GEOM
A Program to Draw Molecules

Harold V. McIntosh
with assistance from
Saturnino Julio De La Trinidad H.
and the participation of
the Spring, 1993, Fortran III Class
Departamento de Aplicación de Microcomputadoras,
Instituto de Ciencias, Universidad Autónoma de Puebla,
Apartado postal 461, 72000 Puebla, Puebla, México.

August 30, 1993

Contents

GEOM is a collection of programs for drawing spheres and families of spheres. One application consists in constructing molecular models, wherein each atom is represented by a sphere of appropriate size. However, there are other applications; inasmuch as Puebla lay very near the center of the path of the memorable eclipse of June, 1991, there was considerable interest in studying the dynamics of the shadow cast by one spherical object onto another. Beyond individual applications, computer technology has evolved over the decades in which **GEOM** has been in devolopment, with the result that several different graphics strategies can be compared.

1 Introduction

GEOM is one of a number of programs dating from the *Academia de Matemáticas Aplicadas* of the *Escuela Superior de Física y Matemáticas* of the *Instituto Politécnico Nacional* in Mexico City. In its original form it was an adaptation of a program from the *Quantum Chemistry Program Exchange* whose purpose was to check the parameters in a molecular orbital program.

Since it was easy to exchange coordinates or merely mistype them, it was a useful precaution to make a crude drawing of the molecule to avoid gross errors in the placement of the atoms. From that time, the program has been gradually extended, using graphical techniques for families of spheres, so that now fairly complicated molecules can be shown in terms of rather elaborate spheres.

The same calculation which suppresses the back side of spheres and the portion of one sphere which lies within the interior of another can be repeated with another angular orientation to study the shadows cast by an infinitely distant point source of illumination. As such, it can be used to approximate the shadows cast during an eclipse, such as the one of local interest which occurred in the summer of 1991.

2 Primitive drawing functions

Drawing functions depend somewhat upon the equipment which will perform the actual drawing, and may be divided into roughly three categories, according to whether the primitive display elements are points, lines, or areas.

Pen and ink plotters draw lines; therefore the primitive function moves the pen from one point to another. Utilizing internal memory and specifying pen up or down, the data required by individual function calls can be limited to two real (or integer) coordinates and a boolean variable.

Even so, it is sometimes convenient to call for a long pen stroke, but to actually draw only a part of it, near the beginning or near the end. Evidently there are slight differences between *physical* primitives which are convenient for plotter operation, and *logical* primitives, which would be convenient for programming.

Drawing images on oscilloscopes can be done through either points or lines. Lines were more common in the days of oscilloscopes, but for television monitors and the associated video controllers, individual points are easier to create. That means that when lines are drawn, they must be constituted from sequences of points, or in the case of incremental plotters, sequences of increments. At the same time, using individual points reduces or eliminates questions of partial visibility.

There was a class of plotter on the market which created images by attracting ink droplets electrostatically to the paper surface, not unlike the mode of operation of contemporary laser printers. Although the droplets were strictly points, the control worked by using them to fill areas, most conveniently small rectangular patches. To use such a plotter, or such a style of image construction, the drawing of an area must be taken as the primitive operation.

Conditions have changed once again, to the extent that the PostScript language emphasizes the use of areas filled with a given color or grey tone.

2.1 Plot primitives

The CalComp plotters which were a standard item of equipment in many computers of the 1960's and 1970's responded to octal commands, whose three bits indicated a raised or lowered position for the pen, rotation of the drum up or down signifying increments in the x-axis, and sliding the pen carriage left or right, along the y-axis.

Above and beyond the data stream arriving from the computer, manual controls turned the instrument on or off, and allowed positioning of the pen. An interlock interrupted horizontal motion, preventing the pen carriage from being forced into the margins, but no mechanical check existed for the presence of paper, or running off the end of a roll.

Programs supplied with the plotters changed absolute coordinates, taken as floating point numbers supplied in terms of inches, into a succession of increments through which stepper motors placed the pen where it was desired. Digital increments distinguished the CalComp line from plotters which used digital-analogue converters and servo motors; placement of the pen was more accurate and reproducible, varying somewhat with the quality of electronics employed.

There were essentially four basic programs accompanying the plotter:

- initialize the plotter
  position the pen
- draw axes
- insert letering.

Strictly, the last two could have been created from pen movements alone; the combination of their utility and complexity meant that they were included amongst the basic offerings.

2.2 PC/DOS primitives

Early microcomputers included varying degrees of graphics presentation, mostly of fairly low resolution. Some were programmed to drive CalComp plotters, just as had been done with earlier minicomputers, such as the PDP-8. With the introduction of the IBM/PC, a standard for graphics presentation was established which was widely disseminated, beginning with the CGA video board.

At first, one of the interrupts of the INTEL 8088, interrupt 10, was dedicated to communication with the video monitor, which was arranged for two modes of operation, graphics and text. The reason for the latter was a matter of bandwidth; the use of a character generator driven by ASCII characters was much better than any other system for placing dots in the graphics bitmap. As a consequence, the ROM BIOS underlying the MS/DOS operating system contained a provision for a mixture of approximately sixteen text and graphics primitives.

Early C compilers offered as close an approximation to these sixteen primitives as was possible within the syntax of the language. Later on, more elaborate graphic functions were included within the runtime libraries, corresponding to a certain perception of users requirements, or simply of their imagined desires.

From the point of view of developing an integrated graphics environment, it turns out that the first offerings were the best; fortunately they already contained direct access to individual pixels, which was an adequate foundation upon which to build a graphics package.

The entire set of functions consisted of

- videobackground - set the color of the frame
  videocmode - Set the cursor style

- **videomode** - Choose between text or graphic mode
- **videopalette** - Include or discard blue in the color image
- **videodot** - Place a dot of given color at prescribed coordinates
- **videoputc** - Place a character on the screen
- **videochar** - similar to videocattr, without color coding
- **videocattr** - deposit a character with prescribed color and background
- **videoscroll** - Block movement of text or clearing of the screen
- **videocursor** - Position the cursor (defining the location of text)
- videopage - Select the text page to be displayed
- videogchar - Report the character at the urrsor location
- videogcmode - Find out the cursor mode
- videogcols - Find out the screen width
- videogcursor - Find out the cursor location
- videogdot - Find out the color of the current pixel
- videogmode - Find out the current mode
- videogpage - Find out the current page
- videogpen - Fetch the light pen parameters

Only the items shown in boldface were ever used to write programs such as **GEOM**, and of those, videodot was the essential function for creating computer graphics. The principal problem in changing over to Objective C lies in substituting printf, which is not a video function at all, but nevertheless uses one of the BIOS interrupts to put text on the screen, irrespective of whether it is in text or graphics mode. It is this detail especially, which complicates any simple substitution of video functions when changing from one operating system to another.

The following comprises the header file which has always been incorporated in TURBO C programs to make them compatible with WIZARD C programs, making the use of any TURBO C functions unnecessary. Such a substitution is feasible because both C compilers still create code for MS/DOS, in which the BIOS and screen interrupts are the same.

As for the code, one places the whole interrupt-10 package in its own `.obj` file. The file is relatively small, inasmuch as the functions merely transfer arguments between the stack and the registers of the CPU, conforming to differences between assembly language and C's stack protocol.

```
/*                                                                   */
/*       VIDEOH.H:       Function prototypes                         */
/*                                                                   */
/*       By :  Saturnino Julio De La Trinidad H.                     */
/*                                                                   */
/*       Departamento de Aplicacion de Microcomputadoras             */
/*       Instituto de Ciencias de la Universidad Autonoma de Puebla  */
/*                                                                   */

#if __STDC__
#define _Cdecl
#else
#define _Cdecl        cdecl
#endif

#if     !defined(__WIZARD_DEF_)
#define __WIZARD_DEF_

 void far _Cdecl  videobackground(int color);
 void far _Cdecl  videocattr(int page, char byte, char attr, int count);
 void far _Cdecl  videochar(int page, char byte, int count);
 void far _Cdecl  videocmode(int startrow, int endrow);
 void far _Cdecl  videocursor(int page, int row, int column);
 void far _Cdecl  videodot(int pixelrow, int pixelcolumn, int color);
 void far _Cdecl  videomode(int mode);
 void far _Cdecl  videopage(int page);
 void far _Cdecl  videopalette(int colorset);
 void far _Cdecl  videoputc(char byte, int color);
 void far _Cdecl  videoscroll(int ulrow, int ulcol, int lrrow, int lrcol,
                                                    int lines, int attr);
 int  far _Cdecl  videogchar(int page);
 int  far _Cdecl  videogcmode(void);
 int  far _Cdecl  videogcols(void);
 int  far _Cdecl  videogcursor(int page);
 int  far _Cdecl  videogdot(int pixelrow, int pixelcolumn);
 int  far _Cdecl  videogmode(void);
 int  far _Cdecl  videogpage(void);

 struct pen {
        char               charrow;
        char               charcolumn;
        unsigned char       pixelrow;
        unsigned char       pixelcolumn;
        };

 int  far _Cdecl  videogpen(struct pen *penv);

 #endifæ
```

2.3 Objective C primitives

Outside of the IBM-oriented world, there have been some remarkable advances in the area og graphical display based on the needs of the publishing industry, in particular computer assisted typesetting. At the same time, the quality of high resolution monitors has continued to improve as their price has declined. The result has been the development of visual interfaces and programs for managing them, which can now be used in conjunction with graphics programs.

Unfortunately there does not seem to be a direct correspondence between the PostScript directives and the C primitives. The difficulty, the same as the discrepancy between WIZARD C and TURBO C, lies in mixing text with graphics. Individually each has its equivalents; but we still have to learn how to make the C function `printf` send printed output directly to a PostScript window.

However, Objective C has such an extensive variety of window manipulation, which goes far beyond anything which is normally seen in the PC world, that it seems to be worthwhile to redesign the visual interface for a previously existing program, and not try for a literal adaptation. Here we present an intermediate, consisting of PostScript functions which directly replace the C principal video functions. In some cases, it has been convenient to modify argument types, to avoid repeatedly changing back and forth between integer and double types.

```
/* - - - - - - - - - - -  V I D E O  - - - - - - - - - - - - - - - - */


void videocursor(p,i,j) int p, i, j; {
     PSmoveto(16.0*(float)j,388.0-16.0*(float)i);
}

void videoputc(c,i) char c; int i; {char s[2]; s[0]=c; PSshow(s);}

void videomode(i) int i; {Vmode=i;}
void videopalette(i) int i; {Vpall=i;}

void videoscroll(i,j,k,l,d,c) int i, j, k, l, d, c; {
float ur, uc, lr, lc;
float de, co;
     ur=16.0*(float)i;
     uc=16.0*(float)j;
     lr=16.0*(float)k;
     lc=16.0*(float)l;
     de=(float)d;
     co=(float)c;
     PSsetgray(NX_WHITE);
     PSrectfill(uc,lr,lc-uc,ur-lr);
}

void videocattr(p,c,n,l) int p, c, n, l; {
char s[2];
  s[0]=c;
  PSshow(s);
}

void videoclear(){}
void videobackground(int c){}

void videodot(i,j,l) int i, j, l; {
  switch (l) {
    case 0: if (drawView==2) PSsetrgbcolor(0.7,1.0,0.6); else
            PSsetrgbcolor(1.0,1.0,1.0); break;
    case 1: PSsetrgbcolor(0.5,0.0,0.0); break;
    case 3: PSsetrgbcolor(0.0,0.5,0.0); break;
    case 2: PSsetrgbcolor(0.0,0.0,0.5); break;
    default: PSsetrgbcolor(0.1,0.1,0.1); break;}
/*  PSsetgray(NX_BLACK); */

/*   drawCircle(y,x,1.0); */

    switch (drawView) {
      case 0: PSrectfill((float)j,(float)i,1.0,1.0); break;
      case 1: PSrectfill(2.0*(float)i,398.0-2.0*(float)j,1.25,1.25); break;
      case 2: PSrectfill(8.0*(float)j+1.5,60.0-8.0*(float)i,4.75,4.75); break;
      case 3: PSrectfill(4.0*(float)j+2.0,48.0-4.0*(float)i,3.0,3.0); break;
      case 4: PSrectfill((float)j,199.0-(float)i,1.25,1.25); break;
      case 5: if (l==-1) PSmoveto((float)j,198.0-(float)i);
                 else PSlineto((float)j,198.0-(float)i);
     default: break;}


}

void videofdot(x,y,l) double x, y, l; {
   PSsetgray(16.0*tanh(l/12.0));
   PSrectfill(x,y,4.0,4.0);
}
```

3 Point and line primitives

```
/* place a dot within range (-2.0, 2.0) */
/* at coordinate (x,y), color l         */

void videoadot(x,y,l) double x, y; int l; {
videodot(199-(int)(42.5*y+100.0),(int)(50.0*x+160.0),l);}


/* place a dot within range (-2.0, 2.0)  */
/* at coordinate (x,y), visibility l     */

void pltms(x,y,l) double x, y; int l; {int xx, yy;
xx=(int)(50.0*x);
yy=(int)(50.0*y);
if (l) videodot(yy,xx,1);
}


void geoms(x,y,l) double x, y; int l; {
if (l) videodot(199-(int)(42.5*y+100.0),(int)(50.0*x+160.0),l);
}


void geomms(x,y,l,m) double x, y; int l, m; {int xx, yy;
if (!l) return;
xx=(int)(100.0*x+320.0); if (xx<0) xx=0; if (xx>640) xx=640;
yy=199-(int)(42.5*y+100.0); if (yy<0) yy=0; if (yy>200) yy=200;
videodot(yy,xx,m);
}
```

```
/* pltil - interrupted line */

pltil(x1,z1,q1,x2,z2,q2,l) double x1, z1, q1, x2, z2, q2; int l; {
int    s1, s2;
double xm, zm;
s1=q1>0.0?1:0;
s2=q2>0.0?1:0;
if (s1!=s2) {
  xm=x1-q1*((x2-x1)/(q2-q1));
  zm=z1-q1*((z2-z1)/(q2-q1));
  pltms(xm,zm,s1);
  }
pltms(x2,z2,s2);
}
```

4 Vector and matrix algebra

Given that **GEOM** works extensively with coordinates in three dimensional space, it is convenient to prepare subroutines which can perform operations on three-dimensional matrices and vectors; this includes the calculation of the distance between points and relative orientations. In addition, certain specific matrices will be required, such as those representing rotations which have been specified by their Euler angles.

4.1 Coordinate system

The conventional representation of planar cartesian coordinates has the positive x-axis running to the left, the positive y-axis upwards.

In three dimensions, if a two dimensional graph were placed on a table in front of the observer, the positive x-axis would still run leftwards, but the y-axis would also be horizontal, running away from the observer. In perspective from a normal seating arrangement, that would still be slightly upwards; the axis would be invisible if the projection were made along the y-axis with the origin at eye-level.

The positive z-axis would run upwards, supplanting the two-dimensional y-axis. The difference in viewpoint corresponds to thinking of a two dimensional graph as being mounted on a wall, whereas the x-y portion of a three dimensional graph lies horizontally, as if placed on the table.

The choice of perspective or isometric drawings is one of taste. An architect would probably prefer a perspective view, according to the way an object would appear to a client. An engineer would probably prefer to measure true distances from the drawing, leaving matters of appearance to the imagination. **GEOM** uses parallel projection, adopting the second alternative; projections are made by suppressing the y-coordinate.

4.2 Representation of a sphere family

Spheres are characterized by their centers and radii; namely by four DOUBLE parameters. Other information may also be associated with them, for example the orientation of their poles and Greenwich meridians, whose specification can be accomplished by additional parameters. If Euler angles are used, three additional parameters are required, for a total of seven.

Beyond that, any amount of additional information may be considered, such as tags, identifying subgroupings to which they belong, associated chemical information when they represents atoms within a molecule, and so on. Foreseeing such an application, let us refer to the whole family as a molecule, individual spheres as atoms. Subgroupings could be called radicals, if needed.

The basic operations to be performed on a sphere family, other than drawing its pictorial representation, consist in altering one or more of the parameters. In particular,

- the individual atoms, complete with orientation, can be placed in the molecule and if need be, adjusted.

  The molecule as a whole can be translated or rotated, without affecting the orientation of the atoms.

- the entire molecule can be scaled; to fit it onto a page, or to change the style of representation — say from van der Waals radii to positions of the nuclei.

- changing the orientation of the atoms without moving their centers. This could be done to identify individual atoms, for example.

- altering any combination of parameters for an individual atom.

To accomplish these transformations, we define several operations from the realm of matrix or vector algebra; in particular, we give a certain prominence to some frequently used combinations of operations.

4.3 Vector Algebra

Few of the operations between vectors and matrices need additional explanation.

4.3.1 cartesian vector from polar coordinates

```
/* sphrv - radius vector                   */
/* cartesian vector from polar coordinates */

void sphrv(w,r,th,ph) double *w, r, th, ph; {double s;
s=r*sin(th);
w[0]=s*cos(ph);
w[1]=s*sin(ph);
w[2]=r*cos(th);
}
```

4.3.2 (vector z) = (zero vector)

```
/* sphzv - zero a vector */

void sphzv(z) double *z; {int i; for (i=0; i<6; i++) z[i]=0.0;}
```

4.3.3 (vector w) = (vector z)

```
/* sphcv - copy vector */

void sphcv(z,x) double *z, *x; {z[0]=x[0]; z[1]=x[1]; z[2]=x[2];}
```

4.3.4 (vector v) = (matrix o) * (vector z)

```
/* sphmv - vector = matrix * vector */

void sphmv(w,o,z) double *w, o[][3], *z; {
double u0, u1, u2;
u0=z[0]; u1=z[1]; u2=z[2];
w[0]=o[0][0]*u0+o[0][1]*u1+o[0][2]*u2;
w[1]=o[1][0]*u0+o[1][1]*u1+o[1][2]*u2;
w[2]=o[2][0]*u0+o[2][1]*u1+o[2][2]*u2;
}
```

4.3.5 (vector w) = (matrix o)*(vector z)+(vector a)

```
/* sphap - affine product                      */
/* w and z may be the same                     */
/* (vector w)=(matrix o)*(vector z)+(vector a) */

void sphap(w,o,z,a) double w[3], o[][3], z[3], a[3]; {
double u0, u1, u2;
u0=z[0]; u1=z[1]; u2=z[2];
w[0]=o[0][0]*u0+o[0][1]*u1+o[0][2]*u2+a[0];
w[1]=o[1][0]*u0+o[1][1]*u1+o[1][2]*u2+a[1];
w[2]=o[2][0]*u0+o[2][1]*u1+o[2][2]*u2+a[2];
}
```

4.3.6 (vector z) = (vector x) + (scalar f) * (vector y)

```
/* sphau - augment one vector by another     */
/* vector z = vector x + scalar f * vector y */

void sphau(z,x,f,y) double z[3], x[3], f, y[3]; {
z[0]=x[0]+f*y[0];
z[1]=x[1]+f*y[1];
z[2]=x[2]+f*y[2];
}
```

4.4 Matrix algebra

There are likewise many completely standard operations on just matrices.

