Chapter 7

Modeling Transformations

Chapter 4 showed that meshed objects are stored in a format that includes a list of vertex positions. These vertex positions are expressed in a coordinate system, but an individual model has no knowledge of its location or orientation in a scene, nor the units used for that scene. Hence the vertex positions of a mesh model are stored in a coordinate system designed for that model, called *model coordinates.*

In model coordinates, a meshed object is typically centered around the origin, aligned along the axes, and often extends about a unit in each direction. A scene will contain many different objects, and uses *world coordinates* to create a single coordinate system for the entire scene. If we loaded all of the objects using their model coodinates, they would just pile on top of each other, as shown on p. 56.

Modeling transformations convert mesh vertices from their stored model coordinate system to the global coordinates used as a a single consistent coordinate system to combine all of the models into a scene. Let A be the current modeling transformation. Then each vertex $\mathbf{v}_i$ is stored in model coordinates, and we transform its position $A\mathbf{v}_i$ to its position, orientation and scale in the scene in world coordinates.

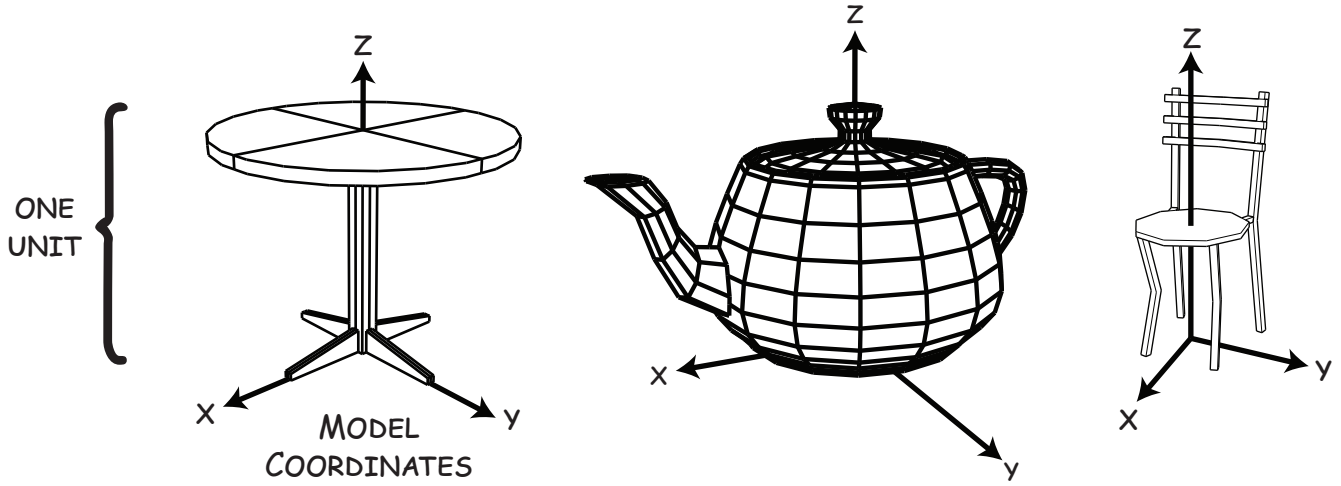
Modeling transformations are applied first in vertex pipeline, operating on model vertices in their stored coordinate system. When we write a program to build the vertex pipeline, each transformation we specify is multiplied on the right of the current transformation. Transformations specified earlier in a program are applied to vertices later, and transformations specified later are applied earlier. Modeling transformations are specified last, just before the commands that send vertices down the graphics pipeline, and so these modeling transformations act on the vertices first.

Since each model is stored in its own model coordinate system, we will need a different model transformation matrix for each model. We do not want to rebuild a vertex transformation pipeline from scratch for each new model, so we need a way to save and restore the vertex transformation pipeline so we can adjust it to properly transform each model. The command `pushmatrix` pushes a copy of the current vertex pipeline

Let's build a scene using

MODELING TRANSFORMATIONS

Each model is stored in its own coordinate system. Each of these models is centered about its own origin, extends about one unit, and is oriented along the main axes, usually extending along the z-axis.

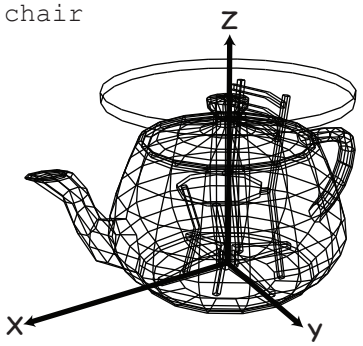


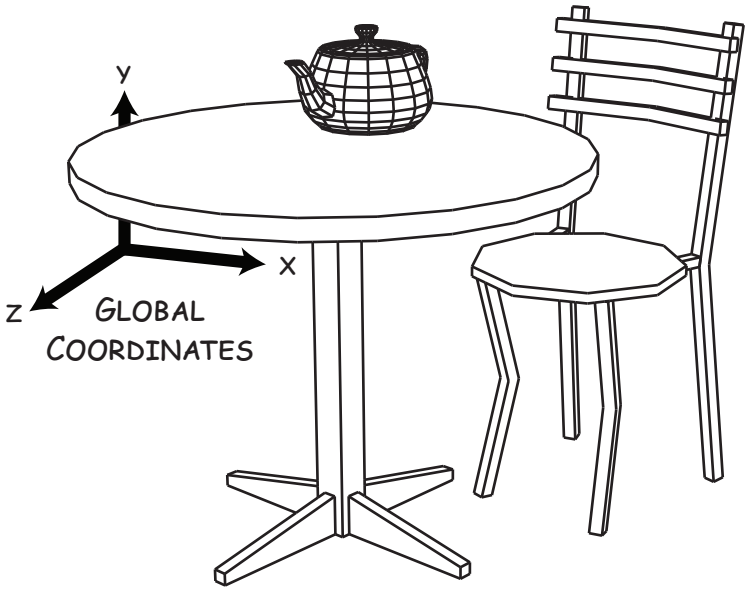
We construct a modeling transformation for feach model that describes how to translate, rotate and scale each model to be placed in a single global coordinate system used to represent the scene.

```
pushmatrix
  translate,rotate,scale
  load table
popmatrix
pushmatrix
  translate,rotate,scale
  load teapot
popmatrix
pushmatrix
  translate,rotate,scale
  load chair
popmatrix
```

If we simply loaded all three models into a single coordinate system we'd get a melange.

```
load table
load teapot
load chair
```





transformation matrix onto a matrix stack. It does not change the current vertex pipeline transformation matrix. The command `popmatrix` removes the most recently added matrix from the matrix stack and uses it to replace the current vertex pipeline transformation matrix.

