SciTech MGL

Professional Graphics Library

Getting Started and Programmer's Guide

Version 4.0

SciTech Software, Inc.
505 Wall Street
Chico, CA 95928

Main: (530) 894-8400
FAX: (530) 894-9069
www.scitechsoft.com

SciTech Software, Inc.
505 Wall Street
Chico, CA 95928 USA
(530) 894-8400
(530) 894-9069 FAX

SciTech MGL Software License Agreement

BEFORE PROCEEDING WITH THE INSTALLATION AND/OR USE OF THIS SOFTWARE, CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS OF THIS LICENSE AGREEMENT AND LIMITED WARRANTY (The "Agreement").

BY INSTALLING OR USING THIS SOFTWARE YOU INDICATE YOUR ACCEPTANCE OF THIS AGREEMENT. IF YOU DO NOT ACCEPT OR AGREE WITH THESE TERMS, YOU MAY NOT INSTALL OR USE THIS SOFTWARE!

PREAMBLE

The terms and conditions of the SciTech MGL Software License Agreement have one major goal in mind; to foster a development community based around the SciTech MGL graphics library and associated source code. SciTech Software does however reserve the right as the sole distributor of the library source code. Hence although we encourage you to change and modify the library to suit your needs, you may not distribute derivative works based on the library source code without express written permission from SciTech Software. Worthwhile derivative works such as ports to other operating systems may be submitted to SciTech Software and we will make them available to the general public via our web pages and ftp site. Worthwhile changes and modifications to the libraries may be submitted to SciTech Software for integration into a future release of the product.

Note that the intent of this license agreement is also to foster development and use of the VESA VBE/Core 3.0, VBE/AF 2.0 standards on Intel based systems, regardless of the operating system. Specifically we want to allow developers to use portions of SciTech MGL in other products and libraries related to using these standards, as well as the WinDirect runtime libraries for Windows 3.1 and Windows 95. Although we do require that you obtain written consent from SciTech Software in order to use portions of the code in your own products and libraries, in general we will not withhold this consent from other developers.

LICENSE

This software, including documentation, source code, object code and/or additional materials (the "Software") is owned by SciTech Software and is protected by copyright law and international treaty provisions. This Agreement does not provide you with title or ownership of Product, but only a right of limited use as outlined in this license agreement. SciTech Software hereby grants you a non-exclusive, royalty free license to use the Software as set forth below:

- integrate the Software with your Applications, subject to the redistribution terms below.
- modify or adapt the Software in whole or in part for the development of Applications based on the Software.
- use portions of the SciTech MGL source code in your own products and libraries (such as the VBE/Core 3.0, VBE/AF 2.0 and WinDirect components), provided you obtain prior written consent from SciTech Software.
- distribute the official Object Code only releases of the Software to other parties for free, (shareware distribution companies may charge their normal shipping and handling fees). Please also note that distribution of the Object Code only releases may only be through the normal shareware distribution channels as a single, complete package.

REDISTRIBUTION RIGHTS

You are granted a non-exclusive, royalty-free right to reproduce and redistribute executable files created using the Software (the "Executable Code") in conjunction with software products that you develop and/or market (the "Applications"). You may also be granted rights to reproduce and distribute components of the Software specified in the "Redistributable Components" section of the "Getting Started" manual.

RESTRICTIONS

Without the expressed, written consent of SciTech Software, you may NOT:

- re-distribute any portion of the Source Code, in whole or in part except as expressly allowed above. SciTech Software reserves the right as the sole distributor of the complete Source Code, which can only be download from SciTech Software's internet sites.
- modify, or distribute the documentation for the Software, in whole or in part.
- distribute modified versions of the Software, in whole or in part, in source or object format. Specifically you may not distribute derivative works based on the SciTech MGL.
- rent or lease the Software.
- use the Software in the development of an operating system, online service or their associated drivers & utilities, or in a graphics library.
- sell any portion of the Software on its own, without integrating it into your Applications as Executable Code.

OBJECT CODE ONLY RELEASES

Object Code only releases include all official installation archives provided by SciTech Software that do not include any of the source code to the SciTech MGL libraries. The MGL Source Code release will always be in a file named MGLSxx.EXE where xx is the version number for that release of the SciTech MGL source code, so the Object Code only releases include all official install archives other than this. This includes the base install archive (MGLBxx.EXE), font and data install archive (MGLFxx.EXE), the MGL object

code archives for the different supported compilers and any other new archives that we add in the future.

SOURCE CODE RELEASE

The Source Code release is the official installation archive provided by SciTech Software that contains all of the source code in the SciTech MGL libraries. The MGL Source Code release will always be in a file named MGLSxx.EXE where xx is the version number for that release of the SciTech MGL source code. You are prohibited from distributing this archive to others via the internet, shareware distribution services or by any other means. SciTech Software reserves the right as the sole distributor of the Source Code release and it can only be downloaded directly from SciTech Software's web and ftp sites.

SELECTION AND USE

You assume full responsibility for the selection of the Software to achieve your intended results and for the installation, use and results obtained from the Software.

LIMITED WARRANTY

THIS SOFTWARE IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PRODUCT IS WITH YOU. SHOULD THE PRODUCT PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING OR ERROR CORRECTION.

SCITECH SOFTWARE DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR FREE.

No oral or written information given by SciTech Software, its agents or employees shall create a warranty.

LIMITATION OF REMEDIES AND LIABILITY.

IN NO EVENT SHALL SCITECH SOFTWARE, OR ANY OTHER PARTY WHO MAY HAVE DISTRIBUTED THE SOFTWARE AS PERMITTED ABOVE, BE LIABLE FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR FAILURE OF THE SOFTWARE TO OPERATE WITH ANY OTHER PRODUCTS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The cumulative liability of SciTech Software to you for all claims relating to the Software, in contract, tort, or otherwise, shall not exceed the total amount of license fees paid to SciTech Software for the relevant Software. The foregoing limitation of liability and exclusion of certain damages shall apply regardless of the success or effectiveness of other remedies.

GENERAL

Notices. All notices or other communications required to be given shall be in writing and delivered either personally or by U.S. mail, certified, return receipt requested, postage prepaid, and addressed as provided in the Agreement or as otherwise requested by the receiving party. Notices delivered personally shall be effective upon delivery and notices delivered by mail shall be effective upon their receipt by the party to whom they are addressed.

Severability. Should any term of this License Agreement be declared void or unenforceable by any court of competent jurisdiction, such declaration shall have no effect on the remaining terms hereof.

Governing Law. This Agreement shall be governed by and construed and enforced in accordance with the laws of the State of California, USA as it applies to a contract made and performed in such state.

No Waiver. The failure of either party to enforce any rights granted hereunder or to take action against the other party in the event of any breach hereunder shall not be deemed a waiver by that party as to subsequent enforcement of rights or subsequent actions in the event of future breaches.

Costs of Litigation. If any action is brought by either party to this License Agreement against the other party regarding the subject matter hereof, the prevailing party shall be entitled to recover, in addition to any other relief granted, reasonable attorney fees and expenses of litigation.

If you have any questions regarding this agreement, please contact SciTech Software at (530) 894-8400.

Contents

This document provides an overview and introduction to the SciTech MGL Graphics Library and associated supplemental libraries. For detailed reference information please consult the MGL Reference Guide.

SciTech MGL Feature Summary

This section provides a summary of the SciTech MGL features, to give you an overview and what the SciTech MGL provides and the features you can expect to use in your products.

General Features

- High performance 32-bit assembler code for maximum speed
- Resolutions from 320x200 to 1600x1200
- Color depths from 4-bits to 32-bits per pixel
- Support for DOS and Windows
- Supports CreateDIBSection under Windows 95 and Windows NT 3.5
- Supports WinDirect full screen under Windows 95
- Supports Microsoft DirectX under Windows 95 and Windows NT 4.0
- Full hardware and software double/multi buffering support
- Rendering direct to video memory
- Rendering direct to offscreen video memory
- Rendering to system memory buffers
- Full linear surface virtualization under DOS and Windows
- Direct surface access to bypass SciTech MGL if desired
- C++ wrapper class API
- Loading of Windows bitmaps, fonts, cursors and icons
- Loading of PCX bitmap files
- High performance 2D and 3D rendering capabilities
- OpenGL® support in windowed and full-screen modes. - *New in 4.0*
- Refresh rate control - *New in 4.0*
- Stereo display support for LC shutter glasses - *New in 4.0*
- Game Framework library for easy game programming - *New in 4.0*
- Sprite Manager library for simplified sprite management - *New in 4.0*
- Native libraries for the DEC Alpha running Windows NT - *New in 4.0*
- Support for hardware triple-buffering - *New in 4.0*
- Support for double buffered-mouse cursors - *New in 4.0*
- Support for Borland Delphi 2.0 and 3.0 - *New in 4.0*

Event Handling Support

- Unified event queue handling mechanism for DOS and Windows
- Supports keyboard events KEYDOWN, KEYUP and KEYREPEAT
- Supports mouse movement, mouse down and mouse up events
- Supports user specified timer events
- Supports user events posted to queue
- Same event functions for DOS, windowed and fullscreen modes
- Supports user supplied Window procedure under Windows

Graphics Output

- Lines
- Rectangles
- Ellipses
- Elliptical arcs
- Text
- Monochrome bitmaps
- Complex regions (including union, difference, intersection etc.)
- BitBlt
- TransBlt (source and destination transparency)
- StretchBlt
- Full 3D rendering functionality via OpenGL®

What is the SciTech MGL Graphics Library?

The SciTech MGL Graphics Library is a full featured 32-bit graphics library for high performance graphics programming in DOS and Windows environments. It provides fast, low level rasterization of 2D and 3D primitives, that can be used for computer games, user interface software and other real-time graphics applications. SciTech MGL fully supports all graphics resolutions from 320x200 right up to 1600x1200, with any pixel depth from 4 bits per pixel to 32-bits per pixel.

Support for Windowed Applications

SciTech MGL can be used for rendering in a windowed environment under Windows 3.*x*, Windows 95 and Windows NT. Under Windows 3.x SciTech MGL will use the WinG library for high performance output. When running under Windows 95 or Windows NT 3.5 or later, SciTech MGL will use CreateDIBSection or DirectDraw with support for all color depths from 8-

bits to 32-bits per pixel.

Support for Full-Screen Applications

SciTech MGL for Windows can also support full screen graphics under Windows using SciTech's WinDirect technology (Windows 3.x and Windows 95), or Microsoft's DirectX technology (Windows 95 and Windows NT). SciTech's WinDirect technology provides high performance, full screen graphics in any supported graphics mode independent of the current GDI graphics mode, and allows the SciTech MGL to work with any existing VBE 2.0 or higher or VBE/AF 1.0 or higher compliant graphics card (or with our SciTech Display Doctor drivers installed).

Support for OpenGL®

SciTech MGL also includes complete support for OpenGL®, the most advanced and widely used API for 3D graphics on PC's and workstations The SciTech MGL provides an open, seamless, easy-to-use interface to the OpenGL API for ultra-fast 3D rendering in software and hardware. The SciTech MGL support for OpenGL is complete and supports both windowed modes and fullscreen graphics modes for high performance games and entertainment titles. SciTech MGL fully supports Microsoft OpenGL, Silicon Graphics OpenGL for Windows and the freeware MESA OpenGL clone when rendering in software. If there is a hardware device with OpenGL drivers installed, SciTech MGL can use either Microsoft OpenGL or Silicon Graphics OpenGL for Windows implementations for maximum performance using the installed graphics hardware.

Support for the Game Framework

SciTech also includes the Game Framework, a library of functions designed to free game programmers from much of the tedium associated with Windows programming and interfacing your game code with the SciTech MGL. The Game Framework contains a complete set of simple, game oriented wrapper functions for the SciTech MGL that take care of the important tasks that all game developers require. These include features such as automatically handling both windowed and fullscreen graphics modes, enumerating the available fullscreen modes, taking over the system static palette entries cleanly in a window, handling alt-tabbing back to the desktop and focus issues, using a single window to simplify DirectSound integration and many other features required of today's advanced Windows

based games.

The Game Framework also allows you to get started with OpenGL 3D rendering in your game with only a single function call to the Game Framework to turn it on!

Of course the Game Framework, like the SciTech MGL is completely portable between DOS and Windows, so the same code you write for DOS can be re-compiled easily for Windows. Best of all the complete source code to the Game Framework is provided so you can see exactly how your game is interacting with the SciTech MGL libraries.

Support for Simple, Fast Hardware Sprites

SciTech also includes the Sprite Manager, a library of functions designed to provide a seamless, intuitively easy to use library for storing sprites and other bitmaps in offscreen video memory on the graphics adapter. Storing bitmaps in offscreen video memory allows the SciTech MGL to move them around on the screen extremely quickly using the hardware built into the graphics adapter. Support is included for both rectangular and linear layout of sprites in offscreen video memory, and the Sprite Manager includes powerful memory management routines to find the best locations to store your sprites.

Of course the sprite manager is intelligent enough to know how to handle the limited resources of offscreen video memory, and will automatically store any overflow or remaining bitmaps that don't fit into offscreen video memory in system memory buffers. Support is included for pre-compiling the system memory sprites into Run Length Encoded (RLE) bitmaps for higher performance drawing to the display memory, so the Sprite Manager provides you with the best of hardware accelerated sprites along with fast, software sprites when there is no hardware memory available.

Device Independent, Directly Accessible Drawing Surfaces

SciTech MGL is device independent, and provides routines for creating device contexts for rendering directly to video memory, and device contexts for rendering directly to system memory. All the high performance 32-bit code in SciTech MGL is written for flat linear framebuffer access and under DOS and Windows full screen modes, SciTech MGL will provide a virtual flat linear surface for normal VBE 1.2 and 2.0 (VBE 2.0 required for

Windows) devices if there is no hardware linear framebuffer support.

SciTech MGL provides a whole host of high performance rendering routines including fast line drawing, BitBlt operations with source transparency and stretching and fast 2D and 3D rendering functions. However if the routines in SciTech MGL do not suit your purposes, SciTech MGL also provides direct access to the display surfaces, so you can render directly to the surface with your own application specific code, including rendering directly to video memory via the hardware or virtualized flat linear framebuffer.

In cases where the virtual linear framebuffer is not available (for instance under Windows and OS/2 DOS sessions for DOS programs), SciTech MGL provides high speed banked framebuffer devices as a fallback measure.

Hardware Acceleration Support

SciTech MGL is also designed extensively to support full hardware acceleration, and will plug in directly with the VBE/AF Graphics Accelerator driver architecture under DOS and WinDirect under Windows, as well as hardware accelerated OpenGL drivers for Windows. SciTech MGL also fully supports the hardware acceleration features exposed by the Microsoft DirectX libraries.

With full hardware acceleration support, SciTech MGL supports hardware accelerated BitBlt and TransBlt (source and destination transparency) operations between offscreen memory surfaces and display memory surfaces, for incredibly fast sprite animation. Of course accelerated line drawing, rectangle filling and polygon filling are also be provided and complete hardware acceleration of the OpenGL 3D functions is available via standard OpenGL accelerator drivers.

Contact SciTech Software for more information on licensing accelerated device driver support for SciTech MGL.

Portable to Other Operating Systems and Architectures

SciTech MGL source code is also fully portable to other operating systems and architectures such as embedded systems development. The portable version of SciTech MGL is a C only version of the library that can be compiled and linked with any standard C compiler. A native version of the complete SciTech MGL is available for the DEC Alpha running Windows

NT.

For more information on the portable version of SciTech MGL for embedded systems development or other operating systems, contact SciTech Software directly for pricing and availability.

Supplemental Libraries

The SciTech MGL ships with a number of supplemental libraries, including the discontinued SuperVGA Kit, the WinDirect libraries for fullscreen SuperVGA graphics under Windows, the PM/Pro library for providing a DOS extender independent API for protected mode services, and the Zen Timer for providing high precision timing under all these environments.

Some of these libraries such as the PM/Pro library and WinDirect library are used directly by SciTech MGL for portability between the DOS and Windows operating systems. For maximum portability you should stick to the standard MGL API, but these supplemental libraries are provided since they may well be useful for developing code that is specific to the DOS and Windows operating systems.

Also included are a set C++ utility libraries for the SciTech MGL. Libraries are provided for performing 2D and 3D vector math and transformations, realtime 2D rendering including arbitrary rotations, scales and shears, realtime 3D rendering including arbitrary parallel and perspective viewing with wireframe, flat shaded and smooth polygons, a 3D modeling system for easy scene management, a C++ abstract data type class library and a C++ GUI framework.

All of these C++ libraries are provided with the SciTech MGL, and the full source code to all the supplemental libraries is available in the SciTech MGL Plus Pack. Most of the libraries are provided on an “as is” basis and are not supported directly by SciTech Software. These libraries were developed in house by SciTech Software and are being made available in the hope that they may help our customers to take full advantage of SciTech MGL’s capabilities.

All libraries are provided pre-built for all supported compilers. The following is a brief description of each of the C++ supplemental libraries:

SciTech MegaVision GUI library

SciTech MegaVision is a C++ based GUI toolkit for SciTech MGL and is the GUI library that was used to build the SciTech MGL demo programs. MegaVision is a fully object oriented user interface library and provides support for moveable, resizeable windows, pull down menus, file browsing, radio buttons, check boxes and message boxes. Full source available in the SciTech MGL Plus Pack.

SciTech Fixed/Floating Point transform library

This is a C and C++ library that provides high performance fixed point or floating point math functions for both DOS and Windows. This library has been designed to provide the ability to write a single set of source code that can be compiled to use either 16.16 fixed point numbers or full floating point numbers.

High performance functions are provided for common math functions like trig functions, 2D, 3D and 4D vector math functions and 2D and 3D transformation matrices. All code has been hand tuned for maximum speed on Pentium processors and this library provides the mathematical foundation for the Quick2D, Quick3D and QuickModeler libraries. Full source available in the SciTech MGL Plus Pack.

SciTech Quick2D Rendering Library

The Quick2D library is a C++ rendering library that provides a fast fixed / floating point two dimensional world coordinate system on top of SciTech MGL. It provides full support for arbitrary 2D transformations such as translates, rotates, scales and shears. It provides 2D versions of the MGL primitives such as pixels, lines, ellipses, polygons and even vector font text output (fully transformed). It relies upon the fixed point library for fast vector and matrix math functions. Full source available in the SciTech MGL Plus Pack.

SciTech Quick3D Rendering Library

The Quick3D library is a C++ rendering library that provides a fast fixed / floating point three dimensional world coordinate system on top of SciTech MGL. It provides support for arbitrary 3D transformations such as translates, rotates, scales and 3D viewing transformations. It provides 3D primitives such as pixels, lines, ellipses, polygons and even vector font text output (fully transformed). It relies upon the fixed point library for fast

vector and matrix math functions.

This library is written entirely in C++ and hence is not as fast as it could be. In fact there are many parts of this library that can be sped up, but it was developed as an experimental 3D library to prototype many of the concepts that will hopefully become available as a new, high performance 3D library running on top of SciTech MGL. This is the library that is used by the MGL 3D demonstration programs. Full source available in the SciTech MGL Plus Pack.

SciTech QuickModeler 3D Modeling Library

The QuickModeler 3D modeling library is a hierarchy of C++ objects that can be rendered directly using SciTech MGL. It provides support for building complete modeling hierarchies and provides support for single polygons and polygonal models. There is much that can be done to this library, and it is intended as a guide to show how you can develop a high performance modeling system on top of MGL and Quick3D. Full source available in the SciTech MGL Plus Pack.

SciTech Techniques Class Library

The Techniques Class Library is high performance C++ class library for neatly implementing various data structures in C++. It uses the C++ template facility to provide type-safe generic data structures such a arrays, stacks, queues, linked lists, hash tables etc. This library is used by all SciTech Software's C++ products as the low level data structure class library. Full source available in the SciTech MGL Plus Pack.

Hardware and Software Requirements

SciTech MGL requires the following minimum system requirements for programs that you will be developing:

- IBM PC compatible
- An 80386 or higher processor
- VGA or SuperVGA display adapter
- VBE 1.2, 2.0 or VBE/AF 1.0 compliance for SuperVGA support
- MS-DOS 3.3 or later (MGL for DOS)
- Windows 95 or Windows NT (MGL for Windows)
- Microsoft DirectX 3a or later (optional)

Installing SciTech MGL

Before you install any SciTech Software developer products, you should decide upon a standard root directory for installing all of the products into. By default the installation programs will choose the C:\SCITECH directory as the installation location. You might like to install the files onto a different drive, but should install all the files for all the different SciTech Software developer products (MGL for DOS, MGL for Windows, Plus Pack etc.) that you have under the same directory tree. Many SciTech Software products use common libraries and common header files, so when you install them into the same directory you will only have one copy of each of these common files and won't run into conflicts with multiple copies of the same files on your system.

Note: If you have installed a previous release of SciTech MGL then you should uninstall the previous release files or install the new release files into a different directory. Please don't install the new release over the top of an existing release, as there may be conflicts with changes in the directory structures for the product.

Setting up Your Compiler Configuration

Once you have installed the files you want from the distribution CD, you

will need inform your compiler where the include files and library files are located. The following steps provide a guide to setting things up correctly for your compiler.

The installation program installs all include files into the \INCLUDE directory under the installation directory where you chose to install the product (so if you installed the product in c:\scitech, the INCLUDE directory is c:\scitech\include). If you are compiling your applications from the Integrated Development Environment (IDE) for your compiler, you will need to set the include directories for your project file's to include the c:\scitech\INCLUDE directory.

If you are compiling from the command line, you simply need to add the c:\scitech\INCLUDE directory to your INCLUDE path environment variable (for Borland C++ users you will need to add this directory to your Borland C++ 'turboc.cfg' and 'bcc32.cfg' configuration files located in the x:\bc\BIN directory where 'x:\bc' is where you installed the compiler; for DJGPP 2.01 users you will need to add this to the DJGPP.ENV file located in your x:\djgpp directory where x:\djgpp is where you install your compiler).

The installation program installs all the library files for the compilers that you select under the \LIB directory under the installation directory where you chose to install the product (so if you installed the product in c:\scitech, the LIB directory is c:\scitech\lib). Beneath this directory is a hierarchy of directories containing library files for different operating systems and different compilers as shown in the table below(some may not be present depending on what libraries you selected at installation time and what platforms are supported by your compiler).

