Revolution

Matrix-Vector Library (MTX)

Version 1.00

The contents in this document are highly confidential and should be handled accordingly.

RVL-06-0038-001-B
Released: October 22, 2007

Contents

Code Examples

Equations

Figures

Tables

Revision History

Version	Date Revised	Item	Description
1.00	2006/03/01	-	First release by Nintendo of America Inc.

1 Introduction

The Matrix-Vector library (MTX) is a collection of routines and types for 3D graphics generation on Revolution. It is not a general purpose mathematical matrix library. It allows you to create and manipulate matrices in ways which are suited to interactive computer graphics in video games and most particularly to the Revolution Graphics library (GX) and Revolution hardware.

1.1 This Guide

This guide explains how to program using the Matrix-Vector library (MTX). In the remainder of this section, we discuss the library in general. Chapter 2 describes the matrix types. Chapters 3 to 9 cover each library routine in turn. Chapter 10 covers some traps and pitfalls. Appendix A lists a summary table of all API calls in the library.

1.2 Useful References

This document assumes that you know how to program in the C language, and that you have some knowledge of matrix and vector mathematics. Otherwise, you might want to do some preparatory reading. Here are some useful sources (check your local bookstore or library for the latest editions):

Foley, James D., et al., *Computer Graphics: Principles and Practice, 2nd Ed.*, Addison-Wesley, Reading, MA, 1990.

Kempf, Renate and Chris Frazier (eds.), *OpenGL Reference Manual, 2nd Ed.*, Addison-Wesley, Reading, MA, 1997.

Woo, Mason, et al., *OpenGL Programming Guide, 2nd Ed.*, Addison-Wesley, Reading, MA, 1997.

1.3 Library Design

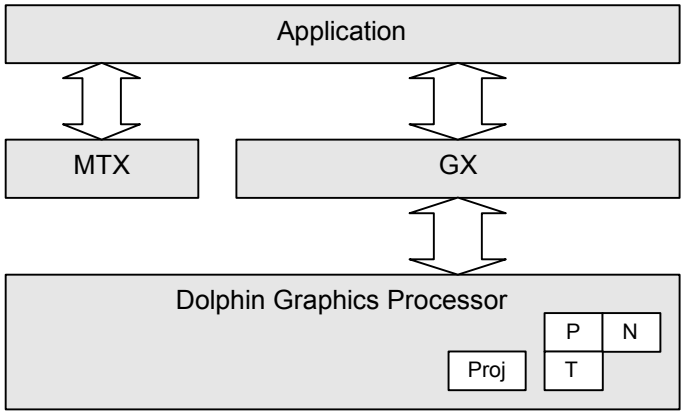
The MTX library is designed to be fairly light-weight and to provide only core functionality, those functions that are essential or extremely useful for most applications on Revolution. It does not provide every matrix type and matrix routine you could possibly need.

- It is designed to be runtime efficient, as far as a general-purpose library for video games can be.
- It is designed to be easy to use; however, ease-of-use is compromised at times to favor of efficiency.
- It is designed to be safe; however, some dangers and pitfalls remain in the interests of efficiency.

1.4 Library Overview

The Matrix-Vector library is used by the application to assist in the generation and animation of 3D matrices for 3D graphics. These matrices can be passed to the Graphics library (GX) as *position* (or modelview) matrices, *normal* matrices, *texture* matrices, and *projection* matrices. GX loads these matrices into the Revolution Graphics Processor matrix register areas (see **P**, **N**, **T**, and **Proj** in Figure 1–1).

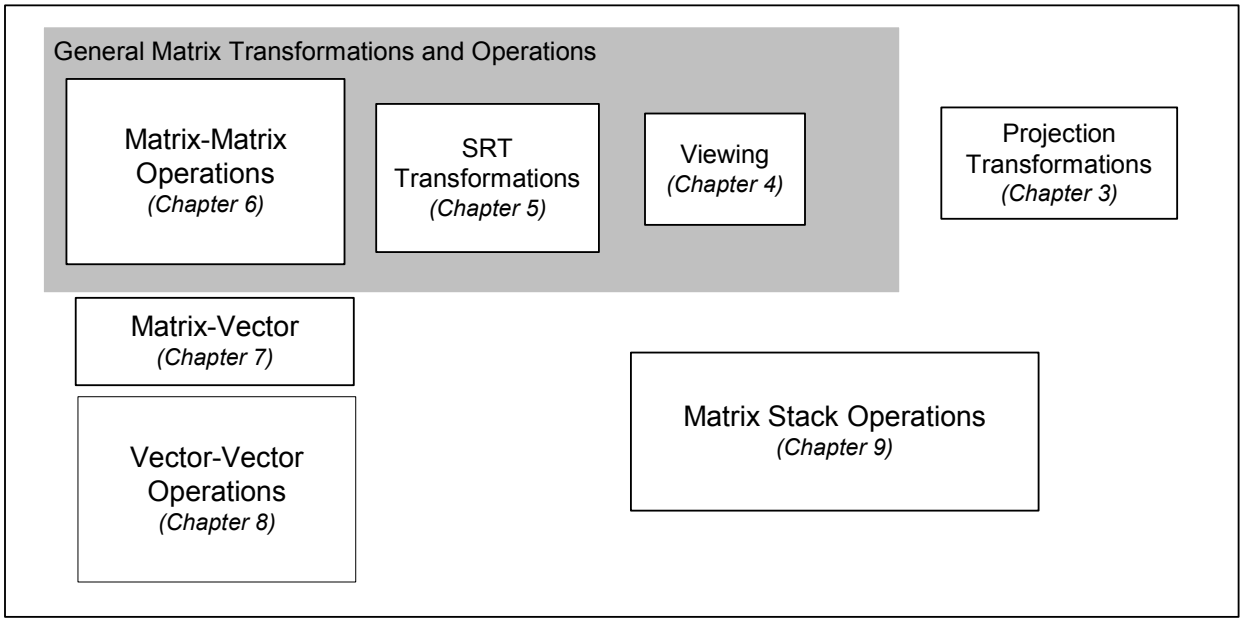
Figure 1–1 Matrix-Vector Library and its Place Within Revolution



Note: The application is not obliged to use MTX to manipulate matrices. However, MTX is designed to match the requirements of GX and is therefore a useful reference point at the very least.

The Matrix-Vector Library contains seven main logical groups of routines. Chapters 3 to 9 of this guide deal with these seven groups in turn.

Figure 1–2 Logical Groups of the Matrix-Vector Library



1.4.1 Projection Transformations

These routines are provided for the generation of perspective and parallel projections, and are suited to the GX library and the Revolution geometry and texture projection hardware. For lighting effects, the library supports geometry projection from 3D (*x, y, z*) to 3D homogeneous (*x, y, z, w*), and texture projection from 3D (*x, y, z*) to 2D homogeneous (*s, t, q*). The geometry projection routines in the library *only* generate (or operate on) 4x4 matrices. In contrast, texture projection routines use 3x4 matrices.

1.4.2 General Matrix Transformations and Operations

These matrix routines are suitable for generation and manipulation of position (modelview) matrices, and normal matrices (and possibly also texture matrices). This set of routines includes three main groups:

1.4.2.1 Viewing Transformations

The generation of a Lookat matrix is supported.

1.4.2.2 Scale, Rotate and Translate (SRT) Transformations

The library provides functions to create matrices for scaling, various rotations, and translation. Also included here is identity-matrix creation, conversion from quaternion to matrix, and conversion between degrees and radians. These transformations are the fundamental modeling operations.

1.4.2.3 Matrix-Matrix Operations

Matrix-matrix operations include copying, concatenation (multiplication), transposition, and inversion. From a mathematical viewpoint, these are the basic operators you can apply to matrices:

Equation 1–1 Basic Matrix Operators

$$A \leftarrow B$$

$$A \leftarrow B \times C$$

$$A \leftarrow B^{T}$$

$$A \leftarrow B^{-1}$$

1.4.3 Matrix-Vector Operations

Multiplication of a matrix by a vector is provided.

1.4.4 Vector-Vector Operations

Although vector support is not the primary focus of the library, some vector-vector and scalar-vector routines are provided, including:

- Vector addition and subtraction.
- Vector scaling.
- Vector magnitude calculation and normalization.
- Vector dot and cross products.
- Vector distance calculation.
- Vector reflection and half-angle generation.

1.4.5 Matrix Stack Operations

Support is also provided for a simple matrix stack type. The user must define the maximum stack depth. Allocating, freeing, initializing, reading, popping, and four flavors of pushing are implemented.

1.5 C Functions and Paired-Single Optimized Functions

Some of the APIs in the library exist in two versions with similar functionality. One version is written in plain C language, while the other has been optimized with paired-single assembler operations to take advantage of a special feature of the Broadway CPU. Where an API has such two versions, the C function is prefixed with "`C_`" in the library (as in `C_MTXConcat`), and the paired-single function is prefixed with "`PS`" (as in `PSMTXConcat`).

Although there are two function prototypes, it is not necessary to concern yourself with which one to call in your application. You can simply use non-prefixed function calls (such as `MTXConcat`). "`C_`" and "`PS`" are defined in build-dependent macros which automatically bind them to the relevant non-prefixed function. In debug builds, the SDK default setting references C functions because of the need for detailed error checks. In release (non-debug) builds, the SDK uses paired-single functions in order to speed up computations.

If this general build rule does not suit your needs, you can call the "`C_`" functions or "`PS`" functions explicitly. Alternatively, you can make the SDK use the C version of non-prefixed function calls exclusively by defining the flag "`MTX_USE_C`" (probably as a debug flag) before including `mtx.h`. Likewise, you can use the flag "`MTX_USE_PS`" to restrict the library to the paired-single versions of functions in the library.

