VIEW - A System for Prototyping Scientific Visualizations
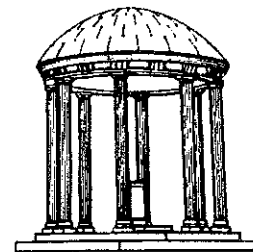
Lawrence D. Bergman

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175

This report is an excerpt from the dissertation, *VIEW - A System for Prototyping Scientific Visualizations,* by Lawrence D. Bergman. It is intended to serve as an introduction to the VIEW system, which is explained more fully in *VIEW: Exploratory Molecular Visualization System* (UNC Technical Report #93-030) and *VIEW Maintenance Guide* (UNC Technical Report #93-034).

Use of the VIEW System

This document presents the VIEW system through a series of examples. The first section presents use of VIEW drawing tools, mostly from the system-supplied tool library, to construct a presentation graphic. The second section introduces the drawing tool language through a development sequence, starting with a very simple tool and progressing through a series of more sophisticated versions. The third and fourth sections present examples of interactive events specified in the drawing tool language. The final section presents an example of the graphical debugging capabilities of the system.

4.1 AN EXAMPLE OF TOOL USE

Professor Jane Richardson of Duke University's Department of Biochemistry produced the visualization of the protein Concanavalin A shown in figure 4.1 using VIEW drawing tools. A variation of this image appeared in *Biophysical Journal* [Richardson 92a]. The image shows the orientation of an amino acid (phenylalanine) and two possible but less favorable orientations. The steps in constructing the visualization were as follows:

1) Richardson selected a tool that creates initial geometry starting with Brookhaven Protein Databank atomic coordinates. The tool in this case is one which creates vectors connecting just the alpha carbons of adjacent amino acids (figure 4.2). This representation gives a clear global view of the structure.

2) Using a mainchain drawing tool, she sketched in atomic level detail for each of three strands of the chain, each time merely selecting a starting
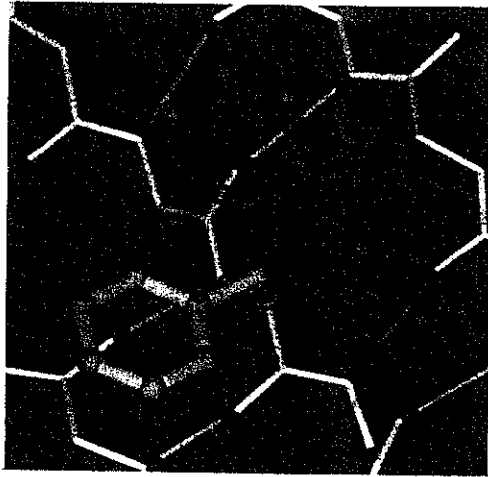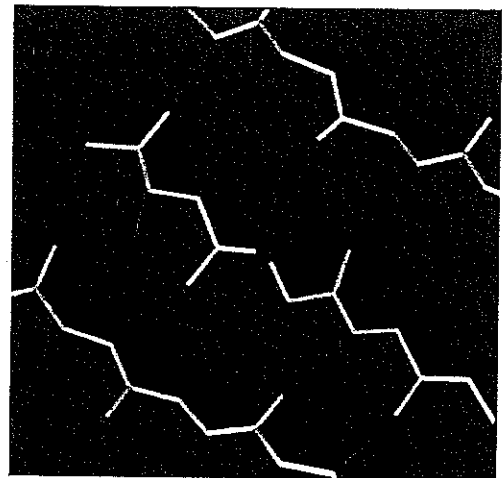
Figure 4.1

Final result of a visualization
construction sequence –
phenylalanine in beta sheet
showing alternate conformations
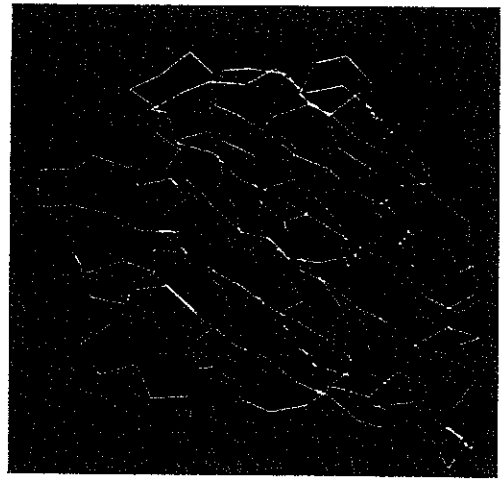
Figure 4.2

