A Basic Implementation Of

IMAGE-BASED VISUAL HULLS

Jesper Carlson Daniel Enetoft Anders Fjeldstad Kristofer Gärdeborg

March 21 2005 Linköping University

ABSTRACT

In this report we present the process of implementing an approach to create an image based visual hull. The approach is very similar to the naive solution described by Matusik (2000). It uses a ray casting technique along with image-based silhouette data to create the visual hull. Our result is a Java application and applet that allows the user to create the visual hull of a convex object from images fulfilling some properties. The user can change the viewpoint through our graphical user interface. The resulting visual hull can be rendered either as a depth-map or textured with pixel values extracted from the reference images.

CONTENTS

INTRODUCTION

The idea of rendering novel views of an object using nothing but static images as input offers many challenges and could very well have applications in a number of areas ranging from movie special effects to online merchandise visualization. We wanted to experiment with a shape-from-silhouette approach to this problem, and decided to implement a basic algorithm for calculating and rendering visual hulls based on image information.

IMAGE-BASED VISUAL HULLS

The technique of image-based visual hulls is a method for obtaining an idea about how an object would look like from novel views based entirely on information available in photographs or rendered images. In this section, we will describe the general idea, our approach to the problem and how we actually implemented it.

GENERAL IDEA

Suppose that you have a number of photographs of an object, and would like to produce a new image that shows the object from another angle. This is a difficult problem, since no geometry or reflectance properties are known in advance. A number of approaches have been proposed, and image-based visual hulls is one of them. It is based on the principle of shape-from-silhouette, that is, to use only silhouette information from a number of reference images to obtain the general shape of the object. For this purpose, the reference images are assumed to contain information about what parts of them are to be considered "object" and what parts are "background".

Imagine a reference image positioned in the 3D scene at the same position and oriented in the same direction as the camera was when the picture was taken or rendered. If one extrudes the silhouette from the camera's center of projection out into 3D space, one will obtain a cone-like volume. Now suppose the same was done with all available reference images. The intersection of the resulting volumes is the visual hull of the object. It is guaranteed to contain the object, and to be an equal or better fit than the convex hull. However, since the visual hull is derived from silhouette information, concavities will not be represented properly.

Furthermore, one can use the reference images as textures for the visual hull, perferably through some view-dependent mapping method.

OUR APPROACH

We started by reading "Image-based visual hulls", a paper written by Matusik et al. in 2000. They had made one naive and one extensive approach in their paper. The differences between the approaches were the time complexity and that the more ambitious implementation could handle non-convex objects. In their extensive approach, Matusik had used epipolar geometry to make the progress of creating and shading the visual hull faster. We found it hard to understand all the parts of their second solution, so we chose to base our implementation on the naive one. In the paper, little is said about the details of the naive approach so in many cases we had to come up with the solutions by ourselves.

By not exploiting the epipolar geometry we also chose to skip the part where it's decided whether a part of an object is visible or not. This restricts our program to calculate visual hulls of convex objects only.

When it comes to shading, we chose just to pick the pixels from the camera closest in angle to the desired view. At first we thought about doing this by using a weighting function as described in Debevec's paper from SIGGRAPH '96, but there was not enough time to implement it. Because of this, the shaded visual hull does not look very good from certain angles.

We also implemented a depth-map function, which is handy for examining the shape of the actual visual hull in 3D. When the depth-map is active, the visual hull is not texturized with pixel values from the reference images, but rather represented with colors ranging from white to black depending on the orthogonal distance to the viewpoint.

IMPLEMENTATION

We wanted to implement our algorithm in Java 2D mainly because we found it convenient to make a graphical user interface for the application with Java and we are comfortable with the language in general. Below is a simplified pseudo-code for our program.

```
for each pixel in novel view
   create a Ray r from position of desired view through the pixel position in 3D
   put it in visual hull vh
end

for each Camera c in refViews
   translate c to origo
   translate vh accordingly
   rotate c to point to 0,0,-1
   rotate vh accordingly

   for each Ray r in vh
      project r to c
      calculate interval
      intersect with vh
   end

   rotate vh back
   rotate c back
   translate vh back
   translate c back
end


for each StartPointInInterval in vh
   if texturized shading selected
      calculate angles to each Camera in refViews
      set pixel to picked value in Camera with smallest angle
   elseif depth-map shading selected
      set the pixel intensity value to the inverse of the distance to the novel view
end
```

At first, the visual hull is simply a collection of imaginary rays whose starting points coincide with the center of projection of the camera of the desired view. The rays are oriented so that each pixel in the image of the desired view has a ray going through it. All rays contain an infinite interval in terms of its scaling parameter.

Each ray is then projected down to the image plane of each reference view and the interval of the ray which is contained within the silhouette of the object in the reference view is saved. To project the ray you need to rotate and translate the whole 3D world, with cameras and rays, so that the camera of the reference view lies in origin and its image lies centered on the negative z axis.

The intervals from each reference view are intersected with each other, ray by ray. When this is done, the visual hull has been created. The visual hull is simply the volume that lies within the point cloud represented by the endpoints of the intervals of the rays. The starting point on each interval is the points on the object that you will see from your desired view.

Now the visual hull is either shaded with a depth-map or with pixels from the reference camera closest in angle. In the case of depth-map shading the pixels are just set to an intensity of gray that represent the inverse orthogonal distance from the point to the desired view. This means that the nearest parts will get a brighter tone than the ones further away.

In the case of texturized shading, the 3D point corresponding to each interval's startpoint is projected down on the image plane of the camera that is closest in angle to the desired view. The pixel value from the projected point is picked and set as the color of the pixel corresponding to the current ray.

Since our program is not very fast at calculating the visual hull, we decided to use a wire-frame cube as a virtual object that the user can rotate the novel view-camera around. That way, we get a framerate suitable for interaction even though the visual hull could never be computed quickly enough.

To be able to demonstrate our work more easily, we created an applet version of our program in addition to the stand-alone application. The applet version lacks some functions, such as the option to add and remove reference images at runtime. The applet comes with a fixed set of reference views and precalculated parameters for the reference cameras.

RESULTS

We have developed a working stand-alone application in Java where it is possible to add reference images with camera parameters and to render novel views of the object described by the supplied images. The rendered image can either be a depth-map of the visual hull or a texturized version where the color information is collected from the reference images. We have also created a Java applet to demonstrate the project.

Currently, our program only supports convex objects. It would be possible to extend the implementation to be able to take images of non-convex objects as input, but the problem of self-occlusion that then arises is non-trivial and we have no solution ready for it, considering the approach we have chosen for the rest of the implementation.

To test our implementation, we have rendered six images of a milk carton to use as reference images. The camera parameters were easy to obtain from the modeling program used to create and render the milk carton model. We chose the milk carton since it is purely convex and have a texture that makes it easy to tell one side from another. The reference images are shown in figures 1-6.

Using the interface of our program, we have rendered a number of novel views of the milk carton, both using the depth-map shading and the simple view-dependent texture mapping. The results of the depth-map renderings are reasonably good in general (see figures 7-8) even if artifacts arise from time to time (see figure 9-10), but the texture mapping tends to be quite buggy (see figures 11-12), mostly because of our very much simplified version of the view-dependent texture mapping algorithm. Note though that we have got some quite pleasing results using the textured shading as well – see figure 13 for an example.

Figure 1. Reference image 1.

Figure 2. Reference image 2.

Figure 3. Reference image 3.

Figure 4. Reference image 4.


Figure 5. Reference image 5.

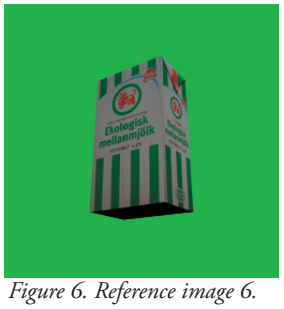

Figure 6. Reference image 6.

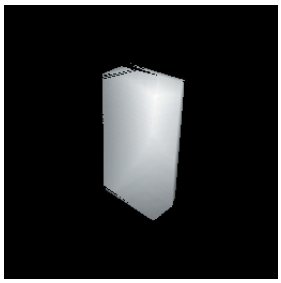

Figure 7. Good depth-map.

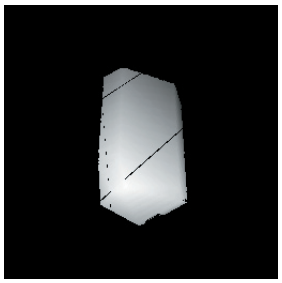

Figure 8. Good depth-map.

Figure 9. Depth-map with artifacts.

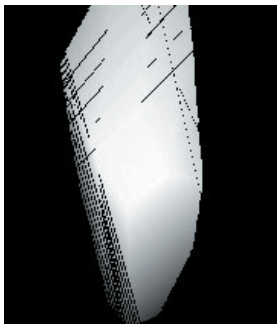

Figure 10. Bad depth-map. These results tend to occur less frequent than the good ones.

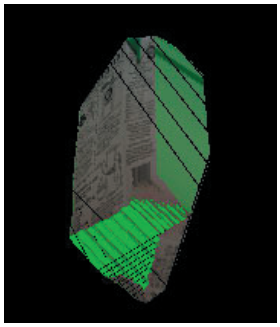

Figure 11. Poorly texturized visual hull.

Figure 12. Poorly texturized visual hull.

Figure 13. Nice result from shading the visual hull.

At some angles, strange artifacts occur which we have not found an explanation to as of yet. These can sometimes be seen as isolated spots outside of the visual hull, where there should not be any non-empty intervals at all (see figure 9). One contributing factor could be rounding errors or some faulty rotation, but we are not sure. We know that the program produces strange results when the angle between the view direction of the novel view and the view direction of a refrence camera is small, but we have avoided this by simply discarding such information.

One of the more negative sides of our implementation is the time required to calculate the visual hull – when using all six reference images of the milk carton the complete process of rendering one frame takes about 90 seconds on an AMD Athlon XP 1800. The low framerate is partly due to the fact that Java is not the best choice of programming language for number-crunching applications, but mostly because our implementation does not exploit the properties of epipolar geometry that Matusik et al. proposed in their 2000 paper. Another factor is the laborious process of accessing the information in the images. One way to solve this would be to represent that information using a more efficient data structure.

CONCLUSION

There are a number of aspects of our implementation that could definitively be improved, but we must consider them as being outside the scope of this project. From the beginning we had thoughts about implementing support not only for convex objects, but for concave as well. But since we ran into many problems on the way we chose to only use convex objects. The structure of the code allow for an extension in the future.

Looking back at the project we have realized the importance of doing extensive testing of the different parts of the program. When we put all the different parts together many issues occurred due to small errors in the different underlying algorithms.

Implementing a visual hull approach is not an easy task. The algorithm for creating the visual hull is quite complex and knowledge in many different areas is required. A lot of work was therefore spent on solving different issues that occurred along the project. We did also suffer from the fact that we did not have any possibility to discuss our issues with anybody but ourselves, since we were the only ones who were involved in the details of our code. This resulted in a slow development but at the same time we got a lot of understanding for the different issues that can occur. Since the real goals of the project were to gain deeper knowledge in an area that is currently in research we think have succeeded. We have learned a lot and we now understand the visual hull approach to shape-from-silhouette and its pros and cons quite well. Since we tried to solve the problems in our own way (not using epipolar geometry) we fell into some pits along the way, but on the other hand we know that this is not a trivial subject. We are satisfied with the results and are proud of our work.

REFERENCES

Debevec, Malik (1996). Modeling and Rendering Architecture from Photographs: A hybrid geometry- and image-based approach. SIGGRAPH 1996. ACM.

Matusik, Buehler, Raskar, Gortler, McMillan (2000). Image-based Visual Hulls. SIGGRAPH 2000. ACM.

SOURCE CODE

Below is all the JAVA source code of our application. We have included the standalone application as well as the applet version. The classes are ordered alphabetically. The class ExampleFileFilter is not included, since it's really an example class written by Jeff Dinkins at Sun Microsystems. The class Matrix4 was originally written by Stefan Gustavsson at ITN, Linköping University.

CAMERA

```
import java.awt.image.BufferedImage;

/**
 * This class represents a viewpoint and a (limited) view plane.
 */

public class Camera {

   public static final int DEFAULT_IMAGE_WIDTH = 256;
   public static final int DEFAULT_IMAGE_HEIGHT = 256;

   String title;
   Vector4 position;
   Vector4 direction;
   Vector4 up;
   float focalLength = 0;
   float fov = 0;
   BufferedImage image = null;
   double deg2rad = Math.PI / 180.0;

   /**
    * Creates an empty <code>Camera</code>.
    */
   public Camera() {
      position = new Vector4();
      direction = new Vector4();
      up = new Vector4();

      // Create empty image with default dimensions
      image = new BufferedImage(DEFAULT_IMAGE_WIDTH,
                           DEFAULT_IMAGE_HEIGHT,
                           BufferedImage.TYPE_INT_RGB);
   }

   /**
    * Creates a new <code>Camera</code> with the specified parameters.
    *
    * @param title the title of the <code>Camera</code>.
    * @param pos the position of the <code>Camera</code> in world coordinates.
    * @param dir the direction of the <code>Camera</code> relative to the
    *            view point.
    * @param up the "up" direction of the <code>Camera</code> relative to the
    *         view point.
    * @param focal the focal length (in meters, assuming world coordinate
    *            units is meters) of the <code>Camera</code>.
    * @param fov the field-of-view of the <code>Camera</code> in degrees.
    * @param im the <code>BufferedImage</code> representing the limited view
    *            plane of this <code>Camera</code>.
    */
   public Camera(String title,
              Vector4 pos,
              Vector4 dir,
              Vector4 up,
              float focal,
              float fov,
              BufferedImage im) {

      this.title = title;
      this.position = pos;
      this.direction = dir;
      this.up = up;
      this.focalLength = focal;
      this.fov = (float)Math.toRadians(fov);
      this.image = im;

      // Normalize vectors
      this.direction.normalize();
      this.up.normalize();
   }

   /**
    * Creates an empty <code>Camera</code> with the specified title.
    *
```

```
 * @param title the title of the <code>Camera</code>.
 */
public Camera(String title) {
   this();
   this.title = title;
}

/**
 * Updates this <code>Camera</code> with the supplied data.
 *
 * @param title the title of the <code>Camera</code>.
 * @param pos the position of the <code>Camera</code> in world coordinates.
 * @param dir the direction of the <code>Camera</code> relative to the
 *            view point.
 * @param up the “up” direction of the <code>Camera</code> relative to the
 *         view point.
 * @param focal the focal length (in meters, assuming world coordinate
 *            units is meters) of the <code>Camera</code>.
 * @param fov the field-of-view of the <code>Camera</code> in degrees.
 * @param im the <code>BufferedImage</code> representing the limited view
 *            plane of this <code>Camera</code>.
 */
public void update(String title,
               Vector4 pos,
               Vector4 dir,
                Vector4 up,
               float focal,
               float fov,
               BufferedImage im) {

   this.title = title;
   this.position = pos;
   this.direction = dir;
   this.up = up;
   this.focalLength = focal;
   this.fov = (float)Math.toRadians(fov);
   this.image = im;

   // Normalize vectors
   this.direction.normalize();
   this.up.normalize();
}

/**
 * Returns the title.
 *
 * @return the title.
 */
public String toString() {
   return title;
}

/**
 * Returns the title.
 *
 * @return the title.
 */
public String getTitle() {
   return title;
}

/**
 * Returns the position in world coordinates.
 *
 * @return the position.
 */
public Vector4 getPos() {
   return position;
}

/**
 * Returns the view direction relative to the view point.
 *
 * @return the view direction.
 */
public Vector4 getDir() {
   return direction;
}

/**
 * Returns the “up” direction relative to the view point.
 *
 * @return the “up” direction.
 */
public Vector4 getUp() {
   return up;
}
```

```
   /**
    * Returns the focal length (in meters).
    *
    * @return the focal length.
    */
   public float getFocalLength() {
      return focalLength;
   }

   /**
    * Returns the field-of-view (in degrees).
    *
    * @return the field-of-view.
    */
   public float getFov() {
      return fov;
   }

   /**
    * Returns the image seen by the <code>Camera</code>.
    *
    * @return the image of this <code>Camera</code>.
    */
   public BufferedImage getImage() {
      return image;
   }

   /**
    * Resets the image seen by this <code>Camera</code> to an empty one.
    */
   public void resetImage() {
      image = new BufferedImage(Camera.DEFAULT_IMAGE_WIDTH,
                           Camera.DEFAULT_IMAGE_HEIGHT,
                           BufferedImage.TYPE_INT_RGB);
   }

   /**
    * Returns the color of the specified pixel in the image of this camera.
    *
    * @param x the X-coordinate of the pixel.
    * @param y the Y-coordinate of the pixel.
    * @return the color of the specified pixel as an int with ARGB format and
    *          8 bits per component. If the supplied coordinates lie outside
    *       of the image, Integer.MAX_VALUE is returned.
    */
   public int getRGB(int x, int y) {
      if (x < 0 || x > image.getWidth()-1 || y < 0 || y > image.getHeight()-1) {
         return Integer.MAX_VALUE;
      }
      else {
         return image.getRGB(x, y);
      }
   }
}
```