4.4.1 Zero matrix

```
/* sphzm - zero matrix */

void sphzm(z) double z[3][3]; {int i, j;
for (i=0; i<3; i++) for (j=0; j<3; j++) z[i][j]=0.0;
}
```

4.4.2 Unit matrix

```
/* sphum - unit matrix */

void sphum(z) double z[3][3]; {int i, j;
for (i=0; i<3; i++) for (j=0; j<3; j++) z[i][j]=0.0;
for (i=0; i<3; i++) z[i][i]=1.0;
}
```

4.4.3 Copy matrix

```
/* sphcm - copy matrix */

void sphcm(z,x) double z[3][3], x[3][3]; {
int i, j; for (i=0; i<3; i++) for (j=0; j<3; j++) z[i][j]=x[i][j];
}
```

4.4.4 Multiply matrices

```
/* sphmm - matrix = matrix * matrix */

void sphmm(z,x,y) double z[3][3], x[3][3], y[3][3]; {
int    i, j, k;
double u[3][3], v[3][3];
sphcm(u,x); sphcm(v,y);
for (i=0; i<3; i++) for (j=0; j<3; j++) {
  z[i][j]=0.0; for (k=0; k<3; k++) z[i][j]+=u[i][k]*v[k][j];
  }
}
```

4.4.5 Distance between centers

```
/* sphdj - distance to center j */

void sphdj(d1,d2,w,s,j) double *d1, *d2, *w, s[][7]; int j; {
double xx, yy, zz, rr;
xx=w[0]-s[j][0]; xx*=xx;
yy=w[1]-s[j][1]; yy*=yy;
zz=w[2]-s[j][2]; zz*=zz;
rr=s[j][3]; rr*=rr;
*d1=xx+yy+zz-rr;
*d2=xx+zz-rr;
}
```

4.5 Rotation matrices

There are two schemes in general use for specifying a rotation; one is in terms of its Euler angles, the other uses its axis and angle of rotation. In reality, there exists a fairly elaborate theory of the three dimensional rotation group, including a third parameterization in terms of quaternions which unifies the other two rather nicely.

The other way to represent rotations is to specify the angle of rotation and the axis about which it occurs. In many ways it gives the most direct representation of a rotation; an additional advantage is that to transform the rotation to a new coordinate system, it is only necessary to rotate the axis.

The user of **GEOM** needs to know little of this, other than to relate rotations to the positioning of the scene that is about to be drawn. For the programmer's part, it suffices to look up the matrix elements of the rotation matrix in some reference work. Nevertheless, without some understanding of the theory, it is difficult to get all the sign conventions and parameter ranges straight.

4.5.1 Angle and axis

In three dimensions, a rotation is defined by a 3×3 orthogonal matrix with positive determinant. In general, such matrices have the real eigenvalue 1 together with two complex conjugate eigenvalues $e^{\pm i\theta}$, wherein θ is the angle of rotation. In turn, there is a real eigenvector lying along the axis of rotation and two complex conjugate null eigenvectors defining the plane of rotation.

However, an orthogonal matrix is also the exponential of a real antisymmetric matrix, having the same eigenvectors but with eigenvalues $0, i\theta$, and $-i\theta$. Three such matrices,

$$\begin{aligned}
\Sigma_x &= \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\
\Sigma_y &= \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix} \\
\Sigma_z &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}
\end{aligned}$$

form a basis for all the 3×3 antisymetric matrices, and are a convenient basis for a Lie Algebra as well. It results that the matrix

$$\begin{aligned}
O &= e^{\theta(\alpha\Sigma_x + \beta\Sigma_y + \gamma\Sigma_z)} \\
&= \begin{bmatrix} \cos\theta + (1-\cos\theta)\alpha^2 & \gamma\sin\theta + (1-\cos\theta)\alpha\beta & -\beta\sin\theta + (1-\cos\theta)\alpha\gamma \\ -\gamma\sin\theta + (1-\cos\theta)\beta\alpha & \cos\theta + (1-\cos\theta)\beta^2 & -\alpha\sin\theta + (1-\cos\theta)\beta\gamma \\ -\beta\sin\theta + (1-\cos\theta)\gamma\alpha & -\alpha\sin\theta + (1-\cos\theta)\gamma\beta & \cos\theta + (1-\cos\theta)\gamma^2 \end{bmatrix}
\end{aligned}$$

rotates vectors through the angle θ about an axis whose direction cosines are (α, β, γ). This gives the *angle-axis* parameterization of a rotation.

Its axis and angle can be recovered by inspection from a rotation matrix. The direction cosines of the axis are proportional to the coefficients of the three anti-symmetric matrices Σ_x, Σ_y, and Σ_z, whereas the angle of rotation is related to the trace via

$$Tr(e^{\theta\Sigma}) \quad = \quad 1 + 2\cos\theta.$$

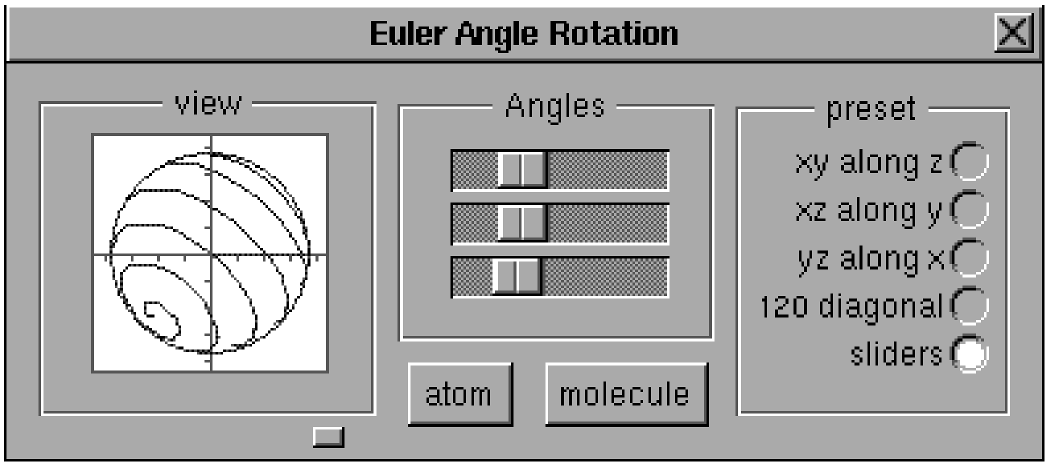


Figure 1: Objective-C panel from which Euler angles may be defined

```
/* sphax - generate rotation from axis and angle */

void sphax(o,th,a,b,c) double o[][3], th, a, b, c; {
double ca, cb, cc, ct, tt, st, r, t;
r=sqrt(a*a+b*b+c*c);
t=(th*3.14159)/180.0;
ca=a/r; cb=b/r; cc=c/r;
ct=cos(t); st=sin(t); tt=1.0-ct;
o[0][0]= ct+tt*ca*ca;
o[0][1]= cc*st+tt*ca*cb;
o[0][2]=-cb*st+tt*ca*cc;
o[1][0]=-cc*st+tt*ca*cb;
o[1][1]= ct+tt*cb*cb;
o[1][2]= ca*st+tt*cb*cc;
o[2][0]= cb*st+tt*ca*cb;
o[2][1]=-ca*st+tt*cb*cc;
o[2][2]= ct+tt*cc*cc;
}
```

4.5.2 Euler angles

To represent a rotation by Euler angles, first rotate the object about the z-axis, then about the y-axis, and finally about the new z-azis.

Rotations about coordinate axes, lying wholly within the plane perpendicular to the axis, have a simple form:

$$
\begin{aligned}
e^{\omega\Sigma_x} &= \begin{bmatrix} \cos\omega & -\sin\omega & 0 \\ \sin\omega & \cos\omega & 0 \\ 0 & 0 & 0 \end{bmatrix} \\
e^{\omega\Sigma_y} &= \begin{bmatrix} \cos\omega & 0 & \sin\omega \\ 0 & 0 & 0 \\ -\sin\omega & 0 & \cos\omega \end{bmatrix} \\
e^{\omega\Sigma_z} &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & \cos\omega & -\sin\omega \\ 0 & \sin\omega & \cos\omega \end{bmatrix}
\end{aligned}
$$

To perform the indicated rotations in succession, through angles ϕ, ρ, and ψ, respectively, means multiplying the corresponding orthogonal matrices.

$$
\begin{aligned}
& e^{\phi\Sigma_x} e^{\rho\Sigma_x} e^{\psi\Sigma_x} = \\
&= \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \cos\rho & -\sin\rho & 0 \\ \sin\rho & \cos\rho & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 0 \end{bmatrix} \\
&= \begin{bmatrix} \cos\phi\cos\psi - \sin\phi\cos\rho\sin\psi & -\cos\phi - \sin\phi\cos\rho\cos\psi & \sin\phi\sin\rho \\ \sin\phi\cos\psi + \cos\phi\cos\rho\sin\psi & -\sin\phi\sin\psi + \cos\phi\cos\rho\cos\psi & -\cos\phi\sin\rho \\ \sin\rho\sin\psi & \sin\rho\cos\psi & \cos\rho \end{bmatrix}
\end{aligned}
$$

```
/* spheu - generate rotation from euler angles */

void spheu(o,a) double o[][3], a[3]; {
double e0, e1, e2, c0, c1, c2, s0, s1, s2;

e0=(a[0]*3.14159)/180.0;
e1=(a[1]*3.14159)/180.0;
e2=(a[2]*3.14159)/180.0;
c0=cos(e0);
c1=cos(e1);
c2=cos(e2);
s0=sin(e0);
s1=sin(e1);
s2=sin(e2);
o[0][0]= c0*c2-s0*c1*s2;
o[0][1]=-c0*s2-s0*c1*c2;
o[0][2]= s0*s1;
o[1][0]= s0*c2+c0*c1*s2;
o[1][1]=-s0*s2+c0*c1*c2;
o[1][2]=-c0*s1;
o[2][0]= s1*s2;
o[2][1]= s1*c2;
o[2][2]= c1;
}
```

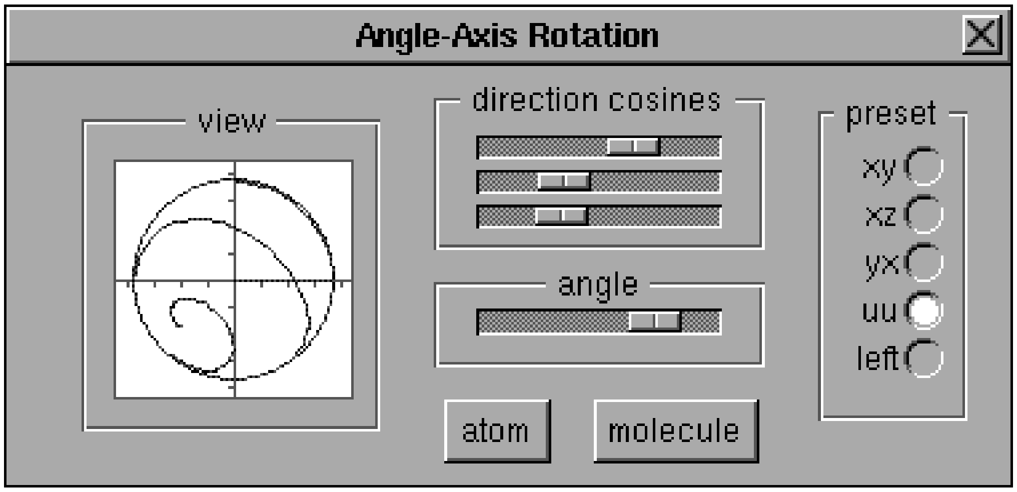


Figure 2: Objective-C panel for defining a rotation via its angle and axis

The formulas for the matrix elements of the Euler angles can be inverted, allowing the angles to be recovered from the matrix. We have:

$$\begin{aligned}\rho &= \arccos O_{33}\\ \psi &= \arccos\,(O_{31}/\sin\rho)\\ \phi &= \arccos\,(-O_{23}/\sin\rho)\end{aligned}$$

wherein the singularity where $\sin\rho$ vanishes should be noted. In that case the first and third rotations occur about the same axis and there is no way to uniquely apportion the rotation between the two.

```
/* sphgeu - find the euler angles of a rotation */

void sphgeu(a,o) double *a, *o[3]; {
double e1, s1;
e1=acos(o[2][2];
s1=sin(e1);
if (s1==0.0) {a[0]=acos(o[1][1]; a[1]=a[2]=0.0;}
    else {a[2]=acos(o[2][1]/s1); a[1]=e1; a[0]=acos(-o[1][2]/s1;}
}
```

4.5.3 Quaternion coordinates

Coordinates can be introduced into four dimensional space accroding to

$$q_0 = \cos\theta$$

$$\begin{aligned} q_1 &= \sin\theta\cos\alpha \\ q_0 &= \sin\theta\cos\beta \\ q_0 &= \sin\theta\cos\gamma, \end{aligned}$$

as well as the alternative,

$$\begin{aligned} q_0 + iq_3 &= \cos\rho e^{i\phi} \\ q_1 + iq_2 &= \sin\rho e^{i\psi}. \end{aligned}$$

The first corresponds to the axis-angle representation, the second to Euler angles [3].

5 Representation of a sphere

Almost anything that recognizably occupies the surface of a sphere can be used for its representation. One obvious choice is to draw the traditional grid of latutudes and longitudes, but another easily programmable choice is to run a spiral from one pole to the other.

Another, more exotic choice, is to select azimuthal and colatitudinal increments which complete different multiples of their circuits as the spiral runs its course, resulting in a sort of spherical Lissajous figure. It will fill the surface of the sphere with more crosshatching than a simple spiral.

Any number of other schemes can be imagined: choose points either randomly or with a density influenced by the intensity of a presumed light source and reflective characteristic of the surface. Draw a fractal curve to fill up the surface. Place identifying letters (such as chemical symbols) on the surface.

5.1 Spherical grid

The most recognizable presentation of a sphere is to draw its grid of latitudes and longitudes, which can be as sparse as simply showing the equator and Greenwich meridian (and antimeridian).

```
/* geosg - represent a sphere by latitude and longitude */

geosg(r0,r,a0,m,l) double *r0, r, *a0; int m, l; {
int    i, j;
double th, ph, dt, dp, y0;
double o[3][3], w[3];

y0=r0[1];
spheu(o,a0);
dt=3.14159/((double)(l+1));
dp=0.0628318;
th=3.14159-dt;
for (i=0; i<l; i++,th-=dt) {
  ph=0.0;
  sphrv(w,r,th,ph);
```

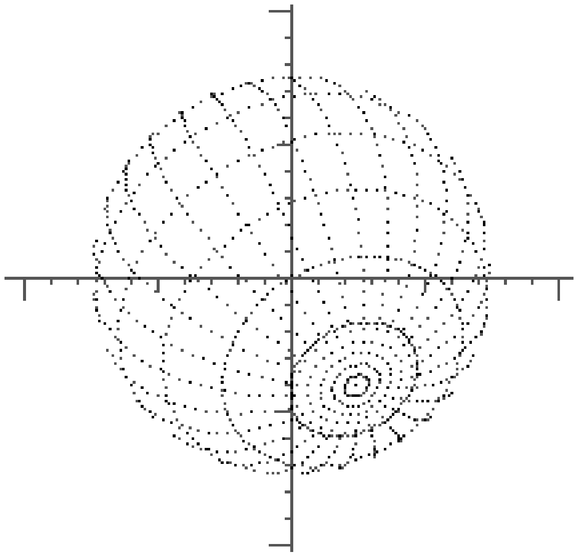


Figure 3: Sphere represented by its latitude-longitude grid

```
  sphap(w,o,w,r0);
  pltms(w[0],w[2],0);
  for (j=0; j<100; j++) {
    ph+=dp;
    sphrv(w,r,th,ph);
    sphap(w,o,w,r0);
    pltms(w[0],w[2],w[1]>=y0);
    }}
dt=0.0628318;
dp=3.14159/((double)m);
for (i=0,ph=0.0; i<m; i++,ph+=dp) {
for (j=0,th=0.0; j<100; j++) {
  th+=dt;
  sphrv(w,r,th,ph);
  sphap(w,o,w,r0);
  pltms(w[0],w[2],w[1]>=y0);
  }}
}
```

5.2 Spiral running from pole to pole

```
/* geosp - scale spherical parameters */

void geosp(s,f,n) double s[][7], f; int n; {int i;
for (i=0; i<n; i++) {s[i][0]*=f; s[i][1]*=f; s[i][2]*=f; s[i][3]*=f;}}
```

Even easier to draw than the traditional grid is a spiral running from pole to pole. It can be created by one continuous pen movement, but still serves to identify the orientation of the sphere by emphasizing the location of the poles. Rotations about the pole are much harder to discern, as they are in most of the presentations.

```
/* geosr - scale spherical radii */

geosr(s,f,n) double s[][7], f; int n; {int i;
for (i=0; i<n; i++) s[i][3]*=f;}
```

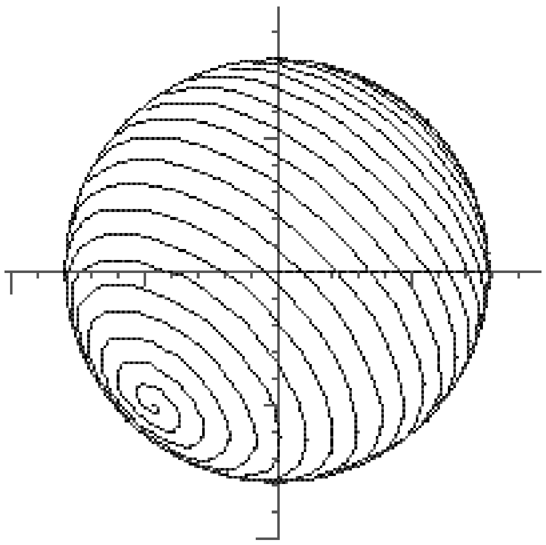


Figure 4: Sphere represented by a spiral running from pole to pole

```
/* geoss - represent a sphere by a spiral running from */
/* pole to pole                                        */

geoss(r0,r,a0) double *r0, r, *a0; {
int    i, n;
double g, th, dh, y0, o[3][3], w[3];
y0=r0[1];
n=2048;
g=64.0;
th=0.0;
dh=3.14159/((double)n);
spheu(o,a0);
sphrv(w,r,th,g*th);
sphap(w,o,w,r0);
pltms(w[0],w[2],0);
for (i=0; i<n; i++) {
  th+=dh;
  sphrv(w,r,th,g*th);
  sphap(w,o,w,r0);
  pltms(w[0],w[2],w[1]>=y0?1:0);
  }
}
```

5.3 Spherical Lissajous figure

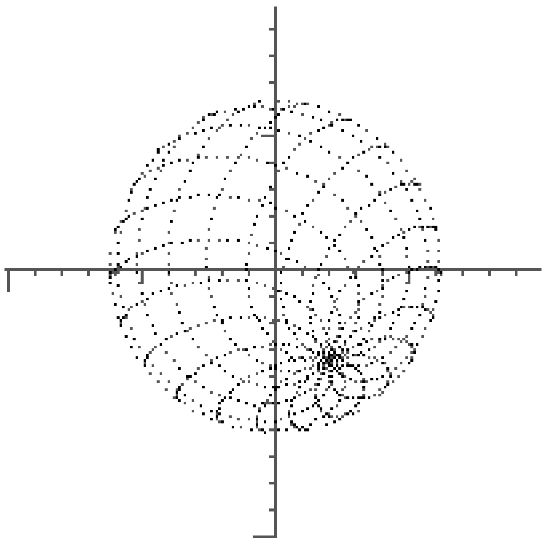


Figure 5: Sphere represented by a Lissajous figure

A mixture of the idea of a spherical grid, which is two dimensional, and a spiral, which is one long line, consists in drawing a Lissajou figure over the spherical surface. To create a closed figure, increments in azimuth should be rational multiples of the increments in longitude, allowing the figure to close when their least common multiple has been reached.

```
/* geosl - represent a sphere by a lissajous figure */
/* extending from pole to pole                      */

void geosl(r0,r,a0,l,m) double *r0, r, *a0; int l, m; {
int    i, n;
double d, th, ph, dt, dp, y0, o[3][3], w[3];
y0=r0[1];
n=2048;
th=0.0;
ph=0.0;
d=6.28318/((double)n);
dt=((double)l)*d;
dp=((double)m)*d;
spheu(o,a0);
sphrv(w,r,th,ph);
sphap(w,o,w,r0);
pltms(w[0],w[2],0);
for (i=0; i<n; i++) {
  th+=dt;
  ph+=dp;
  sphrv(w,r,th,ph);
  sphap(w,o,w,r0);
  pltms(w[0],w[2],w[1]>=y0);
  }
}
```

5.4 Lettering

A representation which could be of interest for chemsits would be to place some lettering upon the surface of the sphere — say the name or the symbol of the chemical element which it is supposed to represent.