The commands leading up to the modeling transformations will construct a vertex pipeline that converts from world coordinates to viewport coordinates. This pipeline will convert the position of vertices in the scene to the position of pixels on the screen. Let M be the transformation matrix that performs this scene-to-screen transformation. Say we have three models, say A, B and C, and let A, B and C be their corresponding model transformation matrices, such that $A\mathbf{v}_i$ transforms model A's vertex $\mathbf{v}_i$ from A's model coordinate system into scene coordinates. Then the following program would properly transform the three models for display.

```
<construct scene-to-screen vertex pipeline>
pushmatrix
  apply matrix A
  load model A
popmatrix
pushmatrix
  apply matrix B
  load model B
popmatrix
pushmatrix
  apply matrix C
  load model C
popmatrix
```

The first line indicates that we have constructes a current transformation matrix, we can denote it M, that converts vertices from the world coordinates of the scene to the viewport coordinates of the pixels on the screen. The first `pushmatrix` command stores a copy of M on the matrix stack. Then the next line `apply matrix A` will replace the current transformation M with the product MA, such that vertices are transformed directly from A's model coordinates to viewport coordinates. The `popmatrix` command then takes the most recent matrix from the matrix stack (M) and replaces the current transformation (MA) with it. Hence, the `popmatrix` command restores the current transformation matrix to its state before the previous `pushmatrix` command.

7.1 Change of Coordinates

There are two ways to look at these transformations. The first way to look at a transformation is by the effect it has on a shape. For example, a two-dimensional non-uniform scale by (1,2) stretches a shape by a factor of two

in the vertical direction, by doubling the y coordinates of the subseqent vertices. Shapes that appear before the transformation are not stretched whereas shapes that appear after it are. Hence the *extrinsic* effect of the transformation is to change the shape.

A second way to apply a transformation is to leave the shape unchanged, but alter the coordinate system it is embedded into. In this *intrinsic* view, the scale by factor (1,2) would create a new coordinate system with new units where the vertical y axis units was half the extent of the horizontal x axis.

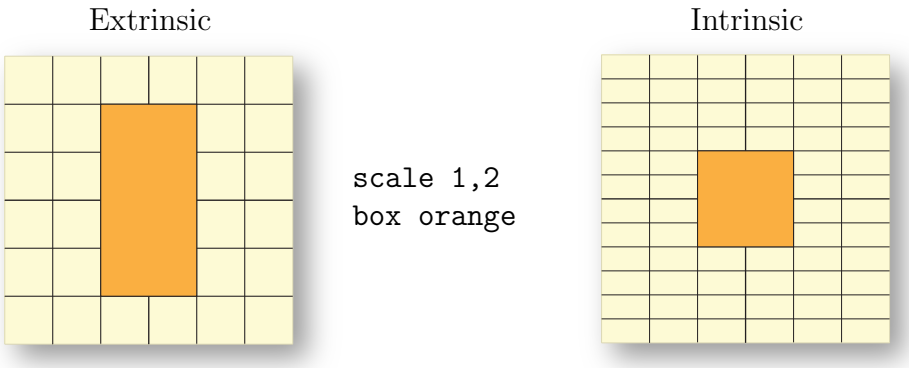


Figure 7.1: The extrinsic view of the scale command is that it transforms the shape and places the result in a global coordinate system. The intrinsic view of the scale command is that the it applies the inverse transformation to the coordinate system, and an undistorted shape is drawn into it.

Figure 7.1 demonstrates this difference by applying the scale to a box. The statement `box orange` creates a box extending from $(-1,-1)$ to $(1,1)$, filled with the color orange. The statement `scale 1,2` creates a scale transformation matrix that is applied to the vertices. The scaled box always extends from $(-2,-1)$ to $(2,1)$. In the extrinsic view, the transformation moves the vertices from their original positions to new positions in the same coordinate system. In the intrinsic view, the vertices remain in the same effective position, but the coordinate system as changed.

The intrinsic view is often called a change of coordinates. One familiar way to view a change of coordinates is as a change of units. Suppose an object is modeled in terms in imperial units, measured in inches, but is placed in a scene that uses metric units, measured in centimeters. If one wanted to read the scene in inches but display it in centimeters, then the model matrix would need to first apply a uniform scale by a factor of 2.2 (centimers per inch) to convert the coordinates from inches to centimeters. In the intrinsic view, we have just changed the coordinate system from inches to centimeters by scaling the grid by a factor of 1/2.2, whereas in the extrinsic view, we have enlarged the shape by a factor of 2.2.

As demonstrated by these examples, the transformation applied to the coordinate system (e.g. the grid) in the intrinsic view is the inverse of the transformation applied to the vertices in the extrinsic view. In computational reality, the transformation is always applied to the vertices, and the intrinsic view is just a way of mentally reasoning about the result.

7.2 Example: Building a Robot

We can demonstrate hierarchical coordinate systems and transformations by building a robot. The robot will consist of a body, an upper arm and a forearm. The upper arm will attach to the body with a shoulder joint, and the forearm will attach to the upper arm with an elbow joint. To keep this first example simple, we'll do this in two dimensions, so the body and arm segments will be rectangles, and the joint configurations will be specified by ordinary rotation angles.

First we need to create the body, as shown in plate Robot-1 on p. 60. The "box" command will create a box, which in 2-D is just a square from (-1,-1) to (1,1). We want the robot's torso to be four units tall and two units wide, so we need to stretch this box by a factor of two in the vertical direction. We implement this by setting up a scaling transformation to apply to the vertices when they are sent down the vertex pipeline. The graphics program in Robot-2 executes the `scale 1,2` command first, which multiplies a scale matrix on the right side of the current transformation, and replaces the current transformation with that product. Then the scale transformation will be applied first to the vertices, followed by the remainder of the vertex pipeline transformations.

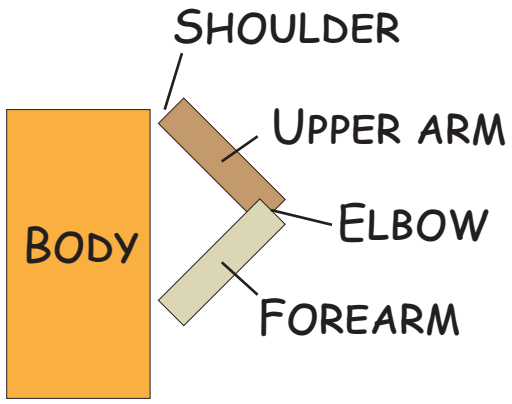
We similarly create the upper arm in Robot-3. We want an upper arm that is two units tall by half a unit wide. We will create a box from (-1,-1) to (1,1) using the `box brown` command, and will scale it to the appropriate size. We apply this scale by preceding the box command with a `scale .25 1` command which shrinks the box by a factor of one-fourth horizontally.