Initial geometry for
data-drawing
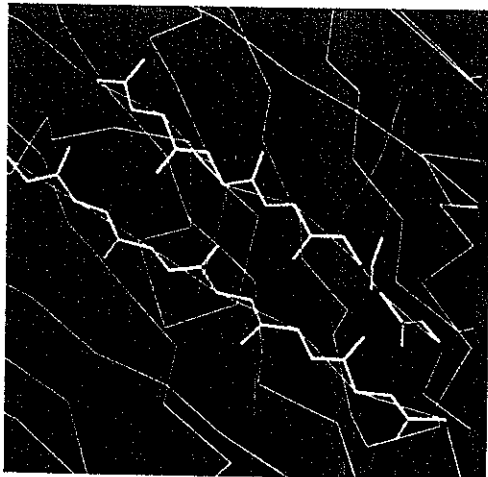
Figure 4.3

Drawing mainchain

Figure 4.4

Drawing in hydrogen bonds

and ending point by picking alpha-carbon atom positions in the original representation. On each pick, the system draws a small red sphere to mark the selection. After both ends are selected, the system draws the connecting mainchain at the atomic level automatically using atom coordinates fetched from the molecular data. After drawing the main chain, the tool removes the marker spheres. Figure 4.3 shows the drawing after specification of the second strand, just before the system removes the red markers. The use of database information by this tool makes it possible to produce this detailed scale drawing with only one tool selection and six datapoint selections.

3) Using a line drawing tool, she sketched in hydrogen bonds that couple the strands together. The particular tool employed knows nothing about hydrogen bonds; Richardson selected the termini of each bond. It would have been possible to write a new tool to automatically draw in all hydrogen bonds for the molecule, but since we wanted only a small, selected set, manual specification of each bond seemed reasonable.

The line-drawing tool bases each drawing operation on two atom selections. This time, the selections are made using the geometry produced in step 1; display of the original representation has been toggled off. In the figures that follow, geometry will often be turned off to simplify the display. VIEW users often do this – the interface allows individual groups of geometry to be toggled using virtual buttons in the interface. Users often switch between sparse global views and detailed local views.

The line-drawing tool allows the user to lengthen or shorten lines that connect the centers of selected atoms. An interactive event for defining the length scaling was triggered by pressing the "l" key. The event pops up a query window requesting a scaling factor. Richardson specified .65 before proceeding with the drawing shown in figure 4.4.

4) Next, she sketched a single sidechain using a sidechain drawing tool, which draws an entire sidechain based on one atom selection and information from the molecular database. Once the bonds of the sidechain were drawn, Richardson used a tool that draws marker spheres

at atom positions to identify each of the atoms in the sidechain. Figure 4.5 shows the result of a single selection with the sidechain tool and eight selections with the sphere marker tool.

5) In order to produce the two rotated positions of the sidechain, Richardson created a duplicate of the marker spheres using a group duplication tool. VIEW places the geometry created by each drawing tool in a separate *geometry group* labeled with the name of the tool that produced it. These groups may be individually manipulated by other tools. The duplication tool requested that she select any element of geometry from the group to be duplicated, and then queried for a name for the new group. This specification of group by identifying a member is common throughout VIEW.

6) Once the group had been duplicated, she used a rotation tool to rotate the duplicate into position. The rotation tool requests selection of a rotation axis, followed by selection of one or more geometry groups that are to be rotated. Rotation may then be performed with a dial or by key presses. In this case, Richardson wanted to rotate the group by a precise amount; an event triggered by pressing the "d" key allowed her to type in the desired angle (120 degrees), and then pressing the "r" key applied the rotation. Figure 4.6 shows the rotated markers. The rotation axis is highlighted in white.

7) Steps 4 and 5 were repeated to produce a second duplicate rotated to 240 degrees.

8) In order to generate the open framework of lines connecting markers at the rotated positions, I scripted a new drawing tool that connects a sequence of selected positions with wireframe cylinders. The tool was created by merging and modifying two prior tools. The first of these tools connected a sequence of positions with solid cylinders (using the system cylinder primitive). The other produced a tessellated cylinder (not using the system cylinder primitive) given two end points and a radius. Development of the new tool required under a half-hour, including testing. Figure 4.7 shows the open cylinders being drawn; the last two selected positions are highlighted white.

