

Current assignments:

Readings: finish Ch. 5, start Ch. 6 and 7; read Redbook Ch. 3 (Viewing and Modeling)

HW3: 2-d rectangular transformations, practice with angles in space, and dot/cross products.

Project 3: DUE Wed. Oct 26. (2 weeks from today)

1. (30 pts.) Add fingers (at least two) to the RedBook Ch. 3 robot. The fingers must be attached and pivot at the end of the forearm. Of course, when the shoulder moves, the forearm and fingers move as if rigidly attached; same for the fingers when the elbow moves.
2. (30 pts.) Make those fingers move independantly (via keyboard, mouse or animated control, your choice.)
3. (30 pts.) Put the moving or moveable robot on the planet of the RedBook Ch. 3 solar system by putting the robot drawing code in a **separate function** (as illustrated by Redbook Example 3-4). The base of the robot should be fixed on the surface of the planet. Therefore, the whole robot arm will move together as the planet revolves around the sun and rotates around its center.

4. (10 pts) Implementing a much better visualization by going beyond wire frame objects and using a better viewing setup.

New studying: HB 5-11 Three-dimensional rotations.

Problem: Givens: $P_1 = (x_1, y_1, z_1)$, $P_2 = (x_2, y_2, z_2)$ and Θ . Goal:

Calculate the 4x4 matrix \mathbf{R} so $\mathbf{R}(x, y, z, 1)^t = (x', y', z', 1)^t$ transforms point $\mathbf{P} = (x, y, z)$ into by (x', y', z') by **rotating \mathbf{P} by angle Θ around the line P_1P_2 .**

We form \mathbf{R} by multiplying 5 matrices corresponding to the following steps:

1. Translate (x, y, z) with the translation that moves $\mathbf{P}_1 = (x_1, y_1, z_1)$ to $(0, 0, 0)$. Easy: $(x', y', z') = (x - x_1, y - y_1, z - z_1)$.
2. Rotate this (x', y', z') with the rotation that **MOVES THE ROTATION AXIS $\mathbf{P}_2 - \mathbf{P}_1 = (x_2 - x_1, y_2 - y_1, z_2 - z_1)$** (which is called \mathbf{V} in equation 5-81) **INTO THE z -axis**. (We will solve this sub-problem today.)
3. Apply the rotation by angle Θ **around the z -axis**. (That was your homework.)
4. Apply the inverse of the transformation of step 2. (That inverse is gotten by **transposing** the matrix because the inverse of a rotation matrix is always the transpose, unlike ordinary matrices).
5. Apply the inverse of the transformation of step 1.

The GIVENS are 7 numbers: $P_1 = (x_1, y_1, z_1)$, $P_2 = (x_2, y_2, z_2)$ and Θ .
 Guide to HB pages 268-269:

$$\mathbf{V} = \mathbf{P}_2 - \mathbf{P}_1 = (x_2 - x_1, y_2 - y_1, z_2 - z_1)$$

$$\mathbf{u} = \frac{\mathbf{V}}{|\mathbf{V}|} \text{ which is called } (a, b, c)$$

The components (a, b, c) are the **direction cosines** of the direction around which we want to rotate. a for example is calculated from the givens by

$$a = \frac{x_2 - x_1}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}} \text{ etc.}$$

