Matrix Conventions Revisited

Technical Memo No. 64

Alvy Ray Smith

Computer Graphics Project
Computer Division
Lucasfilm Ltd

Revised 24 June 1983

This document was reentered by Alvy Ray Smith in Microsoft Word on May 5, 1999. Spelling and punctuation are generally preserved, but trivially minor spelling errors are corrected. The quaint form of the early language C is preserved. Otherwise additions or changes made to the original are noted inside square brackets or in footnotes.

Introduction

Our matrix and transformation conventions need to be restated and backed up with code to encourage their use. Since the modeling and rendering systems are currently being redesigned, it is a good time to ask that they be built according to convention. Some packages to ease this task will be presented here.

An example of what I would like to avoid in the future is the state of affairs currently existing between "med" and "reyes"[1]. The field of view for "med" is taken to be one-half the total view angle in the vertical direction. For "reyes", it is taken to be the total horizontal viewing angle. In "med" the aspect ratio is the ratio of the Picture System[2] width to height in the display (non-menu) portion of the screen. For "reyes", the aspect ratio is the aspect ratio of an Ikonas[3] framebuffer pixel[4] (which is not the aspect ratio of the corresponding video). It is not at all obvious to a user how to get a picture displayed by "med" in the Picture System to appear in the framebuffer with exactly the same relative shape and orientation when rendered there by "reyes".

What we want is a default path from model language through calligraphic display to raster display which is natural and requires no input from the user. And all along the path control would be according to convention.

Table of Contents

In addition to the text there are included in this memo several appendices:

---

[1] The modeling and rendering programs, respectively, in use at Lucasfilm at the time.

[2] A calligraphic line-drawing system by Evans & Sutherland Corporation.

[3] With a full-color raster scan display device.

[4] I would now say "pixel spacing", pixels themselves being point samples without height and width.

Appendix 1: The MxMatrix Package
Appendix 2: The CmxMatrix Package
Appendix 3: The aa_matrix Package
Appendix 4: The VxVector Package
Appendix 5: The VuSpec Package
Appendix 6: The CvuSpec Package
Appendix 7: The aa_view, aa_display, and aa_vutrix Packages
Appendix 8: The mx Matrix Desk Calculator

These may be considered to be manuals for the routines implementing the ideas here and those of Tech Memo 84. The modules are located in **/u0/gfx/lib/libmx.a**.

A Proposed Scenario

In Tech Memo 84, "The Viewing Transformation", the entire viewing transformation of a point **p** in model space is given by

pNPS,

where **N** is the *normalizing transformation* which maps a given viewing frustum in model space into the *canonical viewing frustum*, **P** is the *perspective transformation* which maps the canonical viewing frustum into *normalized device coordinates (NDC)*, and **S** is the *screen mapping* which maps NDC into a desired display space.

The canonical viewing frustum has its eyepoint at the origin and its far face in the $z = 1$ plane, and the far face has *xy* widths of 2 and is centered on the *z* axis. The **P** matrix transforms the canonical viewing frustum into NDC which is a rectangular prism having every constant *z* section equal in shape and size to the far face of the canonical viewing volume, for $0 \le z \le 1$. Thus *x* and *y* are on [-1, 1].

Clipping occurs on the points **pN** in the canonical viewing frustum (where clipping is relatively easy). Division by the homogeneous coordinate is performed on the points **pNP** in NDC. Hidden surface calculations are then applied to these points in NDC. Any further transformations of the image are applied at this juncture—for example, mapping the result onto a view screen seen obliquely (e.g., as a tactical display in a movie scene). Then, finally, the image is mapped to a desired display such as a Picture System, or Iris[5], or Ikonas, or Pixar[6] memory window. This screen mapping takes nonsquare pixels[7] into account, if necessary, and turns the *y* coordinate upside down, if necessary.

Modeling and animation perform operations on a model database. Let **q** be a point in such a database. Then $\mathbf{p} = \mathbf{qT}$ represents the end result of these operations, where **T** represents a sequence of transformations such as scales, translates, and rotates. The points **p** are those delivered to the viewing pipeline discussed above. In general, **qT** should be followed by a homogeneous divide. This is not true if only scales, translates, and rotates are permitted. In most systems this is

[5] By Silicon Graphics.
[6] A Pixar Image Computer.
[7] Ie, nonsquare pixel spacing.

the case, but I will propose below that global scale (which affects the [3, 3] element of the transformation matrix) and "full perspective" (affecting the [0, 3], [1, 3], or [2, 3] elements) be allowed. If the 4th column of the transformation matrix is affected, then the divide is necessary.

Animation also performs operations on the viewing parameters—i.e., the simulated camera. These are realized by the matrices **N** and **P. N** is conveniently decomposed into $\mathbf{N}_L$ and $\mathbf{N}_R$ when animating the camera; only $\mathbf{N}_L$ changes as the camera moves. Clipping is not performed by the animation process. For generality of application, the modeling/animation pipeline could be asked to return models in NDC as well as in model space.

Rendering is the process which takes the modified database and viewing parameters from the modeling/animation processes, clips the results, performs hidden surface calculations if necessary, and then, of course, renders the result into some display. So rendering needs the points $\mathbf{p} = \mathbf{qT}$ and the matrices **N** and **P** to perform clipping on the point **pN** and homogeneous divide on **pNP** for those points which survive clipping. Again, for generality of application, rendering could be asked to return results in NDC.

Variations on the Basic Scenario

Animating the camera is almost identical to animating any other rigid model. In fact, we could represent the viewing parameters by a special model, the *camera model*, and a transformation $\mathbf{T}_c$, the *camera transformation*. $\mathbf{T}_c$ would be restricted to scales, translates, rotates, and skews[8]. The camera model would be a default camera viewing pyramid with the eyepoint at the origin of model space being the apex and the view window being the base. The sizes of the default camera pyramid would be dictated by the default viewing parameters (see Tech Memo 84 and below). Only two parameters are missing from the camera model necessary for defining the view: hte near and far clipping plane locations. These two parameters plus the camera model and camera transformation are equivalent to the set of viewing parameters. This could be an alternative convention for the view spec. It has the feature that it makes the camera animation just like that of any other model. Depicting the view of an oracle camera would be straightforward also.

However, it has been pointed out that the camera is treated in special ways. For example, it is common to want to pan the camera through an angle. If only position information were stored for keyframe camera information, then interpolation through angles would not be straightforward. But this same argument could be used for arbitrary models. Just as we save transformation parameters for an arbitrary model in order to do correct interpolation (as opposed to saving just the matrix), we would do the same for the camera model.

[8] A future project: Extend the camera transformation (hence view spec) to arbitrary 4x4 transformations.

Nonstandard Transformations

It is well-known that with one 4x4 matrix all standard transformations can be effected. Thus a 3-D transformation is accomplished by multiplying a matrix of form

$$\mathbf{M} = \begin{bmatrix} s & S & S & p \\ S & s & S & p \\ S & S & s & p \\ t & t & t & g \end{bmatrix}$$

times each 4-D point in a model (three coordinates for position and a unity homogeneous coordinate). Here *s* stands for scale, *S* for skew (sometimes called shear), *t* for translate, *p* for perspective, and *g* for global scale. Most transformation systems realize rotation, scale, and translation with such a matrix. Rotation is realized by a combination of *s* and *S* entries, it being known that rotation about a principal axis can be accomplished with two skews and two scales.

It is not so common for systems to make use of the skew entries as such (i.e., independently of rotation), nor do they use the global scale *g* entry nor two of the three perspective *p* entries. This is a shame since they are freely given parameters in a 4x4 matrix system. Global scaling by a factor *a* is just as easily accomplished by setting the three *s* entries to *a* as by setting *g* to $\frac{1}{a}$, so it is not too surprising that this entry is ignored. But skew and perspective are quite useful parameters. The problem is probably due to the lack of intuitive controls for these parameters, or perhaps it is due to confusion of these parameters with "standard perspective" as typically implemented in the viewing transformation (see Tech Memo 84).

A command format for utilizing the skew *S* entries is exemplified by *skew x y .5*, read "skew *x*'s as *y* increases by .5*y*" or "skew towards *x*, as a function of *y*". This means that each point of the model is shifted right (meaning positive *x* here) depending on how large its *y* value is. In fact, $x = x+.5y$. It is implemented by simply setting $\mathbf{M}_{ij}$ to .5, where $\mathbf{M}_{ij}$ is the *S* entry for the $(i+1)^{th}$ row and $(j+1)^{th}$ column of 4x4 transformation matrix **M**. The mapping of *i* and *j* to principal axes is 0 for *x*, 1 for *y*, and 2 for *z*. *j* is assigned to the first argument of the skew command and *i* to the second; so $\mathbf{M}_{10} = .5$ in this example. Figure SKEW may be helpful in calibrating one's intuition on this command. It is the unit square with the origin at lower left. The skew command above causes the upper edge to shift right by one half its length.

A decision has to be made on this command as to what to do for the case *skew x x .5*. It is either disallowed, or defaults to scale *x* by .5 or by the logically consistent value 1.5. I prefer this last option.

Skew is handy, not for viewing, but for modeling. Simimlarly, perspective can be useful for modeling. It can be implemented as a command such as *pers x a* which is read "apply perspective along *x* axis with vanishing point at $x = a$". See

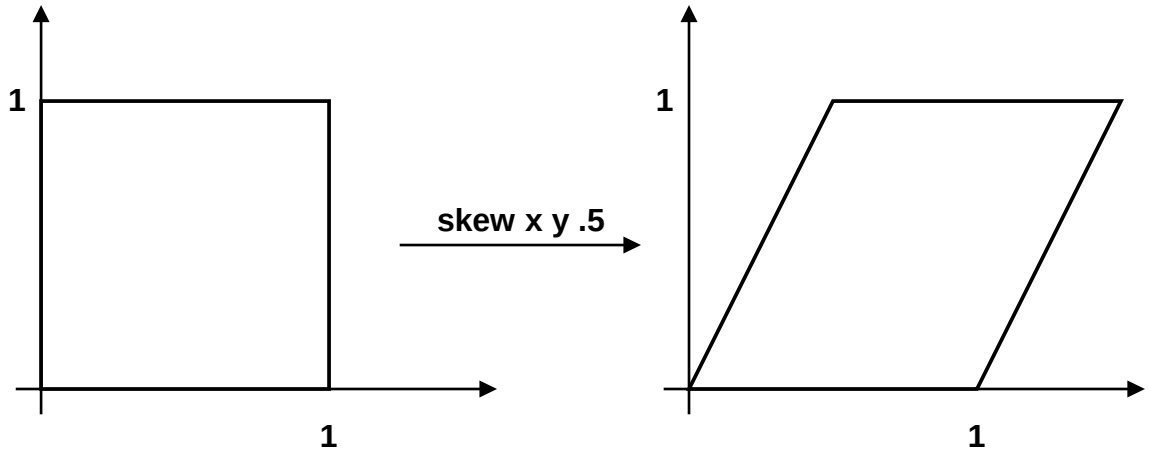

Figure SKEW

Figure PERSP for intuition building. The corresponding implementation is simply $\mathbf{M}_{i3} = \frac{1}{a}$, where i is mapped to the principal axes as above. In particular, $\mathbf{M}_{03} = \frac{1}{a}$ in our sample perspective command above. Clearly $a = 0$ is a problem, so this value should be disallowed. It is desirable to be able to set a to ∞ occasionally. The disallowed value of 0 can be used to force a 0 entry in the matrix for this case. As indicated in the figure, values of a which fall inside the model may cause problems. It is this strangeness which perhaps has kept this set of parameters unpopular. Nevertheless, they are powerful shapers and give a user a very good sense of perspective in general—i.e., including "standard perspective". For example, notice in the figure that the point $\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$ is unchanged by the perspective matrix.

