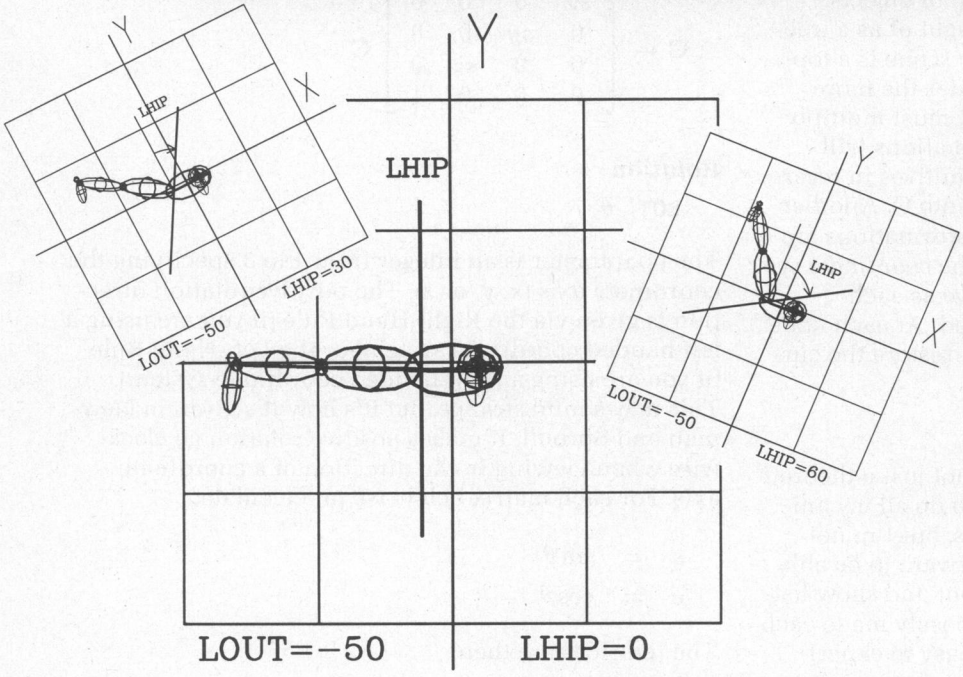
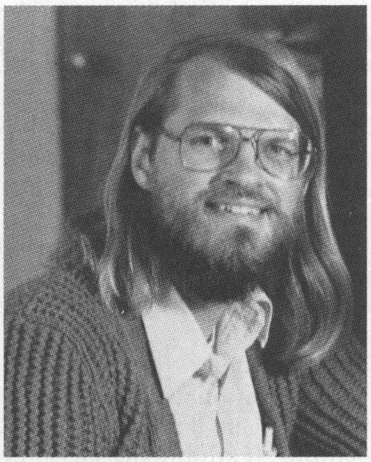
Jim Blinn's Corner





Nested Transformations and Blobby Man

James F. Blinn, Jet Propulsion Lab, Caltech

There are a lot of interesting things you can do with transformation matrices. Future columns will deal with this quite a bit, so I will spend some time here describing my notational scheme for nested transformations. As a nontrivial example I will include the database for an articulated human figure called *Blobby Man*. (Those of you who already know how to do human articulation, don't go away; there are some cute tricks that are very useful.)

This is all pretty standard stuff. That is an advantage of writing a column: You don't have to make sure everything is new. The ideas are not new; I'm just defining a notation.

The mechanism

This is an implementation of the well-known technique of nested transformations. (Don't you just hate it when somebody calls something *well known* and you have never heard of it? It sounds like they are showing off how many things they know. Well, admittedly we can't derive *everything* from scratch. But it sure would be nice to find a less smug way of saying so.) For those to whom this is not so "well known," the basic idea behind nested transformations appears in several places, notably in Foley and van Dam (*Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Mass.) and in Glassner (*Computer Graphics User's Guide*, Sams and Co., Indianapolis, Ind.). It is just an organizational scheme to make it easier to deal with a hierarchy of accumulated transformations. It shows up in various software systems and has hardware implementations in the E&S Picture System or the Silicon Graphics IRIS.