CAMERADIALOG

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.awt.image.*;
import java.io.*;
import javax.swing.border.*;

/**
 * This class is a dialog used to create new <code>Camera</code>s and
 * edit existing ones.
 */

public class CameraDialog extends JDialog implements ActionListener {

   private JPanel mainpane;
   private JButton chooseFileButton, saveButton;
   private JFileChooser fc;
   private ExampleFileFilter fileFilter;
   private Frame parent;
   private boolean firstTimeOpened;
   private ImageIcon thumbnail;
   private String imageFilename;
   private BufferedImage image;
   private JLabel thumb;

   private JLabel titleLabel;
   private JTextField titleField;

   private JPanel posPanel;
```

```
private JLabel xPosLabel, yPosLabel, zPosLabel;
private JTextField xPosField, yPosField, zPosField;

private JPanel dirPanel;
private JLabel xDirLabel, yDirLabel, zDirLabel;
private JTextField xDirField, yDirField, zDirField;

private JPanel upPanel;
private JLabel xUpLabel, yUpLabel, zUpLabel;
private JTextField xUpField, yUpField, zUpField;

private JPanel miscPanel;
private JLabel focalLengthLabel, fovLabel;
private JTextField focalLengthField, fovField;

private boolean confirmPressed;
private byte currentMode;
private final byte NONE = 0;
private final byte NEW_CAMERA = 1;
private final byte EDIT_CAMERA = 2;

private Camera currentCamera;

/**
 * Creates an empty <code>CameraDialog</code>.
 */
public CameraDialog() {
   this(null, true);
}

/**
 * Creates a <code>CameraDialog</code> with the specified parent frame
 * and modal setting.
 *
 * @param parent the parent <code>Frame</code> of this dialog.
 * @param modal whether the dialog should be modal or not.
 */
public CameraDialog(Frame parent, boolean modal) {
   super(parent, modal);
   this.parent = parent;

   firstTimeOpened = true;
   confirmPressed = false;
   currentMode = NONE;

   mainpane = new JPanel();
   setContentPane(mainpane);

   titleLabel = new JLabel("Title:");
   titleLabel.setFont(IBVHApplication.DEFAULT_FONT);
   titleLabel.setBounds(10, 10, 128, 20);
   mainpane.add(titleLabel);
   titleField = new JTextField();
   titleField.setFont(IBVHApplication.DEFAULT_FONT);
   titleField.setBounds(10, 30, 128, 20);
   mainpane.add(titleField);

   chooseFileButton = new JButton("Choose file...");
   chooseFileButton.setFont(IBVHApplication.DEFAULT_FONT);
   chooseFileButton.setBounds(10, 60, 128, 20);
   chooseFileButton.addActionListener(this);
   mainpane.add(chooseFileButton);

   thumbnail = new ImageIcon();
   thumb = new JLabel(thumbnail);
   thumb.setBounds(10, 90, 128, 96);
   mainpane.add(thumb);
   thumb.setVisible(false);

   fileFilter = new ExampleFileFilter();
   fileFilter.addExtension("jpg");
   fileFilter.setDescription("JPEG Images");

   buildPanels();

   saveButton = new JButton("Save camera");
   saveButton.setFont(IBVHApplication.DEFAULT_FONT);
   saveButton.setBounds(10, 232, 128, 20);
   saveButton.addActionListener(this);
   mainpane.add(saveButton);

   fc = new JFileChooser();
   fc.setFileFilter(fileFilter);

   mainpane.setLayout(null);
   setResizable(false);
   setSize(430, 300);
```

```
}

public void actionPerformed(ActionEvent e) {
   if (e.getSource() == chooseFileButton) {
      int optionVal = fc.showOpenDialog(CameraDialog.this);
      if (optionVal == JFileChooser.APPROVE_OPTION) {
         imageFilename = fc.getSelectedFile().getPath();
         image = ImageControl.getBufferedImage(imageFilename, mainpane);
         thumbnail.setImage(image.getScaledInstance(128,
                                    96,
                                    Image.SCALE_FAST));
         thumb.setVisible(true);
      }
   }
   if (e.getSource() == saveButton) {
      if (currentMode == EDIT_CAMERA) {
         // Save changes to the loaded camera
         saveCurrentCamera();
      }
      confirmPressed = true;
      currentMode = NONE;
      this.setVisible(false);
   }
}

/**
 * Shows this <code>CameraDialog</code> as an empty dialog.
 */
public void showNewCamera() {
   setTitle("New camera");
   confirmPressed = false;
   currentMode = NEW_CAMERA;
   if (firstTimeOpened) {
      firstTimeOpened = false;
      centerOverParent();
   }
   setVisible(true);
}

/**
 * Shows this <code>CameraDialog</code> filled with the information from
 * the specified <code>Camera</code>.
 *
 * @param cam the <code>Camera</code> to edit.
 */
public void showEditCamera(Camera cam) {
   setTitle("Edit camera");
   confirmPressed = false;
   currentMode = EDIT_CAMERA;
   if (firstTimeOpened) {
      firstTimeOpened = false;
      centerOverParent();
   }
   editCamera(cam);
   setVisible(true);
}

/**
 * Center this <code>CameraDialog</code> over its parent <code>Frame</code>.
 */
private void centerOverParent() {
   if (parent != null) {
      int parX = parent.getX();
      int parY = parent.getY();
      int parW = parent.getWidth();
      int parH = parent.getHeight();
      int thisW = getWidth();
      int thisH = getHeight();

      int thisX = parX + parW / 2 - thisW / 2;
      int thisY = parY + parH / 2 - thisH / 2;

      setLocation(thisX, thisY);
   }
}

private void buildPanels() {
         // Position panel
   posPanel = new JPanel();
   posPanel.setLayout(null);
   posPanel.setBounds(148, 10, 128, 116);
   posPanel.setBorder(new TitledBorder(LineBorder.createGrayLineBorder(),
                                 "Position",
                                 TitledBorder.DEFAULT_POSITION,
                                 TitledBorder.DEFAULT_JUSTIFICATION,
                                 IBVHApplication.DEFAULT_FONT));
   xPosLabel = new JLabel("X:");
```

```
yPosLabel = new JLabel("Y:");
zPosLabel = new JLabel("Z:");
xPosField = new JTextField();
yPosField = new JTextField();
zPosField = new JTextField();
xPosLabel.setFont(IBVHApplication.DEFAULT_FONT);
yPosLabel.setFont(IBVHApplication.DEFAULT_FONT);
zPosLabel.setFont(IBVHApplication.DEFAULT_FONT);
xPosField.setFont(IBVHApplication.DEFAULT_FONT);
yPosField.setFont(IBVHApplication.DEFAULT_FONT);
zPosField.setFont(IBVHApplication.DEFAULT_FONT);
xPosLabel.setBounds(10, 20, 15, 20);
yPosLabel.setBounds(10, 50, 15, 20);
zPosLabel.setBounds(10, 80, 15, 20);
xPosField.setBounds(25, 20, 90, 20);
yPosField.setBounds(25, 50, 90, 20);
zPosField.setBounds(25, 80, 90, 20);
posPanel.add(xPosLabel);
posPanel.add(yPosLabel);
posPanel.add(zPosLabel);
posPanel.add(xPosField);
posPanel.add(yPosField);
posPanel.add(zPosField);
mainpane.add(posPanel);

// Direction panel
dirPanel = new JPanel();
dirPanel.setLayout(null);
dirPanel.setBounds(286, 10, 128, 116);
dirPanel.setBorder(new TitledBorder(LineBorder.createGrayLineBorder(),
                            "View direction",
                            TitledBorder.DEFAULT_POSITION,
                            TitledBorder.DEFAULT_JUSTIFICATION,
                            IBVHApplication.DEFAULT_FONT));
xDirLabel = new JLabel("X:");
yDirLabel = new JLabel("Y:");
zDirLabel = new JLabel("Z:");
xDirField = new JTextField();
yDirField = new JTextField();
zDirField = new JTextField();
xDirLabel.setFont(IBVHApplication.DEFAULT_FONT);
yDirLabel.setFont(IBVHApplication.DEFAULT_FONT);
zDirLabel.setFont(IBVHApplication.DEFAULT_FONT);
xDirField.setFont(IBVHApplication.DEFAULT_FONT);
yDirField.setFont(IBVHApplication.DEFAULT_FONT);
zDirField.setFont(IBVHApplication.DEFAULT_FONT);
xDirLabel.setBounds(10, 20, 15, 20);
yDirLabel.setBounds(10, 50, 15, 20);
zDirLabel.setBounds(10, 80, 15, 20);
xDirField.setBounds(25, 20, 90, 20);
yDirField.setBounds(25, 50, 90, 20);
zDirField.setBounds(25, 80, 90, 20);
dirPanel.add(xDirLabel);
dirPanel.add(yDirLabel);
dirPanel.add(zDirLabel);
dirPanel.add(xDirField);
dirPanel.add(yDirField);
dirPanel.add(zDirField);
mainpane.add(dirPanel);

// Up panel
upPanel = new JPanel();
upPanel.setLayout(null);
upPanel.setBounds(148, 136, 128, 116);
upPanel.setBorder(new TitledBorder(LineBorder.createGrayLineBorder(),
                            "Up direction",
                            TitledBorder.DEFAULT_POSITION,
                            TitledBorder.DEFAULT_JUSTIFICATION,
                            IBVHApplication.DEFAULT_FONT));
xUpLabel = new JLabel("X:");
yUpLabel = new JLabel("Y:");
zUpLabel = new JLabel("Z:");
xUpField = new JTextField();
yUpField = new JTextField();
zUpField = new JTextField();
xUpLabel.setFont(IBVHApplication.DEFAULT_FONT);
yUpLabel.setFont(IBVHApplication.DEFAULT_FONT);
zUpLabel.setFont(IBVHApplication.DEFAULT_FONT);
xUpField.setFont(IBVHApplication.DEFAULT_FONT);
yUpField.setFont(IBVHApplication.DEFAULT_FONT);
zUpField.setFont(IBVHApplication.DEFAULT_FONT);
xUpLabel.setBounds(10, 20, 15, 20);
yUpLabel.setBounds(10, 50, 15, 20);
zUpLabel.setBounds(10, 80, 15, 20);
xUpField.setBounds(25, 20, 90, 20);
yUpField.setBounds(25, 50, 90, 20);
zUpField.setBounds(25, 80, 90, 20);
```

```
   upPanel.add(xUpLabel);
   upPanel.add(yUpLabel);
   upPanel.add(zUpLabel);
   upPanel.add(xUpField);
   upPanel.add(yUpField);
   upPanel.add(zUpField);
   mainpane.add(upPanel);

   // Misc. panel
   miscPanel = new JPanel();
   miscPanel.setLayout(null);
   miscPanel.setBounds(286, 136, 128, 116);
   miscPanel.setBorder(new TitledBorder(LineBorder.createGrayLineBorder(),
                               "Miscellaneous",
                               TitledBorder.DEFAULT_POSITION,
                               TitledBorder.DEFAULT_JUSTIFICATION,
                               IBVHApplication.DEFAULT_FONT));
   focalLengthLabel = new JLabel("Focal length:");
   fovLabel = new JLabel("Field of view:");
   focalLengthField = new JTextField();
   fovField = new JTextField();
   focalLengthLabel.setFont(IBVHApplication.DEFAULT_FONT);
   fovLabel.setFont(IBVHApplication.DEFAULT_FONT);
   focalLengthField.setFont(IBVHApplication.DEFAULT_FONT);
   fovField.setFont(IBVHApplication.DEFAULT_FONT);
   focalLengthLabel.setBounds(10, 20, 105, 20);
   fovLabel.setBounds(10, 60, 105, 20);
   focalLengthField.setBounds(10, 40, 105, 20);
   fovField.setBounds(10, 80, 105, 20);
   miscPanel.add(focalLengthLabel);
   miscPanel.add(fovLabel);
   miscPanel.add(focalLengthField);
   miscPanel.add(fovField);
   mainpane.add(miscPanel);
}

/**
 * Clears all the fields in this <code>CameraDialog</code>.
 */
public void clearAll() {
   titleField.setText("");
   xPosField.setText("");
   yPosField.setText("");
   zPosField.setText("");
   xDirField.setText("");
   yDirField.setText("");
   zDirField.setText("");
   xUpField.setText("");
   yUpField.setText("");
   zUpField.setText("");
   focalLengthField.setText("");
   fovField.setText("");

   thumb.setVisible(false);
   image = null;
}

/**
 * Returns whether the confirm (save) button was pressed or not.
 *
 * @return true if the confirm button was pressed.
 */
public boolean isConfirmed() {
   return confirmPressed;
}

/**
 * Returns a new <code>Camera</code> based on the information that is
 * currently contained in this <code>CameraDialog</code>.
 *
 * @return a new <code>Camera</code> object.
 */
public Camera getNewCamera() {
   return new Camera(titleField.getText(),
                new Vector4(Float.parseFloat(xPosField.getText()),
                            Float.parseFloat(yPosField.getText()),
                            Float.parseFloat(zPosField.getText())),
                new Vector4(Float.parseFloat(xDirField.getText()),
                            Float.parseFloat(yDirField.getText()),
                            Float.parseFloat(zDirField.getText())),
                new Vector4(Float.parseFloat(xUpField.getText()),
                            Float.parseFloat(yUpField.getText()),
                            Float.parseFloat(zUpField.getText())),
                Float.parseFloat(focalLengthField.getText()),
                Float.parseFloat(fovField.getText()),
                image);
}
```

```
    /**
     * Fills this <code>CameraDialog</code> with the information from the
     * specified <code>Camera</code> object.
     *
     * @param c the <code>Camera</code> containing the desired information.
     */
    private void editCamera(Camera c) {
        titleField.setText(c.getTitle());
        xPosField.setText(Float.toString(c.getPos().x));
        yPosField.setText(Float.toString(c.getPos().y));
        zPosField.setText(Float.toString(c.getPos().z));
        xDirField.setText(Float.toString(c.getDir().x));
        yDirField.setText(Float.toString(c.getDir().y));
        zDirField.setText(Float.toString(c.getDir().z));
        xUpField.setText(Float.toString(c.getUp().x));
        yUpField.setText(Float.toString(c.getUp().y));
        zUpField.setText(Float.toString(c.getUp().z));
        focalLengthField.setText(Float.toString(c.getFocalLength()));
        fovField.setText(Float.toString(c.getFov()));
        image = c.getImage();
        thumbnail.setImage(image.getScaledInstance(128,
                                        96,
                                        Image.SCALE_FAST));
        thumb.setVisible(true);

        currentCamera = c;
    }

    /**
     * Updates the current <code>Camera</code> with the current information
     * of this <code>CameraDialog</code>.
     */
    private void saveCurrentCamera() {
        currentCamera.update(titleField.getText(),
                        new Vector4(Float.parseFloat(xPosField.getText()),
                                    Float.parseFloat(yPosField.getText()),
                                    Float.parseFloat(zPosField.getText())),
                        new Vector4(Float.parseFloat(xDirField.getText()),
                                       Float.parseFloat(yDirField.getText()),
                                       Float.parseFloat(zDirField.getText())),
                        new Vector4(Float.parseFloat(xUpField.getText()),
                                    Float.parseFloat(yUpField.getText()),
                                    Float.parseFloat(zUpField.getText())),
                        Float.parseFloat(focalLengthField.getText()),
                        Float.parseFloat(fovField.getText()),
                        image);
    }
}
```

CAMERAROT

```
/**
 * This is a class that contains methods to create transform matrices that
 * can transform our camera to (0,0,0), its direction to (0,0,-1) and its
 * up direction to (0,1,0)
 *
 */

public class CameraRot {

    static Matrix4 rotMatrix, invRotMatrix, tlMatrix, invTlMatrix;
    static Vector4 desiredDir;

    /**
     * This is a static method that creates the transformation matrices
     * used to transform the objects in 3D space.
     *
     * @param c the <code>Camera</code>
     */
    public static void createMatrices(Camera c) {
        desiredDir = new Vector4(0, 0, -1);
        calculateMatrices(c, desiredDir);
    }

    public static void createMatrices(Camera c, Vector4 desiredDir) {
        calculateMatrices(c, desiredDir);
    }

    public static void calculateMatrices(Camera c, Vector4 desiredDir){

        Vector4 dir = c.getDir();
        Vector4 up = c.getUp();
        Vector4 rotAxis;
        Vector4 upTemp = new Vector4(up.x, up.y, up.z);
        Vector4 pos = c.getPos();
        rotMatrix = new Matrix4();
```

