TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

ACTA TECHNICA NAPOCENSIS

Series: Applied Mathematics, Mechanics, and Engineering
Vol. 64, Issue I, March, 2021

USING JAVA AFFINE TRANSFORMATION IN A SWING BASED 2DOF PLANAR ROBOT SIMULATION

Tiberiu Alexandru ANTAL

Abstract: *Computer graphics that study and implement the simulation of the operation of robots is a topical field. Java programming language provides a ready-implemented package for the programmer with a set of 2D geometric transformations implemented in the form of homogeneous coordinate matrices in the AffineTransform class. The following paper aims to show how they are used concretely in simulating the operation of a simple 2R, planar (2D) robot and draws conclusions about the performance of the simulation by comparing the simulation based on the mathematical model with that based on affine transformations.*
Key words: *four-bar linkage, bisection, java, multithreaded, simulation.*

1. INTRODUCTION

1.1 Some words on vector graphics

Geometrical objects need a model in order to be shown on devices like the monitor or the printer. One way of modelling the geometrical objects is based on vectors and is called vector graphics. Complex objects are obtained by combining primitive (or elementary) objects like lines, rectangles, arcs, circles or ellipses. Primitive graphical objects are described in terms of position and some parameters that are object specific. For example, a circle has a center point against which the radius parameter is given, a rectangle an upper-left point against which the parameters length and height are given. Although the vector based construction of complex geometric figures is simple using graphical primitives the vector modelling of objects is not directly suitable for drawing on pixel oriented devices like the monitor or a printer. Monitors and printers are based on raster graphics also called pixel oriented graphic. Raster graphics is using a fixed size matrix of elements called pixels where each pixel has an associated color. In order to display a vector modelled geometrical object on a raster graphics device the object must be converted to colored pixels. The procedure is called scan conversion and involves:

- high computational effort and
- some aliasing effects (as the continuous model must be sampled to get the discrete one).

Compared to raster graphics the advantages of vector graphics are:

- smaller size;
- ability to zoom indefinitely;
- moving, scaling, filling, or rotating do not degrade the quality of an image.

Although the objects must be brought to a rasterized form in order to be displayed on the monitor or printer it would be a great advantage to keep them stored as vector graphics. One of the benefits of using Java is that it implements rasterized two- and three-dimensional graphical primitives while preserving the vector storage strategy of the graphical objects to be displayed.

1.2 Vector graphics in Java

2D geometrical objects in Java were based initially on the Java 2D API extensions of the *Abstract Windowing Toolkit* (AWT) [1]. The Java 2D API stores and works with two coordinate spaces:

- User - the space in which graphics primitives are specified ;
- Device - the coordinate system of an output device such as a screen, window, or a printer.

The necessary conversions between user space and device space are performed automatically during rendering. The *java.awt.Graphics* class is used for custom painting. It manages the graphics context and provides methods for rendering of three types of graphical objects: text, graphical primitives and raster (also called bitmap) graphics. The graphics to be displayed is defined inside the *paint()* (for *JFrame*) or *paintComponent()* (for *JPanel*) methods while the device context is a parameter of the *paint()* (or *paintComponent()*) method passed as *paint(Graphics g)*. The user coordinate system in which the programmer specifies the graphical primitive coordinates starts at *(0, 0)* in the upper left corner of the window (JFrame of JPanel) and extends to the right for the x-axis and down for the y-axis. This means that the y-axis points downwards that is reversed to how are used to consider it. The window corresponding to the container we use to draw has a size (in pixels) and has margins (in pixels) on all the four sides. It is not possible to draw outside the size of window and the drawing space of the window in smaller than its size because of the margins. The exact size of the margins can be determined inside the *paint()* method by calling the *getInsets()* method which indicates the size of the *JFrame* (or *JPanel*) border using the line:

Inset ins = this.getInsets();

with the following properties: *ins.left*, *ins.right*, *ins.top* and *ins.bottom*. However, since these dimensions are platform-dependent a valid insets value cannot be obtained until the panel is made visible. The *Graphics2D* class extends the *Graphics* class by adding support to operations or new attributes like:

- Geometric transformations (translation, rotation, scaling and shearing);
- Pen (outline of a Shape) and Stroke (point-size, dashing-pattern, end-cap and join decorations);
- Fill (interior of a shape) and Paint (fill Shapes with solid colors, gradients, and patterns);
- Constructive Geometry of Shapes (for overlapping shapes);
- Clip (display area) etc.;

Graphics2D is designed as a subclass of *Graphics*. The *Graphics2D* context must downcast the *Graphics g* in *paintComponent()* to *g2* as shown in the following piece of code:

```
public void paintComponent(Graphics g) {
// graphics subsystem passes a Graphis2D
// subclass object as argument
// paint parent's background
// must be the first line in the method
  super.paintComponent(g);
// downcast the Graphics object
// back to Graphics2D

  Graphics2D g2 = (Graphics2D) g;

 // Perform custom drawing using g2 handle
  .....
}
```

The Java 2D based of the *Graphics2D* class distinguishes between the definition and the drawing of a graphical primitive. All these primitives (definitions) can be manipulated using some general drawing methods which operate on the *Shape* interface. One a graphical primitive object is defined is can be drawn using the *draw()* or *fill()* methods that will receive as argument the object. This is how Java 2D implements the vector representation of the object and the raster drawing of the vector representation of object for the screen. The vector representation is based on floating point arithmetic while the raster drawing is based on integer arithmetic obtained by scan conversion. The abstract class *Shape* has various subclasses for defining graphical primitives. The coordinates used to position and to define the dimensions of the *Shapes* are of *float* or *double* type. The abstract class *Line2D* is used to store lines. A *line* object is created by the following code:

Line2D.Double line = new Line2D.Double(x1, y1, x2, y2);
The *x1, y1, x2, y2* parameters are of *double* type and are representing the coordinates of the start point and the end point of the vector representation of the line object. The line object is drawn only by calling the *draw()* method as follows: *g2.draw(line)*. A *line* object can be used in a simulation to model a link. The *Ellipse2D* abstract class describes an ellipse that is defined by a framing rectangle. If the rectangle is a box then the ellipse is a circle. An *ellipse* object can be created by the following code:

Ellipse2D.Double ellipse = new Ellipse2D.Double(x, y, w, h);

An *ellipse* object can be used in a simulation to model a revolute joint.

The *Path2D* abstract class provides the way of creating a shape of an arbitrary geometric path. For example the following code can be used to create an end effector shaped path:

```
int xpoints[]={ 40, 30, 10, 0, 10, 30, 40 };
int ypoints[]={ 5, 15, 15, 0, -15, -15, -5 };
Path2D PD = new Path2D.Double();
PD.moveTo(xpoints[0],ypoints[0]);
PD.lineTo(xpoints[1],ypoints[1]);
. . .
PD.lineTo(xpoints[6],ypoints[6]);
g2.draw(PD);
```

1.3 Some word on geometric transformations and homogenous coordinates

Computer graphics is using coordinates and parameters to describe geometric objects. The position of the geometric objects can be defined using geometric transformation. The primitive geometric transformations used in simulations are translation, rotation and scaling. All these transformations are carried out with respect of the origin of the coordinate system. A $T(d_x, d_y)$ translation, defined by two distances d_x and d_y, causes the (x,y) point to shift to the (x_T, y_T) point based on the equations:

$$\begin{pmatrix} x_T \\ y_T \end{pmatrix} = \begin{pmatrix} x + d_x \\ y + d_y \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} d_x \\ d_y \end{pmatrix} \quad (1)$$

A rotation $R(\varphi)$, defined by the φ angle, causes the (x,y) point to rotate anticlockwise around the origin to the (x_R, y_R) point based on the equations:

$$\begin{pmatrix} x_R \\ y_R \end{pmatrix} = \begin{pmatrix} x \cdot cos(\varphi) - y \cdot \sin(\varphi) \\ x \cdot sin(\varphi) + y \cdot \cos(\varphi) \end{pmatrix} \quad (2)$$

or

$$\begin{pmatrix} x_R \\ y_R \end{pmatrix} = \begin{pmatrix} cos(\varphi) & -\sin(\varphi) \\ sin(\varphi) & \cos(\varphi) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \quad (3)$$