32-bit DOS protected mode support:	
DOS32\BC4	Borland C++ 4.52 32-bit DOS libraries
DOS32\BC5	Borland C++ 5.02 32-bit DOS libraries
DOS32\WC10	Watcom C++ 10.6 32-bit DOS libraries
DOS32\WC11	Watcom C++ 11.0 32-bit DOS libraries
DOS32\DJ2	DJGPP 2.01 32-bit DOS libraries

32-bit Windows support:	
WIN32\BC4	Borland C++ 4.52 Win32 libraries
WIN32\BC5	Borland C++ 5.02 Win32 libraries
WIN32\SC7	Symantec C++ 7.5 Win32 libraries
WIN32\VC4	Microsoft Visual C++ 4.2 Win32 libraries
WIN32\VC5	Microsoft Visual C++ 5.0 Win32 libraries
WIN32\WC10	Watcom C++ 10.6 Win32 libraries
WIN32\WC11	Watcom C++ 11.0 Win32 libraries
WIN32\IC35	IBM VisualAge for C++ 3.5 Win32 libraries
WIN32\DELPHI2	Borland Delphi 2.0 Win32 libraries
WIN32\DELPHI3	Borland Delphi 3.0 Win32 libraries
NT-AXP\VC4	Microsoft Visual C++ 4.1 DEC Alpha Win32 libraries
NT-AXP\VC5	Microsoft Visual C++ 5.0 DEC Alpha Win32 libraries

Note that the compiler versions listed are those that were used to compile the library files that you will find in those directories. In most cases the libraries should work fine for previous versions of the compiler for the standard C libraries (for C++ libraries such as the Techniques Class Library and MGL Plus Pack libraries you may need to recompile them with your compiler).

If you are compiling your applications from the IDE for your compiler, you will need to set the library directories for your project file to include the c:\scitech\LIB\xxx\xx directory (select the appropriate directory from Table 1 above). If you are compiling from the command line, you simply need to add the c:\scitech\LIB\xxx\xxx path to your LIB path environment variable (for Borland C++ users you will need to add this directory to your Borland C++ 'tlink.cfg' and 'tlink32.cfg' configuration files; for DJGPP 2.01 users you will need to add this to the DJGPP.ENV file located in your x:\djgpp directory where x:\djgpp is where you install your compiler).

Note: For Watcom C++ Users, by default Watcom C++ compiles all source code using register based parameter passing, so by default all SciTech Software libraries are compiled with register based parameter passing. If you are compiling and linking you code for stack based parameter passing, you will need to link with a different set of libraries. All libraries are provided with both stack and register

based versions for Watcom C++ and the default libraries use register based parameter passing. The stack based libraries will have the same name as the register based versions of the libraries, but will have an extra ‘s’ added to the front of the library name. Hence the SciTech MGL/Lite library for stack based parameters is called SMGLLT.LIB rather than MGLLT.LIB.

Once you have done this, you can simply start using the library files as provided. If you intend to re-compile any of the sample programs using the supplied makefiles from the command line, you will need to follow the additional steps outlined below.

Using the Optional Makefile Utilities

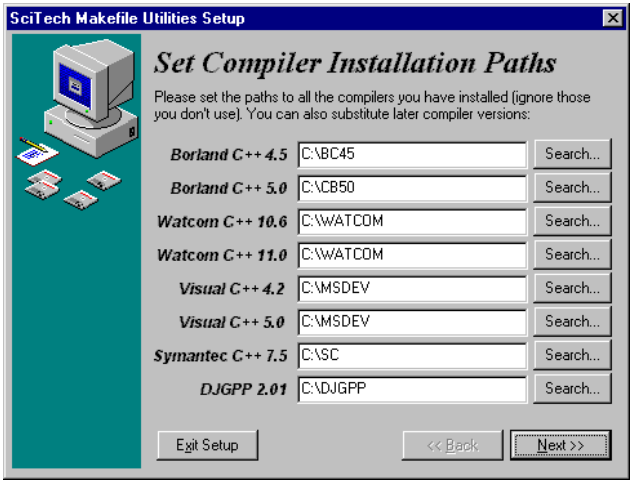
In order to be able to re-compile any of the sample programs using the supplied makefiles from the command line, or re-build any of the libraries that come with source code, you will also need to install the SciTech Software Makefile Utilities package (re-run the installation program and install this if you have not done so already). This installs all of the relevant executable utility files (including the freeware make program called DMAKE), batch files and DMAKE startup scripts required to re-compile the examples for any of the supported compilers.

The makefile utilities package was developed by SciTech Software to allow us to build all of our code from the command line for any of the supported compilers and operating systems using a common set of makefiles. This is achieved by using a standard make program that supports powerful make startup scripts which are changed to reflect the currently selected compiler. To complete the process we also provide a number of utility programs and batch files that can be used to fully automate this process.

The installation program also installs the MKSETUP.EXE configuration program that can be used to re-configure the makefile utilities using a convenient ‘wizard’ style interface. This configuration program allows you to set up the locations to all your compilers and select which compiler you want to be your default compiler. Once you have done this the configuration program will edit all the configuration files for you, so that you can get up and running quickly. If you move your compilers or change any of this information, you can simply re-run the configuration program (located conveniently on the Start Menu in the folder you install the SciTech MGL program folder icons into). The following explains the setup and

configuration steps for the Makefile Utilities setup program:

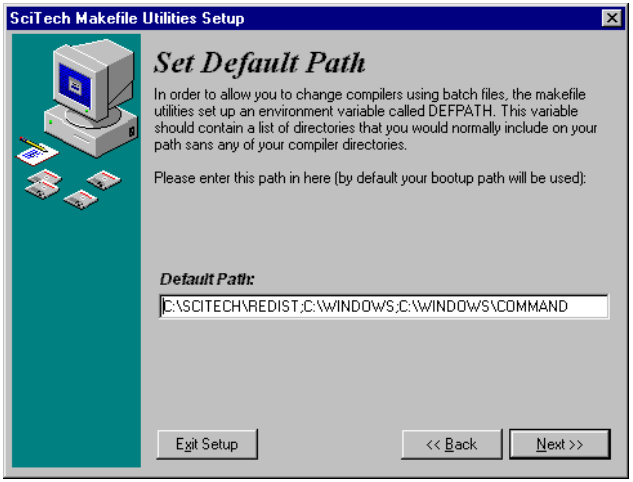
1. **Run the Makefile Utilities Configuration Program (located on the Start Menu under the program folder you installed the SciTech MGL into):**



   This dialog box contains locations to the root directories for the compilers supported by the Makefile Utilities. The initial values will be suggested values based on the defaults directories that the compilers normally install into, but you should modify all the entries for the compiler that you will be using (you can safely ignore any entries for compilers you don't use).

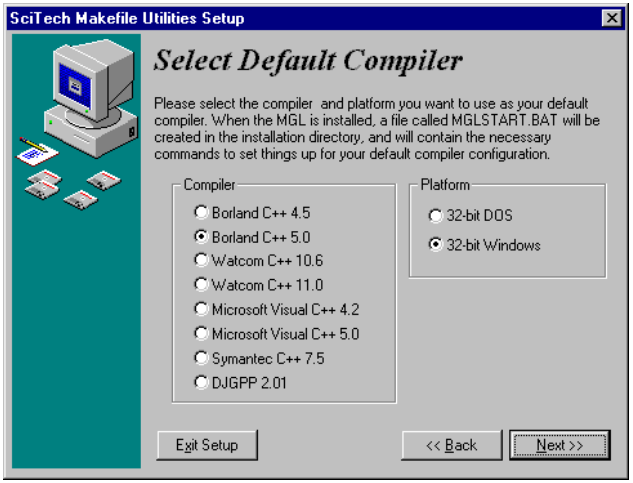
   Click on the 'Search' button to browse for the directory on your hard drive.

2. **Click 'Next' to bring up the 'Set Default Path' screen:**



The path listed will be the current path used by Windows, and you should modify this path to include all the directories you want on the path minus any compiler specific paths. This variable is used by the SciTech Makefile Utilities batch files to construct the complete path for the selected compiler by appending the above path to the end of the compiler specific paths.

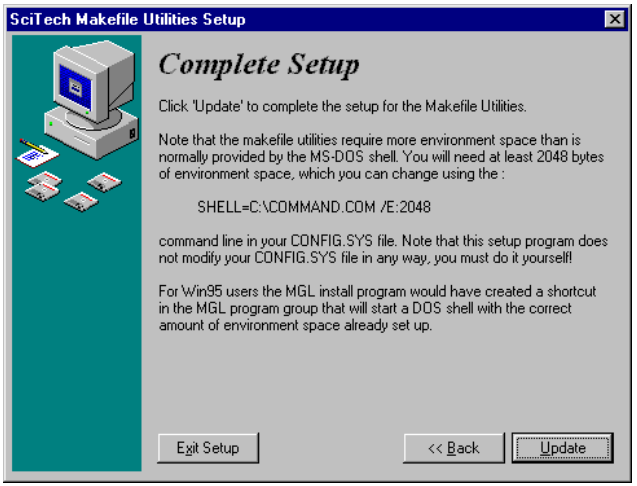
3. **Click 'Next' to bring up the 'Select Default Compiler' screen:**



In here you should select the compiler that you want to be selected as the default compiler when you start an 'MGL Compilation Shell' command prompt. You can of course change the selected compiler at any time using the SciTech Makefile Utilities batch files, but this provides a mechanism to automatically enable the compiler you use the

most as the default.

4. **Click 'Next' to bring up the 'Complete Setup' screen:**



You have now completed the setup process. You can click 'Back' to go back and review all the setup information, and the clock 'Update' to complete the updating of the configuration files.

Once you have complete the setup process, you can simply run the C:\SCITECH\STARTMGL.BAT batch file and you are ready to begin compiling your programs with your default compiler. For Windows 95 and Windows NT 4.0 the install program will have added a shortcut to your Start Menu (called the 'MGL Compilation Shell') which will execute the STARTMGL.BAT batch file with the appropriate settings.

Note: For the Makefile Utilities to work properly, you must have at least 2048 bytes of environment space available. To change this with a normal DOS configuration, add the /E:2048 command line switch to the end of your SHELL= command line. If you are using the latest 4DOS from JP Software (highly recommended) then you can do this in the 4DOS startup files. For Windows 95 and Windows NT 4.0 users, you can simply use the provided 'MGL Compilation Shell' shortcut installed in your Start Menu that is pre-configured with the appropriate command line switches.

Manually Configuring the Makefile Utilities

Once you have installed the makefile utilities onto your hard drive, you may

want to manually edit the configuration files. This section contains detailed information about how to set up your compiler configuration by manually editing all the configuration files.

Change the default executable path in your AUTOEXEC.BAT file to include the c:\scitech\BIN directory (where 'c:\scitech' is where you installed the SciTech MGL files). This can be placed anywhere on your path, so long as the DMAKE.EXE file in the BIN directory will be found first (if there is another program with the same name).

Edit the C:\scitech\BIN\SET-VARS.BAT batch file to set environment variable SCITECH to point to the c:\scitech directory. The SCITECH environment variable is used by the batch files in the BIN directory for setting up for compiling with a particular compiler, and by the DMAKE program so that it can find all of the relevant files during compilation (such as include files).

Edit the C:\scitech\BIN\SET-VARS.BAT batch file to set the environment variable SCITECH_LIB to point to the c:\scitech directory. The SCITECH_LIB environment variable is used by the batch files in the BIN directory to determine where to find all library files, and where to install new files when they are built. By default this is usually the same as the SCITECH variable, but you can make this point to a different drive if your libraries are not located in the normal location. This is extremely useful if you are re-building libraries over a network, and want all the final builds of the libraries to be located on a single machine on the network.

Edit the C:\scitech\BIN\SET-VARS.BAT batch file to set up the environment variables needed by the remainder of the utility batch files. This file is an example that we use for DOS, so you can start with this to build your own configuration file. This should be the only batch file in the BIN directory that should need to be modified as all the remaining batch files feed off the environment variables set up by this master batch file. This batch file essentially lets the rest of the system know where you have installed all of your compiler specific files such as your include and library files.

So an example SET-VARS.BAT configuration might be as follows:

```
@echo off
SET SCITECH=c:\scitech
SET SCITECH_LIB=c:\scitech

SET BC3_PATH=C:\C\BC3
SET BC4_PATH=C:\C\BC45
SET BC5_PATH=C:\C\BC50
SET VC_PATH=C:\C\MSVC
SET VC4_PATH=C:\C\VC42
SET VC5_PATH=C:\C\VC50
SET SC70_PATH=C:\C\SC75
SET WC10_PATH=C:\C\WC10
SET WC11_PATH=C:\C\WC11
SET TNT_PATH=C:\C\TNT
SET DJ_PATH=C:\C\DJGPP
```

You should call this batch file from your AUTOEXEC.BAT file if you will use it all the time, or run this batch file before you use any of the remaining utilities.

Using the Makefile Utilities

To get up and running with your default compiler configuration, you can simply run the c:\scitech\STARTMGL.BAT batch file, or you can add the commands from this batch file to your AUTOEXEC.BAT startup file. Once you have done that, you can switch between different compilers on the command line simply by running the batch file that is appropriate for the compiler you want to use:

32-bit DOS protected mode support:	
bc45-d32.bat	Borland C++ 4.52 32-bit DPMI32
bc50-d32.bat	Borland C++ 5.0 32-bit DPMI32
dj20-d32.bat	DJGPP 2.01 32-bit
wc10-d32.bat	Watcom C++ 10.6 32-bit DOS4GW
wc11-d32.bat	Watcom C++ 11.0 32-bit DOS4GW

32-bit Windows support:	
bc45-w32.bat	Borland C++ 4.52 32-bit
bc50-w32.bat	Borland C++ 5.0 32-bit
vc40-w32.bat	Microsoft Visual C++ 4.2 32-bit
vc50-w32.bat	Microsoft Visual C++ 5.0 32-bit
sc70-w32.bat	Symantec C++ 7.5 32-bit
wc10-w32.bat	Watcom C++ 10.6 32-bit
wc11-w32.bat	Watcom C++ 11.0 32-bit

Note: When using Borland C++ 4.x/5.x with these batch files, you will need to edit the supplied TURBOC.*, TLINK.*, BCC32.* and TLINK32.* to contain the proper information for your installation. These files are then copied by the batch files into the proper Borland C++ installation directories to correctly set up the compiler for compiling and linking code for the specified target environment. By default the Makefile Utilities Configuration program will automatically edit these files for you, but if you need to link with different libraries that are not supplied by Borland or SciTech, you will need to edit these files to reflect their location (or simply put them in the SCITECH\LIB directories or your compiler's LIB directories).

Once you have everything set up, you should be able to run DMAKE from the directory containing the sample programs that you wish to compile. If things run smoothly you should get a resulting executable file that you can run.

Standard Makefile Targets

All of the makefile utilities DMAKE startup scripts support a standard set of targets for controlling the compilation for the current makefile. The most common commands and useful targets that you may want to use when building examples and re-compiling any libraries are listed in the table below:

dmake	Running dmake by itself in a directory will select the default target for the makefile, which is usually to compile and link all sample programs in that directory. Some makefiles only support building libraries so the default target may produce an error.
dmake -u	The -u command line option forces a complete re-build of all files, so it is useful to re-build an entire directory from scratch.
dmake lib	This re-builds the library for the directory.
dmake install	This re-builds the library for the directory and installs the final library into the appropriate c:\scitech\LIB\xxx\xx directory. You should only do this once you are *sure* that everything is working correctly! The old library will simply be overwritten by the new library.
Dmake clean	This cleans out all object files, libraries, pre-compiled header files etc. from the directory, but leaves all executable files and DLL's.
dmake cleanexe	This cleans out all non-source files including all DLL's and EXE files.

Standard Makefile Options

All of the makefile utilities DMAKE startup scripts support a standard set of options for controlling the way that the compilation is performed. Makefile options are provided for turning on debug information, speed or size optimizations and inline floating point instructions. By default when you build files, no optimizations and no debugging information is generated. The following table lists the most common and useful of these options for building examples and re-compiling any libraries.

DBG	Turns on debugging information
OPT	Turns on speed optimizations
OPT_SIZE	Turns on size optimizations
FPU	Turns on inline floating point arithmetic
STKCALL	Turns on stack calling conventions for Watcom C++
USE_PMODEW	Use PMODE/W replacement DOS extender for Watcom C++
USE_CAUSEWAY	Use CauseWay replacement DOS extender for Watcom C++

Options can be passed to DMAKE in one of two ways: on the command line or as global environment variables. For instance the following are equivalent:

```
dmake DBG=1 OPT=1 install
```

or

```
set DBG=1
set OPT=1
dmake install
```

Assembling 32-bit code

All of SciTech Software's assembler code is written in Borland TASM IDEAL mode, so you will need a copy of Borland TASM in order to re-assemble the assembler code. If you are assembling for 32-bit protected mode, you *MUST* use TASM 4.0 or later, since TASM 3.1 and earlier do not generate correct 32-bit code in some instances. If you don't have a copy of Borland TASM but you wish to re-build the C code portions of the libraries, you can recompile and link with just the module you need, or you can use your compiler's librarian utility to extract the pre-assembled modules from the libraries that you wish to build (see your compiler's documentation for more information).

Changing the default DOS extender

All of the SciTech Software DOS libraries are DOS extender independent. All DOS extender dependent information is encapsulated in the PMODE.LIB library files. The default library provided for each of the

compilers is compiled for the default DOS extender normally used by that compiler. All you need to do in order to use a different DOS extender is re-compile the PM/Pro library with the appropriate command line options, and then link with this new library.

SciTech Standard Directory Tree

All SciTech Software products install into a common directory structure so that all header files and library files are all stored in common locations. This makes it very easy to find particular library files and include files that you need, but it also means that once you have set yourself up for compiling and linking with one SciTech Software product, installing and using another is simple because everything will already be set up.

The following is a brief outline of the common directory tree structure.

```
SCITECH             ROOT directory (SCITECH is default directory name)
³
ÃÄÄÄBIN             Useful tools and batch files for library building
³
ÃÄÄÄDOC             Documentation files. Each individual product has
³   ÃÄÄÄMGL         its own separate directory which contains all files
³   ÃÄÄÄ...         specific to that product.
³   ÀÄÄÄSVGAKIT
³
ÃÄÄÄREDIST          All reditributable DLL and EXE components
³
ÃÄÄÄRDST-AXP        DEC ALpha reditributable DLL and EXE components
³
ÃÄÄÄFONTS           All MGL font files installed here.
³
ÃÄÄÄBITMAPS         All MGL bitmap files installed here.
³
ÃÄÄÄCURSORS         All MGL mouse cursor files installed here.
³
ÃÄÄÄICONS           All MGL icon files installed here.
³
ÃÄÄÄEXAMPLES        Any example source code. Separate sub-directories
³   ÃÄÄÄMGL         are provided for each of the different products.
³   ÃÄÄÄ...
³   ÀÄÄÄOPENGL
³
ÃÄÄÄINCLUDE         All installed header files.
³
ÃÄÄÄLIB             All installed library files.
³   ÃÄÄÄDOS32       All 32 bit DOS libraries
³   ³   ÃÄÄBC4          Borland C++ 4.52
³   ³   ÃÄÄBC5          Borland C++ 5.02
³   ³   ÃÄÄWC10         Watcom C++ 10.6
³   ³   ÃÄÄWC11         Watcom C++ 11.0
³   ³   ÀÄÄDJ2          DJGPP 2.01
³   ÃÄÄÄWIN32       All 32 bit Windows libraries
³   ³   ÃÄÄDELPHI2     Borland Delphi 2.0
³   ³   ÃÄÄDELPHI3     Borland Delphi 3.0
³   ³   ÃÄÄBC4         Borland C++ 4.52
³   ³   ÃÄÄBC5         Borland C++ 5.02
³   ³   ÃÄÄSC7         Symantec C++ 7.5
³   ³   ÃÄÄVC4         Visual C++ 4.2
³   ³   ÃÄÄVC5         Visual C++ 5.0
³   ³   ÃÄÄWC10        Watcom C++ 10.6
³   ³   ÃÄÄWC11        Watcom C++ 11.0
³   ³   ÀÄÄIC35        IBM VisualAge for C++ 3.5
³   ÀÄÄÄNT-AXP      All 32 bit Windows libraries for DEC Alpha
³       ÃÄÄVC4         Visual C++ 4.1
³       ÀÄÄVC5         Watcom C++ 4.0
³
ÀÄÄÄSRC             All source code for building the libraries (if
    ÃÄÄÄMGL         provided). Each package has a separate directory
    ÃÄÄÄ...         that contains all of the source code and makefiles
    ÀÄÄÄGM          for building the library and installing it under
                    \SCITECH\LIB.
```

Letting SciTech MGL Know Where to Find Shared Resources

SciTech MGL provides a number of functions for loading shared resource files such as Windows bitmaps, fonts, cursor and icons. In order for SciTech MGL to be able to find these files, a standard search algorithm is always used (see `MGL_loadBitmap` in the MGL Reference Guide for more

information). The last resort used is to look for all files relative to the MGL_ROOT environment variable, which is normally set to point to the directory where you installed SciTech MGL. Under the MGL root directory will be a set of BITMAPS, FONTS, CURSORS and ICONS that contain the resource files that all the MGL sample programs use. When you installed SciTech MGL the installation program would have modified the STARTMGL.BAT batch file in the directory that you install SciTech MGL into to set up this environment variable, and you may want to move the commands from this batch file to your AUTOEXEC.BAT file.

If you don't set this environment variable and you compile any of the MGL sample programs and run them directly from the installation directories, the samples will not be able to find the files they need and will exit with an error message. In this case simply set the MGL_ROOT variable manually to point to the directory where you installed SciTech MGL.

When you distribute your applications built with SciTech MGL, you can simply store all shared resource files in subdirectories below where your application program is located and you won't need to set this environment variable.

Getting Started with SciTech MGL

This section provides an overview of the SciTech MGL graphics library, including information on how to get started compiling and linking your code with the SciTech MGL libraries.

Differences Between MGL/Lite and Regular Libraries

SciTech MGL is provided with two different sets of libraries. The primary reason for this is to cut down on the amount of code that needs to be linked into an application if you are not using all of SciTech MGL's functionality. For instance, if you are developing a game and have all your own custom rendering code, you can link with the MGL/Lite library to only link with the absolute minimum functionality that you require.

Essentially the MGL/Lite library contains only the basic initialization and setup code, pixel plotting, line drawing, basic blitting and device clearing code. Below is a summary of the functionality that is included and excluded from the MGL/Lite libraries.

Functions included in MGL/Lite
All initialization and setup functions (MGL_init, MGL_createDisplayDC)
MGL_setColor family
MGL_pixel family
MGL_line family (single pixel solid lines only)
MGL_clearDevice, MGL_clearViewport
MGL_bitBlt from system memory to screen and back (no write modes)
MGL_stretchBlt from system memory to screen and back (no write modes)
Mouse cursor support
Viewport and clip rectangle support

Although the SciTech MGL API functions themselves are only linked if you call those functions, the low level device driver rendering code is all linked in whenever you register a specific device driver for use. Since a large percentage of the high performance code in SciTech MGL is located in the device drivers, linking with the full SciTech MGL libraries can bring in a lot

of extra baggage if you are not all the rendering functions.

