A short course on

GRAPHICAL USER INTERFACE DEVELOPMENT

using Java AWT

© Dr. Elpida TZAFESTAS
ICCS, Athens, GR
brensham@softlab.ece.ntua.gr

Last updated February 2004

Contents

Source Package Contents

DrawingEditor01.java	Basic interface + object input
DrawingEditor02.java	DrawingEditor01 + object selection
DrawingEditor03.java	DrawingEditor02 + object translation
DrawingEditor04.java	DrawingEditor03 + clipboard
DrawingEditor05.java	DrawingEditor04 + object editing
DrawingEditor06.java	DrawingEditor05 + object I/O
DrawingEditor07.java	DrawingEditor06 + operation undo
DrawingEditor08.java	DrawingEditor07 + operation redo
DrawingEditor09.java	DrawingEditor08 + rubberbanding
DrawingEditor10.java	DrawingEditor09 + object rotation
DrawingEditor11.java	DrawingEditor10 + object resizing + animation
DrawingEditor12.java	DrawingEditor11 + align/distribute/equalize
DrawingEditor13.java	DrawingEditor12 + polygon fitting
DrawingEditor14.java	DrawingEditor13 + vertex insertion/deletion
DrawingEditor15.java	DrawingEditor14 + vertex dragging
DrawingEditor16.java	DrawingEditor15 + polygon splitting/joining
DrawingEditor17.java	DrawingEditor16 + background/foreground
DrawingEditor18.java	DrawingEditor17 + tip drawing
DrawingEditor19.java	DrawingEditor18 + F8 through selections
DrawingEditor20.java	DrawingEditor19 + color invertion + transparency
DrawingEditor21.java	DrawingEditor20 + drawing grid
DrawingEditor22.java	DrawingEditor21 + grouping/ungrouping
KeyGame.java	Key game

INSTALLATION NOTE

To use one of the drawing programs, copy it to a folder, rename it ro plain “DrawingEditor.java” and compile and run it as usually.

Chapter 1 Introduction

1.1 Interface types

Graphical user interfaces (GUIs) are nowadays used extensively in lots of environments and applications; for example, in office automation, the use of tools endowed with intricate user-friendly GUIs has spread dramatically in the past ten years and is by now almost exclusive. In general, graphical user interfaces may bear different properties and belong to different classes, according to the type and the features of the application underlying the image and with whom users interact.

In that sense, and according to the static or dynamic character of the application, user interfaces may be classified to one of the following two categories :

- ***Graphical editors***, where the application is static, for instance, during processing of text or drawing of any type, there is a static object (the text or the drawing) that the user may modify in various ways and whose state or form is continuously displayed graphically on the screen. The user has absolute control over this object, being generally unable to modify it in different ways or by different means outside the editor.
- ***Dynamic visualization systems***, where the application is dynamic, that is its state changes dynamically with time, either thanks to some internal process, such as in simulated systems, or because of continuous flow of external data, such as in systems that are regularly updated by external databases or other data sources. Most modern virtual reality tools belong to this category, as well.

Most computer games belong to one of the former two classes, for example crosswords are graphical editors, while adventure and arcade games are dynamic visualization systems. The latter have the additional feature of allowing the user to actively take part in the dynamic process by adopting a particular role, for example in many cases the user takes the role of a movable target being “chased” by a number of enemies.

There are many user interface-based applications combining features from both categories or comprising components and subsystems from both categories. For instance, Integrated Development Environments (IDEs) for specific programming languages comprise components for graphical editing of code and window graphics, together with components for dynamic execution monitoring. The same apply to most visual programming systems. On the other hand, most simulation and/or monitoring systems comprise, beside their dynamic execution and display components, a number of specialized graphical editors for various parameters/objects. Program visualization systems, that attempt to visualize program execution in a visual, user-friendly way, are dynamic visualization tools as well.

User interface systems may be final systems, i.e. self-contained applications, or relatively general-purpose graphical widgets that may be integrated and used in various

applications. For example, a text editor is a self-contained application, whereas a graphical selector of objects from a list is better conceived and implemented as an independent parametric widget that may integrated in various applications.

1.2 Modern computer systems features with application to graphical user interface design

Graphical user interface design and development profits directly from most of the emergent computer technologies and continuously welcomes new possibilities in its own pool of tools and techniques.

The appearance of ***multimedia systems*** that manage text, graphics, sound and video in a uniform interchangeable way, triggered the extensive and rapid spread of applications and especially in areas related to education and entertainment, for example with the emergence of interactive learning systems for a wealth of educational subjects at all kinds of levels. Modern user interfaces are therefore not just plain graphical interfaces but complex multimedia interfaces, where knowledge representation may encompass textual, graphical, sound or video media in a dynamic way. Interaction with the user is actually limited to just textual and graphical processing and manipulation, but is being extended to include ways of interaction with other media.

Hypermedia systems, that are older than general multimedia systems, allow the system engineer to organize information in networked ways, where the user has direct access to information of various forms available at various "addresses" worldwide and are interconnected via links of various types. The development and spread of the World Wide Web provided the opportunity of connecting an application and its user interface to the external world in order to refer to external sources, obtain new data, update continuously etc.

The static nature of hypermedia systems prompted for the need to extend networked applications and services with dynamicity, i.e. to allow users to execute programs and applications situated in remote network sites. As a first step, *CGI scripts* (*Common Gateway Interface scripts*) were developed, that are programs available for execution by remote users over the Internet. CGI scripts present two major drawbacks regarding interaction with the user. Firstly, they accept as input command lines, that is primarily textual data (for example, copy "a" "b"). Secondly, interaction is the least possible, it usually occurs just once in the beginning of the execution, when all the necessary data is fed to a program that executes and outputs the results, again in textual form. While it is fairly easy to imagine continuous interaction through interconnected scripts, this is in practice hard to implement and conceptually confusing. In any case, it is impossible to have the user interact with the system in a graphical and direct way. The Java programming language, that first appeared in 1995, answered to those needs in an elegant and systematic way by allowing the development of complete graphical applications (applets) that may be run across the WWW. Today, ***network applications*** are so common that this possibility evolves toward a prerequisite for any application.

The most recent technological development with an immediate application to graphical user interface design and development is the so-called ***component-based programming***. Component-based programming is the successor of the traditional object-oriented programming paradigm and is targeted to the homogenization and prototyping of application design, so as to allow individual applications to be freely interconnected and integrated as components of other more complex systems without drastically intervening at the source code level. Furthermore, component-based programming purports to be the framework that will allow the coexistence of applications written in different languages and with different tools. Management in such a context should be made in a simple and straightforward way, hence graphically. According to this principle, every application or program that is supposed to act as a component of a large system, should be accompanied by an appropriate customization interface that allows the designer or the user to adapt it to a particular context and integrate it with other components with minimal effort. The most common component architectures are JavaBeans by Sun (based on the Java programming language) and ActiveX by Microsoft (based on the OLE/COM/DCOM or Object Linking and Embedding/Component Object Model/Distributed Component Object Model). The CORBA (Common Object Request Broker Architecture) protocol by OMG (Object Management Group), specifies in detail the features of programs that serve as components. The ultimate goal of this technology is the systematization and simplification of the design of distributed applications, that comprise many independent components situated at discrete network addresses.

Chapter 2 Graphical editors

2.1 Introduction

A ***graphical editor*** is a program allowing the presentation and processing of the "content" or the parameters of an object in a *graphical* (*visual*) way, for instance a word processor allows the presentation and processing of formatted text in a graphical way and precisely in its printable form.

Many of the best known applications with a graphical user-interface are graphical editors (for example, word processors, spreadsheets and all office automation tools), while many specialized and complex systems encompass a number of special-purpose graphical editors. For example, an integrated development environment for a programming language usually comprises among other things a code editor, a user interface editor, an image editor etc.

Finally, the relatively recent research domain of ***visual*** or ***graphical programming*** is concerned exclusively with methods and tools for the presentation and manipulation of arbitrary structures or objects in a graphical way and seeks to the improvement of the quality of user interaction with more traditional, generally non-commercial, applications. As an example of this approach, a system of graphical programming for scientific experimentation does not have the goal of gaining access to new non-scientific audiences, but to accelerate and systematize the experimentation procedures.

Many graphical editors present and allow the manipulation of a set of numeric or symbolic parameters or properties of a target object. Such editors include the Windows 95 και NT property sheets, such as the page setup popup dialog. From a programming and development point of view, the hardest but most challenging are those that manipulate a graphical representation of a complex object, such as an electronic circuit.

Most of them are targeted toward objects that belong to one of the following categories :

- Text. For example, the code processor for a programming language, as mentioned earlier.
- A set of independent objects in different positions of the screen. For example, an architectural drawing may be conceived as a set of objects (walls, doors, windows etc.) that may be positioned in an arbitrary way.
- A set of independent objects in predefined positions of the screen. For example, a digital image where the position of each pixel on the screen is predefined and depends just on the current presentation scale, or a mathematical equation that may be represented as a row of objects (symbols, variables, functions, embedded equations etc.).
- A set of interdependent objects. For example, an electronic circuit comprises a set of objects such as resistances, diods, transistors and other elements interconnected via

cables. The repositioning of an element in such a drawing does not change its relationships with the other elements.

More intricate editing applications not falling clearly on some of the above categories are sheet music processing, biomolecular structures editing (where the relationships between composing elements are not purely positional but they include complex chemical dependencies), sound processing (where some sort of conventional representation is used, such as the wave representation of sound signals) and so forth.

In what follows we will describe and develop step by step the operations of a typical graphical editor, based on the example of a simple drawing editor that uses just circles, lines and polygons. A snapshot of this editor is given in figure 2.1

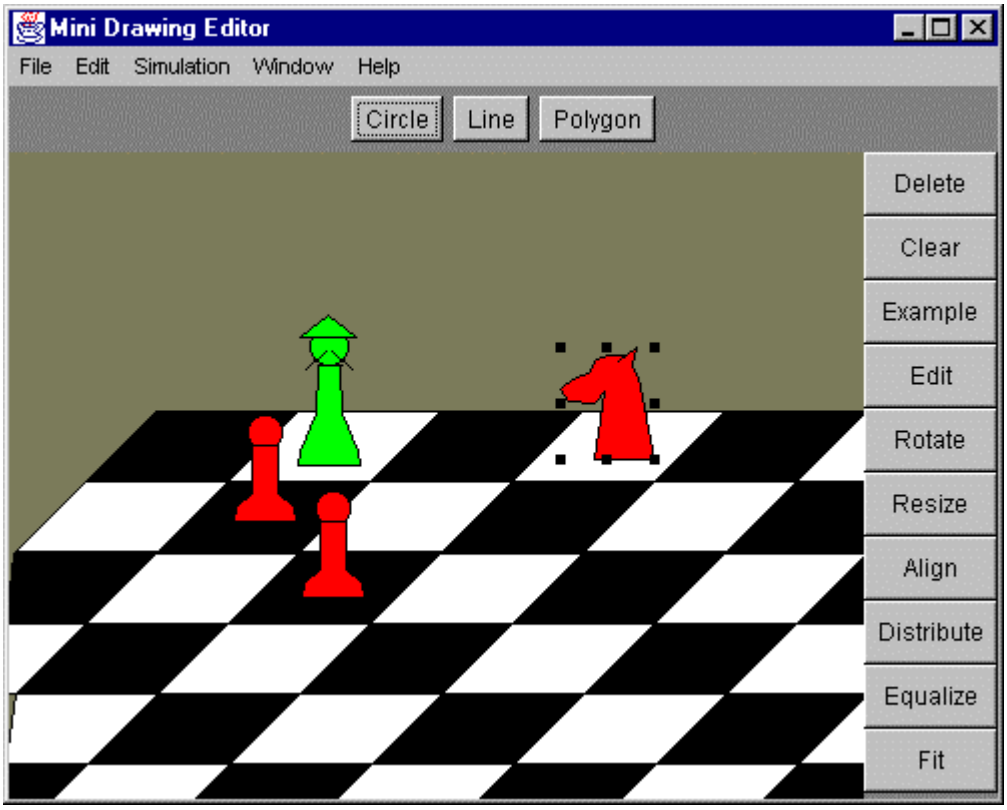

Figure 2.1. A snapshot of a simple drawing editor.

2.2 Elementary operations of graphical editing

2.2.1 Object input

Our example editor supports three types of shape, the circle, the line and the polygon. Input is done by first pressing the button that corresponds to the desired shape and subsequently clicking twice or more times on the desired positions to actually define and position such a shape on the screen : to input a circle the user clicks on two antidiametrical positions, to input a line (segment) the user clicks on its two ends, and

finally to input a polygon a user clicks continuously to define the series of vertices (the polygon is finalized when the user double-clicks or presses Control-C or when the polygon has reached a maximum of 30 vertices). The subsequent points clicked upon by the user are visualized to track input process (see figure 2.2).

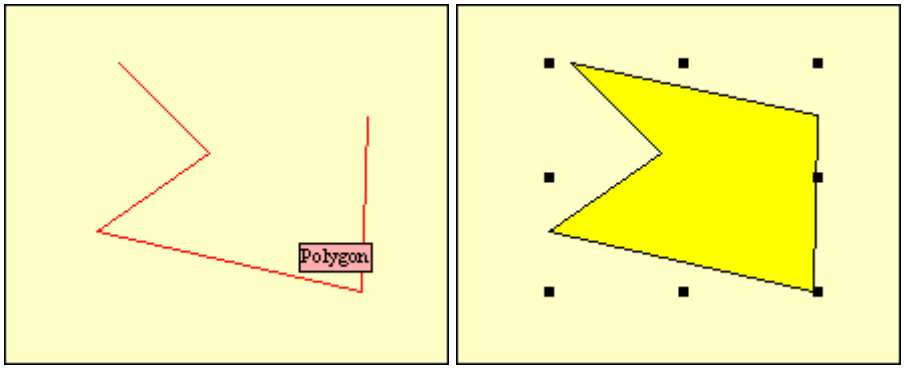

Figure 2.2. Intermediate and final state of a polygon input.

