GAME 2302-0145: Getting Started with the Vector Dot Product*

Richard Baldwin

Abstract

Learn the fundamentals of the vector dot product in both 2D and 3D. Learn how to update the game-math library to support various aspects of the vector dot product. Learn how to write 2D and 3D programs that use the vector dot product methods in the game-math library.

1 Table of Contents

*Version 1.1: Oct 21, 2012 12:26 pm -0500

†http://creativecommons.org/licenses/by/3.0/

2 Preface

This module is one in a collection of modules designed for teaching *GAME2302 Mathematical Applications for Game Development* at Austin Community College in Austin, TX.

What you have learned

In the previous module, you learned how to write your first interactive 3D game using the game-math library. You also learned how to write a Java program that simulates flocking behavior such as that exhibited by birds and fish and how to incorporate that behavior into a game. Finally, you examined three other programs that illustrate various aspects of both 2D and 3D animation using the game-math library.

What you will learn

This module is the first part of a two-part mini-series on the *vector dot product* . By the time you finish both parts, you will have learned

- the fundamentals of the vector dot product in both 2D and 3D,
- how to update the game-math library to support various aspects of the vector dot product, and
- how to write 2D and 3D programs that use the vector dot product for various applications such as the back-face culling procedure that was used to convert the image in Figure 1 (p. 2) to the image in Figure 2 (p. 3) .

A 3D image before back-face culling.

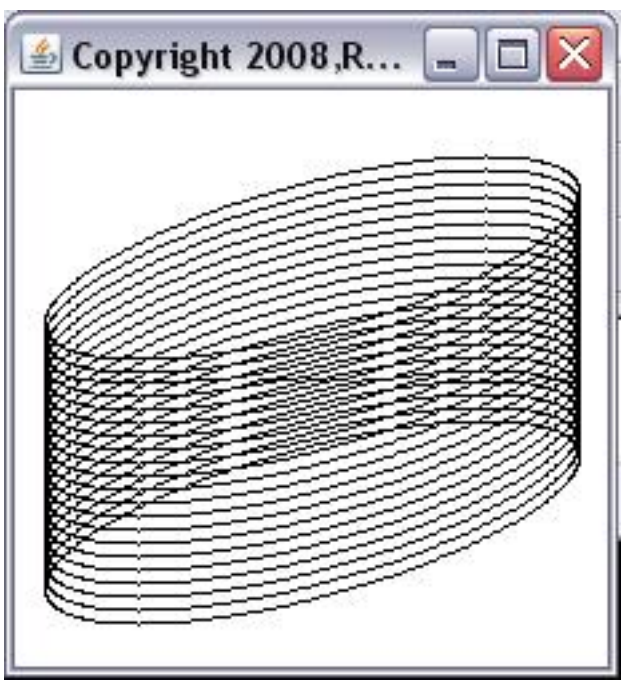

Figure 1: A 3D image before back-face culling.

.

The 3D image after back-face culling.

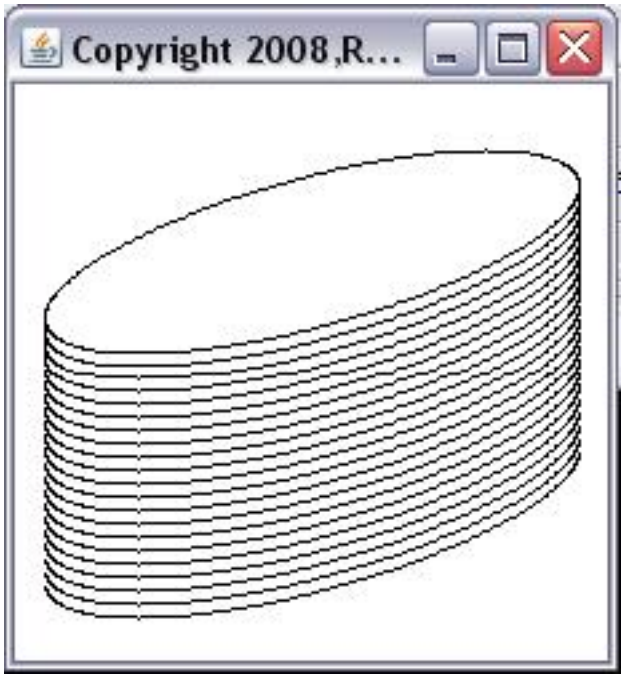

Figure 2: The 3D image after back-face culling.

2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

2.1.1 Figures

2.1.2 Listings

3 Preview

The homework assignment for this module was to study the Kjell tutorial through *Chapter 10, Angle between 3D Vectors* .

I won't repeat everything that Dr. Kjell has to say. However, there are a few points that I will summarize in this section.

Basic definition of the vector dot product

The vector dot product is a special way to multiply two vectors to produce a real result. **A** description of the vector dot product follows.

> NOTE: **The vector dot product:** the vector dot product of two vectors is the product of the lengths of the vectors multiplied by the cosine of the angle between them

By *the angle between them* , I mean the angle that would be formed if you were to draw the two vectors with their tails in the same location.

For example, Figure 3 (p. 5) shows a black vector and a magenta vector drawn with their tails at the origin. Eyeballing the picture suggests that the angle between the two vectors is forty or fifty degrees.

Two vectors with their tails at the origin, program DotProd2D02.

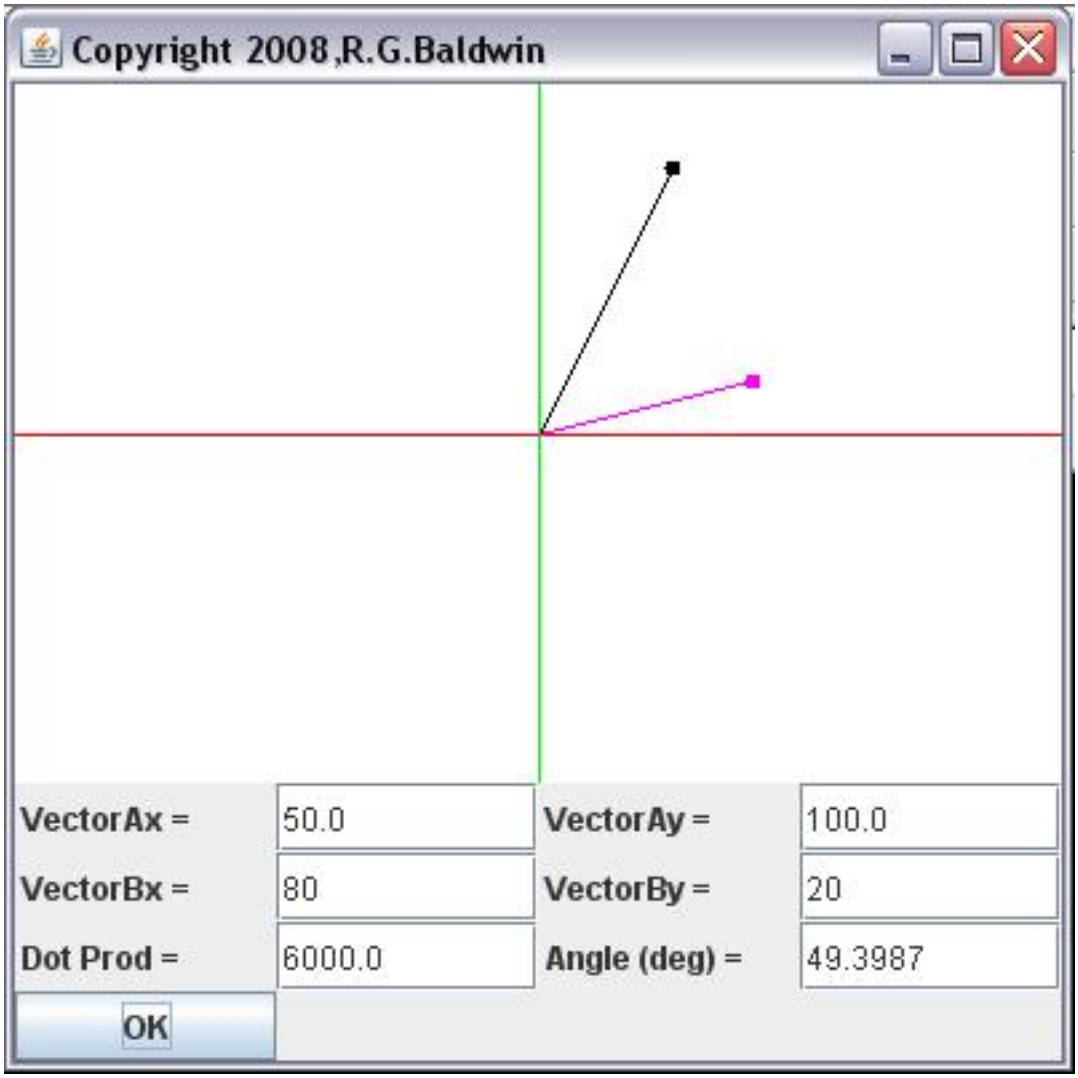

Figure 3: Two vectors with their tails at the origin, program DotProd2D02.

Can do more than eyeball

Fortunately, we can do more than eyeball the angle between two vectors. Figure 3 (p. 5) shows the screen output produced by the program named **DotProd2D02** that I will explain in this module. **DotProd2D02** is a 2D program. I will also explain a 3D version named **DotProd3D02** in this module as well.

In Figure 3 (p. 5) , the top four user input fields allow the user to enter the x and y coordinate values of two vectors according to the labels that identify those fields. When the user clicks the OK button, the first vector is drawn in black with its tail at the origin and the second vector is drawn in magenta with its tail at

the origin. The dot product of the two vectors is computed and displayed in the bottom left text field, and the angle between the two vectors is computed and displayed in the bottom right text field.

Don't need to know the angle between the vectors

Upon seeing the description of the dot product given above (p. 4) , you may reasonably be concerned about needing to know the angle between the vectors before you can compute the dot product. Fortunately, as you will see later (p. 11) , it is possible to compute the dot product of two vectors without knowing the angle. In fact, being able to compute the dot product is one way to determine the angle between two vectors.

As you can see, the value of the dot product of the two vectors shown in Figure 3 (p. 5) is 6000 and the angle between the vectors is 49.3987 degrees. You will learn how those values were computed shortly.

Major properties of the dot product

Here are some of the major properties of the dot product of two vectors:

1. The dot product of two vectors with the same orientation is the product of their lengths.
2. The length of a vector is the square root of the dot product of the vector with itself.
3. The dot product of two vectors having opposite orientations is the negative of the product of their lengths.
4. The dot product of perpendicular vectors is zero.
5. The angle between two vectors is the same as the angle between normalized versions of the vectors, which is equal to the arc cosine of the dot product of the normalized vectors.

As you will see later, these properties apply to both 2D and 3D vectors. In many cases, they also apply to vectors having more than three dimensions as well.

Dot product of two vectors with the same orientation in 2D

Figure 4 (p. 7) illustrates the first property in the above list: *The dot product of two vectors with the same orientation is the product of their lengths.*

Dot product of two vectors with the same orientation in 2D.

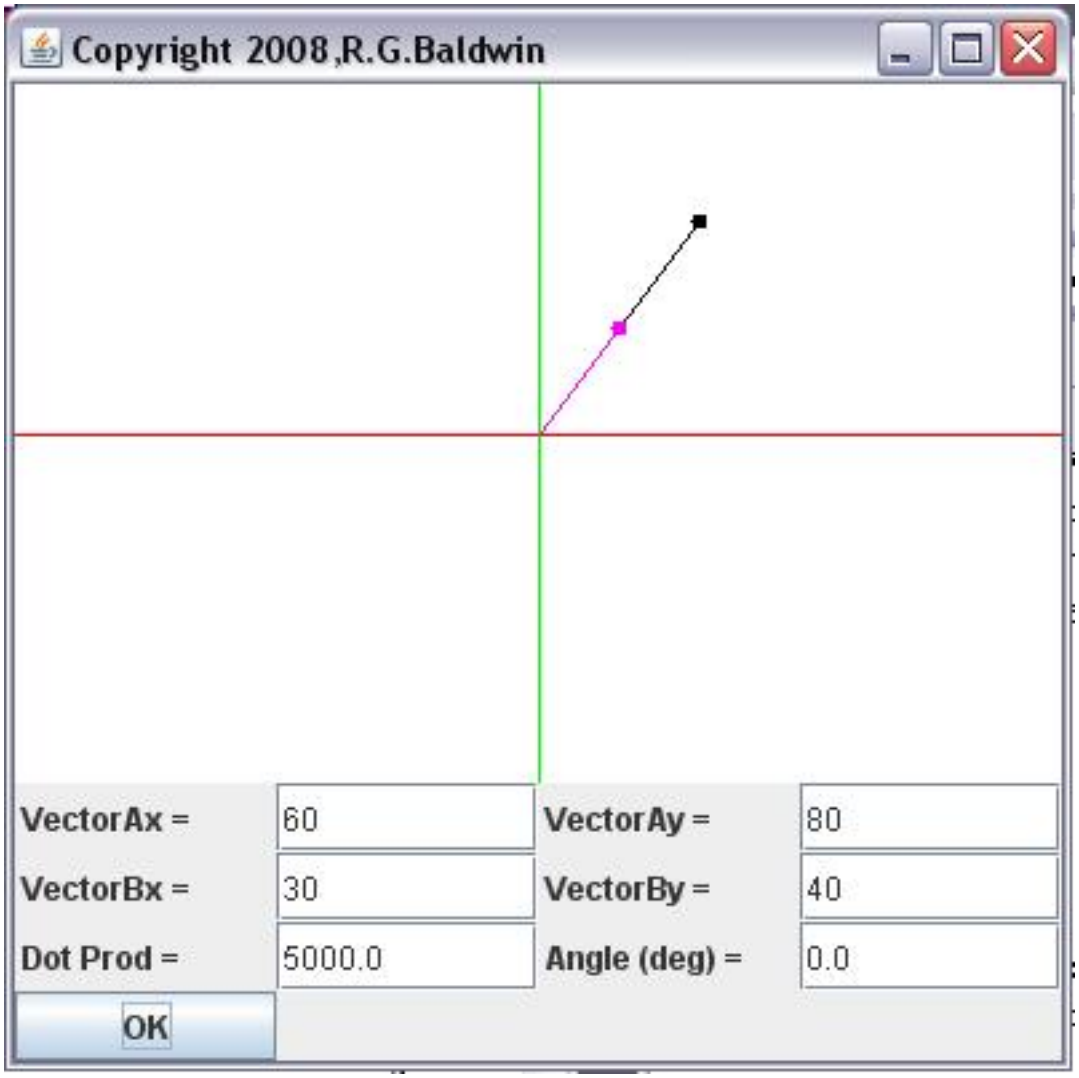

Figure 4: Dot product of two vectors with the same orientation in 2D.

You may recall from some earlier experience that when a right triangle has sides with lengths of 30 and 40, the length of the hypotenuse is 50. That is the case for the magenta vector shown in Figure 4 (p. 7) . Similarly, when the sides of the triangle are 60 and 80, the length of the hypotenuse is 100, as is the case for the black vector in Figure 4 (p. 7) .

From the property given above, we know that the dot product of the black and magenta vectors shown in Figure 4 (p. 7) is 5000, which agrees with the value shown in the **Dot Prod** output field in Figure 4 (p. 7) .

Dot product of two vectors with the same orientation in 3D

Figure 5 (p. 8) shows the dot product of two vectors with the same orientation in 3D. The image in Figure 5 (p. 8) was produced by the program named **DotProd3D02** .

Dot product of two vectors with the same orientation in 3D.

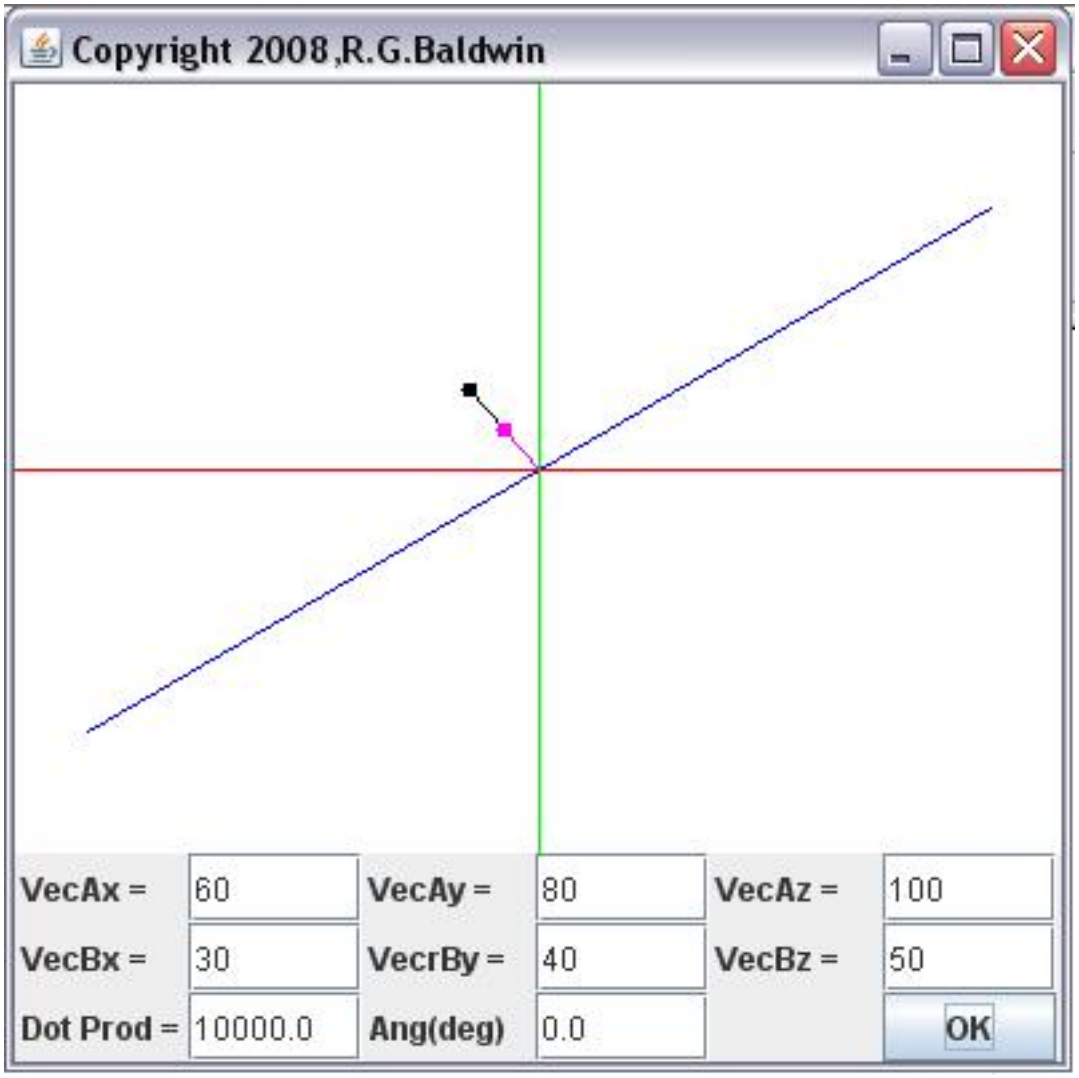

Figure 5: Dot product of two vectors with the same orientation in 3D.

Manually calculate the value of the dot product

You may need to get your calculator out to manually compute the lengths of the two vectors in Figure 5 (p. 8) . Computing the lengths as the square root of the sum of the squares of the three components of each vector gives me the following lengths:

- Black length = 141.42

- Magenta length = 70.71

Rounding the product of the two lengths gives the dot product value of 10000, which matches the value shown in the bottom left output field in Figure 5 (p. 8) .