A $S(s_x, s_y)$ scaling, defined by two scaling factors s_x and s_y, causes the (x,y) point to map to the (x_S, y_S) point based on the equations:

$$\begin{pmatrix} x_S \\ y_S \end{pmatrix} = \begin{pmatrix} s_x \cdot x \\ s_y \cdot y \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \quad (4)$$

If the absolute value of the scaling factor is over one we have a stretching on that direction, if it's less then one we have a shrinking. Negative scaling factors define a reflection with respect of the corresponding axis. Homogenous coordinates use an additional dimension to represent a point. In our case for a 2D Cartesian point a 3D representation is going to be used by adding a w coordinate. Instead of (x,y) we are using (x,y,w) where the correspondence with the 2D Cartesian representation is $(\frac{x}{w}, \frac{y}{w})$, with $w \neq 0$. Any representation of the form $(w \cdot x, w \cdot y, w)$ encodes the same point 2D point (2D Cartesian point can be anywhere on a line that goes through the origin of the homogenous coordinate system). Fixing the value of w to one ($w = 1$) in the homogenous plan the Cartesian plan is represented as a parallel plan at the corresponding w value. The origin of the Cartesian coordinate system corresponds to any point in the homogenous coordinate system with the form $(0,0,w)$. The origin of the 2D Cartesian coordinate system is no longer fixed and can be mapped to another point in homogenous coordinates. The rotation and the scaling primitive transformation in the matrix formulation can be extended to homogenous coordinates in a straightforward way. In the case of translation in homogenous coordinates the following matrix multiplication can be written:

$$\begin{pmatrix} x_T \\ y_T \\ 1 \end{pmatrix} = \begin{pmatrix} x + d_x \\ y + d_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad (5)$$

Rotation and translation transformation preserve lengths and angles, scaling, in general, will not preserve lengths and angles but parallel lines will be mapped to parallel lines. The following table (*Table 1*) gives the primitive geometric transformations matrix description for homogenous coordinates:

Table 1

Primitive transformations in homogenous coordinates.

Transformation	Notation	Matrix
Translation	$T(d_x,d_y)$	$\begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix}$
Rotation	$R(\varphi)$	$\begin{pmatrix} \cos(\varphi) & -\sin(\varphi) & 0 \\ \sin(\varphi) & \cos(\varphi) & 0 \\ 0 & 0 & 1 \end{pmatrix}$
Scaling	$S(s_x,s_y)$	$\begin{pmatrix} 1 & s_x & 0 \\ s_y & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

The homogenous coordinate system has the advantage over the Cartesian 2D coordinate system of describing all the geometric transformation of objects using only matrix multiplication. However, when composing transformation is important to remember that matrix multiplication is a noncommutative operation meaning the order in which the transformation are applied is important (and will lead to different results). An expression of matrix multiplications that results by composing transformations is evaluated from left to right. All matrices that describe primitive geometric transformation in homogenous coordinates have the following form:

$$\begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{pmatrix} \quad (6)$$

As a result, their implementation in graphical libraries is simple, compact and easy to optimize, all coming down to the multiplications of 3x3 matrices.

1.4 Affine transformations in Java

In Java the *java.awt.geom.AffineTransform* class has been created to support affine transformations or geometric transformations in homogenous coordinates implemented by matrices as given in (6) to support operations like translation, rotation or scaling. The most important constructors are:

- *AffineTransform()* which generates the unity matrix that maps every point to itself;
- *AffineTransform(m_{11}, m_{22}, m_{21}, m_{13}, m_{12}, m_{23})* which generates an arbitrary transformation matrix (where all arguments are of type double).

Some of the primitive geometrical transformation methods are:

Rotation

setToRotation(angle) defines a rotation transformation around the origin with *angle*;
setToRotation(angle,x,y) defines a rotation transformation around the *(x,y)* point with *angle*;
getRotateInstance(angle) returns a transform representing a rotation transformation.