5.5 Random points

One especially effective way to represent spheres is to scatter a random selection of points across the surface. Seen in projection, they will congregate near the rim, creating a plausible perspective effect. An even better representation is to further modify the density of points according to the brightness of the sphere as seen under an assumed light source. In fact, different laws of reflection can be used to give the sphere a wide variety of textures.

5.6 Fractals

A fractal curve results from defining a rule of substitution, whereunder a single line segment is replaced by a whole series of segments. Each segment is a scaled version of the original, altogether they must span the same arc. One famous example is the result of dividing the segment into thirds, intending to consider the middle segment as the base of an equilateral triangle, which is replaced by the remaining two edges.

Starting from an equilateral triangle as the basic figure and repeating the replacement infinitely often for each new line segment. The limiting figure is called a snowflake curve; it has infinite length (because every arc is replaced by one 4/3 as long) while both enclosing a finite area and remaining bounded within the original circumscribing circle.

Nevertheless the snowflake does not fill up an area, so a more elaborate rule of substitution would make a better figure to portray a spherical surface.

```
/* geofr - represent a sphere by a fractal curve        */

void geofr(r0,r,a0,l) double *r0, r, *a0; int l; {
int    g0, g1, g2, i, j, k, n;
double d, th, ph, y0, o[3][3], w[3], aa;
int    a[16]={0,1,0,5,3,4,0,1,1,0,4,3,5,0,1,0};
```

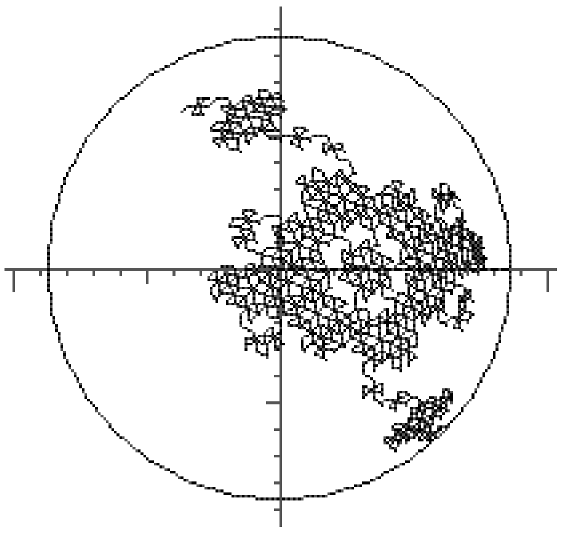


Figure 6: The surface of a sphere can be represented by trying to fill it with a fractal curve. The effect is most successful when the structure of the fractal can be clearly perceived as varying from the center of the sphere towards the rim, when it has a clear orientation, and when it fills a substantial part of the surface.

```
y0=r0[1];
if (l==0) return;
if (l==1) n=16*16*6;
if (l==2) n=16*6*6;
if (l==3) n=6*6*6;
if (l>3) return;
th=1.5707;
ph=0.0;
d=6.28318/((double)n);
spheu(o,a0);
sphrv(w,r,th,ph);
sphap(w,o,w,r0);
pltms(w[0],w[2],0);
for (i=0; i<16; i++) {
  g0=a[i];
  for (j=0; j<16; j++) {
    if (l>1) g1=a[j]; else g1=0;
    for (k=0; k<16; k++) {
      if (l>2) g2=a[k]; else g2=0;
      aa=((double)((g0+g1+g2)%6))*(6.28318/6.0);
      ph+=d*cos(aa); th+=0.866*d*sin(aa);
      sphrv(w,r,th,ph);
      sphap(w,o,w,r0);
      pltms(w[0],w[2],w[1]>=y0);
      }
    }
  }
}

/* geofs - represent a sphere by another fractal curve        */

void geofs(r0,r,a0,l) double *r0, r, *a0; int l; {
int    g0, g1, g2, i, j, k, n;
double d, th, ph, y0, o[3][3], w[3], aa;
int    a[10]={0,5,4,2,0,0,2,4,5,0};
y0=r0[1];
if (l==0) return;
if (l==1) n=10*10*4;
if (l==2) n=10*4*4;
if (l==3) n=4*4*4;
if (l>3) return;
th=1.5707;
ph=0.0;
d=6.28318/(((double)n)*1.732);
spheu(o,a0);
sphrv(w,r,th,ph);
sphap(w,o,w,r0);
pltms(w[0],w[2],0);
for (i=0; i<10; i++) {
  g0=a[i];
  for (j=0; j<10; j++) {
    if (l>1) g1=a[j]; else g1=0;
    for (k=0; k<10; k++) {
      if (l>2) g2=a[k]; else g2=0;
      aa=((double)((2*g0+2*g1+2*g2+1)%12))*(6.28318/12.0);
      ph+=d*cos(aa); th+=0.866*d*sin(aa);
      sphrv(w,r,th,ph);
      sphap(w,o,w,r0);
      pltms(w[0],w[2],w[1]>=y0);
      }
    }
  }
}
```

6 Accounting for the intersection of spheres

The precautions required for drawing multiple spheres, particularly when they intersect or obscure one another, depends somewhat on the technique by which the surfaces are represented. In the point representation, the only decision necessary concerns the visibility of the point, but in the line representation, the two endpoints of the line segment may have different visibilities.

Letting one endpoint determine the visibility of the whole line is only satisfactory when the line segments are very short; otherwise linear interpolation of the visibility criterion greatly improves the appearance of the image. The subroutine which performs this service is pltil.

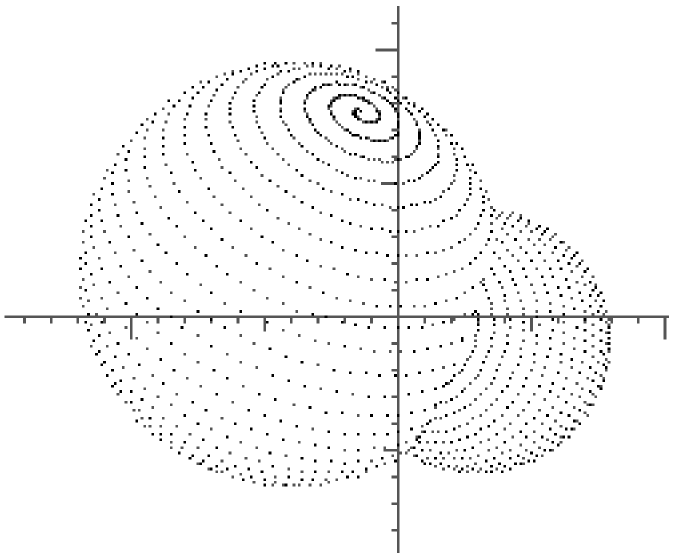


Figure 7: Parts of their surfaces lying within the intersection of two spheres should not be visible.

6.1 Locating the intersection

```
/* geosi - intersection of two spheres */

geosi(s) double s[][7]; {
int    i, j, k, n, m, m0;
double b1, b2, c1, c2, g, th, dh, ri;
double xx, yy, zz, rr;
double w[3], u[3], o[3][3];
n=4096;
n=2048;
g=128.0;
g=64.0;
g=53.0;
dh=3.14159/((double)n);
for (i=0; i<2; i++)
for (j=0; j<2; j++) {
  if (i==j) continue;
  th=0.0;
  ri=s[i][3];
/*  twopv(&s[i][0]); */
/*  twopv(&s[i][4]); */
  spheu(o,&s[i][4]);
/*  twopm(o); */
  sphrv(w,ri,th,g*th);
  sphap(u,o,w,&s[i][0]);
  c1=-1.0;
  c2=-1.0;
  m0=3;
```

```
    pltms(u[0],u[2],0);
    for (k=0; k<n; k++) {
      th+=dh;
      sphrv(w,ri,th,g*th);
      sphap(w,o,w,&s[i][0]);
      xx=w[0]-s[j][0]; xx*=xx;
      yy=w[1]-s[j][1]; yy*=yy;
      zz=w[2]-s[j][2]; zz*=zz;
      rr=s[j][3]; rr*=rr;
      b2=xx+zz-rr;
      b1=b2+yy;
      m=3;
      if      (s[i][1]<w[1]) m0=4;
      else if (b1<0.0) {m=1; m0=1;}
      else if (w[1]<=s[j][1]) m0=3;
      else if (b2>=0.0) m0=3;
      else    m=2;
      switch (m0) {
        case 1: pltil(u[0],u[2],c1,w[0],w[2],b1,i+1); break;
        case 2: pltil(u[0],u[2],c2,w[0],w[2],b2,i+1); break;
        case 3: pltil(u[0],u[2],1.0,w[0],w[2],1.0,i+1); break;
        case 4: pltil(u[0],u[2],-1.0,w[0],w[2],-1.0,i+1); break;
        case 5: pltms(w[0],w[2],i+1); break;
        default: break;
        }

      u[0]=w[0]; u[1]=w[1]; u[2]=w[2];
      c1=b1;
      c2=b2;
      m0=m;
      }  /* end for k */
    }    /* end for j */
  }
```

6.2 The circle of intersection

Two spheres intersect in a circle (which will be imaginary unless they are close enough together) which lies in a plane perpendicular to the line joining their centers. But we need to know the radius of the circle and how far along the line joining the two centers it lies. These quantities are not difficult to obtain by using a modest amount of trigonometry, but there is an interesting formulation of the problem via projective geometry [4, 2] which includes various limiting cases. Among them are point spheres and spheres of infinite radius, which are planes.

6.2.1 Trigonometric solution

Suppose that circle i has radius r_i, that d is the distance between centers, and that we form a plane section including the two centers and p, one of the points where the circles intersect. Let the perpendicular from p to the line joining centers have length c while dividing that line into segments of length a and b. By the

Pythagorean theorem,

$$\begin{aligned} a^2 + c^2 &= r_1^2 \\ b^2 + c^2 &= r_2^2, \end{aligned}$$

whence we conclude

$$a^2 - b^2 = r_1^2 - r_2^2.$$

At the same time,

$$a + b = d.$$

Manipulating these last two equations gives forms for $a+b$ and $a-b$ which in turn yield

$$a = \frac{r_1^2 - r_2^2 + d^2}{2d};$$

the formula for b results from exchangiong r_1 and r_2. Interestingly, the spheres need not even intersect, but then c would be imaginary:

$$c^2 = \frac{2r_1^2 d^2 + 2r_2^2 d^2 + 2r_1^2 r_2^2 - r_1^4 - r_2^4 - d^4}{4d^2}.$$

6.2.2 A projective version

Recalling that the equation of a sphere is

$$\alpha(x^2 + y^2 + z^2) + \beta x + \gamma y + \delta z + \epsilon = 0,$$

regard the coefficients as a vector in a five dimensional space.

Because any non-zero multiple of the equation defines the same sphere, α may as well be chosen equal to 1 unless it, itself, is zero. Except for this degenerate case, substituting $f = \beta/\alpha$, $g = \gamma/\alpha$, and $h = \delta/\alpha$ gives a vector

$$(f, g, h) = -2\mathbf{r},$$

with $\mathbf{r}$ the center of the sphere. Similarly, set $k = \epsilon/\alpha$ to get

$$k = r^2 - R^2,$$

when R is the radius of the sphere and r the distance between its center and the origin.

Defining a metric matrix

$$M = \begin{bmatrix} . & . & . & . & -1/2 \\ . & 1 & . & . & . \\ . & . & 1 & . & . \\ . & . & . & 1 & . \\ -1/2 & . & . & . & . \end{bmatrix}.$$

creates an inner product for which, given coefficient vectors $x_i^T = (\alpha_i, \beta_i, \gamma_i, \delta_i, \epsilon_i)$,

$$x_1^T M x_2 \quad = \quad \beta_1\beta_2 + \gamma_1\gamma_2 + \delta_1\delta_2 - \frac{1}{2}(\alpha_1\epsilon_2 + \alpha_2\epsilon_1)$$

If it were agreed to use normalized vectors $x_i^T = (1, f, g, h, k)$, we would have instead,

$$\begin{aligned} x_1^T M x_2 &= f_1 f_2 + g_1 g_2 + h_1 h_2 - \frac{1}{2}(k_1 + k_2) \\ &= r_1^T r_2 - \frac{1}{2}(R_1^2 + R_2^2 - r_1^2 - r_2^2) \\ &= \frac{1}{2}(R_1^2 + R_2^2 - D^2), \end{aligned}$$

The last two lines result from normalizing the vectors; the very last is a consequence of the cosine law and calling D the distance between centers. According to this, the norm of a sphere is its radius, no matter where it sits.

Some further trigonometry produces the most interesting version of the formula,

$$x_1^T M x_2 \quad = \quad R_1 R_2 \cos\theta$$

wherein θ is the angle at which the two spheres intersect, taken to be the angle between their tangent planes at the point of intersection. This is the same as the angle between the normals to the surface, which are themselves radii.

The remarkable conclusion is that orthogonal circles are those which intersect orthogonally, just as parallel or antiparallel circles are those which are tangent at their intersection. Parallel means internally tangent, antiparallel means externally tangent.

6.3 A general procedure

It is typical of projective geometry that quantities, such as planes, quadrics, or the like, are defined by the vanishing of some linear relation. Here, one five-dimensional coefficient vector represents the surface of a sphere, leaving a four-dimensional space of vectors which could represent points on the sphere. Since at most four of them are linearly independent, four points suffice to define a sphere.

Two spheres whose coefficients are not multiples of each other, define an intersection, which we know to be a circle. Three points must be found, from which to construct the circle. In generality, the problem is to partition space into two sets of linearly independent vectors; one of which holds given conditions, the other the points fulfilling them.

Projective geometry abounds with symbolism which describes the solutions, mostly in terms of cross products, determinants, cofactors, and similar entities. Numerically, it is simpler to start with a matrix of linearly independent vectors, and follow some procedure which will adapt the basis to the conditions at hand, without using all the vocabulary and symbolism of projective geometry.

As it happens, finding families of vectors whose inner products are mostly zero and occasionally not, results by inverting a matrix, which accounts for the presence of cofactors and their ilk in symbolic analyses. Thus, we need a procedure for completing and inverting partially defined matrices.

In the present situation, we have two conditions, namely the coefficients of the two spheres. We require three vectors, each orthogonal to the first two, namely the coordinates of three points which lie on the surfaces of both spheres; the remainder of the circle of intersection consists of linear combinations of these first three.

7 Families of spheres

The ideas used in presenting a pair of spheres can be extended to entire families. In drawing each sphere, it is compared against all the other spheres to determine whether the view of each surface element has been blocked.

By such a scheme, the amount of comparison increases with the square of the number of spheres, so that what is acceptable for tens of spheres may not be acceptable for hundreds of spheres; in that case the introduction of some sort of ordering and horizons would be appropriate.

The calculation of dipole and quadrupole moments for the sphere family will assist in positioning and scaling the drawing for a more effective view.

7.1 Drawing the family

```
/* eomsf - intersection of a family of spheres */

void eomsf(s,n) double s[][7]; int n; {
int    i, j, k, m, mi;
double b1, b2, g, th, dh, ri;
double emi, p[3], o[3][3];
g=57.0;
for (i=0; i<n; i++) {
  ri=s[i][3];
  emi=2000.0*ri;
  mi=(int)emi;
  dh=3.14159/emi;
  th=0.0;
  spheu(o,&s[i][4]);
  for (k=0; k<mi; k++) {
    th+=dh;
    sphrv(p,ri,th,g*th);
    sphap(p,o,p,&s[i][0]);
    if  (p[1]>s[i][1]) continue;
    m=1;
    for (j=0; j<n; j++) {
      if (i==j) continue;
      sphdj(&b1,&b2,p,s,j);
      if (b1<0.0)        {m=0; break;}
      if (p[1]<=s[j][1]) continue;
      if (b2<=0.0)       {m=0; break;}
      }  /* end for j */
```

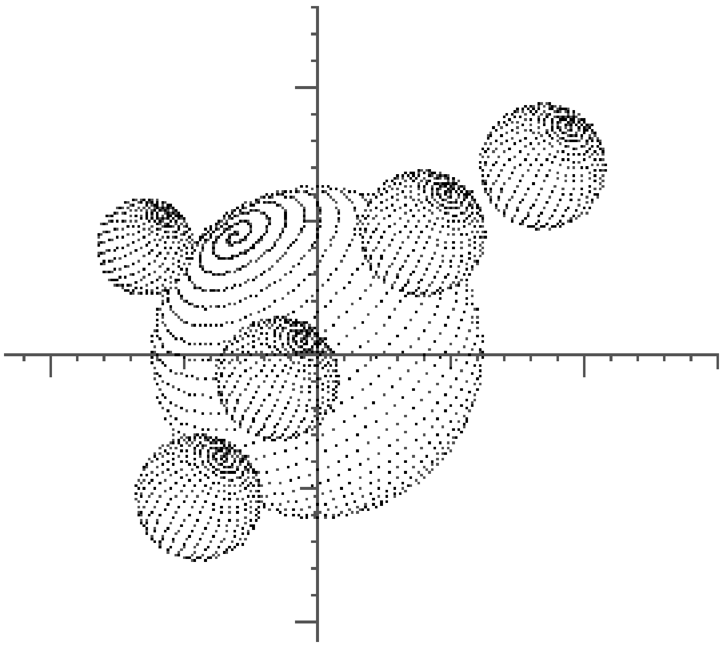


Figure 8: The visibility of surface patches in an entire family of spheres is gotten by comparing all the pairs.

```
      pltms(p[0],p[2],m);
      }  /* end for k */
   }     /* end for i */
}
```

7.2 Dipole moment

"Dipole moment" is a fancy word, for chemists, to designate a center of mass. The point is, that the center of mass (giving each sphere unit mass) is a desirable origin of coordinates, which for artistic reasons might reasonably occupy the center of a drawing of the spheres. It is hard to imagine that scientific reasons might dictate another choice, but the possibility always exists.