4
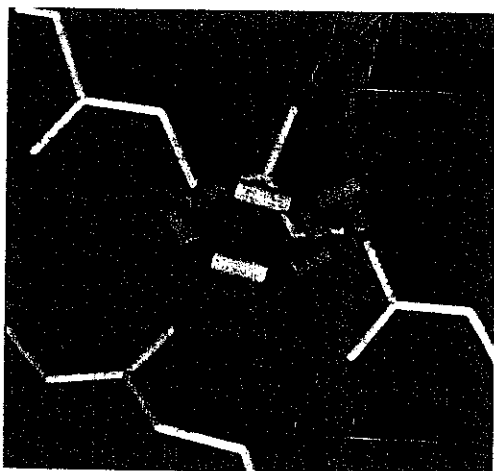
Figure 4.5

Phenylalanine drawn with all
atoms marked
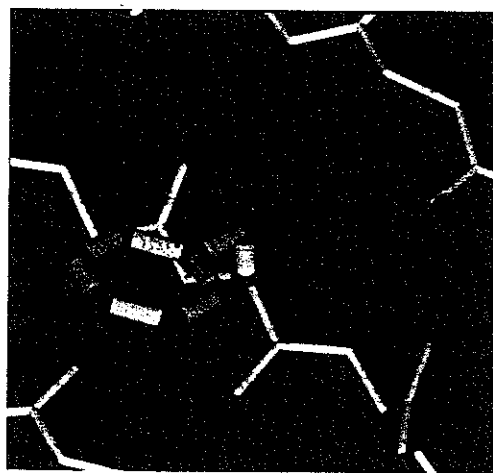
Figure 4.6

Atom markers duplicated and
rotated
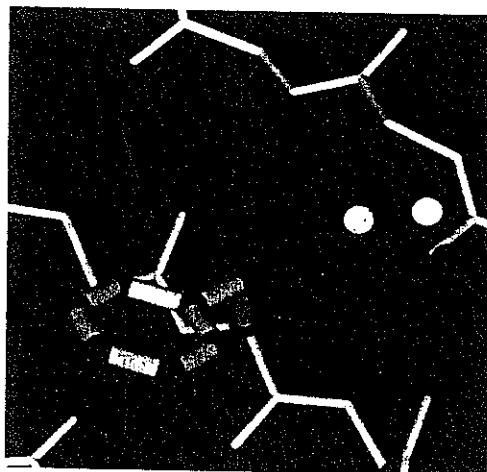
Figure 4.7

Sketching in wireframe cylinders

9) Richardson used a group recoloring tool to finalize colors in the image, shown in figure 4.1. The sequence of operations described can be carried out in about fifteen minutes (excluding the tool development in step 7). In actuality, Richardson spent a couple of hours producing the visualization. A large amount of trial-and-error is required to design a useful image. She tried a score of possibilities before settling on the above result, including: changing colors, radii, lengths, and number of facets for the wireframe cylinders.

4.2 AN EXAMPLE OF TOOL DEVELOPMENT

The following example describes the development of a drawing tool using the VIEW tool language. The example is a modified version of a tool developed with Alisa Wolberg, a graduate student in Darrel Stafford's lab in the Biology Department at UNC, Chapel Hill.

The final tool to be developed is one which displays spheres (constant-sized) for all atoms in a molecule that belong to residues of a specified type. I will develop the code gradually, in a series of steps. Each working tool will be a bit more complex. This is the way that we normally develop VIEW tools – by growing them.

Each code example will contain comments on selected lines that are of the form:

$$! \qquad (1)$$

The exclamation point is the VIEW comment character. Any text on that line following the exclamation is ignored by the VIEW interpreter. The number will indicate that a corresponding comment with that number follows the code, explaining the construct.

All keywords in the language are capitalized. Lower case words in the examples will be user-defined variable names, text strings, or attributes.