The above are implemented as follows* :

```
public class DrawingEditor extends Applet
      implements Constants, MouseListener, KeyListener, ActionListener
{
// Shapes input by the user
Drawing target;

// Components
Button circle, line, polygon;

// Positions defined with the mouse
Vector positions = new Vector(10,5);
int curr_x = -1, curr_y = -1;

// Variable used for input
int input_flag = 0;

// Colors
static Color BackgroundColor = Color.white;
static Color InputColor = Color.red;

// Sizes
public static int SizeX=500, SizeY=400;

public void actionPerformed(ActionEvent evt)
{
      if (evt.getSource() instanceof Button)
      {
            Button tgt = (Button)evt.getSource();
```

* This section describes just the code necessary to implement the corresponding operation. For further details as well as for the way the various operations are composed in a consistent way to form the final program, see the complete programs (DrawingEditor01-22.java) given along with this tutorial.

```
            if (tgt == line) input_flag = LINE;
            else if (tgt == circle) input_flag = CIRCLE;
            else if (tgt == polygon) input_flag = POLYGON;
      }
}


public void mousePressed(MouseEvent evt)
{
      int x = evt.getX(), y = evt.getY();
      Shape comp;

      boolean
            shift = evt.isShiftDown(),
            control = evt.isControlDown(),
            alt = evt.isMetaDown() || evt.isAltDown();

      // Input of a shape
      if (!target.includes(x,y)) {reset_input(); repaint(); return;}
      if (input_flag == LINE)
      {
            positions.addElement((Object)(new Point(x,y)));
            curr_x = x; curr_y = y;
            if (positions.size() == 2)
            {
                  comp = new Shape(LINE,
                              (Point)positions.elementAt(0),
                              (Point)positions.elementAt(1));
                  target.add(comp); reset_input();
            }
            repaint(); return;
      }
      else if (input_flag == CIRCLE)
      {
            positions.addElement((Object)(new Point(x,y)));
            curr_x = x; curr_y = y;
            if (positions.size() == 2)
            {
                  comp = new Shape(CIRCLE,
                              (Point)positions.elementAt(0),
                              (Point)positions.elementAt(1));
                  target.add(comp); reset_input();
            }
            repaint(); return;
      }
      else if (input_flag == POLYGON)
      {
            positions.addElement((Object)(new Point(x,y)));
            curr_x = x; curr_y = y;
            if ((positions.size() == Shape.SizeLimit)
                  || (evt.getClickCount() == 2))
            {
                  comp = new Shape(POLYGON,positions);
                  target.add(comp); reset_input();
            }
            repaint(); return;
```

```
	}
}

public void mouseMoved(MouseEvent evt)
{
	int x = evt.getX(), y = evt.getY();
	Shape el;

	if (input_flag != 0)
		if (positions.size() > 0)
		{curr_x = x; curr_y = y; repaint();}
}

public void keyPressed(KeyEvent evt)
{
	char ch = evt.getKeyChar();

	boolean shift = evt.isShiftDown();
	boolean control = evt.isControlDown();
	boolean alt = (evt.isMetaDown() || evt.isAltDown());

	boolean onlyShift = shift && (!control) && (!alt);
	boolean onlyControl = (!shift) && (control) && (!alt);
	boolean onlyAlt = (!shift) && (!control) && (alt);
	if (((ch == 'c') || (ch == 'C')) && onlyControl)
	{
		if (input_flag == POLYGON)
		{
			if (positions.size() > 0)
			{
				Shape comp = new Shape(POLYGON,positions);
				target.add(comp); reset_input();
			}
			repaint();
		}
	}
}

void reset_input()
{
	positions.removeAllElements();
	input_flag = 0;
	curr_x = -1;
	curr_y = -1;
}

void drawPositions(Graphics g)
{
//    This method tracks user input in red : all given points are drawn
//    and each one of them is connected to the next one with a straight
//    line. This arrangement is particularly useful for polygon input,
//    where one can see the intermediate state of the polygon as it
//    evolves. Circle and line input is traced in a similar manner.
//    The final mouse pointer is also tracked as moved around.

	Point prev, curr;
```

```
        Color pen = g.getColor();
        g.setColor(InputColor);

        if ((input_flag == POLYGON) || (input_flag == LINE))
        {
              if (positions.size() > 0)
              {
                    prev = (Point)positions.elementAt(0);
                    drawPoint(g,prev);
                    for (int i=1;i<positions.size();i++)
                    {
                          curr = (Point)positions.elementAt(i);
                          g.drawLine(prev.x,prev.y,curr.x,curr.y);
                          prev=curr;
                    }
                    g.drawLine(prev.x,prev.y,curr_x,curr_y);
              }
        }
        else if (input_flag == CIRCLE)
        {
              if (positions.size() > 0) // It will be necessarily 1.
              {
                    prev = (Point)positions.elementAt(0);
                    drawPoint(g,prev);

              // Draw the circle from (prev.x,prev.y)
              // to (curr_x,curr_y)
                    int diameter = (int)Math.sqrt(((prev.x-curr_x)
                                          *(prev.x-curr_x))
                                + ((prev.y-curr_y)*(prev.y-curr_y)));
                    Point up = new Point((prev.x+curr_x-diameter)/2,
                                         (prev.y+curr_y-diameter)/2);
                    g.drawOval(up.x,up.y,diameter,diameter);
              }
        }

        g.setColor(pen);
}

void drawPoint(Graphics g, Point po) {g.drawLine(po.x,po.y,po.x,po.y);}

public void paint(Graphics g)
{
        Color pen = g.getColor();
        super.paint(g);
        g.setColor(BackgroundColor);
        g.fillRect(0,0,SizeX,SizeY);
        g.setColor(pen);
        target.draw(g);
        drawPositions(g);
}

}       // End DrawingEditor

class Drawing implements Constants
{
```

```
Vector shapes;
Dimension size;

DrawingEditor parent;

//    Light yellow background
Color backgroundColor=LightYellow;

public boolean includes(int x, int y)
{
      if ((x>0) && (x<size.width) && (y>0) && (y<size.height))
            return true;
      return false;
}

public void add(Shape comp)
      { shapes.addElement(comp); }

public void draw(Graphics g)
{
      Color c = g.getColor();

      g.setColor(backgroundColor);
      g.fillRect(0,0,size.width,size.height);
      g.setColor(c);

      for (int i=0;i<shapes.size();i++)
            ((Shape)(shapes.elementAt(i))).draw(g);
}

}     // End Drawing

class Shape implements Constants
{
Color fillColor = Color.yellow;
Color borderColor = Color.black;
static int SizeLimit = 30;
int type;
int size;
int xpoints[] = new int[SizeLimit];
int ypoints[] = new int[SizeLimit];

Shape(int typ, Point start, Point end)
{
      // Line or circle
      switch (typ)
      {
            case LINE:
            default:
                  type = LINE;
                  xpoints[0]=start.x; xpoints[1]=end.x;
                  ypoints[0]=start.y; ypoints[1]=end.y;
                  size=2;
                  break;
            case CIRCLE:
                  type = CIRCLE;
```

```
                    int cx = (start.x+end.x)/2;
                    int cy = (start.y+end.y)/2;
                    int r = (int)(Math.sqrt(
                                ((start.x-end.x)*(start.x-end.x))
                                + ((start.y-end.y)*(start.y-end.y))) / 2);
                    xpoints[0]= cx - r; xpoints[1]= cx + r;
                    ypoints[0]= cy - r; ypoints[1]= cy + r;
                    size=2;
                    break;
        }
}

Shape(int typ, Vector points)
{
        // Polygon
        Point temp;

        type = POLYGON;
        for (int i=0;i<points.size();i++)
        {
                temp = (Point)points.elementAt(i);
                xpoints[i]=temp.x; ypoints[i]=temp.y;
        }
        size = points.size();
}

void draw(Graphics g)
{
        int diameter;
        Color pen = g.getColor();

        switch (type)
        {
                case LINE:
                        g.setColor(borderColor);
                        g.drawLine(xpoints[0],ypoints[0],
                                    xpoints[1],ypoints[1]);
                        g.setColor(pen);
                        break;
                case CIRCLE:
                        diameter = xpoints[1] - xpoints[0];
                        g.setColor(fillColor);
                        g.fillOval(xpoints[0],ypoints[0],diameter,diameter);
                        g.setColor(borderColor);
                        g.drawOval(xpoints[0],ypoints[0],diameter,diameter);
                        g.setColor(pen);
                        break;
                case POLYGON:
                        g.setColor(fillColor);
                        g.fillPolygon(xpoints,ypoints,size);
                        g.setColor(borderColor);
                        g.drawPolygon(xpoints,ypoints,size);
                        g.setColor(pen);
                        break;
        }
}
```

```
}     // End Shape

interface Constants
{
// Constants used for input
static int LINE = 1;
static int CIRCLE = 2;
static int POLYGON = 3;
}     // End Constants
```

2.2.2 Object selection

A fundamental operation of any graphical editor is that of selecting one or more objects by mouse-clicking on the screen. The selected object(s) will be subsequently used in another operation, for example to move them around with the keyboard arrows, to delete or copy them with a combination of keyboard keys etc.

Our example editor supports selection of a single object by clicking within the area of the object (shape) and multiple selection (that is, selection of many objects simultaneously) when the Shift key is kept pressed while clicking. Selection of all objects with Control-a[†] is also supported. Each selected object is drawn on the screen in the typical way used by most drawing programs, i.e. with small black squares marking the corners and the middles of the edges of the surrounding rectangle (see the little horse in figure 2.1).

The above are implemented as follows[*] :

```
public class DrawingEditor extends Applet
      implements Constants, MouseListener, KeyListener, ActionListener
{
. . . . . .
// Selections
Vector selections = new Vector(10,5);

public void mousePressed(MouseEvent evt)
{
      . . . . . .
      // Selection
      int i = target.shapes.size()-1;
      while (i >= 0)
      {
            comp = (Shape)target.shapes.elementAt(i);
            if (comp.includes(evt.x,evt.y))
            {
                  if (!shift) clearSelections();
                  select(comp);
                  repaint(); return;
```

[†] Some key combinations that use the Control modifier may not work on some computers, because of possible confusion with existing system shortcuts. Control-a, as well as Control-c, control-x and Control-v, that will be found later, belong to this category. To bypass this difficulty, our drawing editor also supports the corresponding Alt conbinations. For example, if Control-a does not work in your system, try Alt-a instead.

```
            }
            i--;
      }
      clearSelections();
      repaint();
}

public void keyPressed (KeyEvent evt)
{
      . . . . . .
      if (((ch == 'a') || (ch == 'A')) && onlyControl)
            {selectAll(); repaint(); return;}
}

public void clearSelections()
{
      for (int i=0;i<selections.size();i++)
      {
            Shape comp = (Shape)selections.elementAt(i);
            comp.deselect();
      }
      selections.removeAllElements();
}

void selectAll()
{
      clearSelections();
      for (int i=0;i<target.shapes.size();i++)
            select((Shape)target.shapes.elementAt(i));
      repaint();
}

void select(Shape s)
{
      if (s.isSelected()) return;
      s.select();
      selections.addElement(s);
}

void deselect(Shape s)
{
      s.deselect();
      selections.removeElement(s);
}

}     // End DrawingEditor

class Shape implements Constants
{
. . . . . .
protected boolean selected = false;

public void select() {selected = true;}
public void deselect() {selected = false;}
public boolean isSelected() {return selected;}
```

```
void draw(Graphics g)
{
      drawUnselected(g);
      if (selected) drawSelected(g);
}

void drawUnselected(Graphics g)
{
//    The previous draw(Graphics g)
}

void drawSelected(Graphics g)
{
      switch (type)

      {
            case LINE: lineDrawSelected(g); break;
            case CIRCLE: circleDrawSelected(g); break;
            case POLYGON: polygonDrawSelected(g); break;
      }
}

void lineDrawSelected(Graphics g)
{
      drawSelectedPoint(g, xpoints[0], ypoints[0]);
      drawSelectedPoint(g, (xpoints[0] + xpoints[1]) / 2,
                        (ypoints[0] + ypoints[1])/ 2);
      drawSelectedPoint(g, xpoints[1], ypoints[1]);
}

void circleDrawSelected(Graphics g)
{
      drawSelectedRectangle(g,new Point(xpoints[0],ypoints[0]),
                                    xpoints[1]-xpoints[0],
                                    ypoints[1]-ypoints[0]);
}

void polygonDrawSelected(Graphics g)
{
      . . . . . .
      drawSelectedRectangle(g,upperleft,width,height);
}

void drawSelectedRectangle(Graphics g, Point upperleft,
            int width, int height)
{
      if (selected)
      {
      drawSelectedPoint(g, upperleft.x, upperleft.y);
      drawSelectedPoint(g, upperleft.x, upperleft.y + (height/2));
      drawSelectedPoint(g, upperleft.x, upperleft.y + height);
      drawSelectedPoint(g, upperleft.x + (width / 2), upperleft.y);
      drawSelectedPoint(g, upperleft.x + (width / 2),
                  upperleft.y + height);
      drawSelectedPoint(g, upperleft.x + width, upperleft.y);
      drawSelectedPoint(g, upperleft.x + width,
```

```
                  upperleft.y + (height/2));
      drawSelectedPoint(g, upperleft.x + width,
                  upperleft.y + height);
      }
}

public void drawSelectedPoint(Graphics g, int x, int y)
      { g.fillRect(x-2, y-2, 5, 5); }

public boolean includes (int x, int y)
{
      switch (type)
      {
            case LINE: return lineIncludes(x,y);
            case CIRCLE: return circleIncludes(x,y);
            case POLYGON: return polygonIncludes(x,y);
      }
      return false;
}

public boolean lineIncludes(int x,int y)
{
      . . . . . .
      if ((x >= lowX) && (x <= upX) && (y >= lowY) & (y <= upY))
            return true;
      return false;
}

public boolean circleIncludes(int x, int y)
{
      return rectangleIncludes(new Point(xpoints[0],ypoints[0]),
                                    xpoints[1]-xpoints[0],
                                    ypoints[1]-ypoints[0],x,y);
}

public boolean polygonIncludes(int x, int y)
{
      . . . . . .
      return rectangleIncludes(upperleft, width, height, x, y);
}

public boolean rectangleIncludes(Point upperleft, int width,
                        int height, int x, int y)
{
      if ((x >= upperleft.x) && (x <= (upperleft.x+width))
                  && (y >= upperleft.y) && (y <= (upperleft.y+height)))
            return true;
      return false;
}

}     // End Shape
```

2.2.3 Object translation

Our example editor supports translation of selected objects with the arrow keys in the corresponding directions by 1 pixel at a time or by 10 pixels when the Alt key is pressed simultaneously. Translation of the selected objects by mouse dragging is also supported. In the latter case, translation is validated only if the final release position lies within the limits of the drawing area, while during dragging the objects are visualized normally in their original position plus as their outline in the intermediate position (see figure 2.3).

