/\*\*

\* File: Aquarium.java

\*

\* Description: A virtual aquarium with swimmers (Anglefish, Clownfish,

\* and Seahorses) and crawlers (Crabs, Lobsters, and Snails).

\*

\* Dr. Bower's Documentation: This code is original, but the

\* inspiration for the virtual aquarium comes from Dr. Schweitzer.

\*

\* Cadet Documentation: INSERT YOUR \_DETAILED\_ DOCUMENTATION STATEMENT HERE!

\*

\* @author Randall.Bower

\* @author YOUR NAME HERE

\*/

import java.awt.\*;

import java.awt.event.\*;

import java.awt.image.BufferedImage;

import java.io.File;

import java.io.IOException;

import javax.imageio.ImageIO;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.Timer;

public class Aquarium

{

/\*\*

\* The dimensions here are the dimensions of a "window" into an

\* aquarium. Creatures are allowed to swim or crawl outside the

\* bounds of this window as if they were just out of view of the

\* window, but will later return.

\*

\* The width of the aquarium should fit on most laptop screens.

\* For best results, use 800, 1024, or 1200.

\*/

public static final int WIDTH = 1200;

/\*\*

\* The height of the aquarium is based on width to get the

\* visually appealing 4:3 ratio.

\*/

public static final int HEIGHT = WIDTH \* 3 / 4;

/\*\*

\* The depth of the aquarium is the visual depth, front to back,

\* not the water depth from the top. This is rather arbitrarily

\* set to one-half of the height.

\*/

public static final int DEPTH = HEIGHT / 2;

/\*\*

\* The main method simply creates a new Aquarium object.

\*

\* @param args Command line arguments; ignored by this application.

\*/

public static void main( String[] args )

{

new Aquarium();

}

/\*\*

\* This constructor creates a new aquarium and starts a timer

\* that triggers the AquariumPanel's actionPerformed method

\* which controls the animation.

\*/

public Aquarium()

{

// Create the frame with the title "Aquarium".

JFrame aquariumFrame = new JFrame( "Aquarium" );

aquariumFrame.setDefaultCloseOperation( JFrame.EXIT\_ON\_CLOSE );

// Set the location of the frame slightly offset from the

// upper-left corner of the screen.

aquariumFrame.setLocation( 16, 16 );

// Set the size of the frame with a few extra pixels for the border and

// title bar so the panel inside the frame is exactly WIDTH x HEIGHT.

aquariumFrame.setSize( WIDTH + 16, HEIGHT + 38 );

// Create the aquarium panel and add it to the frame.

AquariumPanel aquariumPanel = new AquariumPanel();

aquariumFrame.add( aquariumPanel );

// Make the frame visible.

aquariumFrame.setVisible( true );

// Create and start a timer that triggers the actionPerfermed

// method in the aquarium panel every 50 milliseconds for a

// roughly 20 frames-per-second animation.

new Timer( 50, aquariumPanel ).start();

}

/\*\*

\* Inner class extending JPanel to inherit the paint functionality.

\* This class also implements the ActionListener interface so it

\* can be used with a Timer to create the animation.

\*/

class AquariumPanel extends JPanel implements ActionListener

{

/\*\* Array of bubbles contained in the aquarium. \*/

private Bubble[] bubbles;

/\*\* Array of angelfish contained in the aquarium. \*/

private Angelfish[] angelfish;

/\*\* Array of clownfish contained in the aquarium. \*/

private Clownfish[] clownfish;

/\*\* Array of seahorses contained in the aquarium. \*/

private Seahorse[] seahorses;

/\*\* Array of crabs contained in the aquarium. \*/

private Crab[] crabs;

/\*\* Array of lobsters contained in the aquarium. \*/

private Lobster[] lobsters;

/\*\* Array of snails contained in the aquarium. \*/

private Snail[] snails;

/\*\* The background image displayed in the aquarium. \*/

private BufferedImage background;

/\*\* How many of each aquatic creature in the aquarium. \*/

private static final int N = 3;

/\*\*

\* The constructor loads all images and creates arrays of creatures.

\*/

public AquariumPanel()

{

super();

// Load all images here so there is only one copy of each in memory.

// Only the background image needs to be held as a class variable;

// pointers to the other images will be kept in each creature object.

this.background = loadImage( "Background.jpg" );

BufferedImage bubbleImage = loadImage( "Bubble.png" );

BufferedImage angelfishImage = loadImage( "Angelfish.png" );

BufferedImage clownfishImage = loadImage( "Clownfish.png" );

BufferedImage seahorseImage = loadImage( "Seahorse.png" );

BufferedImage crabImage = loadImage( "Crab.png" );

BufferedImage lobsterImage = loadImage( "Lobster.png" );

BufferedImage snailImage = loadImage( "Snail.png" );

// Create the bubbles. Four bubbles for each aquatic creature.

this.bubbles = new Bubble[ N \* 4 ];

for( int i = 0; i < this.bubbles.length; i++ )

{

this.bubbles[ i ] = new Bubble( bubbleImage );

}

/\*

\* PROBLEM: In the code below, one loop initializes all creatures.

