3 Technical Solution

3.1 Run.bat

```
java -jar "%~dp0RefractionSimulator.jar" %*
```

3.2 RefractionSimulator.java

```
package RefractionSim;
import java.awt.BorderLayout;
import java.awt.GraphicsEnvironment;
import javax.swing.JFrame;
import javax.swing.JPanel;

/**
 * Class for windows of the application also containing the public static main method which the system calls to start the application
 * @author William Platt
 *
 */
public class RefractionSimulator extends JFrame {
	private UIController userInterface;
	private JPanel content;

	/**
	 * Called by the system when the application is started. This creates an instance of RefractionSimulator and sets up the window
	 * @param args the parameter passed by the system which isn't needed in this application
	 */
	public static void main(String[] args) {
		JFrame window = new RefractionSimulator();
		window.setResizable(false);
		window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
	  window.setVisible(true);
	}

	/**
```

```
	 * Constructor for the RefractionSimulator class which sets properties for the window as well as generating the contents of the window and drawing it
	 */
	public RefractionSimulator() {
		super("Refraction Simulator");
		setSize(GraphicsEnvironment.getLocalGraphicsEnvironment().getMaximumWindowBounds().getSize()); // Set the window to fill all of the screen except for the task bar
		setResizable(false);
		setVisible(true);
		content = new JPanel();
		content.setLayout(new BorderLayout());
		userInterface = new UIController(this); // Generate the viewport, menu bar and properties panel
		content.add(userInterface.getViewport(), BorderLayout.CENTER); // Add the viewport to the centre area of the content component
		content.add(userInterface.getPropertiesPanel(), BorderLayout.WEST); // Add the properties panel to the far left area of the content component
		setContentPane(content); // Set the content component as the window's content pane
		setJMenuBar(userInterface.getMenuBar()); // Set the menu bar as the window's menu bar
		setLocation(0, 0); // Position the window in the top left of the available screen area
		this.revalidate(); // Process the layout of the window so that it can be drawn correctly
		this.repaint(); // Paint the contents of the window
	}

	/**
	 * Regenerates the properties panel for rayBox as the selected ray box and replaces the old the old properties panel
	 * @param rayBox the selected ray box which the properties panel will display information for. For an empty selection, null should be used as the parameter
	 */
	public void updatePropertiesPanel(RayBox rayBox) {
		BorderLayout layout = (BorderLayout)(content.getLayout());
		content.remove(layout.getLayoutComponent(BorderLayout.WEST)); // Remove the old properties panel from the window
		userInterface.buildPropertiesPanel(rayBox); // Regenerate the new properties panel
		content.add(userInterface.getPropertiesPanel(), BorderLayout.WEST); // Add the new panel to the window in replace of the old panel
		userInterface.getPropertiesPanel().revalidate(); // Process the layout of the properties panel so that it can be drawn correctly
		userInterface.getPropertiesPanel().repaint(); // Paint the contents of the properties panel
```

```
	}

	/**
	 * Regenerates the menu bar and replaces the old menu bar
	 */
	public void updateMenuBar() {
		userInterface.createMenuBar(); // Regenerate the menu bar with the new settings
		setJMenuBar(userInterface.getMenuBar()); // Set the window's menu bar as the new menu bar (replacing the old one)
		userInterface.getMenuBar().revalidate(); // Process the layout of the menu bar so that it can be drawn correctly
		userInterface.getMenuBar().repaint(); // Pain the contents of the menu bar
	}

}
```