Before we attach the arm to the robot torso, we need to set up its shoulder rotation. Our rotate command generates a rotation matrix about the origin, so before we rotate the upper arm, we need to move the shoulder to the origin, as shown in Robot-4. Our scaled box is centered at the origin with its shoulder at $(0, 1)$. We move the shoulder to the origin with a `translate 0,-1` command, which translates the entire upper arm one unit down.

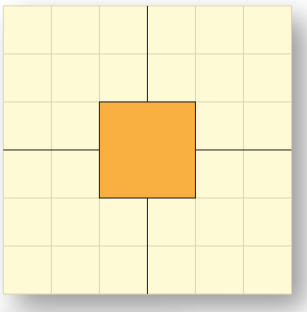
The command `rotate shoulder_angle` in Robot-5 on p. 61 now rotates the upper arm about the shoulder, because the shoulder is positioned at the origin. Notice that the rotation command appears *before* the tranlation and scale commands that shape and maneuver the upper arm into

Let's Build a Robot

(really simple double-jointed arm)



First, make a box

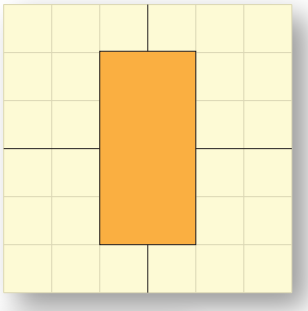


```
box orange
```

"box <color>" makes a box from (-1,-1) to (1,1) of the given color

ROBOT-1

And scale it into a body

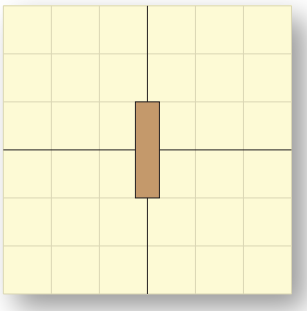


```
scale 1,2
box orange
```

Remember that we *first* set up transformations, *then* send geometry to be transformed

ROBOT-2

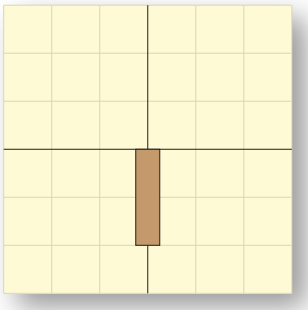
Then make an arm



```
scale 0.25,1
box brown
```

ROBOT-3

And move its shoulder to the origin



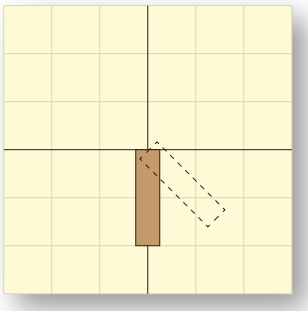
```
translate 0,-1
scale 0.25,1
box brown
```

ROBOT-4

THE *LAST* TRANSFORMATION SPECIFIED BEFORE THE GEOMETRY WILL BE THE *FIRST* APPLIED TO THE GEOMETRY:

[TRANS.][SCALE][VERTS]

SINCE THE SHOULDER IS AT THE ORIGIN A SIMPLE ROTATION IMPLEMENTS THE SHOULDER ANGLE



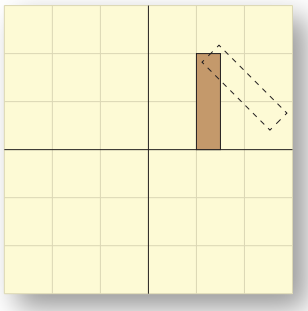
```
rotate shoulder_angle
translate 0,-1
scale 0.25,1
box brown
```

THE SHOULDER ANGLE INDICATES HOW MUCH TO ROTATE THE UPPER ARM IN A CCW DIRECTION

[ROT.][TRANS.][SCALE][VERTS]

ROBOT-5

NOW WE CAN MOVE A ROTATED ARM INTO POSITION



```
translate 1.25,2
rotate shoulder_angle
translate 0,-1
scale 0.25,1
box brown
```

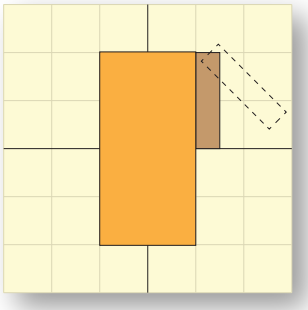
SINCE WE TRANSLATE THE UPPER ARM TO (1,2) AFTER ROTATION THIS MOVES THE CENTER OF ROTATION TO (1,2)

[T(1,2)][ROT.][T(0,-1)][S][VERTS]

T(1,2): TRANSLATES RESULT TO POSITION
ROT.: ROTATES ABOUT ORIGIN
T(0,-1): MOVES SHOULDER TO ORIGIN

ROBOT-6

AND ADD THE BODY BACK IN



```
pushmatrix
  scale 1,2
  box orange
popmatrix

translate 1,2
rotate shoulder_angle
translate 0,-1
scale 0.25,1
box brown
```

WE PUSH THE CURRENT MATRIX ONTO A STACK

AND POP IT BACK AFTER DRAWING THE BODY

SO THAT THE TRANSFORMATIONS APPLIED TO THE BODY DO NOT AFFECT THE ARM

ROBOT-7

place so its shoulder is at the origin, which means the rotation is applied *after* the shoulder has been positioned at the origin.

We can now attach the rotated upper arm to the upper right corner of the robot torso. We want to position the shoulder in the robot's coordinate system at $(1.25, 2)$ to give the half-unit wide upper arm room to rotate without overlapping the torso. Hence Robot-6 moves the upper arm using the command `translate 1.25,2` specified before the shoulder rotation command so it is appled afterwards.

We can now specify any shoulder angle and re-run the program. The new shoulder angle will change the rotation matrix, yielding a new composite matrix for the upper arm. The result will be an upper arm rectangle that rotates about its top center point located at $(1.25, 1)$, but the robot's torso is missing.

