128 short integers. The array is called kbdon. Each element of the array is either a 0 or a 1. Each key on the keyboard is assigned a scancode which is represented by the same element in the array. To determine if a key has been pressed, simply check to see if the corresponding element of kbdon is equal to 1.

You must uninstall the keyboard handler before attempting any normal keyboard input routines or before leaving the program. To determine the scancodes of the keys you want, refer to a DOS programmer's manual, or run the included SCANCODE.EXE utility.

A variable called keypressed is increased each time a key is pressed. You may set this variable to 0 and then wait until it reaches 1 (or more) to determine if a key has been pressed. At this point you may check the kbdon array to determine which keys have been pressed. Reset the variable to 0 again after checking the array.

Parameters: None.

Return Value: None.

See also: **uninstallkbd**

Examples: 56, 57, 58, 59, 62, 63, 65, 67

lib2buf

Function: Allocates memory for a file, and loads it into memory.

Declaration: **void *lib2buf (char *filename)**

Remarks: This allows any type of file to be loaded into memory from the WGT Data Library file. It gets the required memory first, and then returns a pointer to the data. It can be used for many things, such as text, digital sound files, calculated tables, and any other data you might need for the duration of your program..

What is done with the data loaded is up to you. If the data is text, you will need to process it from the pointer into a usable form.

Parameters: filename - The name of the file inside the WGT Data Library file which will be loaded into memory.

Return Value: A pointer to the data which was loaded.

See Also: **setlib, setpassword**

Examples: 31

mdeinit

Function: Removes the custom mouse handler.

Declaration: **void mdeinit (void)**

Remarks: WGT installs a custom mouse handler when the mouse is initialized with **minit**. This routine MUST be called before your program ends when using the mouse routines. If the custom mouse interrupt is not removed, moving the mouse when inside a rodent-free application will usually lock the computer.

Parameters: None

Return Value: None

See Also: **minit**

Examples: 12, 13, 14, 16, 18, 19, 20, 22, 23, 25, 30, 34, 36, 37, 39, 40, 47, 60, 61, 64, 66

minit

Function: Initializes the mouse driver.

Declaration: **short minit (void)**

Remarks: This command initializes the mouse driver. It must be a Microsoft compatible driver. Other drivers may behave strangely using the standard mouse calls. You must call this first if you want to use the mouse. Initially the mouse cursor is not visible. Use **mon** to turn on the display of the cursor. You should also note that all mouse routines work in graphics and text mode.

Upon initialization, a custom mouse interrupt is installed. Whenever an action occurs with the mouse, such as it being moved, or a button being depressed, the interrupt is called.

Within the interrupt, the variables **mouse.mx**, **mouse.my**, and **mouse.but** are updated to contain the information about the mouse. **mouse.mx** and **mouse.my** are the coordinates on the screen. **mouse.mx** is in the range 0 to 320, and **mouse.my** ranges 0 to 200.

If you are using the mouse routines in text mode, they will still report values in the ranges mentioned above. To change the values into character values, you must divide the coordinates by a number. For example, if you are in 80 column mode, the actual character column would be **mouse.mx**/4, since 320/4 gives you 80 possible values. Also, make a copy of the **mouse.mx** and **mouse.my** variables and divide them instead. This is because **mouse.mx** and **mouse.my** may change at any time from the custom mouse interrupt.

The **mouse.but** variable contains information for all of the buttons. Each button is represented by a bit within this number.

Left button: bit 1 (value of 1)
Right button: bit 2 (value of 2)
Middle button bit 3 (value of 4)

Therefore to see if a certain button is depressed,

```
if (mouse.but & 1)  button=LEFT;
if (mouse.but & 2)  button=RIGHT;
if (mouse.but & 4)  button=MIDDLE;
```

You can create defines for each button to make your code easier to understand.

Parameters: None

Return Value: Number of buttons on the mouse if the mouse driver is found.
If the number is less than 1, no mouse driver is present.

See Also: **mdeinit, moff, mon, mouseshape, msetbounds, msetspeed, msetthreshhold, noclick**

Examples: 12, 13, 14, 16, 18, 19, 20, 22, 23, 25, 30, 34, 36, 37, 39, 47, 60, 61, 64, 66

moff

Function: Turns off the display of the mouse cursor.

Declaration: **void moff (void)**

Remarks: This routine is usually called when you are drawing to the visual screen, otherwise the mouse will disturb the drawing operation. This function does nothing if the mouse cursor is already turned off.

Parameters: None

Return Value: None

See Also: **minit, mon**

Examples: 12, 14, 16, 30, 34, 47, 64

mon

Function: Turn on the display of the mouse cursor.

Declaration: **void mon (void)**

Remarks: If the mouse is displayed while you are using some drawing procedures, it will destroy the background of the screen if you move it over the area being updated. Turn it off during all drawing procedures with **moff**, and restore it when finished drawing with **mon**.

Note: If **wgetpixel** is used on an area where the mouse cursor appears, it will return the color within the mouse cursor, not what is on the screen behind it.

Parameters: None

Return Value: None

See Also: **minit, moff**

Examples: 12, 13, 30, 34, 37, 40, 64

mouseshape

Function: Sets the bitmap shape and hotspot of the mouse cursor.

Declaration: **void mouseshape (short x_hotspot, short y_hotspot, unsigned short *mousebitmap)**

Remarks: This procedure sets the cursor hotspot as defined by the variables x_hotspot and y_hotspot. The hotspot is the place on the mouse cursor where the actual coordinates will be returned. Usually the hotspot is in the top left corner, but you are free to modify this depending on your mouse cursor shape. The actual bitmap information is stored in an array of unsigned short integers called mousebitmap. It is defined as follows:

The first 16 words are used to set the cursor shape, while the last 16 words set the mask. To create this monochrome bitmap data, use the WGT Sprite Editor's mouse cursor tool.

Parameters:

x_hotspot	- Hot Spot X position (-16 to 16).
y_hotspot	- Hot Spot Y position (-16 to 16).
mousebitmap	- A pointer to the screen and cursor masks.

Return Value: None

Examples: 13

msetbounds

Function: Sets the movement boundaries of the mouse cursor.

Declaration: **void msetbounds (short x, short y, short x2, short y2)**

Remarks: The cursor's movement will be restricted to the specified region after this routine is called. This routine is used to control the user's actions by limiting where they can click on the screen.

Parameters:

x	- The left X coordinate of the mouse boundary.
y	- The top Y coordinate of the mouse boundary.
x2	- The right X coordinate of the mouse boundary.
y2	- The bottom Y coordinate of the mouse boundary.

Return Value: None

Examples: 12, 13, 19, 20, 60, 61, 66

msetspeed

Function: Sets the speed of the mouse.

Declaration: **void msetspeed (short x_speed, short y_speed)**

Remarks: This routine sets the speed of the mouse in each direction. 10 is about average for both values. 1 is fast, 40 is slower. In other words, if you set the speed to be 40 for the x, and 1 for the y, you will have to move the mouse a greater distance horizontally to move the cursor one pixel. On the other hand, you can move the cursor very quickly up and down.

Mouse movement is measured in mickeys. This routine sets how many mickeys are required to move the mouse one pixel on the screen. One mickey is about 1/200 of an inch.

Parameters: x_speed - Horizontal mickeys per pixel.

y_speed - Vertical mickeys per pixel.

Return Value: None

Examples: 13

msetthreshhold

Function: Sets the speed the mouse must reach before it doubles its movements.

Declaration: **void msetthreshhold (short speed)**

Remarks: When the mouse is moved faster than this speed, all movements are doubled. This is used to control the sensitivity of the mouse.

Parameters: speed - Threshhold speed in mickeys per second.

Return Value: None

See Also: **msetspeed**

Examples: 13

msetxy

Function: Sets the screen location of the mouse cursor.

Declaration: **void msetxy (short x, short y)**

Remarks: You may set the location of the mouse cursor on the screen with this command. Use this whenever you want to force the user to move to a certain region of the screen.

Parameters: x - Horizontal screen coordinate.

y - Vertical screen coordinate.

Return Value: None

Examples: 37, 60, 61

noclick

Function: Waits until all buttons are released.

Declaration: **void noclick (void)**

Remarks: This routine waits in a loop until all the mouse buttons are released. It is useful if you want the user to click on a button, perform an action, and wait for another button click. Without this command, the user could hold down the button and the program will think you pressed it a bunch of times. Put this command in after you have found the user pressed a button.
For example:

```
if (mouse.but == 1)
{
  ctr++;            /* Keep track of how many times you press the left
button */
  noclick ();       /* Wait until user lets go */
}
```

Parameters: None

Return Value: None

See Also: **minit, moff, mon**

Examples: 13, 19, 20, 36

setlib

Function: Sets the WGT Data Library name.

Declaration: **void setlib (char *libraryname)**

Remarks: WGT has the ability to store all of your graphics or data files into one large library file. Use the utility called WGTLIB to manage your WGT Data Library files. These data library files are not the library files made by WLIB, but rather those created by WGTLIB. Set the name of the data library file in the beginning of your program, and all WGT commands which load something will use the library instead of individual files. Pass NULL as libname to disable library files.

Parameters: libraryname - Filename of the WGT Data Library file.

Return Value: None

See Also: **lib2buf, setpassword**

Examples: 31

setpassword

Function: Sets the WGT Data Library password.

Declaration: **void setpassword (char *password)**

Remarks: WGT library files can be protected by a password. If you choose the incorrect password, files cannot be loaded in. This prevents other WGT users from taking your graphics and sound files. Use NULL to indicate no password. If you change your library file partway through the execution of your program and it uses a different password, you should call **setpassword** immediately after **setlib**.

Parameters: password - The password used to access the currently open WGT Data Library file.

Return Value: None

See Also: **lib2buf, setlib**

Examples: 31

uninstallkbd

Function: Removes the WGT custom keyboard handler.

Declaration: **void uninstallkbd (void)**

Remarks: Any program which calls the **installkbd** command must at some point call this routine as well. If the program fails to remove the keyboard handler, strange things may happen once the program ends (usually the system locks up).

Parameters: None.

Return Value: None.

See Also: **installkbd**

Examples: 56, 57, 58, 59, 62, 63, 65, 67

vga256

Function: Initializes the 320x200x256 VGA/MCGA graphics mode.

Declaration: **void vga256 (void)**

Remarks: This command will change the video mode into mode 0x13, which is 320x200 pixels with 256 colors. Mode 0x13 is the most popular mode for games because it is very fast and easy to program. **vga256** must be called before any other drawing commands.

Upon initialization, **vga256** sets a pointer called abuf to the visual screen of the VGA card. abuf will contain a pointer to the current graphics page throughout your program.

Parameters: None

Return Value: None

See Also: **vgadetected, wgetmode, wsetmode**

Examples: 1-29

vgadetected

Function: Determines if a VGA card is available.

Declaration: **char vgadetected (void)**

Remarks: If a VGA card is found, this routine returns 1, otherwise your program should stop immediately since WGT's graphics commands will not function.

Parameters: None

Return Value: 1 = VGA card found
0 = VGA card not found

See Also: **vga256, wgetmode, wsetmode**

Examples: 1-29

wallocblock

Function: Allocates enough memory for an image of given dimensions and returns a pointer.

Declaration: **block wallocblock (short width, short height)**

Remarks: This call is exactly the same as the **wnewblock** command except for the fact that the memory is left uninitialized. This is useful when you want to create a virtual buffer but do not wish to copy the screen's contents into it yet. It is most commonly used when allocating virtual screens larger than the visual screen.

Parameters: width - Desired image width in pixels

height - Desired image height in pixels

Return Value: Pointer to allocated memory

See also: **wnewblock, wsetscreen**

Examples: 34, 63

wbar

Function: Draws a filled rectangle.

Declaration: **void wbar (short x1, short y1, short x2, short y2)**

Remarks: (x1,y1) defines the upper left corner of the bar, and (x2,y2) defines the lower right. The rectangle will be clipped to the clipping boundaries if needed.

Parameters:

x1	- The left X coordinate of the bar.
y1	- The top Y coordinate of the bar.
x2	- The right X coordinate of the bar.
y2	- The bottom Y coordinate of the bar.

Return Value: None

See Also: **wrectangle**

Examples: 4, 18, 27, 29, 42, 54, 55, 60, 61

wbezier

Function: Calculates the points of a bezier curve.

Declaration: **void wbezier (tpolypoint *rawpts, short numraw, tpolypoint *curvepts, short numcurve)**

Remarks: **wbezier** takes a list of polygon points and creates a smooth curve between them. It puts the points of the curve into another vertex list that can be used with WGT's polygon routines.

Parameters:

rawpts	- an array of initial data points.
numraw	- contains the number of data points in the rawpts array.
curvepts	- the array which contains the points of the bezier curve.
numcurve	- the number of points on the curve. Greater values of numcurve will produce smoother bezier curves. numcurve must always be greater than numraw.

Return Value: None

See Also: whollowpoly

Examples: 33

wbutt

Function: Draws a 3-dimensional button.

Declaration: **void wbutt (short x1, short y1, short x2, short y2)**

Remarks: **wbutt** draws a 3D button by using different colors for shading around the sides. It is useful for creating user interfaces which often require a screen which is appealing to the eye. This routine uses colors 253 to 255 for shading the button. These colors must be of decreasing intensity for the button to look correct.

Parameters:

x1	- The left X coordinate of the button.
y1	- The top Y coordinate of the button.
x2	- The right X coordinate of the button.
y2	- The bottom Y coordinate of the button.

Return Value: None

See Also: **wbar, wrectangle**

Examples: 4, 13, 20, 58, 59

wcircle

Function: Draws a circle with a given radius.

Declaration: **void wcircle (short x_center, short y_center, short radius)**

Remarks: **wcircle** draws a hollow circle using the given location and radius.

Parameters:

x_center	- The X coordinate of the center of the circle.
y _center	- The Y coordinate of the center of the circle.
radius	- The radius of the circle. It must be greater than 0.

Return Value: None

See Also: **wellipse, wfill_circle, wfill_ellipse**

Examples: 4, 8

wclip

Function: Sets the current clipping area for all graphics pages.

Declaration: **void wclip (short x1, short y1, short x2, short y2)**

Remarks: (x1,y1) defines the upper left corner of the clipping area, and (x2,y2) defines the lower right. All of WGT's drawing commands will draw within this boundary unless mentioned otherwise. Set the coordinates to the visual screen size to 'disable' clipping.

Parameters:

x1	- The left X coordinate of the clipping region.
y1	- The top Y coordinate of the clipping region.
x2	- The right X coordinate of the clipping region.
y2	- The bottom Y coordinate of the clipping region.

Return Value: None

Examples: 4, 8, 11, 12, 17, 32, 61

wcls

Function: Clears the currently selected graphics page with a specified color number.

Declaration: **void wcls (unsigned char col)**

Remarks: This command fills the current drawing page with the color given. Clipping is not performed.

Parameters: col - The color used to fill the screen.

Return Value: None

See Also: **wnormscreen, wsetscreen**

Examples: 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 17, 18, 20, 27, 31, 32, 33, 42, 46, 51, 58, 59, 62, 65, 67

wcolrotate

Function: Cycles the colors between two indices in a palette.

Declaration: **void wcolrotate (unsigned char start, unsigned char finish, short dir, color *pal)**

Remarks: This routine takes the colors from start to finish and shifts them once in the direction specified by dir. This effect can be used to simulate animation.

dir = 0, rotates up.
dir = 1, rotates down.

This function does NOT set the palette. You must call **wsetpalette** in order to see the changes. This allows you to define multiple rotation areas at a time, by several calls to **wcolrotate**.

Parameters:

start	- First palette entry in rotation.
finish	- Last palette entry in rotation.
dir	- Direction of rotation.
pal	- A pointer to the palette

Return Value: None

See Also: **wsetpalette, wsetrgb**

Examples: 5, 20, 27

wcopyscreen

Function: Copies a section of one block onto a different block.

Declaration: **void wcopyscreen (short x1, short y1, short x2, short y2, block source, short destx, short desty, block destination)**

Remarks: This command copies a region of one block to a new location. The source and destination blocks may be the same as long as the source and destination region do not overlap. It is commonly used to copy from one page to another.

The area defined by (x1,y1,x2,y2) is copied from the image source, and is placed with the upper left corner at (dx,dy) on the screen referred to by destination. For example, to copy screen second (entirely) to the screen first, type wcopyscreen(0,0,319,199,second,0,0,first).

The visual page can be accessed by replacing the destination or source with NULL.

For example to copy firstscr to the visual page:
```
wcopyscreen (0, 0, 319, 199, firstscr, 0, 0, NULL);
```
or the other way around:
```
wcopyscreen (0, 0, 319, 199, NULL, 0, 0, firstscr);
```

You should also note that in the last example, firstscr must already be allocated from another image function such as **wnewblock**, or **wloadblock**. It is not the same as firstscr = wnewblock (0, 0, 319, 199) because **wcopyscreen** assumes there is already memory allocated for firstscr.

You can also copy a region of one screen to a different location on another. For example,
wcopyscreen(30, 20, 180, 190, first, 60, 90, NULL);

Full clipping is performed, so you do not have to worry about the coordinates being completely within the destination screen. Source and destination images do not required the same dimensions.

Parameters: x1 - The left X coordinate of the source region.

y1 - The top Y coordinate of the source region.

	x2	- The right X coordinate of the source region.
	y2	- The bottom Y coordinate of the source region.
	source	- A pointer to the source image.
	destx	- The X coordinate in the destination region.
	desty	- The Y coordinate in the destination region.
	destination	- A pointer to the destination image.

Return Value: None

Examples: 16, 18, 19, 20, 23, 25, 29, 42, 54, 55, 60, 61

wdeinitpoly

Function: Deinitializes the WGT polygon system and frees internal buffers.

Declaration: **void wdeinitpoly (void)**

Remarks: Every program which uses a polygon routine must call this before it exits. Internal buffers are deallocated and memory is returned to the heap space. This call must be matched with a previous call to **winitpoly** or else the program will crash.

Parameters: None

Return Value: None

See Also: **winitpoly**

Examples: 42, 51, 54, 55

wdeinit_triangle_renderer

Function: Deinitializes the WGT triangle rendering system and frees internal buffers.

Declaration: **void wdeinit_triangle_renderer (void)**

Remarks: Every program which uses a triangle routine must call this before it exits. Internal buffers are deallocated and memory is returned to the heap space. This call must be matched with a previous call to **winit_triangle_renderer** or else the program will crash.

Parameters: None

Return Value: None

See Also: **winit_triangle_renderer, wtriangle_flat_shaded_texture, wtriangle_gouraud, wtriangle_gouraud_shaded_texture, wtriangle_solid, wtriangle_texture, wtriangle_translucent_gouraud, wtriangle_translucent_texture**

Examples: 3D_CAM

wdissolve

Function: Dissolves a block onto the active display page, with a user-defined pattern.