The length of a vector

Figure 6 (p. 9) illustrates the second property in the above list (p. 6) : *The length of a vector is the square root of the dot product of the vector with itself* .

Dot product of a 3D vector with an identical vector.

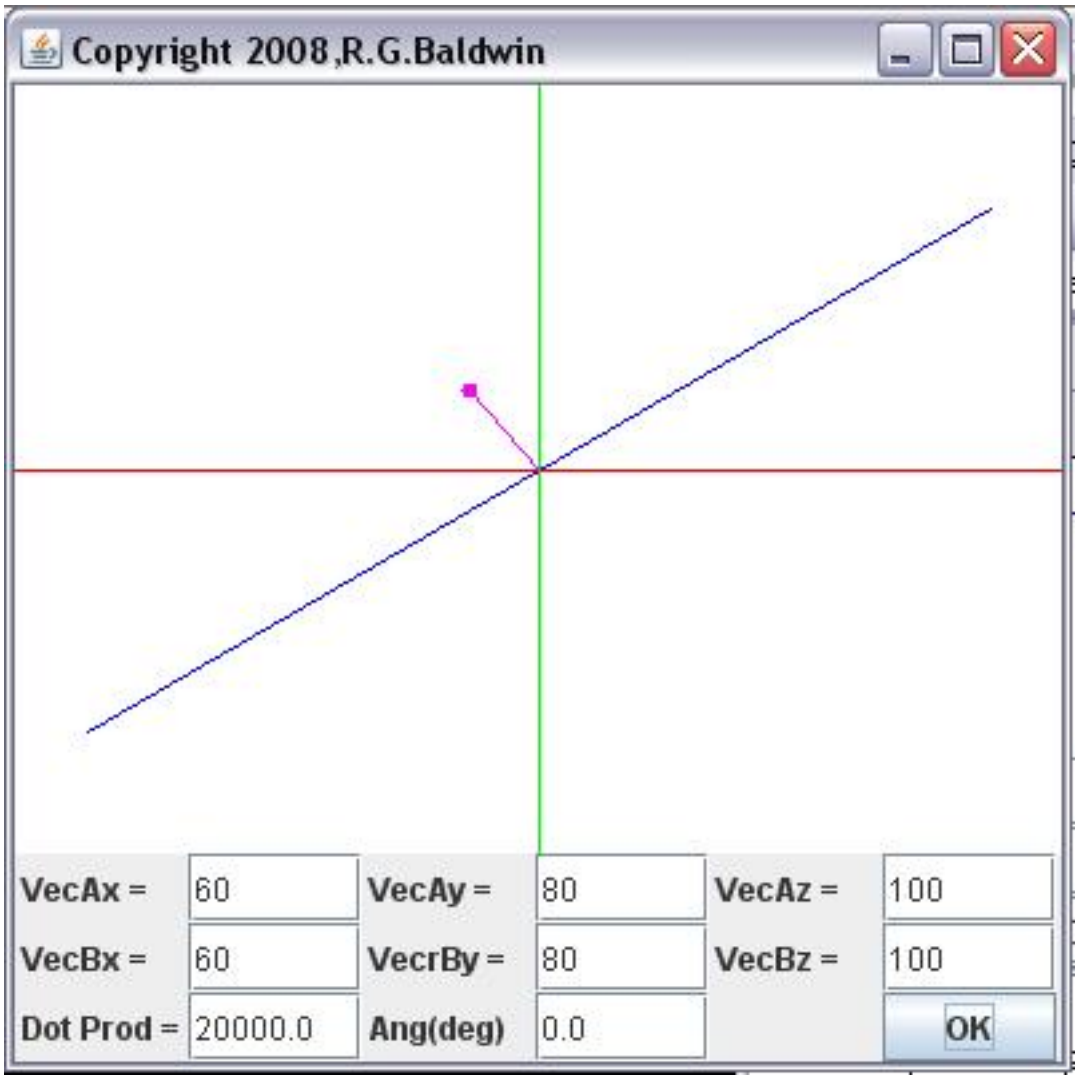

Figure 6: Dot product of a 3D vector with an identical vector.

Figure 6 (p. 9) displays two vectors having the same coordinate values as the black vector in Figure 5 (p. 8) . *(The black vector is hidden by the magenta vector in Figure 6 (p. 9) .)* Because these two vectors have identical coordinate values, the dot product of these two vectors is the same as the dot product of either vector with itself.

We concluded earlier that the length of each of these vectors is 141.42. This length is the square root of the dot product of the vector with itself. Squaring and rounding this length gives a dot product value of 20000, which matches the value shown in the bottom left output field in Figure 6 (p. 9) .

Dot product of vectors with opposite orientations

Figure 7 (p. 11) illustrates the third property of the dot product given above (p. 6) : *The dot product of two vectors having opposite orientations is the negative of the product of their lengths* .

Dot product of vectors with opposite orientations.

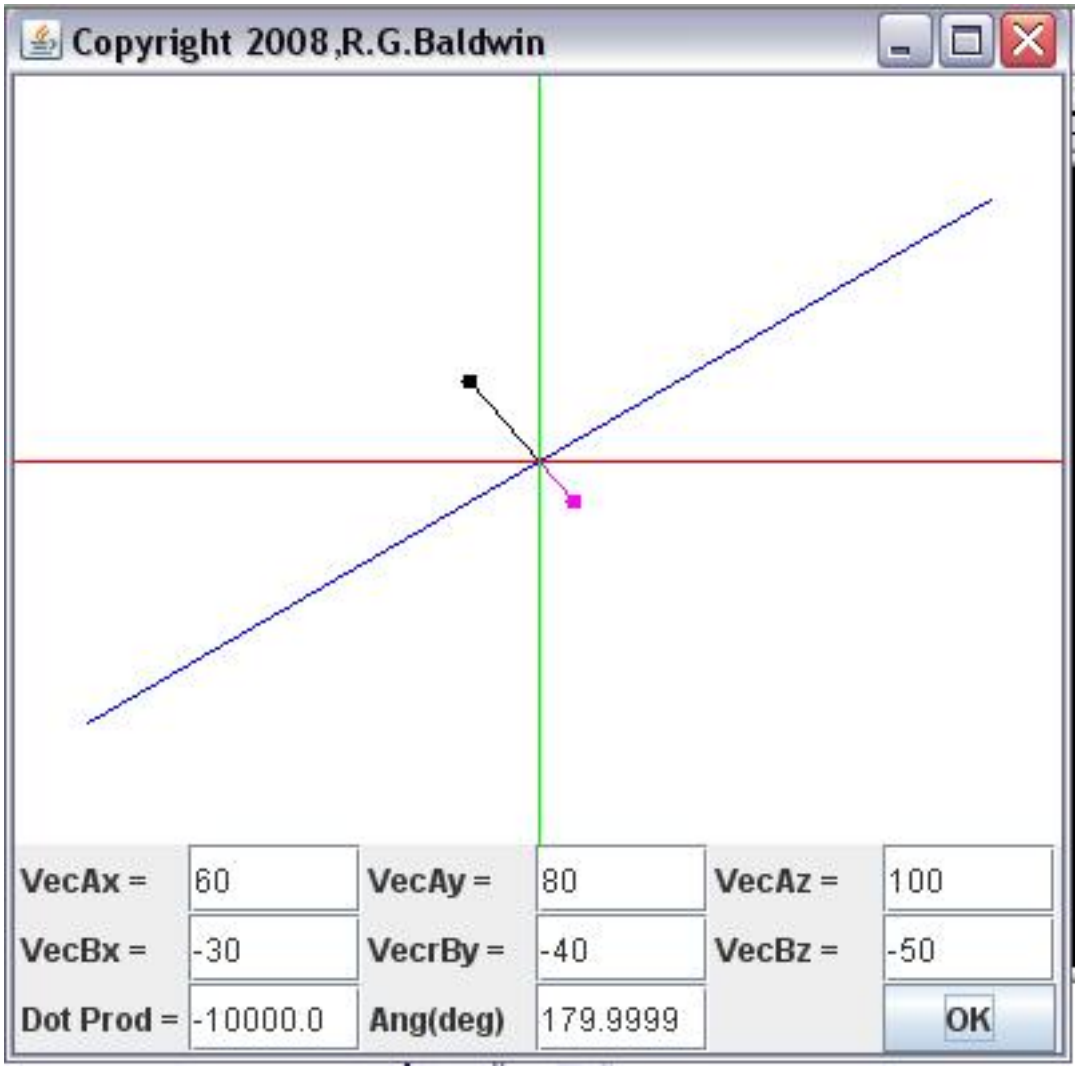

Figure 7: Dot product of vectors with opposite orientations.

The two vectors shown in Figure 7 (p. 11) have the same absolute coordinates as the two vectors shown in Figure 5 (p. 8) . However, the algebraic signs of the coordinates of the magenta vector in Figure 7 (p. 11) were reversed relative to Figure 4, causing the magenta vector to point in the opposite direction from the black vector. *(Note that the angle between the two vectors, as reported by the program is zero degrees in Figure 5 (p. 8) and is 180 degrees in Figure 7 (p. 11) .)*

The point here is that the dot product of the two vectors in Figure 7 (p. 11) is the negative of the dot product of the two vectors in Figure 5 (p. 8) . This property will be used in another program in the second part of this two-part miniseries to achieve the back-face culling shown in Figure 1 (p. 2) .

The computational simplicity of the vector dot product

If you have studied the Kjell tutorial through *Chapter 10, Angle between 3D Vectors* you have learned that the dot product of two vectors can be computed as the sum of products of the corresponding x, y, and z components of the two vectors. In particular, **in the 2D case** , the dot product is given by:

2D dot product = x1*x2 + y1*y2

Similarly, **in the 3D case** , the dot product is given by:

3D dot product = x1*x2 + y1*y2 + z1*z2

Note that these two formulations don't require the program to know anything about the angle between the two vectors, as is the case in the earlie (p. 4) r definition.

The dot product of perpendicular vectors in 2D

The dot product has many uses in game programming, not the least of which is determining if two vectors are perpendicular.

Figure 8 (p. 13) illustrates the fourth property in the above list (p. 6) : *The dot product of perpendicular vectors is zero* . This is an extremely important property in that it allows game programs to easily determine if two vectors are perpendicular. I will begin with a 2D discussion because the topic of perpendicularly of vectors is **less complicated in 2D than in 3D** .

The dot product of perpendicular vectors in 2D.

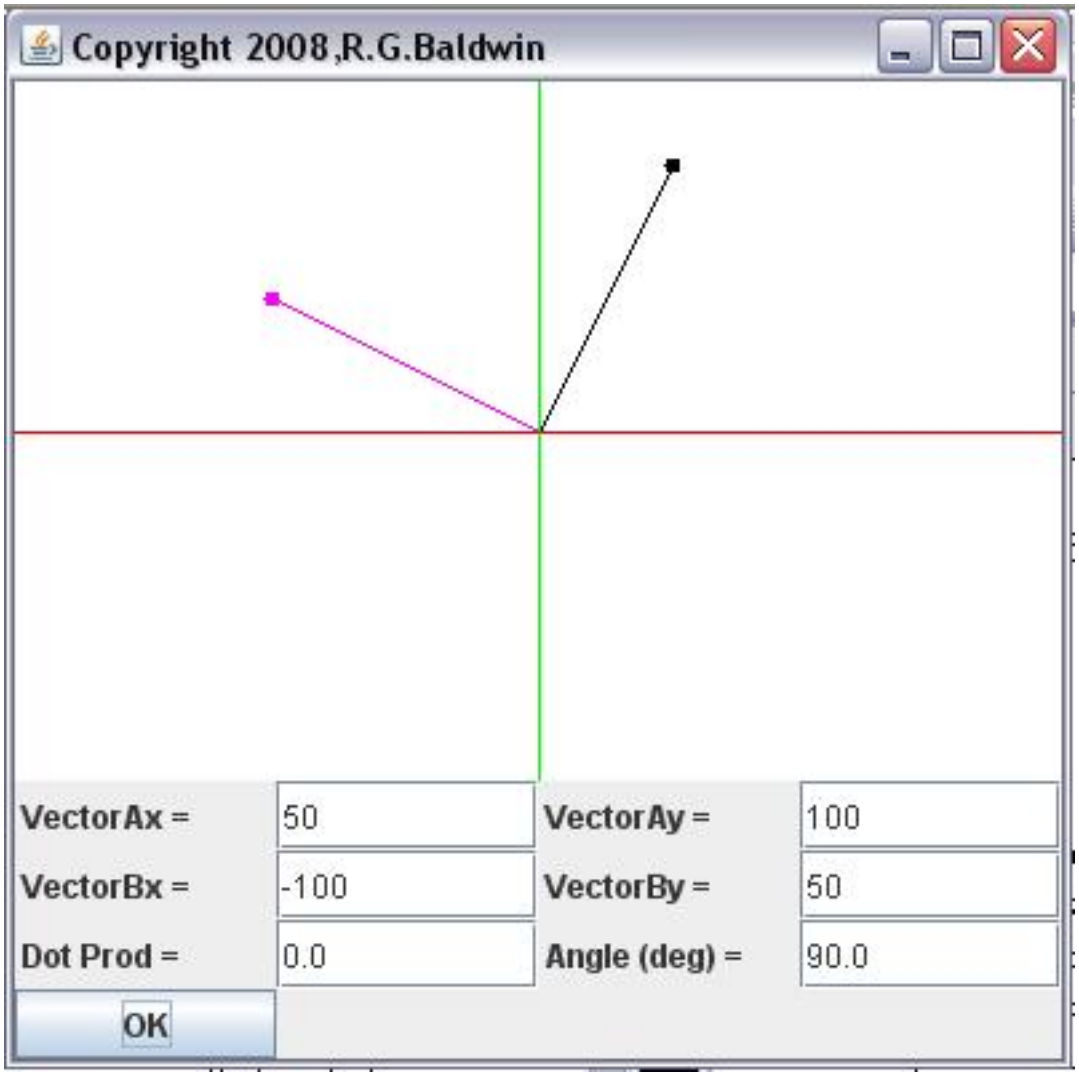

Figure 8: The dot product of perpendicular vectors in 2D.

By eyeballing Figure 8 (p. 13) , you can see that the magenta vector is probably perpendicular to the black vector. Looking at the output fields at the bottom left and the bottom right in Figure 8 (p. 13) , you will see that the dot product of the two vectors is zero and the angle between the two vectors is 90 degrees.

Restating the perpendicularity property

Most of us learned in our earlier mathematics classes that the angle between perpendicular lines is 90 degrees. Therefore, the angle between perpendicular vectors must be 90 degrees. *(See a definition of perpendicularity* [1] *.)*

[1] http://www.thefreedictionary.com/perpendicularity

Most of us also learned in our trigonometry classes that the cosine of 90 degrees is 0.0. Since, by definition (p. 4) , *the value of the dot product of two vectors is the product of the lengths of the vectors multiplied by the cosine of the angle between them* , and the angle must be 90 degrees for two vectors to be perpendicular, then the dot product of perpendicular vectors must be zero as stated by the fourth property in the above list of properties (p. 6) .

An infinite number of perpendicular vectors

By observing Figure 8 (p. 13) , it shouldn't be too much of a stretch for you to recognize that there are an infinite number of different vectors that could replace the magenta vector in Figure 8 (p. 13) and be perpendicular to the black vector. However, since we are discussing the 2D case here, all of those vectors must lie in the same plane and must have the same orientation *(or the reverse orientation)* as the magenta vector. In other words, all of the vectors in the infinite set of vectors that are perpendicular to the black vector must lie on the line defined by the magenta vector, pointing in either the same direction or in the opposite direction. However, those vectors can be any length and still lie on that same line.

A general formulation of 2D vector perpendicularity

By performing some algebraic manipulations on the earlier (p. 12) 2D formulation of the dot product, we can formulate the equations shown in Figure 9 (p. 14) that define the infinite set of perpendicular vectors described above.

A general formulation of 2D vector perpendicularity.

```
    dot product = x1*x2 + y1*y2

If the two vectors are perpendicular:

x1*x2 + y1*y2 = 0.0
x1*x2 =  -y1*y2
x2 = -y1*y2/x1
```

Figure 9: A general formulation of 2D vector perpendicularity.

As you can see from Figure 9 (p. 14) , we can assume any values for y1, y2, and x1 and compute a value for x2 that will cause the two vectors to be perpendicular.

A very interesting case

One very interesting 2D case is the case shown in Figure 8 (p. 13) . In this case, I initially specified one of the vectors to be given by the coordinate values (50,100). Then I assumed that y2 is equal to x1 and computed the value for x2. The result is that the required value of x2 is the negative of the value of y1.

Thus, in the 2D case, we can easily define two vectors that are perpendicular by

- swapping the coordinate values between two vectors and
- negating one of the coordinate values in the second vector

The actual direction that the second vector points will depend on which value you negate in the second vector.

Another interesting 2D case of perpendicular vectors

Another interesting 2D case is shown in Figure 10 (p. 15) .

Another interesting 2D case of perpendicular vectors.

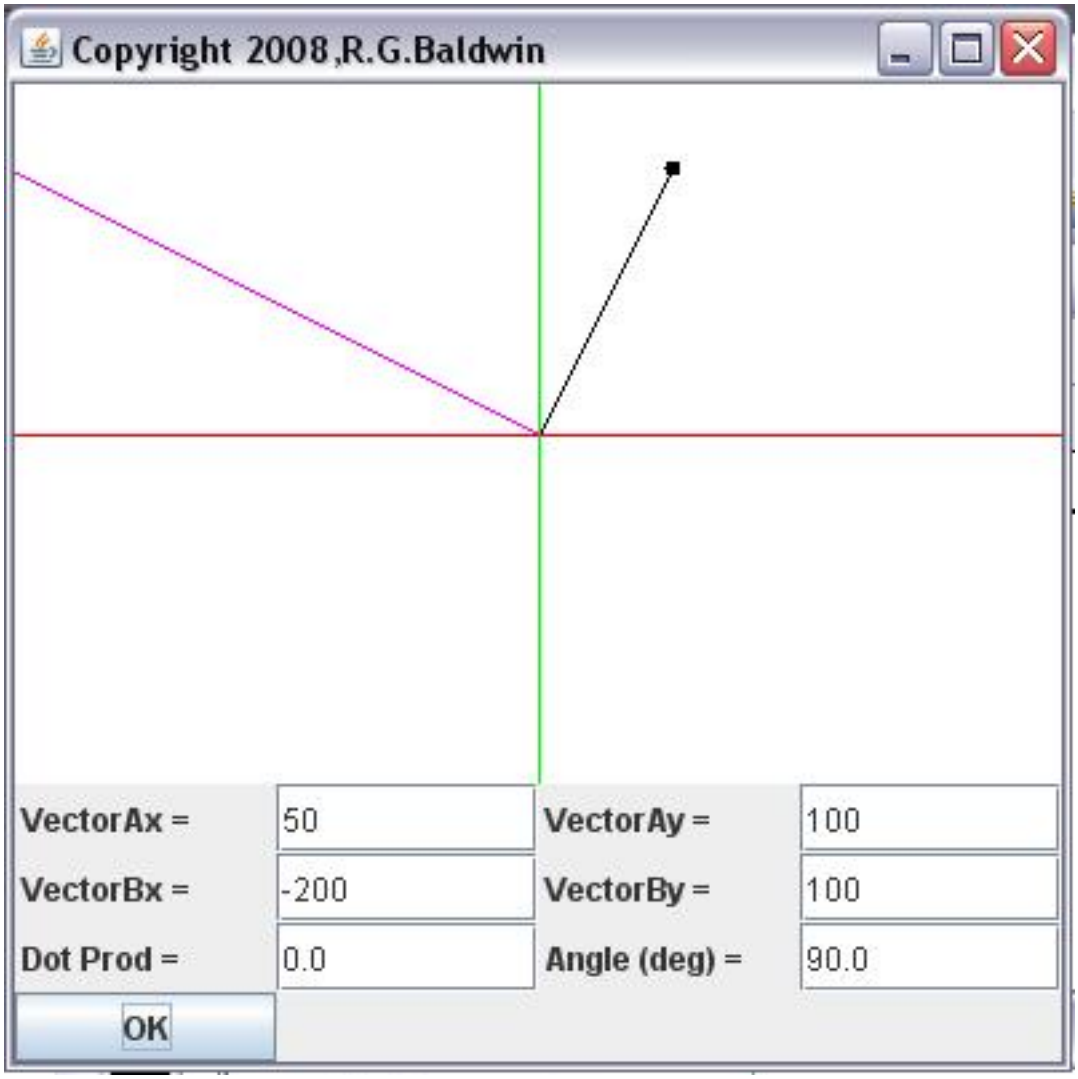

Figure 10: Another interesting 2D case of perpendicular vectors.