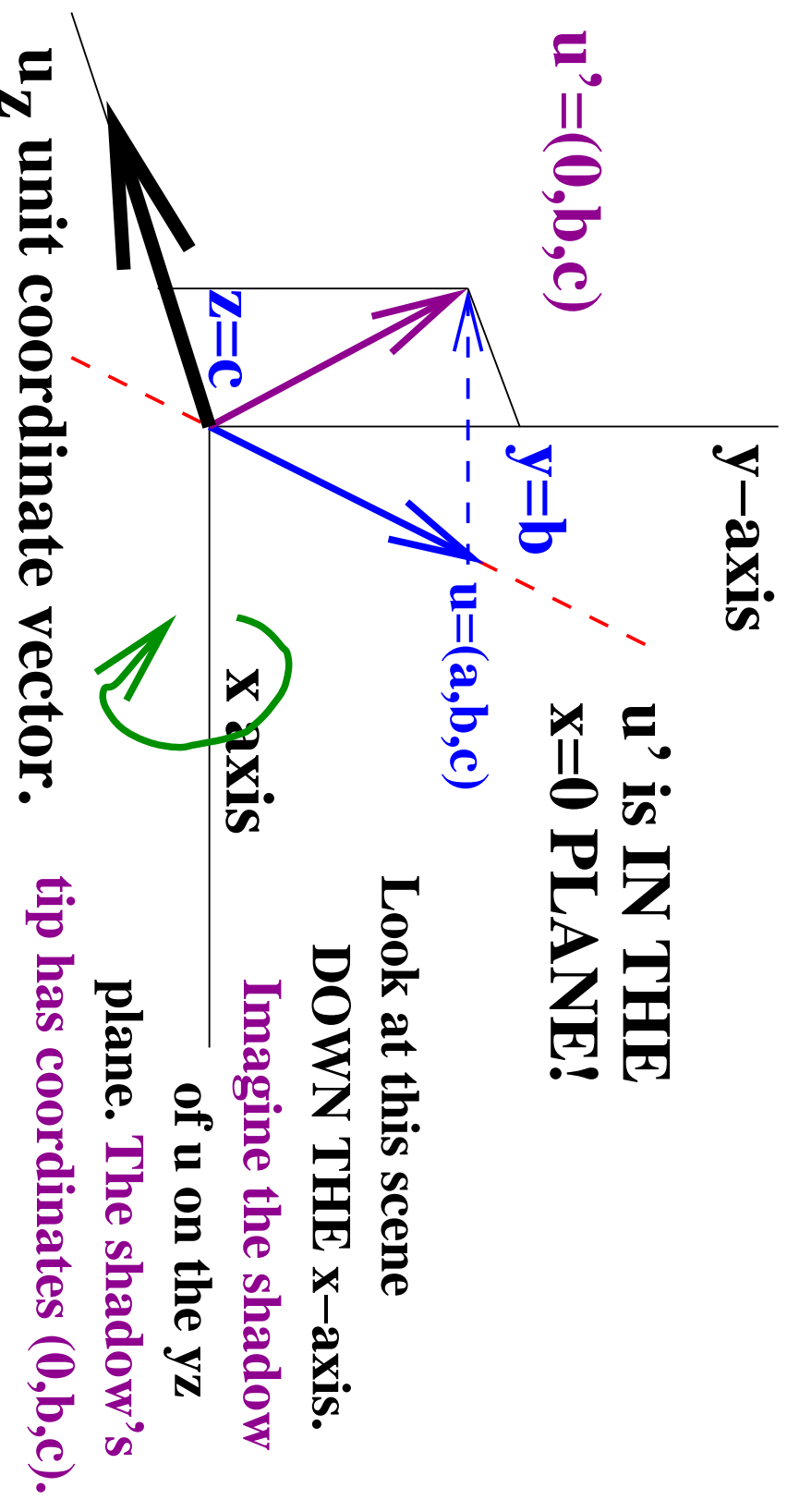
Of course real CS people would do it more efficiently (how??).

We will form the matrix for step 2 (rotate \mathbf{u} into the z axis) by multiplying two rotation matrices $\mathbf{R}_y(\beta)\mathbf{R}_x(\alpha)$.

$\mathbf{R}_x(\alpha)$ will rotate \mathbf{u} around the x axis so \mathbf{u} is transformed into a vector in the xz plane. HB call this vector \mathbf{u}'' .

$\mathbf{R}_y(\beta)$ will rotate \mathbf{u}'' around the y axis so \mathbf{u}'' is transformed into the unit vector in the z direction.

Let's see how to find $\mathbf{R}_x(\alpha)$ in terms of the NUMBERS (a, b, c) we had calculated from the first 6 givens.



What rotation AROUND THE x-AXIS will rotate $\mathbf{u}' = (0, b, c)$ into the z-axis? This will rotate (a, b, c) into $(a, 0, \text{something})$ since a rotation around the x axis doesn't change x-coordinates (like a).

Well, here is the rotation around the x-axis that rotates \mathbf{u}_z into the unit vector in the direction of \mathbf{u}' :

Let $d = \sqrt{b^2 + c^2}$

$$\mathbf{R}_x(-\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{c}{d} & \frac{b}{d} & 0 \\ 0 & -\frac{b}{d} & \frac{c}{d} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Check: this transforms $\mathbf{u}_z = (0, 0, 1, 0)^t$ into $(0, b/d, c/d, 0)^t$.