Declaration: **void wdissolve (block sourceimage, short *pattern, short speed)**

Remarks: The block described by sourceimage is dissolved onto the active screen using the pattern specified by pattern. Pattern is an array of short integers. It can be defined by the program called dissolve.exe, included with WGT. After saving a pattern with this program, include the code generated into your program, and pass the array to this procedure. The speed is the amount to delay after each pixel is copied.

Parameters:

sourceimage	- Pointer to image which will replace active screen after dissolve.
pattern	- Array of integers used to define fade pattern.
speed	- Delay in milliseconds between pixel updates.

Return Value: None

Examples: 17

wdonetimer

Function: Removes the custom timer interrupt.

Declaration: **void wdonetimer (void)**

Remarks: This command will reset the speed of the system timer to its original speed, and remove the custom timer interrupt.

Parameters: None

Return Value: None

See Also: **winittimer, wsettimerspeed, wstarttimer, wstoptimer**

Examples: 3, 11, 46, 55, 56, 57, 58, 59, 61, 62, 63, 68

wdraw_scanpoly

Function: Draws a concave polygon which was previously scan converted with the **wscan_convertpoly** routine.

Declaration: **void wdraw_scanpoly (short x, short y, wscanpoly *scanpoly, void (*customline)(short x1, short x2, short y))**

Remarks: The polygon must have been scan converted by the **wscan_convertpoly** routine before you call this routine.

X and Y are added to each vertex before drawing the polygon, allowing you to move the shape to a different location.

customline is a pointer to a function which will draw a horizontal line. If you set this to NULL, it will use the whline routine. customline accepts three parameters, the first two being the x coordinates of the line. The third is the y coordinate of the horizontal line.

You can write your own horizontal line routines that don't fill the line with a solid color. Some possibilities include plasma, wall papering, or fill patterns.

Parameters:

x	- Horizontal offset added to each vertex.
y	- Vertical offset added to each vertex.
scanpoly	- the scan converted polygon structure containing the polygon to draw.
customline	- A pointer to a function which draws a horizontal line.

Return Value: None

See Also: **wdeinitpoly, wfree_scanpoly, winitpoly, wscan_convertpoly**

Examples: 51

wellipse

Function: Draws a hollow ellipse.

Declaration: **void wellipse (short x_center, short y_center, short x_radius, short y_radius)**

Remarks: Draws a hollow ellipse with center (x_center,y_center) using radius (x_radius,y_radius). Both x_radius and y_radius must be greater than 0.

Parameters:

x_center	- Center of ellipse along horizontal axis.
y_center	- Center of ellipse along vertical axis.
x_radius	- Horizontal radius in pixels.
y_radius	- Vertical radius in pixels.

Return Value: None

See Also: **wcircle, wfill_circle, wfill_ellipse**

Examples: 4

wfade_between

Function: Gradually fades in a group of colors from one palette to another palette using the specified speed.

Declaration: **void wfade_between (unsigned char start, unsigned char finish, short speed, color *pal1, color *pal2)**

Remarks: The colors from start to finish are faded from pal1 to pal2. The speed may range from 0 to 32767, and is similar to using the DELAY command between each change of the colors.

Parameters:

start	- Start index of palette array for fade range.
finish	- End index of palette array for fade range.
speed	- Delay in milliseconds between color updates.
pal1	- Starting palette.
pal2	- Ending palette.

Return Value: None

See Also: **wfade_between_once, wfade_in, wfade_in_once, wfade_out, wfade_out_once**

Examples: 49

wfade_between_once

Function: Fades in a group of colors from one palette to another, performing only 1/64th of the fade operation.

Declaration: **void wfade_between_once (unsigned char start, unsigned char finish, color *pal1, color *pal2)**

Remarks: The colors from start to finish are faded from pal1 to pal2. This routine only performs one step of the fade to allow other operations to be performed while the colors are fading. It must be called at least 64 times to complete the fade between the palettes. The palette is not set by this routine, so you must use **wsetpalette** to do so. After this is called 64 times, pal1 contains pal2, so be sure to preserve pal1 in another palette array if you don't want to lose it.

Parameters:

start	- Start index of palette array for fade range.
finish	- End index of palette array for fade range.
pal1	- Starting palette.
pal2	- Palette which will result from fade.

Return Value: None

See Also: **wfade_between, wfade_in, wfade_in_once, wfade_out, wfade_out_once**

Examples: 49

wfade_in

Function: Gradually fades in a group of colors from black to a palette variable using a specified speed.

Declaration: **void wfade_in (unsigned char start, unsigned char finish, short speed, color *pal)**

Remarks: The colors from start to finish are faded in starting from black. The speed may range from 0 to 32767, and is similar to using the DELAY command between each change of the colors.

Parameters:

start	- Start index of palette array for fade range.
finish	- End index of palette array for fade range.
speed	- Delay in milliseconds between color updates.
pal	- Colors will fade from black to the ones contained in this array.

Return Value: None

See Also: **wfade_between, wfade_between_once, wfade_in_once, wfade_out, wfade_out_once**

Examples: 6, 20

wfade_in_once

Function: Fades in a group of colors from black to a palette variable, performing only 1/64th of the fade operation.

Declaration: **void wfade_in_once (unsigned char start, unsigned char finish, color *pal, color *temppal)**

Remarks: The colors from start to finish are faded from black to pal. This routine only performs one step of the fade to allow other operations to be performed while the colors are fading. It must be called at least 64 times to complete the fade. temppal is a temporary palette array which holds the initial values for the fade operation. Usually it would contain a totally black palette. The palette is not set by this routine, so you must use **wsetpalette** to do so.

Parameters:

start - Start index of palette array for fade range.

finish - End index of palette array for fade range.

pal - Colors will fade from black to the ones contained in this array.

temppal- A temporary buffer which is filled with black colors.

Return Value: None

See Also: **wfade_between, wfade_between_once, wfade_in, wfade_out, wfade_out_once**

Examples: 6

wfade_out

Function: Gradually fades a group of colors from a palette variable to black, using a specified speed.

Declaration: **void wfade_out (unsigned char start, unsigned char finish, short speed, color *pal)**

Remarks: The colors from start to finish are faded out to black starting from the palette given. The speed may range from 0 to 32767, and is similar to using the DELAY command between each change of the colors.

Parameters:

start	- Start index of palette array for fade range.
finish	- End index of palette array for fade range.
speed	- Delay in milliseconds between color updates.
pal	- Palette will fade from these colors to black.

Return Value: None

See Also: **wfade_between, wfade_between_once, wfade_in, wfade_in_once, wfade_out_once**

Examples: 6, 64

wfade_out_once

Function: Fades out a group of colors from a palette variable to black, performing only 1/64th of the fade operation.

Declaration: **void wfade_out_once (unsigned char start, unsigned char finish, color *pal)**

Remarks: The colors from start to finish are faded from pal to black. This routine only performs one step of the fade to allow other operations to be performed while the colors are fading. It must be called at least 64 times to complete the fade. The palette is not set by this routine, so you must use **wsetpalette** to do so.

Parameters:

start	- Start index of palette array for fade range.
finish	- End index of palette array for fade range.
pal	- Colors will fade from this palette to black.

Return Value: None

See Also: **wfade_between, wfade_between_once, wfade_in, wfade_in_once, wfade_out**

Examples: 6

wfastputpixel

Function: Plots a pixel at the given screen coordinate without clipping.

Declaration: **void wfastputpixel (short x, short y)**

Remarks: This routine operates the same as wputpixel only it ignores clipping. This is the fastest pixel plotting method.

Parameters: x - Pixel coordinate along horizontal axis.

y - Pixel coordinate along vertical axis.

Return Value: None

See Also: **wgetpixel, wputpixel**

Examples: 2, 39, 51

wfill_circle

Function: Draws a filled circle given a center and radius.

Declaration: **void wfill_circle (short x, short y, short radius)**

Remarks: (x,y) is the center of the circle to be drawn. Radius is the size in pixels horizontally. It must be greater than 0.

Parameters:

x	- Pixel coordinate along horizontal axis.
y	- Pixel coordinate along vertical axis.
radius	- Radius of circle in pixel units.

Return Value: None

See Also: **wcircle, wellipse, wfill_ellipse**

Examples: 4, 10, 12, 18, 27

wfill_ellipse

Function: Draws a filled ellipse given a center, a horizontal radius, and a vertical radius.

Declaration: **void wfill_ellipse (short x_center, short y_center, short x_radius, short y_radius)**

Remarks: Draws a filled ellipse with center (x_center,y_center) using radius (x_radius,y_radius). Both x_radius and y_radius must be greater than 0.

Parameters:

x_center	- Center of ellipse on horizontal axis.
y_center	- Center of ellipse on vertical axis.
x_radius	- Radius of ellipse along horizontal axis in pixels.
y_radius	- Radius of ellipse along vertical axis in pixels.

Return Value: None

See Also: **wcircle, wellipse, wfill_circle**

Examples: 4

wflashcursor

Function: Flash the cursor using the values set by **wsetcursor** and the global variable curspeed.

Declaration: **void wflashcursor (void)**

Remarks: This routine uses the cursor coordinates set by the global variables xc and yc. Curspeed is another global variable which can be set to change the flashing speed.

wflashcursor will display the software emulated cursor, delay according to curspeed, erase the cursor, delay again, and then return. It will erase the cursor and return immediately if a key is pressed.

Parameters: None

Return Value: None

See Also: **wstring**

Examples: 14

wfline

Function: Draws a line between two points, without clipping.

Declaration: **void wfline (short x1, short y1, short x2, short y2)**

Remarks: This routine is similar to **wline**, except that no clipping is performed on the line. This results in a slightly faster routine, although care must be taken to ensure that the endpoints are within screen boundaries.

Parameters:

	x1	- Horizontal coordinate of first endpoint.
	y1	- Vertical coordinate of first endpoint.
	x2	- Horizontal coordinate of second endpoint.
	y2	- Vertical coordinate of second endpoint.

Return Value: None

See Also: **wline, wstyleline**

Examples: 4, 16, 20, 25

wflipblock

Function: Flips a block in one of two directions. The block is physically changed in memory, not on the screen.

Declaration: **void wflipblock (block image, short direction)**

Remarks: This procedure will flip a previously stored image either vertically or horizontally. The stored image is physically changed, and therefore the results will not be seen until the block is displayed with **wputblock** or another image transfer routine.

direction is defined as the following:

vertical = 0
horizontal = 1

Parameters
image - Pointer to the image which will be flipped.

direction - Indicates axis along which the flip is performed.

Return Value: None

See Also: **wfreeblock, wnewblock, wputblock, wresizeblock**

Examples: 10, 19

wfree_scanpoly

Function: Frees memory used by a polygon buffer.

Declaration: **void wfree_scanpoly (wscanpoly *scanpoly)**

Remarks: When you use **wscan_convertpoly**, the polygon is stored in a buffer which must be deallocated before you use the buffer again. This routine will free the memory used by one of these scan converted polygons.

Parameters: scanpoly - Buffer in which the polygon was stored, and will now be deallocated.

Return Value: None

See Also: **wdeinitpoly, wdraw_scanpoly, winitpoly, wscan_convertpoly**

Examples: 51

wfreeblock

Function: Releases memory used to store an image.

Declaration: **void wfreeblock (block image)**

Remarks: When memory is allocated for a block, it is allocated dynamically using malloc. Most commands that return a block variable allocate this memory for you. To release this memory, use **wfreeblock**. As with malloc, you cannot use the same pointer twice in a row without freeing the memory.

For example, the following code would produce memory errors:

```
pic = wloadblock ("myblock.blk");
pic = wloadblock ("myblock.blk");
...
wfreeblock (pic);
```

It should read:

```
pic = wloadblock ("myblock.blk");
wfreeblock (pic);
pic = wloadblock ("myblock.blk");
...
wfreeblock (pic);
```

Parameters: image - Pointer to the image which is to be deallocated.

Return Value: None

See Also: **wallocblock, wnewblock**

Examples: 9, 10, 11, 15, 16, 17, 18, 19, 20, 21, 24, 31, 32, 33, 34, 35, 42, 47, 48, 49, 51, 53, 64, 65, 66

wfreefont

Function: Frees memory previously allocated for a font.

Declaration: **void wfreefont (wgtfont font)**

Remarks: When a bitmapped font is loaded with **wloadfont**, memory is allocated dynamically. This is similar in operation to the block commands. Use **wfreefont** to release the memory for the font.

Parameters: font - Pointer to font buffer which will be deallocated.

Return Value: None

See Also: **wloadfont**

Examples: 40, 41, 47

wfreesprites

Function: Frees memory from an array of blocks.

Declaration: **void wfreesprites (block *image_array, short start, short end)**

Remarks: When a sprite file is loaded, several blocks are allocated at once. This routine will free all of the blocks in the array from start to end. It should be noted that this routine will deallocate the images, but will not set the pointers to NULL. You must do this yourself if required.

Example usage:

```
block sprites[1001];
...
wfreesprites (sprites, 0, 1000);
```

Parameters:

image_array	- Array containing sprite pointers.
start	- Start index for block deallocation.
end	- End index for block deallocation.

Return Value: None

See Also: **wloadsprites**

Examples: 19, 20, 22, 23, 25, 29, 31, 45, 56, 58, 59, 62, 63, 65, 66, 67, 68

wget_capslock

Function: Returns the state of the Capslock key.

Declaration: **short wget_capslock (void)**

Remarks: This routine is used to report the state of the caps lock key. It cannot be used when the custom keyboard interrupt is installed.

Parameters: None

Return Value 0 = Capslock is off
1 = Capslock is on

See Also: **wset_capslock**

Examples: 44

wget_imagebit

Function: Retrieves a byte of information from the end of allocated memory from an image.

Declaration: **unsigned char wget_imagebit (block image)**

Remarks: This routine retrieves a value from the last allocated byte of memory for that image. The value returned indicates whether or not the first bit is set. The value indicates the following information:

If the result is 1, the image contains some pixels of color index 0. This means that an XRAY putblock would be effective to make the image see-through for parallax scrolling and other such effects. If the status is 0, a normal putblock would be the fastest way of blasting it to the screen. None of the images contain this status unless either **wupdate_imagebytes** or **wset_imagebyte** have been used previously.

Parameters: image - Pointer to the image in memory.

Return Value: Status bit

See Also: **wset_imagebyte, wget_imagebyte, wupdate_imagebytes**

Examples: None

wget_imagebyte

Function: Retrieves a byte of information from the end of allocated memory from an image.

Declaration: **unsigned char wget_imagebyte (block image)**

Remarks: This routine retrieves a value from the last allocated byte of memory for that image and it is recommended the the information is used in the following format:

Bit 0 - Indicates Xray or Normal block
Bits 1-7 are unused.

If bit 0 is set (status is 1), the image contains some pixels of color index 0. This means that an XRAY putblock would be effective to make the image see-through for parallax scrolling and other such effects. If the status is 0, a normal putblock would be the fastest way of blasting it to the screen. None of the images contain this byte unless either **wupdate_imagebytes** or **wset_imagebyte** have been used previously.

Parameters: image - Pointer to the image in memory.

Return Value: Status byte

See Also: **wset_imagebyte, wget_imagebit, wupdate_imagebytes**

Examples: None

wget_lalt

Function: Returns the state of the left Alt key.

Declaration: **short wget_lalt (void)**

Remarks: This routine is used to report the state of the left Alt key. It cannot be used when the custom keyboard interrupt is installed.

Parameters: None

Return Value: 0 = Left Alt is released
1 = Left Alt is depressed

See Also: **wget_ralt**

Examples: 44

wget_lctrl

Function: Returns the state of the Left Control key.

Declaration: **short wget_lctrl (void)**

Remarks: This routine is used to report the state of the left control key. It cannot be used when the custom keyboard interrupt is installed.

Parameters: None

Return Value: 0 = Left Control is released
1 = Left Control is depressed

See Also: **wget_rctrl**

Examples: 44

wget_lshift

Function: Returns the state of the Left Shift key.

Declaration: **short wget_lshift (void)**

Remarks: This routine is used to report the state of the left shift key. It cannot be used when the custom keyboard interrupt is installed.

Parameters: None

Return Value: 0 = Left Shift is released
1 = Left Shift is depressed

See Also: **wget_rshift**

Examples: 44

wget_numlock

Function: Returns the state of the Numlock key.

Declaration: **short wget_numlock (void)**

Remarks: This routine is used to report the state of the numlock key. It cannot be used when the custom keyboard interrupt is installed.

Parameters: None

Return Value: 0 = Numlock is off
1 = Numlock is on

See Also: **wset_numlock**

Examples: 44

wget_ralt

Function: Returns the state of the Right Alt key.

Declaration: **short wget_ralt(void)**

Remarks: This routine is used to report the state of the right Alt key. It cannot be used when the custom keyboard interrupt is installed.

Parameters: None

Return Value: 0 = Right Alt is released
1 = Right Alt is depressed

See Also: **wget_lalt**

Examples: 44

wget_rctrl

Function: Returns the state of the Right Control key.

Declaration: **short wget_rctrl (void)**

Remarks: This routine is used to report the state of the right control key. It cannot be used when the custom keyboard interrupt is installed.

Parameters: None

Return Value: 0 = Right Control is released
1 = Right Control is depressed

See Also: **wget_lctrl**

Examples: 44

wget_rshift

Function: Returns the state of the Right Shift key.

Declaration: **short wget_rshift (void)**

Remarks: This routine is used to report the state of the right shift key. It cannot be used when the WGT keyboard interrupt is installed.

Parameters: None

Return Value: 0 = Right Shift is released
1 = Right Shift is depressed

See Also: **wget_lshift**

Examples: 44

wget_scrlock

Function: Returns the state of the Scroll Lock key.

Declaration: **short wget_scrlock (void)**

Remarks: This routine is used to report the state of the scroll lock key. It cannot be used when the custom keyboard interrupt is installed.

Parameters: None

Return Value: 0 = Scroll Lock is off
1 = Scroll Lock is on

See Also: **wset_scrlock**

Examples: 44

wgetblockheight

Function: Returns height of a previously stored block.

Declaration: **short wgetblockheight (block image)**

Remarks: This routine returns the height of the block in pixels. If the block has not been allocated, the result is undefined.

Parameters: image - Pointer to image which will be measured.

Return Value: Height of the block, in pixels.

See Also: **wgetblockwidth, wnewblock**

Examples: 11, 19, 57, 64, 66, 67

wgetblockwidth

Function: Returns width of a previously stored block.

Declaration: **short wgetblockwidth (block image)**

Remarks: This function returns the width of the block in pixels. If the block has not been allocated, the result is undefined.

Parameters: image - Pointer to image which will be measured.

Return Value: Width of the block, in pixels.

See Also: **wgetblockheight, wnewblock**

Examples: 11, 19, 29, 57, 64, 66, 67

wgetmode

Function: Returns the current video mode number.

Declaration: **short wgetmode (void)**

Remarks: This is used to store the current video state before changing into graphics mode. You can use **wsetmode** to restore the mode before ending your program.

