SILICON GRAPHICS

PIPELINE

Volume II, Number 1 Fall 1985

IRIS 2400 Turbo Option

The Turbo Option upgrade for the IRIS 2400 provides a number of computing, floating point, and graphics performance enhancements:

- 68020 CPU
- Faster memory and memory access
- New floating-point processor board
- New Extent file system
- New optimized 68020 compilers

Upward compatibility
The IRIS 2400 Turbo Option software is compatible with IRIS 2400 release GL2-W2.3. Programs that run on an IRIS 2400 need only be recompiled to run on the Turbo.

New CPU and memory
The IRIS 2400 Turbo Option upgrade replaces the 68010 CPU with a 68020 CPU. Faster memory boards are also included. With the complimentary software upgrade, this simple field modification provides a two to three-fold increase over the computing performance of the IRIS 2400 workstation.

New CPU features:
- 16 MHz Motorola 68020 CPU
- 64 Mbytes virtual address space
- Real-time clock with battery backup
- Vectored interrupts
- Fast Geometry Engine[1] (GE) port (32-bit)
- Fast FPA interface (32-bit)

Fast memory features:
- 2 or 4 Mbyte boards, up to 16 Mbytes
- 250 ns cycle time
16 Mbyte/second bandwidth
- Full 32-bit data path
- Fast parity checking and error reporting

Faster file system
In order to take advantage of our new hardware features, new system software has been developed. The Extent file system offers a significant improvement in file handling performance over the System V file system.

Under the System V file system, data blocks are stored almost randomly on the disk. In reading or writing a file, the System V file system typically performs a seek for each block of recorded data. In contrast to the System V file system, the Extent file system stores groups of blocks (extents) contiguously, minimizing

the number of disk seeks required to access a file.

While offering these increases in performance, the Extent file system is program-compatible with the System V file system. The user interface is identical, and existing programs will work without change. Read/write operations and the use of *stdio* are identical. If you are using *stdio*, you need only recompile your program to improve performance. If you are using read/write, you can take full advantage of the new file system by increasing your buffer size to at least 4K.

Optimized compilers
New FORTRAN and Pascal compilers generate optimized 68020 instructions. Compile times are approximately half of non-Turbo compile times.

The new FORTRAN compiler generates object files compatible with those produced by the C and Pascal compilers. C, Pascal, and FORTRAN can now use common libraries.

In addition to these features, the new compilers also generate in-line code to support our new floating-point accelerator. Overall FORTRAN floating-point performance is ten times faster with the new compiler than with the IRIS 2400 compiler.

Greater floating-point performance
The IRIS 2400 Turbo without the floating-point accelerator option provides two to three times the floating-point power of the 68010 processor. The new floating-point accelerator has been designed to boost IRIS floating-point performance by a factor of 10. The floating-point accelerator option consists of a single field-installable board. All that is needed to use the floating-point accelerator is a compiler flag.

Floating-point accelerator features:
- 32-bit read/write operations
- Full 64-bit arithmetic
- Overlap and debugging modes
- Single- and double-precision to integer conversions

Floating-point performance
Against standard Whetstone and LINPACK benchmarks, the IRIS 2400 Turbo with the floating-point accelerator out-performs a VAX[2] 11/780 with floating-point accelerator by as much as 80%. With the floating-point accelerator, the IRIS 2400 Turbo performs the Whetstone benchmark at 2050K Whetstones. The LINPACK (coded BLAS) executes at .55 MFLOPS.

Turbo Option upgrade issues
- The non-Turbo floating-point board is not compatible with an IRIS 2400 with the Turbo Option.
- The new floating-point accelerator board is not compatible with an IRIS 2400 without the Turbo Option.
- Standard IRIS 2400 memory is not compatible with the new Turbo Option memory.
- Factory installation is available at no charge. Field Installation is available for a charge, and can be performed only by a Silicon Graphics Representative. It takes about four hours to install the upgrade, including time to back up the user's disk. However, it is recommended that users back up their own disk before the installation.

[1]Geometry Engine is a trademark of Silicon Graphics, Inc.
[2]VAX is a trademark of Digital Equipment Corporation.

New Geometry Engines

8MHz Geometry Engines are standard on all IRIS 2300s, 2400s, and 2500s shipped after August 1, 1985. On the IRIS 2400 Turbo, the new Geometry Engines perform 85700 3D 32-bit floating-point or 101700 3D 16-bit integer transformations per second. For the standard IRIS 2400, the Geometry Engines perform 69000 3D 32-bit floating-point or 93000 3D 16-bit integer transformations per second.