A final transformation which is not always implemented is rotation about an

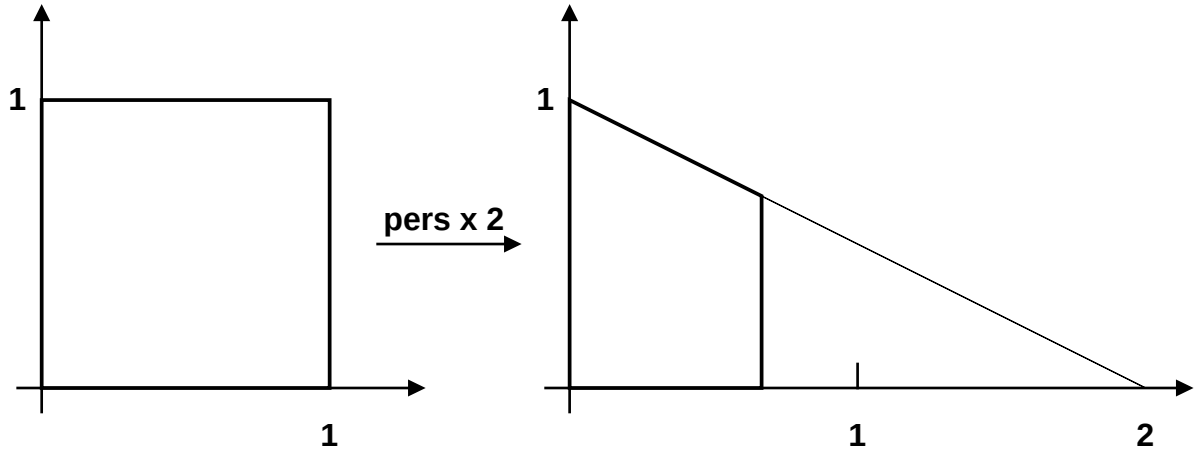

Figure PERSP

arbitrary axis through the origin. This is probably omitted in many implementations because it does not fall out trivially. It is not difficult, however. Let α, β, and γ represent the direction cosines for the desired axis with respect to the *x*, *y*, and *z* axes, respectively. So $[\alpha \quad \beta \quad \gamma]$ is the unit vector along the axis of rotation. Let θ be the desired angle of rotation about the new axis. Then the rotation matrix is ([ROGADM]):

$$\mathbf{R} = \begin{bmatrix} \alpha^2 + (1-\alpha^2)\cos\theta & \alpha\beta(1-\cos\theta) + \gamma\sin\theta & \alpha\gamma(1-\cos\theta) - \beta\sin\theta & 0 \\ \alpha\beta(1-\cos\theta) - \gamma\sin\theta & \beta^2 + (1-\beta^2)\cos\theta & \beta\gamma(1-\cos\theta) + \alpha\sin\theta & 0 \\ \alpha\gamma(1-\cos\theta) + \beta\sin\theta & \beta\gamma(1-\cos\theta) - \alpha\sin\theta & \gamma^2 + (1-\gamma\alpha^2)\cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Positive rotation is defined by the righthand rule along the axis of rotation.

Proposed 3-D Transformation Conventions

Here are a set of proposed conventions:

1. Computer Graphics Normal Form will be assumed: Vectors (points) are assumed to be row vectors. They are post-multiplied by matrices—i.e., the matrix is written to the right of the row vector it multiplies. Successive transformations premultiply the current transformation matrix. $\mathbf{M}[i][j]$ is the $(i+1)^{th}$ row, $(j+1)^{th}$ column of matrix **M**. This is how C defines matrices but is the reverse of the usual *xy* order for specifying 2-D arrays.
2. Model space is righthanded with *z* up, *x* right, and *y* in. It is sometimes convenient to think of model space as a coordinate system on a map spread on a table. *x* points east, *y* points north, and *z* points up from the table.
3. The viewing transformation is as defined in Tech Memo 84 with Normalized Device Coordinates (NDC) defined as lefthanded, *z* in, *x* right, and *y* up with *z* on [0., 1.] and *x* and *y* each on [-1., 1.].
4. The view spec is as defined in Tech Memo 84. Routines for reading and writing it will be provided. The default values are as given in Tech Memo 84. *The ascii version of this spec is the conventional form for communication between programs.*
5. Aspect ratio is defined as width to height. Field-of-view angle is defined as the full horizontal viewing angle. These parameters do not appear in the view spec but routines are provided which make all necessary conversions.
6. Modeling/animation programs will return models in model space, and the view as the conventional view spec (ascii form). They will return a view of a model in NDC if requested.
7. Rendering programs will read a model in model space with a view specified conventionally. They will return a view of a model in NDC if requested.
8. The view spec is in world space coordinates.
9. *All* angles are positive by the righthand rule whether the coordinate system is righthanded or lefthanded. The only exceptions are those special angles known as *azimuth*, *pitch*, and *roll*. Azimuth is positive about the *z* axis by the

lefthand rule, with 0 degrees being at the positive *y* axis. Pitch is positive above the *xy* plane (in the positive *z* direction), negative below, and 0 degrees at the plane. Roll is positive by the lefthand rule along its axis of rotation, the ViewNormal. It is 0 degrees when a radius vector from the axis of rotation points right, parallel the *xy* plane, as viewed looking in the direction of the ViewNormal.

10. The display spec is as in Tech Memo 84. Routines for reading and writing it will be provided. The default values are as given in Tech Memo 84. *The ascii version of this spec is the conventional form of inter-program communication*.
11. Since the full screen aspect ratio of a display device may be changed on purpose (as for *Star Trek II*) or by ordinary drift, 3-D programs should take it as an argument.
12. Far planes at infinity should be allowed, but it must be understood that they are a special case requiring extraordinary treatment. For example, the notion of canonical viewing frustum must be extended to infinity as explained in Tech Memo 84.

The matrix packages below are meant to help with these conventions.

Matrix Packages

I have noticed that there are several matrix packages around:

1. My own in /u0/alvy/lib/libfbx.a which is used in "fbx" and "gene". This package is closely related to an "aarg" front end. It has the standard stuff for 4x4 transformation matrix formation and not so usual routines for matrix printing and setting, unusual perspective manipulations, global scaling, and skew. It does not have matrix-by-matrix operations or matrix-by-scalar operations but does have matrix inversion. The package works only for 4x4 matrices.
2. The **matrix(3)** package written by Tom Duff and used with variations by Bill Reeves and Tom Duff. This has the usual transformation stuff plus windowing and viewport. It has matrix-by-matrix ops and matrix-by-scalar ops. It does not allow one to get at all the entries of the matrix. Skew and full perspective are not implemented. It has dangerous names such as "scale", "rotate", and "ident". It is general for NxN matrices but compiled for 4x4s.
3. **Cpac(3)** uses its own version of a matrix package because it is doing the weird stuff required by the Picture System. It uses dangerous names such as "Scale" and "Rotate".
4. Loren doesn't use any matrices and uses his own viewport routines for "reyes".

A Matrix and Viewing Transformation Package

A matrix package is presented here which incorporates the best features of the packages above, hopefully replacing all of them (and simply being used in those cases where matrices are currently not used). See Appendices 1-7. It is

meant to make implementation of the transformation conventions above a straightforward matter. It features the following:

1. All package routines have distinctive and mnemonic prefixes. All matrix routines to be used globally are prefixed with "Mx" (for "*M*atri*x*" or for "*M*atrix *x*formation")—e.g., MxScale(), MxRotate()—or with "Cmx" (for "*C*urrent *m*atri*x*" or "*C*urrnet *m*atrix *x*formation"). See Appendices 1 and 2. All viewing routines are prefixed with "Vu" or with "Cvu" (for "*C*urrent *v*iew"). See Appendices 5 and 6.
2. All angles are expressed in degrees. Routines are provided for converting to and from radians.
3. The primary axes are referred to by one of '[xyzXYZ]'. Upper case and lower case designations are interchangeable.
4. The following traditional transformations are available: identity, scale, translate, rotate about the major axes.
5. The following less common transformations are also available: skew, full perspective (not to be confused with the "real world" perspective transformation defined in Tech Memo 84), global scale, and rotate about arbitrary axes through the origin.
6. The notion of current transformation matrix is supported and a set of routines provided for its maintenance: push, pop, concatenate (from left and right), print it, set it, etc. Similarly, the notion of current view is supported by a set routines for its maintenance: push, pop, print it, set it, etc.
7. The matrix package is independent of the current transformation matrix if desired. For example, there should be two routines for rotation: one premultiplies the current transformation matrix with the matrix describing the given rotation command, the other simply returns the matrix describing the rotation. Similarly, the view package is independent of the current view if desired.
8. The matrix package supports all common matrix-by-matrix operations and matrix inversion and matrix transposition. It also supports vector-by-matrix, scalar-by-matrix, vector-by-vector, and scalar-by-vector operations.
9. There should be an "aarg" interface to the packages. See Appendices 3 and 7.
10. A vector package is provided with routines prefixed by "Vx". It is intimate with the matrix package. All common vector-vector operations are provided. Dot product and length are provided.
11. The screen map of Tech Memo 84 is fully supported by the view package.
12. Tech Memo 84 is fully supported in the sense that all matrices described in the memo are available with routines in the view package.
13. Infinite far planes are supported but the user must be careful (as explained in Tech Memo 84).

Reference

[ROGADM]

David F. Rogers and J. Alan Adams, **Mathematical Elements for Computer Graphics**, McGraw-Hill Book Company, San Francisco, 1976.

Appendix 1: The MxMatrix Package

Following is a brief description of the routines in this package. This set of routines has no notion of a Cmx (current transform matrix) [see Appendix 2]. I borrow extensively from code by Tom Duff and earlier versions of my own matrix package. In particular, I preserve Tom's notion of leaving the code general for MX_N × MX_N matrices but compiling it for MX_N ≡ 4. Those routines, such as rotate, which are inherently 3-D routines and hence not independent of MX_N are compiled into a separate module. The full rotate code is based on the [ROGADM] derivation.

In no case will parameter name cause trouble. For example, **MxMultiply(a, b, b)** doesn't cause any part of *b* to be clobbered prematurely.

The source resides in files *MxMatrix.h*, *MxMatrix.c*, *MxMatrix4.c*.

```
#define MX_N 4
#include <MxMatrix.h>
typedef double matrix[MX_N][MX_N];
typedef double vector[MX_N];
matrix Cmx;
```

MxCopy(a, b)
matrix a, b;

Copy *a* to *b*.

MxTranspose(a, r)
matrix a, r;

Transpose *a* into result *r*.

MxIdentity(a)
matrix a;

Store the identity matrix in *a*.

MxScalar Multiply(s, a, r)
matrix a, r;
double s;

Multiply *a* by scalar *s*; put result in *r*.

MxVectorMultiply(p, a, q)
matrix a;
vector p, q;

Multiply vector *p* by matrix *a*; put result in *q*. No perspective division is done.

MxNegate(a, r)
matrix a, r;

Negate *a*; put result in *r*.

MxAdd(a, b, r)
matrix a, b, r;
Add *a* and *b*; put result in *r*.

MxSubtract(a, b, r)
matrix a, b, r;
Subtract *b* from *a*; put result in *r*.

MxMultiply(a, b, r)
matrix a, b, r;
Multiply *a* by *b*; put result in *r*.

MxDivide(a, b, r)
matrix a, b, r;
Multiply *a* by the inverse of *b*; put result in *r*.

MxReverseDivide(a, b, r)
matrix a, b, r;
Multiply the inverse of *a* by *b*; put result in *r*.

MxStackInit()
Initialize the matrix stack.

char* MxPush(m)
matrix m;
Push *m* onto the matrix stack. Normal return is NULL. There is an error message returned in case of stack overflow. The length of the stack MX_STKLEN is defined in **MxMatrix.h**.

char* MxPop(m)
matrix m;
Pop the matrix stack into *m*. Normal return is NULL. There is an error message returned in case of stack underflow.

double MxTransform(a, b, m)
vector a, b;
matrix m;
Transform *a* by *m* (including perspective divide); put result in *b*. The return value is the homogeneous coordinate before dividing. If the transformed point is at infinity, no division is done.

double MxInvert(a, r)
matrix a, r;
Invert matrix *a*; put result in *r*. The return value is the determinant of *a*. If the determinant is zero, *m* is singular, and *r* contains garbage. The method used is Gaussian elimination with partial pivoting. Note that this method tends to be ill-conditioned for large MX_N.

double MxDeterminant(a)
matrix a;

Return the determinant of matrix *a*.

MxPrint(m)
matrix m;
Print matrix *m*.

MxGet(m)
matrix m;
Get MX_N × MX_N parameters from a tty to define matrix *m*, row-first order.

The following routines are dependent on MX_N ≡ 3 or MX_N ≡ 4:

MxScale(sx, sy, sz, m)
double sx, sy, sz;
matrix m;
Place a scaling transformation in *m*.