We can combine our commands to draw the torso and the upper arm, but both sets of commands change the vertex pipeline transformation matrix. For example, if we draw the body first, then the `scale 1,2` transformation would be applied to all the vertices specified after it, including the upper arm vertices. Plate Robot-7 fixes this by performing the `pushmatrix` command before we draw the body, and the `popmatrix` command after we draw the body but before we draw the upper arm. By saving a copy of the current transformation matrix before we draw the body, and restoring the transformation matrix immediately afterward, the transformations used to shape the body are forgotten and the upper arm is constructed in the original coordinate system.

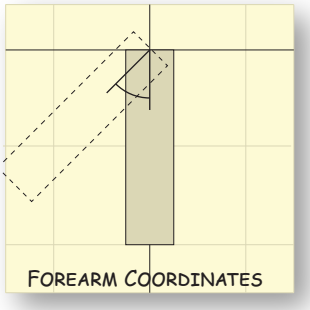
Plate Robot-8 on p. 63 constructs the gray forearm in the same shape and position as the brown upper arm, as a one-half by two unit rectangle positioned so its top-center point is at the origin. This top-center point will be the elbow, so we rotate by the elbow angle (negated since we want a clockwise rotation).

If we just move the rotated forearm to the end of the upper arm in the robot's coordinate system, then rotating the upper arm about the shoulder would separate it from the forearm. We can't automatically know where the end of the upper arm is in order to translate the forearm's elbow joint there. Instead we place the forearm at the end of the upper arm *in the upper arm's coordinate system.*

Recall that the upper arm was placed in a coordinate system where the shoulder lied at the origin and the upper arm extended two units below it. In this upper arm coordinate system shown in Robot-9, we place the forearm at the end of the upper arm by translating it two units down. This moves the forearm's elbow, which was at the origin, to the bottom of the upper arm, precisely at $(0, -2)$ in the arm coordinate system.

If we change the elbow angle, then the forearm rotates about the origin in the forearm coordinate system. But when it is translated two units down to be placed in the upper arm's coordinate system, changing the elbow

We similarly make a forearm with an elbow at the origin



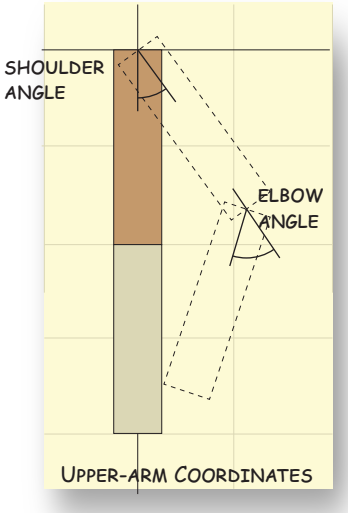

```
rotate -elbow_angle

translate 0,-1
scale 0.25,1
box gray
```

Everything below is rotated by the elbow angle

Draw the forearm

Robot-8

And move the forearm to the bottom of the arm




```
rotate shoulder_angle

pushmatrix
  translate 0,-1
  scale 0.25,1
  box brown
popmatrix

translate 0,-2

rotate -elbow_angle

translate 0,-1
scale 0.25,1
box gray
```

Everything below is rotated by the shoulder angle

Enclose the upper arm's drawing transformations in a push/popmatrix pair so they don't affect the forearm

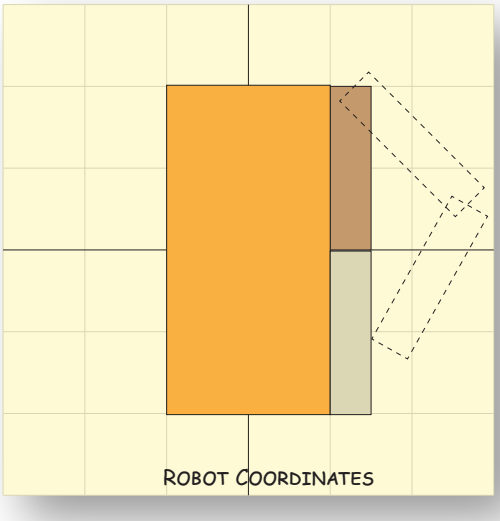
Move the elbow to the bottom of the upper arm

Everything below is rotated by the elbow angle

Draw the forearm

Robot-9

And add the body back in




```
pushmatrix
  scale 1,2
  box orange
popmatrix
translate 1.25,2
rotate shoulder_angle
pushmatrix
  translate 0,-1
  scale 0.25,1
  box brown
popmatrix
translate 0,-2
rotate -elbow_angle
translate 0,-1
scale 0.25,1
box gray
```

Robot-10

angle now causes a rotation of the forearm about the point $(0, -2)$. Note that the shoulder angle rotation continues to rotate everything specified after it, which includes the upper arm and the forearm, even if the forearm has itself been rotated about the elbow.

Note also that Robot-9 encloses the transformations that shape and position the upper arm with a `pushmatrix`, `popmatrix` pair, so they don't affect the transformations that shape and position the forearm and rotate it about the elbow.

Now the same `translate 1.25,2` command that attached the upper arm to the robot body now brings the forearm with it. Changing the shoulder angle results in a rotation of the upper arm and forearm about the shoulder point $(1.25, 2)$, whereas changing the elbow angle rotates only the forearm about a point at the end of the upper arm.

Finally, Robot-10 positions the arm coordinate system with respect to the robot coordinate system containing the body. Commands that appear before the commands in this plate affect the entire robot and operate in the robot coordinate system shown in Robot-10. Commands that appear just after the shoulder rotation affect the entire arm assembly of the robot in the upper-arm's coordinate system shown in Robot-9. Commands that appear just after the elbow rotation only affect the forearm and operate in the forearm's coordinate system shown in Robot-8. Hence these three coordinate systems: robot, upper arm and forearm, form a hierarchy.