In Figure 10 (p. 15) , I assumed the same coordinate values for the black vector as in Figure 8 (p. 13) . Then I assumed that the values of the y-coordinates for both vectors are the same. Using those values along with the equation in Figure 8, I manually computed a required value of -200 for the x-coordinate of the magenta vector. I entered that value into the field labeled **VectorBx =** in Figure 10 (p. 15) and clicked the **OK** button.

And the result was...

You can see that the value of the dot product of the two vectors in Figure 10 (p. 15) is 0.0, and the angle between the two vectors is 90 degrees. Therefore, although the magenta vector in Figure 10 (p. 15) is much longer than the magenta vector in Figure 8 (p. 13) , the magenta vector in Figure 10 (p. 15) is still perpendicular to the black vector. Thus, Figure 8 (p. 13) and Figure 10 (p. 15) show two of the infinite number of magenta vectors that are perpendicular to the black vector in those figures.

The dot product of perpendicular vectors in 3D

As I mentioned earlier, the topic of perpendicularity in 3D is more complicated (p. 12) than is the case in 2D. As is the case in 2D, there are an infinite number of vectors that are perpendicular to a given vector in 3D. In 2D, the infinite set of perpendicular vectors must have different lengths taken in pairs, and the vectors in each pair must point in opposite directions.

An infinite number of perpendicular vectors having the same length

However, in 3D there are an infinite number of vectors having the same length that are perpendicular to a given vector. All of the perpendicular vectors having the same length must point in different directions and they must all lie in a plane that is perpendicular to the given vector.

Perpendicular vectors having different lengths may point in the same or in different directions but they also must lie in a plane that is perpendicular to the given vector.

A wagon-wheel analogy

Kjell explains the situation of an infinite set of 3D vectors that are perpendicular to a given vector by describing an old-fashioned wagon wheel with spokes that emanate directly from the hub and extend to the rim of the wheel. The hub surrounds the axle and each of the spokes is perpendicular to the axle. Depending on the thickness of the spokes, a large *(but probably not infinite)* number of spokes can be used in the construction of the wagon wheel.

Another wheel at the other end of the axle

In this case, the wagon wheel lies in a plane that is perpendicular to the axle. There is normally another wheel at the other end of the axle. Assuming that the axle is straight, the second wheel is in a different plane but that plane is also perpendicular to the axle. Thus, the spokes in the second wheel are also perpendicular to the axle.

If there were two identical wheels at each end of the axle for a total of four wheels *(the predecessor to the modern 18-wheel tractor trailer)* , the spokes in all four of the wheels would be perpendicular to the axle. Again, the point is that there are an infinite number of vectors that are perpendicular to a given vector in 3D.

A general formulation of 3D vector perpendicularity

By performing some algebraic manipulations on the earlier (p. 12) 3D formulation of the dot product, we can develop the equations shown in Figure 11 (p. 17) that describe an infinite set of vectors that are perpendicular to a given vector.

A general formulation of 3D vector perpendicularity.

```
    dot product = x1*x2 + y1*y2 + z1*z2

If the two vectors are perpendicular:

x1*x2 + y1*y2 + z1*z2 = 0.0
x1*x2 =  -(y1*y2 + z1*z2)

x2 = -(y1*y2 + z1*z2)/x1

or

y2 = -(x1*x2 + z1*z2)/y1

or

z2 = -(x1*x2 + y1*y2)/z1
```

Figure 11: A general formulation of 3D vector perpendicularity.

(Although I didn't do so in Figure 11 (p. 17) , I could have written three more equations that could be used to solve for x1, y1, and z1 if the given vector is notated as x2, y2, and z2.)

No simple case

Unlike with 2D vectors, *(to my knowledge)* , there is no case that simply allows you to swap coordinate values and change the sign on one of them to produce a vector that is perpendicular to another vector. However, given the equations in Figure 11 (p. 17) , and given values for x1, y1, and z1, we can assume values for y2 and z2 and determine the value for x2 that will cause the two vectors to be perpendicular.

While this is a fairly tedious computation with a hand calculator, it is very easy to write Java code to perform the computation. Therefore, given a 3D vector, it is fairly easy to write Java code that will compute an infinite number of vectors that are perpendicular to the given vector.

A pair of perpendicular 3D vectors

Figure 12 (p. 18) and Figure 13 (p. 19) each show a magenta 3D vector that is part of the infinite set of vectors that are all perpendicular to the black 3D vector. In Figure 12 (p. 18) , y1 was assumed to be equal to y2, and z1 was assumed to be equal to z2. For an x1 value of 25, a value of -125 was required to cause the magenta vector to be perpendicular to the black vector.

A pair of perpendicular 3D vectors.

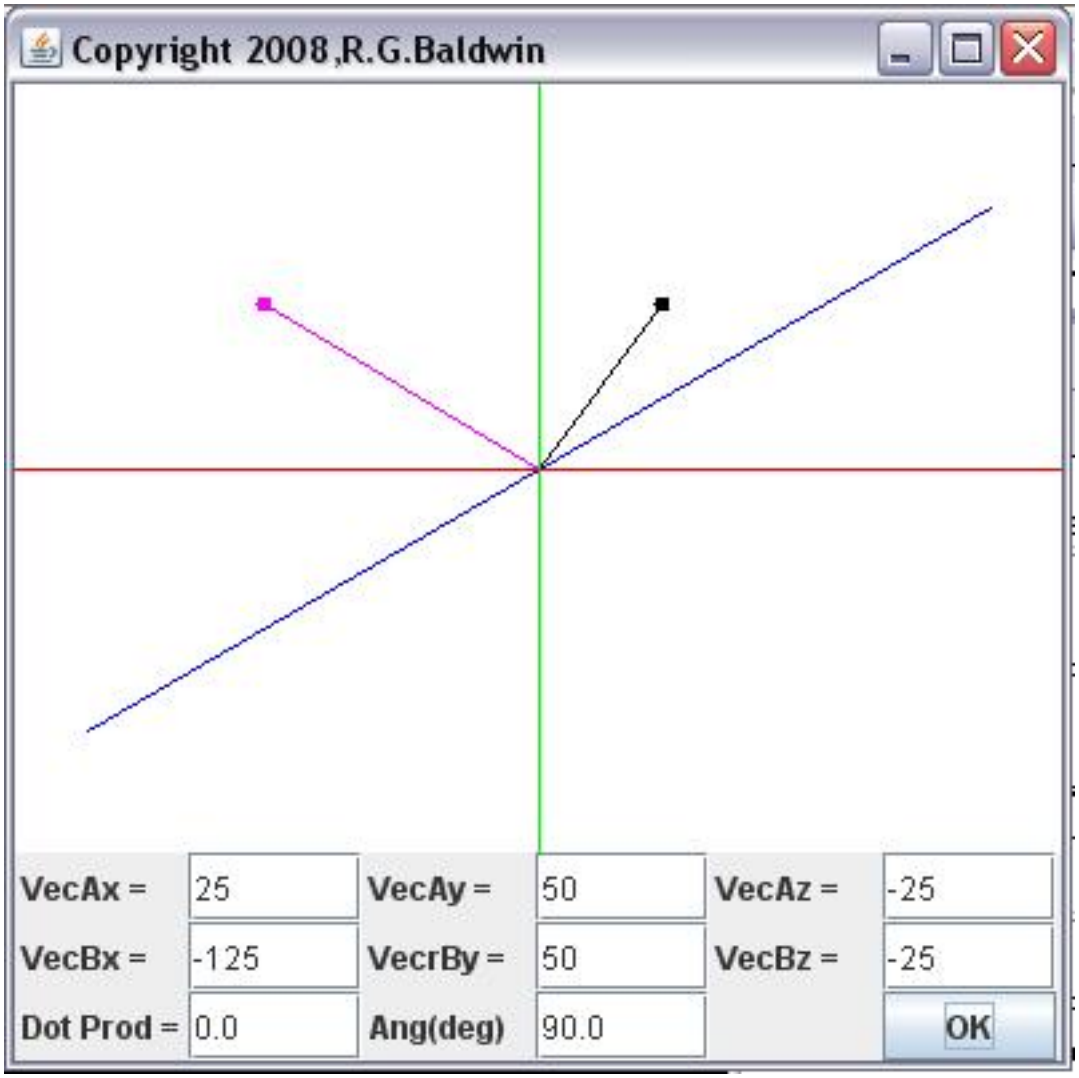

Figure 12: A pair of perpendicular 3D vectors.

Another pair of perpendicular 3D vectors

In Figure 13 (p. 19) , x1 was assumed to be equal to x2, and z1 was assumed to be equal to z2.

Another pair of perpendicular 3D vectors.

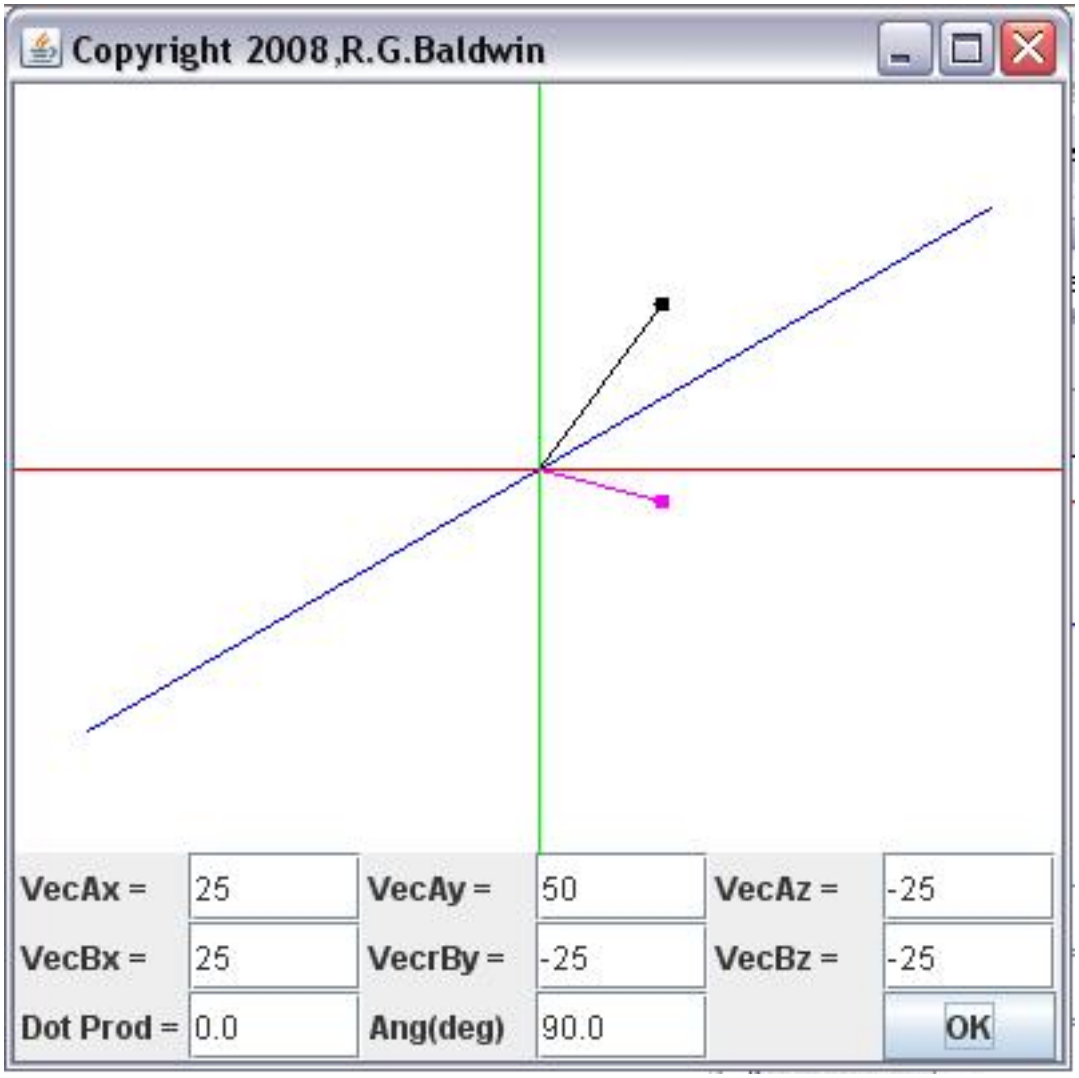

Figure 13: Another pair of perpendicular 3D vectors.

For a y1 value of 50, a y2 value of - 25 was required to cause the magenta vector to be perpendicular to the black vector.

The black vector didn't change

Note that the black vector is the same in Figure 12 (p. 18) and Figure 13 (p. 19) . However, the magenta vectors are different in Figure 12 (p. 18) and Figure 13 (p. 19) . They represent just two of the infinite set of vectors that are perpendicular to the black vector. *(They are two of the vectors in the infinite set of perpendicular vectors that are relatively easy to compute using program code.)*

Computing the angle between two vectors

The fifth property in the previous list (p. 6) reads: *The angle between two vectors is the same as the angle between normalized versions of the vectors, which is equal to the arc cosine of the dot product of the normalized vectors.*

In other words, given two vectors, we can compute the angle between the two vectors by first normalizing each vector and then computing the dot product of the two normalized vectors.

(By normalizing, I mean to change the coordinate values such that the direction of the vector remains the same but the length of the vector is converted to 1.0.)

The dot product is equal to the cosine of the angle. The actual angle can be obtained by using one of the methods of the **Math** class to compute the arc cosine of the value of the dot product.

Used to compute the output angle in the programs

This is the procedure that is used by the programs in this module to compute and display the angle between two vectors as illustrated by the contents of the output fields labeled **Ang(deg)** or **Angle (deg)** = in many of the figures in this module.

I will have more to say about this topic as I explain the code in the upgraded game-math library named **GM02** as well as the following four programs that demonstrate the use of the upgraded game-math library:

- DotProd2D01
- DotProd2D02
- DotProd3D01
- DotProd3D02

I will also provide exercises for you to complete on your own at the end of the module. The exercises will concentrate on the material that you have learned in this module and previous modules.

4 Discussion and sample code

4.1 The game-math library named GM02

In this section, I will present and explain an updated version of the game-math library named **GM02** .

This game-math library is an update to the game-math library named **GM01** . The main purpose of this update was to add *vector dot product* and related capabilities, such as the computation of the angle between two vectors to the library.

The following methods are new instance methods of the indicated static top-level classes belonging to the class named **GM02** .

- **GM02.ColMatrix2D.dot** - computes dot product of two ColMatrix2D objects.
- **GM02.Vector2D.dot** - computes dot product of two Vector2D objects.
- **GM02.Vector2D.angle** - computes angle between two Vector2D objects.
- **GM02.ColMatrix3D.dot** - computes dot product of two ColMatrix3D objects
- **GM02.Vector3D.dot** - computes dot product of two Vector3D objects.
- **GM02.Vector3D.angle** - computes angle between two Vector3D objects.

Will only explain the new 3D methods

I have explained much of the code in the game-math library in previous modules, and I won't repeat those explanations here. Rather, I will explain only the new 3D code in this module. Once you understand the new 3D code, you should have no difficulty understanding the new 2D code.

You can view a complete listing of the updated game-math library in Listing 8 (p. 30) near the end of the module.

Source code for the method named GM02.ColMatrix3D.dot

Listing 1 (p. 21) shows the source code for the new instance method named **GM02.ColMatrix3D.dot** . If you have studied the Kjell tutorials, you have learned that the dot product can be applied not only to two vectors, but can also be applied to the column matrix objects that are used to represent the vectors.

Listing 1 (p. 21) computes the dot product of two **ColMatrix3D** objects and returns the result as type double.

Listing 1: Source code for the method named GM02.ColMatrix3D.dot.

```
    public double dot(GM02.ColMatrix3D matrix){
  return getData(0) * matrix.getData(0)
      + getData(1) * matrix.getData(1)
      + getData(2) * matrix.getData(2);
}//end dot
```

This is one of those cases where it is very easy to write the code once you understand the code that you need to write. Listing 1 (p. 21) implements the equation for the 3D dot product that I provided earlier (p. 12) and returns the result of the computation as type **double** .

Source code for the method named GM02.Vector3D.dot

Listing 2 (p. 21) shows the source code for the method named **GM02.Vector3D.dot** . This method computes the dot product of two **Vector3D** objects and returns the result as type **double** .

Listing 2: Source code for the method named GM02.Vector3D.dot.

```
    public double dot(GM02.Vector3D vec){
  GM02.ColMatrix3D matrixA = getColMatrix();
  GM02.ColMatrix3D matrixB = vec.getColMatrix();
  return matrixA.dot(matrixB);
}//end dot
```

Once again, the code is straightforward. All **Vector3D** objects instantiated from classes in the game-math library are represented by objects of the **ColMatrix3D** class. Listing 2 (p. 21) gets a reference to the two **ColMatrix3D** objects that represent the two vectors for which the dot product is needed. Then it calls the **GM02.ColMatrix3D.dot** method shown earlier in Listing 1 (p. 21) to get and return the value of the dot product of the two vectors.

Source code for the method named GM02.Vector3D.angle

Listing 3 (p. 21) shows the source code for the method named **GM02.Vector3D.angle** . This method computes and returns the angle between two **Vector3D** objects. The angle is returned in degrees as type **double** .

Listing 3: Source code for the method named GM02.Vector3D.angle.

```
    public double angle(GM02.Vector3D vec){
  GM02.Vector3D normA = normalize();
  GM02.Vector3D normB = vec.normalize();
  double normDotProd = normA.dot(normB);
  return Math.toDegrees(Math.acos(normDotProd));
}//end angle
```

You need to understand trigonometry here

If you understand trigonometry, you will find the code in Listing 3 (p. 21) straightforward. If not, simply take my word for it that the method shown in Listing 3 (p. 21) behaves as described above.

The method begins by calling the **normalize** method on each of the two vectors for which the angle between the vectors is needed. *(See the definition of the* ***normalize*** *method in Listing 8 (p. 30) .)*

Then Listing 3 (p. 21) computes the dot product of the two normalized vectors. The value of the dot product is the cosine of the angle between the two original vectors.

After that, Listing 3 (p. 21) calls the **acos** method of the **Math** class to get the *arc (inverse) cosine (see Inverse Cosine* [2] *)* of the dot product value. The **acos** method returns the angle in radians.

Finally, Listing 3 (p. 21) calls the **toDegrees** method of the **Math** class to convert the angle from radians to degrees and to return the angle in degrees as type **double** .