Briefly, it works like this: We maintain a global 4×4 homogeneous coordinate transformation matrix called the *current transformation*, C, containing the transformation from a primitive's "definition space" onto a desired location in "screen space." I will assume a "device independent" (buzz, buzz) screen space ranging from −1 to +1 in x and *y* and where z goes *into* the screen. This is a *left-handed* coordinate system.

Each time a primitive is drawn, it is implicitly transformed by C. For example, the transformation of a (homogeneous) point is accomplished by simple matrix multiplication.

$$[x, y, z, w]_{scrn} = [x, y, z, w]_{defn}\,\mathbf{C}$$

Other primitives are transformed by some other arithmetic using this matrix.

C is typically the product of a perspective transformation and various rotations, translations, and scales. It is built up with a series of matrix multiplications by

0272-1716/87/1000-0059$01.00 © 1987 IEEE

simpler matrices. Each multiplication premultiplies a new matrix into C.

$$\mathbf{C} \leftarrow \mathbf{T}_{new}\ \mathbf{C}$$

Why in this order? Because a collection of objects, subobjects, sub-subobjects, etc., is thought of as a tree-like structure. Drawing a picture of the scene is a top-down traversal of this tree. You encounter the more global of the transformations first, and must multiply them in as you see them. The transformations will therefore seem to be applied to the primitives in *reverse* order to the way they were mulitplied into C. Another way you can think of it is that the transformations are applied in the same order stated, but the *coordinate system* transforms along with the primitive as each elementary transformation is multiplied. At each node in the tree, of course, you can save and restore the current contents of C on a stack.

The language

The notational scheme I will use is not just a theoretical construct; it's what I actually use to do all my animations. It admittedly has a few quirks, but I'm not going to try to sanitize them because I want to be able to use databases I have actually tried out and show listings that I know will work. I have purposely made each operation *very* elementary to make it easy to experiment with various combinations of transformations. Most reasonable graphics systems use something like this, so it shouldn't be too hard for you to translate my examples into your own language.

Instructions for rendering a scene take the form of a list of commands and their parameters. These will be written here in "typewriter" type. All commands will have four or fewer letters. (The number 4 is used because of its ancient numerological significance.) Parameters will be separated by commas, not blanks. (Old-time Fortran programmers don't even *see* blanks, let alone use them as delimiters.) Don't complain; just be glad I'm not using "O language." (Maybe I'll tell you about that sometime.)

Basic command set

These commands modify C and pass primitives through it. Each modification command premultiplies some simple matrix into C. No other action is taken. The command descriptions below will explicitly show the matrices used.

Translation

TRAN x, y, z

Premultiplies C by an elementary translation matrix.

$$\mathbf{C} \leftarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & y & z & 1 \end{bmatrix} \mathbf{C}$$

Scaling

SCAL sx, sy, sz

premultiplies C by an elementary scaling matrix.

$$\mathbf{C} \leftarrow \begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{C}$$

Rotation

ROT θ, j

The j parameter is an integer from 1 to 3 specifying the coordinate axis (x, y, or z). The positive rotation direction is given via the Right-Hand Rule (if you are using a left-handed coordinate system) or the Left-Hand Rule (if you are using a right-handed coordinate system). This may sound strange, but it's how it's given in Newman and Sproull. It makes *positive* rotation go *clockwise* when viewing in the direction of a coordinate axis. For each matrix below we precalculate

$$s = \sin\theta$$
$$c = \cos\theta$$

The matrices are then

$$\mathbf{C} \leftarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{C}$$

$j = 2$ (y axis)

$$\mathbf{C} \leftarrow \begin{bmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{C}$$

$j = 3$ (z axis)

$$\mathbf{C} \leftarrow \begin{bmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{C}$$

Perspective

PERS α, z_n, z_f

This transformation combines a perspective distortion with a depth (z) transformation. The perspective assumes the eye is at the origin, looking down the (+ z) axis. The field of view is given by the angle α.

The depth transformation is specified by two values—z_n (the location of the *near* clipping plane), and z_f (the location of the *far* clipping plane). The matrix

transforms z_n to +0, and z_f to +1. I know that the traditional names for these planes are *hither* and *yon*, but for some reason I always get these words mixed up, so I call them *near* and *far*.