A simple hardware upgrade is available for series 2000 workstations shipped before August 1. This upgrade replaces the 6MHz Geometry Engines with 8MHz Geometry Engines. To order the upgrade, contact your Sales Representative.

Software Release 2.3

The main features of the latest software release for IRIS 1000 and 2000 series workstations are summarized here. This release is distributed automatically to maintenance customers, in-warranty customers, OEM customers, Geometry Partners, and those with a specific major need for 2.3 functionality. Other customers who would like to order the release should contact the Hotline. Please note that this release is required for source code compatibility with the Turbo Option.

IRIS 2000 series workstations

New peripheral options
- Stereo optic viewer
- Digitizer tablet
- Half-inch tape drives (PE 1600 bpi or GCR 6250 bpi)
- Genlockable RS-170A or Genlockable European Video Standard
- IBM Geometry Link with 3270 Emulation

Graphics
- Expanded IRIS window manager demos
- Faster algorithm for drawing small polygons
- Graphics bug fixes

Systems and languages
- More reliable serial communications resolving problems with RS-232 printers, *cu*, and connections to IRIS terminals
- Block mode serial driver allowing communication rates up to 19200 baud
- Floppy formatter program for workstations with floppy disks
- Enhanced and stable *dbx*
- Symbolic links from 4.2 BSD UNIX[1]
- New calling convention allowing FORTRAN to call C directly as well as through wrappers
- FORTRAN character strings up to 4096 characters

IRIS 1000 series workstations

- Graphics bug fixes
- More reliable serial communications resolving problems with RS-232 printers, *cu*, and connections to IRIS terminals
- Block mode serial driver allowing communication rates up to 19200 baud
- Floppy formatter program for workstations with floppy disks
- Enhanced and stable *dbx*
- Symbolic links from 4.2 BSD UNIX
- New calling convention allowing FORTRAN to call C directly as well as through wrappers

[1] UNIX is a trademark of Bell Laboratories.

Geometry Partners Program

Silicon Graphics has established a third-party software program called the Geometry Partners Program. An increasing number of software applications programs for the IRIS are available through this program. To date, over 30 vendors representing more than 60 distinct application programs are enrolled in this program. The types of applications available span Silicon Graphics' major markets:

- Mechanical computer-aided engineering
- Electrical computer-aided engineering
- Molecular modeling
- Seismic and geophysical modeling
- Graphic arts and animation
- Engineering support tools
- Simulation

Through the Geometry Partners Program, Silicon Graphics provides systems and technical support to the vendors to ensure that each program in our catalog actually runs on our equipment and can use the powerful features of the IRIS.

A catalog describing each application program is available from your Silicon Graphics Sales Representative. The catalog includes a contact name, configuration information, and data on the unique capabilities of each program.

Tips for speeding up IRIS graphics programs in C

- Declare heavily used variables as register variables.
- Arrays can be slow, especially 2D arrays. Here are some general rules for speeding them up:
  - Use single-dimensional arrays rather than multi-dimensional arrays. (**struct** is more efficient than arrays.)
  - Use pointers instead of arrays where possible.
  - Use **int** instead of **short** for array indexing.

For example, the classic 2D array can be slow:

```
for (i = 0; i <4; i++)
    for (j = 0; j <4; j++)
        mat1[j] [i] = mat2 [i] [j];
```

Using pointers is faster:

```
pl = mat1;
pl2 = mat2;

for (i = 0; i <4; i++) {
    for(pl=mat1+i;j=0;j<4;j++){
        *pl = *pl2++;
        pl += 4;
    }
}
```

Using hard-coded array indices is even faster.

- Use integers instead of floats whenever possible.
- Eliminate function calls within critical-loops.
- Where possible, use shifts (<< or >>) to handle divides or multiplies, particularly in heavily used functions.
- Use parallelism: start raster subsystem processing, then start 68000 or Geometry Engine processing. The following code gives an 18-millisecond window after the **clear()** for the processor to perform other tasks:

```
color(BLACK);
clear();
gflush();/*1000 series only*/
multmatrix or build object, etc.
```

- Use **rectf** to clear small areas.
- When editing, use **objreplace** whenever possible. **objinsert** and **objdelete** are much slower.
- Reserve bit planes for static objects or background and paint only once for a given buffer. For example, in the *flight* program, the brown and orange gauges are drawn only twice (once for each buffer) during setup. Those planes are then masked off and "forgotten."
- Don't use **lookat** or **polarview** in immediate mode. They require 20 to 30 floating operations per call. You can often simulate the effect of moving the eye location by actually moving the world around it. (The *arch* demo does this.) If you must use **lookat** and **polarview**, put them inside a viewing object and limit the frequency of their execution.
- Immediate mode is several (two to four) times slower than display list mode. (See article below.) Anything that doesn't change, but is called more than once, should be put into a display list, e.g., **perspective**.
- Rather than using circles, which are implemented as 80-point polygons, use **polys** or **polfs** with fewer than 80 points.
- Use **pushattributes** and **popattributes** sparingly.