MxGlobalScale(s, m)
double s;
matrix m;
Place a global scaling transformation in *m*. The inverse of *s* is inserted in the *m*[3][3] element, except *s* ≡ 0. causes a 0. to be inserted in the *m*[3][3] element. It is probably good practice to use **MxScale()**, with equal arguments, instead.

MxTranslate(dx, dy, dz, m)
double dx, dy, dz;
matrix m;
Place a translation in *m*.

double MxRadians(angle)
double angle;
Converts *angle* in degrees to radians.

double MxDegrees(angle)
double angle;
Converts *angle* in radians to degrees.

MxRotate(angle, axis, m)
double angle;
char axis;
matrix m;
Place a rotation by *angle* degrees about the given axis in *m*. The axis is denoted by one of '[XYZxyz]'. Positive rotation follows the righthand rule.

MxFullRotate(angle, axisvector, m)
double angle;
vector axisvector;
matrix m;

Place a rotation by *angle* degrees about the given axis in *m*. The *axisvector* is the direction of an axis which passes through the origin. Positive rotation follows the righthand rule about the given *axisvector*.

MxSkew(coordinate, axis, factor, m)
char coordinate, axis;
double factor;
matrix m;

Skew *coordinate* as a linear function of *axis*—e.g., skew *y* as a linear function of *x*. The function computed is *coordinate* += *axis***factor*. If *coordinate* is the same as *axis*, then it is simply scaled by (1 + *factor*). Place resulting transformation in *m*.

MxFullPerspective(a, b, c, m)
double a, b, c;
matrix m;

For arbitrary mucking about with the "perspective" elements of the standard 4x4 matrix transform. The three arguments are assumed to be vanishing points on the corresponding *x*, *y*, *z* axes. The resulting transformation is placed in *m*. If *a*, *b*, or *c* is 0., the corresponding vanishing point is set to infinity.

Appendix 2: The CmxMatrix Package

This is the specialization of the *MxMatrix* package for the notion of a "current matrix" or "Cmx". These routines are not simply calls to the *MxMatrix* package but have been optimized for speed in the case of scale, translate, rotate, skew, global scale, full rotate, full perspective, copy, push, and pop. MX_N ≡ 4 is assumed. The source is *CmxMatrix.c*.

CmxCopyTo(m)
matrix m;

Copy *Cmx* to *m*.

CmxCopyFrom(m)
matrix m;

Copy *m* into *Cmx*.

CmxTranspose()

Transpose *Cmx*.

CmxIdentity()

Store identity matrix in *Cmx*.

CmxScalarMultiply(s)
double s;

Multiply *Cmx* by scalar *s*.

CmxVectorMultiply(p, q)
vector p, q;
Multiply point (vector) *p* by *Cmx* and place result in *q*. No perspective division is done.

CmxNegate()
Negate *Cmx*.

CmxAdd(a)
matrix a;
Add *a* to *Cmx*.

CmxSubtract(a)
matrix a;
Subtract *a* from *Cmx*.

CmxConcat(m)
matrix m;
Concatenate (multiply) *m* onto *Cmx* (from the left).

CmxRightConcat(m)
matrix m;
Concatenate (multiply) *m* onto *Cmx* from the right.

CmxDivide(a)
matrix a;
Multiply *Cmx* by inverse of *a*—i.e., *Cmx* = *Cmx**inverse(*a*).

CmxReverseDivide(a)
matrix a;
Multiply *Cmx* by inverse of *a* from reverse side—i.e., *Cmx* = inverse(*a*)* *Cmx*.

CmxStackInit()
Initialize the matrix stack. This is identical to **MxStackInit()**.

char* CmxPush()
Push *Cmx* on the matrix stack. An error message is returned for stack overflow, else NULL is returned.

char* CmxPop()
Pop the matrix stack into *Cmx*. An error message is returned for stack underflow, else NULL is returned.

double CmxTransform(a, b)
vector a, b;
Transform *a* by *Cmx* (including perspective divide); put result in *b*. The return value is the homogeneous coordinate before dividing. If the transformed point is at infinity, no division is done.

double CmxInvert()
Invert *Cmx*. The return value is the determinant of *Cmx*. If the determinant is zero, *Cmx* is singular and is left unchanged.

double CmxDeterminant()
Return the determinant of *Cmx*.

CmxPrint()
Print *Cmx*.

CmxGet()
Get MX_N × MX_N parameters from a tty to define *Cmx*, row-first order.

CmxScale(sx, sy, sz)
double sx, sy, sz;
Concatenate a scaling transformation onto *Cmx*.

CmxGlobalScale(s)
double s;
Concatenate a global scaling transformation onto *Cmx*. $s \equiv 0$. may place points at infinity so should be avoided. **CmxScale()** should normally be used instead of this routine.

CmxTranslate(dx, dy, dz)
double dx, dy, dz;
Concatenate a translation onto *Cmx*.

CmxRotate(angle, axis)
double angle;
char axis;
Concatenate a rotation by *angle* degrees about the given *axis* onto *Cmx*. The *axis* is denoted by one of '[XYZxyz]'. Positive rotation follows the righthand rule.

CmxFullRotate(angle, axisvector)
double angle;
vector axisvector;
Concatenate a rotation by *angle* degrees about the given axis onto *Cmx*. The *axisvector* is the direction of an axis which passes through the origin. Positive rotation follows the righthand rule about the given *axisvector*.

CmxSkew(coordinate, axis, factor)
char coordinate, axis;
double factor;
Skew *coordinate* as a linear function of *axis*—e.g., skew *y* as a linear function of *x*. The function computed is *coordinate* += *axis***factor*. If *coordinate* is the same as *axis*, then it is simply scaled by (1 + *factor*). [The skew is concatenated onto *Cmx*.]

CmxFullPerspective(a, b, c)
double a, b, c;

For arbitrary mucking about with the "perspective" elements of the standard 4x4 matrix transform. The three arguments are assumed to be vanishing points on the corresponding *x*, *y*, *z* axes. The resulting matrix is concatenated with *Cmx*. If *a*, *b*, or *c* is 0., the corresponding vanishing point is set to infinity.

Appendix 3: The aa_matrix Package

Following is the help message for the *aa_matrix* package. It is intended to be a front-end to those parts of the *CmxMatrix* package typically used for computer graphics manipulations. The source is files *matrixaarg.h*, *aa_matrix.c*.

[-ident]	Set current matrix to identity
[-move %F [%F [%F]]]	Translate x y z [0. by default]
[-trans %F [%F [%F]]]	Translate x y z [0. by default]
[-rot %F %c]	Rotate angle (degrees) about axis ([xyzXYZ])
[-rotate %F %F %F %F]	Rotate angle about axis given by vector (x, y, z)
[-radians [%c]]	Radians switch (arg=[TFtf]) [default F]
[-scale %F [%F [%F]]]	Scale x y z [1. by default]
[-gscale %F]	Global scale
[-skew %c %c %f]	Skew coordinate in direction of axis
[-pers [%F [%F [%F]]]]	Set perspective vanishing points in x, y, z
[-pop]	Pop matrix stack into current matrix
[-push]	Push current matrix onto matrix stack
[-stkinit]	Initialize matrix stack
[-transpose]	Transpose current matrix
[-invert]	Invert current matrix
[-mset %d<0,3> %d<0,3> %F]	Set current matrix element
[-mget]	Set current matrix from keyboard
[-mprint]	Print current matrix
[-mverbose [%c]]	Verbose matrix switch (arg=[TFtf]) [default F]

Appendix 4: The VxVector Package

Following are the vector routines in the *VxVector* package. The source files are *VxVector.c*, *VxVector_4.c*. The latter is similar to the *MxMatrix* package in being general for arbitrary MX_N at compile time, compiled for MX_N ≡ 4 here. The former assumes 4-element vectors also, but the fourth element is assumed to be a homogeneous coordinate and treated accordingly. The routines described here are principally those of *VxVector.c*. For nearly every routine there is a corresponding one in *VxVector_4.c* which does not treat the fourth element specially. These routines have the same name with *_4* appended. They are not described except for *VxCross_4*() which is substantially different from *VxCross*().

For the module *VxVector.o*, the homogeneous coordinate is simply copied into the result vector in case of single operand functions, such as *VxNegate*(). For double operand functions, such as *VxAdd*(), the result homogeneous coordinate depends on those of the two operands. In general, if they are equal then the result vector homogeneous coordinate is simply a copy of one of the homogeneous coordinates of the operands. If unequal, the two operands are silently divided through by their respective homogeneous coordinates (see *VxHomoDivide*()) before the operation is performed, and the result vector homogeneous coordinate set to 1.

```
#define MX_N 4
#include <MxMatrix.h>
typedef double vector[MX_N];
typedef double matrix[MX_N][MX_N];
matrix Cmx;
```

VxCopy(a, b)
vector a, b;
Copies *a* to *b*.

double* VxScalarMultiply(s, a, r)
vector a, r;
double s;
Multiply *a* by scalar *s*; put result in *r* and return its address[9].

double* VxNegate(a, r)
vector a, r;
Negate *a*; put result in *r* and return its address.

double* VxAdd(a, b, r)
vector a, b, r;
Add *a* and *b*; put result in *r* and return its address.

double* VxSubtract(a, b, r)
vector a, b, r;
Subtract *b* from *a*; put result in *r* and return its address.

double VxDot(a, b)
vector a, b;
Return the dot product of vectors *a* and *b*. That is, multiply row vector *a* times column vector *b* to get a scalar. If the homogeneous coordinates are unequal to 1., a division of the vectors by their respective homogeneous coordinates is performed before the dot product is computed.

double VxLength(a)
vector a;

[9] Seems quaint now. Cleaner to define it **vector VxScalarMultiply(s, a, r)**. Similarly for the following routines. Couldn't do so at the time, as I recall.

Return the length of vector *a*. The vector is divided through by the homogeneous coordinate *a*[3], if it is unequal to 1, before the length is computed.

double* VxNormalize(a, r)
vector a, r;
Normalize vector *a*; put resulting unit vector in *r* and return its address.

double* VxFloor(a, r)
vector a, r;
Take the floor of each element of vector *a*; put result in *r* and return its address.

VxMultiply(a, b, m)
vector a, b;
matrix m;
Multiply column vector *a* times row vector *b* to get matrix *m*.

double* VxLerp(s, a, b, r)
double s;
vector a, b, r;
Perform the linear interpolation from *a* to *b* by amount *s* and place result in *r*; return a pointer to *r*. So $r = a + s*(b - a)$.

double* VxHomoDivide(a, r)
vector a, r;
Do homogeneous divide of vector *a* by *a*[3] and store result in vector *r* with homogeneous coordinate $r[3] \equiv 1$. Return a pointer to *r*.

VxPrint(v)
vector v;
Print vector *v*.

VxGet(v)
vector v;
Get MX_N parameters from a tty to define vector *v*.

double* VxCross(a, b, r)
vector a, b, r;
Returns the cross product of *a* and *b* in *r* and returns pointer to *r*.

VxZero(a)
vector a;
Returns TRUE if each of the first three components of *a* is 0. (within MX_EPSILON of 0.), else it returns FALSE. The *a*[3] element is ignored.

The following routine is the three-vector generalization of the two-vector cross product. An example of its use is the generation of the vector of the plane which passes through three given points [BLIN77]. It is in the module *VxVector_4.o*.

double* VxCross_4(a, b, c, r)

vector a, b, c, r;

Returns the generalized cross product of *a*, *b*, and *c* in *r*. (Cf. **VxCross()** above.)

Reference

[BLIN77]

James F. Blinn, *A Homogeneous Formulation for Lines in 3 Space*, **Computer Graphics**, Vol 11, No 2, Jul 1977, 237-241 (SIGGRAPH '77 Proceedings).

Appendix 5: The VuSpec Package

This package supports the view spec outlined in the text and Tech Memo 84. The view spec is stored in a struct LFL_View defined in *VuSpec.h*. It similarly supports the display spec and struct LFL_Display. The source resides in files *VuSpec.h*, *VuSpec.c*, *VuDisplay.c*, *VuMatrix.c*, *VuClip.c*.

```
#include <MxMatrix.h>
#include <VuSpec.h>
#include <stdio.h>
typedef double MxWorldType;
```

VuViewInit(vup)
struct LFL_View* vup;

Initialize given view struct to default values defined in *struct LFL_View VuReference* which is a perspective view.

VuSetViewPoint(vup, v)
struct LFL_View* vup;
vector v;

Set the viewpoint of the given view structure to the given point in world space.