3.3 UIController.java

```
package RefractionSim;
import java.awt.*;
import java.awt.event.*;
import java.io.File;
import java.io.IOException;
import java.util.Hashtable;
import javax.swing.*;
import javax.swing.border.TitledBorder;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;
import javax.swing.filechooser.FileNameExtensionFilter;

/**
 * Class for objects that generate the user interface
 * @author William Platt
 *
 */
public class UIController {
	private Viewport viewport;
	private JMenuBar menuBar;
	private JPanel propertiesPanel;
	private int fullWidth;
	private int fullHeight;
	private final static int PROPS_PANEL_WIDTH = 250;

	/**
	 * Constructor for the UIController class which generates the viewport, menu bar and properties panel
	 * @param window the window which will contain this new viewport, menu bar and properties panel
	 */
	public UIController(RefractionSimulator window) {
		fullWidth = (int)(window.getWidth() - window.getInsets().left - window.getInsets().right); // Store the
width of the inside of the window
		fullHeight = (int)(window.getHeight() - window.getInsets().top - window.getInsets().bottom); // Store
the height of the inside of the window
		createViewport();
		RayBox rayBox1 = new RayBox();
		viewport.addRayBox(rayBox1);
		createMenuBar();
```

```
			buildPropertiesPanel(rayBox1);
		}

		/**
		 * Generates a viewport to fill all of the space in the window left by the properties panel and menu bar. The
viewport can then be retrieved using getViewport().
		 */
		private void createViewport() {
			// Create a menu bar to find out how tall the real menu bar will be and therefore how much space there
is for the viewport
			JMenuBar dummyMenuBar = new JMenuBar();
			dummyMenuBar.add(new JMenu("File"));
			viewport = new Viewport(fullWidth - PROPS_PANEL_WIDTH, fullHeight -
(int)(dummyMenuBar.getPreferredSize().getHeight()));
		}

		/**
		 * Returns the generated viewport
		 * @return the viewport that was generated by createViewport()
		 */
		public Viewport getViewport() {
			return viewport;
		}

		/**
		 * Class for dialog boxes which allow the user to save an image
		 * @author William Platt
		 *
		 */
		private class FileSaver extends JFileChooser {

			private String gifDescription = "GIF image (limited colour pallete but small file sizes)";
			private String pngDescription = "PNG image (low compression and large file sizes)";
			private String jpgDescription = "JPEG image (high compression more suited to photographs)";

			/**
			 * Constructor for the FileSaver class which sets the allowable file formats and sets gif as the default
			 */
			public FileSaver() {
```

```
                super();
                FileNameExtensionFilter gifFilter = new FileNameExtensionFilter(gifDescription, "gif");
                addChoosableFileFilter(gifFilter);
                addChoosableFileFilter(new FileNameExtensionFilter(pngDescription, "png"));
                addChoosableFileFilter(new FileNameExtensionFilter(jpgDescription, "jpg"));
                setAcceptAllFileFilterUsed(false);
                setFileFilter(gifFilter);
          }

          /**
           * Called when the user clicks 'Save' to open a dialog box if a file already exists with this name in
this directory
           */
          @Override
          public void approveSelection() {
                boolean approveFileName = false;
                File newFile = getFileWithExtension();
                System.out.println(newFile.getName());
                if (newFile.getName().matches(".*[\\\\/?<>:|\"%*].*")) { // If the filename contains any invalid
characters
                      JOptionPane.showMessageDialog(this, "The filename entered was invalid; the following
characters are not allowed: \\ / ? < > : | \" % *");
                } else if (newFile.exists()) { // If a file with this name and directory already exists, ask the
user if they would like to overwrite it
                      int overwriteReturn = JOptionPane.showConfirmDialog(this, "A file with this name already
exists, would you like to overwrite it?", "File already exists", JOptionPane.YES_NO_OPTION);
                      if (overwriteReturn == JOptionPane.YES_OPTION) { // If the user chose to overwrite the file
                            approveFileName = true;
                      }
                } else {
                      approveFileName = true;
                }
                if (approveFileName) {
                      try {
                            if (newFile.createNewFile()) { // Attempts to create the new file (with no contents)
and returns true if it is created successfully
                                  super.approveSelection(); // Allow the dialog to close and the showSaveDialog()
method to return APPROVE_OPTION
                            }
```

```
                        } catch (IOException e) { // Display a dialog if the empty file couldn't be created
                              JOptionPane.showMessageDialog(this, "There was a problem saving the image; you might
not have permission to save to the directory you chose.");
                        }
                  }
            }

            /**
             * Returns a file (with the file extension included in its path) into which image data can be written
             * @return a file (with the file extension included in its path) into which image data can be written
             */
            public File getFileWithExtension() {
                  File newFile = getSelectedFile(); // Get the path of the new file including the name entered by
the user
                  System.out.println(newFile.getName());
                  System.out.println();
                  String fileExtension = extensionFromFileName(newFile.getName());
                  if ((!fileExtension.equals("png")) && (!fileExtension.equals("jpg")) &&
(!fileExtension.equals("gif"))) { // If the user did not add a valid file extension to the file's name
                        fileExtension = getFileFilter().getDescription(); // Get the description of the file format
that was selected from the drop down menu when the user clicked 'Save'
                        if (fileExtension.equals(gifDescription)) {
                              fileExtension = "gif";
                        } else if (fileExtension.equals(pngDescription)) {
                              fileExtension = "png";
                        } else {
                              fileExtension = "jpg";
                        }
                        newFile = new File(newFile.getPath() + "." + fileExtension); // Add the file extension of
the selected file format to the end of the file name
                  }
                  return newFile;
            }

            /**
             * Returns the string of characters after the last "." in fileName in lower case; if there is no "." in
fileName then "" will be returned
             * @param fileName the name or path of a file
```

```
             * @return the file extension of the file name (characters after the last "." in lower case) or an empty
string
             */
            public String extensionFromFileName(String fileName) {
                  String[] substrings = fileName.split("\\."); // The split parameter string represents a regular
expression, so the . must be escaped
                  if (substrings.length > 0) {
                        return substrings[substrings.length - 1].toLowerCase(); // Return everything after the last
. (converted to lower case)
                  } else {
                        return "";
                  }
            }

      }

      /**
       * Generates a menu bar which can then be retrieved using getMenuBar()
       */
      public void createMenuBar() {
            menuBar = new JMenuBar();
            menuBar.add(getFileMenu());
            menuBar.add(getAddMenu());
            addTargetWorldMenus(menuBar);
            menuBar.add(getViewMenu());
      }

      /**
       * Generates and returns the File menu which is to be added to the menu bar
       * @return the File menu
       */
      private JMenu getFileMenu() {
            JMenu fileMenu = new JMenu("File");
            JMenuItem saveImage = new JMenuItem("Save as image", KeyEvent.VK_S); // Allows the user to click 'S' as
a shortcut when the file menu is open (the 'S' in save is also underlined to indicate this)
            saveImage.addActionListener(new ActionListener() { // Use adapter class rather than making a new class

                  /**
                   * Called when the saveImage menu item is clicked (or selected using the keyboard)
```

```
                     * @param event contains details of the action that triggered this event
                     */
                    public void actionPerformed(ActionEvent event) {
                          FileSaver fileSaver = new FileSaver();
                          int saveFileReturn = fileSaver.showSaveDialog(SwingUtilities.windowForComponent(viewport));
                          if (saveFileReturn == FileSaver.APPROVE_OPTION) { // If the user clicked save rather than
cancel or closing the dialog box
                                File newFile = fileSaver.getFileWithExtension();
                                viewport.saveImage(newFile, fileSaver.extensionFromFileName(newFile.getName())); //
Write the image file to newFile
                          }
                    }

              });
              fileMenu.add(saveImage);
              return fileMenu;
        }

        /**
         * Generates and returns the Add menu which is to be added to the menu bar
         * @return the Add menu
         */
        private JMenu getAddMenu() {
              JMenu addMenu = new JMenu("Add");
              JMenuItem addRayBox = new JMenuItem("Ray box", KeyEvent.VK_R); // Shortcut is 'R'
              ActionListener addRayBoxListener = new ActionListener() { // Use an adapter class

                    /**
                     * Called when the addRayBox menu item is clicked (or selected using the keyboard)
                     * @param event contains details of the action that triggered this event
                     */
                    public void actionPerformed(ActionEvent event) {
                          RayBox newRayBox = new RayBox();
                          viewport.addRayBox(newRayBox);
                    }

              };
              addRayBox.addActionListener(addRayBoxListener);
              addMenu.add(addRayBox);
```

```
            JMenuItem addMaterial = new JMenuItem("Custom material", KeyEvent.VK_C); // Shortcut is 'C'
            addMaterial.addActionListener(new ActionListener() { // Use an adapter class

                  /**
                   * Called when the addMaterial menu item is clicked (or selected using the keyboard)
                   * @param event contains details of the action that triggered this event
                   */
                  public void actionPerformed(ActionEvent event) {
                        JTextField materialNameField = new JTextField();
                        MaterialNameActionListener matNameActionListener = new MaterialNameActionListener();
                        materialNameField.addActionListener(matNameActionListener);
                        materialNameField.addFocusListener(new TextFieldFocusListener(matNameActionListener)); //
The TextFieldFocusListener requires the ActionListener so that it can trigger an ActionEvent when a FocusEvent
occurs
                        JTextField refractiveIndexField = new JTextField("1.0"); // Pre-populate the field with a
value of 1.0
                        RefractiveIndexActionListener refIndexActionListener = new
RefractiveIndexActionListener("1.0");
                        refractiveIndexField.addActionListener(refIndexActionListener);
                        refractiveIndexField.addFocusListener(new TextFieldFocusListener(refIndexActionListener));
                        JComponent[] inputs = new JComponent[] {
                                    new JLabel("Material name:"),
                                    materialNameField,
                                    new JLabel("Refractive index:"),
                                    refractiveIndexField
                        };
                        int choice = JOptionPane.showOptionDialog(SwingUtilities.windowForComponent(viewport),
inputs, "Add a custom material", JOptionPane.CANCEL_OPTION, JOptionPane.PLAIN_MESSAGE, null, null, null); //
PLAIN_MESSAGE allows me to pass a message rather than use a standard message; in this case the message is a list of
components (JLabels and JTextFields)
                        if (choice == JOptionPane.OK_OPTION) { // Only add the material if the user clicked OK
                              String name = materialNameField.getText().trim(); // Get the material name entered and
remove whitespace from the beginning and end of the material name
                              if (name.equals("")) { // Strings are objects, so == won't be true unless both sides
refer to the same memory location
                                    name = "Custom material"; // Replace a blank name with "Custom material"
                              }
                              viewport.addMaterial(name, Double.parseDouble(refractiveIndexField.getText())); // Add
the material to the list; the refractive index field's listeners will have ensured that its value is a number
```

```
                        }
                    }

            });
            addMenu.add(addMaterial);
            return addMenu;
      }

      /**
       * Generates the Target and World menus and adds them to the menu bar. This method is more efficient than
generating the menus separately because the target's Material menu and the World menu are similar
       * @param menuBar the menu bar which the menus are to be added to
       */
      private void addTargetWorldMenus(JMenuBar menuBar) {
            JMenu targetMenu = new JMenu("Target");
            JMenu targetShapeMenu = new JMenu("Shape"); // This will be a submenu of the Target menu; it will
automatically expand when the cursor is over it or its shortcut is used with the Target menu open
            targetShapeMenu.setMnemonic(KeyEvent.VK_S); // Set the shortcut to 'S' and underline the 'S' in Shape;
this can't be done in the constructor like it can be for JMenuItems
            ButtonGroup shapeRadioButtons = new ButtonGroup(); // Radio buttons must be grouped so that no more than
one can be selected at a time; the selected radio button is then automatically handled when one is clicked
            Mesh.Primitive[] shapes = Mesh.Primitive.values(); // Store a list of all primitive meshes
            ActionListener shapeActionListener = new ActionListener() { // Use an adapter class

                  /**
                   * Called when a radio button/menu item in the Shape menu is clicked (or selected using the
keyboard)
                   * @param event contains details of the action that triggered this event
                   */
                  public void actionPerformed(ActionEvent event) {
                        JRadioButtonMenuItem source = (JRadioButtonMenuItem)(event.getSource()); // Get the menu
item selected and typecast it to a JRadioButtonMenuItem
                        viewport.setTargetShape(Mesh.primitiveFromStr(source.getText())); // Convert the text of the
selected menu item into a mesh primitive which the viewport will set the target's shape to
                  }

            };
            for (Mesh.Primitive shape : shapes) { // Set shape to each element of shapes in turn
                  String checkBoxLabel = shape.toString(); // Get the corresponding string for the mesh primitive
```

```
                JRadioButtonMenuItem shapeMenuItem = new JRadioButtonMenuItem(checkBoxLabel); // Set the text of
the menu item to the string for the mesh primitive
                if (viewport.getTargetShape() == checkBoxLabel) { // Select the radio button matching the target's
current shape
                      shapeMenuItem.setSelected(true);
                }
                shapeMenuItem.addActionListener(shapeActionListener); // Add the ActionListener to each radio
button
                shapeRadioButtons.add(shapeMenuItem);
                targetShapeMenu.add(shapeMenuItem);
          }
          targetMenu.add(targetShapeMenu);
          JMenu targetMaterialMenu = new JMenu("Material");
          targetMaterialMenu.setMnemonic(KeyEvent.VK_M); // Shortcut is 'M'
          ButtonGroup targetMaterialRadioButtons = new ButtonGroup();
          int numOfMaterials = Viewport.getNumOfMaterials();
          String[] materials = Viewport.getMaterials();
          double[] refractiveIndices = Viewport.getRefractiveIndices();
          ActionListener targetMaterialListener = new ActionListener() { // Adapter class

                /**
                 * Called when a radio button/menu item in the Material menu is clicked (or selected using the
keyboard)
                 * @param event contains details of the action that triggered this event
                 */
                public void actionPerformed(ActionEvent event) {
                      JRadioButtonMenuItem source = (JRadioButtonMenuItem)(event.getSource());
                      viewport.setTargetMaterial(Integer.parseInt(source.getActionCommand())); // Each menu item
has a string which will contain the index of the material it corresponds to
                }

          };

          JMenu worldMenu = new JMenu("World");
          ButtonGroup worldMaterialRadioButtons = new ButtonGroup();
          ActionListener worldMaterialListener = new ActionListener() { // Adapter class

                /**
```

```
                 * Called when a radio button/menu item in the World menu is clicked (or selected using the
keyboard)
                 * @param event contains details of the action that triggered this event
                 */
                public void actionPerformed(ActionEvent event) {
                      JRadioButtonMenuItem source = (JRadioButtonMenuItem)(event.getSource());
                      viewport.setWorldMaterial(Integer.parseInt(source.getActionCommand())); // Each menu item
has a string with the index of the material just as in the target's material menu
                }

          };

          for (int i = 0; i < numOfMaterials; i++) { // For each material generate a target and world menu item
                String label = materials[i] + " (" + refractiveIndices[i] + ")"; // Add the refractive index in
parentheses after the material's name
                String actionCommand = Integer.toString(i); // Associate each menu item with the material's index

                JRadioButtonMenuItem targetMenuItem = new JRadioButtonMenuItem(label);
                targetMenuItem.setActionCommand(actionCommand);
                if (viewport.getTargetMaterial() == i) {
                      targetMenuItem.setSelected(true); // Select the menu item matching the target's current
material
                }
                targetMenuItem.addActionListener(targetMaterialListener);
                targetMaterialRadioButtons.add(targetMenuItem);
                targetMaterialMenu.add(targetMenuItem);

                JRadioButtonMenuItem worldMenuItem = new JRadioButtonMenuItem(label);
                worldMenuItem.setActionCommand(actionCommand);
                if (Viewport.getWorldMaterial() == i) {
                      worldMenuItem.setSelected(true); // Select the menu item matching the world's current
material
                }
                worldMenuItem.addActionListener(worldMaterialListener);
                worldMaterialRadioButtons.add(worldMenuItem);
                worldMenu.add(worldMenuItem);
          }
          targetMenu.add(targetMaterialMenu);
          menuBar.add(targetMenu);
```

```
            menuBar.add(worldMenu);
      }

      /**
       * Generates and returns the View menu which is to be part of the menu bar
       * @return the View menu
       */
      private JMenu getViewMenu() {
            JMenu viewMenu = new JMenu("View");
            JCheckBoxMenuItem perspectiveCheckBox = new JCheckBoxMenuItem("Perspective"); // The checkbox is
automatically toggled when selected
            perspectiveCheckBox.setMnemonic(KeyEvent.VK_P); // Shortcut is 'P'
            if (!viewport.isOrthographic()) {
                  perspectiveCheckBox.setSelected(true); // Check the checkbox if perspective projection is being
used
            }
            perspectiveCheckBox.addActionListener(new ActionListener() { // Adapter class

                  /**
                   * Called when the perspective checkbox/menu item is clicked (or selected using the keyboard)
                   * @param event contains details of the action that triggered this event
                   */
                  public void actionPerformed(ActionEvent event) {
                        viewport.toggleOrthographic();
                  }

            });
            viewMenu.add(perspectiveCheckBox);
            JCheckBoxMenuItem radiansCheckBox = new JCheckBoxMenuItem("Angles in radians");
            radiansCheckBox.setMnemonic(KeyEvent.VK_A); // Shortcut is 'A'
            if (!viewport.areAnglesInDegrees()) {
                  radiansCheckBox.setSelected(true); // Check the checkbox if angles are being displayed in radians
            }
            radiansCheckBox.addActionListener(new ActionListener() {

                  /**
                   * Called when the angles in radians checkbox/menu item is clicked (or selected using the
keyboard)
                   * @param event contains details of the action that triggered this event
```

```
                 */
                public void actionPerformed(ActionEvent event) {
                      viewport.toggleAngleUnits();
                }

          });
          viewMenu.add(radiansCheckBox);

          viewMenu.add(getCameraPositionsMenu()); // Add a submenu
          return viewMenu;
    }

    /**
     * Generates and returns the menu of preset camera positions/orientations which is to be a submenu of the View
menu
     * @return the menu of camera positions/orientations
     */
    private JMenu getCameraPositionsMenu() {
          JMenu cameraMenu = new JMenu("Camera position"); // Submenu of View menu
          cameraMenu.setMnemonic(KeyEvent.VK_C); // Shortcut is 'C'
          JMenuItem frontView = new JMenuItem("Front (shortcut 1)", KeyEvent.VK_F); // Menu item of Camera
position submenu; shortcut is 'F' within the submenu - 1 is the shortcut from the viewport
          frontView.addActionListener(new ActionListener() { // Adapter class

                /**
                 * Called when the front menu item is clicked (or selected using the keyboard)
                 * @param event contains details of the action that triggered this event
                 */
                public void actionPerformed(ActionEvent event) {
                      viewport.setViewFront();
                }

          });
          cameraMenu.add(frontView);
          JMenuItem backView = new JMenuItem("Back (shortcut 2)", KeyEvent.VK_B); // Shortcut from submenu is 'B'
          backView.addActionListener(new ActionListener() { // Adapter class

                /**
                 * Called when the back menu item is clicked (or selected using the keyboard)
```

```
         * @param event contains details of the action that triggered this event
         */
        public void actionPerformed(ActionEvent event) {
              viewport.setViewBack();
        }

});
cameraMenu.add(backView);
JMenuItem leftView = new JMenuItem("Left (shortcut 3)", KeyEvent.VK_L); // Shortcut is 'L'
leftView.addActionListener(new ActionListener() { // Adapter class

        /**
         * Called when the left menu item is clicked (or selected using the keyboard)
         * @param event contains details of the action that triggered this event
         */
        public void actionPerformed(ActionEvent event) {
              viewport.setViewLeft();
        }

});
cameraMenu.add(leftView);
JMenuItem rightView = new JMenuItem("Right (shortcut 4)", KeyEvent.VK_R); // Shortcut is 'R'
rightView.addActionListener(new ActionListener() { // Adapter class

        /**
         * Called when the right menu item is clicked (or selected using the keyboard)
         * @param event contains details of the action that triggered this event
         */
        public void actionPerformed(ActionEvent event) {
              viewport.setViewRight();
        }

});
cameraMenu.add(rightView);
JMenuItem topView = new JMenuItem("Top (shortcut 5)", KeyEvent.VK_T); // Shortcut is 'T'
topView.addActionListener(new ActionListener() { // Adapter class

        /**
         * Called when the top menu item is clicked (or selected using the keyboard)
```

```
                 * @param event contains details of the action that triggered this event
                 */
                public void actionPerformed(ActionEvent event) {
                      viewport.setViewTop();
                }

          });
          cameraMenu.add(topView);
          JMenuItem bottomView = new JMenuItem("Bottom (shortcut 6)", KeyEvent.VK_B); // Shortcut is 'B'
          bottomView.addActionListener(new ActionListener() { // Adapter class

                /**
                 * Called when the back menu item is clicked (or selected using the keyboard)
                 * @param event contains details of the action that triggered this event
                 */
                public void actionPerformed(ActionEvent event) {
                      viewport.setViewBottom();
                }

          });
          cameraMenu.add(bottomView);
          return cameraMenu;
    }

    /**
     * Returns the last menu bar that was generated by createMenuBar()
     * @return the most recent menu bar that was generated by a call to createMenuBar()
     */
    public JMenuBar getMenuBar() {
          return menuBar;
    }

    private final static int SLIDER_MAX = 150; // The maximum value of each slider; the minimum value is -
SLIDER_MAX. Slider values are not in standard angular units and must be converted to radians for processing

    /**
     * Generates a properties panel that can be retrieved by getPropertiesPanel()
     * @param rayBox the selected ray box
     */
```

```
	public void buildPropertiesPanel(RayBox rayBox) {
		propertiesPanel = new JPanel();
		propertiesPanel.setBackground(new Color(220, 220, 220)); // Set the background to light grey
		propertiesPanel.setLayout(new FlowLayout(FlowLayout.LEFT, 5, 5));
		propertiesPanel.setPreferredSize(new Dimension(PROPS_PANEL_WIDTH, fullHeight -
(int)(menuBar.getPreferredSize().getHeight())));

		if (rayBox != null) { // If a ray box is selected (otherwise the properties panel is empty
			JPanel labelPanel = getLabelPanel(rayBox);
			propertiesPanel.add(labelPanel);

			JPanel anglesPanel = getDisplayAnglesPanel(rayBox);
			propertiesPanel.add(anglesPanel);

			JPanel beamThicknessPanel = getBeamThicknessPanel(rayBox);
			propertiesPanel.add(beamThicknessPanel);

			// Add a horizontal line to separate the upper and lower sections of the properties panel
			JSeparator horizontalLine = new JSeparator(SwingConstants.HORIZONTAL);
			horizontalLine.setPreferredSize(new Dimension(PROPS_PANEL_WIDTH - 10, 15));
			propertiesPanel.add(horizontalLine);

			// Calculate the height in pixels of the remaining section of the properties panel
			int heightRemaining = (int)(propertiesPanel.getPreferredSize().getHeight() -
labelPanel.getPreferredSize().getHeight() - anglesPanel.getPreferredSize().getHeight() -
beamThicknessPanel.getPreferredSize().getHeight() - horizontalLine.getPreferredSize().getHeight() - 40);

			JPanel globalPanel = new JPanel();
			TitledBorder title = BorderFactory.createTitledBorder("Global position/orientation"); // Give
globalPanel a border with a title
			globalPanel.setBorder(title);
			globalPanel.setOpaque(false);
			globalPanel.setLayout(new FlowLayout(3, 3, FlowLayout.LEFT));
			int halfHeight = heightRemaining / 2;
			globalPanel.setPreferredSize(new Dimension(PROPS_PANEL_WIDTH - 10, halfHeight));

			Vector origin = rayBox.getOrigin();
			double globalHeading = (Math.PI / 2) - Math2.atan(origin.getElement(2), origin.getElement(0)); //
Calculate the global heading of the selected ray box in radians
```

```
                if (globalHeading > Math.PI) {
                      globalHeading = globalHeading - (2 * Math.PI); // Wrap the heading to the range -pi to pi
                }

                JPanel globalHeadingPanel = new JPanel();
                globalHeadingPanel.setPreferredSize(new Dimension(100, (int)(halfHeight - 30)));
                globalHeadingPanel.setOpaque(false);
                globalHeadingPanel.setLayout(new FlowLayout(20, 20, FlowLayout.LEFT));
                JLabel headingLabel = new JLabel("Heading");
                headingLabel.setPreferredSize(new Dimension(120, 20));
                globalHeadingPanel.add(headingLabel);

                JSlider globalHeadingSlider = getHeadingSlider(halfHeight, globalHeading);
                globalHeadingSlider.setName("globalHeading"); // The SliderListener will use this name to identify
the type of slider
                globalHeadingPanel.add(globalHeadingSlider);
                globalPanel.add(globalHeadingPanel);

                double globalPitch = Math.asin(origin.getElement(1) / origin.modulus());

                JPanel globalPitchPanel = new JPanel();
                globalPitchPanel.setPreferredSize(new Dimension(100, (int)(halfHeight - 30)));
                globalPitchPanel.setOpaque(false);
                globalPitchPanel.setLayout(new FlowLayout(20, 20, FlowLayout.LEFT));
                JLabel pitchLabel = new JLabel("  Pitch");
                pitchLabel.setPreferredSize(new Dimension(120, 20));
                globalPitchPanel.add(pitchLabel);

                JSlider globalPitchSlider = getPitchSlider(halfHeight, globalPitch);
                globalPitchSlider.setName("globalPitch");
                globalPitchPanel.add(globalPitchSlider);
                globalPanel.add(globalPitchPanel);
                propertiesPanel.add(globalPanel);

                JPanel localPanel = new JPanel();
                title = BorderFactory.createTitledBorder("Local Orientation");
                localPanel.setBorder(title);
                localPanel.setOpaque(false);
                localPanel.setLayout(new FlowLayout(3, 3, FlowLayout.LEFT));
```

```
                localPanel.setPreferredSize(new Dimension(PROPS_PANEL_WIDTH - 10, halfHeight));

                EulerTriple orientation = rayBox.getOrientation().eulerObToUp();

                JPanel localHeadingPanel = new JPanel();
                localHeadingPanel.setPreferredSize(new Dimension(100, (int)(halfHeight - 30)));
                localHeadingPanel.setOpaque(false);
                localHeadingPanel.setLayout(new FlowLayout(20, 20, FlowLayout.LEFT));
                JLabel localHeadingLabel = new JLabel("Heading");
                localHeadingLabel.setPreferredSize(new Dimension(120, 20));
                localHeadingPanel.add(localHeadingLabel);

                JSlider localHeadingSlider = getHeadingSlider(halfHeight, orientation.getHeading()); // The
heading of orientation is the local heading
                localHeadingSlider.setName("localHeading");
                localHeadingPanel.add(localHeadingSlider);
                localPanel.add(localHeadingPanel);

                JPanel localPitchPanel = new JPanel();
                localPitchPanel.setPreferredSize(globalHeadingPanel.getPreferredSize());
                localPitchPanel.setOpaque(false);
                localPitchPanel.setLayout(new FlowLayout(20, 20, FlowLayout.LEFT));
                JLabel localPitchLabel = new JLabel("  Pitch");
                localPitchLabel.setPreferredSize(new Dimension(120, 20));
                localPitchPanel.add(localPitchLabel);

                JSlider localPitchSlider = getPitchSlider(halfHeight, -orientation.getPitch()); // It is more
intuitive for the user if pitch represents the angle above the horizontal rather than below it as it does in the
code
                localPitchSlider.setName("localHeading");
                localPitchPanel.add(localPitchSlider);
                localPanel.add(localPitchPanel);

                if (rayBox.isLocalPitchInverted()) {
                      localPitchSlider.setName("localPitchInv");
                } else { // This is the usual case. Local pitch is inverted when the local pitch goes beyond -pi/2
or pi/2 so that the ray box is always 'the right way up' and pitch is always the angle above the horizontal
                      localPitchSlider.setName("localPitch");
                }
```

```
                localPitchSlider.addChangeListener(new SliderListener(-orientation.getPitch())); // The
SliderListener needs to know the original angle so that it can calculate the change in angle when the slider is
dragged and therefore the transformation to apply to the ray box
                localHeadingSlider.addChangeListener(new SliderListener(orientation.getHeading()));
                globalHeadingSlider.addChangeListener(new SliderListener(globalHeading, localHeadingSlider));
                globalPitchSlider.addChangeListener(new SliderListener(globalPitch, localPitchSlider,
localHeadingSlider, rayBox));
                propertiesPanel.add(localPanel);
            }
      }

      /**
       * Generates and returns the panel for setting the label of the selected ray box or deleting the selected ray
box
       * @param rayBox the selected ray box
       * @return the panel containing a label, a text field and a delete button
       */
      private JPanel getLabelPanel(RayBox rayBox) {
            JPanel labelPanel = new JPanel();
            labelPanel.setPreferredSize(new Dimension(PROPS_PANEL_WIDTH, 30)); // Set width to that of the
properties panel and height to 30 pixels
            labelPanel.setOpaque(false); // No background colour
            labelPanel.setLayout(new FlowLayout(3, 3, FlowLayout.LEFT));
            labelPanel.add(new JLabel("Label:"));
            JTextField labelInput = new JTextField(rayBox.getLabel(), 10); // Set the text contents to the selected
ray box's label
            labelInput.setMargin(new Insets(2, 2, 2, 2));
            ActionListener labelActionListener = new ActionListener() { // Adapter class

                  /**
                   * Updates the label of the selected ray box when a new label is entered in the text field
                   * @param event contains details of the action that triggered this event
                   */
                  public void actionPerformed(ActionEvent event) {
                        JTextField source = (JTextField)(event.getSource());
                        viewport.updateLabel(source.getText());
                  }
```

```
            };
            labelInput.addActionListener(labelActionListener);
            labelInput.addFocusListener(new TextFieldFocusListener(labelActionListener)); // The
TextFieldFocusListener requires the field's ActionListener so that a FocusEvent can trigger an ActionEvent
            labelPanel.add(labelInput);
            JButton delete = new JButton("Delete");
            delete.setMargin(new Insets(1, 3, 1, 3));
            ActionListener deleteActionListener = new ActionListener() { // Adapter class

                  /**
                   * Called when the Delete button is clicked
                   * @param event contains details of the action that triggered this event
                   */
                  public void actionPerformed(ActionEvent event) {
                        viewport.removeRayBox();
                  }

            };
            delete.addActionListener(deleteActionListener);
            labelPanel.add(delete);
            return labelPanel;
      }

      /**
       * Generates and returns the panel containing the checkbox indicating whether angles are displayed for the
selected ray box
       * @param rayBox the selected ray box
       * @return the display angles panel
       */
      private JPanel getDisplayAnglesPanel(RayBox rayBox) {
            JPanel anglesPanel = new JPanel();
            anglesPanel.setPreferredSize(new Dimension(PROPS_PANEL_WIDTH, 30));
            anglesPanel.setOpaque(false);
            anglesPanel.setLayout(new FlowLayout(3, 3, FlowLayout.LEFT));
            JCheckBox anglesCheckBox = new JCheckBox("Display angles");
            anglesCheckBox.setSelected(rayBox.getAnglesVisible()); // Check the Display angles checkbox if angles
are being displayed for the selected ray box
            ActionListener checkBoxListener = new ActionListener() { // Adapter class
```

```
                /**
                 * Called when the Display angles checkbox is toggled
                 * @param event contains details of the action that triggered this event
                 */
                public void actionPerformed(ActionEvent event) {
                      viewport.toggleShowAngles();
                }

            };
            anglesCheckBox.addActionListener(checkBoxListener);
            anglesCheckBox.setOpaque(false);
            anglesPanel.add(anglesCheckBox);
            return anglesPanel;
      }

      /**
       * Generates and returns the panel containing a beam thickness input field, label and increment and decrement
buttons
       * @param rayBox the selected ray box
       * @return the panel containing components relating to the beam thickness setting of the selected ray box
       */
      private JPanel getBeamThicknessPanel(RayBox rayBox) {
            JPanel beamThicknessPanel = new JPanel();
            beamThicknessPanel.setPreferredSize(new Dimension(PROPS_PANEL_WIDTH, 30));
            beamThicknessPanel.setOpaque(false);
            beamThicknessPanel.setLayout(new FlowLayout(0, 0, FlowLayout.LEFT));

            JLabel beamThicknessLabel = new JLabel("Beam thickness: ");
            beamThicknessPanel.add(beamThicknessLabel);

            JTextField inputField = new JTextField(String.valueOf(rayBox.getBeamThickness()), 3); // Set the
contents of the Beam thickness field to the current beam thickness for the beam corresponding to the selected ray
box
            inputField.setMargin(new Insets(2, 2, 2, 2));
            ActionListener actionListener = new
BeamThicknessActionListener(String.valueOf(rayBox.getBeamThickness())); // Pass the starting value of the Beam
thickness field
            inputField.addActionListener(actionListener);
            inputField.addFocusListener(new TextFieldFocusListener(actionListener));
```

```
            beamThicknessPanel.add(inputField);

            JPanel incDecButtons = new JPanel();
            incDecButtons.setLayout(new GridLayout(2, 1));
            incDecButtons.setPreferredSize(new Dimension(20, 30));

            JButton increment = new JButton("+");
            increment.setMargin(new Insets(0, 0, 0, 0));
            increment.setActionCommand("Increment");
            increment.addActionListener(new IncDecActionListener(inputField)); // Pass the Beam thickness field so
that its value can be incremented by 1 when the + button is clicked

            JButton decrement = new JButton("-");
            decrement.setMargin(new Insets(0, 0, 0, 0));
            decrement.setActionCommand("Decrement");
            decrement.addActionListener(new IncDecActionListener(inputField));

            incDecButtons.add(increment);
            incDecButtons.add(decrement);
            beamThicknessPanel.add(incDecButtons);
            return beamThicknessPanel;
      }

      /**
       * Generates and returns a labelled vertical slider between -SLIDER_MAX (representing -pi radians) and
SLIDER_MAX (representing pi radians) with the initial value as heading converted from radians. Labels are in the
angular units specified by the user (degrees or radians) and ticks indicate increments on the slider
       * @param parentHeight the height in pixels of the component that is to contain the slider
       * @param heading the global or local heading angle of the selected ray box in radians
       * @return the slider representing a local or global heading of the selected ray box
       */
      private JSlider getHeadingSlider(int parentHeight, double heading) {
            JSlider headingSlider = new JSlider(JSlider.VERTICAL, -SLIDER_MAX, SLIDER_MAX,
(int)(Math.round(SLIDER_MAX * heading / Math.PI))); // Create a vertical slider between -SLIDER_MAX (representing -
pi radians) and SLIDER_MAX (pi radians) currently set to the value that represents heading radians
            headingSlider.setOpaque(false);
            headingSlider.setPreferredSize(new Dimension(100, (int)(parentHeight - 50)));
            headingSlider.setMajorTickSpacing(SLIDER_MAX / 2); // Use longer lines to mark -pi, 0 and pi
```

```
            headingSlider.setMinorTickSpacing(SLIDER_MAX / 10); // Use shorter lines to mark -pi, -9pi/10, -8pi/10,
... 9pi/10, pi; major ticks replace minor ticks where they overlap
            headingSlider.setPaintTicks(true);
            Hashtable labelTable = new Hashtable(); // Use custom labels for points on the slider since generated
labels would be in non-standard units
            // Specify slider values to display labels at and give the label for each
            if (viewport.areAnglesInDegrees()) { // Use labels in degrees if angles are to be displayed in degrees
                  labelTable.put(new Integer(-SLIDER_MAX), new JLabel("-180\u00B0")); // \u00B0 represents the
degree symbol
                  labelTable.put(new Integer(-SLIDER_MAX / 2), new JLabel("-90\u00B0"));
                  labelTable.put(new Integer(0), new JLabel("0\u00B0"));
                  labelTable.put(new Integer(SLIDER_MAX / 2), new JLabel("90\u00B0"));
                  labelTable.put(new Integer(SLIDER_MAX), new JLabel("180\u00B0"));
                  headingSlider.setLabelTable(labelTable);
            } else {
                  labelTable.put(new Integer(-SLIDER_MAX), new JLabel("-\u03C0")); // \u030C0 represents the pi
symbol
                  labelTable.put(new Integer(-SLIDER_MAX / 2), new JLabel("-\u03C0/2")); // -pi/2
                  labelTable.put(new Integer(0), new JLabel("0"));
                  labelTable.put(new Integer(SLIDER_MAX / 2), new JLabel("\u03C0/2")); // pi/2
                  labelTable.put(new Integer(SLIDER_MAX), new JLabel("\u03C0")); // pi
                  headingSlider.setLabelTable(labelTable);
            }
            headingSlider.setPaintLabels(true);
            return headingSlider;
      }

      /**
       * Generates and returns a labelled vertical slider between -SLIDER_MAX (representing -pi/2 radians) and
SLIDER_MAX (representing pi/2 radians) with the initial value as pitch converted from radians. Labels are in the
angular units specified by the user (degrees or radians) and ticks indicate increments on the slider
       * @param parentHeight the height in pixels of the component that is to contain the slider
       * @param pitch the global or local pitch angle of the selected ray box in radians
       * @return the slider representing a local or global pitch of the selected ray box
       */
      private JSlider getPitchSlider(int parentHeight, double pitch) {
            JSlider pitchSlider = new JSlider(JSlider.VERTICAL, -SLIDER_MAX, SLIDER_MAX, (int)(Math.round(2 *
SLIDER_MAX * pitch / Math.PI)));
            pitchSlider.setOpaque(false);
```

```
            pitchSlider.setPreferredSize(new Dimension(100, (int)(parentHeight - 50)));
            pitchSlider.setMajorTickSpacing(SLIDER_MAX / 2);
            pitchSlider.setMinorTickSpacing(SLIDER_MAX / 10);
            pitchSlider.setPaintTicks(true);
            Hashtable labelTable = new Hashtable();
            if (viewport.areAnglesInDegrees()) {
                  labelTable.put(new Integer(-SLIDER_MAX), new JLabel("-90\u00B0")); // \u00B0 represents the degree
symbol
                  labelTable.put(new Integer(-SLIDER_MAX / 2), new JLabel("-45\u00B0"));
                  labelTable.put(new Integer(0), new JLabel("0\u00B0"));
                  labelTable.put(new Integer(SLIDER_MAX / 2), new JLabel("45\u00B0"));
                  labelTable.put(new Integer(SLIDER_MAX), new JLabel("90\u00B0"));
                  pitchSlider.setLabelTable(labelTable);
            } else {
                  labelTable.put(new Integer(-SLIDER_MAX), new JLabel("-\u03C0/2")); // \u03C0 represents the pi
symbol
                  labelTable.put(new Integer(-SLIDER_MAX / 2), new JLabel("-\u03C0/4"));
                  labelTable.put(new Integer(0), new JLabel("0"));
                  labelTable.put(new Integer(SLIDER_MAX / 2), new JLabel("\u03C0/4"));
                  labelTable.put(new Integer(SLIDER_MAX), new JLabel("\u03C0/2"));
                  pitchSlider.setLabelTable(labelTable);
            }
            pitchSlider.setPaintLabels(true);
            return pitchSlider;
      }

      /**
       * Returns the properties panel generated by the last call to buildPropertiesPanel
       * @return the properties panel generated by the last call to buildPropertiesPanel
       */
      public JPanel getPropertiesPanel() {
            return propertiesPanel;
      }

      /**
       * Class for ActionListeners of buttons that increment or decrement the value in a JTextField by 1
       * @author William Platt
       *
       */
```

```
	private class IncDecActionListener implements ActionListener {

		private JTextField textField;

		/**
		 * Constructor for the IncDecActionListener class
		 * @param textField the text field that an ActionEvent increments or decrements the value of
		 */
		public IncDecActionListener(JTextField textField) {
			this.textField = textField;
		}

		/**
		 * Called when the button is clicked and increments or decrements the value in textField by 1
		 * @param event contains details of the action that triggered this event
		 */
		public void actionPerformed(ActionEvent event) {
			JButton button = (JButton)(event.getSource());
			if (button.getActionCommand().equals("Increment")) {
				textField.setText(String.valueOf(Integer.parseInt(textField.getText()) + 1)); // Convert the
current entry to from a String to an integer, add 1 and convert back to a String to set as the contents of the text
field
			} else if (button.getActionCommand().equals("Decrement")) {
				textField.setText(String.valueOf(Integer.parseInt(textField.getText()) - 1));
			}
			textField.requestFocusInWindow(); // Give the text field focus so that the FocusListener triggers
an ActionEvent which will validate the new value and also so that it is easier for the user to type a new value
		}

	}

	/**
	 * Class for FocusListeners of text fields that need to trigger an ActionEvent when they lose or gain focus
	 * @author William Platt
	 *
	 */
	private class TextFieldFocusListener implements FocusListener {

		private ActionListener actionListener;
```

```
		/**
		 * Constructor for the TextFieldFocusListener class
		 * @param actionListener the ActionListener of the text field
		 */
		public TextFieldFocusListener(ActionListener actionListener) {
			super();
			this.actionListener = actionListener;
		}

		/**
		 * Called when the text field gains focus to trigger an ActionEvent
		 * @param event contains details of the action that triggered this event
		 */
		public void focusGained(FocusEvent event) {
			focusLost(event);
		}

		/**
		 * Called when the text field loses focus to trigger an ActionEvent
		 * @param event contains details of the action that triggered this event
		 */
		public void focusLost(FocusEvent event) {
			actionListener.actionPerformed(new ActionEvent(event.getSource(), ActionEvent.ACTION_PERFORMED,
null) {});
		}

	}

	/**
	 * Class for ActionListeners of beam thickness text fields that validates the data entered, rounding to an
appropriate value or reverting to the last appropriate value
	 * @author William Platt
	 *
	 */
	private class BeamThicknessActionListener implements ActionListener {

		private String lastValid;
```

```
		/**
		 * Constructor for the BeamThicknessActionListener class
		 * @param initialString the initial string in the text field
		 */
		public BeamThicknessActionListener(String initialString) {
			lastValid = initialString;
		}

		/**
		 * Called when the text field gains or loses focus or the user hits enter with it in focus
		 * @param event contains details of the action that triggered this event
		 */
		public void actionPerformed(ActionEvent event) {
			JTextField source = (JTextField)(event.getSource());
			try {
				int newBeamThickness = (int)(Math.round(Double.parseDouble(source.getText()))); // Round to
the nearest integer
				// Values below 1 become 1 and those above 10 become 10
				if (newBeamThickness < 1) {
					newBeamThickness = 1;
				} else if (newBeamThickness > 10) {
					newBeamThickness = 10;
				}
				String newText = Integer.toString(newBeamThickness);
				source.setText(newText);
				viewport.updateBeamThickness(newBeamThickness);
				lastValid = newText; // Store the most recent valid entry so that an non-numerical entry can
be reverted to this
			} catch (Exception e) { // If the contents of the text field is non-numerical (causing
parseDouble() to throw an exception)
				source.setText(lastValid); // Revert to the last valid entry
			}
		}
	}

	/**
	 * Class for ActionListeners of material name input fields that restricts the name to 30 characters with no
whitespace at the beginning or end (blank string replaced with "Custom material")
	 * @author William Platt
```

```
	 *
	 */
	private class MaterialNameActionListener implements ActionListener {

		/**
		 * Called when the text field loses or gains focus or the user hits enter with it in focus
		 * @param event contains details of the action that triggered this event
		 */
		public void actionPerformed(ActionEvent event) {
			JTextField source = (JTextField)(event.getSource());
			if (!source.hasFocus()) { // Only validate and sanitise when the text field has just lost focus
				String name = source.getText().trim(); // Store name with whitespace at the beginning and
end removed
				if (name.length() > 30) {
					name = name.substring(0, 30).trim(); // Store the trimmed (the end has changed so
needs trimming again) first 30 characters of the trimmed name
				}
				if (name.equals("")) {
					source.setText("Custom material"); // Replace a blank name with "Custom material"
				} else {
					source.setText(name);
				}
			}
		}

	}

	/**
	 * Class for ActionListeners of refractive index input fields that restricts values between 1.0 and 100.0
inclusive and reverts non-numerical input to the last valid value
	 * @author William Platt
	 *
	 */
	private class RefractiveIndexActionListener implements ActionListener {

		private String lastValid;

		/**
		 * Constructor for the RefractiveIndexActionListener class
```

```
             * @param initialString the initial contents of the text field
             */
            public RefractiveIndexActionListener(String initialString) {
                  lastValid = initialString;
            }

            /**
             * Called when the text field loses or gains focus or the user hits enter with it in focus
             * @param event contains details of the action that triggered this event
             */
            public void actionPerformed(ActionEvent event) {
                  JTextField source = (JTextField)(event.getSource());
                  try {
                        double newRefractiveIndex = Double.parseDouble(source.getText());
                        if (newRefractiveIndex < 1.0) {
                              newRefractiveIndex = 1.0; // Inputs below 1 are changed to 1
                        } else if (newRefractiveIndex > 100) {
                              newRefractiveIndex = 100; // Inputs above 100 are changed to 100
                        }
                        String newText = Double.toString(newRefractiveIndex);
                        source.setText(newText);
                        lastValid = newText; // Store the most recent valid entry so that a non-numerical entry can
be reverted to this value
                  } catch (Exception e) { // If the input is not numerical (causing parseDouble() to throw an
exception)
                        source.setText(lastValid);
                  }
            }

      }

      /**
       * Class for ChangeListeners of sliders that apply the relevant transformation to the selected ray box while
the slider is being dragged
       * @author William Platt
       *
       */
      private class SliderListener implements ChangeListener {
```

```
            private double prevValue;
            private JSlider parallelSlider;
            private JSlider secondaryParallelSlider;
            private RayBox rayBox;

            /**
             * A constructor for SliderListeners that apply to sliders which don't have any affect on other sliders.
These are the local orientation sliders
             * @param previousValue the initial value of the slider converted to radians
             */
            public SliderListener(double previousValue) {
                  prevValue = previousValue;
            }

            /**
             * A constructor for a SliderListener that applies to a slider which affects one other slider (the
global heading slider affects the local heading)
             * @param previousValue the initial value of the slider this object is to be a listener for in radians
             * @param affectedSlider the slider which is affected by changes to the other slider
             */
            public SliderListener(double previousValue, JSlider affectedSlider) {
                  prevValue = previousValue;
                  parallelSlider = affectedSlider;
                  secondaryParallelSlider = null;
            }

            /**
             * A constructor for a SliderListener that applies to a slider which affects one other slider and
occasionally a second other slider (the global pitch slider directly and sometimes needs to change the local
heading)
             * @param previousValue the initial value of the slider this object is to be a listener for in radians
             * @param directlyAffectedSlider the slider which is always affected by changes to the slider being
listened for
             * @param indirectlyAffectedSlider the slider that is sometimes affected by changes to the slider being
listened for
             * @param rayBox the selected ray box
             */
            public SliderListener(double previousValue, JSlider directlyAffectedSlider, JSlider
indirectlyAffectedSlider, RayBox rayBox) {
```

```
                prevValue = previousValue;
                parallelSlider = directlyAffectedSlider;
                secondaryParallelSlider = indirectlyAffectedSlider;
                this.rayBox = rayBox;
            }

            /**
             * Called when the slider is dragged (including while it is still being dragged) or the value changed by
another slider that affects it
             * @param event contains details of the action that triggered this event
             */
            public void stateChanged(ChangeEvent event) {
                JSlider source = (JSlider)(event.getSource());
                int sliderValue = source.getValue();
                switch (source.getName()) {
                    case "globalHeading":
                        updateParallelSlider(sliderValue); // Change the slider that is affected by this one
(local heading slider)
                        viewport.globallyRotateRayBox(headingFromSliderVal(sliderValue), 0);
                        break;
                    case "globalPitch":
                        updateParallelSlider(sliderValue); // Change the slider that is affected by this one
(local pitch and maybe local heading)
                        viewport.globallyRotateRayBox(0, pitchFromSliderVal(sliderValue));
                        break;
                    case "localHeading":
                        if (source.getValueIsAdjusting()) { // If the slider was changed by the user
                            viewport.locallyRotateRayBox(headingFromSliderVal(sliderValue), 0); //
Calculates the change in heading in radians and stores the new heading, then the ray box is transformed by
locallyRotateRayBox()
                        } else { // The transformation is to be handled by the slider that affects this one
                            prevValue = sliderValue * Math.PI / SLIDER_MAX; // Store the most recent heading
in radians
                        }
                        break;
                    case "localPitch":
                        if (source.getValueIsAdjusting()) {
                            viewport.locallyRotateRayBox(0, -pitchFromSliderVal(sliderValue)); // The code
interprets pitch as the angle below the horizontal, so the pitch must be negated for use in transformations
```

```
                        } else { // Transformation handled by the slider that affects this one
                              prevValue = sliderValue * Math.PI / (2 * SLIDER_MAX); // Store the most recent
pitch in radians as pitchFromSliderVal() would normally do
                        }
                        break;
                  case "localPitchInv": // If the ray box is upside down (object y-axis pointing below the
horizontal); this is not obvious to the user because the ray box is a cube and no axes are shown, so we treat the
ray box as though it is still the right way up
                        if (source.getValueIsAdjusting()) {
                              viewport.locallyRotateRayBox(0, pitchFromSliderVal(sliderValue)); // Pitch needs
to be negated twice, so no changes needed
                        } else {
                              prevValue = sliderValue * Math.PI / (2 * SLIDER_MAX);
                        }
                  }
            }

            /**
             * Returns the change in heading in radians from the last time the slider was changed (or from when the
slider was generated) and updates prevValue
             * @param sliderValue the new value of the slider in non-standard units
             * @return the change in heading in radians
             */
            private double headingFromSliderVal(int sliderValue) {
                  double newHeading = sliderValue * Math.PI / SLIDER_MAX; // Convert sliderValue to radians
                  double changeInHeading = newHeading - prevValue;
                  prevValue = newHeading; // Update prevValue for the next time the slider is changed
                  return changeInHeading;
            }

            /**
             * Returns the change in pitch in radians from the last time the slider was changed (or from when the
slider was generated) and updates prevValue
             * @param sliderValue the new value of the slider in non-standard units
             * @return the change in pitch in radians
             */
            private double pitchFromSliderVal(int sliderValue) {
                  double newPitch = sliderValue * Math.PI / (2 * SLIDER_MAX);
                  double changeInPitch = newPitch - prevValue;
```

```
                prevValue = newPitch;
                return changeInPitch;
        }

        /**
         * For a global slider, this method changes the sliders it affects
         * @param newSliderValue the new value of the slider this object is a listener for in non-standard units
         */
        private void updateParallelSlider(int newSliderValue) {
                int changeInValue;
                if (secondaryParallelSlider != null) { // If this SliderListener is for a global pitch slider
                        changeInValue = (int)(newSliderValue - Math.round(2 * SLIDER_MAX * prevValue / Math.PI)); //
Calculate the slider's change in non-standard units
                        int newValue0;
                        if (parallelSlider.getName().equals("localPitch")) {
                                newValue0 = parallelSlider.getValue() - changeInValue; // Increase in global pitch
normally causes a decrease in local pitch
                        } else { // if the name is "localPitchInv"
                                newValue0 = parallelSlider.getValue() + changeInValue;
                        }
                        if (newValue0 > SLIDER_MAX) { // If the new pitch is off the top end of the slider
                                newValue0 = 2 * SLIDER_MAX - newValue0; // The amount newValue0 was above SLIDER_MAX
becomes how much it is below SLIDER_MAX
                                invertPitchSlider();
                        } else if (newValue0 < -SLIDER_MAX) { // If the new pitch is off the bottom end of the
slider
                                newValue0 = -newValue0 - 2 * SLIDER_MAX; // The amount newValue0 was under -SLIDER_MAX
becomes how much it is above -SLIDER_MAX
                                invertPitchSlider();
                        }
                        parallelSlider.setValue(newValue0);
                } else {
                        changeInValue = (int)(newSliderValue - Math.round(SLIDER_MAX * prevValue / Math.PI));
                        int newValue = parallelSlider.getValue() + changeInValue;
                        // Wrap new heading so that the slider goes off the bottom and onto the top or vice versa
                        if (newValue > SLIDER_MAX) {
                                newValue -= 2 * SLIDER_MAX; // The amount above SLIDER_MAX becomes the amount above -
SLIDER_MAX
                        } else if (newValue < -SLIDER_MAX) {
```

```
                        newValue += 2 * SLIDER_MAX; // The amount below -SLIDER_MAX becomes the amount below
SLIDER_MAX
                    }
                    parallelSlider.setValue(newValue);
                }
            }

            /**
             * For a global pitch slider, this method inverts the local pitch slider by changing its name, toggling
a ray box property and adding or subtracting pi radians from the local heading
             */
            private void invertPitchSlider() {
                if (parallelSlider.getName().equals("localPitch")) {
                    parallelSlider.setName("localPitchInv"); // The ray box is no longer upside down, so local
pitch needs to go down when global pitch goes up
                } else {
                    parallelSlider.setName("localPitch"); // The ray box is now upside down, so local pitch
needs to go up when global pitch goes up
                }
                rayBox.toggleLocalPitchInverted();

                int newValue1 = secondaryParallelSlider.getValue();
                // To avoid aliasing (where one orientation can be expressed in more than one way), the amount a
pitch is outside the interval [-pi/2, pi/2] becomes the amount it is within the interval, and the heading is offset
by pi and then a multiple of 2pi to keep it between -pi and pi
                newValue1 += SLIDER_MAX;
                if (newValue1 > SLIDER_MAX) {
                    newValue1 -= 2 * SLIDER_MAX; // The previous heading was already canonical, so the heading
only needs to be offset by 2pi at the most
                }
                secondaryParallelSlider.setValue(newValue1);
            }

        }

}
```