One point of caution: the artistic dipole moment described here is neither the center of mass (since individual atomic weights are not taken into account) nor the electrical dipole moment (since no charges are considered).

```
/* calculate dipole moment */

void geodm(w,s,n) double w[3], *s[7]; int n; {
double en; int i;
en=(double)n;
w[0]=w[1]=w[2]=0.0;
for (i=0; i<n; i++) {w[0]+=s[0][i]; w[1]+=s[1][i];w[2]+=s[2][i];}
w[0]/=en; w[1]/=en; w[2]/=en;
}
```

7.3 Quadrupole moment

Just as the dipole moment yields a reasonable origin of coordinates, the quadrupole moment establishes a plausible scale of coordinates. This use is based on Tchebychev's theorem, which states that the fraction of the data lying more than n standard deviations from the mean is $1/n^2$. This is an absolute guarantee, not depending on a Gaussian or any other distribution; consequently it tends to be fairly conservative. In rough terms, less than 11% of the data lies more than 3 standard deviations away from the average.

The theorem is so easy to prove, that it might as well be done right here. By definition, the square of the standard deviation is the second moment of the data with respect to the first mommment (suppose it is zero to simplify the calculation):

$$\sigma^2 \quad = \quad \frac{1}{N}\sum_{i=1}^{N} x_i^2.$$

Divide the data into two subsets, A for which $|x_i| \le \rho\sigma$ and B for the remainder:

$$\sigma^2 \quad = \quad \frac{1}{N}\sum_{A} x_i^2 + \frac{1}{N}\sum_{B} x_i^2.$$

Create an inequality by discarding data from set A and underestimating each term in the sum for B:

$$\sigma^2 \quad \ge \quad \frac{1}{N}\sum_{B}(\rho\sigma)^2.$$

Suppose there were M items in B, the set with the larger data. Then

$$\sigma^2 \quad \ge \quad \frac{M}{N}(\rho\sigma)^2,$$

which readily translates into

$$\text{Probability}(x \ge \rho\sigma) \quad \le \quad \frac{1}{\rho^2}.$$

Applied to multidimensional data, a similar case could be made for quadratic forms constructed from the components of the data, but it is preferable to go ahead and use linear algebra to realize that the eigenvalues of the second moment, or quadrupole, matrix have a similar significance, providing three separate scales for three orthogonal directions. An average, provided by the trace of the quadrupole matrix, is adequate when isotropic coordinates are preferred. The plane of the largest two eigenvalues would be the most revealing, in the case that the third eigenvalue were significantly smaller than the other two.

```
/* calculate quadrupole moment */

void geoqm(q,s,n) double q[3][3], *s[7]; int n; {
double en; int i, j, k;
en=(double)n;
for (i=0; i<3; i++) for (j=0; j<3; j++) q[i][j]=0.0;
for (k=0; k<n; k++)
  for (i=0; i<3; i++) for (j=0; j<3; j++) q[i][j]+=s[k][i]*s[k][j];
for (i=0; i<3; i++) for (j=0; j<3; j++) q[i][j]/=en;
}
```

Once the quadrupole matrix exists, we require a means of diagonalizing it. Many diagonalization programs exist, especially for symmetric matrices. For pedagogical reasons, **GEOM** uses the Jacobi method [1], which is not necessarily the most effective procedure for very large matrices. Nevertheless, it is simple and reliable, so there is no reason not to use it for the matrices which we are likely to encounter.

The Jacobi method consists in using planar rotations to reduce the matrix element belonging to that plane to zero. However, trying to eliminate all the off-diagonal elements is like swatting flies. As a new element is eliminated, many of those previously destroyed partially revive, so they must be confronted over and over again.

In the history of computing methods, different strategies for deciding the order in which to attack the off-diagonal elements, as well as the validity of different approximations to the angle of rotation and the rotation matrices have been studied in considerable detail; all of this goes far beyond the requirements of **GEOM**.

Amongst the idiosyncracies of the Jacobi method, other than the fact that it was far too computation intensive to be practical before the advent of electronic computers, is its behavior with respect to degeneracies. Two diagonal elements which are more nearly equal than even the tiniest off-diagonal element connecting them will provoke a 45^o rotation trying to break the degeneracy. This can be an annoyance in symmetric molecules, where symmetry implies degeneracy and near symmetry implies near degeneracy.

The formulas required by the Jacobi method can be obtained from 2×2 matrices. Let us call c and s, respectively, the sine and cosine of an angle of rotation, θ. Then

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}\begin{bmatrix} a & b \\ b & d \end{bmatrix}\begin{bmatrix} c & -s \\ s & c \end{bmatrix} = \begin{bmatrix} c^2a+2scb+s^2d & sc(d-a)+(c^2-s^2)b \\ sc(d-a)+(c^2-s^2)b & s^2a-2scb+c^2d \end{bmatrix}$$

Some algebra and use of standard trigonometric identities shows that the condition for vanishing of the new off-diagonal element is

$$\tan 2\theta = \frac{2b}{a-d},$$

which tells us the angle of rotation to use. The trace of the new matrix is the same as that of the old, as it should be.

However, the trace of the square of the matrix is also preserved, a bit of algebra that we would like to avoid performing in detail. In the process, the square-sum of the diagonal elements grows, at the expense of the off-diagonal elements. By the nature of the transformation, the same is true for any larger matrix in which this 2×2 submatrix is embedded, providing something of a convergence criterion.

If all the square-sum of a matrix is on the diagonal, it itself is diagonal. So at any moment we can sweep the largest off-diagonal element onto the diagonal, but without knowing the exact way the remaining elements will rearrange themselves, or which of them will be the largest remaining. So there are strategies for selecting the order in which to crush off-diagonal elements.

```
/* make a jacobi rotation of the rows of the    */
/* matrix z through angle t in plane i, j       */
/* two columns will be affected                 */

void sphjr(z,i,j,th) double z[3][3]; int i, j; double th; {
int    k;
double c, s, t;
s=sin(th); c=cos(th);
for (k=0; k<3; k++) {
  t=z[k][i];
  z[k][i]= c*t+s*z[k][j];
  z[k][j]=-s*t+c*z[k][j];
  }
}


/* make a jacobi rotation of the columns of the */
/* matrix z through angle t in plane i, j       */
/* two rows will be affected                    */

void sphjc(z,i,j,th) double z[3][3]; int i, j; double th; {
int    k;
double c, s, t;
s=sin(th); c=cos(th);
for (k=0; k<3; k++) {
  t=z[i][k];
  z[i][k]= c*t+s*z[j][k];
  z[j][k]=-s*t+c*z[j][k];
  }
}


/* diagonalize a symmetric matrix and record its */
/* eigenvectors (using Jacobi's method).         */
/* u[3][3] = matrix of eigenvectors              */
/* v[3][3] = matrix being diagonalized           */

void sphji(u,v) double u[3][3], v[3][3]; {
int    i, j, k, l, ll;
double e, th;
for (i=0; i<3; i++) for (j=0; j<3; j++) u[i][j]=0.0;
for (i=0; i<3; i++) u[i][i]=1.0;
  ll=0;
for (k=0, e=1.0; k<10; k++, e/=3.0) {
  l=0; do {
  for (i=0; i<2; i++) for (j=i+1; j<3; j++) {
```

```
ll++; if (ll>5000) return;
   if(v[i][j]>-e&&v[i][j]<e) continue;
   th=0.5*atan2(v[i][i]-v[j][j],v[i][j]+v[j][i]);
   sphjr(u,i,j,th); sphjr(v,i,j,th); sphjc(v,i,j,th);
   l=1;
   }}  while (l==1);
 }
}
```

8 Quartets of spheres

Projective geometry's sphere space has five-dimensional homogeneous coordinates within which linear algebra may be performed, but that is only for convenience in computation. Since all equations in homogeneous space require the vanishing of inner products, there is a matrix of coefficients whose determinant must vanish. Given such a restriction, the coefficient matrix will always have a maximum of four linearly independent rows or columns; consequently the projective dimension will be four.

For example, four points identify a unique sphere, just as it requires four spheres to intersect at a single point. There are many other situations in which four spheres define a unique environment.

8.1 Four points determine a sphere

To see how this philosophy works, consider four prescribed points (x_i, y_i, z_i), a fifth arbitrary point (x, y, z) and the set of five equations which assert that these five points all lie on the surface of a sphere (setting $x^2 + y^2 + z^2 = r^2$):

$$\begin{aligned}
r_1^2\alpha + x_1\beta + y_1\gamma + z_1\delta + \epsilon &= 0 \\
r_2^2\alpha + x_2\beta + y_2\gamma + z_2\delta + \epsilon &= 0 \\
r_3^2\alpha + x_3\beta + y_3\gamma + z_3\delta + \epsilon &= 0 \\
r_4^2\alpha + x_4\beta + y_4\gamma + z_4\delta + \epsilon &= 0 \\
r^2\alpha + x\beta + y\gamma + z\delta + \epsilon &= 0.
\end{aligned}$$

The singularity of the coefficient matrix requires that

$$\left| \begin{array}{ccccc} r_1^2 & x_1 & y_1 & z_1 & 1 \\ r_2^2 & x_2 & y_2 & z_2 & 1 \\ r_3^2 & x_3 & y_3 & z_3 & 1 \\ r_4^2 & x_4 & y_4 & z_4 & 1 \\ r^2 & x & y & z & 1 \end{array} \right|. = 0$$

Laplace's expansion of this determinant according to its last row yields explicit values for the coefficients $(\alpha, \beta, \gamma, \delta, \epsilon)$. Numerically, relatively arbitrary values can be placed it the last row, and the whole matrix inverted; we want the last row of the inverse, which is insensitive to such choices.

8.2 A sphere orthogonal to four others

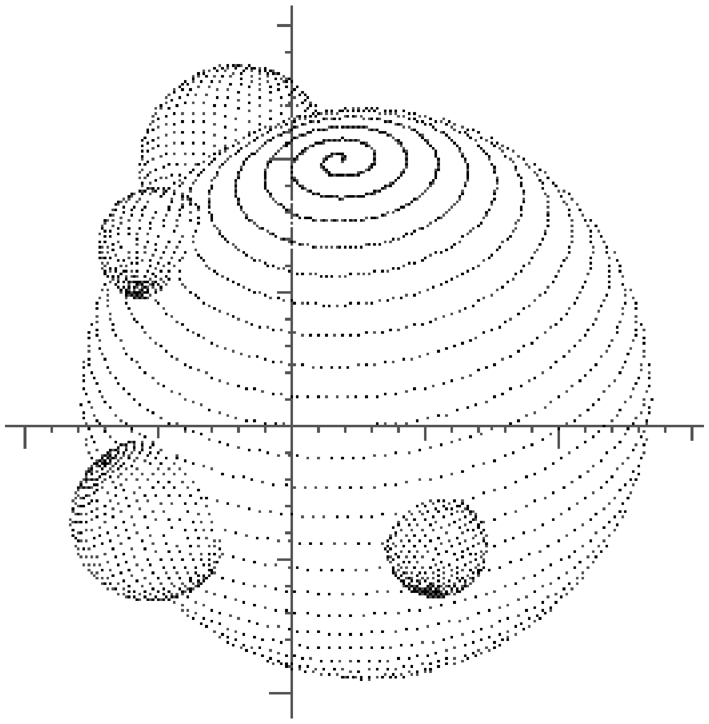


Figure 9: Appolonius' problem of antiquity, to draw a circle tangent to three others can be generalized to finding a fifth sphere tangent to four others. Another generalization varies the angle of intersection of the spheres; the solution is very clean when the fifth sphere must be orthogonal.

To find the parameters of a sphere which is mutually orthogonal to each of four others, it is necessary to solve the four equations

$$\begin{bmatrix} f_1 & g_1 & h_1 & -1/2 \\ f_2 & g_2 & h_2 & -1/2 \\ f_3 & g_3 & h_3 & -1/2 \\ f_4 & g_4 & h_4 & -1/2 \end{bmatrix} \cdot \begin{bmatrix} f \\ g \\ h \\ k \end{bmatrix} = \frac{1}{2} \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{bmatrix},$$

which is a routine job of matrix inversion. Given the solution, the first three components not only locate the center of the desired sphere, they also determine its distance from the origin. With this information, the last coefficient yields the radius of the orthogonal sphere.

The program which calculates the orthogonal sphere can be made slightly more general, for arbitrary angles of intersection

```
void fifth(ss,r,t) double ss[][7], r, t; {
int    i, j;
double w[4][4], z[4][4], p[4][4];
double a[4], b[4], ri[4], rr, s;

for (i=0; i<4; i++) {for (j=0; j<3; j++) z[i][j]=ss[i][j]; z[i][3]=-0.5;}
for (i=0; i<4; i++) for (j=0, ri[i]=0; j<3; j++) ri[i]+=z[i][j]*z[i][j];
for (i=0; i<4; i++) a[i]=0.5*(ri[i]-ss[i][3]);
if (t!=0.0) for (i=0; i<4; i++) {
  s=(r>0?sqrt(r):sqrt(-r));
  a[i]+=s*t*sqrt(ri[i]);
  }
geomi4(w,z);
for (i=0; i<4; i++) for (j=0, b[i]=0.0; j<4; j++) b[i]+=w[i][j]*a[j];
rr=b[0]*b[0]+b[1]*b[1]+b[2]*b[2]-b[3];
ss[4][0]=b[0]; ss[4][1]=b[1]; ss[4][2]=b[2]; ss[4][3]=rr;
}
```

8.3 A sphere tangent to four others

One of the interesting results from the geometry of antiquity is the construction by Appolonius of Perga of a fourth circle tangent to three arbitrary circles. In general there are eight solutions to the problem, according to whether the three tangencies are internal or external. The construction clearly generalizes to spheres and hyperspheres, leading us to expect sixteen different spheres tangent to each of four arbitrarily placed spheres.

The system of equations which has to be solved is slightly more complicated than when finding the orthogonal sphere, because the basic equation is

$$f_i f + g_i g + h_i h - \frac{1}{2}(k_i + k) \quad = \quad R_i R.$$

In fact, we could write a similar system of equations,

$$\begin{bmatrix} f_1 & g_1 & h_1 & -1/2 \\ f_2 & g_2 & h_2 & -1/2 \\ f_3 & g_3 & h_3 & -1/2 \\ f_4 & g_4 & h_4 & -1/2 \end{bmatrix} \cdot \begin{bmatrix} f \\ g \\ h \\ k \end{bmatrix} \quad = \quad \frac{1}{2}\begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{bmatrix} + R \begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix},$$

Unfortunately R, the radius of the tangent sphere, is dependent upon k, so that this is not a system of linear equations; this no doubt reflects the fact that for certain configurations of spheres, not all of the tangencies may be real. Thus another approach is needed, one of which is to assume a value for R and try to arrive at a self-consistent value by iteration.

There is no reason that each sphere should not be given its own angle of intersection, or that a common angle different from 0^o, 90^o or 180^o could not to be chosen.

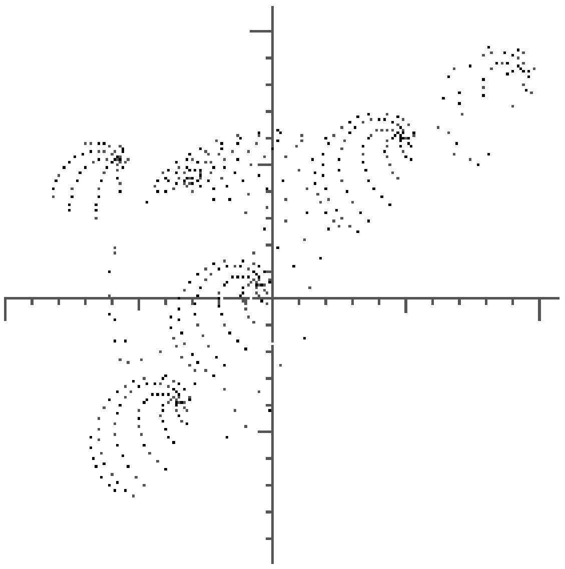


Figure 10: Shadows cast by a distant point light source can be obtained by using a view from a different angle.

9 Casting shadows

The only difference between visibility and illumination, in the scheme of parallel projection used by **GEOM**, lies in the direction of view. Therefore maintaining a double calculation, one for each direction, yields criteria for suppressing (or changing the color of) the picture elements representing the spherical surface.

```
/* shadsf - present a family of spheres in color */
/* s[n][7] - family of spheres                   */
/* u[3][3] - direction of shadow                 */
/* t[n] -    color codes                         */


void shadsf(s,t,u,n) double s[][7], u[3][3]; int *t, n; {
int    i, j, k, m, mi;
double b1, b2, g, th, dh, ri;
double emi, p[3], q[3], a[3], b[3], o[3][3];
g=(double)sspch;
for (i=0; i<n; i++) {
  ri=s[i][3];
  emi=((double)sslen)*ri;
  mi=(int)emi;
  dh=3.14159/emi;
```

```
    th=0.0;
    spheu(o,&s[i][4]);
    for (k=0; k<mi; k++) {
      th+=dh;
      sphrv(a,ri,th,g*th);
      sphmv(p,o,a);
      if (p[1]>0.0) continue;
      sphmv(q,u,p);
      if (q[1]>0.0) continue;
      sphvs(p,p,&s[i][0]);
      sphmv(b,u,p);
      m=1;
      for (j=0; j<n; j++) {
        if (i==j) continue;
        sphdj(&b1,&b2,p,&s[j][0],s[j][3]);
        if (b1<0.0)        {m=0; break;}
        if (p[1]<=s[j][1]) continue;
        if (b2<=0.0)       {m=0; break;}
        sphmv(q,u,&s[j][0]);
        sphdj(&b1,&b2,b,q,s[j][3]);
        if (b1<0.0)        {m=0; break;}
        if (b[1]<=q[1])    continue;
        if (b2<=0.0)       {m=0; break;}
        }  /* end for j */
//    geomms(p[0],p[2],m,t[i]);
      pltms(p[0],p[2],m);
      }  /* end for k */
//  if (kbdst()) {kbdin(); break;}
  }    /* end for i */
}
```

10 Molecules

Although chemists are accustomed to representing the atoms within molecules by spheres, it is still a task which can be accomplished in many different ways.

The most rudimentary is the so-called “ball and stick” model, which can even be realized with gumdrops and toothpicks. All atoms have are taken to have the same size, but do not fill space. In that sense, they represent the nuclei, separated at distances taken from crystallographic data, for example.

When electrons are also taken into account, an atom can be assigned a radius corresponding, say, to where 90% of its electrons are to be found. In this presentation, atoms overlap (bonding can be attributed to electron pairs occupying the same region). Mechanical models traditionally slice away parts of the sphere at stereotyped bonding positions, still maintaining some flexibility of interconnection.

Elaborate model kits maintain variants of each kind of atom, to account for the fact that bond distances and angles are not the same in all molecules, differing slightly with the environment.

Of course, a graphical presentation can adapt itself to all of these styles of presentation, according to the degree of refinement which is required at the moment.

