SILICON GRAPHICS

PIPELINE

Volume III, Number 1 Spring 1986

2.4 and 3.4 releases

Releases 2.4 and 3.4 for IRIS series 2000 and 3000 systems will be available in May. These will be the last releases sent automatically to all customers. To receive these releases, you need only complete the software form enclosed with your IRIS User Survey (see page 20).

To receive future releases automatically, your system must be in the warranty period or covered by a support agreement (see page 3). Releases will be available to others for a fee. The 2.3 release was the last general release for IRIS 1000 systems.

Releases 2.4 and 3.4 include these features:

- Shared memory
- Many Graphics Library bug fixes, especially to the window manager
- Window manager enhancements, including a pop-up menu package and multiple windows and processes for IRIS models 2300 and above
- Support for rational cubic splines
- A timer pseudo-device to introduce timer events into the event queue at specified intervals
- New light pen peripheral option
- The source-level debugger *dbx* for FORTRAN as well as C, and a new tutorial for novice users
- Pascal Graphics Library for IRIS workstations
- The capability to intermix C, FORTRAN, and Pascal routines without wrappers in an IRIS workstation program
- The ability to trap floating-point exceptions extended to Pascal

Release 2.4 for IRIS 2400 and 2500 workstations includes features already incorporated in the 3000 series:

- Extent File System, Silicon Graphics' fast file system
- Color printer support
- *.o* files (standard UNIX binaries) instead of *.j* files (special FORTRAN binaries) for FORTRAN and Pascal to allow use of UNIX utilities
- Many bug fixes to utilities

Because of the level of increased functionality provided in this release, users must recompile all programs. The conversion to the Extent File System on IRIS 2300, 2400, and 2500 systems requires users to back up all personal files onto tape or another machine since disks are wiped clean during the upgrade procedure.

[1] UNIX is a trademark of AT&T Bell Laboratories.

Silicon Graphics Pipeline

Editor: Marcia Allen
Editorial assistant: Diane Wilford
Photographer: Henry Moreton
Masthead designer: Hulda Nelson
Page designer: Robin Bristow
Contributors and reviewers:

Chris Blumenthal	**Susan Luttner**
Greg Boyd	**Gail Madison**
Al Casarez	**Zsuzsanna Molnar**
Dennis Daniels	**Henry Moreton**
Tom Davis	**Beat Poltera**
Susan Ellis	**Amy Smith**

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: 007-7086-010

3.3.1 update release

A 3.3.1 update package has been sent to all customers with series 2000 Turbo systems. If you have not received your package, please call Susan Wilson at (415) 960-1980, ext. 636. Be ready to provide this information:

- The model number of your system
- The serial number of your system

- All software options you have purchased
- Your correct address

IRIS series 3000

Three new 68020-based products were added to the IRIS family in February. These products offer two to three times the computing performance of the 68010-based IRIS series 2000 products. The IRIS series 3000 systems with FPA run up to twice as fast as a VAX[1] 11/780 with FPA.

The IRIS series 3000 product line includes the IRIS 3010 terminal and two workstations, the IRIS 3020 and the IRIS 3030. The new series offers these features:

- 68020 CPU and expandable memory (from 2 to 16 MB for the workstations; from 2 to 12 MB for the terminal)
- Optional floating-point accelerator
- 125-nanosecond Geometry Engines[2]
- 19-inch tilt-and-swivel monitor (optional 15-inch monitor)
- Optical mouse
- Extent File System

The Extent File System is four to six times faster than the standard System V file system.

The IRIS 3010 terminal runs a terminal emulation program over the UNIX operating system. A 20 MB hard disk and 1 MB floppy disk drive are standard on the system. The IRIS 3010 functions as either a host-dependent terminal or as an execute-only workstation.

A 72 MB hard disk is standard on the IRIS 3020. The IRIS 3030 features a 170 MB hard disk with ESDI interface and the Storager II disk controller. A second disk is available on both workstations. The IRIS 3030 offers six to eight times the performance of the standard System V file system.

The 68010-based IRIS series 2000 products are available at advantageous pricing for those who need 3D graphics, but who do not require the additional computing power.

The IRIS series 3000 systems are software compatible with series 2000 systems. You need only recompile when moving between the 68010- and 68020-based systems.

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

Customer support programs

Field Engineering is expanding the support options available to customers. In addition to the hardware support programs currently offered, we are developing a program to allow users to purchase a Software Maintenance Agreement or a Software Subscription Service. Benefits of the Software Maintenance Agreement include:

- telephone support for quick response to software questions and problems,
- new releases and updates,
- new or improved documentation.

The Software Subscription Service is designed to keep your software and documentation up to date, but does not include access to the Hotline.

Beginning July 1, 1986, Field Engineering's Hotline support will be available only to certain customers:

- those with a system in the warranty period,
- those with a hardware or software support agreement,
- those seeking Time and Material support.

Additional details on the two new software support programs will be in the mail to you shortly.

Current support programs
The remainder of this article summarizes these current support programs:

- Product Warranty
- Field Installation
- Basic Support Program
- Full Support Program
- Extended Warranty

Only highlights of the programs are presented here; contact your Silicon Graphics Sales or Service Representative for a more complete explanation of our support programs.

Product Warranty
All customers receive a 90-day warranty covering the cost to repair or replace any part returned to Silicon Graphics that proves to be defective. Return-to-customer freight charges

are also covered, but installation and on-site service are not. This program includes emergency parts exchange to resolve hardware failures, and hardware and software updates during the warranty period.

Field Installation
Under this program, Silicon Graphics provides installation, on-site service, and full support for its products during the warranty period. System problems are resolved by a combination of home-office and on-site support; defective parts are replaced on-site at no additional charge.