3.4 Viewport.java

```
package RefractionSim;

import java.awt.*;
import java.awt.font.FontRenderContext;
import java.awt.image.BufferedImage;
import java.awt.image.MemoryImageSource;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

/**
 * Class for 3-D viewports which deals with the user interface requirements of being a subclass of JPanel as well as
handling its own 'scene' of objects
 * @author William Platt
 *
 */
public class Viewport extends JPanel {
	private int frameWidth;
	private int frameHeight;
	private Color[][] frameBuffer;
	private double[][] depthBuffer;
	private int[][] objectBuffer; // Stores the ID of the object in the foreground for each pixel
	private Color bgColor = new Color(0, 0, 0); // Black
	private static boolean orthographic = false; // By default, parallel lines converge to a vanishing point (as
in real life)
	private double zoomX;
	private double zoomY;
	private static final double NEAR_CLIP = 0.01; // The closest a point on a face can be to the camera before it
is no longer rendered
	private static final double FAR_CLIP = 10000; // The furthest a point on a face can be from the camera before
it is no longer rendered
	private Matrix clipMatrix = new Matrix(4, 4); // Matrix for transforming camera-space co-ordinates into clip
space co-ordinates
```

```
	private static Object3D[] objectList = new Object3D[100]; // Array of all objects in the scene where an object's index in this list is equal to its ID
	private static int objectListLength = 0;
	private static int worldMaterial; // Index of the material of the surroundings
	private int selectedObjID = -1; // No object selected by default
	private boolean anglesInDegrees; // Whether angles should be output in degrees or radians
	private static String[] materials = new String[100]; // List of the names of all materials
	private static double[] refractiveIndices = new double[100]; // List of the refractive indices of all materials
	private static int numOfMaterials = 0;

	/**
	 * Constructor for the Viewport class which sets its size, prepares it for rendering, adds the camera and target to the scene and sets a listener for all events within the viewport that need handling
	 * @param frameX width of the viewport in pixels
	 * @param frameY height of the viewport in pixels
	 */
	public Viewport(int frameX, int frameY) {
		this.frameWidth = frameX;
		this.frameHeight = frameY;
		this.frameBuffer = new Color[frameX][frameY];
		this.depthBuffer = new double[frameX][frameY];
		this.objectBuffer = new int[frameX][frameY];
		this.clearBuffers();
		initialiseMaterials();

		this.anglesInDegrees = true;
		setBackground(this.bgColor);
		double verticalFOV = 20 * Math.PI / 180; // Up/down field of view in radians
		double horizontalFOV = 2 * Math.atan(Math.tan(verticalFOV / 2) * frameX / frameY); // Left/right field of view in radians
		// Set how quickly objects shrink as they get further away
		this.zoomX = 1 / Math.tan(horizontalFOV);
		this.zoomY = 1 / Math.tan(verticalFOV);
		calcClipMatrix();
		initialiseScene();
		ViewportListener listener = new ViewportListener();
		this.addMouseListener(listener);
		this.addMouseMotionListener(listener);
```

```
            this.addMouseWheelListener(listener);
            this.addKeyListener(listener);
      }

      /**
       * Defines some common materials by giving them names and refractive indices
       */
      private void initialiseMaterials() {
            Viewport.worldMaterial = 0;
            materials[0] = "Air";
            materials[1] = "Water";
            materials[2] = "Typical glass (soda-lime)";
            materials[3] = "Human eye";
            materials[4] = "Ice";
            materials[5] = "Diamond";
            materials[6] = "Ethanol";
            materials[7] = "PLA plastic";
            materials[8] = "Sapphire";
            refractiveIndices[0] = 1.00;
            refractiveIndices[1] = 1.33;
            refractiveIndices[2] = 1.52;
            refractiveIndices[3] = 1.39;
            refractiveIndices[4] = 1.31;
            refractiveIndices[5] = 2.42;
            refractiveIndices[6] = 1.36;
            refractiveIndices[7] = 1.46;
            refractiveIndices[8] = 1.77;
            numOfMaterials = 9;
      }

      /**
       * Sets up the default camera and target and adds them to the scene/objectList; objectList[0] is always the
camera and objectList[1] is always the target
       */
      private void initialiseScene() {
            Object3D camera = new Object3D(null, null);
            Vector cameraOffset = new Vector(3);
            cameraOffset.setElement(0, 0);
            cameraOffset.setElement(1, 0);
```

```
            cameraOffset.setElement(2, -6);
            camera.displace(cameraOffset);
            objectList[0] = camera;
            camera.setID(0);
            objectList[1] = new Target(Mesh.Primitive.CUBE, new Color(50, 200, 100, 100), 2);
            objectList[1].setID(1);
            objectListLength = 2;
      }

      /**
       * Redraws the contents of the viewport; the scene is re-rendered into the buffers including the outline for
the selected ray box, then the buffers are drawn inside the viewport component and ray box labels and angles drawn
on top.
       * This method should be called via repaint() which decides whether it is worthwhile drawing the component and
passes an appropriate parameter.
       */
      @Override
      public void paintComponent(Graphics g) {
            super.paintComponent(g); // Call the method as it is defined in the JComponent class of which the
Viewport class is a descendant
            render(); // Clear the buffers and re-render the 3-D objects to them
            outlineSelectedObj(); // Add to the buffers the outline around the selected ray box so the user can see
which is selected

            // Draw frame in the viewport
            int[] pixelValues = new int[frameWidth * frameHeight];
            for (int i = 0; i < frameHeight; i++) {
                  for (int j = 0; j < frameWidth; j++) {
                        pixelValues[j + frameWidth * i] = frameBuffer[j][i].getRGB();
                  }
            }
            // Putting pixel values into an Image and drawing the Image inside the viewport is faster than drawing
each pixel straight to the viewport
            Image img = createImage(new MemoryImageSource(frameWidth, frameHeight, pixelValues, 0, frameWidth));
            g.drawImage(img, 0, 0, null); // Draw the image to the viewport with top left at (0, 0) relative to the
viewport (the top left of the viewport)

            // Write angles and ray box labels
            Graphics2D g2 = (Graphics2D)(g);
```

```
            g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON); // Use anti-
aliasing for text with smoother edges
            FontRenderContext frc = g2.getFontRenderContext();
            writeAngles(g, frc); // Write the angles over the image in the viewport
            writeRayBoxLabels(g, frc); // Write the ray box labels over the image in the viewport (this includes the
angles which were drawn first)
      }

      /**
       * Clears the buffers and renders the 3-D scene to the buffers from the camera's point of view
       */
      private void render() {
            clearBuffers();
            Matrix uprightToCamera = objectList[0].getOrientation().transpose(); // Store the matrix for
transforming points from the camera's upright space to the camera's object space
            for (int i = objectListLength - 1; i > 0; i--) { // objectList[0] is the camera and isn't rendered
                  if (objectList[i] == null) { // Deleted objects leave null pointers in objectList where they once
were, so skip the rendering of these
                        continue; // Skip to the end of this iteration (meaning move on to the next object in the
scene)
                  }
                  if (inView(objectList[i])) { // Check the object is potentially in view of the camera before
spending time attempting to render it
                        Color objectColor = objectList[i].getColor();
                        Matrix objectToUpright = objectList[i].getOrientation(); // Store the matrix for
transforming points from the current object's object space to the current object's upright space
                        int[][] faces = objectList[i].getMesh().getFaces();
                        Vector[] verts = objectList[i].getMesh().getVerts();
                        Vector[] normalisedSpaceVerts = new Vector[verts.length];
                        Vector[] screenSpaceVerts = new Vector[verts.length];
                        for (int j = 0; j < faces.length; j++) { // Iterate through each face of the object
                              for (int k = 0; k < faces[j].length; k++) { // Iterate through each vertex of the face
                                    int vertIndex = faces[j][k];
                                    if (screenSpaceVerts[vertIndex] == null) { // Faces share vertices, so some
vertices may have been mapped to screen space already
                                          Vector worldCoord =
objectToUpright.multiply(verts[vertIndex]).add(objectList[i].getOrigin()); // Map the point from object space to
world space (via the object's upright space)
```

```
                                        Vector cameraCoord =
uprightToCamera.multiply(worldCoord.subtract(objectList[0].getOrigin())); // Map the point from world space to
camera space (via the camera's upright space)
                                        Vector normalisedCoord = project(cameraCoord); // Map the point from
camera space to normalised clip space
                                        normalisedSpaceVerts[vertIndex] = new Vector(3);
                                        normalisedSpaceVerts[vertIndex].setElements(normalisedCoord);
                                        Vector screenCoord = new Vector(3);
                                        screenCoord.setElement(0, (normalisedCoord.getElement(0) + 1) * frameWidth
/ 2); // Map normalised x co-ordinate to screen space
                                        screenCoord.setElement(1, frameHeight * (0.5 -
normalisedCoord.getElement(1) * 0.5)); // Map normalised y co-ordinate to screen space - notice that the normalised
y co-ordinate is negated, causing the face's normal to flip
                                        screenCoord.setElement(2, normalisedCoord.getElement(2)); // Store the
normalised depth along with each screen space co-ordinate
                                        screenSpaceVerts[vertIndex] = screenCoord;
                                    }
                                }
                                Vector p0 = screenSpaceVerts[faces[j][0]];
                                Vector p1 = screenSpaceVerts[faces[j][1]];
                                Vector p2 = screenSpaceVerts[faces[j][2]];
                                Vector normal = p1.subtract(p0).crossProduct(p2.subtract(p0)).normalise(); //
Calculate a normalised (length 1) screen space normal to the face
                                if ((normal.getElement(2) > 0) || (objectColor.getAlpha() < 255)) { // Don't render
the face if it is facing away from the camera and the object is opaque; remember that the normal is flipped in the
mapping to screen space
                                    double d = p0.dotProduct(normal); // The equation of a plane is p.n = d where p
is a point in the plane and n is the normal
                                    if ((p0.getElement(0) >= 0) && (p0.getElement(0) <= frameWidth) &&
(p0.getElement(1) >= 0) && (p0.getElement(1) <= frameHeight) && (p0.getElement(2) >= 0) && (p0.getElement(2) <= 1)
||
                                            (p1.getElement(0) >= 0) && (p1.getElement(0) <= frameWidth) &&
(p1.getElement(1) >= 0) && (p1.getElement(1) <= frameHeight) && (p1.getElement(2) >= 0) && (p1.getElement(2) <= 1)
                                            || (p2.getElement(0) >= 0) && (p2.getElement(0) <= frameWidth) &&
(p2.getElement(1) >= 0) && (p2.getElement(1) <= frameHeight) && (p2.getElement(2) >= 0) && (p2.getElement(2) <= 1))
{ // Render the face if at least one of the vertices is visible to the camera
                                        Color faceColor = calcFaceColor(normalisedSpaceVerts[faces[j][0]],
normalisedSpaceVerts[faces[j][1]], normalisedSpaceVerts[faces[j][2]], objectList[i]); // Shading calculations work
better in normalised clip space than screen space
```

```
                                        rasterise(p0, p1, p2, normal, d, faceColor, i); // Draw the visible parts
of the triangle into the buffers using screen space co-ordinates and a screen space normal vector
                                    }
                                }
                            }
                        }
                    }
                }

                /**
                 * Overwrites areas of the frame buffer in order to create a bright orange outline 1 pixel thick around the
selected object so that the user can identify which object is selected
                 */
                private void outlineSelectedObj() {
                    if ((selectedObjID > 1) && (selectedObjID < objectListLength)) { // Check there is an object selected
                        if (objectList[selectedObjID] != null) { // Check that the selected object wasn't deleted
                            if (objectList[selectedObjID].getMesh() != null) { // Check the selected object has a mesh
(unlike the camera)

                                Color outlineColor = new Color(255, 170, 64); // Bright orange
                                for (int i = 0; i < frameWidth; i++) { // For each column of pixels, place dots where
there are boundaries of the selected object
                                    boolean lastBelongsToObject = false;
                                    for (int j = 0; j < frameHeight; j++) {
                                        if (!lastBelongsToObject) {
                                            if (objectBuffer[i][j] == selectedObjID) {
                                                if (j > 0) { // If the object starts above the screen or at
the very top then the outline cannot be drawn here
                                                    frameBuffer[i][j - 1] = outlineColor;
                                                }
                                                lastBelongsToObject = true;
                                            }
                                        } else {
                                            if (objectBuffer[i][j] != selectedObjID) {
                                                frameBuffer[i][j] = outlineColor;
                                                lastBelongsToObject = false;
                                            }
                                        }
                                    }
```

```
                                    }
                                    for (int j = 0; j < frameHeight; j++) { // For each row of pixels, place dots where
there are boundaries of the selected object
                                          boolean lastBelongsToObject = false;
                                          for (int i = 0; i < frameWidth; i++) {
                                                if (!lastBelongsToObject) {
                                                      if (objectBuffer[i][j] == selectedObjID) {
                                                            if (i > 0) { // If the object starts left of the screen or at
the very left then the outline cannot be drawn here
                                                                  frameBuffer[i - 1][j] = outlineColor;
                                                            }
                                                            lastBelongsToObject = true;
                                                      }
                                                } else {
                                                      if (objectBuffer[i][j] != selectedObjID) {
                                                            frameBuffer[i][j] = outlineColor;
                                                            lastBelongsToObject = false;
                                                      }
                                                }
                                          }
                                    }

                              }
                        }
                  }
            }

            /**
             * Draws/Writes all angles of incidence, refraction and reflection inside the viewport (on top of anything
already drawn in the viewport)
             * @param g the graphics context for the viewport which allows text to be drawn in the viewport
             * @param frc the FontRenderContext of the 2-D graphics context (which should have anti-aliasing) which allows
the width of text to be determined without drawing
             */
            private void writeAngles(Graphics g, FontRenderContext frc) {
                  Font angleFont = new Font("SansSerif", Font.PLAIN, 13);
                  double halfWidth;
                  if (anglesInDegrees) {
```

```
                halfWidth = angleFont.getStringBounds("00.00\u00B0", frc).getWidth() / 2.0; // Calculate half the
typical width of an angle written in degrees (in pixels)
            } else {
                halfWidth = angleFont.getStringBounds("0.000\u03C0", frc).getWidth() / 2.0; // Calculate half the
typical width of an angle written in radians (in pixels)
            }
            g.setFont(angleFont);
            g.setColor(Color.WHITE);
            Matrix uprightToCamera = objectList[0].getOrientation().transpose(); // Matrix for transforming points
from the camera's upright space to the camera's object space (camera space)
            for (int i = 2; i < objectListLength; i++) { // Iterate through each object in the scene except for the
camera and target (IDs 0 and 1)
                if (objectList[i] instanceof Beam) {
                      Beam lightBeam = (Beam)(objectList[i]);
                      if (lightBeam.getAnglesVisible()) {
                            int numOfAngles = lightBeam.getNumOfAngles();
                            double[] angles = lightBeam.getAngles();
                            Vector[] anglePositions = lightBeam.getAnglePositions(); // Points in world space of
the angles
                            for (int j = 0; j < numOfAngles; j++) {
                                  Vector cameraCoord =
uprightToCamera.multiply(anglePositions[j].subtract(objectList[0].getOrigin())); // Map the world space points to
camera space (via the camera's upright space)
                                  Vector normalisedCoord = project(cameraCoord);
                                  Vector screenCoord = new Vector(3);
                                  // Map the points in normalised clip space to screen space and offset slightly
because text position is defined by its top left, not its centre
                                  screenCoord.setElement(0, Math.round(((normalisedCoord.getElement(0) + 1) *
frameWidth / 2) - halfWidth));
                                  screenCoord.setElement(1, Math.round(frameHeight * (0.5 -
normalisedCoord.getElement(1) * 0.5) - 5));
                                  if (anglesInDegrees) {
                                        double angle = Math.round(18000.0 * angles[j] / Math.PI) / 100.0; // 2
decimal places
                                        g.drawString(Double.toString(angle) + "\u00B0",
(int)(screenCoord.getElement(0)), (int)(screenCoord.getElement(1))); // Write the angle in the appropriate place
within the viewport
                                  } else {
```

```
                                        double angle = Math.round(1000.0 * angles[j] / Math.PI) / 1000.0; // 3
decimal places
                                        g.drawString(Double.toString(angle) + "\u03C0",
(int)(screenCoord.getElement(0)), (int)(screenCoord.getElement(1)));
                                  }
                            }
                      }
                }
          }

          /**
           * Draws/Writes all ray box labels inside the viewport (on top of anything that has already been drawn)
           * @param g the graphics context for the viewport which allows text to be drawn in the viewport
           * @param frc the FontRenderContext of the 2-D graphics context (which should have anti-aliasing) which allows
the ascent (maximum height above the baseline) of text to be determined without drawing
           */
          private void writeRayBoxLabels(Graphics g, FontRenderContext frc) {
                Font rayBoxLabelFont = new Font("SansSerif", Font.BOLD, 20);
                int minX = 5; // Left margin preventing the label being written too close to the left edge of the
viewport
                int minY = (int)(Math.round(rayBoxLabelFont.getLineMetrics("W", frc).getAscent() + 5)); // Top margin
preventing the label being written too close to the top of the viewport
                g.setFont(rayBoxLabelFont);
                g.setColor(Color.BLUE);
                for (int i = 1; i < objectListLength; i++) {
                      if (objectList[i] instanceof RayBox) {
                            boolean found = false;
                            // Working from left to right down the screen, find the first pixel belonging to this
particular ray box
                            for (int y = 0; (y < frameHeight) && (found == false); y++) {
                                  for (int x = 0; (x < frameWidth) && (found == false); x++) {
                                        if (objectBuffer[x][y] == i) {
                                              if (x < minX) {
                                                    x = minX;
                                              }
                                              if (y < minY) {
                                                    y = minY;
                                              }
```

```
                                        RayBox rayBox = (RayBox)(objectList[i]);
                                        g.drawString(rayBox.getLabel(), x, y);
                                        found = true;
                                    }
                                }
                            }
                        }
                    }
                }

                /**
                 * Returns the list of 3-D objects in the scene where the item at index 0 is the camera and the item at index
1 is the target
                 * @return the list of all 3-D objects in the scene
                 */
                public static Object3D[] getObjectList() {
                        return objectList;
                }

                /**
                 * Returns true if the bounding box of an object is at least partially visible to the camera (but the object
itself may still be completely out of sight)
                 * @param obj the 3-D object to be checked for visibility
                 * @return whether or not the bounding box of obj is in view of the camera
                 */
                private boolean inView(Object3D obj) { // Even if this returns true, the object may not be in view, as this is
a quick algorithm
                        boolean inView = false;
                        // For each array variable, index 0 represents the x-axis, index 1 represents the y-axis and index 2
represents the z-axis
                        boolean[] below = {false, false, false};
                        boolean[] above = {false, false, false};
                        boolean[] span = {false, false, false}; // Each span element is set to true if there is a normalised
clip space point in the visible range or one point above the range and one point below the range
                        int[][] bounds = {{-1, 1}, {-1, 1}, {0, 1}}; // For each inner array, index 0 is the lower bound for
that axis and index 1 is the upper bound
                        Matrix objectToUpright = obj.getOrientation(); // Matrix for transforming points from object space to
upright space
```

```
            Matrix uprightToCamera = objectList[0].getOrientation().transpose(); // Matrix for transforming points
from the camera's upright space to the camera's object space (camera space)
            Vector[] boxVerts = obj.getBoxVerts();
            int i = 0;
            while ((i < boxVerts.length) && (!inView)) {
                  Vector worldCoord = objectToUpright.multiply(boxVerts[i]).add(obj.getOrigin()); // Map the point
in object space to world space (via upright space)
                  Vector cameraCoord = uprightToCamera.multiply(worldCoord.subtract(objectList[0].getOrigin())); //
Map the point in world space to camera space (via the camera's upright space)
                  Vector normalisedCoord = project(cameraCoord);
                  for (int j = 0; (j < 3) && (inView == false); j++) { // For each of the x, y and z axes
                        if (span[j] == false) {
                              if (normalisedCoord.getElement(j) >= bounds[j][0]) {
                                    if (normalisedCoord.getElement(j) <= bounds[j][1]) {
                                          span[j] = true; // A point within the bounds is treated like one on each
side because both contribute to making the box visible on that axis
                                    } else {
                                          above[j] = true;
                                    }
                              } else {
                                    below[j] = true;
                              }
                              if ((below[j]) && (above[j])) {
                                    span[j] = true;
                              }
                              if ((span[0]) && (span[1]) && (span[2])) {
                                    inView = true;
                              }
                        }
                  }
                  i++;
            }
            return inView;
      }

      /**
       * Maps a point from camera space to normalised clip space and returns the point as a Vector object
       * @param cameraCoord the point in camera space which is to be mapped to normalised clip space
       * @return cameraCoord mapped to normalised clip space
```

```
	 * @throws IllegalArgumentException if the cameraCoord parameter is not a 3-row vector
	 */
	private Vector project(Vector cameraCoord) {
		if (cameraCoord.getN() != 3) {
			throw new IllegalArgumentException("A point in camera space must be a 3-D vector");
		} else {
			if (orthographic) {
				double zoom = objectList[0].getOrigin().modulus() / 3000; // Orthographic visualisation
means that object size is independent of distance, but this means that moving further away does not give a wider
view, so the view cube (rather than the view frustum in perspective projection) is stretched with distance to make
the view wider when the camera is further away
				// The clip space is the same as camera space
				Vector normalised3D = new Vector(3);
				normalised3D.setElement(0, cameraCoord.getElement(0) / (frameWidth * zoom));
				normalised3D.setElement(1, cameraCoord.getElement(1) / (frameHeight * zoom));
				normalised3D.setElement(2, cameraCoord.getElement(2) / FAR_CLIP);
				return normalised3D;
			} else {
				Vector camera4D = new Vector(4); // Perspective projection in 3-D is not a linear
transformation in three dimensions, so cannot be performed using 3 by 3 matrices and 3-row vectors; 4 by 4 matrices
and 4-row vectors are needed
				camera4D.setElement(0, cameraCoord.getElement(0));
				camera4D.setElement(1, cameraCoord.getElement(1));
				camera4D.setElement(2, cameraCoord.getElement(2));
				camera4D.setElement(3, 1); // Element 3 is set to 1 so that it becomes the old value of
element 2 after the clip matrix has been applied
				Vector clip4D = clipMatrix.multiply(camera4D);
				if (clip4D.getElement(3) == 0) {
					clip4D.setElement(3, 0.0001); // Avoid divide by zero
				}
				Vector normalised3D = new Vector(3);
				// Perform the perspective divide to make clip space into normalised clip space
				normalised3D.setElement(0, clip4D.getElement(0) / clip4D.getElement(3));
				normalised3D.setElement(1, clip4D.getElement(1) / clip4D.getElement(3));
				normalised3D.setElement(2, clip4D.getElement(2) / clip4D.getElement(3));
				return normalised3D;
			}
		}
	}
```

```
	/**
	 * Returns the colour to render a particular face in based on the object's overall colour and how much the face is pointing towards the camera in normalised clip space
	 * @param p0 the first vertex of the face in normalised clip space. It is important that the order of the vertices is correct
	 * @param p1 the second vertex of the face
	 * @param p2 the third vertex of the face
	 * @param object the object to which the face belongs
	 * @return the colour to render the face defined by the three input points
	 * @throws IllegalArgumentException if any of p0, p1 and p2 is not a 3-row vector
	 */
	private Color calcFaceColor(Vector p0, Vector p1, Vector p2, Object3D object) {
		if ((p0.getN() != 3) || (p1.getN() != 3) || (p2.getN() != 3)) {
			throw new IllegalArgumentException("Face colour can only be calculated from points in 3-D space");
		} else {
			Color faceColor; // The more the face is pointing towards the camera, the lighter the colour will be
			Color objectColor = object.getColor();
			if (object instanceof Beam) { // Beams are rendered as solid colour without shadows (although a face in front of it with some transparency may affect its colour)
				faceColor = objectColor;
			} else {
				Vector screenNormal = p1.subtract(p0).crossProduct(p2.subtract(p0)).normalise();

				double brightFactor; // A value between 0.2 and 1 where 0 (if zero were allowed) would make faceColor completely black and 1 would make faceColor the same as objectColor
				if (objectColor.getAlpha() < 255) {
					if (screenNormal.getElement(2) < 0) { // If the face is pointing towards the camera
						brightFactor = 1 - 262144 * 2 * (1 + screenNormal.getElement(2)); // 262144 is a value obtained through experimentation and is a power of 2, reducing computation time
						if (brightFactor < 0.5) {
							brightFactor = 0.5;
						}
					} else {
						brightFactor = -(screenNormal.getElement(2) - 1);
						if (brightFactor < 0.2) {
							brightFactor = 0.2;
						}
```

```
                        }
                    } else {
                        brightFactor = 1 - 524288 * (1 + screenNormal.getElement(2)); // 524288 is another
experimental power of 2
                        if (brightFactor < 0.7) {
                            brightFactor = ((brightFactor - 0.7) / 2) + 0.7;
                            if (brightFactor < 0.6) {
                                brightFactor = ((brightFactor - 0.6) / 2) + 0.6;
                                if (brightFactor < 0.5) {
                                    brightFactor = ((brightFactor - 0.5) / 2) + 0.5;
                                }
                            }
                        }
                        if (brightFactor < 0.4) {
                            brightFactor = 0.4;
                        }
                    }
                    faceColor = new Color((int)(Math.round(objectColor.getRed() * brightFactor)),
(int)(Math.round(objectColor.getGreen() * brightFactor)),
                            (int)(Math.round(objectColor.getBlue() * brightFactor)),
objectColor.getAlpha());
                }
                return faceColor;
            }
        }

        /**
         * Adds the parameter ray box and its beam to the scene and makes the ray box the selected object before
updating the user interface and viewport. A dialog box informs the user if they have too many ray boxes to add any
more
         * @param newRayBox the ray box to add to the scene along with its beam
         */
        public void addRayBox(RayBox newRayBox) {
            RefractionSimulator window = (RefractionSimulator)(SwingUtilities.windowForComponent(this));
            int i = 1; // IDs 0 and 1 are the camera and target, so can be skipped
            // Find the first null element in objectList, the next element will also be null (unless there is an
overflow) because ray boxes and their beams are consecutive and both are removed at a time
            while (i < objectListLength) {
                if (objectList[i] == null) {
```

```
                        break;
                    }
                    i++;
                }
                if (i < objectList.length - 1) {
                    newRayBox.setID(i);
                    objectList[i] = newRayBox;
                    Beam newBeam = newRayBox.getLightBeam();
                    newBeam.setID(i + 1);
                    objectList[i + 1] = newBeam;
                    selectedObjID = i;
                    if (i + 1 >= objectListLength) {
                        objectListLength = i + 2;
                    }
                    if (window != null) {
                        window.updatePropertiesPanel(newRayBox);
                    }
                    newBeam.update();
                    repaint();
                } else { // Not enough room for a new ray box and light beam
                    JOptionPane.showMessageDialog(window, "You have too many ray boxes to add another one. You must
delete an existing ray box if you wish to add another");
                }
            }

            /**
             * Removes the selected ray box and its beam from the scene and updates the user interface and viewport; there
is no longer a selected object
             * @throws IllegalArgumentException if the selected object isn't a ray box
             */
            public void removeRayBox() {
                if (objectList[selectedObjID] instanceof RayBox) {
                    RayBox toRemove = (RayBox)(objectList[selectedObjID]);
                    objectList[toRemove.getID()] = null;
                    objectList[toRemove.getLightBeam().getID()] = null;
                    if (toRemove.getLightBeam().getID() + 1 == objectListLength) { // The light beam was the last non-
null object
                        objectListLength -= 2; // Reduce the length to reflect the last two items being removed
                    }
```

```
                selectedObjID = -1;
                RefractionSimulator window = (RefractionSimulator)(SwingUtilities.windowForComponent(this));
                window.updatePropertiesPanel(null);
                repaint();
            } else {
                throw new IllegalArgumentException("Cannot remove a ray box if one is not selected");
            }
        }

        /**
         * Resets the frame buffer, depth buffer and object buffer in preparation for re-rendering
         */
        private void clearBuffers() {
            for (int i = 0; i < this.frameWidth; i++) {
                for (int j = 0; j < this.frameHeight; j++) {
                    this.frameBuffer[i][j] = this.bgColor;
                    this.depthBuffer[i][j] = 1; // Clip space points have a depth mapped between 0 (near clip)
and 1 (far clip)
                    this.objectBuffer[i][j] = -1; // No object
                }
            }
        }

        /**
         * Returns the width of the viewport in pixels
         * @return the width of the viewport in pixels
         */
        @Override
        public int getWidth() {
            return this.frameWidth;
        }

        /**
         * Returns the height of the viewport in pixels
         * @return the height of the viewport in pixels
         */
        @Override
        public int getHeight() {
            return this.frameHeight;
```

```
	}

	/**
	 * Returns an array of the names of all the materials - preset and custom - in the same order as the
refractive indices array
	 * @return an array of the names of all preset and custom materials
	 */
	public static String[] getMaterials() {
		return materials;
	}

	/**
	 * Returns an array of the absolute refractive indices of all the preset and custom materials in the same
order as the material names array.
	 * @return the refractive indices of all the materials
	 */
	public static double[] getRefractiveIndices() {
		return refractiveIndices;
	}

	/**
	 * Returns the index of the material that the world is set to
	 * @return the index of the material that the world is set to
	 */
	public static int getWorldMaterial() {
		return worldMaterial;
	}

	/**
	 * Sets the material of the world to that with the index passed as a parameter
	 * @param materialID the index of the material that the world should be
	 * @throws IllegalArgumentException if there is currently not a material with the index materialID
	 */
	public void setWorldMaterial(int materialID) {
		if ((materialID < 0) || (materialID >= numOfMaterials)) {
			throw new IllegalArgumentException("World material cannot be set to one which does not exist");
		} else {
			worldMaterial = materialID;
			recalculateBeams();
```

```
            repaint();
      }
}

/**
 * Returns the index of the material that the target is set to
 * @return the index of the material that the target is set to
 */
public int getTargetMaterial() {
      Target target = (Target)(objectList[1]);
      return target.getMaterial();
}

/**
 * Sets the material of the target to that with the index passed as a parameter
 * @param materialID the index of the material that the target should be set to
 * @throws IllegalArgumentException if there is currently not a material with the index materialID
 */
public void setTargetMaterial(int materialID) {
      if ((materialID < 0) || (materialID >= numOfMaterials)) {
            throw new IllegalArgumentException("World material cannot be set to one which does not exist");
      } else {
            Target target = (Target)(objectList[1]);
            target.setMaterial(materialID);
            recalculateBeams();
            repaint();
      }
}

/**
 * Returns the total number of materials that are defined
 * @return the total number of materials
 */
public static int getNumOfMaterials() {
      return numOfMaterials;
}

/**
```

```
	 * Creates a new material with the properties specified by the parameters and adds it to the end of the list of materials (which is a combination of the list of material names and the list of refractive indices)
	 * @param materialName the name of the new material
	 * @param refractiveIndex the absolute refractive index of the new material
	 */
	public void addMaterial(String materialName, double refractiveIndex) {
		materials[numOfMaterials] = materialName;
		refractiveIndices[numOfMaterials] = refractiveIndex;
		numOfMaterials++;
		RefractionSimulator window = (RefractionSimulator)(SwingUtilities.windowForComponent(this));
		window.updateMenuBar(); // Material menus will need a new item
	}

	/**
	 * Returns the name of the current shape of the target material
	 * @return the name of the current shape of the target material
	 */
	public String getTargetShape() {
		Target target = (Target)(objectList[1]);
		return target.getShape();
	}

	/**
	 * Sets the geometry of the target to that defined by newShape and recalculates the paths of all beams in the scene
	 * @param newShape the new shape of the target material
	 */
	public void setTargetShape(Mesh.Primitive newShape) {
		Target oldTarget = (Target)(objectList[1]);
		objectList[1] = new Target(newShape, oldTarget.getColor(), oldTarget.getMaterial()); // Create a new target to replace the old one; the new target has the same colour and material as the old one but a different shape
		recalculateBeams();
		repaint();
	}

	/**
	 * Recalculates the paths of all beams in the scene but does not re-render
	 */
	public void recalculateBeams() {
```

```
            for (int i = 2; i < objectListLength; i++) {
                  if (objectList[i] instanceof Beam) {
                        Beam lightBeam = (Beam)(objectList[i]);
                        lightBeam.update();
                  }
            }
      }

      /**
       * Returns true if orthographic projection is currently being used
       * @return whether orthographic projection is being used
       */
      public boolean isOrthographic() {
            return orthographic;
      }

      /**
       * Switches from orthographic projection to perspective projection or from perspective to orthographic and re-
renders
       */
      public void toggleOrthographic() {
            orthographic = !orthographic;
            RefractionSimulator window = (RefractionSimulator)(SwingUtilities.windowForComponent(this));
            window.updateMenuBar(); // Perspective checkbox needs changing
            this.repaint();
      }

      /**
       * Returns true if angles are set to display in degrees rather than radians
       * @return whether angles are displayed in radians
       */
      public boolean areAnglesInDegrees() {
            return anglesInDegrees;
      }

      /**
       * Changes angles from degrees to radians or vice versa and updates the interface accordingly
       */
      public void toggleAngleUnits() {
```

```
            anglesInDegrees = !anglesInDegrees;
            RayBox rayBox;
            try {
                  rayBox = (RayBox)(objectList[selectedObjID]);
                  RefractionSimulator window = (RefractionSimulator)(SwingUtilities.windowForComponent(this));
                  window.updatePropertiesPanel(rayBox); // Labels on sliders need to be changed
            } catch (Exception e) { // If no ray box is selected

            }
            repaint(); // Angles in the viewport need to be changed
      }

      /**
       * Orbits the selected ray box about the world origin by the heading and pitch specified
       * @param heading the angular displacement about the world's y-axis
       * @param pitch the angular displacement about the world's x-axis after rotation by the heading
       */
      public void globallyRotateRayBox(double heading, double pitch) {
            RayBox rayBox = (RayBox)(objectList[selectedObjID]);
            rayBox.orbitAboutOrigin(heading, pitch);
            rayBox.getLightBeam().update(); // Recalculate the path and geometry of the ray box's light beam
            repaint(); // Re-render the viewport with the new ray box position and recalculated light beam
      }

      /**
       * Rotates the selected ray box about its origin by the heading and pitch specified
       * @param heading the angular displacement about the world's y-axis
       * @param pitch the angular displacement about the world's x-axis after rotation by the heading
       */
      public void locallyRotateRayBox(double heading, double pitch) {
            RayBox rayBox = (RayBox)(objectList[selectedObjID]);
            rayBox.rotate(heading, pitch);
            rayBox.getLightBeam().update();
            repaint();
      }

      /**
       * Sets up clipMatrix to map camera space to clip space (when using perspective projection) for the given
zoomX, zoomdY, NEAR_CLIP and FAR_CLIP
```

```
	 */
	private void calcClipMatrix() {
		clipMatrix.setElement(0, 0, zoomX);
		clipMatrix.setElement(1, 1, zoomY);
		double element = FAR_CLIP / (FAR_CLIP - NEAR_CLIP);
		clipMatrix.setElement(2, 2, element);
		clipMatrix.setElement(3, 2, -NEAR_CLIP * element);
		clipMatrix.setElement(2, 3, 1);
	}

	/**
	 * From information in extended screen space (screen space with depth), draws the visible parts of a face into the buffers
	 * @param point0 the first vertex (index 0) of the face in screen space
	 * @param point1 the second vertex (index 1) of the face in screen space
	 * @param point2 the third vertex (index 2) of the face in screen space
	 * @param normal the normalised normal to the face in extended screen space
	 * @param d the value in the expression p.n = d where p is a point on the face and n is the normalised normal to the face (all in extended screen space)
	 * @param faceColor the colour to render the face
	 * @param objectID the ID of the object to which this face belongs (the index of the object in objectList)
	 */
	private void rasterise(Vector point0, Vector point1, Vector point2, Vector normal, double d, Color faceColor, int objectID) {
		// Find the depth of the closest point to save on depth calculations later
		double minDepth = point0.getElement(2);
		if (point1.getElement(2) < minDepth) {
			minDepth = point1.getElement(2);
		}
		if (point2.getElement(2) < minDepth) {
			minDepth = point2.getElement(2);
		}
		Edge2D edge0 = new Edge2D(point0.getElement(0), point0.getElement(1), point1.getElement(0), point1.getElement(1));
		Edge2D edge1 = new Edge2D(point1.getElement(0), point1.getElement(1), point2.getElement(0), point2.getElement(1));
		Edge2D edge2 = new Edge2D(point2.getElement(0), point2.getElement(1), point0.getElement(0), point0.getElement(1));
		// Find the tallest if the three edges
```

```
            Edge2D tallEdge = edge0;
            Edge2D shortEdge0 = edge1;
            Edge2D shortEdge1 = edge2;
            if (edge1.getHeight() > tallEdge.getHeight()) {
                  tallEdge = edge1;
                  shortEdge0 = edge0;
            }
            if (edge2.getHeight() > tallEdge.getHeight()) {
                  tallEdge = edge2;
                  shortEdge0 = edge0;
                  shortEdge1 = edge1;
            }
            if (tallEdge.getY0() != shortEdge0.getY0()) { // shortEdge0 should share the lowest point with tallEdge
(lowest in terms of y-value, not position on the screen)
                  Edge2D temp = shortEdge0;
                  shortEdge0 = shortEdge1;
                  shortEdge1 = temp;
            }
            // Fill in the pixels with centres contained by the precise triangle where the triangle is in front of
anything rendered so far
            int initialY = (int)(Math.round(tallEdge.getY0()));
            if (initialY < 0) {
                  initialY = 0;
            }
            int finalY = (int)(Math.round(shortEdge0.getY1()));
            if (finalY >= frameHeight) {
                  finalY = frameHeight - 1;
            }
            double dxTall = (tallEdge.getX1() - tallEdge.getX0()) / tallEdge.getHeight(); // Increase in x for the
tallest edge when y increases by 1
            double dxShort = (shortEdge0.getX1() - shortEdge0.getX0()) / shortEdge0.getHeight(); // Increase in x
for shortEdge0 when y increases by 1
            double ySkip = initialY - tallEdge.getY0(); // The signed change in y from the lowest point of the face
to the first pixel with centre inside the triangle
            double xTall = (ySkip + 0.5) * dxTall + tallEdge.getX0(); // The precise x co-ordinate of the tallest
edge for the y co-ordinate of the lowest row of pixels in the triangle
            double xShort = (ySkip + 0.5) * dxShort + shortEdge0.getX0(); // The precise x co-ordinate of shortEdge0
for the y co-ordinate of the lowest row of pixels in the triangle
```

```
            rasteriseHalfFace(initialY, finalY, xShort, xTall, dxShort, dxTall, minDepth, normal, d, faceColor,
objectID); // Draw the rows of pixels spanned by shortEdge0

            xTall = xTall + dxTall * (finalY - initialY);
            initialY = finalY;
            finalY = (int)(Math.round(shortEdge1.getY1()));
            if (finalY >= frameHeight) {
                  finalY = frameHeight - 1;
            }
            dxShort = (shortEdge1.getX1() - shortEdge1.getX0()) / shortEdge1.getHeight();
            ySkip = initialY - shortEdge1.getY0();
            xShort = (ySkip + 0.5) * dxShort + shortEdge1.getX0();
            rasteriseHalfFace(initialY, finalY, xShort, xTall, dxShort, dxTall, minDepth, normal, d, faceColor,
objectID); // Draw the rows of pixels spanned by shortEdge1
      }

      /**
       * Sets pixels in the buffers where the face is visible for initialY <= y < finalY
       * @param initialY the first row of pixels (lowest y value)
       * @param finalY the row of pixels after the last (highest y value)
       * @param xShort the x co-ordinate of the shorter edge when the y co-ordinate is initialY
       * @param xTall the x co-ordinate of the taller edge when the y co-ordinate is initailY
       * @param dxShort the change in x of the shorter edge when y is increased by 1
       * @param dxTall the change in x of the taller edge when y is increased by 1
       * @param minDepth the lowest depth value of any point on the face in extended screen space
       * @param normal the normalised normal to the face in extended screen space
       * @param d the value in the expression p.n = d where p is a point on the face and n is the normalised normal
to the face (all in extended screen space)
       * @param faceColor the colour to render the face
       * @param objectID the ID of the object to which the face belongs (the index of the object in objectList)
       */
      private void rasteriseHalfFace(int initialY, int finalY, double xShort, double xTall, double dxShort, double
dxTall, double minDepth, Vector normal, double d, Color faceColor, int objectID) {
            for (int pixelY = initialY; pixelY < finalY; pixelY++) {
                  int roundedxShort = (int)(Math.round(xShort));
                  int roundedxTall = (int)(Math.round(xTall));
                  if (roundedxTall <= roundedxShort) {
                        rasteriseFaceRow(roundedxTall, roundedxShort, pixelY, minDepth, normal, d, faceColor,
objectID);
```

```
                } else {
                        rasteriseFaceRow(roundedxShort, roundedxTall, pixelY, minDepth, normal, d, faceColor,
objectID);
                }
                xTall += dxTall;
                xShort += dxShort;
            }
        }

        /**
         * Sets pixels in the buffers where the face is visible for startX <= x < endX and y co-ordinate pixelY
         * @param startX the x co-ordinate of the first pixel on this row contained by the triangle
         * @param endX the x co-ordinate of the pixel after the last on this row contained by the triangle
         * @param pixelY the y co-ordinate of the row of pixels being set
         * @param minDepth the lowest depth value of any point on the face in extended screen space
         * @param normal the normalised normal to the face in extended screen space
         * @param d the value in the expression p.n = d where p is a point on the face and n is the normalised normal
to the face (all in extended screen space)
         * @param faceColor the colour to render the face
         * @param objectID the ID of the object to which the face belongs (the index of the object in objectList)
         */
        private void rasteriseFaceRow(int startX, int endX, int pixelY, double minDepth, Vector normal, double d,
Color faceColor, int objectID) {
            for (int pixelX = startX; pixelX < endX; pixelX++) {
                if ((pixelX >= 0) && (pixelX < frameWidth)) {
                    if ((pixelY >= 0) && (pixelY < frameHeight)) {
                        if (minDepth < depthBuffer[pixelX][pixelY]) { // If minDepth is too large then the
depth at this point will be
                                // Find the 3rd element of p by rearranging p.n = d to p[2] = (d - n[0] * p[0] -
n[1] * p[1]) / n[2]
                                double depth = (d - normal.getElement(0) * (pixelX + 0.5) - normal.getElement(1)
* (pixelY + 0.5)) / normal.getElement(2); // 0.5 is added to pixelX and pixelY to get the depth at the centre of the
pixel
                                if (depth < 0) { // Don't render in front of the near clip plane (or behind the
camera for orthographic projection)
                                    continue;
                                }
                                if (depth < depthBuffer[pixelX][pixelY]) { // If this face is closer than
anything else at this point so far, alter the buffers
```

```
                                        setPixel(pixelX, pixelY, depth, faceColor, objectID);
                                    }
                                }
                            }
                        }
                    }
                }

    /**
     * Changes the frame buffer, depth buffer and object buffer for a single pixel
     * @param x the x co-ordinate of the pixel to change
     * @param y the y co-ordinate of the pixel to change
     * @param depth the depth of the current face at the centre of this pixel
     * @param color the rendered colour of the face (including alpha)
     * @param objectID the ID of the object this face belongs to (the index of the object in objectList)
     */
    private void setPixel(int x, int y, double depth, Color color, int objectID) {
        if (color.getAlpha() != 255) { // Combine the previous colour of this pixel with the new semi-
transparent colour
            double opacity = color.getAlpha() / 255.0;
            double transparency = 1 - opacity;
            color = new Color((int)(Math.round(frameBuffer[x][y].getRed() * transparency + color.getRed() *
opacity)),
                    (int)(Math.round(frameBuffer[x][y].getGreen() * transparency + color.getGreen() *
opacity)),
                    (int)(Math.round(frameBuffer[x][y].getBlue() * transparency + color.getBlue() *
opacity))); // The produced colour will be opaque
        }
        this.frameBuffer[x][y] = color;
        this.depthBuffer[x][y] = depth;
        this.objectBuffer[x][y] = objectID;
    }

    /**
     * Orbit the camera around the world's origin where xChange is directly proportional to the heading and
yChange directly proportional to the pitch
     * @param xChange the signed change in the x position of the user's cursor between dragging events
     * @param yChange the signed change in the y position of the user's cursor between dragging events
```

```
	 */
	public void orbit(int xChange, int yChange) {
		double heading = Math.PI * xChange / frameWidth; // Dragging from far left to far right gives a heading of positive pi radians
		double pitch = 0.5 * Math.PI * yChange / frameHeight; // Dragging from top to bottom gives a pitch of positive 0.5pi radians
		objectList[0].orbit(heading, pitch);
		this.repaint();
	}

	/**
	 * Moves the camera towards or away from the world's origin
	 * @param scrollAmount the signed number of notches the mouse wheel is moved down or the equivalent amount dragged down with the right mouse button held
	 */
	public void zoom(double scrollAmount) {
		Vector newOrigin = objectList[0].getOrigin().scale(Math.pow(0.8, -scrollAmount));
		double distance = newOrigin.modulus();
		if (distance < 3) {
			newOrigin = newOrigin.scale(3 / distance); // Make newOrigin 0.5 units from the world's origin
		} else if (distance > 100) {
			newOrigin = newOrigin.scale(100 / distance); // Make newOrigin 100 units from the world's origin
		}
		objectList[0].setOrigin(newOrigin); // For each notch up the camera's distance from the origin reduces by 20%
		this.repaint();
	}

	/**
	 * Changes the selection to the ray box visible at position (x, y) in the viewport (otherwise the selection becomes empty) and updates the properties panel
	 * @param x the x co-ordinate of the pixel the user clicked relative to the top left of the viewport
	 * @param y the y co-ordinate of the pixel the user clicked relative to the top left of the viewport
	 */
	public void click(int x, int y) {
		RefractionSimulator window = (RefractionSimulator)(SwingUtilities.windowForComponent(this));
		if (!(window.getFocusOwner() instanceof JTextField)) { // Don't change the selected object while a text field is in focus because it is just about to lose focus and its value used to update the selected object
```

```
                  if ((objectBuffer[x][y] > 0) && (objectBuffer[x][y] < objectListLength)) { // Check if an object
was clicked
                        if (objectList[objectBuffer[x][y]] instanceof RayBox) { // Check if the clicked object was a
ray box
                              if (selectedObjID != objectBuffer[x][y]) { // Only spend time updating if the clicked
object wasn't already selected
                                    selectedObjID = objectBuffer[x][y]; // Select the ray box clicked
                                    window.updatePropertiesPanel((RayBox)(objectList[objectBuffer[x][y]])); //
Change the properties panel to show details of the newly selected object
                              }
                        } else {
                              selectedObjID = -1; // Make the selection empty
                              window.updatePropertiesPanel(null); // Clear the properties panel
                        }
                  } else {
                        selectedObjID = -1;
                        window.updatePropertiesPanel(null);
                  }
                  this.repaint();
            }
      }

      /**
       * Sets the label of the selected object to newLabel
       * @param newLabel the string to set the label of the selected object to
       */
      public void updateLabel(String newLabel) {
            RayBox rayBox = (RayBox)(objectList[selectedObjID]);
            rayBox.setLabel(newLabel);
            repaint(); // Redraw the label
      }

      /**
       * Sets the thickness of the selected ray box's beam to newThickness
       * @param newThickness the relative value that the thickness of the selected ray box's beam should be set to
       */
      public void updateBeamThickness(int newThickness) {
            RayBox rayBox = (RayBox)(objectList[selectedObjID]);
            rayBox.setBeamThickness(newThickness);
```

```
            repaint();
      }

      /**
       * Changes angle visibility from on to off or off to on for the selected ray box and refreshes the viewport
including angles
       */
      public void toggleShowAngles() {
            RayBox rayBox = (RayBox)(objectList[selectedObjID]);
            rayBox.setAnglesVisible(!rayBox.getAnglesVisible());
            repaint(); // Angles must be removed or drawn
      }

      /**
       * Positions and orientates the camera to face directly forwards at the same distance from the origin as
before
       */
      public void setViewFront() {
            Vector newOrigin = new Vector(3);
            newOrigin.setElement(2, -objectList[0].getOrigin().modulus()); // Keep the modulus the same
            Matrix newOrientation = new Matrix(3, 3);
            newOrientation.setElements(new double[] {1, 0, 0,   0, 1, 0,   0, 0, 1});
            objectList[0].setOrigin(newOrigin);
            objectList[0].setOrientation(newOrientation);
            this.repaint();
      }

      /**
       * Positions and orientates the camera to face directly backwards at the same distance from the origin as
before
       */
      public void setViewBack() {
            Vector newOrigin = new Vector(3);
            newOrigin.setElement(2, objectList[0].getOrigin().modulus());
            Matrix newOrientation = new Matrix(3, 3);
            newOrientation.setElements(new double[] {-1, 0, 0,   0, 1, 0,   0, 0, -1});
            objectList[0].setOrigin(newOrigin);
            objectList[0].setOrientation(newOrientation);
            this.repaint();
```

```
}

/**
 * Positions and orientates the camera to face directly right at the same distance from the origin as before
 */
public void setViewLeft() {
	Vector newOrigin = new Vector(3);
	newOrigin.setElement(0, -objectList[0].getOrigin().modulus());
	Matrix newOrientation = new Matrix(3, 3);
	newOrientation.setElements(new double[] {0, 0, -1,   0, 1, 0,   1, 0, 0});
	objectList[0].setOrigin(newOrigin);
	objectList[0].setOrientation(newOrientation);
	this.repaint();
}

/**
 * Positions and orientates the camera to face directly left at the same distance from the origin as before
 */
public void setViewRight() {
	Vector newOrigin = new Vector(3);
	newOrigin.setElement(0, objectList[0].getOrigin().modulus());
	Matrix newOrientation = new Matrix(3, 3);
	newOrientation.setElements(new double[] {0, 0, 1,   0, 1, 0,   -1, 0, 0});
	objectList[0].setOrigin(newOrigin);
	objectList[0].setOrientation(newOrientation);
	this.repaint();
}

/**
 * Positions and orientates the camera to face directly down at the same distance from the origin as before
 */
public void setViewTop() {
	Vector newOrigin = new Vector(3);
	newOrigin.setElement(1, objectList[0].getOrigin().modulus());
	Matrix newOrientation = new Matrix(3, 3);
	newOrientation.setElements(new double[] {1, 0, 0,   0, 0, 1,   0, -1, 0});
	objectList[0].setOrigin(newOrigin);
	objectList[0].setOrientation(newOrientation);
	this.repaint();
```

```
	}

	/**
	 * Positions and orientates the camera to face directly up at the same distance from the origin as before
	 */
	public void setViewBottom() {
		Vector newOrigin = new Vector(3);
		newOrigin.setElement(1, -objectList[0].getOrigin().modulus());
		Matrix newOrientation = new Matrix(3, 3);
		newOrientation.setElements(new double[] {1, 0, 0,   0, 0, -1,   0, 1, 0});
		objectList[0].setOrigin(newOrigin);
		objectList[0].setOrientation(newOrientation);
		this.repaint();
	}

	/**
	 * Saves the contents of the viewport to a bitmapped image file with path outputFile and of type expressed by
fileExtension
	 * @param outputFile the path (including name with file extension) to save the image under
	 * @param fileExtension one of "gif", "jpg" and "png" which indicates the format for the file
	 */
	public void saveImage(File outputFile, String fileExtension) {
		System.out.println(outputFile.getName());
		BufferedImage image = new BufferedImage(frameWidth, frameHeight, BufferedImage.TYPE_INT_RGB);
		paint(image.getGraphics()); // Run paintComponent, but drawing to image rather than the viewport
		try {
			ImageIO.write(image, fileExtension, outputFile); // Write the image file
		} catch (IOException e) {
			JOptionPane.showMessageDialog(SwingUtilities.windowForComponent(this), "There was a problem saving
the image; you might not have space to save the image.");
		}
	}
}
```