```
   invRotMatrix = new Matrix4();
   tlMatrix = new Matrix4();
   invTlMatrix = new Matrix4();
   float angle;

   Matrix4.getTranslateInstance(tlMatrix, -pos.x, -pos.y, -pos.z);
   Matrix4.getTranslateInstance(invTlMatrix, pos.x, pos.y, pos.z);

   //Special cases if the diretion and desired direction are parallel
   if(dir.dotProduct(desiredDir) == 1) {
      angle = 0;
      rotAxis = new Vector4();
   }
   else if(dir.dotProduct(desiredDir) == -1) {
      angle = (float)Math.PI;
      rotAxis = new Vector4(0,1,0);
   }
   else {
      angle = (float)Math.acos(dir.dotProduct(desiredDir));
      rotAxis = Vector4.crossProduct(dir,desiredDir);
      rotAxis.normalize();
   }

   float cos = (float)Math.cos(angle);
   float sin = (float)Math.sin(angle);

   float u = rotAxis.x;
   float v = rotAxis.y;
   float w = rotAxis.z;

   rotMatrix.a11 =      cos + u*u*(1-cos);
   rotMatrix.a21 =  w * sin + v*u*(1-cos);
   rotMatrix.a31 = -v * sin + w*u*(1-cos);
   rotMatrix.a12 = -w * sin + u*v*(1-cos);
   rotMatrix.a22 =      cos + v*v*(1-cos);
   rotMatrix.a32 =  u * sin + w*v*(1-cos);
   rotMatrix.a13 =  v * sin + u*w*(1-cos);
   rotMatrix.a23 = -u * sin + v*w*(1-cos);
   rotMatrix.a33 =      cos + w*w*(1-cos);

   cos = (float)Math.cos(-angle);
   sin = (float)Math.sin(-angle);

   invRotMatrix.a11 =      cos + u*u*(1-cos);
   invRotMatrix.a21 =  w * sin + v*u*(1-cos);
   invRotMatrix.a31 = -v * sin + w*u*(1-cos);
   invRotMatrix.a12 = -w * sin + u*v*(1-cos);
   invRotMatrix.a22 =      cos + v*v*(1-cos);
   invRotMatrix.a32 =  u * sin + w*v*(1-cos);
   invRotMatrix.a13 =  v * sin + u*w*(1-cos);
   invRotMatrix.a23 = -u * sin + v*w*(1-cos);
   invRotMatrix.a33 =      cos + w*w*(1-cos);

   //Calculates the z angle using a temp vector.
   Matrix4.mult(rotMatrix, upTemp);
   float zAngle = (float)Math.atan2(upTemp.x, upTemp.y);

   rotMatrix = Matrix4.mult(Matrix4.getRotateZInstance(zAngle), rotMatrix);
   invRotMatrix = Matrix4.mult(invRotMatrix,
                       Matrix4.getRotateZInstance(-zAngle));

}

/**
 * This method returns the rotation matrix,
 * rotating the camera direction to (0,0,-1)
 * and up direction to (0,1,0)
 *
 * @return A rotation matrix
 */

public static Matrix4 getRotMatrix() {
   return rotMatrix;
}

/**
 * This method returns the inverse rotation matrix
 * rotating the camera direction and up direction
 * to its initial direction.
 *
 * @return A rotation matrix
 */

public static Matrix4 getInvRotMatrix() {
   return invRotMatrix;
}
```

```
   /**
    * This method returns the translation matrix that translates to (0,0,0).
    *
    * @return A translation matrix
    */

   public static Matrix4 getTlMatrix() {
      return tlMatrix;
   }

   /**
    * This method returns the translation matrix that translates
    * to original position.
    *
    * @return A translation matrix
    */

   public static Matrix4 getInvTlMatrix() {
      return invTlMatrix;
   }

}
```

COMPUTE3DRAY

```
/**
 * This is a class that creates rays in 3d space
 */
public class Compute3DRay {

   /**
    * Creates a new <code>Ray3D</code> that starts in the supplied
    * <code>Cameras</code> position and get a direction depending on through
    * which pixel it is choosen to go.
    *
    * @param camera the camera from who the ray get its starting point
    * @param x the pixel number in horizontal way
    * @param y the pixel number in vertical way
    *
    * @return the ray.
    */
   static Ray3D compute3DRay(Camera camera, int x, int y) {

      Vector4 pos = new Vector4(camera.getPos());
      Vector4 normDir = new Vector4(camera.getDir());
      Vector4 normUp = new Vector4(camera.getUp());
      float fov = camera.getFov();
      float focalLength = camera.getFocalLength();

      //Move (0,0) to the middle of the picture
      int pixelX = x-camera.getImage().getWidth()/2;
      int pixelY = camera.getImage().getHeight()/2 - y;

      // (0,0, focal length) i 3d-coordinates
      Vector4 centrumPoint = new Vector4();

      centrumPoint.x = pos.x + normDir.x * focalLength;
      centrumPoint.y = pos.y + normDir.y * focalLength;
      centrumPoint.z = pos.z + normDir.z * focalLength;

      //distance in 2d space between (0,0) and given pixel
      float dxIn2d = (float)Math.tan(fov/2) * focalLength * 2 * pixelX/
                     camera.getImage().getWidth();
      float dyIn2d = (float)Math.tan(fov/2) * focalLength * 2 * pixelY/
                     camera.getImage().getHeight();

      Vector4 sideVector = Vector4.crossProduct(normUp, normDir);
        sideVector.normalize();

      Vector4 deltaIn3d = new Vector4();

      //distance in 3d space between (0,0, focal length) and given pixel
      deltaIn3d.x = -sideVector.x * dxIn2d + normUp.x * dyIn2d;
      deltaIn3d.y = -sideVector.y * dxIn2d + normUp.y * dyIn2d;
      deltaIn3d.z = -sideVector.z * dxIn2d + normUp.z * dyIn2d;


      Vector4 pixelIn3D = new Vector4();

      //pixel in 3D coordinates
      pixelIn3D.x = centrumPoint.x + deltaIn3d.x;
      pixelIn3D.y = centrumPoint.y + deltaIn3d.y;
      pixelIn3D.z = centrumPoint.z + deltaIn3d.z;

      Vector4 rayDir = new Vector4();

      rayDir = pixelIn3D.sub(pos);
```

```
      rayDir.normalize();

      Ray3D r3D = new Ray3D(pos,rayDir);

      return r3D;
   }
}
```

IBVHAPPLET

```
import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.util.Vector;
import java.awt.geom.Point2D;
import java.applet.*;
import java.net.URL;
import java.net.MalformedURLException;

/**
 * This is the main class of the project, containing the GUI among other
 * components. Note that this is the applet version, and therefore lacks
 * features like customizable reference cameras etc. For the standalone
 * application, see IBVHApplication.java.
 */

public class IBVHApplet extends Applet implements TreeSelectionListener {

   /** The font used in the GUI */
   public static final Font DEFAULT_FONT = new Font("Verdana", Font.PLAIN, 11);

   private JPanel mainpane;
   private JScrollPane treepane;
   private JTree tree;
   private DefaultTreeModel treeModel;
   private DefaultMutableTreeNode sceneNode, refViewsNode, novelViewNode;
   private Vector refCameras;
   private Camera novelCamera;
   private VisualHullCreator vhcreator;
   private Timer timer;
   private ProgressMonitor progressMonitor;
   private RotatingCube rotatingCube;
   private MouseClickListener clickListener;

   public ImagePanel viewpane;
   private BufferedImage currentImage;

   private CardLayout viewdeck;
   private JPanel viewpanel;

   private boolean cubeVisible = false;

   private float originalPhi, originalTheta, phi, theta;
   private Point2D.Float sumAngle;
   private Vector4 sideVector, upVector, projPoint, upY,
         tempDirection, tempPoint;
   private Matrix4 rotateMatrix, translateMatrix, rotTransMatrix;

   /**
    * Initializes the GUI and data representation.
    */
   public void init() {

      mainpane = new JPanel();
      this.add(mainpane);
      this.setLayout(null);

      refCameras = new Vector(10, 10);

      novelCamera = new Camera("Novel view",
                        new Vector4(0.5f, 0, 0),
                        new Vector4(-1, 0, 0),
                        new Vector4(0, 1, 0),
                        0.043456f,
                        45,
                        new BufferedImage(Camera.DEFAULT_IMAGE_WIDTH,
                          Camera.DEFAULT_IMAGE_HEIGHT,
                          BufferedImage.TYPE_INT_RGB));

      vhcreator = new VisualHullCreator(refCameras, novelCamera);

      timer = new Timer(1000, new TimerListener());
```

```
clickListener = new MouseClickListener();

rotatingCube = new RotatingCube();
rotatingCube.setBackground(new Color(0, 0, 0));
rotatingCube.setSize(256, 256);
rotatingCube.initStuff(clickListener);

sceneNode = new DefaultMutableTreeNode("Scene");
refViewsNode = new DefaultMutableTreeNode("Reference views");
novelViewNode = new DefaultMutableTreeNode(novelCamera);

sceneNode.add(refViewsNode);
sceneNode.add(novelViewNode);

treeModel = new DefaultTreeModel(sceneNode);
tree = new JTree(treeModel);
tree.setFont(DEFAULT_FONT);
tree.setEditable(false);
tree.addTreeSelectionListener(this);

treepane = new JScrollPane(tree);
treepane.setBounds(10, 10, 200, 256);
mainpane.add(treepane);

currentImage = novelCamera.getImage();
viewpane = new ImagePanel();
viewpane.setSize(256, 256);

viewdeck = new CardLayout();

viewpanel = new JPanel();
viewpanel.setBounds(220, 10, 256, 256);
viewpanel.setBackground(Color.white);
viewpanel.add(viewpane);
viewpanel.add(rotatingCube);

viewdeck.addLayoutComponent(viewpane, "View");
viewdeck.addLayoutComponent(rotatingCube, "Cube");

viewpanel.setLayout(viewdeck);
viewpanel.addMouseListener(clickListener);
mainpane.add(viewpanel);

viewdeck.show(viewpanel, "View");

try {
   addCamera(new Camera("Camera 1",
                    new Vector4(0, 0, 0.400f),
                    new Vector4(0, 0, -1),
                    new Vector4(0, 1, 0),
                    0.043456f,
                    45,
                    ImageControl.getBufferedImage(getImage(
 new URL("http://www.fjeldstad.se/ibr/refs/camera01.png")), mainpane)));

   addCamera(new Camera("Camera 2",
                    new Vector4(0.400f, 0, -0.250f),
                    new Vector4(-0.400f, 0, 0.250f),
                    new Vector4(0, 1, 0),
                    0.043456f,
                    45,
                    ImageControl.getBufferedImage(getImage(
 new URL("http://www.fjeldstad.se/ibr/refs/camera02.png")), mainpane)));

   addCamera(new Camera("Camera 3",
                    new Vector4(-0.300f, 0, -0.200f),
                    new Vector4(0.300f, 0, 0.200f),
                    new Vector4(0, 1, 0),
                    0.043456f,
                    45,
                    ImageControl.getBufferedImage(getImage(
 new URL("http://www.fjeldstad.se/ibr/refs/camera03.png")), mainpane)));

   addCamera(new Camera("Camera 4",
                    new Vector4(0.250f, 0.200f, 0.200f),
                    new Vector4(-0.250f, -0.200f, -0.200f),
                    new Vector4(-0.250f, 0.5125f, -0.200f),
                    0.043456f,
                    45,
                    ImageControl.getBufferedImage(getImage(
 new URL("http://www.fjeldstad.se/ibr/refs/camera04.png")), mainpane)));

   addCamera(new Camera("Camera 5",
                    new Vector4(-0.150f, 0.200f, -0.300f),
                    new Vector4(0.150f, -0.200f, 0.300f),
                    new Vector4(0.150f, 0.5625f, 0.300f),
                    0.043456f,
```

```
                    45,
                    ImageControl.getBufferedImage(getImage(
    new URL("http://www.fjeldstad.se/ibr/refs/camera05.png")), mainpane)));

      addCamera(new Camera("Camera 6",
                    new Vector4(-0.300f, -0.150f, 0.200f),
                    new Vector4(0.300f, 0.150f, -0.200f),
                    new Vector4(-0.3196f, 0.92329f, 0.21307f),
                    0.043456f,
                    45,
                    ImageControl.getBufferedImage(getImage(
    new URL("http://www.fjeldstad.se/ibr/refs/camera06.png")), mainpane)));
   }
   catch (MalformedURLException ex) {
   }

   mainpane.setLayout(null);
   setSize(490, 280);
   mainpane.setSize(490, 280);
   mainpane.setVisible(true);
   setVisible(true);
}

public void valueChanged(TreeSelectionEvent e) {
   DefaultMutableTreeNode node =
      (DefaultMutableTreeNode) tree.getLastSelectedPathComponent();

   if (node != null &&
      node != sceneNode &&
      node != refViewsNode &&
      !vhcreator.isRunning()) {

      // Camera camera = (Camera) node.getUserObject();
      // Update view with selected camera
      currentImage = ((Camera) node.getUserObject()).getImage();
      viewdeck.show(viewpanel, "View");
      viewpane.repaint();
   }
}

/**
 * Adds a new <code>Camera</code> to the scene.
 *
 * @param newCamera the new <code>Camera</code>
 */
private void addCamera(Camera newCamera) {
   DefaultMutableTreeNode childNode =
            new DefaultMutableTreeNode(newCamera);
     treeModel.insertNodeInto(childNode, refViewsNode,
                              refViewsNode.getChildCount());
     tree.scrollPathToVisible(new TreePath(childNode.getPath()));
   tree.clearSelection();
   tree.addSelectionPath(
      new TreePath(refViewsNode.getLastLeaf().getPath()));
   refCameras.add(newCamera);
}

/**
 * Starts the calculation of the visual hull based on the information
 * in the available reference cameras.
 */
private void startCalculating() {

   progressMonitor = new ProgressMonitor(mainpane,
                              "Constructing visual hull...",
                              " ",
                              0,
                              100);
   progressMonitor.setProgress(0);
   progressMonitor.setMillisToDecideToPopup(0);
   progressMonitor.setMillisToPopup(0);

   vhcreator.startProcess();
   timer.start();
   tree.setEnabled(false);
   novelCamera.resetImage();
}

private class TimerListener implements ActionListener {
   public void actionPerformed(ActionEvent e) {

      progressMonitor.setProgress(vhcreator.getProgress());
      progressMonitor.setNote(vhcreator.getNote());

      if (progressMonitor.isCanceled() || vhcreator.isFinished()) {
         progressMonitor.close();
         vhcreator.stopProcess();
```

```
            timer.stop();
            currentImage = novelCamera.getImage();
            viewpane.repaint();
            tree.setEnabled(true);
         }
      }
   }

   private class MouseClickListener extends MouseAdapter {

      public void mouseClicked(MouseEvent e) {
         if (!vhcreator.isRunning()) {
            if (!cubeVisible) {
               // Calculate the camera phi angle and send
               // it to rotatingCube
               float length = novelCamera.getPos().getLength();
               originalPhi =
                  (float)Math.acos(novelCamera.getPos().y/length);
               rotatingCube.setCameraPhi(originalPhi);

               cubeVisible = !cubeVisible;
               viewdeck.show(viewpanel, "Cube");
               rotatingCube.repaint();
            }
            else {
               cubeVisible = !cubeVisible;
               currentImage = novelCamera.getImage();
               tree.clearSelection();
               tree.addSelectionPath(
                  new TreePath(novelViewNode.getLastLeaf().getPath()));
               viewdeck.show(viewpanel, "View");
               viewpane.repaint();

               // Fetch the rotation matrix from rotatingCube
               // Rotate desired view
               // Rotate the camera of the novel view

               tempDirection = new Vector4(novelCamera.getDir().x,
                                   novelCamera.getDir().y,
                                   novelCamera.getDir().z, 1);

               tempPoint = new Vector4(novelCamera.getPos().x,
                                 novelCamera.getPos().y,
                                 novelCamera.getPos().z, 1);

               float length = novelCamera.getPos().getLength();
               sumAngle = rotatingCube.getAngles();

               //Calculate new pos
               originalTheta = (float)Math.atan2(novelCamera.getPos().x,
                                        novelCamera.getPos().z);
               originalPhi =
                  (float)Math.acos(novelCamera.getPos().y / length);
               novelCamera.getPos().setVector4(
                  (float)(length*Math.sin(sumAngle.y + originalPhi) *
                     Math.sin(originalTheta + sumAngle.x)),
                  (float)(length*Math.cos(sumAngle.y + originalPhi)),
                  (float)(length*Math.sin(sumAngle.y + originalPhi) *
                     Math.cos(originalTheta + sumAngle.x)),
                  1);

               //Calculate new dir
               novelCamera.getDir().setVector4(-novelCamera.getPos().x,
                                       -novelCamera.getPos().y,
                                       -novelCamera.getPos().z,
                                       1);
               novelCamera.getDir().normalize();