This transforms \mathbf{u}_y into a unit vector $(0, c/d, -b/d, 0)$ in the yz plane orthogonal to $(0, b/d, c/d, 0)$.

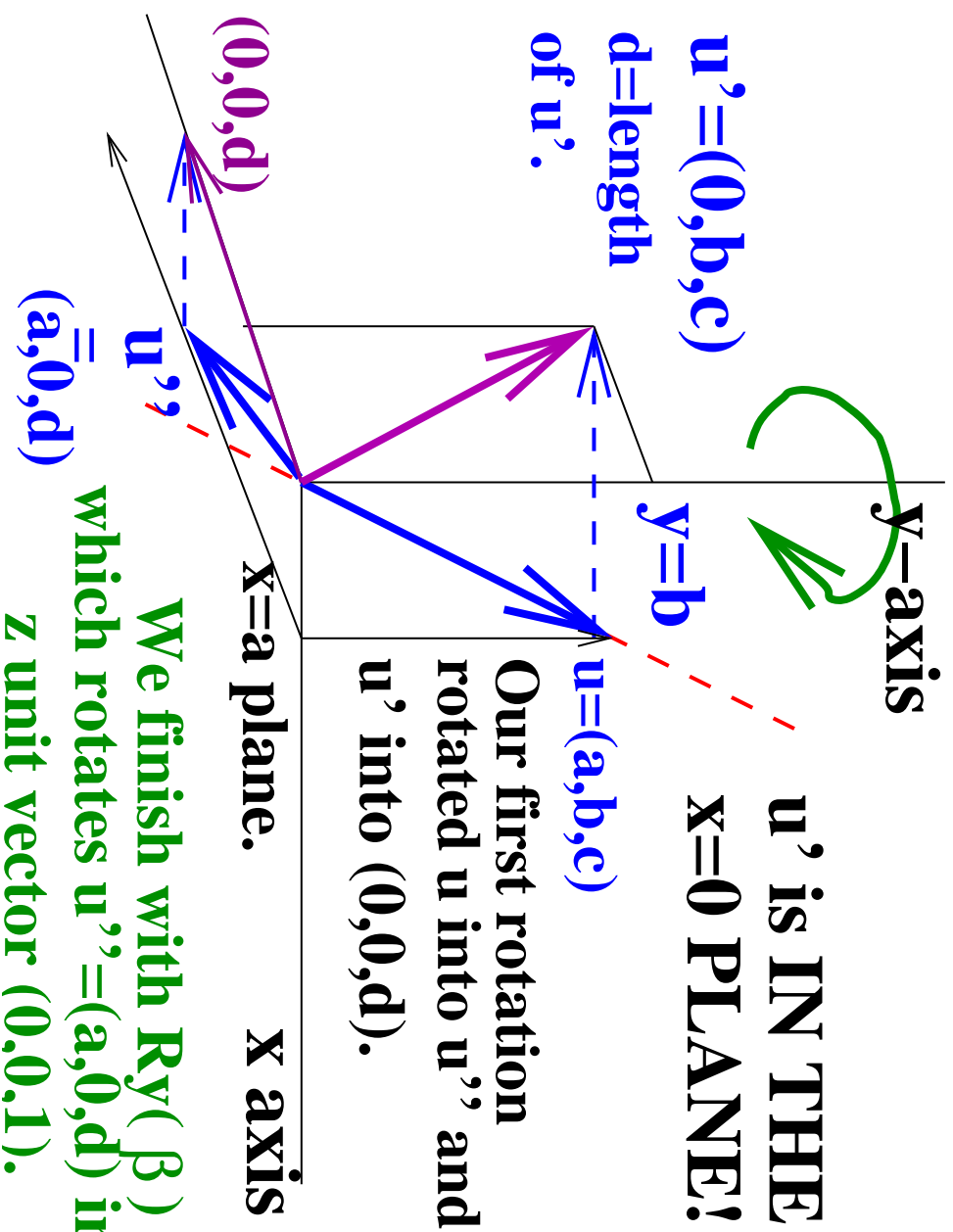
SO... (great!) the desired rotation matrix for step 2 is the transpose of this matrix.