3.5 ViewportListener.java

```
package RefractionSim;
import java.awt.event.*;
import javax.swing.SwingUtilities;

/**
 * Class for all-purpose viewport listeners handling both mouse and keyboard events
 * @author William Platt
 *
 */
public class ViewportListener implements MouseListener, MouseMotionListener, MouseWheelListener, KeyListener {
	private boolean dragging;
	private int prevX;
	private int prevY;

	/**
	 * Called when any of the mouse buttons are depressed in the viewport to give the viewport focus and prepare
for dragging
	 */
	public void mousePressed(MouseEvent event) {
		dragging = true;
		// Store the co-ordinates of the cursor so that the change in position can be determined if the user
drags
		prevX = event.getX();
		prevY = event.getY();
		Viewport viewport = (Viewport)(event.getSource());
		viewport.requestFocusInWindow();
	}

	/**
	 * Called when the user drags after depressing any of the mouse buttons in the viewport to orbit the camera
around the world's origin or zoom if the right mouse button is used
	 */
	public void mouseDragged(MouseEvent event) {
		if (dragging) {
			Viewport viewport = (Viewport)(event.getSource());
			int newX = event.getX();
			int newY = event.getY();
			if (SwingUtilities.isRightMouseButton(event)) {
```

```
                        viewport.zoom((newY - prevY) * 0.01);
                  } else {
                        viewport.orbit(newX - prevX, newY - prevY);
                  }
                  // Store the new co-ordinates so that the change in position can be calculated again if the user
keeps dragging
                  prevX = newX;
                  prevY = newY;
            }
      }

      /**
       * Called when any mouse button is released to end a drag
       */
      public void mouseReleased(MouseEvent event) {
            dragging = false;
      }

      /**
       * Called when a mouse button is released without having moved since it was depressed to change the selection
if the left mouse button was used
       */
      public void mouseClicked(MouseEvent event) {
            if (SwingUtilities.isLeftMouseButton(event)) {
                  Viewport viewport = (Viewport)(event.getSource());
                  viewport.click(event.getX(), event.getY());
            }
      }

      /**
       * Called when the user scrolls with the mouse wheel (middle mouse button) to move the camera closer to or
further away from the world's origin (zooming)
       */
      public void mouseWheelMoved(MouseWheelEvent event) {
            Viewport viewport = (Viewport)(event.getSource());
            viewport.zoom(event.getWheelRotation());
      }

      /**
```

```
	 * Called when a key on the keyboard is released or held down to set the camera position and orientation or
toggle between orthographic and perspective projection
	 */
	public void keyReleased(KeyEvent event) {
		int key = event.getKeyCode();
		Viewport viewport = (Viewport)(event.getSource());
		switch (key) {
			case KeyEvent.VK_1:
				viewport.setViewFront();
				break;
			case KeyEvent.VK_2:
				viewport.setViewBack();
				break;
			case KeyEvent.VK_3:
				viewport.setViewLeft();
				break;
			case KeyEvent.VK_4:
				viewport.setViewRight();
				break;
			case KeyEvent.VK_5:
				viewport.setViewTop();
				break;
			case KeyEvent.VK_6:
				viewport.setViewBottom();
				break;
			case KeyEvent.VK_P:
				viewport.toggleOrthographic(); // This method also regenerates the menu bar so that the
perspective checkbox is toggled
		}
	}

	// Method definitions required by KeyListener, MouseListener and MouseMotionListener

	/**
	 * Called when a key is pressed; no action is taken
	 */
	public void keyPressed(KeyEvent event) {

	}
```

```
        /**
         * Called when a key is pressed or held such that a character would be typed if a text field was in focus
         */
        public void keyTyped(KeyEvent event) {

        }

        /**
         * Called when the cursor enters the viewport
         */
        public void mouseEntered(MouseEvent event) {

        }

        /**
         * Called when the mouse exits the viewport
         */
        public void mouseExited(MouseEvent event) {

        }

        /**
         * Called when the mouse is moved inside the viewport
         */
        public void mouseMoved(MouseEvent event) {

        }
}
```