Basic Support Program
This program provides materials for product maintenance, but the customer is liable for time and travel charges for on-site service, if required. Hardware and software updates are provided. This program is recommended for customers who have the technical expertise necessary to provide the labor required for product maintenance. It is strongly recommended that customer representatives attend the Silicon Graphics Product Maintenance class before beginning this program.

Full Support Program
The Full Support Program provides a comprehensive product support plan. Initial equipment installation and full remedial and preventive maintenance is provided on-site during the standard hours of coverage (8 a.m. to 5 p.m. Monday through Friday excluding holidays); extended coverage is available for an additional charge when approved by Silicon Graphics Field Engineering. Hardware and software updates are provided and installed, where applicable (most software updates are easily installable by the customer). Please note that the equipment must be maintainable in its current condition. This is automatically assumed if the equipment was covered by any hardware support program immediately before the Full Support Agreement is signed. If not, Silicon Graphics must certify the equipment; there may be a charge for certification. If you select the Full Support Program when you initially purchase equipment, the program covers fifteen months, with twelve months billed.

Extended Warranty
The Extended Warranty is available only when you initially purchase equipment. This enhanced warranty includes all Full Support Program benefits, but for only the first twelve months after installation.

Support program pricing
The following customers are eligible for Support Program discounts:

- Customers who have purchased the Extended Warranty, the Full Support Program, or the Field Installation and who have more than ten units at a single location.

- Customers who pay for the full yearly term of the Basic or Full Support Program in advance.

Additional charges may apply to customers located outside a Silicon Graphics Local Service Area. A Local Service Area is defined as a 100-mile radius in driving miles around the city center in an area where a Silicon Graphics service center has been established.

UNIX manuals

Some of our early customers received comb-bound UNIX manuals, rather than manuals in three-ring binders. If your editions of the *UNIX Programmer's Manual, Volume IA* and *Volume IB* are comb-bound, please leave a message at (415) 960-1020, ext. 950. We'll send you new Volume I manuals in three-ring binders.

Training and consulting

The goal of a recent research project at Silicon Graphics has been to identify methods to improve your productivity as you use your IRIS system. Two methods stand out: learning about 3-D graphics programming and spending hands-on time with an expert programmer. To meet these needs, Silicon Graphics is pleased to announce the formation of a new Education/Consulting Group.

Silicon Graphics provides a full program of scheduled software and hardware training classes throughout the year. The IRIS Graphics I course is designed to give applications programmers a comfortable level of proficiency in the use of the IRIS Graphics Library.[1] All major Graphics Library command structures are presented in the context of practical application through extensive classroom instruction and hands-on projects.

The class features a new on-line learning environment to demonstrate transformations and viewing operations. The Training Group is also available to teach the graphics course at your site. For more information on training, contact Monica Schulze in the Technical Marketing Department.

The goal of the Applications Consulting Group is to shorten your implementation cycle. The Applications Consulting Group uses standard Silicon Graphics software, such as the Graphics Library, to help you:

- design your application code
- optimize your code
- integrate standard multibus boards and devices

For more information, please contact Zsuzsanna Molnar, Technical Marketing Manager.

[1] IRIS Graphics Library is a trademark of Silicon Graphics, Inc.

Bugs and fixes

Unwanted horizontal lines
Unwanted horizontal lines may suddenly appear on the screens of series 2000 and 3000 systems. These lines have a dashed appearance and are spaced about one character apart. They appear because of a Graphics Library bug that corrupts the space for font 0.

This bug will be fixed in Releases 2.4 and 3.4. To eliminate these lines in the meantime, compile the following program with **-Zg** (and also **-DIP2** for 68020-based systems). This program allows you to rewrite the ordinarily unwritable font 0. Without this program, it is necessary to reboot the system to eliminate the lines.

```
#include <gl2/globals.h>
#include <gl2/immed.h>
```

```
main(argc, argv)
int argc;
char **argv;
{
    im_setup;
    register x, i;

    noport();
    getport();
    x = -1;
    im_outfontbase(0);
    im_passthru(18);
    im_outshort(FBCloadmasks);
    im_outshort(0);
    /* pattern 0 address */
    for(i=0; i<8; i++)
        im_outlong(x);
    im_outfontbase
        (gl_wstatep->fontrambase);
    im_freepipe;
    gexit();
}
```

fastimmed.h

The last issue of *Pipeline* (Fall 1985) contained an error in the article on fast immediate mode. The line on page 6, column 1 that reads:

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

should read:

```
#include "gl2/fastimmed.h"
```

sqrt(0.0) on series 2000 systems

With the FPA board used in series 2000 non-Turbo systems (IRIS 2300, 2400, and 2500), *sqrt(0.0)* is not quite zero. This isn't usually a problem, except for the `lookat` command. `lookat` calculates an incorrect viewing matrix in the case where one looks down the *y*-axis. The bug makes the view appear flat.

The command `lookat(0.0, 100.0, 0.0, 0.0, 0.0, 0.0, 0)` should generate the matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 100 & 1 \end{bmatrix}$$

With the IRIS 2300, 2400, and 2500 FPA board and associated software, it generates:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 100 & 1 \end{bmatrix}$$

This problem will be fixed in Release 2.4. In the meantime, avoid calling `lookat` looking directly down the *y* axis; rather, use suitably small numbers for either the *x* or *z* values.

Perspective

The `perspective` command should theoretically report an error if the near clipping plane is zero. `perspective` transforms a truncated viewing pyramid into a unit cube for clipping. If the near clipping plane is at zero, the pyramid is no longer truncated. There is no homogeneous way to transform the five-sided pyramid into a six-sided cube.

However, no error is reported in the Graphics Library. Setting the near clipping plane to zero eliminates both near and far clipping, so everything is visible regardless of its *z* coordinate. If you need far clipping but no near clipping in perspective, set *near* to .00001 (or some other suitably small number).

Z clipping option

The z clipping option is required for most applications that use depth-cueing or z buffering, or that use feedback mode to get *z* values in screen coordinates. These applications require z clipping because of the way the graphics hardware performs transformations.