$\mathbf{R}_x(-\alpha)$ rotates \mathbf{u}_z into $(0, b/d, c/d)$.

$\mathbf{R}_x(-\alpha)^{-1} = \mathbf{R}_x(\alpha)$ rotates $\mathbf{u}' = (0, b/d, c/d)$ into \mathbf{u}_z :

$$\mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{c}{d} & -\frac{b}{d} & 0 \\ 0 & \frac{b}{d} & \frac{c}{d} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$\mathbf{R}_x(\alpha)$ doesn't change the x component. It rotates $\mathbf{u} = (a, b, c)$ into $(a, 0, d)$. (d is the length of the projection $(0, b, c)$ "shadow" into the yz plane. The rotation rotates this $(0, b, c)$ into the vector with the same length, d , along the z axis: $(0, 0, d)$.)



$$R_y(\beta) = \begin{bmatrix} d & 0 & -a & 0 \\ 0 & 1 & 0 & 0 \\ a & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{since } R_y(\beta)^{-1} = \begin{bmatrix} d & 0 & a & 0 \\ 0 & 1 & 0 & 0 \\ -a & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

So, with

$$\mathbf{R}_z(\Theta) = \begin{bmatrix} \cos \Theta & -\sin \Theta & 0 & 0 \\ \sin \Theta & \cos \Theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

the problem is solved by

$$\mathbf{R}(\Theta) = \mathbf{T}^{-1} \mathbf{R}_x^{-1}(\alpha) \mathbf{R}_y(\beta)^{-1} \mathbf{R}_z(\Theta) \mathbf{R}_y(\beta) \mathbf{R}_x(\alpha) \mathbf{T}$$

Although HB explain the rotation matrices $R_x(\alpha)$ and $R_y(\beta)$ with angles α , β and their sines and cosines, numerically computing α and β is NOT NECESSARY, and would be a waste of time.

Only a , b , c , $d = \sqrt{b^2 + c^2}$, b/d , c/d , $\sin \Theta$, $\cos \Theta$, and the matrix multiplications actually have to be computed when $\mathbf{R}(\Theta)$ is computed with this method.

METHOD 2: Remember \mathbf{u} ? The unit vector pointing along the axis of rotation.

If $\mathbf{u} = (0, 0, 1)$ or $(0, 0, -1)$ we needn't work hard. So, first test

```
if (a==0.0 && b==0.0){ /* just use Rz(z*Theta) */}
else { /* use stuff below */}
```

So, $\mathbf{u} = (a, b, c)^t$ and one or both of a, b is non-zero. Rename \mathbf{u} by \mathbf{u}'_z , and calculate:

$$\mathbf{u}'_y = \frac{\mathbf{u} \times \mathbf{u}_z}{|\mathbf{u} \times \mathbf{u}_z|}$$

This is guaranteed to be non-zero. The case when \mathbf{u}_z has the same direction as $\mathbf{u}'_z = \mathbf{u}$ was eliminated by the if (. . .) test above!

This is guaranteed to be perpendicular to \mathbf{u}'_z because of the geometric meaning of cross product. The division by $|\mathbf{u} \times \mathbf{u}_z|$ makes it unit length.

Next, calculate

$$\mathbf{u}'_x = \mathbf{u}'_y \times \mathbf{u}'_z$$

This is guaranteed to be unit length and perpendicular to both \mathbf{u}'_y and \mathbf{u}'_z because these two vectors have unit length and are perpendicular to each other (so the sine of the angle between them is ± 1 .)

The order $\mathbf{u}'_y \times \mathbf{u}'_z$ rather than $\mathbf{u}'_z \times \mathbf{u}'_y$ was chosen to make $\mathbf{u}'_x, \mathbf{u}'_y, \mathbf{u}'_z$ a right-handed coordinate system.

Now the matrix:

$$\begin{bmatrix} [\mathbf{u}'_x] & [\mathbf{u}'_y] & [\mathbf{u}'_z] \end{bmatrix}$$

rotates the original axes into orthogonal axes in which u_z is rotated into OUR DESIRED axis direction.

So the inverse of this matrix will rotate OUR DESIRED DIRECTION unit vector \mathbf{u} into the $z - axis$.