3.6 Object3D.java

```
package RefractionSim;
import java.awt.Color;

/**
 * General class for any object that exists in 3-D space such as the camera
 * @author William Platt
 *
 */
public class Object3D {
	protected int ID = -1; // Has no ID until added to the viewport's object list, so is set to -1
	protected Mesh mesh;
	protected Color color;
	protected Matrix orientation = new Matrix(3, 3); // Represents the object to upright transformation
	protected Vector origin = new Vector(3);
	protected Vector[] boxVerts; // Vertices of the arbitrarily orientated bounding box (AOBB)

	/**
	 * Constructor for the Object3D class
	 * @param mesh the geometry of the 3-D object (null if there isn't any)
	 * @param color the colour of the 3-D object (irrelevant if the mesh is null and may also be null in this
case)
	 */
	public Object3D(Mesh mesh, Color color) {
		this.mesh = mesh;
		this.color = color;
		double[] elements = {1, 0, 0,   0, 1, 0,   0, 0, 1}; // 3 by 3 identity matrix
		this.orientation.setElements(elements); // Set the default orientation to that of the world
		if (mesh != null) { // Objects such as the camera may not have a mesh and won't need an AOBB
			boxVerts = mesh.getBoxVerts();
		} else {
			boxVerts = new Vector[0]; // Empty list of Vectors
		}
	}

	/**
	 * Sets the value of ID to newID
	 * @param newID the new value for ID
	 */
```

```
public void setID(int newID) {
	this.ID = newID;
}

/**
 * Returns ID, the index of the object in the viewport's objectList
 * @return the object's ID
 */
public int getID() {
	return this.ID;
}

/**
 * Returns the geometry of the object; null if there is no geometry
 * @return the geometry of the object
 */
public Mesh getMesh() {
	return this.mesh;
}

/**
 * Returns the a list of the vertices of the object's arbitrarily orientated bounding box (AOBB)
 * @return the vertices of the object's bounding box
 */
public Vector[] getBoxVerts() {
	return this.boxVerts;
}

/**
 * Returns the object's colour
 * @return the colour of the object
 */
public Color getColor() {
	return this.color;
}

/**
 * Moves the object in 3-D space by the displacement vector
 * @param displacement 3-row vector representing movement in the world's x, y and z directions
```

```
	 * @throws IllegalArgumentException if displacement is not a 3-row vector
	 */
	public void displace(Vector displacement) {
		if (displacement.getN() > 3) {
			throw new IllegalArgumentException("Displacements must be in three dimensions or fewer");
		} else {
			for (int i = 0; i < displacement.getN(); i++) {
				this.origin.setElement(i, this.origin.getElement(i) + displacement.getElement(i));
			}
		}
	}

	/**
	 * Returns the location of the object's origin which other points are measured relative to in object space
	 * @return the location of the object's origin in world space
	 */
	public Vector getOrigin() {
		return this.origin;
	}

	/**
	 * Moves the object to a new position
	 * @param origin the new position of the object's origin in world space
	 * @throws IllegalArgumentException if origin is not a 3-row vector
	 */
	public void setOrigin(Vector origin) {
		if (origin.getN() != 3) {
			throw new IllegalArgumentException("The origin of an object must be a 3-D vector");
		} else  {
			this.origin = origin;
		}
	}

	/**
	 * Rotates an object about its origin; the vertices' co-ordinates aren't changed, but the object's basis
vectors (columns of the orientation matrix) are changed
	 * @param rotation a 3 by 3 matrix representing the angular displacement of the new orientation from the old
one
```

```
	 * @throws IllegalArgumentException if rotation is not a 3 by 3 matrix; the matrix should also represent a
rotation, although this will not throw an exception
	 */
	public void rotate(Matrix rotation) {
		if ((rotation.getM() != 3) || (rotation.getN() != 3)) {
			throw new IllegalArgumentException("A 3 by 3 matrix is needed to rotate an object");
		} else {
			this.orientation = rotation.multiply(this.orientation);
		}
	}

	/**
	 * Returns the matrix representing the orientation of the object relative to world/upright space
	 * @return the 3 by 3 matrix representing the object's orientation relative to world/upright space
	 */
	public Matrix getOrientation() {
		return this.orientation;
	}

	/**
	 * Sets the orientation of the object to the
	 * @param orientation a 3 by 3 matrix representing an orientation
	 * @throws IllegalArgumentException if the matrix is not 3 by 3; it should also represent a rotation, although
this will not throw an exception
	 */
	public void setOrientation(Matrix orientation) {
		if ((orientation.getM() != 3) && (orientation.getN() != 3)) {
			throw new IllegalArgumentException("The orientation of an object must be a 3 by 3 matrix");
		} else {
			this.orientation = orientation;
		}
	}

	/**
	 * Rotates the object about the origin of world space; the object's origin and rotation are affected. This
method assumes the object to be facing the world's origin
	 * @param heading the angle of rotation clockwise around the world's y-axis
	 * @param pitch the angle of rotation clockwise around the object's x-axis
	 */
```

```
	public void orbit(double heading, double pitch) {
		Matrix verticalRot = new Matrix(3, 3);
		verticalRot.setToRotation(this.orientation.getVector(0), pitch); // Rotation about the object's x-axis
in world space
		Matrix horizontalRot = new Matrix(3, 3);
		Vector verticalAxis = new Vector(3);
		verticalAxis.setElements(new double[] {0, 1, 0}); // World's y-axis
		horizontalRot.setToRotation(verticalAxis, heading);
		// Rotate about the world origin without changing the object's orientation
		setOrigin(horizontalRot.multiply(verticalRot.multiply(this.origin)));
		// Rotate about the object's origin (changing the orientation and not the origin)
		this.rotate(verticalRot);
		this.rotate(horizontalRot);
	}

}
```

3.7 Target.java

```
package RefractionSim;
import java.awt.Color;

/**
 * Class for 3-D objects which act as a transmission medium for light
 * @author William Platt
 *
 */
public class Target extends Object3D {

	private int materialID;
	private String shape;

	/**
	 * Constructor for the Target class
	 * @param shape the shape of the target object from the selection of primitive geometries
	 * @param color the overall colour of the object including transparency
	 * @param materialID the index of the material of the object in the viewport's materials list
	 */
	public Target(Mesh.Primitive shape, Color color, int materialID) {
		super(new Mesh(shape), color); // Call the Object3D constructor
		this.shape = shape.toString(); // Store the name of the material
		this.materialID = materialID; // Store the index that identifies the material of the target
	}

	/**
	 * Returns the index of the target's material in the viewport's list of materials
	 * @return the index of the target's material
	 */
	public int getMaterial() {
		return materialID;
	}

	/**
	 * Sets the material to that referenced by the materialID parameter
	 * @param materialID the index of the material the target is to be changed to
	 */
	public void setMaterial(int materialID) {
```

```
            this.materialID = materialID;
        }

        /**
         * Returns the name of the shape of the target
         * @return the name of the target's shape
         */
        public String getShape() {
            return shape;
        }

}
```

3.8 RayBox.java

```
package RefractionSim;
import java.awt.Color;

/**
 * Class for ray boxes
 * @author William Platt
 *
 */
public class RayBox extends Object3D {

	private String label = "Ray box"; // Default ray box label is "Ray box"
	private Beam lightBeam;
	private boolean localPitchInverted;

	/**
	 * Constructor for the RayBox class that generates the ray box's geometry, sets its colour, position and orientation and creates the light beam with geometry
	 */
	public RayBox() {
		super(new Mesh(Mesh.Primitive.CUBE), new Color(200, 200, 200)); // The ray box is a cube with a light grey object colour
		mesh.scale(0.5, 0.5, 0.5); // Shrink the ray box
		this.lightBeam = new Beam(new Color(200, 20, 20), 0.015); // Set the ray box's light beam as a new Beam with a red colour and radius of 0.015 units (3 in the beam thickness text field)
		Vector newOrigin = new Vector(3);
		newOrigin.setElement(2, -5); // heading: pi, pitch: 0
		this.setOrigin(newOrigin);
		this.orbit((Math.random() - 0.5) * 2 * Math.PI, 0); // Set a random heading
		localPitchInverted = false; // Ray box is not upside down
	}

	/**
	 * Sets the origin of both the ray box and light beam
	 * @param orgin the new origin for the ray box and light beam
	 * @throws IllegalArgumentException if origin is not a 3-row vector
	 */
	@Override
	public void setOrigin(Vector origin) {
```

```
        this.origin = origin;
        lightBeam.setOrigin(origin);
    }

    /**
     * Returns the label of the ray box
     * @return the label of the ray box
     */
    public String getLabel() {
        return this.label;
    }

    /**
     * Sets the ray box's label to the newLabel parameter
     * @param newLabel the new label for the ray box
     */
    public void setLabel(String newLabel) {
        this.label = newLabel;
    }

    /**
     * Returns the light beam for the ray box
     * @return the light beam for the ray box
     */
    public Beam getLightBeam() {
        return lightBeam;
    }

    /**
     * Returns whether angles are set to be visible for the ray box/light beam
     * @return whether or not angle visibility is on
     */
    public boolean getAnglesVisible() {
        return lightBeam.getAnglesVisible();
    }

    /**
     * Sets the visibility of angles for the ray box/light beam
     * @param showAngles whether or not angles should be displayed for the ray box/light beam
```

```
	 */
	public void setAnglesVisible(boolean showAngles) {
		lightBeam.setAnglesVisible(showAngles);
	}

	/**
	 * Returns the thickness of the ray box's light beam between 1 and 10 inclusive
	 * @return the thickness of the light beam
	 */
	public int getBeamThickness() {
		return (int)(lightBeam.getRadius() * 200);
	}

	/**
	 * Sets the radius of the light beam from a thickness value between 1 and 10 inclusive
	 * @param newThickness
	 */
	public void setBeamThickness(int newThickness) {
		lightBeam.setRadius(newThickness * 0.005);
	}

	/**
	 * Returns whether or not the ray box is upside down, meaning that the effect of a change in the local pitch
slider is negated
	 * @return whether or not the local pitch slider is inverted for the ray box
	 */
	public boolean isLocalPitchInverted() {
		return localPitchInverted;
	}

	/**
	 * Toggles whether or not the local pitch slider is inverted for the ray box
	 */
	public void toggleLocalPitchInverted() {
		localPitchInverted = !localPitchInverted;
	}

	/**
```

```
	 * Rotates both the ray box and light beam about their origins by the angular displacement represented by the rotation matrix
	 * @param rotation a 3 by 3 matrix representing the angular displacement of the new orientation from the old one
	 * @throws IllegalArgumentException if rotation is not a 3 by 3 matrix; the matrix should also represent a rotation, although this will not throw an exception
	 */
	@Override
	public void rotate(Matrix rotation) {
		if ((rotation.getM() != 3) || (rotation.getN() != 3)) {
			throw new IllegalArgumentException("A 3 by 3 matrix is needed to rotate an object");
		} else {
			this.orientation = rotation.multiply(this.orientation);
			lightBeam.setOrientation(this.orientation);
		}
	}

	/**
	 * Rotates both the ray box and light beam about their origins by the heading and pitch angles
	 * @param heading the angle of rotation clockwise about the world's y-axis
	 * @param pitch the angle of rotation about the ray box's x-axis
	 */
	public void rotate(double heading, double pitch) {
		Matrix verticalRot = new Matrix(3, 3);
		verticalRot.setToRotation(this.orientation.getVector(0), pitch);
		Matrix horizontalRot = new Matrix(3, 3);
		Vector verticalAxis = new Vector(3);
		verticalAxis.setElements(new double[] {0, 1, 0});
		horizontalRot.setToRotation(verticalAxis, heading);
		rotate(verticalRot);
		rotate(horizontalRot);
	}

	/**
	 * Rotates both the ray box and light beam about the world's origin by the heading and pitch angles. Unlike the orbit method of Object3D this doesn't assume that the ray box is facing the world's origin
	 * @param heading the angle of rotation clockwise about the world's y-axis
	 * @param pitch the angle of rotation clockwise about the horizontal vector perpendicular to the vector from the origin to the ray box if the ray box had a y co-ordinate of 0
```

```
	 */
	public void orbitAboutOrigin(double heading, double pitch) {
		Matrix verticalRot = new Matrix(3, 3);
		Vector pitchAxis = new Vector(3);
		pitchAxis.setElements(new double[] {origin.getElement(0), 0, origin.getElement(2)}); // Vector from the world's origin to the ray box's origin if the ray box was in the horizontal plane
		verticalRot.setElements(new double[] {0, 0, 1,   0, 1, 0,   -1, 0, 0}); // Represents a rotation pi/2 radians anticlockwise about the world's y-axis
		pitchAxis = verticalRot.multiply(pitchAxis); // Horizontal vector perpendicular to the horizontal vector from the world's origin to the ray box's origin
		verticalRot.setToRotation(pitchAxis, pitch);
		Matrix horizontalRot = new Matrix(3, 3);
		Vector verticalAxis = new Vector(3);
		verticalAxis.setElements(new double[] {0, 1, 0}); // World's y-axis
		horizontalRot.setToRotation(verticalAxis, heading);
		setOrigin(horizontalRot.multiply(verticalRot.multiply(this.origin)));
		this.rotate(verticalRot);
		this.rotate(horizontalRot);
	}

}
```

