

Chapter 17

Visualization and Rigid Body Dynamics

©2005 by Harvey Gould, Jan Tobochnik, and Wolfgang Christian
2 April 2005

We study affine transformations in order to visualize objects in three dimensions. We then solve Euler's equation of motion for rigid body dynamics using the quaternion representation of rotations.

17.1 Two-Dimensional Transformations

Physicists frequently use transformations to convert from one system of coordinates to another. A very common transformation is an *affine transformation*, which has the ability to rotate, scale, stretch, skew, and translate an object. Such a transformation maps straight lines to straight lines. They often are represented using matrices and are manipulated using the tools of linear algebra such as matrix multiplication and matrix inversion. The Java 2D API defines a set of classes designed to create high-quality graphics using image composition, image processing, anti-aliasing, and text layout. Because linear algebra and affine transformations are used extensively in this API, we begin our study of two- and three-dimensional visualization techniques by studying this framework.

It is straightforward to rotate a point (x, y) about the origin by an angle θ or scale the distance from the origin by (s_x, s_y) using matrices:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad (17.1)$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}. \quad (17.2)$$

Unfortunately, the translation of the point (x, y) by (d_x, d_y) is treated as an addition and not as a multiplication and must be written differently:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} d_x \\ d_y \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix}. \quad (17.3)$$

This inconsistency is easily overcome if points are expressed in terms of *homogeneous coordinates* by adding a third coordinate w . Homogeneous coordinates are used extensively in computer graphics to treat all transformations in a consistent way. Instead of representing a point by a pair of numbers (x, y) , each point is represented by a triple (x, y, w) . Because two homogeneous coordinates represent the same point if one is a multiple of the other, we usually *homogenize* the point by dividing by w and write the coordinates in the form $(x, y, 1)$. (The w coordinate can be used to add perspective (see Foley et al.).) By using homogeneous coordinates, an arbitrary affine transformation can be written as:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \quad (17.4)$$

A translation, for example, can be expressed as:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \quad (17.5)$$

Exercise 17.1. Homogeneous coordinates

How are the rotation and scaling transformations expressed in matrix notation using homogeneous coordinates? Write the transformation matrices for a 30° clockwise rotation and for a scaling along the x -axis by a factor of two. Do these matrices commute?

Exercise 17.1 shows that a coordinate transformation can be broken into parts using a *block matrix* format.

$$\begin{bmatrix} \mathcal{A} & \mathbf{d}^T \\ \mathbf{0} & 1 \end{bmatrix}. \quad (17.6)$$

We will use boldface for row vectors such as $\mathbf{0}$ and $\mathbf{d}$ and calligraphic letters to represent matrices. The translation vector $\mathbf{d}$ is transposed in order to convert it to a column vector. The upper left-hand sub-matrix $\mathcal{A}$ produces rotation and scaling while the vector $\mathbf{d} = [d_x, d_y]$ produces translation.

Homogeneous coordinates have another advantage in that they can be used to distinguish between points and vectors. Unlike points, vectors should remain invariant under translation. To transform vectors using the same transformation matrices that we use to transform points, we set w to zero, thereby removing the effect of the last column. Note that the difference between two homogeneous points produces a w equal to zero. The elimination of the effect of translation makes sense because the difference between two points is a displacement vector, and vectors are defined in terms of their components, not their location.

The `AffineTransform` class in the `java.awt.geom` package defines two-dimensional affine transformations. Instances of this class are constructed as:

```
AffineTransform at = new AffineTransform(double m00, double m10, double m01, double m11,
                                         double m02, double m12);
```

Methods in this class encapsulate most of the matrix arithmetic that is required for two-dimensional visualization. For example, there are methods to calculate a transformation's inverse and to combine transformations using the rules of matrix multiplication. There also are static methods for constructing pure rotations, scalings, and translations that require only one or two parameters.

```
double theta = Math.PI/6;
AffineTransform at = AffineTransform.getRotateInstance(theta);
```

A method such as `getRotateInstance` is known as a *convenience method* because it simplifies a complicated API.

The `AffineTransform` class can transform geometric objects, images, and even text. The following code fragment shows how this class is used to rotate a point and a rectangle.

```
Point2D.Double pt = Point2D.Double(2.0,3.0);
pt = AffineTransform.getRotateInstance(Math.PI/3).transform(pt,null);
Shape shape = new Rectangle2D.Double(50,50,100,150);
shape = AffineTransform.getRotateInstance(Math.PI/3).createTransformedShape(shape);
```

The `Affine2DApp` class in Listing 17.1 allows us to visualize affine transformations by applying them to a rectangle.

Listing 17.1: The `Affine2DApp` class demonstrates how to apply an affine transformation to a two-dimensional shape.

```
package org.opensourcephysics.sip.ch17;
import java.awt.*;
import java.awt.geom.*;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.*;

public class Affine2DApp extends AbstractCalculation {
    DisplayFrame frame = new DisplayFrame("Affine Transformation");
    RectShape rect = new RectShape();
    public void calculate() {
        double[][] matrix = new double[3][]; // allocate 3 rows but not the row elements
        // set the first row of the matrix
        matrix[0] = (double[]) control.getObject("row 0"); // returned array becomes the first row
        // set the second row
        matrix[1] = (double[]) control.getObject("row 1"); // returned array becomes the second row
        // set the third row
        matrix[2] = (double[]) control.getObject("row 2"); // returned array becomes the third row
        rect.transform(matrix);
    }

    public void reset() {
        control.clearMessages();
        control.setValue("row 0", new double[]{1, 0, 0});
    }
}
```

```

        control.setValue("row 1", new double[]{0, 1, 0});
        control.setValue("row 2", new double[]{0, 0, 1});
        rect = new RectShape();
        frame.clearDrawables();
        frame.addDrawable(rect);
        calculate ();
    }

    public static void main(String[] args) {
        CalculationControl.createApp(new Affine2DApp());
    }

    class RectShape implements Drawable { // inner class
        Shape shape = new Rectangle2D.Double(50, 50, 100, 100);
        public void draw(DrawingPanel panel, Graphics g) {
            Graphics2D g2 = ((Graphics2D) g);
            g2.setPaint(Color.BLUE);
            g2.fill (shape);
            g2.setPaint(Color.RED);
            g2.draw(shape);
        }

        public void transform(double[][] mat) {
            shape = (new AffineTransform(mat[0][0], mat[1][0], mat [0][1], mat [1][1], mat [0][2], mat [1][2])).createTransformedShape(shape);
        }
    }
}

```

Exercise 17.2. Two-dimensional affine transformations

- a. Run `Affine2DApp` and describe the effect of the affine transformation:

$$\begin{bmatrix} 1 & 0.2 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \quad (17.7)$$

- b. Enter an affine transformation for a 30° clockwise rotation. About what point does the rectangle rotate? Why?
- c. Add a convenience method named `translate` to the `RectShape` class that takes two parameters (d_x, d_y) . Add a custom button to invoke this method. Test the `translate` method.
- d. Add a convenience method named `rotate` to the `RectShape` class that takes a θ parameter. Test this method.
- e. You can rotate an object about its center by first translating the object to the center of rotation, performing the rotation, and then translating the object back to its original position. Implement a method that performs a rotation about the center of the rectangle by invoking a sequence of `translate` and `rotate` methods.

- f. Affine transformations have the interesting property that transformed parallel lines remain parallel. Demonstrate that this property is plausible by transforming a rectangle using random values for the transformation matrix.

To facilitate the creation of simple geometric shapes using a drawing panel's world coordinates, the Open Source Physics library defines the `DrawableShape` and `InteractiveShape` classes in the drawing package. These classes define convenience methods to create common drawable shapes whose (x, y) location is the geometric center. The following code fragment shows how they are used.

```
// frame is any container that accepts drawables
// a circle of radius 3 centered at the origin
DrawableShape circle = InteractiveShape.createCircle(0,0,3);
frame.addDrawable(circle);
// a rectangle of width 2 and height 1 centered at (3,4)
InteractiveShape rect = InteractiveShape.createRectangle(3,4,2,1);
rect.transform(new AffineTransform(2,1,0,1,0,0));
frame.addDrawable(rect);
```

Because `DrawableShape` and `InteractiveShape` classes are written using the Java 2D API, the objects that they define are fundamentally different from the objects that use the `awt` API such as the `Circle` class.

```
// a circle with a 10 pixel radius centered at the origin
Drawable circle = new Circle(0,0,10);
frame.addDrawable(circle);
```

The Open Source Physics shape classes can be manipulated using a wide variety of linear algebra-based tools.

Exercise 17.3. Open Source Physics shape classes

Modify the `Affine2DApp` program so that it instantiates and transforms a rectangular `DrawableShape`. Test your program by repeating Exercise 17.2.

17.2 Three-Dimensional Transformations

There are a number of APIs available for three-dimensional visualizations using Java. Although Sun has developed the Java 3D package, this package is currently not included in the standard Java runtime environment. The `gl4java` and `jogl` libraries are also popular because they are based on the Open GL language. Because 3D graphics libraries might need to be installed on a client computer and are in a state of active environment and because we want a three-dimensional visualization framework designed for physics simulations, we have developed our own three-dimensional visualization framework that relies only on the standard Java API. This section describes the mathematics that forms the basis of all three-dimensional libraries.