Functions *NOT* included in MGL/Lite
Scanline filling
Rectangle filling
Text output
Markers
Wide pen support
Bitmap pattern support
Polyline drawing
Polygon filling
Scanline color scanning
Border drawing
Ellipse drawing
Stretch Blting
Divot support
Onscreen BitBlt's
Transparent Blting
Offscreen hardware Blting
Monochrome bitmap support
Complex region support
Icon loading / drawing support
Bitmap loading / drawing support
PCX file loading / drawing support
Dithering support
Bitmap translation or palette translation support

Compiling and Linking with SciTech MGL

To compile standard C code to use SciTech MGL, you only need to include the MGRAPH.H header file in your source files. This file contains all the information required to compile MGL code, and you simply link your application with the appropriate MGLxx.LIB library supplied for your compiler. These libraries are listed in the table below individually for both DOS and Windows. If you are compiling C++ code you can either stick to

the C based MGL API, or you can also use the optional C++ wrapper functions for SciTech MGL (used by all of the Plus Pack C++ libraries). To use the C++ wrapper API simply include the MGRAPH.HPP header file and link with both the MGLxx.LIB and MGLCPP.LIB library files for your compiler.

SciTech MGL consists of a number of header files and static link libraries, and the following sections describes the files and their purpose.

Standard C Header Files

Header File	Purpose
MGRAPH.H	Main MGL header file
MGLDOS.H	MGL DOS specific header file
ZTIMER.H	Zen Timer library header file
GL\GL.H	Main OpenGL header file
GL\GLU.H	Main OpenGL Utility Library header file
GL\GLUT.H	GLUT OpenGL sample program framework header
GL\GLUTDLG.RH	Resource header for GLUT sample programs
GL\GLUTDLG.RC	Resource file for GLUT sample programs
GM\GM.H	Game Framework header file
GM\SPRITE.H	Sprite Manager header file

Note: You do not need to directly include the MGLDOS.H or MGLWIN.H header files. These header files are automatically included when you build your applications for either DOS or Windows (the header files automatically determine the target environment).

Standard C++ Header Files

Header File	Purpose
MGRAPH.HPP	Main MGL C++ header file
MGLPOINT.HPP	MGL C++ header file for MGLPoint class
MGLRECT.HPP	MGL C++ header file for MGLRect class
GM\SPRBMP.HPP	Sprite Manager C++ header file

DOS Runtime Libraries

SciTech MGL for DOS consists of a number of static link libraries, and the following files comprise the full MGL library package (see Installation for details on where the files will be located).

Library File	Purpose
MGLLT.LIB	MGL/Lite static link library
MGLFX.LIB	MGL static link library
MGLDB.LIB	MGL debug static library for Borland C++
MGLCPP.LIB	MGL C++ binding static link library
MESAGL.LIB	Mesa OpenGL implementation for DOS
MESAGLU.LIB	Mesa OpenGL utility library for DOS
GLUT.LIB	GLUT OpenGL sample program library
GM.LIB	Game Framework library
ZTIMER.LIB	Zen Timer Library

Note: Watcom C++ users are also provided with a complete set of stack calling convention libraries with the same names as the above libraries, but with a pre-pended ‘s’ at the start (i.e.: SMGLFX.LIB is the main MGL library for stack calling conventions).

Special DOS Debugging Libraries for Borland C++

For Borland C++ users we have also supplied special debugging libraries to get around certain problems with the Borland debuggers.

If you are compiling and linking code for the Borland C++ DOS IDE and wish to debug the code from within the IDE, you cannot link your code with the standard MGLFX.LIB library. SciTech MGL installs its own keyboard interrupt handling routines, and the IDE debugger does not correctly save and restore these vectors while it debugs you code, so the IDE will hang once you hit your first breakpoint and attempt to step into any code. To get around this problem link your code with the MGLDB.LIB library in the DOS16\BC3 directory. This version does not install a keyboard interrupt handler. However your application may not receive keyboard and mouse events in the order that the user issued them, and you won’t get separate

KEYDOWN, KEYREPEAT and KEYUP events (you only get KEYDOWN's).

If you are compiling and linking code for the Borland C++ 4.52 32-bit DOS PowerPack, and you wish to debug your code with the TD32 debugger, you cannot link with the standard MGLFX.LIB library. For the same reason outlined above, the TD32 debugger does not correctly save and restore DPMI interrupt handlers and hence as soon as SciTech MGL is initialized, the debugger's keyboard handler is locked out and you cannot debug your code! The MGLDB.LIB library provided in the DOS32\BC directory does not include a keyboard interrupt handler and has the same caveats as the BC++ 3.1 version. Also note that TD32 does not correctly handle user installed mouse handlers, and hence while debugging MGL code the TD32 mouse handler will be locked out and you will not be able to use the mouse from within the debugger.

Windows Runtime Libraries

SciTech MGL for Windows can also be compiled to use the DLL version of the SciTech MGL libraries. By default when you compile the code it selects the static link libraries, so to select the DLL version you must `#define MGL_DLL` in your IDE's project options or on the compiler command line. Then to use the DLL you need to link with the appropriate MGLxxI.LIB import library for you compiler. When you have compiled for the DLL version, your application will require the appropriate MGLxx.DLL library file to be present on the path when it is run (or in the same directory as your application).

SciTech MGL for Windows consists of a number of static link libraries, and optional 32-bit DLL's for Windows applications. The following files comprise the full MGL library package (see installation section for details on where the files will be located).

Library File	Purpose
MGLLT.LIB	MGL/Lite static link library
MGLLTI.LIB	MGL/Lite DLL import library
MGLFX.LIB	MGL static link library
MGLFXI.LIB	MGL DLL import library
MGLCPP.LIB	MGL C++ binding static link library
MGLLTWI.LIB	Watcom C++ MGL/Lite DLL import library
MGLFXWI.LIB	Watcom C++ MGL DLL import library
GLUT.LIB	GLUT OpenGL sample program library
GM.LIB	Game Framework library
ZTIMER.LIB	Zen Timer Library

SciTech MGL also requires a number of runtime DLL's which are all located in the c:\scitech\REDIST directory where you installed the SciTech MGL. All the DLL's are listed below, however you only need to ship the necessary DLL's for your runtime environment and selected compiler (see the Appendix A : Shipping your MGL for more information on the DLL's you need to ship with your product).

Runtime DLL	Purpose
MGLLT.DLL	MGL/Lite Win32 DLL
MGLFX.DLL	MGL Win32 DLL
MGLLTW.DLL	Watcom C++ specific MGL/Lite Win32 DLL
MGLFXW.DLL	Watcom C++ specific MGL Win32 DLL
MGLGM.DLL	MGL Game Framework DLL for Delphi
MGLGLUT.DLL	MGL OpenGL GLUT DLL for Delphi
ZTIMER.DLL	Zen Timer Library DLL for Delphi
SGIGL.DLL	Silicon Graphics OpenGL for Windows
SGIGLU.DLL	Silicon Graphics OpenGL for Windows
OPENGL95.DLL	Microsoft Windows 95 OpenGL
GLU95.DLL	Microsoft Windows 95 OpenGL
MESAGL.DLL	Mesa OpenGL clone
WDIR16.DLL	SciTech WinDirect 16-bit side DLL (required!)
WDIR32.DLL	SciTech WinDirect 32-bit side DLL

Note: The OPENGL95.DLL and GLU95.DLL libraries must be renamed to OPENGL32.DLL and GLU32.DLL respectively, and then installed into the Windows 95 system directory. The library files are Windows 95 specific and *must not* be installed on Windows NT!

Special Win32 DLL's for Watcom C++ users

The standard MGL Win32 DLL's use the *__cdecl* calling conventions for all public functions, as this is a common standard supported by all compilers. However by default Watcom C++ always uses register based parameter passing and although you can compile and link with the standard MGLxx.DLL's, the calling conventions used will be different than those used by the standard MGLxx.LIB static link libraries.

If however you plan to compile and link your code with the C++ wrapper API or with the SciTech MGL for Windows Plus Pack libraries, these libraries are all compiled to use the faster register based calling conventions. You might also want to use it because it will provide for faster code that is comparable to the static link library version. For this reason we have also provided a special DLL for Watcom C++ that is compiled to use register

based calling conventions for all public functions. To use this DLL simply need to link you code with appropriate MGLxxWI.LIB import library rather than the MGLxx.LIB static link library. You don't need to #define MGL_DLL as this is only required if you are linking with the MGLxx.DLL library.

When you have compiled for the Watcom C++ specific DLL, your application will require the appropriate MGLxxW.DLL library file to be present on the path when it is run (or in the same directory as the application).

Note: This DLL can only be called from Watcom C++ code and cannot be used with any other compiler.

Where to Next? The MGL Sample Programs

This section gives a brief overview of the MGL sample programs to give you an idea of the best place to start learning about SciTech MGL. All programs except for the WMGLDOG and WSHOWBMP demos can be compiled and run under both fullscreen DOS and fullscreen Windows (fullscreen Windows support requires WinDirect or Microsoft DirectX to be installed; WinDirect is provided as part of the SciTech MGL). Note also that none of the C++ sample programs have been ported to Borland Delphi, only the C sample programs.

Note that all MGL sample programs are provided with Integrated Development Environment (IDE) project files for all supported compilers. All of the SciTech MGL example projects are included in a single IDE. IDE files exist for each of the compilers supported by the SciTech MGL, and there will be a directory for each supported compiler within the c:\scitech\EXAMPLES\MGL directory. For instance the IDE files for Borland C++ 4.5 will be in the BC4-IDE directory, those for Borland C++ 5.0 will be in the BC5-IDE directory and those for Visual C++ 4.2 will be in the VC4-IDE directory etc.

MGL Sample Programs

All the MGL sample programs are located in the c:\scitech\EXAMPLES\SAMPLES directory, and most of them use the MGLSAMP.C utility module to initialize and set up SciTech MGL. Hence the sample code itself only contains the functions illustrated by that sample

program to keep it clear what is code related to that function and what is general SciTech MGL maintenance code.

BITBLT.EXE

c:\scitech\EXAMPLES\MGL\SAMPLES\BITBLT.C

This simple sample program shows how to use the MGL_bitBlt functions to move blocks of data around on the screen.

BITMAP.EXE

c:\scitech\EXAMPLES\MGL\SAMPLES\BITMAP.C

This simple sample program shows how to use SciTech MGL to load a bitmap file from disk and display it on the screen.

DIRECT.EXE

c:\scitech\EXAMPLES\MGL\SAMPLES\DIRECT.C

This is a very simple sample program that shows how to directly access the display surface for a device context, and shows how you can integrate your own custom rendering code with SciTech MGL rendering code. It shows drawing lines directly to video memory and directly to a system memory buffer using identical code. Can be compiled and linked with the MGL/Lite libraries.

ELLIPSES.EXE

c:\scitech\EXAMPLES\MGL\SAMPLES\ELLIPSES.C

This simple sample program shows how to use SciTech MGL ellipse functions to draw ellipses and elliptical arcs on the screen.

HELLO.EXE

c:\scitech\EXAMPLES\MGL\SAMPLES\HELLO.C

This is a simple hello world style program that shows how to get up and running with your first MGL program. It simply uses the 640x480x4 standard VGA 16 color mode and draws a bunch of lines. This would be the

first program you should try to get up and running on your system.

LINES.EXE

c:\scitech\EXAMPLES\MGL\SAMPLES\BITBLT.C

This simple sample program shows how to use the MGL line drawing functions to draw lines on the screen in interesting patterns.

MGLDEMO.EXE

c:\scitech\EXAMPLES\MGL\SAMPLES\MGLDEMO

This is a full featured demo program that exercises nearly every part of the SciTech MGL API. By default this demo runs in the 640x480x4 standard VGA mode, but you can select the graphics mode to use from the command line (run MGLDEMO -h for a list of graphics modes). This is a good example showing you how to select any graphics mode at runtime, and how to handle color issues for a single executable that needs to run in both 4/8-bit palette modes and 15/16/24/32-bit RGB modes.

The demo is quite long but it contains code showing how to do just about everything with the SciTech MGL API, from palette fades to floodfills to high speed polygon filling.

MOUSE.EXE

c:\scitech\EXAMPLES\MGL\SAMPLES\MOUSE.C

This simple sample program shows how to use SciTech MGL to display a mouse cursor in fullscreen graphics modes, and how to change the shape of the mouse cursor from your code.

MOUSEDB.EXE

c:\scitech\EXAMPLES\MGL\SAMPLES\MOUSEDB.C

This simple sample program shows how to use SciTech MGL to display flicker free mouse cursors while doing double buffered animation. Two different methods are outline for two different types of animation that can be used.

PAGEFLIP.EXE

c:\scitech\EXAMPLES\MGL\SAMPLES\PAGEFLIP.C

This simple sample program shows how to use the MGL page flipping functions to implement double and multi-buffered, smooth animation. Includes support for triple buffering as well.

PALETTE.EXE

c:\scitech\EXAMPLES\MGL\SAMPLES\PALETTE.C

This simple sample program shows how to use the MGL palette manipulation functions for changing the color palette in 8-bit graphics modes, as well as doing palette fades and rotates.

PCX.EXE

c:\scitech\EXAMPLES\MGL\SAMPLES\PCX.C

This simple sample program shows how to use SciTech MGL for loading and displaying a PCX bitmap file from disk.

PIXELFMT.EXE

c:\scitech\EXAMPLES\MGL\SAMPLES\PIXELFMT.C

This simple sample program shows how to use SciTech MGL to load a bitmap with a different pixel format to the screen, and use SciTech MGL to Blt it to the screen with automatic color depth translation. This sample also shows how to create your own memory buffers with specific pixel formats for 15-bit and above memory buffers.

POLYS.EXE

c:\scitech\EXAMPLES\MGL\SAMPLES\POLYS.C

This simple sample program shows how to use SciTech MGL to draw polygons on the screen.

RECTS.EXE

c:\scitech\EXAMPLES\MGL\SAMPLES\RECTS.C

This simple sample program shows how to use SciTech MGL to display both filled and outlined rectangles on the screen with different pen widths.

REGIONS.EXE

c:\scitech\EXAMPLES\MGL\SAMPLES\REGIONS.C

This simple sample program shows how to use the MGL region manipulation functions to create regions, display them on the screen and do math operations such as unions, intersections and differences using SciTech MGL.

SPRITES.EXE

c:\scitech\EXAMPLES\MGL\SAMPLES\SPRITES.C

This simple sample program shows how to use SciTech MGL for displaying sprites on the screen, and include code to automatically use an offscreen memory device context for storing the sprites in offscreen video memory and using the hardware blitter to move the sprites around on the screen.

STEREO.EXE

c:\scitech\EXAMPLES\MGL\SAMPLES\STEREO.C

This simple sample program shows how to use the MGL stereo LC shutter glasses support to load and display stereo bitmaps, and enable and display stereo viewing with SciTech MGL. You will require a pair of supported stereo LC shutter glasses to be able to view the images generated by this sample program!

STRETCH.EXE

c:\scitech\EXAMPLES\MGL\SAMPLES\STRETCH.C

This simple sample program shows how to use SciTech MGL to stretch bitmaps from a system memory buffer to the screen. SciTech MGL supports both fast shrinking and expanding of bitmaps, and this sample program demonstrates both types of stretching.

TEXTDEMO.EXE

c:\scitech\EXAMPLES\MGL\SAMPLES\TEXTDEMO.C

This simple sample program shows how to display text on the screen using SciTech MGL, and shows both bitmap fonts and vector fonts.

VIEWPORT.EXE

c:\scitech\EXAMPLES\MGL\SAMPLES\VIEWPORT.C

This simple sample program shows how to use the MGL viewport functions to change the local coordinate system for all output and move it to a different location on the screen.

Game Framework Sample Programs

The Game Framework sample programs include simple sample programs that show how to get started quickly, along with full fledged sprite demos and OpenGL demonstration programs. Most but not all are located in the c:\scitech\EXAMPLES\MGL\GM directory.

BOUNCE.EXE

c:\scitech\EXAMPLES\GM\BOUNCE.C

This is a very simple Game Framework sample program that shows how to use the Game Framework and show a bouncing ball animation around on the screen. It uses all of the Game Framework features such as automatically switching between windowed and fullscreen modes, and supporting all available color depths.

FOXBEAR.EXE

c:\scitech\EXAMPLES\MGL\FOXBEAR

This is complete sprite based C++ animation demo, based on ATI Technologies Fox & Bear benchmark program. It fully supports both high performance RLE based software transparent sprites combined with hardware accelerated transparent sprites via the SciTech Game Framework and the SciTech Sprite Manager libraries. This sample programs makes extensive use of the SciTech Game Framework and the SciTech Sprite Manager, so it is a gold mine of information on using these libraries for sprite based animation.

Unlike the original ATI demo, this version was essentially rebuilt as a set of real C++ classes and provides full support for scaling the original 640x480 8-bit bitmap data to any graphics mode resolution ranging from 320x200x8 up to 1600x1200x32! With hardware accelerated rendering this demo can reach speeds of 72fps at 1024x768x8 with double buffering (on a 486DX266 with a 4Mb ATI Mach64). With software only rendering it can reach speeds of 30-40fps at 640x480x8 on a fast Pentium90 machine with VBE 2.0 PCI graphics card.

Also included in this demo is optional sound support using the DiamondWare Sound ToolKit for DOS and Windows, showing how you can include sound support in your SciTech MGL applications. For more information on these libraries, check out the shareware versions supplied on the \COOLSTUFF\DWSTK directory of the installation CD-ROM.

GEARS.EXE

c:\scitech\EXAMPLES\GM\GEARS.C

This is a very simple Game Framework sample program that shows how to use the Game Framework and OpenGL to draw 3D scenes on the screen in real time. It uses all of the Game Framework features such as automatically switching between windowed and fullscreen modes, and supporting all available color depths with OpenGL.

For more complete sample programs specific to OpenGL, see the c:\scitech\EXAMPLES\OPENGL directory which contain many more OpenGL sample programs, some of which use the Game Framework and others which use the GLUT Utility libraries.

SKYFLY.EXE

c:\scitech\EXAMPLES\OPENGL\SKYFLY

This is a neat OpenGL sample program developed originally by Silicon Graphics and ported to SciTech MGL by SciTech Software. This sample program shows how to use the Game Framework with OpenGL, as well as showing a simple terrain renderer in action based on OpenGL.

This sample program also shows how to set up and display stereo 3D scenes using SciTech MGL and OpenGL (run with -stereo from the command line if you have LC shutter glasses!).

General MGL Demo Programs

The following MGL demo programs show off different aspects of SciTech MGL, and as such as not specifically good sample programs to learn how to get started with SciTech MGL. However most do contain lots of interesting things that can be done with SciTech MGL, so when you are more experienced with SciTech MGL you might want to come back and investigate what these sample programs do.

DEMO.EXE

c:\scitech\EXAMPLES\MGL\DEMO

This sample program shows off most of the capabilities of SciTech MGL using the MegaVision C++ GUI interface class library. It is essentially a GUI version that contains much the same functionality as the MGLDEMO sample program, but allows the user to switch graphics modes on the fly using the Graphics mode dialog box.

DEMO3D.EXE

c:\scitech\EXAMPLES\MGL\DEMO3D

This sample program shows off the 3D capabilities of SciTech MGL using the Quick3D and QuickModeler libraries. This application itself is actually very simple, as the Quick3D and QuickModeler libraries take care of most of the details related to performing the 3D rendering. It is a good reference to determine how to use the MegaVision file browser dialog box and how to perform hardware double buffered animation in a MegaVision GUI window.

MGLDOG.EXE

c:\scitech\EXAMPLES\MGL\MGLDOG

This sample program is a simple program showing how you can use the SciTech MGL for displaying transparent bitmaps for sprite and animation effects. It is loosely based on the WinG DOGGIE sample program, but fully supports all color depths and resolutions.

This sample is a good place to get information on how to load bitmaps, convert them to the color format of the display, allowing for maximum

performance when blitting the bitmaps to the screen.

PLAY.EXE

c:\scitech\EXAMPLES\MGL\SMACKER

This is a fullscreen Smacker video player for both DOS and Windows. Smacker is a cool package for compressing animation and sound files into 8-bits per pixel videos, and is used in many commercial games for video playback. This demo shows how you can integrate SciTech MGL and Smacker together for high performance video playback in any of the resolutions supported by SciTech MGL. For more information on Smacker check out the Smacker Toolkit in the \COOLSTUFF\SMACKER directory on your installation CD-ROM.

SHOWBMP.EXE

c:\scitech\EXAMPLES\MGL\SHOWBMP

This is a fullscreen version of the WSHOWBMP demo, and is based on the SHOWBMP.EXE demo in the WinG SDK. It shows how to load a bitmap or PCX file directly into a device context surface in fullscreen graphics modes, and will automatically do on the fly color conversion of bitmaps to the selected graphics mode.

WMGLDOG.EXE

c:\scitech\EXAMPLES\MGL\WMGLDOG

This sample program is the same as the MGLDOG program above, but displays all output in a window on the desktop rather than fullscreen. It is good sample program to start with if you want to learn how to use the SciTech MGL for drawing to a window on the desktop.

WSHOWBMP.EXE

c:\scitech\EXAMPLES\MGL\WSHOWBMP

This is a windowed version of the SHOWBMP.EXE demo, and is based on the SHOWBMP.EXE demo in the WinG SDK. It shows how to load a bitmap directly into a system memory device context surface and display that in a Window. It automatically does on the fly color conversion of bitmaps to the

current GDI display mode and will handle HiColor and TrueColor graphics modes properly under Windows 95 and Windows NT.

This demo also contains code showing how to create and use the SYS_PAL static mode to properly create a palette that will allow your applications to display 254 colors out of the total 256 (256 if you can use black and white as 0 and 255).

OpenGL Sample Programs

All of the SciTech MGL OpenGL specific sample programs and located under the c:\scitech\EXAMPLES\OPENGL directory, and may be located in sub-directories specific to each sample program. Most of the sample programs use the free OpenGL Utility Library (GLUT) ported to the SciTech MGL, which allows you to download most OpenGL sample code of the internet based on GLUT and simple compile and link it with SciTech MGL.

Note that we have only listed a small selection of the available OpenGL sample programs, so please check the directories as we have included as many sample programs as we could lay out hands on for your enjoyment.

ATLANTIS.EXE

c:\scitech\EXAMPLES\OPENGL\ATLANTIS

This is a simple OpenGL sample program based on GLUT that shows a bunch of fish swimming around in the ocean.

GEARS.EXE

c:\scitech\EXAMPLES\OPENGL\DEMOS\GEARS.C

This is a simple OpenGL sample program based on GLUT that shows some gears rotating around on the screen. This is the same code that the Game Framework gears program is based on, and shows the difference between using the Game Framework and the GLUT libraries for OpenGL development.

GEARS2.EXE

c:\scitech\EXAMPLES\OPENGL\DEMOS\GEARS2.C

This is a simple OpenGL sample program based on GLUT that shows a different set of gears spinning around on the screen.

MECH.EXE

c:\scitech\EXAMPLES\OPENGL\DEMOS\MECH.C

This is a simple OpenGL sample program based on GLUT that shows a mechanical robot strolling down the street of a futuristic city.