Points, lines, and polygons are transformed in three steps:

- A matrix multiplication transforms the world coordinates so that the three-dimensional region to be viewed is mapped into a cube whose *x*, *y*, and *z* coordinates are between –1.0 and 1.0.
- The geometry is clipped to eliminate points outside this range.
- The points (now guaranteed to have coordinates between –1.0 and 1.0) are scaled to screen coordinates with an appropriate multiplication and addition.

Systems without *z* clippers clip only in the *x* and *y* directions, so the resulting *z* coordinates may fall outside the range –1.0 to 1.0. *z* values outside this range are not scaled correctly by the hardware. For almost all drawing, the *z* values are meaningless and are discarded by the scan-conversion hardware, so the lack of z clippers has no effect. However, depth-cueing and z buffering require correctly scaled *z* values.

If the visible geometry happens to be transformed so that all *z* values lie between –1.0 and 1.0, the picture will be displayed correctly. Although this range cannot be guaranteed for arbitrary geometry, it can be guaranteed if you know the range of the data. Z buffering and depth-cueing work correctly if you set the near and far planes in the viewing transformations far enough apart so that all geometry lies between them.

Object headers

The use of objects in GL2 entails a small memory overhead. This overhead can normally be ignored, but in some cases it is important to understand its implications. In GL2, every object built by the graphics library with `makeobj()`–`closeobj()` has a small associated object header. The header contains, among other things, pointers to included tag information, the location of the object's contents, and statistics for garbage collection. It also contains one bit that tells whether the object is deleted.

Assuming that objects 2 and 3 are already built, consider the following code sequence:

```
makeobj(1);
callobj(2);
callobj(3);
closeobj();
delobj(2);
callobj(1);
```

The result is the same as if the deleted object 2 were present, but empty. This behavior is possible because the header for object 2 is

still present; in the header, object 2 is marked as deleted.

Even after the object is deleted, the header remains for two reasons:

• An object need not be defined before it is used. Object 1, built in the example above, works even if objects 2 and 3 had never existed. `callobj(2)` makes a header (marked deleted) for object 2 if object 2 does not exist at the time.

• Display list traversal is more efficient if actual addresses of the object headers are contained so no lookup is required. (In GL1, a lookup was done for every `callobj()`; GL2 performs `callobj()` 20 times faster.)

The fact that headers remain after objects have been deleted does, however, present these disadvantages:

• Some space is taken up by the headers of deleted objects. The headers are relatively small (currently 36 bytes), but programs that make lots of objects using `genobj()` may run low on space even if they carefully delete unused objects. Since `genobj()` never returns the same number, a program that uses `genobj()` to create and display an object for each frame may lose 36 bytes 60 times each second.

• Although object headers are relatively small and there may be only a moderate number of them, they may be scattered throughout memory so that the free list becomes badly fragmented. This causes a slowdown in memory allocation. In extreme cases, a program could fail because all the fragments would be too small.

Object headers are reused, so if object 2 is deleted and then created

again with `makeobj(2)`, the same header is used, and no additional memory is lost.

The Silicon Graphics display list implementation is fairly general and easy to use. However, if your application is memory intensive, consider designing a custom display list tailored to it. For more information, see "Fast immediate vs. display list mode", *Pipeline* II, 1 (Fall 1985).

Cartridge tape maintenance

For reliable cartridge tape operation, the tape must be handled, stored, and cleaned properly. If you have had problems with the cartridge tape on your IRIS, please review the guidelines in this article.

Handling and storage

• Avoid exposing the tape cartridge to dirt, moisture, or extreme temperatures (outside the range of 41 to 131 degrees Farenheit or 5 to 45 degrees Celsius). If the tape cartridge's environment temperature changes dramatically, give the cartridge time to acclimatize before use.

• Store the tape cartridge on its edge.

• Place labels on the tape only in the area where the manufacturer's label is affixed.

• Do not open the tape access cover to expose the tape when the cartridge is not in use.

• Do not touch the tape; this can contaminate the magnetic coating.

• Do not attempt to use a damaged cartridge; this can damage the tape drive.

Hand cleaning

Clean the recording head and the tape cleaners two hours after a new tape cartridge has been inserted, and then after every 8 hours of actual use. You will need these supplies:

• Plastic foam swabs (not cotton swabs, which leave fibers that can cause oxide build-up on the read/write head)

• Trichlorotrifluoroethane solvent, also known as Freon T F (not alcohol, which leaves a residue that can cause load failures and data errors)

The right supplies are included in the Cipher Cleaning Kit, part number 960855-001.

To clean the recording head and tape cleaners, follow these steps:

• Wet the swab with the solvent.

• Rub the tape head with the wet swab in the direction of tape motion.

• Rub the tape head with a clean, dry swab in the same direction.

• Clean the roller guides.

Tape retension

Take the tape to the end and then rewind it in these situations:

• if it is a new tape,

• if it has been dropped,

• if it has been in storage,

• after 8 hours of use,

• before using it to make a system backup.

On IRIS series 2000 Turbo and series 3000 workstations, issue the command *mt ret* to perform tape retension. On series 1000 and 2000 workstations with Release 2.3, issue the command *smt ret*.

Density and suppliers

Tape density is certified for 10,000 flux changes per second, 100k per foot. Tape suppliers are 3M/Scotch; Data Electronics, Inc.; and Control Data Corporation.

Technical Publications

The Technical Publications Department at Silicon Graphics is working on several projects to improve the quality of technical manuals.

An introductory guide, *Getting Started with Your IRIS Workstation*, is now available. This pamphlet, designed for programmers with little or no UNIX experience, shows the new user how to issue basic UNIX commands, get started with *vi*, and create and run two simple graphics programs.

New communications manuals that are scheduled to appear soon include *TCP/IP User's Guide* and *IBM Terminal Emulation*. A communications overview document is scheduled for this summer.