That completes the discussion of the updates to the game-math library resulting in the new library named **GM02** .

4.2 The program named DotProd2D01

To understand this program, you need to understand the material in the Kjell tutorial through *Chapter 8 - Length, Orthogonality, and the Column Matrix Dot product* .

The purpose of this program is simply to confirm proper operation of the **GM02.ColMatrix2D.dot** method. The program output is shown in Figure 14 (p. 22) .

GUI for the program named DotProd2D01.

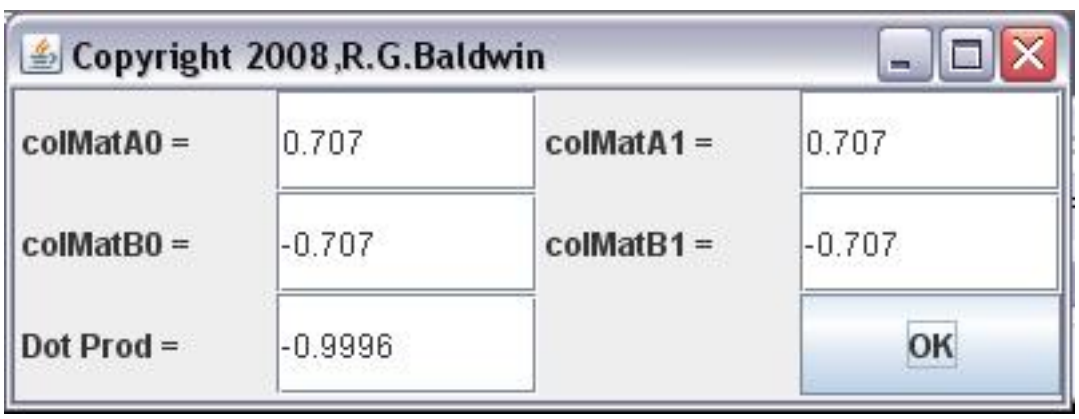

Figure 14: GUI for the program named DotProd2D01.

A graphical user interface

The program creates a GUI that allows the user to enter the first and second values for each of a pair of **ColMatrix2D** objects into four text fields. The GUI also provides a button labeled **OK** . When the user clicks the **OK** button, the dot product of the two **ColMatrix2D** objects is computed. The resulting value is formatted to four decimal digits and displayed in a text field in the lower left of the GUI.

Similar to previous programs

Much of the code in this program is similar to code that I have explained in earlier modules, so I won't repeat those explanations here. I will explain only the code contained in the **actionPerformed** method that is new to this module. A complete listing of this program is shown in Listing 9 (p. 54) .

The actionPerformed method in the program named DotProd2D01

The beginning of the **actionPerformed** method is shown in Listing 4 (p. 23) . This method is called to respond to a click on the **OK** button shown in Figure 14 (p. 22) .

[2] http://mathworld.wolfram.com/InverseCosine.html

Listing 4: The actionPerformed method in the program named DotProd2D01.

```
  public void actionPerformed(ActionEvent e){
//Create two ColMatrix2D objects.
GM02.ColMatrix2D matrixA = new GM02.ColMatrix2D(
  Double.parseDouble(colMatA0.getText()),
  Double.parseDouble(colMatA1.getText()));

GM02.ColMatrix2D matrixB = new GM02.ColMatrix2D(
  Double.parseDouble(colMatB0.getText()),
  Double.parseDouble(colMatB1.getText()));

//Compute the dot product.
double dotProd = matrixA.dot(matrixB);
```

Listing 4 (p. 23) begins by instantiating a pair of **GM02.ColMatrix2D** objects using data provided by the user in the top four fields shown in Figure 14 (p. 22) .

Then Listing 4 (p. 23) calls the **dot** method on the object referred to by **matrixA** , passing the object referred to by **matrixB** as a parameter. The **dot** method computes the dot product of the two column matrix objects, returning the result as type **double** .

Format the dot product value for display in the GUI

In some cases, the format of the returned value is not very suitable for display in the lower-left field in Figure 14 (p. 22) . For example, if the value is very small, it is returned in an exponential notation requiring a large number of digits for display. Similarly, sometimes the dot-product value is returned in a format something like 0.33333333 requiring a large number of digits to display.

Listing 5 (p. 23) formats the dot product value to make it suitable for display in the text field in Figure 14 (p. 22) . *(There may be an easier way to do this, but I didn't want to take the time to figure out what it is.)*

Listing 5: Format the dot product value for display in the GUI.

```
    //Eliminate exponential notation in the display.
if(Math.abs(dotProd) < 0.001){
  dotProd = 0.0;
}//end if

//Convert to four decimal digits and display.
dotProd =((int)(10000*dotProd))/10000.0;
dotProduct.setText("" +  dotProd);

}//end actionPerformed
```

Eliminate exponential format and format to four decimal digits

Listing 5 (p. 23) begins by simply setting small values that are less than 0.001 to 0.0 to eliminate the exponential format for very small, non-zero values.

Then Listing 5 (p. 23) executes some code that formats the dot product value to four decimal digits. I will leave it as an exercise for the student to decipher how this code does what it does.

I recommend that you try it

I recommend that you plug a few values into the input fields, click the **OK** button, and use your calculator to convince yourself that the program properly implements the 2D dot product equation shown earlier (p. 12) .

That concludes the discussion of the program named **DotProd2D01** .

4.3 The program named DotProd2D02

To understand this program, you need to understand the material in the Kjell tutorial through *Chapter 9, The Angle Between Two Vectors* .

This program allows the user to experiment with the dot product and the angle between a pair of **GM02.Vector2D** objects.

A screen shot of the output from this program is shown in Figure 3 (p. 5) . The GUI shown in Figure 3 (p. 5) is provided to allow the user to enter four double values that define each of two **GM02.Vector2D** objects. The GUI also provides an **OK** button as well as two text fields used for display of computed results.

In addition, the GUI provides a 2D drawing area. When the user clicks the **OK** button, the program draws the two vectors, *(one in black and the other in magenta)* , on the output screen with the tail of each vector located at the origin in 2D space. The program also displays the values of the dot product of the two vectors and the angle between the two vectors in degrees.

Once again, much of the code in this program is similar to code that I have explained before. I will explain only the method named **actionPerformed** , for which some of the code is new to this module. A complete listing of this program is provided in Listing 10 (p. 56) .

Beginning of the actionPerformed method in the program named DotProd2D02

This method is called to respond to a click on the **OK** button in Figure 2.

Listing 6: Beginning of the actionPerformed method in the program named DotProd2D02.

```
  public void actionPerformed(ActionEvent e){

//Erase the off-screen image and draw the axes.
setCoordinateFrame(g2D);

//Create two ColMatrix2D objects based on the user
// input values.
GM02.ColMatrix2D matrixA = new GM02.ColMatrix2D(
              Double.parseDouble(vectorAx.getText()),
              Double.parseDouble(vectorAy.getText()));

GM02.ColMatrix2D matrixB = new GM02.ColMatrix2D(
              Double.parseDouble(vectorBx.getText()),
              Double.parseDouble(vectorBy.getText()));


//Use the ColMatrix2D objects to create two Vector2D
// objects.
GM02.Vector2D vecA = new GM02.Vector2D(matrixA);
GM02.Vector2D vecB = new GM02.Vector2D(matrixB);

//Draw the two vectors with their tails at the origin.
g2D.setColor(Color.BLACK);
vecA.draw(
     g2D,new GM02.Point2D(new GM02.ColMatrix2D(0,0)));

g2D.setColor(Color.MAGENTA);
vecB.draw(
```

```
        g2D,new GM02.Point2D(new GM02.ColMatrix2D(0,0)));
```

Listing 6 (p. 24) gets the four user input values that define the two vectors and draws them in black and magenta on the GUI shown in Figure 3 (p. 5) . There is nothing new in the code in Listing 6 (p. 24) .

Compute the dot product and the angle between the two vectors

Listing 7 (p. 25) computes the dot product and the angle between the two vectors by first calling the **dot** method and then the **angle** method on the object referred to by **vecA** , passing the object referred to by **vecB** as a parameter.

Listing 7: Compute the dot product and the angle between the two vectors.

```
        //Compute the dot product of the two vectors.
    double dotProd = vecA.dot(vecB);

//Output formatting code was deleted for brevity

    //Compute the angle between the two vectors.
    double angle = vecA.angle(vecB);

//Output formatting code was deleted for brevity

    myCanvas.repaint();//Copy off-screen image to canvas.
  }//end actionPerformed
```

In both cases, the code in Listing 7 (p. 25) formats the returned **double** values to make them appropriate for display in the bottom two text fields in Figure 3 (p. 5) . This code was deleted from Listing 7 (p. 25) for brevity.

Confirm that the results are correct

Because this is a 2D display, it is easy to make an eyeball comparison between the drawing of the two vectors and the reported angle between the two vectors to confirm agreement. However, I recommend that you use this program to define several vectors and then use your scientific calculator to confirm that the results shown are correct.

That concludes the explanation of the program named **DotProd2D02** .

4.4 The program named DotProd3D01

To understand this program, you need to understand the material in the Kjell tutorial through *Chapter 8 - Length, Orthogonality, and the Column Matrix Dot product* .

The purpose of this program is to confirm proper operation of the **ColMatrix3D.dot** method. The program creates a GUI that allows the user to enter three values for each of a pair of **ColMatrix3D** objects along with a button labeled **OK** . When the user clicks the **OK** button, the dot product of the two **ColMatrix3D** objects is computed and displayed.

A screen shot of the output from the program named DotProd3D01

A screen shot of the output from the program named **DotProd3D01** is shown in Figure 15 (p. 26) . *(Compare Figure 15 (p. 26) with Figure 14 (p. 22) .)*

A screen shot of the output from the program named DotProd3D01.

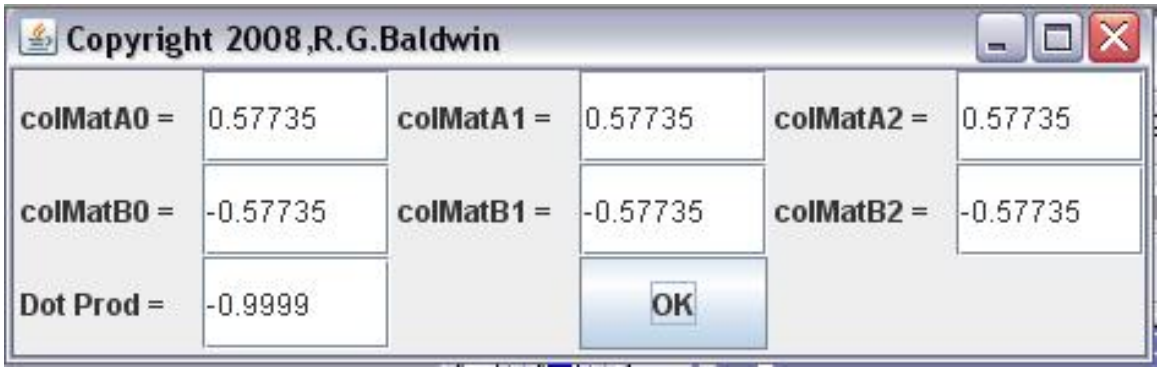

Figure 15: A screen shot of the output from the program named DotProd3D01.

Very similar to a previous program

Except for the fact that this program calls the **dot** method on an object of the **GM02.ColMatrix3D** class instead calling the **dot** method on an object of the **GM02.ColMatrix2D** class, this program is essentially the same at the program named **DotProd2D01** that I explained earlier. Therefore, you should have no trouble understanding this program without further explanation from me. A complete listing of this program is provided in Listing 11 (p. 61) .

That concludes the explanation of the program named **DotProd3D01** .

4.5 The program named DotProd3D02

You need to understand the material in the Kjell tutorial through *Chapter 10, Angle between 3D Vectors* to understand this program.

Program output

A screen shot of the output of this program is shown in Figure 12 (p. 18) . This program allows the user to experiment with the dot product and the angle between a pair of **GM02.Vector3D** objects. A GUI is provided that allows the user to enter six **double** values that define each of two **GM02.Vector3D** objects. The GUI also provides an **OK** button as well as two text fields used for display of the computed results.

In addition, the GUI provides a 3D drawing area. When the user clicks the **OK** button, the program draws the two vectors, *(one in black and the other in magenta)* , on the output screen with the tail of each vector located at the origin in 3D space. The program also displays the values of the dot product of the two vectors and the angle between the two vectors in degrees. *(Compare the output of this 3D program in Figure 12 (p. 18) with the output from the 2D program in Figure 3 (p. 5) .)*

Program code

A complete listing of this program is provided in Listing 12 (p. 64) . Almost all of the new and interesting code in this program is in the method named **actionPerformed** .

Very similar to a previous program

If you compare the method named **actionPerformed** in Listing 12 (p. 64) with the **actionPerformed** method for the program named **DotProd2D02** in Listing 10 (p. 56) , you will see that they are very similar. One calls 2D methods in the game-math library while the other calls 3D methods in the same

library. Therefore, you should have no difficulty understanding this program without further explanation from me.

That concludes the explanation of the program named **DotProd3D02** .

4.6 Interpreting the vector dot product

In effect, the dot product of two vectors provides a measure of the extent to which the two vectors have the same orientation. If the two vectors are parallel, the dot product of the two vectors has a maximum value of 1.0 multiplied by the product of the lengths of the two vectors. Normalizing the vectors before computing the dot product will eliminate the effect of the vector lengths and will cause the results to be somewhat easier to interpret.

If the normalized vectors are parallel and point in the same direction, the dot product of the normalized vectors will be 1.0. If the normalized vectors are parallel and point in opposite directions, the value of the dot product will be -1.0. If the vectors are perpendicular, the dot product of the two vectors will be 0.0 regardless of whether or not they are normalized.

For all orientations, the value of the dot product of normalized vectors will vary between -1.0 and +1.0.

4.7 More than three dimensions

You may have it in your mind that the use of mathematical concepts such as the vector dot product are limited to the three dimensions of width, height, and depth. If so, that is a false impression. The vector dot product is very useful for systems with more than three dimensions. In fact, engineers and scientists deal with systems every day that have more than three dimensions. While it usually isn't too difficult to handle the math involved in such systems, we have a very hard time drawing pictures of systems with more than three or four dimensions.

Digital convolution is a good example

I have spent much of my career in digital signal processing where I have routinely dealt with systems having thirty or forty dimensions. For example, in the module titled *Convolution and Frequency Filtering in Java* [3] and some other related modules as well, I describe the process of digital *convolution filtering* .

One way to think of digital convolution is that it is a running dot product computation between one vector *(the convolution filter)* and a series of other vectors, each comprised of successive chunks of samples from the incoming data. In non-adaptive systems, the vector that represents the convolution filter usually has a set of fixed coordinate values, and there may be dozens and even hundreds of such coordinates. *(For an adaptive system, the values that define the vector typically change as a function of time or some other parameter.)*

Often, the input data will consist of samples taken from a channel consisting of signal plus noise. The objective is often to design a vector *(the convolution filter)* that is parallel to all of the signal components in the incoming data and is perpendicular to all of the noise components in the incoming data. If that objective is achieved, the noise will be suppressed while the signal will be passed through to the output.

5 Documentation for the GM02 library

Click here [4] to download a zip file containing standard javadoc documentation for the library named **GM02** . Extract the contents of the zip file into an empty folder and open the file named **index.html** in your browser to view the documentation.

Although the documentation doesn't provide much in the way of explanatory text *(see Listing 8 (p. 30) and the explanations given above)* , the documentation does provide a good overview of the organization and structure of the library. You may find it helpful in that regard.

[3] http://www.developer.com/java/other/article.php/3484591/Convolution-and-Frequency-Filtering-in-Java.htm
[4] http://cnx.org/content/m45018/1.1/GM02docs.zip

6 Homework assignment

The homework assignment for this module was to study the Kjell tutorial through *Chapter 10, Angle between 3D Vectors* .

The homework assignment for the next module is to continue studying that same material.

In addition to studying the Kjell material, you should read at least the next two modules in this collection and bring your questions about that material to the next classroom session.

Finally, you should have begun studying the physics material [5] at the beginning of the semester and you should continue studying one physics module per week thereafter. You should also feel free to bring your questions about that material to the classroom for discussion.

7 Run the programs

I encourage you to copy the code from Listing 8 (p. 30) through Listing 12 (p. 64) . Compile the code and execute it in conjunction with the game-math library named **GM02** provided in Listing 8 (p. 30) . Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

8 Summary

In this module, you learned the fundamentals of the vector dot product in both 2D and 3D. You learned how to update the game-math library to support various aspects of the vector dot product, and you learned how to write 2D and 3D programs that use the vector dot product methods in the game-math library.

9 What's next?

In the next module, which will be the second part of this two-part miniseries on the vector dot product, you will learn how to use the dot product to compute nine different angles of interest that a vector makes with various elements in 3D space.

You will learn how to use the dot product to find six of the infinite set of vectors that are perpendicular to a given vector as shown in Figure 16 (p. 29) .

[5] http://cnx.org/content/m44992/latest/

Six (magenta) vectors that are perpendicular to a given (black) vector.

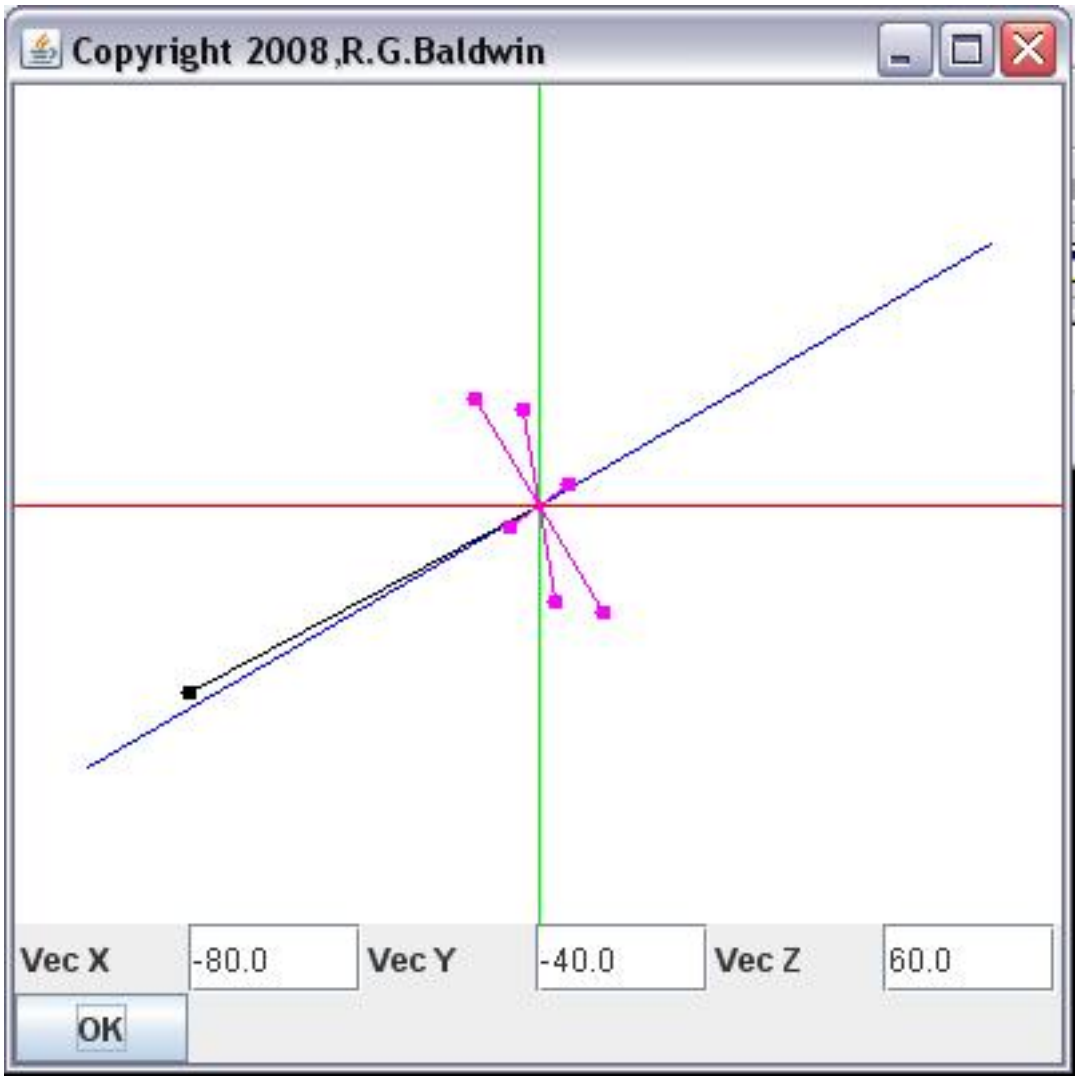

Figure 16: Six (magenta) vectors that are perpendicular to a given (black) vector.