               //Calculate new up
               upY = new Vector4(0,1,0,1);
               projPoint = new Vector4(novelCamera.getDir().x,
                                 0,
                                 novelCamera.getDir().z,
                                 1);
               sideVector = Vector4.crossProduct(projPoint, upY);
               upVector = Vector4.crossProduct(sideVector,
                                       novelCamera.getDir());
               novelCamera.getUp().setVector4(upVector.x,
                                       upVector.y,
                                       upVector.z,
                                       1);

               //Calculate the rotatation matrix
               CameraRot.createMatrices(novelCamera, tempDirection);
               rotateMatrix = CameraRot.getInvRotMatrix();

               vhcreator = new VisualHullCreator(refCameras, novelCamera);
```

```
                rotatingCube.setOrginalCube();
                rotatingCube.resetAngles();
                rotatingCube.setFirstTime(true);

                // Calculate visual hull
                startCalculating();
             }
          }
       }
    }

    class ImagePanel extends JPanel {
       public void paintComponent(Graphics g) {
          super.paintComponent(g);
          g.setColor(Color.white);
          g.drawRect(0,0,getWidth(), getHeight());
          g.drawImage(currentImage, 0, 0, this);
       }
    }
}
```

IBVHAPPLICATION

```
import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.util.Vector;
import java.awt.geom.Point2D;


/**
 * This is the main class of the project, containing the GUI among other
 * components.
 */

public class IBVHApplication extends JFrame implements MouseListener,
                                          ActionListener,
                                          TreeSelectionListener {

   /** The font used in the GUI */
   public static final Font DEFAULT_FONT = new Font("Verdana", Font.PLAIN, 11);

   private JPanel mainpane;
   private JScrollPane treepane;
   private JTree tree;
   private DefaultTreeModel treeModel;
   private DefaultMutableTreeNode sceneNode, refViewsNode, novelViewNode;
   private JPopupMenu rightMenuOne, rightMenuTwo;
   private JMenuItem newCamera, editCamera, deleteCamera;
   private CameraDialog cameraDialog;
   private Vector refCameras;
   private Camera novelCamera;
   private VisualHullCreator vhcreator;
   private Timer timer;
   private ProgressMonitor progressMonitor;
   private RotatingCube rotatingCube;
   private MouseClickListener clickListener;

   public ImagePanel viewpane;
   private BufferedImage currentImage;

   private Rectangle imageRect;
   private TexturePaint imagePaint;

   private CardLayout viewdeck;
   private JPanel viewpanel;

   private boolean cubeVisible = false;

   private float originalPhi, originalTheta, phi, theta;
   private Point2D.Float sumAngle;
   private Vector4 sideVector, upVector, projPoint, upY,
         tempDirection, tempPoint;
   private Matrix4 rotateMatrix, translateMatrix, rotTransMatrix;

   /**
    * Initializes the GUI and data representation.
    */
   public IBVHApplication() {
      cameraDialog = new CameraDialog(this, true);

      mainpane = new JPanel();
      setContentPane(mainpane);
```

```
refCameras = new Vector(10, 10);

novelCamera = new Camera("Novel view",
                  new Vector4(0.5f, 0, 0),
                  new Vector4(-1, 0, 0),
                  new Vector4(0, 1, 0),
                  0.043456f,
                  45,
                  new BufferedImage(Camera.DEFAULT_IMAGE_WIDTH,
                    Camera.DEFAULT_IMAGE_HEIGHT,
                    BufferedImage.TYPE_INT_RGB));

vhcreator = new VisualHullCreator(refCameras, novelCamera);

timer = new Timer(1000, new TimerListener());

clickListener = new MouseClickListener();

rotatingCube = new RotatingCube();
rotatingCube.setBackground(new Color(0, 0, 0));
rotatingCube.setSize(256, 256);
rotatingCube.initStuff(clickListener);

sceneNode = new DefaultMutableTreeNode("Scene");
refViewsNode = new DefaultMutableTreeNode("Reference views");
novelViewNode = new DefaultMutableTreeNode(novelCamera);

sceneNode.add(refViewsNode);
sceneNode.add(novelViewNode);

treeModel = new DefaultTreeModel(sceneNode);
tree = new JTree(treeModel);
tree.setFont(DEFAULT_FONT);
tree.setEditable(false);
tree.addMouseListener(this);
tree.addTreeSelectionListener(this);

treepane = new JScrollPane(tree);
treepane.setBounds(10, 10, 200, 256);
mainpane.add(treepane);

currentImage = novelCamera.getImage();
viewpane = new ImagePanel();
viewpane.setSize(256, 256);

viewdeck = new CardLayout();

viewpanel = new JPanel();
viewpanel.setBounds(220, 10, 256, 256);
viewpanel.setBackground(Color.white);
viewpanel.add(viewpane);
viewpanel.add(rotatingCube);

viewdeck.addLayoutComponent(viewpane, "View");
viewdeck.addLayoutComponent(rotatingCube, "Cube");

viewpanel.setLayout(viewdeck);
viewpanel.addMouseListener(clickListener);
mainpane.add(viewpanel);

rightMenuOne = new JPopupMenu();
rightMenuTwo = new JPopupMenu();
newCamera = new JMenuItem("New camera...");
editCamera = new JMenuItem("Edit camera...");
deleteCamera = new JMenuItem("Delete camera");
newCamera.setFont(DEFAULT_FONT);
editCamera.setFont(DEFAULT_FONT);
deleteCamera.setFont(DEFAULT_FONT);

rightMenuOne.add(newCamera);
rightMenuTwo.add(editCamera);
rightMenuTwo.add(deleteCamera);

newCamera.addActionListener(this);
editCamera.addActionListener(this);
deleteCamera.addActionListener(this);

viewdeck.show(viewpanel, "View");

mainpane.setLayout(null);
setSize(490, 300);
setResizable(false);
setTitle("Image-based Visual Hulls");
setVisible(true);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
   addCamera(new Camera("Camera 1",
                    new Vector4(0, 0, 0.400f),
                    new Vector4(0, 0, -1),
                    new Vector4(0, 1, 0),
                    0.043456f,
                    45,
                    ImageControl.getBufferedImage(
                      "./refs/camera01.png", mainpane)));

   addCamera(new Camera("Camera 2",
                    new Vector4(0.400f, 0, -0.250f),
                    new Vector4(-0.400f, 0, 0.250f),
                    new Vector4(0, 1, 0),
                    0.043456f,
                    45,
                    ImageControl.getBufferedImage(
                      "./refs/camera02.png", mainpane)));

   addCamera(new Camera("Camera 3",
                    new Vector4(-0.300f, 0, -0.200f),
                    new Vector4(0.300f, 0, 0.200f),
                    new Vector4(0, 1, 0),
                    0.043456f,
                    45,
                    ImageControl.getBufferedImage(
                      "./refs/camera03.png", mainpane)));

   addCamera(new Camera("Camera 4",
                    new Vector4(0.250f, 0.200f, 0.200f),
                    new Vector4(-0.250f, -0.200f, -0.200f),
                    new Vector4(-0.250f, 0.5125f, -0.200f),
                    0.043456f,
                    45,
                    ImageControl.getBufferedImage(
                      "./refs/camera04.png", mainpane)));

   addCamera(new Camera("Camera 5",
                    new Vector4(-0.150f, 0.200f, -0.300f),
                    new Vector4(0.150f, -0.200f, 0.300f),
                    new Vector4(0.150f, 0.5625f, 0.300f),
                    0.043456f,
                    45,
                    ImageControl.getBufferedImage(
                      "./refs/camera05.png", mainpane)));

   addCamera(new Camera("Camera 6",
                    new Vector4(-0.300f, -0.150f, 0.200f),
                    new Vector4(0.300f, 0.150f, -0.200f),
                    new Vector4(-0.3196f, 0.92329f, 0.21307f),
                    0.043456f,
                    45,
                    ImageControl.getBufferedImage(
                      "./refs/camera06.png", mainpane)));
}

public void mouseClicked(MouseEvent e) {
   TreePath path = tree.getPathForLocation(e.getX(), e.getY());
   if (path == null) {
      tree.clearSelection();
   }

   if (e.getButton() == MouseEvent.BUTTON2 ||
       e.getButton() == MouseEvent.BUTTON3) {

      if ((path != null &&
           path.getLastPathComponent() != sceneNode) ||
          path == null) {

         if (path == null ||
            path.getLastPathComponent() == refViewsNode) {

            if (path != null &&
                path.getLastPathComponent() == refViewsNode) {

               tree.clearSelection();
               tree.addSelectionPath(path);
            }
            // Insert new node
            rightMenuOne.show(mainpane, e.getX(), e.getY());
         } else {
            // Highlight the node clicked on
            tree.clearSelection();
            tree.addSelectionPath(path);
            if (path.getLastPathComponent() != novelViewNode) {
               // Display options for node
               rightMenuTwo.show(mainpane, e.getX(), e.getY());
            }
```

```
            }
         }
      }
   }

   public void actionPerformed(ActionEvent e) {
      if (e.getSource() == newCamera) {
         cameraDialog.showNewCamera();
         // If the user pressed OK, get the info from the dialog and
         // create a new Camera object, adding it to the tree.
         if (cameraDialog.isConfirmed()) {
            addCamera(cameraDialog.getNewCamera());
            cameraDialog.clearAll();
         }
      }
      if (e.getSource() == editCamera) {
         DefaultMutableTreeNode selected =
            (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
         Camera c = (Camera)selected.getUserObject();
         cameraDialog.showEditCamera(c);
      }
      if (e.getSource() == deleteCamera) {
         DefaultMutableTreeNode selected =
            (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
         Camera c = (Camera)selected.getUserObject();
         if (JOptionPane.showConfirmDialog(mainpane,
                                 "Do you really want to delete " +
                                 "the camera?",
                                 "Delete camera",
                                 JOptionPane.YES_NO_OPTION,
                                 JOptionPane.QUESTION_MESSAGE) ==
            JOptionPane.YES_OPTION) {

            treeModel.removeNodeFromParent(selected);
            refCameras.remove(c);
         }
      }
   }

   public void valueChanged(TreeSelectionEvent e) {
      DefaultMutableTreeNode node =
         (DefaultMutableTreeNode) tree.getLastSelectedPathComponent();

      if (node != null &&
         node != sceneNode &&
         node != refViewsNode &&
         !vhcreator.isRunning()) {

         // Camera camera = (Camera) node.getUserObject();
         // Update view with selected camera
         currentImage = ((Camera) node.getUserObject()).getImage();
         viewdeck.show(viewpanel, "View");
         viewpane.repaint();
      }
   }

   /**
    * Adds a new <code>Camera</code> to the scene.
    *
    * @param newCamera the new <code>Camera</code>
    */
   private void addCamera(Camera newCamera) {
      DefaultMutableTreeNode childNode =
                new DefaultMutableTreeNode(newCamera);
        treeModel.insertNodeInto(childNode, refViewsNode,
                                 refViewsNode.getChildCount());
        tree.scrollPathToVisible(new TreePath(childNode.getPath()));
      tree.clearSelection();
      tree.addSelectionPath(
         new TreePath(refViewsNode.getLastLeaf().getPath()));
      refCameras.add(newCamera);
   }

   /**
    * Starts the calculation of the visual hull based on the information
    * in the available reference cameras.
    */
   private void startCalculating() {

      progressMonitor = new ProgressMonitor(IBVHApplication.this,
                                 "Constructing visual hull...",
                                 " ",
                                 0,
                                 100);
      progressMonitor.setProgress(0);
      progressMonitor.setMillisToDecideToPopup(0);
      progressMonitor.setMillisToPopup(0);
```

```
   vhcreator.startProcess();
   timer.start();
   tree.setEnabled(false);
   novelCamera.resetImage();
}

private class TimerListener implements ActionListener {
   public void actionPerformed(ActionEvent e) {

      progressMonitor.setProgress(vhcreator.getProgress());
      progressMonitor.setNote(vhcreator.getNote());

      if (progressMonitor.isCanceled() || vhcreator.isFinished()) {
         progressMonitor.close();
         vhcreator.stopProcess();
         timer.stop();
         currentImage = novelCamera.getImage();
         viewpane.repaint();
         tree.setEnabled(true);
      }
   }
}

private class MouseClickListener extends MouseAdapter {

   public void mouseClicked(MouseEvent e) {
      if (!vhcreator.isRunning()) {
         if (!cubeVisible) {
            // Calculate the camera phi angle and
            // send it to rotatingCube
            float length = novelCamera.getPos().getLength();
            originalPhi =
               (float)Math.acos(novelCamera.getPos().y / length);
            rotatingCube.setCameraPhi(originalPhi);

            cubeVisible = !cubeVisible;
            viewdeck.show(viewpanel, "Cube");
            rotatingCube.repaint();
         }
         else {
            cubeVisible = !cubeVisible;
            currentImage = novelCamera.getImage();
            tree.clearSelection();
            tree.addSelectionPath(
               new TreePath(novelViewNode.getLastLeaf().getPath()));
            viewdeck.show(viewpanel, "View");
            viewpane.repaint();

            // Fetch the rotation matrix from rotatingCube
            // Rotate desired view
            // Rotate the camera of the novel view

            tempDirection = new Vector4(novelCamera.getDir().x,
                                 novelCamera.getDir().y,
                                 novelCamera.getDir().z,
                                 1);

            tempPoint = new Vector4(novelCamera.getPos().x,
                               novelCamera.getPos().y,
                               novelCamera.getPos().z,
                               1);

            float length = novelCamera.getPos().getLength();
            sumAngle = rotatingCube.getAngles();

            //Calculate new pos
            originalTheta = (float)Math.atan2(novelCamera.getPos().x,
                                        novelCamera.getPos().z);
            originalPhi =
               (float)Math.acos(novelCamera.getPos().y / length);

            novelCamera.getPos().setVector4(
               (float)(length*Math.sin(sumAngle.y + originalPhi) *
                  Math.sin(originalTheta + sumAngle.x)),
               (float)(length*Math.cos(sumAngle.y + originalPhi)),
               (float)(length*Math.sin(sumAngle.y + originalPhi) *
                  Math.cos(originalTheta + sumAngle.x)),
              1);

            //Calculate new dir
            novelCamera.getDir().setVector4(-novelCamera.getPos().x,
                                    -novelCamera.getPos().y,
                                    -novelCamera.getPos().z,
                                    1);
            novelCamera.getDir().normalize();
```

```
                //Calculate new up
                upY = new Vector4(0,1,0,1);
                projPoint = new Vector4(novelCamera.getDir().x,
                                0,
                                novelCamera.getDir().z,
                                1);

                sideVector = Vector4.crossProduct(projPoint, upY);

                upVector = Vector4.crossProduct(sideVector,
                                        novelCamera.getDir());

                novelCamera.getUp().setVector4(upVector.x,
                                        upVector.y,
                                        upVector.z,
                                        1);

                //Calculate the rotate matrix
                CameraRot.createMatrices(novelCamera, tempDirection);
                rotateMatrix = CameraRot.getInvRotMatrix();

                vhcreator = new VisualHullCreator(refCameras, novelCamera);

                rotatingCube.setOrginalCube();
                rotatingCube.resetAngles();
                rotatingCube.setFirstTime(true);

                // Calculate visual hull
                startCalculating();
             }
          }
       }
    }

    class ImagePanel extends JPanel {
       public void paintComponent(Graphics g) {
          super.paintComponent(g);
          g.setColor(Color.white);
          g.drawRect(0,0,getWidth(), getHeight());
          g.drawImage(currentImage, 0, 0, this);
       }
    }

    public void mouseEntered(MouseEvent e) {
    }
    public void mouseExited(MouseEvent e) {
    }
    public void mousePressed(MouseEvent e) {
    }
    public void mouseReleased(MouseEvent e) {
    }

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

IMAGECONTROL

```
import java.awt.*;
import java.awt.image.*;

/**
 * This class icludes methods for reading images from file
 */

public class ImageControl {

    /**
     * Reads a image from file, making it a buffered image.
     *
     * @param imagefile the local file specifing the image
     * @param c a graphical component to use
     * @return the buffered image of the local file
     */
    public static BufferedImage getBufferedImage(String imageFile,
                                                 Component c) {
        Image image = c.getToolkit().getImage(imageFile);
        waitForImage(image, c);

        BufferedImage bufferedImage =
          new BufferedImage(image.getWidth(c), image.getHeight(c),
                            BufferedImage.TYPE_INT_RGB);

        Graphics2D g2d = bufferedImage.createGraphics();
        g2d.drawImage(image, 0, 0, c);
        return(bufferedImage);
```

```
    }

    public static BufferedImage getBufferedImage(Image image, Component c) {
        waitForImage(image, c);

        BufferedImage bufferedImage =
          new BufferedImage(image.getWidth(c), image.getHeight(c),
                            BufferedImage.TYPE_INT_RGB);

        Graphics2D g2d = bufferedImage.createGraphics();
        g2d.drawImage(image, 0, 0, c);
        return(bufferedImage);
    }

    /**
    * Take an Image associated with a file, and wait until it is
    * done loading (just a simple application of MediaTracker).
    * If you are loading multiple images, don't use this
    * consecutive times; instead, use the version that takes
    * an array of images.
    */
    public static boolean waitForImage(Image image, Component c) {
       MediaTracker tracker = new MediaTracker(c);
       tracker.addImage(image, 0);
       try {
         tracker.waitForAll();
       } catch(InterruptedException ie) {}
       return(!tracker.isErrorAny());
    }
}
```