The simplest rotation is a rotation around one of the coordinate axes with the center of rotation at the origin. This transformation can be written using a 3×3 matrix acting on the coordinates

(x, y, z) . For example, a rotation about the z axis can be written as:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \mathcal{R}_z \begin{bmatrix} x \\ y \\ z \end{bmatrix}. \quad (17.8)$$

The extension of homogeneous coordinates to three dimensions is straightforward. We add a w -coordinate to the spatial coordinates in order to create a homogenous point $(x, y, z, 1)$. This point is transformed as:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \mathcal{R}_z & \mathbf{d}^T \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}. \quad (17.9)$$

Although rotations about one of the coordinate axes are easy to derive and can be combined using the rules of linear algebra to produce an arbitrary orientation, the general case of rotation about the origin by an angle θ around an arbitrary axis $\hat{\mathbf{r}}$ can be constructed directly. The strategy is to decompose the vector $\mathbf{v}$ into components that are parallel and perpendicular to the direction $\hat{\mathbf{r}}$. The parallel part $\mathbf{v}_{\parallel}$ does not change while the perpendicular part $\mathbf{v}_{\perp}$ is a two-dimensional rotation in a plane perpendicular to $\hat{\mathbf{r}}$. The parallel part is the projection of $\hat{\mathbf{r}}$ onto $\mathbf{v}$,

$$\mathbf{v}_{\parallel} = (\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}}, \quad (17.10)$$

and the perpendicular part is what remains of $\hat{\mathbf{r}}$ after we subtract the parallel part:

$$\mathbf{v}_{\perp} = \mathbf{v} - (\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}}. \quad (17.11)$$

To calculate the rotation of $\mathbf{v}_{\perp}$, we need two perpendicular basis vectors in the plane of rotation. If we use $\mathbf{v}_{\perp}$ as the first basis vector, then we can take the cross product with $\hat{\mathbf{r}}$ to produce a vector $\mathbf{w}$ that is guaranteed to be perpendicular to $\mathbf{v}_{\perp}$ and $\hat{\mathbf{r}}$:

$$\mathbf{w} = \hat{\mathbf{r}} \times \mathbf{v}_{\perp} = \hat{\mathbf{r}} \times \mathbf{v}. \quad (17.12)$$

The rotation of $\mathbf{v}_{\perp}$ is now calculated in terms of this new basis.

$$\mathcal{R}(\mathbf{v}_{\perp}) = \cos \theta \mathbf{v}_{\perp} + \sin \theta \mathbf{w}. \quad (17.13)$$

The final result is the sum of this rotated vector and the parallel part that does not change:

$$\mathcal{R}(\mathbf{v}) = \mathcal{R}(\mathbf{v}_{\perp}) + \mathbf{v}_{\parallel} \quad (17.14a)$$

$$= \cos \theta \mathbf{v}_{\perp} + \sin \theta \mathbf{w} + \mathbf{v}_{\parallel} \quad (17.14b)$$

$$= \cos \theta [\mathbf{v} - (\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}}] + \sin \theta (\hat{\mathbf{r}} \times \mathbf{v}) + (\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}} \quad (17.14c)$$

$$= [1 - \cos \theta](\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}} + \sin \theta (\hat{\mathbf{r}} \times \mathbf{v}) + \cos \theta \mathbf{v}. \quad (17.14d)$$

Equation (17.14d) is known as the *Rodrigues formula* and provides a way of constructing rotation matrices in terms of the direction of rotation $\hat{\mathbf{r}} = (r_x, r_y, r_z)$, the cosine of the rotation

angle $c = \cos \theta$, and the sine of the rotation angle $s = \sin \theta$. If we expand the vector products in (17.14d), we obtain the matrix:

$$\mathcal{R} = \begin{bmatrix} tr_x r_x + c & tr_x r_y - sr_z & tr_x r_z + sr_y \\ tr_x r_y + sr_z & tr_y r_y + c & tr_y r_z - sr_x \\ tr_x r_z - sr_y & tr_y r_z + sr_x & tr_z r_z + c \end{bmatrix}, \quad (17.15)$$

where $t = 1 - \cos \theta$. Homogeneous coordinates are transformed using

$$\begin{bmatrix} \mathcal{R} & \mathbf{0}^T \\ \mathbf{0} & 1 \end{bmatrix}, \quad (17.16)$$

where the $\mathcal{R}$ submatrix is given in (17.15). The `Rotation3D` class constructor (see Listing 17.2) computes the rotation matrix. The `direct` method uses this matrix to transform a point. Note that the point passed into this method as an argument is copied into a temporary vector and that the point's coordinates are then changed. You will define an `inverse` method that reverses this operation in Exercise 17.5.

Listing 17.2: The `Rotation3D` class implements three-dimensional rotations using a matrix representation.

```
package org.opensourcephysics.sip.ch17;
public class Rotation3D {
    private double[][] mat = new double[4][4]; // the transformation matrix
    public Rotation3D(double theta, double[] axis) {
        double norm = Math.sqrt(axis[0]*axis[0]+axis[1]*axis[1]+axis[2]*axis[2]);
        double x = axis[0]/norm, y = axis[1]/norm, z = axis[2]/norm;
        double c = Math.cos(theta), s = Math.sin(theta);
        double t = 1-c;
        // matrix elements not listed are zero
        mat[0][0] = t*x*x+c;
        mat[0][1] = t*x*y-s*z;
        mat[0][2] = t*x*y+s*y;
        mat[1][0] = t*x*y+s*z;
        mat[1][1] = t*y*y+c;
        mat[1][2] = t*y*z-s*x;
        mat[2][0] = t*x*z-s*y;
        mat[2][1] = t*y*z+s*x;
        mat[2][2] = t*z*z+c;
        mat[3][3] = 1;
    }

    public void direct(double[] point) {
        int n = point.length;
        double[] pt = new double[n];
        System.arraycopy(point, 0, pt, 0, n);
        for(int i = 0; i < n; i++) {
            point[i] = 0;
            for(int j = 0; j < n; j++) {
                point[i] += mat[i][j]*pt[j];
            }
        }
    }
}
```

```

    }
  }
}

```

Exercise 17.4. Rodrigues formula

Show that a rotation about the z -axis is consistent with (17.14d) and (17.15). That is, define the direction of rotation to be $\hat{\mathbf{r}} = (0, 0, 1)$ and show that both formulas give the same result and that this result is consistent with a two-dimensional rotation in the xy plane. Write a short program to test the `Rotation3D` class.

Exercise 17.5. Inverse

Consult a linear algebra book for formulas to calculate the inverse of a 3×3 matrix and define an `inverse` method that multiplies a given point by the inverse of the rotation matrix. Although it is slow and prone to round-off errors, Cramer's rule works for a simple rotation matrix. It should not, however, be used for serious computations. Write a short program to test the `inverse` method. Show that the original vector is recovered if the `inverse` and `direct` methods are applied in succession.

A projection transforms an object in a coordinate system of dimension d into another object in a coordinate system less than d . The simplest projection is an *orthographic parallel projection*, which maps an object onto a plane perpendicular to a coordinate axis. For example, if we choose to project along the z -axis, the point (x, y, z) is mapped to the point (x, y) by dropping the third coordinate. A line is projected by projecting the end points and then connecting the projected values. A sphere with radius R is displayed by projecting the center and drawing a circle with the sphere's radius. Listing 17.3 uses these simple projections to visualize the structure of the methane CH_4 molecule.

Listing 17.3: The `Methane` class implements a visualization of the methane molecule CH_4 .

```

package org.opensourcephysics.sip.ch17;
import java.awt.*;
import org.opensourcephysics.display.*;

public class Methane implements Drawable {
    static final double cos30 = Math.cos(Math.PI/6); //cosine of 30 degrees
    static final double sin30 = Math.sin(Math.PI/6); //sine of 30 degrees
    static final double h = Math.sqrt(1.0-4.0*cos30*cos30/9.0); // trapezoid height
    double[][] atoms = new double[5][];
    Circle circle = new Circle();
    public Methane() {
        // atom locations in 3D homogeneous coordinates
        atoms[0] = new double[]{0, 0, 0, 1}; // C atom at origin
        atoms[1] = new double[]{0, 0, 0.75*h, 1}; // H atom on z axis
        atoms[2] = new double[]{2.0*cos30/3.0, 0, -0.25*h, 1}; // H atom
        atoms[3] = new double[]{-cos30/3.0, sin30, -0.25*h, 1}; // H atom
        atoms[4] = new double[]{-cos30/3.0, -sin30, -0.25*h, 1}; // H atom
    }
}

```

```

    }

    void transform(Rotation3D t) {
        for(int i = 0, n = atoms.length; i < n; i++) {
            t.direct(atoms[i]);
        }
    }

    public void draw(DrawingPanel panel, Graphics g) {
        g.setColor(Color.black);
        int x0 = panel.xToPix(0);
        int y0 = panel.yToPix(0);
        for(int i = 0, n = atoms.length; i < n; i++) {
            int xpix = panel.xToPix(atoms[i][0]);
            int ypix = panel.yToPix(atoms[i][1]);
            g.drawLine(x0, y0, xpix, ypix);
        }
        for(int i = 0, n = atoms.length; i < n; i++) {
            circle.setXY(atoms[i][0], atoms[i][1]);
            circle.draw(panel, g);
        }
    }
}

```

The carbon and hydrogen atom coordinates are given in the **Methane** constructor. These coordinates are stored in a multidimensional array so that we can loop over the atoms in the **draw** method. The carbon atom (the center of symmetry) is placed at the origin and a hydrogen atom is placed above it along the z axis. The three remaining hydrogen atoms are placed so as to form a tetrahedron with a H-H separation equal to unity. This orientation can be changed using the **transform** method to rotate the coordinates. Note that the **draw** method draws lines from the origin to each hydrogen atom's coordinates and then draws circles at these coordinates.

Exercise 17.6. Methane

Determine the angle between two hydrogen bonds using the data in the **Methane** class.

The **MethaneApp** program uses the **Rotation3D** and **Methane** classes by rotating the methane molecule about the origin in response to mouse actions. The **handleMouseAction** method stores the current mouse position when the mouse is pressed. A **Rotation3D** object is created when the mouse is dragged and the drag distance determines the angle of rotation. If the mouse is dragged vertically, the molecule is rotated about the y axis, and if the mouse is dragged horizontally, the molecule is rotated about the z axis. The **direct** transform is then applied to every atom in the methane molecule.