The above are implemented as follows* :

```
public class DrawingEditor extends Applet
      implements Constants, MouseListener, MouseMotionListener,
            KeyListener, ActionListener
{
. . . . . .
// Variables used for shapes dragging : flag and previous drag position
boolean shapeMouseDrag = false;
int prev_x = -1, prev_y = -1;

public void mousePressed(MouseEvent evt)
{
      . . . . . .
      // Selection with dragging
      int i = target.shapes.size()-1;
      while (i >= 0)
      {
            comp = (Shape)target.shapes.elementAt(i);
            if (comp.includes(evt.x,evt.y))
            {
                  if (!shift) clearSelections();
                  select(comp);

                  // Start dragging selections
                  setShapeMouseDrag();
                  prev_x = x; prev_y = y;
                  for (int ind=0;ind<selections.size();ind++)
                        ((Shape)selections.elementAt(ind)).
                              startDragging();

                  repaint(); return;
            }
            i--;
      }
      clearSelections();
      repaint();
}

public void mouseReleased(MouseEvent evt)
{
      int x = evt.getX(), y = evt.getY();
      Shape comp;

      if (!shapeMouseDrag) return;
```

```
        if (shapeMouseDrag)
        {
              resetShapeMouseDrag();
              if (!target.includes(x,y)) return;
              Shape sel = (Shape)selections.elementAt(0);
              if (sel.hasBeenDragged())
              {
                    for (int i=0;i<selections.size();i++)
                    {
                          comp = (Shape)selections.elementAt(i);
                          comp.dragBy(x-prev_x,y-prev_y);
                          comp.confirmDragged(target);
                    }
              }
              prev_x = -1; prev_y = -1;
        }
        repaint();
}
```

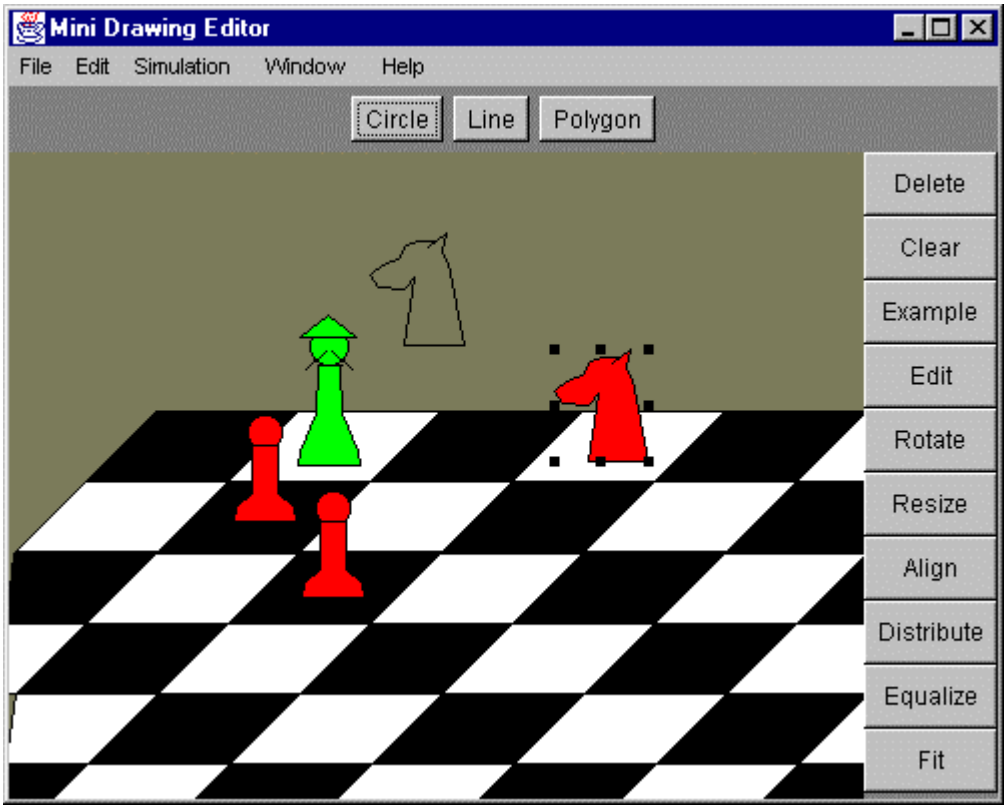

Figure 2.3. While dragging the little horse around with the mouse, only its outline is visualized in the intermediate position, and the final release position is validated only if it lies within the limits of the drawing area.

```
public void mouseDragged(MouseEvent evt)
{
        Shape comp;
        int x = evt.getX(), y = evt.getY();
```

```
      if (!shapeMouseDrag) return;
      if (shapeMouseDrag)
      {
            for (int i=0;i<selections.size();i++)
            {
                  comp = (Shape)selections.elementAt(i);
                  comp.dragBy(x-prev_x,y-prev_y);
            }
            prev_x = x; prev_y = y;
      }
}

public void keyPressed(KeyEvent evt)
{
      int key = evt.getKeyCode();
      . . . . . .

      // Move with the arrows
      if (key == KeyEvent.VK_LEFT)
      {
            if ( (!shift) && (!control) && (!alt) )
            {     // Move by 1
                  for (int i=0;i<selections.size();i++)
                  {
                        comp = (Shape)selections.elementAt(i);
                        comp.moveBy(-1,0);
                  }
                  repaint(); return;
            }
            if ( (!shift) && (!control) && (alt) )
            {     // Move by 10
                  for (int i=0;i<selections.size();i++)
                  {
                        comp = (Shape)selections.elementAt(i);
                        comp.moveBy(-10,0);
                  }
                  repaint(); return;
            }
      }
      // Similar things for the rest of the keys (RIGHT, DOWN and UP)
}

public void paint(Graphics g)
{
      . . . . . .
      if (shapeMouseDrag)
            for (int i=0;i<selections.size();i++)
                  ((Shape)selections.elementAt(i)).
                        drawDragged(g);
}

private void setShapeMouseDrag() { shapeMouseDrag = true;}
private void resetShapeMouseDrag() { shapeMouseDrag = false;}

}     // End DrawingEditor
```

```
class Shape implements Constants
{
. . . . . .
Point dragPosition = new Point(0,0);

void drawDragged(Graphics g)
{ drawDraggedAt(g,dragPosition.x,dragPosition.y); }

void drawDraggedAt(Graphics g, int x, int y)
{
      Point up, offset, pos = getPosition();
      int diameter;
      Color pen = g.getColor();

      offset = new Point(x - getPosition().x, y - getPosition().y);

      switch (type)
      {
            case LINE:
                  g.setColor(borderColor);
                  g.drawLine(xpoints[0]+offset.x,ypoints[0]+offset.y,
                              xpoints[1]+offset.x,ypoints[1]+offset.y);
                  g.setColor(pen);
                  break;
            case CIRCLE:
                  g.setColor(borderColor);
                  g.drawOval(xpoints[0]+offset.x,ypoints[0]+offset.y,
                              xpoints[1]-xpoints[0],
                              ypoints[1]-ypoints[0]);
                  g.setColor(pen);
                  break;
            case POLYGON:
                  g.setColor(borderColor);
                  int newxpoints[] = new int[SizeLimit];
                  int newypoints[] = new int[SizeLimit];
                  for (int i=0;i<size;i++)
                  {
                        newxpoints[i]=xpoints[i]+offset.x;
                        newypoints[i]=ypoints[i]+offset.y;
                  }
                  g.drawPolygon(newxpoints,newypoints,size);
                  g.setColor(pen);
                  break;
      }
}

public void moveBy(int x, int y)
{
      int i;
      for (i=0;i<xpoints.length;i++) xpoints[i]+=x;
      for (i=0;i<ypoints.length;i++) ypoints[i]+=y;
}

public void dragBy(int x, int y)
      {dragPosition.x +=x; dragPosition.y +=y;}
```

```
public void startDragging()
      {dragPosition.x = getPosition().x;
      dragPosition.y = getPosition().y;}

public boolean hasBeenDragged()
{
      Point pos = getPosition();
      return ((dragPosition.x != pos.x) || (dragPosition.y != pos.y));
}

public void confirmDragged(Drawing target)
{
      if
      (target.includes(dragPosition.x,dragPosition.y) &&
      target.includes(dragPosition.x+width(),dragPosition.y+height()))
      // then
            setPosition(dragPosition.x,dragPosition.y);
}

public void setPosition(int x, int y)
{
//    Find leftmost and topmost
      . . . . . .
// Reset topmost and leftmost (reset all points relative to them)
      for (i=0;i<size;i++) xpoints[i] += (x-leftmost);
      for (j=0;j<size;j++) ypoints[j] += (y-topmost);
}

public void setPosition(Point po)
      {setPosition(po.x,po.y);}

}     // End Shape
```

2.2.4 Clipboard

Another very common and useful operation of a graphical editor is that of cutting, copying and pasting objects, i.e. the clipboard operations.

Our example editor supports cut, copy and paste operations for the selected objects with the key combinations Control-x, Control-c and Control-v, respectively [†]. In the latter case, all the objects in the cliboard are inserted with the upper left corner of their surrounding rectangle on the position of the latest mouse click. The deletion of selected objects is also possible with the special DELETE and BACKSPACE keys or with the "Delete" button, without using the clipboard.

The above are implemented as follows[*] :

```
public class DrawingEditor extends Applet
      implements Constants, MouseListener, MouseMotionListener,
            KeyListener, ActionListener
{
. . . . . .
Button deleteButton;
```

```
// Clipboard
Vector clipboard = new Vector(10,5);

// Position set with the mouse (used for pasting)
int posX = -1, posY = -1;

public void actionPerformed(ActionEvent evt)
{
      . . . . . .
      else if (tgt == deleteButton)
      {
            deleteSelections();
            repaint();
      }
}

public void mousePressed(MouseEvent evt)
{
      . . . . . .
      // Selection with dragging and position setting
      {
            posX = x; posY = y;
            int i = target.shapes.size()-1;
            while (i >= 0)
            {
                  comp = (Shape)target.shapes.elementAt(i);
                  if (comp.includes(evt.x,evt.y))
                  {
                        if (!shift) clearSelections();
                        comp.select(); selections.addElement(comp);
                        // Start dragging selections
                        . . . . . .
                        posX = x; posY = y;
                        repaint(); return;
                  }
            i--;
            }
            clearSelections();
            repaint();
      }
}

public void keyPressed(KeyEvent evt)
{
      . . . . . .
      boolean del=false;

      if ( (key == KeyEvent.VK_BACK_SPACE) ||
                  (key == KeyEvent.VK_DELETE) )
            del=true;

      if (del)
      {
            deleteSelections();
            repaint(); return;
```

```
        }
        if (((ch == 'c') || (ch == 'C')) && onlyControl)
        {
              if (input_flag == POLYGON)
                    { . . . /* Close polygon */ . . . }
              // Copy selections
              copy(); return;
        }
        if (((ch == 'x') || (ch == 'X')) && onlyControl)
        {     // Cut selections
              cut(); return;
        }
        if (((ch == 'v') || (ch == 'V')) && onlyControl)
        {     // Paste selections
              paste(); return;
        }

}

void copy()
{
      clipboard.removeAllElements();
      for (int i=0;i<selections.size();i++)
      {
            Shape comp = ((Shape)selections.elementAt(i)).copy();
            comp.deselect();
            clipboard.addElement(comp);
      }
}

void cut()
{
      clipboard.removeAllElements();
      for (int i=0;i<selections.size();i++)
      {
            Shape comp = (Shape)selections.elementAt(i);
            clipboard.addElement(comp);
            target.remove(comp);
      }
      clearSelections();
      repaint();
}

void paste()
{
      if ((posX < 0) || (posY < 0)) return;
      for (int i=0;i<clipboard.size();i++)
      {
            Shape comp = (Shape)clipboard.elementAt(i);
            comp.setPosition(posX,posY);
            comp.deselect();
            comp = comp.copy();
            target.add(comp);
      }
      repaint();
}
```

```
public void deleteSelections()
{
      for (int i=0;i<selections.size();i++)
      {
            Shape comp = (Shape)selections.elementAt(i);
            target.remove(comp);
      }
      clearSelections();
}

}     // End DrawingEditor

class Shape implements Constants
{
. . . . . .
Shape copy()
{
      return new Shape(type, xpoints, ypoints, size,
                        fillColor, borderColor);
}

}     // End Shape
```

2.2.5 Individual object editing

Another common feature of a graphical editor is that of editing of an individual object.