INTERVAL1D

```
/**
 * This class represents a simple scalar one-dimensional interval, together
 * with a method for aquiring the intersection with another interval.
 */

public class Interval1D {

    /** Start point of the interval. */
    public float start;
    /** End point of the interval. */
    public float end;

    /**
     * Creates a new <code>Interval1D</code> spanning the whole real axis.
     */
    public Interval1D() {
       start = Float.NEGATIVE_INFINITY;
       end = Float.POSITIVE_INFINITY;
    }

    /**
     * Creates a new <code>Interval1D</code> spanning the specified range.
     *
     * @param start the start point of the interval.
     * @param end the end point of the interval.
     */
    public Interval1D(float start, float end) {
       this.start = start;
       this.end = end;
    }

    /**
     * Indicates whether this <code>Interval1D</code> is empty.
     *
     * @return true if the interval is empty, false otherwise.
     */
    public boolean isEmpty() {
       return (Float.isNaN(start) || Float.isNaN(end));
    }

    /**
     * Sets this <code>Interval1D</code> to the empty one.
     */
    public void makeEmpty() {
       start = Float.NaN;
       end = Float.NaN;
    }

    /**
     * Returns whether the <code>Interval1D</code> is valid. An interval is
     * considered valid if the end point is greater than the start point. An
     * empty interval is also counted as a valid one.
     *
     * @return true if the interval is valid, false otherwise.
```

```
     */
    public boolean isValid() {
       return ((this.start <= this.end) || this.isEmpty());
    }

    /**
     * Performs the intersection operation between this interval and the
     * supplied one. This interval is updated accordingly while the supplied
     * one is left unchanged.
     *
     * @param other the interval to intersect this one with.
     */
    public void intersection(Interval1D other) {

       if (this.isEmpty() || other.isEmpty()) {
          this.makeEmpty();

       }
       else if (this.isValid() == false || other.isValid() == false) {

          if (this.isValid() == false && other.isValid() == false) {
             this.makeEmpty();

          }
          else if (this.isValid() == false) {
             this.start = other.start;
             this.end = other.end;

          }
          else if (other.isValid() == false) {
             // Do nothing
          }
       }
       else {

          float tempStart, tempEnd;

          // Check whether the intervals intersect at all
          if ((this.start < other.start && this.end < other.start) ||
             (this.start > other.end && this.end > other.end)) {

             this.makeEmpty();
          }
          else {
             if (this.start < other.start) {
                tempStart = other.start;
                if (this.end > other.end) {
                   tempEnd = other.end;
                }
                else {
                   tempEnd = this.end;
                }
             }
             else {
                tempStart = this.start;
                if (this.end > other.end) {
                   tempEnd = other.end;
                }
                else {
                   tempEnd = this.end;
                }
             }
             this.start = tempStart;
             this.end = tempEnd;
          }
       }
    }
    /**
     * Performs the intersection operation between the two supplied
     * <code>Interval1D</code>s. The result is returned as a new
     * <code>Interval1D</code>.
     *
     * @param first the first interval.
     * @param second the second interval.
     * @return a new <code>Interval1D</code> as a result of the operation.
     */
    public static Interval1D intersection(Interval1D first, Interval1D second) {

       Interval1D isect = new Interval1D(first.start, first.end);
       isect.intersection(second);
       return isect;
    }


}
```

MATRIX4

```
/* @(#)Matrix4.java 1.0 01/04/17
 * Stefan Gustavson, ITN-LiTH 2001 (stegu@itn.liu.se)
 * Modified by Jesper Carlson, Anders Fjeldstad, Daniel Enetoft, Kristofer Gärdeborg
 */

public class Matrix4 {

   /**
   * The matrix coefficients. Element <code>aij</code> is at row i, column j.
   */
   public float a11,a12,a13,a14,
         a21,a22,a23,a24,
         a31,a32,a33,a34,
         a41,a42,a43,a44;

   /**
   * Default constructor, yields indentity matrix
   */
   public Matrix4() {
      a11=1.0f; a12=0.0f; a13=0.0f; a14=0.0f;
      a21=0.0f; a22=1.0f; a23=0.0f; a24=0.0f;
      a31=0.0f; a32=0.0f; a33=1.0f; a34=0.0f;
      a41=0.0f; a42=0.0f; a43=0.0f; a44=1.0f;
   }

   /**
   * Matrix copy constructor, copies an existing matrix
   */
   public Matrix4(Matrix4 m) {
      a11=m.a11; a12=m.a12; a13=m.a13; a14=m.a14;
      a21=m.a21; a22=m.a22; a23=m.a23; a24=m.a24;
      a31=m.a31; a32=m.a32; a33=m.a33; a34=m.a34;
      a41=m.a41; a42=m.a42; a43=m.a43; a44=m.a44;
   }

   /**
   * Factory method to create a transformation matrix for rotation around X
   * @param theta The rotation angle
   * @return A transformation matrix
   */
   public static Matrix4 getRotateXInstance(float theta) {
      Matrix4 m = new Matrix4();
      m.a22=(float)Math.cos(theta);     m.a23=(float)Math.sin(-theta);
      m.a32=(float)Math.sin(theta);     m.a33=(float)Math.cos(theta);
      return m;
   }

   /**
   * Factory method to create a transformation matrix for rotation around Y
   * @param theta The rotation angle
   * @return A transformation matrix
   */
   public static Matrix4 getRotateYInstance(float theta) {
      Matrix4 m = new Matrix4();
      m.a11=(float)Math.cos(theta);  m.a13=(float)Math.sin(theta);
      m.a31=(float)Math.sin(-theta);    m.a33=(float)Math.cos(theta);
      return m;
   }

   /**
   * Factory method to create a transformation matrix for rotation around Z
   * @param theta The rotation angle
   * @return A transformation matrix
   */
   public static Matrix4 getRotateZInstance(float theta) {
      Matrix4 m = new Matrix4();
      m.a11=(float)Math.cos(theta);     m.a12=(float)Math.sin(-theta);
      m.a21=(float)Math.sin(theta);     m.a22=(float)Math.cos(theta);
      return m;
   }

   /**
   * Factory method to create a transformation matrix for translation
   * @param tx The translation along the x dimension
   * @param ty The translation along the y dimension
   * @param tz The translation along the z dimension
   * @return A transformation matrix
   */
   public static void getTranslateInstance(Matrix4 m,
                               float tx,
                               float ty,
                               float tz) {
      m.a14=tx; m.a24=ty; m.a34=tz;
   }
```

```
/**
* Factory method to create a transformation matrix for uniform scaling
* @param s The scaling factor
* @return A transformation matrix
*/
public static Matrix4 getScaleInstance(float s) {
   Matrix4 m = new Matrix4();
   m.a11=s;
   m.a22=s;
   m.a33=s;
   return m;
}

/**
* Factory method to create a transformation matrix for non-uniform scaling
* @param sx The scaling factor along the x dimension
* @param sy The scaling factor along the y dimension
* @param sz The scaling factor along the z dimension
* @return A transformation matrix
*/
public static Matrix4 getScaleInstance(float sx, float sy, float sz) {
   Matrix4 m = new Matrix4();
   m.a11=sx;
   m.a22=sy;
   m.a33=sz;
   return m;
}

/**
* Copy the coefficient vaules from another matrix
* @param m The matrix to copy
*/
public void set(Matrix4 m) {
   a11=m.a11; a12=m.a12; a13=m.a13; a14=m.a14;
   a21=m.a21; a22=m.a22; a23=m.a23; a24=m.a24;
   a31=m.a31; a32=m.a32; a33=m.a33; a34=m.a34;
   a41=m.a41; a42=m.a42; a43=m.a43; a44=m.a44;
}



/**
* Rotating matrix
* @param m1 The first (left) matrix for the multiplication
* @param m2 The second (middle) matrix for the multiplication
* @param m3 The third (right) matrix for the multiplication
* @return The resulting matrix product
*/
public Matrix4 getRotMatrix(float phi, float theta, float sigma) {

   Matrix4 m = new Matrix4();
   m.getRotateXInstance(phi);
   this.mult(m);
   m.getRotateYInstance(theta);
   this.mult(m);
   m.getRotateZInstance(sigma);
   this.mult(m);
   return this;
}

/* Matrix multiply with a vector
 * @param v The vector for the multiplication
 * @return The resulting vector
 */
public static void mult(Matrix4 m, Vector4 v) {
   float x, y, z, w;
   x = m.a11*v.x+m.a12*v.y+m.a13*v.z+m.a14*v.w;
   y = m.a21*v.x+m.a22*v.y+m.a23*v.z+m.a24*v.w;
   z = m.a31*v.x+m.a32*v.y+m.a33*v.z+m.a34*v.w;
   w = m.a41*v.x+m.a42*v.y+m.a43*v.z+m.a44*v.w;
   v.x = x; v.y = y; v.z = z; v.w = w;
}


/**
* Matrix multiply two matrices
* @param m1 The first (left) matrix for the multiplication
* @param m2 The second (right) matrix for the multiplication
* @return The resulting matrix product
*/
public static Matrix4 mult(Matrix4 m1, Matrix4 m2) {
   Matrix4 m3 = new Matrix4();
   // An array for a(i,j) and loops would make this code more compact,
   // but slower.
   m3.a11 = m1.a11*m2.a11+m1.a12*m2.a21+m1.a13*m2.a31+m1.a14*m2.a41;
   m3.a12 = m1.a11*m2.a12+m1.a12*m2.a22+m1.a13*m2.a32+m1.a14*m2.a42;
   m3.a13 = m1.a11*m2.a13+m1.a12*m2.a23+m1.a13*m2.a33+m1.a14*m2.a43;
```

```
      m3.a14 = m1.a11*m2.a14+m1.a12*m2.a24+m1.a13*m2.a34+m1.a14*m2.a44;

      m3.a21 = m1.a21*m2.a11+m1.a22*m2.a21+m1.a23*m2.a31+m1.a24*m2.a41;
      m3.a22 = m1.a21*m2.a12+m1.a22*m2.a22+m1.a23*m2.a32+m1.a24*m2.a42;
      m3.a23 = m1.a21*m2.a13+m1.a22*m2.a23+m1.a23*m2.a33+m1.a24*m2.a43;
      m3.a24 = m1.a21*m2.a14+m1.a22*m2.a24+m1.a23*m2.a34+m1.a24*m2.a44;

      m3.a31 = m1.a31*m2.a11+m1.a32*m2.a21+m1.a33*m2.a31+m1.a34*m2.a41;
      m3.a32 = m1.a31*m2.a12+m1.a32*m2.a22+m1.a33*m2.a32+m1.a34*m2.a42;
      m3.a33 = m1.a31*m2.a13+m1.a32*m2.a23+m1.a33*m2.a33+m1.a34*m2.a43;
      m3.a34 = m1.a31*m2.a14+m1.a32*m2.a24+m1.a33*m2.a34+m1.a34*m2.a44;

      m3.a41 = m1.a41*m2.a11+m1.a42*m2.a21+m1.a43*m2.a31+m1.a44*m2.a41;
      m3.a42 = m1.a41*m2.a12+m1.a42*m2.a22+m1.a43*m2.a32+m1.a44*m2.a42;
      m3.a43 = m1.a41*m2.a13+m1.a42*m2.a23+m1.a43*m2.a33+m1.a44*m2.a43;
      m3.a44 = m1.a41*m2.a14+m1.a42*m2.a24+m1.a43*m2.a34+m1.a44*m2.a44;

      return m3;
   }

    /**
    * Matrix multiply two matrices
    * @param m2 The second (right) matrix for the multiplication
    * @return The resulting matrix product
    */
   public void mult(Matrix4 m2) {
      float f11,f12,f13,f14,f21,f22,f23,f24,f31,f32,f33,f34,f41,f42,f43,f44;
      f11 = this.a11*m2.a11+this.a12*m2.a21+this.a13*m2.a31+this.a14*m2.a41;
      f12 = this.a11*m2.a12+this.a12*m2.a22+this.a13*m2.a32+this.a14*m2.a42;
      f13 = this.a11*m2.a13+this.a12*m2.a23+this.a13*m2.a33+this.a14*m2.a43;
      f14 = this.a11*m2.a14+this.a12*m2.a24+this.a13*m2.a34+this.a14*m2.a44;

      f21 = this.a21*m2.a11+this.a22*m2.a21+this.a23*m2.a31+this.a24*m2.a41;
      f22 = this.a21*m2.a12+this.a22*m2.a22+this.a23*m2.a32+this.a24*m2.a42;
      f23 = this.a21*m2.a13+this.a22*m2.a23+this.a23*m2.a33+this.a24*m2.a43;
      f24 = this.a21*m2.a14+this.a22*m2.a24+this.a23*m2.a34+this.a24*m2.a44;

      f31 = this.a31*m2.a11+this.a32*m2.a21+this.a33*m2.a31+this.a34*m2.a41;
      f32 = this.a31*m2.a12+this.a32*m2.a22+this.a33*m2.a32+this.a34*m2.a42;
      f33 = this.a31*m2.a13+this.a32*m2.a23+this.a33*m2.a33+this.a34*m2.a43;
      f34 = this.a31*m2.a14+this.a32*m2.a24+this.a33*m2.a34+this.a34*m2.a44;

      f41 = this.a41*m2.a11+this.a42*m2.a21+this.a43*m2.a31+this.a44*m2.a41;
      f42 = this.a41*m2.a12+this.a42*m2.a22+this.a43*m2.a32+this.a44*m2.a42;
      f43 = this.a41*m2.a13+this.a42*m2.a23+this.a43*m2.a33+this.a44*m2.a43;
      f44 = this.a41*m2.a14+this.a42*m2.a24+this.a43*m2.a34+this.a44*m2.a44;

      this.a11 = f11; this.a12 = f12; this.a13 = f13; this.a14 = f14;
      this.a21 = f21; this.a22 = f22; this.a23 = f23; this.a24 = f24;
      this.a31 = f31; this.a32 = f32; this.a33 = f33; this.a34 = f34;
      this.a41 = f41; this.a42 = f42; this.a43 = f43; this.a44 = f44;
   }
}
```