Listing 17.4: The **MethaneApp** class instantiates a **Methane** object and rotates this object using mouse actions.

```

package org.opensourcephysics.sip.ch17;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;

```

```

import org.opensourcephysics.frames.*;
import org.opensourcephysics.display.InteractiveMouseHandler;
import org.opensourcephysics.display.InteractivePanel;

public class MethaneApp implements InteractiveMouseHandler {
    DisplayFrame frame = new DisplayFrame("Methane");
    Methane methane = new Methane();
    double mousex = 0, mousey = 0;
    public MethaneApp() {
        frame.addDrawable(methane);
        frame.setPreferredMinMax(-1, 1, -1, 1);
        frame.setInteractiveMouseHandler(this);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public void handleMouseAction(InteractivePanel panel, MouseEvent evt) {
        switch(panel.getMouseAction()) {
            case InteractivePanel.MOUSE_DRAGGED :
                double dx = panel.getMouseX()-mousex;
                double dy = panel.getMouseY()-mousey;
                Rotation3D rotation = new Rotation3D(Math.sqrt(dx*dx+dy*dy), new double[] {dy,
                                                                                               0,
                                                                                               dx});

                methane.transform(rotation);
                mousex += dx;
                mousey += dy;
                panel.repaint();
                break;
            case InteractivePanel.MOUSE_PRESSED :
                mousex = panel.getMouseX();
                mousey = panel.getMouseY();
                break;
        }
    }

    public static void main(String[] args) {
        new MethaneApp();
    }
}

```

Exercise 17.7. Methane rotation

Modify the methane program by replacing the **direct** transformation with the **inverse** transformation. Run and compare this program to the original program. How did this change affect the mouse actions? Why?

Exercise 17.8. Hidden surface removal

Modify the **Methane** class so that the carbon atom is drawn as a green circle and run the **MethaneApp** program. Note that the carbon atom is hidden by hydrogen atoms with negative z values due to

the incorrect drawing order. Recode the `Methane` class so that the drawing order is determined by the atom's z coordinate. Ordering along the line of sight often is used in graphics programs for hidden line and hidden surface removal.

17.3 Display EJS

The `dispayejs` package contains a three-dimensional drawing framework that makes it easy to create simple visualizations. In the spirit of object oriented programming, we will not study the EJS implementation in detail, and will describe only its key concepts. The `BoxEJS` class creates a simple three-dimensional visualization. Run the program and drag the mouse within the panel. We need not concern ourselves with details such as hidden line removal, perspective, or rotation. EJS hides these and many other details from the user.

Listing 17.5: The `BoxEJSApp` class creates a box within an EJS drawing panel.

```
package org.opensourcephysics.sip.ch17;
import java.awt.*;
import javax.swing.*;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.displayejs.*;

public class BoxEJSApp {
    public static void main(String[] args) {
        // create a drawing frame and a drawing panel
        EJSFrame frame = new EJSFrame("EJS Demo");
        frame.setPreferredMinMax(-10, 10, -10, 10, -10, 10);
        frame.setDecorationType(DrawingPanel3D.DECORATION_AXES);
        InteractiveElement block = new InteractiveBox();
        block.setXYZ(0, 0, 0);
        block.setSizeXYZ(6, 6, 3);
        block.getStyle().setFillPattern(Color.RED);
        block.setResolution(Resolution.createDivisions(1.0)); // divide the block in subblocks
        frame.addDrawable(block);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

The `EJSFrame` class behaves very much like its two-dimensional counterpart `DisplayFrame`. Some methods, such as `setPreferredMinMax` have been extended by adding parameters for the third dimension. New methods such as `setDisplayMode` and `setDecorationType` enable the programmer to control the three-dimensional viewing perspective and axis types, respectively. Three-dimensional interactive elements, such as `InteractiveBox`, `InteractiveCylinder`, and `InteractiveCircle`, have position, size, and style properties that can be set. Just as in the two-dimensional case, these objects are added to a display panel. The program must repaint this panel if either the panel's or the drawable object's properties change.

Drawable objects in the `displayejs` package implement the `InteractiveElement` interface. The `BoxEJSApp` program demonstrates that this interface provides an API for objects that have a

position and size and accept `Style` and `Resolution` objects to control their visual representation. Interactive elements also can generate interaction events and can have one or more interaction targets. Interaction events are action events that are generated within an object when the object is accessed using a mouse. Interaction targets receive and process these events. Listing 17.6 creates a particle and an arrow and responds to interaction events when these objects are moved. You can move an interactive element by dragging it with the mouse. You may generate interaction events at an arbitrary location within the three-dimensional viewing space, but you must first press the `<alt>` (option on Mac OS X) key to disable the viewing space rotation. First pressing the `<shift>` key enables you to zoom in and out.

Listing 17.6: The `InteractionEJSApp` class demonstrates how to respond to interaction events.

```
package org.opensourcephysics.sip.ch17;
import javax.swing.*;
import org.opensourcephysics.displayjs.*;
import org.opensourcephysics.frames.*;

public class InteractionEJSApp implements InteractionListener {
    InteractionEJSApp() {
        EJSFrame frame = new EJSFrame("EJS Interactions");
        frame.setPreferredMinMax(-0.25, 0.25, -0.25, 0.25, -0.25, 0.25);
        frame.addListener(this); // send interactions FROM the panel to this app
        InteractiveParticle particle = new InteractiveParticle();
        particle.setEnabled(true); // enables interactions such as dragging
        particle.addListener(this); // sends interactions FROM the particle to this app
        frame.addDrawable(particle); // adds the particle to the panel
        InteractiveArrow arrow = new InteractiveArrow();
        arrow.setEnabled(true); // enables interactions such as dragging
        arrow.addListener(this); // sends interactions FROM the arrow to this app
        frame.addDrawable(arrow); // adds the arrow to the panel
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

    public void interactionPerformed(InteractionEvent _evt) {
        if(_evt.getID() != InteractionEvent.MOUSE_PRESSED) {
            return;
        }
        if(_evt.getSource() instanceof InteractiveParticle) {
            System.out.println("A particle has been hit");
        } else if(_evt.getSource() instanceof InteractiveArrow) {
            if(_evt.getTarget().getClass() == InteractionTargetElementSize.class) {
                System.out.println("An arrow's head has been hit");
            } else {
                System.out.println("An arrow's tail has been hit");
            }
        }
    }
}

static public void main(String args[]) {
```

```

        new InteractionEJSApp();
    }
}

```

Interactive elements can be grouped together and manipulated as a single object by creating geometric shapes such as spheres, boxes, and arrows and adding them to a **GroupDrawable**. The easiest way to do this is to subclass **GroupDrawable** and instantiate the shapes in the constructor. Listing 17.7 shows an example.

Listing 17.7: The **BarbellEJS** creates a compound object by instantiating simpler shapes and adding them to a **GroupDrawable**.

```

package org.opensourcephysics.sip.ch17;
import org.opensourcephysics.displayejs.*;