Parameters: None

Return Value: The mode number of the current video mode.

See Also: **vga256, vgadetected, wsetmode**

Examples: 1-29

wgetpixel

Function: Gets the pixel value at the given screen coordinate.

Declaration: **unsigned char wgetpixel (short x, short y)**

Remarks: This reads the color of a pixel from the current page and returns the color value at the coordinate given.

Parameters:

x	-	Horizontal coordinate of pixel to check.
y	-	Vertical coordinate of pixel to check.

Return Value: Color value of the pixel, between 0 and 255.

See Also: **wfastputpixel, wputpixel**

Examples: 2, 60

wgettextheight

Function: Returns the maximum height of the string.

Declaration: **short wgettextheight (char *string, wgtfont font)**

Remarks: This function is used for justification of strings. If font is NULL, it always returns 8, which is the height of the default font. If font is a pointer to a bitmapped font, the height of the tallest character in the string is returned.

Parameters: string - The string to measure.

font - Pointer to the font used in measurement.

Return Value: The height in pixels of the tallest letter in the string.

See Also: **wgettextwidth**

Examples: 41

wgettextwidth

Function: Returns the width of the string.

Declaration: **short wgettextwidth(char *string, wgtfont font)**

Remarks: This function is used for justification of strings. If font is NULL, the width is 8 for each character. If font is a pointer to a bitmapped font, the width of each character may vary.

Parameters: string - The string to be measured.

font - Pointer to the font used in measurement.

Return Value: The total width in pixels of the string.

See Also: **wgettextheight**

Examples: 41

wgouraudpoly

Function: Draws a Gouraud shaded convex polygon.

Declaration: **void wgouraudpoly (tpolypoint *vertexlist, short numvertex, short x, short y)**

Remarks: The **wgouraudpoly** routine draws a Gouraud shaded polygon by shading the polygon between the colors at each vertex. Each vertex is has the offset (x,y) added to it before the polygon is drawn. vertexlist is an array of vertices, which contains numvertex vertices.

The tpolypoint structure is defined as:

```
typedef struct
{
  short x, y;       /* Coordinate on the screen */
  short sx, sy;     /* Coordinate on the texture */
} tpolypoint;
```

wgouraudpoly uses the sx variable to store the color of the vertex. This means each vertex has a color value (0-255). For Gouraud shading, the polygon uses color numbers between the ones stored at the vertices. For example, if one vertex is color 5, and another is color 30, the polygon will use colors 5-30. Therefore in order to make a shaded polygon, those color values must be smoothly blended from one to the other. You can use the blend function in the WGT Sprite Editor to create a blended range of colors.

Parameters:

vertexlist	- Array of vertices defining the polygon.
numvertex	- Total number of vertices in the array.
x	- Horizontal offset added to each vertex.
y	- Vertical offset added to each vertex.

Return Value: None

See Also: **wdeinitpoly, whollowpoly, winitpoly, wsolidpoly, wtexturedpoly**

Examples: 42, 54, 55

wgtprintf

Function: Prints text using the standard printf format.

Declaration: **void wgtprintf (short x, short y, wgtfont font, char *fmt, ...)**

Remarks: Allows the user to print to the screen using the same format as the printf command from ANSI C. The text is printed at the screen coordinates (x,y) and uses the font. The format is the same as the printf command, so you can easily print numbers within strings very easily.

Example:
wgtprintf (30, 40, bigfont, "Hello %s! You are %hi years old.", name, years);

If font is NULL, it uses the default 8x8 font.

Note: Special characters such as newline, bell, or carriage return are NOT interpreted. **wgtprintf** will display the character used by these codes.

Parameters:

x	- Horizontal location for string display.
y	- Vertical location for string display.
font	- Pointer to font which will be used.
fmt	- String for formatted output (see printf).
...	- Variables used for formatted string output.

Return Value: None

See Also: **wouttextxy**

Examples: 12, 20, 26, 34, 35, 41, 49, 51, 58, 59, 63, 67

whline

Function: Draws a horizontal line between two points.

Declaration: **void whline (short x1, short x2, short y)**

Remarks: This routine will draw horizontal lines faster than the normal **wline** routine. Be careful when passing parameters because the two x coordinates are passed before the single y coordinate (different format than **wline**).

Parameters:

	x1	- Horizontal coordinate of first endpoint.
	x2	- Horizontal coordinate of second endpoint.
	y	- Vertical coordinate for the whole line.

Return Value: None

See Also: **wfline, wline, wstyleline**

Examples: 46

whollowpoly

Function: Draws a hollow polygon.

Declaration: **void whollowpoly (tpolypoint *vertexlist, short numvertex, short x, short y, short closemode)**

Remarks: The **whollowpoly** routine draws a hollow polygon using the current color. Each vertex is has the offset (x,y) added to it before the polygon is drawn. vertexlist is an array of vertices, which contains numvertex vertices. closemode tells whether the polygon is open or closed. If closemode is 1, the first and last points are connected.

The tpolypoint structure is defined as:

```
typedef struct
{
  short x, y;       /* Coordinate on the screen */
  short sx, sy;     /* Coordinate on the texture */
} tpolypoint;
```

whollowpoly does not use the sx,sy variables. While this is wasteful of memory, it allows you to switch between polygon rendering methods without altering your vertex data variables.

Parameters:

vertexlist	- Array of vertices defining the polygon.
numvertex	- Total number of vertices in the array.
x	- Horizontal offset added to each vertex.
y	- Vertical offset added to each vertex.
closemode	- Indicates if the polygon is open or closed.

Return Value: None

See Also: **wgouraudpoly, wsolidpoly, wtexturedpoly**

Examples: 33, 42, 54, 55

winitpoly

Function: Initializes the WGT polygon system to a maximum number of scanlines.

Declaration: **void winitpoly (short maxrows)**

Remarks: Before performing any polygon routines this function must be called to initialize the internal buffers. Simply pass the maximum number of scanlines which will be used when manipulating polygons. Usually this number refers the the screen dimension for height (WGT_SYS.yres).

A call to **wdeinitpoly** must be made before the program exits in order to free the buffers used by the polygon system. This should also be called if you want to re-initialize the system to a different number of scanlines at some point.

Parameters: maxrows - The maximum number of scanlines which will be used for any polygon routines.

Return Value: None

See Also: **wdeinitpoly**

Examples: 42, 54, 51, 55

winittimer

Function: Installs the custom timer interrupt.

Declaration: **void winittimer (void)**

Remarks: This command will replace the current timer interrupt and prepare it for a new interrupt handler. Use **wstarttimer** to replace the interrupt routine with you own.

Parameters: None

Return Value: None

See Also: **wdonetimer, wsettimerspeed, wstarttimer, wstoptimer**

Examples: 3, 11, 46, 55, 56, 57, 58, 59, 61, 62, 63, 68

winit_triangle_renderer

Function: Initializes the WGT triangle rendering system to a maximum number of scanlines.

Declaration: **void winit_triangle_renderer (short maxrows)**

Remarks: Before performing any triangle routines this function must be called to initialize the internal buffers. Simply pass the maximum number of scanlines which will be used when manipulating triangles. Usually this number refers the the screen dimension for height (WGT_SYS.yres).

A call to **wdeinit_triangle_renderer** must be made before the program exits in order to free the buffers used by the polygon system. This should also be called if you want to re-initialize the system to a different number of scanlines at some point.

Parameters: maxrows - The maximum number of scanlines which will be used for any triangle routines.

Return Value: None

See Also: **wdeinit_triangle_renderer, wtriangle_flat_shaded_texture, wtriangle_gouraud, wtriangle_gouraud_shaded_texture, wtriangle_solid, wtriangle_texture, wtriangle_translucent_gouraud, wtriangle_translucent_texture**

Examples: 3D_CAM

wline

Function: Draws a line between two points.

Declaration: **void wline (short x1, short y1, short x2, short y2)**

Remarks: (x1,y1) defines the first endpoint of the line. (x2,y2) defines the second endpoint of the line. The line is clipped if needed.

Parameters:

x1	- Horizontal coordinate of first endpoint.
y1	- Vertical coordinate of first endpoint.
x2	- Horizontal coordinate of second endpoint.
y2	- Vertical coordinate of second endpoint.

Return Value: None

See Also: **wfline, wstyleline**

Examples: 1, 4, 5, 6, 9, 10, 12, 14, 15, 17, 18, 19, 23, 28, 29, 36, 47, 60, 67

wloadblock

Function: Loads a block file from disk into memory.

Declaration: **block wloadblock (char *filename)**

Remarks: **wloadblock** automatically allocates the appropriate amount of memory, by calling malloc, and loads the data into the space allocated. It then returns a pointer to this memory so you may perform other actions on the block. You do should NOT call **wnewblock** before you load the block. **wnewblock** is used for reading a block off the current video page only.

Block files are the simplest of graphics file formats. A block has the following format:

width : short integer
height : short integer
data : width*height bytes (no compression)

No palette data is stored in a block. You must load palettes separately with **wloadpalette** if you use block files.

Parameters: filename - Full pathname of image to be loaded.

Return Value: A pointer to the block loaded. If the block could not be found on disk, it returns NULL.

See Also: **wsaveblock**

Examples: 35, 47

wloadbmp

Function: Loads a BMP image and converts it into block format.

Declaration: **block wloadbmp (char *filename, color *pal)**

Remarks: Given a filename, this function loads the BMP image and converts it to block format (image may be mono, 2/4/8 or 24-bit color). 24-bit color images are converted to greyscale.

Parameters: filename - Full pathname of image to be loaded.

pal - Color array for image palette.

Return Value: Pointer to the loaded BMP, in block format. If the BMP could not be found on disk, it returns NULL.

See Also: **wsavebmp**

Examples: 31, 35

wloadcel

Function: Loads a CEL file (AutoDesk Animator format) from disk into memory.

Declaration: **block wloadcel (char *filename, color *palette)**

Remarks: CEL files are created with AutoDesk Animator. The newer version, Animator Pro, is not compatible with WGT's CEL format. You should use the POCO program oldcel included with Animator Pro to save the old format supported by WGT.

wloadcel automatically allocates the appropriate amount of memory by calling malloc, and loads the data into the space allocated. It then returns a pointer to this memory so you may perform other actions on the block. You should NOT call **wnewblock** before you load the block.

Parameters: filename - Full pathname of image to be loaded.

palette - Color array for image palette.

Return Value: Pointer to the loaded cel, in block format. If the CEL could not be found on disk, it returns NULL.

See Also: **wsavecel**

Examples: 35, 42

wloadfont

Function: Loads a bitmapped font from disk and returns a pointer to it.

Declaration: **wgtfont wloadfont (char *filename)**

Remarks: Fonts can be used with **wouttextxy**, **wgtprintf**, or **wstring**. Custom fonts are created with the WGT Sprite Editor.

Example:
myfont = wloadfont ("tiny.wfn");

Bitmapped fonts require memory to be allocated when loading them, so you should use **wfreefont** to release this memory when you are finished using the font.

Parameters: filename - Full pathname of font to load.

Return Value: Pointer to font in memory.

See Also: **wfreefont, wouttextxy, wstring**

Examples: 26, 40, 41, 47

wloadiff

Function: Loads an Amiga IFF/LBM format image and its associated palette.

Declaration: **block wloadiff (char *filename, color *pal)**

Remarks: Given a filename and a palette buffer, this function will load an IFF or LBM format image and return the pointer to it. The image may be of any dimensions or color depth.

If the call is successful, the pointer will indicate where the image is stored (uncompressed) in memory. An unsuccessful call will return NULL as the pointer.

Parameters:

filename	- The full pathname (or filename) of the image to be loaded
pal	- The image palette will be stored in this buffer

See Also: none

Examples: 64

wloadpak

Function: Loads a compressed PAK file from disk into memory.

Declaration: **block wloadpak (char *filename)**

Remarks: wloadpak automatically allocates the appropriate amount of memory by calling malloc, and loads the data into the space allocated. It then returns a pointer to this memory so you may perform other actions on the block. You do should NOT call **wnewblock** before you load the block.

PAK files are a custom format, only used within WGT applications. It uses a run-length encoding compression scheme, and compresses a bit better than the PCX format. No palette information is saved.

Parameters: filename - Full pathname for image to be loaded.

Return Value: Pointer to the loaded PAK file, in block format. If the PAK could not be found on disk, it returns NULL.

See Also: **wsavepak**

Examples: 35, 61, 64

wloadpalette

Function: Loads a palette file from disk into a palette variable.

Declaration: **void wloadpalette (char *filename, color *pal)**

Remarks: Palette files created by WGT or popular graphics programs may be loaded into the variable specified by palette. Changes are not made to the currently displayed palette. A good example of compatibility would be the AutoDesk Animator. It's .COL files are of this type. Many commercial games use these files as well. Each file must be exactly 768 bytes, or it is an incorrect format.

The format of a palette file is:

```
unsigned char red1
unsigned char green1
unsigned char blue1
unsigned char red2
unsigned char green2
unsigned char blue2
...
unsigned char red256
unsigned char green256
unsigned char blue256
```

These values are repeated for each of the 256 palette registers, giving you a 768 byte file.

Parameters: filename - Full pathname for palette file to be loaded.

pal - Array in which to load the palette.

Return Value: None

See Also: **wsavepalette, wsetpalette, wsetrgb**

Examples: 35, 39, 54, 55

wloadpcx

Function: Loads a PCX file from disk into memory.

Declaration: **block wloadpcx (char *filename, color *pal)**

Remarks: **wloadpcx** automatically allocates the appropriate amount of memory by calling malloc, and loads the data into the space allocated. It then returns a pointer to this memory so you may perform other actions on the block. You do should NOT call **wnewblock** before you load the block. PCX files are created from many drawing programs. They use a run-length encoding compression scheme and unlike the PAK format, it includes palette information. Images may be any dimensions and any color depth up to 8-bit.

Parameters: filename - Full pathname for PCX image to be loaded.

pal - Array in which the PCX's palette will be stored.

Return Value: Pointer to the loaded PCX, in block format. If the PCX could not be found on disk, it returns NULL.

See Also: **wsavepcx**

Examples: 11, 21, 24, 31, 32, 34, 35, 48, 49, 53, 64

wloadsprites

Function: Loads in a sprite file.

Declaration: **short wloadsprites (color *pal, char *filename, block *image_array, short start, short end)**

Remarks: **wloadsprites** will load a sprite file created with the WGT Sprite Editor. pal is the variable to store the palette of the sprites into. The command only loads sprites with numbers start to end. image_array is an array of blocks to load the sprites into. Your array must be large enough to hold the number of sprites between start and end.

Example usage:

```
block mysprites[800];
...
ok = wloadsprites (palette, "game.spr", mysprites, 0, 500);
```

Furthermore, you can "chain" sprite files together by giving an offset of the block array. For example to load another file at the end of the previous file,

```
ok = wloadsprites (palette, "stuff.spr", &mysprites[501], 0, 200);
```

Parameters:

pal	- Array of colors used in sprites.
filename	- Full pathname of sprite file to load from.
image_array	- Array of pointers to the sprites loaded.
start	- Start index of sprites to load.
end	- End index of sprites to load.

Return Value: -1 = unsuccessful
0 = successful

See Also: **wfreesprites, wsavesprites**

Examples: 19, 20, 22, 23, 25, 29, 31, 45, 56, 57, 58, 59, 60, 61, 62, 63, 65, 66, 67, 68

wnewblock

Function: Captures a section of the screen into a block.

Declaration: **block wnewblock (short x1, short y1, short x2, short y2)**

Remarks: Dynamic memory is used to automatically calculate the size of the data, and to store it in the standard block format. See **wloadblock** for a description of this format.

The size of the block is calculated, and then malloc is allocated to reserve enough memory for the block. The section of the current page is copied into this memory, and a pointer is returned to the new block. (x1,y1) defines the upper left corner of the region, and (x2,y2) defines the lower right corner.

example:

```
block mypicture;
void main (void)
{
  ...
  mypicture = wnewblock (50, 50, 60, 60);
  ...
}
```

Parameters:

x1 - Horizontal coordinate of left edge.

y1 - Vertical coordinate of upper edge.

x2 - Horizontal coordinate of right edge.

y2 - Vertical coordinate of lower edge.

Return Value: A pointer the to block. If there is not enough memory to create the block, it returns NULL.

See Also: **wallocblock, wflipblock, wfreeblock, wgetblockheight, wgetblockwidth, wloadblock, wputblock, wsaveblock**

Examples: 9, 10, 15, 16, 17, 18, 19, 20, 24, 27, 33, 42, 43, 47, 51, 54, 55, 67

wnormscreen

Function: Sets the active page back to the original visual page.

Declaration: **void wnormscreen (void)**

Remarks: This is used to return all drawing operations to the normal (visual) screen.

Parameters: None

Return Value: None

See Also: **wfreeblock, wnewblock, wsetscreen**

Examples: 11, 15, 16, 17, 18, 20, 24, 33, 34, 56, 60, 61, 62, 67

wouttextxy

Function: Outputs a string of text.

Declaration: **void wouttextxy (short x, short y, wgtfont font, char *string)**

Remarks: The coordinate system may be be character or pixel based, depending on the state of the text grid system. See the **wtextgrid** procedure for more information. It uses the font for displaying bitmapped fonts which were loaded with **wloadfont**.

In place of the wgtfont, use NULL to display text with the default 8x8 font.

Parameters:

x	- Horizontal coordinate for string display.
y	- Vertical coordinate for string display.
font	- Pointer to font which will be used.
string	- String to display.

See Also: **wgtprintf, wloadfont, wstring, wtextgrid**

Return Value: None

Examples: 4, 7, 13, 14, 15, 20, 24, 34, 48

wpan

Function: Changes the offset of the visual screen.

Declaration: **void wpan (unsigned short pan_offset)**

Remarks: This can be used for special effects to make the screen shake. Each row contains a certain number of pixels across. Therefore the screen will shift up one if the offset is equal to the width of the logical screen. Offsets with multiples of this width will shift the screen up or down, and other offsets will move left or right. For a shaking effect, use small offsets (0-5).

Parameters: pan_offset - Offset in pixel units used to shift screen display.

Return Value: None

Examples: 28

wputblock

Function: Displays a block onto the current graphics page.

Declaration: **void wputblock (short x, short y, block image, short method)**

Remarks: This routine is used in a variety of situations for displaying bitmapped imaged onto the screen. It can be used to show single sprites, or full screen pictures which have been loaded with the set of image commands. X and y are the coordinates on the screen to show the bitmap image, and method can be either:

0 = Normal
1 = XRay

The XRay mode uses the 0 color as a "see-through" color. Any occurrence of this color is not drawn on the screen. This is useful for drawing sprites which overlay a background screen.

Parameters:

x	- Horizontal coordinate for image display.
y	- Vertical coordinate for image display.
image	- Pointer to image which will be displayed.
method	- Technique used to display image.