\* This is possible because there are N of each creature. What if

\* we wanted a different number of each creature in the aquarium?

\* For example, two angelfish, three clownfish, eight snails, etc.

\*/

// Create the aquatic creatures.

this.angelfish = new Angelfish[ N ];

this.clownfish = new Clownfish[ N ];

this.seahorses = new Seahorse[ N ];

this.crabs = new Crab[ N ];

this.lobsters = new Lobster[ N ];

this.snails = new Snail[ N ];

for( int i = 0; i < N; i++ )

{

this.angelfish[ i ] = new Angelfish( angelfishImage );

this.clownfish[ i ] = new Clownfish( clownfishImage );

this.seahorses[ i ] = new Seahorse( seahorseImage );

this.crabs[ i ] = new Crab( crabImage );

this.lobsters[ i ] = new Lobster( lobsterImage );

this.snails[ i ] = new Snail( snailImage );

}

}

/\*\*

\* The actionPerformed event is fired by the timer created in the

\* Aquarium constructor.

\*

\* @param ae ActionEvent object associated with this event.

\*/

@Override

public void actionPerformed( ActionEvent ae )

{

// First, ask all the bubbles to move.

for( int i = 0; i < this.bubbles.length; i++ )

{

this.bubbles[ i ].move();

}

/\*

\* PROBLEM: In the code below, one loop moves all the creatures.

\* This is possible because there are N of each creature. What if

\* we wanted a different number of each creature in the aquarium?

\* For example, two angelfish, three clownfish, eight snails, etc.

\*/

// Ask all the aquatic creatures to move.

for( int i = 0; i < N; i++ )

{

this.angelfish[ i ].move();

this.clownfish[ i ].move();

this.seahorses[ i ].move();

this.crabs[ i ].move();

this.lobsters[ i ].move();

this.snails[ i ].move();

}

// See if any of the aquatic creatures have met any other creatures.

for( int i = 0; i < N; i++ )

{

for( int j = 0; j < N; j++ )

{

if( met( this.angelfish[ i ], this.angelfish[ j ] ) )

{

this.angelfish[ i ].meet();

this.angelfish[ j ].meet();

}

if( met( this.angelfish[ i ], this.clownfish[ j ] ) )

{

this.angelfish[ i ].meet();

this.clownfish[ j ].meet();

}

if( met( this.angelfish[ i ], this.seahorses[ j ] ) )

{

this.angelfish[ i ].meet();

// Seahorses do nothing when they meet another creature.

}

if( met( this.angelfish[ i ], this.crabs[ j ] ) )

{

this.angelfish[ i ].meet();

this.crabs[ j ].meet();

}

if( met( this.angelfish[ i ], this.lobsters[ j ] ) )

{

this.angelfish[ i ].meet();

this.lobsters[ j ].meet();

}

if( met( this.angelfish[ i ], this.snails[ j ] ) )

{

this.angelfish[ i ].meet();

// Snails do nothing when they meet another creature.

}

if( met( this.clownfish[ i ], this.clownfish[ j ] ) )

{

this.clownfish[ i ].meet();

this.clownfish[ j ].meet();

}

if( met( this.clownfish[ i ], this.seahorses[ j ] ) )

{

this.clownfish[ i ].meet();

// Seahorses do nothing when they meet another creature.

}

if( met( this.clownfish[ i ], this.crabs[ j ] ) )

{

this.clownfish[ i ].meet();

this.crabs[ j ].meet();

}

if( met( this.clownfish[ i ], this.lobsters[ j ] ) )

{

this.clownfish[ i ].meet();

this.lobsters[ j ].meet();

}

if( met( this.clownfish[ i ], this.snails[ j ] ) )

{

this.clownfish[ i ].meet();

// Snails do nothing when they meet another creature.

}

if( met( this.seahorses[ i ], this.crabs[ j ] ) )

{

// Seahorses do nothing when they meet another creature.

this.crabs[ j ].meet();

}

if( met( this.seahorses[ i ], this.lobsters[ j ] ) )

{

// Seahorses do nothing when they meet another creature.

this.lobsters[ j ].meet();

}

if( met( this.crabs[ i ], this.crabs[ j ] ) )

{

this.crabs[ i ].meet();

this.crabs[ j ].meet();

}

if( met( this.crabs[ i ], this.lobsters[ j ] ) )

{

this.crabs[ i ].meet();

this.lobsters[ j ].meet();

}

if( met( this.crabs[ i ], this.snails[ j ] ) )

{

this.crabs[ i ].meet();

// Snails do nothing when they meet another creature.