VuSetViewNormal(vup, v)
struct LFL_View* vup;
vector v;

Set the view normal of the given view struct to the given vector in world space.

VuSetViewUp(vup, v)
struct LFL_View* vup;
vector v;

Set the view up of the given view struct to the given vector in world space. The component of this vector perpendicular to the view normal vector defines the up direction. This routine does not check for the view up vector collinear with the view normal.

VuSetViewDistance(vup, d)
struct LFL_View* vup;
MxWorldType d;

Set the distance of the given view struct to the given distance, which is the distance along the view normal from the view point at which the view plane lies. The distance is in world space.

VuSetNearFar(vup, near, far)
struct LFL_View* vup;
MxWorldType near, far;

Set the near and far clipping plane distances of the given view struct. The planes are located at the given distances from the view point along the view normal. If *far* is 0., then the far clipping plane is at infinity. The distances are in world space.

VuSetViewWindow(vup, centeru, centerv, halfsizeu, halfsizev)
struct LFL_View* vup;
MxWorldType centeru, centerv, halfsizeu, halfsizev;

Set the window of the given view struct by specifying the location of its center relative the point where the view normal intersects the view plane. The coordinates are in world space. The horizontal and vertical halfsizes complete the definition. They are also expressed in world space. Coordinate *v* is parallel the up vector (not view up) and coordinate *u* is perpendicular to it, parallel the right vector which points to the right when looking out the view normal with the up vector pointing up.

VuSetProjectionType(vup, type)
struct LFL_View* vup;

Set the projection type of the given view struct to either VU_PERSPECTIVE or VU_ORTHOGRAPHIC (defined in **VuSpec.h**).

VuSetAspectRatio(vup, aspectratio)
struct LFL_View* vup;
double aspectratio;

Resets WindowHalfsize.v in the given view struct so that the width to height aspect ratio is as given. WindowHalfsize.u is unchanged, so the field-of-view angle is unchanged (cf. **VuSetFieldOfView()** below) when meaningful. Special aspect ratios VU_SILENT, VU_VIDEO, VU_ACADEMY, VU_PANAVISION, VU_CINEMASCOPE, VU_TODDAO, VU_SUPERPANAVISION, VU_ULTRAPANAVISION, VU_IMAX, and VU_OMNIMAX are defined in **VuSpec.h**.

VuSetFieldOfView(vup, angle)
struct LFL_View* vup;
double angle;

Resets WindowHalfsize.u in the given view struct so that the full horizontal field-of-view angle is *angle* degrees. WindowHalfsize.v is also reset to maintain aspect ratio. ViewDistance is unchanged. N.B. Field of view for off-axis

view windows is invalid. (Cf. **VuSetAspectRatio()** above.) The routine returns a 0 normally but a –1 in case WindowHalfsize.u is 0.

VuGetViewPoint(vup, v)
struct LFL_View* vup;
vector v;

Returns the view point from the given view struct. *v*[3] is set to 1.

VuGetViewNormal(vup, v)
struct LFL_View* vup;
vector v;

Returns the view normal from the given view struct. *v*[3] is set to 1.

VuGetViewUp(vup, v)
struct LFL_View* vup;
vector v;

Returns the view up from the given view struct. *v*[3] is set to 1. This routine does not check for view up vector collinear with the view normal.

VuGetUnitNormal(vup, n)
struct LFL_View* vup;
vector n;

Returns the unit vector in the viewing direction of the given view struct. *n*[3] is set to 1.

VuGetUnitUp(vup, v)
struct LFL_View* vup;
vector v;

Returns the unit vector in the up direction of the given view struct. This is the normalized component of the view up vector orthogonal to the view normal and in the plane defined by the up and normal vectors. *v*[3] is set to 1. This routine checks for view up collinear with view normal. If collinear, *v* is set arbitrarily to be orthogonal to view normal, an error message is printed (if **VuPrintError** is TRUE), and –1 is returned instead of the normal 0. The view struct is not changed.

VuGetUnitRight(vup, u)
struct LFL_View* vup;
vector u;

Returns the unit vector towards the right for the given view struct. This is the unit vector orthogonal to the unit up and unit normal vectors defined above, by the lefthand rule. *u*[3] is set to 1. This routine checks for view up collinear with view normal. If collinear, an arbitrary view up orthogonal to view normal is used to compute *u*, an error message is printed (if **VuPrintError** is TRUE), and –1 is returned instead of the normal 0. The view struct is not changed.

MxWorldType VuGetViewDistance(vup)

struct LFL_View* vup;
Returns the view distance of the given view struct.

VuGetNearFar(vup, nearp, farp)
struct LFL_View* vup;
MxWorldType *nearp, *farp;
Returns the near distance and far distance of the given view struct.

double VuGetAspectRatio(vup)
struct LFL_View* vup;
Returns the aspect ratio of the viewing window defined in the given view struct. It is defined as *WindowHalfsize.u / WindowHalfsize.v*. An error return of –1. occurs for a WindowHalfsize.v of 0.

double VuGetFieldOfView(vup)
struct LFL_View* vup;
Returns the full horizontal field-of-view angle (in degrees) associated with the given view struct. It is computed as *2 * atan(WindowHalfsize.u / ViewDistance)*. This angle is valid only for centered view windows.

VuGetViewWindow(vup, centerup, centervp, halfsizeup, halfsizevp)
struct LFL_View* vup;
MxWorldType *centerup, *centervp, *halfsizeup, *halfsizevp;
Returns the view window description from the given view struct.

VuGetProjectionType(vup)
struct LFL_View* vup;
Returns the projection type of the given view struct (VU_PERSPECTIVE or VU_ORTHOGRAPHIC, defined in **VuSpec.h**).

VuViewPrint(vup)
struct LFL_View* vup;
Print an ascii version of the given view struct on *stdout*.

VuFileViewPrint(file, vup)
FILE* file;
struct LFL_View* vup;
Print an ascii version of the given view struct to the given file. This is the conventional way of passing viewing definitions between programs (cf. **VuFileViewScan()** below).

VuMiscPrint(vup)
struct LFL_View* vup;
Print an ascii version of miscellaneous derived parameters from the given view struct on *stdout*. These parameters are field-of-view angle, view window aspect ratio, and the azimuth, pitch, and roll of the given view (cf. **VuGetAspectRatio()**, **VuGetFieldOfView()**, and **VuGetViewAngles()**).

VuFileMiscPrint(file, vup)

FILE* file;
struct LFL_View* vup;

Print an ascii version of miscellaneous derived parameters from the given view struct to the given file. See **VuMiscPrint()** above.

VuViewScan(vup)
struct LFL_View* vup;

Scan an ascii version of a viewing definition into the given view struct from *stdin*.

VuFileViewScan(file, vup)
FILE* file;
struct LFL_View* vup;

Scan an ascii version of a viewing definition into the given view struct from the given file. This is the conventional way of passing viewing definitions between programs (cf. **VuFileViewPrint()** above).

VuStackInit(vup)
struct LFL_View* vup;

Initialize the view stack.

char* VuPush(vup)
struct LFL_View* vup;

Push the given view struct onto the view stack. Normal return is NULL. There is an error message returned in case of stack overflow. The length of the stack VU_STKLEN is defined in **VuSpec.h**.

char* VuPop(vup)
struct LFL_View* vup;

Pop the view stack into the given view struct. Normal return is NULL. There is an error message returned in case of stack underflow.

VuPerspNormalization(vup, m)
struct LFL_View* vup;
matrix m;

The perspective normalization matrix (**N** in Tech Memo 84) is derived from the given view struct and returned in matrix *m*. An error return of –1 occurs for collinear ViewUp and ViewNormal (see **VuGetUVN()** below for full explanation and the actions taken) else 0 is returned. The special case of far plane at infinity is handled as explained in Tech Memo 84. A ViewDistance of zero is an error in this case and causes a –1 return.

VuOrthoNormalization(vup, m)
struct LFL_View* vup;
matrix m;

The orthographic normalization matrix ($\mathbf{N}_o$ in Tech Memo 84) is derived from the given view struct and returned in matrix *m*. An error return of –1 occurs for collinear ViewUp and ViewNormal (see **VuGetUVN()** below for

full explanation and the actions taken) else 0 is returned. FarDisance equal NearDistance causes an error return also. The special case of far plane at infinity is handled as explained in Tech Memo 84. A ViewDistance of zero is an error in this case and causes a –1 return.

VuPerspTransformation(vup, m)
struct LFL_View* vup;
matrix m;

The perspective transformation matrix (**P** in Tech Memo 84) is derived from the given view struct and returned in matrix *m*. The special case of far plane at infinity is handled as explained in Tech Memo 84. A –1 is returned in case ViewDistance is 0 and an error message is printed (suppressed if **VuPrintError** is set to FALSE); normal return is 0. FarDisance equal NearDistance causes an error return also.

VuPerspProjection(vup, m)
struct LFL_View* vup;
matrix m;

The perspective projection matrix (**Q** in Tech Memo 84) is derived from the given view struct and returned in matrix *m*. The special case of far plane at infinity is handled as explained in Tech Memo 84. A –1 is returned in case ViewDistance is 0 and an error message is printed (suppressed if **VuPrintError** is set to FALSE); normal return is 0.

VuCollinear(vup, up)
struct LFL_View* vup;
vector up;

Checks the given view struct to see if the view up vector is collinear with the view normal vector. It returns TRUE in case of collinearity and FALSE otherwise. In either case, *up* contains a vector orthogonal to the view normal upon return. In case of collinearity, this vector is arbitrarily chosen. Otherwise, it is just the view up vector. The view struct is not changed.

VuGetUVN(vup, u, v, n)
struct LFL_View* vup;
vector u, v, n;

Computes the three orthogonal unit vectors defining the view space described in the given view struct. (The first three elements of each vector is a unit vector; the fourth element is arbitrarily set to 1. for all three vectors.) *n* is the unit normal obtained from ViewNormal. *v* is the unit up vector obtained from ViewUp as defined in Tech Memo 84. The ViewUp vector is not necessarily perpendicular to *n* so *v* is that component which is. Then *u*, the unit right vector, is obtained by cross product to be orthogonal to *v* and *n* and forms a lefthanded coordinate system with them. If the ViewUp vector happens to be collinear with the ViewNormal vector, an error message is printed, *v* is made arbitrarily orthogonal to *n* (it may be meaningless however),

and a –1 is returned. Normal return is a 0. The view struct is not changed. The error message may be suppressed by setting **VuPrintError** to FALSE.

Following are several convenient interfaces to the view spec. They follow closely routines by similar names written by Bill Reeves in an earlier package.

VuSetViewFromAngles(vup, azimuth, pitch, roll)
struct LFL_View* vup;
double azimuth, pitch, roll;

ViewNormal and ViewUp of the given view are determined from angles *azimuth*, *pitch*, and *roll* as defined in **VuView()** below.

VuGetAnglesFromView(vup, azimuthp, pitchp, rollp)
struct LFL_View* vup;
double *azimuthp, *pitchp, *rollp;

The inverse of **VuSetViewFromAngles()** above. Angles returned are not necessarily those used to generate the view. For example, 480 degrees would be returned as 120, and a roll of 200 degrees would be returned as –160. A collinearity check is made identical in action to **VuGetUnitRight()** above. The view struct is not changed.

VuView(vup, distance, azimuth, pitch, roll)
struct LFL_View* vup;
double distance, azimuth, pitch, roll;

Defines the location of the ViewPoint by giving its *distance* from the world space origin along a radius vector at angle *azimuth* degrees in the *xy* plane, measured positive by the lefthand rule on the *z* axis, and at angle *pitch* degrees from the *xy* plane, measured positive above the *xy* plane (positive *z* direction) and negative below. Azimuth is measured relative the positive *y* axis. The vector from the origin to this point defines the ViewNormal. ViewUp is derived from the angle *roll* measured about the ViewNormal using the lefthand rule. Roll equal 0 degrees implies the rightward view vector is parallel the *xy* plane with the *z* component of ViewUp positive. ViewUp is always set perpendicular to ViewNormal by this routine which may be used with **VuWindow()** or **VuPerspective()** to completely fill a view struct. Alternatively, **VuFullView()**, **VuLookAt()**, or **VuCamera()** may be used in its place.