public class BarbellEJS extends GroupDrawable {
    public BarbellEJS() {
        InteractiveCylinderSimple bar = new InteractiveCylinderSimple();
        bar.setXYZ(0, 0, 5);
        bar.setSizeXYZ(0, 0, 10);
        bar.setRadius(0.2);
        add(bar);
        InteractiveElement sphere = new InteractiveSphere();
        sphere.setXYZ(0, 0, -5);
        sphere.setSizeXYZ(4, 4, 4);
        add(sphere);
        sphere = new InteractiveSphere();
        sphere.setXYZ(0, 0, 5);
        sphere.setSizeXYZ(4, 4, 4);
        add(sphere);
    }
}

```

Exercise 17.9. Group drawable

Write a small test program that instantiates and displays a **Barbell**. Describe the change in rendering while dragging within the view. Why does the EJS author change the rendering?

The code package for this chapter includes an EJS version of the methane molecule. It is not listed here because of its length. As in the previous example, an EJS **GroupDrawable** is used to define a **Methane** class. This model is instantiated and added to a **EJSFrame** in the **MethaneEJSApp** class.

17.4 Dynamics

The dynamical behavior of a rigid body is determined by

$$\frac{d\mathbf{P}}{dt} = \mathbf{F} \quad (17.17a)$$

$$\frac{d\mathbf{L}}{dt} = \mathbf{N}, \quad (17.17b)$$

where the rate of change of the total linear momentum $\mathbf{P}$ and total angular momentum $\mathbf{L}$ about a point O is determined by the total force $\mathbf{F}$ on the body and the total torque $\mathbf{N}$ about O . These momenta are expressed in terms of the translational $\mathbf{V}$ and rotational velocity $\boldsymbol{\omega}$ by

$$\mathbf{P} = M\mathbf{V} \quad (17.18a)$$

$$\mathbf{L} = \mathcal{I}\boldsymbol{\omega}, \quad (17.18b)$$

where M is the mass and $\mathcal{I}$ is the moment of inertia tensor. For an unconstrained body, the point O is usually taken to be the center of mass. For a constrained body, such as a spinning top, the point O is usually taken to be the point of support.

Although the translational and rotational equations of motion appear similar, the fact that the inertia tensor is not always constant with respect to axes fixed in space complicates the analysis. To use a constant inertia tensor, we must describe the motion using a non-inertial reference frame that is fixed in the body. Because we are free to orient the axes within the body, we choose axes in which the moment of inertia tensor is diagonal. These axes are referred to as the body's principal axes and are easy to determine for symmetrical objects. The diagonal elements of the moment of inertia tensor are calculated using the volume integrals

$$I_1 = \int_V \rho(y^2 + z^2) dV \quad (17.19a)$$

$$I_2 = \int_V \rho(z^2 + x^2) dV \quad (17.19b)$$

$$I_3 = \int_V \rho(x^2 + y^2) dV, \quad (17.19c)$$

where ρ is the mass density. The off-diagonal elements are zero.

Given that the general relation between the derivative in the space frame to the derivative in the body frame is

$$\left(\frac{d\mathbf{L}}{dt}\right)_{\text{space}} = \left(\frac{d\mathbf{L}}{dt}\right)_{\text{body}} + \boldsymbol{\omega} \times \mathbf{L}, \quad (17.20)$$

it is easy to show that the rotational equation of motion in the body frame can be written as:

$$\frac{d\mathbf{L}}{dt} + \boldsymbol{\omega} \times (\mathcal{I}\boldsymbol{\omega}) = \mathbf{N}. \quad (17.21)$$

Equation (17.21) is Euler's equation for the motion of a rigid body. It may be written in component form as

$$I_1 \dot{\omega}_1 + (I_3 - I_2) \omega_3 \omega_2 = N_1 \quad (17.22a)$$

$$I_2 \dot{\omega}_2 + (I_1 - I_3) \omega_1 \omega_3 = N_2 \quad (17.22b)$$

$$I_3 \dot{\omega}_3 + (I_2 - I_1) \omega_2 \omega_1 = N_3. \quad (17.22c)$$

A complete description of a rigid body in space requires three angular orientation variables as well as three angular velocity variables. The orientation often is given in terms of the Euler angles ψ , θ , and ϕ . These angles specify the orientation as a series of three independent rotations about pre-chosen axes and must be applied in exactly the order given because the rotation matrices do not commute. To use these angles as differential equation state variables, we must be able to calculate their rate as a function of the body-frame angular velocity in (17.22). The expression for the angular velocity in the body frame in terms of the Euler angles is (see Goldstein)

$$\begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} = \begin{bmatrix} \sin \theta \sin \psi & \cos \psi & 0 \\ \sin \theta \cos \psi & -\sin \psi & 0 \\ \cos \theta & 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}. \quad (17.23)$$

Unfortunately, the matrix in (17.23) is singular when $\sin \theta = 0$. Because we must invert the above matrix to solve for the rate of the Euler angles, the differential equations become unstable whenever θ approaches 0 or π . A better approach for numerical computation is to abandon Euler angles and to use *quaternions*.

17.5 Quaternion Arithmetic

The rotation of an arbitrary two-dimensional vector $\mathbf{A} = a_x \hat{x} + a_y \hat{y}$ by an angle θ can be reformulated using complex numbers as

$$a' = ae^{i\theta} = e^{i\theta/2} a e^{i\theta/2}, \quad (17.24)$$

where the vector $\mathbf{A}$ is expressed as a complex number $a = a_x + ia_y$. The real and imaginary components corresponding to the vector components. This idea can be extended to three dimensions using quaternions. A quaternion can be represented in terms of real and *hyper-complex* numbers i , j , and k as

$$\hat{q} = q_0 + iq_1 + jq_2 + kq_3 = (q_0, q_1, q_2, q_3), \quad (17.25)$$

where the hyper-complex numbers obey Hamilton's rules

$$i^2 = j^2 = k^2 = ijk = -1. \quad (17.26)$$

Like imaginary numbers, the quaternion conjugate is defined as

$$\hat{q}^* = q_0 - iq_1 - jq_2 - kq_3 = (q_0, -q_1, -q_2, -q_3). \quad (17.27)$$

Unlike imaginary numbers, quaternion multiplication does not commute and obeys the rules:

$$ij = k, \quad jk = i, \quad ki = j, \quad ji = -k, \quad kj = -i, \quad ik = -j. \quad (17.28)$$

Although it would be a mistake to identify hyper-complex numbers with unit vectors in three-dimensional space (just as it would be a mistake to identify the imaginary number i with the y direction in a two-dimensional space), it is convenient to think of a quaternion as the sum of a scalar q_0 and a vector $\mathbf{q}$

$$\hat{q} = q_0 + \mathbf{q} = (q_0, \mathbf{q}). \quad (17.29)$$

The quaternion is said to be *pure* if the scalar part is zero.

By using the above definitions, the product of two quaternions $\hat{p}$ and $\hat{q}$ can be shown to be

$$\hat{p}\hat{q} = (q_0, \mathbf{q})(p_0, \mathbf{p}) = q_0p_0 - \mathbf{q} \cdot \mathbf{p} + q_0\mathbf{p} + p_0\mathbf{q} + \mathbf{p} \times \mathbf{q}. \quad (17.30)$$

Note that except for the additional cross product term, quaternion multiplication is similar to complex multiplication, $(a_0, a_1)(b_0, b_1) = (a_0b_0 - a_1b_1, a_1b_0 + b_1a_0)$. The norm (length) of a quaternion is defined to be $|\hat{q}\hat{q}^*| = q_0q_0 + q_1q_1 + q_2q_2 + q_3q_3$.

Exercise 17.10. Quaternion multiplication

Show that Hamilton's rules for hyper-complex numbers in (17.26) lead to (17.30).

Euler has shown that the most general change of a rigid body with one point fixed is a rotation about a fixed axis. Quaternions provide an elegant representation of both the rotation angle and the axis orientation. It can be shown (see Shoemaker) that a rotation through an angle θ about an axis with direction cosines (u_1, u_2, u_3) can be represented as a unit quaternion with components

$$\hat{q} = \cos \frac{\theta}{2} + (iu_1 + ju_2 + ku_3) \sin \frac{\theta}{2}. \quad (17.31)$$

To stress the analogy with the complex exponential $e^{i\theta} = \cos \theta + i \sin \theta$, some textbooks go so far as to represent this rotation through θ about the axes $\mathbf{u}$ using exponential notation $\hat{q} = e^{\mathbf{u}\theta/2}$.

Given a unit quaternion $\hat{q}$ that represents a rotation, how do we apply this rotation to an arbitrary vector $\mathbf{A}$? If we define a pure quaternion $\hat{a} = (0, a_x, a_y, a_z)$, it can be shown that

$$\hat{a}' = \hat{q}\hat{a}\hat{q}^*, \quad (17.32)$$

where the resulting quaternion $\hat{a}' = (0, a'_x, a'_y, a'_z)$ contains the components of the rotated vector $\mathbf{A}'$ (see Rapaport). Note the similarity to (17.24) if the quaternion is represented using exponential notation.

Exercise 17.11. Quaternion rotation

- Use the properties of the direction cosines to show that (17.31) defines a quaternion of unit length.
- Show that the length of the vector $\mathbf{A}$ does not change when using (17.32).

Bodies can be oriented using any representation of a rotation including quaternions, Euler angles, and rotation matrices. Because the quaternion representation does not involve trigonometric functions, it is very efficient for computing rigid body dynamics, and we prefer this representation. To make it easy to create other representations, EJS defines a **Transformation** interface. This interface is defined in the EJS display package and a concrete implementation based on rotation matrices is described in Appendix A. A quaternion implementation is similar and is available in EJS. Listing 17.8 shows how the quaternion implementation is used to rotate a **BoxWithArrows**. Because **BoxWithArrows** is similar to Listing 17.7 it is not shown here.

Listing 17.8: A class that tests the quaternion representation of rotations.

```
package org.opensourcephysics.sip.ch17;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.displayjs.*;
import org.opensourcephysics.displayjs.utils.*;
import org.opensourcephysics.frames.*;

public class QuaternionApp extends AbstractCalculation {
    EJSFrame frame = new EJSFrame("Quaternion Rotations");
    QuaternionRotation transformation = new QuaternionRotation(1, 0, 0, 0);
    BoxWithArrows box = new BoxWithArrows();
    public QuaternionApp() {
        frame.setDecorationType(DrawingPanel3D.DECORATION_AXES);
        frame.setPreferredMinMax(-6, 6, -6, 6, -6, 6);
        box.setTransformation(transformation);
        frame.addDrawable(box);
        frame.setVisible(true);
    }

    public void calculate() {
        double q0 = control.getDouble("q0");
        double q1 = control.getDouble("q1");
        double q2 = control.getDouble("q2");
        double q3 = control.getDouble("q3");
        transformation.setCoordinates(q0, q1, q2, q3);
        box.setTransformation(transformation);
    }

    public void reset() {
        control.clearMessages();
        control.setValue("q0", 1);
        control.setValue("q1", 0); // initial orientation is along x axis
        control.setValue("q2", 0);
        control.setValue("q3", 0);
        calculate();
    }