}

if( met( this.lobsters[ i ], this.lobsters[ j ] ) )

{

this.lobsters[ i ].meet();

this.lobsters[ j ].meet();

}

if( met( this.lobsters[ i ], this.snails[ j ] ) )

{

this.lobsters[ i ].meet();

// Snails do nothing when they meet another creature.

}

}

}

// Sort the creatures so some will appear "behind" others.

// See comment on the sort method below.

sortBubbles( this.bubbles );

sortAngelfish( this.angelfish );

sortClownfish( this.clownfish );

sortSeahorses( this.seahorses );

sortCrabs( this.crabs );

sortLobsters( this.lobsters );

sortSnails( this.snails );

// Finally, after all creatures have moved and all meetings are

// taken care of, repaint the entire aquarium. NOTE: None of the

// above code tried to paint anything!! Everything is moved at

// once and only then is the entire aquarium repainted.

repaint();

}

/\*\*

\* This method paints the aquarium.

\*

\* @param g The Graphics object to use to paint.

\*/

@Override

public void paintComponent( Graphics g )

{

// Make sure the JPanel does whatever it needs to do when painted.

super.paintComponent( g );

// Draw the background image.

g.drawImage( this.background, 0, 0,

this.getWidth(), this.getHeight(), null );

// Paint all the bubbles.

for( int i = 0; i < this.bubbles.length; i++ )

{

this.bubbles[ i ].paint( g );

}

/\*

\* PROBLEM: The last bit of code in the actionPerformed method sorts

\* each of these arrays so creatures closer to the back of the aquarium

\* will be drawn before creatures closer to the front. This gives the

\* illusion of depth when creatures closer to the front overlap those

\* closer to the back.

\*

\* The problem is what happens when the first angelfish is closer to

\* the front than the first clownfish, but the angelfish is painted

\* before the clownfish because that array is first in the loop below?

\*

\* In other words, ensuring the angelfish are painted in the correct

\* order with respect to all other angelfish is easy, but doing so

\* with respect to all other creatures of any kind is quite a task

\* (and therefore not even attempted in this solution).

\*/

// Paint all the aquatic creatures.

for( int i = 0; i < N; i++ )

{

this.angelfish[ i ].paint( g );

this.clownfish[ i ].paint( g );

this.seahorses[ i ].paint( g );

this.crabs[ i ].paint( g );

this.lobsters[ i ].paint( g );

this.snails[ i ].paint( g );

}

}

/\*\*

\* This method sorts the array of bubbles so those with the smallest

\* z positions are first. This means when the bubbles are drawn, the

\* bubbles with smaller z values (closer to the back of the tank)

\* will appear to be behind the other bubbles when they overlap.

\* The method uses a simple selection sort algorithm.

\*/

private void sortBubbles( Bubble[] array )

{

Bubble temp; // Temporary varuable to use when swapping.

for( int i = 0; i < array.length - 1; i++ )

{

int smallest = i;

for( int j = i + 1; j < array.length; j++ )

{

if( array[ j ].getZ() < array[ smallest ].getZ() )

{

smallest = j;

}

}

temp = array[ i ];

array[ i ] = array[ smallest ];

array[ smallest ] = temp;

}

}

/\*\*

\* This method sorts an array of creatures so those with the smallest

\* z positions are first. This means when the creatures are drawn, the

\* creatures with smaller z values (closer to the back of the tank)

\* will appear to be behind the other creatures when they overlap.

\* The method uses a simple selection sort algorithm.

\*/

private void sortAngelfish( Angelfish[] array )

{

Angelfish temp; // Temporary varuable to use when swapping.

for( int i = 0; i < array.length - 1; i++ )

{

int smallest = i;

for( int j = i + 1; j < array.length; j++ )

{

if( array[ j ].getZ() < array[ smallest ].getZ() )

{

smallest = j;

}

}

temp = array[ i ];

array[ i ] = array[ smallest ];

array[ smallest ] = temp;

}

}

/\*\*

\* This method sorts an array of creatures so those with the smallest

\* z positions are first. This means when the creatures are drawn, the

\* creatures with smaller z values (closer to the back of the tank)

\* will appear to be behind the other creatures when they overlap.

\* The method uses a simple selection sort algorithm.

\*/

private void sortClownfish( Clownfish[] array )

{

Clownfish temp; // Temporary varuable to use when swapping.

for( int i = 0; i < array.length - 1; i++ )

{

int smallest = i;

for( int j = i + 1; j < array.length; j++ )

{

if( array[ j ].getZ() < array[ smallest ].getZ() )

{

smallest = j;

}

}

temp = array[ i ];

array[ i ] = array[ smallest ];

array[ smallest ] = temp;

}

}

/\*\*

\* This method sorts an array of creatures so those with the smallest

\* z positions are first. This means when the creatures are drawn, the

\* creatures with smaller z values (closer to the back of the tank)

\* will appear to be behind the other creatures when they overlap.