PROJECT3DRAY

```
import java.awt.geom.Point2D;
import java.awt.image.*;


public class Project3DRay {

// private static final float BIG_NUM = 100.0f;
   private static Vector2 p1, p2;
   private static Ray2D ray;
   private static float imW, imH, imTop, imRight, imBottom, imLeft;
   private static float stepsize;
   private static Interval1D interval;

   /**
    * Projects the supplied <code>Ray3D</code> to the supplied
    * <code>Camera</code>, which is assumed to be located in origo looking
    * down the negative Z-axis. The method then computes which part of the
    * ray that intersects the silhouette of the camera's image.
    * If no part of the ray is contained within the silhouette, an
    * empty interval is returned. In the current implementation, only single
    * intervals are supported, limiting the objects to be convex.
    *
    * @param r the ray to project.
    * @param c the camera to project to.
    * @return the interval of the ray that is contained in the silhouette.
    */
   public static Interval1D getProjectedInterval(Ray3D r, Camera c) {
      // Project two points on the ray to the view plane of the camera
      // and construct a Ray2D from the points.
      p1 = Vector2.project(r.getPosition(0), c);
      p2 = Vector2.project(r.getPosition(1000), c);
```

```
   ray = new Ray2D(p1, p2.sub(p1).normalize());;

   // Reposition the ray so that it sits on the edge of the camera's image
   // and points inwards into the image.
   repositionToImageEdge(ray, c);

   // Calculate stepsize, chosen so that it represents one pixel's width.
   stepsize = Vector2.pixel2World(1, c);

   // Step along the ray to find the start and end of the interval
   // contained within the silhouette.
   interval = new Interval1D();
   interval.makeEmpty();
   Vector2 currentPos;
   float tStart = 0;
   float tEnd = 0;
   int currentColor = 0;
   int step = 0;
   boolean searchForStart = true;
   boolean searchForEnd = true;
   for (; searchForStart; step++) {
      // Look for start of the interval
      currentPos = ray.getPosition(step * stepsize);
      currentPos.x = Math.round(Vector2.world2Pixel(currentPos.x, c) +
                  c.getImage().getWidth() / 2);
      currentPos.y = Math.round(c.getImage().getHeight() / 2 -
                  Vector2.world2Pixel(currentPos.y, c));
      currentColor = c.getRGB((int)currentPos.x, (int)currentPos.y);

      if (currentColor == Integer.MAX_VALUE) {
         // Pixel coordinates lay outside of the image
         searchForStart = false;
         searchForEnd = false;
      }
      else if (!isBackgroundColor(currentColor)) {
         interval.start = step * stepsize;
         searchForStart = false;
      }
   }
   for (; searchForEnd; step++) {
      // Look for end of the interval
      currentPos = ray.getPosition(step * stepsize);
      currentPos.x = Math.round(Vector2.world2Pixel(currentPos.x, c) +
                    c.getImage().getWidth() / 2);
      currentPos.y = Math.round(c.getImage().getHeight() / 2 -
                  Vector2.world2Pixel(currentPos.y, c));
      currentColor = c.getRGB((int)currentPos.x, (int)currentPos.y);
      if (currentColor == Integer.MAX_VALUE) {
         // Pixel coordinates lay outside of the image
         searchForEnd = false;
      }
      else if (isBackgroundColor(currentColor)) {
         interval.end = step * stepsize;
         searchForEnd = false;
      }
   }

   // Now the interval is in 2D space. The next step is to calculate
   // the corresponding 3D ray parameters.
   if (!interval.isEmpty()) {
      interval.start = backproject(r, ray.getPosition(interval.start), c);
      interval.end = backproject(r, ray.getPosition(interval.end), c);
   }

   return interval;
}

/*
 * Repositions the supplied <code>Ray2D</code> so that its start point
 * sits on the edge of the supplied <code>Camera</code>s image and its
 * direction points towards the interior of the image. The ray is assumed
 * to lie in the image plane of the camera.
 *
 * @param r the ray to reposition.
 * @param c the camera to fetch the image from.
 * @return true if the ray could be repositioned to the edge of the image,
 *       false otherwise.
 */
private static boolean repositionToImageEdge(Ray2D r, Camera c) {
   boolean hitImage = false;
   float newX, newY;
   Vector2 p = r.getPoint();
   Vector2 d = r.getDirection();
   Vector2 testOne = null;
   Vector2 testTwo = null;
   Vector2 testResult = null;
```

```
   // Read info about image dimensions
   imW = Vector2.pixel2World(c.getImage().getWidth()-1, c);
   imH = Vector2.pixel2World(c.getImage().getHeight()-1, c);
   imTop = imH/2;
   imRight = imW/2;
   imBottom = -imTop;
   imLeft = -imRight;

   // Figure out which edge to place the ray point on
   if (d.x >= 0 && d.y >= 0) {
      // Left or bottom edge
      testOne = r.getYForX(imLeft);
      testTwo = r.getXForY(imBottom);
   }
   else if (d.x < 0 && d.y < 0) {
      // Right or top edge
      testOne = r.getYForX(imRight);
      testTwo = r.getXForY(imTop);
   }
   else if (d.x >= 0 && d.y < 0) {
      // Left or top edge
      testOne = r.getYForX(imLeft);
      testTwo = r.getXForY(imTop);
   }
   else if (d.x < 0 && d.y >= 0) {
      // Right or bottom edge
      testOne = r.getYForX(imRight);
      testTwo = r.getXForY(imBottom);
   }
   testResult = Vector2.getShortest(testOne, testTwo);
   if (testResult != null) {
      r.setPoint(testResult);
      hitImage = true;
   }
   return hitImage;
}

/*
 * Calculates the parameter for the supplied ray that will correspond to
 * the supplied 2D point backprojected into 3D. The supplied camera is
 * assumed to be positioned in origo looking down the negative Z-axis.
 *
 * @param r3D the ray in 3D space.
 * @param p2D the point in 2D space.
 * @param cam the camera looking at the 3D ray.
 * @return the parameter for the (normalized) 3D ray that corresponds to the
 *         backprojection of the supplied 2D point.
 */
private static float backproject(Ray3D r3D, Vector2 p2D, Camera cam) {
   Vector4 p, q, d;
   p = new Vector4(r3D.getPoint());
   q = new Vector4(p2D.x, p2D.y, -cam.getFocalLength());
   d = new Vector4(r3D.getDirection());
   float lengthOfEpipolarLine = p.getLength();
   p.normalize();
   q.normalize();
   d.normalize();

   return lengthOfEpipolarLine *
         (float)Math.sin(Math.acos(q.dotProduct(p))) /
         (float)Math.sin(Math.acos(d.dotProduct(q)));
}

/**
 * Determines whether the supplied color is background or not. The color is
 * considered background if its red component is in the interval [0, 10],
 * its green component is in the interval [245, 255] and the blue component
 * is in the interval [0, 10].
 *
 * @param color the color to test.
 * @return true if the supplied color is a background color.
 */
public static boolean isBackgroundColor(int color) {

   // Extract rgb components
   int red = (color >> 16) & 0xff;
   int green = (color >> 8) & 0xff;
   int blue = color & 0xff;

   return (red <= 10 && green >= 245 && blue <= 10);
}

public static void printColor(int color) {

   // Extract rgb components
   int red = (color >> 16) & 0xff;
   int green = (color >> 8) & 0xff;
```

```
      int blue = color & 0xff;

      System.out.println(red + "\t" + green + "\t" + blue);
   }
}
```

RAY2D

```
/**
 * This class contains methods to handle Ray2D objects.
 */

public class Ray2D {


   Vector2 p = new Vector2();
   Vector2 d = new Vector2();

   /**
    * Constructor to create a <code>Ray2D</code>.
    *
    * @param p a <code>Vector2</code> containing the startposition of the ray.
    * @param d a <code>Vector2</code> containing the direction of the ray.
    */

   public Ray2D(Vector2 p, Vector2 d) {
      this.p = p;
      this.d = d;
   }

   /**
    * Constructor to create a <code>Ray2D</code>.
    */
   public Ray2D() {
      this(new Vector2(), new Vector2());
   }

   /**
    * Sets a new direction to the <code>Ray2D</code>.
    *
    * @param d a <code>Vector2</code> containing the new direction.
    */
   public void setDirection(Vector2 d) {
      this.d = d;
   }

   /**
    * Sets a new starting point to the <code>Ray2D</code>.
    *
    * @param p a <code>Vector2</code> containing the new startpoint.
    */
   public void setPoint(Vector2 p) {
      this.p = p;
   }

   /**
    * Returns the direction of the <code>Ray2D</code>.
    *
    * @return a <code>Vector2</code> containing the direction.
    */
   public Vector2 getDirection() {
      return this.d;
   }

   /**
    * Returns the starting point of the <code>Ray2D</code>.
    *
    * @return a <code>Vector2</code> containing the point.
    */
   public Vector2 getPoint() {
      return this.p;
   }

   /**
    * Returns a position along a <code>Ray2D</code>
    *
    * @param t the parameter
    *
    * @return a <code>Vector2</code> containing the position.
    */
   public Vector2 getPosition(float t) {
      return new Vector2(this.p.x + t*this.d.x, this.p.y + t*this.d.y);
   }

   /**
    * Returns a position along a <code>Ray2D</code> knowing the x-value
    *
```

```
     * @param x the x-value of the position
     *
     * @return a <code>Vector2</code> containing the position.
     */
    public Vector2 getYForX(float x) {
        if (d.x == 0) {
            return null;   // Avoid division by zero
        } else {
            float t = (x - p.x) / d.x;
            return new Vector2(x, p.y + t * d.y);
        }
    }

    /**
     * Returns a position along a <code>Ray2D</code> knowing the y-value
     *
     * @param y the y-value of the position
     *
     * @return a <code>Vector2</code> containing the position.
     */
    public Vector2 getXForY(float y) {
        if (d.y == 0) {
            return null;   // Avoid division by zero
        } else {
            float t = (y - p.y) / d.y;
            return new Vector2(p.x + t * d.x, y);
        }
    }
}
```

RAY3D

```
/**
 * This class contains methods to handle Ray3D objects.
 */
public class Ray3D {

    Vector4 p;
    Vector4 d;

    /**
     * Constructor to create a <code>Ray3D</code>.
     *
     * @param p a <code>Vector4</code> containing the startposition of the ray.
     * @param d a <code>Vector4</code> containing the direction of the ray.
     */
    public Ray3D(Vector4 p, Vector4 d) {
        this.p = p;
        this.d = d;
    }

    /**
     * Constructor to create a <code>Ray3D</code>.
     */
    public Ray3D() {
        this(new Vector4(), new Vector4(0.0f, 0.0f, 0.0f, 0.0f));
    }

    /**
     * Sets a new direction to the <code>Ray3D</code>.
     *
     * @param d a <code>Vector4</code> containing the new direction.
     */
    public void setDirection(Vector4 d) {
        this.d = d;
    }

    /**
     * Sets a new starting point to the <code>Ray3D</code>.
     *
     * @param p a <code>Vector4</code> containing the new startpoint.
     */
    public void setPoint(Vector4 p) {
        this.p = p;
    }

    /**
     * Returns the direction of the <code>Ray3D</code>.
     *
     * @return a <code>Vector4</code> containing the direction.
     */
    public Vector4 getDirection() {
        return d;
    }

    /**
     * Returns the starting point of the <code>Ray3D</code>.
```

```
     *
     * @return a <code>Vector4</code> containing the point.
     */
    public Vector4 getPoint() {
       return p;
    }

    /**
     * Returns a position along a <code>Ray3D</code>
     *
     * @param t the parameter
     *
     * @return a <code>Vector4</code> containing the position.
     */
    public Vector4 getPosition(float t) {
       return new Vector4(p.x + t*d.x,
                          p.y + t*d.y,
                          p.z + t*d.z);
    }

}
```

RAYINTERVALS

```
import java.util.Vector;

/**
 * This class can contain a <code>Ray3D</code> together with a number of
 * <code>Interval1D</code>s. This tuple represents a collection of intervals
 * along the ray. The intervals should be defined in terms of the parameter
 * of the ray.
 */

public class RayIntervals {

    /** The ray. */
    private Ray3D ray;
    /** Intervals along the ray. */
    private Vector intervals;

    /**
     * Creates an empty <code>RayIntervals</code>
     */
    public RayIntervals() {
    }

    /**
     * Creates a new <code>RayIntervals</code> object and sets its ray
     * to the supplied one.
     *
     * @param ray the new ray.
     */
    public RayIntervals(Ray3D ray) {
       this.ray = ray;
       intervals = new Vector(1,1);
       intervals.addElement(new Interval1D());
    }

    /**
     * Returns the current ray.
     *
     * @return the ray.
     */
    public Ray3D getRay() {
       return ray;
    }

    /**
     * Returns the current intervals along the ray.
     *
     * @return the intervals.
     */
    public Vector getIntervals() {
       return intervals;
    }
}
```