Return Value: None

See Also: **wloadblock, wloadbmp, wloadcel, wloadpak, wloadpcx, wnewblock**

Examples: 9, 10, 11, 15, 18, 19, 20, 21, 29, 31, 33, 34, 35, 45, 47, 48, 49, 56, 57, 58, 59, 60, 61, 62, 64, 65, 67

wputblock_shade

Function: Displays a block onto the current graphics page, using a shade table.

Declaration: **void wputblock_shade (short x, short y, block image, unsigned char *shadetable, short mode)**

Remarks: This routine is used for displaying bitmapped imaged onto the screen with a special lighting effect. X and y are the coordinates on the screen to show the bitmap image.

For the first three shade methods, shadetable is an array of 256 characters which defines how each color is processed. In general, each color turns into a different color depending on the value in this array. For instance if a pixel from image was color 34, the color written to the screen would be shadetable[34].

For the SHADE_TRANSLUCENT method, shadetable must be an array of 65536 characters. This technique takes a pixel from the image AND the destination screen and combines them into a number between 0 and 65535. The new color is taken from this location in the table.

For the SHADE_MONO method, shadetable is a pointer to a single character. The value of this character will be used as a color to draw the shape. This is useful for highlighting an irregular object.

mode tells which special lighting method to use on the image. It can be one of the following:

SHADE_NORMAL (0)	Each pixel in the image is mapped to the new color in the shade table, including color 0.
SHADE_XRAY (1)	Each pixel in the image is mapped to the new color in the shade table, but color 0 is not drawn. Note that if a color is set to 0 after passing through shadetable, it is drawn.
SHADE_SHADOW (2)	For each pixel in the image which is not color 0, a pixel is taken from the destination screen and passed through shadetable. This lets you alter an area on a previously drawn screen in the

		shape of the image. The actual colors contained in image are not used. This method is only concerned about the pixel if it is greater than color 0.
	SHADE_TRANSLUCENT(3)	For each pixel in the image which is not color 0, a pixel is taken from both the destination screen and the source image and passed through shadetable. This is usually for creating translucent images where you can see part of the background and foreground at the same time.
	SHADE_MONO(4)	For each pixel in the image which is not color 0, a pixel is drawn with the color in shadetable.

Parameters:

x	- Horizontal coordinate for image display.
y	- Vertical coordinate for image display.
image	- Pointer to image which will be displayed.
shadetable	- Pointer to the shade table
method	- Technique used to display image.

Return Value: None

See Also: **wputblock**, **wresize_shade**

Examples: 68

wputpixel

Function: Plots a pixel at the given screen coordinate.

Declaration: **void wputpixel (short x, short y)**

Remarks: Draws a point in the current drawing color (set by **wsetcolor**) at the coordinate (x,y) of the current graphics page.

Parameters:

x	-	Horizontal coordinate of pixel to set.
y	-	Vertical coordinate of pixel to set.

Return Value: None

See Also: **wfastputpixel, wgetpixel**

Examples: 2, 8, 67

wreadpalette

Function: Reads the current palette into a palette variable.

Declaration: **void wreadpalette (unsigned char start, unsigned char finish, color *pal)**

Remarks: Reads the palette registers between start and finish and stores them into the palette variable. This is used to capture the palette being used at the time.

Parameters:

start	- Start index of palette array to set.
finish	- End index of palette array to set.
pal	- Array of colors to be altered.

Return Value: None

See Also: **wloadpalette, wsavepalette, wsetcolor, wsetpalette, wsetrgb**

Examples: 4, 27, 40, 47, 48, 49

wrectangle

Function: Draws a rectangle using the current color and graphics page.

Declaration: **void wrectangle (short x1, short y1, short x2, short y2)**

Remarks: (x1,y1) defines the upper left corner of the rectangle, and (x2,y2) defines the lower right.

Parameters:

x1	- Horizontal coordinate of left edge.
y1	- Vertical coordinate of upper edge.
x2	- Horizontal coordinate of right edge.
y2	- Vertical coordinate of lower edge.

Return Value: None

See Also: **wbar**

Examples: 4, 60

wregionfill

Function: Fills an area of the screen bounded by any color.

Declaration: **void wregionfill (short x, short y)**

Remarks: Fills an enclosed area starting at (x,y), which is bounded by pixels of any color other than the one at the original point. Full clipping is performed.

Parameters: x - Horizontal coordinate of starting pixel.

y - Vertical coordinate of starting pixel.

Return Value: None

Examples: 8

wremap

Function: Remaps the colors of a block or the visual screen to a new palette.

Declaration: **void wremap (color *pal1, block image, color *pal2)**

Remarks: pal1 is the palette belonging to the block image. pal2 contains the new palette the block will use after remapping. This changes every pixel in the block from the normal palette, to the closest color in the new palette pal2. If image is NULL, the visual screen is used.

Parameters:

pal1	- Array of colors normally used for image.
image	- Image to be remapped.
pal2	- Palette to remap block with.

Return Value: None

See Also: **wloadblock, wloadpalette**

Examples: 48

wresize

Function: Draws a previously stored image on the screen to fit within a given boundary.

Declaration: **void wresize (short x1, short y1, short x2, short y2, block image, short mode)**

Remarks: This routine will stretch a bitmapped image to a different size than it was originally stored in. The bitmap is displayed on the current graphics page. If the new size is larger than the original, the image becomes blockier. If the new size is smaller, some of the pixels within the image are removed. Resizing is generally used in special effects sequences, not within a game animation loop. It is too slow to draw several resized bitmaps on the screen at once, so you should resize each image beforehand and store each one as a block. mode is similar to the mode in **wputblock**.

mode = 0 : Normal
mode = 1 : X-ray (color 0 is not copied)

Parameters:

x1	- Horizontal coordinate of left edge.
y1	- Vertical coordinate of upper edge.
x2	- Horizontal coordinate of right edge.
y2	- Vertical coordinate of lower edge.
image	- Pointer to image which will be resized.
mode	- Technique used when resizing (xray or normal)

Return Value: None

See Also: **wresize_column, wresize_shade, wvertres, wwarp**

Examples: 11, 19

wresize_column

Function: Resizes a single column of a bitmap to a new height on the screen.

Declaration: **void wresize_column (short x, short y, short y2, block image, short column, short mode)**

Remarks: This routine will stretch a single column from a bitmapped image to a different height than it was originally stored in. The column is displayed on the current graphics page. If the new size is larger than the original, the image becomes blockier. If the new size is smaller, some of the pixels within the image are removed. mode is similar to the mode in **wputblock**.

mode = 0 : Normal
mode = 1 : X-ray (color 0 is not copied)

Parameters:

x	- Horizontal coordinate of column on screen.
y	- Top y coordinate of column on screen.
y2	- Botton y coordinate of column on screen.
image	- Pointer to image which will be resized.
column	- Horizontal coordinate of column on image. This must be less than the width of image.
mode	- Technique used when resizing (xray or normal)

Return Value: None

See Also: **wresize, wresize_shade, wvertres**

Examples: None

wresize_shade

Function: Draws a previously stored image on the screen to fit within a given boundary, using a shade table.

Declaration: **void wresize (short x1, short y1, short x2, short y2, block image, unsigned char *shadetable, short mode)**

Remarks: This routine will stretch a bitmapped image to a different size than it was originally stored in, while performing a special light effect on the image.

For the first three shade methods, shadetable is an array of 256 characters which defines how each color is processed. In general, each color turns into a different color depending on the value in this array. For instance if a pixel from image was color 34, the color written to the screen would be shadetable[34].

For the SHADE_TRANSLUCENT method, shadetable must be an array of 65536 characters. This technique takes a pixel from the image AND the destination screen and combines them into a number between 0 and 65535. The new color is taken from this location in the table.

For the SHADE_MONO method, shadetable is a pointer to a single character. The value of this character will be used as a color to draw the shape. This is useful for highlighting an irregular object.

mode tells which special lighting method to use on the resized image. It can be one of the following:

SHADE_NORMAL (0)	Each pixel in the image is mapped to the new color in the shade table, including color 0.
SHADE_XRAY (1)	Each pixel in the image is mapped to the new color in the shade table, but color 0 is not drawn. Note that if a color is set to 0 after passing through shadetable, it is drawn.
SHADE_SHADOW (2)	For each pixel in the image which is not color 0, a pixel is taken from the destination screen and passed through shadetable. This lets you alter an area on a previously drawn screen in the

shape of the image. The actual colors contained in image are not used. This method is only concerned about the pixel if it is greater than color 0.

SHADE_TRANSLUCENT(3)	For each pixel in the image which is not color 0, a pixel is taken from both the destination screen and the source image and passed through shadetable. This is usually for creating translucent images where you can see part of the background and foreground at the same time.
SHADE_MONO(4)	For each pixel in the image which is not color 0, a pixel is drawn with the color in shadetable.

Parameters:

x1	- Horizontal coordinate of left edge.
y1	- Vertical coordinate of upper edge.
x2	- Horizontal coordinate of right edge.
y2	- Vertical coordinate of lower edge.
image	- Pointer to image which will be resized.
shadetable	- Pointer to the shade table
mode	- Technique used when resizing.

Return Value: None

See Also: **wresize, wresize_column, wvertres**

Examples: 68

wretrace

Function: Waits for the vertical retrace.

Declaration: **void wretrace (void)**

Remarks: Use the **wretrace** command when you get flickering graphics. This command waits for the vertical retrace on your monitor, so you can write to the screen while it is not refreshing the screen.

It will not work perfectly unless you are writing small amounts to the screen, but it is worth a try if you get flickering screens. This routine can also be used for timing (obtaining constant speeds on different CPUs).

Parameters: None

Return Value: None

Examples: 3, 5, 20, 21, 22, 23, 24, 25, 27, 29, 37, 39, 51, 54, 56, 57, 59, 60, 61, 65

wsaveblock

Function: Saves a block to a disk file in raw format.

Declaration: **short wsaveblock (char *filename, block image)**

Remarks: A file is created which holds the data stored at the pointer given by image. This allows your programs to store images that can be loaded at a later time.

Parameters: filename - Full pathname for image to save.

image - Pointer to the image which will be saved.

Return Value: 0 if successful, 1 if it could not save the file

See Also: **wfreeblock, wloadblock, wnewblock**

Examples: 35

wsavebmp

Function: Saves a block image to disk in BMP format.

Declaration: **short wsavebmp (char *filename, block image, color *pal)**

Remarks: Given a filename and a pointer to a block, this function saves the image in BMP format (8-bit color).

Parameters:

filename	- Full pathname of image to save.
image	- Pointer to actual image.
pal	- Palette to save with image.

Return Value: None

See Also: **wloadbmp**

Examples: 35

wsavecel

Function: Saves a CEL file (AutoDesk Animator format) to disk.

Declaration: **short wsavecel (char *filename, block image, color *pal)**

Remarks: A file is created which holds the data stored at the pointer given by image. This allows your programs to store images that can be loaded at a later time.

Parameters:

filename	- Full pathname of image to save.
image	- Pointer to actual image in memory.
pal	- Palette to save with image.

Return Value: 0 if succesful, 1 if it could not save the file

See Also: **wloadcel**

Examples: 35

wsavepak

Function: Saves a block to disk in compressed format.

Declaration: **short wsavepak (char *filename, block image)**

Remarks: This is similar to **wsaveblock** except that the image is in a file which is compressed. The compression used is "similar" to PCX format (sometimes better). The more simplistic the image, the smaller the file. The image still requires the same amount of memory when loading it in, but takes up less disk space.

Parameters: filename - Full pathname of image to save.

image - Pointer to image.

Return Value: 0 if succesful, 1 if it could not save the file

See Also: **wloadblock, wloadpak**

Examples: 35

wsavepalette

Function: Save a palette variable's contents to a file.

Declaration: **void wsavepalette (char *filename, color *pal)**

Remarks: Creates a 768 byte palette file to disk from a variable specified by palette. This is useful for storing different palettes for games and their various screens.

Parameters: filename - Full pathname of palette file to save.

pal - Actual palette to save.

Return Value: None

See Also: **wloadpalette, wsetpalette, wsetrgb**

Examples: 35

wsavepcx

Function: Saves a block to disk in the PCX compressed format.

Declaration: **short wsavepcx (char *filename, block image, color *pal)**

Remarks: This is similar to **wsaveblock** except that the image is in a file which is compressed. The compression used is the PCX format. The more simplistic the image, the smaller the file. The image still requires the same amount of memory when loading it in, but takes up less disk space.

Parameters:

	filename	- Full pathname of PCX file to save.
	image	- Pointer to image in memory.
	pal	- Palette to save with image.

Return Value: None

See Also: **wloadblock, wloadpak, wloadpcx**

Examples: 35

wsavesprites

Function: Savess a sprite file using a palette array and a set of blocks.

Declaration: **short wsavesprites (color *pal, char *filename, block *image_array, short start, short end)**

Remarks: **wsavesprites** will save a sprite file compatible with the WGT Sprite Editor. pal is the array of color info used for the images. image_array is the array of blocks containing the sprites. This function will save from index "start" to whatever value you specify as the "end" sprite in the array. Your array must be large enough to hold the given number of sprites between start and end or a page fault will occur.

Parameters:

pal	- Array of colors used in sprites.
filename	- Full pathname of sprite file to save.
image_array	- Array of pointers to the sprites in memory.
start	- Lowest index of sprites to save.
end	- Highest index of sprites to save.

Return Value: -1 = unsuccessful
0 = successful

See Also: **wloadsprites**

Examples: 45

wscan_convertpoly

Function: Scan converts a concave polygon into a rendered polygon buffer.

Declaration: **void wscan_convertpoly (tpolypoint *vertexlist, short numvertex, wscanpoly *scanpoly)**

Remarks: Concave polygons allow multiple horizontal line segments on each row of the polygon. This means you can have any number of vertices in any position and the polygon will still be drawn correctly.

Rendered polygons are only stored in an allocated buffer. They are not drawn until **wdraw_scanpoly** is called. The benefit of storing the polygon in a buffer is you can draw the polygon in a different location on the screen without doing the mathematical calculations again.

wscanpoly is a structure to hold the polygon buffer. It is defined as:

```
typedef struct
{
  char **pointbuffer;       /* Holds a buffer of x coordinates for each row
                               of the polygon. */
  unsigned short *numpoints;  /* Holds the size of the point buffer in
bytes. */
} wscanpoly;
```

Parameters:

vertexlist - Array of vertices that define the concave polygon.

numvertex - Total number of vertices in the array.

scanpoly - Array of vertices defining the rendered polygon.

Return Value: None

See Also: **winitpoly, wdeinitpoly, wdraw_scanpoly, wfree_scanpoly**

Examples: 51

wset_capslock

Function: Sets the state of the capslock key.

Declaration: **void wset_capslock (short state)**

Remarks: This routine will set the state of the capslock key and change the keyboard LEDS appropriately.

Parameters: state - Indicates on/off status of key.
0=off 1=on

Return Value: None

See Also: **wget_capslock, wset_numlock, wset_scrlock**

Examples: 44

wset_imagebyte

Function: Stores a byte of information at the end of allocated memory for an image.

Declaration: **void wset_imagebyte (block image, unsigned char status)**

Remarks: This routine stores a value as the last allocated byte of memory for that image and it is recommended the the information is used in the following format:

Bit 0 - Indicates Xray or Normal block
Bits 1-7 are unused.

If bit 0 is set (status is 1), the image contains some pixels of color index 0. This means that an XRAY putblock would be effective to make the image see-through for parallax scrolling and other such effects. If the status is 0, a normal putblock would be the fastest way of blasting it to the screen.

Parameters: image - Pointer to the image in memory.

status - A single byte of information to be stored.

Return Value: None

See Also: **wget_imagebyte, wget_imagebit, wupdate_imagebytes**

Examples: None

wset_numlock

Function: Sets the state of the numlock key.

Declaration: **void wset_numlock (short state)**

Remarks: This will set the state of the numlock key and change the keyboard LEDS appropriately.

Parameters: state - Indicates on/off status of key.
0=off 1=on

Return Value: None

See Also: **wget_numlock, wset_capslock, wset_scrlock**

Examples: 44

wset_scrlock

Function: Sets the state of the scrlock key.

Declaration: **void wset_scrlock (short state)**

Remarks: This will set the state of the scroll lock key and change the keyboard LEDS appropriately.

Parameters: state - Indicates on/off status of key.
0=off 1=on

Return Value: None

See Also: **wget_scrlock, wset_capslock, wset_numlock**

Examples: 44

wsetcolor

Function: Sets the current drawing color.

Declaration: **void wsetcolor (unsigned char col)**

Remarks: Col may be in the range 0 to 255. All drawing procedures used after setting this will use the color you choose.

Parameters: col - Color index to use for graphics primitives.

Return Value: None

See Also: **wloadpalette, wsavepalette, wsetpalette, wsetrgb**

Examples: 1, 2, 4, 5, 6, 8, 9, 10, 12, 14, 15, 16, 17, 18, 19, 20, 23, 25, 27, 28, 29, 33, 36, 39, 42, 46, 47, 51, 53, 54, 55, 60, 61, 66, 67

wsetcursor

Function: Sets the shape of the software emulated text cursor.

Declaration: **void wsetcursor (short start, short end)**

Remarks: start and end are the y coordinates of the upper and lower lines for a cursor 8 pixels wide. The X coordinates are fixed to 8 pixels wide. The cursor is not the hardware text cursor, but rather a software simulation which flashes during text input. A solid box would be (0,7).

Parameters: start - Upper coordinate of cursor.

end - Lower coordinate of cursor.

See Also: **wflashcursor, wstring**

Return Value: None

Examples: 14

wsetmode

Function: Sets the video mode.

Declaration: **void wsetmode (short mode)**

Remarks: This is mainly used to restore the video mode. The most common mode to restore is 0x03, which is 80x25 Text mode.

Parameters: mode - Video mode to switch into.

Return Value: None

See Also: **vga256, vgadetected, wgetmode**

Examples: 1-29

wsetpalette

Function: Sets a group of colors from a specified palette variable.

Declaration: **void wsetpalette (unsigned char start, unsigned char finish, color *pal)**

Remarks: Colors from start to finish are set using the values stored in the palette variable. The variable type color is specific to WGT and may not be used elsewhere. It is defined as follows:

```
typedef struct {
  unsigned char r, g, b;
} color;
```

To define a palette, you must make an array of colors like this:

```
color palette[256];
```

Parameters:

start	- Start index of palette array to set.
finish	- End index of palette array to set.
pal	- Array of colors to use.

Return Value: None

See Also: **wloadpalette, wreadpalette, wsavepalette, wsetcolor, wsetrgb**

Examples: 4, 5, 6, 7, 11, 13, 19, 20, 21, 22, 23, 24, 25, 27, 31, 32, 34, 35, 39, 40, 42, 45, 47, 48, 49, 51, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 63, 64, 65, 66, 67

wsetrgb

Function: Sets a color's red, green, and blue values.