    public static void main(String[] args) {
        CalculationControl.createApp(new QuaternionApp());
    }
}
```

}

Exercise 17.12. Quaternion representation of rotations

- Compile and run the `QuaternionApp` target class. Find and then test a quaternion that orients the long side of the box at 45° in the xy plane. Repeat for the zy plane.
- Use a quaternion to orient the long axis of the box along an axis in the $(1, 2, 1)$ direction.
- What happens if the quaternion does not have unit norm?

17.6 Quaternion equations of motion

Because we often will transform torque and other vectors to and from the rotating object's body frame, we require the transformation matrix from the space frame to the body frame using quaternions. The derivation is straightforward. We represent a rotation using a unit quaternion (q_0, q_1, q_2, q_3) , carry out (17.32) using (17.30), and express the result in matrix form. The resulting rotation matrix is

$$\mathcal{R} = 2 \begin{bmatrix} \frac{1}{2} - q_2^2 - q_3^2 & q_1 q_2 + q_0 q_3 & q_1 q_3 - q_0 q_2 \\ q_1 q_2 - q_0 q_3 & \frac{1}{2} - q_1^2 - q_3^2 & q_2 q_3 + q_0 q_1 \\ q_1 q_3 + q_0 q_2 & q_2 q_3 - q_0 q_1 & \frac{1}{2} - q_1^2 - q_2^2 \end{bmatrix}. \quad (17.33)$$

This matrix is equal to the rotation matrix derived from the Rodrigues formula (17.14d).

The angular velocity in the body frame can be written as $\dot{\hat{q}}(t) = \frac{1}{2}\hat{\omega}(t)\hat{q}(t)$, where $\hat{\omega}$ is a pure quaternion $(0, \omega_1, \omega_2, \omega_3)$. The derivation is as follows. The time dependence of a vector $\mathbf{r}$ can be expressed as a transformation of its initial value $\mathbf{r}_0$ as

$$\hat{\mathbf{r}}(t) = \hat{q}(t)\hat{\mathbf{r}}_0\hat{q}^*(t). \quad (17.34)$$

If we differentiate $\hat{\mathbf{r}}(t)$ with respect to time, we have

$$\dot{\hat{\mathbf{r}}} = \dot{\hat{q}}\hat{\mathbf{r}}_0\hat{q}^* + \hat{q}\hat{\mathbf{r}}_0\dot{\hat{q}}^*, \quad (17.35)$$

where we have dropped the explicit time dependence of $\hat{q}$. We then substitute $\hat{\mathbf{r}}_0 = \hat{q}^*\hat{\mathbf{r}}\hat{q}$ and obtain

$$\dot{\hat{\mathbf{r}}} = \dot{\hat{q}}\hat{q}^*\hat{\mathbf{r}} + \hat{\mathbf{r}}\dot{\hat{q}}\hat{q}^* = \dot{\hat{q}}\hat{q}^*\hat{\mathbf{r}} - \hat{\mathbf{r}}\dot{\hat{q}}\hat{q}^*, \quad (17.36)$$

where we have used the fact $\hat{q}\hat{q}^* = 1$. The only part of the $\hat{\mathbf{r}}$ and $\dot{\hat{q}}\hat{q}^*$ product that does not commute is the vector cross product. The scalar part commutes and is zero. If we denote the pure (vector) part of the quaternion $\dot{\hat{q}}\hat{q}^*$ by $\mathbf{u}$, we find

$$\dot{\hat{\mathbf{r}}} = \mathbf{u} \times \mathbf{r} - \mathbf{r} \times \mathbf{u} = 2\mathbf{u} \times \mathbf{r}. \quad (17.37)$$

Because $\dot{\mathbf{r}} = \boldsymbol{\omega} \times \mathbf{r}$ for rotational motion, we obtain

$$\boldsymbol{\omega} = 2\mathbf{u}. \quad (17.38)$$

This result can be expressed using components by expressing $\mathbf{u}$ as

$$\begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = \frac{1}{2} \mathcal{Q}^T \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ 0 \end{bmatrix}, \quad (17.39)$$

or

$$\begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ 0 \end{bmatrix} = 2\mathcal{Q} \begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix}, \quad (17.40)$$

where

$$\mathcal{Q} = \begin{bmatrix} -q_1 & q_0 & q_3 & -q_2 \\ -q_2 & -q_3 & q_0 & q_1 \\ -q_3 & q_2 & -q_1 & q_0 \\ q_0 & q_1 & q_2 & q_3 \end{bmatrix}. \quad (17.41)$$

Note that because $\mathcal{Q}$ is orthogonal, $\mathcal{Q}^T \mathcal{Q} = \mathbf{1}$.

Finally, Euler's equation of motion (17.22) must be expressed using quaternion acceleration. If we start with

$$\frac{d}{dt} \hat{q}^* \dot{\hat{q}} = \dot{\hat{q}}^* \dot{\hat{q}} + \hat{q}^* \ddot{\hat{q}}, \quad (17.42)$$

multiply by $\hat{q}$ and rearrange the terms, we obtain

$$\ddot{\hat{q}} = \hat{q} \left(\frac{d}{dt} \hat{q}^* \dot{\hat{q}} - \dot{\hat{q}}^* \dot{\hat{q}} \right). \quad (17.43)$$

Equation (17.43) can be written in matrix form as

$$\begin{bmatrix} \ddot{q}_0 \\ \ddot{q}_1 \\ \ddot{q}_2 \\ \ddot{q}_3 \end{bmatrix} = \frac{1}{2} \mathcal{Q}^T \begin{bmatrix} \dot{\omega}_1 \\ \dot{\omega}_2 \\ \dot{\omega}_3 \\ -2\sum \dot{q}_m^2 \end{bmatrix}. \quad (17.44)$$

The dynamical behavior of a rigid body can now be expressed in terms of a quaternion state vector $(q_0, \dot{q}_0, q_1, \dot{q}_1, q_2, \dot{q}_2, q_3, \dot{q}_3, t)$. The following steps summarize the algorithm for computing the rate for this state:

- i. Use the state vector to compute the angular velocity $(\omega_1, \omega_2, \omega_3)$ in the body frame using (17.40);
- ii. Project the external torques onto the body frame and compute $\dot{\boldsymbol{\omega}}$ using Euler's equation (17.22);

iii. Compute the quaternion acceleration using (17.44).

The quaternion acceleration is the rate for the quaternion velocity in a differential equation solver. This recipe is implemented in the `RigidBody` class shown in Listing 17.9.

Listing 17.9: The `RigidBody` class solves Euler’s equation of motion for a rotating rigid body using quaternions.

```
package org.opensourcephysics.sip.ch17;
import org.opensourcephysics.displayjs.utils.*;
import org.opensourcephysics.numerics.*;

public class RigidBody implements ODE {
    QuaternionRotation rotation;
    double[] state = new double[9];
    ODESolver solver;
    double[] orientation = new double[3];
    double[] omegaBody = new double[3]; // body frame omega
    double[] omegaSpace = new double[3]; // space frame omega
    double[] angularMomentumBody = new double[3]; // body frame angular momentum
    double[] angularMomentumSpace = new double[3]; // space frame angular momentum
    double I1 = 1, I2 = 4, I3 = 4; // principal moments of inertia
    double a1 = 0, a2 = 0, a3 = 0; // angular acceleration in the body frame
    double t1 = 0, t2 = 0, t3 = 0; // torque in the body frame
    public RigidBody(double[] q) {
        state[0] = q[0];
        state[2] = q[1];
        state[4] = q[2];
        state[6] = q[3];
        rotation = new QuaternionRotation(q[0], q[1], q[2], q[3]);
        solver = new RK45MultiStep(this);
    }

    QuaternionRotation getQuaternionRotation() {
        rotation.setCoordinates(state[0], state[2], state[4], state[6]);
        return rotation;
    }

    void setBodyOmegaFromSpaceOmega(double[] omega) {
        RigidBodyUtil.spaceToBody(state, omega);
        setBodyFrameOmega(omega);
    }

    void setBodyFrameOmega(double[] omega) {
        // use components for clarity
        double q0 = state[0], q1 = state[2], q2 = state[4], q3 = state[6];
        double wx = omega[0];
        double wy = omega[1];
        double wz = omega[2];
        state[1] = 0.5*(-q1*wx-q2*wy-q3*wz); // dq0/dt
```

```

    state[3] = 0.5*(q0*wx-q3*wy+q2*wz); // dq1/dt
    state[5] = 0.5*(q3*wx+q0*wy-q1*wz); // dq2/dt
    state[7] = 0.5*(-q2*wx+q1*wy+q0*wz); // dq3/dt
    updateVectors();
}

void setInertia(double[] inertia) {
    I1 = inertia[0];
    I2 = inertia[1];
    I3 = inertia[2];
    updateVectors();
}

void setOrientation(double[] q) {
    state[0] = q[0];
    state[2] = q[1];
    state[4] = q[2];
    state[6] = q[3];
    RigidBodyUtil.normalize(state);
    rotation.setCoordinates(state[0], state[2], state[4], state[6]);
}

public double[] getSpaceFrameOmega() {
    System.arraycopy(omegaBody, 0, omegaSpace, 0, 3);
    RigidBodyUtil.bodyToSpace(state, omegaSpace);
    return omegaSpace;
}

double[] getZOrientation() {
    orientation[0] = 0;
    orientation[1] = 0;
    orientation[2] = 1;
    RigidBodyUtil.bodyToSpace(state, orientation);
    return orientation;
}

public double[] getSpaceFrameAngularMomentum() {
    System.arraycopy(angularMomentumBody, 0, angularMomentumSpace, 0, 3);
    RigidBodyUtil.bodyToSpace(state, angularMomentumSpace);
    return angularMomentumSpace;
}

void updateVectors() {
    double q0 = state[0], q1 = state[2], q2 = state[4], q3 = state[6];
    omegaBody[0] = 2*(-q1*state[1]+q0*state[3]+q3*state[5]-q2*state[7]);
    omegaBody[1] = 2*(-q2*state[1]-q3*state[3]+q0*state[5]+q1*state[7]);
    omegaBody[2] = 2*(-q3*state[1]+q2*state[3]-q1*state[5]+q0*state[7]);
    angularMomentumBody[0] = I1*omegaBody[0];
    angularMomentumBody[1] = I2*omegaBody[1];
    angularMomentumBody[2] = I3*omegaBody[2];
}

```