You will also learn how to use the dot product to perform back-face culling as shown in Figure 1 (p. 2) and Figure 2 (p. 3) .

10 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: GAME 2302-0145: Getting Started with the Vector Dot Product
- File: Game0145.htm
- Published: 10/21/12

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

11 Complete program listings

Complete listings of the programs discussed in this module are shown in Listing 8 (p. 30) through Listing 12 (p. 64) below

Listing 8: Source code for the game-math library named GM02.

```
    /*GM02.java
Copyright 2008, R.G.Baldwin
Revised 02/08/08

This is an update to the game-math library named GM01.

The main purpose of this update was to add vector dot
product and related capabilities to the library.

Please see the comments at the beginning of the library
class named GM01 for a general description of the library.

The following methods are new instance methods of the
indicated static top-level classes belonging to the class
named GM02.

GM02.ColMatrix2D.dot - compute dot product of two
 ColMatrix2D objects.
GM02.Vector2D.dot - compute dot product of two
 Vector2D objects.
GM02.Vector2D.angle - compute angle between two Vector2D
 objects.

GM02.ColMatrix3D.dot - compute dot product of two
```

```
 ColMatrix3D objects
GM02.Vector3D.dot - compute dot product of two
 Vector3D objects.
GM02.Vector3D.angle - compute angle between two Vector3D
 objects.

Tested using JDK 1.6 under WinXP.
*******************************************************/
import java.awt.geom.*;
import java.awt.*;

public class GM02{
  //-----------------------------------------------------//

  //This method converts a ColMatrix3D obj to a
  // ColMatrix2D object. The purpose is to accept
  // x, y, and z coordinate values and transform those
  // values into a pair of coordinate values suitable for
  // display in two dimensions.
  //See http://local.wasp.uwa.edu.au/~pbourke/geometry/
  // classification/ for technical background on the
  // transform from 3D to 2D.
  //The transform equations are:
  // x2d = x3d + z3d * cos(theta)/tan(alpha)
  // y2d = y3d + z3d * sin(theta)/tan(alpha);
  //Let theta = 30 degrees and alpha = 45 degrees
  //Then:cos(theta) = 0.866
  //     sin(theta) = 0.5
  //     tan(alpha) = 1;
  //Note that the signs in the above equations depend
  // on the assumed directions of the angles as well as
  // the assumed positive directions of the axes. The
  // signs used in this method assume the following:
  //   Positive x is to the right.
  //   Positive y is up the screen.
  //   Positive z is protruding out the front of the
  //     screen.
  //   The viewing position is above the x axis and to the
  //     right of the z-y plane.
  public static GM02.ColMatrix2D convert3Dto2D(
                                 GM02.ColMatrix3D data){
    return new GM02.ColMatrix2D(
                 data.getData(0) - 0.866*data.getData(2),
                 data.getData(1) - 0.50*data.getData(2));
  }//end convert3Dto2D
  //-----------------------------------------------------//

  //This method wraps around the translate method of the
  // Graphics2D class. The purpose is to cause the
  // positive direction for the y-axis to be up the screen
  // instead of down the screen. When you use this method,
```

```
  // you should program as though the positive direction
  // for the y-axis is up.
  public static void translate(Graphics2D g2D,
                               double xOffset,
                               double yOffset){
    //Flip the sign on the y-coordinate to change the
    // direction of the positive y-axis to go up the
    // screen.
    g2D.translate(xOffset,-yOffset);
  }//end translate
  //-----------------------------------------------------//

  //This method wraps around the drawLine method of the
  // Graphics class. The purpose is to cause the positive
  // direction for the y-axis to be up the screen instead
  // of down the screen. When you use this method, you
  // should program as though the positive direction for
  // the y-axis is up.
  public static void drawLine(Graphics2D g2D,
                              double x1,
                              double y1,
                              double x2,
                              double y2){
    //Flip the sign on the y-coordinate value.
    g2D.drawLine((int)x1,-(int)y1,(int)x2,-(int)y2);
  }//end drawLine
  //-----------------------------------------------------//

  //This method wraps around the fillOval method of the
  // Graphics class. The purpose is to cause the positive
  // direction for the y-axis to be up the screen instead
  // of down the screen. When you use this method, you
  // should program as though the positive direction for
  // the y-axis is up.
  public static void fillOval(Graphics2D g2D,
                              double x,
                              double y,
                              double width,
                              double height){
    //Flip the sign on the y-coordinate value.
    g2D.fillOval((int)x,-(int)y,(int)width,(int)height);
  }//end fillOval
  //-----------------------------------------------------//

  //This method wraps around the drawOval method of the
  // Graphics class. The purpose is to cause the positive
  // direction for the y-axis to be up the screen instead
  // of down the screen. When you use this method, you
  // should program as though the positive direction for
  // the y-axis is up.
  public static void drawOval(Graphics2D g2D,
```

```
                              double x,
                              double y,
                              double width,
                              double height){
    //Flip the sign on the y-coordinate value.
    g2D.drawOval((int)x,-(int)y,(int)width,(int)height);
  }//end drawOval
  //-----------------------------------------------------//

  //This method wraps around the fillRect method of the
  // Graphics class. The purpose is to cause the positive
  // direction for the y-axis to be up the screen instead
  // of down the screen. When you use this method, you
  // should program as though the positive direction for
  // the y-axis is up.
  public static void fillRect(Graphics2D g2D,
                              double x,
                              double y,
                              double width,
                              double height){
    //Flip the sign on the y-coordinate value.
    g2D.fillRect((int)x,-(int)y,(int)width,(int)height);
  }//end fillRect
  //-----------------------------------------------------//



  //An object of this class represents a 2D column matrix.
  // An object of this class is the fundamental building
  // block for several of the other classes in the
  // library.
  public static class ColMatrix2D{
    double[] data = new double[2];

    public ColMatrix2D(double data0,double data1){
      data[0] = data0;
      data[1] = data1;
    }//end constructor
    //-------------------------------------------------//

    //Overridden toString method.
    public String toString(){
      return data[0] + "," + data[1];
    }//end overridden toString method
    //-------------------------------------------------//

    public double getData(int index){
      if((index < 0) || (index > 1)){
        throw new IndexOutOfBoundsException();
      }else{
        return data[index];
```

```
    }//end else
  }//end getData method
  //-----------------------------------------------------//

  public void setData(int index,double data){
    if((index < 0) || (index > 1)){
      throw new IndexOutOfBoundsException();
    }else{
      this.data[index] = data;
    }//end else
  }//end setData method
  //-----------------------------------------------------//

  //This method overrides the equals method inherited
  // from the class named Object. It compares the values
  // stored in two matrices and returns true if the
  // values are equal or almost equal and returns false
  // otherwise.
  public boolean equals(Object obj){
    if(obj instanceof GM02.ColMatrix2D &&
       Math.abs(((GM02.ColMatrix2D)obj).getData(0) -
                                 getData(0)) <= 0.00001 &&
       Math.abs(((GM02.ColMatrix2D)obj).getData(1) -
                                  getData(1)) <= 0.00001){
      return true;
    }else{
      return false;
    }//end else

  }//end overridden equals method
  //-----------------------------------------------------//

  //Adds one ColMatrix2D object to another ColMatrix2D
  // object, returning a ColMatrix2D object.
  public GM02.ColMatrix2D add(GM02.ColMatrix2D matrix){
    return new GM02.ColMatrix2D(
                          getData(0)+matrix.getData(0),
                          getData(1)+matrix.getData(1));
  }//end add
  //-----------------------------------------------------//

  //Subtracts one ColMatrix2D object from another
  // ColMatrix2D object, returning a ColMatrix2D object.
  // The object that is received as an incoming
  // parameter  is subtracted from the object on which
  // the method is called.
  public GM02.ColMatrix2D subtract(
                                 GM02.ColMatrix2D matrix){
    return new GM02.ColMatrix2D(
                          getData(0)-matrix.getData(0),
                          getData(1)-matrix.getData(1));
```

```
    }//end subtract
    //-----------------------------------------------------//

    //Computes the dot product of two ColMatrix2D
    // objects and returns the result as type double.
    public double dot(GM02.ColMatrix2D matrix){
      return getData(0) * matrix.getData(0)
           + getData(1) * matrix.getData(1);
    }//end dot
    //-----------------------------------------------------//
  }//end class ColMatrix2D
  //=====================================================//


  //An object of this class represents a 3D column matrix.
  // An object of this class is the fundamental building
  // block for several of the other classes in the
  // library.
  public static class ColMatrix3D{
    double[] data = new double[3];

    public ColMatrix3D(
                  double data0,double data1,double data2){
      data[0] = data0;
      data[1] = data1;
      data[2] = data2;
    }//end constructor
    //-----------------------------------------------------//

    public String toString(){
      return data[0] + "," + data[1] + "," + data[2];
    }//end overridden toString method
    //-----------------------------------------------------//

    public double getData(int index){
      if((index < 0) || (index > 2)){
        throw new IndexOutOfBoundsException();
      }else{
        return data[index];
      }//end else
    }//end getData method
    //-----------------------------------------------------//

    public void setData(int index,double data){
      if((index < 0) || (index > 2)){
        throw new IndexOutOfBoundsException();
      }else{
        this.data[index] = data;
      }//end else
    }//end setData method
    //-----------------------------------------------------//
```

```
//This method overrides the equals method inherited
// from the class named Object. It compares the values
// stored in two matrices and returns true if the
// values are equal or almost equal and returns false
// otherwise.
public boolean equals(Object obj){
  if(obj instanceof GM02.ColMatrix3D &&
     Math.abs(((GM02.ColMatrix3D)obj).getData(0) -
                             getData(0)) <= 0.00001 &&
     Math.abs(((GM02.ColMatrix3D)obj).getData(1) -
                             getData(1)) <= 0.00001 &&
     Math.abs(((GM02.ColMatrix3D)obj).getData(2) -
                              getData(2)) <= 0.00001){
    return true;
  }else{
    return false;
  }//end else

}//end overridden equals method
//-----------------------------------------------------//

//Adds one ColMatrix3D object to another ColMatrix3D
// object, returning a ColMatrix3D object.
public GM02.ColMatrix3D add(GM02.ColMatrix3D matrix){
  return new GM02.ColMatrix3D(
                        getData(0)+matrix.getData(0),
                        getData(1)+matrix.getData(1),
                        getData(2)+matrix.getData(2));
}//end add
//-----------------------------------------------------//

//Subtracts one ColMatrix3D object from another
// ColMatrix3D object, returning a ColMatrix3D object.
// The object that is received as an incoming
// parameter is subtracted from the object on which
// the method is called.
public GM02.ColMatrix3D subtract(
                             GM02.ColMatrix3D matrix){
  return new GM02.ColMatrix3D(
                        getData(0)-matrix.getData(0),
                        getData(1)-matrix.getData(1),
                        getData(2)-matrix.getData(2));
}//end subtract
//-----------------------------------------------------//

//Computes the dot product of two ColMatrix3D
// objects and returns the result as type double.
public double dot(GM02.ColMatrix3D matrix){
  return getData(0) * matrix.getData(0)
       + getData(1) * matrix.getData(1)
```

```
          + getData(2) * matrix.getData(2);
    }//end dot
    //-----------------------------------------------------//
  }//end class ColMatrix3D
  //=====================================================//
  //=====================================================//


  public static class Point2D{
    GM02.ColMatrix2D point;

    public Point2D(GM02.ColMatrix2D point){//constructor
      //Create and save a clone of the ColMatrix2D object
      // used to define the point to prevent the point
      // from being corrupted by a later change in the
      // values stored in the original ColMatrix2D object
      // through use of its set method.
      this.point = new ColMatrix2D(
                       point.getData(0),point.getData(1));
    }//end constructor
    //-----------------------------------------------------//

    public String toString(){
      return point.getData(0) + "," + point.getData(1);
    }//end toString
    //-----------------------------------------------------//

    public double getData(int index){
      if((index < 0) || (index > 1)){
        throw new IndexOutOfBoundsException();
      }else{
        return point.getData(index);
      }//end else
    }//end getData
    //-----------------------------------------------------//

    public void setData(int index,double data){
      if((index < 0) || (index > 1)){
        throw new IndexOutOfBoundsException();
      }else{
        point.setData(index,data);
      }//end else
    }//end setData
    //-----------------------------------------------------//

    //This method draws a small circle around the location
    // of the point on the specified graphics context.
    public void draw(Graphics2D g2D){
      drawOval(g2D,getData(0)-3,
                   getData(1)+3,6,6);
    }//end draw
```

```
//-------------------------------------------------//

//Returns a reference to the ColMatrix2D object that
// defines this Point2D object.
public GM02.ColMatrix2D getColMatrix(){
  return point;
}//end getColMatrix
//-------------------------------------------------//

//This method overrides the equals method inherited
// from the class named Object. It compares the values
// stored in the ColMatrix2D objects that define two
// Point2D objects and returns true if they are equal
// and false otherwise.
public boolean equals(Object obj){
  if(point.equals(((GM02.Point2D)obj).
                                    getColMatrix())){
    return true;
  }else{
    return false;
  }//end else

}//end overridden equals method
//-------------------------------------------------//

//Gets a displacement vector from one Point2D object
// to a second Point2D object. The vector points from
// the object on which the method is called to the
// object passed as a parameter to the method. Kjell
// describes this as the distance you would have to
// walk along the x and then the y axes to get from
// the first point to the second point.
public GM02.Vector2D getDisplacementVector(
                                  GM02.Point2D point){
  return new GM02.Vector2D(new GM02.ColMatrix2D(
                        point.getData(0)-getData(0),
                        point.getData(1)-getData(1)));
}//end getDisplacementVector
//-------------------------------------------------//

//Adds a Vector2D to a Point2D producing a
// new Point2D.
public GM02.Point2D addVectorToPoint(
                                  GM02.Vector2D vec){
  return new GM02.Point2D(new GM02.ColMatrix2D(
                      getData(0) + vec.getData(0),
                      getData(1) + vec.getData(1)));
}//end addVectorToPoint
//-------------------------------------------------//
```

```
//Returns a new Point2D object that is a clone of
// the object on which the method is called.
public Point2D clone(){
  return new Point2D(
              new ColMatrix2D(getData(0),getData(1)));
}//end clone
//-----------------------------------------------------//

//The purpose of this method is to rotate a point
// around a specified anchor point in the x-y plane.
//The rotation angle is passed in as a double value
// in degrees with the positive angle of rotation
// being counter-clockwise.
//This method does not modify the contents of the
// Point2D object on which the method is called.
// Rather, it uses the contents of that object to
// instantiate, rotate, and return a new Point2D
// object.
//For simplicity, this method translates the
// anchorPoint to the origin, rotates around the
// origin, and then translates back to the
// anchorPoint.
/*
See http://www.ia.hiof.no/~borres/cgraph/math/threed/
p-threed.html for a definition of the equations
required to do the rotation.

x2 = x1*cos - y1*sin
y2 = x1*sin + y1*cos
*/
public GM02.Point2D rotate(GM02.Point2D anchorPoint,
                           double angle){
  GM02.Point2D newPoint = this.clone();

  double tempX ;
  double tempY;

  //Translate anchorPoint to the origin
  GM02.Vector2D tempVec =
        new GM02.Vector2D(anchorPoint.getColMatrix());
  newPoint =
          newPoint.addVectorToPoint(tempVec.negate());

  //Rotate around the origin.
  tempX = newPoint.getData(0);
  tempY = newPoint.getData(1);
  newPoint.setData(//new x coordinate
                  0,
                  tempX*Math.cos(angle*Math.PI/180) -
                  tempY*Math.sin(angle*Math.PI/180));
```

```
      newPoint.setData(//new y coordinate
                       1,
                       tempX*Math.sin(angle*Math.PI/180) +
                       tempY*Math.cos(angle*Math.PI/180));

      //Translate back to anchorPoint
      newPoint = newPoint.addVectorToPoint(tempVec);

      return newPoint;

    }//end rotate
    //-----------------------------------------------------//

    //Multiplies this point by a scaling matrix received
    // as an incoming parameter and returns the scaled
    // point.
    public GM02.Point2D scale(GM02.ColMatrix2D scale){
      return new GM02.Point2D(new ColMatrix2D(
                          getData(0) * scale.getData(0),
                          getData(1) * scale.getData(1)));
    }//end scale
    //-----------------------------------------------------//
  }//end class Point2D
  //=====================================================//


  public static class Point3D{
    GM02.ColMatrix3D point;

    public Point3D(GM02.ColMatrix3D point){//constructor
      //Create and save a clone of the ColMatrix3D object
      // used to define the point to prevent the point
      // from being corrupted by a later change in the
      // values stored in the original ColMatrix3D object
      // through use of its set method.
      this.point =
         new ColMatrix3D(point.getData(0),
                         point.getData(1),
                         point.getData(2));
    }//end constructor
    //-----------------------------------------------------//

    public String toString(){
      return point.getData(0) + "," + point.getData(1)
                                  + "," + point.getData(2);
    }//end toString
    //-----------------------------------------------------//

    public double getData(int index){
      if((index < 0) || (index > 2)){
        throw new IndexOutOfBoundsException();
```

```
    }else{
      return point.getData(index);
    }//end else
  }//end getData
  //-----------------------------------------------------//

  public void setData(int index,double data){
    if((index < 0) || (index > 2)){
      throw new IndexOutOfBoundsException();
    }else{
      point.setData(index,data);
    }//end else
  }//end setData
  //-----------------------------------------------------//

  //This method draws a small circle around the location
  // of the point on the specified graphics context.
  public void draw(Graphics2D g2D){

    //Get 2D projection coordinate values.
    ColMatrix2D temp = convert3Dto2D(point);
    drawOval(g2D,temp.getData(0)-3,
                 temp.getData(1)+3,
                 6,
                 6);
  }//end draw
  //-----------------------------------------------------//

  //Returns a reference to the ColMatrix3D object that
  // defines this Point3D object.
  public GM02.ColMatrix3D getColMatrix(){
    return point;
  }//end getColMatrix
  //-----------------------------------------------------//

  //This method overrides the equals method inherited
  // from the class named Object. It compares the values
  // stored in the ColMatrix3D objects that define two
  // Point3D objects and returns true if they are equal
  // and false otherwise.
  public boolean equals(Object obj){
    if(point.equals(((GM02.Point3D)obj).
                                      getColMatrix())){
      return true;
    }else{
      return false;
    }//end else

  }//end overridden equals method
  //-----------------------------------------------------//
```

```
//Gets a displacement vector from one Point3D object
// to a second Point3D object. The vector points from
// the object on which the method is called to the
// object passed as a parameter to the method. Kjell
// describes this as the distance you would have to
// walk along the x and then the y axes to get from
// the first point to the second point.
public GM02.Vector3D getDisplacementVector(
                                GM02.Point3D point){
  return new GM02.Vector3D(new GM02.ColMatrix3D(
                      point.getData(0)-getData(0),
                      point.getData(1)-getData(1),