And the inverse of this matrix is its transpose.

METHOD 3: algebraic work on the matrices—no visualization needed.
(future handout).

METHOD 4: Quaternions: Cool for continuous animated rotation.
(Professor must do homework.)

OpenGL state includes 3 matrices: **modelview**, **projection** and **texture mapping**.

The state also includes 3 **matrix stacks**.

OpenGL API style:

- Each command modifies part of the state.
- The effect of each command is determined by modifications done by *previous* commands.

For example, **COLOR**: Current color assigns a color to all subsequent *vertices*. You can change the current color between calls to `glVertex*(...);`

When the object is rendered, the fill or edge colors are **interpolated** from corner or endpoint vertices.

EG, to make the current *modelview* matrix be the identity matrix:

```
glMatrixMode ( GL_MODELVIEW );  
//Subsequent matrix ops apply to the MODELVIEW matrix.  
glLoadIdentity( );  
//Now the MODELVIEW matrix equals I.
```

RedBook p.104: “Viewing and modeling transformations are inextricably related in OpenGL and are in fact combined into a single modelview matrix.”

Confused??

The OpenGL modelview matrix is called the modelview matrix because *YOU* can use it for *either modeling or viewing or BOTH!*

So, it’s *your job* to create modeling and viewing transformations to create your graphics, and to combine them into OpenGL in the way OpenGL accepts them.

OpenGL’s access to its matrices is restricted because OpenGL drives modern graphics card hardware.

Modern graphics (and CPU) hardware is **pipelined**: A series of separate stages all operate simultaneously; each command progresses from stage to stage at each clock step.

Study HB 6-1 to 6-4; 7-1, 7-2, 7-3: 2 and 3-d Graphics **Pipelines**:

1. Input of Master or template **model**
2. → **Modeling Transformation** to produce an instance (in the world).
3. → **Viewing Transformation** which moves the world in front of the standard camera. (The numbers now describing the primitives are sometimes called viewing coordinates.)
4. → **Projection Transformation** (produces a perspective or other projection when z coordinate is ignored.)
5. → **Normalization Transformation and Clipping** (normalizes coordinates within the view volume to $[0,1]$ or $[-1,1]$ range and clips primitives outside the view volume)
6. → **Viewport Transformation** maps the 2-d $[0,1]$ or $[-1,1]$ coordinates to given rectangle in the screen or graphics window, expressed in pixel coordinates.

The current **OpenGL modelview matrix** MV transforms coordinates of the primitives

- entered via `glVertex*(...)`
- or indirectly via GLUT models like `glutWireSphere(...)`

by

$$V' = MV \cdot V$$

LEFT MULTIPLICATION.

Many graphics textbooks and manuals use the word “concatenation” for matrix multiplication.

```
glMatrixMode( GL_MODELVIEW ); glLoadIdentity( );
```

set the modelview matrix to I .

```
glLoadMatrix( ... );
```

 sets modelview matrix to a given matrix

```
glMultMatrix( T );
```

 DOES $MV = MV \cdot T$

Multiplies the given matrix on the RIGHT

```
glTranslatef(20.0,30.0,40.0);
```

MULTIPLIES ON THE RIGHT the matrix that translates $(0,0,0)$ to $(20.0,30.0,40.0)$.

(1) Using the (20,30,40) translation for MODELING: (See Example 3-4 in the RedBook)

```
glTranslatef(20.0,30.0,40.0);
draw_wheel_and_bolts();
```

draws an instance of wheel and bolts AT POSTION (20,30,40).

Think: Transformations MOVE or CHANGE OBJECTS all described using a *fixed* coordinate system.

(2) Using the translation (-100,-100,0) for VIEWING:

```
glTranslatef(-100,-100,0);
draw_THE_WORLD();
```

draws an image of the WORLD as if the CAMERA were centered at point (100,100,0).

Think: Transformations CHANGE the SYSTEM by which numeric coordinates describe *fixed* objects.

(don't worry—glu makes it easy to code viewing transformations)

REMEMBER the OpenGL **right matrix multiply** rule:

```
glTranslatef(20.0, 30.0, 40.0);  
glRotatef(45.0, 1.0, 0.0, 0.0 );  
draw_wheel_and_bolts();  
draws wheel and bolts AFTER
```

1. **FIRST**, rotating the model around x axis by 45° ,
2. and **SECOND**, translating (the rotated model) so its origin is at $(20,30,40)$.