If *at* is an *AffineTransform* writing *at.rotation(angle)* or *at.rotation(angle,x,y)* extend the *at* transformation by a rotation that correspond to a multiplication from the right of the *at* transform.

Translation

setToTranslation(dx,dy) defines a translation transformation by the *(dx,dy)*;
at.translate(dx,dy) extends the *at* transformation by a translation as matrix multiplication from the right;
getTranslateInstance(dx,dy) returns a transform representing a translation transformation.

Scaling

setToScale(sx,sy) defines the scaling transformation by the *sx* and *sy* scaling factors for *x* and *y*;
at.scale(sx,sy) extends the *at* transformation by a scaling as matrix multiplication from the right;
getScaleInstance(sx,sy) returns a transform representing a scaling transformation.

Composition

The presented affine transformation can be composed using the following methods:
at.concatenate(at1) to multiply the *at* transformation by *at1* from the right (*at* x *at1*) with the result stores in *at*;
at.oreConcatenate(at1) to multiply the *at* transformation by *at1* from the left (*at1* x *at*) with the result stores in *at*;

Successive transformations are *concatenated*, until it is reset (to the identity transform) or overwritten.

```
// create the identity transform
AffineTransform id = new AffineTransform();
// overwrite the transform associated with the
current Graphics2D context
g2.setTransform(id);
// translates from (0, 0) to the current (x, y)
position
g2.translate(x, y);
// scaling by
g2.scale(sx, sy);
// rotation clockwise about (0, 0), by angle (in
radians)
g2.rotate(angle);
```

The current transformation associated with the Graphics2D context can be saved and restored as follows:

```
//save
AffineTransform saveAt = g2.getTransform();
...
//restore
g2.setTransform(saveAt);
```

2. JAVA SWING MOTION SIMULATION USING AFFINE TRANSFORMATIONS

The following simulation is based on the 2 DOF planar manipulator from [2] (see Figure 2 also). The principles of the Java simulation and robot driving based on a mathematical model and the graphical primitives used locally and in a client-server implementation were discussed in [3] - [9] . Compared to those this implementation evades the mathematical model and completes the simulation using only affine transformations. Also, it demonstrates how to use Swing events to transfer data computed in a *JPanel* to components of the *JFrame* to which the *JPanel* is added. The class diagrams of the implementation are given in Figure 1. The *GrPanel* class does all the work inside the *paintComponent()* method.