Fast immediate vs. display list mode

The Graphics Library 2 (GL2) provides an additional access mode that can improve performance in some applications. In the Graphics Library I (GL1), commands can be executed in *immediate mode* or *display list mode*. In immediate mode, direct calls are made to the Graphics Library subroutines. In display list mode, Graphics Library commands are compiled into display lists and executed as objects.

GL2 provides three modes for accessing the Graphics Library: *immediate mode*, *display list mode*, and *fast immediate mode*. Immediate mode in GL2 is ten times faster than it was in GL1. The addition of *fast immediate mode* in GL2 makes simple graphics commands even faster. Fast immediate mode uses in-line C macros, eliminating subroutine call overhead. In general, the simple com-

mands (**move**, **draw**, **poly**, **rect**, **pnt**, **color**, **writemask**, **translate**, **rotate**, **scale**) have fast immediate mode versions.

GL1 was designed to be used with display lists, but because of hardware improvements, display lists are often not necessary in GL2.

In GL2, use fast immediate mode when high performance is critical, for example in inner loops of a program. Fast immediate mode is also useful for designing a customized display list.

Fast immediate mode macros are written in C and do not work in FORTRAN or Pascal. However, C routines can be called from FORTRAN or Pascal to do fast drawing.

Here is a simple fast immediate mode example. (Note: this is not necessarily an *advantageous* use of fast immediate mode, since registers are set up for only one routine. A more realistic example would call several fast immediate mode routines, and would benefit more from setting up registers only once.)

```
#include "fastimmed.h"
#include "gl.h"

main()
{
    ginit();
    fastrect();
    gexit();
}

fastrect()
{
    im_setup;
    /* to assure address */
    /* register assignment */
    color(BLACK);
    clear();
    color(RED);
    im_rects(100, 100, 300, 300);
    /*immediate mode version of*/
    /*"rects" command; actually*/
    /*draws the rectangle*/
}
```

Note that even this simple example must call a subroutine. This is because **ginit()** must appear first in a graphics program, and **im_setup** should appear first in every routine that uses fast immediate mode. **im_setup** initializes two address registers in the 68010, leaving two for the user. Check assembler listings to make sure address registers have been assigned.

Fast immediate mode macros are accessible through the file *fastimmed.h*, listed below. All the include files are in */usr/include/gl2*.

```
#define UNIX
#define PM2
#define DC4
#define UC4
#include "gl2/globals.h"
#include "gl2/gltypes.h"
#include "gl2/immed.h"
#include "gl2/imsetup.h"
```

Fast immediate mode macros work differently from subroutines in C. The expressions are evaluated in order left to right. Thus, if **array[]** is a list of short integer coordinates ordered x, y, z, the routine:

```
im_setup;  /* here to be sure */
           /* it gets registers */
register int ptr = 0;

while(ptr<3000)
      im_pnts(array[ptr++],
      array[ptr++],array[ptr++]);
```


draws 1000 points correctly in fast immediate mode.

In some cases, it is not advantageous to use fast immediate mode:

- Since performance would not improve when subroutine call time is small compared to command execution time, some commands have no fast immediate mode equivalents. There are no fast immediate mode equivalents for **clear**, **circle**, **arc**, **ortho**, **polarview**, and **lookat**. To see which commands have fast immediate mode versions, check the file *fastimmed.h*.
- It is probably not worthwhile to use fast immediate mode for menus and backgrounds, because backgrounds take a long time to draw, and menus have to react at only human speeds.

Display lists can still be used in GL2. They run at almost the same speed as fast immediate mode, since the overhead is only two machine language instructions per graphics command. The code executed during display list traversal is generated by the same immediate mode macros, plus two extra instructions to get to the next display list command. Registers are set up only once.

If code developed under GL1 is already structured to use display lists, it may be easier to continue using them under GL2. The GL1 and GL2 display list formats are nearly compatible. Some points to consider when using display lists:

- The standard display list format is not always the best one for a particular application. The memory management scheme is only an average strategy that works well for some applications and less well for others. The standard format stores only geometric information. Extra data, such as electrical information in a circuit design system or type of material in a mechanical CAD system, must be stored in a second display list. However, you can design your own display list format. Use fast immediate mode in the routines that traverse the display list.
- Display lists can be slow to edit.
- During development, it is easy to use immediate mode to get a program running correctly and then convert the time-critical parts to fast immediate mode.