Our example editor supports manipulation of the border and fill color of a selected object by double clicking on it, provided it is the only selected object (see figure 2.4). This operation is also possible through the use of the "Enter" or the "Return" keys or through the "Edit" button.

The above are implemented as follows* :

```
public class DrawingEditor extends Applet
      implements Constants, MouseListener, MouseMotionListener,
            KeyListener, ActionListener
{
. . . . . .
Button editButton;

public void actionPerformed(ActionEvent evt)
{
      . . . . . .
      else if (tgt == editButton)
      {
            if (selections.size() == 1)
            {
                  ((Shape)selections.elementAt(0)).edit(this);
                  return;
            }
            else if (selections.size() > 1)
            {
```

```
                MessageDialog.open(
            "Warning : Cannot edit multiple selections at once!");
                        return;
                }
        }
}
```

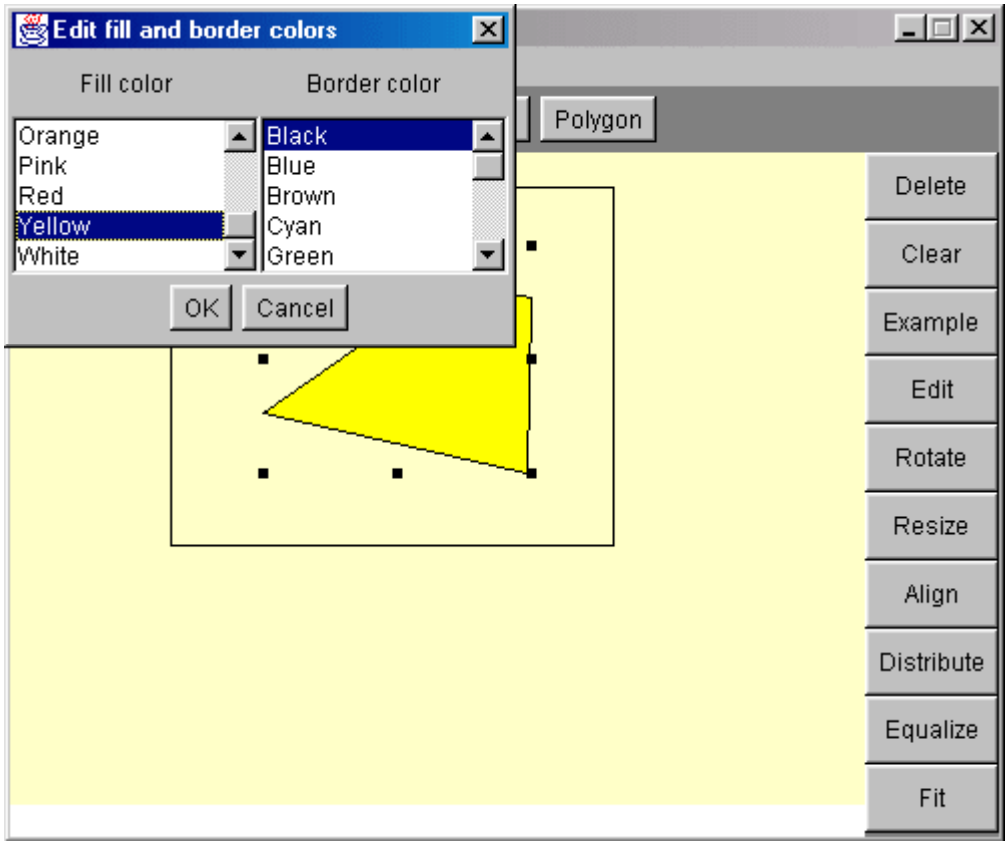

Figure 2.4. Fill and color editing.

```
public void keyPressed(KeyEvent evt)
{
      . . . . . .
      boolean edit=false;

      if (key == KeyEvent.VK_ENTER) edit=true;
      if (edit)
      {
            if (selections.size() == 1)
            {
                  ((Shape)selections.elementAt(0)).edit(this);
                  return;
            }
            else if (selections.size() > 1)
                  MessageDialog.open(
            "Warning : Cannot edit multiple selections at once!");
            return;
      }
}
```

```
public void mousePressed(MouseEvent evt)
{
      . . . . . .
      // Selection with dragging, position setting and color edition
      {
            posX = x; posY = y;
            int i = target.shapes.size()-1;
            while (i >= 0)
            {
                  comp = (Shape)target.shapes.elementAt(i);
                  if (comp.includes(evt.x,evt.y))
                  {
                        . . . . . .
                        posX = x; posY = y;
                        // If double-clicked, edit the selection
                        if (evt.getClickCount() == 2)
                        {
                              resetShapeMouseDrag();
                              if (selections.size() == 1)
                              {
                                    ((Shape)selections.elementAt(0)).
                                          edit(this);
                                    repaint(); return;
                              }
                              else if (selections.size() > 1)
                              {
                                    MessageDialog.open(
            "Warning : Cannot edit multiple selections at once!");
                                    return;
                              }
                        }
                        repaint(); return;
                  }
                  i--;
            }
            clearSelections();
            repaint();
      }
}

}     // End DrawingEditor

class Shape implements Constants
{
. . . . . .
public void edit(DrawingEditor ed)
{
      ColorEditor.open("Edit fill and border colors",this,ed);
}

}     // End Shape
```

2.2.6 Object I/O

Input/output from/to a file may use either a textual or a binary structure that contains the information in "compressed" form using a data compression method. The selection or design of the structure used for I/O may reveal crucial for the performance of the I/O operation as well as for the reliability of the data.

Our example editor supports output in textual form, whereas during input the text is read into the application and parsed in the inverse way, that is from text to the data that the graphical editor can represent and manipulate. Text is output in the following form :

```
450 450 255 200 255
      // Dimension (width and height) of the drawing and background
      // color as an (R,G,B) triplet, with numeric intensities
      // for the red, green and blue component of the color.
// Foreach object are given its type (1 for line, 2 for circle,
// 3 for polygon), its number of vertices (corners)
// its fill and border colors as (R,G,B) triplets as before,
// finally the coordinates (x,y) of its vertices, i.e. the
// content of its "points" variable.
2 2 0 255 255 0 0 0 226 154 234 163
2 2 255 175 175 0 0 0 206 168 215 177
3 17 255 175 175 0 0 0 214 177 177 153 205 185 219 200 192 224 205 224
       185 253 214 222 223 221 229 257 230 219 239 214 228 197 224 186
       236 186 222 173 221 175
3 14 0 255 255 0 0 0 178 155 211 163 235 165 242 167 241 192 258 255 233
       207 218 214 222 251 206 214 230 194 223 176 195 162 194 162
3 8 0 255 255 0 0 0 218 151 243 157 236 156 237 150 228 148 226 153 221
       152 221 152
```

During input the text is read and parsed as follows :

- Firstly, the dimension and color of the drawing is read.
- Subsequently and until the end of file, the individual objects (shapes) are read in. Foreach object, its type, number of vertices, fill and border colors and point coordinates are read in. Coordinates are read as pairs of numbers (x,y) and the number of pairs must match the number of vertices parsed just before.

To parse the input data, we use the java.util.StringTokenizer class that automatically splits a text to "words", subsequently translated to program (shapes) data. The I/O functions are connected to appropriate buttons or key actions when the application is run in standalone mode rather than as an applet (for details see the complete code given along with this tutorial).

The above are implemented as follows* :

```
public class Drawing implements Constants
{
. . . . . .
public void parse(String str)
{
```

```
        String token=null;
        int r=255,g=255,b=255;

        StringTokenizer description = new StringTokenizer(str);

        token = nextToken(description);
        if (token == null) return;
        size.width=parseWidth(token);

        token = nextToken(description);
        if (token == null) return;
        size.height=parseHeight(token);

        token = nextToken(description);
        if (token == null) return;
        r=parseRGB(token);

        token = nextToken(description);
        if (token == null) return;
        g=parseRGB(token);

        token = nextToken(description);
        if (token == null) return;
        b=parseRGB(token);

        backgroundColor = new Color(r,g,b);

        while (description.hasMoreTokens())
        {
              token = nextToken(description);
              int typ = Shape.parseShapeType(token);
              if (Shape.validType(typ))
              {
                    Shape s = new Shape();
                    s.setType(typ);
                    s.parseWithoutType(description);
                    if (validate(s)) shapes.addElement(s);
              }
        }
}

private boolean validate(Shape s)
{
        if (s == null) return false;
        return (s.size >= 2);
}

private String nextToken(StringTokenizer tokenizer)
{
        if (tokenizer.hasMoreTokens()) return tokenizer.nextToken();
        return null;
}

private int parseWidth(String str)
{
        int n = Integer.parseInt(str);
```

```
      if ((n>0) && (n<=MaximumSize.width)) return n;
      return DefaultSize.width;
}

private int parseHeight(String str)
{
      int n = Integer.parseInt(str);
      if ((n>0) && (n<=MaximumSize.height)) return n;
      return DefaultSize.height;
}

private int parseRGB(String str)
{
      int n = Integer.parseInt(str);
      if ((n>=0) && (n<=255)) return n;
      return 255;
}

public String toString()
{
      String str="";

      str += (size.width+" "+size.height+" "+
                  backgroundColor.getRed()+" "+
                  backgroundColor.getGreen()+" "+
                  backgroundColor.getBlue()+"\n");
      for (int i=0;i<shapes.size();i++)
            str+=(((Shape)shapes.elementAt(i)).toString()+"\n");
      return str;
}

}     // End Drawing

class Shape implements Constants
{
. . . . . .
public void parse(StringTokenizer tokenizer)
{
      String token;
      int r=255,g=255,b=255;

      // Parse type, first.
      token = nextToken(tokenizer);
      if (token == null) return;
      type=parseType(token);

      // Then parse size, fill color and border color from tokenizer
      parseWithoutType(tokenizer);
}

public void parseWithoutType(StringTokenizer tokenizer)
{
      String token;
      int r=255,g=255,b=255;

      // Parse size, fill color and border color from tokenizer
```

```
        token = nextToken(tokenizer);
        if (token == null) return;
        size=parseSize(token);

        token = nextToken(tokenizer);
        if (token == null) return;
        r=parseRGB(token);

        token = nextToken(tokenizer);
        if (token == null) return;
        g=parseRGB(token);

        token = nextToken(tokenizer);
        if (token == null) return;
        b=parseRGB(token);

        fillColor = new Color(r,g,b);

        token = nextToken(tokenizer);
        if (token == null) return;
        r=parseRGB(token);

        token = nextToken(tokenizer);
        if (token == null) return;
        g=parseRGB(token);

        token = nextToken(tokenizer);
        if (token == null) return;
        b=parseRGB(token);

        borderColor = new Color(r,g,b);

        // Parse xpoints and ypoints from tokenizer
        for (int i=0;i<size;i++)
        {
              int x, y;
              token = nextToken(tokenizer);
              if (token == null) {size = 0; return;}
              x = parsePosition(token);

              token = nextToken(tokenizer);
              if (token == null) {size = 0; return;}
              y = parsePosition(token);

              if (i < SizeLimit)
              {
                    xpoints[i] = x;
                    ypoints[i] = y;
              }
        }
}

protected static boolean validType(int typ)
{
        switch (typ)
        {
```

```
            case LINE:
            case CIRCLE:
            case POLYGON:
                  return true;
            default:
                  return false;
      }
}

protected String nextToken(StringTokenizer tokenizer)
{
      if (tokenizer.hasMoreTokens()) return tokenizer.nextToken();
      return null;
}

public static int parsePosition(String str)
{
      int n = Integer.parseInt(str);
      if (n>0) return n;
      return 1;
}

public static int parseRGB(String str)
{
      int n = Integer.parseInt(str);
      if ((n>=0) && (n<=255)) return n;
      return 255;
}

protected int parseType(String str)
{
      int n = Integer.parseInt(str);
      switch (n)
      {
            case LINE:
            case CIRCLE:
            case POLYGON:
                  return n;
            default:
                  return CIRCLE;
      }
}

public static int parseShapeType(String str)
{
      int n = Integer.parseInt(str);
      return n;
}

protected int parseSize(String str)
{
      int n = Integer.parseInt(str);
      if ((n>0) && (n<=SizeLimit)) return n;
      return 2;
}
```

```
public String toString()
{
      String str = new String(type + " " + size);
      str += (" "+fillColor.getRed()+" "+
            fillColor.getGreen()+" "+
            fillColor.getBlue());
      str += (" "+borderColor.getRed()+" "+
            borderColor.getGreen()+" "+
            borderColor.getBlue());
      for (int i=0;i<size;i++)
            str += (" "+xpoints[i]+" "+ypoints[i]);
      return str;
}

}     // End Shape
```

2.2.7 Operation undo

Cancelation of operations necessitates recording. Recording may be done in various ways that belong to one of two categories :

- Recording of operations performed. These will be reversed during undoing. For example, a translation by (x,y) will be reversed by translating by (-x,-y).
- Recording directly the equivalent undo operations. For example, a translation by (MOVE,x,y) will trigger the recording of (MOVE,xposition,yposition), where xposition and yposition are the coordinates of the initial object position before moving by (x,y).