How to Read the Robot Program

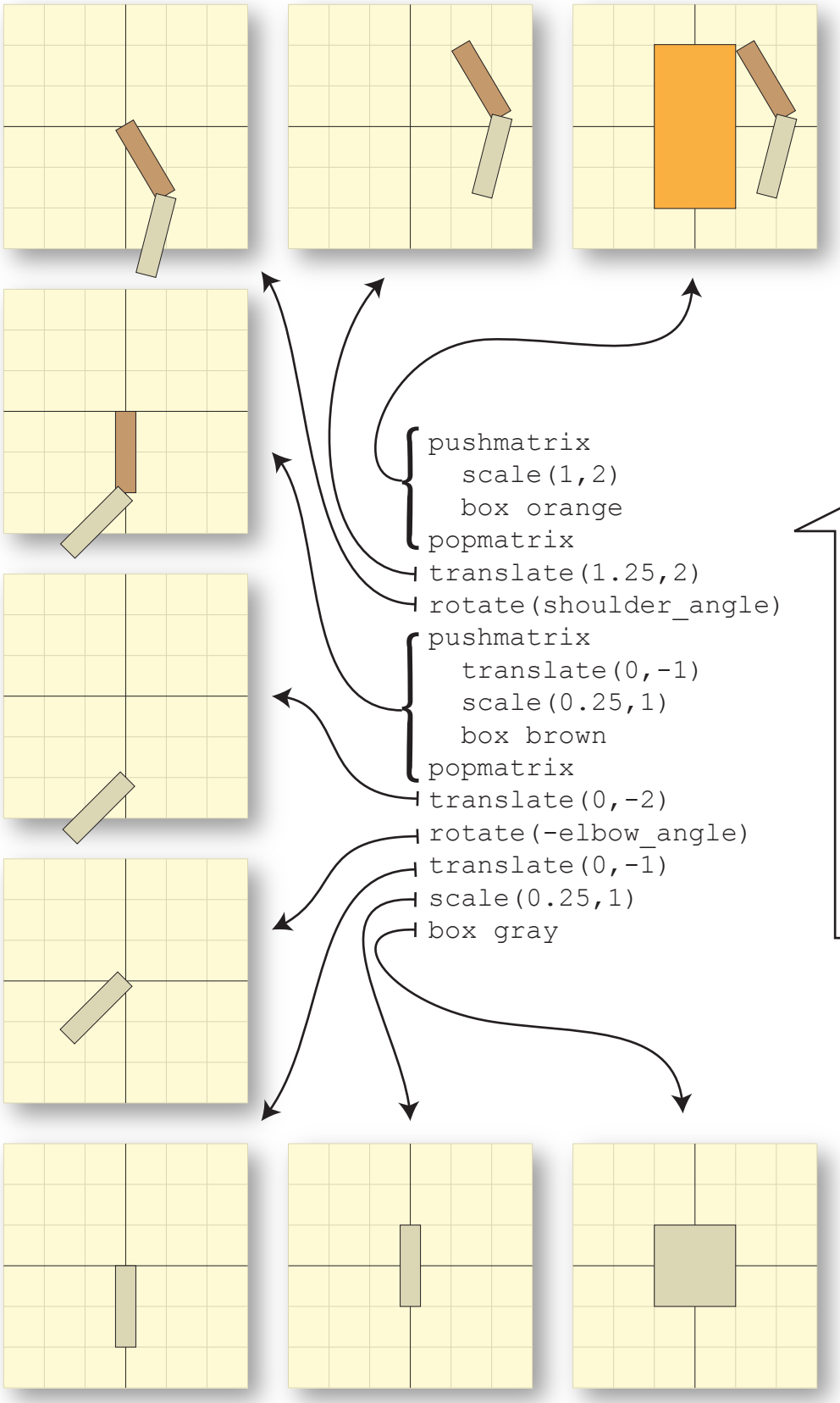
Each transformation creates a new coordinate system, and so each transformation affects the commands that come after it, up to the next `popmatrix` command, if there is one. While the computer executes a graphics program from top to bottom, it is easier to understand a graphics program if we read it from bottom to top.

Reading the robot program on p. 65 this way from bottom to top, we create a gray box scaled into a thin rectangle and moved down so its top-center elbow point lies at the origin. We then apply the elbow rotation and move the rotated forearm two units down so it can be placed at the end of the upper arm we are about to construct.

We then construct the upper arm as a box scaled into a thin rectangle moved down so its top-center shoulder point is at the origin, enclosed in a push/popmatrix pair so the shaping and positioning of the box into an upper arm don't affect the forearm. Now the shoulder rotation rotates the upper arm and the elbow rotated lower arm at its end about the origin which coincides with its shoulder point.

We translate this rotated arm assembly to the upper right corner of the robot's body we are about to build. Finally, we build the robot's body as a box stretched vertically, and surrounded by a push/popmatrix pair to prevent the stretch from affecting the construction of the arm assembly.

How to read a graphics program



```
pushmatrix
  scale(1,2)
  box orange
popmatrix
translate(1.25,2)
rotate(shoulder_angle)
pushmatrix
  translate(0,-1)
  scale(0.25,1)
  box brown
popmatrix
translate(0,-2)
rotate(-elbow_angle)
translate(0,-1)
scale(0.25,1)
box gray
```

The trick is to start at the bottom and work your way up

7.3 Example: Modeling the Earth

We can further illustrate hierarchical coordinate systemg by modeling the solar system. The earth spins on a tilted axis, orbitted by a spinning moon while it travels around the sun, which itself is orbitted by seven (or eight) other planets, several of which are orbitted by one or more spinning moons. Each of these elements can be modeled in a local coordinate system. These coordinate systems organize into a hierarchy, so the moon is dragged by the earth as it orbits the sun.

Before we start, we should understand that this is a *model* of the solar system, and so contains many simplifications. This is a hierarchical model, as opposed to a dynamical system governed by gravity (which we could model using techniques from the later motion chapters of this book). Our orbits are circular and planets and moons are spherical, all using averaged radii. Furthermore, some geometric details are left out, or as a later exercise, such as the orbiting plane and tilted axis of the moon.

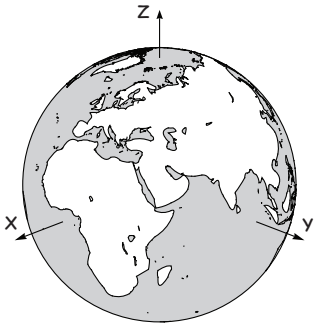


Figure 7.2: The earth local coordinate system.

We begin by modeling the earth. We start with an earth-cenetered coordinate system as shown in Fig. 7.2 and plate Earth-1, where the z-axis points north and the x-axis extends through the intersection of the equator (0° latitude) and the prime (Greenwich) meridian (0° longitude), just off the coast of Africa. This is a right-handed coordinate system so the y-axis comes out of the Indian Ocean.