\* The method uses a simple selection sort algorithm.

\*/

private void sortSeahorses( Seahorse[] array )

{

Seahorse temp; // Temporary varuable to use when swapping.

for( int i = 0; i < array.length - 1; i++ )

{

int smallest = i;

for( int j = i + 1; j < array.length; j++ )

{

if( array[ j ].getZ() < array[ smallest ].getZ() )

{

smallest = j;

}

}

temp = array[ i ];

array[ i ] = array[ smallest ];

array[ smallest ] = temp;

}

}

/\*\*

\* This method sorts an array of creatures so those with the smallest

\* z positions are first. This means when the creatures are drawn, the

\* creatures with smaller z values (closer to the back of the tank)

\* will appear to be behind the other creatures when they overlap.

\* The method uses a simple selection sort algorithm.

\*/

private void sortCrabs( Crab[] array )

{

Crab temp; // Temporary varuable to use when swapping.

for( int i = 0; i < array.length - 1; i++ )

{

int smallest = i;

for( int j = i + 1; j < array.length; j++ )

{

if( array[ j ].getZ() < array[ smallest ].getZ() )

{

smallest = j;

}

}

temp = array[ i ];

array[ i ] = array[ smallest ];

array[ smallest ] = temp;

}

}

/\*\*

\* This method sorts an array of creatures so those with the smallest

\* z positions are first. This means when the creatures are drawn, the

\* creatures with smaller z values (closer to the back of the tank)

\* will appear to be behind the other creatures when they overlap.

\* The method uses a simple selection sort algorithm.

\*/

private void sortLobsters( Lobster[] array )

{

Lobster temp; // Temporary varuable to use when swapping.

for( int i = 0; i < array.length - 1; i++ )

{

int smallest = i;

for( int j = i + 1; j < array.length; j++ )

{

if( array[ j ].getZ() < array[ smallest ].getZ() )

{

smallest = j;

}

}

temp = array[ i ];

array[ i ] = array[ smallest ];

array[ smallest ] = temp;

}

}

/\*\*

\* This method sorts an array of creatures so those with the smallest

\* z positions are first. This means when the creatures are drawn, the

\* creatures with smaller z values (closer to the back of the tank)

\* will appear to be behind the other creatures when they overlap.

\* The method uses a simple selection sort algorithm.

\*/

private void sortSnails( Snail[] array )

{

Snail temp; // Temporary varuable to use when swapping.

for( int i = 0; i < array.length - 1; i++ )

{

int smallest = i;

for( int j = i + 1; j < array.length; j++ )

{

if( array[ j ].getZ() < array[ smallest ].getZ() )

{

smallest = j;

}

}

temp = array[ i ];

array[ i ] = array[ smallest ];

array[ smallest ] = temp;

}

}

/\*\*

\* Determine if the two given aquatic creatures have met.

\*

\* @param a First creature of the pair to be checked.

\* @param b Second creature of the pair to be checked.

\* @return true of the two creatures have met; false otherwise.

\*/

private boolean met( Angelfish a, Angelfish b )

{

double dx = Math.abs( a.getX() - b.getX() );

if( a.getX() <= b.getX() && dx <= a.getWidth() ||

b.getX() <= a.getX() && dx <= b.getWidth() )

{

double dy = Math.abs( a.getY() - b.getY() );

if( a.getY() <= b.getY() && dy <= a.getHeight() ||

b.getY() <= a.getY() && dy <= b.getHeight() )

{

double dz = Math.abs( a.getZ() - b.getZ() );

if( a.getZ() <= b.getZ() && dz <= a.getDepth() ||

b.getZ() <= a.getZ() && dz <= b.getDepth() )

{

// Creatures only "meet" if they are going in opposite directions.

if( a.getSpeed() < 0 && b.getSpeed() > 0 ||

a.getSpeed() > 0 && b.getSpeed() < 0 )

{

return true;

}

}

}

}

return false;

}

/\*\*

\* Determine if the two given aquatic creatures have met.

\*

\* @param a First creature of the pair to be checked.

\* @param b Second creature of the pair to be checked.

\* @return true of the two creatures have met; false otherwise.

\*/

private boolean met( Angelfish a, Clownfish b )

{

double dx = Math.abs( a.getX() - b.getX() );

if( a.getX() <= b.getX() && dx <= a.getWidth() ||

b.getX() <= a.getX() && dx <= b.getWidth() )

{

double dy = Math.abs( a.getY() - b.getY() );

if( a.getY() <= b.getY() && dy <= a.getHeight() ||

b.getY() <= a.getY() && dy <= b.getHeight() )

{

double dz = Math.abs( a.getZ() - b.getZ() );

if( a.getZ() <= b.getZ() && dz <= a.getDepth() ||

b.getZ() <= a.getZ() && dz <= b.getDepth() )

{

// Creatures only "meet" if they are going in opposite directions.

if( a.getSpeed() < 0 && b.getSpeed() > 0 ||

a.getSpeed() > 0 && b.getSpeed() < 0 )

{

return true;

}

}

}

}

return false;

}

/\*\*

\* Determine if the two given aquatic creatures have met.