VMS XNS and graphics software available

Distribution of the VMS XNS N2.3 release started in early fall.[1] N2.3 is the XNS communications software for VAX/VMS 4.0. Both terminals and workstations are supported. The N2.3 release supports file transfer and remote login, and has an applications library callable from C. If you ordered XNS but haven't received your copy, contact the Geometry Hotline.

The GL2 Remote FORTRAN Graphics Library for VMS is also available. The release is shipped automatically to customers who have already ordered it. If you have not ordered it but would like to, please contact your Sales Representative.

[1] VMS is a trademark of Digital Equipment Corporation. XNS is a trademark of Xerox Corporation.

Exploring Julia sets: floating point performance, z-buffering, depth cueing

Julia sets, and the Mandelbrot set in particular, have received attention in *The Fractal Geometry of Nature* by Mandelbrot, *Frontiers of Chaos* by Peitgen and Richter, and the *Computer Recreations* column of the August 1985 issue of *Scientific American*. The Mandelbrot set shown in Plate 1 and discussed here is based on iterating $x_{k+1} = x_k^2 + c$ in the complex plane. By choosing different values of c and iterating the function until x_k begins to diverge, the number of iterations, k, may be used to color or give height to the starting points c_{real}, $c_{imaginary}$. The C program shown below prints 1600 points of a Mandelbrot set. Plate 1 was created by displaying a Mandelbrot subset using c_{real} and $c_{imaginary}$ as x and y coordinates and coloring $pixel_{xy}$ black if $k_{diverge}$ was odd and white if even. Plate 2 was created by using $k_{diverge}$ to control altitude and gray scale color.

Floating-point performance

The evaluation of points in the Mandelbrot set is floating-point intensive. Calculating a large number of points in the Mandelbrot set may be used to compare the floating-point performance of various processors. The following program was run on several different processors; the relative performances are noted in the table below.

```
#define D 0.1
#define C 4.0
main ()
{
register long float z, zi;
register long float c, ci;
register long float temp, z2, zi2;
register int i;

for (c=-2.0; c<=2.0; c+=D)
 for (ci=-2.0; ci<=2.0; ci+=D){
  z = 0;
  zi = 0;
  for (i = 0; i < 200; i++) {
   z2 = z*z;
   zi2 = zi*zi;
   if (z2 + zi2 > C) break;
   temp = z2 - zi2 + c;
   zi = 2.0*z*zi + ci;
   z = temp;
   }
#ifndef TIMING
 printf("%lg %lg %d\n", c, ci, i);
#endif
  }
}
```

Julia set timings

processor	hardware floating point	user	system	wall clock
IRIS 2400	no	41.4u	2.9s	0:44
IRIS 2400	yes	12.8u	0.1s	0:13
IRIS 2400 Turbo	no	16.5u	0.4s	0:18
IRIS 2400 Turbo	yes	1.8u	0.1s	0:02
VAX 11/780	yes	1.8u	0.1s	0:02
VAX 11/750	yes	9.5u	0.7s	0:20