ROTATINGCUBE

```
import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import javax.swing.JPanel;
import java.awt.image.BufferedImage;
import java.awt.event.MouseMotionListener;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.geom.Point2D;

/**
 * Class that shows a cube that the user can spin round the x-axis and the y-axis.
 * It represent moving the camera around the object
 */
public class RotatingCube extends JPanel implements MouseMotionListener {


   float theta;
   float phi;
   float b;
   float originalPhi;

   Camera c;

   Matrix4 rotMatrixX, rotMatrixY, rotMatrix, transToOrigo, transBack;

   Vector4 p1, p2, p3, p4, p5, p6, p7, p8, c1, c2, c3, c4, c5, c6, c7, c8;

   float m2p;

   Point2D.Float sumAngle;

   int height,width;
   int xPos,yPos;
   int oldY,oldX;
   int startX,startY;

   boolean firstTime;

   int[] xCordTop; int[] xCordBottom; int[] xCordLeft; int[] xCordRight;
   int[] xCordFront; int[] xCordBack;

   int[] yCordTop; int[] yCordBottom; int[] yCordLeft; int[] yCordRight;
   int[] yCordFront; int[] yCordBack;

   Color red, blue;

   private double t=0;
   private Image buf;

   /**
    * Initialize all the parameters needed.
    *
    * @param ma the MouseAdapter sent from test()
    */
   public void initStuff(MouseAdapter ma) {

      this.addMouseListener(ma);

      firstTime = true;

      red = new Color(255 , 0, 0);

      sumAngle = new Point2D.Float();

      xPos = 0;
      yPos = 0;
      oldX = -1;
      oldY = -1;

      this.addMouseMotionListener(this);

      height = this.getHeight()/2;
      width = this.getWidth()/2;
      b = 0.1f;

      p1 = new Vector4(b,b,b-1);
      p2 = new Vector4(-b,b,b-1);
      p3 = new Vector4(-b,-b,b-1);
      p4 = new Vector4(b,-b,b-1);
      p5 = new Vector4(b,b,-b-1);
      p6 = new Vector4(-b,b,-b-1);
      p7 = new Vector4(-b,-b,-b-1);
      p8 = new Vector4(b,-b,-b-1);
```

```
    c1 = new Vector4();
    c2 = new Vector4();
    c3 = new Vector4();
    c4 = new Vector4();
    c5 = new Vector4();
    c6 = new Vector4();
    c7 = new Vector4();
    c8 = new Vector4();

    xCordTop = new int[4];
    yCordTop = new int[4];
    xCordLeft = new int[4];
    yCordLeft = new int[4];
    xCordRight = new int[4];
    yCordRight = new int[4];
    xCordFront = new int[4];
    yCordFront = new int[4];
    xCordBack = new int[4];
    yCordBack = new int[4];
    xCordBottom = new int[4];
    yCordBottom = new int[4];

    Vector4 up = new Vector4(0,1,0);
    Vector4 pos = new Vector4(0,0,0);
    Vector4 dir = new Vector4(0,0,-1);
    BufferedImage bi = new BufferedImage(Camera.DEFAULT_IMAGE_WIDTH,
                                Camera.DEFAULT_IMAGE_HEIGHT,
                                BufferedImage.TYPE_INT_RGB);

    c = new Camera("",pos,dir,up,0.043456f,(float)(45*Math.PI/180),bi);

    /** Change meters in to pixels. */
    m2p = (float)(256/0.036);

    transToOrigo = new Matrix4();
    transBack = new Matrix4();

    Matrix4.getTranslateInstance(transToOrigo, 0f, 0f, 1f);
    Matrix4.getTranslateInstance(transBack, 0f, 0f, -1f);
}

/**
 * Render the 3D graphics to the window using Java2D and JPanel for drawing
 * Change the points in 3d space and project them down so it looks like a box
 * spinning around.
 */
public void paintComponent(Graphics g) {


    super.paintComponent(g);
    g.setColor(Color.black);
    g.fillRect(0,0, getWidth(), getHeight());

    Graphics2D g2d = (Graphics2D)g;

    rotMatrixX = Matrix4.getRotateXInstance(phi);
    rotMatrixY = Matrix4.getRotateYInstance(theta);

    phi = 0;
    theta = 0;

    rotMatrix = rotMatrixX.mult(rotMatrixY, rotMatrixX);

    Matrix4.mult(transToOrigo, p1);
    Matrix4.mult(transToOrigo, p2);
    Matrix4.mult(transToOrigo, p3);
    Matrix4.mult(transToOrigo, p4);
    Matrix4.mult(transToOrigo, p5);
    Matrix4.mult(transToOrigo, p6);
    Matrix4.mult(transToOrigo, p7);
    Matrix4.mult(transToOrigo, p8);

    Matrix4.mult(rotMatrix, p1);
    Matrix4.mult(rotMatrix, p2);
    Matrix4.mult(rotMatrix, p3);
    Matrix4.mult(rotMatrix, p4);
    Matrix4.mult(rotMatrix, p5);
    Matrix4.mult(rotMatrix, p6);
    Matrix4.mult(rotMatrix, p7);
    Matrix4.mult(rotMatrix, p8);

    Matrix4.mult(transBack, p1);
    Matrix4.mult(transBack, p2);
    Matrix4.mult(transBack, p3);
    Matrix4.mult(transBack, p4);
    Matrix4.mult(transBack, p5);
```

```
Matrix4.mult(transBack, p6);
Matrix4.mult(transBack, p7);
Matrix4.mult(transBack, p8);

c1.x = p1.x;  c1.y = p1.y; c1.z = p1.z;
c2.x = p2.x;  c2.y = p2.y; c2.z = p2.z;
c3.x = p3.x;  c3.y = p3.y; c3.z = p3.z;
c4.x = p4.x;  c4.y = p4.y; c4.z = p4.z;
c5.x = p5.x;  c5.y = p5.y; c5.z = p5.z;
c6.x = p6.x;  c6.y = p6.y; c6.z = p6.z;
c7.x = p7.x;  c7.y = p7.y; c7.z = p7.z;
c8.x = p8.x;  c8.y = p8.y; c8.z = p8.z;

c1.project(c1, c);
c2.project(c2, c);
c3.project(c3, c);
c4.project(c4, c);
c5.project(c5, c);
c6.project(c6, c);
c7.project(c7, c);
c8.project(c8, c);

xCordFront[0] = Math.round(c1.x*m2p+width);
xCordFront[1] = Math.round(c2.x*m2p+width);
xCordFront[2] = Math.round(c3.x*m2p+width);
xCordFront[3] = Math.round(c4.x*m2p+width);
yCordFront[0] = Math.round(height-(-c1.y*m2p));
yCordFront[1] = Math.round(height-(-c2.y*m2p));
yCordFront[2] = Math.round(height-(-c3.y*m2p));
yCordFront[3] = Math.round(height-(-c4.y*m2p));

xCordBack[0] = Math.round(c6.x*m2p+width);
xCordBack[1] = Math.round(c5.x*m2p+width);
xCordBack[2] = Math.round(c8.x*m2p+width);
xCordBack[3] = Math.round(c7.x*m2p+width);
yCordBack[0] = Math.round(height-(-c6.y*m2p));
yCordBack[1] = Math.round(height-(-c5.y*m2p));
yCordBack[2] = Math.round(height-(-c8.y*m2p));
yCordBack[3] = Math.round(height-(-c7.y*m2p));

xCordLeft[0] = Math.round(c2.x*m2p+width);
xCordLeft[1] = Math.round(c6.x*m2p+width);
xCordLeft[2] = Math.round(c7.x*m2p+width);
xCordLeft[3] = Math.round(c3.x*m2p+width);
yCordLeft[0] = Math.round(height-(-c2.y*m2p));
yCordLeft[1] = Math.round(height-(-c6.y*m2p));
yCordLeft[2] = Math.round(height-(-c7.y*m2p));
yCordLeft[3] = Math.round(height-(-c3.y*m2p));

xCordRight[0] = Math.round(c1.x*m2p+width);
xCordRight[1] = Math.round(c4.x*m2p+width);
xCordRight[2] = Math.round(c8.x*m2p+width);
xCordRight[3] = Math.round(c5.x*m2p+width);
yCordRight[0] = Math.round(height-(-c1.y*m2p));
yCordRight[1] = Math.round(height-(-c4.y*m2p));
yCordRight[2] = Math.round(height-(-c8.y*m2p));
yCordRight[3] = Math.round(height-(-c5.y*m2p));

xCordTop[0] = Math.round(c1.x*m2p+width);
xCordTop[1] = Math.round(c5.x*m2p+width);
xCordTop[2] = Math.round(c6.x*m2p+width);
xCordTop[3] = Math.round(c2.x*m2p+width);
yCordTop[0] = Math.round(height-(-c1.y*m2p));
yCordTop[1] = Math.round(height-(-c5.y*m2p));
yCordTop[2] = Math.round(height-(-c6.y*m2p));
yCordTop[3] = Math.round(height-(-c2.y*m2p));

xCordBottom[0] = Math.round(c3.x*m2p+width);
xCordBottom[1] = Math.round(c7.x*m2p+width);
xCordBottom[2] = Math.round(c8.x*m2p+width);
xCordBottom[3] = Math.round(c4.x*m2p+width);
yCordBottom[0] = Math.round(height-(-c3.y*m2p));
yCordBottom[1] = Math.round(height-(-c7.y*m2p));
yCordBottom[2] = Math.round(height-(-c8.y*m2p));
yCordBottom[3] = Math.round(height-(-c4.y*m2p));

g2d.setColor(red);

if(isFaceVisible(xCordFront[0], xCordFront[1], xCordFront[2],
             yCordFront[0], yCordFront[1], yCordFront[2])){
   g2d.drawPolygon(xCordFront, yCordFront, 4);
}

if(isFaceVisible(xCordBack[0], xCordBack[1], xCordBack[2],
              yCordBack[0], yCordBack[1], yCordBack[2])){
   g2d.drawPolygon(xCordBack, yCordBack, 4);
}
```

```
   if(isFaceVisible(xCordRight[0], xCordRight[1], xCordRight[2],
               yCordRight[0], yCordRight[1], yCordRight[2])){
      g2d.drawPolygon(xCordRight, yCordRight, 4);
   }

   if(isFaceVisible(xCordLeft[0], xCordLeft[1], xCordLeft[2],
               yCordLeft[0], yCordLeft[1], yCordLeft[2])){
      g2d.drawPolygon(xCordLeft, yCordLeft, 4);
   }
   if(isFaceVisible(xCordTop[0], xCordTop[1], xCordTop[2],
               yCordTop[0], yCordTop[1], yCordTop[2])){
      g2d.drawPolygon(xCordTop, yCordTop, 4);
   }

   if(isFaceVisible(xCordBottom[0], xCordBottom[1], xCordBottom[2],
               yCordBottom[0], yCordBottom[1], yCordBottom[2])){
      g2d.drawPolygon(xCordBottom, yCordBottom, 4);
   }
}

public void mouseDragged(MouseEvent e) {
}

/**
 * Reacts if the mouse is moved. The movement is used to change the points
 * in paintComponent()
 */
public void mouseMoved(MouseEvent e) {
   xPos = e.getX();
   yPos = e.getY();
   if (firstTime){
      startX = xPos;
      startY = yPos;
      theta = 0f;
      phi = 0f;
      firstTime = false;
   }
   else{
      theta = (float)(xPos - startX)/50;
      phi = (float)(yPos - startY)/50;

      if((originalPhi + phi) >= (float)Math.PI){
         phi = (float)Math.PI - originalPhi;
      }
      if((originalPhi + phi) <= 0){
         phi = 0 - originalPhi;
      }

      sumAngle.x = theta;
      sumAngle.y = phi;
      setOrginalCube();
   }

   repaint();
}

/**
 * Checks if the polygon that the param belongs to is visible when it is
 * projected down.
 */
public boolean isFaceVisible(int x1, int x2, int x3, int y1, int y2, int y3){
   int zeta = (x2-x1) * (y3-y2) - (y2-y1) * (x3 - x2);
   if (zeta > 0){
      return true;
   }
   else{
      return false;
   }
}

/**
 * Reset the angles with who the cube is spinned around its axis with to zero
 */
public void resetAngles() {
   sumAngle.x = 0f;
   sumAngle.y = 0f;
}
/**
 * set the cube to its original position
 */
public void setOrginalCube(){
   p1.setVector4(b,b,b-1, 1f);
   p2.setVector4(-b,b,b-1,1f);
   p3.setVector4(-b,-b,b-1,1f);
   p4.setVector4(b,-b,b-1, 1f);
   p5.setVector4(b,b,-b-1,1f);
```

```
      p6.setVector4(-b,b,-b-1,1f);
      p7.setVector4(-b,-b,-b-1,1f);
      p8.setVector4(b,-b,-b-1,1f);
   }

   public void setFirstTime(boolean b){
      firstTime = b;
   }

   /**
    * @return the angles with who the cube is spinned around its axis with
    */
   public Point2D.Float getAngles(){
      return new Point2D.Float(-sumAngle.x, sumAngle.y);
   }

   /**
    * Set the cameras present phi angle, this is used to make sure that
    * the cube never can rotate so it will be upside down
    */
   public void setCameraPhi(float p){
      originalPhi  = p;
   }
}
```

VECTOR2

```
/**
 * This class contains methods to handle <code>Vector2</code> objects.
 */
public class Vector2 {

   public float x, y;


   /**
    * Constructor to create a <code>Vector2</code>.
    */
   public Vector2() {
      this.x = 0;
      this.y = 0;
   }

   /**
    * Constructor to create a <code>Vector2</code>.
    *
    * @param x the x value.
    * @param y the y value.
    */
   public Vector2(float x, float y) {
      this.x = x;
      this.y = y;
   }

   /**
    * Method to copy a <code>Vector2</code> object.
    *
    * @param v the <code>Vector2</code> that should be copied.
    */
   public Vector2(Vector2 v) {
      this.x = v.x;
      this.y = v.y;
   }

   /**
    * Method to copy a the first 2 components of a <code>Vector4</code> object
    * to a <code>Vector2</code> object.
    *
    * @param v the <code>Vector4</code> of which the first 2 component shall be
    * copied.
    */
   public Vector2(Vector4 v) {
      this.x = v.x;
      this.y = v.y;
   }

   /**
    * Static method to add two <code>Vector2</code>.
    *
    * @param v1 a <code>Vector2</code>.
    * @param v2 a <code>Vector2</code>.
    *
    * @return A new <code>Vector2</code>.
    */

   public static Vector2 add(Vector2 v1, Vector2 v2) {
      return new Vector2(v1.x + v2.x, v1.y + v2.y);
```

```
}

/**
 * Method to add a <code>Vector2</code> to an existing <code>Vector2</code>.
 *
 * @param v a <code>Vector2</code>.
 *
 * @return a <code>Vector2</code>.
 */
public Vector2 add(Vector2 v) {
   this.x += v.x;
   this.y += v.y;
   return this;
}

/**
 * Static method to subtract a <code>Vector2</code> from another
 *
 * @param v1 a <code>Vector2</code>
 * @param v2 a <code>Vector2</code> to subtract
 *
 * @return A new <code>Vector2</code>
 */
public static Vector2 sub(Vector2 v1, Vector2 v2) {
   return new Vector2(v1.x - v2.x, v1.y - v2.y);
}

/**
 * Method to subtract a <code>Vector2</code> from an current
 * <code>Vector2</code>.
 *
 * @param v a <code>Vector2</code> that should be subtracted.
 *
 * @return a <code>Vector2</code>.
 */
public Vector2 sub(Vector2 v) {
   this.x -= v.x;
   this.y -= v.y;
   return this;
}

/**
 * Method that returns the length of current <code>Vector2</code>.
 *
 *  @return the length.
 */
public float getLength() {
   return (float)Math.sqrt(this.x * this.x + this.y * this.y);
}

/**
 * Method to normalize the current <code>Vector2</code>.
 *
 * @return the normalized <code>Vector2</code>.
 */
public Vector2 normalize() {
   float len = this.getLength();
   this.x /= len;
   this.y /= len;
   return this;
}

/**
 * Method to project a <code>Vector4</code> onto an imageplane
 *
 * @param c the Camera containing the imageplane
 * @param p the <code>Vector4</code> to project
 *
 * @return the resulting <code>Vector2</code>
 */
public static Vector2 project(Vector4 p, Camera c) {
   float f = c.getFocalLength();
   float w = Math.abs(p.z / f);
   return new Vector2(p.x / w, p.y / w);
}

/**
 * Method to convert a world position (1D) to image coordinates.
 *
 * @param c the camera containing the image coordinates.
 * @param world the point in the world.
 *
 * @return the converted point
 */
public static int world2Pixel(float world, Camera c) {
   float width = c.getImage().getWidth();
   float fov = c.getFov();
```

```
      float focalLength = c.getFocalLength();
      float factor = width / (2 * (focalLength * (float)Math.tan(fov / 2)));
      return Math.round(world * factor);
   }

   /**
    * Method to convert a image coordinate (1D) to image coordinates.
    *
    * @param c the camera containing the image coordinates.
    * @param pixel the point in the image (1D).
    *
    * @return the converted point
    */
   public static float pixel2World(int pixel, Camera c) {
      float width = c.getImage().getWidth();
      float fov = c.getFov();
      float focalLength = c.getFocalLength();
      float factor = width / (2 * (focalLength * (float)Math.tan(fov / 2)));
      return pixel / factor;
   }

   /**
    * Returns the shortest of the two supplied vectors. If the vectors are
    * of equal length, the second one is returned.
    *
    * @param v1 the first vector.
    * @param v2 the second vector.
    * @return the shortest of the two vectors or the second one if they are
    *          of equal length.
    */
   public static Vector2 getShortest(Vector2 v1, Vector2 v2) {
      if (v1 == null && v2 != null) {
         return v2;
      }
      else if (v1 != null && v2 == null) {
         return v1;
      }
      else if (v1 == null && v2 == null) {
         return null;
      }
      else {
         if (v1.getLength() < v2.getLength()) {
            return v1;
         }
         else {
            return v2;
         }
      }
   }

}
```

VECTOR4

```
/**
 * Class for handeling all vector computations
 */
public class Vector4 {

   public float x;
   public float y;
   public float z;
   public float w;


    /**
     * Default constructor
     */
    public Vector4() {
        this(0, 0, 0, 1);
    }

   /**
    * Constructor
    *
    * @param x The vectors x coeficient
     * @param y The vectors y coeficient
     * @param z The vectors z coeficient
     * @param w The vectors w coeficient
     */
   public Vector4(float x, float y, float z, float w) {
      this.x = x;
      this.y = y;
      this.z = z;
      this.w = w;
   }
```

```
 /**
  * Constructor
  *
  * @param x The vectors x coeficient
  * @param y The vectors y coeficient
  * @param z The vectors z coeficient
  */
public Vector4(float x, float y, float z) {
   this(x, y, z, 1);
}

 /**
  * Copy constructor
  *
  * @param original The vector to be copied
  */
 public Vector4(Vector4 original) {
     this.x = original.x;
     this.y = original.y;
     this.z = original.z;
     this.w = original.w;
 }

/**
 * Set new values to the vector
 *
 * @param x x-value
 * @param y y-value
 * @param z z-value
 * @param w w-value
 */
public void setVector4(float x, float y, float z, float w) {
   this.x = x;
   this.y = y;
   this.z = z;
   this.w = w;
}

 /**
  * Negate a vector
  */
public Vector4 negate() {
   this.x = -this.x;
   this.y = -this.y;
   this.z = -this.z;
   return this;
}

  /**
  * Negate a vector
  *
  * @param v The vector to negate
  * @return a new vector = - v
  */

public static Vector4 negate(Vector4 v) {
   Vector4 v2 = new Vector4();
   v2.x = -v.x;
   v2.y = -v.y;
   v2.z = -v.z;
   return v2;
}

 /**
  * Gets the length of a vector
  *
  * @return the lengt as a float
  */
public float getLength() {
   return (float)Math.sqrt(Math.pow(this.x,2) + Math.pow(this.y,2)
                     + Math.pow(this.z,2));
}

 /**
  * Normalize
  *
  * @return the normalized vector
  */
public Vector4 normalize() {
   float den = (float)Math.sqrt(Math.pow(this.x,2) + Math.pow(this.y,2)
                        + Math.pow(this.z,2));
   this.x = this.x / den;
   this.y = this.y / den;
   this.z = this.z / den;
   return this;
}
```

```
 /**
  * Normalize
  *
  * @param n The vector to normalize
  * @return a new normalized vector
  */
public Vector4 normalize(Vector4 n) {
   float den = (float) Math.sqrt(Math.pow(n.x,2) + Math.pow(n.y,2)
                          + Math.pow(n.z,2));

   this.x = n.x / den;
   this.y = n.y / den;
   this.z = n.z / den;
   return this;
}

 /**
  * Compute the cross product between two vectors
  *
  * @param n The first vector
  * @param o The second vector
  * @return a new vector orthogonal to the others
  */
 public static Vector4 crossProduct(Vector4 n, Vector4 o){
     Vector4 cp = new Vector4();
     cp.x = n.y * o.z - n.z * o.y;
     cp.y = n.z * o.x - n.x * o.z;
     cp.z = n.x * o.y - n.y * o.x;
     return cp;
 }

 /**
  * Compute the cross product between two vectors
  *
  * @param o The second vector
  * @return the resulting vector
  */
 public Vector4 crossProduct(Vector4 o){
     this.x = this.y * o.z - this.z * o.y;
     this.y = this.z * o.x - this.x * o.z;
     this.z = this.x * o.y - this.y * o.x;
     return this;
 }

 /**
  * Compute the dot product of two vectors
  * @param v the second vector
  * @return the dotproduct in floating point
  */
 public float dotProduct(Vector4 v) {
   float dot = this.x * v.x + this.y * v.y + this.z * v.z;
   return dot;
 }

 /**
  * Adds a vector to current vector.
  *
  * @param b The vector to add
  * @return the resulting vector
  */
public Vector4 add(Vector4 b) {
   this.x = this.x + b.x;
   this.y = this.y + b.y;
   this.z = this.z + b.z;
   return this;
}

 /**
  * Subtracts a vector from current vector.
  *
  * @param b The vector to subtract
  * @return the resulting vector
  */
public Vector4 sub(Vector4 b) {
   this.x = this.x - b.x;
   this.y = this.y - b.y;
   this.z = this.z - b.z;
   return this;
}

 /**
  * Subtracts a vector from current vector.
  *
  * @param a The first vector
  * @param b The vector to subtract
  * @return a new resulting vector
  */
```

```
    public static Vector4 sub(Vector4 a, Vector4 b) {
       Vector4 v = new Vector4();
       v.x = a.x - b.x;
       v.y = a.y - b.y;
       v.z = a.z - b.z;
       return v;
    }


     /**
      * Projects a point in world coordinates to the origin cameras viewplane.
      * The coordinates are not converted to pixel coordinates.
      * The projected vector is NOT changed.
      *
      * @param vect3D the 3d-point to project
      * @param c the camera whose viewplane the point is to be projected to
      * @return the projected point as a vector
      */
    public Vector4 project(Vector4 vect3D, Camera c) {
       this.w = -vect3D.z/c.getFocalLength();
       this.x = vect3D.x / this.w;
       this.y = vect3D.y / this.w;
       this.z = vect3D.z / this.w;
       this.w = 1;
       return this;
    }

     /**
      * Projects a point in world coordinates to the origin cameras viewplane.
      * The coordinates are not converted to pixel coordinates.
      * The projected vector is changed.
      *
      * @param c the camera whose viewplane the point is to be projected to
      * @return the projected point as a vector
      */
    public Vector4 project(Camera c) {
       this.w = -this.z/c.getFocalLength();
       this.x = this.x / this.w;
       this.y = this.y / this.w;
       this.z = this.z / this.w;
       this.w = 1;
       return this;
    }

    /**
     * Convert world coordinates to image coordinates, no projection.
     *
     * @param c The camera containing the image coordinates.
     * @return the converted vector
     */
    public Vector4 world2pixel(Camera c) {
       double width = c.getImage().getWidth();
       double fov = c.getFov();
       double focalLength = c.getFocalLength();
       double factor = width / (2 * (focalLength * Math.tan(fov / 2)));
       this.x *= factor;
       this.y *= factor;
       return this;
    }

     /**
      * Convert image coordinates to world coordinates, no projection.
      *
      * @param c The camera containing the image coordinates.
      * @return the converted vector
      */
    public Vector4 pixel2world(Camera c) {
       double width = c.getImage().getWidth();
       double fov = c.getFov();
       double focalLength = c.getFocalLength();
       double factor = width / (2 * (focalLength * Math.tan(fov / 2)));
       this.x /= factor;
       this.y /= factor;
       return this;
    }
}
```