The first tool draws a single sphere at a selected atom location. The user clicks on some element of geometry that has an atom identifier associated with it. The sphere drawn is pink and 1.5 angstroms in radius.

```
sph_color = COLOR(255,60,60);   ! define the sphere color
sph_radius = 1.5;               ! define the sphere radius


SELECT_DB (item, "Select an atom");                  ! (1)
pnt = item.atom.position;                            ! (2)
sph = SPHERE (pnt, sph_radius, sph_color);           ! (3)
sph.db_ptr = item.atom;                              ! (4)
DISPLAY(sph);                    ! display the sphere
```

(1) The SELECT_DB statement indicates that the tool is to pause, waiting for the user to select an element of geometry that has a database pointer associated with it (in this case the database pointer is to an atom record). The statement will return the element of geometry selected in the variable *item*. The second argument is a message to be displayed at the top of the screen while the system awaits the selection.

(2) This statement retrieves the position field from the atom record in the database that is associated with the geometric element *item*. The position is assigned the variable name *pnt*.

(3) This statement constructs a sphere and assigns it to the variable *sph*.

(4) A database pointer to the atom record associated with *item* is assigned to the sphere *sph*.

The second version of the tool displays a sphere for each atom of the residue that is selected. The atom selection is used as a starting point in the database, and a search for all records with the same residue number is conducted, first forward in the database from the initial selection, and then backward. Note that this tool assumes that all atom records for a given residue are contiguous in the database (this is always the case for Brookhaven Protein Databank data). Code that was incorporated directly from the previous version is shown in plain type; new code is displayed in bold type.

7

```
sph_color = COLOR(255,60,60);    ! define the sphere color
sph_radius = 1.5;                ! define the sphere radius

SELECT_DB (item, "Select an atom");
rec = item.atom;                                           !  (1)
start_rec = rec;
res_number = rec.res_num;                                  !  (2)

! look forward through the database for atoms from same residue
WHILE ((rec != EOD) AND (rec.res_num == res_number))       !  (3)
{
  pnt = rec.position;                                      !  (4)
  sph = SPHERE (pnt, sph_radius, sph_color);
  sph.db_ptr = rec;
  DISPLAY(sph); ! display the sphere
  rec = NEXT_RECORD(rec);                                  !  (5)
}

! look backward through the database for atoms from same residue
! start one record prior to selection
rec = PREV_RECORD(start_rec);                              !  (6)
WHILE ((rec != EOD) AND (rec.res_num == res_number))
{
  pnt = rec.position;
  sph = SPHERE (pnt, sph_radius, sph_color);
  sph.db_ptr = rec;
  DISPLAY(sph); ! display the sphere
  rec = PREV_RECORD(rec);
}
```

(1) The variable name *rec* is assigned to the atom record associated with item.

(2) The *res_num* field, which contains the residue number, is retrieved from the atom record *rec* and assigned to the variable *res_number*.

(3) The tool will search forward through the database, retrieving one record at a time. The retrieval will continue as long as we are not at the end of the data, and the residue number for the current record is equal to the residue number of the initial selection. Note use of the C relational operators != (not equal to) and == (equal to).

(4) The atom position is retrieved from the current atom record, *rec*.

(5) This statement retrieves the next record in the database. The record that follows *rec* (indicated within the parentheses) will be assigned to *rec* (the variable to the left of the equals). Thus, the value of *rec* is updated to be the next record in the database. If *rec* was the last record in the database prior to this statement, *rec* will receive the value **EOD**.

(6) A block of code that searches backward through the database in exactly the same way as the forward search begins here. Note that we start the search with the record before the initial selection (**PREV_RECORD** gets the previous record in the database exactly as **NEXT_RECORD** gets the following record), to avoid drawing the initial selection twice.