\*

\* @param a First creature of the pair to be checked.

\* @param b Second creature of the pair to be checked.

\* @return true of the two creatures have met; false otherwise.

\*/

private boolean met( Angelfish a, Seahorse b )

{

double dx = Math.abs( a.getX() - b.getX() );

if( a.getX() <= b.getX() && dx <= a.getWidth() ||

b.getX() <= a.getX() && dx <= b.getWidth() )

{

double dy = Math.abs( a.getY() - b.getY() );

if( a.getY() <= b.getY() && dy <= a.getHeight() ||

b.getY() <= a.getY() && dy <= b.getHeight() )

{

double dz = Math.abs( a.getZ() - b.getZ() );

if( a.getZ() <= b.getZ() && dz <= a.getDepth() ||

b.getZ() <= a.getZ() && dz <= b.getDepth() )

{

// Creatures only "meet" if they are going in opposite directions.

if( a.getSpeed() < 0 && b.getSpeed() > 0 ||

a.getSpeed() > 0 && b.getSpeed() < 0 )

{

return true;

}

}

}

}

return false;

}

/\*\*

\* Determine if the two given aquatic creatures have met.

\*

\* @param a First creature of the pair to be checked.

\* @param b Second creature of the pair to be checked.

\* @return true of the two creatures have met; false otherwise.

\*/

private boolean met( Angelfish a, Crab b )

{

double dx = Math.abs( a.getX() - b.getX() );

if( a.getX() <= b.getX() && dx <= a.getWidth() ||

b.getX() <= a.getX() && dx <= b.getWidth() )

{

double dy = Math.abs( a.getY() - b.getY() );

if( a.getY() <= b.getY() && dy <= a.getHeight() ||

b.getY() <= a.getY() && dy <= b.getHeight() )

{

double dz = Math.abs( a.getZ() - b.getZ() );

if( a.getZ() <= b.getZ() && dz <= a.getDepth() ||

b.getZ() <= a.getZ() && dz <= b.getDepth() )

{

// Creatures only "meet" if they are going in opposite directions.

if( a.getSpeed() < 0 && b.getSpeed() > 0 ||

a.getSpeed() > 0 && b.getSpeed() < 0 )

{

return true;

}

}

}

}

return false;

}

/\*\*

\* Determine if the two given aquatic creatures have met.

\*

\* @param a First creature of the pair to be checked.

\* @param b Second creature of the pair to be checked.

\* @return true of the two creatures have met; false otherwise.

\*/

private boolean met( Angelfish a, Lobster b )

{

double dx = Math.abs( a.getX() - b.getX() );

if( a.getX() <= b.getX() && dx <= a.getWidth() ||

b.getX() <= a.getX() && dx <= b.getWidth() )

{

double dy = Math.abs( a.getY() - b.getY() );

if( a.getY() <= b.getY() && dy <= a.getHeight() ||

b.getY() <= a.getY() && dy <= b.getHeight() )

{

double dz = Math.abs( a.getZ() - b.getZ() );

if( a.getZ() <= b.getZ() && dz <= a.getDepth() ||

b.getZ() <= a.getZ() && dz <= b.getDepth() )

{

// Creatures only "meet" if they are going in opposite directions.

if( a.getSpeed() < 0 && b.getSpeed() > 0 ||

a.getSpeed() > 0 && b.getSpeed() < 0 )

{

return true;

}

}

}

}

return false;

}

/\*\*

\* Determine if the two given aquatic creatures have met.

\*

\* @param a First creature of the pair to be checked.

\* @param b Second creature of the pair to be checked.

\* @return true of the two creatures have met; false otherwise.

\*/

private boolean met( Angelfish a, Snail b )

{

double dx = Math.abs( a.getX() - b.getX() );

if( a.getX() <= b.getX() && dx <= a.getWidth() ||

b.getX() <= a.getX() && dx <= b.getWidth() )

{

double dy = Math.abs( a.getY() - b.getY() );

if( a.getY() <= b.getY() && dy <= a.getHeight() ||

b.getY() <= a.getY() && dy <= b.getHeight() )

{

double dz = Math.abs( a.getZ() - b.getZ() );

if( a.getZ() <= b.getZ() && dz <= a.getDepth() ||

b.getZ() <= a.getZ() && dz <= b.getDepth() )

{

// Creatures only "meet" if they are going in opposite directions.

if( a.getSpeed() < 0 && b.getSpeed() > 0 ||

a.getSpeed() > 0 && b.getSpeed() < 0 )

{

return true;

}

}

}

}

return false;

}

/\*\*

\* Determine if the two given aquatic creatures have met.