```

    }

    public void advanceTime() {
        solver.step();
        RigidBodyUtil.normalize(state);
        updateVectors();
    }

    public double[] getState() {
        return state;
    }

    public void getRate(double[] state, double[] rate) {
        computeBodyFrameAcceleration(state);
        double sum = 0;
        for(int i = 1; i < 9; i += 2) { // sum the q-dot values
            sum += state[i]*state[i];
        }
        sum = -2.0*sum;
        // use q components for clarity
        double q0 = state[0], q1 = state[2], q2 = state[4], q3 = state[6];
        rate[0] = state[1];
        rate[1] = 0.5*(-q1*a1-q2*a2-q3*a3+q0*sum);
        rate[2] = state[3];
        rate[3] = 0.5*(q0*a1-q3*a2+q2*a3+q1*sum);
        rate[4] = state[5];
        rate[5] = 0.5*(q3*a1+q0*a2-q1*a3+q2*sum);
        rate[6] = state[7];
        rate[7] = 0.5*(-q2*a1+q1*a2+q0*a3+q3*sum);
        rate[8] = 1.0; // time rate
    }

    void computeBodyFrameTorque(double[] state) {
        t1 = t2 = t3 = 0;
    }

    void computeBodyFrameAcceleration(double[] state) {
        // use components for clarity
        double q0 = state[0], q1 = state[2], q2 = state[4], q3 = state[6];
        double wx = 2*(-q1*state[1]+q0*state[3]+q3*state[5]-q2*state[7]);
        double wy = 2*(-q2*state[1]-q3*state[3]+q0*state[5]+q1*state[7]);
        double wz = 2*(-q3*state[1]+q2*state[3]-q1*state[5]+q0*state[7]);
        computeBodyFrameTorque(state);
        a1 = (t1-(I3-I2)*wz*wy)/I1; // Euler's equations of motion
        a2 = (t2-(I1-I3)*wx*wz)/I2;
        a3 = (t3-(I2-I1)*wy*wx)/I3;
    }
}

```

The `RigidBody` class makes use of the `RigidBodyUtil` utility class in the `ch17` package to

handle quaternion normalization and transformations between space and body frames. The class is not listed because it is uninteresting. The static `spaceToBody` method multiplies the given vector by (17.33) and the static `bodyToSpace` method multiplies the given vector by the inverse.

17.7 Free rotation

We begin by visualizing the torque-free rotation of a body about its center of mass. In general, this simple rigid body problem does not have an analytical solution, although the special case in which the body possess an axis of symmetry does. We use this special case to test the validity of our numerical solutions in Exercises 17.13 and 17.14.

The `RigidBodyApp` program animates torque-free rotation by extending `AbstractSimulation` and implementing the `doStep` method. Because this class is similar to other concrete animations, we do not list it here. The program uses the `RigidBodySpaceView` class shown in Listing 17.10 to display the body, the angular momentum vector, and the angular velocity vector. This program uses the `EJSFrame` class and drawable elements from the `displayejs` package to produce a three-dimensional visualization.

Listing 17.10: The `RigidBodySpaceView` class displays the rotation of a rigid body as seen in the space frame.

```
package org.opensourcephysics.sip.ch17;
import org.opensourcephysics.displayejs.*;
import org.opensourcephysics.frames.*;

public class RigidBodySpaceView {
    InteractiveSphere ellipsoid = new InteractiveSphere();
    InteractiveArrow omega = new InteractiveArrow();
    InteractiveArrow angularMomentum = new InteractiveArrow();
    InteractiveTrace omegaTrace = new InteractiveTrace();
    InteractiveTrace momentumTrace = new InteractiveTrace();
    EJSFrame frame = new EJSFrame("Space View");
    RigidBody rigidBody;
    double scale = 1;
    public RigidBodySpaceView(RigidBody _rigidBody) {
        rigidBody = _rigidBody;
        frame.setSize(600, 600);
        frame.setDecorationType(DrawingPanel3D.DECORATION_AXES);
        omega.getStyle().setFillPattern(java.awt.Color.RED);
        omegaTrace.getStyle().setEdgeColor(java.awt.Color.RED);
        angularMomentum.getStyle().setFillPattern(java.awt.Color.GREEN);
        momentumTrace.getStyle().setEdgeColor(java.awt.Color.GREEN);
        ellipsoid.setTransformation(rigidBody.getQuaternionRotation());
        frame.addDrawable(ellipsoid);
        frame.addDrawable(omega);
        frame.addDrawable(angularMomentum);
        frame.addDrawable(omegaTrace);
        frame.addDrawable(momentumTrace);
    }
}
```

```

        frame.repaint ();
        frame.setVisible (true);
    }

    void initialize () {
        double dx = 1/Math.sqrt(rigidBody.I1); // dimension of ellipsoid is inverse to inertia
        double dy = 1/Math.sqrt(rigidBody.I2);
        double dz = 1/Math.sqrt(rigidBody.I3);
        ellipsoid .setSizeXYZ(dx, dy, dz);
        scale = Math.max(Math.max(2*dx, 2*dy), 2*dz); // bounding dimension
        frame.setPreferredMinMax(-scale, scale, -scale, scale, -scale, scale);
        omegaTrace.clear();
        momentumTrace.clear();
        update();
    }

    void update() {
        ellipsoid .setTransformation(rigidBody.getQuaternionRotation());
        double[] vec = rigidBody.getSpaceFrameOmega();
        double norm = Math.sqrt(vec[0]*vec[0]+vec[1]*vec[1]+vec[2]*vec[2]);
        norm = Math.max(norm, 1.0e-6);
        double s = 1.25*scale/norm;
        omega.setSizeXYZ(s*vec[0], s*vec[1], s*vec[2]);
        omegaTrace.addPoint(s*vec[0], s*vec[1], s*vec[2]);
        vec = rigidBody.getSpaceFrameAngularMomentum();
        norm = Math.sqrt(vec[0]*vec[0]+vec[1]*vec[1]+vec[2]*vec[2]);
        norm = Math.max(norm, 1.0e-6);
        s = 1.25*scale/norm;
        angularMomentum.setSizeXYZ(s*vec[0], s*vec[1], s*vec[2]);
        momentumTrace.addPoint(s*vec[0], s*vec[1], s*vec[2]);
        frame.repaint ();
    }
}

```

The `RigidBodySpaceView` class displays the physics of rigid body motion by retrieving data that has been computed in `RigidBody`. This data is used to set the position and orientation of various three-dimensional objects. The rotating body is represented using an inertia (or *Poinsot*) ellipsoid whose principal axes are in the ratios $I_1^{-1/2} : I_2^{-1/2} : I_3^{-1/2}$ (see Symon). These ellipsoid diameters give a visual representation of a uniform mass distribution that produces the given principal moments. The angular momentum and angular velocity vectors are retrieved from `RigidBody` and used to set the direction of the corresponding arrows. In addition, the paths of these two arrows through space are recorded by adding points to two `InteractiveTrace` objects.

Problem 17.13. Torque-free rotation about a principal axes

- Add a second visualization showing the motion as viewed in the non-inertial body frame.
- If the angular velocity vector coincides with a body's principal axis, the angular momentum and the angular velocity coincide. The body should then rotate steadily about the corresponding

principal axis because the net torque is zero and the angular momentum is constant. Does the simulation show this result for all three axes if the moments of inertia are unequal? Perturb the angular velocity. Are rotations about the three principal axes stable or unstable? Check each axis. Repeat this simulation with a different set of moments of inertia.

Problem 17.14. Symmetric body torque-free rotation

- Add a plot showing the time dependence of the angular velocity components $\omega_1(t)$, $\omega_2(t)$, and $\omega_3(t)$ to the `RigidBodyApp` class.
- Verify that $\omega_3(t)$ is constant if $I_1 = I_2 = I_s$.
- Verify that $\omega_1(t)$ and $\omega_2(t)$ exhibit an out of phase sinusoidal dependence if $I_1 = I_2$.
- If α denotes the angle between the body-frame $\hat{3}$ -axis and the axis of rotation, verify that

$$\Omega = \left(\frac{I_3}{I_s} - 1 \right) \omega \cos \alpha, \quad (17.45)$$

where Ω is the angular velocity of the $\boldsymbol{\omega}$ vector's precession about the body frame's $\hat{3}$ -axis.

Problem 17.15. Free rotation of a thin cylindrical rod

Model the free rotation of a thin cylindrical rod. Show that $\boldsymbol{\omega}$ precesses about the axis of the rod. Why does $\boldsymbol{\omega}$ precess? Why does $\mathbf{L}$ not precess? How is the frequency of precession related to the moments of inertia of the rod?

17.8 Motion of a spinning top

Object oriented programming makes it easy to add a torque to a `RigidBody`. All that needs to be done is to override the `computeBodyFrameTorque` method in the superclass. A spinning top is a body that is free to rotate about a pivot. Because the pivot point is not the center of mass, there is an external torque due to the weight of the top. We must compute the torque's vector components in the body frame to use Euler's equation of motion.

The `SpinningTop` shown in Listing 17.11 models a symmetric body rotating about the axis of symmetry. The axis of symmetry is taken to be the $\hat{3}$ -axis. If we assume that the center of mass lies a unit distance from the pivot along the $\hat{3}$ -axis, then the torque is computed by taking the cross product after projecting the force into the body frame $\boldsymbol{\tau} = \hat{3} \times F_{\text{body}}$. Listing 17.11 assumes that the center of mass is one unit from the pivot and is acted on by an external force in the z -direction $(0, 0, 1)$.

Listing 17.11: The `SpinningTop` class models a symmetric body rotating about the axis of symmetry.

```
package org.opensourcephysics.sip.ch17;
public class SpinningTop extends RigidBody {
    public SpinningTop(double[] q) {
```

```

    super(q);
}

void computeBodyFrameTorque(double[] state) {
    double[] vec = new double[] {0, 0, 1}; // external force in space frame
    RigidBodyUtil.spaceToBody(state, vec);
    t1 = -vec[1];
    t2 = vec[0];
    t3 = 0;
}
}

```

The visualization of a spinning top can take many forms. Listing 17.12 presets the spinning top model as a table-top gyroscope using cylinders and arrows. The shaft of the gyroscope is the body's axis of symmetry (the body frame's $\hat{3}$ -axis) and draws a trace showing the gyroscope's precession. Note that the spinning about the $\hat{3}$ -axis might appear to be incorrect if the animation step is too large. This aliasing effect often is seen in Hollywood movies showing carriage wheel spokes rotating backward to the direction of travel.

Listing 17.12: The `SpinningTopSpaceView` class models a symmetric body rotating about the axis of symmetry.