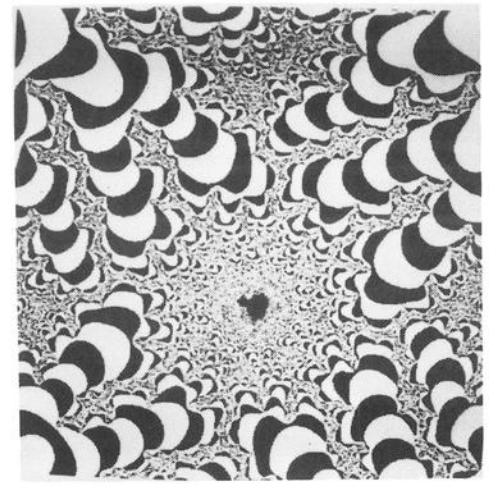

Plate 1

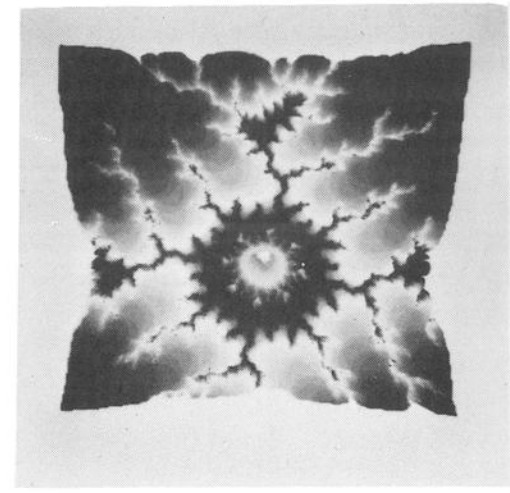

Plate 2

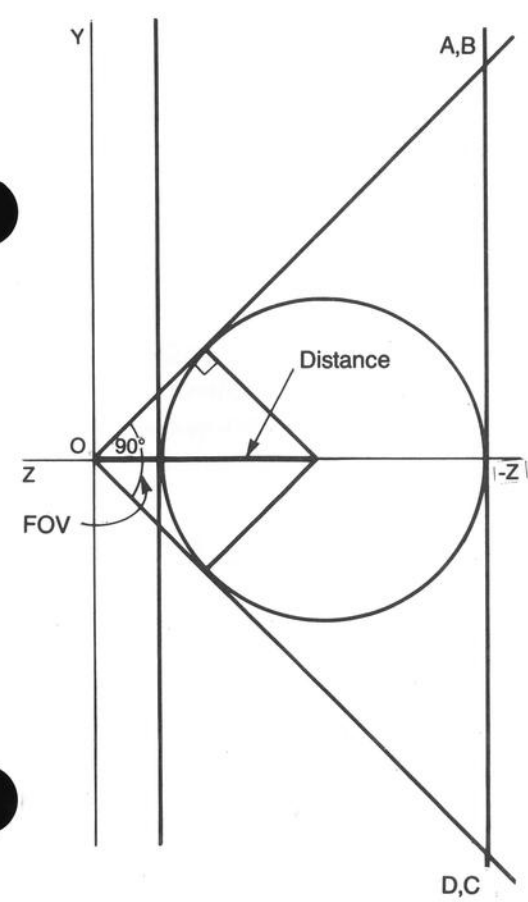

Figure 1a: Profile view

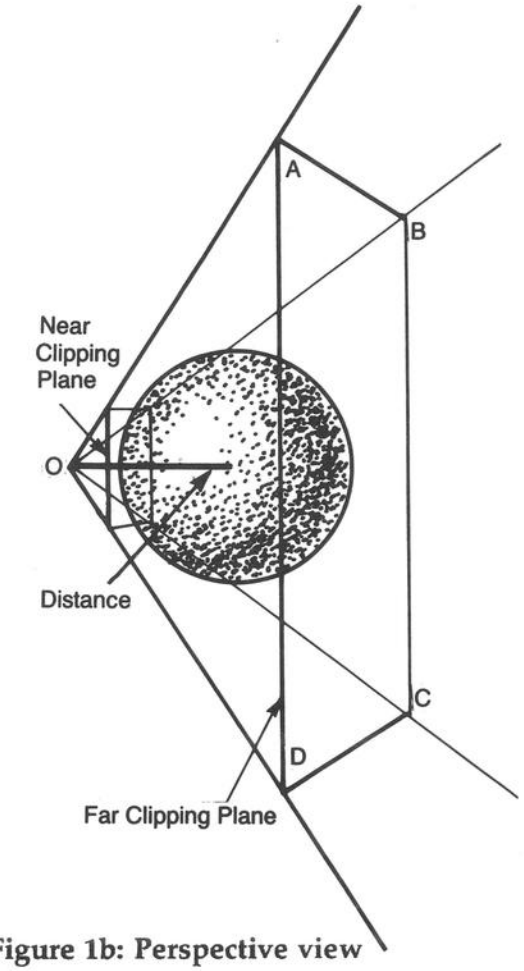

Figure 1b: Perspective view

Enclosing sphere and viewing frustum

Z-buffering and depth-cueing

The problem of effectively displaying a data set is common to all of computer graphics. The three dimensional Mandelbrot set shown in Plate 2 was displayed using depth-cued points to interactively orient the data set and z-buffered polygons to produce the final image.

Because depth-cueing and z-buffering use 16-bit screen space *z* values, the view of the data set must be carefully defined in order to get the most effective final image. The best use of the 16 bits of *z* value is made by choosing a view that tightly sandwiches the object to be viewed between the near and far clipping planes. In specifying an orthographic or perspective projection, a set of view parameters must be calculated or chosen to satisfy viewing requirements:

- static vs. dynamic
- clipped
- region occupied by data set
- aspect ratio
- field of view

For example, if the desired view is to be an orthographic projection of the data set with no portions clipped, no specific aspect ratio, and no movement or reorientation, the appropriate **ortho** is trivial to calculate. The **ortho** should define a volume that just encloses the data set. The data may be thought of as being enclosed in a box whose corners are the *min/max* values of the points defining the object. The following function finds a set of *max/min* values in an array of points.

```
#define MAX(x,y) (x>y?x:y)
#define MIN(x,y) (x<y?x:y)
find_maxmin(points,n,max,min)
float points[][3];
int n;
float max[3], min[3];
{
int i,j;
/* initialize max min values */
for (i = 0; i < 3; i++)
 min[i] = max[i] = points[0][i];
for (j = 1; j < n; j++)
 for (i = 0; i < 3; i++) {
  max[i] = MAX(max[i],
               points[j][i]);
  min[i] = MIN(min[i],
               points[j][i]);
 }
}
```