3.9 Beam.java

```
package RefractionSim;
import java.awt.Color;

/**
 * Class for light beams
 * @author William Platt
 *
 */
public class Beam extends Object3D {

	private Vector[] points = new Vector[100];
	private int numOfPoints = 0;
	private double[] angles = new double[196]; // 2 angles for every point other than the first and last
	private Vector[] anglePositions = new Vector[196]; // Position in 3-D space at which to write angles when displaying them in the viewport
	private int numOfAngles = 0;
	private double radius;
	private boolean anglesVisible;

	/**
	 * Constructor for the Beam class that sets its colour, radius, default position and orientation, and sets angles to be displayed in the viewport
	 * @param color the colour of the light beam (the beam will be a solid colour and not have shading to imitate lighting)
	 * @param radius half of the width of the square cross-section of the beam
	 */
	public Beam(Color color, double radius) {
		super(null, color); // Create the light beam as a 3-D object with no geometry/mesh
		this.orientation = new Matrix(3, 3); // heading: 0, pitch: 0
		this.origin = new Vector(3); // [0, 0, 0]
		this.radius = radius;
		this.anglesVisible = true;
	}

	/**
	 * Sets the colour of the beam to newColor
	 * @param newColor the new colour for the beam
	 */
```

```
	public void setColor(Color newColor) { // getColor is already defined by the parent class
		this.color = newColor;
	}

	/**
	 * Recalculates the beam's path and regenerates its geometry
	 */
	public void update() {
		calculateRays();
		generateMesh();
	}

	/**
	 * Returns half of the width of the square cross-section of the beam
	 * @return half of the width of the square cross-section of the beam
	 */
	public double getRadius() {
		return radius;
	}

	/**
	 * Sets the width of the square cross-section of the beam to be 2 * newRadius and regenerates the geometry of
the beam
	 * @param newRadius half of the width of the new square cross-section of the beam
	 */
	public void setRadius(double newRadius) {
		radius = newRadius;
		generateMesh();
	}

	/**
	 * Returns whether or not angles are set to be visible for the light beam
	 * @return if angles are to be visible
	 */
	public boolean getAnglesVisible() {
		return anglesVisible;
	}

	/**
```

```
	 * Sets whether or not angles should be displayed for the light beam
	 * @param showAngles if angles are to be visible
	 */
	public void setAnglesVisible(boolean showAngles) {
		anglesVisible = showAngles;
	}

	/**
	 * Returns the list of angles in order from the ray box
	 * @return the list of angles between the beam and the surface normals in order from the ray box
	 */
	public double[] getAngles() {
		return angles;
	}

	/**
	 * Returns the list of positions in 3-D space for each of the angles in order from the ray box
	 * @return the list of positions for the angles in the same order as the list of angles
	 */
	public Vector[] getAnglePositions() {
		return anglePositions;
	}

	/**
	 * Returns the number of angles in the list of angles
	 * @return the number of angles in the list of angles
	 */
	public int getNumOfAngles() {
		return numOfAngles;
	}

	/**
	 * Generates the geometry of the beam if the rays have been calculated
	 */
	private void generateMesh() {
		int[][] faces = new int[(numOfPoints - 1) * 8][3]; // Between every square of vertices (every point)
there are 4 square surfaces each comprised of 2 triangular faces
		Vector[] verts = new Vector[4 * numOfPoints]; // There is a square of vertices at each point
		Vector displace0 = new Vector(3);
```

```
            displace0.setElements(new double[] {-radius, radius, 0}); // Displacement of the top left vertex of a
square from the point where the ray and face intersect
            Vector displace1 = new Vector(3);
            displace1.setElements(new double[] {radius, radius, 0}); // Displacement of the top right vertex
            int j = 0; // Vertex counter
            int k = 0; // Face counter
            Matrix rotation = orientation.transpose();
            for (int i = 0; i < numOfPoints; i++) {
                  Vector centerPoint = rotation.multiply((points[i].subtract(origin))); // Map point from world
space to object space
                  verts[j] = centerPoint.add(displace0); // Create top left vertex
                  verts[j+1] = centerPoint.add(displace1); // Create top right vertex
                  verts[j+2] = centerPoint.subtract(displace0); // Create bottom right vertex
                  verts[j+3] = centerPoint.subtract(displace1); // Create bottom left vertex
                  if (i > 0) { // Create faces between the vertices just created and the last square of vertices
                        // For each face, list vertices in clockwise order when looking at the face from outside of
the beam
                        faces[k] = new int[] {j-4, j, j+1};
                        faces[k+1] = new int[] {j-4, j+1, j-3};

                        faces[k+2] = new int[] {j-3, j+1, j+2};
                        faces[k+3] = new int[] {j-3, j+2, j-2};

                        faces[k+4] = new int[] {j-2, j+2, j+3};
                        faces[k+5] = new int[] {j-2, j+3, j-1};

                        faces[k+6] = new int[] {j-1, j+3, j};
                        faces[k+7] = new int[] {j-1, j, j-4};
                        k += 8;
                  }
                  j += 4;
            }
            this.mesh = new Mesh(faces, verts); // Set the faces and vertices as the mesh of the beam so that it can
be rendered
            this.boxVerts = this.mesh.getBoxVerts(); // Store the vertices of the bounding box for the new mesh
      }

      /**
```

```
	 * Calculates the path of the beam as a sequence of rays and stores the points in 3-D space where the path switches between rays and the angles of rays to surface normals
	 */
	private void calculateRays() {
		numOfPoints = 0;
		numOfAngles = 0;
		Vector p = this.origin; // Starting point of the beam in world space
		Vector v = this.orientation.getVector(2); // Initial direction of the beam in world space
		Ray currentRay = new Ray(p, v);
		Target target = (Target)(Viewport.getObjectList()[1]);
		double[] refractiveIndices = Viewport.getRefractiveIndices();
		double targetIndexRelToWorld = refractiveIndices[target.getMaterial()] / refractiveIndices[Viewport.getWorldMaterial()];
		double criticalAngle;
		if (targetIndexRelToWorld > 1) {
			criticalAngle = Math.asin(1 / targetIndexRelToWorld);
		} else {
			criticalAngle = Math.asin(targetIndexRelToWorld);
		}
		int i = 0;
		do {
			points[i] = currentRay.getP(); // Store the starting point of the ray
			v = currentRay.getV();
			currentRay = calcNextRay(currentRay, targetIndexRelToWorld, criticalAngle); // Calculate the next ray based on the current one. Null is returned if the current ray doesn't intersect any faces of the target object
			i++;
		} while ((currentRay != null) && (i < points.length - 1)); // Repeat until the beam carries on to infinity without hitting a boundary between media or no more points can be stored (given that one more point is added after this loop)
		if (i == 1) { // The beam never hit the target object
			points[i] = points[i - 1].add(v.scale(10)); // Continue the beam in along its original line for 10 units
		} else {
			points[i] = points[i - 1].add(v.scale(8)); // Continue the beam along the line of the last ray for 8 units
		}
		numOfPoints = i + 1; // i started at 0
	}
```

```
      /**
       * Calculates and returns the next ray of the beam based on the intersection of the current ray and the target
object
       * @param incidentRay the last ray that was calculated
       * @param targetIndexRelToWorld the refractive index of the target material relative to the world
       * @param criticalAngle the minimum angle from the normal needed for total internal reflection within the
denser material
       * @return the next ray which the beam follows
       */
      private Ray calcNextRay(Ray incidentRay, double targetIndexRelToWorld, double criticalAngle) {
            Target target = (Target)(Viewport.getObjectList()[1]);
            Mesh mesh = target.getMesh();
            int[][] faces = mesh.getFaces();
            Vector[] verts = mesh.getVerts();
            Vector[] normals = mesh.getNormals();
            double[] ds = mesh.getDs(); // The equation of a plane is p.n = d where p is a point in the plane and n
is the normal to the plane; ds is a list containing the value of d for each face
            Vector p = incidentRay.getP(); // The starting point of the current ray
            Vector v = incidentRay.getV(); // The direction of the current ray
            Vector n;
            double d;
            double vMultiple = -1; // The displacement of the closest point of intersection so far from p in terms
of v; it remains -1 until an intersection is found
            int faceIntersected = -1; // The face of the target that is first intersected by the current ray; it
remains -1 until an intersection is found
            Vector finalPoint = new Vector(3);
            for (int i = 0; i < faces.length; i++) { // Iterate through all of the target's faces and check if the
current ray intersects the face
                  n = normals[i]; // Normal for the current face
                  d = ds[i]; // Value of d for the current face
                  double nDotP = n.dotProduct(p);
                  double nDotV = n.dotProduct(v); // The compononet of v in the direction of n
                  // Go to the next face if the value of lambda will not be positive (meaning the face is in the
opposite direction to v from p)
                  if (nDotP < d) {
                        if (nDotV <= 0) {
                              continue;
                        }
                  } else if (nDotP == d) {
```

```
                    continue;
                } else if (nDotV >= 0) {
                    continue;
                }
                double lambda = (d - nDotP) / nDotV; // d - nDotP gives the shortest distance from p to the plane,
so lambda is the number of times p must be displaced by v to be in the plane of the face
                if (lambda > 0.0001) { // Prevent a beam interacting with the same face twice consecutively due to
floating point error
                    if ((vMultiple == -1) || (lambda < vMultiple)) { // If there have been no intersections so
far or the distance to the plane in terms of v is less than the shortest found so far
                        // The three edges of the face
                        Edge2D edge0;
                        Edge2D edge1;
                        Edge2D edge2;
                        double intersectX;
                        double intersectY;
                        Vector pointInPlane = p.add(v.scale(lambda)); // Point where the current ray
intersects the face
                        // Orthographicallly project the plane into 2-D
                        if (n.getElement(2) == 0) { // The face is not tilted forwards or backwards
                            if (n.getElement(0) == 0) { // The face is horizontal, so x and z co-ordinates
can be used without the vertices becoming colinear in two dimensions
                                // In this projection, x co-ordinates remain x co-ordinates and z co-
ordinates become y co-ordinates
                                edge0 = new Edge2D(verts[faces[i][0]].getElement(0),
verts[faces[i][0]].getElement(2), verts[faces[i][1]].getElement(0), verts[faces[i][1]].getElement(2));
                                edge1 = new Edge2D(verts[faces[i][1]].getElement(0),
verts[faces[i][1]].getElement(2), verts[faces[i][2]].getElement(0), verts[faces[i][2]].getElement(2));
                                edge2 = new Edge2D(verts[faces[i][2]].getElement(0),
verts[faces[i][2]].getElement(2), verts[faces[i][0]].getElement(0), verts[faces[i][0]].getElement(2));
                                intersectX = pointInPlane.getElement(0); // x --> x
                                intersectY = pointInPlane.getElement(2); // z --> y
                            } else { // The face is not titled forwards/backwards but is tilted left/right,
so z and y co-ordinates can be used
                                // z --> x, y --> y
                                edge0 = new Edge2D(verts[faces[i][0]].getElement(2),
verts[faces[i][0]].getElement(1), verts[faces[i][1]].getElement(2), verts[faces[i][1]].getElement(1));
                                edge1 = new Edge2D(verts[faces[i][1]].getElement(2),
verts[faces[i][1]].getElement(1), verts[faces[i][2]].getElement(2), verts[faces[i][2]].getElement(1));
```

```
                                        edge2 = new Edge2D(verts[faces[i][2]].getElement(2),
verts[faces[i][2]].getElement(1), verts[faces[i][0]].getElement(2), verts[faces[i][0]].getElement(1));
                                        intersectX = pointInPlane.getElement(2); // z --> x
                                        intersectY = pointInPlane.getElement(1); // y --> y
                                  }
                            } else { // We can use x and y co-ordinates and the vertices won't become colinear
                                  // x --> x, y --> y
                                  edge0 = new Edge2D(verts[faces[i][0]].getElement(0),
verts[faces[i][0]].getElement(1), verts[faces[i][1]].getElement(0), verts[faces[i][1]].getElement(1));
                                  edge1 = new Edge2D(verts[faces[i][1]].getElement(0),
verts[faces[i][1]].getElement(1), verts[faces[i][2]].getElement(0), verts[faces[i][2]].getElement(1));
                                  edge2 = new Edge2D(verts[faces[i][2]].getElement(0),
verts[faces[i][2]].getElement(1), verts[faces[i][0]].getElement(0), verts[faces[i][0]].getElement(1));
                                  intersectX = pointInPlane.getElement(0); // x --> x
                                  intersectY = pointInPlane.getElement(1); // y --> y
                            }
                            // Find the tallest of the edges (greatest change in y)
                            Edge2D tallEdge = edge0;
                            Edge2D shortEdge0 = edge1;
                            Edge2D shortEdge1 = edge2;
                            if (edge1.getHeight() > tallEdge.getHeight()) {
                                  tallEdge = edge1;
                                  shortEdge0 = edge0;
                            }
                            if (edge2.getHeight() > tallEdge.getHeight()) {
                                  tallEdge = edge2;
                                  shortEdge0 = edge0;
                                  shortEdge1 = edge1;
                            }
                            if ((tallEdge.getY0() != shortEdge0.getY0()) || (tallEdge.getX0() !=
shortEdge0.getX0())) { // shortEdge0 should share the lowest point with tallEdge
                                  if ((tallEdge.getY0() == shortEdge0.getY0()) && (tallEdge.getX0() ==
shortEdge0.getX1())){

                                  } else {
                                        Edge2D temp = shortEdge0;
                                        shortEdge0 = shortEdge1;
                                        shortEdge1 = temp;
                                  }
```

```
                        }

                        boolean pointInFace = false;
                        if (intersectY > shortEdge0.getY1()) { // The ray passes through the top part of the
face if at all
                              // Check that the intersection of the ray and plane is between the long edge and
top edge
                              double dxTall = (tallEdge.getX1() - tallEdge.getX0()) / tallEdge.getHeight(); //
Increase in x for the tallest edge when y increases by 1
                              double dxShort = (shortEdge1.getX1() - shortEdge1.getX0()) /
shortEdge1.getHeight(); // Increase in x for shortEdge1 when y increases by 1
                              double yFromTop = intersectY - tallEdge.getY1(); // Change in y from the top of
the intersection to the top of the face
                              double xTall = yFromTop * dxTall + tallEdge.getX1(); // x co-ordinate of the
tallest edge when its y co-ordinate is that of the point of intersection
                              double xShort = yFromTop * dxShort + tallEdge.getX1(); // x co-ordinate of
shortEdge1 when its y co-ordinate is that of the point of intersection
                              if (dxTall > dxShort) { // If shortEdge1 has a greater gradient than tallEdge,
meaning that shortEdge1 is always to the right of tallEdge within the top part of the face
                                    if ((intersectX <= xShort) && (intersectX >= xTall)) { // If the point of
intersection is between the two edges in the top part of the face
                                          pointInFace = true;
                                    }
                              } else if ((intersectX >= xShort) && (intersectX <= xTall)) { // If the point of
intersection is between the two edges in the top part of the face
                                    pointInFace = true;
                              }
                        } else { // The ray passes through the bottom part of the face if at all
                              // Check that the intersection of the ray and plane is between the long edge and
bottom edge
                              double dxTall = (tallEdge.getX1() - tallEdge.getX0()) / tallEdge.getHeight(); //
Increase in x for the tallest edge when y increases by 1
                              double dxShort = (shortEdge0.getX1() - shortEdge0.getX0()) /
shortEdge0.getHeight();
                              double yFromBottom = intersectY - tallEdge.getY0();
                              double xTall = yFromBottom * dxTall + tallEdge.getX0();
                              double xShort = yFromBottom * dxShort + tallEdge.getX0();
                              if (dxShort > dxTall) { // If tallEdge is always right of shortEdge in the
bottom part of the face
```

```
                                        if ((intersectX <= xShort) && (intersectX >= xTall)) { // If the point of
intersection is in the bottom part of the face
                                              pointInFace = true;
                                        }
                                  } else if ((intersectX >= xShort) && (intersectX <= xTall)) {
                                        pointInFace = true;
                                  }
                            }
                            if (pointInFace) {
                                  vMultiple = lambda; // Update the shortest distance (multiple of v) so far
                                  faceIntersected = i; // Update the closest face intersected so far
                                  // Update the point of intersection with the closest face so far
                                  finalPoint.setElement(0, pointInPlane.getElement(0));
                                  finalPoint.setElement(1, pointInPlane.getElement(1));
                                  finalPoint.setElement(2, pointInPlane.getElement(2));
                            }
                      }
                }
          }
          if (faceIntersected == -1) { // If the ray didn't intersect any faces
                return null;
          } else {
                return new Ray(finalPoint, nextVector(v, normals[faceIntersected], targetIndexRelToWorld,
criticalAngle, finalPoint)); // Calculate the direction of the next ray and return create a new ray which the method
will return
          }
    }

    /**
     * Calculates the direction of the next ray and returns the result as a normalised 3-row vector
     * @param vector the direction of the current ray
     * @param normal the vector perpendicular to the face being intersected
     * @param targetIndexRelToWorld the refractive index of the target material relative to the world
     * @param criticalAngle the minimum angle between vector and normal within the denser material that would
cause total internal reflection
     * @param intersection the point of intersection with the face in world space
     * @return the direction (as a unit vector) of the next ray after the one with direction vector that
intersects the face with normal normal at the point intersection
     */
```

```
	private Vector nextVector(Vector vector, Vector normal, double targetIndexRelToWorld, double criticalAngle,
Vector intersection) {
		vector = vector.normalise(); // Ensure that vector has a length of 1
		Vector incidentAnglePosition;
		normal = normal.normalise(); // Ensure that the normal to the face has unit length
		double vDotN = vector.dotProduct(normal); // v.n = |v||n|cos(x) where x is the angle between v and n.
|v| and |n| are both 1 in this case, so v.n = cos(x)
		if (vDotN == 0) { // If the current ray is perpendicular to the face (in the plane of the face)
			return vector; // Treat the ray as not intersecting the face
		} else {
			if (vDotN > 0) { // If the angle between v and n is less than pi/2 radians
				incidentAnglePosition = intersection.subtract(Math2.midpoint(vector.scale(0.3),
normal.scale(0.3))); // Set the position of the angle of incidence halfway between vector and normal at about 0.3
units from the point of intersection
			} else {
				incidentAnglePosition = intersection.subtract(Math2.midpoint(vector.scale(0.3),
normal.scale(-0.3))); // Use a negative scale factor for normal because it is facing the opposite way to vector
			}
			// There is a 2-D plane containing both vector and normal; working in 2-D is simpler than 3-D.
Derive the matrix to rotate the vectors so that they lie horizontally (in the plane y = 0; each vector has its tail
at the origin)
			Vector p2 = vector.scale(-1); // Negations such as these are used to ensure the new co-ordinate
system is left-handed so that the resulting matrix doesn't represent reflection as well as rotation
			Matrix rotateVecsToYEquals0 = new Matrix(3, 3);
			Vector xBasis = normal.scale(-1); // In the co-ordinate system where vector and normal are in the
plane y = 0, the x-axis is -normal
			rotateVecsToYEquals0.setElement(0, 0, xBasis.getElement(0));
			rotateVecsToYEquals0.setElement(0, 1, xBasis.getElement(1));
			rotateVecsToYEquals0.setElement(0, 2, xBasis.getElement(2));
			Vector yBasis = p2.crossProduct(normal).normalise(); // Use the cross product on two points in the
plane to give a vector perpendicular to y = 0, the y-axis
			rotateVecsToYEquals0.setElement(1, 0, yBasis.getElement(0));
			rotateVecsToYEquals0.setElement(1, 1, yBasis.getElement(1));
			rotateVecsToYEquals0.setElement(1, 2, yBasis.getElement(2));
			Vector zBasis = yBasis.scale(-1).crossProduct(xBasis).normalise(); // Use the cross-product on two
points in the plane z = 0 to give the z-axis
			rotateVecsToYEquals0.setElement(2, 0, zBasis.getElement(0));
			rotateVecsToYEquals0.setElement(2, 1, zBasis.getElement(1));
			rotateVecsToYEquals0.setElement(2, 2, zBasis.getElement(2));
```

```
                rotateVecsToYEquals0 = rotateVecsToYEquals0.transpose(); // This inverts an orthogonal matrix such
as this. Before doing this the matrix of basis vectors for the new co-ordinate space converts points in this co-
ordinate space to world space; we want the reverse

                double angle = Math.acos(vDotN); // The angle between vector and normal because they were both
normalised
                if (angle > Math.PI / 2) { // If the vectors were in opposite directions (vDotN < 0), then we will
have the larger of the two angles between them
                      angle = Math.PI - angle; // This gives the desired angle because angles on a straight line
sum to pi/2 radians
                }
                addAngle(angle, incidentAnglePosition); // Add an angle to the list of angles and a position to
the list of angle positions

                vector = rotateVecsToYEquals0.multiply(vector); // Map vector to the new co-ordinate system where
it is in the plane y = 0. normal doesn't need mapping because it is along the x-axis
                if (vDotN < 0) { // World to target transition

                      if (targetIndexRelToWorld > 1) { // Target is the denser material
                            vector = refract(vector, targetIndexRelToWorld); // Calculate the next (normalised)
vector in the new co-ordinate system
                      } else { // World is the denser material
                            if (angle >= criticalAngle) {
                                  vector.setElement(0, -vector.getElement(0)); // Reflect the vector
                            } else {
                                  vector = refract(vector, targetIndexRelToWorld);
                            }
                      }
                } else { // Target to world transition
                      if (targetIndexRelToWorld > 1) { // Target is the denser material
                            if (angle >= criticalAngle) {
                                  vector.setElement(0, -vector.getElement(0)); // Reflect the vector
                            } else {
                                  vector = refract(vector, 1 / targetIndexRelToWorld); // Pass the refractive
index of the world relative to the target material
                            }
                      } else {
                            vector = refract(vector, 1 / targetIndexRelToWorld); // Pass the refractive index of
the world relative to the target material
```

```
                        }
                  }
                  vector = rotateVecsToYEquals0.transpose().multiply(vector); // Map the next vector from the new
co-ordinate space to world space
            }
            vDotN = vector.dotProduct(normal); // Cosine of the angle between the new vector and the normal
            Vector finalAnglePosition;
            // Calculate the position of the angle of refraction similarly to incidentAnglePosition
            if (vDotN > 0) {
                  finalAnglePosition = intersection.add(Math2.midpoint(vector.scale(0.3), normal.scale(0.3)));
            } else {
                  finalAnglePosition = intersection.add(Math2.midpoint(vector.scale(0.3), normal.scale(-0.3)));
            }
            double angle = Math.acos(vDotN);
            if (angle > Math.PI / 2) { // Find the smaller of the two angles between normal and the new vector
                  angle = Math.PI - angle;
            }
            addAngle(angle, finalAnglePosition); // Store the angle and its position in 3-D space
            return vector; // Reurn the new vector
      }

      /**
       * Calculates and returns the vector produced by the refraction of incidentVector passing from material A to
material B where refractiveIndex is the refractive index of material B relative to material A
       * @param incidentVector the direction of the previous ray rotated into the plane y = 0 with x-axis -normal
       * @param refractiveIndex the refractive index of the destination material relative to the source material
       * @return the direction of the next ray
       */
      private Vector refract(Vector incidentVector, double refractiveIndex) { // Take the x-axis to be the normal
and all y-values should be zero
            // Snell's law: refractive index = sin(i) / sin(r)
            double sinR = incidentVector.getElement(2) / refractiveIndex; // getElement(2) represents the component
of incidentVector perpendicular to the normal (sin(i)). sinR represents the component of the new vector
perpendicular to the normal
            double cosR = Math.cos(Math.asin(sinR)); // The component of the new vector parallel to the normal
            Vector refractedVector = new Vector(3);
            if (incidentVector.getElement(0) < 0) {
                  refractedVector.setElement(0, -cosR); // If the incident ray was going against the direction of
the normal, the refracted ray will also be more than pi/2 radians from the direction of the normal
```

```
            } else {
                  refractedVector.setElement(0, cosR); // If the incident ray was in a similar direction to the
normal, the refracted ray will also be in a similar direction to the normal
            }
            refractedVector.setElement(2, sinR); // Set the new vector's z component to sinR (z-axis is
perpendicular to x-axis and normal)
            return refractedVector;
      }

      /**
       * Appends an angle to the list of angles and an angle position to the list of angle positions
       * @param angle the angle to append to the list
       * @param position the position in 3-D space of the angle to append to the list
       */
      private void addAngle(double angle, Vector position) {
            angles[numOfAngles] = angle; // ArrayIndexOutOfBoundsException should be avoided because the number of
points is limited
            anglePositions[numOfAngles] = position; // Exception avoided here also
            numOfAngles++;
      }
}
```