```

package org.opensourcephysics.sip.ch17;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.displayjs.*;
import org.opensourcephysics.displayjs.InteractiveArrow;
import org.opensourcephysics.displayjs.utils.*;

public class SpinningTopSpaceView {
    AbstractInteractiveTile shaft = new InteractiveCylinder();
    AbstractInteractiveTile disk = new InteractiveCylinder();
    AbstractInteractiveTile base = new InteractiveCylinder();
    AbstractInteractiveTile post = new InteractiveCylinder();
    InteractiveArrow omega = new InteractiveArrow();
    InteractiveArrow angularMomentum = new InteractiveArrow();
    InteractiveArrow orientation = new InteractiveArrow();
    InteractiveTrace orientationTrace = new InteractiveTrace();
    EJSFrame frame = new EJSFrame("Space View");
    SpinningTop rigidBody;
    double scale = 1;
    public SpinningTopSpaceView(SpinningTop _rigidBody) {
        rigidBody = _rigidBody;
        frame.setSize(600, 600);
        // panel.setDisplayMode(DrawingPanel3D.DISPLAY_NO_PERSPECTIVE);
        frame.setDecorationType(DrawingPanel3D.DECORATION_AXES);
        omega.getStyle().setFillPattern(java.awt.Color.YELLOW);
        angularMomentum.getStyle().setFillPattern(java.awt.Color.GREEN);
        orientation.getStyle().setFillPattern(java.awt.Color.RED);
        orientationTrace.getStyle().setEdgeColor(java.awt.Color.RED);
        base.setSizeXYZ(2, 2, 0.15);
    }
}

```

```

    base.setResolution(new Resolution(4, 12, 1));
    base.setStyle().setFillPattern(java.awt.Color.RED);
    base.setZ(-3);
    post.setSizeXYZ(0.2, 0.2, 3);
    post.setResolution(new Resolution(2, 10, 15));
    post.setZ(-1.5); // shift by half the length
    post.setStyle().setFillPattern(java.awt.Color.RED);
    shaft.setSizeXYZ(0.2, 0.2, 3);
    shaft.setOrigin(0, 0, 0, false);
    shaft.setResolution(new Resolution(1, 10, 15));
    disk.setSizeXYZ(1.5, 1.5, 0.25);
    disk.setOrigin(0, 0, -2, false);
    disk.setResolution(new Resolution(4, 12, 1));
    shaft.setTransformation(rigidBody.getQuaternionRotation());
    frame.addDrawable(base);
    frame.addDrawable(post);
    frame.addDrawable(shaft);
    frame.addDrawable(disk);
    frame.addDrawable(orientation);
    frame.addDrawable(omega);
    frame.addDrawable(angularMomentum);
    frame.addDrawable(orientationTrace);
}

void initialize () {
    double dx = 1/Math.sqrt(rigidBody.I1); // dimension of ellipsoid is inverse to inertia
    double dy = 1/Math.sqrt(rigidBody.I2);
    double dz = 1/Math.sqrt(rigidBody.I3);
    scale = Math.max(Math.max(4*dx, 4*dy), 4*dz); // bounding dimension
    frame.setPreferredMinMax(-scale, scale, -scale, scale, -scale, scale);
    orientationTrace.clear();
    update();
}

void update() {
    QuaternionRotation transformation = rigidBody.getQuaternionRotation();
    shaft.setTransformation(transformation);
    disk.setTransformation(transformation);
    double[] vec = rigidBody.getSpaceFrameOmega();
    double norm = Math.sqrt(vec[0]*vec[0]+vec[1]*vec[1]+vec[2]*vec[2]);
    norm = Math.max(norm, 1.0e-6);
    double s = 1.5*scale/norm;
    omega.setSizeXYZ(s*vec[0], s*vec[1], s*vec[2]);
    vec = rigidBody.getSpaceFrameAngularMomentum();
    norm = Math.sqrt(vec[0]*vec[0]+vec[1]*vec[1]+vec[2]*vec[2]);
    norm = Math.max(norm, 1.0e-6);
    s = 1.5*scale/norm;
    angularMomentum.setSizeXYZ(s*vec[0], s*vec[1], s*vec[2]);
    s = 4;
    vec = rigidBody.getZOrientation();

```

```

        orientation.setSizeXYZ(s*vec[0], s*vec[1], s*vec[2]);
        orientationTrace.addPoint(s*vec[0], s*vec[1], s*vec[2]);
        frame.repaint();
    }
}

```

Problem 17.16. Uniform precession

If the angular velocity about the axis of symmetry is large, there are two possible rates of steady precession. These precession rates $\dot{\phi}$ are approximately

$$\dot{\phi} \approx \frac{I_3}{I_s} \frac{\omega_3}{\cos \theta_0}, \quad (17.46)$$

and

$$\dot{\phi} \approx \frac{mgl}{I_3 \omega_3}, \quad (17.47)$$

where θ_0 is the angle between the vertical and the axis of gyroscope and mgl is the weight times the distance from the pivot to the center of mass. Demonstrate both precession rates.

Appendix 17A: Matrix Transformations

Transformations, such as rotations, can be implemented using matrices, Euler angles, or analytic functions. All that is required is that a transformation provide a rule for mapping points in a domain to points in a range. Some, but not all, transformations have an inverse that reverses this operation. To abstract the concept of a transformation, EJS defines the **Transformation** interface. This interface contains two methods that define the mapping and a third method that creates a new transformation that is an exact duplicate of the existing transformation. Objects that can make copies of themselves implement the **clone** interface and are said to be *cloneable*.

```

package org.opensourcephysics.displayejs;

public interface Transformation extends Cloneable {
    public void direct (double[] point);
    public void inverse (double[] point) throws UnsupportedOperationException;
    public Object clone();
}

```

The **Affine3DMatrix** matrix class shown in Listing 17.13 implements the **Transformation** interface. Because rotations are very common, the class implements the **Rotation** convenience method to create a matrix using (17.15). The **direct** and **inverse** methods use the matrix and the inverse matrix to transform a point, respectively. Because a client may not need the inverse, because the inverse may not exist, and because an inverse is expensive to compute, we do not compute this matrix until it is needed, in which case the inverse is computed only once. Note that the inverse is calculated using a *lower upper (LU) matrix decomposition*. LU decomposition comes

from the realization that a nonsingular square matrix $\mathcal{A}$ can be decomposed into a product of two matrices $\mathcal{L}$ and $\mathcal{U}$ whose components below and above the diagonal are zero, respectively.

$$\mathcal{A} = \mathcal{L}\mathcal{U} \quad (17.48)$$

We will not describe this technique further here, but you are encouraged to test that the routine in the numerics package works and to study the algorithm further in *Numerical Recipes* or in other numeral methods texts.

Listing 17.13: The `Affine3DMatrix` class implements the `Transformation` interface using a matrix representation for the affine transformations.

```
package org.opensourcephysics.sip.ch17;
import org.opensourcephysics.numerics.*;

public class Affine3DMatrix implements Transformation {
    private double[][] matrix = new double[4][4]; // the transformation matrix
    private double[][] inverse = null;           // the inverse transformation matrix if it exists
    private boolean inverted = false;            // true if the inverse has been computed
    public Affine3DMatrix(double[][] matrix) {
        if(matrix==null) { // identity matrix
            this.matrix[0][0] = this.matrix[1][1] = this.matrix[2][2] = this.matrix[3][3] = 1;
            return;
        }
        for(int i = 0; i < matrix.length; i++) { // loop over the rows
            System.arraycopy(matrix[i], 0, this.matrix, 0, matrix[i].length);
        }
    }

    public static Affine3DMatrix Rotation(double theta, double[] axis) {
        Affine3DMatrix at = new Affine3DMatrix(null);
        double[][] atMatrix = at.matrix;
        double norm = Math.sqrt(axis[0]*axis[0]+axis[1]*axis[1]+axis[2]*axis[2]);
        double x = axis[0]/norm, y = axis[1]/norm, z = axis[2]/norm;
        double c = Math.cos(theta), s = Math.sin(theta);
        double t = 1-c;
        // matrix elements not listed are zero
        atMatrix[0][0] = t*x*x+c;
        atMatrix[0][1] = t*x*y-s*z;
        atMatrix[0][2] = t*x*y+s*y;
        atMatrix[1][0] = t*x*y+s*z;
        atMatrix[1][1] = t*y*y+c;
        atMatrix[1][2] = t*y*z-s*x;
        atMatrix[2][0] = t*x*z-s*y;
        atMatrix[2][1] = t*y*z+s*x;
        atMatrix[2][2] = t*z*z+c;
        atMatrix[3][3] = 1;
        return at;
    }
}
```

```

public static Affine3DMatrix Translation(double dx, double dy, double dz) {
    Affine3DMatrix at = new Affine3DMatrix(null);
    double[][] m = at.matrix;
    // matrix elements not listed are zero
    m[0][0] = 1;
    m[0][3] = dx;
    m[1][1] = 1;
    m[1][3] = dy;
    m[2][2] = 1;
    m[2][3] = dz;
    m[3][3] = 1;
    return at;
}

public Object clone() {
    return new Affine3DMatrix(matrix);
}

public void direct(double[] point) {
    int n = point.length;
    double[] tempPoint = new double[n];
    System.arraycopy(point, 0, tempPoint, 0, n);
    for(int i = 0; i < n; i++) {
        point[i] = 0;
        for(int j = 0; j < n; j++) {
            point[i] += matrix[i][j]*tempPoint[j];
        }
    }
}

public void inverse(double[] point) throws UnsupportedOperationException {
    if (!inverted) {
        calcInverse(); // computes the inverse using LU decomposition
    }
    if (inverse == null) { // inverse does not exist
        throw new UnsupportedOperationException("The inverse matrix does not exist.");
    }
    int n = point.length;
    double[] pt = new double[n];
    System.arraycopy(point, 0, pt, 0, n);
    for(int i = 0; i < n; i++) {
        point[i] = 0;
        for(int j = 0; j < n; j++) {
            point[i] += inverse[i][j]*pt[j];
        }
    }
}