                      point.getData(2)-getData(2)));
}//end getDisplacementVector
//-----------------------------------------------------//

//Adds a Vector3D to a Point3D producing a
// new Point3D.
public GM02.Point3D addVectorToPoint(
                                GM02.Vector3D vec){
  return new GM02.Point3D(new GM02.ColMatrix3D(
                     getData(0) + vec.getData(0),
                     getData(1) + vec.getData(1),
                     getData(2) + vec.getData(2)));
}//end addVectorToPoint
//-----------------------------------------------------//

//Returns a new Point3D object that is a clone of
// the object on which the method is called.
public Point3D clone(){
  return new Point3D(new ColMatrix3D(getData(0),
                                     getData(1),
                                     getData(2)));
}//end clone
//-----------------------------------------------------//

//The purpose of this method is to rotate a point
// around a specified anchor point in the following
// order:
// Rotate around z - rotation in x-y plane.
// Rotate around x - rotation in y-z plane.
// Rotate around y - rotation in x-z plane.
//The rotation angles are passed in as double values
// in degrees (based on the right-hand rule) in the
// order given above, packaged in an object of the
// class GM02.ColMatrix3D. (Note that in this case,
// the ColMatrix3D object is simply a convenient
// container and it has no significance from a matrix
// viewpoint.)
//The right-hand rule states that if you point the
// thumb of your right hand in the positive direction
```

```
// of an axis, the direction of positive rotation
// around that axis is given by the direction that
// your fingers will be pointing.
//This method does not modify the contents of the
// Point3D object on which the method is called.
// Rather, it uses the contents of that object to
// instantiate, rotate, and return a new Point3D
// object.
//For simplicity, this method translates the
// anchorPoint to the origin, rotates around the
// origin, and then translates back to the
// anchorPoint.
/*
See http://www.ia.hiof.no/~borres/cgraph/math/threed/
p-threed.html for a definition of the equations
required to do the rotation.
z-axis
x2 = x1*cos - y1*sin
y2 = x1*sin + y1*cos

x-axis
y2 = y1*cos(v) - z1*sin(v)
z2 = y1*sin(v) + z1* cos(v)

y-axis
x2 = x1*cos(v) + z1*sin(v)
z2 = -x1*sin(v) + z1*cos(v)
*/
public GM02.Point3D rotate(GM02.Point3D anchorPoint,
                           GM02.ColMatrix3D angles){
  GM02.Point3D newPoint = this.clone();

  double tempX ;
  double tempY;
  double tempZ;

  //Translate anchorPoint to the origin
  GM02.Vector3D tempVec =
        new GM02.Vector3D(anchorPoint.getColMatrix());
  newPoint =
          newPoint.addVectorToPoint(tempVec.negate());

  double zAngle = angles.getData(0);
  double xAngle = angles.getData(1);
  double yAngle = angles.getData(2);

  //Rotate around z-axis
  tempX = newPoint.getData(0);
  tempY = newPoint.getData(1);
  newPoint.setData(//new x coordinate
                  0,
```

```
                    tempX*Math.cos(zAngle*Math.PI/180) -
                    tempY*Math.sin(zAngle*Math.PI/180));

    newPoint.setData(//new y coordinate
                    1,
                    tempX*Math.sin(zAngle*Math.PI/180) +
                    tempY*Math.cos(zAngle*Math.PI/180));

    //Rotate around x-axis
    tempY = newPoint.getData(1);
    tempZ = newPoint.getData(2);
    newPoint.setData(//new y coordinate
                    1,
                    tempY*Math.cos(xAngle*Math.PI/180) -
                    tempZ*Math.sin(xAngle*Math.PI/180));

    newPoint.setData(//new z coordinate
                    2,
                    tempY*Math.sin(xAngle*Math.PI/180) +
                    tempZ*Math.cos(xAngle*Math.PI/180));

    //Rotate around y-axis
    tempX = newPoint.getData(0);
    tempZ = newPoint.getData(2);
    newPoint.setData(//new x coordinate
                    0,
                    tempX*Math.cos(yAngle*Math.PI/180) +
                    tempZ*Math.sin(yAngle*Math.PI/180));

    newPoint.setData(//new z coordinate
                   2,
                   -tempX*Math.sin(yAngle*Math.PI/180) +
                   tempZ*Math.cos(yAngle*Math.PI/180));

    //Translate back to anchorPoint
    newPoint = newPoint.addVectorToPoint(tempVec);

    return newPoint;

  }//end rotate
  //-----------------------------------------------------//

  //Multiplies this point by a scaling matrix received
  // as an incoming parameter and returns the scaled
  // point.
  public GM02.Point3D scale(GM02.ColMatrix3D scale){
    return new GM02.Point3D(new ColMatrix3D(
                         getData(0) * scale.getData(0),
                         getData(1) * scale.getData(1),
                         getData(2) * scale.getData(2)));
  }//end scale
```

```
    //-----------------------------------------------------//
  }//end class Point3D
  //=====================================================//
  //=====================================================//


  public static class Vector2D{
    GM02.ColMatrix2D vector;

    public Vector2D(GM02.ColMatrix2D vector){//constructor
      //Create and save a clone of the ColMatrix2D object
      // used to define the vector to prevent the vector
      // from being corrupted by a later change in the
      // values stored in the original ColVector2D object.
      this.vector = new ColMatrix2D(
                      vector.getData(0),vector.getData(1));
    }//end constructor
    //-----------------------------------------------------//

    public String toString(){
      return vector.getData(0) + "," + vector.getData(1);
    }//end toString
    //-----------------------------------------------------//

    public double getData(int index){
      if((index < 0) || (index > 1)){
        throw new IndexOutOfBoundsException();
      }else{
        return vector.getData(index);
      }//end else
    }//end getData
    //-----------------------------------------------------//

    public void setData(int index,double data){
      if((index < 0) || (index > 1)){
        throw new IndexOutOfBoundsException();
      }else{
        vector.setData(index,data);
      }//end else
    }//end setData
    //-----------------------------------------------------//

    //This method draws a vector on the specified graphics
    // context, with the tail of the vector located at a
    // specified point, and with a small filled circle at
    // the head.
    public void draw(Graphics2D g2D,GM02.Point2D tail){

      drawLine(g2D,
               tail.getData(0),
               tail.getData(1),
```

```
            tail.getData(0)+vector.getData(0),
            tail.getData(1)+vector.getData(1));

      fillOval(g2D,
            tail.getData(0)+vector.getData(0)-3,
            tail.getData(1)+vector.getData(1)+3,
            6,
            6);
    }//end draw
    //-----------------------------------------------//

    //Returns a reference to the ColMatrix2D object that
    // defines this Vector2D object.
    public GM02.ColMatrix2D getColMatrix(){
      return vector;
    }//end getColMatrix
    //-----------------------------------------------//

    //This method overrides the equals method inherited
    // from the class named Object. It compares the values
    // stored in the ColMatrix2D objects that define two
    // Vector2D objects and returns true if they are equal
    // and false otherwise.
    public boolean equals(Object obj){
      if(vector.equals((
                  (GM02.Vector2D)obj).getColMatrix())){
        return true;
      }else{
        return false;
      }//end else

    }//end overridden equals method
    //-----------------------------------------------//

    //Adds this vector to a vector received as an incoming
    // parameter and returns the sum as a vector.
    public GM02.Vector2D add(GM02.Vector2D vec){
      return new GM02.Vector2D(new ColMatrix2D(
                     vec.getData(0)+vector.getData(0),
                     vec.getData(1)+vector.getData(1)));
    }//end add
    //-----------------------------------------------//

    //Returns the length of a Vector2D object.
    public double getLength(){
      return Math.sqrt(
            getData(0)*getData(0) + getData(1)*getData(1));
    }//end getLength
    //-----------------------------------------------//

    //Multiplies this vector by a scale factor received as
```

```
    // an incoming parameter and returns the scaled
    // vector.
    public GM02.Vector2D scale(Double factor){
      return new GM02.Vector2D(new ColMatrix2D(
                                  getData(0) * factor,
                                  getData(1) * factor));
    }//end scale
    //-----------------------------------------------------//

    //Changes the sign on each of the vector components
    // and returns the negated vector.
    public GM02.Vector2D negate(){
      return new GM02.Vector2D(new ColMatrix2D(
                                         -getData(0),
                                         -getData(1)));
    }//end negate
    //-----------------------------------------------------//

    //Returns a new vector that points in the same
    // direction but has a length of one unit.
    public GM02.Vector2D normalize(){
      double length = getLength();
      return new GM02.Vector2D(new ColMatrix2D(
                                    getData(0)/length,
                                    getData(1)/length));
    }//end normalize
    //-----------------------------------------------------//

    //Computes the dot product of two Vector2D
    // objects and returns the result as type double.
    public double dot(GM02.Vector2D vec){
      GM02.ColMatrix2D matrixA = getColMatrix();
      GM02.ColMatrix2D matrixB = vec.getColMatrix();
      return matrixA.dot(matrixB);
    }//end dot
    //-----------------------------------------------------//

    //Computes and returns the angle between two Vector2D
    // objects. The angle is returned in degrees as type
    // double.
    public double angle(GM02.Vector2D vec){
      GM02.Vector2D normA = normalize();
      GM02.Vector2D normB = vec.normalize();
      double normDotProd = normA.dot(normB);
      return Math.toDegrees(Math.acos(normDotProd));
    }//end angle
    //-----------------------------------------------------//
  }//end class Vector2D
  //=====================================================//
```

```
  public static class Vector3D{
    GM02.ColMatrix3D vector;

    public Vector3D(GM02.ColMatrix3D vector){//constructor
      //Create and save a clone of the ColMatrix3D object
      // used to define the vector to prevent the vector
      // from being corrupted by a later change in the
      // values stored in the original ColMatris3D object.
      this.vector = new ColMatrix3D(vector.getData(0),
                                    vector.getData(1),
                                    vector.getData(2));
    }//end constructor
    //-----------------------------------------------------//

    public String toString(){
      return vector.getData(0) + "," + vector.getData(1)
                               + "," + vector.getData(2);
    }//end toString
    //-----------------------------------------------------//

    public double getData(int index){
      if((index < 0) || (index > 2)){
        throw new IndexOutOfBoundsException();
      }else{
        return vector.getData(index);
      }//end else
    }//end getData
    //-----------------------------------------------------//

    public void setData(int index,double data){
      if((index < 0) || (index > 2)){
        throw new IndexOutOfBoundsException();
      }else{
        vector.setData(index,data);
      }//end else
    }//end setData
    //-----------------------------------------------------//

    //This method draws a vector on the specified graphics
    // context, with the tail of the vector located at a
    // specified point, and with a small circle at the
    // head.
    public void draw(Graphics2D g2D,GM02.Point3D tail){

      //Get a 2D projection of the tail
      GM02.ColMatrix2D tail2D = convert3Dto2D(tail.point);

      //Get the 3D location of the head
      GM02.ColMatrix3D head =
                      tail.point.add(this.getColMatrix());
```

```
    //Get a 2D projection of the head
    GM02.ColMatrix2D head2D = convert3Dto2D(head);
    drawLine(g2D,tail2D.getData(0),
                 tail2D.getData(1),
                 head2D.getData(0),
                 head2D.getData(1));

    //Draw a small filled circle to identify the head.
    fillOval(g2D,head2D.getData(0)-3,
                 head2D.getData(1)+3,
                 6,
                 6);

  }//end draw
  //-----------------------------------------------------//

  //Returns a reference to the ColMatrix3D object that
  // defines this Vector3D object.
  public GM02.ColMatrix3D getColMatrix(){
    return vector;
  }//end getColMatrix
  //-----------------------------------------------------//

  //This method overrides the equals method inherited
  // from the class named Object. It compares the values
  // stored in the ColMatrix3D objects that define two
  // Vector3D objects and returns true if they are equal
  // and false otherwise.
  public boolean equals(Object obj){
    if(vector.equals((
                   (GM02.Vector3D)obj).getColMatrix())){
      return true;
    }else{
      return false;
    }//end else

  }//end overridden equals method
  //-----------------------------------------------------//

  //Adds this vector to a vector received as an incoming
  // parameter and returns the sum as a vector.
  public GM02.Vector3D add(GM02.Vector3D vec){
    return new GM02.Vector3D(new ColMatrix3D(
                      vec.getData(0)+vector.getData(0),
                      vec.getData(1)+vector.getData(1),
                      vec.getData(2)+vector.getData(2)));
  }//end add
  //-----------------------------------------------------//

  //Returns the length of a Vector3D object.
  public double getLength(){
```

```
    return Math.sqrt(getData(0)*getData(0) +
                     getData(1)*getData(1) +
                     getData(2)*getData(2));
  }//end getLength
  //-----------------------------------------------------//

  //Multiplies this vector by a scale factor received as
  // an incoming parameter and returns the scaled
  // vector.
  public GM02.Vector3D scale(Double factor){
    return new GM02.Vector3D(new ColMatrix3D(
                                 getData(0) * factor,
                                 getData(1) * factor,
                                 getData(2) * factor));
  }//end scale
  //-----------------------------------------------------//

  //Changes the sign on each of the vector components
  // and returns the negated vector.
  public GM02.Vector3D negate(){
    return new GM02.Vector3D(new ColMatrix3D(
                                       -getData(0),
                                       -getData(1),
                                       -getData(2)));
  }//end negate
  //-----------------------------------------------------//

  //Returns a new vector that points in the same
  // direction but has a length of one unit.
  public GM02.Vector3D normalize(){
    double length = getLength();
    return new GM02.Vector3D(new ColMatrix3D(
                                   getData(0)/length,
                                   getData(1)/length,
                                   getData(2)/length));
  }//end normalize
  //-----------------------------------------------------//

  //Computes the dot product of two Vector3D
  // objects and returns the result as type double.
  public double dot(GM02.Vector3D vec){
    GM02.ColMatrix3D matrixA = getColMatrix();
    GM02.ColMatrix3D matrixB = vec.getColMatrix();
    return matrixA.dot(matrixB);
  }//end dot
  //-----------------------------------------------------//

  //Computes and returns the angle between two Vector3D
  // objects. The angle is returned in degrees as type
  // double.
  public double angle(GM02.Vector3D vec){
```

```
      GM02.Vector3D normA = normalize();
      GM02.Vector3D normB = vec.normalize();
      double normDotProd = normA.dot(normB);
      return Math.toDegrees(Math.acos(normDotProd));
    }//end angle
    //-----------------------------------------------------//
  }//end class Vector3D
  //=====================================================//
  //=====================================================//


  //A line is defined by two points. One is called the
  // tail and the other is called the head. Note that this
  // class has the same name as one of the classes in
  // the Graphics2D class. Therefore, if the class from
  // the Graphics2D class is used in some future upgrade
  // to this program, it will have to be fully qualified.
  public static class Line2D{
    GM02.Point2D[] line = new GM02.Point2D[2];

    public Line2D(GM02.Point2D tail,GM02.Point2D head){
      //Create and save clones of the points used to
      // define the line to prevent the line from being
      // corrupted by a later change in the coordinate
      // values of the points.
      this.line[0] = new Point2D(new GM02.ColMatrix2D(
                         tail.getData(0),tail.getData(1)));
      this.line[1] = new Point2D(new GM02.ColMatrix2D(
                         head.getData(0),head.getData(1)));
    }//end constructor
    //-----------------------------------------------------//

    public String toString(){
      return "Tail = " + line[0].getData(0) + ","
             + line[0].getData(1) + "\nHead = "
             + line[1].getData(0) + ","
             + line[1].getData(1);
    }//end toString
    //-----------------------------------------------------//

    public GM02.Point2D getTail(){
      return line[0];
    }//end getTail
    //-----------------------------------------------------//

    public GM02.Point2D getHead(){
      return line[1];
    }//end getHead
    //-----------------------------------------------------//

    public void setTail(GM02.Point2D newPoint){
```

```
      //Create and save a clone of the new point to
      // prevent the line from being corrupted by a
      // later change in the coordinate values of the
      // point.
      this.line[0] = new Point2D(new GM02.ColMatrix2D(
                newPoint.getData(0),newPoint.getData(1)));
    }//end setTail
    //-----------------------------------------------//

    public void setHead(GM02.Point2D newPoint){
      //Create and save a clone of the new point to
      // prevent the line from being corrupted by a
      // later change in the coordinate values of the
      // point.
      this.line[1] = new Point2D(new GM02.ColMatrix2D(
                newPoint.getData(0),newPoint.getData(1)));
    }//end setHead
    //-----------------------------------------------//

    public void draw(Graphics2D g2D){
      drawLine(g2D,getTail().getData(0),
                   getTail().getData(1),
                   getHead().getData(0),
                   getHead().getData(1));
    }//end draw
    //-----------------------------------------------//
  }//end class Line2D
  //===============================================//


  //A line is defined by two points. One is called the
  // tail and the other is called the head.
  public static class Line3D{
    GM02.Point3D[] line = new GM02.Point3D[2];

    public Line3D(GM02.Point3D tail,GM02.Point3D head){
      //Create and save clones of the points used to
      // define the line to prevent the line from being
      // corrupted by a later change in the coordinate
      // values of the points.
      this.line[0] = new Point3D(new GM02.ColMatrix3D(
                                       tail.getData(0),
                                       tail.getData(1),
                                       tail.getData(2)));
      this.line[1] = new Point3D(new GM02.ColMatrix3D(
                                       head.getData(0),
                                       head.getData(1),
                                       head.getData(2)));
    }//end constructor
    //-----------------------------------------------//
```

```
  public String toString(){
    return "Tail = " + line[0].getData(0) + ","
                     + line[0].getData(1)  + ","
                     + line[0].getData(2)
                     + "\nHead = "
                     + line[1].getData(0) + ","
                     + line[1].getData(1) + ","
                     + line[1].getData(2);
  }//end toString
  //-----------------------------------------------//

  public GM02.Point3D getTail(){
    return line[0];
  }//end getTail
  //-----------------------------------------------//

  public GM02.Point3D getHead(){
    return line[1];
  }//end getHead
  //-----------------------------------------------//

  public void setTail(GM02.Point3D newPoint){
    //Create and save a clone of the new point to
    // prevent the line from being corrupted by a
    // later change in the coordinate values of the
    // point.
    this.line[0] = new Point3D(new GM02.ColMatrix3D(
                                 newPoint.getData(0),
                                 newPoint.getData(1),
                                 newPoint.getData(2)));
  }//end setTail
  //-----------------------------------------------//

  public void setHead(GM02.Point3D newPoint){
    //Create and save a clone of the new point to
    // prevent the line from being corrupted by a
    // later change in the coordinate values of the
    // point.
    this.line[1] = new Point3D(new GM02.ColMatrix3D(
                                 newPoint.getData(0),
                                 newPoint.getData(1),
                                 newPoint.getData(2)));
  }//end setHead
  //-----------------------------------------------//

  public void draw(Graphics2D g2D){

    //Get 2D projection coordinates.
    GM02.ColMatrix2D tail =
                        convert3Dto2D(getTail().point);
    GM02.ColMatrix2D head =
```

```
                          convert3Dto2D(getHead().point);

      drawLine(g2D,tail.getData(0),
                   tail.getData(1),
                   head.getData(0),
                   head.getData(1));
    }//end draw
    //-----------------------------------------------------//
  }//end class Line3D
  //=====================================================//

}//end class GM02
```

.