The latter method is simpler and more reliable for complex operations whose reversal may be equally or more complex, for example in an operation of distribution of positions along a line, the reversal is not uniquely defined. Our example editor supports operation undo according to the second recording method. For operations refering to multiple objects at once, such as translation of multiple objects, two signals of begin and end of recording are used : START_BLOCK and END_BLOCK. In this way, all undo operations between the two signals are executed in order as a block.

The last operation performed is undoed by pressing Control-z or Control-Z [†].

The above are implemented as follows[*] :

```
public class DrawingEditor extends Applet
      implements Constants, MouseListener, MouseMotionListener,
            KeyListener, ActionListener
{
. . . . . .
// Operations
Vector operations = new Vector(10,5);

//    Two examples of recording of undo actions  are given
//    in keyPressed() and mouseReleased().
//    The rest of the undo actions are recorded in a similar manner.
```

```
public void keyPressed(KeyEvent evt)
{
      . . . . . .
      boolean single_selection = (selections.size() == 1);

      if (key == KeyEvent.VK_LEFT)
      {
            if ( (!shift) && (!control) && (!alt) )
            {     // Move by 1
                  if (!single_selection) record(null,END_BLOCK);
                  for (int i=0;i<selections.size();i++)
                  {
                        comp = (Shape)selections.elementAt(i);
                        record(comp,MOVE,
                              comp.getPosition().x,
                              comp.getPosition().y);
                        comp.moveBy(-1,0);
                  }
                  if (!single_selection) record(null,START_BLOCK);
                  repaint(); return;
            }
      . . . . . .
      }

      if (((ch == 'z') || (ch = 'Z')) && onlyControl)
      {
            undo();
            repaint(); return;
      }
}

public void mouseReleased(MouseEvent evt)
{
      int x = evt.getX(), y = evt.getY();
      Shape comp;

      if (!shapeMouseDrag) return;
      if (shapeMouseDrag)
      {
            resetShapeMouseDrag();
            if (!target.includes(x,y)) return;
            Shape sel = (Shape)selections.elementAt(0);
            if (sel.hasBeenDragged())
            {
                  if (selections.size()>1) record(null,END_BLOCK);
                  for (int i=0;i<selections.size();i++)
                  {
                        comp = (Shape)selections.elementAt(i);
                        record(comp,MOVE,
                              comp.getPosition().x,
                              comp.getPosition().y);
                        comp.dragBy(x-prev_x,y-prev_y);
                        comp.confirmDragged(target);
                  }
                  if (selections.size()>1) record(null,START_BLOCK);
            }
```

```
            prev_x = -1; prev_y = -1;
      }
      repaint();
}

public void undo()
{
//    The operations vector is read from end to start, so that more
//    recent operations are cancelled first.

      int index;
      UndoOperation current;

      if (operations.size() == 0) return;
      clearSelections();
      index = operations.size()-1;
      current = (UndoOperation)operations.elementAt(index);
      if (current.operator == CLEAR_ALL) clear();
      else if (current.operator == START_BLOCK)
      {
            operations.removeElement(current);
            index--;
            current = (UndoOperation)operations.elementAt(index);
            while (current.operator != END_BLOCK)
            {
                  current.target.undo(current.operator,
                        current.operands[0],current.operands[1],
                        current.operands[2],this);
                  operations.removeElement(current);
                  index--;
                  current = (UndoOperation)operations.elementAt(index);
            }
            // Now, current.operator == END_BLOCK,
            //so remove the UndoOperation.
            operations.removeElement(current);
      }
      else
      {
            current.target.undo(current.operator, current.operands[0],
                  current.operands[1],current.operands[2],this);
            operations.removeElement(current);
      }
}

public void record(UndoEntity target, int operator)
{
      operations.addElement(new UndoOperation(target,operator));
}

public void record(UndoEntity target, int operator, int op1)
{
      operations.addElement(new UndoOperation(target,operator,op1));
}

public void record(UndoEntity target, int operator, int op1, int op2)
{
```

```
      operations.addElement(new UndoOperation(target,operator,
                                                op1,op2));
}

public void record(UndoEntity target, int operator, int op1, int op2,
int op3)
{
      operations.addElement(new UndoOperation(target,operator,
                                                op1,op2,op3));
}

}     // End DrawingEditor

class Shape implements Constants, UndoEntity
{
. . . . . .
public void undo(int operator, int op1, int op2, int op3,
                        DrawingEditor editor)
{
      editor.select(this);
      switch (operator)
      {
            case ADD:
                  this.setPosition(op1,op2);
                  editor.target.add(this);
                  break;
            case DELETE:
                  editor.target.remove(this);
                  break;
            case MOVE:
                  setPosition(op1,op2);
                  break;
            case SET_FILL_COLOR:
                  setFillColor(new Color(op1,op2,op3));
                  break;
            case SET_BORDER_COLOR:
                  setBorderColor(new Color(op1,op2,op3));
                  break;
      }
}

}     // End Shape

class UndoOperation
{
      public UndoEntity target;
      public int operator;
      public int operands[] = new int[5];

      UndoOperation(UndoEntity elem, int op)
      {
            target = elem;
            operator=op;
      }

      UndoOperation(UndoEntity elem, int op, int op1)
```

```
        {
                target = elem;
                operator=op;
                operands[0]=op1;
        }

        UndoOperation(UndoEntity elem, int op, int op1, int op2)
        {
                target = elem;
                operator=op;
                operands[0]=op1;
                operands[1]=op2;
        }

        UndoOperation(UndoEntity elem, int op, int op1, int op2, int op3)
        {
                target = elem;
                operator=op;
                operands[0]=op1;
                operands[1]=op2;
                operands[2]=op3;
        }
}

interface UndoEntity
{
public void undo(int operand, int op1, int op2, int op3, DrawingEditor
editor);
}

interface Constants
{
. . . . . .
//    Undo constants
int ADD = 600;
int DELETE = 601;
int MOVE = 602;
int START_BLOCK = 606;
int END_BLOCK = 607;
int SET_FILL_COLOR = 610;
int SET_BORDER_COLOR = 611;
. . . . . .
}     // End Constants
```

2.2.8 Operation redo

Redoing of operations necessitates recording of the last operation performed. Unlike undoing, redo is possible through direct recording of the operation performed, for example a translation of an object by (dx,dy) is recorded as (MOVE_BY,dx,dy). Obviously recording for redo does not need the START_BLOCK and END_BLOCK markers. Also each operation performed through redo is recorded for undoing as usually.

The last operation performed is redoed by pressing Control-y or Control-Y [†].

The above are implemented as follows* :

```
public class DrawingEditor extends Applet
      implements Constants, MouseListener, MouseMotionListener,
            KeyListener, ActionListener
{
. . . . . .
// Redo data
int redo_operator = 0;
int redo_operands_size = 6;
int redo_operands[] = new int[redo_operands_size];

//    Ενα παράδειγμα καταγραφής ενέργειας επανάληψης δίνεται στην
//    keyPressed(). Οι άλλες ενέργειες καταγράφονται αντίστοιχα.

public void keyPressed(KeyEvent evt)
{
      . . . . . .
      boolean single_selection = (selections.size() == 1);

      if (key == KeyEvent.VK_LEFT)
      {
            if ( (!shift) && (!control) && (!alt) )
            {     // Move by 1
                  . . . . . .
                  if (selections.size() > 0) set_redo(MOVE_BY,-1,0);
                  repaint(); return;
            }
            . . . . . .
      }

      if (((ch == 'y') || (ch = 'Y')) && onlyControl)
      {
            redo();
            repaint(); return;
      }
}

public void redo()
{
      if (selections.size() == 0) return;
      switch (redo_operator)
      {
            case -1 :
                  break;
            default:
                  if (selections.size() > 1) record(null,END_BLOCK);
                  for (int i=0;i<selections.size();i++)
                        ((Shape)selections.elementAt(i)).redo(
                              redo_operator,redo_operands,this);
                  if (selections.size() > 1) record(null,START_BLOCK);
      }
}

public void set_redo(int operator)
```

```
{
      redo_operator = operator;
      for (int i=0;i<redo_operands_size;i++) redo_operands[i]=0;
}

public void set_redo(int operator, int op1)
{
      redo_operator = operator;
      for (int i=0;i<redo_operands_size;i++) redo_operands[i]=0;
      redo_operands[0]=op1;
}

public void set_redo(int operator, int op1, int op2)
{
      redo_operator = operator;
      for (int i=0;i<redo_operands_size;i++) redo_operands[i]=0;
      redo_operands[0]=op1;
      redo_operands[1]=op2;
}

public void set_redo(int operator, int op1, int op2, int op3)
{
      redo_operator = operator;
      for (int i=0;i<redo_operands_size;i++) redo_operands[i]=0;
      redo_operands[0]=op1;
      redo_operands[1]=op2;
      redo_operands[2]=op3;
}

public void set_redo(int operator, int op1, int op2, int op3,
                              int op4, int op5, int op6)
{
      redo_operator = operator;
      for (int i=0;i<redo_operands_size;i++) redo_operands[i]=0;
      redo_operands[0]=op1;
      redo_operands[1]=op2;
      redo_operands[2]=op3;
      redo_operands[3]=op4;
      redo_operands[4]=op5;
      redo_operands[5]=op6;
}

}     // End DrawingEditor


class Shape implements Constants, UndoEntity
{
. . . . . .
public void redo(int operator, int[] operands, DrawingEditor editor)
{
      Color c;
      switch (operator)
      {
            case MOVE_BY:
                  editor.record(this,MOVE,
                                    getPosition().x,getPosition().y);
```

```
                    moveBy(operands[0],operands[1]);
                    break;
              case DELETE:
                    editor.record(this,ADD);
                    editor.target.remove(this);
                    break;
        }
}

}     // End Shape


interface Constants
{
. . . . . .
//    Redo constants (some undo constants are also used)
int MOVE_BY = 650;

}     // End Constants
```

2.2.9 Rubberbanding

This is a multiple selection operation through drawing a rectangle (rubberband) around the desired objects. Usually this is done by first clicking on the free space in the topleft corner of the desired rubberband area and dragging the mouse until the corresponding bottomright corner. The rectangle defined in this way is drawn in its final as well as in its intermediate forms with a dashed line of a caracteristic color (blue, in our case, see figure 2.5). When the mouse button is subsequently released, the objects included in the rubberband in their entirety are selected and the rubberband vanishes.

The above are implemented as follows* :

```
public class DrawingEditor extends Applet
      implements Constants, MouseListener, MouseMotionListener,
            KeyListener, ActionListener
{
//    Variables used for rubberbanding : flag and start position
boolean rubberbandMouseDrag = false;
Point rubberbandStart = new Point(-1,-1);
Point rubberbandEnd = new Point(-1,-1);

public void mousePressed(MouseEvent evt)
{
      . . . . . .
      // Selection with dragging, edition of shapes etc.
      {
            . . . . . .
            while (i >= 0)
            {
                  . . . . . .
            }
            clearSelections();
```

```
                setRubberbandMouseDrag();
                rubberbandStart.x = rubberbandEnd.x = x;
                rubberbandStart.y = rubberbandEnd.y = y;
                repaint();
        }
}
```