Our most recent owner's manuals have been reorganized to include simple instructions for booting systems, as well as disk configuration and other new procedures. Plans are underway to modularize manuals, so that only customers who order an option receive documentation for that option.

We're looking forward to receiving customer input via the IRIS User Survey (see page 20). This survey gives you the opportunity to influence the future development of our technical manuals.

Graftals: trees, weeds, and grasses

The *graftal* as a means for describing plant life was introduced by Alvy Ray Smith in the 1984 SIGGRAPH proceedings.[1] Graftals are based on a class of formal language. This language class is defined by parallel graph grammars called L-systems.[2] Each word in such a language, as defined by a specific graph grammar, describes the branching structure of a plant. These words may then be used to generate renderings of plants. The tree shown in plate 1 is based on a word from a very simple graph grammar; it illustrates the compact descriptive power of graftals.

Parallel graph grammars

Context-free parallel graph grammars, called 0L-systems, are defined by an alphabet, a substitution rule or production for each member of the alphabet, and a start symbol or string of symbols. Context-sensitive grammars have multiple substitution rules. For each member of the alphabet, rules are selected depending on the context in which the symbol appears. The extent of the context classifies the grammar, e.g., a grammar that uses one symbol on either side of the symbol for substitution is called a 2L-system.

Each word in the language of a grammar corresponds to a string generated by applying the substitution rules to the start string or other words in the language. All words are therefore descendants of the start string. When productions are applied to the symbols of a string, they are applied in parallel, producing a new string or word in the language.

The following example illustrates a simple context-free parallel graph grammar with the alphabet {*01[]*}. When this grammar is related to plant structure, {*01*} represent plant stems and limbs, and {*[]*} represent the beginnings and ends of branches, with leaves placed at the ends of branches. The right hand side of the 0 and 1 productions are shown in italics and bold respectively.

alphabet:
{01[]}

start string:
0

productions:
0 → *1[0]0*
1 → **11**
[→ [
] →]

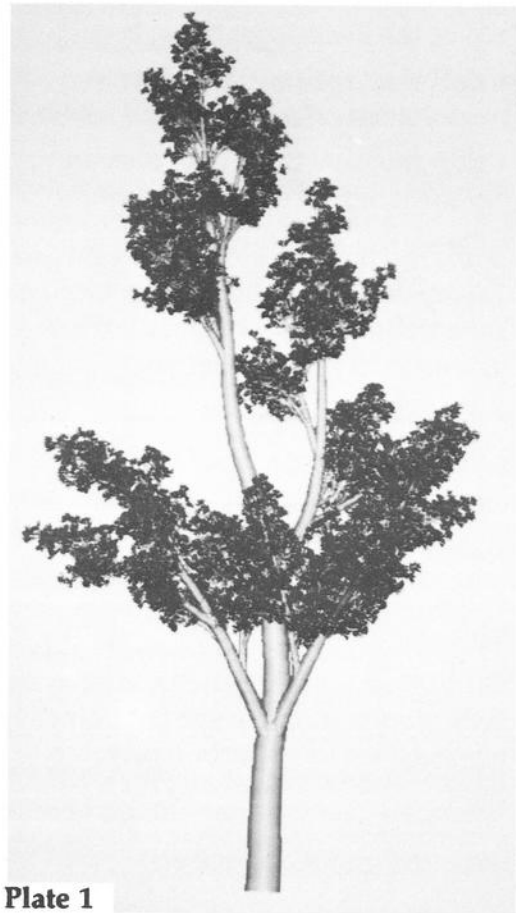


Plate 1

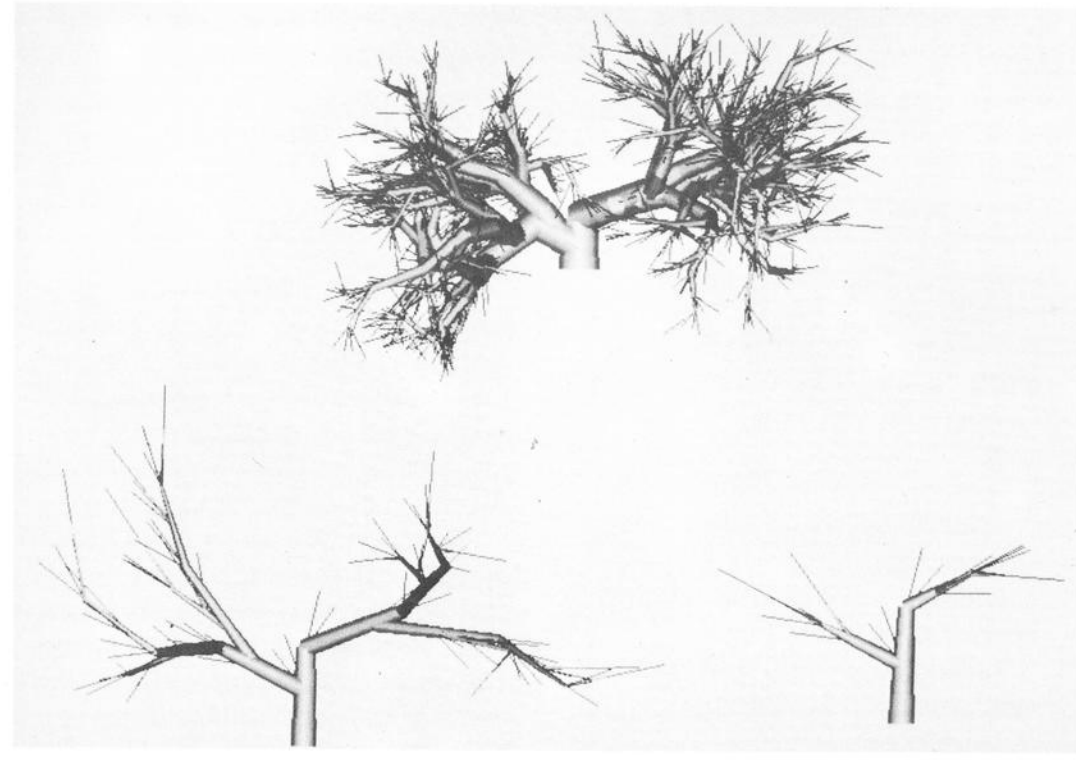


Plate 2

words in the language:

#	word
0	0
1	*1[0]0*
2	**11**[*1[0]0*]*1[0]0*
3	**1111**[**11**[*1[0]0*]*1[0]0*]**11**[*1[0]0*...

Plate 2 illustrates three words from a grammar. The three plants are described by words two, four, and eight from the following grammar:

alphabet:
{01[]}
start string:
0
productions:
0 → *1[[[0]]]1[[[0]]]*
1 → **1**
[→ [
] →]

Word generators

The following C function generates a word in a language from an input word and grammar. The parameters to the function are the input `word`, the `alphabet` of the grammar, the substitution `rules`, and the lengths of the respective substitution strings `rulen`. The function returns a pointer to the newly generated word, `newword`.

```
char *
grow(word,alphabet,rules,rulen)
register char
  /* input word */
  *word,
  /* alphabet of grammar */
  *alphabet,
  /* replacement rules for each symbol
  in alphabet */
  **rules;
  /* length of each replacement string */
register int  *rulen;
{
  register char *newword,*symbol;
```

```
register int i,newlength,length;
char *oldstring;
newword = malloc(1);
newword[0] = 0;
length = 1;
/* loop until end of input word */
while(*word) {
  symbol = alphabet;
  i = 0;
  /* find symbol in alphabet */
  while(*symbol &&(*a != *word)) {
    symbol++;
    i++;
  }
  /* replace with corresponding string
  of symbols */
  newlength = rulen[i] + length;
  newword = (char *)
     realloc(newword,newlength);
  strcpy(newword+length-1,
     rules[i]);
  length = newlength;
  word++;
}
return newword;
}
```

The following program segment prints ten generations of words from the sample grammar shown above:

```
alphabet = "01[]";
rules = (char **)
   malloc(4*sizeof(char *));
rules[0] = "1[0]0";
rulen[0] = strlen(rules[0]);
rules[1] = "11";
rulen[1] = strlen(rules[1]);
rules[2] = "[";
rulen[2] = strlen(rules[2]);
rules[3] = "]";
rulen[3] = strlen(rules[3]);
word = malloc(2);
word[0] = '0';
word[1] = 0;
for (i = 0; i < 10; i++) {
  newword = grow(word,
     alphabet,rules,rulen);
  free(word);
  word = newword;
  printf("%s\n",word);
}
```

Plant rendering

Supplied with words from a language, the program can create realistic images of plants. Programs for rendering plants from strings of symbols could be developed to take many complex factors into account. Factors such as geo- and phototropism typically affect plant growth. Surprisingly realistic images may be created from very simple programs that completely ignore these factors. The program described below allows the user to specify a branch angle range, trunk thickness, and numeric seed used to generate random numbers. Fixed ranges of colors are used for shading leaves, branches, and the trunk. These colors may be altered using the editing tools *cedit* and *interp*. Use *showmap* to display the color map.

Provided with a string representing a word from a plant grammar, a trunk thickness, a branch angle, and a random seed, the rendering program displays the corresponding plant in wire frame or Gouraud shaded form. The drawing is generated by parsing the string, translating and rotating a local coordinate system for every letter encountered.

When "stems" ('0' or '1') are encountered, a move to the origin (0,0,0) is made followed by two random rotations of $\pm 3^\circ$ about x and z. A stem segment length is then looked up in a small table using a random index, and a segment of that length is drawn along the y axis of the local coordinate system.

After the segment is drawn, the local coordinate system is translated by the length of the stem segment along the y axis; this leaves the origin of the coordinate system at the end of the current stem segment.

When the beginning of a branch ('[') is encountered, the current matrix is saved using **pushmatrix**, and a random rotation about the y axis is performed, followed by a rotation about the z axis of the fixed branch angle. This leaves the local coordinate system with the y axis lined up along the branch of the tree; subsequent "stems" will be drawn along the branch.

When the end of a branch (']') is encountered, a leaf of random color is drawn and a **popmatrix** command is issued. The **popmatrix** command returns the local coordinate system to the base of the branch, oriented as before the branch was encountered.

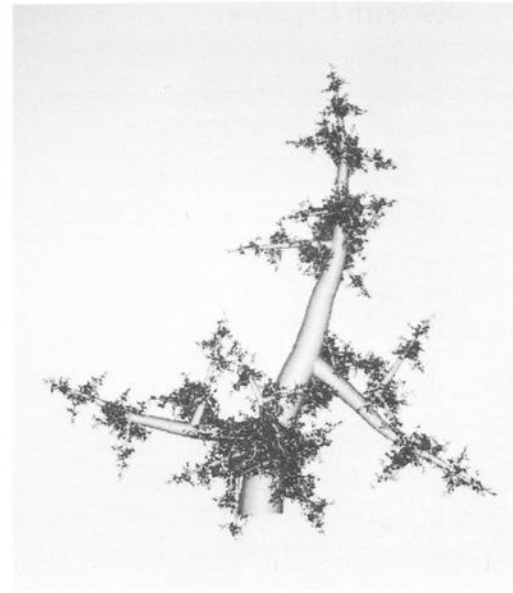


Plate 3

Plate 4

Typically, the user interactively modifies the branch angle and random seed until a structurally satisfactory plant/tree is arrived at. Once the skeletal appearance of the plant is satisfactory, a shaded version may be drawn. The trunk thickness is adjustable and renderings of the same plant with various stem thicknesses may be created. Plates 3 and 4 depict the same plant with differing branch angles and trunk thicknesses.