Listing 9: Source code for the program named DotProd2D01.

```
    /*DotProd2D01.java
Copyright 2008, R.G.Baldwin
Revised 03/04/08

Study Kjell through Chapter 8 - Length, Orthogonality, and
the Column Matrix Dot product.

The purpose of this program is to confirm proper operation
of the ColMatrix2D.dot method.

The program creates a GUI that allows the user to enter
the first and second values for a pair of ColMatrix2D
objects along with a button labeled OK.

When the user clicks the OK button, the dot product
between the two ColMatrix2D objects is computed. The
resulting value is converted to four decimal digits and
displayed in a text field on the GUI.

Tested using JDK 1.6 under WinXP.
*******************************************************/
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class DotProd2D01{
  public static void main(String[] args){
    GUI guiObj = new GUI();
  }//end main
}//end controlling class DotProd2D01
//=====================================================//

class GUI extends JFrame implements ActionListener{
  //Specify the horizontal and vertical size of a JFrame
```

```
  // object.
  int hSize = 400;
  int vSize = 150;

  JTextField colMatA0 = new JTextField("0.707");
  JTextField colMatA1 = new JTextField("0.707");
  JTextField colMatB0 = new JTextField("-0.707");
  JTextField colMatB1 = new JTextField("-0.707");
  JTextField dotProduct = new JTextField();
  JButton button = new JButton("OK");

  //-----------------------------------------------------//

  GUI(){//constructor

    //Set JFrame size, title, and close operation.
    setSize(hSize,vSize);
    setTitle("Copyright 2008,R.G.Baldwin");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);


    //Instantiate a JPanel that will house the user input
    // components and set its layout manager.
    JPanel controlPanel = new JPanel();
    controlPanel.setLayout(new GridLayout(0,4));

    //Add the user input components and appropriate labels
    // to the control panel.
    controlPanel.add(new JLabel(" colMatA0 = "));
    controlPanel.add(colMatA0);

    controlPanel.add(new JLabel(" colMatA1 = "));
    controlPanel.add(colMatA1);

    controlPanel.add(new JLabel(" colMatB0 = "));
    controlPanel.add(colMatB0);

    controlPanel.add(new JLabel(" colMatB1 = "));
    controlPanel.add(colMatB1);

    controlPanel.add(new JLabel(" Dot Prod = "));
    controlPanel.add(dotProduct);

    controlPanel.add(new JLabel(""));//spacer
    controlPanel.add(button);

    //Add the control panel to the CENTER position in the
    // JFrame.
    this.getContentPane().add(
                        BorderLayout.CENTER,controlPanel);
    setVisible(true);
```

```
    //Register this object as an action listener on the
    // button.
    button.addActionListener(this);

  }//end constructor
  //-----------------------------------------------------//

  //This method is called to respond to a click on the
  // button.
  public void actionPerformed(ActionEvent e){
    //Create two ColMatrix2D objects.
    GM02.ColMatrix2D matrixA = new GM02.ColMatrix2D(
      Double.parseDouble(colMatA0.getText()),
      Double.parseDouble(colMatA1.getText()));

    GM02.ColMatrix2D matrixB = new GM02.ColMatrix2D(
      Double.parseDouble(colMatB0.getText()),
      Double.parseDouble(colMatB1.getText()));

    //Compute the dot product.
    double dotProd = matrixA.dot(matrixB);

    //Eliminate exponential notation in the display.
    if(Math.abs(dotProd) < 0.001){
      dotProd = 0.0;
    }//end if

    //Convert to four decimal digits and display.
    dotProd =((int)(10000*dotProd))/10000.0;
    dotProduct.setText("" +  dotProd);

  }//end actionPerformed
  //=====================================================//

}//end class GUI
```

.

Listing 10: Source code for the program named DotProd2D02.

```
    /*DotProd2D02.java
Copyright 2008, R.G.Baldwin
Revised 03/07/08

This program allows the user to experiment with the dot
product and the angle between a pair of GM02.Vector2D
objects.

Study Kjell through Chapter 9, The Angle Between Two
Vectors.
```

```
A GUI is provided that allows the user to enter four
double values that define each of two GM02.Vector2D
objects.  The GUI also provides an OK button as well as
two text fields used for display of computed results.

In addition, the GUI provides a 2D drawing area.

When the user clicks the OK button, the program draws the
two vectors, one in black and the other in magenta, on
the output screen with the tail of each vector located at
the origin in 2D space.

The program also displays the values of the dot product
of the two vectors and the angle between the two vectors
in degrees.

Tested using JDK 1.6 under WinXP.
*******************************************************/
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class DotProd2D02{
  public static void main(String[] args){
    GUI guiObj = new GUI();
  }//end main
}//end controlling class DotProd2D02
//=====================================================//

class GUI extends JFrame implements ActionListener{
  //Specify the horizontal and vertical size of a JFrame
  // object.
  int hSize = 400;
  int vSize = 400;
  Image osi;//an off-screen image
  int osiWidth;//off-screen image width
  int osiHeight;//off-screen image height
  MyCanvas myCanvas;//a subclass of Canvas
  Graphics2D g2D;//off-screen graphics context.

  //User input components.
  JTextField vectorAx = new JTextField("50.0");
  JTextField vectorAy = new JTextField("100.0");
  JTextField vectorBx = new JTextField("-100.0");
  JTextField vectorBy = new JTextField("-50.0");
  JTextField dotProduct = new JTextField();
  JTextField angleDisplay = new JTextField();
  JButton button = new JButton("OK");

  //-----------------------------------------------------//
```

```
  GUI(){//constructor

    //Set JFrame size, title, and close operation.
    setSize(hSize,vSize);
    setTitle("Copyright 2008,R.G.Baldwin");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //Instantiate a JPanel that will house the user input
    // components and set its layout manager.
    JPanel controlPanel = new JPanel();
    controlPanel.setLayout(new GridLayout(0,4));

    //Add the user input components and appropriate labels
    // to the control panel.
    controlPanel.add(new JLabel(" VectorAx = "));
    controlPanel.add(vectorAx);

    controlPanel.add(new JLabel(" VectorAy = "));
    controlPanel.add(vectorAy);

    controlPanel.add(new JLabel(" VectorBx = "));
    controlPanel.add(vectorBx);

    controlPanel.add(new JLabel(" VectorBy = "));
    controlPanel.add(vectorBy);

    controlPanel.add(new JLabel(" Dot Prod = "));
    controlPanel.add(dotProduct);

    controlPanel.add(new JLabel(" Angle (deg) = "));
    controlPanel.add(angleDisplay);

    controlPanel.add(button);

    //Add the control panel to the SOUTH position in the
    // JFrame.
    this.getContentPane().add(
                        BorderLayout.SOUTH,controlPanel);


    //Create a new drawing canvas and add it to the
    // CENTER of the JFrame above the control panel.
    myCanvas = new MyCanvas();
    this.getContentPane().add(
                            BorderLayout.CENTER,myCanvas);

    //This object must be visible before you can get an
    // off-screen image.  It must also be visible before
    // you can compute the size of the canvas.
    setVisible(true);
```

```
    //Make the size of the off-screen image match the
    // size of the canvas.
    osiWidth = myCanvas.getWidth();
    osiHeight = myCanvas.getHeight();

    //Create an off-screen image and get a graphics
    // context on it.
    osi = createImage(osiWidth,osiHeight);
    g2D = (Graphics2D)(osi.getGraphics());


    //Translate the origin to the center.
    GM02.translate(g2D,0.5*osiWidth,-0.5*osiHeight);


    //Register this object as an action listener on the
    // button.
    button.addActionListener(this);

    //Cause the overridden paint method belonging to
    // myCanvas to be executed.
    myCanvas.repaint();

  }//end constructor
  //-----------------------------------------------------//

  //This method is used to draw orthogonal 2D axes on the
  // off-screen image that intersect at the origin.
  private void setCoordinateFrame(Graphics2D g2D){

    //Erase the screen
    g2D.setColor(Color.WHITE);
    GM02.fillRect(g2D,-osiWidth/2,osiHeight/2,
                                     osiWidth,osiHeight);

    //Draw x-axis in RED
    g2D.setColor(Color.RED);
    GM02.Point2D pointA = new GM02.Point2D(
                     new GM02.ColMatrix2D(-osiWidth/2,0));
    GM02.Point2D pointB = new GM02.Point2D(
                      new GM02.ColMatrix2D(osiWidth/2,0));
    new GM02.Line2D(pointA,pointB).draw(g2D);

    //Draw y-axis in GREEN
    g2D.setColor(Color.GREEN);
    pointA = new GM02.Point2D(
                    new GM02.ColMatrix2D(0,-osiHeight/2));
    pointB = new GM02.Point2D(
                     new GM02.ColMatrix2D(0,osiHeight/2));
    new GM02.Line2D(pointA,pointB).draw(g2D);
```

```
  }//end setCoordinateFrame method
  //-----------------------------------------------------//

  //This method is called to respond to a click on the
  // button.
  public void actionPerformed(ActionEvent e){

    //Erase the off-screen image and draw the axes.
    setCoordinateFrame(g2D);

    //Create two ColMatrix2D objects based on the user
    // input values.
    GM02.ColMatrix2D matrixA = new GM02.ColMatrix2D(
                  Double.parseDouble(vectorAx.getText()),
                  Double.parseDouble(vectorAy.getText()));

    GM02.ColMatrix2D matrixB = new GM02.ColMatrix2D(
                  Double.parseDouble(vectorBx.getText()),
                  Double.parseDouble(vectorBy.getText()));


    //Use the ColMatrix2D objects to create two Vector2D
    // objects.
    GM02.Vector2D vecA = new GM02.Vector2D(matrixA);
    GM02.Vector2D vecB = new GM02.Vector2D(matrixB);

    //Draw the two vectors with their tails at the origin.
    g2D.setColor(Color.BLACK);
    vecA.draw(
         g2D,new GM02.Point2D(new GM02.ColMatrix2D(0,0)));

    g2D.setColor(Color.MAGENTA);
    vecB.draw(
         g2D,new GM02.Point2D(new GM02.ColMatrix2D(0,0)));

    //Compute the dot product of the two vectors.
    double dotProd = vecA.dot(vecB);

    //Eliminate exponential notation in the display.
    if(Math.abs(dotProd) < 0.001){
      dotProd = 0.0;
    }//end if

    //Convert to four decimal digits and display.
    dotProd =((int)(10000*dotProd))/10000.0;
    dotProduct.setText("" +  dotProd);


    //Compute the angle between the two vectors.
    double angle = vecA.angle(vecB);
```

```
    //Eliminate exponential notation in the display.
    if(Math.abs(angle) < 0.001){
      angle = 0.0;
    }//end if

    //Convert to four decimal digits and display.
    angle =((int)(10000*angle))/10000.0;
    angleDisplay.setText("" +  angle);

    myCanvas.repaint();//Copy off-screen image to canvas.
  }//end actionPerformed
  //=====================================================//


  //This is an inner class of the GUI class.
  class MyCanvas extends Canvas{
    //Override the paint() method. This method will be
    // called when the JFrame and the Canvas appear on the
    // screen or when the repaint method is called on the
    // Canvas object.
    //The purpose of this method is to display the
    // off-screen image on the screen.
    public void paint(Graphics g){
      g.drawImage(osi,0,0,this);
    }//end overridden paint()

  }//end inner class MyCanvas

}//end class GUI
```

.

Listing 11: Source code for the program named DotProd3D01.

```
    /*DotProd3D01.java
Copyright 2008, R.G.Baldwin
Revised 03/04/08

Study Kjell through Chapter 8 - Length, Orthogonality, and
the Column Matrix Dot product.

The purpose of this program is to confirm proper operation
of the ColMatrix3D.dot method.

The program creates a GUI that allows the user to enter
three values for each of a pair of ColMatrix3D objects
along with a button labeled OK.

When the user clicks the OK button, the dot product
between the two ColMatrix3D objects is computed. The
```

```
resulting value is converted to four decimal digits and
displayed in a text field on the GUI.

Tested using JDK 1.6 under WinXP.
*******************************************************/
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class DotProd3D01{
  public static void main(String[] args){
    GUI guiObj = new GUI();
  }//end main
}//end controlling class DotProd3D01
//=====================================================//

class GUI extends JFrame implements ActionListener{
  //Specify the horizontal and vertical size of a JFrame
  // object.
  int hSize = 470;
  int vSize = 150;

  JTextField colMatA0 = new JTextField("0.57735");
  JTextField colMatA1 = new JTextField("0.57735");
  JTextField colMatA2 = new JTextField("0.57735");

  JTextField colMatB0 = new JTextField("-0.57735");
  JTextField colMatB1 = new JTextField("-0.57735");
  JTextField colMatB2 = new JTextField("-0.57735");

  JTextField dotProduct = new JTextField();
  JButton button = new JButton("OK");

  //-----------------------------------------------------//

  GUI(){//constructor

    //Set JFrame size, title, and close operation.
    setSize(hSize,vSize);
    setTitle("Copyright 2008,R.G.Baldwin");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);


    //Instantiate a JPanel that will house the user input
    // components and set its layout manager.
    JPanel controlPanel = new JPanel();
    controlPanel.setLayout(new GridLayout(0,6));

    //Add the user input components and appropriate labels
    // to the control panel.
    controlPanel.add(new JLabel(" colMatA0 = "));
```

```
    controlPanel.add(colMatA0);

    controlPanel.add(new JLabel(" colMatA1 = "));
    controlPanel.add(colMatA1);

    controlPanel.add(new JLabel(" colMatA2 = "));
    controlPanel.add(colMatA2);

    controlPanel.add(new JLabel(" colMatB0 = "));
    controlPanel.add(colMatB0);

    controlPanel.add(new JLabel(" colMatB1 = "));
    controlPanel.add(colMatB1);

    controlPanel.add(new JLabel(" colMatB2 = "));
    controlPanel.add(colMatB2);

    controlPanel.add(new JLabel(" Dot Prod = "));
    controlPanel.add(dotProduct);

    controlPanel.add(new JLabel(""));//spacer
    controlPanel.add(button);

    //Add the control panel to the CENTER position in the
    // JFrame.
    this.getContentPane().add(
                        BorderLayout.CENTER,controlPanel);
    setVisible(true);

    //Register this object as an action listener on the
    // button.
    button.addActionListener(this);

  }//end constructor
  //-----------------------------------------------------//

  //This method is called to respond to a click on the
  // button.
  public void actionPerformed(ActionEvent e){
    //Create two ColMatrix3D objects.
    GM02.ColMatrix3D matrixA = new GM02.ColMatrix3D(
      Double.parseDouble(colMatA0.getText()),
      Double.parseDouble(colMatA1.getText()),
      Double.parseDouble(colMatA2.getText()));

    GM02.ColMatrix3D matrixB = new GM02.ColMatrix3D(
      Double.parseDouble(colMatB0.getText()),
      Double.parseDouble(colMatB1.getText()),
      Double.parseDouble(colMatB2.getText()));

    //Compute the dot product.
```

```
    double dotProd = matrixA.dot(matrixB);

    //Eliminate exponential notation in the display.
    if(Math.abs(dotProd) < 0.001){
      dotProd = 0.0;
    }//end if

    //Convert to four decimal digits and display.
    dotProd =((int)(10000*dotProd))/10000.0;
    dotProduct.setText("" +  dotProd);

  }//end actionPerformed
  //=====================================================//

}//end class GUI
```

.