Precalculate the following quantities (note that far clipping can be effectively disabled by setting $z_f = \infty$ which makes $Q = s$).

$$\begin{aligned} s &= \sin \alpha/2 \\ c &= \cos \alpha/2 \\ Q &= \frac{s}{1 - z_n/z_f} \end{aligned}$$

The matrix is then

$$\mathbf{C} \leftarrow \begin{bmatrix} c & 0 & 0 & 0 \\ 0 & c & 0 & 0 \\ 0 & 0 & Q & s \\ 0 & 0 & -Qz_n & 0 \end{bmatrix} \mathbf{C}$$

Orientation

ORIE *a, b, c, d, e, f, p, q, r*

Sometimes it's useful to specify the rotation (orientation) portion of the transformation explicitly. There is nothing, though, to enforce it being a pure rotation so it can be used for skew transformations.

$$\mathbf{C} \leftarrow \begin{bmatrix} a & d & p & 0 \\ b & e & q & 0 \\ c & f & r & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{C}$$

Transformation stack

```
PUSH
POP
```

These two commands push and pop C on and off the stack.

Primitives

DRAW *name*

A primitive could be a list of vector endpoints, points-and-polygons, implicit surfaces, cubic patches, blobbies, etc. This command means "pass the elements in primitive *name* (however it's defined) through C and onto the screen."

Example

A typical scene will consist of an alternating sequence of C-alteration commands and of primitive-drawing commands. At the beginning of the command list, C is assumed to be initialized to the identity matrix. Here is a typical sequence of commands to draw a view of two cubes sitting on a grid plane. The

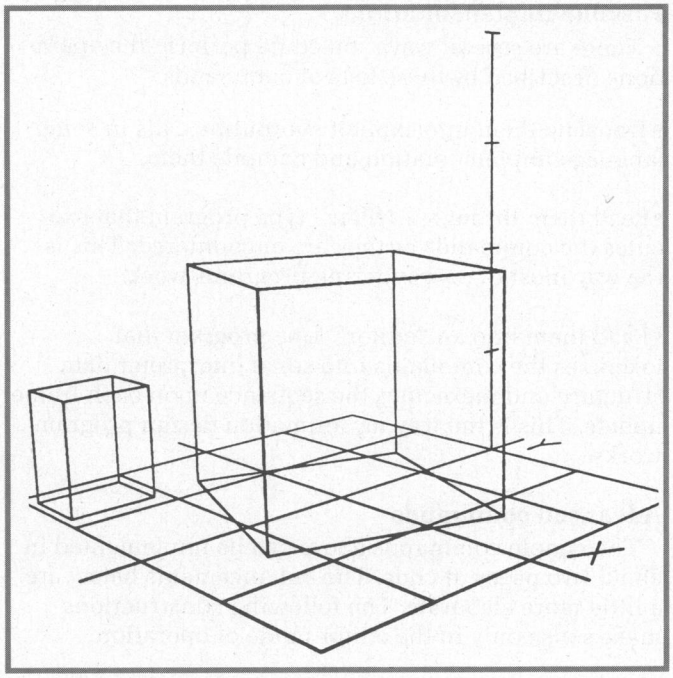


Figure 1. Cubes on parade.

primitive GPLANE consists of a grid of lines in the xy plane covering −2 to +2 along each axis, along with some labels and a tic-marked pole in the +z direction, which is placed at $y = 2$. The primitive CUBE consists of a cube whose vertices have coordinates $[\pm 1, \pm 1, \pm 1]$—that is, it is centered at the origin and has edge length equal to 2. Notice the scale by −1 in z to convert from the right-handed system in which the scene is defined to the left-handed system in which it is rendered.

```
PERS  45, 6.2, 11.8
TRAN  0, -1.41, 9
ROT   -80, 1
ROT   48, 3
SCAL  1, 1, -1
DRAW  GPLANE
PUSH
TRAN  0,0,1
ROT   20,3
DRAW  CUBE
POP
PUSH
SCAL  .3,.4,.5
TRAN  -5,-3.8,1
DRAW  CUBE
POP
```

The results of executing these instructions appear in Figure 1.