3.10 Mesh.java

```
package RefractionSim;

/**
 * Class for the geometries of objects based on the object3D class and its descendant classes
 * @author William Platt
 *
 */
public class Mesh {

	private int[][] faces; // All faces are triangles; the first index defines a triangle and the second defines a vertex number
	private Vector[] verts; // Each vector in this array has 3 rows
	private Vector[] normals;
	private double[] ds; // ds[i] is the d value for faces[i] where d = p.n (n is the normal to a plane and p is a point in that plane)
	private Vector[] boxVerts; // A list of the vertices for the smallest box that will contain all of the mesh's vertices (the box is aligned to the object space axes)

	/**
	 * An enumerated type that specifies the shapes for which the Mesh class can generate geometry
	 * @author William Platt
	 *
	 */
	public enum Primitive {
		// Calls to Primitive(shape) passing a String for shape
		CUBE("Cube"), CUBOID("Cuboid"), TRIANGULAR_PRISM("Triangular prism"), SPHERE("Sphere"), CONVEX_LENS("Convex lens"), CONCAVE_LENS("Concave lens"), HALF_CYLINDER("Half-cylinder");

		private String shape;

		/**
		 * Stores the user-friendly name of the shape
		 * @param shape the string representation of the shape
		 */
		private Primitive(String shape) {
			this.shape = shape;
		}
```

```
		/**
		 * Returns the user-friendly name of the shape
		 * @return the string representation of the shape
		 */
		public String toString() {
			return shape;
		}

	}

	/**
	 * A constructor for the mesh class for non-primitive geometries
	 * @param faces a list of the faces of the object; each item/face contains 3 items which are the indices of
the vertices in the verts list that make up the face
	 * @param verts a list of vertices; each vertex is represented by a position in 3-D space
	 */
	public Mesh(int[][] faces, Vector[] verts) { // Will be read in from a file
		this.faces = faces;
		this.verts = verts;
		normals = new Vector[faces.length]; // One normal for each face
		ds = new double[faces.length]; // One d value for each face
		for (int i = 0; i < faces.length; i++) {
			normals[i] = this.normal(faces[i]); // Calculate the normal to the face
			ds[i] = verts[faces[i][0]].dotProduct(normals[i]); // Calculate the d value for the face
		}
		calcBoxVerts(); // Calculates and stores the vertices of the arbitrarily orientated bounding box
	}

	/**
	 * A constructor for the mesh class for primitive geometries
	 * @param shape the shape represented by the new mesh
	 * @throws IllegalArgumentException if shape is null
	 */
	public Mesh(Primitive shape) {
		switch (shape) {
			case CUBE:
				generateCube();
				break;
			case CUBOID:
```

```
                        generateCube();
                        scale(2, 1, 1); // Stretch the cube along the x-axis to give a cuboid
                        break;
                  case TRIANGULAR_PRISM:
                        generatePrism();
                        break;
                  case SPHERE:
                        generateSphere();
                        break;
                  case CONVEX_LENS:
                        generateSphere();
                        scale(0.6, 2, 2); // Increase the size (to allow a better demonstration of how the lens
works) and then squash the sphere along the x-axis
                        break;
                  case CONCAVE_LENS:
                        generateSphere();
                        scale(0.6, 2, 2); // Increase the size and squash along the x-axis

                        for (int i = 0; i < verts.length; i++) {
                              if (verts[i].getElement(0) < -0.0001) { // Don't move points in the middle of the x-
axis
                                    verts[i].setElement(0, verts[i].getElement(0) + 0.8); // Move points on the left
to the right. This part still bulges out to the left, but when it is on the right it is concave
                              } else if (verts[i].getElement(0) > 0.0001) {
                                    verts[i].setElement(0, verts[i].getElement(0) - 0.8); // Move points on the
right to the left.
                              }
                        }
                        // The faces are 'inside out', so the ordering of the vertices must be reversed in order to
make the normals point the correct way
                        for (int i = 0; i < faces.length; i++) {
                              // Swap the first and last vertices (0 and 2)
                              int temp = faces[i][0];
                              faces[i][0] = faces[i][2];
                              faces[i][2] = temp;
                        }
                        break;
                  case HALF_CYLINDER:
                        generateHalfCylinder();
```

```
                        break;
                  default:
                        throw new IllegalArgumentException("Mesh constructor cannot take a null primitive");
            }
            normals = new Vector[faces.length];
            ds = new double[faces.length];
            for (int i = 0; i < faces.length; i++) {
                  normals[i] = normal(faces[i]); // Calculate the normal for the face
                  ds[i] = verts[faces[i][0]].dotProduct(normals[i]); // Calculate the value of d for the face
            }
            calcBoxVerts(); // Calculate and store the vertices of the AOBB
      }

      /**
       * A constructor for the Mesh class for primitive geometries where the shape needs to be determined from its
name
       * @param shape the name/String representation of the shape
       */
      public Mesh(String shape) {
            this(primitiveFromStr(shape)); // Call the other constructor for primitive geometries
      }

      /**
       * Returns a primitive shape as a Primitive object from its name/String representation
       * @param shape the name of the shape
       * @return the shape represented by the name
       */
      public static Primitive primitiveFromStr(String shape) {
            shape = shape.trim();
            switch (shape) {
                  case "Cube":
                        return Primitive.CUBE;
                  case "Cuboid":
                        return Primitive.CUBOID;
                  case "Triangular prism":
                        return Primitive.TRIANGULAR_PRISM;
                  case "Sphere":
                        return Primitive.SPHERE;
                  case "Convex lens":
```

```
                        return Primitive.CONVEX_LENS;
                  case "Concave lens":
                        return Primitive.CONCAVE_LENS;
                  case "Half-cylinder":
                        return Primitive.HALF_CYLINDER;
                  default:
                        return null;
            }
      }

      /**
       * Creates the vertices and faces that define a cube of side length 2 units
       */
      private void generateCube() {
            verts = new Vector[] {new Vector(3), new Vector(3), new Vector(3), new Vector(3), new Vector(3), new
Vector(3), new Vector(3), new Vector(3)};
            verts[0].setElement(0, -1);
            verts[0].setElement(1, -1);
            verts[0].setElement(2, -1);

            verts[1].setElement(0, 1);
            verts[1].setElement(1, -1);
            verts[1].setElement(2, -1);

            verts[2].setElement(0, -1);
            verts[2].setElement(1, -1);
            verts[2].setElement(2, 1);

            verts[3].setElement(0, 1);
            verts[3].setElement(1, -1);
            verts[3].setElement(2, 1);

            verts[4].setElement(0, -1);
            verts[4].setElement(1, 1);
            verts[4].setElement(2, -1);

            verts[5].setElement(0, 1);
            verts[5].setElement(1, 1);
            verts[5].setElement(2, -1);
```

```
            verts[6].setElement(0, -1);
            verts[6].setElement(1, 1);
            verts[6].setElement(2, 1);

            verts[7].setElement(0, 1);
            verts[7].setElement(1, 1);
            verts[7].setElement(2, 1);

            faces = new int[][] {{0, 3, 2}, {0, 1, 3}, {0, 4, 5}, {0, 5, 1}, {0, 2, 6}, {0, 6, 4}, {2, 7, 6}, {2, 3,
7}, {3, 1, 5}, {3, 5, 7}, {4, 7, 5}, {4, 6, 7}}; // List the vertices of each face by index in verts. Vertices must
be listed in clockwise order from outside of the shape so that the faces pointing away from the camera can be culled
or shaded differently
      }

      /**
       * Creates the vertices and faces that define a triangular prism with sides of length 2
       */
      private void generatePrism() {
            double halfAltitude = Math.sin(Math.PI / 3); // The cross-section is an equilateral triangle with sides
of length 2 units. The altitude is the height of the triangle with one edge horizontal
            verts = new Vector[] {new Vector(3), new Vector(3), new Vector(3), new Vector(3), new Vector(3), new
Vector(3)};
            verts[0].setElements(new double[] {-1, -halfAltitude, -1});
            verts[1].setElements(new double[] {0, halfAltitude, -1});
            verts[2].setElements(new double[] {1, -halfAltitude, -1});
            // Use the same triangle of vertices but offset by 2 units along the z-axis
            verts[3].setElements(verts[0]);
            verts[4].setElements(verts[1]);
            verts[5].setElements(verts[2]);
            verts[3].setElement(2, 1);
            verts[4].setElement(2, 1);
            verts[5].setElement(2, 1);

            faces = new int[][] {{0, 1, 2}, {0, 5, 3}, {0, 2, 5}, {0, 3, 4}, {0, 4, 1}, {1, 4, 5}, {1, 5, 2}, {3, 5,
4}};
      }

      /**
```

```
	 * Creates the vertices and faces that define an approximation of a sphere with radius 1
	 */
	private void generateSphere() {
		int segments = 14;
		int rings = 15; // Use an odd number of rings of faces so that halfway up the sphere is the middle of a ring and not a loop of edges
		verts = new Vector[segments * (rings - 1) + 2]; // There are rings + 1 rings of vertices, but the first and last of these are each a single vertex
		faces = new int[2 * segments * (rings - 1)][3]; // Apart from the first and last, each ring has segments number of square faces, so 2 * segments triangular faces. The first and last each have segments triangular faces
		verts[0] = new Vector(3);
		verts[0].setElement(1, -1); // The lowest point of the sphere
		for (int i = 0; i < segments; i++) {
			if (i == segments - 1) {
				faces[i] = new int[] {0, i + 1, 1}; // The last face involves the last vertex in the second ring and loops back to the first vertex in the second ring
			} else {
				faces[i] = new int[] {0, i + 1, i + 2}; // Triangles involving the lowest vertex and two consecutive vertices in the second ring of vertices
			}
		}
		double pitchIncrement = Math.PI / rings; // The increment in pitch (angle above horizontal) between rings of vertices
		double pitch = pitchIncrement - Math.PI / 2; // The lowest point had a pitch of -pi/2
		double headingIncrement = Math.PI * 2.0 / segments; // The increment in heading between segments
		double heading = -Math.PI;
		for (int r = 0; r < rings - 1; r++) { // Last ring is a single point and must be treated separately
			double y = Math.sin(pitch); // The y co-ordinate for each vertex in this ring
			double modulus = Math.cos(pitch); // The radius of the circle which this ring lies on
			for (int s = 0; s < segments; s++) {
				double x = modulus * Math.cos(heading); // x co-ordinate for the next vertex
				double z = modulus * Math.sin(heading); // z co-ordinate for the next vertex
				verts[segments * r + s + 1] = new Vector(3);
				verts[segments * r + s + 1].setElements(new double[] {x, y, z});
				heading += headingIncrement;
			}
			// Make faces between the vertices just added and the next ring of vertices to be added
			if (r != rings - 2) { // The second to last ring doesn't make faces with the next ring up in the same way because the last ring is a single vertex
```

```
                        for (int i = 0; i < segments; i++) {
                              if (i == segments - 1) { // The last two faces make use of the first vertex in the
next ring by looping back to the start
                                    // Two faces in the same plane
                                    faces[i * 2 + segments * (2 * r + 1)] = new int[] {segments * r + i + 1,
(segments * r + i + 1) + segments, segments * r + 1 + segments};
                                    faces[i * 2 + segments * (2 * r + 1) + 1] = new int[] {segments * r + i + 1,
segments * r + 1 + segments, segments * r + 1};
                              } else {
                                    // Two faces that are in the same plane and appear as a quadrilateral
                                    faces[i * 2 + segments * (2 * r + 1)] = new int[] {segments * r + i + 1,
(segments * r + i + 1) + segments, (segments * r + i + 1) + segments + 1};
                                    faces[i * 2 + segments * (2 * r + 1) + 1] = new int[] {segments * r + i + 1,
(segments * r + i + 1) + segments + 1, (segments * r + i + 1) + 1};
                              }
                        }
                  }
                  pitch += pitchIncrement;
            }
            verts[verts.length - 1] = new Vector(3);
            verts[verts.length - 1].setElement(1, 1); // The last and highest vertex
            for (int i = 0; i < segments; i++) {
                  if (i == segments - 1) { // Last face completes the ring and includes the last vertex of the
second to last ring
                        faces[2 * segments + segments * (2 * rings - 5) + i] = new int[] {segments * (rings - 2) + 1
+ i, segments * (rings - 1) + 1, segments * (rings - 2) + 1};
                  } else { // Faces involving the last vertex and two consecutive vertices in the second to last
ring
                        faces[2 * segments + segments * (2 * rings - 5) + i] = new int[] {segments * (rings - 2) + 1
+ i, segments * (rings - 1) + 1, segments * (rings - 2) + 1 + i + 1};
                  }
            }
      }

      /**
       * Creates the vertices and faces that define the approximation of a cylinder of radius 1 and height 2 that
has been cut in vertically in half
       */
      private void generateHalfCylinder() {
```

```
            int segments = 32;
            verts = new Vector[segments * 2];
            faces = new int[4 * segments - 4][3];
            double heading = 0;
            double headingIncrement = Math.PI / (segments - 1); // The increment in heading between segments of
vertices
            for (int s = 0; s < segments; s++) {
                  double x = Math.cos(heading); // x co-ordinate of points on the segment
                  double z = Math.sin(heading); // z co-ordinate of points on the segment
                  verts[s] = new Vector(3);
                  verts[s].setElements(new double[] {x, -1, z}); // Vertex on the bottom semi-circle
                  verts[s + segments] = new Vector(3);
                  verts[s + segments].setElements(new double[] {x, 1, z}); // Vertex on the top semi-circle
                  heading += headingIncrement;
            }
            for (int i = 0; i < segments - 1; i++) { // Vertical faces approximating the curved surface
                  faces[i * 2] = new int[] {i, i + segments, i + segments + 1}; // Face involving a point on the
bottom semi-circle, the point directly above it (top semi-circle and the same segment) and the point directly above
and one segment across
                  faces[i * 2 + 1] = new int[] {i, i + segments + 1, i + 1}; // Face involving a point on the bottom
semi-circle, the point above and one segment across and the point one segment across on the bottom semi-circle
            }
            for (int i = 0; i < segments - 2; i++) { // Horizontal faces approximating the semi-circles at the top
and bottom
                  faces[segments * 2 - 2 + i] = new int[] {0, i + 1, i + 2}; // For the bottom semi-circle, the
first vertex connected to the (i + 1)th vertex and the (i + 2)th vertex
                  faces[segments * 2 - 2 + i + segments - 2] = new int[] {segments, segments + i + 2, segments + i +
1}; // The same as above but for the top semi-circle
            }
            // Faces representing the vertical square cross-section
            faces[4 * segments - 6] = new int[] {0, segments * 2 - 1, segments}; // The first vertex, the last
vertex and the one above the first
            faces[4 * segments - 5] = new int[] {0, segments - 1, segments * 2 - 1}; // The first vertex, the last
vertex on the bottom and the last vertex (on the top)
      }

      /**
       * Stretches the geometry parallel to the object space axes
       * @param xScale the scale factor of enlargement parallel to the x-axis
```

```
     * @param yScale the scale factor of enlargement parallel to the y-axis
     * @param zScale the scale factor of enlargement parallel to the z-axis
     */
    public void scale(double xScale, double yScale, double zScale) {
          // Create a matrix that will transform vertices to their new positions
          Matrix scaleMatrix = new Matrix(3, 3);
          scaleMatrix.setElement(0, 0, xScale);
          scaleMatrix.setElement(1, 1, yScale);
          scaleMatrix.setElement(2, 2, zScale);
          for (int i = 0; i < verts.length; i++) {
                verts[i] = scaleMatrix.multiply(verts[i]);
          }
    }

    /**
     * Returns a list of the vertices for the mesh's arbitrarily orientated bounding box (AOBB)
     * @return the AOBB vertices for the geometry
     */
    public Vector[] getBoxVerts() {
          return boxVerts;
    }

    /**
     * Calculates the vertices for the mesh's AOBB
     */
    private void calcBoxVerts() {
          if (verts != null) {
                double minX = verts[0].getElement(0);
                double maxX = minX;
                double minY = verts[0].getElement(1);
                double maxY = minY;
                double minZ = verts[0].getElement(2);
                double maxZ = minZ;
                for (int i = 1; i < verts.length; i++) {
                      if (verts[i].getElement(0) < minX) {
                            minX = verts[i].getElement(0);
                      } else if (verts[i].getElement(0) > maxX) {
                            maxX = verts[i].getElement(0);
                      }
```

```
                if (verts[i].getElement(1) < minY) {
                        minY = verts[i].getElement(1);
                } else if (verts[i].getElement(1) > maxY) {
                        maxY = verts[i].getElement(1);
                }
                if (verts[i].getElement(2) < minZ) {
                        minZ = verts[i].getElement(2);
                } else if (verts[i].getElement(2) > maxZ) {
                        maxZ = verts[i].getElement(2);
                }
            }
            Vector[] boxVerts = new Vector[8];
            boxVerts[0] = new Vector(3);
            boxVerts[0].setElements(new double[] {minX, minY, minZ});
            boxVerts[1] = new Vector(3);
            boxVerts[1].setElements(new double[] {maxX, minY, minZ});
            boxVerts[2] = new Vector(3);
            boxVerts[2].setElements(new double[] {minX, minY, maxZ});
            boxVerts[3] = new Vector(3);
            boxVerts[3].setElements(new double[] {maxX, minY, maxZ});
            boxVerts[4] = new Vector(3);
            boxVerts[4].setElements(new double[] {minX, maxY, minZ});
            boxVerts[5] = new Vector(3);
            boxVerts[5].setElements(new double[] {maxX, maxY, minZ});
            boxVerts[6] = new Vector(3);
            boxVerts[6].setElements(new double[] {minX, maxY, maxZ});
            boxVerts[7] = new Vector(3);
            boxVerts[7].setElements(new double[] {maxX, maxY, maxZ});
            this.boxVerts = boxVerts;
        } else {
            this.boxVerts = null;
        }
    }

    /**
     * Returns the list of faces
     * @return the list of faces; each face is a list of 3 integers which are indices for the list of vertices
     */
    public int[][] getFaces() {
```

```
		return this.faces;
	}

	/**
	 * Returns the list of vertices
	 * @return the list of vertices
	 */
	public Vector[] getVerts() {
		return this.verts;
	}

	/**
	 * Returns the list of normals
	 * @return a list of normals corresponding to the faces with the same subscripts
	 */
	public Vector[] getNormals() {
		return this.normals;
	}

	/**
	 * Returns the list of values for d
	 * @return a list of the d values corresponding to the faces with the same indices
	 */
	public double[] getDs() {
		return this.ds;
	}

	/**
	 * Calculates and returns the normalised normal to face; the normal is in the direction the face is 'facing',
which is the direction from which the face's vertices are listed in clockwise order
	 * @param face the face which the normal needs to be calculated for
	 * @return the unit length normal to face
	 */
	private Vector normal(int[] face) {
		Vector point0 = verts[face[0]];
		Vector point1 = verts[face[1]];
		Vector point2 = verts[face[2]];
		return point1.subtract(point0).crossProduct(point2.subtract(point0)).normalise();
	}
```

```
}
```

3.11 Matrix.java

```
package RefractionSim;
/**
 * Class for two-dimensional column-major (each column is a vector) matrices
 * @author William Platt
 *
 */
public class Matrix {
	private int m; // Number of columns
	private int n; // Number of rows
	private double[][] mat; // Elements of the matrix
	private double det; // The determinant of the matrix
	private boolean detKnown; /* States whether the value stored in det is correct; the determinant will change
	                             when the matrix is changed but we don't want to recalculate the
determinant unless
	                             we have to. This will be false for a non-square matrix because it
won't have a
	                             determinant */

	/**
	 * Constructor for the Matrix class which sets all elements in the new matrix to zero
	 * @param m number of columns that the new matrix is to have
	 * @param n number of rows that the new matrix is to have
	 * @throws IllegalArgumentException if either m or n is less than 2 (as this would produce a vector or single
	 * value)
	 * */
	public Matrix(int m, int n) {
		if ((m < 2) || (n < 2)) {
			throw new IllegalArgumentException("Matrix is not two-dimensional");
		} else {
			this.m = m;
			this.n = n;
			this.mat = new double[m][n];
			for (int i = 0; i < m; i++) {
				for (int j = 0; j < n; j++) {
					this.mat[i][j] = 0;
				}
			}
			this.det = 0;
```

```
            if (m == n) {
                  this.detKnown = true;
            }
            this.detKnown = false;
      }
}

/**
 * Returns the number of columns the matrix has
 * @return the number of columns the matrix has
 */
public int getM() {
      return m;
}

/**
 * Returns the number of rows the matrix has
 * @return the number of rows the matrix has
 */
public int getN() {
      return n;
}

/**
 * Sets the value of a single element in the matrix to that of the newValue parameter
 * @param i the column of the element to change (indices start at 0)
 * @param j the row of the element to change (indices start at 0)
 * @param newValue the value which the specified element should be changed to
 * @throws ArrayIndexOutOfBoundsException if i >= number of columns or j >= number of rows
 */
public void setElement(int i, int j, double newValue) {
      if ((i >= this.m) || (j >= this.n) || (i < 0) || (j < 0)) {
            throw new ArrayIndexOutOfBoundsException("Matrix element does not exist");
      } else {
            this.mat[i][j] = newValue;
            this.detKnown = false;
      }
}
```

```
	/**
	 * Sets all elements of the matrix to the values in the newValues array (the array represents the matrix with
	 * columns joined end-to-end)
	 * @param newValues the array of new values that the matrix elements should be set to; the matrix elements are
	 * changed down the columns starting with the leftmost column
	 * @throws IllegalArgumentException if the length of the array is not equal to the number of elements in the
	 * matrix
	 */
	public void setElements(double[] newValues) {
		if (newValues.length != this.m * this.n) {
			throw new IllegalArgumentException("Array argument of different length to number of matrix
elements");
		} else {
			for (int i = 0; i < this.m; i++) {
				for (int j = 0; j < this.n; j++) {
					this.mat[i][j] = newValues[i * this.n + j];
				}
			}
			this.detKnown = false;
		}
	}

	/**
	 * Copies the values of the elements of the matrix newValues to the corresponding elements in the matrix for
	 * which this method is being called (the two matrices then do not reference the same memory locations)
	 * @param newValues the matrix of new values that the matrix elements should be set to
	 * @throws IllegalArgumentException if the newValues matrix does not have the same dimensions as the matrix
for
	 * which this method is being called
	 */
	public void setElements(Matrix newValues) {
		if ((newValues.m != this.m) || (newValues.n != this.n)) {
			throw new IllegalArgumentException("Matrices size mismatch");
		} else {
			for (int i = 0; i < this.m; i++) {
				for (int j = 0; j < this.n; j++) {
					this.mat[i][j] = newValues.mat[i][j];
				}
			}
```

```
                if (newValues.detKnown) {
                        this.detKnown = true;
                        this.det = newValues.det;
                } else {
                        this.detKnown = false;
                }
        }
    }

    /**
     * Sets the elements of the 3 by 3 matrix to represent an a specified 3-D angular displacement (rotation by a
specific amount about a vector in 3-D space)
     * @param axis the 3-D vector/axis about which to rotate
     * @param angle the number of radians by which to rotate
     * @throws IllegalArgumentException if the matrix is not 3 by 3 or if the axis is not a 3-row vector
     */
    public void setToRotation(Vector axis, double angle) {
        if ((this.m != 3) || (this.n != 3)) {
                throw new IllegalArgumentException("A non-3 by 3 matrix cannot be made a rotation matrix");
        } else if (axis.getN() != 3) {
                throw new IllegalArgumentException("A 3-D axis is needed to construct a 3-D rotation matrix");
        } else {
                axis = axis.normalise();
                // Store values needed multiple times so as to reduce the number of calculations
                double x = axis.getElement(0);
                double y = axis.getElement(1);
                double z = axis.getElement(2);
                double cos = Math.cos(angle);
                double sin = Math.sin(angle);
                double oneMinusCos = 1 - cos;
                double xsin = x * sin;
                double ysin = y * sin;
                double zsin = z * sin;
                double xy = x * y * oneMinusCos;
                double xz = x * z * oneMinusCos;
                double yz = y * z * oneMinusCos;
                // Apply the general formula for a 3-D matrix rotating about an arbitrary axis
                this.setElement(0, 0, x * x * oneMinusCos + cos);
                this.setElement(0, 1, xy + zsin);
```

```
            this.setElement(0, 2, xz - ysin);
            this.setElement(1, 0, xy - zsin);
            this.setElement(1, 1, y * y * oneMinusCos + cos);
            this.setElement(1, 2, yz + xsin);
            this.setElement(2, 0, xz + ysin);
            this.setElement(2, 1, yz - xsin);
            this.setElement(2, 2, z * z * oneMinusCos + cos);
      }
}

/**
 * Gets the value of a single element in the matrix
 * @param i the column of the element to return (indices start at 0)
 * @param j the row of the element to return (indices start at 0)
 * @return the value of the specified element
 * @throws ArrayIndexOutOfBoundsException if i >= number of columns or j >= number of rows or i or j < 0
 */
public double getElement(int i, int j) {
      if ((i >= this.m) || (j >= this.n) || (i < 0) || (j < 0)) {
            throw new ArrayIndexOutOfBoundsException("Matrix element does not exist");
      } else {
            return this.mat[i][j];
      }
}

/**
 * Returns the values of all elements in a one-dimensional array where the matrix's columns have been joined
 * end-to-end
 * @return the array of values in the matrix where the columns have been joined end-to-end
 */
public double[] getElements() {
      double[] elements = new double[this.m * this.n];
      for (int i = 0; i < this.m; i++) {
            for (int j = 0; j < this.n; j++) {
                  elements[i * this.n + j] = this.mat[i][j];
            }
      }
      return elements;
}
```

```
	/**
	 * Returns a column of the matrix as a Vector object (this program uses column-vectors)
	 * @param i the index of the column to return as a Vector object (indices start at zero)
	 * @return a Vector object representing the specified column of the matrix
	 * @throws IllegalArgumentException if a the specified column does not exist
	 */
	public Vector getVector(int i) {
		if (i >= this.m) {
			throw new IllegalArgumentException("Matrix column does not exist");
		} else {
			Vector result = new Vector(this.n);
			result.setElements(this.mat[i]);
			return result;
		}
	}

	/**
	 * Returns the sum of this matrix and the toAdd parameter matrix
	 * @param toAdd the two-dimensional matrix to add to the matrix for which this method is being called
	 * @return the sum of this matrix and the matrix passed as a parameter (the returned matrix will be the same
	 * size as both matrices being added)
	 * @throws IllegalArgumentException if the toAdd matrix is not the same size as the matrix for which this
method
	 * is being called
	 */
	public Matrix add(Matrix toAdd) {
		if ((this.m != toAdd.m) || (this.n != toAdd.n)) {
			throw new IllegalArgumentException("Matrices size mismatch for addition");
		} else {
			Matrix result = new Matrix(this.m, this.n);
			for (int i = 0; i < this.m; i++) {
				for (int j = 0; j < this.n; j++) {
					result.setElement(i, j, this.mat[i][j] + toAdd.mat[i][j]);
				}
			}
			return result;
		}
	}
```

```
	/**
	 * Returns A - B where A is the matrix for which this method is called and B is the toSubtract parameter
matrix
	 * @param toSubtract the matrix to subtract from the matrix for which this method is called
	 * @return A - B where A is the matrix for which this method is called and B is the toSubtract parameter
matrix
	 * @throws IllegalArgumentException if the two matrices don't have the same dimensions
	 */
	public Matrix subtract(Matrix toSubtract) {
		if ((this.m != toSubtract.m) || (this.n != toSubtract.n)) {
			throw new IllegalArgumentException("Matrices size mismatch for subtraction");
		} else {
			// Scaling toSubtract by -1 and adding it to `this` is avoided in order to reduce computation time
			Matrix result = new Matrix(this.m, this.n);
			for (int i = 0; i < this.m; i++) {
				for (int j = 0; j < this.n; j++) {
					result.setElement(i, j, this.mat[i][j] - toSubtract.mat[i][j]);
				}
			}
			return result;
		}
	}

	/**
	 * Returns the matrix AB, where A is the matrix for which this method is being called and B is the toMultiply
	 * parameter matrix
	 * @param toMultiply the matrix to post-multiply by (toMultiply is pre-multiplied by the matrix for which this
	 * method is being called)
	 * @return the matrix AB, where A is the matrix for which this method is being called and B is the toMultiply
	 * parameter
	 * @throws IllegalArgumentException if the number of rows in the toMultiply matrix does not equal the number
of
	 * columns in the matrix for which this method is being called
	 */
	public Matrix multiply(Matrix toMultiply) {
		if (this.m != toMultiply.n) {
			throw new IllegalArgumentException("Matrices size mismatch for multiplication");
		} else {
```

```
                Matrix result = new Matrix(toMultiply.m, this.n);
                for (int thisRow = 0; thisRow < this.n; thisRow++) {
                      for (int toMultiplyCol = 0; toMultiplyCol < toMultiply.m; toMultiplyCol++) {
                            double sum = 0;
                            for (int thisCol = 0; thisCol < this.m; thisCol++) {
                                  sum += this.mat[thisCol][thisRow] * toMultiply.mat[toMultiplyCol][thisCol];
                            }
                            result.mat[toMultiplyCol][thisRow] = sum;
                      }
                }
                if ((this.detKnown) && (toMultiply.detKnown)) {
                      result.det = this.det * toMultiply.det;
                } else {
                      result.detKnown = false;
                }
                return result;
          }
    }

    /**
     * Returns the vector AB where A is the matrix for which this method is being called and B is the toMultiply
     * parameter vector
     * @param toMultiply the vector to post-multiply by (toMultiply is pre-multiplied by the matrix for which this
     * method is being called)
     * @return the vector AB, where A is the matrix for which this method is being called and B is the toMultiply
     * parameter
     * @throws IllegalArgumentException if the number of rows in the toMultiply vector does not equal the number
of
     * columns in the matrix for which this method is being called
     */
    public Vector multiply(Vector toMultiply) {
          if (this.m != toMultiply.getN()) {
                throw new IllegalArgumentException("Matrix/Vector size mismatch for multiplication");
          } else {
                Vector result = new Vector(this.n);
                for (int thisRow = 0; thisRow < this.n; thisRow++) {
                      double sum = 0;
                      for (int thisCol = 0; thisCol < this.m; thisCol++) {
                            sum += this.mat[thisCol][thisRow] * toMultiply.getElement(thisCol);
```

```
                        }
                        result.setElement(thisRow, sum);
                  }
                  return result;
            }
      }

      /**
       * Returns the matrix multiplied by a scalar value
       * @param scaleFactor the scalar value by which the matrix is to be multiplied
       * @return the matrix produced when the original matrix is multiplied by scaleFactor
       */
      public Matrix scale(double scaleFactor) {
            Matrix result = new Matrix(this.m, this.n);
            for (int i = 0; i < this.m; i++) {
                  for (int j = 0; j < this.n; j++) {
                        result.mat[i][j] = this.mat[i][j] * scaleFactor;
                  }
            }
            if (this.m == this.n) {
                  if (this.detKnown) {
                        result.det = this.det * Math.pow(scaleFactor, this.m);
                  } else {
                        result.detKnown = false;
                  }
            }
            return result;
      }

      /**
       * Returns the inverse of the matrix (if A is the original matrix and B is the inverse, AB = BA = I where I is
       * the identity matrix (all elements are zero except for the diagonal from top left to bottom right on which
the
       * elements have the value 1)
       * @return the inverse of the matrix
       * @throws Exception if the matrix is not square (the number of columns equals the number of rows if a matrix
       * is square) or the matrix has no inverse
       */
      public Matrix inverse() throws Exception {
```

```
            if (this.m != this.n) {
                  throw new Exception("Non-square matrices have no inverse");
            } else {
                  Matrix inverse = new Matrix(this.m, this.n);
                  double determinant;
                  if (this.m == 2) { // Method for 2 by 2 matrices
                        if (!this.detKnown) {
                              this.det = this.det2By2();
                              this.detKnown = true;
                        }
                        determinant = this.det;
                        if (determinant == 0) { // Preventing runtime error of dividing by zero
                              throw new Exception("Singular matrix has no inverse");
                        } else {
                              inverse.mat[0][0] = this.mat[1][1];
                              inverse.mat[1][1] = this.mat[0][0];
                              inverse.mat[0][1] = - this.mat[0][1];
                              inverse.mat[1][0] = - this.mat[1][0];
                              inverse = inverse.scale(1 / determinant);
                              System.out.println("Determinant:" + determinant);
                        }
                  } else {
                        Matrix cofactors = this.cofactors();
                        if (!this.detKnown) {
                              this.det = this.detFromCofactors(cofactors);    /* More efficient than det() because
the
                                                                                                    required
cofactors have already been

      calculated */
                              this.detKnown = true;
                        }
                        determinant = this.det;
                        if (determinant == 0) {
                              throw new Exception("Singular matrix has no inverse");
                        } else {
                              inverse = cofactors.transpose().scale(1 / determinant);     /* Find the adjugate and
scale by the
```

```
	reciprocal of the determinant */
                        }
                    }
                    return inverse;
              }
        }

        /**
         * Returns the transpose of the matrix in which the columns of the original become the rows of the transpose
and
         * the rows of the original become the columns of the transpose (equivalent to reflecting the elements in the
         * diagonal through the top left and bottom right elements of the matrix
         * @return the transpose of the matrix
         */
        public Matrix transpose() {
              Matrix transpose = new Matrix(this.n, this.m);  /* Number of cols of transpose equals number of rows of
                                                                                                  original and vice versa */
              for (int i = 0; i < this.m; i++) {
                    for (int j = 0; j < this.n; j++) {
                          transpose.mat[j][i] = this.mat[i][j];     // Rows become columns and columns become rows
                    }
              }
              transpose.det = this.det; /* If the transpose is not square, transpose.detKnown will be false anyway and
                                                      this.det will be the default value, so transpose.det will not
change */
              return transpose;
        }

        /**
         * Returns the determinant of the matrix
         * @return the determinant of the matrix
         * @throws IllegalArgumentException if the matrix is not square (the number of columns equals the number of
rows
         * if a matrix is square)
         */
        public double det() throws IllegalArgumentException {
              if (this.m == this.n) {
                    if (!this.detKnown) {
```

```
                        this.detKnown = true;
                        this.det = this.expand();
                  }
                  return this.det;
            } else {
                  throw new IllegalArgumentException("Rectangular matrices have no determinant");
            }
      }

      /**
       * Returns the matrix of cofactors corresponding to the matrix for which this method is being called
       * @return the matrix of cofactors corresponding to the matrix for which this method is being called
       */
      private Matrix cofactors() {
            Matrix cofactors = new Matrix(this.m, this.n);
            for (int i = 0; i < this.m; i++) {
                  for (int j = 0; j < this.n; j++) {
                        if ((i + j) % 2 == 0) {
                              cofactors.mat[i][j] = this.crop(i, j).expand();
                        } else {
                              cofactors.mat[i][j] = - this.crop(i, j).expand();
                        }
                  }
            }
            return cofactors;
      }

      /**
       * An alternative to the det() method which resuses the cofactors calculated for the inverse method so that
they
       * don't need to be recalculated
       * @param cofactors the matrix of cofactors corresponding to the matrix for which this method is being called
       * @return the determinant of the matrix for which this method is being called
       * @throws IllegalArgumentException if the matrix of cofactors doesn't have the same number of rows as the
       * matrix for which this method is called
       */
      private double detFromCofactors(Matrix cofactors) {
            if (this.m != cofactors.m) {
```

```
                throw new IllegalArgumentException("A matrix and its matrix of cofactors must have the same number
of rows");
            } else {
                double determinant = 0;
                for (int i = 0; i < this.m; i++) {
                    determinant += this.mat[i][0] * cofactors.mat[i][0];
                }
                return determinant;
            }
        }

        /**
         * Recursive method for finding the determinant of a matrix by finding the determinants of sub-matrices
         * @return the determinant of the matrix for which the method is being called
         */
        private double expand() {
            // This is a private method only called by other methods which first checked that the matrix is square,
so this check is not needed here
            if (this.m == 3) {
                return this.det3By3();
            } else if (this.m == 2) {
                return this.det2By2();
            } else {
                Matrix reducedMatrix = this.eliminated();
                double determinant = 0;
                for (int i = 0; i < reducedMatrix.m; i++) {
                    if (reducedMatrix.mat[i][0] == 0) { /* We know that the determinant will not change if the
                                                                element is zero */
                        continue;
                    }
                    if (i % 2 == 0) {
                        determinant += reducedMatrix.mat[i][0] * reducedMatrix.crop(i, 0).expand();
                    } else {
                        determinant -= reducedMatrix.mat[i][0] * reducedMatrix.crop(i, 0).expand();
                    }
                }
                return determinant;
            }
        }
```

```
	/**
	 * Returns a version of the matrix where the columns have been manipulated and rows swapped such that the
	 * determinant is unchanged but calculation of the determinant through expanding of the top row is quicker
	 * (because the top row has many zeroes, so the determinant will be a sum of values of which many will be zero
	 * and we know which will be zero without having to do the full calculations)
	 * @return a 'simpler' matrix with the same determinant as the original
	 */
	public Matrix eliminated() {
		Matrix result = new Matrix(this.m, this.n);
		result.setElements(this);
		int zeroes;
		int maxZeroes = 0;
		int bestRow = 0;
		for (int i = 0; i < result.n; i++) {
			zeroes = 0;
			for (int j = 0; j < result.m; j++) {
				if (result.mat[j][i] == 0) {
					zeroes++;
				}
			}
			if (zeroes > maxZeroes) {
				maxZeroes = zeroes;
				bestRow = i;
			}
		}
		if (bestRow != 0) {
			/* 3 rows are swapped (equivalent of swapping 0 and bestRow then bestRow and another row (not 0))
			 * because a swap negates the determinant, so an even number of swaps are used to leave the
determinant
			 * unchanged*/
			double tempElement;
			int swapRow = 1;
			if (bestRow == 1) {
				swapRow = 2;
			}
			for (int i = 0; i < result.m; i++) {
				tempElement = result.mat[i][0];
				result.mat[i][0] = result.mat[i][bestRow];
```

```
                        result.mat[i][bestRow] = result.mat[i][swapRow];
                        result.mat[i][swapRow] = tempElement;
                  }
            }
            if (maxZeroes < result.m - 1) {
                  int preserveCol = -1;
                  for (int i = 0; i < result.m; i++) {
                        if ((preserveCol >= 0) && (result.mat[i][0] != 0)) {
                              double factor = result.mat[i][0] / result.mat[preserveCol][0];
                              result.mat[i][0] = 0;
                              for (int j = 1; j < result.n; j++) {
                                    result.mat[i][j] -= result.mat[preserveCol][j] * factor;
                              }
                        }
                        if (result.mat[i][0] != 0) {
                              preserveCol = i;
                        }
                  }
            }
            if (this.detKnown) {
                  result.detKnown = true;
                  result.det = this.det;
            }
            return result;
      }

      /**
       * Returns a sub-matrix (known as a minor) of the matrix for which this method is called by removing the
column and the row
       * specified
       * @param col the column to crop in order to create the sub-matrix
       * @param row the row to crop in order to create the sub-matrix
       * @return the matrix for which the method is called but with a column and a row removed
       */
      private Matrix crop(int col, int row) {
            Matrix result = new Matrix(this.m - 1, this.n - 1);
            for (int i = 0; i < col; i++) {
                  for (int j = 0; j < row; j++) {
                        result.mat[i][j] = this.mat[i][j];
```

```
                }
                for (int j = row + 1; j < this.n; j++) {
                        result.mat[i][j - 1] = this.mat[i][j];
                }
        }
        for (int i = col + 1; i < this.m; i++) {
                for (int j = 0; j < row; j++) {
                        result.mat[i - 1][j] = this.mat[i][j];
                }
                for (int j = row + 1; j < this.n; j++) {
                        result.mat[i - 1][j - 1] = this.mat[i][j];
                }
        }
        return result;
    }

    /**
     * Quick non-recursive method for calculating the determinant of a 3 by 3 matrix
     * @return the determinant of the matrix (must be 3 by 3)
     */
    private double det3By3() {
        // This private method is only called when the matrix is already known to be 3 by 3, so we don't need a
check here
        return this.mat[0][0] * this.mat[1][1] * this.mat[2][2]
                + this.mat[1][0] * this.mat[2][1] * this.mat[0][2]
                + this.mat[2][0] * this.mat[0][1] * this.mat[1][2]
                - this.mat[0][2] * this.mat[1][1] * this.mat[2][0]
                - this.mat[1][2] * this.mat[2][1] * this.mat[0][0]
                - this.mat[2][2] * this.mat[0][1] * this.mat[1][0];
    }

    /**
     * Quick non-recursive method for calculating the determinant of a 2 by 2 matrix
     * @return the determinant of the matrix (must be 2 by 2)
     */
    private double det2By2() {
        // This private method is only called when the matrix is already known to be 2 by 2, so we don't need a
check here
        return this.mat[0][0] * this.mat[1][1] - this.mat[1][0] * this.mat[0][1];
```

```
	}

	/**
	 * Returns the EulerTriple (three euler angles) representing the orientation of an object when this matrix is interpreted as converting points from an object space to the corresponding upright space
	 * @return the orientation/angular displacement represented by the matrix when interpreted as converting points from object space to upright space in EulerTriple form
	 * @throws IllegalArgumentException if the matrix is not 3 by 3 or if the matrix does not have determinant 1 (a matrix that satisfied these conditions is not necessarily a rotation matrix, so it is the responsibility of the code that calls this method to ensure the matrix represents a rotation)
	 */
	public EulerTriple eulerObToUp() {
		if ((this.m != 3) && (this.n != 3)) {
			throw new IllegalArgumentException("Only 3 by 3 matrices can be converted to Euler angles");
		} else if ((det() < 0.9999) || (det() > 1.0001)) { // Allow some leeway due to matrix creep
			throw new IllegalArgumentException("Only rotation matrices (which have a determinant of 1) can be converted to Euler angles");
		} else {
			double heading; // Rotation about the body y-axis
			double pitch; // Rotation about the body x-axis
			double bank; // Rotation about the body z-axis
			double sinPitch = - this.mat[2][1];
			if (sinPitch <= -1) {
				pitch = - Math.PI / 2;
			} else if (sinPitch >= 1) {
				pitch = Math.PI / 2;
			} else {
				pitch = Math.asin(sinPitch);
			}
			if (Math.abs(sinPitch) > 0.9999) {
				bank = 0;
				heading = Math2.atan(-this.mat[0][2], this.mat[0][0]);
			} else {
				heading = Math2.atan(this.mat[2][0], this.mat[2][2]);
				bank = Math2.atan(this.mat[0][1], this.mat[1][1]);
			}
			return new EulerTriple(heading, pitch, bank);
		}
	}
```

```
	/**
	 * Returns the EulerTriple (three euler angles) representing the orientation of an object when this matrix is interpreted as converting points from an upright space to the corresponding upright space
	 * @return the orientation/angular displacement represented by the matrix when interpreted as converting points from object space to upright space in EulerTriple form
	 * @throws IllegalArgumentException if the matrix is not 3 by 3 or if the matrix does not have determinant 1 (a matrix that satisfied these conditions is not necessarily a rotation matrix, so it is the responsibility of the code that calls this method to ensure the matrix represents a rotation)
	 */
	public EulerTriple eulerUpToOb() {
		try {
			return this.transpose().eulerObToUp(); // Rotation matrices can be inverted by transposition (object-to-upright and upright-to-object matrices are inverses of each other)
		} catch (IllegalArgumentException e) {
			throw e;
		}
	}
}
```