10.1 Benzene ring

Benzene is a planar molecule; six carbon atoms and six hydrogen atoms are arranged in a ring, with double bonds between the carbon atoms. Bonds, nuclei, and electron spheres with van der Walls radii can all be shown. Their positions are generated by subroutines which can generate rings with an arbitrary number of atoms, but six is the c

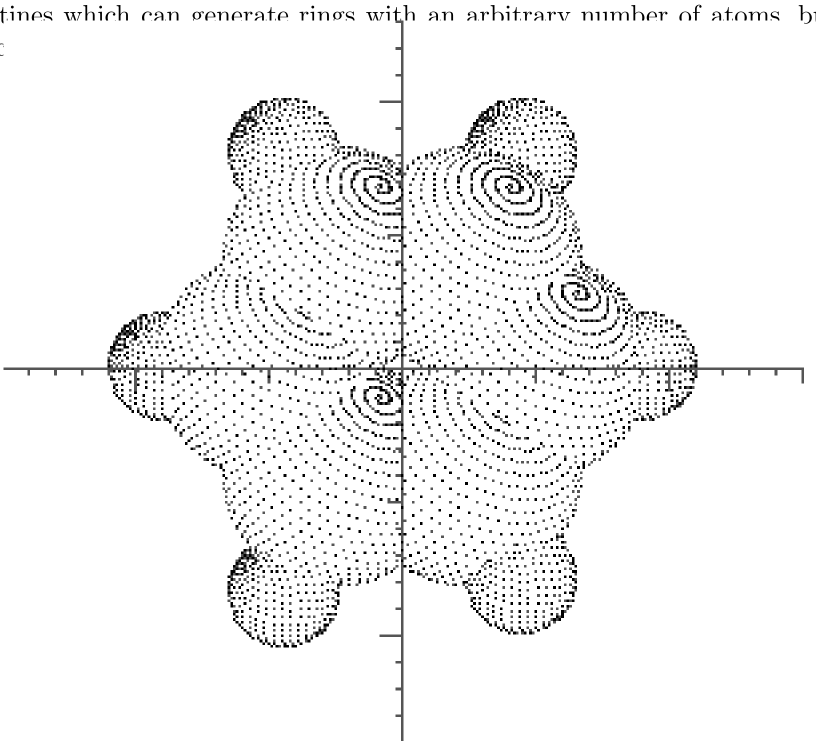


Figure 11: A representation of a benzene molecule using van der Waals radii.

```
/* molhr - generate a hexagonal ring */

void molhr(s,r,rr,a,b,c) double s[][7], r, rr, a, b, c; {int i; double ph;
for (i=0, ph=0.0; i<6; i++,ph+=(60.0*3.14159)/180.0) {
  s[i][0]=r*cos(ph); s[i][1]=0.0; s[i][2]=r*sin(ph);
  s[i][3]=rr;
  s[i][4]=a; s[i][5]=b; s[i][6]=c;}
}


/* molbe - generate a benzene molecule */

void molbe(s,ac,rc,ah,rh) double s[][7], ac, rc, ah, rh; {
molhr(&s[0][0],rc,ac,30.0,50.0,40.0);
molhr(&s[6][0],rh,ah,-60.0,40.0,30.0);
}
```

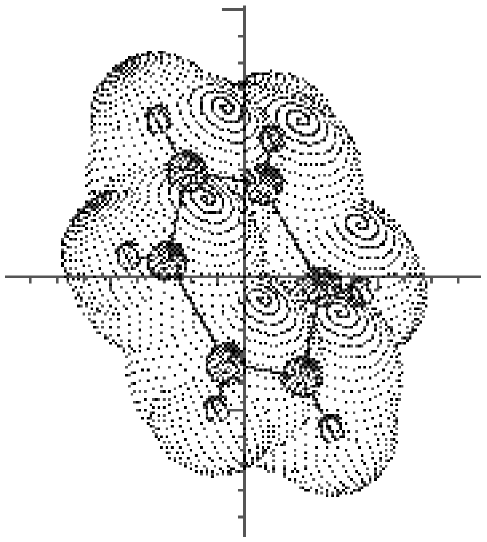


Figure 12: Another representation of a benzene molecule in which smaller spheres are used to represent the nuclei and lines representing bonds have been inserted.

10.2 Sulfur ring

Sulfur is capable of forming chain polymers or rings of various lengths. With a valence of 2, no other kinds of atoms are required, although the chains require monovalent terminators at each end. In a gaseous state, the molecules can be quite irregular, but crystallized, tend to be more regular. The rings and chains are still puckered, and so are neither planar (unlike benzene) nor simple straight lines. Consequently the subroutines generating such molecules have to take three dimensional details into account.

```
/* molsu - generate coordinates of a sulfur ring */

void molsu(s,radsu) double s[][7], radsu; {int i;
for (i=0; i<8; i++) {
molsc(&s[i][0],&s[i][1],&s[i][2],i);
s[i][3]=radsu;
s[i][4]=30.0;
s[i][5]=20.0;
s[i][6]=15.0;
}}

void molsc(x,y,z,p) double *x, *y, *z; int p; {
double d, al, be, t;
```

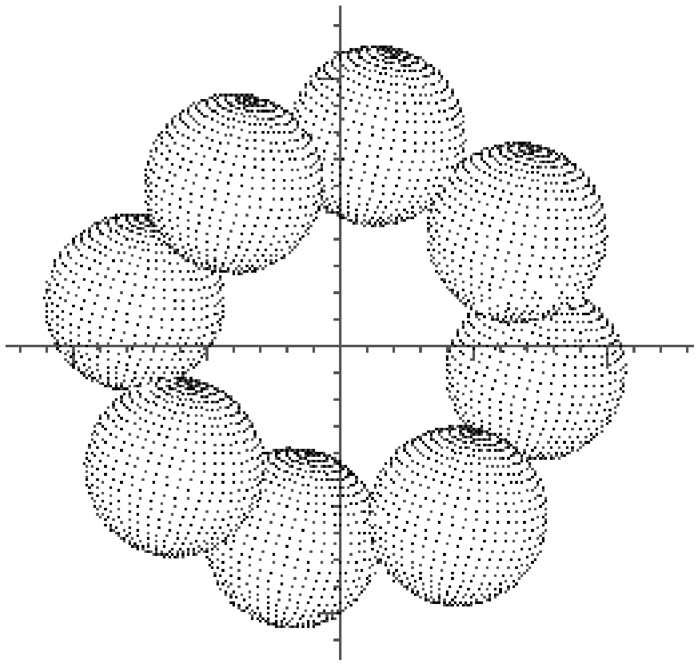


Figure 13: Sulfur readily polymerizes into rings and chains, which are forced into regular forms in crystals. Nevertheless the rings are puckered, not planar. The ring shown, of eight atoms, is one of the most stable.

```
d=2.048;
al=3.1415926/4.0;
be=(107.9*3.1415926)/360.0;
t=sin(be)/cos(al/2.0);
*x=d*sin(be)*cos(p*al)/sin(al);
*y=d*sin(be)*sin(p*al)/sin(al);
*z=(p%2==0?1.0:-1.0)*(d/2.0)*sqrt(1.0-t);
}
```

10.3 Ferrocene molecule

Ferocene is a member of a class of organometallic compounds, in which heavy metal atoms, with disposable d-electrons, can interact with aromatic rings having mobile pi-electrons, to form stable compounds. From a graphical point of view, the question is one of complexity, given the number of atoms present.

```
/* molfc - generate a ferrocene molecule */

\index{molfc}
\begin{quote}
{\footnotesize
```

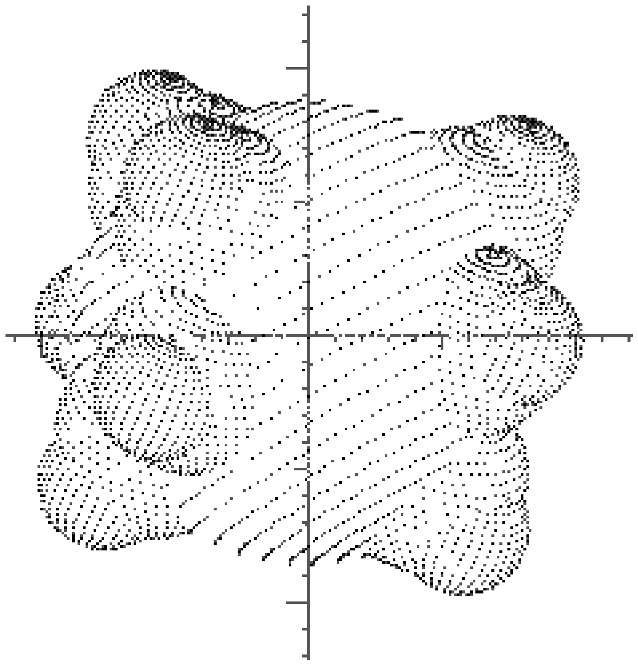


Figure 14: The ferrocene molecule.

```
\begin{verbatim}
void molfc(s,r1,r2,r3) double s[][7], r1, r2, r3; {
molpr(&s[0][0],1.4,r3,1.3,30.0,30.0,30.0);
molpr(&s[5][0],1.0,r2,1.0,30.0,30.0,30.0);
s[10][0]=s[10][1]=s[10][2]=0.0; s[10][3]=r1;
s[10][4]=80.0; s[10][5]=-30.0; s[10][6]=100.0;
molps(&s[11][0],1.0,r2,-1.0,30.0,30.0,30.0);
molps(&s[16][0],1.4,r3,-1.3,30.0,30.0,30.0);
}


/* molpr - generate a pentagonal ring */

void molpr(s,r,rr,h,a,b,c) double s[][7], r, rr, h, a, b, c; {
int    i;
double ph;
for (i=0, ph=0.0; i<5; i++,ph+=(72.0*3.14159)/180.0) {
  s[i][0]=r*cos(ph); s[i][1]=h; s[i][2]=r*sin(ph);
  s[i][3]=rr;
  s[i][4]=a; s[i][5]=b; s[i][6]=c;}
}


/* molps - generate a staggered pentagonal ring */

void molps(s,r,rr,h,a,b,c) double s[][7], r, rr, h, a, b, c; {
```

```
int    i;
double ph;
for (i=0, ph=(36.0*3.14159)/180.0; i<5; i++,ph+=(72.0*3.14159)/180.0) {
  s[i][0]=r*cos(ph); s[i][1]=h; s[i][2]=r*sin(ph);
  s[i][3]=rr;
  s[i][4]=a; s[i][5]=b; s[i][6]=c;}
}
```

11 Random spheres

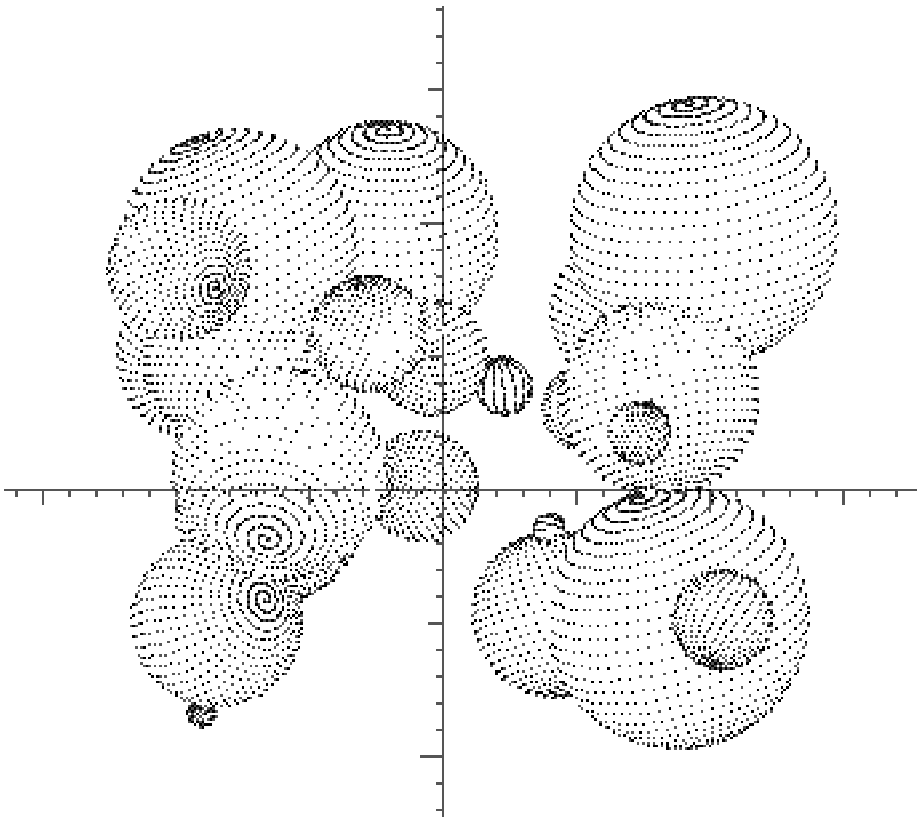


Figure 15: 27 Spheres, all of whose parameters were chosen at random.

Interesting artistic effects can be produced by simply generating a collection of spheres with random parameters. More subtle variations of the same effect would arise from varying the probability distributions, or even giving them a spatial dependence.

```
/* Generate a random assortment of n atoms */
```

```
void molra(s,n) double s[][7]; int n; {int i;
for (i=0; i<n; i++){
  s[i][0]=((double)(rand()%4096))/1024.0-2.0;
  s[i][1]=((double)(rand()%4096))/1024.0-2.0;
  s[i][2]=((double)(rand()%4096))/1024.0-2.0;
  s[i][3]=((double)(rand()%4096))/4096.0+0.1;
  s[i][4]=(((double)(rand()%4096))/4096.0)*180.0;
  s[i][5]=(((double)(rand()%4096))/4096.0)*180.0;
  s[i][6]=(((double)(rand()%4096))/4096.0)*180.0;
  }
}
```

12 Sphere Lattices

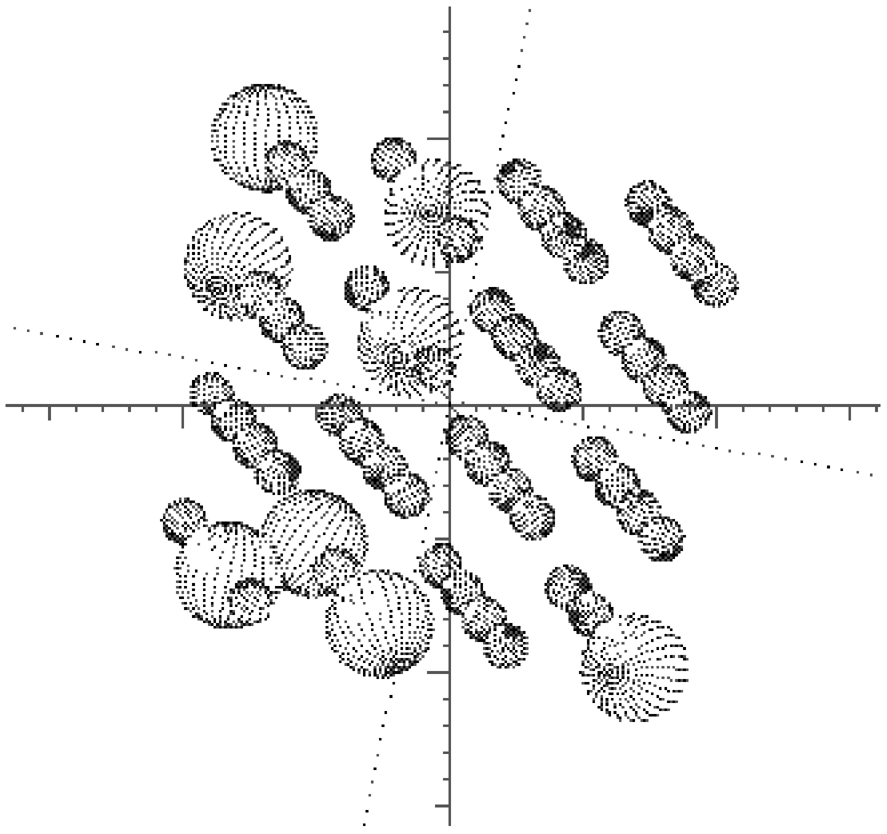


Figure 16: A 4x4x4 lattice containing spheres of two sizes. Such lattices arise in crystallography, the evolution of cellular automata, and in many other places (like video games).

Just as molecules can be represented by collections of spheres, so also can crystal lattices. To avoid an endless proliferation of spheres, only a single unit cell would probably be drawn, although drawing a $3\times3\times3$ cluster would serve to illustrate the central environment better. Provided that the whole assemblage did not become too cluttered, that is.

Another interesting application of a sphere lattice is the representation of a three-dimensional cellular automaton; either the size or the color (or both) of the spheres would serve as an indicator of the state of the cell occupied by the sphere.

```
/* Fill a three dimensional lattice randomly */
/* p is probaility in mils                   */
/* n is lattice width (not number of points) */

void molla(s,p,n) double s[][7]; int p, n; {
int    i, j, k, ii;
double dx;
  dx=3.0/((double)(n-1));
for (i=0; i<n; i++) for (j=0; j<n; j++) for (k=0; k<n; k++) {
  ii=i+n*(j+n*k);
  s[ii][0]=-1.5+((double)i)*dx;
  s[ii][1]=-1.5+((double)j)*dx;
  s[ii][2]=-1.5+((double)k)*dx;
  s[ii][3]=(rand()%1024)<p?0.4*dx:0.166;
  s[ii][4]=(((double)(rand()%4096))/4096.0)*180.0;
  s[ii][5]=(((double)(rand()%4096))/4096.0)*180.0;
  s[ii][6]=(((double)(rand()%4096))/4096.0)*180.0;
  }
}
```

13 Main menu in Turbo C

Unless one wants to use the windowing facilities of a language like TURBO C, the interaction of the program with the user consists mostly of placing informative messages and data at various places on the screen, and in reading the keyboard.

To a certain extent mouse movements can be incorporated along with keystrokes, especially to enhance the use of arrows to move the cursor. As extensions of the alphabet, both the function keys and the editing keys can be placed in the menu, all with the intention of confining commands to single letter sequences.