Notice that the z_n and z_f variables are selected to bound the scene as closely as possible, so that depth cuing will work. And, hey, it's called depth *cuing* not depth *queuing* as I've heard some people say. (Depth queuing *could* perhaps be used to refer to a depth-priority rendering algorithm...Hmmm.)

Possible implementations

There are several ways you could perform the operations described by these lists of commands.

•Translate them into explicit subroutine calls in some language implementation and compile them.

•Read them through a "filter" type program that executes the commands as they are encountered. This is the way most of my rendering programs work.

•Read them into an "editor" type program that tokenizes the commands into some interpreter data structure and reexecutes the sequence upon each frame update. This is the way my animation design program works.

Advanced commands

The simple commands above can be implemented in about two pages of code. The enhancements below are a little more elaborate. The following constructions make sense only in the editor mode of operation.

Parameters

Any numeric parameter can be given a symbolic name. A symbol table will be maintained and the current numeric value of the symbol used when the instruction is executed. For example, our cube scene could be

```
PERS  FOV, ZN, ZF
TRAN  XSCR, YSCR, ZSCR
ROT   BACK, 1
ROT   SPIN, 3
SCAL  1, 1, -1
DRAW  GPLANE
PUSH
TRAN  X1,Y1,Z1
ROT   ANG,3
DRAW  CUBE
POP
PUSH
SCAL  .3,.4,.5
TRAN  -5,-3.8,Z1
DRAW  CUBE
POP
```

By setting the variables

```
FOV=45      ZN=6.2       ZF=11.8
XSCR=0      YSCR=-1.41   ZSCR=9
BACK=-80    SPIN=48
X1=0        Y1=0         Z1=1
ANG=20
```

and executing the command list, the same results would be generated. The same symbol can appear in more than one place, allowing a certain amount of constraint satisfacton.

Abbreviations

Each time a subobject is positioned relative to a containing object, the instructions usually look something like

```
PUSH
:
various TRAN, ROT, SCAL commands
:
DRAW primitive
POP
```

While explicit, the above notation is sometimes a bit spread out and hard to follow. This sort of thing happens so often that it's helpful to define an abbreviation for it. We do so by following the DRAW command (on the same line) with the list of transformation commands, separated by commas. An implied PUSH and POP encloses the transformation list and DRAW. Our cube scene now looks like

```
PERS  FOV, ZN, ZF
TRAN  XSCR, YSCR, ZSCR
ROT   BACK, 1
ROT   SPIN, 3
SCAL  1, 1, -1
DRAW  GPLANE
DRAW  CUBE, TRAN,X1,Y1,Z1, ROT ,ANG,3
DRAW  CUBE, SCAL,.3,.4,.5, TRAN,-5,-3.8,Z1
```

Subassembly definitions

These are essentially subroutines. A subassembly is declared and named by bracketing its contents with the commands

```
DEF name
:
any commands
:
----
```

Once defined, a subassembly can be thought of as just another primitive. In fact, the "designer" of a list of commands should not know or care if the thing being drawn is a primitive or a subassembly, so a subassembly is "called" by the same command as a primitive.

```
DRAW assy_name
```

The subassembly calling and return process is completely independent of the matrix stack PUSH and POP process. Interpretation of commands begins at the built-in name WORLD. I typically organize my definitions so that WORLD contains only the *viewing transformation*, i.e., its rotations and transformations tell where the "camera" is and in which direction it is looking. My favorite all-purpose viewing transform is

```
DEF   WORLD
PERS  FOV, ZN, ZF
TRAN  XSCR, YSCR, ZSCR
ROT   BACK, 1
ROT   SPIN, 3
ROT   TILT, 1
TRAN  -XLOOK, -YLOOK, -ZLOOK
SCAL  1, 1, -1
DRAW  SCENE
----
```

The variables XLOOK, YLOOK, and ZLOOK determine the "look-at" point. BACK, SPIN, and TILT tumble the scene about this point. Then XSCR, YSCR, and ZSCR position the look-at point on the screen. XSCR and YSCR might very well be zero, but ZSCR needs to be some positive distance to move the scene away from the eye.