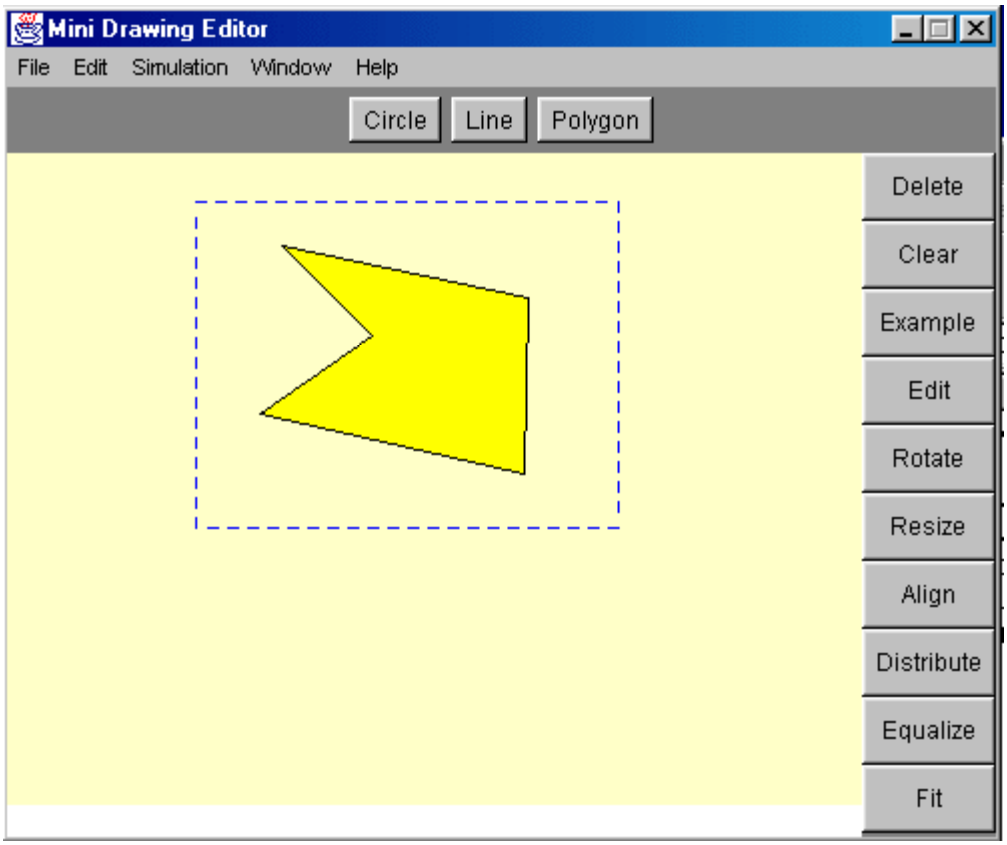

Figure 2.5. Rubberbanding.

```
public void mouseReleased(MouseEvent evt)
{
        . . . . . .
        if ((!shapeMouseDrag) && (!rubberbandMouseDrag)) return;
        . . . . . .
        if (rubberbandMouseDrag)        // rubberbandMouseDrag
        {
                clearSelections();
                for (int i=0;i<target.shapes.size();i++)
                {
                        comp = (Shape)target.shapes.elementAt(i);
                        if (inRubberband(comp.topleft())
                                        && inRubberband(comp.bottomright()))
                                comp.doSelect(this);
                }
                resetRubberbandMouseDrag();
                rubberbandStart.x = rubberbandStart.y
                        = rubberbandEnd.x = rubberbandEnd.y = -1;
```

```
        }
        repaint();
}

public void mouseDragged(MouseEvent evt)
{
        . . . . . .
        if ((!shapeMouseDrag) && (!rubberbandMouseDrag)) return;
        . . . . . .
        if (rubberbandMouseDrag)      // rubberbandMouseDrag
        {
              rubberbandEnd.x = x;
              rubberbandEnd.y = y;
        }
        repaint();
}

public void paint(Graphics g)
{
        . . . . . .
        if (rubberbandMouseDrag)
              drawRubberband(g,rubberbandStart,rubberbandEnd);
}

public void drawRubberband(Graphics g, Point start, Point end)
{
        Color pen = g.getColor();
        int   originx = Math.min(start.x, end.x),
              originy = Math.min(start.y, end.y),
              width = Math.abs(start.x - end.x),
              height = Math.abs(start.y - end.y),
              endx = Math.max(start.x, end.x),
              endy = Math.max(start.y, end.y);

        g.setColor(Color.blue);

//      Draw a dashed rectangle
        int i, j, step = 5;
        boolean draw = true;

        for (i=originx;i<=(endx-step);i+=step)
        {
              if (draw) g.drawLine(i, originy, i+step, originy);
              draw = (!draw);
        }
        draw = true;
        for (j=originy;j<=(endy-step);j+=step)
        {
              if (draw) g.drawLine(endx, j, endx, j+step);
              draw = (!draw);
        }
        draw = true;
        for (i=endx;i>=(originx+step);i-=step)
        {
              if (draw) g.drawLine(i, endy, i-step, endy);
              draw = (!draw);
```

```
        }
        draw = true;
        for (j=endy;j>=(originy+step);j-=step)
        {
                if (draw) g.drawLine(originx, j, originx, j-step);
                draw = (!draw);
        }

        g.setColor(pen);
}

public boolean inRubberband(Point po)
{
        int   leftmost = Math.min(rubberbandStart.x, rubberbandEnd.x),
              topmost = Math.min(rubberbandStart.y, rubberbandEnd.y),
              rightmost = Math.max(rubberbandStart.x, rubberbandEnd.x),
              bottommost = Math.max(rubberbandStart.y, rubberbandEnd.y);

        return ((po.x >= leftmost) && (po.x <= rightmost)
                    && (po.y >= topmost) && (po.y <= bottommost));
}

private void setRubberbandMouseDrag() {rubberbandMouseDrag = true;}
private void resetRubberbandMouseDrag() {rubberbandMouseDrag = false;}

}     // End DrawingEditor
```

2.3 Advanced operations

2.3.1 Additional geometric operations

Just like any other graphical editor, our example editor defines many special purpose operations, mostly geometrical manipulations :

- Rotation with Control+arrow (clockwise or counterclockwise according to the arrow) or with the “Turn” button. Rotation of a polygon is performed around its mass or geometric center (i.e. the point having as coordinates the averages of the x- and y-coordinates of the vertices). This operation is implemented as follows* :

```
public class DrawingEditor extends Applet
        implements Constants, MouseListener, MouseMotionListener,
                KeyListener, ActionListener, UndoEntity
{
. . . . . .
Button rotateButton;

public void actionPerformed(ActionEvent evt)
{
        . . . . . .
        else if (tgt == rotateButton)
        {
                rotateSelections();
```

```
            repaint();
      }
}

public void keyPressed(KeyEvent evt)
{
      . . . . . .
      // Move and rotate with the arrows
      if (key == KeyEvent.VK_LEFT)
      {
            if ( (!shift) && (!control) && (!alt) )
            {     // Move by 1
                  . . . . . .
            }
            if ( (!shift) && (!control) && (alt) )
            {     // Move by 10
                  . . . . . .
            }
            if ( (!shift) && control && (!alt) )
            {     // Rotate
                  for (int i=0;i<selections.size();i++)
                  {
                        comp = (Shape)selections.elementAt(i);
                        comp.rotateBy(-1,0);
                  }
                  repaint(); return;
            }
      }
      // Similar things for the rest of the keys (RIGHT, DOWN and UP)
      . . . . . .
}

public void rotateSelections()
{
      for (int i=0;i<selections.size();i++)
            ((Shape)selections.elementAt(i)).rotate(this);
      repaint();
}

}     // End DrawingEditor

class Shape implements Constants, UndoEntity
{
. . . . . .

public void rotate(DrawingEditor editor)
{
      rotateBy(1,0);
}

protected boolean rotateClockwise(int x, int y)
{
      // Clockwise : (+1,0) -- right -- or (0,+1) --down --
      // Counterclockwise : (-1,0) -- left -- or (0,-1) -- up --
      return (x+y > 0);
}
```

```
public void rotateBy(int x, int y)
{
      switch (type)
      {
            case LINE:
                  rotateLineBy(x,y);
                  break;
            case CIRCLE:
                  // No in situ rotation necessary !
                  break;
            case POLYGON:
                  rotatePolygonBy(x,y);
                  break;
      }
}

public void rotateLineBy(int x, int y)
{
      int i, j, dx, dy, centerx, centery;

//    Compute centerx and centery
      centerx = (xpoints[0]+xpoints[1])/2;
      centery = (ypoints[0]+ypoints[1])/2;

//
//    Exchange all points x and y relative to centerx and centery
//
      if (rotateClockwise(x,y))
      {
            for (i=0;i<size;i++)
            {
                  dx = xpoints[i]-centerx;
                  dy = ypoints[i]-centery;
                  xpoints[i] = centerx - dy;
                  ypoints[i] = centery + dx;
            }
      }
      else
      {
            for (i=0;i<size;i++)
            {
                  dx = xpoints[i]-centerx;
                  dy = ypoints[i]-centery;
                  xpoints[i] = centerx + dy;
                  ypoints[i] = centery - dx;
            }
      }
}

public void rotatePolygonBy(int x, int y)
{
      int i, j, dx, dy, centerx, centery;

//    Compute center
      Point c = center();
```

```
        centerx = c.x;
        centery = c.y;

//
//    Exchange all points x and y relative to centerx and centery
//
//                (as above)
}

private Point center()
{
        Point p = new Point(0,0);
        for (int i=0;i<size;i++)
        {
                p.x += xpoints[i];
                p.y += ypoints[i];
        }
        double s = (double)size;
        p.x = (int)Math.round(p.x/s);
        p.y = (int)Math.round(p.y/s);
        return p;
}

public void rotateAround(int cx, int cy, boolean right)
{
        int i;
        if (right)
        {
                for (i=0;i<size;i++)
                {
                        int x = xpoints[i], y = ypoints[i];
                        xpoints[i] = cx + cy - y;
                        ypoints[i] = cy - cx + x;
                }
        }
        else // left
        {
                for (i=0;i<size;i++)
                {
                        int x = xpoints[i], y = ypoints[i];
                        xpoints[i] = cx - cy + y;
                        ypoints[i] = cy + cx - x;
                }
        }
        if (type == CIRCLE) validateCircle();
}

public void validateCircle()
{
        int temp;
        if (xpoints[0] > xpoints[1])
        {       // Swap xpoints
                temp = xpoints[0];
                xpoints[0] = xpoints[1];
                xpoints[1] = temp;
        }
```

```
      if (ypoints[0] > ypoints[1])
      {     // Swap ypoints
            temp = ypoints[0];
            ypoints[0] = ypoints[1];
            ypoints[1] = temp;
      }
}

}     // End Shape
```

- Object resizing with the arrows (increase by 1 pixel with Shift+arrow, decrease by 1 pixel with Shift+Control+arrow, increase or decrease by 10 pixels when moreover Alt is pressed). Also absolute resizing with control-r † or with the "Resize" button. The above are implemented as follows* :

```
public class DrawingEditor extends Applet
      implements Constants, MouseListener, MouseMotionListener,
            KeyListener, ActionListener, UndoEntity
{
. . . . . .
Button resizeButton;

public void actionPerformed(ActionEvent evt)
{
      . . . . . .
      else if (tgt == resizeButton)
      {
            reset_input();
            if (selections.size() == 1)
            {
                  ((Shape)selections.elementAt(0)).resize(this);
                  return;
            }
            if (selections.size() > 1)
            {
                  MessageDialog.open(
            "Warning : Cannot resize multiple selections at once!");
                  return;
            }
      }

}

public void keyPressed(KeyEvent evt)
{
      . . . . . .
      // Move, rotate and resize with the arrows
      if (key == KeyEvent.VK_LEFT)
      {
            if ( (!shift) && (!control) && (!alt) )
            {     // Move by 1
                  . . . . . .
            }
```

```
        else if ( (!shift) && (!control) && (alt) )
        {     // Move by 10
              . . . . . .
        }
        else if ( (!shift) && control && (!alt) )
        {     // Rotate
              . . . . . .
        }
        else if ( shift && (!control) && (!alt) )
        {     // Increase size by 1
              for (int i=0;i<selections.size();i++)
              {
                    comp = (Shape)selections.elementAt(i);
                    comp.increaseBy(-1,0,this);
              }
              repaint(); return;
        }
        else if ( shift && control && (!alt) )
        {     // Decrease size by by 1
              for (int i=0;i<selections.size();i++)
              {
                    comp = (Shape)selections.elementAt(i);
                    comp.decreaseBy(-1,0,this);
              }
              repaint(); return;
        }
        else if ( shift && (!control) && alt )
        {     // Increase size by 10
              for (int i=0;i<selections.size();i++)
              {
                    comp = (Shape)selections.elementAt(i);
                    comp.increaseBy(-10,0,this);
              }
              repaint();return;
        }
        else if ( shift && control && alt )
        {     // Decrease size by by 10
              for (int i=0;i<selections.size();i++)
              {
                    comp = (Shape)selections.elementAt(i);
                    comp.decreaseBy(-10,0,this);
              }
              repaint();return;
        }
  }
  // Similar things for the rest of the keys (RIGHT, DOWN and UP)

  if (((ch == 'r') || (ch == 'R')) && onlyControl)
  {     // Resize the selection
        reset_input();
        if (selections.size() == 1)
        {
              ((Shape)selections.elementAt(0)).resize(this);
              return;
        }
        if (selections.size() > 1)
```

```
            {
                  MessageDialog.open(
            "Warning : Cannot resize multiple selections at once!");
                  return;
            }
            repaint(); return;
      }
}

}     // End DrawingEditor

class Shape implements Constants, UndoEntity
{
. . . . . .
public int width()
{
      // Find rightmost and leftmost
      . . . . . .
      return rightmost-leftmost;
}

public int height()
{
      // Find bottommost and topmost
      . . . . . .
      return bottommost-topmost;
}

public Dimension getSize()
{
      return new Dimension(width(),height());
}

public void setSize(int wi, int hei)
{
//    Find leftmost and topmost
      . . . . . .

//    Scale all points x's and y's relative to leftmost and topmost
      for (i=0;i<size;i++)
            xpoints[i] = (int)(leftmost +
                  ((xpoints[i]-leftmost)*((float)wi/(float)oldwi)));
      for (j=0;j<size;j++)
            ypoints[j] = (int)(topmost +
                  ((ypoints[j]-topmost)*((float)hei/(float)oldhei)));
}

public void increaseBy(int x, int y, DrawingEditor editor)
{
      if (type == CIRCLE)
            increaseCircleBy(x,y,editor);
      else
            increaseLineOrPolygonBy(x,y,editor);
}

public void increaseCircleBy(int x, int y, DrawingEditor editor)
```

```
{
      int i, j;

      if (x>0)
      {     // Find rightmost x
            xpoints[1] += x; ypoints[0] -= x;
      }
      else if (x<0)
      {     // Find leftmost x
            xpoints[0] += x; ypoints[1] -= x;
      }
      if (y>0)
      {     // Find bottommost y
            ypoints[1] += y; xpoints[0] -= y;
      }
      else if (y<0)
      {     // Find topmost y
            ypoints[0] += y; xpoints[1] -= y;
      }
}

public void increaseLineOrPolygonBy(int x, int y, DrawingEditor editor)
{
      int chosen_x, chosen_y, chosen_i = -1, chosen_j = -1, i, j;

      if (x>0)
      {
      // Find rightmost x (chosen_x at chosen_i)
            . . . . . .
            xpoints[chosen_i] += x;
      }
      else if (x<0)
      {
      // Find leftmost x (chosen_x at chosen_i)
            . . . . . .
            xpoints[chosen_i] += x;
      }
      if (y>0)
      {
      // Find bottommost y (chosen_y at chosen_j)
            . . . . . .
            ypoints[chosen_j] += y;
      }
      else if (y<0)
      {
      // Find topmost y (chosen_y at chosen_j)
            . . . . . .
            ypoints[chosen_j] += y;
      }
}

public void decreaseBy(int x, int y, DrawingEditor editor)
{
      if (type == CIRCLE)
            decreaseCircleBy(x,y,editor);
      else
```

```
            decreaseLineOrPolygonBy(x,y,editor);
}

public void decreaseCircleBy(int x, int y, DrawingEditor editor)
{
      int i, j;

      if (x>0)
      {     // Find leftmost x
            xpoints[0] += x; ypoints[1] -= x;
      }
      else if (x<0)
      {     // Find rightmost x
            xpoints[1] += x; ypoints[0] -= x;
      }
      if (y>0)
      {     // Find topmost y
            ypoints[0] += y; xpoints[1] -= y;
      }
      else if (y<0)
      {     // Find bottommost y
            ypoints[1] += y; xpoints[0] -= y;
      }
}

public void decreaseLineOrPolygonBy(int x, int y, DrawingEditor editor)
{
      int chosen_x, chosen_y, chosen_i = -1, chosen_j = -1, i, j;

      if (x>0)
      {
      // Find leftmost x (chosen_x at chosen_i)
            . . . . . .
            xpoints[chosen_i] += x;
      }
      else if (x<0)
      {
      // Find rightmost x (chosen_x at chosen_i)
            . . . . . .
            xpoints[chosen_i] += x;
      }
      if (y>0)
      {
      // Find topmost y (chosen_y at chosen_j)
            . . . . . .
            ypoints[chosen_j] += y;
      }
      else if (y<0)
      {
      // Find bottommost y (chosen_y at chosen_j)
            . . . . . .
            ypoints[chosen_j] += y;
      }
}

public void resize(DrawingEditor ed)
```

```
{
      ResizeDialog.open(this,ed);
}

}     // End Shape
```