MOTH.EXE

c:\scitech\EXAMPLES\OPENGL\DEMOS\MOTH.C

This is a simple OpenGL sample program based on GLUT that shows a view of a moth spinning around a lamp post in the middle of a room, as the room zooms in from the distance toward the viewer.

RINGS.EXE

c:\scitech\EXAMPLES\OPENGL\DEMOS\RINGS.C

This is a simple OpenGL sample program based on GLUT that shows the classic spinning Olympic rings.

IDEAS.EXE

c:\scitech\EXAMPLES\OPENGL\IDEAS

This is a simple OpenGL sample program based on GLUT that shows the classic Silicon Graphics ‘Ideas in Motion’ animation sequence.

Building Your First Fullscreen MGL Program

To get started with SciTech MGL, we'll build a simple, full-screen application which has minimal logic but which does contain the basic elements of any full-scale SciTech MGL application.

The MGLDOG.EXE Sample Program

To open and browse your first fullscreen MGL program:

1. **Open the Examples IDE for your compiler.**

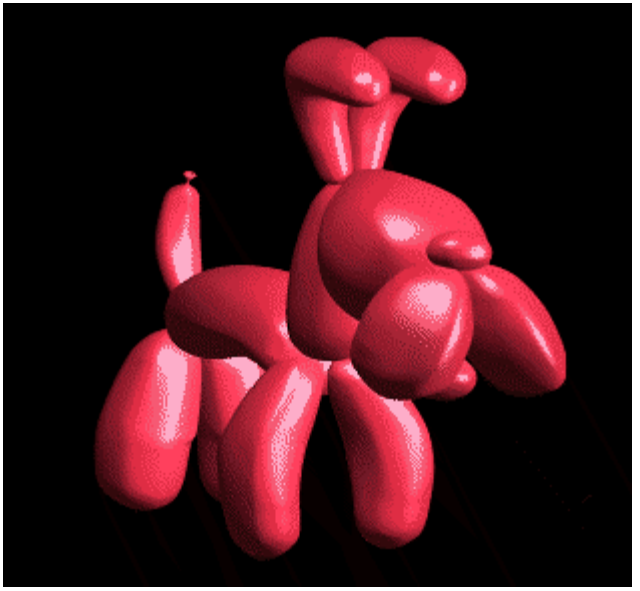
   All of the SciTech MGL example projects are included in a single IDE. IDE files exist for each of the compilers supported by the SciTech MGL.

   For example, if you are using Borland's C++ v5.02, your IDE would be found in this directory:

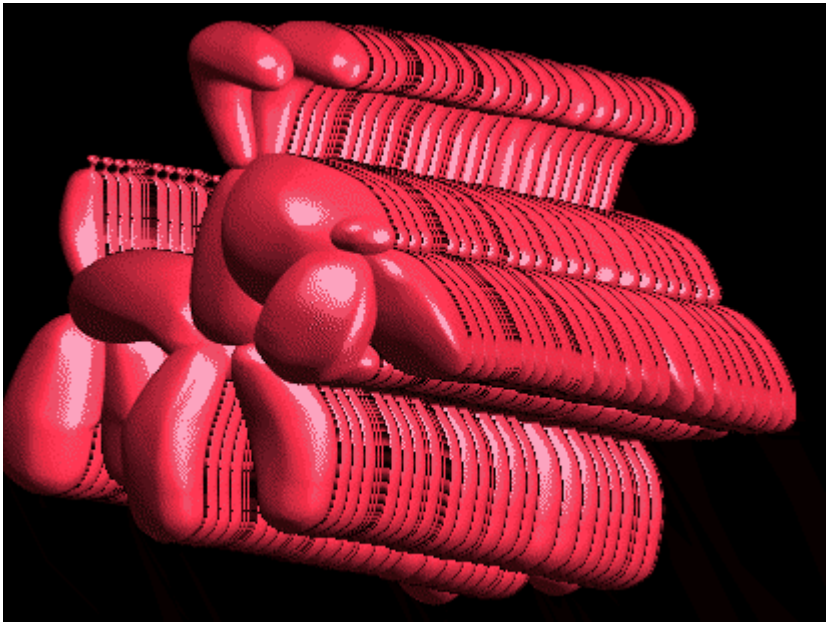
   <root>\scitech\MGL\examples\bc50-ide\

2. **Run the MGLDOG node in the project.**

   The MGLDOG program is quite simple. An image of a red dog appears in the center of the screen:



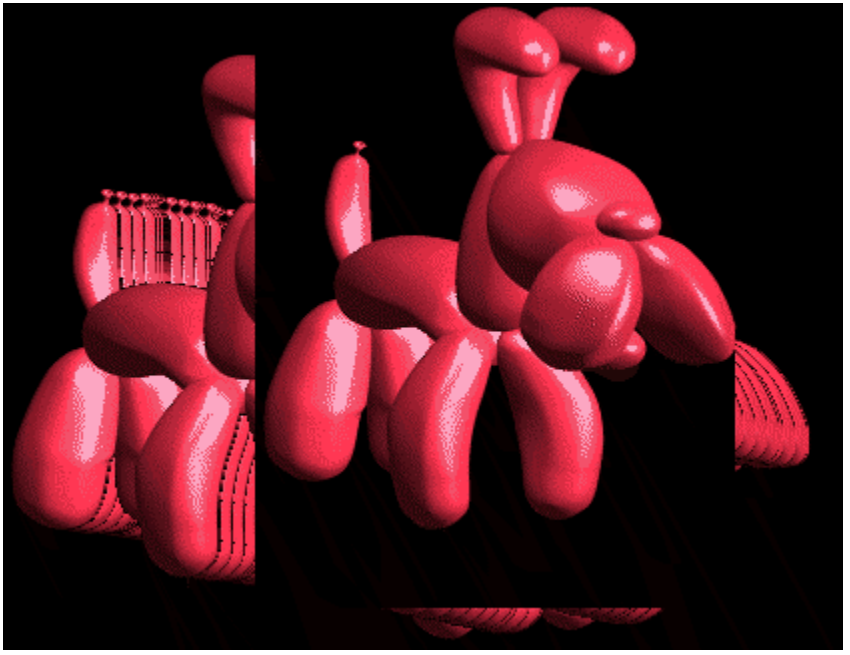
- Hold the left mouse button down to paint the screen with dogs



Note that the dogs are copied with *Source Transparency*. Pixels in the new image which are the same color as pixels in the underlying image are not copied.

- Click the right mouse button to copy an instance of the dog sprite to the

screen:



Note that this time the dog sprite is copied *without* source transparency. The black pixels around the figure of the dog overwrite the pixels of the underlying image.

What is a Fullscreen MGL Program?

Most people are familiar with windowed applications. A windowed application runs in the context of a window, an object provided by the operating system. The operating system provides various services to windows, including the transmission of system *events*. Examples of events include the user pressing a key on the keyboard, moving the mouse, minimizing or maximizing the window. A windowed applications window can be programmed to respond in various ways to these events. For game programmers, events are the key to controlling user interaction with the game.

Windows provide a framework within which a complex application can run, and also provide an interface governing the interaction between the application and the rest of the system.

Windows does not, however, provide the maximum in flexibility in terms of the parameters game developers are often most interested in: fullscreen

graphics. A windowed application can only run in the graphics mode (resolution and color depth) in which the operating system as a whole is running. For example, if a user chooses to keep their desktop at 1024x768 pixels and 16.7 million of color, your game will have to provide graphics output at that resolution. This can mean a serious degradation in performance in order for your code to generate the necessary number of pixels (or if a translation step must be done on the fly every frame for the display data), or a minute display area if code intended for lower resolutions is run at the higher resolution.

Fortunately, you can run your games in a full-screen mode. A full-screen mode occupies the entire display, concealing the desktop. Furthermore, full-screen mode provide complete control over graphics mode, including resolution and color-depth. Full-screen modes are also normal windows, however, in the sense that all the facilities for handling events and manipulating the window are still in place. You can use the same event handling code in a windowed and in a full-screen mode.

Initializing the MGL Fullscreen Environment

The SciTech MGL simplifies the interface to the Windows API, so initializing the full-screen MGL environment is a matter of a simple function call. Browse the source code for MGLDOG.C and move to the WinMain function:

```
int PASCAL WinMain(HINSTANCE hInst,HINSTANCE hPrev,LPSTR
szCmdLine,int sw)
{
    ... Some code for linking with other SciTech libraries is here...
    DialogBox(hInst,MAKEINTRESOURCE(IDD_MAINDLG),NULL,
        (DLGPROC)MainDlgProc);
}
```

Displaying the Initial Dialog Box

The first step in this particular program is to display an initial dialog box allowing the user to select the fullscreen graphics mode to run the demo in. The call above initializes a dialog box, with which the user can select a graphics mode, color depth, and driver type to use when running the demo. Execution now shifts to the process which handles events for the window, `MainDlgProc`:

```
    DialogBox(hInst,MAKEINTRESOURCE(IDD_MAINDLG),NULL,
        (DLGPROC)MainDlgProc);
```

Of course, in a real application these parameters would be determined at

runtime. SciTech MGL and its companion library for game developers, the Game Framework, include routines for handling hardware detection and selecting graphics modes. For now, however, we'll ignore these complications and focus on the methods by which device drivers are registered in SciTech MGL.

Note: We'll be covering device detection in a later chapter, but if you'd like to see how SciTech MGL detects video device drivers, have a look in the `RefreshModeList` routine.

Specifying the Initial Display Mode, and Initializing SciTech MGL

The call to `MGL_init` establishes the initial graphics mode used when the SciTech MGL creates Display Contexts. This mode can be overridden later "on the fly" (see below). You should always pass in `grDETECT` for the driver parameter to allow the SciTech MGL to detect and use the highest performance driver available on your system.

```
MGLDC *initGraphics(void)
{
    MGLDC   *dc;
    /* Start SciTech MGL and create a display device context
*/
    MGL_unregisterAllDrivers();
    MGL_registerAllDispDrivers(useLinear,useDirectDraw,
        useWinDirect);
    MGL_registerAllMemDrivers();
    if (!MGL_init(&driver,&modeNum,"..\\..\\..\\"))
        initFatalError();
    if ((dc = MGL_createDisplayDC(1)) == NULL)
        initFatalError();
    MGL_makeCurrentDC(dc);

    /* Register our suspend application callback */
    MGL_setSuspendAppCallback(doSuspendApp);

    /* Turn off identity palette checking for maximum speed */
    MGL_checkIdentityPalette(false);
    return dc;
}
```

Registering the Device Drivers

Without bothering too much with the specifics of the dialog box implementation of graphics mode and driver selection, move to the `initGraphics` function in MGLDOG:

The first step is to unregister any drivers which may be registered. This clears the way for initializing and registering all the display drivers which are present on the system with a call to `MGL_registerAllDispDrivers`. Next, all known packed pixel memory drivers are detected and linked in with a call to `MGL_registerAllMemDrivers`.

Note: These two functions, `MGL_registerAllDispDrivers` and `MGL_registerAllMemDrivers`, are useful for getting code up and running quickly. In a production environment, however, you'll want to specify the drivers you want registered to reduce the total size of the resulting executable by linking in only those drivers you application will be using. SciTech MGL will only end up linking with the device support code for the specific drivers you specific if you include the code in your application to register them.

Note: If you are running at a lower color-depth, and use `MGL_loadBitmapIntoDC` to load a bitmap with a deeper color-depth, SciTech MGL will translate the color-depth of the source bitmap to the color-depth of the target DC. However, this translation operation requires that the appropriate packed pixel memory drivers for the source bitmap be loaded. So, you must load packed pixel memory drivers for all the color-depths you plan to support, even if you never intend to display images at these color depths (i.e.: if you need to convert 24-bit bitmaps to other pixel formats, make sure you register the PACKED24 memory driver).

Specifying Which Drivers to Support

The parameters to `MGL_registerAllDispDrivers` tell that function whether or not to register all Linear, DirectDraw, and WinDirect drivers. If you choose to do so, you can remove these drivers from the list of drivers to be registered *even if they are installed and available in the system*. To do so, just pass in a value of False for these parameters. At the same time you can force the use of WinDirect or DirectDraw drivers by excluding other possibilities.

Note: You might want to take a look at the `MGL_registerAllDispDriversExt` function which provides more fine grained control over exactly which drivers are registered, and for simplicity the above sample program does not use this function.

Creating a Display Device Context

If the drivers are registered correctly, then the next step is to create an MGL Display Device Context, or MGLDC. The display device context is the section of memory, basically an array of pixels, to which we will render our graphics along with the associated structures for describing that memory. In our example, the MGLDC is created with a call to `MGL_createDisplayDC`:

```
if ((dc = MGL_createDisplayDC(1)) == NULL)
    initFatalError();
```

`MGL_createDisplayDC` returns NULL if there is not enough memory to create the MGLDC. This would effectively end the program, so the call to `MGL_createDisplayDC` is wrapped in the appropriate error trapping code.

Note: `MGL_createDisplayDC` creates a device context for writing directly to the hardware display device in full screen modes. The SciTech MGL uses other functions to create device contexts in windowed applications. See the MGL Library Reference for more information.

The Current Device Context

The *Current Device Context* is the MGLDC used for all the SciTech MGL rendering routines. So, if you want to see the results of your rendering code, make sure you've got the right MGLDC selected as the current DC. You can make an MGLDC the current MGLDC with a call to `MGL_makeCurrentDC`.

Fullscreen Applications and Focus

In Windows, the application with which the user is currently interacting is said to have the *focus*. Gaining or losing focus occurs when the user clicks on another window, or uses a system key combination like ALT-TAB to shift focus to another application. Gaining or losing focus also generates a system event, for which you can code a specific response in your application.

You can include code in a single routine to set up your application for losing and regaining focus with a call to `MGL_setSuspendAppCallback`. Just provide the name of a function you want executed when your application gains or loses the focus, as on line 100.

```
MGL_setSuspendAppCallback(doSuspendApp);
```

In general you'll want to be sure that you reset the system to a reasonable

state in your callback function when it loses the focus, and restore the system to the state required by your application when it regains the focus. In MGLDOG, the callback function restores a clear image when the application regains focus (because it is not possible to save the previous image when the focus is lost, since by the time your application gets the message that the has been lost the display memory has already been changed):

```
{
    if (flags == MGL_REACTIVATE) {
        MGLDC *oldDC = MGL_makeCurrentDC(dc);
        MGL_clearDevice();
        MGL_makeCurrentDC(oldDC);
        }
    return MGL_SUSPEND_APP;
}
```

The flags argument can have one of two values, depending upon whether the application is gaining or losing the focus:

- WD_DEACTIVATE

  Losing focus.

- WD_REACTIVATE

  Gaining focus

Identity Palettes and Performance

The last step in the initialization of the SciTech MGL environment in MGLDOG is related strictly to performance. `MGL_checkIdentityPalette` toggles identity palette checking, in our example, it is toggled off:

```
MGL_checkIdentityPalette(false);
```

When identity palette checking is off, no effort is made to translate colors from a source bitmap to a destination bitmap during BitBlt operations. You must handle palette translation yourself. In this case, we're only working with a single bitmap so we don't really have to worry about weird colors popping up due to different, untranslated palettes. And things go much more quickly without palette checking on because the SciTech MGL does not have to check for an identity palette prior to every Blt operation.

Following this execution returns to the `doGraphics` routine.

Now that we've created our display device context, we need to create another device context to hold the "doggie" image. We'll transfer this device context onto the display device context in different positions when the user moves the mouse.

The implementation is relatively simple:

1. **Create another device context and load a bitmap file into it in a single operation.**

   This is accomplished with a call to `MGL_loadBitmapIntoDC`, in `doGraphics`:

   ```
   memDC = loadBitmapIntoMemDC(dc,"doggie2.bmp");
   ```

   This system memory MGLDC now contains the doggie image.

2. **Set the transparent color for subsequent Blting operations.**

   In this case, we choose the color of the lowest left pixel of the doggie bitmap as our transparent color, first by making the MGLDC with the doggie image the current device context, then by calling `MGL_getPixelCoord` to extract the color of this pixel:

   ```
   MGL_makeCurrentDC(memDC);
   transparent = MGL_getPixelCoord(0,0);
   MGL_makeCurrentDC(dc);
   ```

3. **Blt the device context with the image to the center of the display device context.**

   ```
   MGL_makeCurrentDC(dc);
   width = MGL_sizex(memDC)+1;
   height = MGL_sizey(memDC)+1;
   MGL_transBltCoord(dc,memDC,0,0,width,height,
       (MGL_sizex(dc) - width)/2,(MGL_sizey(dc) -
        height)/2,transparent,true);
   ```

The actual Blt is done with a call to `MGL_transBltCoord`. The display MGLDC (dc) is the *destination* device context, while the offscreen MGLDC (memDC) containing the doggie image is the *source* device context. The next four arguments are pixel coordinates defining a rectangle within the source image to be Blted (in this case, the entire image), while the next two arguments determine the upper left corner of the area in the destination

image into which the source image will be Blted.

The `transparent` parameter is the color we selected earlier as the transparency color. Pixels of this color will be skipped during the Blting operation.

The final parameter determines whether the Blt operation will use source or destination transparency. In *source transparency*, pixels in the source image which match the transparent color are ignored (not drawn). In *destination transparency*, pixels in the destination image which match the transparent color are ignored (not drawn).

Note the use of the `MGL_sizex` and `MGL_sizey` routines to determine the size of the image and the destination device context.

What the Heck is a Blt?

Blt (pronounced "Blit") is a word derived from the Windows API function, `BitBlt`, which in turn stands for Bit Block Transfer.

Drawing the Mouse Cursor

So, now that we have an image, we need to provide the user with a way to manipulate it. Since we're in a full-screen application, there is no default windows mouse handling. That means we'll have to use SciTech MGL to manage the mouse. That's easy enough, fortunately, with a call to `MS_show`. But first, if we're running in 8-bit (color mapped) mode, we need to explicitly set the color of the cursor by passing in the index into the system palette for the color we want. In this case, white is 255 (0xFF).

```
if (MGL_getBitsPerPixel(dc) == 8)
    MS_setCursorColor(0xFF);
MS_show();
```

You can hide the mouse cursor with a call to `MS_hide`.

Interacting with the User

Interacting with the user means tracking which keys are pressed, where the mouse is moved to, and which mouse buttons are pressed. In the case of MGL dog, we want to draw transparent doggies while the left mouse button is pressed and opaque doggies while the right mouse button is pressed. We also want to continue doing this until the user presses the Escape key

(which has ASCII code 0x1B, or 27). So, we loop until the escape key is pressed:

```
while (!done) {
if (EVT_getNext(&evt,EVT_EVERYEVT)) {
    switch (evt.what) {
        case EVT_KEYDOWN:
            if (EVT_asciiCode(evt.message) == 0x1B)
                done = true;
                break;
        case EVT_MOUSEMOVE:
            x = evt.where_x - width/2;
            y = evt.where_y - height/2;
            MS_obscure();
            if (evt.modifiers & EVT_LEFTBUT) {
                MGL_transBltCoord(dc,memDC,0,0,width,height,x,y,
                    transparent,true);
                }
            else if (evt.modifiers & EVT_RIGHTBUT) {
                MGL_bitBltCoord(dc,memDC,0,0,width,height,x,y,
                    MGL_REPLACE_MODE);
                }
            MS_show();
            break;
            }
        }
    }
MGL_exit();
```

Using the MGL Event Handling Functions

The core of the loop are the MGL event handling functions. MGL provides a *unified event queue*, which means you only have to look in one place to find out about keyboard, mouse, and other system events. To get the most recent event in the event queue, call `EVT_getNext`, passing by reference an object of type `event_t` as well as a flag indicating which events to include (line 20 in the code segment above):

```
EVT_getNext(&evt, EVT_EVERYEVENT)
```

This fills the members of the `evt` object with information about the most recent event on the stack. The type of event is stored in `evt.what`. In MGLDOG, we're interested in four types of events:

- EVT_MOUSEDOWN, EVT_MOUSEUP, EVT_MOUSEMOVE

  A mouse button has been pressed, released, or the mouse has moved.

- EVT_KEYDOWN

  A key has been pressed.

The coordinates at which a mouse event occurred are stored in `evt.where_x` and `evt.where_y`. You can determine which buttons are down by examining the `evt.message` field. There are three possible values for a mouse event:

- EVT_LEFTBUT

  The left button was down.

- EVT_RIGHTBUT

  The right button was down.

- EVT_DBLCLK

  The mouse down event was a double-click.

Check for a button down by using a bitwise AND to compare the value in `evt.modifiers` with the constant value for EVT_LEFTBUT, as in line 120 above:

```
if (evt.modifiers & EVT_LEFTBUT)
```

For an event of type EVT_KEYDOWN, the particulars are stored in `evt.message`. The data stored in `evt.message` are in bits, but you can extract the ASCII code for the key which was pressed by calling `EVT_asciiCode`, as on line 50 in the code segment above:

```
EVT_asciiCode(evt.message)
```

So, in the code segment above we're looping until the user quits by pressing the escape key. Whenever the mouse is moved, we check to see whether or not a button was pressed. If a button was pressed, we Blt the doggie image onto the display DC again, creating the overlapping pattern of doggie images:

```
//  If the mouse is moved

    case EVT_MOUSEMOVE:
// Record the position of the cursor

        x = evt.where_x - width/2;
        y = evt.where_y - height/2;

// Hide the mouse cursor.

        MS_obscure();

// Check to see if the left button was pressed
        if (evt.modifiers & EVT_LEFTBUT) {

// If so, transparently Blt the doggie image onto the display DC

            MGL_transBltCoord(dc,memDC,0,0,width,height,x,y,
                transparent,true);
            }

// Check to see if the right button was pressed.

        else if (evt.modifiers & EVT_RIGHTBUT) {

// If so, Blt the doggie onto the display DC with no transparency

            MGL_bitBltCoord(dc,memDC,0,0,width,height,x,y,
                MGL_REPLACE_MODE);
            }
```

Using Your Own Window Procedure

The SciTech MGL event handling routines simplify Windows event handling, and have the added advantage of allowing your applications to use the same event handling code whether running as windowed or as full screen applications. In addition, DOS and Windows applications can also use the same event handling code so you can re-compile for DOS and Windows without having to change your code. However, you may have a requirement to supply your own windows procedure. For example, in legacy code situations it may be more trouble to recode your windows procedures than to just include them as they are. You can register your window procedure with the SciTech MGL, and then all Windows events will be passed to and handled by that routine rather than by the MGL event handling routines.

To register your own Windows Event procedure, call `MGL_registerEventProc`:

```
MyEventProc(HWND hWnd, UINT message UINT wParam, LONG lParam)
{
    ...your code here...
}

MGL_registerEventProc(myEventProc)
```

Changing Display Modes on the Fly

When you initialize SciTech MGL, you specify a graphics mode during the call to `MGL_init`. This mode is used by `MGL_createDisplayDC` to initialize the desired fullscreen graphics mode. However, to change the graphics mode you don't have to exit MGL and reinitialize. Instead use the following steps:

1. **Call MGL_destroyDC on the existing display DC and all offscreen DCs.**
2. **Call MGL_changeDisplayMode, passing in the new graphics mode.**
3. **Call MGL_createDisplayDC again.**

`MGL_destroyDC` late binds destruction of the graphics mode so that the SciTech MGL remains in full-screen mode until another DC is created. This avoids the expensive process of returning to the GDI desktop, and the resulting flickering of the display. The effect is similar to switching graphics modes in a pure DOS environment.