VISUALHULL

```
/**
 * This class represents the visual hull of an object in the scene. The hull
 * consists of a series of intervals along rays shot from the desired view.
 */

public class VisualHull {

   /** 2D array of <code>RayIntervals</code> representing the visual hull. */
   private RayIntervals[][] intervalImage;
   /** <code>Camera</code> of the desired view. */
   private Camera cam;
   /** Width of the interval image. */
   private int width;
   /** Height of the interval image. */
   private int height;

   /**
    * Creates an empty <code>VisualHull</code>.
    */
   public VisualHull() {
   }

   /**
    * Creates an empty <code>VisualHull</code> with dimensions set to the
    * dimensions of the image contained in the supplied <code>Camera</code>.
    * Also creates <code>Ray3D</code>s from the projection point of the
    * <code>Camera</code> through all the pixels of its image.
    *
    * @param c the camera of the desired view. Must contain an image.
    */
   public VisualHull(Camera c) {
      this.cam = c;
      this.width = c.getImage().getWidth();
      this.height = c.getImage().getHeight();

      intervalImage = new RayIntervals[this.height][];
      for (int i = 0; i < this.height; i++) {
         intervalImage[i] = new RayIntervals[this.width];
         for (int j = 0; j < this.width; j++) {
            intervalImage[i][j] =
               new RayIntervals(Compute3DRay.compute3DRay(c, j, i));
         }
      }
   }

   /**
    * Inserts the supplied <code>RayIntervals</code> into the specified
    * position in the interval image.
    *
    * @param posX the x-position.
    * @param posY the y-position.
    * @param ri the <code>RayIntervals</code> object to insert.
    */
   public void insertRayIntervals(int posX, int posY, RayIntervals ri) {
      intervalImage[posY][posX] = ri;
   }

   public RayIntervals getRayIntervals(int posX, int posY) {
      return intervalImage[posY][posX];
   }

   /**
    * Transforms the complete visual hull by multiplying all its rays with
    * the supplied matrix. Note that no rescaling of the intervals is performed
    * so the matrix should imply any scaling either.
    *
    * @param t the transformation matrix.
    */
   public void transform(Matrix4 t) {
      Ray3D temp;
      for (int row = 0; row < this.height; row++) {
         for (int col = 0; col < this.width; col++) {
            temp = intervalImage[row][col].getRay();
            Matrix4.mult(t, temp.getPoint());
            Matrix4.mult(t, temp.getDirection());
         }
      }
   }

   public void transformRayDirections(Matrix4 t) {
      Ray3D temp;
      for (int row = 0; row < this.height; row++) {
         for (int col = 0; col < this.width; col++) {
            temp = intervalImage[row][col].getRay();
            Matrix4.mult(t, temp.getDirection());
```

```
         }
      }
   }

   public void transformRayPositions(Matrix4 t) {
      Ray3D temp;
      for (int row = 0; row < this.height; row++) {
         for (int col = 0; col < this.width; col++) {
            temp = intervalImage[row][col].getRay();
            Matrix4.mult(t, temp.getPoint());
         }
      }
   }
}
```

VISUALHULLCREATOR

```
import java.util.Vector;

/**
 * This class takes care of all the steps of constructing a visual hull
 * from a set of reference views and a desired view.
 */
public class VisualHullCreator implements Runnable {

   /** Reference views */
   private Vector refViews;
   /** Desired view */
   private Camera desView;
   /** The actual visual hull */
   private VisualHull vh;
   private boolean running;
   private int progress;
   private float progressStep;
   private String note;
   public Thread thread;

   private static final int BG_COLOR = ((0xff << 24) & 0xffffffff) |
                              (((0xff & 0) << 16) & 0xffffffff) |
                              (((0xff & 255) << 8) & 0xffffffff) |
                              ((0xff & 0) & 0xffffffff);

   private static final int BLACK = ((0xff << 24) & 0xffffffff) |
                            (((0xff & 0) << 16) & 0xffffffff) |
                            (((0xff & 0) << 8) & 0xffffffff) |
                            ((0xff & 0) & 0xffffffff);

   /**
    * Creates a <code>VisualHullCreator</code> with the supplied reference
    * views and desired view.
    *
    * @param refs a <code>Vector</code> of <code>Camera</code>s representing
    *             the reference views of the scene.
    * @param desired a <code>Camera</code> representing the desired view.
    */
   public VisualHullCreator(Vector refs, Camera desired) {

      this.running = false;
      this.progress = 0;
      this.note = "";
      this.refViews = refs;
      this.desView = desired;
      this.vh = new VisualHull(desView);
   }

   /**
    * Returns whether the process is running.
    *
    * @return true if the process is running, false otherwise.
    */
   public boolean isRunning() {
      return this.running;
   }

   /**
    * Starts the calculation of the visual hull.
    */
   public void startProcess() {
      this.running = true;
      this.progress = 0;
      thread = new Thread(this);
      thread.setPriority(Thread.MIN_PRIORITY);
      thread.start();
   }

   /**
    * Interrupts the calculation of the visual hull at the next loop.
```

```
 */
public void stopProcess() {
   this.running = false;
}

/**
 * Returns how far the calculation of the visual hull has progressed.
 * This is indicated as a nuber between 0 and 100.
 *
 * @return the progress of the calculation (number between 0 and 100).
 */
public int getProgress() {
   return this.progress;
}

/**
 * Returns a <code>String</code> that describes what phase the calculations
 * are in.
 *
 * @return a <code>String</code> describing the state of the calculations.
 */
public String getNote() {
   return this.note;
}

/**
 * Returns whether the calculation of the visual hull is finished.
 *
 * @return true if the calculation is finished, false otherwise.
 */
public boolean isFinished() {
   return (progress == 100);
}

/**
 * Returns the <code>VisualHull</code> object contained in the
 * <code>VisualHullCreator</code>.
 *
 * @return the visual hull.
 */
public VisualHull getVisualHull() {
   return this.vh;
}

public void run() {
   // Calculate everything
   // Don't forget to update this.progress

   Camera c;
   RayIntervals ri;
   Matrix4 rot, iRot, trans, iTrans;
   Ray3D r;
   int notEmpty = 0;
   int camWidth = 0;
   int camHeight = 0;

   progressStep = 1.0f/(float)(refViews.size()+1);

   for (int camNum = 0; camNum < refViews.size(); camNum++) {

      c = (Camera) refViews.elementAt(camNum);

      this.note = "Intersecting rays with " + c.getTitle();

      CameraRot.createMatrices(c);

      rot = CameraRot.getRotMatrix();
      iRot = CameraRot.getInvRotMatrix();
      trans = CameraRot.getTlMatrix();
      iTrans = CameraRot.getInvTlMatrix();

      camWidth = desView.getImage().getWidth();
      camHeight = desView.getImage().getHeight();

      for (int posY = 0; posY < camHeight; posY++) {
         for (int posX = 0; posX < camWidth; posX++) {

            this.progress = (int)(100 * ((float)camNum *
               (float)progressStep + (float)progressStep *
               ((float)posY * (float)camWidth + posX) /
               ((float)camWidth * (float)camHeight)));

            ri = vh.getRayIntervals(posX, posY);
            r = ri.getRay();

            // Translate ray position
            Matrix4.mult(trans, ri.getRay().getPoint());
```

```
            // Rotate ray position and direction
            Matrix4.mult(rot, r.getPoint());
            Matrix4.mult(rot, r.getDirection());

            // Check if the angle between the ray direction and
            // the view direction of the camera is close to 0 degrees
            // or 180 degrees. If so, skip interval calculation for
            // this ray.
            float angle = (float)Math.toDegrees(Math.acos(
                        (new Vector4(0, 0, -1)).dotProduct(
                         r.getDirection())));

            if (angle >= 20 && angle <= 160) {

               ((Interval1D) ri.getIntervals().elementAt(0)).intersection(
                  Project3DRay.getProjectedInterval(r, c));
            }

            // Rotate ray position and direction back
            Matrix4.mult(iRot, r.getPoint());
            Matrix4.mult(iRot, r.getDirection());
            // Translate ray position back
            Matrix4.mult(iTrans, r.getPoint());

            thread.yield();
         }
      }
   }

   // Perform shading
   this.note = "Shading the visual hull";

   // Create a depth map of the visualhull
   createDepthMap(vh, desView);

   // Shade our visual hull
   // shade(vh,refViews,desView);

   this.progress = 100;
}

/**
 * Depth map for the visual hull.
 *
 * @param vh The visual hull.
 * @param novel The novel camera.
 */
 private void createDepthMap(VisualHull vh, Camera novel) {

   float near = Float.NEGATIVE_INFINITY;
   float far = Float.POSITIVE_INFINITY;
   float intensity;
   int tempColor;
   Vector4 point;
   RayIntervals ri;
   Interval1D interval;

   CameraRot.createMatrices(novel);
   Matrix4 rotationMatrix = CameraRot.getRotMatrix();
   Matrix4 translationMatrix = CameraRot.getTlMatrix();

   Matrix4.mult(translationMatrix, novel.getPos());
   Matrix4.mult(rotationMatrix, novel.getDir());

   int camWidth = novel.getImage().getWidth();
   int camHeight = novel.getImage().getWidth();

   // Find maximum and minimum depth
   for (int posY = 0; posY < camHeight; posY++) {
      for (int posX = 0; posX < camWidth; posX++) {

         progress += 50 * progressStep * (float)(posY * camWidth + posX)/
                                    (float)(camWidth * camHeight);

         ri = vh.getRayIntervals(posX, posY);
         interval = (Interval1D) ri.getIntervals().elementAt(0);
         if (!interval.isEmpty()) {
            point = ri.getRay().getPosition(interval.start);

            Matrix4.mult(translationMatrix, point);
            Matrix4.mult(rotationMatrix, point);

            if(point.z > near) {
               near = point.z;
            }
            if(point.z < far) {
```

```
                far = point.z;
            }

         }

      }
   }

   // Shade the depth map
   for (int posY = 0; posY < camHeight; posY++) {
      for (int posX = 0; posX < camWidth; posX++) {

         progress += 50 * progressStep * (float)(posY * camWidth + posX)/
                                    (float)(camWidth * camHeight);

         ri = vh.getRayIntervals(posX, posY);
         interval = (Interval1D) ri.getIntervals().elementAt(0);
         if (interval.isEmpty()) {
            novel.getImage().setRGB(posX,posY,BLACK);
            continue;
         }
         point = ri.getRay().getPosition(interval.start);

         Matrix4.mult(translationMatrix, point);
         Matrix4.mult(rotationMatrix, point);
         intensity = 255 - (near - point.z) * 200 / (near - far);

         tempColor = ((0xff << 24) & 0xffffffff) |
                  (((0xff & (int)intensity) << 16) & 0xffffffff) |
                  (((0xff & (int)intensity) << 8) & 0xffffffff) |
                  ((0xff & (int)intensity) & 0xffffffff);

         novel.getImage().setRGB(posX,posY,tempColor);
      }
   }
   rotationMatrix = CameraRot.getInvRotMatrix();
   translationMatrix = CameraRot.getInvTlMatrix();

   Matrix4.mult(rotationMatrix, novel.getDir());
   Matrix4.mult(translationMatrix, novel.getPos());
}

/** A method to shade our visual hull
 *
 *   @param vh the <code>visual hull</code>
 * @param refs a vector containing the reference cameras
 * @param desView a camera containing our novel view
 */
private void shade(VisualHull vh, Vector refs, Camera desView) {

   Camera c;
   RayIntervals ri;
   Ray3D r;
   Interval1D interval;
   Matrix4 rotationMatrix;
   Matrix4 translationMatrix;
   Vector4 point;
   Vector4 desPos;
   Vector4 dir1;
   Vector4 dir2;
   float t;
   float angle = 0;
   float bestAngle = -1;
   int bestCamera = 0;
   int pixel;
   Vector2 projectedPoint = new Vector2();

   desPos = desView.getPos();

   // For every ray in the novel image
   for (int posY = 0; posY < desView.getImage().getHeight(); posY++) {
      for (int posX = 0; posX < desView.getImage().getWidth(); posX++) {

         ri = vh.getRayIntervals(posX, posY);
         r = ri.getRay();
         interval = (Interval1D) ri.getIntervals().elementAt(0);

         // If the intreval in the current position is empty
         if (interval.isEmpty()) {
            // Set background color
            desView.getImage().setRGB(0,0,0);
         }
         else {

                  // Get the intervals startpoint in 3d
            t = interval.start;
                  // Reference camera position
```

```
                point = r.getPosition(t);
                     // Epipolar line betwwen reference and novel camera
                dir1 = Vector4.sub(point, desPos);
                dir1.normalize();

                bestAngle = -1;

                     // Find the camera, which direction is closest to the novel
                     // view direction
                for (int camNum = 0; camNum < refs.size(); camNum++) {
                   c = (Camera) refs.elementAt(camNum);
                   dir2 = Vector4.sub(point, c.getPos());
                   dir2.normalize();
                   angle = (float)Math.acos(dir1.dotProduct(dir2));

                   if(bestAngle == -1) {
                      bestCamera = camNum;
                      bestAngle = angle;
                   }
                   else if(angle < bestAngle) {
                      bestAngle = angle;
                      bestCamera = camNum;
                   }
                }

                // Get the best camera
                c = (Camera) refs.elementAt(bestCamera);

                // Make matrices to rotate the camera to origin and it's
                // direction to 0,0,-1
                CameraRot.createMatrices(c);
                rotationMatrix = CameraRot.getRotMatrix();
                translationMatrix = CameraRot.getTlMatrix();

                // Rotate the camera and do corresponding rotation on the
                // interval point
                Matrix4.mult(translationMatrix, c.getPos());
                Matrix4.mult(translationMatrix, point);
                Matrix4.mult(rotationMatrix, c.getDir());
                Matrix4.mult(rotationMatrix, point);

                     // Project the interval point to the cameras viewplane
                projectedPoint = Vector2.project(point, c);

                // Pick pixel coordinates  .....
                projectedPoint.x = Math.round(
                               Vector2.world2Pixel(projectedPoint.x, c)
                               + c.getImage().getWidth() / 2);
                projectedPoint.y = Math.round(
                              c.getImage().getHeight() / 2 -
                             Vector2.world2Pixel(projectedPoint.y, c));

                pixel = c.getRGB((int)projectedPoint.x,
                                 (int)projectedPoint.y);

                     // If the picked pixel is outside of the object or of
                     // background color, set it to black.
                if(Project3DRay.isBackgroundColor(pixel) ||
                   pixel == Integer.MAX_VALUE) {

                   pixel = BLACK;
                }

                desView.getImage().setRGB(posX, posY, pixel);

                rotationMatrix = CameraRot.getInvRotMatrix();
                translationMatrix = CameraRot.getInvTlMatrix();
                     // Rotate the camera back to it's previous location and
                     // direction.
                Matrix4.mult(rotationMatrix, c.getDir());
                Matrix4.mult(translationMatrix, c.getPos());

             }
          }
       }

    }
}
```