3.12 Vector.java

```
package RefractionSim;
/**
 * Class for vectors of all kinds. All vectors are column vectors because matrices are column-major.
 * @author William Platt
 *
 */
public class Vector {
	private int n; // Size of vector (number of rows or number of elements)
	private double[] vec; // Elements of the vector
	private double modulus; /* The 'length' or 'magnitude' of a vector (or mathematically, the euclidean norm or
 2-norm) */
	private boolean modulusKnown; /* States whether the value stored in modulus is correct; the modulus will
 will change when the vector is changed but we don't want to
recalculate the
 modulus unless we have to */

	/**
	 * Constructor for the Vector class which sets all elements to zero
	 * @param n number of elements that the new vector is to have
	 * @throws IllegalArgumentException if n is less than or equal to 2 (as this would produce nothing or a single
	 * value)
	 */
	public Vector(int n) {
		if (n <= 1) {
			throw new IllegalArgumentException("A vector needs at least two rows");
		} else {
			this.n = n;
			this.vec = new double[n];
			for (int i = 0; i < n; i++) {
				this.vec[i] = 0;
			}
			this.modulus = 0;
			this.modulusKnown = true;
		}
	}

	/**
```

```
 * Returns the number of rows the matrix has
 * @return the number of rows the matrix has
 */
public int getN() {
	return n;
}

/**
 * Sets the value of a single element in the vector to that of the newValue parameter
 * @param i the index (row) of the element to change (indices start at 0)
 * @param newValue the value which the specified element should be changed to
 * @throws ArrayIndexOutOfBoundsException if i >= number of elements (rows)
 */
public void setElement(int i, double newValue) {
	if (i >= this.n) {
		throw new ArrayIndexOutOfBoundsException("Vector element does not exist");
	} else {
		this.vec[i] = newValue;
		modulusKnown = false;
	}
}

/**
 * Sets all elements of the vector to the values in the newValues array
 * @param newValues the array of new values that the vector elements should be set to
 * @throws IllegalArgumentException if the length of the array is not equal to the number of elements in the
 * vector
 */
public void setElements(double[] newValues) {
	if (newValues.length != this.n) {
		throw new IllegalArgumentException("Array argument of different length to vector");
	} else {
		for (int i = 0; i < this.n; i++) {
			this.vec[i] = newValues[i];
		}
		modulusKnown = false;
	}
}
```

```
/**
 * Copies the values of the elements of the vector newValues to the corresponding elements in the vector for
 * which this method is being called (the two vectors then don't reference the same locations)
 * @param newValues the vector of new values the vector elements should be set to
 * @throws IllegalArgumentException if the newValues parameter is not the same length as the vector for which
 * this method is called
 */
public void setElements(Vector newValues) {
	try {
		this.setElements(newValues.vec);
	} catch (IllegalArgumentException e) {
		throw new IllegalArgumentException("Vector size mismatch");
	}
}

/**
 * Gets the value of a single element in the vector
 * @param i the index (row) of the element to return the value of
 * @return the value of the specified element
 * @throws ArrayIndexOutOfBoundsException if i >= number of elements or i < 0
 */
public double getElement(int i) {
	if ((i >= this.n) || (i < 0)) {
		throw new ArrayIndexOutOfBoundsException("Vector element does not exist");
	} else {
		return this.vec[i];
	}
}

/**
 * Returns the values of all elements in an array
 * @return the array of values in the vector
 */
public double[] getElements() {
	double[] elements = new double[this.n];
	for (int i = 0; i < this.n; i++) {
		elements[i] = this.vec[i];
	}
	return elements;
```

```
	}

	/**
	 * Returns the sum of this vector and the toAdd parameter vector
	 * @param toAdd the vector to add to the vector for which this method is being called
	 * @return the sum of this vector and the vector passed as a parameter (the return vector will be the same
size
	 * as both vectors being added)
	 * @throws IllegalArgumentException if the toAdd vector is not the same size as the vector for which this
method
	 * is being called
	 */
	public Vector add(Vector toAdd) {
		if (this.n != toAdd.n) {
			throw new IllegalArgumentException("Vectors size mismatch for addition");
		} else {
			Vector result = new Vector(this.n);
			for (int i = 0; i < this.n; i++) {
				result.setElement(i, this.vec[i] + toAdd.vec[i]); // Set element sets result.modulusKnown to
false
			}
			return result;
		}
	}

	/**
	 * Returns a - b where a is the vector for which this method is called and b is the toSubtract parameter
vector
	 * @param toSubtract vector to subtract from the vector for which this method is called
	 * @return a - b where a is the vector for which this method is called and b is the toSubtract parameter
vector
	 * @throws IllegalArgumentException if the two vectors don't have the same dimensions
	 */
	public Vector subtract(Vector toSubtract) {
		if (this.n != toSubtract.n) {
			throw new IllegalArgumentException("Vectors size mismatch for subtraction");
		} else {
			// Scaling toSubtract by -1 and adding it to `this` is avoided in order to reduce computation time
			Vector result = new Vector(this.n);
```

```
                for (int i = 0; i < this.n; i++) {
                        result.setElement(i, this.vec[i] - toSubtract.vec[i]);
                }
                return result;
        }
    }

    /**
     * Returns the dot product of the vector for which this method is called and the toDot parameter vector. The
dot
     * product is associative, meaning that a.b = b.a
     * @param toDot the vector to dot with
     * @return the dot product of the two vectors (representing |a||b|cos(x) where a and b are the two vectors and
     * x is the angle between them - this is the same as the component of a parallel to b multiplied by |a|)
     */
    public double dotProduct(Vector toDot) {
        if (this.n != toDot.n) {
                throw new IllegalArgumentException("Vectors size mismatch for dot product");
        } else {
                double result = 0;
                for (int i = 0; i < this.n; i++) {
                        result += this.vec[i] * toDot.vec[i];
                }
                return result;
        }
    }

    /**
     * Returns the vector multiplied by a scalar value
     * @param scaleFactor the scalar value by which the vector is to be multiplied
     * @return the vector produced when the original vector is multiplied by scaleFactor
     */
    public Vector scale(double scaleFactor) {
        Vector result = new Vector(this.n);
        for (int i = 0; i < this.n; i++) {
                result.vec[i] = this.vec[i] * scaleFactor;
        }
        if (this.modulusKnown) {
```

```
                result.modulus = this.modulus * scaleFactor; /* result.modulusKnown will already be true because
result
                                                                                was just constructed */
            } else {
                result.modulusKnown = false;
            }
            return result;
      }

      /**
       * Returns a x b (a crossed with b) where a is the vector for which this method is being called and b is the
       * toCross parameter vector. Both a and b must have 3 elements (rows) and the cross product will also have 3.
       * The cross product represents a vector perpendicular to both vectors with modulus |a||b|sin(x) where x is
the
       * angle between the vectors a and b
       * @param toCross the 3-row vector b in a x b where a is the vector for which this method is being called
       * @return a vector with 3 elements (rows) which represents a x b (a crossed with b) a is the vector for which
       * this method is being called and b is the toCross parameter vector
       */
      public Vector crossProduct(Vector toCross) {
            if ((this.n != 3) || (toCross.n != 3)) {
                throw new IllegalArgumentException("Two vectors must have exactly 3 rows in order to be crossed");
            } else {
                Vector result = new Vector(3);
                result.setElement(0, this.vec[1] * toCross.vec[2] - this.vec[2] * toCross.vec[1]); /* setElement
sets

                                result.modulusKnown

                                to false */
                result.setElement(1, this.vec[2] * toCross.vec[0] - this.vec[0] * toCross.vec[2]);
                result.setElement(2, this.vec[0] * toCross.vec[1] - this.vec[1] * toCross.vec[0]);
                return result;
            }
      }

      /**
       * Returns the modulus (a.k.a. length, euclidean norm or 2-norm) of the vector
       * @return the modulus/length/euclidean norm/2-norm of the vector
```

```
	 */
	public double modulus() {
		if (this.modulusKnown) {
			return this.modulus;
		} else {
			double result = 0;
			for (int i = 0; i < this.n; i++) {
				result += Math.pow(this.vec[i], 2);
			}
			result = Math.sqrt(result);
			this.modulus = result;
			this.modulusKnown = true;
			return result;
		}
	}

	/**
	 * Returns true if the vector is normalised (has modulus 1). Some leeway is acceptable for the vector to be
	 * considered normalised
	 * @return whether or not the vector is normalised
	 */
	public boolean isNormalised() {
		if ((this.modulus() < 1.00000001) && (this.modulus() > 0.99999999)) { /* Consider a vector with modulus
between
                                                                                                    1.0001
and 0.9999 inclusive to be normalised
                                                                                                    (this is
the convention throughout this
                                                                                                    project)
*/
			return true;
		} else {
			return false;
		}
	}

	/**
	 * Returns the normalised version of the original vector (meaning it has modulus 1 with slight leeway)
	 * @return the normalised version of the original vector
```

```
	 */
	public Vector normalise() {
		if (this.isNormalised()) {
			return this;
		} else {
			return this.scale(1 / this.modulus());
		}
	}
}
```

3.13 EulerTriple.java

```
package RefractionSim;
/**
 * Class for Euler angle triples (these represent an orientation in 3-D space)
 * @author William Platt
 *
 */
public class EulerTriple {

	private double heading; // Rotation about the vertical (y) axis
	private double pitch; // Rotation about the object space x-axis after applying the heading rotation
	private double bank; // Rotation about the object space z-axis after applying the heading and pitch rotations

	/**
	 * Constructor for the EulerTriple class which sets the three angles to the three parameters
	 * @param heading the rotation about the vertical (y) axis
	 * @param pitch the rotation about the object space x-axis after applying the heading rotation
	 * @param bank the rotation about the object space z-axis after applying the heading and pitch rotations
	 */
	public EulerTriple(double heading, double pitch, double bank) {
		// Ensure angles are in canonical form (-pi <= heading <= pi, -halfPi <= pitch <= halfPi, -pi <= bank <=
pi)
		double pi = Math.PI;
		double twoPi = pi * 2;
		double halfPi = pi / 2;

		if (Math.abs(pitch) > halfPi) {
			pitch += halfPi;
			pitch = pitch % twoPi;
			if (pitch > pi) {
				heading += pi;
				pitch = (3 * pi / 2) - pitch;
			} else {
				pitch -= halfPi;
			}
		}
		if (Math.abs(pitch) >= 0.99999 * halfPi) { // If there is gimbal lock, assign all rotation to pitch and
bank
			bank += heading;
```

```
            heading = 0;
        } else {
            if (Math.abs(heading) > pi) {
                heading += pi;
                heading = heading % twoPi;
                heading -= pi;
            }
        }
        if (Math.abs(bank) > pi) {
            bank += pi;
            bank = bank % twoPi;
            bank -= pi;
        }
        this.heading = heading;
        this.pitch = pitch;
        this.bank = bank;
}

/**
 * Returns the heading angle
 * @return the angle of rotation about the vertical axis
 */
public double getHeading() {
        return heading;
}

/**
 * Returns the pitch angle
 * @return the angle of declination
 */
public double getPitch() {
        return pitch;
}

/**
 * Returns the bank angle
 * @return the angle of rotation along the body z-axis
 */
public double getBank() {
```

```
            return bank;
      }

      /**
       * Returns the matrix for transforming points from object space to upright space where the EulerTriple is the
angular displacement of object space from upright space
       * @return the object space to upright space matrix represented by the EulerTriple
       */
      public Matrix matrixObToUp() {
            Matrix objectToUpright = new Matrix(3, 3);
            double ch = Math.cos(this.heading);
            double sh = Math.sin(this.heading);
            double cp = Math.cos(this.pitch);
            double sp = Math.sin(this.pitch);
            double cb = Math.cos(this.bank);
            double sb = Math.sin(this.bank);
            double chcb = ch * cb;
            double shsb = sh * sb;
            double shcb = sh * cb;
            double chsb = ch * sb;
            objectToUpright.setElement(0, 0, chcb + shsb * sp);
            objectToUpright.setElement(0, 1, sb * cp);
            objectToUpright.setElement(0, 2, chsb * sp - shcb);
            objectToUpright.setElement(1, 0, shcb * sp - chsb);
            objectToUpright.setElement(1, 1, cb * cp);
            objectToUpright.setElement(1, 2, shsb + chcb * sp);
            objectToUpright.setElement(2, 0, sh * cp);
            objectToUpright.setElement(2, 1, -sp);
            objectToUpright.setElement(2, 2, ch * cp);
            return objectToUpright;
      }

      /**
       * Returns the matrix for transforming points from upright space to object space where the EulerTriple is the
angular displacement of object space from upright space
       * @return the upright space to object space matrix represented by the EulerTriple
       */
      public Matrix matrixUpToOb() {
```

```
            return this.matrixObToUp().transpose(); // The transpose is the same as the inverse for rotation
matrices
      }

      /**
       * Returns the sum of the EulerTriple for which this method is called and the toAdd parameter
       * @param toAdd representation of an angular displacement to add to the angular displacement represented by
the EulerTriple for which this method is called
       * @return the EulerTriple representing the sum of the two angular displacements
       */
      public EulerTriple add(EulerTriple toAdd) {
            return new EulerTriple(this.heading + toAdd.heading, this.pitch + toAdd.pitch, this.bank + toAdd.bank);
      }

}
```

3.14 Ray.java

```
package RefractionSim;

/**
 * Class for rays; lines with a starting point and a direction
 * @author William Platt
 *
 */
public class Ray {

	private Vector p;
	private Vector v;

	/**
	 * Constructor for the Ray class
	 * @param p the starting point of the ray
	 * @param v the direction of the ray
	 */
	public Ray(Vector p, Vector v) {
		this.p = p;
		this.v = v;
	}

	/**
	 * Returns the starting point of the ray
	 * @return the point where the ray starts
	 */
	public Vector getP() {
		return p;
	}

	/**
	 * Returns the direction the ray extends from its starting point
	 * @return the direction of the ray
	 */
	public Vector getV() {
		return v;
	}
```

```
}
```

3.15 Edge2D.java

```
package RefractionSim;

/**
 * Class for edges (line segments) in two dimensions
 * @author William Platt
 *
 */
public class Edge2D {

	private double x0; // x co-ordinate of the lower point
	private double y0; // y co-ordinate of the lower point
	private double x1; // x co-ordinate of the higher point
	private double y1; // y co-ordinate of the higher point
	private double height; // Positive difference between the y co-ordinates of the two points

	/**
	 * Constructor for the Edge2D class
	 * @param x0 the x co-ordinate of the first end of the edge
	 * @param y0 the y co-ordinate of the first end of the edge
	 * @param x1 the x co-ordinate of the second end of the edge
	 * @param y1 the y co-ordinate of the second end of the edge
	 */
	public Edge2D(double x0, double y0, double x1, double y1) {
		if (y0 < y1) {
			this.x0 = x0;
			this.y0 = y0;
			this.x1 = x1;
			this.y1 = y1;
			this.height = y1 - y0; // Height is positive
		} else { // Reverse the points so that (x0, y0) is lower than (x1, y1)
			this.x0 = x1;
			this.y0 = y1;
			this.x1 = x0;
			this.y1 = y0;
			this.height = y0 - y1; // Height is positive
		}
	}
```

```
/**
 * Returns the x co-ordinate of the lower end of the edge
 * @return the x co-ordinate of the lower of the two end points of the edge
 */
public double getX0() {
      return x0;
}

/**
 * Returns the x co-ordinate of the higher end of the edge
 * @return the x co-ordinate of the higher of the two end points of the edge
 */
public double getX1() {
      return x1;
}

/**
 * Returns the y co-ordinate of the lower end of the edge
 * @return the y co-ordinate of the lower of the two end points of the edge
 */
public double getY0() {
      return y0;
}

/**
 * Returns the y co-ordinate of the higher end of the edge
 * @return the y co-ordinate of the higher of the two end points of the edge
 */
public double getY1() {
      return y1;
}

/**
 * Returns the vertical height of the edge
 * @return the y component of the edge's length
 */
public double getHeight() {
      return height;
}
```

```
}
```

3.16 Math2.java

```
package RefractionSim;
/**
 * Class for mathematical functions which are not already provided in Math.
 * @author William Platt
 *
 */
public class Math2 {

	/**
	 * The constructor for the Math2 class is private because there aren't supposed to be any instances of this
	 * class. The purpose of this class is the provide publicly accessible static methods like the Math class
	 */
	private Math2() {}

	/**
	 * Returns an angle (in radians) from a vertical component and a horizontal component. All vectors in the
plane
	 * can be given a proper angle by this function; Math.atan limits angles to the interval (-pi/2, pi/2)
	 * @param y the vertical component of a 2-D vector
	 * @param x the horizontal component of a 2-D vector
	 * @return the angle (in radians) of a vector to the positive x-axis under standard mathematical conventions
	 */
	public static double atan(double y, double x) {
		if (x == 0) {
			if (y == 0) {
				return 0;
			} else if (y > 0) {
				return (Math.PI / 2);
			} else {
				return (-Math.PI / 2);
			}
		} else if (x > 0) {
			return Math.atan(y / x);
		} else {
			if (y >= 0) {
				return (Math.atan(y / x) + Math.PI);
			} else {
				return (Math.atan(y / x) - Math.PI);
```

```
                }
            }
        }

        /**
         * Returns the midpoint of the line segment between p0 and p1 in n-dimensional space
         * @param p0 one end of a line segment
         * @param p1 the other end of the line segment
         * @return the point on the line between p0 and p1 that is halfway between the two points
         */
        public static Vector midpoint(Vector p0, Vector p1) {
            if (p0.getN() != p1.getN()) {
                throw new IllegalArgumentException("A line segment must be between two points in the same number
of dimensions");
            } else {
                Vector midpoint = new Vector(p0.getN());
                for (int i = 0; i < p0.getN(); i++) {
                    midpoint.setElement(i, (p0.getElement(i) + p1.getElement(i)) / 2);
                }
                return midpoint;
            }
        }

}
```