The assembly SCENE contains the contents of the scene, and can be designed independently of how it is being viewed. Our cube scene again:

```
DEF SCENE
DRAW  GPLANE
DRAW  CUBE, TRAN,X1,Y1,Z1, ROT ,ANG,3
DRAW  CUBE, SCAL,.3,.4,.5, TRAN,-5,-3.8,Z1
----
```

Blobby Man

A few years ago I made a short animation of a human figure called "Blobby Man" to illustrate a new surface modeling technique. Leaving aside issues of modeling, the figure itself is an interesting example of nested transformations. I have, in fact, used it as a homework assignment for my computer graphics class. (Gee, I guess I can't do that any more.)

Here is Blobby Man. His origin is in his stomach, and he stands with the z axis vertical. The only primitive element is a unit radius SPHERE centered at the origin. The parameterized variables are all rotation angles. Their usage is defined in Table 1.

The WORLD is the standard one given above. SCENE looks like

```
DEF SCENE
DRAW  GPLANE
DRAW  TORSO , TRAN,XM,YM,ZM, ROT ,RZM,3,
----
```

The actual articulated parts are in Figure 2. Some primitive body parts are defined as translated and squashed spheres in Figure 3. A picture of the result appears in Figure 4. The viewing parameters are

```
ZN=5.17      ZF=10.7
XSCR=-.1     YSCR=-1.6     ZSCR=7.9
BACK=-90     SPIN=-30      TILT=0
XLOOK=0      YLOOK=0       ZLOOK=0
XM=0         YM=0          ZM=1.75
```

Table 1. Meanings of Blobby Man variables

EXTEN	Extension. A dancers term for bending forward and backwards (x axis)
ROT	Rotation. A dancers term for rotating the body and shoulders left and right about the vertical (z axis)
BTWIS	Angle of body leaning left and right (y axis)
NOD	Head nod
NECK	Head shake
LHIP,RHIP	Angular direction that the leg is kicked
LOUT,ROUT	Angular distance that the leg is kicked
LTWIS,RTWIS	Angle the leg is twisted about its length
LKNEE,RKNEE	Knee bend
LANKL,RANKL	Ankle bend
LSID,RSID	Arm rotation to side
LSHOU,RSHOU	Arm rotation forward and back
LATWIS,RATWIS	Arm rotation about its own length
LELBO,RELBO	Elbow angle

```
DEF TORSO
DRAW  LEFTLEG , TRAN,-0.178,0,0,
DRAW  RGHTLEG , TRAN,0.178,0,0,
DRAW  SPHERE  , TRAN,0,0,0.08, SCAL,0.275,0.152,0.153,
DRAW  BODY    , ROT,EXTEN,1, ROT,BTWIS,2, ROT,ROT,3,
----

DEF BODY
DRAW  SPHERE  , TRAN,0,0,0.62, SCAL,0.306,0.21,0.5,
DRAW  SHOULDER, TRAN,0,0,1, ROT,EXTEN,1, ROT,BTWIS,2, ROT,ROT,3,
----

DEF SHOULDER
DRAW  SPHERE  , SCAL,0.45,0.153,0.12,
DRAW  HEAD    , TRAN,0,0,0.153, ROT,NOD,1, ROT,NECK,3,
DRAW  LEFTARM , TRAN,-0.45,0,0, ROT,LSID,2, ROT,LSHOU,1, ROT,LATWIS,3,
DRAW  RGHTARM , TRAN, 0.45,0,0, ROT,RSID,2, ROT,RSHOU,1, ROT,RATWIS,3,
----

DEF LEFTLEG                    DEF RGHTLEG
PUSH                           PUSH
ROT   LHIP, 3,                 ROT   RHIP, 3,
ROT   LOUT, 2,                 ROT   ROUT, 2,
ROT  -LHIP, 3,                 ROT  -RHIP, 3,
ROT   LTWIS, 3,                ROT   RTWIS, 3,
DRAW  THIGH   ,                DRAW  THIGH   ,
TRAN  0, 0, -0.85,             TRAN  0, 0, -0.85,
ROT   LKNEE, 1,                ROT   RKNEE, 1,
DRAW  CALF    ,                DRAW  CALF    ,
TRAN  0, 0, -0.84,             TRAN  0, 0, -0.84,
ROT   LANKL, 1                 ROT   RANKL, 1
DRAW  FOOT                     DRAW  FOOT
POP                            POP
----                           ----

DEF LEFTARM                    DEF RGHTARM
PUSH                           PUSH
DRAW  UPARM                    DRAW  UPARM
TRAN  0, 0, -0.55,             TRAN  0, 0, -0.55,
ROT   LELBO, 1,                ROT   RELBO, 1,
DRAW  LOWARM                   DRAW  LOWARM
TRAN  0, 0, -0.5,              TRAN  0, 0, -0.5,
DRAW  HAND                     DRAW  HAND
POP                            POP
----                           ----
```