Given the *min/max* values of a data set, the command

```
ortho(min[0],max[0],
      min[1],max[1],
      min[2],max[2]);
```

sets up the appropriate static orthographic view. If an aspect ratio of 1.0 is desired, the following code segment sets up the correct orthographic projection:

```
minval = MIN(min[0], min[1]);
maxval = MAX(max[0], max[1]);
ortho(minval, maxval,
    minval, maxval,
    min[2], max[2]);
```

As a second example, a perspective view of the data allows a user to interactively rotate the data set about its center. The desired view should never clip, should have an aspect ratio of 1.0, and should have a 90° field of view. To prevent clipping, the object to be viewed must be centered in the field of view at a distance from the view point such that it is not clipped in any rotational orientation. The distance is calculated from the radius of the sphere that completely encloses the object, and the field of view. The sphere diameter may be calculated by computing the maximum of all point-to-point distances in the object. For efficiency reasons, the diagonal of the bounding box is used as an approximation. The following function sets up and draws the object in the appropriate view:

```
#include "math.h"
#define FOV 90
#define ASPECT 1.0
#define PI 3.14159265
#define RADCONV (PI/180.0)
#define DEPTHCUE 0
#define ZBUFFER 1
#define SQ(x) ((x)*(x))

displayobj(points,n,max,min)
float points[][3];
int n;
float max[3], min[3];
{
float center[3],
  distance, radius;
int i;
short drawmode;
drawmode = DEPTHCUE;
depthcue(1);
shaderange(128,255,
    0xc000,0x3fff);
radius  = SQ(max[0]-min[0]);
radius += SQ(max[1]-min[1]);
radius += SQ(max[2]-min[2]);
radius = sqrt(radius)/2.0;
distance = radius/
    (sin((FOV/2.0)*RADCONV));
perspective(FOV*10,
    ASPECT,
    distance-radius,
    distance+radius);
translate(0.0, 0.0, -distance);
for (i = 0; i < 3; i++)
 center[i]=
     (max[i]+min[i])/2.0;
pushmatrix();
translate(center[0],
        center[1],
       -center[2]);
color(BLACK);
clear();
drawpoints(points,n);
popmatrix();
while (1) {
 pushmatrix();
 /* do Q handling: rotations,
 z-buffer/depth cueing mode
 select, and redraw
 interaction */
 drawmode =
       dointeraction(drawmode);
 translate(center[0],
        center[1],
       -center[2]);
 color(BLACK);
 clear();
 if (drawmode == ZBUFFER) {
  zclear();
  drawpolys(points,n);
 } else
  drawpoints(points,n);
 popmatrix();
}
}
```

The value of **distance** is calculated from the geometric relationship illustrated in Figure 1. The **perspective** command positions the near and far clipping planes tangent to the front and back sides of the enclosing sphere, maximizing the range of screen space *z* values.

The following main program initializes the graphics system by requesting a port from the window

manager, sets the z scalars, queues the desired input devices, and calls **displayobj** to display the object. The z scalars are set to range from **0xc000** to **0x3fff** to avoid possible overflows that can occur in 16-bit arithmetic. The function **dointeraction** reads the input queue until it is empty, taking action corresponding to each input event.

```
#include "gl.h"
#include "device.h"
extern float points[][3];
extern int n;
main ()
{
float max[3],min[3];
getport("julia");
setdepth(0xc000, 0x3fff);
find_maxmin(points,n,
   max,min);
qdevice(REDRAW);
qdevice(MOUSE3);
qdevice(MOUSEX);
qdevice(MOUSEY);
displayobj(points,n,max,min);
gexit();
}

dointeraction(drawmode)
short drawmode;
{
short val, dev;
static short xrot=0,yrot=0;
do
 switch (dev=qread(&val)){
 case REDRAW :
  reshapeviewport();
  break;
 case MOUSE3:
  if (!val) break;
  drawmode = !drawmode;
  if (drawmode == DEPTHCUE) {
   zbuffer(0);
   depthcue(1);
   shaderange(128,255,
```



```
          0xc000,0x3fff);
  } else {
   depthcue(0);
   zbuffer(1);
  }
  break;
 case MOUSEX:
  xrot = val;
  break;
 case MOUSEY:
  yrot = val;
  break;
 }
while (qtest());
rotate(xrot, 'x');
rotate(yrot, 'y');
return(drawmode);
}
```

Double buffered window manager programs

Programs written for the window manager in double buffer mode must handle situations that are not encountered in single buffer programs. This article first reviews general window manager programming, and then discusses special issues raised by double buffered window manager programs.