OpenGL for Beginners: Use only one view.

FIRST, initialize the modelview matrix with the VIEWING transformation: Use the handy `gluLookAt()`;

SECOND, use the modelview matrix stack to save the current matrix when you temporarily want a MODELLING transformation to apply to primitives, `glut` models, or your own model-drawing functions:

```
glPushMatrix();           //SAVE current MV matrix.
glTranslatef( ... );     //where I want my rotated model to go.
glRotatef( ... );       //rotate the model.
draw_model( ... );      //uses MODELLING coordinates.
glPopMatrix();          //RESTORE saved MV matrix
```

Current assignments:

Readings: finish Ch. 5, start Ch. 6 and 7; read Redbook Ch. 3 (Viewing and Modeling)

HW3: 2-d rectangular transformations, practice with angles in space, and dot/cross products.

Project 3: DUE Wed. Oct 26. (2 weeks from today)

1. (30 pts.) Add fingers (at least two) to the RedBook Ch. 3 robot. The fingers must be attached and pivot at the end of the forearm. Of course, when the shoulder moves, the forearm and fingers move as if rigidly attached; same for the fingers when the elbow moves.
2. (30 pts.) Make those fingers move independantly (via keyboard, mouse or animated control, your choice.)

3. (30 pts.) Put the moving or moveable robot on the planet of the RedBook Ch. 3 solar system by putting the robot drawing code in a **separate function** (as illustrated by Redbook Example 3-4). The base of the robot should be fixed on the surface of the planet. Therefore, the whole robot arm will move together as the planet revolves around the sun and rotates around its center.

4. (10 pts) Implementing a much better visualization by going beyond wire frame objects and using a better viewing setup.

So, the pattern for modeling transformation usage is

```
glPushMatrix();  
glTranslate, glRotate, glScale, etc to apply to the model.  
draw_model_fun(); //function to draw the model should be used.  
glPopMatrix();
```

The two ways to understand modeling transformations:

1. The current modeling transformation **APPLIES TO** the vertices defining the model.
2. The current modeling transformation **TRANSFORMS** **COORDINATES** so when the model is defined using its **OWN** (local, modeling) coordinates, it will be drawn in your desired location and scale within your scene.

For example, `robot.c` uses `glutWireCube(1.0)` which draws a $1 \times 1 \times 1$ cube parallel to the axes and **CENTERED** at the origin of OpenGL's world coordinate system.

in anticipation of drawing the upper arm as a $2 \times 0.4 \times 1$ rectangular solid,
`void display(void)`

```
{
    glClearColor (GL_COLOR_BUFFER_BIT);
    glPushMatrix();
    glTranslatef (-1.0, 0.0, 0.0);
    glRotatef ((GLfloat) shoulder, 0.0, 0.0, 1.0);
    glTranslatef (1.0, 0.0, 0.0);
    //The 2x0.4x1 rectangular solid and all other future
    //objects WILL BE ROTATED around the left end
    //of our solid WHEN THEY ARE DRAWN.
    //Here is a case of (Tinv)(Rot)(T)!

    //Alternative view: The current coordinate
    //system now aligned with its x-axis along
    //the rotated upper arm. Let's call this the
    //OLD system now, because we want to position
    //future objects relative to the rotated
    //upper arm.
    glPushMatrix();
```

```
glScalef (2.0, 0.4, 1.0);
glutWireCube (1.0);
glPopMatrix();

//Scaled cube as been drawn around the origin.
//No permanent change to MV transformation.

    glTranslatef (1.0, 0.0, 0.0);
//TRANSFORM THE COORDINATE SYSTEM so
//THE ORIGIN in NEW SYSTEM
//is (x=1,y=0,z=0) in the OLD SYSTEM.
//That is the FAR END of the upper arm.

    glRotatef ((GLfloat) elbow, 0.0, 0.0, 1.0);
//WILL rotate AROUND THE NEW origin
//(point (1,0,0) in OLD system is at the
//end of the upper arm.
//ALSO, changes to coordinate system to a
//NEW-NEW system so the x- and y-directions
//are NOW rotated relative
```

```
//the old x- and y-directions.
```

```
    glTranslatef (1.0, 0.0, 0.0);
```

```
//TRANSFORM THE COORDINATE SYSTEM so the
```

```
//origin in the NEW-NEW-NEW system
```

```
//is (x=1,0,0) in the NEW-NEW system.
```

```
    glPushMatrix();
```

```
    glScalef (2.0, 0.4, 1.0);
```

```
    glutWireCube (1.0);
```

```
    glPopMatrix();
```

```
//A scaled cube is draw CENTERED AT THE
```

```
//NEW-NEW-NEW origin, which is the proper
```

```
//place to put the center of the forearm.
```

```
    glPopMatrix();
```

```
//Undo the damage to MV caused by drawing
```

```
//the robot.
```

```
    glutSwapBuffers();  
}
```