The third version of the tool draws spheres for every atom of each arginine residue in an entire molecule. The user specifies the molecule by selecting some element of geometry from that molecule on-screen. We implement this tool by converting the previous tool into a subroutine, *Draw_residue*. The subroutine is passed an atom record and locates all other atoms in the same residue. Spheres are drawn for each atom. Note that unlike C, the user does not name a subroutine within the drawing-tool language. The name of the file containing the routine is used as the routine name; this name is assigned through the user interface. The subroutine is:

```
PARAMETERS (rec IN);                                    !   (1)

    sph_color = COLOR(255,60,60);   ! define the sphere color
    sph_radius = 1.5;               ! define the sphere radius

    start_rec = rec;
    res_number = rec.res_num;
```

```
! look forward through the database for atoms from same residue
WHILE ((rec != EOD) AND (rec.res_num == res_number))
{
  pnt = rec.position;
  sph = SPHERE (pnt, sph_radius, sph_color);
  sph.db_ptr = rec;
  DISPLAY(sph); ! display the sphere
  rec = NEXT_RECORD(rec);
}


! look backward through the database for atoms from same residue
  ! start one record prior to selection
rec = PREV_RECORD(start_rec);
WHILE ((rec != EOD) AND (rec.res_num == res_number))
{
  pnt = rec.position;
  sph = SPHERE (pnt, sph_radius, sph_color);
  sph.db_ptr = rec;
  DISPLAY(sph); ! display the sphere
  rec = PREV_RECORD(rec);
}
```

(1) This statement defines parameters for the subroutine. A single input parameter *rec* is passed to the routine. This is a database record, and substitutes for the record retrieved from the selected geometry in the previous version of the tool.

Note that this routine differs from the previous tool in two ways. First is the addition of the **PARAMETER** statement that defines *rec* as a parameter to be passed by the calling program. Second is the removal of the **SELECT_DB** statement and the statement following it which retrieves the database pointer from the element of geometry and assigns it to *rec*. Thus, *rec* is no longer specified through on-screen selection, but through a subroutine parameter.

The main program (i.e. the code for the tool itself) is:

10

```
SELECT_DB (item, "Select the molecule");
dbase = item.DB;                                    !   (1)


FOREACH (rec IN dbase.atom)                         !   (2)
{ IF ((rec.res_name == "ARG") AND                   !   (3)
      (rec.atom_wqual == "CA"))
            Draw_residue (rec);                     !   (4)
}
```

(1) The database associated with the selected element, *item*, is assigned the variable name *dbase*.

(2) This is an iterator (much like a C **FOR** loop or a FORTRAN **DO** loop). The atom records in the database assigned to *dbase* are to be iterated on. *rec* is the iteration variable. Each record in the atom portion of the database will be assigned to *rec* in turn, and the code within the braces will be executed for each.

(3) We wish to draw residue atoms for each arginine, but we must ensure that the atoms are drawn only once. For this reason only the alpha carbon (atom name CA) is passed to the residue drawing routine. This scheme is somewhat inefficient, since the *Draw_residue* routine will re-process database records scanned by the main routine. However, we have produced a very readable program, using already-built code.

(4) The subroutine that draws the residue is invoked with an alpha carbon record for each arginine.

The final version of the tool draws all residues of a specified type for a molecule. The molecule is selected once on-screen. The user then enters the residue type in a query, and that type is drawn. The user may repeat this process for as many residue types as she wishes. Each type is placed in a separate geometry group (each of which can be individually toggled, removed, renamed, written to file, or operated on by other drawing tools) identified by the residue type name. The *Draw_residue* subroutine was modified as follows and renamed *Draw_residue_grp*.

```
PARAMETERS (rec IN, grp IN_OUT);                                    !   (1)


  sph_color = COLOR(255,60,60);   ! define the sphere color
  sph_radius = 1.5;              ! define the sphere radius


  start_rec = rec;
  res_number = rec.res_num;


  ! look forward through the database for atoms from same residue
  WHILE ((rec != EOD) AND (rec.res_num == res_number))
  {
    pnt = rec.position;
    sph = SPHERE (pnt, sph_radius, sph_color);
    sph.db_ptr = rec;
    grp &= sph;                                                     !   (2)
    rec = NEXT_RECORD(rec);
  }


  ! look backward through the database for atoms from same residue
    ! start one record prior to selection
  rec = PREV_RECORD(start_rec);
  WHILE ((rec != EOD) AND (rec.res_num == res_number))
  {
    pnt = rec.position;
    sph = SPHERE (pnt, sph_radius, sph_color);
    sph.db_ptr = rec;
    grp &= sph;
    rec = PREV_RECORD(rec);
  }
```

(1) An extra argument has been added to the routine. *grp*, which is both an input and an output parameter (**IN_OUT**), stores the geometry group to which all spheres are to be added.

(2) The subroutine no longer displays the spheres, instead they are added to the geometry group, *grp*, using the concatenation operator, **&=**.