For simplicity, this document uses non-prefixed definitions in its descriptions of functions.

2 The Matrix Types

2.1 The Mtx Type

The focus of MTX is to support common modeling transformations. For this reason the standard matrix type in the library is a 3x4 matrix. This type is named `Mtx`, and is defined thus:

Code 2–1 Mtx Type Definition

```
typedef f32 Mtx[3][4];
Mtx m;
```

There are three major ways to view and to index this type.

First, the mathematical view of a 3x4 matrix *m*:

Equation 2–1 3x4 Matrix *m*

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

From this point of view, all matrices are *really* 4x4 row-major (4 rows by 4 columns): the `Mtx` type has an *implicit* 4th row of (0,0,0,1), and only the upper 12 elements are actually stored in memory.

Because the matrix is *row-major*, we state the *y*-coordinate or dimension first (the row), and the *x*-coordinate or dimension second (the column). This has two consequences. One, the `Mtx` type can be said to represent 3x4 matrices (3 rows by 4 columns). Two, when indexing into the matrix, m_{20} is the bottom left element (*y*=2, *x*=0).

The second viewpoint is that of the Revolution hardware. Because we want the library to be efficient, the hardware dictates how the matrix elements must be stored in memory. The 3x4 matrix type is defined by the Revolution hardware to be *row-major* in memory. This means the first row (row 0) is stored first in memory.

The third viewpoint is that of the C language. In order to reference a 3x4 row-major matrix that is stored row-major in memory, you must declare the type to be `[3][4]`, and index into `m[0..2][0..3]`. So the example element m_{20} is stored in `m[2][0]`. Therefore, when indexing into a matrix in C, you must type the y-coordinate or dimension (the row) first.

These three viewpoints are summarized in the following table to minimize confusion.

Table 2–1 Methods of Viewing and Indexing the Mtx Type

Address	Mathematical	C Arrays
base + 0x00	m_{00}	m[0][0]
base + 0x04	m_{01}	m[0][1]
base + 0x08	m_{02}	m[0][2]
base + 0x0c	m_{03}	m[0][3]
base + 0x10	m_{10}	m[1][0]
base + 0x14	m_{11}	m[1][1]
base + 0x18	m_{12}	m[1][2]
base + 0x1c	m_{13}	m[1][3]
base + 0x20	m_{20}	m[2][0]
base + 0x24	m_{21}	m[2][1]
base + 0x28	m_{22}	m[2][2]
base + 0x2c	m_{23}	m[2][3]

The `Mtx` type is “array of 3 (arrays of 4 (f32s))”, and `sizeof(Mtx)` is 48 bytes. The Matrix-Vector library does not require any alignment other than word-alignment for matrices.

Note: 8-byte or 16-byte alignment may be recommended for maximum performance in the future. See "1.5 C Functions and Paired-Single Optimized Functions" on page 11.

2.2 The MtxPtr Type

Code 2–2 MtxPtr Type Definition

```
typedef f32 Mtx[3][4];
typedef f32 (*MtxPtr)[4];
```

The type MtxPtr behaves as a pointer to Mtx in most respects; however, its type is *really* “pointer to (array of 4 (f32s))”. This has two implications:

First, you must assign an Mtx directly to an MtxPtr without using the address-of operator &. This is demonstrated in the following example:

Code 2–3 Assignment to a MtxPtr

```
#include <revolution.h>

Mtx m;
MtxPtr mp;

void main(void) {
    mp=(MtxPtr)m;
    m[2][0]=1.0;     // references m20 (bottom-left element)
    mp[2][0]=2.0;    // references m20 (bottom-left element)
}
```

Second, you can’t just increment or decrement a MtxPtr, because sizeof(*MtxPtr) is 16; it is *not* 48. Therefore, we provide the constant MTX_PTR_OFFSET, which holds the integer multiplier necessary to increment or decrement a MtxPtr by one matrix. The correct use of this pointer is illustrated below:

Code 2–4 MtxPtr++

```
#include <revolution.h>

Mtx m[10];
MtxPtr mp;

void main(void) {
    mp=(MtxPtr)m;
    mp[2][0]=2.0;         // references first matrix,  m[0][2][0]
    mp+=MTX_PTR_OFFSET;   // Increment over 3 rows
    mp[2][0]=3.0;         // references second matrix, m[1][2][0]
    mp+=MTX_PTR_OFFSET*2; // Increment over 6 rows
    mp[2][0]=4.0;         // references fourth matrix, m[3][2][0]
}
```

2.3 The Mtx44 Type

This Mtx44 type is required for the routines that generate a projection matrix. These are MTXPerspective, MTXFrustum, and MTXOrtho. A Mtx44 cannot be passed to any other MTX routine. The type is defined as shown in Code 2–5.

Code 2–5 Mtx44 Type Definition

```
typedef f32 Mtx44[4][4];
Mtx44 p;
```

This represents a full 4x4 matrix *p*:

Equation 2–2 Full 4x4 Matrix *p*

$$\begin{pmatrix} p_{00} & p_{01} & p_{02} & p_{03} \\ p_{10} & p_{11} & p_{12} & p_{13} \\ p_{20} & p_{21} & p_{22} & p_{23} \\ p_{30} & p_{31} & p_{32} & p_{33} \end{pmatrix}$$

The mathematical indexing and C array indexing of this type are also equivalent:

Equation 2–3 Math and C Array Indexing

$$p_{i,j} = \mathbf{p[i][j]}$$

2.4 The Mtx44Ptr Type

Just as a `MtxPtr` points to a `Mtx`, a `Mtx44Ptr` points to a `Mtx44`. The types `MtxPtr` and `Mtx44Ptr` are alike in all other respects. `MTX44_PTR_OFFSET` is provided as an index multiplier for `Mtx44Ptr` types.

Code 2–6 Mtx44Ptr Type Definition

```
typedef f32 (*Mtx44Ptr)[4];
```

2.5 The MTXRowCol Macro

We provide the macro `MTXRowCol(m, r, c)` in `mtx.h` as a mnemonic to remind you to specify the row index first, then the column index. This macro also insulates the application code from changes in the representation of matrices in memory. It can be used for reading from or writing to individual matrix elements in `Mtx`, `MtxPtr`, `Mtx44` and `Mtx44Ptr` types. It is basically defined as:

Code 2–7 MTXRowCol Macro Definition

```
#define MTXRowCol(m,r,c) m[r][c]
```

This macro can be used in all the places that a reference to a matrix element can:

Code 2–8 Two MTXRowCol Examples

```
#include <revolution.h>

Mtx m;
Mtx44 p;

void main(void) {

    MTXIdentity(m);       // m=I
    m[2][3]=-10;          // Set m to be a translation by -10 along z-axis.
    MTXRowCol(m,2,3)=-10; // The same as the previous line.

    MTXPerspective(p,60,4.0/3,1,10);  // Setup p as perspective projection.
    p[0][0]*=0.5;                     // Double the width of the frustum in p.
    MTXRowCol(p,0,0)*=0.5;            // The same as the previous line.
}
```

3 Projection Transformations

3.1 MTXPerspective

Code 3–1 MTXPerspective

```
void MTXPerspective   ( Mtx44 m, f32 fovy, f32 aspect, f32 n, f32 f );
```

`MTXPerspective` is used to describe the mapping from camera-space coordinates into clip-space. It is designed to be used in tandem with `GXSetProjection`. However, you can also use the resulting matrix to transform points from world-space to screen-space on the CPU.

Unlike other MTX functions, `MTXPerspective`, `MTXFrustum` and `MTXOrtho` all operate on 4x4 matrices.

`MTXPerspective` will generate 3D perspective transformations, encapsulating some of the properties of a camera, including field-of-view angle and aspect ratio. It also controls the near and far plane positions. It is the most suitable function for creating general purpose 3D-viewing projection matrices.

The argument *fovy* specifies the desired field-of-view from screen top to screen bottom, in degrees. This argument cannot be 0° or 180°.

The argument *aspect* specifies the ratio between screen width and height (width/height). It must not be zero.

The arguments *n* and *f* specify the position of the near and far planes respectively. `MTXPerspective` requires that *n* and *f* are not equal. Normally, for most projections, *n* will be less than (or much less than) *f*, and both *n* and *f* will be positive. (Even though the planes are typically thought of as being positioned down the negative *z*-axis, the values specify positive distances from the eye-point at the origin.)

`MTXPerspective` assigns the following to the user-allocated 4x4 matrix *m*:

Equation 3–1 Matrix Assigned by MTXPerspective

$$\begin{pmatrix} 1/(\tan(fovy/2)*aspect) & 0 & 0 & 0 \\ 0 & 1/\tan(fovy/2) & 0 & 0 \\ 0 & 0 & -(n)/(f-n) & -(f*n)/(f-n) \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

The following example sets up and loads a perspective projection matrix with a field-of-view of 70° and near and far planes at 50 units and 1000 units, respectively:

Code 3–2 Typical Perspective Projection Matrix Using MTXPerspective

```
#include <revolution.h>
:
Mtx44 m;

MTXPerspective(m, 70.0, 640.0/480.0, 50.0, 1000.0);
GXSetProjection(m, GX_PERSPECTIVE);
```

Note: `GXSetProjection` *must* be passed the `GX_PERSPECTIVE` flag if the projection matrix has the perspective projection form (shown in Code 3–2).