VuFullView(vup, V, azimuth, pitch, roll)
struct LFL_View* vup;
vector V;
double azimuth, pitch, roll;

The ViewPoint is placed at the location specified by vector *V* in world space. The ViewNormal is located as described above with the angles *azimuth* and *pitch*, but they are measured relative a coordinate system centered at the

ViewPoint which is just a simple translation of the world space system. ViewUp is determined as above from the *roll* angle.

VuLookAt(vup, V, p, roll)
struct LFL_View* vup;
vector V, p;
double roll;

Locates the ViewPoint at $[Vx \quad Vy \quad Vz \quad 1]$. The ViewNormal direction is that defined by a vector from ViewPoint to $[px \quad py \quad pz \quad 1]$. *Roll* is interpreted as above. In particular, ViewUp is always set perpendicular to ViewNormal.

This routine does not supply sufficient data for computing a unique view when $p - V$ is parallel the z axis. In this case, ViewUp can be anything so this routine sets it to $[0 \quad -1 \quad 0]$ if the z component of ViewNormal is positive or to $[0 \quad 1 \quad 0]$ if negative, prints an error message, and returns –1 (0 is the normal return). The view struct is changed to hold the altered view up. The error message may be suppressed by setting **VuPrintError** to FALSE.

VuCamera(vup, r, n, up, deye)
struct LFL_View* vup;
vector r, n, up;
MxWorldType deye;

The point $[xr \quad yr \quad zr \quad 1]$ is any convenient reference point. The vector $[xn \quad yn \quad zn \quad 1]$ defines the ViewNormal. The vector $[xup \quad yup \quad zup \quad 1]$ defines ViewUp. In this case ViewUp is not necessarily perpendicular to ViewNormal, as permitted by the view spec. A collinearity check is made and the view up portion of the view struct is changed to be arbitrarily orthogonal to the view normal in case of collinearity. A return of –1 (rather than the normal 0) occurs in this case. An error message is printed in this case unless **VuPrintError** is set to FALSE. The ViewPoint is located distance *deye* from the reference point along and in the direction of the ViewNormal.

This routine is more powerful and computationally cheaper than either of the preceding two routines.

VuPerspective(vup, fieldofview, aspectratio, near, far)
struct LFL_View* vup;
double fieldofview, aspectratio;
MxWorldType near, far;

Sets the WindowCenter, WindowHalfsize, NearDistance, and FarDistance of the given view struct. FarDistance of 0 represents a far clipping plane at infinity. A view window specified with this routine is assumed to be centered on this ViewNormal. The view plane and the near clipping plane are assumed to be the same in this routine—i.e., NearDistance equals ViewDistance. The *fieldofview* argument is the full horizontal field-of-view angle in

degrees for the desired viewing window. *aspectratio* is the desired width-to-height aspect ratio for it. If the window is subsequently moved off center, it remains the same size and shape but the field-of-view angle is no longer accurate.

VuWindow(vup, wleft, wright, wtop, wbottom, near, far)
struct LFL_View* vup;
MxWorldType wleft, wright, wtop, wbottom, near, far;

Sets the WindowCenter, WindowHalfsize, NearDistance, and FarDistance of the given view struct. FarDistance of 0 represents a far clipping plane at infinity. A view window specified by this routine does not have to be centered on the ViewNormal. The view plane and the near clipping plane are assumed to be the same in this routine—i.e., NearDistance equals ViewDistance.

The following routines generate the matrices used in Tech Memo 84:

VuGet[ABCDEFGHNPQR](vup, m)
struct LFL_View* vup;
matrix m;

This notation stands for the routines **VuGetA()**, **VuGetB()**, etc. Each routine returns in *m* the matrix of the corresponding name from Tech Memo 84. Thus, for example, **VuGetA()** returns matrix **A**. This set of routines returns those matrices dependent on a view structure. **VuGetN()** is the same as **VuPerspNormalization()**; **VuGetP()** is the same as **VuPerspTransformation()**; and **VuGetQ()** is the same as **VuPerspProjection()**. In all cases affected by the far plane at infinity, appropriate action is taken as explained in Tech Memo 84.

VuGet[JKLMS](disp, m)
struct LFL_Display* disp;
matrix m;

Another class of routines like that above for generating the matrices mentioned in Tech Memo 84. In this case, however, the matrices are those dependent on a display structure—i.e., those used for generating a screen mapping.

VuGetNsubL(vup, m)
struct LFL_View* vup;
matrix m;

Generates the matrix $\mathbf{N}_L = \mathbf{AB}$ of Tech Memo 84. This is the part of the perspective normalization which changes as the camera moves. The camera is otherwise rigid—i.e., its viewing frustum is rigid, only its position and orientation change. An error return of –1 occurs for collinear ViewUp and ViewNormal (see **VuGetUVN()** above for full explanation and the actions taken) else 0 is returned.

VuGetNsubR(vup, m)
struct LFL_View* vup;
matrix m;

The complement of the preceding routine, this generates matrix $\mathbf{N}_R = \mathbf{CD}$ of Tech Memo 84. This is the part of the perspective normalization which is fixed for a rigid, but moving camera.

VuGetNsuboR(vup, m)
struct LFL_View* vup;
matrix m;

Same as above but for orthographic normalization. The matrix generated is $\mathbf{N}_{oR}$ of Tech Memo 84.

VuGetJsubr(disp, m)
struct LFL_Display* disp;
matrix m;

Generates matrix $\mathbf{J}_r$ of Tech Memo 84. This matrix is the coordinate transformation part of the screen mapping specialized for raster devices.

VuGetJsubc(disp, m)
struct LFL_Display* disp;
matrix m;

Generates matrix $\mathbf{J}_c$ of Tech Memo 84. This matrix is the coordinate transformation part of the screen mapping specialized for calligraphic devices.

VuGetNsubo(vup, m)
struct LFL_View* vup;
matrix m;

Similar to **VuGetN()** but for orthographic normalization. This routine is the same as **VuOrthoNormalization()**. An error return of –1 occurs for collinear ViewUp and ViewNormal (see **VuGetUVN()** above for full explanation and the actions taken) else 0 is returned.

VuGetSsubr(disp, m)
struct LFL_Display* disp;
matrix m;

Generates the raster screen mapping $\mathbf{S}_r$ of Tech Memo 84. The same as **VuRasterMap()**.

VuGetSsubc(disp, m)
struct LFL_Display* disp;
matrix m;

Generates the calligraphic screen mapping $\mathbf{S}_c$ of Tech Memo 84. The same as **VuCalligraphicMap()**.

The following routines are devoted to the screen mapping part of the view transformation.

VuRasterInit(disp)
struct LFL_Display* disp;

Initializes the given display to the default raster values defined in *struct LFL_Display VuRasterDisplay*.

VuCalligraphicInit(disp)
struct LFL_Display* disp;

Initializes the given display to the default calligraphic values defined in *struct LFL_Display VuCalligraphicDisplay*.

VuGetScreenMin(disp, v)
struct LFL_Display* disp;
vector v;

Returns X_{min}, Y_{min}, and Z_{min} of the current window in the given display (as opposed to the full screen window).

VuGetScreenMax(disp, v)
struct LFL_Display* disp;
vector v;

Returns X_{max}, Y_{max}, and Z_{max} of the current window in the given display (as opposed to the full screen window).

VuGetScreenWindow(disp, xminp, xmaxp, yminp, ymaxp, zminp, zmaxp)
struct LFL_Display* disp;
double *xminp, *xmaxp, *yminp, *ymaxp, *zminp, *zmaxp;

A non-vector form of the two preceding routines.

VuGetFullScreenMin(disp, v)
struct LFL_Display* disp;
vector v;

Returns X_{min}, Y_{min}, and Z_{min} of the full screen display window. Although the current display window may vary, the full screen window is fixed for a given device.

VuGetFullScreenMax(disp, v)
struct LFL_Display* disp;
vector v;

Returns X_{max}, Y_{max}, and Z_{max} of the full screen display window.

VuGetFullScreenWindow(disp, xminp, xmaxp, yminp, ymaxp, zminp, zmaxp)
struct LFL_Display* disp;
double *xminp, *xmaxp, *yminp, *ymaxp, *zminp, *zmaxp;

A non-vector form of the two preceding routines.

double VuGetFullScreenAspectRatio(disp)
struct LFL_Display* disp;

Returns the width to height of the full screen window. This is fixed for any given device (unless it is physically altered).

double VuGetPixelAspectRatio(disp)
struct LFL_Display* disp;

Derives and returns the pixel [spacing] aspect ratio of the given device. This is a measure of the nonsquareness of a device. This is fixed for any given device (unless it is physically altered). It is determined by the formula *PixelAspectRatio = FullScreenAspectRatio * VResolution / HResolution*. VResolution and HResolution can be determined from FullScreenMin and FullScreenMax.

VuGetScreenNormal(disp, n)
struct LFL_Display* disp;
vector n;

Returns the ScreenNormal in Normalized Device Coordinates (NDC). This is assumed to be a unit vector (disregarding *n*[3]). *n*[3] is set to 1.

VuGetScreenUp(disp, v)
struct LFL_Display* disp;
vector v;

Returns the ScreenUp in NDC. This is assumed to be a unit vector (disregarding *v*[3]). *v*[3] is set to 1.

VuGetScreenRight(disp, u)
struct LFL_Display* disp;
vector u;

Returns the screen right vector in NDC. This is assumed to be $\begin{bmatrix}1 & 0 & 0 & 1\end{bmatrix}$ for all devices.

VuRasterMap(disp, m)
struct LFL_Display* disp;
matrix m;

Generates the raster screen mapping $\mathbf{S}_r$ of Tech Memo 84. The same as **VuGetSsubr()**.

VuCalligraphicMap(disp, m)
struct LFL_Display* disp;
matrix m;

Generates the calligraphic screen mapping $\mathbf{S}_c$ of Tech Memo 84. The same as **VuGetSsubc()**.

VuDisplayPrint(disp)
struct LFL_Display* disp;

Print an ascii version of the given display struct on *stdout*.

VuFileDisplayPrint(file, disp)
FILE* file;
struct LFL_Display* disp;

Print an ascii version of the given display struct to the given file. This is the conventional way of passing display definitions between programs (cf. **VuFileDisplayScan()** below).

VuMiscDisplayPrint(disp)
struct LFL_Display* disp;

Print an ascii version of miscellaneous derived parameters of the given display struct. In particular, the pixel [spacing] aspect ratio is derived and printed.

VuDisplayScan(disp)
struct LFL_Display* disp;

Scan an ascii version of a display definition into the given display struct from *stdin*.

VuFileDisplayScan(file, disp)
FILE* file;
struct LFL_Display* disp;

Scan an ascii version of a display definition into the given display struct from the given file. This is the conventional way of passing display definitions between programs (cf. **VuFileDisplayPrint()** above).

VuSetScreenMin(disp, v)
struct LFL_Display* disp;
vector v;

Set ScreenMin of the given display struct to the given values. *v*[3] is ignored.

VuSetScreenMax(disp, v)
struct LFL_Display* disp;
vector v;

Set ScreenMax of the given display struct to the given values. *v*[3] is ignored.

VuSetScreenWindow(disp, xmin, xmax, ymin, ymax, zmin, zmax)
struct LFL_Display* disp;
double xmin, xmax, ymin, ymax, zmin, zmax;

A non-vector form of the two routines above.

VuSetScreenWindow2D(disp, aspectratio, xmin, ymin, pixelwidth,pixelheight)
struct LFL_Display* disp;
double aspectratio, xmin, ymin, pixelwidth, pixelheight;

The screen window of the display described by the given display structure is set to have the given *aspectratio* which is assumed to be the aspect ratio of a view window meant to be mapped to the specified display window. The corner of the desired window is pegged at (*xmin*, *xmax*). It is scaled to have either width *pixelwidth* or height *pixelheight*, but not both. So one of the arguments, but not both, should be 0. An error message is printed otherwise and a –1 is returned; 0 is the normal return. The error message may be suppressed by setting **VuPrintError** to FALSE. The *z* component of the screen

mapping is not affected by this routine and must be set separately. No checks for invalid windows are made.

VuSetFullScreenMin(disp, v)
struct LFL_Display* disp;
vector v;
Set FullScreenMin of the given display struct to the given values. *v*[3] is ignored.

VuSetFullScreenMax(disp, v)
struct LFL_Display* disp;
vector v;
Set FullScreenMax of the given display struct to the given values. *v*[3] is ignored.