\*

\* @param a First creature of the pair to be checked.

\* @param b Second creature of the pair to be checked.

\* @return true of the two creatures have met; false otherwise.

\*/

private boolean met( Clownfish a, Clownfish b )

{

double dx = Math.abs( a.getX() - b.getX() );

if( a.getX() <= b.getX() && dx <= a.getWidth() ||

b.getX() <= a.getX() && dx <= b.getWidth() )

{

double dy = Math.abs( a.getY() - b.getY() );

if( a.getY() <= b.getY() && dy <= a.getHeight() ||

b.getY() <= a.getY() && dy <= b.getHeight() )

{

double dz = Math.abs( a.getZ() - b.getZ() );

if( a.getZ() <= b.getZ() && dz <= a.getDepth() ||

b.getZ() <= a.getZ() && dz <= b.getDepth() )

{

// Creatures only "meet" if they are going in opposite directions.

if( a.getSpeed() < 0 && b.getSpeed() > 0 ||

a.getSpeed() > 0 && b.getSpeed() < 0 )

{

return true;

}

}

}

}

return false;

}

/\*\*

\* Determine if the two given aquatic creatures have met.

\*

\* @param a First creature of the pair to be checked.

\* @param b Second creature of the pair to be checked.

\* @return true of the two creatures have met; false otherwise.

\*/

private boolean met( Clownfish a, Seahorse b )

{

double dx = Math.abs( a.getX() - b.getX() );

if( a.getX() <= b.getX() && dx <= a.getWidth() ||

b.getX() <= a.getX() && dx <= b.getWidth() )

{

double dy = Math.abs( a.getY() - b.getY() );

if( a.getY() <= b.getY() && dy <= a.getHeight() ||

b.getY() <= a.getY() && dy <= b.getHeight() )

{

double dz = Math.abs( a.getZ() - b.getZ() );

if( a.getZ() <= b.getZ() && dz <= a.getDepth() ||

b.getZ() <= a.getZ() && dz <= b.getDepth() )

{

// Creatures only "meet" if they are going in opposite directions.

if( a.getSpeed() < 0 && b.getSpeed() > 0 ||

a.getSpeed() > 0 && b.getSpeed() < 0 )

{

return true;

}

}

}

}

return false;

}

/\*\*

\* Determine if the two given aquatic creatures have met.

\*

\* @param a First creature of the pair to be checked.

\* @param b Second creature of the pair to be checked.

\* @return true of the two creatures have met; false otherwise.

\*/

private boolean met( Clownfish a, Crab b )

{

double dx = Math.abs( a.getX() - b.getX() );

if( a.getX() <= b.getX() && dx <= a.getWidth() ||

b.getX() <= a.getX() && dx <= b.getWidth() )

{

double dy = Math.abs( a.getY() - b.getY() );

if( a.getY() <= b.getY() && dy <= a.getHeight() ||

b.getY() <= a.getY() && dy <= b.getHeight() )

{

double dz = Math.abs( a.getZ() - b.getZ() );

if( a.getZ() <= b.getZ() && dz <= a.getDepth() ||

b.getZ() <= a.getZ() && dz <= b.getDepth() )

{

// Creatures only "meet" if they are going in opposite directions.

if( a.getSpeed() < 0 && b.getSpeed() > 0 ||

a.getSpeed() > 0 && b.getSpeed() < 0 )

{

return true;

}

}

}

}

return false;

}

/\*\*

\* Determine if the two given aquatic creatures have met.

\*

\* @param a First creature of the pair to be checked.

\* @param b Second creature of the pair to be checked.

\* @return true of the two creatures have met; false otherwise.

\*/

private boolean met( Clownfish a, Lobster b )

{

double dx = Math.abs( a.getX() - b.getX() );

if( a.getX() <= b.getX() && dx <= a.getWidth() ||

b.getX() <= a.getX() && dx <= b.getWidth() )

{

double dy = Math.abs( a.getY() - b.getY() );

if( a.getY() <= b.getY() && dy <= a.getHeight() ||

b.getY() <= a.getY() && dy <= b.getHeight() )

{

double dz = Math.abs( a.getZ() - b.getZ() );

if( a.getZ() <= b.getZ() && dz <= a.getDepth() ||

b.getZ() <= a.getZ() && dz <= b.getDepth() )

{

// Creatures only "meet" if they are going in opposite directions.

if( a.getSpeed() < 0 && b.getSpeed() > 0 ||

a.getSpeed() > 0 && b.getSpeed() < 0 )

{

return true;

}

}

}

}

return false;

}

/\*\*

\* Determine if the two given aquatic creatures have met.