- Operations “align”, “distribute” and “equalize” with the buttons “Align”, “Distribute” and “Equalize”, respectively.

  During an alignment operation, the user selects a number of shapes and defines the parameters of the alignment in a special-purpose popup dialog. There are three alignment papameters taking each a number of predefined values : alignment direction (horizontal or vertical), alignment position (top, bottom or center for horizontal alignment; left, right or center for vertical alignment) and alignment reference (first selection, last selection, min, max or average across selections).

  During a distribution operation, the user selects a number of shapes and defines the parameters of the distribution in a special-purpose popup dialog. There are three distribution papameters, two of which are taking each a number of predefined values : distribution type (absolute or relative), distribution direction (horizontal or vertical) and distribution spacing (a number of pixels used for absolute distribution).

  During an equalization operation, the user selects a number of shapes and defines the parameters of equalization in a special-purpose popup dialog. There are two equalization papameters taking each a number of predefined values : equalization dimension (width, height or both) and equalization reference (first selection, last selection, min, max or average across selections).

  The above are implemented as follows [*] :

```
public class DrawingEditor extends Applet
      implements Constants, MouseListener, MouseMotionListener,
            KeyListener, ActionListener, UndoEntity
{
. . . . . .
Button alignButton, distributeButton, equalizeButton;

public void actionPerformed(ActionEvent evt)
{
      . . . . . .
      else if (tgt == alignButton)
      {
            reset_input();
            alignSelections();
            repaint();
      }
      else if (tgt == distributeButton)
      {
            reset_input();
            distributeSelections();
            repaint();
      }
```

```
      else if (tgt == equalizeButton)
      {
            reset_input();
            equalizeSelections();
            repaint();
      }
}

public void alignSelections()
{
      if (selections.size() >= 2) AlignDialog.open(this);
}

public void alignSelections(int direction, int reference, int position)
{
//    This method will be called by the AlignDialog on return.
//    Align the given position of all selections (topleft, bottomleft
//    etc.) in the given direction according to the given reference.

      Point pos;
      Shape comp;

      switch (reference)
      {
      // Align according to the first selection, the last selection,
      // the average, the maximum or the minimum of all selections.
            case AlignConstants.ALIGN_FIRST:
                  pos = ((Shape)selections. elementAt(0)).
                              getAlignPosition(position);
                  break;
            case AlignConstants.ALIGN_LAST:
                  pos = ((Shape)selections.elementAt(
                                    selections.size()-1)).
                              getAlignPosition(position);
                  break;
            case AlignConstants.ALIGN_AVG:
                  pos = compute_avg_selections_pos(position);
                  break;
            case AlignConstants.ALIGN_MAX:
                  pos = compute_max_selections_pos(position);
                  break;
            case AlignConstants.ALIGN_MIN:
                  pos = compute_min_selections_pos(position);
                  break;
            default:
                  pos = ((Shape)selections.
                        elementAt(0)).getAlignPosition(position);
                  break;
      }
      for (int i=0;i<selections.size();i++)
      {
            comp = (Shape)selections.elementAt(i);
            comp.align(direction, position, pos);
      }
}
```

```
public void distributeSelections()
{
      if (selections.size() >= 2) DistributeDialog.open(this);
}

public void distributeSelections(int type, int dimension, int spacing)
{
//    This method will be called by the DistributeDialog on return.
//    Distribute all selections in the given dimension
//          using the given type and spacing.

      Point rel;
      Shape el, comp;
      Vector sels;

      switch (type)
      {
            case DistributeConstants.DISTRIBUTE_ABSOLUTE:
                  rel = new Point (spacing,spacing);
                  break;
            case DistributeConstants.DISTRIBUTE_RELATIVE:
                  rel = compute_uniform_indent_selections();
                  break;
            default:
                  rel = new Point (spacing,spacing);
                  break;
      }
      switch (dimension)
      {
            case DistributeConstants.DISTRIBUTE_HORIZONTAL:
                  sels = sortLeftmostFirst(selections);
                  el = (Shape)sels.elementAt(0);
                  for (int i=1;i<sels.size();i++)
                  {
                        comp = (Shape)sels.elementAt(i);
                        comp.setPosition((el.relativePosition(rel)).x,
                                                comp.getPosition().y);
                        el = (Shape)sels.elementAt(i);
                  }
                  break;
            case DistributeConstants.DISTRIBUTE_VERTICAL:
                  sels = sortTopmostFirst(selections);
                  el = (Shape)sels.elementAt(0);
                  for (int i=1;i<sels.size();i++)
                  {
                        comp = (Shape)sels.elementAt(i);
                        comp.setPosition(comp.getPosition().x,
                                          (el.relativePosition(rel)).y);
                        el = (Shape)sels.elementAt(i);
                  }
                  break;
            default:
                  break;
      }
}
```

```
public Vector sortLeftmostFirst(Vector v)
{
      int i;
      Vector vec = new Vector(v.size());
      for (i=0;i<v.size();i++) vec.addElement(v.elementAt(i));

      // Use bubble sort
      for (int start=0;start<vec.size();start++)
      for (i=start+1;i<vec.size();i++)
      {
            Shape s_i = (Shape)(vec.elementAt(i));
            Shape s_start = (Shape)(vec.elementAt(start));
            int pos_i = s_i.getPosition().x;
            int pos_start = s_start.getPosition().x;
            if (pos_i<pos_start)
            {
                  Shape temp = s_start;
                  vec.setElementAt(s_i,start);
                  vec.setElementAt(temp,i);
            }
      }
      return vec;
}

public Vector sortTopmostFirst(Vector v)
{
      // Same as sortLeftmostFirst(), but for minimum y instead of x.
}

public void equalizeSelections()
{
      if (selections.size() >= 2) EqualizeDialog.open(this);
}

public void equalizeSelections(int reference, int dimension)
{
//    This method will be called by the EqualizeDialog on return.
//    Equalize the given dimension of all selections
//          according to the given reference.

      Dimension theSize;
      Shape comp;

      switch (reference)
      {
      // Equalize according to the first selection, the last selection,
      // the average, the maximum or the minimum of all selections.
            case EqualizeConstants.EQUALIZE_FIRST:
                  theSize = ((Shape)selections.elementAt(0)).getSize();
                  break;
            case EqualizeConstants.EQUALIZE_LAST:
                  theSize = ((Shape)selections.
                        elementAt(selections.size()-1)).getSize();
                  break;
            case EqualizeConstants.EQUALIZE_AVG:
                  theSize = compute_avg_selections_size();
```

```
                break;
            case EqualizeConstants.EQUALIZE_MAX:
                theSize = compute_max_selections_size();
                break;
            case EqualizeConstants.EQUALIZE_MIN:
                theSize = compute_min_selections_size();
                break;
            default:
                theSize = ((Shape)selections.elementAt(0)).getSize();
                break;
        }
        for (int i=0;i<selections.size();i++)
        {
            if ( ((reference == EqualizeConstants.EQUALIZE_FIRST)
                        && (i == 0)) ||
                 ((reference == EqualizeConstants.EQUALIZE_LAST)
                        && (i == (selections.size()-1))) )
                 ; // Ignore the reference, if there is one
            else
            {
                comp = (Shape)selections.elementAt(i);
                comp.equalize(dimension, theSize);
            }
        }

        repaint();
    }

    private Point compute_uniform_indent_selections()
    {
        Point min = new Point(10000,10000);
        Point max = new Point(0,0);
        Point total = new Point(0,0);
        int n = selections.size()-1;

        for (int i=0;i<selections.size();i++)
        {
            min.x = Math.min(min.x,
                ((Shape)selections.elementAt(i)).
                getPosition().x);
            min.y = Math.min(min.y,
                ((Shape)selections.elementAt(i)).
                getPosition().y);
            max.x = Math.max(max.x,
                (((Shape)selections.elementAt(i)).
                bottomright()).x);
            max.y = Math.max(max.y,
                (((Shape)selections.elementAt(i)).
                bottomright()).y);
            total.x += ((Shape)selections.elementAt(i)).
                width();
            total.y += ((Shape)selections.elementAt(i)).
                height();
        }
        return new Point( ((max.x-min.x-total.x)/n),
                          ((max.y-min.y-total.y)/n) );
```

```
}

}     // End DrawingEditor

class Shape implements Constants, UndoEntity
{
. . . . . .

public Point getPosition()
{
//    Find leftmost and topmost
      . . . . . .
      return new Point(leftmost,topmost);
}

public Point relativePosition(Point rel)
{
      Point pos = getPosition();
      return new Point(pos.x+width()+rel.x,pos.y+height()+rel.y);
}

public Point getAlignPosition(int pos)
{
      switch (pos)
      {
            case AlignConstants.ALIGN_TOP:
                  return getPosition();
            case AlignConstants.ALIGN_BOTTOM:
                  return bottomright();
            case AlignConstants.ALIGN_CENTER:
                  return new Point((getPosition().x+bottomright().x)/2,
                        (getPosition().y+bottomright().y)/2);
            case AlignConstants.ALIGN_LEFT:
                  return getPosition();
            case AlignConstants.ALIGN_RIGHT:
                  return topright();
            default:
                  return getPosition();
      }
}

public void align(int type, int pos, Point po)
{
      Point oldPosition = getPosition();

      switch (type)
      {
            case AlignConstants.ALIGN_HORIZONTAL: // Align y coordinates
                  switch (pos)
                  {
                        case AlignConstants.ALIGN_TOP:
                              setPosition(oldPosition.x,po.y);
                              break;
                        case AlignConstants.ALIGN_BOTTOM:
                              setPosition(oldPosition.x,po.y-height());
                              break;
```

```
                        case AlignConstants.ALIGN_CENTER:
                              setPosition(oldPosition.x,
                                    po.y-(height()/2));
                              break;
                  }
                  break;
            case AlignConstants.ALIGN_VERTICAL: // Align x coordinates
                  switch (pos)
                  {
                        case AlignConstants.ALIGN_LEFT:
                              setPosition(po.x,oldPosition.y);
                              break;
                        case AlignConstants.ALIGN_RIGHT:
                              setPosition(po.x-width(),oldPosition.y);
                              break;
                        case AlignConstants.ALIGN_CENTER:
                              setPosition(po.x-(width()/2),
                                    oldPosition.y);
                              break;
                  }
                  break;
      }
}

public void equalize(int dim, Dimension po)
{
      if (type == CIRCLE)
      {
            int new_dim = 0, old_dim = width();
            if (dim == EqualizeConstants.EQUALIZE_WIDTH)
                        new_dim = po.width;
            else if (dim == EqualizeConstants.EQUALIZE_HEIGHT)
                        new_dim = po.height;
            else if (dim == EqualizeConstants.EQUALIZE_BOTH)
                        new_dim = Math.min(po.width,po.height);
            xpoints[1] = xpoints[0] + (xpoints[1]-xpoints[0])
                                    *new_dim/old_dim;
            ypoints[1] = ypoints[0] + (ypoints[1]-ypoints[0])
                                    *new_dim/old_dim;
      }
      else switch (dim)
      {
            case EqualizeConstants.EQUALIZE_WIDTH:
                  while (Math.abs(width()-po.width) > 1)
                  {
                        // Find rightmost and leftmost
                        . . . . . .
                        xpoints[rightmost_index]=leftmost+po.width;
                  }
                  break;
            case EqualizeConstants.EQUALIZE_HEIGHT:
                  while (Math.abs(height()-po.height) > 1)
                  {
                        // Find topmost and bottommost
                        . . . . . .
                        ypoints[bottommost_index]=topmost+po.height;
```

```
                }
                break;
            case EqualizeConstants.EQUALIZE_BOTH:
                while (Math.abs(width()-po.width) > 1)
                {
                    // Find rightmost and leftmost
                    . . . . . .
                    xpoints[rightmost_index]=leftmost+po.width;
                }
                while (Math.abs(height()-po.height) > 1)
                {
                    // Find topmost and bottommost
                    . . . . . .
                    ypoints[bottommost_index]=topmost+po.height;
                }
                break;
        }
}

}     // End Shape

interface AlignConstants
{
//    Alignment constants
final static int ALIGN_HORIZONTAL = 256;
final static int ALIGN_VERTICAL = 257;
final static int ALIGN_TOP = 270;
final static int ALIGN_CENTER = 271;
final static int ALIGN_BOTTOM = 272;
final static int ALIGN_LEFT = 273;
final static int ALIGN_RIGHT = 274;
final static int ALIGN_FIRST = 280;
final static int ALIGN_LAST = 281;
final static int ALIGN_AVG = 282;
final static int ALIGN_MIN = 283;
final static int ALIGN_MAX = 284;
}     // End AlignConstants

interface DistributeConstants
{
//    Distribute constants
final static int DISTRIBUTE_HORIZONTAL = 420;
final static int DISTRIBUTE_VERTICAL = 421;
final static int DISTRIBUTE_RELATIVE = 430;
final static int DISTRIBUTE_ABSOLUTE = 431;
}     // End DistributeConstants

interface EqualizeConstants
{
//    Equalize constants
final static int EQUALIZE_WIDTH = 340;
final static int EQUALIZE_HEIGHT = 341;
final static int EQUALIZE_BOTH = 342;
final static int EQUALIZE_FIRST = 350;
final static int EQUALIZE_LAST = 351;
final static int EQUALIZE_AVG = 352;
```

```
final static int EQUALIZE_MIN = 353;
final static int EQUALIZE_MAX = 354;
}     // End EqualizeConstants
```