The thickness parameter controls the diameter of the plant at its base. The stem thickness tapers linearly as the stem segments are closer and closer to the tips of branches. When the shaded plant is being drawn, an exhaustive search is made to find the longest path in stem segments (0s and 1s) to the most distant branch tip (**taper=thickness/distance**). The stem thickness or diameter is then divided by the distance to the most distant branch tip. This value is used as the rate of taper of the plant's stem.

Shading is done by drawing a tapered octagonal prism in feedback mode, and computing the shading of the prism in 3D screen space. The diameters of the prism ends are determined by their distance to the most distant branch tip. The two diameters are `distance*taper` and `(distance-1)*taper`. The shade of each of the prism's polygons is taken as the z component of the polygon's normal vector. This model assumes diffuse lighting with the light at infinity on the z axis. The shades of neighboring polygons are then averaged to yield vertex shades for each polygon. The polygons are then redrawn in shaded z buffer mode.

Since a large `viewport` and `setdepth` range are defined, enough precision is available to do reasonable lighting calculations. The rendering program saves the current viewport using `pushviewport`. `viewport (-1000,1500,-1000,1500)` and `setdepth(-1000,1500)` set up the new viewport. The program then draws the appropriate prism in feedback mode and calculates the lighting based on the 3D screen coordinates. The original viewport is restored by `popviewport`, and the prism is redrawn in shaded form. This process is performed for every stem segment making up a plant.

[1] Smith, Alvy, "Plants, Fractals, and Formal Languages," *Computer Graphics* 18(3), pp. 1-10 (July 1984). SIGGRAPH '84 Proceedings.

[2] Lindenmayer, Aristid, "Mathematical Models for Cellular Interactions in Development, Parts I and II," *Journal of Theoretical Biology* 18, pp. 280-315 (1968).

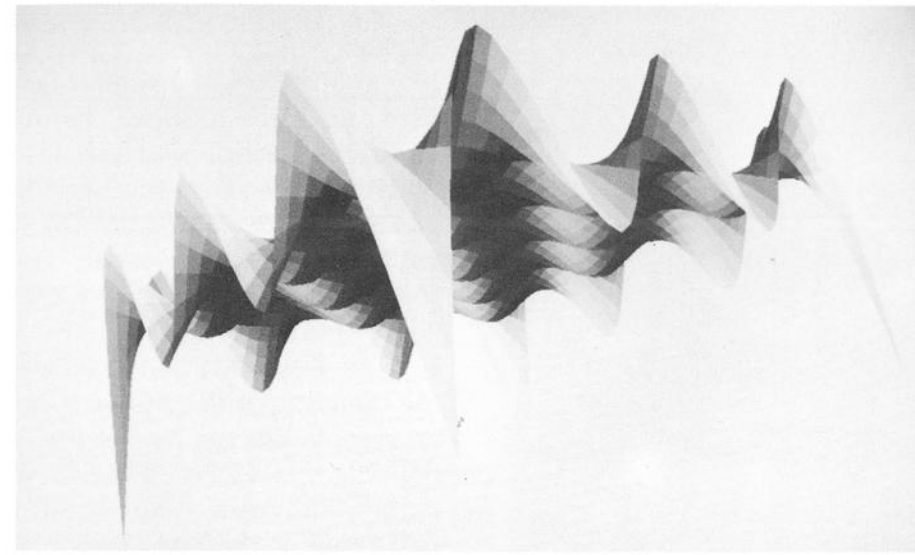
Curve and patch subdivision

[Editor's note: This article is reprinted from Pipeline 1, 2 (Summer 1984), which is now out of print. While most of that issue is now obsolete, this article contains useful information for users of GL2 systems. We hope you find it helpful.]

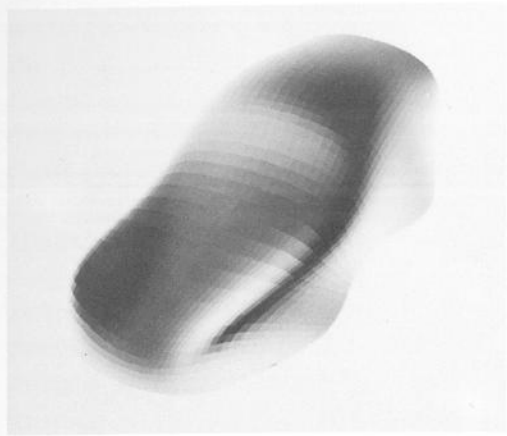
In applications requiring curve and patch intersection, repeated subdivision is frequently used to reduce a curve or patch to a collection of linear approximations, line segments, or quadrilaterals. The intersection operation is then performed on the linear approximations. Subdivision is also useful for creating a faceted rendering of a patch. By breaking a patch into a collection of small polygons and shading each according to its orientation, a computationally cheap, very effective rendering may be produced.

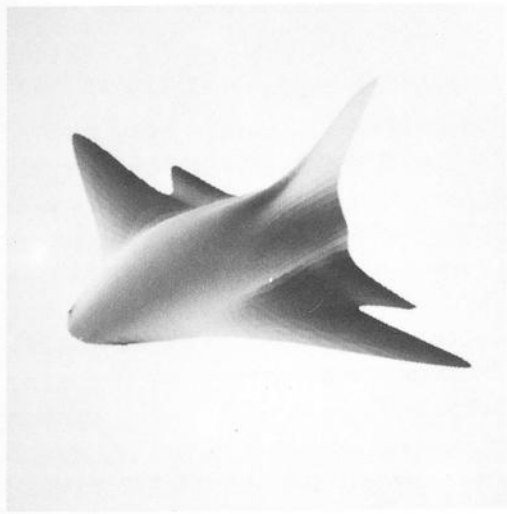
The Geometry Engines may be used to perform subdivision of parametric cubic curves and patches. Subdivision is a technique for deriving the set of four control points (p'_i) defining a curve $C'(s)$ from a parent curve $C(t)$ such that $C'(s) = C(t)$ where $t = 2s$; i.e., $C'(s)$ is the first half of the parent curve $C(t)$. Equation 1 (see page 16) illustrates the matrix form of a parametric cubic curve. Equation 2 follows from the definition of subdivision and equation 1. The relationship of the control points p'_i to the parent curve's control points p_i may be derived as shown in equations 3 and 4.