Reviewing window manager programming

A window manager program has an initialization stage. During this stage, the characteristics of the window, such as aspect ratio and size limits, are defined. The event queue is initialized to recognize tokens **REDRAW** and **INPUTCHANGE** that will be encountered while running in the window manager. For a double buffer program, the display is set to double buffer mode during the initialization stage. This code shows a sample initialization sequence:

```
#include "gl.h"
#include "device.h"

main()
{
    keepaspect(3,2);
    getport("sample");

    qdevice(INPUTCHANGE);
    qdevice(REDRAW);
    qdevice(ESCKEY);

    doublebuffer();
    gconfig();
```

The main section of the program, which follows initialization, is a loop that continually processes devices. Each pass through the main loop has two parts. The first part of the main section is itself a loop that processes the event queue. Tokens in the event queue are read and processed until the event queue is emptied. The second part of the main section handles input from polled (not queued) devices. A sample main loop is shown below:

```
while(TRUE) {
    while (qtest()) {
    /* process queued tokens */
        dev = qread(&val);
        switch(dev) {
        case ESCKEY:
        /* exit program with ESC */
            exit(0);
            break;
        case INPUTCHANGE:
            attached = val;
            if (!attached) {
/* be sure both buffers have */
/* same scene when unattached */
                frontbuffer(TRUE);
                drawscene(dx, dy);
                frontbuffer(FALSE);
```

```
            {
            break;
        case REDRAW:
            reshapeviewport();
            frontbuffer(TRUE);
            drawscene(dx, dy);
            frontbuffer(FALSE);
            break;
        default:
            break;
        }  /* end switch(dev) */
        if (!attached)
/* swap buffers if not attached*/
            while (!qtest())
                swapbuffers();
    }  /* end while (qtest()) */

    if (attached) {
    /* process polled devices */
        oldx = x; oldy = y;
        x = getvaluator(MOUSEX);
        y = getvaluator(MOUSEY);
    /* redraw scene if LEFTMOUSE */
    /* button is pressed */
        if (getbutton(LEFTMOUSE)) {
            dx = dx + (x - oldx);
            dy = dy + (y - oldy);
            drawscene(dx, dy);
        }
    }  /* end if (attached) */
    swapbuffers();
}  /* end while (TRUE) */
```

The double buffer difference

Special situations are encountered in double buffer window manager programs that are not encountered in single buffer mode. Every double buffer program must continually issue **swapbuffers()** calls. If the program blocks for input without constantly swapping buffers, buffer swapping does not take place in other windows. The **swapbuffers()** call is made each time through the main loop. Most important, even when the double buffered graphics

program is detached and merely idling, buffers must be swapped. (When the user cannot interact directly with a program, the program is said to be detached.) The code below, taken from the larger code sample shown previously, shows proper handling of an unattached program:

```
if (!attached)
/* swap buffers if not attached */
    while (!qtest())
        swapbuffers();
```

Since the program is swapping buffers when idle, the information in both buffers must be the same. To ensure this, whenever the program becomes detached, both buffers should be updated with the same image. When the **INPUTCHANGE** token is received with a data value of 0, the graphics program is detached. Both buffers can be enabled for writing with the graphics library call, **frontbuffer(TRUE)**. Then the scene should be drawn into both buffers. The code below, also taken from the earlier code sample, shows how to detach gracefully from a double buffered program:

```
case INPUTCHANGE:
    attached = val;
    if (!attached) {
/* be sure both buffers have */
/* same scene when unattached */
        frontbuffer(TRUE);
        drawscene(dx, dy);
        frontbuffer(FALSE);
    }
    break;
```


Release 2.3 bugs and fixes

shaderange

shaderange has two more parameters in release 2.3, but its FORTRAN wrapper has not been updated. Until it is fixed in the next release, **shaderange** should not be called from FORTRAN. An interim solution is to write your own wrapper, as documented in Appendix C, *Generating C/FORTRAN Interface Routines*, in the *IRIS Workstation Guide, Series 2000*.