We model the globe of the earth as a perfect sphere, using spherical coordinates parameterized by latitude and longitude

$$\begin{aligned} s &= \frac{\pi}{180^\circ}\text{latitude}, \\ t &= \frac{\pi}{180^\circ}\text{longitude}. \end{aligned} \tag{7.1}$$

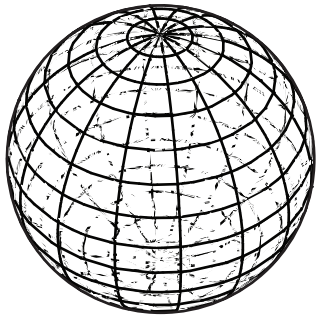


Figure 7.3: Wireframe sphere plotted using spherical coordinates (latitude and longitude).

As latitude ranges from -90° (south pole) to 90° (north pole), s ranges from $-\pi/2$ to $\pi/2$ radians, and as longitude ranges from -180° (west) to 180° (east), $t \in [-\pi, pi]$. The parameters s, t are spherical coordinates, which we convert to Euclidean x, y, z coordinates with the formula

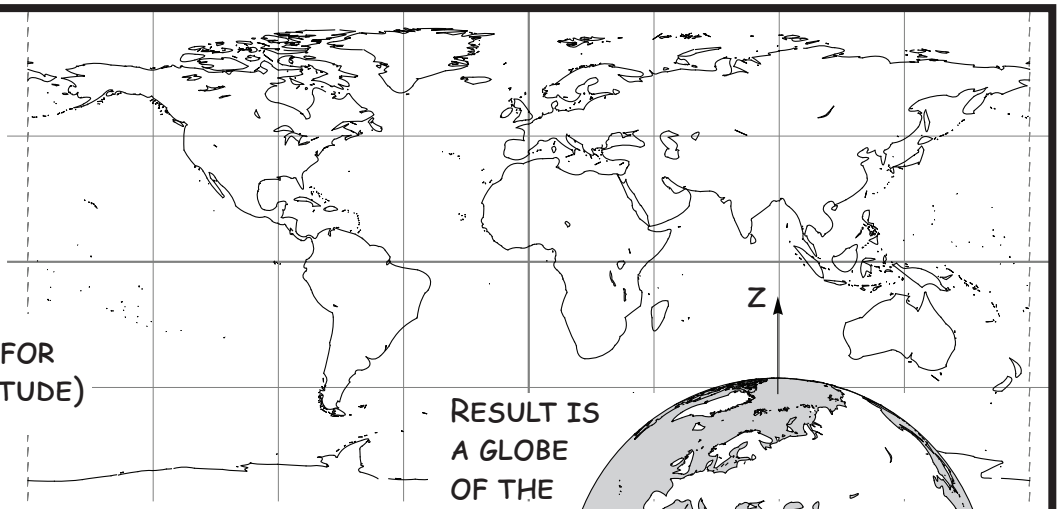
$$\begin{aligned} x &= \cos(s)\cos(t), \\ y &= \cos(s)\sin(t), \\ z &= \sin(s). \end{aligned} \tag{7.2}$$

We can make a mesh from points on this sphere by enumerating values of s and t across the globe. To sample n longitude points along a latitude, we would vary t from $-pi$ to $pi - \Delta t$ in units of $\Delta t = 2\pi/n$. We'd similarly sample m latitude points along a longitude for s varying from $-\pi/2$ to $\pi/2 - \Delta s$ in units of $\Delta s = \pi/m$. We then get the positions of four vertices of a quad by applying (7.2) to $(s, t), (s, t+\Delta t), (s+\Delta s, t+\Delta t)$ and $(s+\Delta s, t)$. (Since this quad is non-planar, it is better drawn as a triangle fan.)

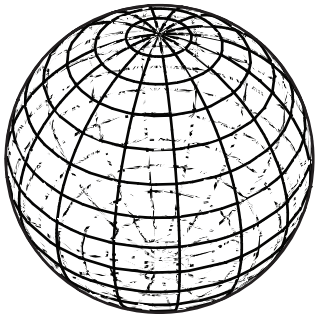
Let's Model the EARTH

EARTH-1

LOAD LATITUDE AND LONGITUDE COORDS FOR CONTINENT OUTLINES



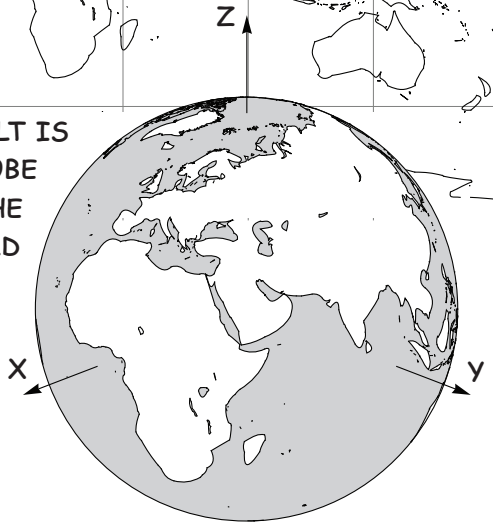
PLOT (X,Y,Z) POSITION FOR EACH (LATITUDE,LONGITUDE) POSITION ON GLOBE



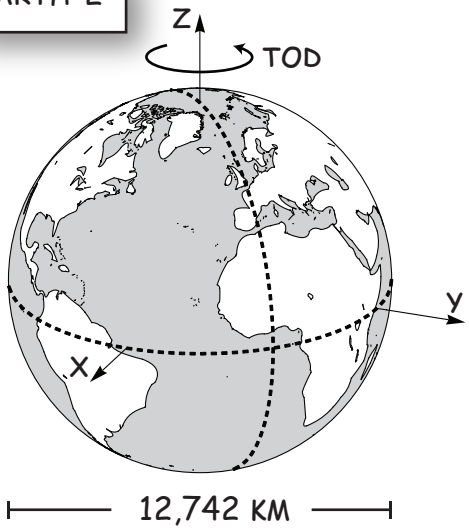
S = LATITUDE x π/180°
T = LONGITUDE x π/180°

X = COS(S) COS(T)
Y = COS(S) SIN(T)
Z = SIN(S)