However if you destroy a full-screen DC and then wish to switch to a windowed mode (such as when the user presses ALT-ENTER), you must call `MGL_changeDisplayMode` and pass in `grWINDOWED` in order to force SciTech MGL to switch back to the GDI desktop so you can created the windowed device context.

Using DirectSound with the SciTech MGL

In order to use DirectSound with your MGL applications, you will need to do the following:

1. **Create a single main window before you initialize the SciTech MGL.**

   The attributes of this window do not matter at this point; it can even be hidden.

2. **Initialize Direct Sound with the window handle.**
3. **Call MGL_registerFullScreenWindow, passing in the handle to your**

main window.

This call informs SciTech MGL to use the window you created rather than one it creates itself for fullscreen display DCs. Be sure to call `MGL_registerFullScreenWindow` *before* any calls to `MGL_createDisplayDC`.

This window can be toggled between windowed and full-screen modes without being destroyed, and DirectSound need only be initialized once. Otherwise, a window created by the MGL internal code is created on the call to `MGL_createDisplayDC` and destroyed on the call to `MGL_destroyDC`, which means that the DirectSound focus is lost. Thus, a change in graphics mode such as a switch to windowed mode or to a different full-screen resolution destroys the window and causes DirectSound to lose focus requiring it to be re-initialized. Due to the problems with DirectSound, it is not possible to properly re-initialize the libraries for a new window handle, which is why we allow you to create a single window for the duration of your MGL program.

Note: If you create your own window and register it with SciTech MGL, *do not* use the `MGL_registerEventProc` and register your window procedure with SciTech MGL or your application will likely crash!

Setting the Task Bar Icon and Program Name

If you create your own window you set the task bar icon when you create the class and the window name when you create the window itself. However, in full screen modes when SciTech MGL creates a window for you the icon used will be the icon with numerical identifier '1' in your resource file. The window name will always be the string with numerical identifier '1' in the stringtable in your resource file.

Destroying SciTech MGL Before Exit

When your application has finished, you must destroy the objects created by the SciTech MGL and deallocate the memory they occupy. Fortunately this is relatively straightforward; just call `MGL_exit`.

Note: `MGL_exit` destroys all existing display and memory device contexts. Thus if your code maintains any pointers to DCs after the call to `MGL_exit`, you must ensure that your code does not reference these pointers after the call to `MGL_exit` (Especially in Windows modes,

where Windows messages may arrive after the destruction of the DC).

Building Your First Windowed MGL Program

In the last example we built a full-screen application using SciTech MGL. Now it's time to try building a standard windows application, which will run like any other application in a window on the desktop.

What is a Windowed MGL Program?

Unlike a full-screen application, a windowed MGL program runs in a window on the desktop. Application windows can be resized, minimized, or maximized to cover the entire desktop. In addition, a windowed application can run in the background when a user shifts focus to a different application. All of these capabilities can have implications for the way in which you must code your windowed applications.

The windowed version of MGLDOG.EXE, called WMGLDOG.EXE, is an excellent example of a windowed MGL application. Because we've already introduced much of the code for the application, we can focus on new concepts strictly related to coding for the windowed environment. Take a moment to load the project in your IDE and browse the code.

Initializing the MGL Windowed Environment

Initialization of the MGL windowed environment consists of these steps:

1. **Registering an instance of WNDCLASS for our applications with Windows**

   The Registration process includes configuring basic parameters for the Window which govern its appearance and behavior.

2. **Initializing the SciTech MGL**
3. **Creating and displaying the main window for our application**
4. **Polling messages from the Windows event queue.**

   Like nearly all windows applications, our application loops through a message processing routine until receipt of the WM_QUIT message.

Creating a Window Manager Window and Initializing SciTech MGL

The following is standard windows code to register, create, and show a window, as well as create a basic event handling loop. Note that immediately after creating the window class SciTech MGL is initialized with a call to `MGL_init`. After SciTech MGL is initialized, the window is displayed which in turn sets of the chain of events that cause SciTech MGL device contexts to be created and something to display on the screen (which appears to be strangely missing from the code below).

```
if (!hPrev) {
    /* Register a class for the main application window */
    WNDCLASS cls;
    cls.hCursor        = LoadCursor(NULL,IDC_ARROW);
    cls.hIcon          = LoadIcon(hInst,"AppIcon");
    cls.lpszMenuName   = "AppMenu";
    cls.lpszClassName  = szAppName;
    cls.hbrBackground  = NULL;
    cls.hInstance      = hInst;
    cls.style          = CS_BYTEALIGNCLIENT | CS_VREDRAW |
                             CS_HREDRAW | CS_DBLCLKS;
    cls.lpfnWndProc    = (LPVOID)AppWndProc;
    cls.cbWndExtra     = 0;
    cls.cbClsExtra     = 0;
    if (!RegisterClass(&cls))
       return FALSE;
    }

/* Create the main window and display it */
hwndApp = CreateWindow(szAppName,szAppName,WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT,0,400,400,NULL,NULL,hInst,NULL);
ShowWindow(hwndApp,sw);

/* Initialise SciTech MGL */
InitMGL();

/* Create the main window and display it */
hwndApp = CreateWindow(szAppName,szAppName,WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT,0,400,400,NULL,NULL,hInst,NULL);
ShowWindow(hwndApp,sw);
```

The `InitMGL` routine includes a call to `MGL_initWindowed`, which initializes the SciTech MGL for operation in windowed only modes.

Registering the Device Drivers

The `InitMGL` routine also takes care of registering device drivers, as well as loading the doggie image into a memory (backbuffer) DC.

You don't register full-screen drivers for windowed modes. However, you must register packed pixel drivers if you want to create a memory DC. So if you want to support say 24 bpp in a memory DC (which lives in system

memory) and then Blt it to the DisplayDC, you must register the appropriate packed pixel drivers to support the creation of the DC. You can choose to support only 8 bpp, in which case you need not register any additional packed pixel drivers other than the packed 8 driver.

```
MGL_registerDriver(MGL_PACKED8NAME,PACKED8_driver);
MGL_registerDriver(MGL_PACKED16NAME,PACKED16_driver);
MGL_registerDriver(MGL_PACKED24NAME,PACKED24_driver);
MGL_registerDriver(MGL_PACKED32NAME,PACKED32_driver);
```

If your display DCs are running at a lower color-depth, and you call `MGL_loadBitmapIntoDC` to load a bitmap with a deeper color-depth, SciTech MGL will translate the color-depth of the source bitmap to the color-depth of the target DC. However, this translation operation requires that the appropriate packed pixel memory drivers for the source bitmap be loaded. So, you must load packed pixel memory drivers for all the color-depths you plan to support, even if you never intend to display images at these color depths.

Creating Device Contexts and Loading the Doggie Sprite

Before we begin to process events we need to create a DC and load the doggie bitmap from file into that DC. The following code creates a windowed DC, then loads the doggie image from disk with a call to `LoadBitmapIntoMemDC`:

```
if ((winDC = MGL_createWindowedDC(GetDesktopWindow())) == NULL)
    MGL_fatalError("Unable to create Windowed DC!");
if ((bits = MGL_getBitsPerPixel(winDC)) < 8)
    MGL_fatalError("This program requires 256 or more colors!");
MGL_getPixelFormat(winDC,&pf);

/* Load the sprite bitmap into the dogDC */
dogDC = LoadBitmapIntoMemDC(winDC,"doggie2.bmp");
```

Note that the code checks that the system supports at least 8-bits per pixel.

Once the dogDC is created with the doggie sprite, we can extract useful information about the image for use later in Blting the image around the display. Such information includes the dimensions of the image, as well as a color to use as the transparency color during Blt operations:

```
MGL_makeCurrentDC(dogDC);
transparent = MGL_getPixelCoord(0,0);
width = MGL_sizex(dogDC)+1;
height = MGL_sizey(dogDC)+1;
MGL_makeCurrentDC(NULL);
```

Creating Windowed Device Contexts

In WMGLDOG, device contexts are created and modified in response to the `WM_SIZE` event, handled in `AppWndProc`. This message is also fired as part of the window creation process, and also allows us to recreate all of the DCs used by the application when the user resizes, minimizes or maximizes the window.

In `AppWndProc`, receipt of a `WM_SIZE` message results in a call to the `CreateMGLDeviceContexts` routine. This argument requires the handle to your application window, so that SciTech MGL will know which window with which to associate any windowed device contexts it creates.

Creation of the windowed DC is handled in a call to `MGL_createWindowedDC`. Note that the window handle is passed in:

```
    if ((winDC = MGL_createWindowedDC(hwnd)) == NULL)
        MGL_fatalError("Unable to create Windowed DC!");
```

Synchronizing Color Depth

After the DC is created, the following call to `GetMemoryBitmapDepth` ensures that the color depth and pixel format of the DC are correct for the current GDI desktop mode:

```
{
    if (!haveTrueColor)
        *bits = 8;
    else {
        *bits = MGL_getBitsPerPixel(dc);
        MGL_getPixelFormat(dc,pf);
        }
}
```

If TrueColor is not supported (for example, is we're running under Windows 3.*x* which only supports 8-bits per pixel memory DC's), then we will want to force 8-bits per pixel. If TrueColor support does exist, we set the pixel format and bit-depth to match that of the windowed DC passed in (i.e.: the color depth and pixel format of the GDI display mode). We'll use this information to ensure that all system memory DCs we create use the same pixel format and color depth as the GDI display mode.

Creating a Memory Device Context

Once the windowed DC is created, we need to create the primary backbuffer DC. In this case we'll create the DC in system memory, which is controlled directly by the operating system, rather than in hardware video memory.

Create the memory DC with a call to `MGL_createMemoryDC`. Note that the x and y dimensions of the windowed DC are used in this call, as well as the bit depth and pixel formats set earlier in the call to `GetMemoryBitmapDepth`:

```
if ((memDC = MGL_createMemoryDC(sizex,sizey,bits,&pf)) == NULL)
    MGL_fatalError("Unable to create Memory DC!");
```

Changing and Realizing the Windows Color Palette

All DCs contain a palette member. Because we intend to Blt pixels from our memory DC to our windowed DC, the palettes for the memory DC must match that of the windowed DC. The SciTech MGL automatically takes care of creating identity palettes when you realize the palette for the memory DC and the windowed DC (A bitmap is said to have an *identity palette* is it's palette matches the palette on the hardware). However, you must ensure that each is *realized*, which means updating the hardware with the color values in the DC palette. Palette realization in the hardware is taken care of with a call to `MGL_realizePalette`. Since realizing a palette is an expensive process, call `MGL_realizePalette` only after you have performed all palette manipulations you wish to before updating the display. Whenever you change the palette, you must again realize the palettes for all DCs. The code for accomplishing all of this looks like this:

```
MGL_getPalette(dogDC,pal,MGL_getPaletteSize(dogDC),0);
MGL_setPalette(memDC,pal,MGL_getPaletteSize(memDC),0);
MGL_realizePalette(memDC,MGL_getPaletteSize(memDC),0,false);
MGL_setPalette(winDC,pal,MGL_getPaletteSize(winDC),0);
MGL_realizePalette(winDC,MGL_getPaletteSize(winDC),0,false);
```

Getting Access to all 254 entries in the Color Palette

Windows uses first and last 10 color entries for system colors. This so called static palette means you only have access to 236 color entries. You can make Windows give up it's hold on these entries by making the system palette non-static with a Windows API call:

```
SetSystemPaletteUse(hdc, SYSPAL_NOSTATIC);
```

To restore the system palette, simply make the same call with SYSPAL_STATIC:

```
SetSystemPaletteUse(hdc, SYSPAL_STATIC);
```

If you choose to use system palette entries in this manner, you should trap the `WM_ACTIVATE` message so that you can restore the system palette when your app loses the focus, and reclaim control when your app regains the

focus. For more information, see WSHOWBMP.C.

Drawing Something to the Memory DC

In the SciTech MGL you never draw directly to the windowed DC. Instead you composite your scenes in a memory DC, then Blt the composited scene to the windowed DC to update the display. This process is exactly the same as for a full-screen display DC, except you don't see the image in the memory DC until you Blt it to the screen.

In WMGLDOG, we copy doggies around the screen as long as the user holds the left or right mouse button down and moves the mouse. In effect, we copy a doggie from `dogDC` to the memory DC (`memDC`) with a call to `MGLtransBltCoord` (This is where the transparency color we extracted earlier in InitMGL comes in handy):

```
leftDown = (GetKeyState(VK_LBUTTON) < 0);
rightDown = (GetKeyState(VK_RBUTTON) < 0);
if (!leftDown && !rightDown)
    break;
hdc = GetDC(hwnd);
GetViewportOrgEx(hdc,&ofs);
ReleaseDC(hwnd,hdc);
x = LOWORD(lParam) - width/2 - ofs.x;
y = HIWORD(lParam) - height/2 - ofs.y;
if (leftDown) {
    MGL_transBltCoord(memDC,dogDC,0,0,width,height,x,y,
        transparent,true);
    }
else if (rightDown) {
    MGL_bitBltCoord(memDC,dogDC,0,0,width,height,x,y,
        MGL_REPLACE_MODE);
    }
```

Note that if the user is holding the right mouse button down, the Blt is accomplished with a call to `MGL_bitBltCoord`, which Blts without transparency.

Blting the Results to the Window

Now that we've created the scene in `memDC`, we still don't see the scene in the window. In order to see the results, we Blt the composited scene to the windowed DC:

```
MGL_bitBltCoord(winDC,memDC,x,y,x+width,y+height,x,y,
    MGL_REPLACE_MODE);
```

Since we know the width and height of the doggie image, and the

coordinate of the new image, we know which portion of the memory DC has actually being changed. For efficiencies sake, we need only Blt that modified portion to the window DC (called the *dirty rectangle*).

Stretching to a Resized Window

When Blting to a window without stretching, you specify only the top and left coordinates of the source image and the top left coordinate in the destination DC to which to Blt the image. To stretch an image when Blting, call `MGL_stretchBLT`, and provide the top left corner of the destination rectangle, as well as the bottom right corner. The SciTech MGL will map pixels in the source DC to the target rectangle. You can shrink or expand images in this way. For more information, see WSHOWBMP.C.

Note: Stretches to arbitrary aspect ratios can be slow. However, both SciTech MGL and Windows are optimized for efficient stretches by a factor of 2 in both the X and Y dimensions (i.e.: stretching a 320x240 bitmap to a 640x480 window).

Repainting the Window Contents

When windows repaints a window, it automatically clips the repaint to that portion of the window which has actually changed (for example, the portion of a window exposed by moving another window away from it). Thus, on the WM_PAINT, we Blt the entire backbuffer DC to the Window DC, trusting Windows to clip out the unmodified portions.

Interacting with the User

You can use the MGL event handling routines in a windowed application in exactly the same way you'd use them in a full-screen application. In fact the portability of this code between full-screen and windowed apps is a major benefit of the SciTech MGL event handling functionality. Of course, this may be impractical for some reason, such as the existence of legacy code, the recoding of which is precluded by time or cost factors. You can create your own window procedure to process events just as you would for any other windowed application.

Destroying SciTech MGL Before Exit

When your program exits you must destroy all the DCs you created, as well

as deallocate the memory they occupy. Fortunately, this is easily accomplished with a call to `MGL_exit`.

Advanced MGL Programming

Page Flipping for Smooth Animation (Double and Triple Buffering)

Page flipping is handled very easily in SciTech MGL. When you create a fullscreen MGLDC with a call to `MGL_createDisplayDC`, you specify the number of pages or *buffers* that you want that device context to support, e.g.,

```
myMGLDC = MGL_createDisplayDC(2);
```

Of course, if the hardware cannot support that many pages in the specified graphics mode the function will fail. You can help to prevent this with a call to `MGL_availablePages` first, to determine how many pages the hardware can support. Thus the code for creating a display DC with the maximum number of pages looks something like this:

```
numPages = MGL_availablePages(mode);
myMGLDC  = MGL_createDisplayDC(numPages);
```

Once you've created your full-screen MGLDC, you must enabled double-buffering with a call to `MGL_doubleBuffer`:

```
if (!MGL_doubleBuffer(myMGDC))
    MGL_fatalError("Unable to start double buffering!");
```

If for some reason double-buffering cannot be initialized this function returns false (such as when you forget to specify more than one page when you create the full-screen DC).

Implementing Page Flipping

All MGL drawing operations are carried out on the *active* page of the *current* device context in fullscreen graphics modes. The pixels displayed on the screen are display from the *visual* page. When you've finished drawing and you're ready to display the scene, you need to flip the pages, making the active page the visual page, and the visual page the active page, ready to be drawn on again.

To flip display pages, call `MGL_swapBuffers` on your MGLDC. You choose whether you want the hardware to wait for vertical retrace or not:

```
MGL_swapBuffers(myMGLDC, MGL_waitVRT);
```

Implementing Multiple Buffering

`MGL_swapBuffers` is an easy way to implement double-buffering (page flipping with two buffers). Today's hardware with lots of available video memory offers the possibility of using 3 or more buffers for multi-buffering. SciTech MGL provides full functionality for multi-buffering with two functions. To select an active page, call `MGL_setActivePage` on your DC, passing in the page you want to make active for drawing:

```
MGL_setActivePage(myMGLDC, 0); //make the front page active
```

Then to set the visual page, call `MGL_setVisualPage` on your DC, passing in the new page to display:

```
MGL_setVisualPage(myMGLDC, 1); // display the back page
```

Obviously you'll not want to specify a page greater than the number initialized with your MGLDC. You can check this value anytime by accessing the Mode Information member of your display DC:

```
maxPages = myMGLDC->mi.maxPage;
```

See the MGL Library Reference for more information.

Swapping the Multiple Buffers

You can use the properties of the mode information structure to manage multiple buffers yourself. For example, to swap buffers:

```
MGL_setVisualPage(activePage);
activePage = (activePage + 1)  %  (maxPage + 1);
MGL_setActivePage(activePage);
```

Directly Accessing the Device Context Surface

The SciTech MGL abstracts video hardware for you, obviating the need for complicated routines for direct access to video memory. However, services exist which allow you to directly manipulate the display surface itself (the video memory which contains pixel information) for custom rendering routines not supported by SciTech MGL.

Before you can access a display surface directly, you must determine which type of access to that surface (if any) your application has. Access type is determined by system configuration; some hardware doesn't allow direct access. Call `MGL_surfaceAccessType` on your MGLDC to determine the

access type:

```
MGL_surfaceAccessType(*myMGLDC);
```

This function returns one of the surface access flags enumerated in `MGL_surfaceAccessFlagsType`, which are summarized in the following table:

Flag	Description
`MGL_NO_ACCESS`	Surface cannot be accessed
`MGL_VIRTUAL_ACCESS`	Surface is virtualized
`MGL_LINEAR_ACCESS`	Surface can be linearly accessed

Using Linear Access

With `MGL_LINEAR_ACCESS`, you can treat the display surface like regular memory. The MGLDC has a member variable, `surface`, which points to the beginning of the surface associated with the MGLDC. Before you can write to the surface, however, you'll need to know how to compute a pixel address. Given an X and Y coordinate, `MGL_computePixelAddress` returns a pointer to the corresponding pixel address:

```
MGL_computePixelAddress(*myMGLDC, x, y);
```

If you want to perform this calculation yourself, you can use the members of the surface structure of your MGLDC. Essentially this function computes the following:

```
addr = dc->surface + (y * dc->mi.bytesPerLine) +
       (x * bytesPerpixel);
```

Accessing Virtual Linear Framebuffers

Many older SuperVGA devices do not include hardware linear framebuffer support, and hence access to the framebuffer on these devices must be done through a small 64Kb bank switched window. By being able to access the entire framebuffer via a 32-bit near pointer, you can use the same code for rendering to a system memory buffer and for rendering directly to video memory (hence enabling you to write one set of code that can be used for drawing to a DIB and Blting to a real GDI window, or rendering directly to the framebuffer for fullscreen MGL code). In a bank switched environment, SciTech MGL may be able to virtualize the framebuffer to make it appears a 32-bit linear address to your application code, even though underneath we

automatically handle bank switching using a page fault handler.

Note: You must ensure that when you directly access the surface you do so on BYTE, WORD and DWORD aligned boundaries. If you access it on a non-aligned boundary across a page fault, you will cause an infinite page fault loop to occur. To copy blocks of memory to the display in a manner that is both fast and *virtual safe*, see the functions `MGL_memcpyVIRTSRC` and `MGL_memcpyVIRTDST`.

Accessing Surface Color Information

If your using an 8 bpp mode, then writing pixel information directly to video memory is easy: just write each byte to the proper address that represents the color you want on the screen. If you're using higher color modes such as 15 bpp and above modes, you'll need to packed the color information for a pixel into the appropriate format before you can write it to memory. This information on how the pixels are packed in memory is stored in the PixelFormat structure of your MGLDC (`MGLDC->pf`). SciTech MGL provides functions to pack the color values for you, such as `MGL_packColor` which packs the colors from a 24-bit RGB tuple to the appropriate pixel format:

```
colorBits = MGL_packColor(&myMGLDC->pf, Red, Green, Blue);
```

Note that this routine returns a `color_t`, which is basically a long integer with the color information correctly packed into the appropriate bit positions.

Note: In order to achieve maximum performance for custom rendering code that does color packing, you might want to use the `MGL_packColorFast` macro or even hard code the equivalent of this macro into your code.

To read color information from the hardware display device, use `MGL_unpackColor` on the pixel you have read from memory:

```
myColor = MGL_getPixelCoord(0,0);
MGL_unpackColor(&myMGLDC->pf, myColor, *red, *green, *blue);
```

This returns an equivalent RGB color for the `color_t`, minus the loss in precision (e.g., 5:6:5 gets converted to 8:8:8, with the bottom bits truncated or set to 0).

For more information on direct memory access, refer to the DIRECT.EXE

sample program.

Creating Offscreen Device Contexts

You can create offscreen device contexts in offscreen video memory for storing bitmaps, sprites, or whatever else you'd like handy in hardware memory. Although the SciTech MGL Sprite Manager can perform most of these functions for you, you can also directly access this functionality if you choose.

You can create two kinds of offscreen DCs:

- Rectangular contexts

  Bitmaps are stored in rectangular chunks of memory, references by their planar (x, y) coordinates in memory. This can require careful fitting to ensure optimal use of video memory (the Sprite Manager has automatic routines to do this for you).

- Linear contexts

  Bitmaps are stored linearly in memory. Not supported by all hardware devices, but completely eliminates wasted memory.