3.2 MTXFrustum

Code 3–3 MTXFrustum

```
void MTXFrustum (
        Mtx44 m,
        f32 t, f32 b,
        f32 l, f32 r,
        f32 n, f32 f
);
```

`MTXFrustum` is used to describe the mapping from camera-space coordinates into clip-space. It is designed to be used in tandem with `GXSetProjection`. However, you can also use the resulting matrix to transform points from world-space to screen-space on the CPU.

Unlike other MTX functions, `MTXPerspective`, `MTXFrustum`, and `MTXOrtho` all operate on 4x4 matrices.

`MTXFrustum` will generate 3D perspective transformations, encapsulating some of the properties of a camera, including field-of-view angle and aspect ratio. It also controls the near and far plane positions.

`MTXFrustum` is more flexible, but less commonly used than its cousin `MTXPerspective`. It is more powerful because it allows you to specify off-center projections. These projections are useful in some applications, for example rendering a high-resolution image by rendering *nxm* sub-rectangles, one at a time.

The arguments *n* and *f* (near and far) specify the position of the near and far planes respectively. Normally, for most projections, *n* will be less than (or much less than) *f*, and both *n* and *f* will be positive. (Even though the planes are typically thought of as being positioned down the negative *z*-axis, the values specify positive distances from the eye-point at the origin.)

The arguments *l* and *r* (left and right), and *t* and *b* (top and bottom) specify the location in camera-space of the front four corners of the viewing frustum. Normally, *l* will be less than *r*, and *b* will be less than *t*. Reversing one pair allows you to redefine the coordinate axes of object space (that is, you can change from a right-hand to a left-hand coordinate system).

`MTXFrustum` requires that *n* and *f* are not equal, that *l* and *r* are not equal, and that *t* and *b* are not equal, so that the resulting matrix is well defined.

`MTXFrustum` generates a projection perspective transformation matrix such that the viewing frustum bounded by the 8 (camera-space) points:

(l, t, -n, 1)	*(l*(f/n), t*(f/n), -f, 1)*
(r, t, -n, 1)	*(r*(f/n), t*(f/n), -f, 1)*
(l, b, -n, 1)	*(l*(f/n), b*(f/n), -f, 1)*
(r, b, -n, 1)	*(r*(f/n), b*(f/n), -f, 1)*

will be mapped to the 8 (clip-space) points:

(-n, n, -n, n)	*(-f, f, 0, f)*
(n, n, -n, n)	*(f, f, 0, f)*
(-n, -n, -n, n)	*(-f, -f, 0, f)*
(n, -n, -n, n)	*(f, -f, 0, f)*

Once these 8 homogeneous coordinates are divided through by *w* (the fourth element), they will be simply the 8 corners of a 2x2x2 cube centered on the origin; that is, the 8 points: (±1.0, ±1.0, ±1.0).

Therefore, `MTXFrustum` assigns the following to the user-allocated 4x4 matrix *m*:

Equation 3–2 Matrix Assigned by MTXFrustum

$$\begin{pmatrix} 2n/(r-l) & 0 & (r+l)/(r-l) & 0 \\ 0 & 2n/(t-b) & (t+b)/(t-b) & 0 \\ 0 & 0 & -(n)/(f-n) & -(f*n)/(f-n) \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

The example shown in Code 3–4 sets up and loads a perspective projection matrix that will only display the top left quarter of the screen, but will scale this quarter up to fill the viewable area:

Code 3–4 Off-Center Projection with MTXFrustum

```
#include <revolution.h>
:
Mtx44 m;

MTXFrustum(m, 480.0, 0.0, -640.0, 0.0, 20.0, 600.0 );  // Just draw TopLeft Quadrant
GXSetProjection(m, GX_PERSPECTIVE);
```

Note: `GXSetProjection` *must* be passed the `GX_PERSPECTIVE` flag if the projection matrix has the perspective projection form (shown in Code 3–4).

3.3 MTXOrtho

Code 3–5 MTXOrtho

```
void MTXOrtho   ( Mtx44 m, f32 t, f32 b, f32 l, f32 r, f32 n, f32 f );
```

`MTXOrtho` is used to describe the mapping from camera-space coordinates into clip-space. It is designed to be used in tandem with `GXSetProjection`.

Unlike other MTX functions, `MTXPerspective`, `MTXFrustum`, and `MTXOrtho` all operate on 4x4 matrices.

`MTXOrtho` will generate 2D non-perspective (or parallel) projection transformations, simulating a camera with an infinite zoom positioned at infinity. It is the most suitable function for creating general purpose 2D-viewing projection matrices.

The arguments *n* and *f* specify the position of the near and far planes respectively. `MTXOrtho` requires that *n* and *f* are not equal. Normally, for most projections, *n* will be less than *f*, and both *n* and *f* will be positive.

The arguments *t* and *b* (top and bottom), *l* and *r* (left and right), and *n* and *f* together specify the location in camera-space of the 8 corners of the viewing frustum. In the case of `MTXOrtho`, this is a cuboid. Normally *l* will be less than *r*, and *b* will be less than *t*.

Therefore, `MTXOrtho` assigns the following to the user-allocated 4x4 matrix *m*:

Equation 3–3 Matrix Assigned by MTXOrtho

$$\begin{pmatrix} 2/(r-l) & 0 & 0 & -(r+l)/(r-l) \\ 0 & 2/(t-b) & 0 & -(t+b)/(t-b) \\ 0 & 0 & -1/(f-n) & -(f)/(f-n) \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The example shown in Code 3–6 sets up and loads a simple orthogonal projection matrix that will map one unit of camera space (in *x* and *y*) onto one pixel of screen space (if used with the correct viewport):

Code 3–6 Pixel-Unit Projection with MTXOrtho

```
#include <revolution.h>
:
Mtx44 m;

MTXOrtho(m, 480.0,0.0, 0.0,640.0,  0.0,255.0);    // Top=480 Bot=0  Left=0 Right=640
GXSetProjection(m, GX_ORTHO);
```

Note: `GXSetProjection` *must* be passed the `GX_ORTHO` flag if the projection matrix has the orthogonal or parallel projection form (shown in Code 3–6).

3.4 MTXLightPerspective

Code 3–7 MTXLightPerspective

```
void MTXLightPerspective(
    Mtx m,
    f32 fovY, f32 aspect,
    f32 scaleS, f32 scaleT,
    f32 transS, f32 transT );
```

`MTXLightPerspective` supports projected texture techniques. It generates a projection transformation that specifies a mapping from light-space (that is, model space for the light) to texture-space (*s*, *t*, *q*). The mapping encapsulates some of the properties of a projective texture light, including the position of the four bounding planes and the placement of the texture, but not including the location and orientation of the light in space. For more details, refer to "3 Texture Projection" in the *Graphics Library (Advanced Rendering)* manual.

Texture projection is in some ways analogous to geometry projection. In both cases, points in a 3D space are projected along rays, to intersect with a 2D plane. However, unlike geometry projection, texture projection does not involve hardware clipping. Also, unlike `MTXPerspective`, `MTXLightPerspective` generates a (smaller) 3x4 projection transformation matrix. A 4x4 matrix is not necessary, because the projection is from a 3D homogeneous space (*x,y,z,*1) into a 2D homogeneous space (*s, t, q*). A 4x4 matrix would also be inconvenient, because MTX contains no calls to manipulate 4x4 matrices.

Under normal usage, the resulting 3x4 projection matrix would be combined with other 3x4 matrices, and then passed to `GXLoadTexMtxImm` or `GXLoadTexMtxIndx`. The common case is to pass *TextureProjection-Matrix * LightModelMatrix*.

The argument *fovY* specifies the height (or *t* extent) of the projected light. The projected texture will form a rectangle (or square) if projected onto a perpendicular surface, and *aspect* controls the width to height ratio of this rectangle. For example, a value of 2.0 would result in the texture being scaled up by two in the horizontal (*s*) direction.

The arguments *scaleS*, *scaleT*, *transS*, and *transT* can be used to move and scale the texture. The normal *scale* and *trans* values will be 0.5, 0.5, 0.5, and 0.5. These values will cause the center of the texture (s=0.5, t=0.5) to be projected along the light direction and are therefore suitable for a light texture that is clamped, but not mirrored in *s* or *t*.

`MTXLightPerspective` assigns the following to the user-allocated 3x4 matrix *m*:

Equation 3–4 Matrix Assigned by MTXLightPerspective

$$\begin{pmatrix} scaleS/(\tan(fovy/2)*aspect) & 0 & -transS & 0 \\ 0 & scaleT/\tan(fovy/2) & -transT & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Code 3–8 demonstrates a rotating light, projected 10° tall and 10° wide, positioned at the origin:

Code 3–8 Using MTXLightPerspective

```
{
    Mtx txproj;
    Mtx lightmodelview;
    Mtx txmtx;

    /* Setup light modelview */
    MTXRotDeg(lightmodelview,'x',lightdegrees);

    MTXLightPerspective(txproj, 10.0, 1.0, 0.5, 0.5, 0.5, 0.5);

    MTXConcat(txproj, lightmodelview, txmtx);

    GXLoadTexMtxImm(txmtx, GX_TEXMTX0, GX_MAT_3x4);
}
```

3.5 MTXLightFrustum

Code 3–9 MTXLightFrustum

```
void MTXLightFrustum   (
    Mtx m,
    f32 t, f32 b,
    f32 l, f32 r,
    f32 n,
    f32 scaleS, f32 scaleT,
    f32 transS, f32 transT );
```

`MTXLightFrustum` supports projected texture techniques. It generates a projection transformation that specifies a mapping from light-space (that is, model space for the light) to texture-space (*s, t, q*). The mapping encapsulates some of the properties of a projective texture light, including the position of the four bounding planes and the placement of the texture, but not including the location and orientation of the light in space. For more details, refer to "3 Texture Projection" in the *Graphics Library (Advanced Rendering)* manual.