So far, **GEOM** has not made as extensive use of these opportunities as some other programs have done.

```
main() {
char   k;
int    i, j;

    videomode(COLGRAF);
    videoclear();
    videopalette(REDYEGR);
    videocursor(0,2,0);


k=' ';
di=9; dj=1;                        /* demonstration menu (dj=1 or DCOLS) */
```

```
pi=6; pj=7;                         /* parameter menu                      */
si=0; sj=1;                         /* screen format menu                  */
dmii=10;                            /* active demonstration                */
bg=17;
smen[0]=0; slim[0]=3;                 /* x-y, y-z, or x-r main graph      */
smen[1]=0; slim[1]=2;                 /* parabolic/cartesian coordinates  */
r0[0]=r0[1]=r0[2]=0.0;
a0[0]=a0[1]=a0[2]=45.0;
b0[0]=b0[1]=b0[2]=30.0;

pitc=86.0;
leng=2500.0;
videobackground(bg);

radhy=0.50; radca=0.75; radsu=1.00; radfe=1.45;

tu[0]=3; for (i=1; i<5; i++) tu[i]=1;  tu[5]=2;

for (i=0; i<6; i++) abe[i]=1; for (i=6; i<12; i++) abe[i]=2;

for (i=0; i<5; i++) afc[i]=2; for (i=5; i<10; i++) afc[i]=1; afc[10]=3;
for (i=11; i<16; i++) afc[i]=1; for (i=16; i<21; i++) afc[i]=2;

    geoat(s,-0.8, 0.0, 0.3, 1.6, 30.0, 40.0, 59.0, 0);
    geoat(s, 0.6, 0.2,-0.2, 1.0,-30.0,-95.0,-22.0, 1);

    geoat(t,-0.8, 0.0, 0.3, 1.6, 30.0, 40.0, 59.0, 0);
    geoat(t, 0.6,-0.8,-0.2, 0.7,-30.0,-95.0,-22.0, 1);
    geoat(t, 0.9, 0.5, 0.0, 1.6, 40.0,-75.0,-62.0, 0);

    geoat(u, 0.0, 1.8, 0.0, 1.25, -45.0, 45.0, 45.0, 0);
    geoat(u, 1.7, 0.0, 1.4, 0.47,  30.0, 50.0, 40.0, 1);
    geoat(u, 0.8, 0.0, 0.9, 0.47,  30.0, 50.0,-40.0, 2);
    geoat(u,-0.3, 0.0,-0.2, 0.47,  30.0, 50.0, 40.0, 3);
    geoat(u,-0.9, 0.0,-1.1, 0.47,  30.0, 50.0, 40.0, 4);
    geoat(u,-1.3, 0.2, 0.8, 0.36,  30.0, 50.0, 40.0, 5);

    molbe(be,radca,1.2*radca,radhy,1.8*radca);

    molsu(su,radsu); geosp(su,0.67,8);

    molfc(fc,radfe,radca,radhy);

for (i=0; i<12; i++) for (j=0; j<12; j++) bbe[i][j]=0;
for (i=0; i<6; i++) bbe[i][(i+1)%6]=1;
for (i=0; i<6; i++) bbe[i][i+6]=1;

while (1!=0) {
  videocursor(0,0,0); printf(" -GEOM-");
  tidat();
  di=dmii-1;
  pi=pj-1;
  spheu(uu,b0);
  videocursor(0,2,0); printf("? %1c",k);
   while (!kbdst()) videocursor(0,2,0);
   k=kbdin();
```

```
 if (k=='q') {videomode(T80X25); return;};
videocursor(0,2,0); printf(" %1c ",k);
 switch (k) {

/* demonstration menu with typical parameters and initial conditions */

case '\273': case '#':                       /* F1 key */
  dmii=dmenu();
  videomode(COLGRAF);
  videopalette(REDYEGR);
  break;

/* parameter and initial condition menu activated for +,- */

case '\274': case '$':                       /* F2 key */
  pj=pmenu();
  videomode(COLGRAF);
  videopalette(REDYEGR);
  break;

/* */

case '\275': case '%':                       /* F3 key */
  sj=smenu();
  videomode(COLGRAF);
  videopalette(REDYEGR);                       /* white/cyan/magenta */
  break;

case '\322':                                       /* INSERT key */
  geopi(dmii);
  break;

case '\323':                                       /* DELETE key */
  geopi(0);
  break;

case '\307':                                       /* PAR UP key */
  pj--;
  if (pj==0) pj=PROWS;
  geoqi(pj);
  break;

case '\317':                                       /* PAR DOWN key */
  if (pj>=PROWS) pj=0;
  pj++;
  geoqi(pj);
  break;

case '\311':                                       /* PAGE UP key */
  dmii--;
  if (dmii==0) dmii=DROWS;
  geopi(dmii);
  break;

case '\321':                                       /* PAGE DOWN key */
  if (dmii==DROWS) dmii=0;
```

```
    dmii++;
    geopi(dmii);
    break;

case 'a':                                       /* show skeleton */
    geoni(dmii);
    break;

case 'b':                                       /* draw bonds         */
    geobo(dmii);
    break;

case 'c':
    eomsf(t,3);
    geosp(t,0.5,3);
    eomsf(t,3);
    geosr(t,2.0,3);
    break;

case 'd':
/*    eomsf(be,12); */
    break;

/* echo character from keyboard */

case 'e':
    videocursor(0,0,35);
    printf("%1o",kbdin());
    break;

/* intersecting spheres */

case 'f':
    geosi(s);
    break;

case 'g':                                       /* draw molecule */
    geodi(dmii);
    break;

case 'h':
    geosg(r0,1.5,a0,14,8);
    break;

case 'i':
    geosl(r0,1.25,a0,14,17);
    break;

case 'j': geori(45.0,1.0,1.0,1.0,dmii); break;

case 'k': geori(60.0,1.0,1.0,1.0,dmii); break;

case 'l': geori(90.0,0.2,0.5,1.0,dmii); break;

case 'm': geori(90.0,1.0,0.3,0.2,dmii); break;
case 'n': geori(90.0,0.4,1.0,0.2,dmii); break;

/* read center of sphere */

case 'o':
    twopc('o');
    scanf("%F %F %F",&r0[0],&r0[1],&r0[2]);
    break;

/* read parameters */

case 'p':
    geoqi(pj);
/*    scanf("%F %F %F %F",&g1,&g2,&z1,&z2); */
    break;

/* read euler angles a0 */

case 'u':
    twopc('u');
    scanf("%F %F %F",&a0[0],&a0[1],&a0[2]);
    break;

/* read euler angles b0 */

case 'v':
    twopc('v');
    scanf("%F %F %F",&b0[0],&b0[1],&b0[2]);
    break;

/* set up background */

case 'w': videobackground(bg); break;

/* clear the screen */

case 'z':
    twopc('z');
    videomode(COLGRAF);
    videoclear();
    videopalette(REDYEGR);
    videobackground(bg);
    videocursor(0,2,0);
    break;

/* increase parameter */

case '+':
  switch (pj) {
    case 1: bg++; if (bg==32) bg=0; break;
    case 2: pitc*=1.125; break;
    case 3: leng*=1.414; break;
    case 4: radhy*=1.414; molfc(fc,radfe,radca,radhy); break;
    case 5: radca*=1.414; molfc(fc,radfe,radca,radhy); break;
    case 6: radsu*=1.250; molsu(su,radsu); geosp(su,0.67,8); break;
    case 7: radfe*=1.250; molfc(fc,radfe,radca,radhy); break;
    default: break;
      }
    geoqi(pj);
    break;

/* decrease parameter */

case '-':
  switch (pj) {
    case 1: if (bg==0) bg=32; bg--; break;
    case 2: pitc*=1.125; break;
    case 3: leng/=1.414; break;
    case 4: radhy/=1.414; molfc(fc,radfe,radca,radhy); break;
    case 5: radca/=1.414; molfc(fc,radfe,radca,radhy); break;
    case 6: radsu/=1.250; molsu(su,radsu); geosp(su,0.67,8); break;
    case 7: radfe/=1.250; molfc(fc,radfe,radca,radhy); break;
    default: break;
      }
    geoqi(pj);
    break;

/* go to next parameter with down arrow */

case '\320':
  twocur(pj); printf("=");
  if (pi==PROWS-1) pi=0; else pi++;
  pj=pi+1;
  break;

/* go to previous parameter with up arrow */

case '\310':
  twocur(pj); printf("=");
  if (pi==0) pi=PROWS-1; else pi--;
  pj=pi+1;
  break;

/* ? --- menu listing */

  case '?':
    twopc('?');
    videomode(COLGRAF);
    videopalette(REDYEGR);
    videocursor(0,9,0);
    printf(" F1 - demonstrations\n");
    printf(" F2 - parameters\n");
    printf(" ins - show demo\n");
    printf(" del - clear demo\n");
    printf(" pg up,dn change demo\n");
    printf(" q - quit\n");
    printf(" z - clear screen\n");
    printf(" ? - show menu\n");
    break;

  default: break;
  }  /* end switch */
}    /* end while  */

}  /* end main */
```

14 Main menu in Objective C

With access to a greatly more detailled screen and a well developed collection of windowing programs, Objective C tempts the programmer to build up much more elaborate displays, which, except for entering numbers or strings of text, can be manipulated almost exclusively by mouse movements.

The following interface has two different parts. One is a simple replacement for the MS/DOS menu; the other elaborates several of the options into full displays with their own windows.

14.1 Simple replacement of Turbo C GEOM

```
/* - - - - - - - - - - - - -  G E O M  - - - - - - - - - - - - - - - - - */

- readEulerAngles:sender
{
  a0[0]=180.0*[[euRot cellAt:0:0] doubleValue];
  a0[1]=180.0*[[euRot cellAt:1:0] doubleValue];
  a0[2]=180.0*[[euRot cellAt:2:0] doubleValue];
return self;
}

- readAngleAxis:(double *)a:(double *)aa
{
  a[0]=[[cosRot cellAt:0:0] doubleValue];
  a[1]=[[cosRot cellAt:1:0] doubleValue];
  a[2]=[[cosRot cellAt:2:0] doubleValue];
  *aa=[angRot doubleValue];
  if (a[0]==0.0&&a[1]==0.0&a[2]==0.0) a[0]=a[1]=a[2]=0.886;
return self;
}

- readParameters:sender
{int i, j; double o[3][3], a[3];
  if (strcmp([[sender selectedCell] title],"sphere 1")==0) i=0;
  if (strcmp([[sender selectedCell] title],"sphere 2")==0) i=1;
  if (strcmp([[sender selectedCell] title],"sphere 3")==0) i=2;
  for (j=0; j<3; j++) [[rightCenter cellAt:j:0] setDoubleValue:u[i][j]];
  [rightRadius setDoubleValue:u[i][3]];
  for (j=0; j<3; j++) [[rightEuler cellAt:j:0] setDoubleValue:u[i][j+4]];
spheu(o,&u[i][4]);
sphdc(a,o);
  [[rightAngles cellAt:0:0] setDoubleValue:a[0]];
  [[rightAngles cellAt:1:0] setDoubleValue:a[1]];
  [[rightAngles cellAt:2:0] setDoubleValue:a[2]];
  [[rightAngles cellAt:3:0] setDoubleValue:sphar(o)];
return self;
```

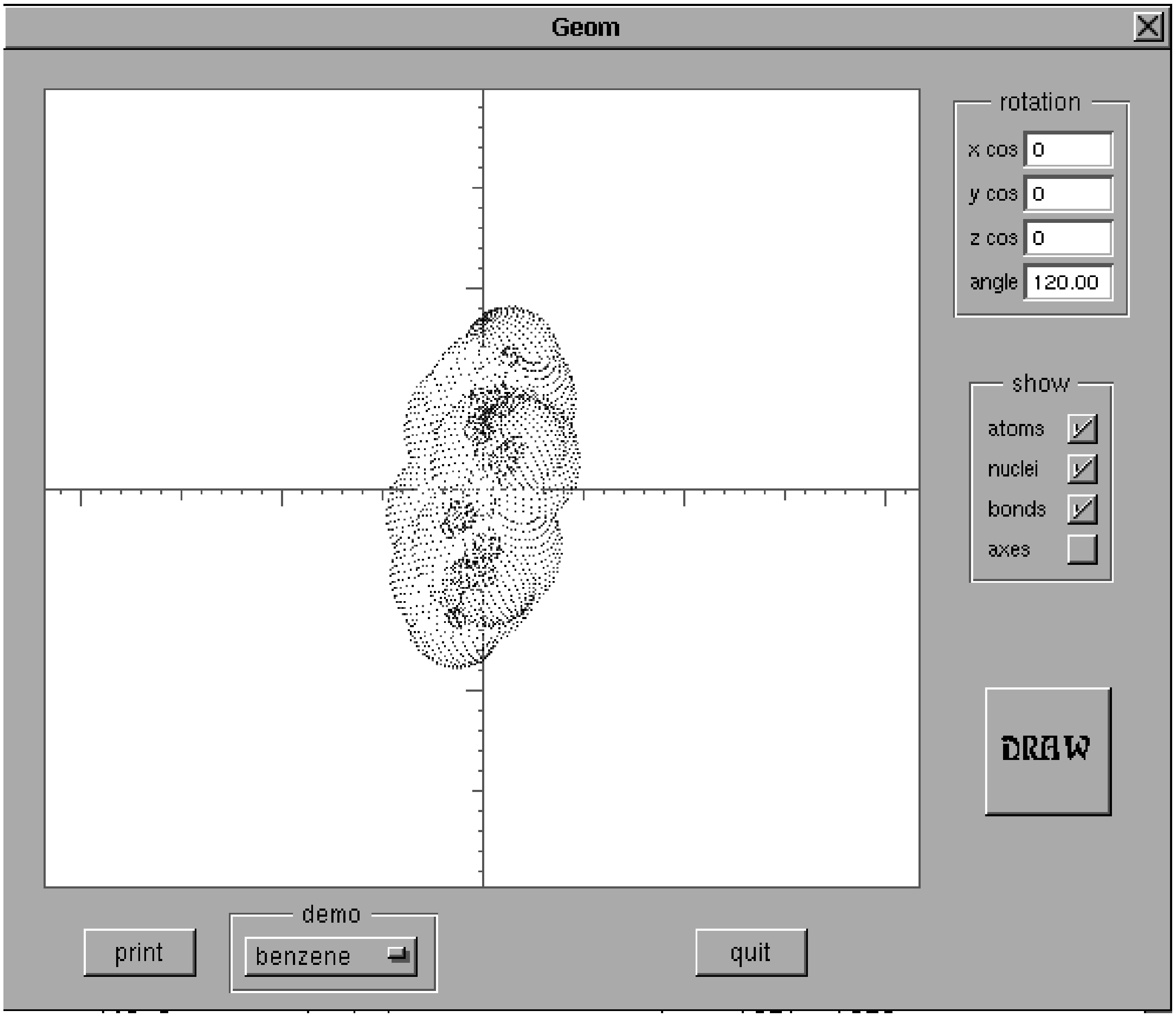


Figure 17: Objective C master panel designed as a simple replacement for the MS-DOS interface. Of course, the resemblance is only approximate.

```
}

- rotateMolecule:sender
{
  m0[0]=[[cosRot cellAt:0:0] doubleValue];
  m0[1]=[[cosRot cellAt:1:0] doubleValue];
  m0[2]=[[cosRot cellAt:2:0] doubleValue];
  m0[3]=360.0*[angRot  doubleValue];
  if (m0[0]==0.0&&m0[1]==0.0&m0[2]==0.0) m0[0]=m0[1]=m0[2]=0.886;
  sphax(mrot,m0[3],m0[0],m0[1],m0[2]);

return self;
}

- setParameters:sender
{int i, j; double o[3][3], a[3];
  if (strcmp([sphereMenu title],"sphere 1")==0) i=0;
  if (strcmp([sphereMenu title],"sphere 2")==0) i=1;
  if (strcmp([sphereMenu title],"sphere 3")==0) i=2;
  for (j=0; j<3; j++) u[i][j]=[[rightCenter cellAt:j:0] doubleValue];
  u[i][3]=[rightRadius doubleValue];
  for (j=0; j<3; j++) u[i][j+4]=[[rightEuler cellAt:j:0] doubleValue];
spheu(o,&u[i][4]);
sphdc(a,o);
  [[rightAngles cellAt:0:0] setDoubleValue:a[0]];
  [[rightAngles cellAt:1:0] setDoubleValue:a[1]];
  [[rightAngles cellAt:2:0] setDoubleValue:a[2]];
  [[rightAngles cellAt:3:0] setDoubleValue:sphar(o)];
return self;
}

- setLighting:sender
{
double a[3], aa;
  [self readAngleAxis:a:&aa];
  sphax(uu,aa,a[0],a[1],a[2]);
  [[leftAngles cellAt:0:0] setDoubleValue:a[0]];
  [[leftAngles cellAt:1:0] setDoubleValue:a[1]];
  [[leftAngles cellAt:2:0] setDoubleValue:a[2]];
  [[leftAngles cellAt:3:0] setDoubleValue:sphar(uu)];
return self;
}

- setRotation:sender
{
  switch ([sender selectedRow]) {
    case 0: [angRot setDoubleValue:90.0/360.0];
            [[cosRot cellAt:0:0] setDoubleValue:0.0];
            [[cosRot cellAt:1:0] setDoubleValue:1.0];
            [[cosRot cellAt:2:0] setDoubleValue:0.0];
            break;
    case 1: [angRot setDoubleValue:90.0/360.0];
            [[cosRot cellAt:0:0] setDoubleValue:0.0];
            [[cosRot cellAt:1:0] setDoubleValue:0.0];
            [[cosRot cellAt:2:0] setDoubleValue:1.0];
            break;
    case 2: [angRot setDoubleValue:90.0/360.0];
            [[cosRot cellAt:0:0] setDoubleValue:1.0];
            [[cosRot cellAt:1:0] setDoubleValue:0.0];
            [[cosRot cellAt:2:0] setDoubleValue:0.0];
            break;
    case 3: [angRot setDoubleValue:60.0/360.0];
            [[cosRot cellAt:0:0] setDoubleValue:0.3];
            [[cosRot cellAt:1:0] setDoubleValue:0.3];
            [[cosRot cellAt:2:0] setDoubleValue:0.3];
            break;
    default:break;}
return self;
}

- showSphere:sender
{
  drawView=0;
  drawSphere=YES;
  [self display];
  drawSphere=NO;
        needsClearing = YES;
return self;
}

- showAxSphere:sender
{
  drawView=5;
  drawAxSphere=YES;
  [self display];
  drawAxSphere=NO;
        needsClearing = YES;
return self;
}

- showEuSphere:sender
{
  drawView=5;
  drawEuSphere=YES;
  [self display];
  drawEuSphere=NO;
        needsClearing = YES;
return self;
}

- showShadows:sender
{
  drawView=0;
  drawShadows=YES;
  [self display];
  drawShadows=NO;
        needsClearing = YES;
return self;
}

- prepSpiral:sender
{
  pushMe=spImButton;
  [spImButton setTitle:"SPIRAL"];
  if ([orientButton state]) [euPanel orderFront:self];
    [gridPanel performClose:self];
    [liPanel performClose:self];
    [spPanel performClose:self];
  if ([paramButton state]) [spPanel orderFront:self];
return self;
}

- prepGrid:sender
{
  pushMe=spImButton;
  [spImButton setTitle:"GRID"];
  if ([orientButton state]) [euPanel orderFront:self];
    [gridPanel performClose:self];
    [liPanel performClose:self];
    [spPanel performClose:self];
  if ([paramButton state]) [gridPanel orderFront:self];
return self;
}

- prepLissa:sender
{
  pushMe=spImButton;
  [spImButton setTitle:"LISSAJOUS"];
  if ([orientButton state]) [euPanel orderFront:self];
    [gridPanel performClose:self];
    [liPanel performClose:self];
    [spPanel performClose:self];
  if ([paramButton state]) [liPanel orderFront:self];
return self;
}

- showUnisphere:sender
{
  drawUnisphere=YES;
  [self display];
  drawUnisphere=NO;
        needsClearing = YES;
return self;
}

- showDisphere:sender
{
int i;
double o[3][3], a[3], aa;
  if (strcmp([[rotateDimer selectedCell] title],"forward")==0) {
     [self readAngleAxis:a:&aa];
     sphax(o,aa,a[0],a[1],a[2]);

     for (i=0; i<3; i++) a[i]=[[leftCenter cellAt:i:0] doubleValue];
     sphmv(a,o,a);
     for (i=0; i<3; i++) [[leftCenter cellAt:i:0] setDoubleValue:a[i]];

     for (i=0; i<3; i++) a[i]=[[rightCenter cellAt:i:0] doubleValue];
     sphmv(a,o,a);
     for (i=0; i<3; i++) [[rightCenter cellAt:i:0] setDoubleValue:a[i]];
     }
   if (strcmp([[rotateDimer selectedCell] title],"backward")==0) {
     [self readAngleAxis:a:&aa];
     sphax(o,-aa,a[0],a[1],a[2]);

     for (i=0; i<3; i++) a[i]=[[leftCenter cellAt:i:0] doubleValue];
     sphmv(a,o,a);
     for (i=0; i<3; i++) [[leftCenter cellAt:i:0] setDoubleValue:a[i]];

     for (i=0; i<3; i++) a[i]=[[rightCenter cellAt:i:0] doubleValue];
     sphmv(a,o,a);
     for (i=0; i<3; i++) [[rightCenter cellAt:i:0] setDoubleValue:a[i]];
     }
 drawDisphere=YES;
  [self display];
  drawDisphere=NO;
        needsClearing = YES;
return self;
}

- printSphere:sender
{
  drawView=0;
  drawSphere=YES;
  [self printPSCode:self];
  drawSphere=NO;
        needsClearing = YES;

return self;
}

- printUnisphere:sender
{
  drawUnisphere=YES;
  [self printPSCode:self];
  drawUnisphere=NO;
        needsClearing = YES;
return self;
}

- printSpIm:sender
{
  sslen=[[spiralData cellAt:1:0] intValue];
  sspch=[[spiralData cellAt:0:0] intValue];
  drawUnisphere=YES;
  [self printPSCode:self];
  drawUnisphere=NO;
        needsClearing = YES;

return self;
}

- printDisphere:sender
{
  drawDisphere=YES;
  [self printPSCode:self];
  drawDisphere=NO;
        needsClearing = YES;

return self;
}

- rotateDimerB:sender
{
  [dimerButton setTitle:"Backward"];
return self;
}

- rotateDimerF:sender
{
  [dimerButton setTitle:"Forward"];
return self;
}

- rotateDimerN:sender
{
  [dimerButton setTitle:"Dimer"];
return self;
}

- pushIt:sender
{
  [pushMe performClick:self];
return self;
}
```