Listing 12: Source code for the program named DotProd3D02.

```
    /*DotProd3D02.java
Copyright 2008, R.G.Baldwin
Revised 03/07/08

This program allows the user to experiment with the dot
product and the angle between a pair of GM02.Vector3D
objects.

Study Kjell through Chapter 10, Angle between 3D Vectors.

A GUI is provided that allows the user to enter six
double values that define each of two GM02.Vector3D
objects.  The GUI also provides an OK button as well as
two text fields used for display of computed results.

In addition, the GUI provides a 3D drawing area.

When the user clicks the OK button, the program draws the
two vectors, one black and the other in magenta, on the
output screen with the tail of each vector located at the
origin in 3D space.

The program also displays the values of the dot product
of the two vectors and the angle between the two vectors
in degrees.

Tested using JDK 1.6 under WinXP.
*******************************************************/
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
```

```
class DotProd3D02{
  public static void main(String[] args){
    GUI guiObj = new GUI();
  }//end main
}//end controlling class DotProd3D02
//=====================================================//

class GUI extends JFrame implements ActionListener{
  //Specify the horizontal and vertical size of a JFrame
  // object.
  int hSize = 400;
  int vSize = 400;
  Image osi;//an off-screen image
  int osiWidth;//off-screen image width
  int osiHeight;//off-screen image height
  MyCanvas myCanvas;//a subclass of Canvas
  Graphics2D g2D;//off-screen graphics context.

  //User input components.
  JTextField vecAx = new JTextField("50.0");
  JTextField vecAy = new JTextField("100.0");
  JTextField vecAz = new JTextField("0.0");
  JTextField vecBx = new JTextField("-100.0");
  JTextField vecBy = new JTextField("-50.0");
  JTextField vecBz = new JTextField("0.0");
  JTextField dotProduct = new JTextField();
  JTextField angleDisplay = new JTextField();
  JButton button = new JButton("OK");

  //-----------------------------------------------------//

  GUI(){//constructor

    //Set JFrame size, title, and close operation.
    setSize(hSize,vSize);
    setTitle("Copyright 2008,R.G.Baldwin");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //Instantiate a JPanel that will house the user input
    // components and set its layout manager.
    JPanel controlPanel = new JPanel();
    controlPanel.setLayout(new GridLayout(0,6));

    //Add the user input components and appropriate labels
    // to the control panel.
    controlPanel.add(new JLabel(" VecAx = "));
    controlPanel.add(vecAx);

    controlPanel.add(new JLabel(" VecAy = "));
    controlPanel.add(vecAy);
```

```
    controlPanel.add(new JLabel(" VecAz = "));
    controlPanel.add(vecAz);

    controlPanel.add(new JLabel(" VecBx = "));
    controlPanel.add(vecBx);

    controlPanel.add(new JLabel(" VecrBy = "));
    controlPanel.add(vecBy);

    controlPanel.add(new JLabel(" VecBz = "));
    controlPanel.add(vecBz);

    controlPanel.add(new JLabel(" Dot Prod = "));
    controlPanel.add(dotProduct);

    controlPanel.add(new JLabel(" Ang(deg)"));
    controlPanel.add(angleDisplay);

    controlPanel.add(new JLabel(""));//spacer

    controlPanel.add(button);

    //Add the control panel to the SOUTH position in the
    // JFrame.
    this.getContentPane().add(
                        BorderLayout.SOUTH,controlPanel);


    //Create a new drawing canvas and add it to the
    // CENTER of the JFrame above the control panel.
    myCanvas = new MyCanvas();
    this.getContentPane().add(
                            BorderLayout.CENTER,myCanvas);

    //This object must be visible before you can get an
    // off-screen image.  It must also be visible before
    // you can compute the size of the canvas.
    setVisible(true);

    //Make the size of the off-screen image match the
    // size of the canvas.
    osiWidth = myCanvas.getWidth();
    osiHeight = myCanvas.getHeight();

    //Create an off-screen image and get a graphics
    // context on it.
    osi = createImage(osiWidth,osiHeight);
    g2D = (Graphics2D)(osi.getGraphics());
```

```
    //Translate the origin to the center.
    GM02.translate(g2D,0.5*osiWidth,-0.5*osiHeight);


    //Register this object as an action listener on the
    // button.
    button.addActionListener(this);

    //Cause the overridden paint method belonging to
    // myCanvas to be executed.
    myCanvas.repaint();

  }//end constructor
  //-----------------------------------------------------//

  //This method is used to draw orthogonal 3D axes on the
  // off-screen image that intersect at the origin.
  private void setCoordinateFrame(Graphics2D g2D){

    //Erase the screen
    g2D.setColor(Color.WHITE);
    GM02.fillRect(g2D,-osiWidth/2,osiHeight/2,
                                    osiWidth,osiHeight);

    //Draw x-axis in RED
    g2D.setColor(Color.RED);
    GM02.Point3D pointA = new GM02.Point3D(
                  new GM02.ColMatrix3D(-osiWidth/2,0,0));
    GM02.Point3D pointB = new GM02.Point3D(
                   new GM02.ColMatrix3D(osiWidth/2,0,0));
    new GM02.Line3D(pointA,pointB).draw(g2D);

    //Draw y-axis in GREEN
    g2D.setColor(Color.GREEN);
    pointA = new GM02.Point3D(
                 new GM02.ColMatrix3D(0,-osiHeight/2,0));
    pointB = new GM02.Point3D(
                  new GM02.ColMatrix3D(0,osiHeight/2,0));
    new GM02.Line3D(pointA,pointB).draw(g2D);

    //Draw z-axis in BLUE. Make its length the same as the
    // length of the x-axis.
    g2D.setColor(Color.BLUE);
    pointA = new GM02.Point3D(
                  new GM02.ColMatrix3D(0,0,-osiWidth/2));
    pointB = new GM02.Point3D(
                   new GM02.ColMatrix3D(0,0,osiWidth/2));
    new GM02.Line3D(pointA,pointB).draw(g2D);

  }//end setCoordinateFrame method
  //-----------------------------------------------------//
```

```
  //This method is called to respond to a click on the
  // button.
  public void actionPerformed(ActionEvent e){

    //Erase the off-screen image and draw the axes.
    setCoordinateFrame(g2D);

    //Create two ColMatrix3D objects based on the user
    // input values.
    GM02.ColMatrix3D matrixA = new GM02.ColMatrix3D(
                  Double.parseDouble(vecAx.getText()),
                  Double.parseDouble(vecAy.getText()),
                  Double.parseDouble(vecAz.getText()));

    GM02.ColMatrix3D matrixB = new GM02.ColMatrix3D(
                  Double.parseDouble(vecBx.getText()),
                  Double.parseDouble(vecBy.getText()),
                  Double.parseDouble(vecBz.getText()));


    //Use the ColMatrix3D objects to create two Vector3D
    // objects.
    GM02.Vector3D vecA = new GM02.Vector3D(matrixA);
    GM02.Vector3D vecB = new GM02.Vector3D(matrixB);

    //Draw the two vectors with their tails at the origin.
    g2D.setColor(Color.BLACK);
    vecA.draw(g2D,new GM02.Point3D(
                          new GM02.ColMatrix3D(0,0,0)));
    g2D.setColor(Color.MAGENTA);
    vecB.draw(g2D,new GM02.Point3D(
                          new GM02.ColMatrix3D(0,0,0)));

    //Compute the dot product of the two vectors.
    double dotProd = vecA.dot(vecB);

    //Eliminate exponential notation in the display.
    if(Math.abs(dotProd) < 0.001){
      dotProd = 0.0;
    }//end if

    //Convert to four decimal digits and display.
    dotProd =((int)(10000*dotProd))/10000.0;
    dotProduct.setText("" +  dotProd);


    //Compute the angle between the two vectors.
    double angle = vecA.angle(vecB);

    //Eliminate exponential notation in the display.
```

```
    if(Math.abs(angle) < 0.001){
      angle = 0.0;
    }//end if

    //Convert to four decimal digits and display.
    angle =((int)(10000*angle))/10000.0;
    angleDisplay.setText("" +  angle);

    myCanvas.repaint();//Copy off-screen image to canvas.
  }//end actionPerformed
  //=====================================================//


  //This is an inner class of the GUI class.
  class MyCanvas extends Canvas{
    //Override the paint() method. This method will be
    // called when the JFrame and the Canvas appear on the
    // screen or when the repaint method is called on the
    // Canvas object.
    //The purpose of this method is to display the
    // off-screen image on the screen.
    public void paint(Graphics g){
      g.drawImage(osi,0,0,this);
    }//end overridden paint()

  }//end inner class MyCanvas

}//end class GUI
```

12 Exercises

12.1 Exercise 1

Using Java and the game-math library named **GM02** , or using a different programming environment of your choice, write a program that behaves as follows.

When the program starts running, an image similar to Figure 17 (p. 70) appears on the screen. Each of the text fields is blank and the drawing area at the top is also blank.

Each time you click the **Replot** button, the program generates three random values in the general range of -128 to 127. The first two values are used as the x and y values for a vector. The two values are displayed in the fields labeled **VectorAx** and **VectorAy** . Also, the two values are used to create and draw a black vector with its tail at the origin as shown in Figure 17 (p. 70) .

The third random value is used as the x-value for a second vector. It is displayed in the field labeled **VectorBx** . A y-value is computed that will cause that vector to be perpendicular to the black vector. That value is displayed in the field labeled **VectorBy** and the two values are used to draw the magenta vector shown in Figure 17 (p. 70) .

The dot product between the two vectors is computed and displayed in the field labeled **Dot Prod** . The angle between the two vectors is computed and displayed in the field labeled **Angle (deg)** .

If the two vectors are perpendicular, the dot product should be close to zero and the angle should be very close to 90 degrees.

Cause your name to appear in the screen output in some manner.

Output from Exercise 1.

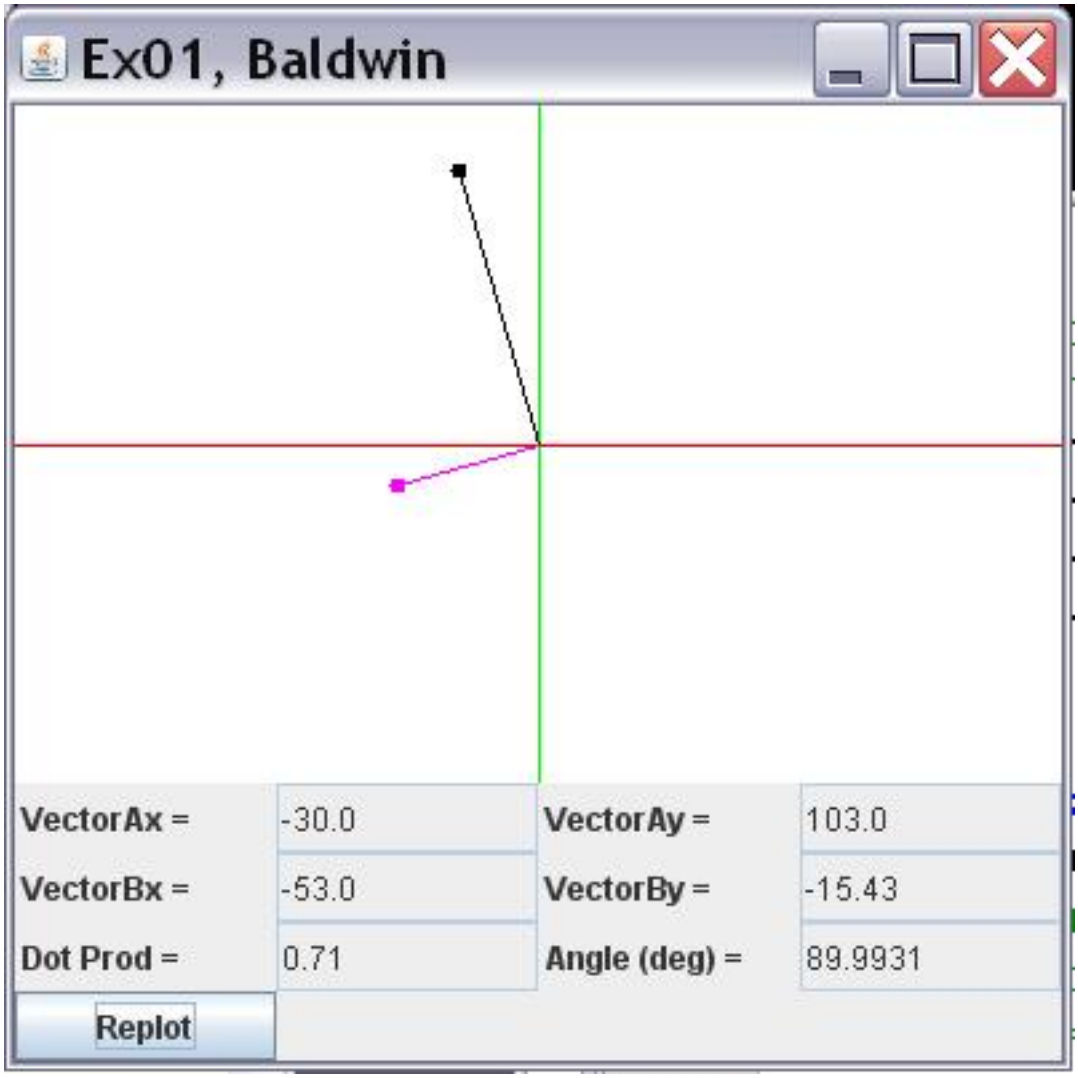

Figure 17: Output from Exercise 1.

12.2 Exercise 2

Using Java and the game-math library named **GM02** , or using a different programming environment of your choice, write a program that behaves as follows.

When the program starts running, an image similar to Figure 18 (p. 72) appears on the screen. Each of the text fields is blank and the drawing area at the top is also blank.

Each time you click the **Plot** button, the program generates three random values in the general range of -128 to 127. The first two values are used as the x and y values for a 3D vector. (Set the z-value for the

vector to 0.0.) The three values are displayed in the fields labeled **VecAx** , **VecAy** , and **VecAz** . Also, the three values are used to create and draw a black 3D vector with its tail at the origin as shown in Figure 18 (p. 72) .

The third random value is used as the x value for a second 3D vector. (Set the z-value for the vector to 0.0.) Those two values are displayed in the fields labeled **VecBx** and **VecBz** . A y-value is computed that will cause that vector to be perpendicular to the black vector. That value is displayed in the field labeled **VecBy** and the three values are used to draw the magenta vector shown in Figure 18 (p. 72) .

The dot product between the two vectors is computed and displayed in the field labeled **Dot Prod** . The angle between the two vectors is computed and displayed in the field labeled **Angle (deg)** .

If the two vectors are perpendicular, the dot product should be close to zero and the angle should be very close to 90 degrees.

Cause your name to appear in the screen output in some manner.

Output from Exercise 2.

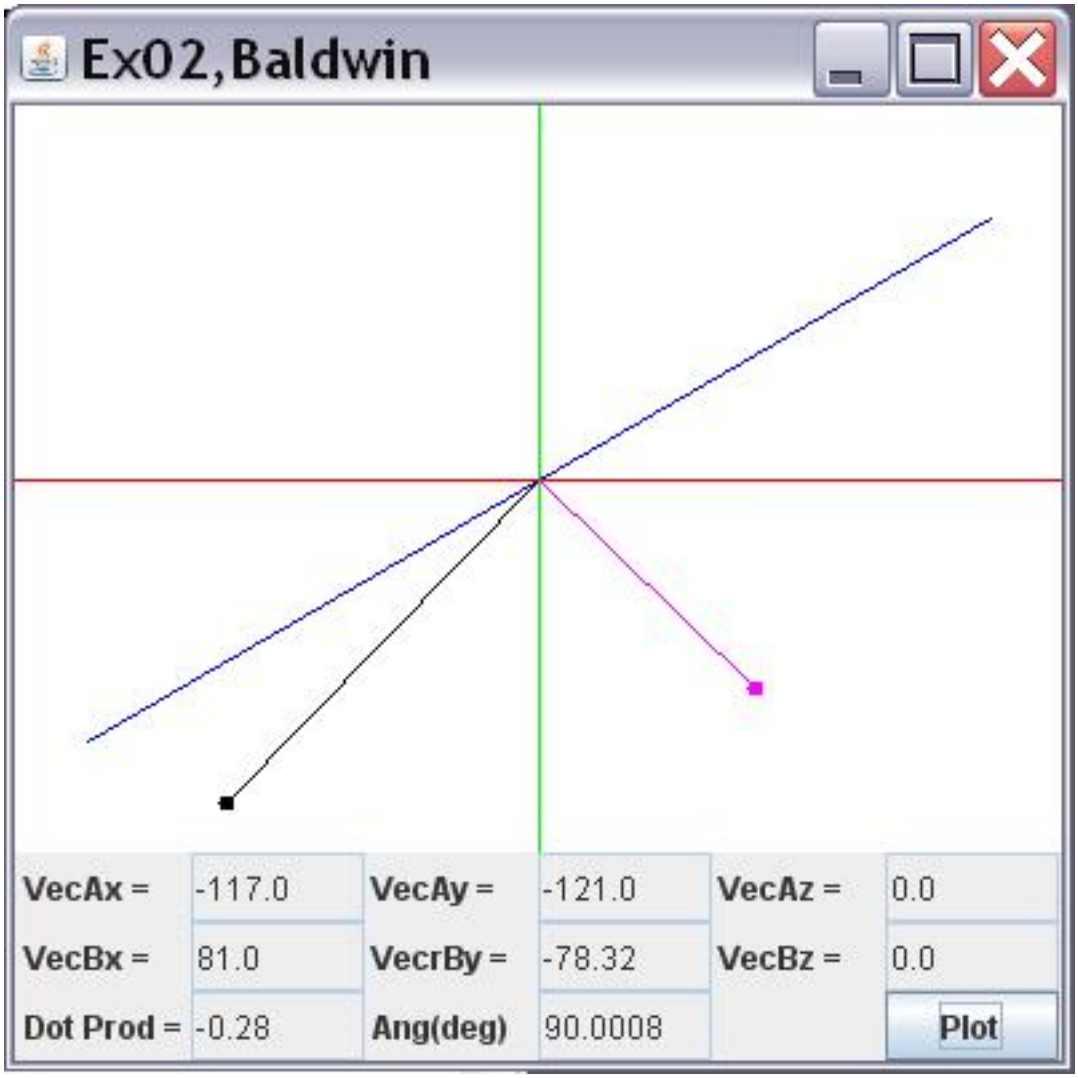

Figure 18: Output from Exercise 2.

12.3 Exercise 3

Using Java and the game-math library named **GM02** , or using a different programming environment of your choice, write a program that behaves as follows.

When the program starts running, an image similar to Figure 19 (p. 74) appears on the screen. Each of the text fields is blank and the drawing area at the top is also blank.

Each time you click the **Plot** button, the program generates five random values in the general range of -128 to 127. The first three values are used as the x, y, and z values for a vector. The three values are

displayed in the fields labeled **VecAx** , **VecAy** , and **VecAz** . Also, the three values are used to create and draw a black vector with its tail at the origin as shown in Figure 19 (p. 74) .

The fourth and fifth random values are used as the x and y values for a second vector. They are displayed in the fields labeled **VecBx** and **VecBy** . A z-value is computed that will cause that vector to be perpendicular to the black vector. That value is displayed in the field labeled **VecBz** and the three values are used to draw the magenta vector shown in Figure 19 (p. 74) .

The dot product between the two vectors is computed and displayed in the field labeled **Dot Prod** . The angle between the two vectors is computed and displayed in the field labeled **Angle (deg)** .

If the two vectors are perpendicular, the dot product should be close to zero and the angle should be very close to 90 degrees.

Cause your name to appear in the screen output in some manner.

Output from Exercise 3.

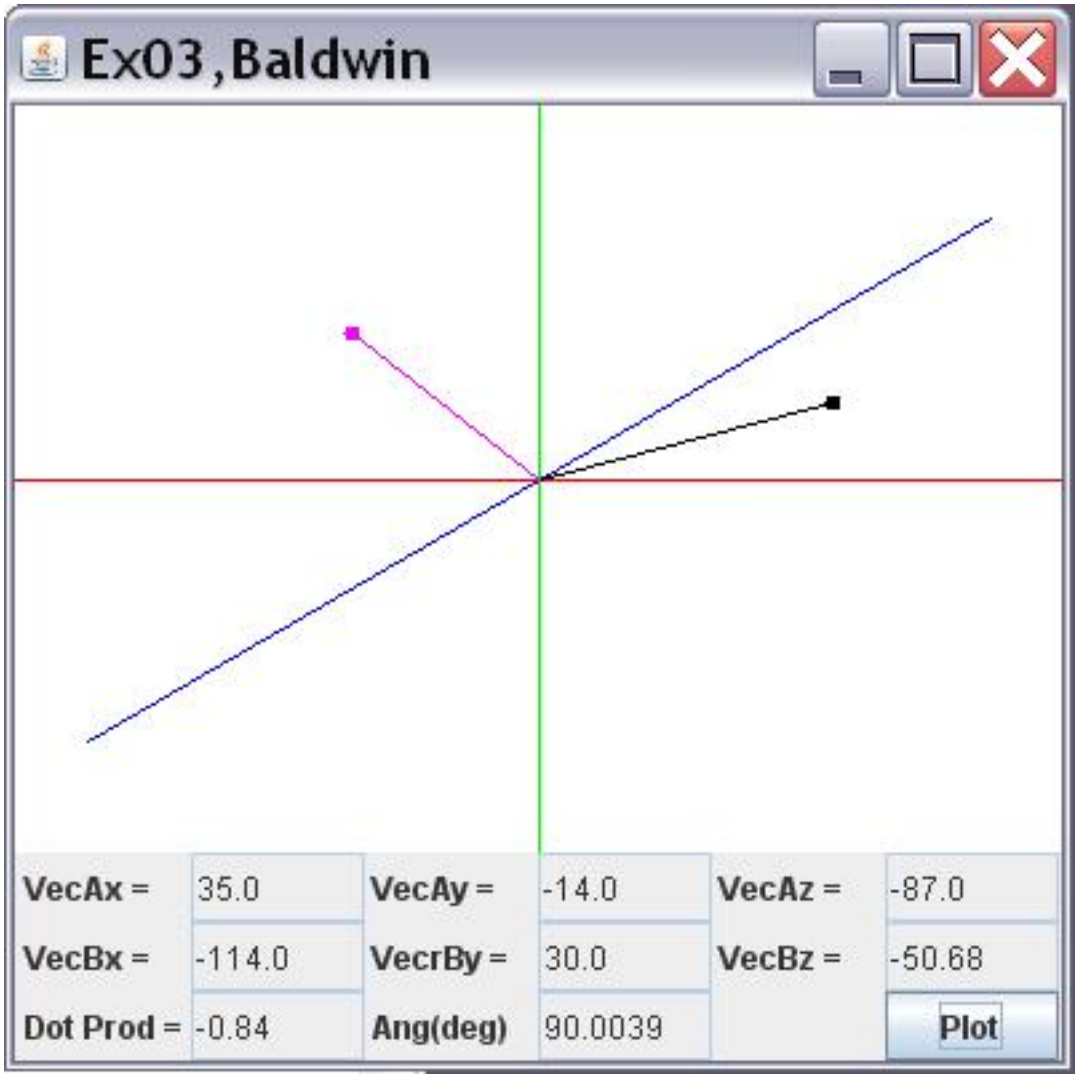

Figure 19: Output from Exercise 3.

-end-