Texture projection is in some ways analogous to geometry projection. In both cases, points in a 3D space are projected along rays to intersect with a 2D plane. However, unlike geometry projection, texture projection does not involve hardware clipping. Also, unlike `MTXFrustum`, `MTXLightFrustum` generates a (smaller) 3x4 projection transformation matrix. A 4x4 matrix is not necessary, because the projection is from a 3D homogeneous space (*x,y,z,*1) into a 2D homogeneous space (*s, t, q*). A 4x4 matrix would also be inconvenient, because MTX contains no calls to manipulate 4x4 matrices.

Under normal usage, the resulting 3x4 projection matrix would be combined with other 3x4 matrices, and then passed to `GXLoadTexMatrixImm` or `GXLoadTexMatrixIndx`. The common case is to pass *TextureProjectionMatrix * LightModelMatrix*.

With normal *scale* and *trans* values, the top, bottom, left, and right planes (arguments *t*, *b*, *l*, *r*) will correspond to the clamp/wrap borders of the texture, that is, the lines *t*=1, *t*=0, *s*=0, *s*=1, respectively. No far plane is specified, and the near plane distance (*n*) is only used to specify or scale the 4 side planes. Therefore, doubling *t*, *b*, *l*, *r*, and *n* will have no effect on the resulting matrix.

The arguments *scaleS*, *scaleT*, *transS*, and *transT* can be used to move and scale the texture. The normal *scale* and *trans* values will be 0.5, 0.5, 0.5, and 0.5. These values will cause the top, bottom, left, and right planes to be mapped to *t*=1, *t*=0, *s*=0, *s*=1 respectively and are therefore suitable for a light texture which is clamped, but not mirrored in *s* or *t*.

With unit scale and zero translate, the four planes would be mapped to *s*=±1, *t*=±1.

`MTXLightFrustum` assigns the following to the user-allocated 3x4 matrix *m*:

Equation 3–5 Matrix Assigned by MTXLightFrustum

$$\begin{pmatrix} 2*scaleS*n/(r-l) & 0 & scaleS*(r+l)/(r-l)-transS & 0 \\ 0 & 2*scaleT*n/(t-b) & scaleT*(t+b)/(t-b)-transT & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

The following demonstrates a rotating light at the origin:

Code 3–10 Using MTXLightFrustum

```
{
    Mtx txproj;
    Mtx lightmodelview;
    Mtx txmtx;

    /* Setup light modelview */
    MTXRotDeg(lightmodelview,'x',lightdegrees);

    MTXLightFrustum(txproj, ymin, ymax, xmin, xmax, nnear,
                    0.5, 0.5, 0.5, 0.5);

    MTXConcat(txproj, lightmodelview, txmtx);

    GXLoadTexMtxImm(txmtx, GX_TEXMTX0, GX_MAT_3x4);
}
```

3.6 MTXLightOrtho

Code 3–11 MTXLightOrtho

```
void MTXLightOrtho(
    Mtx m,
    f32 t, f32 b,
    f32 l, f32 r,
    f32 scaleS, f32 scaleT,
    f32 transS, f32 transT );
```

`MTXLightOrtho` supports non-perspective projected texture techniques. It generates a parallel projection transformation that specifies a mapping from light-space (that is, model space for the light) to texture-space (*s,t,*1). The mapping encapsulates some of the properties of a projective texture light, including the position of the four bounding planes and the placement of the texture, but not including the location and orientation of the light in space. For more details, refer to "3 Texture Projection" in the *Graphics Library (Advanced Rendering)* manual.

Texture projection is in some ways analogous to geometry projection. In both cases, points in a 3D space are projected along rays, to intersect with a 2D plane. However, unlike geometry projection, texture projection does not involve hardware clipping. Also, unlike `MTXOrtho`, `MTXLightOrtho` generates a (smaller) 3x4 projection transformation matrix. A 4x4 matrix is not necessary, because the projection is from a 3D homogeneous space (*x,y,z,*1) into a 2D homogeneous space (*s,t,*1). A 4x4 matrix would also be inconvenient, because MTX contains no calls to manipulate 4x4 matrices. In fact, with `MTXOrtho`, only a 2x4 is necessary, because (*q*=1) for all points in space. However, a 2x4 would also be inconvenient — unless a 2x4 matrix library was implemented as well, but this would increase code space. Worse yet, it would increase the instruction cache working set, in return for a fairly small functional gain.

With normal *scale* and *trans* values, the top, bottom, left, and right planes (arguments *t*, *b*, *l*, *r*) will correspond to the clamp/wrap borders of the texture, that is, the lines *t*=1, *t*=0, *s*=0, *s*=1, respectively. No far plane or near plane is specified.

Under normal usage, the resulting 3x4 projection matrix would be combined with other 3x4 matrices and then passed to `GXLoadTexMtxImm` or `GXLoadTexMtxIndx`. The common case is to pass *TextureProjectionMatrix * LightModelMatrix*.

The arguments *scaleS*, *scaleT*, *transS*, and *transT* can be used to move and scale the texture. The normal scale and translate values will be 0.5, 0.5, 0.5, and 0.5. These values will cause the center of the texture (*s*=0.5, *t*=0.5) to be projected along the light direction and are therefore suitable for a light texture that is clamped, but not mirrored in *s* or *t*.

`MTXLightOrtho` assigns the following to the user-allocated 3x4 matrix *m*:

Equation 3–6 Matrix Assigned by MTXLightOrtho

$$\begin{pmatrix} 2*scaleS/(r-l) & 0 & -scaleS*(r+l)/(r-l)+transS & 0 \\ 0 & 2*scaleT/(t-b) & -scaleT*(t+b)/(t-b)+transT & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The following demonstrates a light at infinity, directed through the origin, and rotating about the origin and the x-axis. The light is slightly taller than it is wide.

Code 3–12 Using MTXLightOrtho

```
{
    Mtx txproj;
    Mtx lightmodelview;
    Mtx txmtx;

    /* Setup light modelview */
    MTXRotDeg(lightmodelview,'x',lightdegrees);

    MTXLightOrtho(txproj, 10.0,-10.0, -8.0,8.0, 0.5, 0.5, 0.5, 0.5);

    MTXConcat(txproj, lightmodelview, txmtx);

    GXLoadTexMtxImm(txmtx, GX_TEXMTX0, GX_MAT_3x4);
}
```

4 Viewing Transformations

4.1 MTXLookAt

Code 4–1 MTXLookAt

```
void MTXLookAt   ( Mtx m, Point3dPtr camPos, VecPtr camUp, Point3dPtr target );
```

`MTXLookAt` generates a modelview rotation-and-translation matrix such that an object centered at *target* will be central on the screen and will be rotated as if seen from a camera at *camPos* and as if the camera up direction is along *camUp*. The matrix is assigned to the user-allocated 3x4 matrix *m*.

The arguments *camPos*, *camUp*, and *target*, can all be considered to be in world-coordinates. `MTXLookAt` requires that *target* is not equal to *camPos*, that *camUp* has non-zero length, and that *camUp* does not point exactly along the *camPos—target* line.

`MTXLookAt` first computes three direction vectors:

Equation 4–1 MTXLookAt Direction Vectors

$$\begin{aligned} \text{look} &= \text{camPos} - \text{target} \\ \text{right} &= \text{camUp} \times \text{look} \\ \text{up} &= \text{look} \times \text{right} \end{aligned}$$

These three direction vectors are then normalized to unit-length, and `MTXLookAt` assigns them to *m* as shown in Equasion 4–2.

Equation 4–2 Matrix Assigned by MTXLookAt

$$\begin{pmatrix} right_x & right_y & right_z & -(\text{right} \cdot \text{camPos}) \\ up_x & up_y & up_z & -(\text{up} \cdot \text{camPos}) \\ look_x & look_y & look_z & -(\text{look} \cdot \text{camPos}) \end{pmatrix}$$

`MTXLookAt` performs two divides, two square-roots, 19 add/subtracts, and 33 multiplications. All calculations are single-precision.

5 Scale, Rotate and Translate Transformations

5.1 MTXIdentity

Code 5–1 MTXIdentity

```
void MTXIdentity      (Mtx m);
```

`MTXIdentity` assigns the following to the user-allocated 3x4 matrix *m*:

Equation 5–1 Matrix Assigned by MTXIdentity

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

5.2 MTXScale

Code 5–2 MTXScale

```
void MTXScale  (Mtx m, f32 x, f32 y, f32 z);
```

`MTXScale` assigns the following to the user-allocated 3x4 matrix *m*:

Equation 5–2 Matrix Assigned by MTXScale

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \end{pmatrix}$$

This matrix represents a transformation which scales by a factor of *x* in the *x*-axis, *y* in the *y*-axis, and *z* in the *z*-axis.

If all three arguments are equal, *x*=*y*=*z*, the resulting scaling transformation will be isotropic (or proportional). Otherwise, the resulting scaling transformation will be anisotropic (or non-proportional). If you use any anisotropic scale factors in modelview matrices or modelview matrix stacks, *and* you use lighting (Graphics Processor vertex-normal-based lighting calculations), you will probably need to calculate and load inverse-transposes of modelview matrices.