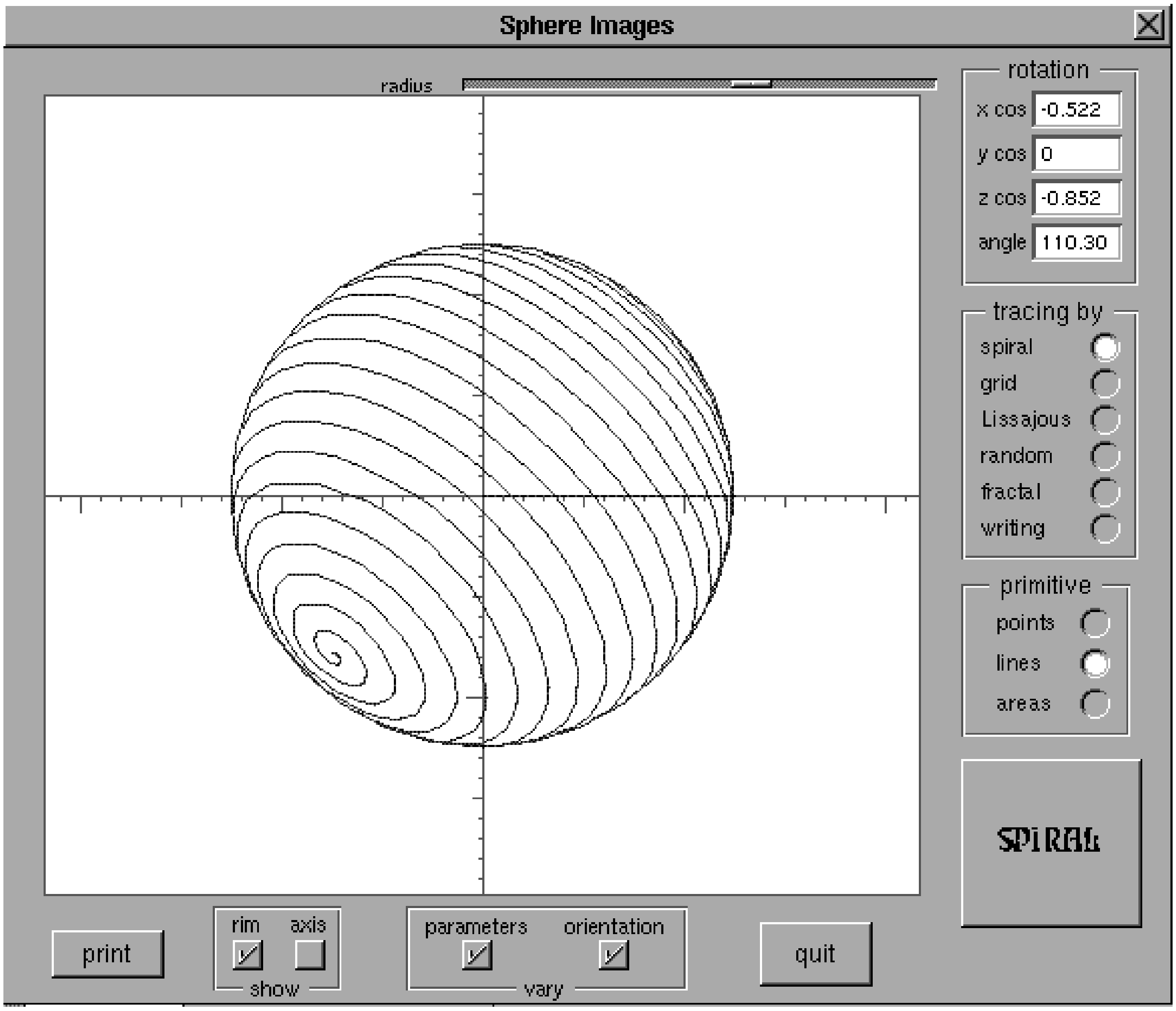
14.2 Single sphere demonstration

The availability of a larger surface in general, and of higher resolution in particular, allows a user to study the details of sphere drawing in much greater detail in the NeXT environment. Consequently, customized windows have been created to elaborate the fairly general menu contained in the TURBO C programs.

Topics which can be emphasized are, the variation of the sphere parameters such as center, radius and orientation of the surface, as well as such aspects of visual appearance as the choice of representation of the surface, the use of point or line elements, the superposition of an axis, and so on.

```
if (drawUnisphere) {

  [self readEulerAngles:self];
  spheu(mrot,a0);
  sphdc(a,mrot);
  [[angleParameters cellAt:0:0] setDoubleValue:a[0]];
  [[angleParameters cellAt:1:0] setDoubleValue:a[1]];
  [[angleParameters cellAt:2:0] setDoubleValue:a[2]];
  [[angleParameters cellAt:3:0] setDoubleValue:sphar(mrot)];
  spRadius=[radiusSlider doubleValue];
```



rotation

x cos	-0.522
y cos	0
z cos	-0.852
angle	110.30

tracing by: [✓ spiral] grid Lissajous random fractal writing

primitive: points [✓ lines] areas

show: [✓ rim] axis

vary: [✓ parameters] [✓ orientation]

Figure 18: Objective C Panel in which the nuances of drawing one single sphere can be studied in greater detail.

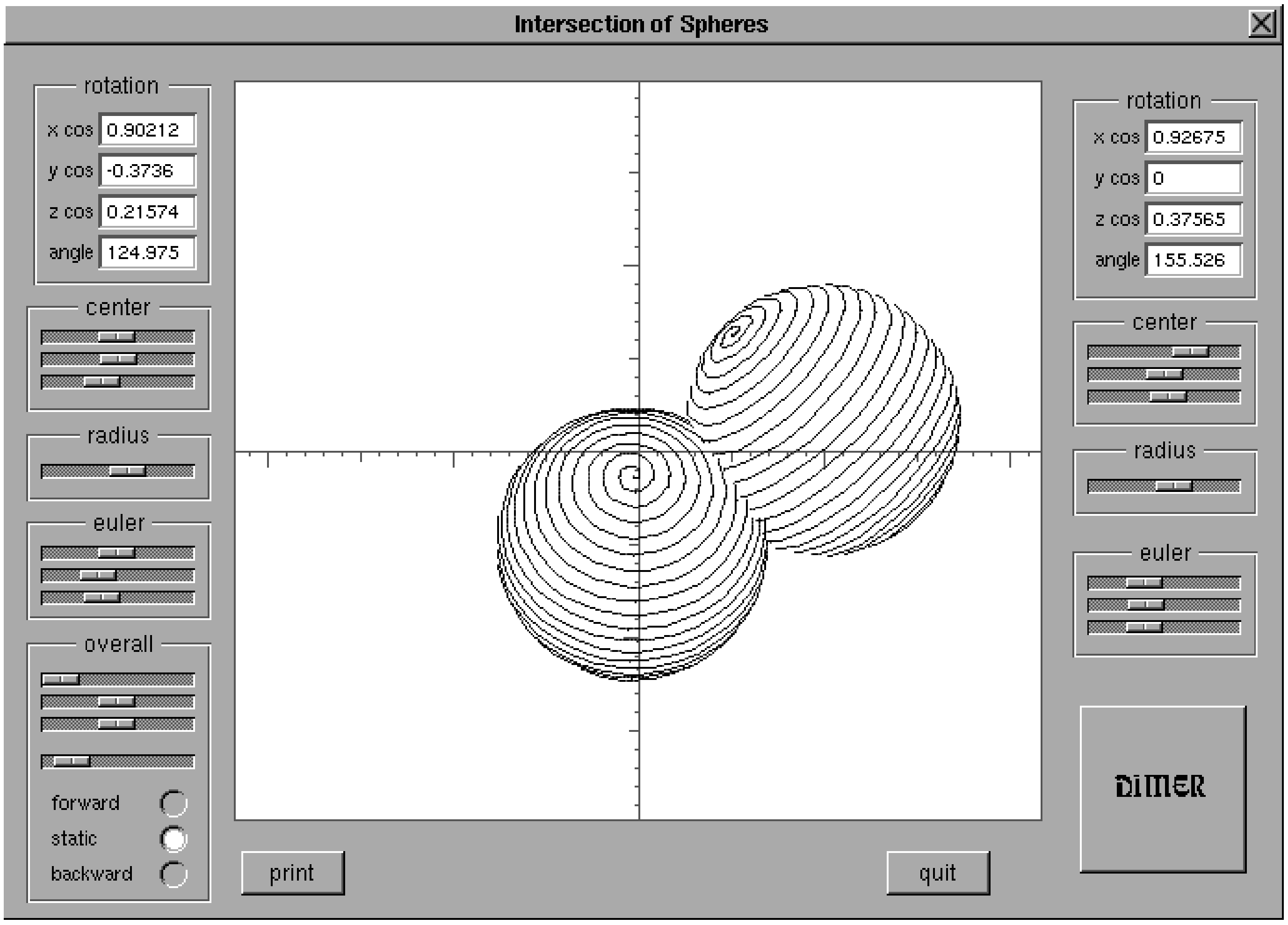
```
  if (strcmp([[primiMenu selectedCell] title],"points")==0) drawView=0;
  if (strcmp([[primiMenu selectedCell] title],"lines")==0) drawView=5;

  if (strcmp([[traceMenu selectedCell] title],"spiral")==0) {
     sslen=[[spiralData cellAt:1:0] intValue];
     sspch=[[spiralData cellAt:0:0] intValue];
     geomss(r0,spRadius,a0,sspch,sslen);
     }
  if (strcmp([[traceMenu selectedCell] title],"grid")==0) {
     nlat=[[gridData cellAt:0:0] intValue];
     nlong=[[gridData cellAt:1:0] intValue];
     geosg(r0,spRadius,a0,nlong,nlat);
     }
  if (strcmp([[traceMenu selectedCell] title],"Lissajous")==0) {
     sllen=[[spiralData cellAt:2:0] intValue];
     nlat=[[gridData cellAt:0:0] intValue];
     nlong=[[gridData cellAt:1:0] intValue];
     geosl(r0,spRadius,a0,nlong,nlat);
     }
  if (strcmp([[traceMenu selectedCell] title],"fractal")==0) {
     frdepth=[[frData selectedCell] title][0]-'0';
     geofs(r0,spRadius,a0,frdepth);
     }
  if ([rimButton state]) {PSmoveto(0.0,0.0);
      PSarc(0.0,0.0,50.0*spRadius,0.0,360.0);}
  if (drawView==5) PSstroke();
  }
```

14.3 Double sphere demonstration

Just as the presentation of a single sphere can be examined in minute detail, so also can the details of sphere intersction and occultation be examined up close. It is particularly convenient to vary the relative positions and radii of the two spheres; for example, to see the differences between clean intersections and near tangencies.

```
if (drawDisphere) {
drawView=5;
s[0][0]=[[leftCenter cellAt:0:0] doubleValue];
s[0][1]=[[leftCenter cellAt:1:0] doubleValue];
s[0][2]=[[leftCenter cellAt:2:0] doubleValue];
s[0][3]=[leftRadius doubleValue];
s[0][4]=[[leftEuler cellAt:0:0] doubleValue];
s[0][5]=[[leftEuler cellAt:1:0] doubleValue];
s[0][6]=[[leftEuler cellAt:2:0] doubleValue];
spheu(o,&s[0][4]);
sphdc(a,o);
  [[leftAngles cellAt:0:0] setDoubleValue:a[0]];
  [[leftAngles cellAt:1:0] setDoubleValue:a[1]];
  [[leftAngles cellAt:2:0] setDoubleValue:a[2]];
  [[leftAngles cellAt:3:0] setDoubleValue:sphar(o)];
s[1][0]=[[rightCenter cellAt:0:0] doubleValue];
s[1][1]=[[rightCenter cellAt:1:0] doubleValue];
s[1][2]=[[rightCenter cellAt:2:0] doubleValue];
s[1][3]=[rightRadius doubleValue];
s[1][4]=[[rightEuler cellAt:0:0] doubleValue];
```



Intersection of Spheres

rotation

x cos 0.90212

y cos -0.3736

z cos 0.21574

angle 124.975

rotation

x cos 0.92675

y cos 0

z cos 0.37565

angle 155.526

Figure 19: Objective C Panel, in which the the difficulties of drawing two spheres, which can occult or intersect one another is shown.

```
s[1][5]=[[rightEuler cellAt:1:0] doubleValue];
s[1][6]=[[rightEuler cellAt:2:0] doubleValue];
spheu(o,&s[1][4]);
sphdc(a,o);
  [[rightAngles cellAt:0:0] setDoubleValue:a[0]];
  [[rightAngles cellAt:1:0] setDoubleValue:a[1]];
  [[rightAngles cellAt:2:0] setDoubleValue:a[2]];
  [[rightAngles cellAt:3:0] setDoubleValue:sphar(o)];

geosi(s);
  if (drawView==5) PSstroke();
}
```

14.4 Quartet demonstration

Spheres are amongst the most symmetrical of objects, so it is not surprising that projective geometry reserves an especially elegant place for them. There is a metric matrix for their coefficients wherein the norm of a sphere is its radius, and the angle between two sphere-vectors turns out to be the angle at which the spheres intersect.

This brings up the three dimensional version of the ancient construction of the circle tangent to any three others. Sometimes the circle is imaginary, but in generality there are eight solutions, depending upon whether any given tangency is internal or external.

Apparently there are sixteen combinations possible for four spheres, but the construction can be generalized still further, wherein the angle with which each sphere intersects the fifth sphere can specified independently of the others. In particular, a real orthogonal fifth sphere can always be found, with the help of a simple matrix inversion.

14.5 Shadow demonstration

The sun is neither a point source of light nor infinitely distant; nevertheless **GEOM** can be used to illustrate all the elements of an eclipse with the exception of the distinction between the umbra and the penumbra.

Besides the kind of detail observable for single and multiple spheres, the direction from which the illumination arrives can be varied and the trace of the shadows on each sphere can followed.

15 geom.h

The header file for the part of **GEOM** written in ordinary C gives the prototypes for all the functions involved.

```
/* GEOM.H                                      */
/* Harold V. McIntosh, 14 December 1972        */
/* 10 November 1990 - "C" version              */
/* 12 April 1992 - prototypes for Objective C */
```