The value of the three matrices on the right side of equation 4 may be precalculated for any curve basis and subsequently used to do curve subdivision at the cost of a single matrix multiplication. By reversing the order of the original control points, the parameterization of the parent curve is reversed resulting in $C(0) = C_{reversed}(1)$ and $C(1) = C_{reversed}(0)$. If the reparameterized curve is now subdivided in the same fashion, the second half of the original curve is calculated. The following code segment subdivides a curve into its two subcurves, using the reparameterized version of the original curve to calculate the second subcurve.



```
subdividecurve(parent_curve,
     first_half, second_half)
  Matrix   parent_curve,
           /* the curve to
           subdivide */
           first_half,
           /* the new curve
           for 0 to 0.5 */
           second_half;
           /* the new curve
           for 0.5 to 1 */
  {
  Matrix   reversed_parent;
  int      i, j;
  /* save the current matrix */
  pushmatrix();
  /* load the matrix of control
  points */
  loadmatrix(parent_curve);
  /* multiply them by the precal-
  culated subdivision matrix */
  multmatrix(Subdivision_matrix);
  /* retrieve the new control
  points */
  getmatrix(first_half);
  /* reverse the order of the
  points */
  for (i = 0; i < 4; i++)
     (j = 0; j < 4; j++)
        reversed_parent[i][j] =
        parent_curve[3-i][j];
  /* load the matrix of control
  points */
  loadmatrix(reversed_parent);
  /* multiply them by the precal-
  culated subdivision matrix */
  multmatrix(Subdivision_matrix);
  /* retrieve the new control
  points */
  getmatrix(second_half);
   /* restore the matrix saved
  on entry */
  popmatrix();
  }
```





A patch $P(u,v)$ may be subdivided into subpatches in a similar fashion, using the same routine. The matrix of sixteen control points that define a patch may be taken row-wise in groups of four points forming four control curves which when blended form the patch. If each of these curves is subdivided using the above routine, the patch that results from the blending of the *first_half* curves represents the low half of the original patch. Likewise, if the *second_half* curves are blended, the resulting patch will be the high half of the parent patch. If both of these subpatches are subdivided in the other parametric direction by taking the control points *column*-wise, the resulting four patches represent the subpatches of the original parent patch subdivided once in each parametric direction. The halftone image on page 15 was rendered by recursively subdividing a set of 36 patches into 2304 patches which were flat shaded as polygons.

$$C(t) = [t^3\ t^2\ t\ 1][\text{Basis}]\begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix} \quad (1)$$

$$[t^3\ t^2\ t\ 1][\text{Basis}]\begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix} = [(2t)^3\ (2t)^2\ 2t\ 1][\text{Basis}]\begin{bmatrix} p'_0 \\ p'_1 \\ p'_2 \\ p'_3 \end{bmatrix} \quad (2)$$

$$[(2t)^3\ (2t)^2\ 2t\ 1] = [8t^3\ 4t^2\ 2t\ 1] = [t^3\ t^2\ t\ 1]\begin{bmatrix} 8 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3)$$

$$\begin{bmatrix} p'_0 \\ p'_1 \\ p'_2 \\ p'_3 \end{bmatrix} = [\text{Basis}]^{-1}\begin{bmatrix} 1/8 & 0 & 0 & 0 \\ 0 & 1/4 & 0 & 0 \\ 0 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}[\text{Basis}]\begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix} \quad (4)$$

Network connections from within a program

The programs below allow you to make an XNS[1] or TCP/IP connection between IRIS systems from within a program. You can make either type of connection with a makefile, a file that makes the connection, and a file that accepts the connection.

XNS

For an XNS connection, compile the files *xnscon.c* and *xnslis.c* with the *makefile* shown below. This builds the executables *xnscon* and *xnslis*.

Run *xnslis* with a socket number (numbers in the range 100-2000 are recommended) on one machine. Then run *xnscon* with a hostname and a socket number on a second machine. Once a connection is established, everything typed on the connect side appears on the listen side. To terminate the session, type EOF (`CTRL-D`).

For example, at site A, type:

```
xnslis 100
```

From site B, type:

```
xnscon siteA 100
this is a message
```

Back at site A,

```
this is a message
```

appears on your screen.

makefile

```
default: xnscon xnslis

xnscon: xnscon.c
    cc -o xnscon xnscon.c -lxns

xnslis: xnslis.c
    cc -o xnslis xnslis.c -lxns
```

xnscon.c

```
#include <stdio.h>

main(ac,av)
int ac;
char **av;
{
  int sockno,fd,rval,cnt;
  char *host;
  char line[80];
  ac--;

/* process arguments */
  if(ac == 0){
    printf("usage: %s ",*av);
    printf("host socket\n");
    exit(0);
  } else {
    *av++;
    host = *av;
    ac--;*av++;
    if(ac == 0)
      sockno = 100;
    else {
      sockno = atoi(*av);

/* these are good numbers to use */
      if((sockno < 100) ||
        (sockno > 2000)){
        printf("socket(%d)", sockno);
        printf("out of range,");
        printf(" 100 used\n");
        sockno = 100;
      }
  }

/* connect to host */
  if((fd = xnsconnect
    (host,sockno)) == -1){
    printf("xnsconnect of %s ", host);
    printf(" on %d failed\n", sockno);
    exit(0);
  }
```

```
/* now write to the socket */
  printf(" - connected -\n");
/* while there is input, read it */
  while((cnt = read(0,line,80))>0){
/* some data massaging */
    printf("sending <");
    fflush(stdout);
    write(1,line,cnt-1);
    printf("\\n>\n");
    fflush(stdout);
/* write to socket */
    if(write(fd,line,cnt)<0){
      perror("write");
      exit(0);
    }
  }
  printf("DONE\n");
/* clean up */
  close(fd);
  }
}
```