5.3 MTXRotRad, MTXRotDeg

Code 5–3 MTXRotRad and MTXRotDeg

```
void MTXRotRad (Mtx m, u8 axis, f32 rad);

#define MTXRotDeg(m, axis, deg) MTXRotRad(m, axis, MTXDegToRad(deg))
```

`MTXRotRad` generates a rotation about one of three major axes: the positive *x*-axis, *y*-axis, or *z*-axis. The rotation is assigned to the user-allocated 3x4 matrix *m*.

The argument *axis* indicates the major axis and should be a lower or upper-case character “*x*,” “*y*,” or “*z*.”

The argument *rad* specifies in radians the counter-clockwise rotation about the major axis.

`MTXRotRad` provides the result to the system functions `sinf()` and `cosf()` from `math.h` (this math library is provided by Metrowerks). The resulting values are then inserted into *m* as follows:

axis = "*x*": rotation about positive *x*-axis (1,0,0):

Equation 5–3 Rotation About Positive X-Axis (1,0,0)m

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos f(deg) & -\sin f(deg) & 0 \\ 0 & \sin f(deg) & \cos f(deg) & 0 \end{pmatrix}$$

axis = "*y*": rotation about positive *y*-axis (0,1,0):

Equation 5–4 Rotation About Positive Y-Axis (0,1,0)m

$$\begin{pmatrix} \cos f(deg) & 0 & -\sin f(deg) & 0 \\ 0 & 1 & 0 & 0 \\ \sin f(deg) & 0 & \cos f(deg) & 0 \end{pmatrix}$$

axis = "*z*": rotation about positive *z*-axis (0,0,1):

Equation 5–5 Rotation About Positive Z-Axis (0,0,1)m

$$\begin{pmatrix} \cos f(deg) & -\sin f(deg) & 0 & 0 \\ \sin f(deg) & \cos f(deg) & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

There is a convenience macro, `MTXRotDeg`, that specifies rotation angles in degrees.

`MTXRotRad` is a convenience wrapper for `MTXRotTrig` (see section 5.4).

5.4 MTXRotTrig

Code 5–4 MTXRotTrig

```
void MTXRotTrig        (Mtx m, u8 axis, f32 sinA, f32 cosA);
```

MTXRotTrig generates a rotation matrix about one of three axes and assigns this matrix to *m* in precisely the same manner as MTXRotRad (see "5.3 MTXRotRad, MTXRotDeg" on page 25). In this case, however, the user must compute the sine and cosine and convert to radians. Therefore, the following two MTX calls are exactly equivalent:

Code 5–5 Comparison of MTXRotDeg and MTXRotTrig

```
#include <revolution.h>
#include <math.h>

    Mtx m;

    MTXRotDeg(m,'x',45.0);      // 45 degrees
    MTXRotTrig(
        m,
        'x',
        sin(MTXDegToRad(45.0)),
        cos(MTXDegToRad(45.0)));
```

Although MTXRotRad is probably the most convenient, using MTXRotTrig will be more efficient in some circumstances. For example, you could pre-compute a table of sin() values and re-use those values multiple times, or you could use a less precise version of the sin() and cos() functions.

Note: The arguments *sinA* and *cosA* should conform with the following formula.

Equation 5–6 MTXRotTrig Argument Logic

$$\sin A2 + \cos A2 = 1$$

Otherwise, the resulting matrix *m* will also perform an anisotropic scale.

5.5 MTXRotAxisRad, MTXRotAxisDeg

Code 5–6 MTXRotAxisRad and MTXRotAxisDeg

```
void MTXRotAxisRad     (Mtx m, VecPtr axis, f32 rad);

#define MTXRotAxisDeg(m, axis, deg) MTXRotAxisRad(m, axis, MTXDegToRad(deg))
```

MTXRotAxisRad generates a rotation about the user-provided axis. The rotation is assigned to the user-allocated 3x4 matrix *m*.

The vector argument *axis* must be non-zero in length. It need not be unit-length.

The argument *rad* (in radians) is provided to the system functions sinf() and cosf() (from math.h).

MTXRotAxisDeg is a convenience macro to specify rotation angles in degrees.

`MTXRotAxisRad` is used to perform the following computations.

Equation 5–7 MTXRotAxisRad Computations

$$u = \text{axis}/|\text{axis}|$$
$$c = \cos f(deg)$$
$$s = \sin f(deg)$$
$$t = 1 - \cos f(deg)$$

Then `MTXRotAxisRad` assigns the following to *m*:

Equation 5–8 Matrix Assigned by MTXRotAxisRad

$$\begin{pmatrix} tu_x^2 + c & tu_xu_y - su_z & tu_xu_z + su_y & 0 \\ tu_yu_x + su_z & tu_y^2 + c & tu_yu_z - su_x & 0 \\ tu_zu_x - su_y & tu_zu_y + su_x & tu_z^2 + c & 0 \end{pmatrix}$$

For example, Code 5–7 shows an expensive way to compute a simple matrix:

Code 5–7 Rotation About the (*x*=*y*=*z*) Axis

```
#include <revolution.h>
    :
    Mtx m;
    Vec axis={1.0,1.0,1.0};

    MTXRotAxisDeg(m,axis,120.0);
```

`MTXRotAxisRad` performs one `sinf()`, one `cosf()`, one `sqrtf()`, one divide, and 30 multiplies. All calculations are single-precision floating point.

5.6 MTXTrans

Code 5–8 MTXTrans

```
void MTXTrans          (Mtx m, f32 x, f32 y, f32 z);
```

`MTXTrans` generates a translation matrix from *x*, *y*, and *z*.

It assigns the following to the user-allocated 3x4 matrix *m*:

Equation 5–9 Matrix Assigned by MTXTrans

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \end{pmatrix}$$

5.7 MTXQuat

Code 5–9 MTXQuat

```
void MTXQuat          (Mtx m, QuaternionPtr q);
```

`MTXQuat` converts the quaternion *q* (which can be seen to represent a rotation in R^3) into the corresponding rotation matrix. Quaternions are the best way to represent rotations which require interpolation.

The argument *q* must have non-zero length. It need not be unit-length.

If *q* is:

$$\begin{pmatrix} q_1 & q_2 & q_3 & q_4 \end{pmatrix}$$

`MTXQuat` calculates:

Equation 5–10 MTXQuat Computation

$$s = 2/(q_1^2 + q_2^2 + q_3^2 + q_4^2)$$

`MTXQuat` then assigns the following to the user-allocated 3x4 matrix *m*:

Equation 5–11 Matrix Assigned by MTXQuat

$$\begin{pmatrix} 1 - s(q_2^2 + q_3^2) & sq_1q_2 - sq_4q_3 & sq_1q_3 + sq_4q_2 & 0 \\ sq_1q_2 + sq_4q_3 & 1 - s(q_1^2 + q_3^2) & sq_2q_3 - sq_4q_1 & 0 \\ sq_1q_3 - sq_4q_2 & sq_2q_3 + sq_4q_1 & 1 - s(q_1^2 + q_2^2) & 0 \end{pmatrix}$$

`MTXQuat` performs one divide and 16 multiplies. All calculations are single-precision.

5.8 MTXDegToRad, MTXRadToDeg

Code 5–10 MTXDegToRad and MTXRadToDeg

```
f32 MTXDegToRad       (f32 degrees);
f32 MTXRadToDeg       (f32 radians);
```

`MTXDegToRad` converts a single-precision floating point (f32) value from degrees to radians. `MTXRadToDeg` converts a single-precision floating point (f32) value from radians to degrees. Neither is fully precise for 64-bit double-precision floating point (f64) computations.

These functions are actually implemented as macros.

6 Matrix-matrix Operations

6.1 MTXConcat

Code 6–1 MTXConcat

```
void MTXConcat          (Mtx a, Mtx b, Mtx axb);
```

`MTXConcat` concatenates (or multiplies) two 3x4 matrices *a* and *b* to produce the result *a·b* and assigns this result to the user-allocated 3x4 matrix *axb*. Any of the arguments *a*, *b*, and *axb* can reference the same matrix.

If you ask, any mathematician will explain that you can't multiply two 3x4 matrices together. Fortunately, you can, and here's how. If *a* and *b* are respectively:

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \end{pmatrix} \qquad \begin{pmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \end{pmatrix}$$

Then *axb* is:

$$\begin{pmatrix} a_{00}b_{00}+a_{01}b_{10}+a_{02}b_{20} & a_{00}b_{01}+a_{01}b_{11}+a_{02}b_{21} & a_{00}b_{02}+a_{01}b_{12}+a_{02}b_{22} & a_{00}b_{03}+a_{01}b_{13}+a_{02}b_{23}+a_{03} \\ a_{10}b_{00}+a_{11}b_{10}+a_{12}b_{20} & a_{10}b_{01}+a_{11}b_{11}+a_{12}b_{21} & a_{10}b_{02}+a_{11}b_{12}+a_{12}b_{22} & a_{10}b_{03}+a_{11}b_{13}+a_{12}b_{23}+a_{13} \\ a_{20}b_{00}+a_{21}b_{10}+a_{22}b_{20} & a_{20}b_{01}+a_{21}b_{11}+a_{22}b_{21} & a_{20}b_{02}+a_{21}b_{12}+a_{22}b_{22} & a_{20}b_{03}+a_{21}b_{13}+a_{22}b_{23}+a_{23} \end{pmatrix}$$

This is mathematically correct, if you recall that every 3x4 matrix has an (implicit) 4th row of (0,0,0,1).

Code 6–2 calculates the square of the matrix *m*:

Code 6–2 MTXConcat Example

```
#include <revolution.h>
:
MTXConcat(m,m,m);      // m*m->m
```

`MTXConcat` performs 36 multiplies. All calculations are single-precision.