VuSetFullScreenWindow(disp, xmin, xmax, ymin, ymax, zmin, zmax)
struct LFL_Display* disp;
double xmin, xmax, ymin, ymax, zmin, zmax;
A non-vector form of the two routines above.

VuSetFullScreenAspectRatio(disp, screenaspectratio)
struct LFL_Display* disp;
double screenaspectratio;
Set the FullScreenAspectRatio of the given display struct to the given value.

VuSetScreenNormal(disp, v)
struct LFL_Display* disp;
vector v;
Set ScreenNormal of the given display struct to the given vector normalized. It is the user's responsibility to make ScreenNormal orthogonal to ScreenUp.

VuSetScreenUp(disp, v)
struct LFL_Display* disp;
vector v;
Set ScreenUp of the given display struct to the given vector normalized. It is the user's responsibility to make ScreenUp orthogonal to ScreenNormal.

Following are routines for clipping points and lines against canonical viewing volumes after perspective or orthographic normalization:

double VuGetCVFzmin(vup)
struct LFL_View* vup;
Returns the minimum *z* coordinate of the canonical viewing frustum (CVF) corresponding to the assumed perspective view given.

VuFastPerspClipPoint(vup, p, zmin)
struct LFL_View* vup;
vector p;
double zmin;

A fast version of **VuPerspClipPoint()** described below. It is faster because *zmin* is provided (perhaps computed with **VuGetCVFzmin()** above). Since *zmin* changes only when near, far, or view plane distances change, it does not need to be recomputed for every clip.

VuPerspClipPoint(vup, p)
struct LFL_View* vup;
vector p;

Clips the given point *p* against the canonical viewing frustum corresponding to the given perspective view. The routine returns the Cohen-Sutherland outcode which has six bits called VU_ABOVE, VU_BELOW, VU_RIGHT, VU_LEFT, VU_BEHIND, or VU_BEFORE (defined in **VuSpec.h**). If none of these bits is set (i.e., outcode is 0), then the point is inside the viewing volume. The special case of far plane at infinity is handled.

VuOrthoClipPoint(vup, p)
struct LFL_View* vup;
vector p;

Clips the given point *p* against the canonical viewing volume for orthographic views. The routine returns the Cohen-Sutherland outcode which has six bits called VU_ABOVE, VU_BELOW, VU_RIGHT, VU_LEFT, VU_BEHIND, or VU_BEFORE (defined in **VuSpec.h**). If none of these bits is set (i.e., outcode is 0), then the point is inside the viewing volume. The special case of far plane at infinity is handled.

VuFastPerspClipLine(vup, p, q, nup, nuq, zmin)
struct LFL_View* vup;
vector p, q, nup, nuq;
double zmin;

This routine is to **VuPerspClipLine()** below as **VuFastPerspClipPoint()** is to **VuPerspClipPoint()** above.

VuPerspClipLine(vup, p, q, nup, nuq)
struct LFL_View* vup;
vector p, q, nup, nuq;

Returns VU_REJECT (defined in **VuSpec.h**) if the line segment from point *p* to point *q* is entirely outside the canonical viewing frustum corresponding to the given perspective view. Otherwise it returns VU_ACCEPT and places the clipped line segment endpoints in *nup* and *nuq* (which are undefined for case VU_REJECT). *p* and *q* are not touched.

VuOrthoClipLine(vup, p, q, nup, nuq)
struct LFL_View* vup;
vector p, q, nup, nuq;

Returns VU_REJECT (defined in **VuSpec.h**) if the line segment from point *p* to point *q* is entirely outside the canonical viewing volume for orthographic

views. Otherwise it returns VU_ACCEPT and places the clipped line segment endpoints in *nup* and *nuq* (which are undefined for case VU_REJECT). *p* and *q* are not touched.

Appendix 6: The CvuSpec Package

This package is the *VuSpec* package (Appendix 5) specialized for the notion of "current view" *Cvu* and "current display" *CurrentDisplay*. These are defined in *VuSpec.h*. The source resides in files *VuSpec.h*, *CvuSpec.c*, *CvuDisplay.c*, *CvuMatrix.c*, *CvuClip.c*.

```
#include <MxMatrix.h>
#include <VuSpec.h>
#include <stdio.h>
typedef double MxWorldType;
```

CvuViewInit()

Initialize current view to default values defined in *struct LFL_View CvuReference* which is a perspective view.

CvuSetViewPoint(v)
vector v;

Set the viewpoint of the current view to the given point in world space.

CvuSetViewNormal(v)
vector v;

Set the view normal of the current view to the given vector in world space.

CvuSetViewUp(v)
vector v;

Set the view up of the current view to the given vector in world space. The component of this vector perpendicular to the view normal vector defines the up direction. This routine does not check for the view up vector collinear with the view normal.

CvuSetViewDistance(d)
MxWorldType d;

Set the distance of the current view to the given distance, which is the distance along the view normal from the view point at which the view plane lies. The distance is in world space.

CvuSetNearFar(near, far)
MxWorldType near, far;

Set the near and far clipping plane distances of the current view. The planes are located at the given distances from the view point along the view normal. If *far* is 0., then the far clipping plane is at infinity. The distances are in world space.

CvuSetViewWindow(centeru, centerv, halfsizeu, halfsizev)

MxWorldType centeru, centerv, halfsizeu, halfsizev;

Set the window of the current view by specifying the location of its center relative the point where the view normal intersects the view plane. The coordinates are in world space. The horizontal and vertical halfsizes complete the definition. They are also expressed in world space. Coordinate *v* is parallel the up vector (not view up) and coordinate *u* is perpendicular to it, parallel the right vector which points to the right when looking out the view normal with the up vector pointing up.

CvuSetProjectionType(type)

Set the projection type of the current view to either VU_PERSPECTIVE or VU_ORTHOGRAPHIC (defined in **VuSpec.h**).

CvuSetAspectRatio(aspectratio)
double aspectratio;

Resets WindowHalfsize.v in the current view so that the width to height aspect ratio is as given. WindowHalfsize.u is unchanged, so the field-of-view angle is unchanged (cf. **CvuSetFieldOfView()** below) when meaningful. Special aspect ratios VU_SILENT, VU_VIDEO, VU_ACADEMY, VU_PANAVISION, VU_CINEMASCOPE, VU_TODDAO, VU_SUPERPANAVISION, VU_ULTRAPANAVISION, VU_IMAX, and VU_OMNIMAX are defined in **VuSpec.h**.

CvuSetFieldOfView(angle)
double angle;

Resets WindowHalfsize.u in the current view so that the full horizontal field-of-view angle is *angle* degrees. WindowHalfsize.v is also reset to maintain aspect ratio. ViewDistance is unchanged. N.B. Field of view for off-axis view windows is invalid. (Cf. **CvuSetAspectRatio()** above.) The routine returns a 0 normally but a –1 in case WindowHalfsize.u is 0.

CvuGetViewPoint(v)
vector v;

Returns the view point from the current view. *v*[3] is set to 1.

CvuGetViewNormal(v)
vector v;

Returns the view normal from the current view. *v*[3] is set to 1.

CvuGetViewUp(v)
vector v;

Returns the view up from the current view. *v*[3] is set to 1. This routine does not check for view up vector collinear with the view normal.

CvuGetUnitNormal(n)
vector n;

Returns the unit vector in the viewing direction of the current view. *n*[3] is set to 1.

CvuGetUnitUp(v)
vector v;

Returns the unit vector in the up direction of the current view. This is the normalized component of the view up vector orthogonal to the view normal and in the plane defined by the up and normal vectors. *v*[3] is set to 1. This routine checks for view up collinear with view normal. If collinear, *v* is set arbitrarily to be orthogonal to view normal, an error message is printed (if **CvuPrintError** is TRUE), and –1 is returned instead of the normal 0. The view struct is not changed.

CvuGetUnitRight(u)
vector u;

Returns the unit vector towards the right for the current view. This is the unit vector orthogonal to the unit up and unit normal vectors defined above, by the lefthand rule. *u*[3] is set to 1. This routine checks for view up collinear with view normal. If collinear, an arbitrary view up orthogonal to view normal is used to compute *u*, an error message is printed (if **CvuPrintError** is TRUE), and –1 is returned instead of the normal 0. The view struct is not changed.

MxWorldType CvuGetViewDistance()

Returns the view distance of the current view.

CvuGetNearFar(nearp, farp)
MxWorldType *nearp, *farp;

Returns the near distance and far distance of the current view.

double CvuGetAspectRatio()

Returns the aspect ratio of the viewing window defined in the current view. It is defined as *WindowHalfsize.u / WindowHalfsize.v*. An error return of –1. occurs for a WindowHalfsize.v of 0.

double CvuGetFieldOfView()

Returns the full horizontal field-of-view angle (in degrees) associated with the current view. It is computed as *2 * atan(WindowHalfsize.u / ViewDistance)*. This angle is valid only for centered view windows.

CvuGetViewWindow(centerup, centervp, halfsizeup, halfsizevp)
MxWorldType *centerup, *centervp, *halfsizeup, *halfsizevp;

Returns the view window description from the current view.

CvuGetProjectionType()

Returns the projection type of the current view (VU_PERSPECTIVE or VU_ORTHOGRAPHIC, defined in **VuSpec.h**).

CvuViewPrint()

Print an ascii version of the current view on *stdout*.

CvuFileViewPrint(file)

FILE* file;

Print an ascii version of the current view to the given file. This is the conventional way of passing viewing definitions between programs (cf. **CvuFileViewScan()** below).

CvuMiscPrint()

Print an ascii version of miscellaneous derived parameters from the current view on *stdout*. These parameters are field-of-view angle, view window aspect ratio, and the azimuth, pitch, and roll of the given view (cf. **CvuGetAspectRatio()**, **CvuGetFieldOfView()**, and **CvuGetViewAngles()**).

CvuFileMiscPrint(file)
FILE* file;

Print an ascii version of miscellaneous derived parameters from the current view to the given file. See **CvuMiscPrint()** above.

CvuViewScan()

Scan an ascii version of a viewing definition into the current view from *stdin*.

CvuFileViewScan(file)
FILE* file;

Scan an ascii version of a viewing definition into the current view from the given file. This is the conventional way of passing viewing definitions between programs (cf. **CvuFileViewPrint()** above).

CvuStackInit()

Initialize the view stack.

char* CvuPush()

Push the current view onto the view stack. Normal return is NULL. There is an error message returned in case of stack overflow. The length of the stack VU_STKLEN is defined in **VuSpec.h**.

char* CvuPop()

Pop the view stack into the current view. Normal return is NULL. There is an error message returned in case of stack underflow.

CvuPerspNormalization(m)
matrix m;

The perspective normalization matrix (**N** in Tech Memo 84) is derived from the current view and returned in matrix *m*. An error return of –1 occurs for collinear ViewUp and ViewNormal (see **CvuGetUVN()** below for full explanation and the actions taken) else 0 is returned. The special case of far plane at infinity is handled as explained in Tech Memo 84. A ViewDistance of zero is an error in this case and causes a –1 return.

CvuOrthoNormalization(m)
matrix m;

The orthographic normalization matrix ($\mathbf{N}_o$ in Tech Memo 84) is derived from the current view and returned in matrix *m*. An error return of –1 occurs for collinear ViewUp and ViewNormal (see **CvuGetUVN()** below for full explanation and the actions taken) else 0 is returned. FarDisance equal NearDistance causes an error return also. The special case of far plane at infinity is handled as explained in Tech Memo 84. A ViewDistance of zero is an error in this case and causes a –1 return.

CvuPerspTransformation(m)
matrix m;

The perspective transformation matrix (**P** in Tech Memo 84) is derived from the current view and returned in matrix *m*. The special case of far plane at infinity is handled as explained in Tech Memo 84. A –1 is returned in case ViewDistance is 0 and an error message is printed (suppressed if **CvuPrintError** is set to FALSE); normal return is 0. FarDisance equal NearDistance causes an error return also.

CvuPerspProjection(m)
matrix m;

The perspective projection matrix (**Q** in Tech Memo 84) is derived from the current view and returned in matrix *m*. The special case of far plane at infinity is handled as explained in Tech Memo 84. A –1 is returned in case ViewDistance is 0 and an error message is printed (suppressed if **CvuPrintError** is set to FALSE); normal return is 0.