\*

\* @param a First creature of the pair to be checked.

\* @param b Second creature of the pair to be checked.

\* @return true of the two creatures have met; false otherwise.

\*/

private boolean met( Clownfish a, Snail b )

{

double dx = Math.abs( a.getX() - b.getX() );

if( a.getX() <= b.getX() && dx <= a.getWidth() ||

b.getX() <= a.getX() && dx <= b.getWidth() )

{

double dy = Math.abs( a.getY() - b.getY() );

if( a.getY() <= b.getY() && dy <= a.getHeight() ||

b.getY() <= a.getY() && dy <= b.getHeight() )

{

double dz = Math.abs( a.getZ() - b.getZ() );

if( a.getZ() <= b.getZ() && dz <= a.getDepth() ||

b.getZ() <= a.getZ() && dz <= b.getDepth() )

{

// Creatures only "meet" if they are going in opposite directions.

if( a.getSpeed() < 0 && b.getSpeed() > 0 ||

a.getSpeed() > 0 && b.getSpeed() < 0 )

{

return true;

}

}

}

}

return false;

}

/\*\*

\* Determine if the two given aquatic creatures have met.

\*

\* @param a First creature of the pair to be checked.

\* @param b Second creature of the pair to be checked.

\* @return true of the two creatures have met; false otherwise.

\*/

private boolean met( Seahorse a, Crab b )

{

double dx = Math.abs( a.getX() - b.getX() );

if( a.getX() <= b.getX() && dx <= a.getWidth() ||

b.getX() <= a.getX() && dx <= b.getWidth() )

{

double dy = Math.abs( a.getY() - b.getY() );

if( a.getY() <= b.getY() && dy <= a.getHeight() ||

b.getY() <= a.getY() && dy <= b.getHeight() )

{

double dz = Math.abs( a.getZ() - b.getZ() );

if( a.getZ() <= b.getZ() && dz <= a.getDepth() ||

b.getZ() <= a.getZ() && dz <= b.getDepth() )

{

// Creatures only "meet" if they are going in opposite directions.

if( a.getSpeed() < 0 && b.getSpeed() > 0 ||

a.getSpeed() > 0 && b.getSpeed() < 0 )

{

return true;

}

}

}

}

return false;

}

/\*\*

\* Determine if the two given aquatic creatures have met.

\*

\* @param a First creature of the pair to be checked.

\* @param b Second creature of the pair to be checked.

\* @return true of the two creatures have met; false otherwise.

\*/

private boolean met( Seahorse a, Lobster b )

{

double dx = Math.abs( a.getX() - b.getX() );

if( a.getX() <= b.getX() && dx <= a.getWidth() ||

b.getX() <= a.getX() && dx <= b.getWidth() )

{

double dy = Math.abs( a.getY() - b.getY() );

if( a.getY() <= b.getY() && dy <= a.getHeight() ||

b.getY() <= a.getY() && dy <= b.getHeight() )

{

double dz = Math.abs( a.getZ() - b.getZ() );

if( a.getZ() <= b.getZ() && dz <= a.getDepth() ||

b.getZ() <= a.getZ() && dz <= b.getDepth() )

{

// Creatures only "meet" if they are going in opposite directions.

if( a.getSpeed() < 0 && b.getSpeed() > 0 ||

a.getSpeed() > 0 && b.getSpeed() < 0 )

{

return true;

}

}

}

}

return false;

}

/\*\*

\* Determine if the two given aquatic creatures have met.

\*

\* @param a First creature of the pair to be checked.

\* @param b Second creature of the pair to be checked.

\* @return true of the two creatures have met; false otherwise.

\*/

private boolean met( Crab a, Crab b )

{

double dx = Math.abs( a.getX() - b.getX() );

if( a.getX() <= b.getX() && dx <= a.getWidth() ||

b.getX() <= a.getX() && dx <= b.getWidth() )

{

double dy = Math.abs( a.getY() - b.getY() );

if( a.getY() <= b.getY() && dy <= a.getHeight() ||

b.getY() <= a.getY() && dy <= b.getHeight() )

{

double dz = Math.abs( a.getZ() - b.getZ() );

if( a.getZ() <= b.getZ() && dz <= a.getDepth() ||

b.getZ() <= a.getZ() && dz <= b.getDepth() )

{

// Creatures only "meet" if they are going in opposite directions.

if( a.getSpeed() < 0 && b.getSpeed() > 0 ||

a.getSpeed() > 0 && b.getSpeed() < 0 )

{

return true;

}

}

}

}

return false;

}

/\*\*

\* Determine if the two given aquatic creatures have met.

\*

\* @param a First creature of the pair to be checked.

\* @param b Second creature of the pair to be checked.

\* @return true of the two creatures have met; false otherwise.