Declaration: **void wsetrgb (unsigned char color, unsigned char red, unsigned char green, unsigned char blue, color *pal)**

Remarks: Color can range from 0 to 255. The Red, Green, and Blue values are in the range 0 to 63 giving a total of 262144 color combinations possible (64*64*64).

The colors do NOT change until you call the **wsetpalette** command. This way you can set a number of colors and change them all simultaneously.

Parameters:

color	- Index within color array to set.
red	- Red level to use (0-63).
green	- Green level to use (0-63).
blue	- Blue level to use (0-63).
pal	- Array of colors to set.

Return Value: None

See Also: **wloadpalette, wsavepalette, wsetcolor, wsetpalette**

Examples: 4, 5, 6, 7, 13, 20, 40, 42, 47, 51, 55

wsetscreen

Function: Makes all drawing procedures use a specified destination video buffer.

Declaration: **void wsetscreen (block image)**

Remarks: Sets the active drawing buffer to image. Every WGT graphics routine will write to this page instead of the visual screen. This is used to create pictures in memory which will be displayed later using **wputblock** or **wcopyscreen**. Clipping values are adjusted to meet the new virtual screen dimensions.

The screen must be first allocated by using **wnewblock** or a block load function. It can be any size.

If image is NULL, the active buffer is set to the visual screen. (equivalent to **wnormscreen**)

Parameters: image - Pointer to screen which becomes the active screen.

Return Value: None

See Also: **wfreeblock, wnewblock**

Examples: 15, 16, 17, 18, 19, 20, 33, 34, 42, 54, 55, 60, 61, 63, 67

wsettimerspeed

Function: Sets the custom timer interrupt to run at a different speed.

Declaration: **void wsettimerspeed (int speed)**

Remarks: This command is used to increase the speed of the timer. Normally the timer interrupt is called at 18.2 times per second. For games with frame rates up to 60 frames per second, this is not fast enough. Typically you will set the timer to the fastest frame rate your game will run, or at the speed of the monitor's retrace.

To convert from ticks per second to the timer speed, use the TICKS macro. For example, wsettimerspeed (TICKS(60));

Parameters: speed - The speed of the new timer

Return Value: None

See Also: **wdonetimer, winittimer, wstarttimer, wstoptimer**

Examples: None

wskew

Function: Skews a block sideways.

Declaration: **void wskew (short x, short y, block image, short degrees)**

Remarks: Shows the bitmap image on the current page at (x,y) using a skew value of degrees. Each row of the bitmap is pushed either left or right of the previous. If degrees is negative, it will be skewed in the opposite direction.

Parameters:

x	- Horizontal display coordinate.
y	- Vertical display coordinate.
image	- Pointer to image to be skewed.
degrees	- Number of degrees to skew image.

Return Value: None

Examples: 27

wsline

Function: Stores the points of a line into an array instead of plotting them on the screen.

Declaration: **void wsline (short x1, short y1, short x2, short y2, short *y_array)**

Remarks: Simply stores the y coordinate values in the appropriate place in the array for every x coordinate on a line. y_array is an array of short integers. See **wwarp** for more info.

Parameters:

x1	- Horizontal coordinate of first endpoint.
y1	- Vertical coordinate of first endpoint.
x2	- Horizontal coordinate of second endpoint.
y2	- Vertical coordinate of second endpoint.
y_array	- Buffer used to store points in the line.

Return Value: None

See Also: **wwarp**

Examples: 32

wsolidpoly

Function: Draws a filled convex polygon.

Declaration: **void wsolidpoly (tpolypoint *vertexlist, short numvertex,**
short x, short y,
void (*customline)(short x1, short x2, short y))

Remarks: The **wsolidpoly** routine draws a filled convex polygon using the current color. Each vertex is has the offset (x,y) added to it before the polygon is drawn. vertexlist is an array of vertices, which contains numvertex vertices. customline is the name of the horizontal line routine to call for each line in the polygon. If NULL is given, it will use **whline**. You can create your own line routines to make other unique effects such as fill patterns using the **wstyleline** command, image copying using memcpy, or shadebob techniques.

The tpolypoint structure is defined as:

```
typedef struct
{
  short x,y;  /* Coordinate on the screen */
  short sx,sy; /* Coordinate on the texture */
} tpolypoint;
```

wsolidpoly does not use the sx,sy variables. While this is wasteful of memory, it allows you to switch between polygon rendering methods without altering your vertex data variables.

Customline is a function which accepts three parameters. The first two are the x coordinates of a horizontal line, and the third is the y coordinate.

Parameters:

vertexlist	- Array of vertices defining the polygon.
numvertex	- Total number of vertices in the array.
x	- Horizontal offset to add to each vertex.
y	- Vertical offset to add to each vertex.
customline	- Address of custom horizontal line routine.

Return Value: none

See Also: **wdeinitpoly, whollowpoly, winitpoly**

Examples: 42, 51, 54, 55

wstarttimer

Function: Installs a custom timer interrupt at the given speed.

Declaration: **void wstarttimer (wtimerproc timer, int speed)**

Remarks: This command will install the procedure called timer as the new timer interrupt, and set the speed at which this routine will be called.

wtimerproc is defined as:
typedef void (*wtimerproc)(void);

Parameters: timer - The procedure name of the new timer routine

speed - The new timer speed

Return Value: None

See Also: **wdonetimer, winittimer, wsettimerspeed, wstoptimer**

Examples: 3, 11, 46, 55, 56, 57, 58, 59, 61, 62, 63, 68

wstoptimer

Function: Resets a custom timer interrupt to an empty procedure.

Declaration: **void wstoptimer (void)**

Remarks: This command will remove the custom timer procedure and replace it with a procedure which does nothing. The timer interrupt itself is still present, and must be removed with **wdonetimer**.

Parameters: None

Return Value: None

See Also: **wdonetimer, winittimer, wsettimerspeed, wstarttimer**

Examples: 3, 11, 46, 55, 56, 57, 58, 59, 61, 62, 63, 68

wstring

Function: Reads in a string of text from the keyboard and displays both the text and a simulated cursor on the graphics page.

Declaration: **void wstring (short x, short y, char *string, char *legal, short maxlength)**

Remarks: This procedure uses the flashing cursor while you input the string. Any characters you wish to be able to type in must be included in LEGAL. For example, if you want a yes or no answer only, make LEGAL="YNyn". num is the maximum number of letters in the string. x and y are the coordinates of the initial cursor position.

STRING must be a pointer to char:

```
char *filename;
```

Along with legal:

```
char *legal_chars = " ABCDEFGHIJKLMNOPQRSTUVWXYZ
                    abcdefghijklmnopqrstuvwxyz1234567890_.";
```

Then, you must allocate memory for the string, like this:

```
filename = (char *) malloc (13);
```

Add 1 to the length for the null terminator.

Call the string procedure:

```
wstring (10, 22, filename, legal_chars, 12);
```

After you're done with the string, free the memory by

```
free (filenameout);
```

INSERT,DELETE,HOME,END and the arrow keys are functional, and backspace is destructive.

If you press insert, the cursor shape changes and toggles between insert and overwrite modes.

Parameters:

x	- Horizontal coordinate for text input.
y	- Vertical coordinate for text input.
string	- Character string input buffer.
legal	- String of legal input characters.

maxlength - Maximum number of characters to input.

Return Value: None

See Also: **wflashcursor, wgtprintf, wouttextxy**

Examples: 14

wstyleline

Function: Draws a line from between two points with a line pattern.

Declaration: **void wstyleline (short x, short y, short x2, short y2, unsigned short style)**

Remarks: (x1,y1) defines the first endpoint of the line. (x2,y2) defines the second endpoint of the line. pattern is a number between 0 and 65535 which defines the line pattern. The style can be thought of as an array of 16 bits, each telling if a pixel is on (1) or off (0).

For example, a dashed line would have half of the pixels turned on, and half of them turned off. This would be represented as 1111111100000000 in binary.

Of course you cannot use binary numbers within your C program, so you must first convert them to hexadecimal. To convert this number, split the digits into groups of four. Below is a table of the binary group on the left, and the hexadecimal equivalent on the right.

0000 0	1000 8
0001 1	1001 9
0010 2	1010 A
0011 3	1011 B
0100 4	1100 C
0101 5	1101 D
0110 6	1110 E
0111 7	1111 F

Some examples of this conversion are:

Binary: 1111 1111 0000 0000
Hexadecimal: F F 0 0 = 0xFF00

Binary: 1000 1000 1001 1001
Hexadecimal: 8 8 9 9 = 0x8899

This conversion process is also important when creating user defined mouse cursors with the **mouseshape** routine.

Parameters: x - Horizontal coordinate of first endpoint.

y - Vertical coordinate of first endpoint.

	x2	- Horizontal coordinate of second endpoint.
	y2	- Vertical coordinate of second endpoint.
	style	- Bitpattern of the line.

Return Value: None

See Also: **wfline, wline**

Examples: 4

wtextbackground

Function: Sets the background text color.

Declaration: **void wtextbackground (unsigned char col)**

Remarks: Col is in the range 0 to 255. The background color behind text may be turned off by calling **wtexttransparent**.

Parameters: col - Color index to use for text background.

Return Value: None

See Also: **wouttextxy, wstring, wtextcolor, wtextcursor, wtextgrid, wtexttransparent**

Examples: 7, 14, 20, 26, 53, 58, 59

wtextcolor

Function: Selects the foreground character color.

Declaration: **void wtextcolor (unsigned char col)**

Remarks: Col is in the range 0 to 255. The foreground color of text (the actual letters) may be turned off by calling **wtexttransparent**.

Parameters: col - Color index to use for text output.

Return Value: None

See Also: **wouttextxy, wstring, wtextbackground, wtextcursor, wtextgrid, wtexttransparent**

Examples: 4, 7, 12, 13, 14, 15, 20, 24, 26, 34, 35, 41, 48, 49, 51, 53, 58, 59, 63, 67

wtextgrid

Function: Switches the text coordinate system between character and pixel based.

Declaration: **void wtextgrid (short state)**

Remarks: xt may be output in regular graphics coordinates in pixels, or a text mode system which ligns the strings to character boundaries. If STATE is set to 1, all input and output values for WGT text functions are expected to be character based. Characters will be 'snapped' to the grid as if it were a text mode. 0 will use the regular graphics system coordinates.

Parameters: state - Indicates text gridlock on/off status.

Return Value: None

See Also: **wouttextxy, wstring**

Examples: 7

wtexttransparent

Function: Sets text output to supress the display of the foreground or background.

Declaration: **void wtexttransparent (short mode)**

Remarks: This will turn off the display of either the foreground or the background, or set it to show both.

The values possible are:
TEXTFG (0): Foreground turned on
TEXTBG (1): Background turned on
TEXTFGBG(2): Both turned on

Parameters: mode - Indicates visibility status of text fore/background.

Return Value: None

See Also: **wtextbackground, wtextcolor**

Examples: 7, 12, 13, 14, 20, 26, 34, 48, 49, 51, 53, 58, 59, 63

wtexturedpoly

Function: Draws a texture mapped convex polygon.

Declaration: **void wtexturedpoly (tpolypoint *vertexlist, short numvertex, short x, short y, block image, short mode)**

Remarks: The **wtexturedpoly** routine draws a texture mapped convex polygon using the texture bitmap image. Each vertex is has the offset (x,y) added to it before the polygon is drawn. vertexlist is an array of vertices, which contains numvertex vertices.

Mode can either be NORMAL (0), or XRAY (1) , which allows for see-through textures. The tpolypoint structure is defined as:

```
typedef struct
{
  short x,y;  /* Coordinate on the screen */
  short sx,sy; /* Coordinate on the texture */
} tpolypoint;
```

(sx,sy) contains the offset within the texture bitmap of the vertex. Each vertex in the polygon has a corresponding coordinate on the texture bitmap.

Parameters:

vertexlist	- Array of vertices which define the polygon.
numvertex	- Total number of vertices in array.
x	- Horizontal offset added to each vertex.
y	- Vertical offset added to each vertex.
image	- Pointer to block used as a texture.
mode	- Indicates copy mode (xray or normal).

Return Value: None

See Also: **wgouraudpoly, whollowpoly, wsolidpoly**

Examples: 42

wtriangle_flat_shaded_texture

Function: Draws a flat shaded texture mapped triangle.

Declaration: **void wtriangle_flat_shaded_texture (tpoint *ttri, block texture, short shade, block shadetable)**

Remarks: The **wtriangle_flat_shaded_texture** routine draws a flat shaded texture mapped triangle on the screen using texture as the image, and shadetable for the lighting table. shadetable is an array of unsigned characters. It can be defined as follows:

unsigned char shadetable[NUMSHADES][256];

shade tells which table of 256 characters to use as the lighting table. It can range from 0 to NUMSHADES - 1.

When drawing the triangle, each pixel is taken from the image texture, then a new color is looked up from shadetable[shade][color] and displayed on the screen. This allows you to remap the texture to a different palette as it is drawn.

texture must have a width of 256, and has a maximum height of 256. This is the only limitation of the rendering system. You can place multiple pictures on a single texture block and show them by assigning different texture coordinates.

ttri is an array of 3 vertices. Each vertex has the following structure:

```
typedef struct
 {
  short x, y;          /* Coordinate on the screen */
  long sx, sy;         /* Coordinate on the texture */
  short col;           /* Shade/color  value */
 } tpoint;
```

For this routine, col is not used.

Before using this routine, you must initialize the triangle rendering engine using **winit_triangle_renderer**.

Parameters: ttri - Pointer to 3 tpoints which hold the vertex information.

texture - Image containing the texture (max 256x256)

shade - Tells the base offset into the shadetable

shadetable - Array of unsigned characters to use for remapping

Return Value: None

See Also: **winit_triangle_renderer, wdeinit_triangle_renderer, wtriangle_gouraud, wtriangle_gouraud_shaded_texture, wtriangle_solid, wtriangle_texture, wtriangle_translucent_gouraud, wtriangle_translucent_texture**

Examples: 3D_CAM

wtriangle_gouraud

Function: Draws a Gouraud shaded triangle.

Declaration: **void wtriangle_gouraud (tpoint *gtri)**

Remarks: The **wtriangle_gouraud** routine draws a Gouraud shaded triangle on the screen. gtri is an array of 3 vertices. Each vertex has the following structure:

```
typedef struct
 {
  short x, y;              /* Coordinate on the screen */
  long sx, sy;             /* Coordinate on the texture */
  short col;               /* Shade/color  value */
 } tpoint;
```

For this routine, sx and sy are not used.

The colors within the Gouraud shaded triangle are in the range 0 to 255. The triangle will use all the colors between the three col values of the vertices.

Before using this routine, you must initialize the triangle rendering engine using **winit_triangle_renderer**.

Parameters: gtri - Pointer to 3 tpoints which hold the vertex information.

Return Value: None

See Also: **winit_triangle_renderer, wdeinit_triangle_renderer, wtriangle_flat_shaded_texture, wtriangle_gouraud_shaded_texture, wtriangle_solid, wtriangle_texture, wtriangle_translucent_gouraud, wtriangle_translucent_texture**

Examples: 3D_CAM

wtriangle_gouraud_shaded_texture

Function: Draws a gouraud shaded texture mapped triangle.

Declaration: **void wtriangle_gouraud_shaded_texture (tpoint *ttri, block texture, block shadetable)**

Remarks: The **wtriangle_gouraud_shaded_texture** routine draws a Gouraud shaded texture mapped triangle on the screen using texture as the image, and shadetable for the lighting table. shadetable is an array of unsigned characters. It can be defined as follows:

unsigned char shadetable[NUMSHADES][256];

When drawing the triangle, each pixel is taken from the image texture, then a new color is looked up from shadetable[shade][color] and displayed on the screen. This allows you to remap the texture to a different palette as it is drawn.

texture must have a width of 256, and has a maximum height of 256. This is the only limitation of the rendering system. You can place multiple pictures on a single texture block and show them by assigning different texture coordinates.

ttri is an array of 3 vertices. Each vertex has the following structure:

```
typedef struct
 {
  short x, y;          /* Coordinate on the screen */
  long sx, sy;         /* Coordinate on the texture */
  short col;           /* Shade/color  value */
 } tpoint;
```

shade ranges from 0 to NUMSHADES - 1 and it is assumed the shadetable is set up correctly. The triangle will use all the shades between the three col values of the vertices.

Before using this routine, you must initialize the triangle rendering engine using **winit_triangle_renderer**.

Parameters: ttri - Pointer to 3 tpoints which hold the vertex information.

texture - Image containing the texture (max 256x256)

shadetable - Array of unsigned characters to use for remapping

Return Value: None

See Also: **winit_triangle_renderer, wdeinit_triangle_renderer, wtriangle_flat_shaded_texture, wtriangle_gouraud, wtriangle_solid, wtriangle_texture, wtriangle_translucent_gouraud, wtriangle_translucent_texture**

Examples: 3D_CAM

wtriangle_solid

Function: Draws a filled triangle.

Declaration: **void wtriangle_solid (tpoint *ftri)**

Remarks: The **wtriangle_solid** routine draws a filled triangle on the screen using the current drawing color. ftri is an array of 3 vertices. Each vertex has the following structure:

```
typedef struct
 {
  short x, y;              /* Coordinate on the screen */
  long sx, sy;             /* Coordinate on the texture */
  short col;               /* Shade/color  value */
 } tpoint;
```

For this routine, sx, sy, and col are not used.

Before using this routine, you must initialize the triangle rendering engine using **winit_triangle_renderer**.

Parameters: ftri - Pointer to 3 tpoints which hold the vertex information.

Return Value: None

See Also: **winit_triangle_renderer, wdeinit_triangle_renderer, wtriangle_flat_shaded_texture, wtriangle_gouraud, wtriangle_gouraud_shaded_texture, wtriangle_texture, wtriangle_translucent_gouraud, wtriangle_translucent_texture**

Examples: 3D_CAM

wtriangle_texture

Function: Draws a flat shaded texture mapped triangle.

Declaration: **void wtriangle_texture (tpoint *ttri, block texture)**

Remarks: The **wtriangle_texture** routine draws a texture mapped triangle on the screen using texture as the image.

texture must have a width of 256, and has a maximum height of 256. This is the only limitation of the rendering system. You can place multiple pictures on a single texture block and show them by assigning different texture coordinates.

ttri is an array of 3 vertices. Each vertex has the following structure:

```
typedef struct
 {
  short x, y;              /* Coordinate on the screen */
  long sx, sy;             /* Coordinate on the texture */
  short col;               /* Shade/color  value */
 } tpoint;
```

For this routine, col is not used.

Before using this routine, you must initialize the triangle rendering engine using **winit_triangle_renderer**.