Figure 2. Body of Blobby Man.

```
DEF HEAD
DRAW  SPHERE  , TRAN,0,     0,0.4  , SCAL,0.2   ,0.23  ,0.3,
DRAW  SPHERE  , TRAN,0,-0.255,0.42 , SCAL,0.035 ,0.075 ,0.035,
DRAW  SPHERE  , TRAN,0,     0,0.07 , SCAL,0.065 ,0.065 ,0.14,
DRAW  SPHERE  , TRAN,0,-0.162,0.239, SCAL,0.0533,0.0508,0.0506,
----

DEF UPARM
DRAW  SPHERE  , TRAN,0,0,-0.275, SCAL,0.09,0.09,0.275,
----

DEF LOWARM
DRAW  SPHERE  , TRAN,0,0,-0.25, SCAL,0.08,0.08,0.25,
----

DEF HAND
DRAW  SPHERE  , TRAN,0,0,-0.116, SCAL,0.052,0.091,0.155,
----

DEF THIGH
DRAW  SPHERE  , TRAN,0,0,-0.425, SCAL,0.141,0.141,0.425,
----

DEF CALF
DRAW  SPHERE  , SCAL,0.05,0.05,0.05,
DRAW  SPHERE  , TRAN,0,0,-0.425, SCAL,0.1,0.1,0.425,
----

DEF FOOT
DRAW  SPHERE  , SCAL,0.05,0.04,0.04,
DRAW  SPHERE  , TRAN,0, 0.05,-0.05, SCAL,0.04,0.04,0.04,
DRAW  SPHERE  , TRAN,0,-0.15,-0.05, ROT,-10,1, SCAL,0.08,0.19,0.05,
----
```

Figure 3. Body parts.

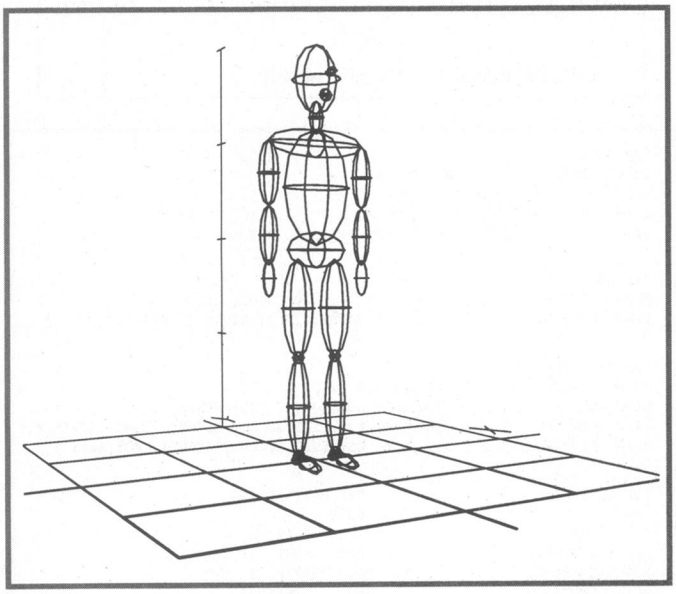


Figure 4. Blobby Man.