The main tool routine is:

```
res_type = "ARG";
SELECT_DB (item," Select the molecule");
dbase = item.DB;

LOOP                                                        !   (1)
{
  ASK_STRING ("Enter the type of residue to display:",
                  res_type);                                !   (2)
  res_grp = GROUP (res_type);                               !   (3)

  FOREACH (rec IN dbase.atom)
  { IF ((rec.res_name == res_type) AND (rec.atom_wqual == "CA"))
    { Draw_residue_grp (rec, res_grp);
      DISPLAY (res_grp);                                    !   (4)
    }
  }
}
```

(1) The **LOOP** statement specifies that the body (in braces) is to be performed repeatedly, without end. **LOOP**s are terminated by exiting the tool, either explicitly, by pressing **EXIT**, or implicitly by executing another tool. **LOOP** substitutes for the awkward construct **WHILE (1)** in C. **LOOP** is used here to permit multiple residue types to be displayed.

(2) **ASK_STRING** creates a user query that returns a string variable. The first argument is the prompt to be printed in the query. The second argument *res_type* is the variable that will contain the string entered by the user. **ASK_STRING** requires that this variable contain an initial value (assigned in the first line of the tool).

(3) This statement creates (or retrieves if it already exists), a geometry group with the name stored in the string variable *res_type*. The group is assigned to the variable *res_grp*.

(4) The group *res_grp* containing the spheres created by *Draw_residue_grp* must be displayed. I chose to display the group after each residue is processed rather than after all residues are processed (in which case the **DISPLAY** would be positioned outside the **FOREACH** loop).

4.3 EXAMPLE OF A TOOL CONTAINING A SIMPLE EVENT

In this section, we will examine a tool which contains an event triggered by dial rotation.

The tool (a simplified version of *change_radius_object* in the tool library) allows us to change the radius of a selected object by turning dial 7. Turning the dial clockwise will cause the radius to increase, counterclockwise to decrease.

```
SELECT (obj, "Select object whose radius is to be changed");   !   (1)


EVENT ("change_radius"; ON DIAL 7)                              !   (2)
{
  IF (EXISTS(obj))                                             !   (3)
  {  obj.RADIUS = obj.RADIUS +
                  (DIALRATE/50 * obj.RADIUS);                  !   (4)
    IF (obj.RADIUS < 0.01) obj.RADIUS = 0.01;                  !   (5)
    REDRAW();                                                   !   (6)
  }
}
```

(1) This is the single line that will be executed when the tool is started. The rest of the tool is an event definition, and will not be triggered until dial 7 is rotated.

(2) This is the header line for the dial event. The event will be named "change_radius" and will be triggered each time dial 7 is moved. On each trigger, the code within the braces will be executed.

(3) The **EXISTS** function returns **TRUE** if *obj* is defined, and returns **FALSE** otherwise. This test prevents access of *obj* before it is defined (*obj* will be undefined until the SELECT statement has completed).

14

(4) This statement modifies the radius of the selected object based on the amount of dial movement. DIALRATE is a system variable which measures the amount that the dial has been rotated since the last event execution.

(5) This statement ensures that the radius of *obj* will always be positive.

(6) Once the radius of *obj* has changed, the screen is updated.

4.4 EXAMPLE OF A TOOL CONTAINING A CONDITIONAL EVENT

The tool presented in this section uses a conditional event to determine if a user-steerable cursor is near any residue of a molecule. If it is, all atoms for that residue are drawn as spheres using the *Draw_residue* subroutine described above. The routine uses only alpha carbon locations to determine proximity.

After the user specifies the molecule to be processed by selecting an element from it, the routine scans the database searching for all alpha carbons of that molecule. Each alpha carbon position in added to an array (using the concatenation operator, &=), and the atom record is stored in a corresponding array. A conditional event checks the distance between a user-defined cursor (*probe*) location and each of the alpha carbon positions. A 3-dimensional search function, SEARCH3D is used to perform this check. If any points fall within the specified distance from the probe, the residue is drawn and then that residue's alpha carbon is removed from the search list.