xnslis.c

```
main(ac,av)
int ac;
char **av;
{
  int sockno,fd,cnt;
  char buf[80];
  ac--;
    if(ac == 0){
      printf("usage:%s ",*av);
      printf("socket\n");
      exit(0);
    } else {
      *av++;
      sockno = atoi(*av);
/* these are good numbers to use */
    if((sockno < 100) ||
      (sockno > 2000)){
      printf("socket(%d)", sockno);
      printf("out of range, 100 used\n");
      sockno = 100;
    }
  }
/* listen till someone tries to connect */
  if((fd = xnslisten
    (sockno)) == -1){
    printf("xnslisten on ");
    printf("%d failed\n",sockno);
    exit(0);
  }
  printf(" - connection ");
  printf("established -\n");
/* now read from the socket */
  while((cnt = read(fd,buf,80))>0){
    write(1,buf,cnt);
  }
  printf("DONE\n");
/* clean up */
  close(fd);
}
```

TCP/IP

For a TCP/IP connection, compile the files *accept.c* and *connect.c* with the *makefile* shown below. This builds the executables *accept* and *connect*.

Run *accept* on one machine; then run *connect* on another. Everything you type on the connect side appears on the accept side. To end the session, type EOF (`CTRL-D`).

For example, at site A, type:

```
accept
```

At site B, type:

```
connect siteA
this is a message
```

Back at site A,

```
this is a message
```

appears on your screen.

makefile

```
default: accept connect

accept: accept.c
    cc -o accept accept.c -lsocket

connect: connect.c
    cc -o connect connect.c -lsocket
```

accept.c

```
#include <stdio.h>
#include <sys/types.h>
#include <EXOS/net/misc.h>
#include <EXOS/sys/socket.h>
#include <EXOS/net/in.h>
#include <errno.h>

struct sockaddr_in sin = {AF_INET};
struct sockaddr_in sfrom =
  {AF_INET};

extern int errno;

main(ac,av)
int ac;
char **av;
{
  int fd;
  int cnt;
  char line[80];

/* make socket connection */
/* use legal port */
    sin.sin_port =
    htons(IPPORT_RESERVED + 1);

/*create socket file descriptor*/
  if(!(fd=socket(SOCK_STREAM,
      0,&sin,SO_ACCEPTCONN|
      SO_KEEPALIVE))){
      perror("socket");
      exit(0);
    }

/* wait till someone tries to connect */
    if(accept(fd,&sfrom) < 0){
      perror("accept");
      exit(0);
    }
    printf("- connection ");
    printf("established -\n");
/* once done, while something */
/* to read, read it */
    while((cnt = read(fd,line,80))>0){
/* write what was read to stdout */
      write(1,line,cnt);
    }
    printf("DONE\n");
/* clean up */
    close(fd);
}
```

connect.c

```
#include <stdio.h>
#include <sys/types.h>
#include <EXOS/net/misc.h>
#include <EXOS/sys/socket.h>
#include <EXOS/net/in.h>
#include <errno.h>

struct sockaddr_in sinto =
  {AF_INET};
struct sockaddr_in sfrom =
  {AF_INET};

extern int errno;

main(ac,av)
int ac;
char **av;
{
  int fd;
  int cnt;
  char line[80];

  if(ac != 2){
    printf("usage:%s host\n",*av);
    exit(0);
  }
  else

/* make socket connection */
  {
  *av++;
```

```
/*get address of requested host*/
  if((sinto.sin_addr.s_addr =
     rhost(av)) < 0){
     perror("rhost");
     exit(0);
    }
/* reserve a legal port */
    sinto.sin_port =
     htons(IPPORT_RESERVED+1);
/*make a socket file descriptor*/
    if(!(fd = socket
    (SOCK_STREAM,0,&sfrom,0))){
     perror("socket");
     exit(0);
    }
/* connect to other host */
    if(connect(fd,&sinto)<0){
     perror("connect");
     exit(0);
    }
    printf(" - connected -\n");
/* while there is input, read it */
    while((cnt = read(0,line,80))>0){
/* some data massaging */
      printf("sending <");
      fflush(stdout);
      write(1,line,cnt-1);
      printf("\\n>\n");
      fflush(stdout);
/* write to socket */
      if(write(fd,line,cnt) < 0){
       perror("write");
       exit(0);
      }
    }
    printf("DONE\n");
/* clean up */
    close(fd);
  }
}
```

[1] XNS is a trademark of Xerox Corporation.

Geometry Hotline

Customer calls during non-business hours will soon be answered by an automated voice storage system.

With this change, you will be able to leave the serial number of your system and a message of unlimited length. Messages will be processed on the next business day in the order received.

We believe that this change will improve the quality of the Hotline service, since we will have a description of your request before we return your call.

To contact the Geometry Hotline, continue to use these numbers:

(800) 345-0222 California

(800) 252-0222 U.S. except California

(800) 443-0222 Canada

IRIS User Survey

Silicon Graphics is currently conducting a survey of IRIS users. This survey gives you the chance to impact our product and services. If you haven't already done so, please take the time to fill it out and return it to us.

To thank you for your participation in the survey, we will hold a drawing for special prizes. To be eligible for the drawing, your survey must be received with a postmark of April 21, 1986 or before.

If you have not received a survey, please contact Monica Schulze. Survey results will be shared with all participants. We look forward to hearing from you.