All other angles are zero. A picture of the man gesturing is in Figure 5. The view is the same, but the body angles are

```
NOD=-25   NECK=28
RHIP=105  ROUT=13   RTWIS=-86   RKNEE=-53
LHIP=0    LOUT=0    LTWIS=0    LKNEE=0
LSID=-45  LSHOU=0   LATWIS=-90  LELBO=90
RSID=112  RSHOU=40  RATWIS=-102 RELBO=85
```

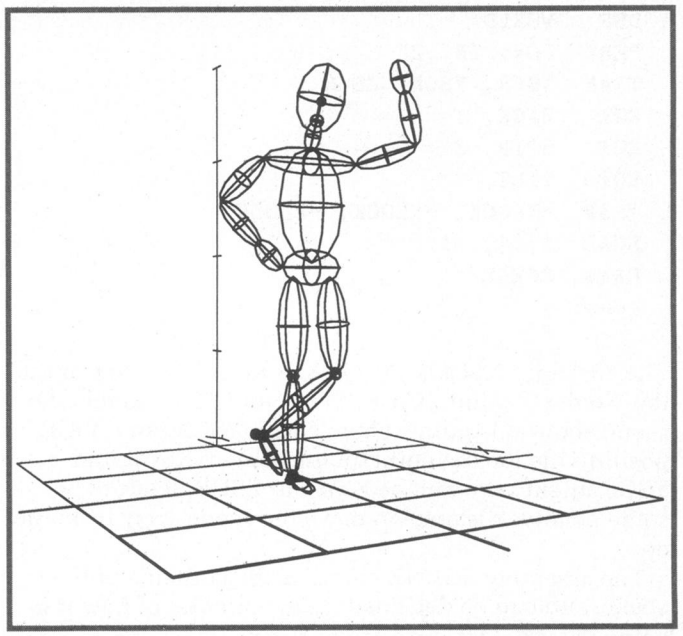


Figure 5. Blobby Man waving.

There are several tricks in the model of Blobby Man that are especially notable.

Cumulative transformations

It is not necessary to POP a transformation just after it is used to DRAW something. Sometimes it is useful to accumulate translations and rotations continuously. For example, Blobby Man's leg could have looked like

```
DEF  LLEG
DRAW  THIGH    ,
DRAW  CALFETC , TRAN,0,0,-0.85, ROT,LKNEE,1
----

DEF CALFETC
DRAW CALF     ,
DRAW FOOT     , TRAN,0,0,-0.84, ROT,LANKL,1
----
```

As long as there are no transformed objects after the last one, some of the nesting can be dispensed with, leaving...

```
DEF LLEG
PUSH
DRAW  THIGH    ,
TRAN  0, 0, -0.85,
ROT   LKNEE, 1,
DRAW  CALF     ,
TRAN  0, 0, -0.84,
ROT   LANKL, 1
DRAW  FOOT
POP
----
```

Repeated variables

The variables EXTEN, BTWIS, and ROT are used twice, once to flex the BODY relative to the TORSO and once to flex the SHOULDER relative to the BODY. This gives a minimal simulation of a flexible spine for the figure.

Rotated rotations

The transformation of the (left) leg relative to the torso contains the sequence

```
ROT   LHIP, 3
ROT   LOUT, 2
ROT  -LHIP, 3
```

This is something I'm especially proud of. It is a not-completely-obvious variation of a common technique—using simple transformations to build rotations or scalings about points other than the origin. For example, if you wanted to rotate a primitive about a point at coordinates (DX, DY), the commands would be

```
TRAN  DX, DY, 0
ROT   ANGLE,3
TRAN -DX,-DY, 0
```

In other words, you translate the desired rotation center to the origin, rotate, and then translate the center back to where it used to be. (Remember that the transformations will be effectively carried out in sequence in the *reverse* order from that seen above.) The rotation sequence used for the leg enables us to rotate the leg about a rotated coordinate axis. The purpose of this is to make the foot always point forward no matter what LHIP and LOUT are. Figure 6 shows how this works. It is a top view of just the legs and hips; the dark line shows the axis of rotation by the angle LOUT. A similar technique could have been used for the arm and shoulder joints, but I didn't happen to need that much flexibility in the animation.