Note that the *lint* program cannot accommodate **shaderange**'s four parameters.

linewidth and depth-cueing

Depth-cue mode is supported only for lines of line width 1. Fat lines are not supported in depth-cue mode.

gclear

The program **gclear** does not work correctly in the window manager. It initializes a textport that is in conflict with the *mex* console textport. In *mex*, use the mouse to change the size and shape of the textport. Exit the window manager before executing **gclear**, or run a program that executes **ginit** with **noport** set. **gclear** resets colors to their original values, clears the screen to black, and resets the textport to the middle of the screen. Using **ginit** and **noport** resets the colors to their original values. The following program can also be used to clear the screen to black within the window manager:

```
main() {
    ginit();
    color(BLACK);
    clear();
}
```

Stereo viewing

For stereo viewing, when the left image is shown on the screen, the left eye should be open. However, the stereo viewer box has only one input. The only command that can be given is to change eyes, but there is no way to query the hardware about which eye is open. The solution to this is to toggle the switch on the stereo viewer box to change eyes. Another possibility is to write software to check whether an image looks right in the viewer and to toggle the switch if not; once the switch is set correctly, it will stay in sync for the rest of the session.

FORTRAN local variables

With the 2.3 release of FORTRAN, local variables in subroutines and functions are allocated on the stack, rather than statically, enabling recursion. This means that local data that remain valid between calls to a subroutine in release 2.2 are not valid between calls in the 2.3 release. Data can be declared as 'static' by use of the **SAVE** statement (see the *FORTRAN Reference Manual* in the *UNIX Programmer's Manual, Volume 2B*). Data declared with the **SAVE** statement remain valid between calls to the routine.

Hardware floating point

The following program behaves incorrectly when it is compiled for the non-Turbo floating-point hardware:

```
main() {
    float *rf0,*rf1;
    *rf0++ += *rf1++;
}
```

This will be fixed in Release 2.4.

We've moved

On September 27, Silicon Graphics moved to a larger facility at Shoreline Park in Mountain View. Our new address is:

2011 Stierlin Road
Mountain View, CA 94043

Our phone number remains the same: 415/960-1980.

Silicon Graphics Pipeline

Editor: Marcia Allen
Technical illustrator: Anna Szabados
Masthead designer: Hulda Nelson
Production assistant: Ken Allen
Photographer: Henry Moreton
Contributors:
Chris Blumenthal, Greg Boyd, Tom Davis, Susan Ellis, Robin Florentine, Paul Haeberli, Steve Johnston, Mark Libby, Susan Luttner, Zsuzsanna Molnar, Henry Moreton, Bob Pearson, Pramod Rustagi, Diane Wilford, Mason Woo

The *Silicon Graphics Pipeline* is published by Silicon Graphics, Inc. as a service to our customers. Please circulate it only within your site. For additional copies, write: Silicon Graphics, Inc., 2011 Stierlin Road, Mountain View, CA 94043, Attn: Marketing Communications.

Document number: 5001-094-003-I

Geometry Hotline and customer training classes

The purpose of the Geometry Hotline is to improve your productivity as you use Silicon Graphics products. Our staff answers technical questions about the functionality of the system software and hardware, handles hardware failure reports, and dispatches Field Service Representatives. Sales calls are referred to a local Sales Representative; application-oriented questions are referred to local Systems Engineers for follow-up.

A Silicon Graphics technical representative is available on the Geometry Hotline Monday through Friday from 7 a.m. to 5:30 p.m. and Saturday and Sunday from 9 a.m. to 5 p.m. Pacific Standard Time. Outside these hours and on holidays, the Hotline uses an answering service to take messages, which are answered at the beginning of the next working day.

To help our representative serve you better when you call the Hotline, please have the following information ready:

- your system's serial number
- your name
- your phone number

It is especially important to remember your serial number; we need it to log your call. We have a computerized call log system that is connected to our customer database. This provides us with complete information about your system and configuration.

Your call is best handled if you can tell us:

- what you are trying to do
- how you attempted to do it
- the results you got, including complete error messages.

Call the Hotline when you are having a problem getting something to work. Have your serial number ready. After hours and on holidays leave a message with the Hotline answering service. Your call will be answered early the next working day. On weekends, a technical representative will respond as soon as possible.

If you are new to graphics, or if you have just purchased your first Silicon Graphics product, we recommend our training classes, listed below. For UNIX training, we recommend AT&T's UNIX System V classes. Call AT&T at 800/221-1647 for information.

Training classes
Silicon Graphics offers two customer training classes, Graphics I and Maintenance I.

- **Graphics I, 5 days**
This course is designed to raise applications programmers to a comfortable level of proficiency with the IRIS graphics system through extensive classroom instruction and hands-on training. The entire Graphics Library command structure is presented in the context of practical application.

- **Maintenance I, 5 days**
This technical course offers hands-on training in the installation and maintenance of IRIS 2000 series terminals and workstations. The course provides the knowledge required to install and maintain IRIS products.