To create a rectangular offscreen MGLDC, call `MGL_createOffscreenDC`:

```
myMGLDC = MGL_createOffscreenDC();
```

To create a linear offscreen MGLDC, call `MGL_createLinearOffscreenDC`:

```
MGL_createLinearOffscreenDC();
```

Storing Bitmaps in Offscreen Device Contexts

With your offscreen DCs created, it's a simple matter of using `MGL_loadBitmapIntoDC` or `MGL_putBitmap` to stuff bitmaps into them. In short, they perform identically to other DCs for the purposes of writing bitmaps.

You need only use `MGL_memcpy` to write a bitmap directly to a linear buffer, although you will need to keep track of exactly where you put the bitmaps in memory:

```
MGL_memcpy(source, pixelAddr, numBytes);
```

Blting Offscreen Memory Bitmaps

For rectangular offscreen DCs, Blt just as you would with any other rectangular DC by using `MGL_bitBlt` or `MGL_transBlt`, except of course that the source DC for these Blts is an offscreen DC.

For linear offscreen DCs, use `MGL_bitBltLin`, and `MGL_transBltLin`, passing in the starting address for the bitmap which is the address that you copied the bitmap to.

Using Mouse Cursors

Loading your own custom mouse cursors can be an easy and effective way of customizing your full-screen game environment. The SciTech MGL makes it easy to load and manipulate mouse cursors using standard Windows 3.1 mouse cursor (.cur) files.

Load a cursor with `MGL_loadCursor`:

```
myCursor = MGL_loadCursor("c:\\stuff\\mycursor.cur");
```

Once a cursor is loaded, make it the current cursor with a call to `MS_setCursor`:

```
MGL_setCursor(*myCursor);
```

If you like, you can change the cursor color in a single operation with `MS_setCursorColor`:

```
MS_setCursorColor(newColor);
```

Make sure that the color you pass in is a `color_t` structure in the correct format for the current display mode!

See the MOUSE.EXE sample program for more information.

Double Buffered Mouse Cursors

Double-buffering mouse cursors provides a quick and easy way to provide flicker free mouse cursor animation. There are two methods for double-buffered cursor animation using the SciTech MGL.

For the first method, all you have to do is hide the cursor with a call to `MS_hide`, and then draw the new cursor on your active buffer with `MS_drawCursor` after drawing the next frame in your animation and then

switch buffers. This type of animation requires that you completely update the entire display memory every frame, to erase the old image of the moue cursor. As long as the frame rate remains relatively high, the mouse will appear to move smoothly. The following code from the MOUSEDB.EXE sample program illustrates the concept:

```
MS_show();
MS_hide();

initAnimation();
do {
    /* Clear the entire display device before drawing the next
     * frame
     */
    mainWindow(dc,"Double buffered mouse cursor");
    statusLine("Method 1: Re-render entire scene per frame");

    /* Draw the clock at the current location and move it */
    drawClock();
    moveClock();

    /* Draw the mouse cursor on top of the current frame and
     * flip
     */
    MS_drawCursor();
    MGL_swapBuffers(dc,true);
    } while (!checkEvent());
waitEvent();
```

The second method can be more appropriate for situations where you may not be re-drawing the entire frame or require the mouse to move in between page flips (i.e.: if you flip on an intermittent basis such as in an RPG). In this case, just turn on the mouse cursor and SciTech MGL will automatically take care of moving it and ensuring it turns up at the right places after a flip. Note that the mouse cursor will *always* be visible on the visible page so you don't need to show and hide the cursor while drawing to the active page in these modes.

```
MGL_doubleBuffer(dc);
mainWindow(dc,"Double buffered mouse cursor");
MGL_swapBuffers(dc,false);
mainWindow(dc,"Double buffered mouse cursor");
MS_show();

initAnimation();
do {
    /* Draw the clock at the current location and move it */
    clearDirtyRect();
    drawClock();
    moveClock();

    /* Flip buffers */
    MGL_swapBuffers(dc,true);
    } while (!checkEvent());
waitEvent();
```

Displaying Stereo Images for LC Shutter Glasses

The SciTech MGL includes full support for creating *stereoscopic* display device contexts for display 3D images that can be viewed in *stereo* by users wearing inexpensive LC shutter glasses (such as the StereoGraphics SimulEyes glasses). Stereo support in SciTech MGL is very simple and intuitive, and you create a stereo display device context just the same as you would for a normal display device context:

```
stereoDC = MGL_createStereoDisplayDC(2,120);
```

Note that the number of buffers you pass in is the number of *stereo* buffers that you want, and the maximum you can create will be half the number of buffers normally available in that mode, because SciTech MGL needs to use one buffer for the left eye image and another buffer for the right eye image.

In order to display the stereo image, when stereo mode is turned on SciTech MGL will automatically flip between the left and right eye images on every vertical refresh of the display adapter (and simultaneously signal to the LC shutter glasses when the left and right image is being display so they can blank out the appropriate eyes). Because of this the effective refresh rate that the users sees through the glasses will be half that of the real refresh rate, and hence at 60Hz the user will perceive 30Hz per eye! In order to get around this problem, SciTech MGL allows you to pass in a desired refresh rate as the second parameter to the above function. SciTech MGL will attempt to use that refresh rate or the next lowest available refresh rate for the mode instead of the adapter default refresh rate, allowing you to display your images at very high refresh rates such as 120Hz for ergonomic stereo viewing.

Once you have created the stereo display device context, the display mode will look like any regular non-stereo display mode. In order to get SciTech MGL to start the automatic stereo page flipping, you must enable stereo page flipping:

```
MGL_startStereo(stereoDC);
```

and you can stop it with the following:

```
MGL_stopStereo(stereoDC);
```

Of course when you are doing all drawing, you also have to tell SciTech MGL which buffer you want to draw to (left or right eye) and you do that with the `MGL_setActivePage` function and pass in the `MGL_LEFT_BUFFER` or `MGL_RIGHT_BUFFER` flags:

```
MGL_setActivePage(0 | MGL_LEFT_BUFFER);
... do drawing to the left buffer
MGL_setActivePage(0 | MGL_RIGHT_BUFFER);
... do drawing to the right buffer
```

For more information on using stereo with SciTech MGL, see the STEREO.EXE sample program or the SKYFLY.EXE OpenGL sample program.

Note: In stereo display modes, because you have to draw both a left eye image and a right eye image, it may be beneficial to run the application in a lower resolution than the regular resolution to cut down on the number of pixels that have to be draw. You might also want to have the vertical resolution and draw to a system memory DC, and then use SciTech MGL to do a fast 1x2 stretch to stretch the buffer to the screen (i.e.: use a 640x240 buffer and stretch it to a 640x480 screen). Otherwise you can expect your application to run at half the regular frame rate due to the increased rendering time for the stereo image pairs.

Debugging Fullscreen SciTech MGL Applications

In order to debug MGL fullscreen applications under Windows 95, you must run your normal debugger in dual monitor mode, with all debugger output displayed on a monochrome monitor, or you must use remote debugging. Using a normal fullscreen or GDI debugger window, you will not be able to see the standard Windows debuggers output screens once you have shut down GDI.

Once you have set up your debugging environment properly, you should be able to step through and trace all MGL code while GDI is still shut down. Note that sometimes crashing your application while running WinDirect modes can cause a system lockup. In these cases unless you a debugging WinDirect specific problems, you might find it more useful to debug in DirectDraw only modes. In order to force SciTech MGL to ignore WinDirect and not load any of the drivers (even if you run with DirectDraw, if SciTech MGL has loaded WinDirect, Windows 95 will not clean up properly when the app terminates), you can set the following environment variable in your code:

```
putenv("MGL_USE_WINDIRECT=0");
```

What is the Game Framework?

The Game Framework is a comprehensive wrapper for the SciTech MGL designed for the Game Programmer. It relieves much of the tedium associated with Windows programming without sacrificing any of the performance inherent in SciTech MGL. For example, using the Game Framework it's possible to code your entire application without constructing a single window, or writing any of the support code you'd normally have to write for a windows application. The Game Framework can take care of it for you, allowing you to concentrate on what's really important: your game.

The Game Framework also includes the Sprite Manager, which you can use to manage all the bitmaps used in your game. All you have to do is tell the Sprite Manager to load your bitmaps, and it knows how to store them in video memory for fast hardware Blts if your system supports it. Sprite Manager can also manage video memory and begin storing sprites in system memory if video memory runs low.

We'll use BOUNCE.EXE as an example of a simple Game Framework application in the following section. Take a moment to load the project into your IDE and browse the code.

Using the Game Framework

Initialization of the Game Framework consists of six primary steps:

1. **Call to GM_setDriverOptions**

   This routine tells the Game Framework which driver technologies you want to support in your game. By default all of them are enabled, and you can use this function to disable certain driver technologies at runtime for compatibility in the field. You may call this function as many times as you wish to change the driver options on the fly, and if the values change the Game Framework will re-enumerate the list of available graphics modes for you

   The `driverOpt` parameter is a variable of type `GM_driverOptions`, which contains several boolean fields for enabling or disabling different

drivers, such as UseWinDirect, UseVGA, etc. You'll want to fill in the fields of this object before calling `GM_setDriverOptions`.

2. **Call to GM_init**

   This call starts the game framework, and also returns an object of type GMDC, which contains information necessary to continue the initialization process.

3. **Registration of your application callback functions.**

   The Game Framework works by calling your registered application callback functions in response to certain events, and regularly in the course of executing it's main loop routine (`GM_mainLoop`). So, all you have to provide are routines for handling game logic, drawing the screen, dealing with changes in graphics modes, etc. We'll cover application callback function registration in detail in the next section.

4. **Selection and setting of Graphics mode.**

   You must find and select the appropriate graphics mode for your application.

5. **Pre-Main Loop initialization**

   Before the Main Loop begins, you'll need to setup the screen and perform any other functions which you only want to happen once on startup. In BOUNCE.C, the `myInit` routine initializes several global variables used by other routines in the application.

6. **Execute the main loop.**

   The Game Framework provides you with a built in main loop routine, which will execute the application callback functions you registered earlier.

   That's all there is to it!

Setting Driver Options

Before your application starts up you'll need to tell the Game Framework which driver technologies you wish to support with a call to `GM_setDriverOptions`:

```
GM_setDriverOptions(&driverOpt);
```

The argument to this function is an object of type `GM_driverOptions`, which you should fill in ahead of time with the drivers you want to support. By default all available drivers are enabled, but you can use the `GM_driverOptions` argument to this function to disable certain driver technologies at runtime for compatibility in the field. For example, in BOUNCE.C, we've opted to enable all driver technologies except hardware OpenGL:

```
GM_driverOptions driverOpt = {
    true,           /* UseWinDirect*/
    true,           /* UseDirectDraw*/
    true,           /* UseVGA*/
    true,           /* UseVGAX*/
    true,           /* UseVBE*/
    true,           /* UseVBEAF*/
    true,           /* UseLinear*/
    true,           /* UseFullscreenDIB*/
    false,          /* UseHWOpenGL*/
    MGL_GL_AUTO,    /* OpenGLType */
    GM_MODE_ALLBPP, /* modeFlags*/
    };
```

You may call this function as many times as you wish to change the driver options on the fly, and if the values change the will re-enumerate the list of available graphics modes for you.

The `GM_driverOptions` structure also contains the `modeFlags` field which represents the color depths that you will be supporting in your application, so that the Game Framwork will only enumerate modes that your game can support. For instance if you only support 8bpp modes, than pass a value of `GM_MODE_8BPP`. If you support 8bpp and 15/16bpp then pass in a value of `GM_MODE_8BPP | GM_MODE_16BPP`. Note also that you can change the supported mode flags at any time, which is useful if your software renderer only supports 8bpp modes, while in 3D hardware accelerated modes you want to support all available color depths.

Initializing the Game Framework

A call to `GM_init` initializes the Game Framework, and returns an object of type GMDC. This object contains information necessary for subsequent steps in the initialization process.

```
if ((gm = GM_init("Bounce")) == NULL)
    MGL_fatalError(MGL_errorMsg(MGL_result()));
```

Note: You should wrap this call in appropriate code to detect and handle a

failure of the Game Framework to initialize properly.

Registering your Application Callbacks

The `GM_mainLoop` routine takes care of all Windows housekeeping tasks for your application. During the loop, the Game Framwork calls functions which you register to (among other things), composite the next frame, draw the next frame to the screen, handle user input, and handle system events. In BOUNCE.EXE, the callback functions are initialized in the `Main` routine:

```
GM_setDrawFunc(draw);
GM_setGameLogicFunc(gameLogic);
GM_setKeyDownFunc(keydown);
GM_setMouseDownFunc(mousedown);
GM_setModeSwitchFunc(switchModes);
GM_setAppActivate(activate);
GM_setSuspendAppCallback(suspendApp);
```

The arguments to these functions are themselves routines defined in the application.

Inside of `GM_mainLoop`, these functions are called in this order:

```
GM_exitMainLoop = false;
while (!GM_exitMainLoop) {
   GM_processEvents ();      //     Farm out keyboard, mouse,
                             //     system events to your
                             //     registered functions
   MyGameLogic();            //     Your registered game logic
                             //     routine
   if (GM_doDraw)
       MyDrawFrame();        //     Your registered draw function
   }
GM_cleanup();
```

Keyboard Callbacks

Initialize your keyboard event handling routines with `GM_setKeyDownFunc`, passing in the name of the function you've written to handle these events. In BOUNCE.C, the `keydown` routine makes use of the MGL unified event queue to trap the only keyboard event we care about, pressing the ESC key:

```
void keydown(event_t *evt)
{
    switch (EVT_asciiCode(evt->message)) {
        case 0x1B:
            GM_exit();
            break;
        }
}
```

Your callback routine will be passed a copy of the event itself, packaged in a variable of type `event_t`. You can parse this object to derive all the information you need about the event. See the MGL Library Reference for more information about the `event_t` structure.

Other keyboard callbacks which you can register with the Game Framework include routines for handling keyup and keyrepeat events. Register these callbacks with calls to these routines:

```
GM_setKeyUpFunc(myKeyUp);
GM_setKeyRepeatFunc(myKeyRepeat);
```

It's up to you whether you wish to handle all keyboard events in a single keyDown routine, or provide separate routines for each of the possible keyboard events.

Note: These callbacks are called from within `GM_mainLoop` several times per frame to ensure that all events are handled correctly until the event queue is empty.

Mouse Callbacks

Your application will almost certainly need to respond to mouse events, and you may wish to manipulate the mouse in other ways, such as drawing custom cursors or obscuring the mouse cursor. In BOUNCE.C, mouse events are handled by the `mousedown` routine, which simply restarts the animation when a mousedown event occurs:

```
void mousedown(event_t *evt)
{
    myInit();
}
```

Your callback event is passed a pointer to a variable of type `event_t`. This variable contains all the relevant information about the event such as mouse x-position, y-position, button states, etc. Refer to the MGL Library Reference for more information about this structure.

Other mouse callbacks are available as well, including callbacks for handling mouseup and mousedown events:

```
GM_setMouseUpFunc(myMouseUp);
GM_setMouseDownFunc(myMouseDown);
```

You can include separate routines for these events, or include all mouse-

handling code in a single mouseEvent routine.

Note: The mouse event callbacks are called several times from within `GM_mainLoop`. All events are passed to your callback functions in the order in which the user performed them.

Trapping Your Own Events

The Game Framework event callbacks cover most of the events your application is likely to see in the event queue. However, they won't trap timer events, nor will they trap any events which you define and post to the event queue. You can trap these events with a "fallthrough case" callback which you set up with a call to `GM_setEventFunc`.

```
GM_setEventFunc(myEventFunc);
```

Your routine will be passed a copy of each event, so you can determine what to do in response. This function is called several times each frame from within `GM_mainLoop` to ensure that all user events are trapped in the correct order.

Game Logic Callback

The Game Logic callback routine determines what will happen in the next scene based on the state of the system and user input. This is the core of your game. In BOUNCE the logic is fairly straightforward; all we need to do is calculate the new position of the ball. If it's at the edge of the display, we bounce it off at a 45 degree angle:

```
void gameLogic(void)
{
    /* Move the ball to new location */
    if (pixelx >= xres-CSIZE)
        crunchx = xres-pixelx-1;
    if (pixely >= yres-CSIZE)
        crunchy = yres-pixely-1;
    if (pixelx <= CSIZE*2)
        crunchx = pixelx-CSIZE-1;
    if (pixely <= CSIZE*2)
        crunchy = pixely-CSIZE-1;
    pixelx += incx;
    pixely += incy;
    if ((pixelx >= xres) || (pixelx <= CSIZE))
        incx = -incx;
    if ((pixely >= yres) || (pixely <= CSIZE))
        incy = -incy;
}
```

You should perform all non-drawing related operations in this routine, such as network updates, sound processing, etc.

Draw Callback

After the game logic function has determined what is happening on the screen, the draw callback routine applies these changes to the display. In BOUNCE.C, this routine draws the current frame to the device context and then swaps the display buffers:

```
{
    rect_t            dirtyRect;
    region_t         *dirty = MGL_newRegion();

    /* Draw the ball at the current location */
    MGL_clearDevice();
    MGL_setColorCI(13);
    MGL_fillEllipseCoord(pixelx,pixely,CSIZE+crunchx,CSIZE+crunchy);
    MGL_setColorCI(12);
    MGL_ellipseCoord(pixelx,pixely,CSIZE+crunchx,CSIZE+crunchy);
    dirtyRect.left = pixelx - (CSIZE+crunchx);
    dirtyRect.right = pixelx + CSIZE+crunchx;
    dirtyRect.top = pixely - (CSIZE+crunchy);
    dirtyRect.bottom = pixely + CSIZE+crunchy;

    /* Swap display buffers with dirty rectangles */
    MGL_unionRegionRect(dirty,&prevDirty);
    MGL_unionRegionRect(dirty,&dirtyRect);
    MGL_optimizeRegion(dirty);
    prevDirty = dirtyRect;
    GM_swapDirtyBuffers(dirty,true);
    MGL_freeRegion(dirty);
}
```

Using Dirty Regions

This routine makes use of SciTech MGL's arbitrary region technology and the Game Frameworks dirty region features. When the next frame is being composited in the backbuffer, SciTech MGL keeps track of a list of those regions (really a union of rectangles) which are actually updated. When it's time to swap the display buffers, only the dirty regions are Blted to the screen with the call to `GM_swapDirtyBuffers`; a more efficient scenario than blindly Blting the whole DC when only a small percentage of it is actually in need of update.

Note: This function is not called when the application is minimized to avoid writing to memory owned by another application.

To swap buffers without the dirty region logic, just call `GM_swapBuffers`,

with a flag to tell the Game Framework whether to wait for vertical retrace or not:

```
GM_swapBuffers(true);
```

More Advanced Callbacks

Activation Callbacks

The activation callback is called when your application starts up, or is activated after being minimized from the taskbar. Register your activation callback function with `GM_setAppActivate`:

```
GM_setAppActivate(myActivate);
```

Your callback is passed a flag that indicates whether your game is now currently active or not, and should be used to enable and disable support for things such as CD-Audio when your application loses activation (or the current focus).

Mode Switch Callback

Use the mode switch callback to provide support for toggling between full-screen and windowed modes. By default, this callback is set to null, so if you wish to provide support for multiple graphics modes you must register a mode switch callback with a call to `GM_setModeSwitchFunc`:

```
GM_setModeSwitchFunc(myModeSwitch);
```

By default the Game Framework contains code to provide two methods of switching to fullscreen modes when running in windowed modes:

- When the user hits the *Alt-Enter* key combination
- When the user clicks the *Maximize* button on the games title bar

Likewise when the game is running in a fullscreen mode and the user hits the *Alt-Enter* key, the graphics mode will automatically be switched to windowed mode. In order to support auto-switching between fullscreen and windowed modes, all the MGL device contexts will be destroyed and re-created during the switch, so you will have to include other code to re-initialize SciTech MGL to the state that the game is currently in (i.e.: setting the color palette etc) in your mode switch callback. You will also need to

code your game in such as way that it can handle dynamic resolution changes on the fly.

Note that your mode switch callback will be passed a variable of type `GM_modeInfo`, containing relevant information about the mode which is about to be switched to.

Mode Filter Callback

The mode filter callback is used to filter out graphics modes which are not appropriate for your application during mode enumeration. For example, you could use the mode filter callback to limit mode enumeration to 1:1 aspect ratio modes. Register your mode filter callback with a call to `GM_setModeFilterFunc`:

```
GM_setModeFilterFunc(myFilterFunc);
```

Pre-Mode Switch Callback

The pre-mode switch callback function is called when switching on the fly between full-screen and windowed modes. This callback is called *before* the mode switch callback, and is the appropriate place to destroy any internal data structures that might need to be cleaned up before the current mode is destroyed and the new one created. Initialize your pre-mode switch callback with a call to `GM_setPreModeSwitchFunc`:

```
GM_setPreModeSwitchFunc(myPreModeFunc);
```

Starting Graphics Modes

After you've initialized the drivers you're application will use, you need to select a graphics mode. Available graphics modes are enumerated in the `modeList` field of the `GMDC` object initialized in the call to `GM_init` (See the MGL Library Reference for more information about the GMDC type).

Finding Supported Graphics Modes

The next step is to find the highest performance mode within this list which supports the resolution and color depth you require with a call to `GM_findMode`:

```
GM_findMode(&info,320,200,8);
```

Pass in a pointer to a GM_modeInfo (info in this call) structure, as well as

desired values for X and Y resolution and color depth.

This is most useful for finding a good default graphics mode to start your game in if the user has not selected a default mode yet. Note that this function searches for the mode from the top of the list backwards, so that we find the highest performance 320x200 and 320x240 modes (i.e.: the Linear Framebuffer modes rather than the VGA ModeX or Standard VGA modes). When it finds an appropriate mode, `GM_findMode` fills in the fields of the supplied GM_modeInfo structure with information about the mode (see the MGL Library Reference for more information about the `GM_modeInfo` structure).

Both `GM_findMode` and `GM_setMode` return True on success and False on failure. Be sure to wrap calls to these functions in the appropriate code to trap and handle errors on initialization!, e.g.:

```
if (!GM_findMode(&info,320,200,8))
    MGL_fatalError("Unable to find 320x200x256 graphics mode!");
```

Setting the Graphics Mode

`GM_findMode` fills in the info structure with information about the selected graphics mode. Now, you use the same structure to set the selected mode with a call to `GM_setMode`:

```
GM_setMode(&info,startWindowed,3,true);
```

The third argument to `GM_setMode` is the number of video buffer pages you'd like to have available to your application. The Game Framework will attempt to provide you with this number of buffers, but if for some reason it cannot (limited memory, for example), it will provide you with as many as it can up to the amount you specify.

You don't need to keep track of the active page when you're rendering animation in your code. The Game Framework keeps track of all that for you. All you have to do is call when you scene is complete and it's time to update the display.