\*/

private boolean met( Crab a, Lobster b )

{

double dx = Math.abs( a.getX() - b.getX() );

if( a.getX() <= b.getX() && dx <= a.getWidth() ||

b.getX() <= a.getX() && dx <= b.getWidth() )

{

double dy = Math.abs( a.getY() - b.getY() );

if( a.getY() <= b.getY() && dy <= a.getHeight() ||

b.getY() <= a.getY() && dy <= b.getHeight() )

{

double dz = Math.abs( a.getZ() - b.getZ() );

if( a.getZ() <= b.getZ() && dz <= a.getDepth() ||

b.getZ() <= a.getZ() && dz <= b.getDepth() )

{

// Creatures only "meet" if they are going in opposite directions.

if( a.getSpeed() < 0 && b.getSpeed() > 0 ||

a.getSpeed() > 0 && b.getSpeed() < 0 )

{

return true;

}

}

}

}

return false;

}

/\*\*

\* Determine if the two given aquatic creatures have met.

\*

\* @param a First creature of the pair to be checked.

\* @param b Second creature of the pair to be checked.

\* @return true of the two creatures have met; false otherwise.

\*/

private boolean met( Crab a, Snail b )

{

double dx = Math.abs( a.getX() - b.getX() );

if( a.getX() <= b.getX() && dx <= a.getWidth() ||

b.getX() <= a.getX() && dx <= b.getWidth() )

{

double dy = Math.abs( a.getY() - b.getY() );

if( a.getY() <= b.getY() && dy <= a.getHeight() ||

b.getY() <= a.getY() && dy <= b.getHeight() )

{

double dz = Math.abs( a.getZ() - b.getZ() );

if( a.getZ() <= b.getZ() && dz <= a.getDepth() ||

b.getZ() <= a.getZ() && dz <= b.getDepth() )

{

// Creatures only "meet" if they are going in opposite directions.

if( a.getSpeed() < 0 && b.getSpeed() > 0 ||

a.getSpeed() > 0 && b.getSpeed() < 0 )

{

return true;

}

}

}

}

return false;

}

/\*\*

\* Determine if the two given aquatic creatures have met.

\*

\* @param a First creature of the pair to be checked.

\* @param b Second creature of the pair to be checked.

\* @return true of the two creatures have met; false otherwise.

\*/

private boolean met( Lobster a, Lobster b )

{

double dx = Math.abs( a.getX() - b.getX() );

if( a.getX() <= b.getX() && dx <= a.getWidth() ||

b.getX() <= a.getX() && dx <= b.getWidth() )

{

double dy = Math.abs( a.getY() - b.getY() );

if( a.getY() <= b.getY() && dy <= a.getHeight() ||

b.getY() <= a.getY() && dy <= b.getHeight() )

{

double dz = Math.abs( a.getZ() - b.getZ() );

if( a.getZ() <= b.getZ() && dz <= a.getDepth() ||

b.getZ() <= a.getZ() && dz <= b.getDepth() )

{

// Creatures only "meet" if they are going in opposite directions.

if( a.getSpeed() < 0 && b.getSpeed() > 0 ||

a.getSpeed() > 0 && b.getSpeed() < 0 )

{

return true;

}

}

}

}

return false;

}

/\*\*

\* Determine if the two given aquatic creatures have met.

\*

\* @param a First creature of the pair to be checked.

\* @param b Second creature of the pair to be checked.

\* @return true of the two creatures have met; false otherwise.

\*/

private boolean met( Lobster a, Snail b )

{

double dx = Math.abs( a.getX() - b.getX() );

if( a.getX() <= b.getX() && dx <= a.getWidth() ||

b.getX() <= a.getX() && dx <= b.getWidth() )

{

double dy = Math.abs( a.getY() - b.getY() );

if( a.getY() <= b.getY() && dy <= a.getHeight() ||

b.getY() <= a.getY() && dy <= b.getHeight() )

{

double dz = Math.abs( a.getZ() - b.getZ() );

if( a.getZ() <= b.getZ() && dz <= a.getDepth() ||

b.getZ() <= a.getZ() && dz <= b.getDepth() )

{

// Creatures only "meet" if they are going in opposite directions.

if( a.getSpeed() < 0 && b.getSpeed() > 0 ||

a.getSpeed() > 0 && b.getSpeed() < 0 )

{

return true;

}

}

}

}

return false;

}

/\*\*

\* This method loads an image with the given file name from the images

\* folder of the project.

\*

\* @param name File name of the image.

\* @return BufferedImage object representing the image;

\* null if there was an error loading the image.

\*/

private BufferedImage loadImage( String name )

{

BufferedImage image = null;

try

{

image = ImageIO.read( new File( "images/" + name ) );

}

catch( IOException e )

{

System.err.println( "Error loading image: " + name );

System.exit( 1 );

}

return image;

}

}

}