A set of events defines movement of the spherical cursor. The "x", "y", and "z" keys cause movement in the horizontal, vertical, and in-out-of-screen directions respectively. Pressing the "n" key causes reversal of movement for all of these keys (e.g. movements to the left become movements to the right).

```
SELECT_DB (item, "Select the molecule");
dbase = item.DB;


! store alpha carbon atom positions in atom_pos array, and
! corresponding atom records in atom_records
atom_pos = ARRAY();                                    !   (1)
```

15

```
atom_records = ARRAY();
FOREACH (atom_rec IN dbase.atom)
{
  IF (atom_rec.atom_wqual == "CA")
  { atom_pos &= atom_rec.position;   ! add the atom position to the array
    atom_records &= atom_rec;
  }
}


! place probe at the center of the screen, define movement events for it
probe = SPHERE (ORIGIN, 2.0, COLOR(255,0,0));
DISPLAY (probe);


move_amt = .5;
EVENT ("toggle_dir"; ON KEY "n")
{ move_amt = -move_amt; }


EVENT ("move_x"; ON KEY "x")
{ TRANSLATE_SCREEN (probe, move_amt, 0, 0);            !   (2)
  REDRAW();
}


EVENT ("move_y"; ON KEY "y")
{ TRANSLATE_SCREEN (probe, 0, move_amt, 0);
  REDRAW();
}


EVENT ("move_z"; ON KEY "z")
{ TRANSLATE_SCREEN (probe, 0, 0, move_amt);
  REDRAW();
}


! define the conditional event that controls residue display
EVENT ("search_side"; (SEARCH3D (1, atom_pos, probe.CENTER, 3.0,
                found_pnts; INDEX = indices)))            !   (3)
{
```

```
        ndx = indices[0];
        rec = atom_records[ndx];                          !   (4)
        Draw_residue(rec);                                !   (5)


        ! remove this alpha carbon from the search list
        REMOVE(atom_records[ndx]);
        REMOVE(atom_pos[ndx]);
    }
```

(1) An empty array is defined.

(2) The TRANSLATE_SCREEN command specifies movement in the plane of the screen for the named geometry. In this case, *probe* is to be moved in the "x" direction by *move_amt* units (angstroms).

(3) The SEARCH3D function is the condition to be checked for each evaluation of this conditional event. This system-defined function returns TRUE if any point in the array *atom_pos* is within 3.0 angstroms of the center of the cursor sphere *probe*. The first argument specifies that only a single such point is to be located, which will be returned in an array *found_pnts*. If a point is located, the array *indices* will contain its index in the original array *atom_pos* (a one-to-one correspondence exists between *found_pnts* and *indices*).

(4) The first element (index number zero) in *indices* is the index number of the atom position found in *atom_pos* and of the corresponding atom record in *atom_rec*.

(5) Call the residue-drawing routine of section 4.2, passing it the alpha carbon record for the residue.

4.5 AN EXAMPLE OF GRAPHICAL DEBUGGING

This example shows some of the graphical debugging features of the VIEW system. Graphical debugging provides a capability for linking the textual representation of a tool with an on-screen graphical representation. This example shows how geometric computations performed in a section of code

were traced, using the graphical display facility in VIEW to create representations of geometric entities on-screen and/or to highlight in the code the line displaying an entity selected on-screen.

The code for this example is a portion of the *Ribgen* subroutine, invoked by both the *ribbon* and *ribbon_select* tools. *Ribgen* creates ribbon geometry from a list of alpha carbon and carbonyl oxygen positions assembled by the calling routine. After the ribbon had been in use for a while, I noted that the ends of the ribbons were unusually narrow, and that the ribbons appeared somewhat kinked. In order to see what the code was doing, I set a breakpoint at the position indicated in the following code segment:

```
FOR (i=0; i<len-1; i=i+1)
{
    a = VECTOR(ca_list[i+1],ca_list[i]);        <——
    b = VECTOR(ca_list[i], o_list[i]);
    c = CROSS(b,a);
    d = CROSS(c,a);
    d_list[i] = d/SQRT(d*d);
    pnt = (ca_list[i] + ca_list[i+1]) / 2.;
    p_list[i] = pnt;

        . . .
}
```

I executed the *ribbon_select* tool and selected a start and end atom for the ribbon. The debugger containing the *Ribgen* code popped up with the breakpoint line highlighted, indicating that the debugger was stopped at that line.