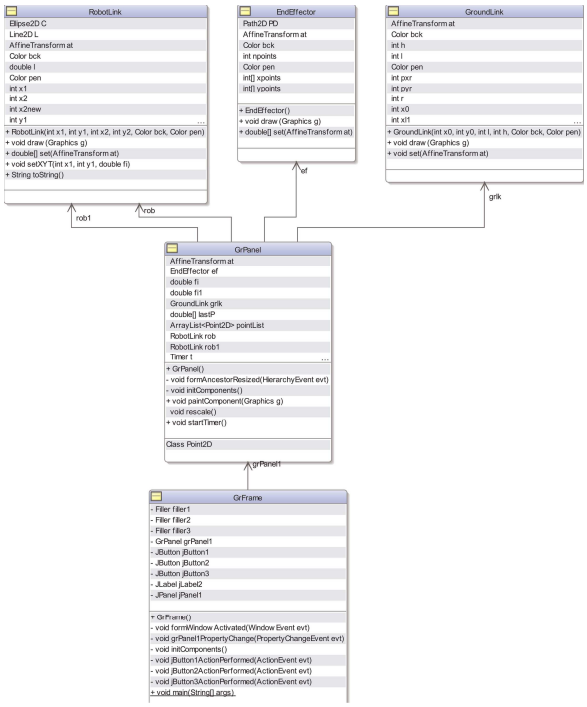

Fig. 1. - Class diagrams of the robot simulator.

```
public void paintComponent(Graphics g) {
// paint parent's background
 super.paintComponent(g);

// draw in RED the points of the end effector
// stored in an ArrayList
 g.setColor(Color.RED);
  for (Point2D p : pointList)
    g.fillOval((int) (p.x - 1), (int) (p.y - 1), 2, 2);

//draw the ground link with translation
// to the  middle of the screen
 at                                                =
AffineTransform.getTranslateInstance(xmid,
ymid);
 grlk.set(at);
 grlk.draw(g);

//change to orientation of the y axis
at.concatenate(AffineTransform.getScaleInstan
ce(1, -1));

//rotate with fi1 angle
```

```
at.concatenate(AffineTransform.getRotateInsta
nce(fi1));
// set the rob object position based on the
// at rotation before drawing and get
// the coordinates of the rotating point
 lastP = rob.set(at);
//draw de rob object
 rob.draw(g);

//translate to the rotating point of rob object
       at                                    =
AffineTransform.getTranslateInstance(lastP[0],
lastP[1]);
// scale the y axis
at.concatenate(AffineTransform.getScaleInstan
ce(1, -1));
// rotate around lastP with fi1 angle
at.concatenate(AffineTransform.getRotateInsta
nce(fi2));
 lastP = rob1.set(at);
 rob1.draw(g);

//add the end effector to the ArrayList
 pointList.add(new Point2D(lastP));

//draw the end effector
 at                                           =
AffineTransform.getTranslateInstance(lastP[0],
lastP[1]);
at.concatenate(AffineTransform.getScaleInstan
ce(1, -1));
at.concatenate(AffineTransform.getRotateInsta
nce(fi2));
 lastP = ef.set(at);
 ef.draw(g);

//fire the property change event for each
// round of computation
 this.firePropertyChange("Label",
"",String.format("(fi1:%-7.3f,fi2:%-7.3f)          -
size:%d ", fi1, fi2, pointList.size()));
}
```

The constructor of the *GrPanel* is:

```
public GrPanel() {
 initComponents();
 setPreferredSize(new Dimension(600, 600));
 xmid = 300;
 ymid = 316;
 grlk = new GroundLink(0, 0, 70,
60,getBackground(), getForeground());
 rob = new RobotLink(0, 0, 50, 0,
getBackground(), getForeground());
 rob1 = new RobotLink(0, 0, 100, 0,
getBackground(), getForeground());
 ef = new EndEffector();
 startTimer();
}
```

As the code from above shows all objects to be drawn are created at position *(0,0)*, then translations, scaling and rotations are used to position them based on the *fi1* and *fi2* angles.

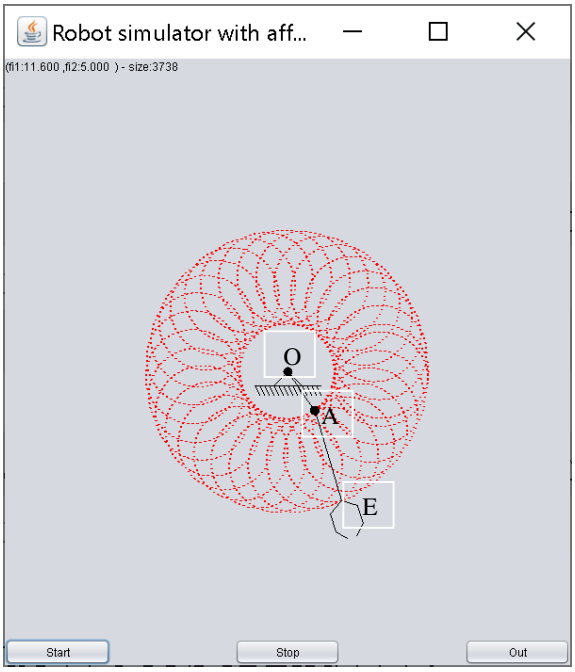

Fig. 2. - Workspace simulation of a 2R planar robot.