RESULT IS A GLOBE OF THE WORLD



EARTH-2

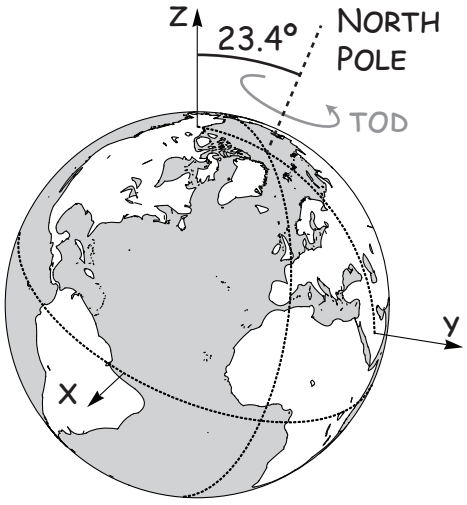


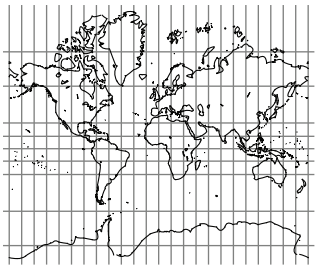

```
rotate 23.4,(1,0,0)
rotate tod,(0,0,1)
scale 6371
draw earth
```

TO ROTATE THE EARTH ABOUT ITS TILTED AXIS:

1. ROTATE THE EARTH ABOUT ITS (Z) AXIS BY "TOD," THE TIME-OF-DAY ANGLE

2. TILT THIS AXIS (AND THE ROTATED EARTH) BY 23.4° ABOUT THE X AXIS





This would be the Mercator projection commonly used for atlases if we set

$$y = \log \tan(\pi/4 + s/2), \quad (7.3)$$

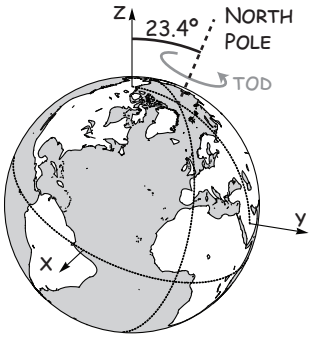
which stretches the higher latitudes vertically as shown above.

We can also draw the continents on the globe. Geographical data is available from a variety of sources. Outline data for the continents is available at:

`http://www.ngdc.noaa.gov/mgg/coast`

as a text file of latitude/longitude coordinates. Plotting the coastline data in 2-D using $x = t$ and $y = s$ produces an atlas of the earth as shown in the upper right corner of Plate Earth-1. These coordinates can be plotted as polylines or filled (concave) polygons on the globe after the coordinates are converted by (7.1) and (7.2). The result is shown in the lower right of Plate Earth-1. Subsequent plates refer to this procedure by the `draw earth` command, which creates a unit sphere and draws the continents on its surface.

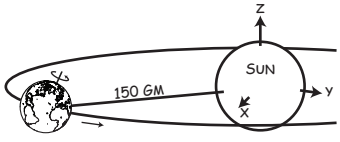
We'll assume the earth is spherical, and scale our sphere to the earth's mean diameter of 12,742 km, as shown in Plate Earth-2. Then we rotate the earth about its axis (the z-axis in our earth coordinate system) with a rotation. The angle of rotation is given as "tod," short for time-of-day. System time functions return time in seconds from a specific reference time. The earth rotates once every 23 hours, 56 minutes and 4 seconds relative to the stars. The current orientation of the earth is given by the fractional part of the current time (in seconds) divided by 86,164 seconds per rotation, multiplied by 360°, and added to the orientation of the earth at the reference time (say tod_0).



$\text{tod} = \text{frac}\left(\frac{\text{time}}{86{,}164}\right) 360° + \text{tod}_0$
$\text{tod} = \text{tod} \bmod 360°$

As we all learn in elemenary school, the earth rotates on a tilted axis, giving us seasons. We tilt the earth's axis (its z axis in the coordinate system we are using) with a rotation about its x-axis. The order of the rotations is important, since we want to spin the earth on a tilted axis, instead of spinning a tilted earth. The time-of-day rotation is applied to vertices first, rotating about the z-axis, then the result is tilted by 23.4°.

The earth rotates about the sun as it spins on its tilted axes. We'll assume it follows a circular path even though it doesn't. As illustrated in Plate Earth-3, we'll need to set up a new sun-centered coordinate system, whose origin is now in the middle of the sun at the center of the solar system. First we move the earth one astronomical unit (149,587,871 km) away from the sun. (Things in space are few and quite far between, so an actual display would want to shorten this distance so one could actually see the sun and earth.) This translation converts the coordinate system from earth-centered to sun-centered.



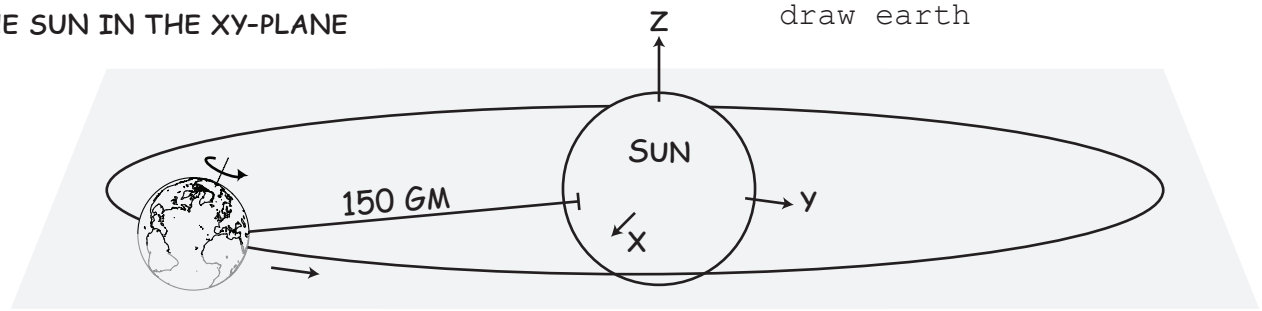
Now the earth can orbit the sun with a rotation about the sun's z-axis. The amount of this rotation is labeled "doy" for day-of-year. The earth spins and orbits in a right-handed orientation (ccw when viewed from the $+z$ direction). This orbit takes 365.256363 days which is one year plus leap year, minus a day every hundred years, plus a day every 400 years, et cetera. This comes to about 31,558,149 seconds for each orbit of the earth

EARTH-3

TO ORBIT THE SUN:

1. DRAW THE SUN AT THE ORIGIN OF SOLAR SYSTEM COORDINATES
2. MOVE THE EARTH ONE ASTRONOMICAL UNIT AWAY FROM THE SUN
3. ROTATE THE EARTH BY "DOY," THE DAY-OF-YEAR ANGLE AROUND THE SUN IN THE XY-PLANE