CvuCollinear(up)
vector up;

Checks the current view to see if the view up vector is collinear with the view normal vector. It returns TRUE in case of collinearity and FALSE otherwise. In either case, *up* contains a vector orthogonal to the view normal upon return. In case of collinearity, this vector is arbitrarily chosen. Otherwise, it is just the view up vector. The view struct is not changed.

CvuGetUVN(u, v, n)
vector u, v, n;

Computes the three orthogonal unit vectors defining the view space described in the current view. (The first three elements of each vector is a unit vector; the fourth element is arbitrarily set to 1. for all three vectors.) *n* is the unit normal obtained from ViewNormal. *v* is the unit up vector obtained from ViewUp as defined in Tech Memo 84. The ViewUp vector is not necessarily perpendicular to *n* so *v* is that component which is. Then *u*, the unit right vector, is obtained by cross product to be orthogonal to *v* and *n* and forms a lefthanded coordinate system with them. If the ViewUp vector happens to be collinear with the ViewNormal vector, an error message is printed, *v* is made arbitrarily orthogonal to *n* (it may be meaningless however),

and a –1 is returned. Normal return is a 0. The view struct is not changed. The error message may be suppressed by setting **CvuPrintError** to FALSE.

Following are several convenient interfaces to the view spec. They follow closely routines by similar names written by Bill Reeves in an earlier package.

CvuSetViewFromAngles(azimuth, pitch, roll)
double azimuth, pitch, roll;

ViewNormal and ViewUp of the current view are determined from angles *azimuth*, *pitch*, and *roll* as defined in **CvuView()** below.

CvuGetAnglesFromView(azimuthp, pitchp, rollp)
double *azimuthp, *pitchp, *rollp;

The inverse of **CvuSetViewFromAngles()** above. Angles returned are not necessarily those used to generate the view. For example, 480 degrees would be returned as 120, and a roll of 200 degrees would be returned as –160. A collinearity check is made identical in action to **CvuGetUnitRight()** above. The view struct is not changed.

CvuView(distance, azimuth, pitch, roll)
double distance, azimuth, pitch, roll;

Defines the location of the ViewPoint by giving its *distance* from the world space origin along a radius vector at angle *azimuth* degrees in the *xy* plane, measured positive by the lefthand rule on the *z* axis, and at angle *pitch* degrees from the *xy* plane, measured positive above the *xy* plane (positive *z* direction) and negative below. Azimuth is measured relative the positive *y* axis. The vector from the origin to this point defines the ViewNormal. ViewUp is derived from the angle *roll* measured about the ViewNormal using the lefthand rule. Roll equal 0 degrees implies the rightward view vector is parallel the *xy* plane with the *z* component of ViewUp positive. ViewUp is always set perpendicular to ViewNormal by this routine which may be used with **CvuWindow()** or **CvuPerspective()** to completely fill the current view struct. Alternatively, **CvuFullView()**, **CvuLookAt()**, or **CvuCamera()** may be used in its place.

CvuFullView(V, azimuth, pitch, roll)
vector V;
double azimuth, pitch, roll;

The ViewPoint is placed at the location specified by vector *V* in world space. The ViewNormal is located as described above with the angles *azimuth* and *pitch*, but they are measured relative a coordinate system centered at the ViewPoint which is just a simple translation of the world space system. ViewUp is determined as above from the *roll* angle.

CvuLookAt(V, p, roll)
vector V, p;
double roll;

Locates the ViewPoint at $[Vx \quad Vy \quad Vz \quad 1]$. The ViewNormal direction is that defined by a vector from ViewPoint to $[px \quad py \quad pz \quad 1]$. *Roll* is interpreted as above. In particular, ViewUp is always set perpendicular to ViewNormal.

This routine does not supply sufficient data for computing a unique view when $p - V$ is parallel the z axis. In this case, ViewUp can be anything so this routine sets it to $[0 \quad -1 \quad 0]$ if the z component of ViewNormal is positive or to $[0 \quad 1 \quad 0]$ if negative, prints an error message, and returns –1 (0 is the normal return). The view struct is changed to hold the altered view up. The error message may be suppressed by setting **CvuPrintError** to FALSE.

CvuCamera(r, n, up, deye)
vector r, n, up;
MxWorldType deye;

The point $[xr \quad yr \quad zr \quad 1]$ is any convenient reference point. The vector $[xn \quad yn \quad zn \quad 1]$ defines the ViewNormal. The vector $[xup \quad yup \quad zup \quad 1]$ defines ViewUp. In this case ViewUp is not necessarily perpendicular to ViewNormal, as permitted by the view spec. A collinearity check is made and the view up portion of the current view is changed to be arbitrarily orthogonal to the view normal in case of collinearity. A return of –1 (rather than the normal 0) occurs in this case. An error message is printed in this case unless **CvuPrintError** is set to FALSE. The ViewPoint is located distance *deye* from the reference point along and in the direction of the ViewNormal.

This routine is more powerful and computationally cheaper than either of the preceding two routines.

CvuPerspective(fieldofview, aspectratio, near, far)
double fieldofview, aspectratio;
MxWorldType near, far;

Sets the WindowCenter, WindowHalfsize, NearDistance, and FarDistance of the current view. FarDistance of 0 represents a far clipping plane at infinity. A view window specified with this routine is assumed to be centered on this ViewNormal. The view plane and the near clipping plane are assumed to be the same in this routine—i.e., NearDistance equals ViewDistance. The *fieldofview* argument is the full horizontal field-of-view angle in degrees for the desired viewing window. *aspectratio* is the desired width-to-height aspect ratio for it. If the window is subsequently moved off center, it remains the same size and shape but the field-of-view angle is no longer accurate.

CvuWindow(wleft, wright, wtop, wbottom, near, far)
MxWorldType wleft, wright, wtop, wbottom, near, far;

Sets the WindowCenter, WindowHalfsize, NearDistance, and FarDistance of the current view. FarDistance of 0 represents a far clipping plane at infinity. A view window specified by this routine does not have to be centered on the

ViewNormal. The view plane and the near clipping plane are assumed to be the same in this routine—i.e., NearDistance equals ViewDistance.

The following routines generate the matrices used in Tech Memo 84:

CvuGet[ABCDEFGHNPQR](m)
matrix m;

This notation stands for the routines **CvuGetA()**, **CvuGetB()**, etc. Each routine returns in *m* the matrix of the corresponding name from Tech Memo 84. Thus, for example, **CvuGetA()** returns matrix **A**. This set of routines returns those matrices dependent on the current view structure. **CvuGetN()** is the same as **CvuPerspNormalization()**; **CvuGetP()** is the same as **CvuPerspTransformation()**; and **CvuGetQ()** is the same as **CvuPerspProjection()**. In all cases affected by the far plane at infinity, appropriate action is taken as explained in Tech Memo 84.

CvuGet[JKLMS](m)
matrix m;

Another class of routines like that above for generating the matrices mentioned in Tech Memo 84. In this case, however, the matrices are those dependent on the current display structure—i.e., those used for generating a screen mapping.

CvuGetNsubL(m)
matrix m;

Generates the matrix $\mathbf{N}_L = \mathbf{AB}$ of Tech Memo 84. This is the part of the perspective normalization which changes as the camera moves. The camera is otherwise rigid—i.e., its viewing frustum is rigid, only its position and orientation change. An error return of –1 occurs for collinear ViewUp and ViewNormal (see **CvuGetUVN()** above for full explanation and the actions taken) else 0 is returned.

CvuGetNsubR(m)
matrix m;

The complement of the preceding routine, this generates matrix $\mathbf{N}_R = \mathbf{CD}$ of Tech Memo 84. This is the part of the perspective normalization which is fixed for a rigid, but moving camera.

CvuGetNsuboR(m)
matrix m;

Same as above but for orthographic normalization. The matrix generated is $\mathbf{N}_{oR}$ of Tech Memo 84.

CvuGetJsubr(m)
matrix m;

Generates matrix $\mathbf{J}_r$ of Tech Memo 84. This matrix is the coordinate transformation part of the current screen mapping specialized for raster devices.

CvuGetJsubc(m)
matrix m;

Generates matrix $\mathbf{J}_c$ of Tech Memo 84. This matrix is the coordinate transformation part of the current screen mapping specialized for calligraphic devices.

CvuGetNsubo(m)
matrix m;

Similar to **CvuGetN()** but for orthographic normalization. This routine is the same as **CvuOrthoNormalization()**. An error return of –1 occurs for collinear ViewUp and ViewNormal (see **CvuGetUVN()** above for full explanation and the actions taken) else 0 is returned.

CvuGetSsubr(m)
matrix m;

Generates the raster screen mapping $\mathbf{S}_r$ of Tech Memo 84. The same as **CvuRasterMap()**.

CvuGetSsubc(m)
matrix m;

Generates the calligraphic screen mapping $\mathbf{S}_c$ of Tech Memo 84. The same as **CvuCalligraphicMap()**.

The following routines are devoted to the screen mapping part of the view transformation.

CvuRasterInit()

Initializes the current display to the default raster values defined in *struct LFL_Display CvuRasterDisplay*.

CvuCalligraphicInit()

Initializes the current display to the default calligraphic values defined in *struct LFL_Display CvuCalligraphicDisplay*.

CvuGetScreenMin(v)
vector v;

Returns $X_{\min}$, $Y_{\min}$, and $Z_{\min}$ of the current window in the current display (as opposed to the full screen window).

CvuGetScreenMax(v)
vector v;

Returns $X_{\max}$, $Y_{\max}$, and $Z_{\max}$ of the current window in the current display (as opposed to the full screen window).

CvuGetScreenWindow(xminp, xmaxp, yminp, ymaxp, zminp, zmaxp)
double *xminp, *xmaxp, *yminp, *ymaxp, *zminp, *zmaxp;

A non-vector form of the two preceding routines.

CvuGetFullScreenMin(v)

vector v;

Returns $X_{\min}$, $Y_{\min}$, and $Z_{\min}$ of the full screen display window. Although the current display window may vary, the full screen window is fixed for a given device.

CvuGetFullScreenMax(v)
vector v;

Returns $X_{\max}$, $Y_{\max}$, and $Z_{\max}$ of the full screen display window.

CvuGetFullScreenWindow(xminp, xmaxp, yminp, ymaxp, zminp, zmaxp)
double *xminp, *xmaxp, *yminp, *ymaxp, *zminp, *zmaxp;

A non-vector form of the two preceding routines.

double CvuGetFullScreenAspectRatio()

Returns the width to height of the full screen window. This is fixed for any given device (unless it is physically altered).

double CvuGetPixelAspectRatio()

Derives and returns the pixel [spacing] aspect ratio of the current display. This is a measure of the nonsquareness of a device. This is fixed for any given device (unless it is physically altered). It is determined by the formula *PixelAspectRatio = FullScreenAspectRatio * VResolution / HResolution*. VResolution and HResolution can be determined from FullScreenMin and FullScreenMax.

CvuGetScreenNormal(n)
vector n;

Returns the ScreenNormal in Normalized Device Coordinates (NDC). This is assumed to be a unit vector (disregarding $n[3]$). $n[3]$ is set to 1.

CvuGetScreenUp(v)
vector v;

Returns the ScreenUp in NDC. This is assumed to be a unit vector (disregarding $v[3]$). $v[3]$ is set to 1.

CvuGetScreenRight(u)
vector u;

Returns the screen right vector in NDC. This is assumed to be $\begin{bmatrix}1 & 0 & 0 & 1\end{bmatrix}$ for all devices.

CvuRasterMap(m)
matrix m;

Generates the raster screen mapping $\mathbf{S}_r$ of Tech Memo 84. The same as **CvuGetSsubr()**.

CvuCalligraphicMap(m)
matrix m;

Generates the calligraphic screen mapping $\mathbf{S}_c$ of Tech Memo 84. The same as **CvuGetSsubc()**.

CvuDisplayPrint()
Print an ascii version of the current display on *stdout*.

CvuFileDisplayPrint(file)
FILE* file;
Print an ascii version of the current display to the given file. This is the conventional way of passing display definitions between programs (cf. **CvuFileDisplayScan()** below).

CvuMiscDisplayPrint()
Print an ascii version of miscellaneous derived parameters of the current display. In particular, the pixel [spacing] aspect ratio is derived and printed.

CvuDisplayScan()
Scan an ascii version of a display definition into the current display struct from *stdin*.

CvuFileDisplayScan(file)
FILE* file;
Scan an ascii version of a display definition into the current display struct from the given file. This is the conventional way of passing display definitions between programs (cf. **CvuFileDisplayPrint()** above).