Setting the Palette

You set the Game Framework palette for the currently active device context with a call to `GM_setPalette`. This argument takes an object of type `palette_t` which has already been initialized, as well as the number of colors and a start index for the palette. The `myInit` routine in BOUNCE.C

shows how to initialize and set the Game Framework palette:

```
palette_t pal[256] = {{0,0,0},{0,0,0},{0,0,0},
                      {0,0,0},{0,0,0},{0,0,0},
                      {0,0,0},{0,0,0},{0,0,0},
                      {0,0,0},{0,0,0},{0,0,0},
                      {255,255,255},{255,0,0}};
GM_setPalette(pal,256,0);
```

Any time you set or change the palette, you must realize the palette in hardware with a call to `GM_realizePalette`:

```
GM_realizePalette(256,0,true);
```

Accessing the Entire Palette

As discussed in a previous chapter, the Windows operating system reserves 20 colors in the system palette for its own use. You can gain access to this palette with a call to `GM_initSysPalNoStatic`:

```
GM_initSysPalNoStatic(true);
```

Starting OpenGL 3D Rendering Support

The SciTech Game Framework provides complete support for OpenGL rendering via the OpenGL 3D API. You can enabled OpenGL support in the Game Framework with a call to `GM_startOpenGL`:

```
GM_startOpenGL(flags);
```

After this call you must do all rendering via calls to the OpenGL API. The flags parameter (of type `MGL_glContextFlagsType`) is used to specify the type of OpenGL rendering context that you want, such as if you want RGB or color index mode, single or double buffering, an alpha buffer, an accumulation buffer, a depth buffer (z-buffer) and a stencil buffer.

If you pass in a value of MGL_GL_VISUAL for the flags parameter, SciTech MGL will use the OpenGL visual that was set by a previous call to `MGL_glSetVisual`. Hence if you require more control over the type of OpenGL rendering context that is created, you can call `MGL_glChooseVisual` and `MGL_glSetVisual` before calling this function. Note that you should *not* call `MGL_glCreateContext` when using the Game Framework, but call this function instead.

Note: After this function has been called, the current rendering context will have been made the current OpenGL rendering context with a call to

`MGL_glMakeCurrent`, so you can simply start issuing OpenGL rendering commands to start drawing after calling this function.

Capturing Window Messages Directly

By default, the Game Framework handles all Windows overhead for you. If you choose to, you can use the event handling services of the Game Framework as well as the SciTech MGL to avoid the need for any windows procedure coding. However, there are situations which require that you write your own windows code. For example, you may have legacy code which is perfectly satisfactory, and which you wish to use with your Game Framework application.

As is the case with SciTech MGL, you need only register your windows procedure with a call to `GM_registerMainWindow` in order to use it with the Game Framework:

```
GM_registerMainWindow(handle);
```

Note: Be sure to make the call to `GM_registerMainWindow` *after* you call `GM_init`.

Your First Game Framework Application

Load and run BOUNCE.C. Although this is a very simple application, you can modify it in many ways to familiarize yourself with the Game Framework and with the SciTech MGL.

What is the Sprite Manager?

The Sprite Manager is designed to manage a list of bitmaps in memory all at once. For example, you might use the Sprite Manager to manage all the sprites you're going to use for a given level in a game. The Sprite Manager can create either opaque or transparent bitmap objects for that level. If the hardware supports accelerated bitmap rendering, the Sprite Manager will cache as many of those bitmaps in an MGL offscreen memory DC as possible. If there is no hardware support (or we have run out of offscreen display memory) bitmaps will be created and rendered in software. Source transparent software bitmaps are actually compiled into RLE encoded bitmaps and rendered directly to the display surface for maximum speed.

You can make your main device context a system memory device context, in which case no hardware rendering will be used. When managing rectangular offscreen memory device contexts, Sprite Manager uses the MGL complex region manipulation routines to keep track of what parts of the DC have been allocated so that we can efficiently utilize available memory by tucking small bitmaps into any gaps that may arise. Bitmaps are always allocated in a left to right and top to bottom fashion, so you may want to experiment when loading your bitmaps to find the order which minimizes the amount of wasted space. Of course, if you are using a linear offscreen DC then there won't be any wasted space. The Sprite Manager will automatically delete all the bitmaps when the bitmap manager is emptied.

Initializing the Sprite Manager

Call `SPR_mgrInit` to initialize the Sprite Manager. Pass in a device context for which to enable the Sprite Manager, and a flag telling the support manager whether or not to support Run-Length Encoding:

```
SPR_mgrInit(myDC, true);
```

Note: You must reinitialize the Sprite Manager and reload all your bitmaps when you switch from full-screen to windowed graphics modes, or between full-screen modes. The reason for this is that the pixel

formats and available display memory may be completely different in windowed modes and fullscreen modes.

Adding a Bitmap to the Sprite Manager

You can add bitmaps to the Sprite Manager in one of two ways:

- As a transparent bitmap
- As an opaque bitmap

Adding a Transparent Bitmap

Add a transparent bitmap to the Sprite Manger with a call to `SPR_addTransparentBitmap`:

```
SPR_mgrAddTransparentBitmap(bmp, transparent);
```

Where `bmp` is a pointer to a bitmap of type `bitmap_t`, and `transparent` (of type `color_t`) is the transparency color. This routine returns a pointer to the bitmap, of type `SPR_bitmap`. Use this pointer to access the bitmap in the Sprite Manager.

This function automatically determines the most efficient way to store and draw the bitmap depending on the underlying hardware configuration.

Note: The Sprite Manager always uses source transparency for transparent bitmaps.

Adding an Opaque Bitmap

Add an opaque bitmap to the Sprite Manager with a call to `SPR_addOpaqueBitmap`:

```
SPR_mgrAddOpaqueBitmap(bmp);
```

This routine returns a pointer to the bitmap, of type `SPR_bitmap`. Use this pointer to access the bitmap in the Sprite Manager.

Drawing a Sprite

Drawing a sprite is a simple matter, accomplished with a call to `SPR_draw`:

```
SPR_draw(bmp, x, y);
```

Simply provide this function with the `SPR_bitmap` pointer returned when you added it to the Sprite Manager.

Reloading the Hardware After Task Switching

When the user switches between a full-screen window and another window on the desktop, all the video memory holding our sprite information is lost. When the user returns to the full-screen application, you'll need to reload your sprites into the hardware. You can accomplish this with a call to `SPR_mgrReloadHW`:

```
SPR_mgrReloadHW();
```

This function should be called when your application is re-activated (in other words, when the argument to your `GM_suspendAppCallback` function is MGL_REACTIVATE). For example:

```
int ASMAPI SuspendApp(MGLDC*,int flags)
{
    if (flags == MGL_REACTIVATE)
        SPR_mgrReloadHW();
    return MGL_NO_SUSPEND_APP;
}
```

Using OpenGL

The SciTech MGL provides complete support for software and hardware OpenGL rendering. Initialization of OpenGL support consists of these steps:

1. **Register OpenGL Drivers.**
2. **Choose a visual for OpenGL rendering.**

   A visual is the SciTech MGL term for the structure which defines pixel formats in OpenGL rendering contexts.

3. **Set the visual.**

   Having found an appropriate visual for your implementation of OpenGL, make it the current visual for OpenGL rendering operations.

4. **Create an OpenGL rendering context**

   An OpenGL rendering context is analogous to an MGLDC, and is the area of memory to which OpenGL rendering operations are written.

5. **Make the OpenGL rendering context current**

   All OpenGL operations are written to the current OpenGL context.

6. **Perform OpenGL rendering operations**

Register the OpenGL Hardware Drivers

Register the OpenGL hardware drivers with `MGL_registerAllOpenGLDrivers`. The SciTech MGL will utilize these drivers if OpenGL support is available in the hardware. Otherwise, SciTech MGL will fall back on software rendering using SGI's OpenGL for Windows.

Choosing a Visual

A visual is analogous to a pixel format in MGL. Refer to the MGL Library

Reference for more information, but the following table describes the `MGLVisual` structure and it's members:

Flag	Type	Description
`rgb_flag`	bool	True for an RGB mode, false for color index modes
`alpha_flag`	bool	True for alpha buffers (8-bits deep)
`db_flag`	bool	True for double buffered, false for single buffered
`depth_size`	Int	Size of depth buffer in bits
`stencil_size`	Int	Size of stencil buffer in bits
`accum_size`	Int	Size of accumulation buffer in bits

Note: Be sure to set up a variable of this type with the parameters you'd like to see in your OpenGL visual before choosing a visual.

Because hardware capabilities and implementations of OpenGL vary, the SciTech MGL provides a flexible interface for finding the visual that most closely matches your requirement. To find an appropriate visual for your system, call `MGL_glChooseVisual`:

```
MGL_glChooseVisual(dc, &visual);
```

If requested capability is not supported, the visual passed in will be modified for the capabilities that the SciTech MGL does support on your system. For example, the depth buffer may be reduced in size to 16-bits when you requested 32-bits.

Once you've chosen a visual, you set it to make it current for the purposes of creating OpenGL rendering contexts with a call to `MGL_glSetVisual`:

```
MGL_glSetVisual(dc, &visual);
```

Creating and Using OpenGL Rendering Contexts

An OpenGL rendering context is analogous to an MGL Device Context, in that it is an area of memory to which OpenGL commands write their output. OpenGL supports multiple rendering contexts, so you must make the rendering context you want to use the current context before you can begin

rendering to it.

Create a context with `MGL_glCreateContext`:

```
MGL_glCreateContext(myMGLDC, MGL_GL_VISUAL);
```

Passing in MGL_GL_VISUAL as the flags argument tells this command to use the pixel format information returned by the preceding call to `MGL_glChooseVisual` and `MGL_glSetVisual` when creating the rendering context.

If you prefer, you can specify a visual for your OpenGL rendering context when you create it by passing in one or more of the flags enumerated in the `MGL_openGLFlagsType` type. For example, to create a context which supports RGB color mode and double buffering:

```
MGL_glCreateContext(myMGLDC, MGL_GL_RGB | MGL_GL_DOUBLE);
```

See the MGL Library Reference for more details.

After you've created the rendering context, make it the current context with a call to `MGL_glMakeCurrent`. The following code shows how you would set up your visual, create a rendering context, then make it the current context.

```
{
    MGLVisual    myVisual;
    MGLDC        *myMGLDC;

    //Set up the visual we'll use later when we create our OpenGL
    //rendering context.
    myVisual.rgb_flag      = true;
    myVisual.alpha_flag    = false;
    myVisual.db_flag       = true;
    myVisual.depth_size    = 24;
    myVisual.stencil_size  = 8;
    myVisual.accum_size    = 4;
    MGL_registerAllOpenGLDrivers();

    //Create display DC for fullscreen modes, with double buffering
    if ((myMGLDC = MGL_createDisplayDC(2)) == NULL)
        MGL_fatalError("Can't create DC!");
    MGL_makeCurrentDC(myMGLDC);

    //Set up the visual
    MGL_glChooseVisual(myMGLDC, &myVisual);
    MGL_glSetVisual(myMGLDC, &myVisual);

    //Create the rendering context and make it current
    if(!MGL_glCreateContext(myMGLDC))
        MGL_fatalError("Can't create OpenGL context!");
    MGL_glMakeCurrent(myMGLDC);

    ... Your OpenGL rendering code here...
}
```

Swapping the Display Buffers

The number of buffers supported by an MGLDC is passed in as a parameter to `MGL_createDisplayDC`. You needn't worry specifically where these buffers are initialized (whether in system memory or in hardware video RAM). They may actually be in both places. In the MGL OpenGL interface, the active buffer is abstracted from you, the programmer. All you have to do to swap buffers is make a call to `MGL_glSwapBuffers`, passing in an MGLDC and whether or not to wait for vertical retrace:

```
MGL_glSwapBuffers(myMGLDC, true);
```

The SciTech MGL keeps track of which page is the active page for drawing operations and which page is being displayed.

Resizing the Display Buffers

You must explicitly respond to a WM_SIZE message, sent by the operating system when the user resizes a window. Fortunately, all you need to do is

call `MGL_glResizeBuffers`, passing in the MGLDC that's been resized:

```
MGL_glResizeBuffers(myMGLDC);
```

Deleting a OpenGL Rendering Context

Destroy OpenGL rendering contexts with a call to `MGL_glDeleteContext`:

```
MGL_glDeleteContext(myMGLDC);
```

Note: Be sure to call `MGL_glDeleteContext` before you destroy your MGLDCs with a call to `MGL_destroyDC` or `MGL_exit`.

Programming the Hardware Palette

Each device context (and by extension each OpenGL rendering context) has its own associated palette. The video hardware also has an onboard palette. When you make changes to a DCs palette, you are only updating the palette values stored in the palette structure of that DC. In order for the change to be reflected in the display, you must also update the hardware palette.

You change the palette with a call to `MGL_glSetPalette`:

```
MGL_glSetPalette(myMGLDC, myPal, 254, 1);
```

This routine is exactly similar to `MGL_setPalette`, but contains internal code appropriate for the OpenGL environment.

After you've changed the palette for an MGLDC, you must update the hardware palette by calling `MGL_glRealizePalette`:

```
MGL_glRealizePalette(myMGLDC, 254, 1, true);
```

This function is exactly the same as `MGL_realizePalette`, but contains internal code appropriate to the OpenGL environment.

Forcing the OpenGL Implementation

The SciTech MGL supports four implementations of OpenGL. Ordinarily when you start OpenGL, the SciTech MGL will automatically detect the most appropriate implementation of OpenGL to use in your application. However, if you choose to do so you can force your application to use one of

the implementations supported by the SciTech MGL.

Forcing a Specific OpenGL Driver

In the beginning of this section we initialized OpenGL with a call to `MGL_registerAllOpenGLDrivers`. This is a good way to get code up and running quickly, and also ensures that some drives will be found for you to work with. However, the SciTech MGL provides a way for you to force your application to register and use a specific driver if you need to.

You can enumerate a list of those OpenGL drivers which are available on the system at run-time with a call to `MGL_ enumerateDrivers`, which returns a NULL-terminated list of drivers available in the system:

```
MGL_enumerateDrivers();
```

The list returned lists software drivers first, then hardware drivers. You can parse this list, select a driver, then force the system to use that driver with a call to `MGL_glSetDriver`:

```
MGL_glSetDriver(driverName);
```

For example, to force the Mesa implementation:

```
MGL_glSetDriver(“mesa”);
```

Note: You must destroy your MGLDCs and OpenGL contexts before a call to `MGL_glSetDriver`, and recreate them afterwards when you reinitialize OpenGL.

Appendix A : Shipping your MGL Product

In order to ship products you've created with the SciTech MGL, you'll need to be sure you distribute the necessary components of SciTech MGL, including WinDirect files, files for support of OpenGL, DirectX runtime files, etc. There are also platform specific and development environment specific files which must be installed as well. This section discusses which runtime components you will need to distribute.

What is WinDirect?

A key component of the SciTech MGL, WinDirect is a runtime package for DOS and Windows 95 that provides direct access to the display hardware for both 16 and 32-bit applications. Traditionally Windows applications have had to perform all graphics output using the standard Graphics Device Interface (GDI). Although the GDI is very extensive and powerful, it is also not particularly fast for the sort of graphics that real time applications like interactive video games require.

WinDirect breaks this barrier by allowing high performance applications to shut down the normal GDI interface, and to take over the entire graphics display hardware just like you would normally do under DOS. Once GDI has been shut down, interactive graphics applications can re-program the display controller and write directly to video memory. A WinDirect application can program any standard VGA graphics mode such as 320x200x256, it can re-program the controller and run standard VGA ModeX style graphics, or it can call the standard VESA BIOS services to run high resolution SuperVGA graphics.

Note that for maximum portability your should avoid directly using the WinDirect API, and use the standard MGL API instead. SciTech MGL includes full support for WinDirect, but the API is portable to future operating system technologies. Note however that if you are developing MGL applications that use WinDirect, you will need to ship the WinDirect DLL and runtime components with your application and ensure they get installed correctly on the users machine.

MGL Redistributable Components

Subject to the terms and conditions of the SciTech Software License Agreement, in addition to any Redistribution Rights granted therein, you are hereby granted a non-exclusive, royalty-free right to reproduce and distribute the Components specified below provided that (a) is distributed as part of and only with your software product; (b) you not suppress, alter or remove proprietary copyright notices contained therein; and (c) you indemnify, hold harmless and defend SciTech Software and it's suppliers from and against any claims or lawsuits, including attorney's fees, that arise or result from the use or distribution of your software product.

Note that we have not listed any standard MGL runtime library DLL's such as MGLFX.DLL, only the runtime DLL's that are required by SciTech MGL that you do not specifically link to in your application. If you compile and link with the DLL versions of those libraries, you must also ship the corresponding DLL with your application and stored it in the same directory as you application. For more information on the runtime DLL supported by SciTech MGL for different compilers, consult the section 'Compiling and Linking with SciTech MGL' above.

Windows 95 Specific Runtime Files

If you wish to be able to get GDI to draw on MGL fullscreen device context surfaces, you will need to install the following files into the Windows 95 system directory:

Runtime Files	**Purpose**
MGLDIB.DRV	MGL DIB driver for providing support for GDI drawing access to fullscreen MGL device context surfaces. *Only* install on Windows 95, and install to the Windows system directory with version checking.

WinDirect Runtime Files

If you wish to support WinDirect for fullscreen modes under Windows 95, you will need to install the WinDirect runtime libraries files into the same directory as your application program. Do *not* install these files into the Windows system directory!

Runtime Files	Purpose
WDIR16.DLL	WinDirect 16-bit side DLL. Contains all 16-bit side WinDirect functions.
WDIR32.DLL	WinDirect 32-bit side DLL. Contains all 32-bit side WinDirect functions. Requires WDIR16.DLL to interface to the 16-bit subsystem code.

OpenGL Runtime Files

Depending on which implementation's of OpenGL you wish to support with your application, you will need to ship at least one set of the following OpenGL runtime libraries (you always need both the xxGL and xxGLU library files). Note that for Mesa and SGI OpenGL you should install all of these runtime libraries into your applications directory to avoid conflicts with other applications possibly using earlier or later versions of the libraries:

Runtime Files	Purpose
MESAGL.DLL	Mesa OpenGL for Windows 95/NT
SGIGL.DLL	Silicon Graphics OpenGL for Windows for Windows 95/NT.
SGIGLU.DLL	Silicon Graphics OpenGL for Windows Utility Library for Windows 95/NT.
OPENGL95.DLL	Microsoft OpenGL for Windows 95. This file must be installed into the Windows 95 system directory and renamed to OPENGL32.DLL. Do *not* install this file on Windows NT!
GLU95.DLL	Microsoft OpenGL for Windows 95. This file must be installed into the Windows 95 system directory and renamed to GLU32.DLL. Do *not* install this file on Windows NT!

Note: If you plan to support DirectX in your application, we *highly* recommend that you at least ship the latest version of DirectX with your product and provide some mechanism for the end user to install it on their system. We have found that many DirectX related runtime problems can be tracked down to badly installed or older

versions of DirectX and doing a re-install of the latest version can clear up the problems.

Note: All the DLL files should be installed into the same directory as your application for correct operation, except where noted.

Appendix B: Using the Zen Timer

This section provides an overview of the Zen Timer Library, and provides background details on the Zen Timer Library's functionality and how to utilize this functionality in your own applications.

What is the Zen Timer?

The Zen Timer is a C callable library for timing code fragments with microsecond accuracy. The code was originally developed by Michael Abrash for his book "Zen of Assembly language - Volume I, Knowledge" and later in his book "Zen of Code Optimization." We modified the code and made it into a C callable library and added a few extra utility routines, the ability to read the current state of the timer and keep it running, and added a set of C++ wrapper classes. We also added a new Ultra Long Period timer that can be used to time code that takes up to 24 hours to between calls to the timer and with an accuracy of 1/10th of a second.

Since the original implementation of the Zen Timer Library, the library now supports the following timing mechanisms in DOS and Windows:

- Pentium RDTSC instruction for cycle accurate timing
- QueryPerformanceCounter for Win32
- timeGetTime for Win32 compatibility
- 8253 timer chip for DOS compatibility

The Long Period Zen Timer uses as many of the above high precision timing mechanisms to obtain microsecond accurate timings results whenever possible. The following different techniques are used depending on the operating system, runtime environment and CPU on the target machine. If the target system has a Pentium CPU installed which supports the Read Time Stamp Counter instruction (RDTSC), the Zen Timer library will use this to obtain the maximum timing precision available.

Under 32-bit Windows, if the Pentium RDTSC instruction is not available, we first try to use the Win32 QueryPerformanceCounter API, and if that is not available we fall back on the timeGetTime API which is always

supported.

Under 32-bit DOS, if the Pentium RDTSC instruction is not available, we then do all timing using the old style 8253 timer chip. The 8253 timer routines provide highly accurate timings results in pure DOS mode, however in a DOS box under Windows or other Operating Systems the virtualization of the timer can produce inaccurate results.

Timing with the Long Period Zen Timer

Before you can use the Zen Timer in your code, you must first always call the ZTimerInit function to initialize the Zen Timer Library. Once you have done this, to use the timer, isolate the piece of code you wish to time and bracket it with calls to LZTimerOn and LZTimerOff. You then call LZTimerCount to obtain the count use it from within your C program. For example:

```
int i;

void main(void)
{
    ulong count;

    ZTimerInit()
    LZTimerOn();
    for (i = 0; i < 20000; i++)
        i = i;                      /* Do some work */
    LZTimerOff();
    count = LZTimerCount();
}
```

While the timer is running, you can call the `LZTimerLap` function to return the current count without stopping the timer from running.

One point to note when using the long period time is that interrupts are ON while this timer executes. This means that every time you hit a key or move the mouse, the timed count will be longer that normal. Thus you should avoid hitting any keys or moving the mouse while timing code fragments if you want highly accurate results. It is also a good idea to insert a delay of about 1-2 seconds before turning the long period timer on if a key has just been pressed by the user (this includes the return key used to start the program from the command line!). Otherwise you may measure the time taken by the keyboard ISR to process the upstroke of the key that was just pressed.

Note that under DOS the Long Period Zen Timer has a cumulative limit of approximately 1 hour and 10 minutes between calls to LZTimerOn and LZTimerOff.

Timing with the Ultra Long Period Zen Timer

As well as the normal long period Zen Timer functions, we also provide functions that implement an Ultra Long Period Zen Timer. This version of the timer has lower accuracy and can time intervals that take up to 24 hours to execute. There are two routines that are used to accomplish this; `ULZReadTime`() and `ULZElapsedTime`(). The way to use these routines is simple:

```
void main(void)
{
    ulong   start,finish,time;

    ZTimerInit()
    start = ULZReadTime();

    /* Do something useful in here */

    finish = ULZReadTime();
    time = ULZElapsedTime(start,finish);
}
```

Calling `ULZReadTime` latches the current timer count and returns it. You call `ULZElapsedTime` to compute the time difference between the start and finishing times, which is returned in 1/10ths of a second. If you are using C++, you may want to use the simpler C++ classes, which have a common interface for all timers.