In order to see graphical display of the computations in this code block, I clicked on the **Construct** button in the debugger. This turned on the construction facility, a graphical debugging feature that automatically produces on-screen displays of points and vectors as they are created by the currently executing tool. As I stepped through the code, one statement at a time, white cylinders appeared on-screen representing the vectors a, b, c, d and a sphere appeared representing the point *pnt*.

18

At any time after a point or vector was created, I was able to highlight it by selecting the name of the variable in the debugger window and then clicking on **Display**. This caused the representation of the variable to change color (if it already existed on-screen) or to be drawn using the same representations as the construction facility.

Figure 4.8 shows the on-screen display after this block of code has been stepped through once. The red spheres show the atom selections. The cylinders and white dots were produced by the construction facility and represent variables as labeled (the labels shown in the figure were added as a post-process). The cylinder representing the variable a was turned red by **Displaying** that variable.

So far, the vectors and computed point were positioned as I expected. I turned construction off (by clicking the **Construct** button), set a new breakpoint further on in the code, and clicked on **Continue**. When the new breakpoint was reached, I again turned construction on, and continued to step through the following block of code:

```
IF (i == 0)    ! handle start of ribbon
{
    d_add = d_list[0]*w;                      <———

    end1_list &= ca_list[0] + d_add;                    !   (1a)
    end2_list &= ca_list[0] - d_add;                    !   (1b)


    ! replicate start point for spline
    end1_list &= ca_list[0] + d_add;                    !   (1a)
    end2_list &= ca_list[0] - d_add;                    !   (1b)


    end1_list &= pnt + d_add;                           !   (2a)
    end2_list &= pnt - d_add;                           !   (2b)
}
end1_list &= (p_list[i+1] + d_list[i+1]*w);             !   (3a)
end2_list &= (p_list[i+1] - d_list[i+1]*w);             !   (3b)
```

The graphics produced after stepping through this code is shown in figure 4.9. Each point is labeled to correspond with the code line that produced it. The

display feature was used to display *a* (turning it back to white), and *d_add* creating the red and white striped cylindrical segment (the striping is an artifact caused by the way the SGI renders coincident geometric primitives; *d_add* is collinear with *d*).

The vector *d_add* was positioned as I expected, but was not as long as I had anticipated. Thus, the six points drawn in this code segment were nearer the original atom positions than I wished. I printed the value of *w* (using **Display**), and noted that it was too small. With a bit of additional code browsing, I located an error in the *w* calculation.

BIBLIOGRAPHY

[Richardson 92a] J.S. Richardson et al, "Looking at Proteins: Representations, Folding, Packing, and Design", *Biophysical Journal*, Vol. 63, Nov. 1992, pp. 1186-1209.
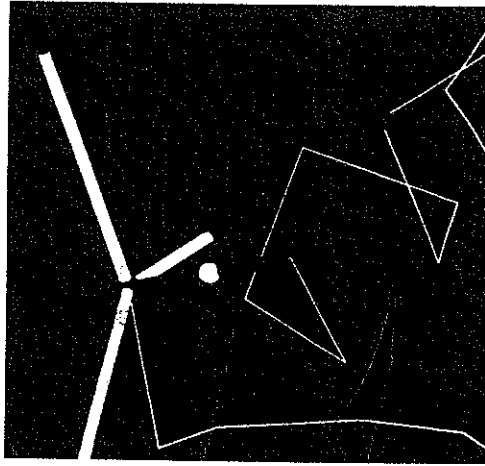
Figure 4.8

Graphical debugging – after
first code block
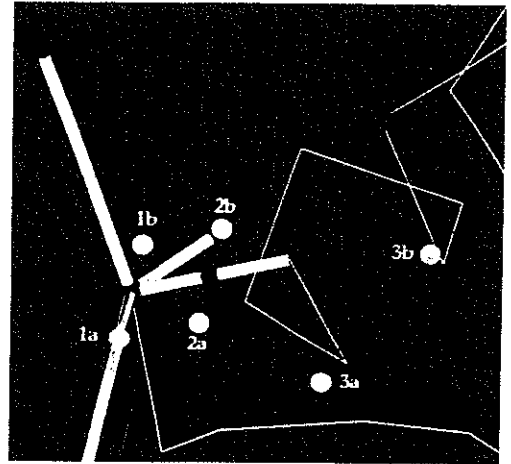
Figure 4.9

Graphical debugging – after
second code block