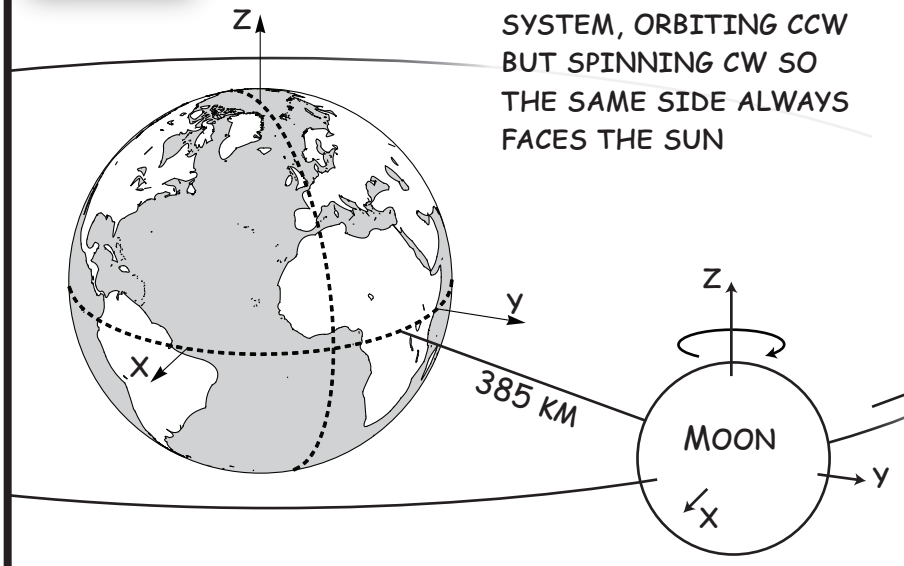
```
pushmatrix
  scale 695842
  sphere yellow
popmatrix
rotate doy,(0,0,1)
translate (149597871,0,0)
rotate 23.4,(1,0,0)
rotate tod,(0,0,1)
scale 6371
draw earth
```



(NONE OF THESE ARE ANYWHERE CLOSE TO SCALE)

EARTH-4

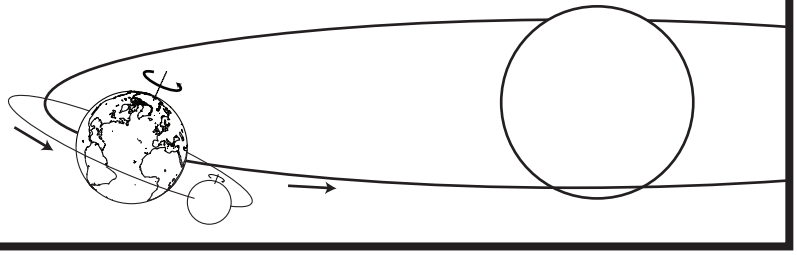
WE INSERT THE MOON IN THE EARTH'S COORD. SYSTEM, ORBITING CCW BUT SPINNING CW SO THE SAME SIDE ALWAYS FACES THE SUN




```
pushmatrix
  scale 695842
  sphere yellow
popmatrix
rotate doy,(0,0,1)
translate (149597871,0,0)
rotate 23.4,(1,0,0)
pushmatrix
  rotate 13.37*doy,(0,0,1)
  translate 385000,0,0
  rotate -13.37*doy,(0,0,1)
  scale 1737
  sphere white
popmatrix
rotate tod,(0,0,1)
scale 6371
draw earth
```

...AND TADA...

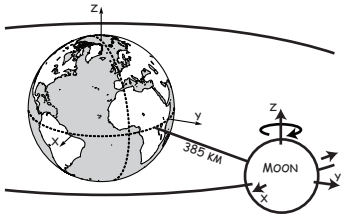
WE END UP WITH THE EARTH, SPINNING ON ITS TILTED AXIS, ORBITTED BY A TIDAL-LOCKED MOON, ALL ORBITTING THE SUN



around the sun. The current angle of the orbit of the earth around the sun is given by the fractional part of the current time in seconds divided by the seconds per earth orbit times 350° plus the angle of the earth about the sun at time zero, say tod_0, (relative to the x-axis we used to translate from earth-centered to sun-centered coordinates).

$$\text{doy} = \text{frac}\left(\frac{\text{time}}{31.6\text{M}}\right) 360° + \text{doy}_0$$
$$\text{doy} = \text{doy} \bmod 360°$$

We also need to draw the sun, shown in Plate Earth-3 as a large sphere. We create a yellow unit sphere and scale it to a diameter of 1,391,684 km, enclosed in a push/popmatrix pair so the scale doesn't affect the sun-centered coordinate system where a unit corresponds to a kilometer.



The moon orbits the earth, and follows the earth as it orbits the sun, so a hierachy of coordinate systems will come in handy. For now, assume the moon orbits the earth around the equator (it doesn't), and assume this orbit is circular (it isn't). Hence we will have a moon-cenetered coordinate system orbiting an earth-centered coordinate system orbiting a sun-centered coordinate system.

Plate Earth-4 draws a moon as a white unit sphere at the origin of the moon-centered coordinate system. This sphere is scaled to a diameter of 3,474 km and translated away from the center of the earth's coordinate system by a distance of 385,000 km along the earth's x-axis.

The moon orbits the earth roughly once a month, about once every 27.3 days, or 13.37 times per year. Hence we can simulate this orbit with a rotation about the earth's z axis by an angle of 13.37 doy. (For now we orbit the tilted earth about its equator, but the actual orbiting plane is offset by about 5° from the plane the earth follows as it orbits the sun.) The moon is tidally locked so the same side always faces the earth. Thus the moon rotates about its z axis by the opposite amount, -13.37 doy. Hence the moon orbits the earth ccw but spins cw. The moons orbit is right handed, but its spin is left-handed. (The actual axis that of the moon's spin is tilted by 6.7°, but we ignore that for now.)

The moon is drawn, scaled and spun, then moved and rotated into its current position in earth orbit, all within a push/popmatrix pair. Hence, the code for the moon is inserted without affecting the earth code or anything else. This moon code is inserted after the rotation that tilts the earth's axis. Hence the moon is inserted into the earth's untilted coordinate system and orbits the earth at its equator. Placing the code before this tilt rotation would cause the moon to orbit the earth in the plane that the earth orbits the sun.

The result is a moon orbitting an earth orbitting the sun. We use hiearchical coordinate systems and push/popmatrix commands to branch off of the hierarchy. We can broaden this hierarchy by adding other planets, and planets like Jupiter with several moons. We can deepen the hierarchy by adding a lunar orbiter, or by following the sun's motion through the galaxy and dragging the rest of the planets and their moons with it.