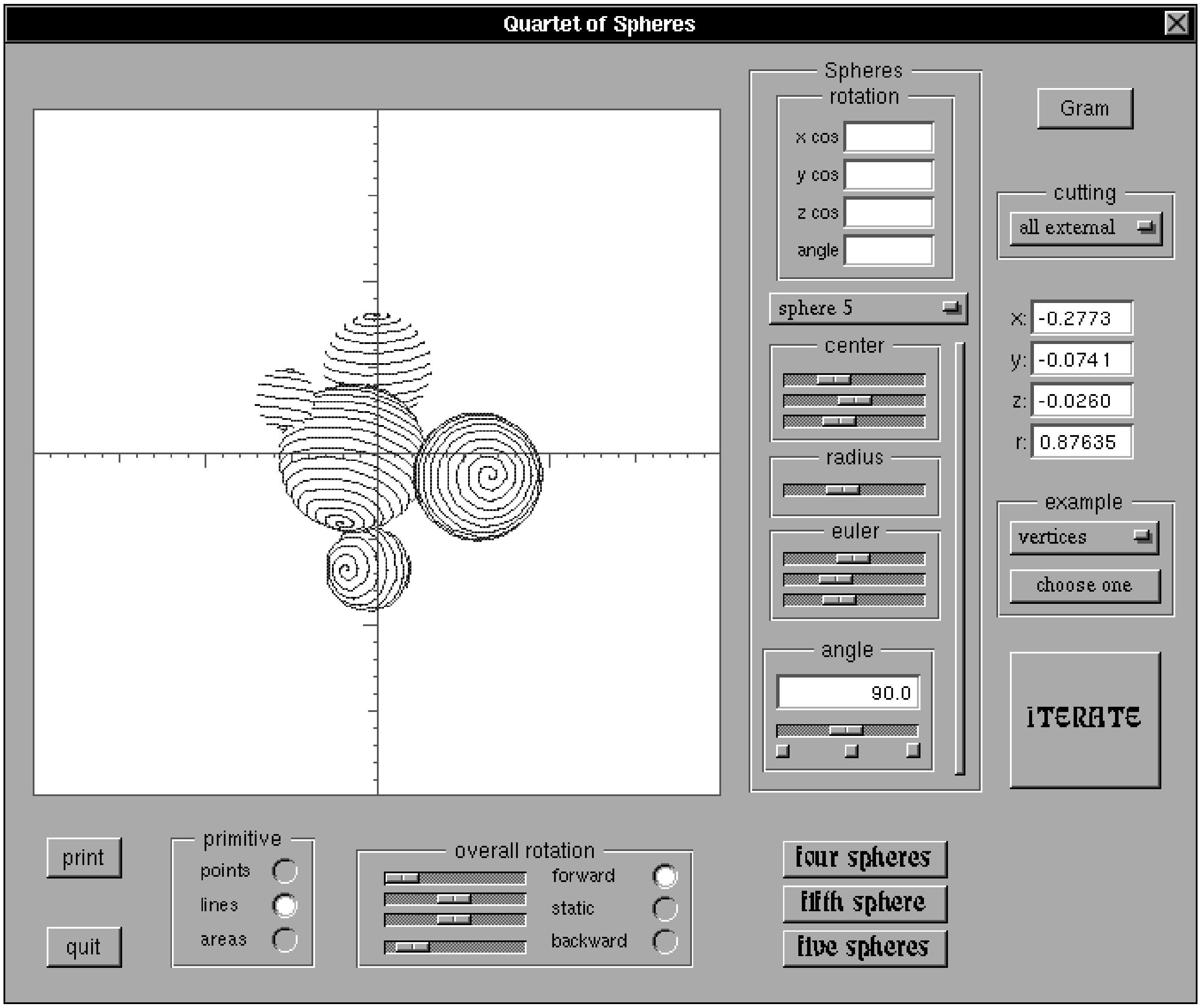
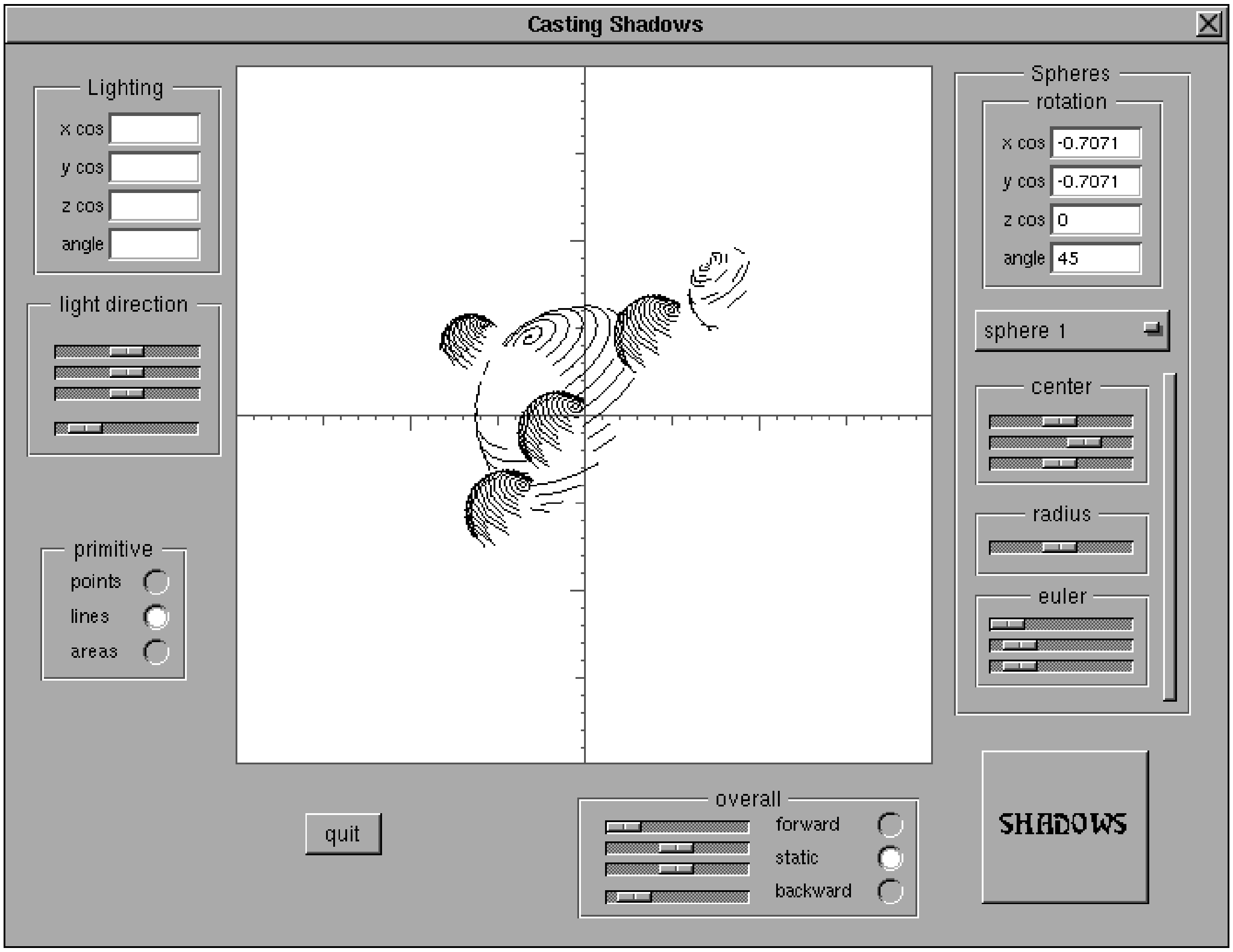


Figure 20: Objective C Panel in which theorems in projective geometry can be studied, especially the analog of Appolonius' classical treatment of finding a circle tangent to three others.



Spheres rotation

x cos -0.7071
y cos -0.7071
z cos 0
angle 45

sphere 1

primitive: points [✓ lines] areas

overall: forward [✓ static] backward

Figure 21: Objective C Panel in which the casting of shadows can be studied in greater detail.

```
void eomsf(double s[][7], int n);
void geoat(double s[][7], double x, double y, double z, double r,
           double a, double b, double c, int n);
void geoav(double s[][7], double x, double y, double z, double r, int n);
void geobo(int i);
void geobf(double s[][7], int b[12][12], int n);
void geodax(double s[][7], double o[3][3], int n);
void geodi(int n);
void geodm(double *w, double *s[7], int n);
void geofr(double *r0, double r, double *a0, int l);
void geofs(double *r0, double r, double *a0, int l);
void geoii();
void geoil(double *p1, double q1, double *p2, double q2);
void geommain();
void geomms(double x, double y, int l, int m);
void geoms(double x, double y, int l);
void geomsf(double s[][7], int *t, int n);
void geoni(int n);
void geopi(int n);
void geoqi(int n);
void geoqm(double q[][3], double s[][7], int n);
void geori(int n);
void georf(double s[][7], double o[][3], int n);
void geosg(double *r0, double r, double *a0, int m, int l);
void geosi(double s[][7]);
void geosl(double *r0, double r, double *a0, int l, int m);
void geosr(double s[][7], double f, int n);
void geoss(double *r0, double r, double *a0);
void geosp(double s[][7], double f, int n);
void geomsg(double *r0, double r, double *a0, int m, int l, int la, int lo);
void geomsl(double *r0, double r, double *a0, int l, int m, int le);
void geomss(double *r0, double r, double *a0, int p, int l);
void molbe(double s[][7], double ac, double rc, double ah, double rh);
void molfc(double s[][7], double r1, double r2, double r3);
void molhr(double s[][7], double r, double rr, double a, double b, double c);
void molla(double s[][7], int p, int n);
void molpr(double s[][7], double r, double rr, double h,
           double a, double b, double c);
void molps(double s[][7], double r, double rr, double h,
           double a, double b, double c);
void molra(double s[][7], int n);
void molsc(double *x, double *y, double *z, int p);
void molsu(double s[][7], double radsu);
void pltfms(double x, double y, int l);
void pltil(double x1, double z1, double q1,double x2, double z2,
           double q2, int l);
void pltms(double x, double y, int l);
void shadsf(double s[][7], int *t, double u[3][3], int n);
void sphau(double *z, double *x, double f, double *y);
void sphap(double w[3], double o[][3], double z[3], double a[3]);
double sphar(double o[3][3]);
void sphax(double o[][3], double th, double a, double b, double c);
void sphcl(double z[][3], double e);
void sphcv(double *z, double *x);
void sphdc(double *a, double o[3][3]);
void sphdj(double *d1, double *d2, double *w, double s[][7], int j);
void spheu(double o[][3], double a[3]);
void sphgeu(double *a, double o[][3]);
void sphjc(double z[3][3], int i, int j, double th);
void sphji(double u[3][3], double v[3][3]);
void sphjr(double z[3][3], int i, int j, double th);
void sphmm(double z[3][3], double x[3][3], double y[3][3]);
void sphmv(double *w, double o[][3], double *z);
void sphrv(double *w, double r, double th, double ph);
void sphum(double z[3][3]);
void sphvc(double *a, double x, double y, double z);
void sphvs(double *z, double *x, double *y);
void sphzm(double z[3][3]);
void sphzv(double *z);
void twopc(int k);
void twopm(double (*z)[3]);
void twopv(double *z);
void videodot(int x, int y, int l);
void videoadot(double x, double y, int l);
void videofdot(double x, double y, int l);

/* end geom.h */
```

16 GeomView.h

The header file for the part of **GEOM** written in Objective C is essentially copied from the NeXT demonstration application PlotView, and is used mostly by the Interface Builder. It contains a list of objects as id's, plus prototypes for all the methods involved. The PlotView class was renamed GeomView, and still retains all those methods to read the mouse cursor, draw the screen, and so on.

```
/*
 * GeomView.h -- Interface file for the GeomView class
 *
 * You may freely copy, distribute, and reuse the code in this example.
 * NeXT disclaims any warranty of any kind, expressed or implied, as to its
 * fitness for any particular use.
 *
 */

#import <appkit/View.h>
#import <appkit/NXCType.h>
// #import <IBViewCache.m>

@interface GeomView:View
{

id euRot;
id cosRot;
id angRot;
id axisButton;
id demoMenu;
id dimerButton;
id frData;
id frPanel;
id goButton;
id gridData;
id gridMenu;
id gridPanel;
id leftAngles;
id leftCenter;
id leftEuler;
id leftRadius;
id orientButton;
id paramButton;
id primiMenu;
id quadDiagon;
id quadEigenv;
id quadMatrix;
id quadPanel;
id radiusSlider;
id rightAngles;
id rightCenter;
id rightEuler;
id rightRadius;
id rimButton;
id rotateDimer;
id showAtoms;
id showAxes;
id showBonds;
id showNuclei;
id sphereMenu;
id spiralData;
id spImButton;
id traceMenu;
id angleParameters;
id spPanel;
id liPanel;
id lissaData;
id laloData;
id euPanel;
id pushMe;

id delegate;
id points;
id crossCursor;
id readOut;
float radius;
BOOL needsClearing;
}

- dismissPanels:sender;
- dLize:header;
- prepSpiral:sender;
- prepGrid:sender;
- prepLissa:sender;
- printSphere:sender;
- printUnisphere:sender;
- printDisphere:sender;
- printSpIm:sender;
- pushIt:sender;
- qMom:sender;
- readAngleAxis:(double *)a:(double *)aa;
- readEulerAngles:sender;
- readParameters:sender;
- rotateDimerB:sender;
- rotateDimerF:sender;
- rotateDimerN:sender;
- rotateMolecule:sender;
- setLighting:sender;
- setParameters:sender;
- setRotation:sender;
- showSphere:sender;
- showShadows:sender;
- showDisphere:sender;
- showAxSphere:sender;
- showEuSphere:sender;
- showUnisphere:sender;

- initFrame:(const NXRect *)frameRect;
- setDelegate:anObject;
- drawSelf:(const NXRect *)rects :(int)rectCount;
- sizeTo:(NXCoord)width :(NXCoord)height;
- clear:sender;
- Geom:sender;
- mouseDown:(NXEvent *)theEvent;
- registerPoint:(NXPoint *)aPoint;
- setRadius:(float)aFloat;
- (float)radius;
- read:(NXTypedStream *)stream;
- write:(NXTypedStream *)stream;
- awake;
- (const char *)inspectorName;
@end

@interface Object(PlotViewDelegate)
- PlotView:sender providePoints:(NXStream **)stream;
- PlotView:sender pointDidChange:(NXPoint *)aPoint;
@end
```

17 Programs for matrix inversion

The fact that C requires relatively fixed dimensions for arrays when working with multiple indices leads to developing special matrix routines for each dimension. **GEOM** needs different dimensions for different applications — three dimensions for routine vector analysis in space, but four and five dimensions for the projective geometry of spheres. The following collection inverts 4×4 matrices.

The method employed is Gaussian elimination with pivoting on the largest elements.

```
/* - - - - - - - - - M A T R I X   I N V E R S E - - - - - - - -  */

/* the subroutines required to invert a matrix of fixed dimension */
/* geomi4 - 4x4 matrix inverse */

void geomi4(z,u) double z[4][4], u[4][4]; {
int    i, j, n, ii[4], geoge4();
double f, w[4][4];
n=4;
geocm4(w,u);
geoum4(z);
for (i=0; i<n; i++) {
  ii[i]=geoge4(w,i);
  geoxr4(w,i,ii[i]);
  for (j=0; j<n; j++) {
    if (i==j) continue;
    f=-w[i][j]/w[i][i];
    geoac4(w,j,j,f,i);
    geoac4(z,j,j,f,i);
    }
  }
for (i=0; i<n; i++) geosm4(z,1.0/w[i][i],i);
for (i=0; i<n; i++) {j=n-i-1; geoxc4(z,j,ii[j]);}
}

/* geoge4 - greatest element in the nth column of 4x4 matrix */

int geoge4(p,n) double p[4][4]; int n; {double g, h; int i, m;
m=n;
g=p[n][n];
g=(g<0.0?-g:g);
for (i=n; i<4; i++) {
  h=p[i][n];
  h=(h<0.0?-h:h);
  if (h<g) continue;
  g=h; m=i;
  }
return m;
}

/* geocm4 - copy 4x4 matrix */

void geocm4(z,x) double z[4][4], x[4][4]; {int i, j;
for (i=0; i<4; i++) for (j=0; j<4; j++) z[i][j]=x[i][j];}

/* geoum4 - 4x4 unit matrix */

void geoum4(z) double z[4][4]; {int i, j;
for (i=0; i<4; i++) {
  for (j=0; j<4; j++) z[i][j]=0.0;
  z[i][i]=1.0;
  }
}

/* geoxc4 - exchange ith and jth columns of a 4x4 matrix */

void geoxc4(z,i,j) double z[4][4]; int i, j; {int k; double t;
if (i==j) return;
for (k=0; k<4; k++) {t=z[k][i]; z[k][i]=z[k][j]; z[k][j]=t;}
}

/* geoxr4 - exchange ith and jth rows of a 4x4 matrix */

void geoxr4(z,i,j) double z[4][4]; int i, j; {int k; double t;
if (i==j) return;
for (k=0; k<4; k++) {t=z[i][k]; z[i][k]=z[j][k]; z[j][k]=t;}
}

/* geoxtc4 - extract nth column from 4x4 matrix */

void geoxtc4(p,z,n) double p[4], z[4][4]; int n; {int i;
for (i=0; i<4; i++) p[i]=z[i][n];}


/* geoac4 - augment column of a 4x4 matrix */

void geoac4(z,i,j,f,k) double z[4][4], f; int i, j, k; {int l;
for (l=0; l<4; l++) z[l][i]=z[l][j]+f*z[l][k];}

/* geosm4 - multiply ith column of 4x4 matrix by a factor */

void geosm4(z,f,i) double z[4][4], f; int i; {int j;
for (j=0; j<4; j++) z[j][i]*=f;}

/* geoome4 - "one minus epsilon" for iterative */
/*      refinement of 4x4 inverse        */

void geoome4(z,x) double z[4][4], x[4][4]; {int i, j;
for (i=0; i<4; i++)
for (j=0; j<4; j++)
  if (i==j) z[i][j]=2.0-x[i][j]; else z[i][j]=-x[i][j];
}

/* geomp4 - product of two 4x4 matrices */

void geomp4(z,x,y) double z[4][4], x[4][4], y[4][4]; {
int i, j, k;
double u[4][4], v[4][4];
geocm4(u,x);
geocm4(v,y);
for (i=0; i<4; i++) for (j=0; j<4; j++) {
  z[i][j]=0.0;
  for (k=0; k<4; k++) z[i][j]+=u[i][k]*v[k][j];
  }
}
```

18 Future developments

Programs are never finished, a measure of their success is the demand for improvements. Several possibilities are evident.

18.1 Program structure; manual presentation

Naturally the technical details of many of the subroutines could be improved. So could the presentations included in this user's manual. In any event, experience shows that programs grow, amongst other ways, by a process which includes proliferating minor variants of some theme untill their number has become unwieldy. At that time common features of their structure will be sought out, the new structure defined and assigned parameters. The original collection will then be consolidated, only to begin the process anew. **GEOM** has, and will continue to, follow such a course.

18.2 Generating compounds from their names

From the viewpoint of chemists, **GEOM** still lacks the facilities present in the original Quantum Chemistry Program Exchange version, for generating all the atomic coordinates from the specification of bond distances, angles, and torsional parameters. Indeed, thought has always been given to creating the molecule from its chemical name, using a pattern recognition program, given that there are naming conventions from which the necessary information can be extracted.

18.3 Chemical data

As presented, **GEOM** is still very much a beginner's tool, especially with its emphasis on the details of sphere construction and the effect of varying various numerical parameters and programming techniques. For the working chemist, good use could be made of the incorporation of a vast store of chemical information, such as typical radii, angles, and so forth. But even at the elementary level, the variety of ways in which a sphere can be represented has hardly begun to be explored.

18.4 Stereopairs; front and side views

Stereopairs and other techniques are useful for visualizing three dimensional objects more comprehensively, as are animated presentations. Experience shows that the combination of a front view with a side view, both taken as parallel projections, is as effective as any when it comes to actually working with three dimensional objects. Display of such a pair of views has a high priority in the further elaboration of **GEOM**. Of course, the other presentations have artistic merit and contribute in their own way to visualizing objects, so their further development must also be considered.

18.5 Three-dimensional cutting and pasting

GEOM does not allow any user interaction with the objects displayed, even though extensive manipulation of their parameters is allowed. To construct a molecule from atoms located by cursor, to isolate radicals and drag them to new positions, or to "cut and paste" are all operations consistent with contemprorary two dimensional images; consideration should be given to extending them to three dimensions.

18.6 Projective geometry

Not to forget astronomers and eclipse watchers, the sun is hardly a point source and perspective is vital. But to orient **GEOM** towards projective geometry is an entirely different story, requiring a completely different program collection; similar as the outward appearance between the two might be. One day this project may be undertaken; it already has a basis in programs **PHOC** and **SHOC**.

19 Acknowledgements

The development of the programs in **GEOM** has spanned a major portion of the history of electronic computers, as a consequence of which it has been subject to many influences. Among the most recent have been the stimulus provided by the *Comité Universitario del Eclipse (1991)*, headed by Dr. Apolonio Juarez Nuñez, but we should also not fail to mention the benefits of having the use of a NeXT computer, with its operating system and graphics capabilities, which was provided to the University by a research contract (#P228CCO X904651) with Mexico's CONACYT.

References

[1] Noel Gastinel, *Linear Numerical Analysis*, Hermann, Paris, 1970.

[2] Henry George Forder, *The Calculus of Extension*, Chelsea Publishing Company, New York, 1960.

[3] H. Goldstein, *Classical Mechanics*, Addison-Wesley Publishing Company, 1950.

[4] Richard E. Pfiefer and Cathleen van Hook, "Circles, Vectors, and Linear Algebra," *Mathematics Magazine* **66** 75-86 (1993).