When using the Ultra Long Period timer class you must ensure that no more than 24 hours elapses between calls to start() and stop() or you will get invalid results. There is no way that we can reliably detect this so the timer will quietly give you a value that is much less than it should be. However, the total cumulative limit for this timer is about 119,000 hours which should be enough for most practical purposes, but you must ensure that no more than 24 hours elapses between calls to start() and stop(). If you wish to use the timer for applications like ray tracing, then latching the timer after every 10 scanlines or so should ensure that this criteria is met.

Using the C++ interface

If you are using C++, you can use the C++ wrapper classes that provide a simpler and common interface to all of the timing routines. There are two classes that are used for this:

LZTimer	C++ Class to access the Long Period Zen Timer
ULZTimer	C++ Class to access the Ultra Long Period Zen Timer

Each class provides the following member functions:

start() member function

The start() member function is called to start the timer counting. It does not modify the internal state of the timer at all.

lap() member function

The lap() member function returns the current count since the timer was started. This count is the total amount of time that the timer has been running since the last call to reset() or restart(), so it is cumulative. The lap() member function does not stop the timer, nor does it change the internal state of the timer.

stop() member function

The stop() member function is called to stop the timer from counting and to update the internal timer count. The internal timer count is the total amount of time that the timer has been running since the last call to reset() or restart() so it is cumulative.

reset() member function

The reset() member function resets the internal state of the timer to a zero count and no overflow. This should be called to zero the state of the timer before timing a piece of code. Note that the reset operation is performed every time that a new instance of one of the timer classes is created.

restart() member function

The restart() member function simply resets the timers internal state to a zero count and begins timing.

count() member function

The count() member function returns the current timer count, which will be in fractions of a second. You can use the resolution() member function to determine how many seconds there are in a count so you can convert it to a meaningful value. Use this routine if you wish to manipulate and display the count yourself. If the timer has overflowed while it was timing, this member function will return a count of 0xFFFFFFFF (-1 long).

overflow() member function

The overflow() member function will return true if the timer has overflowed while it was counting.

resolution() member function

The resolution() member function returns the number of seconds in a timer count, so you can convert the count returned by the count() member function to a time in seconds (or minutes, or whatever). The value returned is a floating point number, which simplifies the conversion process.

operator << () friend function

This a convenience function that outputs a formatted string to a C++ output stream that represents the value of the internal timer count in seconds. The string represents the time to the best accuracy possible with the timer being used.

Appendix C: Developing for Maximum Compatibility

This section contains information relating to developing application software with maximum compatibility in mind, without sacrificing performance or features. Although the VBE standard defines how the specification should work, there are many different flavors of hardware out in the field. It is very important that you design your application with the following special cases in mind so that you application will run on the widest variety of hardware possible.

Note that many of the issues in this section are only related to directly programming for the VBE 1.2/2.0 interfaces. If you are doing all your development with the native MGL API, SciTech MGL insulates you from many of these issues. However some issues such as being aware of the different types of hardware configurations that will be out there (such as hardware that cannot do page flipping) affect MGL applications as well.

Provide for Solid Backwards Compatibility

If you are developing your application to take advantage of the latest VBE 2.0 standards, you should ensure that you all provide a good set of compatibility fallbacks for your application. There will be cases in the field where your customer may not be able to get a proper VBE 2.0 driver running on their system, and may not be able to get even a VBE 1.2 driver working properly. Hence you should always provide support for at least a standard VGA mode if possible (Mode 13h or ModeX will suffice) or VBE 1.2 support. If you are developing an application that runs in only SuperVGA modes (640x480 and above) then you should at least ensure that your application runs properly on systems with only VBE 1.2 drivers installed.

Although the performance will not be nearly as great with VBE 1.2, a customer is less likely to be raving mad when they call your tech support lines if the game at least runs. Once they have the game running and wish to get more performance, they will spend more time seeking out higher performance drivers, or will eventually upgrade their graphics card.

Don’t Assume all SVGA Low Res Modes are Available

Also note that on some systems, high performance low resolution graphics modes are not always available, so you should not develop your game to rely on the presence of these modes. On some systems modes below 512x384 are not available, so the only available low resolution modes may be the standard VGA Mode 13h and ModeX modes. Hence if you wish to use low resolution, high performance graphics modes you should always check to see if the mode are available, and provide options for the user to select other modes that may be available (on some systems 200/240 line modes are not available, but 400/480 line modes are).

Note that systems that do not support high performance SVGA low resolution modes are few and far between (less than 5% of the installed base), but you should ensure that your code is ready to handle situations where the exact modes that you want are not available.

NOTE: SciTech MGL provides support for Standard VGA Mode 13h and ModeX modes as well as all the VBE 2.0 SuperVGA low resolution modes. If you are planning on using these modes, you should also plan on supported Mode13h or ModeX modes as a fallback measure for cases where these low resolution modes may not be available.

Develop for the Future with Scalability

An important criteria for developing a successful application is to attempt to obtain maximum performance across a variety of target hardware systems. You should develop your applications to be as scalable as possible, both in terms of the resolutions and color depths that are supported. If you can get your game to run in 320x200x256 linear framebuffer mode, this will probably provide the absolute maximum performance and compatibility in the field. However customers with high end systems will be wanting to run your games at higher resolutions and color depths if possible. Hence you should also develop your games to be fully scalable in terms of resolutions and color depths if possible. Even though the performance may not be so great at 640x480x256 on present day systems, a year from the time that your game is released it may well be possible to support this mode with enough speed to run your game.

If you are developing a 3D game that relies heavily on texture mapping and detailed 3D worlds, you should consider developing the game with multiple levels of detail for the world and the textures. This will allow customers

with lower performance machines (like 486/66 VLB systems) to be able to tune the details of the game down to increase performance. Customers with high performance systems or with systems that will ship after your game has been completed can crank up the details and resolution to get a richer game playing experience.

Include an Option for Rendering to a System Buffer

Cards Affected:

Diamond Viper series (Weitek P9000 and P9100)

One of the main reasons for having an option to render to a system buffer is for compatibility. It fixes two problems: 1.) Some cards cannot double buffer, so you will only get one page of video RAM when you query the card. For example, cards based on the Weitek P9x000 chips (like the Diamond Viper) only support a single VBE buffer in many modes. In order to make your software compatible with the Diamond Viper, you need to have an option to render into system memory. 2.) It will allow you to support cards that may not have enough video memory to properly double buffer in the modes that you need. This gives you a fall back so the user with less than the required RAM can still run your application.

Redistributable Components

Subject to the terms and conditions of the SciTech Software License Agreement, in addition to any Redistribution Rights granted therein, you are hereby granted a non-exclusive, royalty-free right to reproduce and distribute the Components specified below provided that (a) is distributed as part of and only with your software product; (b) you not suppress, alter or remove proprietary copyright notices contained therein; and (c) you indemnify, hold harmless and defend SciTech Software and it's suppliers from and against any claims or lawsuits, including attorney's fees, that arise or result from the use or distribution of your software product.

The redistributable MGL components are:

MGLLT.DLL (c) 1996-98 SciTech Software, Inc.
MGLFX.DLL (c) 1996-98 SciTech Software, Inc.
MGLLTW.DLL (c) 1996-98 SciTech Software, Inc.
MGLFXW.DLL (c) 1996-98 SciTech Software, Inc.
MGLGM.DLL (c) 1997-98 SciTech Software, Inc.
MGLGLUT.DLL (c) 1997-98 SciTech Software, Inc.
ZTIMER.DLL (c) 1997-98 SciTech Software, Inc.
SGIGL.DLL (c) 1997 Silicon Graphics Inc.
SGIGLU.DLL (c) 1997 Silicon Graphics Inc.
OPENGL95.DLL (c) 1996 Microsoft Corporation.
GLU95.DLL (c) 1996 Microsoft Corporation.
MESAGL.DLL (c) 1997 Brian Paul.
All font files in the SCITECH\FONTS directory, copyrighted by the various companies listed in the copyright headers in the font files.
All cursor files in the SCITECH\CURSORS directory (c) 1996-98 SciTech Software.

The redistributable WinDirect components are:

WDIR16.DLL (c) 1995-98 SciTech Software.
WDIR32.DLL (c) 1995-98 SciTech Software.
DVA.386 (c) 1995 Microsoft Corporation.

Bank Switching

Due to limitations of the PC architecture, application software can only access graphics memory in 64K banks when running in real mode. When application software needs to modify the image being displayed, it must switch to the bank of memory on the graphics card that contains the part of the image that it wants to change. Each graphic chip implements bank switching in a different way, so VESA created the VBE Core standard to provide a common way to do bank switching. Newer graphics chips fix this problem by implementing a "linear frame buffer" mode. (see also: Linear Frame Buffer)

BIOS

Acronym for Basic Input Output System. This is the low level code that makes the graphics card start up and operate correctly. It is normally stored in a non-volatile Read Only Memory (ROM) chip on the graphics board and it can be upgraded or supplemented to provide additional functionality with a program like SciTech Display Doctor or a TSR from the graphics card supplier.

BitBlt

Abbreviation for Bit Block Transfer. It means moving a block of pixels from one area of memory to another. Since it is the most common function used in graphical applications and operating systems, it is the primary hardware function available in graphics accelerators. Much of the performance gained in graphics accelerators is a result of a hardware BitBlt function.

BPP

Acronym for Bits Per Pixel. It signifies how many colors can be displayed in a particular graphics mode. 4BPP=16 colors, 8BPP=256 colors, 15BPP=32,768 colors, 16BPP=64,536 colors, 24BPP=16.7 million colors. (see also: Color Depth)

Clipping

Clipping is used to limit the drawing of graphics primitives to a specified area and stopping them from being drawn into unwanted regions. In SciTech MGL, all primitives are clipped to a rectangle. Clipping is useful for implementing viewports and windowing systems to restrict graphics output to specific portions of the display screen.

Clock Chip

Every graphics board has a clock chip that allows it to output varying frequencies to the display. Clock chips are necessary to create each graphics mode and vary refresh rates. Each clock chip must be programmed differently. There are several types of clocks used in typical PC graphics cards. Early graphics cards used "discrete" clocks, which means that there was a physical component on the graphics board for each frequency required by the graphics card. Those were replaced by "mask programmable" clock chips that supported several different frequencies on one component. Most of today's modern graphics cards use "fully programmable" clock generators so they allow a wide selection of graphics modes and refresh rates and are much more flexible. It is now common for clock chip technology to be included inside the main graphics chip instead of as a separate

component.

Color Depth
This refers to how many colors are being displayed. The higher the color depth, the greater number of colors. The more colors, the more memory is required on the graphics card. Generally, the higher the color depth, the more realistic the image displayed and the more processing power is required to manipulate the image. (see also: BPP)

CRTC
Acronym for Cathode Ray Tube Controller. A standalone or integrated chip that generates signals (creates the frequencies necessary for an image to be displayed at a certain resolution) necessary to drive the Cathode Ray Tube (CRT).

DAC
Acronym for Digital to Analog Converter. This is the chip that converts the digital signals in the graphics chip to the analog signals that a standard super VGA monitor requires. The DAC chip also generates all of the colors for a graphics chip. Generally, the better the DAC, the more colors you get.

Device Context
A region where graphics are drawn. It may be on the graphics card in video memory, exist in offscreen video memory or be in a system memory buffer.

DirectDraw
The graphical API part of Microsoft's DirectX. It provides more direct access to the video hardware and memory than is normally available under standard Windows GDI functions.

DirectX
A family of API's designed by Microsoft for Windows 95 and Windows NT 4.0 based games. They allow more direct access to the hardware than would normally be available under the standard Windows GDI functions.

Discrete Clock
see: Clock Chip.

Double Buffering
This is a programming technique that capitalizes the fact that most graphics cards have more memory on them then is actually used to display an image. This extra, or "off screen" memory is used to begin generating the next image so that the user can instantly see the next frame when it is completed. This way the user does not have to watch each frame be rendered by the computer. This technique is used extensively in computer games to produce smooth animation.

DPMI
An acronym for DOS Protected Mode Interface. An interface that allows multiple protected mode applications to run in the one system and share important resources like memory and interrupt handlers.

8bit DAC
An 8bit DAC allows the color palette to be selected from a range of 16.7 million colors rather than the usual 256k colors available in 6bit DAC mode. The 8bit DAC allows the

256 color modes to display a full range of 256 grayscales, while the 6-bit mode only allows a selection of 64 grayscales.

Firmware

This is low-level code that sits between application software and hardware. It is generally stored in non-volatile ROM, but it can be updated with programs from disk, such as SciTech Display Doctor. In a graphics card, firmware is know as a VGA ROM BIOS.

Frame Buffer

This is the memory where a computer image is stored. This memory is usually located on the graphics card and is completely separate from the computer system's main memory. When you are looking at an image displayed on a computer screen, you are viewing the actual image stored in the frame buffer.

GDI

An acronym for Graphics Device Interface. It is the set of functions from within Windows that perform all graphics output. GDI is very restrictive in the types of primitives that it supports, and as such complex graphics applications may run slowly using GDI drawing functions.

Glyph

A small monochrome Bitmap or a character from a graphical font.

GUI

An acronym for Graphical User Interface. It provides support for windowed operations, such as scrolling, maximizing and minimizing.

Hardware Cursor

A mouse cursor drawn with hardware. It is more stable because the application does not have to attempt to draw the pointer on top of the image that is being generated, but the icon is usually monochromatic.

Horizontal Sync Polarity

see: Sync Polarity.

Interlaced

Interlacing is a technique whereby the monitor displays every even line of pixels and then goes back and displays the odd line of pixels in the next pass. Interlacing can result in significant flicker perception and most users prefer a non-interlaced display for this reason. Interlacing does enable higher resolution display on many monitors and it is useful for stereo glasses, where the image needs to be split between the left and right eye to produce a 3D effect.

Linear Frame Buffer

This is a high performance replacement for bank switching whereby all of the graphics memory can be accessed in one contiguous block of memory. In order to use linear frame buffer mode, a graphics chip must support it, must be running VBE 2.0 or other specialized driver and must be running in protected mode. Using a linear frame buffer requires protected mode access because the computer maps the graphics memory into the system memory address space and in order to do this, it must have access to more than the 640K that it would get in "real mode."

LSB
An acronym for Least Significant Bit. In rounding it is the bit that would make the least error if it were dropped. (i.e. the 2 in 1,000,002)

Mask
A mask is a numerical (usually binary) pattern that covers or controls some other type of data. A mask can prevent certain bits from being modified, colors from changing or provide the pattern for an overlay. There is a bit to bit correlation between items in the mask and the data to which the mask is applied.

MSB
An acronym for Most Significant Bit. In rounding it is the bit that would make the most error if it were dropped. (i.e. the 1 in 1,000,002)

Multibuffering
Takes the double buffering concept one step further and allows three or more hidden display images. Multibuffering is used to increase the frame rate in applications by allowing the application to generate images at the maximum rate possible. (see also: Double Buffering)

Non-Interlaced
see: Interlaced.

Off Screen Memory
Graphics cards have a dedicated memory buffer separate from the main system memory. Typically, there is more graphics memory available than is required to actually display an image. This is known as off screen memory and it is used by high-performance applications as a place to store the next image to be displayed or other images that require frequent access. (see also: Frame Buffer)

Packed Pixel
The bits for the image are stored contiguously in memory (packed together) rather than being grouped in planes. Older style EGA / VGA graphics modes are planar and complicated to program, while newer style SuperVGA modes for 8+ bits per pixel color depths are always packed pixel formats and very easy to program.

Page Flipping
(see also: Double Buffering)

PCI
Acronym for Peripheral Component Interconnect. A local bus specification, developed by Intel, that allows high bandwidth peripherals such as graphics cards to run at maximum performance.

Pixels
A pixel is an individual dot of light that the graphics card can turn on or off or change the color. A monitor can display many thousands of pixels. The more pixels, the more defined a computer image becomes. Generally, as the number of pixels displayed increases, the performance of the computer decreases, thus many computer games run in lower resolutions with a higher (more realistic) color depth.

Primitive

A primitive is the smallest component that can be used to build larger complex objects. In SciTech MGL graphics primitives form the building blocks of all graphics output, and includes things such as pixels, lines, rectangle, ellipses and polygons. All other complicated shapes and objects can be broken down into a number of smaller graphics primitives, and those primitives can be used to drawn the objects on the graphics screen.

Protected Mode

Normally a PC can access 640K of RAM. By going into protected mode, software has full access to all of the memory in a computer system. The problem with protected mode is that when running in protected mode, the software no longer has access to some of the computer's resources. This is why VBE 2.0 was created; software can jump down to "real mode" get the information from VBE and switch back to protected mode. Before VBE 2.0, an application had to do that each time that it wanted to access the graphics card. With VBE 2.0, it only has to do it at initialization time, so VBE 2.0 is much faster then earlier versions of VBE.

Real Mode

This is the opposite of protected mode. In this mode of operation, a software application can only access 640K of RAM. If an application runs in real mode, it can only access the graphics card using bank switching. In order to access VBE, an application must be in real mode, therefore protected mode applications will switch to real mode, access VBE, then switch back to protected mode to continue executing. With VBE 2.0, this only has to happen when the application starts up, not every time it accesses graphics memory.

Refresh Rate

An image on a display that appears constant is actually repainting (refreshing) many times per second. The refresh rate is measured in how many times the screen is updated per second, or hertz (Hz). For instance, a typical display refreshes the screen 60 times per second, or 60Hz. The reason that a display image appears constant is that the graphics card is repainting the screen faster than your eye can perceive. If the screen does not refresh fast enough, a person will perceive flicker and may experience headaches. There are many other factors that contribute to the perception of flicker including: ambient lighting, screen size, display brightness (and other screen adjustments) and if the image being displayed has a bright background (like many Windows applications). There is no "best" refresh rate, but it is generally agreed that refresh rates between 70 and 90Hz are in the best range for most people in most computer applications. Refresh rates lower than 70Hz can result in eye strain and headaches, higher refresh rates can result in decreased graphics performance and little or no ergonomic benefit.

Resolution

This refers to the number of pixels that are displayed on the screen. For instance, 1024x768 means that 1024 pixels are displayed across and 768 pixels are displayed down for a total of 786,432 pixels.

ROM

Acronym for Read Only Memory. This is a chip where the BIOS is stored on a graphics card.

Software Cursor

A cursor that is created using software instead of hardware. It can be multicolored but it

may flicker if the image that it is displayed on is rapidly changing.

SVGA

Acronym for Super VGA. The original IBM Video Graphics Array, or VGA card had limited ability to display graphics. Quickly, other manufacturers enhanced the original design of IBM's VGA, while still maintaining compatibility with IBM's original VGA card. These enhanced cards have come to be known collectively as "Super VGA" cards.

Sync Polarity

Polarity, horizontal and vertical, are the signals that the monitor uses to identify a particular mode. They were defined when IBM first introduced the VGA and can be sent via either the horizontal or vertical sync lines in a standard VGA cable. Generally, they are only useful for modes that are less than 640x480 in size, but they can be useful to allow monitors to differentiate between two different modes.

Thunk

This is when an application changes from protected mode back to real mode. Thunking will slow down an application considerably. By minimizing thunking, VBE 2.0 will dramatically increase the performance of a graphics application.

TSR

Acronym for Terminate and Stay Resident program. This is a program that runs, loads itself into memory and then allows the user to run other programs. TSR's have gotten a bad name because many early applications tried to be TSR's and they conflicted with many other applications. Low level drivers like UniVBE or mouse drivers are also TSR's. Since they operate at a very low level, they almost never conflict with applications.

Tuple

A set of three values. Tuple is usually used in the context of color values such as RGB and HSV (i.e. an RGB tuple is the set of three components that make up the RGB color; Red, Green and Blue).

UniVBE

The Universal VESA BIOS Extension 2.0 driver that comes with the SciTech Display Doctor suite. UniVBE enables much of the functionality of SciTech Display Doctor. UniVBE is a trademark of SciTech Software

Vertical Retrace

The point at which the gun drawing the screen travels from the bottom back to the top of the display in order to start drawing the next page. When the application is creating pages faster than the monitor can display them, flickering and noise may occur. By enabling Wait for Vertical Retrace, the next page is not displayed until the application receives a signal that the vertical retrace has occurred and the hardware is ready to accept the next page.

Vertical Sync Polarity

see: Sync Polarity.

VBE

VESA BIOS Extension. This is an extension of the BIOS defined in the original IBM VGA card. VESA added new functionality to the BIOS to provide software developers access to

new features that have been added to SVGA cards. There are several VBE modules that address new features that have been added over the years.

VBE/AF

VESA BIOS Extension / Acceleration Functions. This is a proposal within the VESA Software Standards Committee to standardize common acceleration functions available on most hardware today. Some of the functions supported in the standard are access to hardware cursors, Bit Block Transfers (BitBlt), off screen sprites, hardware panning and drawing. This proposal has the potential to dramatically increase the baseline performance of games, operating systems and other applications.

VBE/PM

VESA BIOS Extension / Power Management - This is a module of VBE that allows applications to issue standard calls to a SVGA card to power down the monitor. VBE / PM allows operating system and application vendors a standard interface to control the monitor. Without VBE / PM a screen saver would be required with special software for every graphic card on the market.

VESA

The Video Electronics Standards Association is an organization that was formed in 1989 to standardize graphics and display hardware and device interfaces. It has over 200 member companies from the display, graphics chip, graphics board, system and software companies. They can be reached at: Video Electronics Standards Association, 2150 North First Street, Suite 440, San Jose, CA 95131. (408) 435-0333, (408) 435-8225 FAX

Viewport

The currently active drawing region of the a device context (like the graphics display). A full graphics display may be 640x480 pixels, while you can have a 320x480 pixel viewport in the center of the screen which would restrict all MGL output to the smaller 320x480 rectangle. When a viewport is active, the logical (0,0) coordinate is mapped to the upper left corner of the viewport. This allows code to be written to draw an object and then, the object can be moved around on the screen simply by changing the position of the currently active viewport.

Virtual Linear Frame Buffer

A virtual linear frame buffer uses the 386+ processors 'memory management unit' (MMU) to emulate a linear frame buffer in software on a standard banked framebuffer device. When the drawing application draws into an area that falls outside of the currently active bank (or 64Kb region of video memory that is currently mapped in by the graphics hardware), the processor raises a page fault exception which is handled by SciTech MGL to automatically re-map the new region of video memory for the graphics card and make it active for drawing.

VL-Bus

Acronym for VESA Local Bus. A local bus specification, developed by VESA, that allows high bandwidth peripherals such as graphics cards to run at maximum performance.

WinDirect

A SciTech API that allows direct access to the hardware and video memory for applications running in Windows 3.x or 95.

WinG
An API developed by MS to allow you to Blt efficiently in Windows 3.*x*.

XY Coordinate System
The Cartesian Coordinate system that is used to define where primitives are drawn on the screen. In SciTech MGL the origin is in the upper left hand corner and the positive directions are down and to the right. This is different than the standard Cartesian Coordinate system where the origin is in the lower left and the positive directions are up and to the right.

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z