So, quoting from the RedBook, Ch.3:

Grand, Fixed Coordinate System: ... if you like to think in terms of a grand, fixed coordinate system—
in which matrix multiplications affect the position, orientation and scaling of your model—

you have to think of the multiplications as occurring in the opposite order from how they appear in the code.

sdc: Well, they do! New transformations are multiplied into the current transformation *from the right* by `glMultMatrix*()` and `glRotate*()`, `glTranslate*()`, etc.

Moving a Local Coordinate System: Another way to view matrix multiplication ... [is to] imagine that a local coordinate system is tied to the object you're drawing.

All operations occur relative to this changing coordinate system.

With this approach, the matrix multiplications now appear in the natural order in the code.

... This approach is what you should use for ... articulated robot arms.

(now to paraphrase:) The results and code are the same either way *you* think about it.

Helpful Tip: When you specify a transformation to be appended, such as `glTranslatef(-1.0, 0.0, 0.0);`, specify it *in terms of the CURRENT LOCAL COORDINATE SYSTEM*. Thus, in the robot, `glTranslatef(-1.0, 0.0, 0.0);` changes the origin from the old origin to $(-1.0, 0.0, 0.0)$ *relative to the old origin*.

Purpose: Set the pivot point for shoulder rotation to the base of the upper arm (left end).

`glRotatef((GLfloat) shoulder, 0.0, 0.0, 1.0);` changes the orientation of the axes so the x-axis and y-axis are rotated by **shoulder** degrees relative to the original axes.

The new x-axis of the current coordinate system is *aligned along* the upper arm.

`glTranslate((GLfloat) 1.0, 0.0, 0.0);` changes the origin from the base to the *CENTER OF THE UPPER ARM* because it is specified relative to the current coordinate system.

Reason???: `glutWireCube();` draws its cube centered at the (current) origin.

Notice the demo writer installed the scale transformation very temporarily:

```
glPushMatrix();  
glScalef(2.0, 0.4, 1.0);  
glutWireCube(1.0);  
glPopMatrix();
```

Try this! Call this function to see the local coordinate system at any point in a series of transformations. Good for debugging!

```
void coords()
{
    glBegin(GL_LINES);
        glColor3f(1.0, 0.0, 0.0); /* Red for X */
        glVertex3f(0.0, 0.0, 0.0);
        glVertex3f(1.0, 0.0, 0.0); /* Draw X-axis */
        glColor3f(0.0, 1.0, 0.0); /* Green for Y */
        glVertex3f(0.0, 0.0, 0.0);
        glVertex3f(0.0, 1.0, 0.0); /* Draw Y-axis */
        glColor3f(0.0, 0.0, 1.0); /* Blue for Z */
        glVertex3f(0.0, 0.0, 0.0);
        glVertex3f(0.0, 0.0, 1.0); /* Draw Z-axis */
    glEnd();
    glColor3f(1.0, 1.0, 1.0); /* Regular Robot Color*/
}
}
```

Very useful to know: Each local coordinate system is installed temporarily in NESTED HIERARCHICAL ORDER when a HIERARCHIAL model is drawn.

The MATRIX STACK is perfectly suited to RETURN TO a pervious coordinate system when any BRANCH of the hierarchy is finished.