// calculates the inverse using Lower-Upper decomposition.
private void calcInverse() {

```

```

    LUPDecomposition lupd = new LUPDecomposition(matrix);
    inverse = lupd.inverseMatrixComponents();
    inverted = true; // signal that the inverse computation has been performed
  }
}

```

Exercise 17.17. Transformation interface

Write a simple program to test the `Affine3DMatrix` class. Show that the direct and inverse transformations reverse each others mappings.

Appendix 17B: Conversions

Physicists use Euler angles because they are useful for analytically solving Euler's rigid body equations of motion in a small number of special cases. Because the quaternion representation is unfamiliar and is not taught in standard texts, we present conversion formulas between quaternions, rotation matrices, and Euler angles. See Shoemake for a more complete discussion of these conversions.

Quaternion to matrix

For a quaternion $q = (q_0, q_1, q_2, q_3)$ that satisfies the normalization condition $1 = q_0^2 + q_1^2 + q_2^2 + q_3^2$, the formula for the rotation matrix is

$$\mathcal{R} = 2 \begin{bmatrix} \frac{1}{2} - q_2^2 - q_3^2 & q_1 q_2 + q_0 q_3 & q_1 q_3 - q_0 q_2 \\ q_1 q_2 - q_0 q_3 & \frac{1}{2} - q_1^2 - q_3^2 & q_2 q_3 + q_0 q_1 \\ q_1 q_3 + q_0 q_2 & q_2 q_3 - q_0 q_1 & \frac{1}{2} - q_1^2 - q_2^2 \end{bmatrix}. \quad (17.49)$$

Matrix to quaternion

The quaternion components can be computed using linear combinations of the rotation matrix elements $\mathcal{R} = [r_{i,j}]_{3 \times 3}$. To avoid the pitfall of dividing by a small number ϵ (the machine precision), we compute quaternion components using if statements:

- Compute $w^2 = (1 + r_{0,0} + r_{1,1} + r_{2,2})/4$.

If $w^2 > \epsilon$, then

$$q_0 = \sqrt{w^2} \quad (17.50a)$$

$$q_1 = (r_{1,2} - r_{2,1})/4q_0 \quad (17.50b)$$

$$q_2 = (r_{2,0} - r_{0,2})/4q_0 \quad (17.50c)$$

$$q_3 = (r_{0,1} - r_{1,0})/4q_0, \quad (17.50d)$$

else compute $x^2 = -1/2(r_{1,1} + r_{2,2})$.

- If $x^2 > \epsilon$, then

$$q_0 = 0 \quad (17.51a)$$

$$q_1 = \sqrt{x^2} \quad (17.51b)$$

$$q_2 = r_{0,1}/2q_1 \quad (17.51c)$$

$$q_3 = r_{0,2}/2q_1, \quad (17.51d)$$

else compute $y^2 = 1/[2(1 - r_{2,2})]$.

- If $y^2 > \epsilon$, then

$$q_0 = 0 \quad (17.52a)$$

$$q_1 = 0 \quad (17.52b)$$

$$q_2 = \sqrt{y^2} \quad (17.52c)$$

$$q_3 = r_{1,2}/2q_2, \quad (17.52d)$$

else compute

$$q_0 = 0 \quad (17.53a)$$

$$q_1 = 0 \quad (17.53b)$$

$$q_2 = 0 \quad (17.53c)$$

$$q_3 = 1. \quad (17.53d)$$

Euler angles to matrix

Euler angles are generally described in physics texts (see Goldstein) as a group of three rotations about a set of body-frame axes. An object is created with the body frame aligned with the space frame. The first rotation is about the body frame's $\hat{z}$ axis by an angle ϕ ; the second rotation is about the new x axis by an angle θ , and the third rotation is about the new z axis by an angle ψ . Other definitions of Euler angles are possible. For example, the Java 3D API defines Euler angles as three rotations about a set of axes fixed in space. All possible positions of an object can be represented using either of these conventions.

The first rotation is about z and is given by

$$\mathcal{A}(\phi) = \begin{bmatrix} \cos \phi & \sin \phi & 0 \\ -\sin \phi & \cos \phi & 0 \\ 0 & 0 & 1 \end{bmatrix}. \quad (17.54)$$

The second rotation is about the new x axis and is given by

$$\mathcal{B}(\theta) = \begin{bmatrix} 0 & 0 & 1 \\ 0 & \cos \theta & \sin \theta \\ 0 & -\sin \theta & \cos \theta \end{bmatrix}. \quad (17.55)$$

And the last last rotation is again about the new z axis

$$\mathcal{C}(\psi) = \begin{bmatrix} \cos \psi & \sin \psi & 0 \\ -\sin \psi & \cos \psi & 0 \\ 0 & 0 & 1 \end{bmatrix}. \quad (17.56)$$

The application of the three Euler rotation matrices $\mathcal{C}(\psi)\mathcal{B}(\theta)\mathcal{A}(\phi)$ in this order produces the transformation:

$$\mathcal{R}(\psi, \theta, \phi) = \begin{bmatrix} \cos \psi \cos \phi - \cos \theta \sin \phi \sin \psi & \cos \psi \sin \phi + \cos \theta \cos \phi \sin \psi & \sin \psi \sin \theta \\ -\sin \psi \cos \phi - \cos \theta \sin \phi \cos \psi & -\sin \psi \sin \phi + \cos \theta \cos \phi \cos \psi & \cos \psi \sin \theta \\ \sin \theta \sin \phi & -\sin \theta \cos \phi & \cos \theta \end{bmatrix}. \quad (17.57)$$

Euler angles to quaternion

There are many possible conventions for the Euler angles. We again use the definition found in Goldstein.

$$q_0 = \cos \theta/2 \cos \frac{1}{2}(\phi + \psi) \quad (17.58a)$$

$$q_1 = \sin \theta/2 \cos \frac{1}{2}(\phi - \psi) \quad (17.58b)$$

$$q_2 = \sin \theta/2 \sin \frac{1}{2}(\phi - \psi) \quad (17.58c)$$

$$q_3 = \sin \theta/2 \sin \frac{1}{2}(\phi + \psi). \quad (17.58d)$$

Matrix to Euler Angles

The conversion from matrix elements to Euler angles is ill-defined because inverse trigonometric functions do not uniquely specify the resulting quadrant. From (17.57) we see that $\cos \theta = r_{2,2}$. We then use $\sin \theta = \sqrt{1 - \cos^2 \theta}$ to compute $\sin \theta$ to within a sign. As in the matrix to quaternion conversion, we again use if statements to avoid dividing a number less than the machine precision ϵ :

If $|r_{2,2}| > \epsilon$, then

$$\cos \theta = r_{2,2} \quad (17.59a)$$

$$\sin \theta = \sqrt{1 - \cos^2 \theta} \quad (17.59b)$$

$$\cos \phi = r_{1,0} / \sin \theta \quad (17.59c)$$

$$\sin \phi = -r_{2,0} / \sin \theta \quad (17.59d)$$

$$\cos \psi = r_{1,2} / \sin \theta \quad (17.59e)$$

$$\sin \psi = r_{0,2} / \sin \theta, \quad (17.59f)$$

else

$$\cos \theta = 0 \quad (17.60a)$$

$$\sin \theta = 1 \quad (17.60b)$$

$$\cos \phi = r_{1,0} \quad (17.60c)$$

$$\sin \phi = -r_{2,0} \quad (17.60d)$$

$$\cos \psi = 1 \quad (17.60e)$$

$$\sin \psi = 0. \quad (17.60f)$$

Quaternions to Euler Angles

Convert the quaternion to a rotation matrix and then convert the matrix to Euler angles.

References and Suggestions for Further Reading

- Didier H. Besset, *Object-Oriented Implementation of Numerical Methods*, Morgan Kaufmann (2001).
- David H. Eberly, *Game Physics*, Morgan Kaufmann (2004)
- Denis J. Evans, “On the representation of orientation space,” *Mol. Phys.* **34** (2) 317–325 (1977).
- James D. Foley, Andries van Dam, Steven Feiner, and John Hughes *Computer Graphics: Principles and Practice*, second edition, Addison Wesley (1990).
- Herbert Goldstein, Charles P. Poole, and John L. Safko, *Classical Mechanics*, third edition, Addison-Wesley (2002). Chapter 4 discusses the kinematics of rigid body motion.
- William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes*, second edition, Cambridge University Press (1992).
- D. C. Rapaport, “Molecular dynamics simulation using quaternions,” *J. Comp. Phys.* **60** 306–314 (1985).
- D. C. Rapaport, *The Art of Molecular Dynamics Simulation*, second edition, Cambridge University Press (2004). Chapter 8 discusses the molecular dynamics of rigid molecules.
- Philip J. Schneider and David H. Eberly, *Geometric Tools for Computer Graphics*, Morgan Kaufmann (2003)
- Ken Shoemake, “Animating rotation with quaternion curves,” *ACM Transactions in Graphics* **19** (3), 256–276 (1994).
- Keith R. Symon, *Mechanics*, Addison-Wesley (1971).
- James M. Van Verth and Lars M. Bishop, *Essential Mathematics for Games and Interactive Applications*, Morgan Kaufmann (2004).