The ground link (*grlk* object of the *GroundLink* class) is created at the *(0,0)* point (upper-left corner) then translated to the middle of the JPanel at the *(xmid,ymid)* position and drawn. As this point is the same with the O point from the following link no translation is need to position the next object. The OA link model (*rob* object of class *RobotLink*) is formed of the O joint (black point in O) and the OA link (OA black) line. Again the object is created in the *(0,0)* point. As this object is going to be moved scaling must be applied to make the y axis point up, then a rotation with *fi1* angle is made before drawing the object. The coordinates of the A point have to be extracted after the rotation in the *lastP* two

element array as this is going to be the rotation point for the AE object. This is another object (*rob1*) of class *RobotLink* transformed from the origin representation by a translation to the A point, a scaling for the reversed y axis and a rotation of *fi2* angle. Again the coordinates of the E point are extracted as the end effector must be positioned in the E point and oriented by the fi1 angle along the AE object.

Each time the *PropertyChange* event is fired inside the *paintComponent()* code of the *grPanel1* object of *GrPanel* class the *grPanel1* object on the *GrFrame* that holds *grPanel1* is catching the event by the handler:

```
private void grPanel1PropertyChange(java.
beans.PropertyChangeEvent evt) {//GEN-
FIRST:event_grPanel1PropertyChange
 // TODO add your handling code here:
 if (evt.getPropertyName().equals("Label"))
 jLabel2.setText(evt.getNewValue().toString());
}//
```

The handler sets the contents of the *jLabel2* object to show the angles and the number of points used to draw the end effector positions.

The red points in Figure 2 are positions of the end effector. For each *(fi1,fi2)* variable pair a point is drawn on the *JPanel*. As the number of the points to be drawn is not known in advance the safest way to deal with this situation is to use a List ([2], [5], [6], [8]) to store the points. For this purpose the *ArrayList* was chosen as this is thread safe.

```
ArrayList<Point2D> pointList = new
ArrayList<Point2D>();
```

As the *ArrayList* only works with objects the following inner class was used to store a point:

```
class Point2D {
   private double x, y;
   Point2D(double[] p) {
      x = p[0];
      y = p[1];
   }
}
```

The objects of the *GrLink* class are drawn to the *JPanel* by the following code:

```
public void draw(Graphics g) {
   Graphics2D g2 = (Graphics2D) g;
   g2.setColor(pen);
   g2.draw(at.createTransformedShape(L));
   g2.setColor(Color.BLACK);
   g2.fill(at.createTransformedShape(C));
 }
```

where *L* and *C* are declared as: *Line2D L; Ellipse2D C;* and the *createTransformedShape()* method returns a new object defined by the geometry of the specified *Shape* after it has been transformed by *at* transform. The *at* is set by the *set()* method of the object and determined in the *paintComponent()* of the *JPanel*. This principle is applied to all the objects that make up the moving parts of the robot simulator.

In terms of resources used to run the applications Figure 3 and Figure 4 are showing the differences between the original application from [2] and this one based on affine transformation. As we can see differences in resources are minimal with the exception that v2 has a better memory usage compared to v1 and v1 a bit lower CPU usage. However, the CPU processing time for v2 is half compared to v1. So the major gain is that simulations based on affine transformation that are implemented in Java are faster than those based on the computations resulting from the brute mathematical model of the robot.

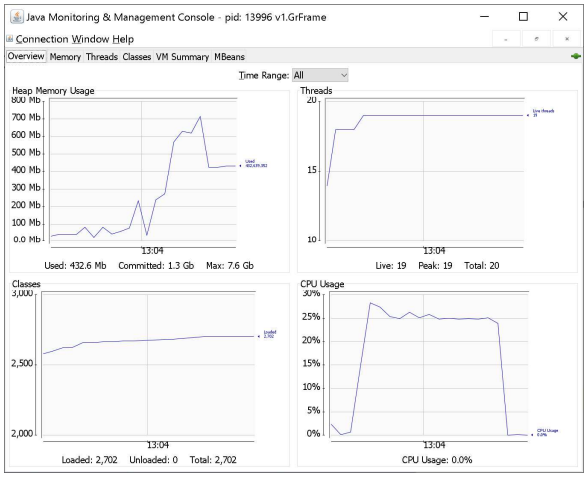

Fig. 3. - Resources for the v1 robot simulator.