6.2 MTXCopy

Code 6–3 MTXCopy

```
void MTXCopy            (Mtx src, Mtx dst);
```

`MTXCopy` copies the 3x4 matrix *src* into the user-allocated 3x4 matrix *dst*. The previous contents of *dst* are lost. The arguments *src* and *dst* can reference the same 3x4 matrix with no ill effects.

6.3 MTXTranspose

Code 6–4 MTXTranspose

```
void MTXTranspose      (Mtx src, Mtx xPose);
```

`MTXTranspose` computes the transpose of the 3x4 matrix *src* and assigns the result to the user-allocated 3x4 matrix *xPose*. The arguments *src* and *xPose* can reference the same 3x4 matrix with no ill effects.

Because *src* and *xPose* are only 3x4 (not 4x4), the 4th column of *src* (the translation part) is lost, and the resulting 4th column of *xPose* (the translation part) is zero:

Equation 6–1 MTXTranspose Computations

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \end{pmatrix}^{T} = \begin{pmatrix} a_{00} & a_{10} & a_{20} & 0 \\ a_{01} & a_{11} & a_{21} & 0 \\ a_{02} & a_{12} & a_{22} & 0 \end{pmatrix}$$

However, the most common reason you need to compute a transpose is to calculate the inverse-transpose of a modelview matrix, in order to correctly transform vertex normals and perform (CPU or GX) lighting. The translation part of the matrix is not relevant in this case.

6.4 MTXInverse

Code 6–5 MTXInverse

```
u32 MTXInverse      ( Mtx src, Mtx inv );
```

If the 3x4 matrix *src* has an inverse, `MTXInverse` computes that inverse, assigns the result to the user-allocated 3x4 matrix *inv* and returns 1, meaning success. Otherwise, if *src* is singular (non-invertible), `MTXInverse` will return 0, meaning failure, and *inv* will be unaffected.

The arguments *src* and *inv* may reference the same matrix.

Matrix inverse operations on 4x4 matrices with a 4th row of (0,0,0,1) yield a 4x4 matrix which still has a 4th row of (0,0,0,1). Therefore, no information is “lost” when using `MTXInverse` (in contrast to `MTXTranspose`).

This implementation of matrix inverse assumes *src* and *inv* both have an implicit 4th row of (0,0,0,1). It requires one divide and 48 multiplies. All calculations are single-precision.

The typical application of `MTXInverse` is in computing inverse-transpose matrices for lighting:

Code 6–6 Generating Normal Matrices Using MTXInverse and MTXTranspose

```
#include <revolution.h>
:

Mtx mv;                                 // Modelview
Mtx mvIT;                               // Inverse-Tranpose of modelview

  :

  generate_modelview(mv);

  MTXInverse(mv,mvIT);
  MTXTranspose(mvIT,mvIT);
  GXLoadPosMatrixImm(&mv, GX_PNMTX0);    // Load (modelview) into Graphics Processor
  GXLoadNormMatrixImm(&mvIT, GX_PNMTX0); // Load (modelview)-1T

  enable_lighting();
  draw_model();
```

Note: It is not necessary to use `MTXInverse` and `MTXTranspose` to compute the inverse-transpose of the modelview matrix, if that matrix is a concatenation of *only* proportional (isotropic) scales, rotations, and translations. In this case, the modelview matrix will function as its own inverse-transpose, and can be passed to the Graphics Processor twice—once as the modelview matrix and once as the Normal matrix.

In contrast, if you do use significantly non-proportional (anisotropic) scales, or if you use other non-length preserving transformations (such as shears) in your modelview matrix, you will need to use `MTXInverse` and `MTXTranspose` to support lighting.

7 Matrix-vector Operations

7.1 MTXMultVec

Code 7–1 MTXMultVec

```
void MTXMultVec        ( Mtx m, VecPtr src, VecPtr dst );
```

`MTXMultVec` multiplies the 3x4 matrix *m* by the 3-element vector *src*, and assigns the result to the user-allocated 3-element vector *dst*. The vectors *src* and *dst* are assumed to have an implicit fourth element (*w*) of 1, and the matrix *m* is assumed to have an implicit fourth row of (0,0,0,1). For this reason, `MTXMultVec` is not suitable for projecting points through a perspective projection matrix on the CPU.

The arguments *src* and *dst* may reference the same vector.

`MTXMultVec` computes:

Equation 7–1 MTXMultVec Computations

$$dst_x = src_x m_{00} + src_y m_{01} + src_z m_{02} + m_{03}$$

$$dst_y = src_x m_{10} + src_y m_{11} + src_z m_{12} + m_{13}$$

$$dst_z = src_x m_{20} + src_y m_{21} + src_z m_{22} + m_{23}$$

It performs 9 multiplication operations.

7.2 MTXMultVecArray

Code 7–2 MTXMultVecArray

```
void MTXMultVecArray   ( Mtx m, VecPtr srcBase, VecPtr dstBase, u32 count );
```

`MTXMultVecArray` multiplies the 3x4 matrix *m* by each vector in the array of 3 element vectors *src[]*, and assigns each result to the user-allocated array of 3 element vectors *dst[]*. All vectors are assumed to have an implicit 4th element (w) of 1, and the matrix *m* is assumed to have an implicit 4th row of (0,0,0,1). (For this reason, `MTXMultVecArray` is not suitable for projecting points through a perspective projection matrix on the CPU). The argument *count* indicates the number of 3-element vectors in both *src[]* and *dst[]*.

Note: `MTXMultVecArray` is a convenience wrapper for `MTXMultVec`. The following two examples are equivalent, except that `MTXMultVecArray` is a little more efficient. In the future, `MTXMultVecArray` may be significantly more efficient.

Code 7–3 Using MTXMultVec with an Array of Vectors

```
#include <revolution.h>

    :

    int i;
    Vec src[50],dst[50];
    Mtx m;
    :
    for (i=0; i<50; i++) MTXMultVec(m,src+i,dst+i);
```

Code 7–4 Using MTXMultVecArray with an Array of Vectors

```
#include <revolution.h>

    :

    int i;
    Vec src[50],dst[50];
    Mtx m;
    :
    MTXMultVecArray(m,src,dst,50);
```

The arguments *src* and *dst* may reference precisely the same vector array (*src*==*dst*). Otherwise, *src*[0]..*src*[*count*-1] must not overlap *dst*[0]..*dst*[*count-1*].

`MTXMultVecArray` performs 9**count* multiplications.

7.3 MTXMultVecSR

Code 7–5 MTXMultVecSR

```
void MTXMultVecSR      ( Mtx m, VecPtr src, VecPtr dst );
```

`MTXMultVecSR` multiplies the 3x3 sub-matrix (scale-and-rotate component) of 3x4 matrix *m* by the 3-element vector *src* and assigns the result to the user-allocated 3-element vector *dst*. The translation component of matrix *m* is ignored.

The arguments *src* and *dst* may reference the same vector.

`MTXMultVecSR` computes:

Equation 7–2 MTXMultVecSR Computations

$$dst_x = src_x m_{00} + src_y m_{01} + src_z m_{02}$$

$$dst_y = src_x m_{10} + src_y m_{11} + src_z m_{12}$$

$$dst_z = src_x m_{20} + src_y m_{21} + src_z m_{22}$$

It performs 9 multiplication operations.

It may help to convert a vector of direction into another coordinate system (e.g., light direction vector into view space).

7.4 MTXMultVecArraySR

Code 7–6 MTXMultVecArraySR

```
void MTXMultVecArraySR ( Mtx m, VecPtr srcBase, VecPtr dstBase, u32 count );
```

`MTXMultVecArraySR` multiplies the 3x3 sub-matrix (scale and rotate component) of 3x4 matrix *m* by each vector in the array of 3-element vectors *src[]* and assigns each result to the user-allocated array of 3-element vectors *dst[]*. The argument *count* indicates the number of 3-element vectors in both *src[]* and *dst[]*.

`MTXMultVecArraySR` is a convenience wrapper for `MTXMultVecSR` (See `MTXMultVecArray`)

The arguments *src* and *dst* may reference precisely the same vector array (*src*==*dst*). Otherwise, *src*[0]..*src*[*count*-1] must not overlap *dst*[0]..*dst*[*count-1*].

`MTXMultVecArraySR` performs 9**count* multiplications.

8 Vector-vector Operations

8.1 The Vector Type

The basic type of these vector operations is the `Vec`, passed by reference using a `VecPtr`:

Code 8–1 Vector and Point Type Definitions

```
typedef struct
{
    f32 x;
    f32 y;
    f32 z;
} Vec;

typedef Vec* VecPtr;
typedef Vec  Point3d;
typedef Vec* Point3dPtr;
```

8.2 The Vector Operations

Code 8–2 Vector Operations

```
void VECAdd             ( VecPtr a, VecPtr b, VecPtr ab );
void VECSubtract        ( VecPtr a, VecPtr b, VecPtr a_b );
void VECScale           ( VecPtr src, VecPtr dst, f32 scale );
void VECNormalize       ( VecPtr src, VecPtr unit );
f32  VECDotProduct      ( VecPtr a, VecPtr b );
void VECCrossProduct    ( VecPtr a, VecPtr b, VecPtr axb );
void VECHalfAngle       ( VecPtr a, VecPtr b, VecPtr half );
void VECReflect         ( VecPtr src, VecPtr normal, VecPtr dst );
f32  VECSquareMag       ( VecPtr v );
f32  VECMag             ( VecPtr v );
f32  VECSquareDistance  ( VecPtr a, VecPtr b );
f32  VECDistance        ( VecPtr a, VecPtr b );
```

All these vector-vector operations act on one or two 3-element vector arguments and return either a scalar result (f32) or a vector result in a user-allocated 3-element vector argument. Any of the one, two, or three vector arguments may reference the same vector. All calculations are single-precision.