For example, you can use a different coordinate system to draw each finger.

```
//The local coordinate system is for the wrist.
glPushMatrix();
... transform so origin is the base of finger 1
... Draw finger 1
glPopMatrix();
//The local coordinate system is for the wrist. (again)
glPushMatrix();
... transform so origin is the base of finger 2
... Draw finger 2
glPopMatrix();
```

The OpenGL Camera.

OpenGL defines a **FIXED STANDARD** “camera” looks toward the $-z$ (**MINUS ZEE**) axis and takes a fixed, standard picture.

Suppose you want a picture taken with a differently shaped, directed and positioned camera. If T transforms the standard GL camera into the camera you want to use, then T^{-1} is used for GL viewing.

T^{-1} transforms objects and your custom camera into objects and the OpenGL standard camera. The transformation preserves relationships between the objects and the camera so the standard picture of the transformed objects is the **SAME** as the custom camera picture of the original objects.

Using T^{-1} is like bringing somebody to a photography studio.

3d-Viewing: Chapter 7.

“Point the Camera” means transform world coordinates into **viewing coordinates**

“Choose lens, film angle, film size, etc.” means transform viewing coordinate by a **projection** onto a film plane or, for sophisticated **normalized view volume**

Typically, the normalized view volume implements **clipping**.

Perspective projections and their approximations are the first technique for making objects appear to be 3-dimensional.

Other techniques:

Depth Queuing: more distant objects have less intense color.

Identifying Visible Lines and Surfaces: and only rasterize them.

Surface Rendering: use computational models for lighting conditions and for how surfaces reflect or scatter light toward the eye.

Exploded or Cutaway Views: Typical of engineering, scientific, medical, etc. illustrations.

Display of hidden lines or surfaces behind the visible surfaces.

User controlled or animated rotation.

Stereoscopic display devices: Just calculate two images, one for each eye, each the same way as for one eye.

The viewing transformation is typically determined by

1. The **view point** aka **viewing position** aka **eye position** aka **camera position**.
2. The **view plane normal**: direction from which the eye is going to look.
3. The **up vector**: Some vector that will be transformed to into the y viewing direction.

(The projection will then transform the y direction into the UP direction on the display.)

Calculation:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{and } \mathbf{R} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

\mathbf{T} translates the view point to the origin. **IN OTHER WORDS**, transforms coordinate systems by translation so the new origin is the view point.

R rotates space so the view plane normal is aligned along the z axis, and the up vector is in the yz plane. **IN OTHER WORDS**, transform coordinate systems so the new z axis is along the view plane normal, etc.

How to get **R**? **GIVENS**:

- **N**: View plane normal.
- **V**: Up vector.

Calculate:

$$\mathbf{n} = \frac{\mathbf{N}}{|\mathbf{N}|} = (n_x, n_y, n_z)^t$$

$$\mathbf{u} = \frac{\mathbf{V} \times \mathbf{n}}{|\mathbf{V}|} = (u_x, u_y, u_z)^t$$

Note: (1) only division by $|\mathbf{V}|$ is necessary, no need to calculate length of $\mathbf{V} \times \mathbf{n}$.

(2) **FAILURE IF** user gives parallel **N** and **V**!

$$\mathbf{v} = \mathbf{n} \times \mathbf{u}$$

$$\mathbf{R}^t = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{n} \\ 0 & 0 & 1 \end{bmatrix}$$

rotates x -axis into \mathbf{u}

rotates y -axis into \mathbf{v}

rotates z -axis into *mathbf{n}*

SO, $\mathbf{R} = (\mathbf{R}^t)^{-1} = (\mathbf{R}^{-1})^{-1}$ does the job.

We must carefully check that the right-hand-rule definition of the two cross products really makes \mathbf{v} point in the right direction. That is, make $\mathbf{v} \cdot \mathbf{V} > 0$

Now let's do two simple projections:

Orthogonal projection onto the xy plane:

$$x' = x \text{ and } y' = y \text{ and } z' = 0$$

really simple, eh?

Perspective projection: Suppose the eye is at $(0, 0, 0)$ and the projection plane is given by the equation $z = z_{\text{proj}}$.

Where in the $z = z_{\text{proj}}$ plane do we draw the intersection of the **RAY** from the eye $(0, 0, 0)$ to the point (x, y, z) ?

It is easy to figure it out with similar triangles!

Do it twice; once for x and once for y .