Parameters: ttri - Pointer to 3 tpoints which hold the vertex information.

texture - Image containing the texture (max 256x256)

Return Value: None

See Also: **winit_triangle_renderer, wdeinit_triangle_renderer, wtriangle_flat_shaded_texture, wtriangle_gouraud, wtriangle_gouraud_shaded_texture, wtriangle_solid, wtriangle_translucent_gouraud, wtriangle_translucent_texture**

Examples: 3D_CAM

wtriangle_translucent_gouraud

Function: Draws a translucent gouraud shaded triangle.

Declaration: **void wtriangle_translucent_gouraud(tpoint *gtri, block shadetable)**

Remarks: The **wtriangle_translucent_gouraud** routine draws a translucent Gouraud shaded triangle on the screen using shadetable for the lighting table. shadetable is an array of unsigned characters. It can be defined as follows:

unsigned char shadetable[256][256];

When drawing the triangle, a pixel is taken from the shaded triangle, and a pixel is taken from the current drawing buffer, then a new color is looked up from shadetable[backgroundcolor][trianglecolor] and displayed on the screen. The table is used to create a single color given two color. Usually this is done by taking a percentage of the first color and adding it to a percentage of the second color.

gtri is an array of 3 vertices. Each vertex has the following structure:

```
typedef struct
 {
  short x, y;          /* Coordinate on the screen */
  long sx, sy;         /* Coordinate on the texture */
  short col;           /* Shade/color  value */
 } tpoint;
```

For this routine, sx and sy are not used.

The colors within the Gouraud shaded triangle are in the range 0 to 255. The triangle will use all the colors between the three col values of the vertices.

Before using this routine, you must initialize the triangle rendering engine using **winit_triangle_renderer**.

Parameters: gtri - Pointer to 3 tpoints which hold the vertex information.

shadetable - Array of unsigned characters to use for remapping

Return Value: None

See Also: **winit_triangle_renderer, wdeinit_triangle_renderer, wtriangle_flat_shaded_texture, wtriangle_gouraud, wtriangle_gouraud_shaded_texture, wtriangle_solid, wtriangle_texture, wtriangle_translucent_texture**

Examples: 3D_CAM

wtriangle_translucent_texture

Function: Draws a translucent texture mapped triangle.

Declaration: **void wtriangle_translucent_texture (tpoint *ttri, block texture, block shadetable)**

Remarks: The **wtriangle_translucent_texture** routine draws a translucent texture mapped triangle on the screen using texture as the image, and shadetable for the lighting table. shadetable is an array of unsigned characters. It can be defined as follows:

unsigned char shadetable[256][256];

When drawing the triangle, a pixel is taken from the image texture, and a pixel is taken from the current drawing buffer, then a new color is looked up from shadetable[backgroundcolor][texturecolor] and displayed on the screen. This allows you to remap the texture to a different palette as it is drawn. The table is used to create a single color given two color. Usually this is done by taking a percentage of the first color and adding it to a percentage of the second color. Besides translucent textures, you can also create tinted textures by adding to the red, green, or blue values of the color.

texture must have a width of 256, and has a maximum height of 256. This is the only limitation of the rendering system. You can place multiple pictures on a single texture block and show them by assigning different texture coordinates.

ttri is an array of 3 vertices. Each vertex has the following structure:

```
typedef struct
 {
  short x, y;              /* Coordinate on the screen */
  long sx, sy;             /* Coordinate on the texture */
  short col;               /* Shade/color  value */
 } tpoint;
```

For this routine, col is not used.

Before using this routine, you must initialize the triangle rendering engine using **winit_triangle_renderer**.

Parameters: ttri - Pointer to 3 tpoints which hold the vertex information.

texture - Image containing the texture (max 256x256)

shadetable - Array of unsigned characters to use for remapping

Return Value: None

See Also: **winit_triangle_renderer, wdeinit_triangle_renderer, wtriangle_flat_shaded_texture, wtriangle_gouraud, wtriangle_gouraud_shaded_texture, wtriangle_solid, wtriangle_texture, wtriangle_translucent_gouraud**

Examples: 3D_CAM

wupdate_imagebytes

Function: Updates status bytes for an array of images.

Declaration: **void wupdate_imagebytes (block *image_array, short start, short end)**

Remarks: This routine updates a value from the last allocated byte of memory for each image in the array of images (from start index to end index). The value indicates the following information:

If the status is 1, the image contains some pixels of color index 0. This means that an XRAY putblock would be effective to make the image see-through for parallax scrolling and other such effects. If the status is 0, a normal putblock would be the fastest way of blasting it to the screen.

Parameters:

image_array	- Array of images in memory.
start	- Index of first image to update
end	- Index of last image to update

Return Value: none

See Also: **wset_imagebyte, wget_imagebyte, wget_imagebit**

Examples: None

wvertres

Function: Draws a previously stored image on the screen to fit within a given boundary but only stretches it vertically.

Declaration: **void wvertres (short x1, short y1 short y2, block image)**

Remarks: This is an extremely fast shrink/expand routine for blocks. It is similar to **wresize** but doesn't change the width of the block. Use this at all times if you are only resizing the height, since it is much faster.

Parameters:

x1	- Horizontal coordinate of image.
y1	- Upper Y coordinate of image.
y2	- Lower Y coordinate of image.
image	- Pointer to image which will be resized.

Return Value: None

See Also: **wresize**

Examples: 21

wwarp

Function: Displays a block on the screen, using different heights for each column displayed.

Declaration: **void wwarp (short x1, short x2, short *top, short *bot, block image, int mode)**

Remarks: Instead of drawing the block straight across in a rectangular manner, this function resizes the block vertically for every column displayed. You can have each column stretched by different amounts allowing for some very interesting effects. X1 and X2 are the two x coordinates the block will be resized between. Top and bot are 320 integer arrays which hold the y values for every x coordinate. Image is the block to warp. Mode is the same as the XRAY or NORMAL modes used in **wputblock**.

Use **wsline** to set up the top and bot arrays if you are using straight lines.

Example:

```
wsline (0, 199, 319, 0, &top);          // sets up line for top
wsline (0, 199, 319, 199, &bot);        // sets up bottom line
// Imagine drawing these two lines on the screen...

wwarp (0, 319, &top, &bot, warp_block);
```

Top should have all points above the corresponding point in bot. The block is resized between these points on the screen.

You do not need to use wsline to make straight lines for warping. Other possibilities include zig-zags, sine waves, etc.

Parameters:

x1	- Horizontal coordinate of left edge.
x2	- Horizontal coordinate of right edge.
top	- Array of y coordinates for upper edge.
bot	- Array of y coordinates for lower edge.
image	- Pointer to image which will be resized.
mode	- Either XRAY or NORMAL display mode

Return Value: None

See Also: **wsline**

Examples: 32

wwipe

Function: Draws a line but uses colors from another virtual screen.

Declaration: **void wwipe (short x1, short y1, short x2, short y2, block image)**

Remarks: This special effect allows you to draw a line on the screen, but instead of using one color,each pixel drawn is the color of the pixel at the same location on the block screen. You can create impressive wipes and make your programs look very professional. You will want to make one or more 'for' statements that copy the whole screen over since this only does it one line at a time. The lines don't have to be horizontal or vertical, so you can create complex patterns with the lines.

Parameters:

x1	- Horizontal coordinate of first endpoint.
y1	- Vertical coordinate of first endpoint.
x2	- Horizontal coordinate of second endpoint.
y2	- Vertical coordinate of second endpoint.
image	- Screen to use as the wipe source.

Return Value: None

Examples: 24

wxorbox

Function: Draws a filled rectangle using exclusive-or mode.

Declaration: **void wxorbox (short x, short y, short x2, short y2, unsigned char col)**

Remarks: This routine performs an XOR on the pixels within the box using the color given. It is useful for drawing outlines or showing button presses. If you draw a box on the same place twice in a row using the same color, the box will erase itself due to the results of the XOR mode.

Parameters:

x	- Horizontal coordinate of first endpoint.
y	- Vertical coordinate of first endpoint.
x2	- Horizontal coordinate of second endpoint.
y2	- Vertical coordinate of second endpoint.
col	- Color index used to draw the xor box.

Return Value: None

Examples: 36

WJOY_WC.LIB

Most games will need some method of detecting and reading from the joystick port(s) of the computer. These routines provide full access to 1 or 2 joysticks which are connected to the system. Routines for detection, calibration, and reading the button and positional values are all found in this library.

wcalibratejoystick

Function: Finds out the coordinate range of the joystick.

Declaration: **short wcalibratejoystick (joystick *joy)**

Remarks: Before using the joystick, it is best to calibrate it. Calibration simply finds out the range of values the joystick can return, and this is done by moving the stick to the extreme direction in each axis.

This routine can be used in two ways:
1. Tell the user to move the joystick to the upper left and press a fire button. Call **wcalibratejoystick**. Tell the user to move the joystick to the lower right and press a fire button. Call **wcalibratejoystick** again.

2. Tell the user to swirl the joystick a few times. This method only requires one **wcalibratejoystick** call.

If you want to calibrate the joystick in a setup program, you can save all the calibrated values in a configuration file, and load them back in for your main program. This way the user only has to calibrate the joystick once.

Here is the data structure you must save:

```
typedef struct {
   short x, y;
   short cenx, ceny;
   short port, buttons;
   short xrange, yrange;
   short scale;
} joystick;
```

Once the joystick has been calibrated, you must set the scale to the maximum absolute value you wish to receive. For example, if you set the scale to be 2000, the value returned by **wreadjoystick** will be between -2000 and +2000.

Parameters: joy - Pointer to the joystick information data structure.

Return Value: 1 if the joystick was calibrated 0 if the joystick was not calibrated.

See Also: **wcheckjoystick, winitjoystick, wreadjoystick**

Examples: 50

wcheckjoystick

Function: Determines which joysticks are connected.

Declaration: **short wcheckjoystick (void)**

Remarks: This routine checks for the presence of each joystick. It returns an integer with 2 bits set. If bit 1 is set, joystick 1 has been found. If bit 2 is set, joystick 2 has been found.

Parameters: None

Return Value: 0 = no joysticks found
1 = joystick 1 found
2 = joystick 2 found
3 = both joysticks found

See Also: **wcalibratejoystick, winitjoystick, wreadjoystick**

Examples: 50

winitjoystick

Function: Initializes a joystick.

Declaration: **void winitjoystick (joystick *joy, short joynum)**

Remarks: Initializes joystick number joynum using the joystick structure joy. This assumes the joystick is centered when you call the routine. joynum can be either 0 (for joystick 1) or 1 (for joystick 2).

You should calibrate the joystick after this routine.

Parameters: joy - Pointer to the joystick information structure.

joynum - Indicates joystick number to initialize.

Return Value: None

See Also: **wcalibratejoystick, wcheckjoystick, wreadjoystick**

Examples: 50

wreadjoystick

Function: Reads the information from a joystick.

Declaration: **short wreadjoystick (joystick *joy)**

Remarks: This routine reads the coordinates of the stick and button state and stores them in the joystick structure.

joy.x is the x coordinate
joy.y is the y coordinate
joy.buttons is the button status.

The buttons flag uses bits to determine which buttons are pressed, similar to the mouse button routines.

Parameters: joy - Pointer to the joystick information structure.

Return Value: 1 if successful, 0 if not successful

See Also: **wcalibratejoystick, wcheckjoystick, winitjoystick**

Examples: 50

WGT3D_WC.LIB

WGT 5.1 features a simple, but effective library of routines for rotating a set of points in all 3 dimensions. When used in combination with the polygon and triangle routines in the main library, this system can produce excellent wireframe and solid polygon-based animations.

winit3d

Function: Initializes the WGT 3D system.

Declaration: **void winit3d (void)**

Remarks: This command must be called in your program once, before the **wrotatepoints** command is used. It creates a sine and cosine lookup table that is used for 3D rotations. WGT's 3D rotation library is meant for simple applications only. The main limitation is that you cannot change the camera's location within the 3D coordinate system. You can move 3D objects around by changing their center point however.

Parameters: None

Return Value: None

See Also: **wrotatepoints, wsetrotation**

Examples: 54, 55

wrotatepoints

Function: Rotates a set of three dimensional points.

Declaration: **void wrotatepoints (point3d *orig_pointlist,**
point3d *rotated_pointlist, short maxpoint)

Remarks: The **wrotatepoints** command will take the array of 3D points called orig_pointlist, with maxpoint points, and rotate them around their axis. It stores the rotated points in the rotated_pointlist array. The rotation amounts are set with the command **wsetrotation**.

The point3d structure is defined as:

```
typedef struct
{
  short x, y, z;
  short sx, sy;
} point3d;
```

sx, and sy are used for storing colors and texture bitmap offsets when using **wtexturedpoly** and **wgouraudpoly**.

In addition, the following variables are used in WGT's 3D system:

```
int sx, sy, sz;                          /* Scale Factors */
short move_x, move_y, move_z;            /* Translational Offsets
                                 - Added to each point after  rotation */
short origin_x, origin_y, origin_z;       /* Point of origin.  All points will
                    be rotated around this point */
```

Parameters: pointlist - Pointer to array of original point3d structure.

rotated_pointlist - Pointer to array of rotated points.

maxpoint - Number of points in the original buffer.

Return Value: None

See Also: **winit3d, wsetrotation**

Examples: 54, 55

wsetrotation

Function: Sets the rotational amount in each axis that all points will be rotated by.

Declaration: **void wsetrotation (short rotate_x, short rotate_y, short rotate_z)**

Remarks: The **wrotatepoints** routine rotates all the points given to it by the amount set with this command. rotate_x, rotate_y and rotate_z can range between 0 and 360.

For example, to rotate the points by 30 degrees in the x axis:
wsetrotation (30,0,0);
wrotatepoints (&stpoint, &finpoint, 4);

Parameters:
- rotate_x - Degrees to rotate in the x-axis.
- rotate_y - Degrees to rotate in the y-axis.
- rotate_z - Degrees to rotate in the z-axis.

Return Value: None

See Also: **winit3d, wrotatepoints**

Examples: 54, 55

WFILE_WC.LIB

A graphical file selector is the easiest way to select files from large directory lists. This library contains a single routine which creates this selector. The file selector is capable of being moved to a different position on the screen by the user, and it provides full access to all available floppy disks and hard drives on the system. File listings are sorted alphabetically, with the drives and subdirectories at the top of each list.

Programs are required to load the font file "LITTLE.WFN" in order to properly display this selector. Larger fonts will not work properly with the display. Refer to the example programs to see how this works.

wfilesel

Function: Activates the WGT file selector routine.

Declaration: **char *wfilesel (char *mask, char *title, short x, short y, block image)**

Remarks: This routine displays a graphical file selector, and allows the user to scroll up and down through the file listing, change drives and directories, and pick a file, by using the mouse. This is the same file selector used in the Sprite Editor and Map Maker.

mask contains a file extension mask, such as "* " or "spr". It will automatically add a "*." to the file mask. It controls what kind of files will be listed in the file selector when first activated. The user can change this from within the selector if needed. It must be 3 characters long or the routine will always return NULL.

title contains the text which is displayed at the top of the selector. This is used to inform what file operation will be performed on the chosen file.

(x,y) is the coordinate where the selector will be drawn. It must fit completely on the screen. The user can move the file selector around, if you give a background screen, by clicking and holding on the title bar of the selector.

image is a full screen block which will be used to restore the background when moving the file selector. If image is NULL, the selector will be fixed at the (x,y) coordinates you pass, and it will not erase itself.

The file selector returns a pointer to a string containing the name of the file selected. If no file was chosen, it returns NULL.

Example usage:

```
char *filename;
...
filename = wfilesel("spr", "Load a sprite file", 10, 10, back);
printf ("filename was: %s", filename);
free (filename);          /* Free the filename once we're finished with it */
```

Parameters: mask - File extension to list as default.

title - Title for file selector.

	x	- Horizontal coordinate of selector on screen.
	y	- Vertical coordinate of selector on screen.
	image	- Pointer to buffer for background preservation.

Return Value: Pointer to the string which contains the filename.

Examples: 47

WSCR_WC.LIB

The most popular games in the past few years have all followed the same trend. Graphics are created using a set of repeated "tiles" to simulate a background. The tiles are placed on a map which is larger than the visual screen and this map is scrolled to show the various regions requested.

In the WGT system, the background tiles are created with the WGT Sprite Editor and may be any size up to 64*64 pixels. The tiles do not need to be square, although map design is simpler with square tiles. The WGT Map Maker is then used to create the map itself. On each map, up to 2000 sprites ("objects") may be placed in their starting positions. Each tile may be assigned a number indicating some property or value that the program may interpret. Maps may be up to 320*200 tiles in size. This results in a maximum pixel resolution of 20480*12800; a size much larger than any available video mode supports at the current time (thus the scrolling).

Maps may be superimposed on each other to produce an effect known as "parallax". Each map may be scrolled independently of the other maps which are loaded. Up to 50 maps may be loaded into memory at any given time.

Full source code for this library is provided so that you may customize the library limitations to suit your program’s needs. We are not responsible for the use of this code once it has been altered in any way.

is_in_window

Function: Determines if the coordinate given is inside the scrolling window.

Declaration: **short is_in_window (short currentwindow, short x, short y, short range)**

Remarks: This command will see if the coordinate (x,y) is within the scrolling window currentwindow. It is commonly used for playing sound effects triggered by a certain action. Since it would be annoying for sound to play because of an action happening somewhere else on the map, **is_in_window** comes in handy for this. range is a distance (in pixels, always >0) to extend the window's dimensions. A range of 0 would return 1 if the coordinate was exactly within the window. A value of 50 would increase the area to check by 50 pixels on each side. This is used in case the point is close to the window and the window may scroll over it in the next few frames.

Parameters:

currentwindow - Scrolling window containing the map.

x - World X coordinate to check.

y - World Y coordinate to check.

range - Number of pixels of to expand the test window by.

Return Value: 0 if the coordinate is outside the window (plus the range) 1 if the coordinate is inside the window (plus the range)

Examples: 62, 63

soverlap

Function: Detects collisions between two scrollsprites.

Declaration: **short soverlap (short s1, scrollsprite *wobjects1,
block *sprites1, short s2,
scrollsprite *wobjects2, block *sprites2)**

Remarks: This command checks for collisions between two scrollsprites. It uses a quick rectangular region check which sees if the bounding rectangles of the sprites overlap. This means the collision is not pixel accurate, but in most computer games, exact collisions are not required. If a collision is found, it will return 1, otherwise it returns 0.

If one of the scrollsprites is turned off (ie. scrollsprite[i].on == 0) then no collision is found.

soverlap is usually used in a loop where ranges of sprites are checked for collisions between each other.

Parameters:

s1	- Scrollsprite number of sprite 1
wobjects1	- Pointer to the array of scrollsprites for sprite 1
sprites1	- Pointer to the images used for sprite 1
s2	- Scrollsprite number of sprite 2
wobjects2	- Pointer to the array of scrollsprites for sprite 2
sprites2	- Pointer to the images used for sprite 2

Return Value: 0 if the sprites are not overlapping
1 if the sprite are overlapping

See Also: **wshowobjects**

Examples: 58, 59

wcopymap

Function: Makes a scrolling window share a map with another window.