`VECAdd` performs the operation ***ab = a + b***. It computes the sum (*a* + *b*) of the two vectors *a* and *b* and assigns the result to *ab*.

`VECSubtract` performs the operation ***a_b = a - b***. It computes the difference (*a* - *b*) of the two vectors *a* and *b* and assigns the result to *a_b*.

`VECScale` performs the operation ***dst*** = *scale*·***src***. It scales the vector *src* by the scalar *scale*, and assigns the result (*scale*·*src*) to *dst*.

`VECNormalize` performs the operation ***unit*** = ***src***/|***src***|. It calculates the length or magnitude of the vector *src*, and normalizes the vector, storing the result in *unit*. The vector *src must* have non-zero length. The computation requires one square-root, one divide, and six multiply operations.

`VECDotProduct` returns the scalar ***a***·***b***. This is a vector dot-product operation. It requires three multiplies.

`VECCrossProduct` performs the operation ***axb*** = ***a*** x ***b***. This is a vector cross-product operation. It requires 6 multiplications.

`VECHalfAngle` performs the operation ***half*** = (|***b***|·***a*** + |***a***|·***b***) / | (|***b***|·***a*** + |***a***|·***b***) |. This calculates a unit vector *half*, which lies halfway between the two vectors *a* and *b*. This is suitable for specular highlight half-angle computations. The vectors *a* and *b must* both have non-zero length, and they *must not* point in exactly opposite directions. `VECHalfAngle` requires three square-roots, one division, and 18 multiplications.

`VECReflect` reflects the vector *src* in the plane defined by all points **n**, such that **n**·***normal*** **= 0**, using the rule “angle-of-reflection equals angle-of-incidence”. The unit-length result is assigned to *dst*. The entire computation requires three square-root operations, three divisions, and 24 multiplications.

`VECSquareMag` returns $|\boldsymbol{v}|^2$, the square of the magnitude of the vector *src*. It requires three multiplications.

`VECMag` returns |***v***|, the scalar length or magnitude of the vector *src*. It requires three multiplications and one square-root operation.

`VECSquareDistance` calculates $|\boldsymbol{a}-\boldsymbol{b}|^2$, or the square of the distance between vectors *a* and *b*. It requires three multiplications.

`VECDistance` calculates |***a-b***|, or the distance between vectors *a* and *b*. It requires three multiplications and one square root operation.

9 Stack Operations

9.1 The Matrix Stack Type

The `MtxStack` is used to record the location of the allocated array, the current position in the array, and the size of the array. `MtxStackPtr` is used as a handle to a stack. This handle is passed to all matrix stack calls, and indicates the stack on which to operate.

Code 9–1 MtxStack and MtxStackPtr Type Definitions

```
typedef struct
{
    u32    numMtx;
    MtxPtr stackBase;
    MtxPtr stackPtr;
} MtxStack;

typedef MtxStack* MtxStackPtr;
```

9.2 MTXAllocStack and MTXFreeStack

Code 9–2 MTXAllocStack and MTXFreeStack

```
void MTXAllocStack      ( MtxStackPtr sPtr, u32 numMtx );
void MTXFreeStack       ( MtxStackPtr sPtr );
```

`MTXAllocStack` allocates an array of *numMtx* 3x4 matrices (type `Mtx`). The memory is allocated using `OSAlloc()`. (The user must allocate the `MtxStackPtr` structure).

`MTXFreeStack` frees the allocated array, using `OSFree()`.

`MTXAllocStack` and `MTXFreeStack` are implemented as macros in `mtx.h`. They are provided for your convenience — it is not essential to allocate and free a stack using `MTXAllocStack`/`MTXFreeStack`. Code 9–3 and Code 9–4 show two ways to allocate a matrix stack:

Code 9–3 Using MTXAllocStack to Allocate a Matrix Stack

```
    #include <revolution.h>

    MtxStackPtr mstack;

        :
        MTXAllocStack(mstack,10);  // calls OSAlloc to allocate 10*(3*4)*4 bytes
        MTXInitStack(mstack,10);

        do_some_work();

        MTXFreeStack(mstack);
```

Code 9–4 Not Using MTXAllocStack to Allocate a Matrix Stack

```
#include <revolution.h>

MtxStackPtr mstack;
Mtx mstackspace[10];

    :
    mstack.stackbase = mstackspace;

    MTXInitStack(mstack,10);
```

9.3 MTXInitStack

Code 9–5 MTXInitStack

```
void    MTXInitStack    ( MtxStackPtr sPtr, u32 numMtx );
```

`MTXInitStack` sets up the `MtxStack` structure referenced by *sPtr*. It must be called before any stack operations are performed on the stack. The argument *numMtx* indicates the maximum legal size for the stack—that is, the maximum number of 3x4 matrices that can be pushed. It must not be zero.

9.4 MTXPush, MTXPushFwd, MTXPushInv, MTXPushInvXpose

Code 9–6 MTXPush, MTXPushFwd, MTXPushInv, and MTXPushInvXpose

```
MtxPtr  MTXPush         ( MtxStackPtr sPtr, Mtx m );
MtxPtr  MTXPushFwd      ( MtxStackPtr sPtr, Mtx m );
MtxPtr  MTXPushInv      ( MtxStackPtr sPtr, Mtx m );
MtxPtr  MTXPushInvXpose ( MtxStackPtr sPtr, Mtx m );
```

These operations push a 3x4 matrix onto the matrix stack referenced by *sPtr*. There must be room in the matrix stack for a new 3x4 matrix. They return a `MtxPtr` reference to the new top-of-stack matrix.

`MTXPush` pushes the matrix ***m*** onto the top of the stack. The matrix is not concatenated or multiplied with any other. The remaining three operations pre- or post-concatenate ***m***, $\boldsymbol{m}^{-1}$, or $(\boldsymbol{m}^{-1})^T$ with the matrix on the top of the stack (**t**), and push the result:

`MTXPush` pushes ***m***.

`MTXPushFwd` pushes **t** x ***m***.

`MTXPushInv` pushes $\boldsymbol{m}^{-1}$ x **t**.

`MTXPushInvXpose` pushes **t** x $(\boldsymbol{m}^{-1})^T$.

If the stack is empty, **t** is taken to be the identity matrix (**I**).

`MTXPush` always performs a `MTXCopy` operation. The remaining three operations perform a `MTXCopy` if the stack was empty, otherwise they perform a `MTXConcat`.

`MTXPushInv` and `MTXPushInvXpose` also perform a `MTXInverse`. Therefore, ***m*** must be non-singular (invertible). `MTXPushInvXpose` also performs a `MTXTranspose`.

9.5 MTXPop

Code 9–7 MTXPop

```
MtxPtr  MTXPop          ( MtxStackPtr sPtr );
```

`MTXPop` discards the top 3x4 matrix element in the stack. It then returns a `MtxPtr` reference to the new top-of-stack, or `NULL` if the stack is now empty.

If the stack is already empty, `MTXPop` has no effect, and returns `NULL`.

9.6 MTXGetStackPtr

Code 9–8 MTXGetStackPtr

```
MtxPtr  MTXGetStackPtr  ( MtxStackPtr sPtr );
```

`MTXGetStackPtr` returns a `MtxPtr` reference to (**t**), the top 3x4 matrix element in the stack. If the stack is empty, it returns `NULL`.

10 Traps and Pitfalls

Because the library is designed to be runtime efficient first, and easy-to-use second, some warnings and cautions are necessary.

10.1 The Standard Matrix Type is 3x4

The standard matrix type is `Mtx`, a 3x4 matrix with an assumed, implicit 4th row of (0,0,0,1). This size was selected because it is sufficient for all 3D scale, rotation, and translate operations and combinations thereof. Although 4x4 matrices could be used as the general standard, they have some disadvantages compared to 3x4 matrices:

- 4x4 matrices require extra memory storage (33% extra).
- In memory-limited situations, 4x4 matrix computations will be 33% slower.
- In CPU-speed limited situations, many general 4x4 operations are computationally much slower (for example, `MTXConcat` requires 64 multiplies verses 36).

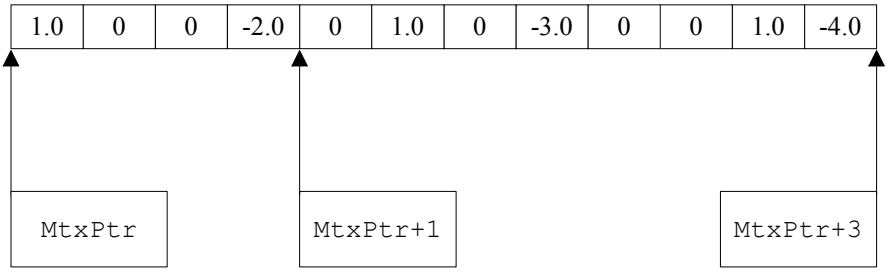
However, 3x4 is not adequate for 3D projection matrices (perspective or parallel). Therefore, a `Mtx44` type is used in the special cases of `MTXPerspective`, `MTXFrustum`, and `MTXOrtho`. This type is incompatible with `Mtx` and cannot be passed to any other API calls in the library. This is slightly inconvenient in some circumstances.

10.2 Mtx is an Array of 3 Arrays of 4 Floats

The definitions of `Mtx` and `MtxPtr` (and their 4x4 cousins) deserve some careful attention—it is possible to get confused.