CvuSetScreenMin(v)
vector v;
Set ScreenMin of the current display to the given values. *v*[3] is ignored.

CvuSetScreenMax(v)
vector v;
Set ScreenMax of the current display to the given values. *v*[3] is ignored.

CvuSetScreenWindow(xmin, xmax, ymin, ymax, zmin, zmax)
double xmin, xmax, ymin, ymax, zmin, zmax;
A non-vector form of the two routines above.

CvuSetScreenWindow2D(aspectratio, xmin, ymin, pixelwidth, pixelheight)
double aspectratio, xmin, ymin, pixelwidth, pixelheight;
The screen window of the display described by the current display structure is set to have the given *aspectratio* which is assumed to be the aspect ratio of a view window meant to be mapped to the specified display window. The corner of the desired window is pegged at (*xmin*, *xmax*). It is scaled to have either width *pixelwidth* or height *pixelheight*, but not both. So one of the arguments, but not both, should be 0. An error message is printed otherwise and a –1 is returned; 0 is the normal return. The error message may be suppressed by setting **CvuPrintError** to FALSE. The *z* component of the screen mapping is not affected by this routine and must be set separately. No checks for invalid windows are made.

CvuSetFullScreenMin(v)

vector v;
Set FullScreenMin of the current display to the given values. *v*[3] is ignored.

CvuSetFullScreenMax(v)
vector v;
Set FullScreenMax of the current display to the given values. *v*[3] is ignored.

CvuSetFullScreenWindow(xmin, xmax, ymin, ymax, zmin, zmax)
double xmin, xmax, ymin, ymax, zmin, zmax;
A non-vector form of the two routines above.

CvuSetFullScreenAspectRatio(screenaspectratio)
double screenaspectratio;
Set the FullScreenAspectRatio of the current display to the given value.

CvuSetScreenNormal(v)
vector v;
Set ScreenNormal of the current display to the given vector normalized. It is the user's responsibility to make ScreenNormal orthogonal to ScreenUp.

CvuSetScreenUp(v)
vector v;
Set ScreenUp of the current display to the given vector normalized. It is the user's responsibility to make ScreenUp orthogonal to ScreenNormal.

Following are routines for clipping points and lines against canonical viewing volumes after perspective or orthographic normalization:

double CvuGetCVFzmin()
Returns the minimum *z* coordinate of the canonical viewing frustum (CVF) corresponding to the assumed perspective current view.

CvuFastPerspClipPoint(p, zmin)
vector p;
double zmin;
A fast version of **CvuPerspClipPoint()** described below. It is faster because *zmin* is provided (perhaps computed with **CvuGetCVFzmin()** above). Since *zmin* changes only when near, far, or view plane distances change, it does not need to be recomputed for every clip.

CvuPerspClipPoint(p)
vector p;
Clips the given point *p* against the canonical viewing frustum corresponding to the current perspective view. The routine returns the Cohen-Sutherland outcode which has six bits called VU_ABOVE, VU_BELOW, VU_RIGHT, VU_LEFT, VU_BEHIND, or VU_BEFORE (defined in **VuSpec.h**). If none of these bits is set (i.e., outcode is 0), then the point is inside the viewing volume. The special case of far plane at infinity is handled.

CvuOrthoClipPoint(p)
vector p;

Clips the given point *p* against the canonical viewing volume for the current orthographic view. The routine returns the Cohen-Sutherland outcode which has six bits called VU_ABOVE, VU_BELOW, VU_RIGHT, VU_LEFT, VU_BEHIND, or VU_BEFORE (defined in **VuSpec.h**). If none of these bits is set (i.e., outcode is 0), then the point is inside the viewing volume. The special case of far plane at infinity is handled.

CvuFastPerspClipLine(p, q, nup, nuq, zmin)
vector p, q, nup, nuq;
double zmin;

This routine is to **CvuPerspClipLine()** below as **CvuFastPerspClipPoint()** is to **CvuPerspClipPoint()** above.

CvuPerspClipLine(p, q, nup, nuq)
vector p, q, nup, nuq;

Returns VU_REJECT (defined in **VuSpec.h**) if the line segment from point *p* to point *q* is entirely outside the canonical viewing frustum corresponding to the current perspective view. Otherwise it returns VU_ACCEPT and places the clipped line segment endpoints in *nup* and *nuq* (which are undefined for case VU_REJECT). *p* and *q* are not touched.

CvuOrthoClipLine(p, q, nup, nuq)
vector p, q, nup, nuq;

Returns VU_REJECT (defined in **VuSpec.h**) if the line segment from point *p* to point *q* is entirely outside the canonical viewing volume for the current orthographic view. Otherwise it returns VU_ACCEPT and places the clipped line segment endpoints in *nup* and *nuq* (which are undefined for case VU_REJECT). *p* and *q* are not touched.

Appendix 7: The aa_view, aa_display, and aa_vutrix Packages

Following are the help messages for the *aa_view*, *aa_display*, and *aa_vutrix* interfaces (in the *aarg.i* style) to the **CvuSpec** view and display specification package. The source resides in files *viewaarg.h*, *aa_view.c*, *displayaarg.h*, *aa_display.c*, *vutrixaarg.h*, *aa_vutrix.c*.

For *aa_view*:

[-vuinit]	Initialize current view to default
[-vupop]	Pop view stack into current view
[-vupush]	Push current view onto view stack
[-vustkinit]	Initialize view stack
[-vuprint]	Print current view
[-vuwrite %s]	Write current view into given file
[-vuread %s]	Read current view from given file
[-vuverbose [%c]]	Verbose view switch (arg=[TFtf]) [default=F]

[-vupoint %F %F %F]	Set view point (world space)
[-vunorm %F %F %F]	Set view normal (world space)
[-vuup %F %F %F]	Set view up (world space)
[-vudist %F]	Set view distance (world space)
[-nearfar %F %F]	Set near and far distances (far=0 means infinity)
[-vuwin %F %F %F %F]	Set view window: center (u, v), halfsize (u, v)
[-vutype %c]	'p'=PERSPECTIVE, 'o'=ORTHOGRAPHIC
[-fov %F]	Set full horizontal field-of-view angle (degrees)
[-aspect %F [%F]]	Set width-to-height aspect ratio (2 args => w h)
[-view %F %F %F %F]	Set view: distance azimuth pitch roll (degrees)
[-fullview %F%F%F %F %F %F]	Set view: viewpoint azimuth pitch roll (degrees)
[-lookat %F%F%F %F%F%F %F]	Set view: ViewPoint(xyz) LookAtPoint(xyz) roll (degrees)
[-camera %F%F%F %F%F%F %F%F%F %F]	Set view: Reference(xyz) Normal(xyz) Up(xyz) deye
[-perspective %F %F %F %F]	Set view window: fovx aspectratio near far
[-window %F%F%F%F%F%F]	Set view window: left right top bottom near far

For *aa_display*:

[-raster]	Set current display to raster type
[-stroke]	Set current display to calligraphic type
[-disprint]	Print current display
[-screenverbose [%c]]	Verbose screen switch (arg=[TFtf]) [default F]
[-screen %F %F %F %F [%F %F]]	Set current screen window: xmin xmax ymin ymax [zmin zmax]
[-screen2D %F %F %F %F [%F]]	Set current screen window: aspect xmin ymin pixelw pixelh. One of pixelw pixelh must be 0.
[-fullscreen %F %F %F %F [%F%F]]	Set full screen window: xmin xmax ymin ymax [zmin zmax]
[-fullaspect %F [%F]]	Set full screen aspect ratio (2 args => w h)
[-dispnorm %F %F %F]	Set screen normal
[-dispup %F %F %F]	Set screen up

For *aa_vutrix*:

[-A]	Get matrix A (Tech Memo 84)
[-B]	Get matrix B (Tech Memo 84)
[-C]	Get matrix C (Tech Memo 84)
[-D]	Get matrix D (Tech Memo 84)
[-E]	Get matrix E (Tech Memo 84)
[-F]	Get matrix F (Tech Memo 84)
[-G]	Get matrix G (Tech Memo 84)
[-H]	Get matrix H (Tech Memo 84)

[-J]	Get matrix J (Tech Memo 84)
[-K]	Get matrix K (Tech Memo 84)
[-L]	Get matrix L (Tech Memo 84)
[-M]	Get matrix M (Tech Memo 84)
[-N]	Get matrix N (Tech Memo 84)
[-P]	Get matrix P (Tech Memo 84)
[-Q]	Get matrix Q (Tech Memo 84)
[-R]	Get matrix R (Tech Memo 84)
[-S]	Get matrix S (Tech Memo 84)
[-NsubL]	Get matrix NsubL (Tech Memo 84)
[-NsubR]	Get matrix NsubR (Tech Memo 84)
[-NsuboR]	Get matrix NsuboR (Tech Memo 84)
[-Jsubr]	Get matrix Jsubr (Tech Memo 84)
[-Jsubc]	Get matrix Jsubc (Tech Memo 84)
[-Nsubo]	Get matrix Nsubo (Tech Memo 84)
[-Ssubr]	Get matrix Ssubr (Tech Memo 84)
[-Ssubc]	Get matrix Ssubc (Tech Memo 84)

Appendix 8: The mx Matrix Desk Calculator

The program *mx* is basically a driver for the *aa_mx* package described below and also the *aa_matrix*, *aa_view*, *aa_display*, *aa_vutrix* packages. It is intended to exercise the full matrix/view set of packages: **MxMatrix**, **CmxMatrix**, **VxVector**, and **CvuSpec**. Following are the help messages for the *aa_mx* package and the *mx* program. The corresponding source files are *mxaarg.h*, *aa_mx.c*, and *mx.c.* A set of 52 matrix registers and 52 vector registers is assumed, with alphabetic names, upper and lower case. An implied matrix is the current matrix **Cmx** and an implied vector is the current vector **Cvx**.

For *aa_mx*:

[-i [%c]]	Set matrix to identity
[-m %c]	Set matrix to current transform
[-mget [%c]]	Set matrix from keyboard
[-v %c [%F %F %F [%F]]]	Set vector to given value
[-vget [%c]]	Set vector from keyboard
[-len [%c]]	Get 3-D length of normalized vector
[-len4 [%c]]	Get 4-D length of vector
[-normalize [%c]]	Normalize 3-D vector
[-normalize4 [%c]]	Normalize 4-D vector
[-det [%c]]	Get determinant of matrix
[-mxm %c [%c]]	Multiply a matrix times a matrix
[-vxm %c [%c]]	Multiply a vector times a matrix
[-vxv %c [%c]]	Multiply a column vector times a row vector
[-sxm %F [%c]]	Multiply a scalar times a matrix
[-sxv %F [%c]]	Multiply a scalar times a vector

[-dot %c [%c]]	Form dot product of two 3-D vectors
[-dot4 %c [%c]]	Form dot product of two 4-D vectors
[-cross %c [%c]]	Form cross product of two 3-D vectors
[-cross4 %c [%c]]	Form cross product of two 4-D vectors
[-add %c [%c]]	Add a matrix to a matrix
[-sub %c [%c]]	Subtract matrix2 from matrix1
[-div %c [%c]]	Divide matrix by matrix
[-rdiv %c [%c]]	The other divide
[-cp %c [%c]]	Copy matrix to matrix
[-vadd %c [%c]]	Add a vector to a vector
[-vsub %c [%c]]	Subtract vector2 from vector1
[-hdiv [%c]]	Do homogeneous divide on vector
[-xf [%c [%c]]]	Transform vector by a matrix
[-vcp %c [%c]]	Copy vector to vector
[-xpose [%c]]	Transpose matrix
[-neg [%c]]	Negate matrix
[-vneg [%c]]	Negate vector
[-inv [%c]]	Invert matrix
[-mp [%c]]	Print matrix
[-vp [%c]]	Print vector

For *mx*:

[-mxinit]	Initialize world
[-op [%d<1,14>]]	Change output precision [14 digits default]
[-do [%c]]	Do full mapping pNPS (Tech Memo 84)
[-Do [%c]]	Do shortcut mapping pNQRS (Tech Memo 84)
[-line %c [%c]]	Do full mapping pNPS on given line segment
[-q]	Another quit command

plus the *aa_mx*, *aa_matrix*, *aa_view*, *aa_display*, and *aa_vutrix* package commands.