Declaration: **void wcopymap (short sourcewindow, short destwindow)**

Remarks: This command allows you to have two scrolling windows on the screen at once which use the same map. This is perfect for two player games. Any changes made to the map with **wputworldblock** update both windows at the same time. Note that you can have as many windows sharing a map as you want. This is also useful for creating a reduced view map which scrolls along with a larger one.

Parameters: sourcewindow - Window containing loaded map.

destwindow - Window to assign duplicate map.

Return Value: None

Examples: 59, 67

wendscroll

Function: Releases memory used for a scrolling window.

Declaration: **void wendscroll (short currentwindow)**

Remarks: **wendscroll** deallocates the memory used to store a scrolling window's buffers. It must be called to shut down the scrolling system.

Parameters: currentwindow - Scrolling window to close.

See Also: **winitscroll**

Examples: 56, 57, 58, 59, 62, 63, 65, 67

wfreemap

Function: Releases memory used to store a map file.

Declaration: **void wfreemap (wgtmap mapname)**

Remarks: When loading a new map file, you must first free the memory used by the previous map before you use the same pointer.

Parameters: mapname - Pointer to the map in memory.

Return Value: None

See Also: **wloadmap**

Examples: 56, 57, 58, 59, 62, 63, 65, 67

wgetworldblock

Function: Returns the tile number at the given location in a map.

Declaration: **unsigned short wgetworldblock (short currentwindow, short posx, short posy)**

Remarks: **wgetworldblock** is used for detecting collisions between sprites and different parts of the map. posy and posy are world coordinates, so you may pass it the coordinates of a scrollsprite and add a pixel offset if needed.

It is essential for determining when the scrollsprites should react with their surrounding map.

Parameters:

currentwindow - Scrolling window of containing the map.

posx - X coordinate of tile. (in world coordinates)

posy - Y coordinate of tile. (in world coordinates)

Return Value: Tile number at the location given.

See Also: **wgetworldpixel, wputworldblock**

Examples: 58, 59, 62, 63

wgetworldpixel

Function: Returns the color of a pixel within a map.

Declaration: **unsigned char wgetworldpixel (short currentwindow, short x, short y)**

Remarks: **wgetworldpixel** will return the color of the pixel at (x,y) within the map. x and y are world coordinates. This command can be used for pixel-precise collision detection if you "color code" your tiles.

Parameters: currentwindow - Scrolling window containing the map.

x - World X coordinate of pixel.

y - World Y coordinate of pixel.

Return Value: Color of the pixel at the location given.

See Also: **wgetworldblock**

Examples: None

winitscroll

Function: Initializes one of the four scrolling windows.

Declaration: **void winitscroll (short currentwindow, short mode, short link, short windowwidth, short windowheight, block *tileref)**

Remarks: The WGT scrolling library has the ability to control 50 scrolling windows at a time. This command will initialize one of these windows. currentwindow is the window to initialize from 0 to 49. mode is either NORMAL (0) or PARALLAX (1). NORMAL windows are used as a base window meaning other PARALLAX windows may be shown overtop it. PARALLAX windows are like the xray mode in **wputblock** where color 0 is considered to be see-through.

link is used for PARALLAX windows only. To use a PARALLAX window, you must also have a NORMAL window which is shown underneath. link is the scrolling window number of this NORMAL window. Also note that you may have more than one NORMAL scrolling window.

windowwidth is the width (in tiles) of the scrolling window. windowheight is the height (in tiles) of the scrolling window. Both of these variables must be at least 1. The maximum value for these variables depends on the size of the tiles being used. When using a PARALLAX window, you must make sure the NORMAL window and the PARALLAX window have the same window dimensions in pixels. If you are using different sizes of tiles for each window, it may be impossible to achieve this. To calculate the window dimensions in pixel, multiply the size of the tiles by the size of the window.

Example of a correct scrolling window setup:
NORMAL window: tile size is 16x16, window size is 20x12 tiles
window width = 320 , window height = 192 (in pixels)
PARALLAX window: tile size is 40x32, window size is 8x6 tiles
window width = 320 , window height = 192 (in pixels)

tileref is a pointer to the block array containing the tiles. winitscroll looks at the first tile in
this array which isn't NULL to obtain the dimensions of the tiles.

Parameters: currentwindow - Scrolling window to initialize (0-49).

mode - Window mode : NORMAL or PARALLAX.

	link	- Number of related NORMAL window, only used if the mode is PARALLAX.
	wwidth	- Width of window (in tiles).
	wheight	- Height of window (in tiles).
	tileref	- Pointer to the array of tile images.

Return Value: None

See Also: **wloadmap, wscrollwindow, wshowwindow**

Examples: 56, 57, 58, 59, 62, 63, 65, 67

wloadmap

Function: Loads a tiled map into a scrolling window.

Declaration: **wgtmap wloadmap (short currentwindow, char *filename, short *tiletypes, scrollsprite *wobjects)**

Remarks: Maps that are created with the WGT Map Maker can be loaded into memory with this command. currentwindow can be a number from 0 to 49, which defines which scrolling window to load the map into. Each scrolling window may have different maps and use different tiles. filename is the name of the WGT map file (.WMP extension) to load.

tiletypes is a pointer to an array of 256 short ints which contains a number for each tile. These numbers can be used to group tiles into certain categories, such as solid or hollow.

wobjects is a scrollsprite structure to load the positions and sprite numbers of the objects in the map. wobjects must be large enough to hold all the objects. For example if the last object number used in a map is 678, you must define wobjects as:
scrollsprite wobjects[679];

Global Variables Set:
mapwidth[currentwindow] contains the width of the map in tiles.
mapheight[currentwindow] contains the height of the map in tiles.

Parameters:

currentwindow	- Scrolling window to load the map into (0-49).
filename	- Filename of the map to load.
tiletypes	- Pointer to an array of 256 short integers used for storing tiletypes.
wobject	- Pointer to an array of scrollsprites.

Return Value: A pointer to the WGT map in conventional memory. NULL is returned map could not loaded due to not enough memory, or the file could not be found.

See Also: **wfreemap**

Examples: 56, 57, 58, 59, 62, 63, 65, 67

wputworldblock

Function: Changes the tile number at the given location on a map.

Declaration: **void wputworldblock (short currentwindow, short posx, short posy, unsigned short tilenum)**

Remarks: **wputworldblock** is used for modifying the map while displaying it in a window. posx and posy are the map coordinates of the tile, and tilenum is the new tile number.

Parameters:

currentwindow	- Scrolling window of containing the map.
posx	- X coordinate of tile.
posy	- Y coordinate of tile.
tilenum	- New tile number

Return Value: None

See Also: **wgetworldblock**

Examples: 58, 59, 62, 63

wsavemap

Function: Saves a WGT map stored in memory into a file.

Declaration: **void wsavemap (short currentwindow, char *filename, wgtmap savemap, short *tiletypes, scrollsprite *wobjects, short numobj)**

Remarks: This command will save the current map in the given scrolling window, along with all of the scrollsprite positions and tiletypes. currentwindow is the scrolling window containing the map to save. filename is the map file to create. savemap is the wgtmap pointer which points to the map data. tiletypes is an array of 256 short integers for the tile types.

wobjects is an array of scrollsprites holding the information about the sprites in the map. numobj contains the size of the wobjects array.

Parameters:

currentwindow	- Scrolling window containing the map to save.
filename	- Filename of the map file to save.
savemap	- Pointer to the map in memory.
tiletypes	- Pointer to the array of tile types. (256 short integers)
wobjects	- Pointer to the array of scrollsprites.
numobj	- Size of the wobjects array.

Return Value: None

See Also: **wfreemap, wloadmap**

Examples: 56

wscreen_coordx

Function: Returns the absolute X screen coordinate of a world coordinate.

Declaration: **short wscreen_coordx (short currentwindow, short xcoord)**

Remarks: This command is used to find out if a sprite is on the screen and where it appears. It can be used for sound effects panning or to find out the scrolling speed of a window.

If returned coordinate is between 0 and the window width, it is within the window.

Parameters: currentwindow - Window to base coordinate system.

xcoord - World coordinate on map.

Return Value: Screen coordinate of the object.

See Also: **wscreen_coordy**

Examples: 67

wscreen_coordy

Function: Returns the absolute Y screen coordinate of a world coordinate.

Declaration: **short wscreen_coordy (short currentwindow, short ycoord)**

Remarks: This command is used to find out if a sprite is on the screen and where it appears. It can be used for sound effects panning or to find out the scrolling speed of a window.

Parameters: currentwindow - Window to base coordinate system.

ycoord - World coordinate on map.

Return Value: Screen coordinate of the object.

See Also: **wscreen_coordx**

Examples: 67

wscrollwindow

Function: Scrolls a window by a distance vertically and horizontally.

Declaration: **void wscrollwindow (short currentwindow, short windowspeedx, short windowspeedy)**

Remarks: **wscrollwindow** scrolls the given window by a number of pixels in any direction. It must be called continuously to update the scrolling window, even if the scrolling distances are 0.

windowspeedx is the number of pixels to move in the x direction. A negative value for windowspeedx will scroll to the left.

windowspeedy is the number of pixels to move in the y direction. A negative value for windowspeedy will scroll upwards.

When using parallax scrolling with many layers, the NORMAL layer must be the first window to be scrolled/drawn. Each parallax layer will be added to the image from back to front when you use **wscrollwindow** to draw it. Since each layer is constructed separately, it easy to change the order of the parallax layers, and draw sprites between layers.

Parameters: currentwindow - Scrolling window of the map to scroll (0-49).

windowspeedx - Number of pixels to scroll in X direction.

windowspeedy - Number of pixels to scroll in Y direction.

Return Value: None

See Also: **winitscroll, wshowwindow**

Examples: 56, 57, 58, 59, 62, 63, 65, 67

wshowobjects

Function: Displays a range of scrollsprites on the scrolling window.

Declaration: **void wshowobjects (short currentwindow, short start, short end, block *image_array, scrollsprite *wobjects)**

Remarks: **wshowobjects** provides an easy way to display your scrollsprites over the scrolling background. start and end refer to the starting and ending indexes of the scrollsprite.

array wobjects. All sprites within this range will be drawn from the lowest index to the highest. Lower indexes will are therefore drawn behind higher indexes. You can also change the order the sprites are drawn by splitting your sprites into ranges, and draw them with a number of calls to **wshowobjects**. Sprites may be placed behind a PARALLAX layer by showing them before the layer is drawn with the **wscrollwindow** command.

Parameters:

currentwindow	- Scrolling window containing the map.
start	- Number of first scrollsprite to show.
end	- Number of last scrollsprite to show.
image_array	- Pointer to the array of images for sprites.
wobjects	- Pointer to the array of scrollsprites.

Return Value: None

See Also: **soverlap, wscrollwindow**

Examples: 56, 58, 59, 62, 63, 67

wshowwindow

Function: Sets the location of the world within a scrolling window.

Declaration: **void wshowwindow (short currentwindow, short posx, short posy)**

Remarks: **wshowwindow** is used to set where the window will be located within the map. It is called once before **wscrollwindow** to set the initial viewing coordinates, and may be used while

scrolling to instantly change the viewing coordinates. posy and posy are the coordinates, in tile measurements, of the new viewing coordinates.

Parameters:

currentwindow	- Scrolling window to set viewpoint.
posx	- Initial X coordinate in map. (in map coords)
posy	- Initial Y coordinate in map. (in map coords)

Return Value: None

See Also: **winitscroll**

Examples: 56, 57, 58, 59, 62, 63, 65, 67

WMENU_WC.LIB

User interfaces can be so confusing at times, and so helpful in others. This library provides a set of routines for providing dropdown menus (such as used in a windowing system). For examples of this library in action, take a look at the Map Maker which is included with this release of WGT. The program uses the dropdown menu system as the core decision maker of the program logic flow.

Full source code for this library is provided so that you may customize the menu limitations to suit your program's needs. We are not responsible for the use of this code once it has been altered in any way.

checkmenu

Function: Polls the mouse until the users clicks a button, and returns which menu choice was selected.

Declaration: **short checkmenu (void)**

Remarks: **checkmenu** loops until the user selects a choice from the menu bar or clicks the mouse button outside of the menu bar. If a selection is made, it returns the number of the menu choice.

The return value is a combination of the menu bar title and the option within that bar. For each menu bar, multiply by 10, and add the choice within the menu.

For example, if you selected the menu below, it would return the number 11.
dropdown[1].choice[1]=" Save ";

Mathematically:
result = menu*10 + choice
= 1*10 + 1
= 11

If the mouse is clicked outside the menu, it returns -1.

Parameters: None

Return Value: The number of the menu choice selected, or -1 if the mouse was clicked elsewhere.

See Also: **initdropdowns, removemenubar, showmenubar**

Examples: 40

initdropdowns

Function: Initializes the drop down menu system.

Declaration: **void initdropdowns (void)**

Remarks: You must call this after you have defined the menu bar and drop down menus. If you change the text in the menus, such as toggling choices, you may need to call this again to get the correct sizes for the drop down menus.

Parameters: None

Return Value: None

See Also: **checkmenu, removemenubar, showmenubar**

Examples: 40

removemenubar

Function: Turns off the menu bar at the top of the screen.

Declaration: **void removemenubar (void)**

Remarks: This command restores the screen underneath the menu bar and shuts down the menu system.

Parameters: None

Return Value: None

See Also: **checkmenu, initdropdowns, showmenubar**

Examples: 40

showmenubar

Function: Displays the menu bar at the top of the screen.

Declaration: **void showmenubar (void)**

Remarks: This will activate the menu you have defined, and display the menu bar at the top of the screen.

Parameters: None

Return Value: None

See Also: **checkmenu, initdropdowns, removemenubar**

Examples: 40

WSPR_WC.LIB

Some of the simplest and most entertaining games are created with the use of animated characters over a non-changing background. This library of routines provides you with the ability to animate images created with the WGT Sprite Editor. The full range of motion, animation and object collision is provided through these routines.

This library is NOT compatible with the WSCR_WC.LIB file. This library can not be used to produce animations over a tiled background created with our other utilities. Limitations to the number of "sprites" onscreen and the specialization of the routines themselves has prompted us to provide full source code for this library. Feel free to alter the code and recompile the library file to meet your own personal needs.

We are not responsible for the use of this code once it has been altered in any way.

animate

Function: Sets the animation sequence for a sprite.

Declaration: **void animate (short spritenum, char *animation_sequence)**

Remarks: **animate** sets up animation for a sprite. Spritenum is the sprite number to animate. The animation_sequence string takes the following form:

"(arrnumber,delay)(arrnumber2,delay2)...(arrnumberN,delayN)R"

Between each set of brackets is the information required to animate the sprite. The first number is the sprite array number, and the second is how long that sprite number is displayed. You then make another set and use different sprite numbers and delays. If you place a CAPITAL R at the end of the string, the animation will repeat itself when it gets to the end. If you leave the R out, the animation will occur once, and stop. You can have up to 40 sets in the animation sequence.

For example:
To animate sprite number one between two sprites, and repeat:

```
Animate(1,"(1,0)(2,1)R");
```

After calling this procedure, you must turn the animation on with animon.

Parameters:

spritenum	- Sprite number to animate.
animation_sequence	- Pointer to the animation control string.

Return Value: None

See Also: **animoff, animon**

Examples: 22, 23, 25, 61

animoff

Function: Freezes animation for the sprite.

Declaration: **void animoff (short spritenum)**

Remarks: **animoff** will temporarily turn off the animation sequence for the given sprite. animation will resume from the point it stopped when animon is called.

Parameters: spritenum - Sprite numver to freeze animation.

Return Value: None

See Also: **animate, animon**

Examples: 25

animon

Function: Turns animation on for a sprite.

Declaration: **void animon (short spritenum)**

Remarks: Animation sequences are set with the **animate** command. To activate the sequence, the sprite's animation must be turned on. Each time **draw_sprites** is called, the animation will advance by one frame, or delay unit.

Parameters: spritenum - Sprite number to turn on animation.

Return Value: None

See Also: **animate, animoff**

Examples: 22, 23, 25, 61

deinitialize_sprites

Function: Deinitializes the sprite system.

Declaration: **void deinitialize_sprites (void)**

Remarks: Before quitting your program, or calling **initialize_sprites** again, you must first shut down the sprite system with **deinitialize_sprites**. This will free the memory used by the sprites on the screen.

Parameters: None

Return Value: None

See Also: **initialize_sprites**

Examples: 22, 23, 25, 60, 61

draw_sprites

Function: Draws the sprites and updates movement and animation counters.

Declaration: **void draw_sprites (int movement_multiplier)**

Remarks: **draw_sprites** will first update the animation and movement of all the sprites, and then draw them in the correct location on the screen. movement_multiplier is the number of times to perform the movement and animation for each sprite. This can used so the sprites move at a constant speed regardless of how fast the computer is at drawing them.

Parameters: movement_multiplier - Number of times to move and animate the sprites

Return Value: None

See Also: **erase_sprites**

Examples: 22, 23, 25, 60, 61

erase_sprites

Function: Erases the sprites from the screen.

Declaration: **void erase_sprites (void)**

Remarks: **erase_sprites** will remove the sprites from the virtual screen spritescreen, in preparation for drawing them in a new location. They will not disappear from the visual screen. To remove a sprite from the visual screen requires you to call **erase_sprites**, turn the sprite off using **spriteoff**, and call **draw_sprites** which updates the screen.

Parameters: None

Return Value: None

See Also: **draw_sprites**

Examples: 22, 23, 25, 60, 61

initialize_sprites

Function: Initializes the sprite library.

Declaration: **void initialize_sprites (block *sprite_blocks)**

Remarks: **initialize_sprites** sets up some internal variables, and makes two virtual screens called spritescreen and backgroundscreen.

It assumes you have loaded sprites into an array called sprite_blocks.

You must call **initialize_sprites** after you load in the sprites using **wloadsprites**. If you need to call thismore than once in a program, you must use the **deinitialize_sprites** command first.

Parameters: sprite_blocks - Array of images used for all sprites

Return Value: None

See Also: **deinitialize_sprites**

Examples: 22, 23, 25, 60, 61

movex

Function: Sets the horizontal movement of a sprite.

Declaration: **void movex (short spritenum, char *movement_sequence)**

Remarks: **movex** works similar to **animate** only there are 3 numbers in each set. You can have up to 15 sets in the movement sequence.

movex (3, "(1,50,1)(-1,50,0)R");

The first number in the set is added to the current x coordinate of the sprite. Therefore if the sprite is at 100,50 on the screen, and you make the number a 5, the sprite will move to 105,50. This number can be anything, so you can have the sprite move right (positive numbers), left (negative numbers) or nowhere (zero).

The second number in the set is the number of times to move the sprite before going on to the next set. It must always be greater then 0.