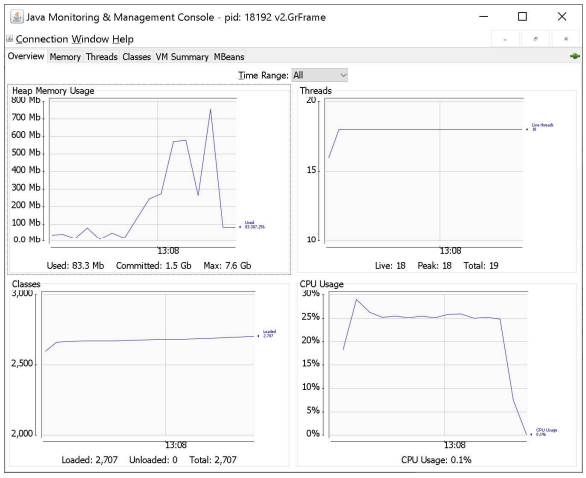

Fig. 4. - Resources for the v2 robot simulator.

3. REFERENCES

[1] https://docs.oracle.com/javase/tutorial/2d/overview/index.html

[2] ANTAL, Tiberiu Alexandru. *Principles of motion simulation of a 2 DOF RR planar manipulator using Java Swing*. Acta Technica Napocensis - Series: Applied Mathematics, Mechanics, and Engineering, v. 63, n. 1, Apr. 2020. ISSN 1221-5872.

[3] ANTAL, T. A., *Elemente de Java cu JDeveloper - îndrumător de laborator*, Editura UTPRES, 2013, p.150, ISBN: 978-973-662-827-6.

[4] ANTAL, T. A., *Java - Iniţiere - îndrumător de laborator*, Editura UTPRES, 2013, p. 246, ISBN: 978-973-662-832-0.

[5] ANTAL, Tiberiu Alexandru; CHELARU, Julieta Daniela. *A multithreaded Java client-server model for robot interaction.* Acta Technica Napocensis - Series: Applied Mathematics, Mechanics, and Engineering, v. 60, n. 3, sep. 2017. ISSN 1221-5872.

[6] ANTAL, Tiberiu Alexandru. *A Java client-server model to solve the forward and the inverse robot kinematics.* Acta Technica Napocensis - Series: Applied Mathematics, Mechanics, and Engineering, v. 62, n. 1, apr. 2019. ISSN 1221-5872.

[7] ANTAL, Tiberiu Alexandru. *3R serial robot control based on arduino/genuino uno, in java, using JDeveloper and Ardulink*. Acta Technica Napocensis - Series: Applied Mathematics, Mechanics, and Engineering, v. 61, n. 1, mar. 2018. ISSN 1221-5872.

[8] ANTAL, Tiberiu Alexandru. *Using networking services for remote access to a Java robot simulator.* Acta Technica Napocensis - Series: Applied Mathematics, Mechanics, and Engineering, v. 63, n. 1, Apr. 2020. ISSN 1221-5872

[9] Tiucă, T., T. Precup, şi T. Antal. *Dezvoltarea aplicaţiilor cu AutoCAD şi AutoLISP*, Editura Promedia Plus Computers, Cluj-Napoca, 1995, p. 304, ISBN: 973-96862-2-2

Utilizarea transformatelor afine din Java în simularea funcţionării cu Swing a unui robot plan 2R

Grafica pe calculator care studiază şi implementează simularea funcţionarii unor roboţi este un domeniu de actualitate. Limbajul Java pune la dispoziţia programatorului, gata implementate, un set de transformări geometrice 2D sub forma unor matrice de transformare care operează în coordonate omogene prin clasa *AffineTransform*. Lucrarea care urmează îşi propune să arate modul în care acestea se utilizează concret în simularea funcţionării unui robot plan 2R şi trage concluzii cu privire la performanţele simulării comparând simularea bazată pe modelul matematic brut cu cea bazată pe transformări afine.

ANTAL Tiberiu Alexandru, Professor, dr. eng., Technical University of Cluj-Napoca, Department of Mechanical System Engineering, antaljr@bavaria.utcluj.ro, 0264-401667, B-dul Muncii, Nr. 103-105, Cluj-Napoca, ROMANIA.