For example, `MtxPtr` is not a pointer to an `Mtx`, it is a pointer to an array of floats:

Figure 10–1 MtxPtr Points at a Mtx in Memory



Therefore, it is important to always remember to increment as shown in the example below:

Code 10–1 Incrementing a MtxPtr—the right way

```
MtxPtr mp;
Mtx m[2];
mp=(MtxPtr)m;
mp+=MTX_PTR_OFFSET;       // Increment to next matrix
```

…instead of:

Code 10–2 Incrementing a MtxPtr—the wrong way

```
MtxPtr mp;
Mtx m[2];
mp=(MtxPtr)m;
mp++;                     // Increment to next matrix (actually to next column).
```

10.3 Multiple References

Although it is safe to pass a reference to one matrix or vector to any routine in the library multiple times, some routines will have to perform an additional copy of the input type. For example, `MTXConcat(a, a, a)` may perform a `MTXCopy(a, temp)`.

10.4 Rules-of-use

Many of these library routines have rules-of-use. These are generally associated with divide-by-zero conditions (but some are associated with Matrix stack bounds-checking). For example, `VECNormalize()` must not be called with a zero-length vector. The library routines are unforgiving. No error conditions or flags are returned to the calling environment (except `MTXInverse`). In this respect, `mtx.h` is unlike `math.h`, for example.

If one of these rules-of-use is broken, the library routine will `ASSERT()` and halt the game, *provided* you are using the debug libraries.

Note: With the non-debug libraries, error behavior is undefined. (In most cases, one or more single-precision floating-point ±infinity value(s) will be yielded in the returned scalar/vector/matrix. These will spread through future operations, until (for example) an infinity is multiplied by zero. An exception will then be raised. However, the exception may be raised a long distance from and a long time after the source of the problem. Your mileage may vary.)

10.5 Macros

Some routines are in fact C-preprocessor macros (`MTXDegToRad`, `MTXRadToDeg`, `MTXAllocStack`, `MTXFreeStack`, `MTXRowCol`). Although macros and functions appear to the user to be identical, macros behave slightly differently in some circumstances. In general, they are slightly more dangerous.

For example, the following will compile (and probably run) without errors or warnings:

Code 10–3 Macros are *not* Functions

```
#include <revolution.h>
void main(void) {
    MTXAllocStack(main /*or-any-ptr/scalar-type*/ ,10);
                                // self-modifying main() code
}
```

Appendix A. Tables of API Calls

A.1 Scale, Rotate, and Translate Transformations

Table A–1 Scale, Rotate, and Translate Transformations

API	Arguments	Summary
MTXIdentity	(Mtx m)	M = I
MTXScale	(Mtx m, f32 x, f32 y, f32 z)	M = Scale by (x, y, z)
MTXRotRad	(Mtx m, u8 axis, f32 rad)	M = Rotate about positive axis ['x'|'y'|'z'] counter-clockwise rad radians
MTXRotTrig	(Mtx m, u8 axis, f32 sinA, f32 cosA)	M = Rotate about positive axis ['x'|'y'|'z'] counter-clockwise angle A
MTXRotAxisRad	(Mtx m, VecPtr axis, f32 rad)	M = Rotate about v_{axis} counter-clockwise by rad radians
MTXTrans	(Mtx m, f32 x, f32 y, f32 z)	M = Translate by (x, y, z)
MTXQuat	(Mtx m, QuaternionPtr q)	M = q
MTXDegToRad	(f32 degrees)	Return degrees/180.0*pi
MTXRadToDeg	(f32 radians)	Return radians/pi*180.0

A.2 View Transformations

Table A–2 View Transformations

API	Arguments	Summary
MTXLookAt	(Mtx m, Point3dPtr camPos, VecPtr camUp, Point3dPtr target)	M = Look at: v_{target} from: v_{camPos}, up is: v_{camUp}

A.3 Projection Transformations

Table A–3 Projection Transformations

API	Arguments	Summary
MTXPerspective	(Mtx44 m, f32 fovY, f32 aspect, f32 n, f32 f)	M: Models camera with field-of-view fovY degrees high, fovY*aspect wide, near n, far f
MTXFrustum	(Mtx44 m, f32 t, f32 b, f32 l, f32 r, f32 n, f32 f)	M: Maps frustum 4 corners ([r|l], [b|t], -n) to (±1.0, ±1.0, 1.0)
MTXOrtho	(Mtx44 m, f32 t, f32 b, f32 l, f32 r, f32 n, f32 f)	M: Maps cube with corners ([l|r],[b|t],-[f|n]) to (±1.0, ±1.0, ±1.0)

A.4 Texture Projection Transformations

Table A–4 Texture Projection Transformations

API	Arguments	Summary
MTXLightPerspective	(Mtx m, f32 fovY, f32 aspect, f32 scaleS, f32 scaleT, f32 transS, f32 transT)	M: Maps (x, y, z) to (s, t, q) for a light with a rectangular aperture fovY degrees tall, aspect ratio (X/Y) aspect. The scales and the translates position the texture; use (0.5, 0.5, 0.5, 0.5) to obtain a single, centralized image of a non-mirrored texture.
MTXLightFrustum	(Mtx m, f32 t, f32 b, f32 l, f32 r, f32 n, f32 scaleS, f32 scaleT, f32 transS, f32 transT)	M: Maps (x, y, z) to (s, t, q) for a light with a rectangular aperture bounded by the points (l/r, t/b, n) in light space. The scales and translates position the texture; use (0.5, 0.5, 0.5, 0.5) to obtain a single, centralized image of a non-mirrored texture.
MTXLightOrtho	(Mtx m, f32 t, f32 b, f32 l, f32 r, f32 scaleS, f32 scaleT, f32 transS, f32 transT)	M: Maps (x, y, z) to (s, t, q) for a light at infinity with a (much closer) rectangular aperture bounded by the points (l/r, t/b, ?) in light space. The scales and translates position the texture; use (0.5, 0.5, 0.5, 0.5) to obtain a single, centralized image of a non-mirrored texture.

A.5 Matrix-matrix Operations

Table A–5 Matrix-matrix Operations

API	Arguments	Summary
MTXConcat	(Mtx a, Mtx b, Mtx axb)	$M_{axb} = M_a \times M_b$
MTXCopy	(Mtx src, Mtx dst)	$M_{dst} = M_{src}$
MTXTranspose	(Mtx src, Mtx xPose)	$M_{dst} = M_{src}^T$
MTXInverse	(Mtx src, Mtx inv)	$M_{dst} = M_{src}^{-1}$ Returns 0 if singular

A.6 Matrix-vector Operations

Table A–6 Matrix-vector Operations

API	Arguments	Summary
MTXMultVec	(Mtx m, VecPtr src, VecPtr dst)	$v_{dst} = M \times v_{src}$
MTXMultVecArray	(Mtx m, VecPtr srcBase, VecPtr dst-Base, int count)	forall(i): $v_{dst,\ i} = M \times v_{src,\ i}$
MTXMultVecSR	(Mtx m, VecPtr src, VecPtr dst)	$v_{dst} = M_{3x3} \times v_{src}$
MTXMultVecArraySR	(Mtx m, VecPtr srcBase, VecPtr dst-Base, int count)	forall(i): $v_{dst,\ i} = M_{3x3} \times v_{src,\ i}$

A.7 Vector-vector Operations

Table A–7 Vector-vector Operations

API	Arguments	Summary
VECScale	(VecPtr src, VecPtr dst, f32 scale)	$v_{dst} = scale \times v_{src}$
VECMag	(VecPtr src)	return $\lvert v_{src} \rvert$
VECNormalize	(VecPtr src, VecPtr unit)	$v_{unit} = v_{src} / \lvert v_{src} \rvert$
VECDotProduct	(VecPtr a, VecPtr b)	return $v_a \cdot v_b$
VECCrossProduct	(VecPtr a, VecPtr b, VecPtr axb)	$v_{axb} = v_a \times v_b$
VECHalfAngle	(VecPtr a, VecPtr b, VecPtr half)	$v_{half} = (\lvert v_b \rvert \cdot v_a + \lvert v_a \rvert \cdot v_b) /$ length
VECReflect	(VecPtr src, VecPtr normal, VecPtr dst)	v_{dst} = reflection of v_{src} in plane $n.v_{normal}=0$
VECSquareDistance	(VecPtr a, VecPtr b)	return |va-vb|2
VECDistance	(VecPtr a, VecPtr b)	return |va-vb|

A.8 Stack Operations

Table A–8 Stack Operations

API	Arguments	Summary
MTXAllocStack	(MtxStackPtr sPtr, u32 numMtx)	sPtr=OSAlloc(numMtx*sizeof(Mtx))
MTXFreeStack	(MtxStackPtr sPtr)	OSFree(sPtr)
MTXInitStack	(MtxStackPtr sPtr, u32 numMtx)	stack = { }; at=0; maxlen=numMtx
MTXPush	(MtxStackPtr sPtr, Mtx m)	stack[at] = M; return &stack[++at]
MTXPushFwd	(MtxStackPtr sPtr, Mtx m)	stack[at] = stack[at-1] [a] x M; return &stack[++at]
MTXPushInv	(MtxStackPtr sPtr, Mtx m)	stack[at] = M^{-1} x stack[at-1]; return &stack[++at]
MTXPushInvXpose	(MtxStackPtr sPtr, Mtx m)	stack[at] = stack[at-1] x $(M-1)^{T}$; return &stack[++at]
MTXPop	(MtxStackPtr sPtr)	return &stack[--at]
MTXGetStackPtr	(MtxStackPtr sPtr)	return &stack[at]

a. The Identity matrix is substituted for references to stack[-1]