The third number in the set is the delay for each time the sprite moves. Try using 0 at first and increase it to slow the sprites down.

If the set was (1,50,1), then the sprite would move to the right one pixel, 50 times, with a delay of 1 for each move. Then the next set would be activated. If you have a CAPITAL R at the end of the string, the movement will repeat.

For example, to make sprite 2 move back and forth on the screen continuously:

movex(2,"(1,200,0)(-1,200,0)R");

To make a sprite move from the left side to the right, and jump back to the left again:

movex(2,"(1,319,0)(-319,1,0)R");

Parameters: spritenum - Sprite number to set movement.

movement_sequence - Movement sequence string.

Return Value: None

See Also: **movexoff, movexon**

Examples: 22, 25, 61

movexoff

Function: Turns horizontal movement for a sprite off.

Declaration: **void movexoff (short spritenum)**

Remarks: This will temporarily turn off the horizontal movement of a sprite. It can be turned back on with **movexon**.

Parameters: spritenum - Sprite number to deactivate movement.

Return Value: None

See Also: **movex, movexon**

Examples: 25

movexon

Function: Turns horizontal movement for a sprite on.

Declaration: **void movexon (short spritenum)**

Remarks: This will turn the horizontal movement for a sprite on. The movement sequence must have been defined previously with **movex**.

Parameters: spritenum - Sprite number to activate movement.

Return Value: None

See Also: **movex, movexoff**

Examples: 22, 25, 61

movey

Function: Sets up vertical movement for a sprite.

Declaration: **void movey (short spritenum, char *movement_sequence)**

Remarks: This command works the same way as **movex**, only it sets the vertical movement of the sprite. See **movex** for the movement_sequence format.

Parameters: spritenum - Sprite number to set movement.

movement_sequence - Movement sequence string.

Return Value: None

See Also: **moveyoff, moveyon**

Examples: 25, 61

moveyoff

Function: Turns vertical movement for a sprite off.

Declaration: **void moveyoff (short spritenum)**

Remarks: This will temporarily turn off the vertical movement of a sprite. It can be turned back on with **moveyon**.

Parameters: spritenum - Sprite number to deactivate movement.

Return Value: None

See Also: **movey, moveyon**

Examples: 25

moveyon

Function: Turns vertical movement for a sprite on.

Declaration: **void moveyon (short spritenum)**

Remarks: This will turn the vertical movement for a sprite on. The movement sequence must have been defined previously with **movey**.

Parameters: spritenum - Sprite number to activate movement.

Return Value: None

See Also: **movey, moveyoff**

Examples: 25, 61

overlap

Function: Tests if two sprites overlap each other on the screen.

Declaration: **short overlap (short spritenum_1, short spritenum_2)**

Remarks: **overlap** checks the bounding rectangles of the two sprites and returns a 1 if they are overlapping. This is used for collision detection between the sprites.

Parameters: spritenum_1 - Sprite number of first sprite.

spritenum_2 - Sprite number of second sprite.

Return Value: 1 if sprites overlap, 0 if they do not.

Examples: 5, 20, 61

spriteoff

Function: Turns off a sprite.

Declaration: **void spriteoff (short spritenum)**

Remarks: This command frees the memory used by a sprite, and removes it from the screen during the next call to **draw_sprites**.

Parameters: spritenum - Sprite number to turn off.

Return Value: None

See Also: **spriteon**

Examples: 22, 23, 25, 61

spriteon

Function: Turns a sprite on at the given location.

Declaration: **void spriteon (short spritenum, short xcoord, short ycoord, short arrnumber)**

Remarks: **spriteon** turns a sprite on at the coordinates, using the arrnumber block from the sprites array. Arrnumber is the same as the sprite numbers in the WGT Sprite Editor. Spritenum can range from 0 to 39. This means you can have 40 sprites on the screen at once. Each call to **spriteon** must be followed later in the program by a **spriteoff**.

Parameters:

spritenum	- Sprite number to turn on (0-39).
xcoord	- X coordinate of sprite.
ycoord	- Y coordinate of sprite.
arrnumber	- Sprite image number.

Return Value: None

See Also: **spriteoff**

Examples: 22, 23, 25, 60, 61

WFLIC_WC.LIB

Version 5.1 of WGT has support for both FLI and FLC format animation files. These files are created using commercial software programs or a shareware program called Dave's Targa Animation.

openflic

Function: Opens an FLI or FLC animation file to be played.

Declaration: **int openflic (char *filename, int mode_flic, int update_colors)**

Remarks: This routine will open an animation file which has been created using the FLI or FLC file format created by Autodesk. These animation files may be played from either disk or memory. Disk accesses are slow, so it is advisable to use memory if possible.

mode_flic can be one of the following:

FLIC_DISK	0
FLIC_MEMORY	1

The function will return an error value if the file could not be found, or if it is not the proper resolution. The error codes are defined as follows:

FLIC_OK	0
FLIC_NOTFOUND	1
FLIC_INVALIDRES	2
FLIC_INVALIDHEADER	3
FLIC_INVALIDFRAME	4
FLIC_INVALIDCHUNK	5
FLIC_DONE	6

Parameters: filename - The full pathname of the animation to load.

mode_flic - The method to show the animation

update_colors - If true, uses the flic's palette

Return Value: Number indicating error during file open (if any).

See also: **closeflic, nextframe**

Examples: 52

nextframe

Function: Advances an open animation file to the next frame.

Declaration: **int nextframe (void)**

Remarks: Once an animation file has been opened with **openflic**, this command is used to update the animation to the next frame. An internal counter keeps track of the last frame played, and the function will return FLIC_DONE (see include file) when it reaches the final frame. If this value is ignored, subsequent calls to nextframe will start the animation over again.

Parameters: None

Return Value: Integer indicating error status or completion status of animation. See **openflic** for the error codes.

See Also: **closeflic, copyflic, openflic**

Examples: 52

copyflic

Function: Copies the animation buffer to the visual screen.

Declaration: **void copyflic (void)**

Remarks: If you are using a secondary screen buffer to perform animations, this command will copy from the virtual screen to the visual screen. The virtual screen must have been created by calling **wnewblock** or **wallocblock**, and you must set the flicscreen variable to point to the buffer.

For example (small FLI/FLC player using virtual screen):

```
block buffer;
buffer = wnewblock (0, 0, 319, 199);      /*Allocate our buffer*/
flicscreen = buffer;                      /*Assign it*/
result = openflic ("myflic.flc", FLIC_DISK, 1);/*Open the file*/
if (result == FLIC_OK)                    /*If successful*/
{
  while (nextframe () == FLIC_OK)         /*and not done flic*/
  {
    copyflic ();                          /*Copy to screen*/
    delay (flicheader.speed);             /*Proper speed*/
  }
  closeflic ();                           /*Close file*/
}
wfreeblock (buffer);                      /*Free buffer*/
```

Parameters: None

Return Value: None

See Also: **closeflic, nextframe, openflic**

Examples: None

closeflic

Function: Closes a previously opened animation file.

Declaration: **void closeflic (void)**

Remarks: If an animation file has been opened successfully with the **openflic** command, this routine will close the file.

Parameters: None

Return Value: None

See Also: **openflic**

Examples: 52

WVESA_WC.LIB

WGT now supports SVGA video modes through the use of a VESA video driver. There are currently a limited number of supported graphics routines for these modes, however they will be expanded over time to include the full set of WGT primitives. X and Y coordinates will depend on the video mode being used. The following commands are SVGA versions of main library commands:

```
void   wvesa_clip (short x, short y, short x2, short y2);
void   wvesa_cls (short color);
void   wvesa_putpixel (short x, short y);
short wvesa_getpixel (short x, short y);
void   wvesa_line (short x, short y, short x2, short y2);
void   wvesa_hline (short x, short x2, short y);
void   wvesa_bar (short x, short y, short x2, short y2);
void   wvesa_rectangle (short x, short y, short x2, short y2);
void   wvesa_copyscreen (short sx, short sy, short sx2, short sy2, block src, short dx,
                  short dy, short method);
block wvesa_newblock (short x, short y, short x2, short y2);
void   wvesa_putblock (short dx, short dy, block src, short method);
void   wvesa_outtextxy (short x, short y, wgtfont fnt , char *printit);
```

NOTE: The wvesa_copyscreen contains an additional parameter compared to the normal wcopyscreen command. This parameter allows solid or xray copy modes.

wvesa_bank

Function: Switches to a different bank on a SVGA card.

Declaration: **short wvesa_bank (short bank)**

Remarks: Generally this command will not be needed. It is included for those who wish to write their own video routines for VESA. You must call this command whenever you want to write to video ram outside the current bank number. The size (in bytes) of the banks are stored in the VESAmode.banksize global variable. To initialize this structure, you must first call **wvesa_getmodeinfo**. See the WGTVESA include file for further details on the global structures available to the programmer.

Parameters: bank - Short integer indicating bank number to switch to. This number will vary in range depending on the video card and the video mode selected.

Return Value: 0 - Successful call.
1 - Error during call.

See Also: **wvesa_detected, wvesa_getmodeinfo, wvesa_init**

Examples: 43

wvesa_detected

Function: Returns 1 if a VESA (SVGA) driver is detected.

Declaration: **short wvesa_detected (void)**

Remarks: To enable a Super VGA video mode, WGT requires the presence of a VESA video driver. These drivers are available with the software included with most SVGA cards, and may be obtained from a local BBS if needed. Do not attempt to use any of the remaining commands in this library if the function returns a 0.

The most common driver is called UNIVESA or UNIVBE. Drivers should support the VESA standards version 1.2 or better to ensure proper program execution. DO NOT trust the video card to provide VESA support without a driver even if the manufacturer claims VESA compatibility.

Parameters: None

Return Value: 0 - VESA driver not found.
1 - Driver was found.

See Also: **wvesa_init**

Examples: 43, 53, 63, 66

wvesa_getmodeinfo

Function: Stores video mode information in the VESAmode global structure.

Declaration: **short wvesa_getmodeinfo (short mode)**

Remarks: After detecting a VESA driver, the user may wish to determine which modes are supported by the video card. This information may be obtained by calling **wvesa_supported**. If the function returns a 1, you may then call **wvesa_getmodeinfo** with the same mode value to get mode specific information. The information is stored in the VESAmode global variable after a successful call. See the WGTVESA include file for a detailed description of the structure and its contents.

Parameters: mode - Video mode to obtain information on.

Return Value: 0 - Call was successful.
1 - Error during call.

See Also: **wvesa_detected, wvesa_init, wvesa_supported**

Examples: 43, 53, 63, 66

wvesa_init

Function: Initializes a SVGA video mode using a VESA driver.

Declaration: **short wvesa_init (short mode)**

Remarks: Instead of calling **vga256** to initialize graphics mode, this routine is used to start a SVGA video mode. The value passed as the video mode may be determined by calling **wvesa_supported** or pre-determined by the programmer. If the video mode is not supported by the video card or the driver, this function will return an error value and the video mode will not be initialized. This function must be called before any graphics operations are performed in SVGA modes.

The VESAmode global structure will contain information about the video mode after a successful call to **wvesa_init**.

Parameters: mode - SVGA video mode to initialize.

Return Value: 0 - Call was successful.
1 - Error during call, video mode not supported.

See Also: **wvesa_detected, wvesa_getmodeinfo, wvesa_supported**

Examples: 43, 53, 63, 66

wvesa_setlogical

Function: Sets the logical scanline to write to in the SVGA memory system. This can be larger than the physical screen dimensions.

Declaration: **void wvesa_setlogical (unsigned short startline)**

Remarks: This function is used to alter the page in video memory where WGT outputs its graphics.

For example, on a 640x400 video mode with 1024k of video ram, 4 pages are supported. Using 400 as a parameter, WGT will now direct output to the second page of video memory. Coordinates are zero-based regardless of video page, so a y value of 50 would actually be referring to the 450th scanline in video ram. Clipping should be altered with **wvesa_clip** from this point on to ensure that the logical output is considered (the standard wclip function does not take logical VESA output into account).

Parameters: startline - Scanline which will become y=0 for all VESA output.

See Also: **wvesa_setphysical**

Examples: BBSDEMO

wvesa_setphysical

Function: Sets the physical screen display region. The leftmost pixel displayed and the starting scanline are parameters. Useful for page flipping techniques.

Declaration: **void wvesa_setphysical (unsigned short left, unsigned short startline)**

Remarks: This function provides access to page flipping and panning using a VESA driver. Panning is achieved by altering the leftmost pixel displayed. Page flipping or vertical scrolling is done bye changing the scanline which is displayed at the top of the screen. The panning techniques work best when combined with **wvesa_setwidth** so that the display does not wrap images around when shifting.

Page flipping will support as many screens as the video card has memory for. This number depends on the video mode and the amount of RAM on the card. Check VGA.TotalMemory and VESAmode.NumberOfImages to verify this for each mode.

To draw on different pages in video ram, use the **wvesa_setlogical** function.

Parameters: left - Leftmost pixel to display on screen.

startline - Scanline to be displayed at top of screen.

See Also: **wvesa_setwidth, wvesa_setlogical**

Examples: BBSDEMO

wvesa_setwidth

Function: Sets the logical SVGA screen width.

Declaration: **short wvesa_setwidth (unsigned short width)**

Remarks: Sets the logical screen width. This should be done before any graphics are drawn on the screen. The video card must have sufficient memory to support the number of bytes used per scanline after this function is called.

Parameters: width - Pixels per scanline (logical).

See Also: **wvesa_setlogical**

Examples: BBSDEMO

wvesa_supported

Function: Returns status of VESA and video card support for a given SVGA video mode.

Declaration: **short wvesa_supported (short mode)**

Remarks: Given a SVGA video mode number, this routine will return the status of VESA support for the mode. Different video cards will support different video modes, so you must make sure that the mode is supported before intializing graphics mode and executing the rest of the program.

Once support has been verified, the video mode may be initialized, or a call to **wvesa_getmodeinfo** can be made to determine information about the mode.

Parameters: mode - SVGA video mode number.

Return Value: 0 - Mode is NOT supported.
1 - The given video mode is supported on this card.

See Also: **wvesa_detected, wvesa_getmodeinfo, wvesa_init**

Examples: 43, 53, 66

WLINK_WC.LIB

This library allows for direct serial link and modem communications between two computers. It has routines for sending commands to the modem, answering and dialing, and sending and receiving data through interrupts. Full source code is provided for this library so that you may alter or expand the existing system.

We are not responsible for the use of this code once it has been altered in any way.

wlink_answer

Function: Waits for a call, and connects when a ring occurs.

Declaration: **int wlink_answer (void)**

Remarks: This command will wait for the modem to respond with a ring. If a ring is detected, it will connect with the remote modem.

Parameters: None

Return Value: 0 if a ring was not detected
1 if a ring was detected and the modem connected

See Also: **wlink_dial, wlink_modemcommand**

Examples: 38

wlink_dial

Function: Dials a number

Declaration: **int wlink_dial (char *dialstring)**

Remarks: This command will dial the number contained in dialstring. If the modem connects with another computer, it will return 1, otherwise it returns 0.

Parameters: dialstring - The number to dial

Return Value: 1 if the modem connects

0 if the modem did not connect

See Also: **wlink_answer, wlink_modemcommand**

Examples: 38

wlink_flush_incoming

Function: Flushes the incoming communications buffer.

Declaration: **void wlink_flush_incoming (void)**

Remarks: This command will reset the incoming communications buffer and ignore any characters which were received but not processed.

Parameters: None

Return Value: None

See Also: **wlink_flush_outgoing**

Examples: None

wlink_flush_outgoing

Function: Flushes the outgoing communications buffer.

Declaration: **void wlink_flush_outgoing (void)**

Remarks: This command will reset the outgoing communications buffer and ignore any characters which are in the buffer but have not been sent.

Parameters: None

Return Value: None

See Also: **wlink_flush_incoming**

Examples: None

wlink_getuart

Function: Finds out what kind of UART is present.

Declaration: **void wlink_getuart (void)**

Remarks: This command will determine what kind of UART is present on your computer. You must determine this before you can start your communications session.

Parameters: None

Return Value: None

Examples: 38

wlink_getvector

Function: Finds out which interrupt vector to use for communications.

Declaration: **void wlink_getvector (void)**

Remarks: This command will determine which interrupt is available to use for the communications driver. You must determine this before you can start your communications session.

Parameters: None

Return Value: None

Examples: 38

wlink_hangup_modem

Function: Sends the hangup string to the modem.

Declaration: **void wlink_hangup_modem (void)**

Remarks: This command wil send the hangup string found in the link_info structure.

Parameters: None

Return Value: None

Examples: None

wlink_initmodem

Function: Sends the modem initialization string.

Declaration: **int wlink_initmodem (void)**

Remarks: This command will send the initialization string found in the link_info structure to the modem. If the modem responds with "OK", 1 is returned. If the modem did not understand the initialization command, it will return 0. **wlink_initmodem** will always return 1 if the initialization string is empty.

Parameters: None

Return Value: 1 if the modem responded ok

0 if the modem did not respond to the string

Examples: None

wlink_initport

Function: Initializes a communications session.

Declaration: **void wlink_initport (void)**

Remarks: You must have the link_info structure filled with the necessary data before you call this procedure. It will use the values stored in this structure to initialize the port and IRQ.

Parameters: None

Return Value: None

Examples: 38

wlink_modemcommand

Function: Sends a command to the modem.

Declaration: **void wlink_modemcommand (char *str)**

Remarks: This command is used to send commands to the modem. It can be used to dial numbers, alter settings, or hang up the line.

Parameters: str - The string to send to the modem

Return Value: None

Examples: None

wlink_modemresponse

Function: Flushes the outgoing communications buffer.

Declaration: **int wlink_modemresponse (char *resp)**

Remarks: This command waits for the modem to respond, and checks for a certain response. If the modem returns the string in resp, the routine returns 1, otherwise it returns 0. It is used after sending a command to the modem, in order to see if the command was processed properly.

Parameters: resp - A string containing the proper response.

Return Value: 1 if the correct response was received

0 if the response was incorrect

Examples: None

wlink_read_byte

Function: Reads a single byte from the incoming data buffer.

Declaration: **short wlink_read_byte (void)**

Remarks: This command will return the next byte found in the incoming data buffer. If the buffer is empty, it will return -1.

Parameters: None

Return Value: The next byte in the incoming buffer, or -1 if the buffer is empty.

Examples: 38

wlink_shutdownport

Function: Closes a communications session.

Declaration: **void wlink_shutdownport (void)**

Remarks: This command will reset the IRQ, and close the communications session for the port used.

Parameters: None

Return Value: None

Examples: 38

wlink_write_buffer

Function: Sends a number of bytes over the communications link.

Declaration: **void wlink_write_buffer (char *buffer, unsigned int count)**

Remarks: This command will copy count bytes from buffer into the outgoing data buffer.

Parameters: buffer - Contains the bytes to send

count - The number of bytes to send

Return Value: None

Examples: 38