Reminder

Cut this column out and put it in your scrapbook so that in later months I can assume you already know the notation. I plan to use it when discussing such topics as: ways of constructing Platonic solids, modeling with transformations, symmetry operations, etc. ■

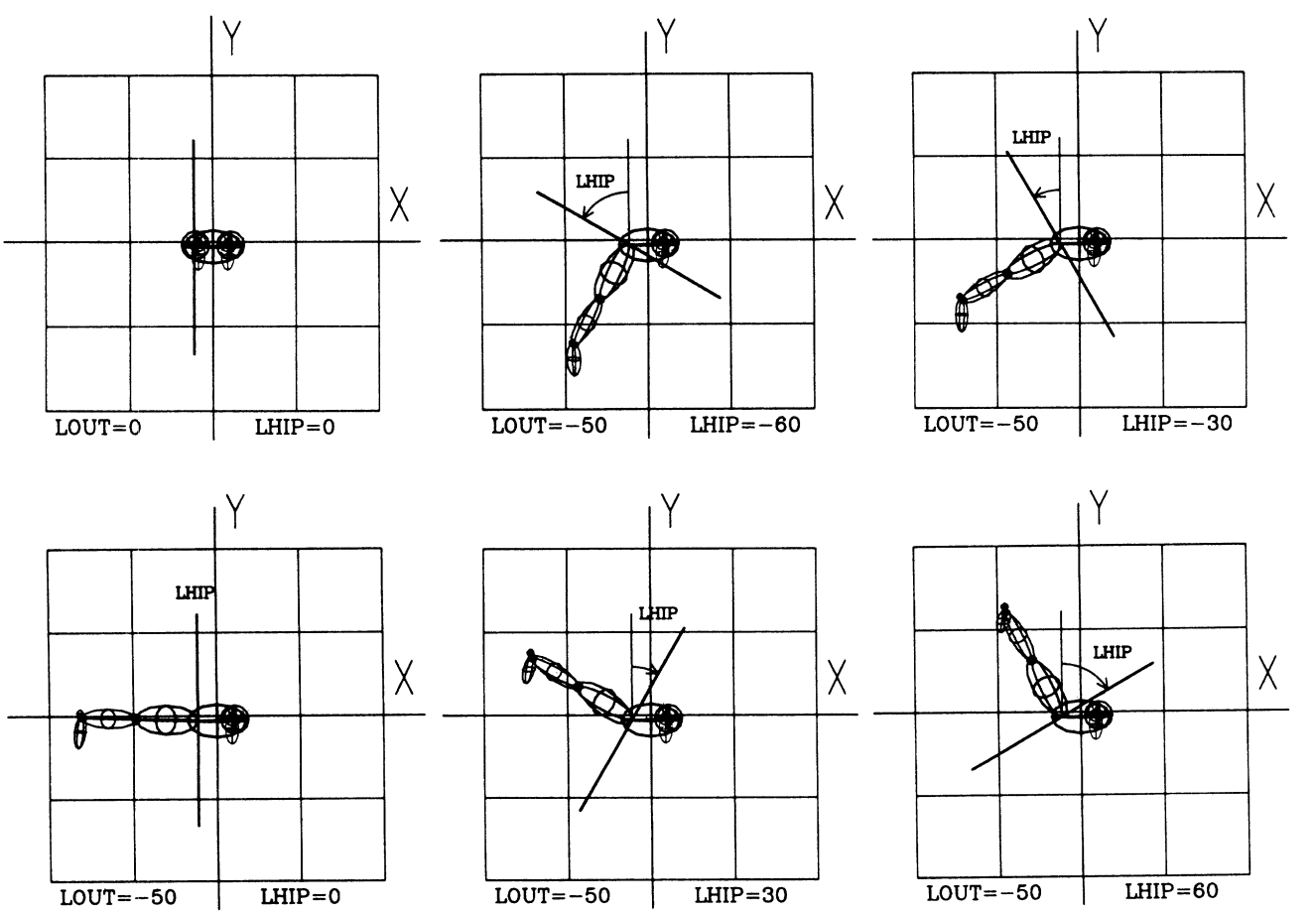


Figure 6. Top view of leg rotation.