- "Fitting" operation with the button "Fit" : translating an object so that one of its corners coincides with a different corner of another object. If both corners belong to the same shape, then it is not translated but deformed accordingly. The two corners are selected with Alt+click and are visualized during selection as red crosses (see figure 2.6).

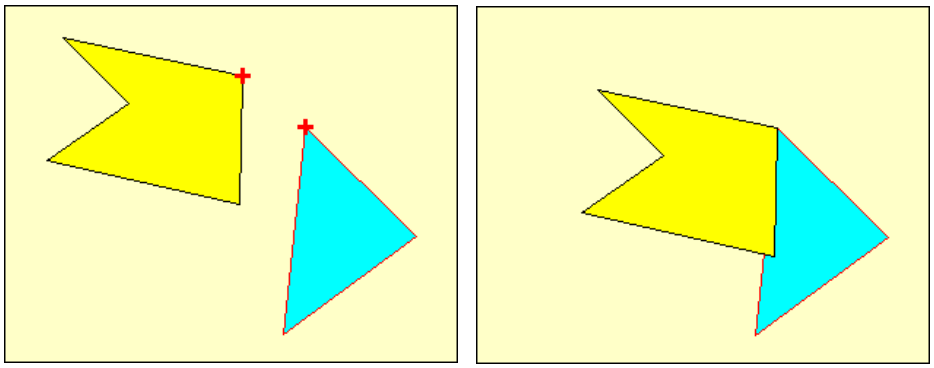

Figure 2.6. Polygon fitting.

This operation is implemented as follows* :

```
public class DrawingEditor extends Applet
      implements Constants, MouseListener, MouseMotionListener,
            KeyListener, ActionListener, UndoEntity
{
. . . . . .
Button fitButton;

//    Variables used for fitting shapes to one another
Shape fitShapes[] = {null,null};
int fitVertices[] = {-1,-1};

public void mousePressed(MouseEvent evt)
{
      . . . . . .
//    Select vertices to fit.
      if ((!shift) && (!control) && alt)
      {
            int i=0, j, vtx;
            while (i<target.shapes.size())
            {
                  comp = (Shape)target.shapes.elementAt(i);
                  if ((vtx=comp.indexOfVertexAt(x,y)) != -1)
                  {
                        if (fitShapes[0] == null) j=0; else j=1;
                        fitShapes[j]=comp;
                        fitVertices[j]=vtx;
```

```
                        repaint(); return;
                  }
                  i++;
            }
      }
      clearFitData();
}

public void actionPerformed(ActionEvent evt)
{
      . . . . . .
      else if (tgt == fitButton)
      {
            reset_input();
            fitSelections();
            repaint();
      }

}

void fitSelections()
{
      if ((fitShapes[0]!=null) && (fitShapes[1]!=null))
      {
            Point pos = fitShapes[1].vertexAt(fitVertices[1]);
            if (fitShapes[0] == fitShapes[1])
                  fitShapes[0].setVertexAt(
                              fitVertices[0],pos.x,pos.y,this);
            else
                  fitShapes[0].fitVertexAt(
                              fitVertices[0],pos.x,pos.y,this);
            clearFitData();
      }
}

public void clearFitData()
{
      for (int i=0;i<2;i++) {fitShapes[i]=null; fitVertices[i]=-1;}
}

public void paint(Graphics g)
{
      . . . . . .
      for (int i=0;i<2;i++)
            if (fitShapes[i]!=null)
                  drawFitPoint(g,fitShapes[i].vertexAt(fitVertices[i]));
}

void drawFitPoint(Graphics g, Point po)
{     // Draw a red cross
      Color pen = g.getColor();
      g.setColor(Color.red);
      g.fillRect(po.x-1,po.y-5,3,11);
      g.fillRect(po.x-5,po.y-1,11,3);
      g.setColor(pen);
}
```

```
}     // End DrawingEditor

class Shape implements Constants, UndoEntity
{
. . . . . .

public int indexOfVertexAt(int x, int y)
{
      for (int i=0;i<size;i++)
      {
            if ((Math.abs(xpoints[i]-x) <= 5)
                  && (Math.abs(ypoints[i]-y) <= 5))
                  return i;
      }
      return -1;
}

public Point vertexAt(int index)
{
      if ((index>=0) && (index<size))
            return new Point(xpoints[index],ypoints[index]);
      return new Point(0,0);
}

public void fitVertexAt(int index, int x, int y, DrawingEditor editor)
{
      Point pos = getPosition();
      if ((index>=0) && (index<size))
            moveVertexTo(index,x,y);
}

public void setVertexAt(int index, int x, int y, DrawingEditor editor)
{
      if ((index>=0) && (index<size))
      {
            xpoints[index]=x;
            ypoints[index]=y;
      }
}

public void moveVertexTo(int index, int x, int y)
{
      Point offset = new Point(x-xpoints[index],y-ypoints[index]);

      for (int i=0;i<size;i++)
            xpoints[i] += offset.x;
      for (int j=0;j<size;j++)
            ypoints[j] += offset.y;
}
}     // End Shape
```

- Insertion/deletion of vertex in an existing polygon. A vertex is firstly selected with Control+click (it is then visualized as a large red dot). Insertion/deletion are performed by subsequent pressing of the '+' or the '-' key. In the former case a new

vertex in inserted between the selected one and the next one in the polygon (the new vertex is automatically selected to prompt for its immediate translation, see next operation), while in the latter case the selected vertex is just removed from the polygon (see figure 2.7).

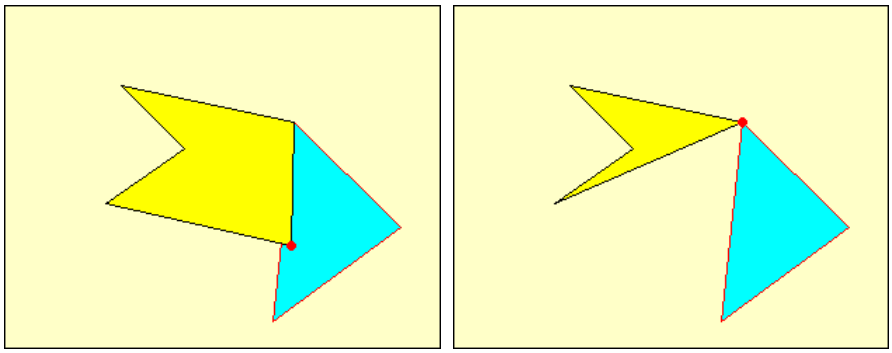

Figure 2.7. Deletion of a selected vertex from a polygon.

The above are implemented as follows* :

```
public class DrawingEditor extends Applet
      implements Constants, MouseListener, MouseMotionListener,
            KeyListener, ActionListener, UndoEntity
{
. . . . . .
// Variable used for vertex selection : selected vertex's shape
Shape selectedVertexShape = null;

public void mousePressed(MouseEvent evt)
{
      . . . . . .
      clearSelectedVertices();
      if ((!shift) && (control) && (!alt))
      {     // Select a vertex
            int i = 0;
            while (i<target.shapes.size())
            {
                  if ((comp = (Shape)target.shapes.elementAt(i)).
                              selectVertex(x,y))
                  {
                        selectedVertexShape = comp;
                        clearFitData();
                        repaint(); return;
                  }
                  i++;
            }
      }

//    Select vertices to fit.
      . . . . . .
}

public void keyPressed(KeyEvent evt)
{
```

```
      . . . . . .
      if ((ch == '+') && ((!shift) && (!control) && (!alt)))
      {
            if (selectedVertexShape != null)
            {
                  if (selectedVertexShape.type == POLYGON)
                  {
                        if (selectedVertexShape.canGrow())
            {selectedVertexShape.createNewVertex(this);repaint();}
                        else MessageDialog.open(
      "This polygon has already the maximum number of vertices !");
                  }
            }
      }
      if ((ch == '-') && ((!shift) && (!control) && (!alt)))
      {
            if (selectedVertexShape != null)
            {
                  if (selectedVertexShape.type == POLYGON)
            {selectedVertexShape.deleteSelectedVertex(this);
                        repaint();}
            }
      }
}

public void clearSelectedVertices()
{
      if (selectedVertexShape != null)
      {
            selectedVertexShape.clearSelectedVertex();
            selectedVertexShape = null;
            repaint();
      }
}

}     // End DrawingEditor

class Shape implements Constants, UndoEntity
{
. . . . . .
int selectedVertexIndex = -1;

public boolean canGrow()
{
      return (size < SizeLimit);
}

public void createNewVertex(DrawingEditor editor)
{
      // Only for polygons
      if (selectedVertexIndex != -1)
      {
            int i0 = selectedVertexIndex;
            for (int j=(size-1);j>i0;j--)
            {
                  xpoints[j+1]=xpoints[j];
```

```
                ypoints[j+1]=ypoints[j];
            }
            int i2 = (i0 == (size-1)) ? 0 : (i0+2);
            // The above is correct for the last value, i.e. the
            // value in position (size-1), because values
            // have already shifted forward one position.
            // The new vertex is now in position (i0+1)
            xpoints[i0+1]=((xpoints[i0]+xpoints[i2])/2) + 1;
            ypoints[i0+1]=((ypoints[i0]+ypoints[i2])/2) - 1;
            size++;
            // Select the new vertex
            selectedVertexIndex++;
      }
}

public void deleteSelectedVertex(DrawingEditor editor)
{
      // Only for polygons
      if (selectedVertexIndex != -1)
      {
            for (int j=selectedVertexIndex+1;j<size;j++)
            {
                  xpoints[j-1]=xpoints[j];
                  ypoints[j-1]=ypoints[j];
            }
            size--;
            if (selectedVertexIndex == size) selectedVertexIndex--;
            if (size == 2) type = LINE;
      }
}

public boolean selectVertex(int x, int y)
{
      // Only for polygons
      for (int i=0;i<size;i++)
      {
            if ((Math.abs(xpoints[i]-x) <= 5)
                  && (Math.abs(ypoints[i]-y) <= 5))
            {
                  selectedVertexIndex = i;
                  return true;
            }
      }
      clearSelectedVertex();
      return false;
}

public void clearSelectedVertex() {selectedVertexIndex = -1;}

void drawUnselected(Graphics g)
{
      . . . . . .
      // Draw a red dot around the selected vertex
      if (selectedVertexIndex != -1)
      {
            g.setColor(Color.red);
```

```
            g.fillOval(xpoints[selectedVertexIndex]-3,
                        ypoints[selectedVertexIndex]-3,
                        7,7);
            g.setColor(pen);
      }
}

//    Undo functions : insert to undo delete, delete to undo create

public void insertVertex(int index, int x, int y)
{
      // Only for polygons
      if (index != -1)
      {
            // There is no need to check for size limits, because
            // this is an undo operation and there has been
            // a valid polygon from which a vertex was deleted
            for (int j=(size-1);j>=index;j--)
            {
                  xpoints[j+1]=xpoints[j];
                  ypoints[j+1]=ypoints[j];
            }
            xpoints[index]=x;
            ypoints[index]=y;
            size++;
            type = POLYGON;
      }
}

public void deleteVertex(int index)
{
      // Only for polygons
      if (index != -1)
      {
            for (int j=index+1;j<size;j++)
            {
                  xpoints[j-1]=xpoints[j];
                  ypoints[j-1]=ypoints[j];
            }
            size--;
            // The following line is unnecessary, because this is
            // an undo operation, and there is no way to add a vertex
            // to a line and turn it into a polygon
            if (size == 2) type = LINE;
      }
}

}     // End Shape
```

- Translation of a polygon vertex or polygon deformation (see figure 2.8). Once a polygon vertex is selected with Control+click, it may be dragged with the mouse (the Control key must be kept pressed).

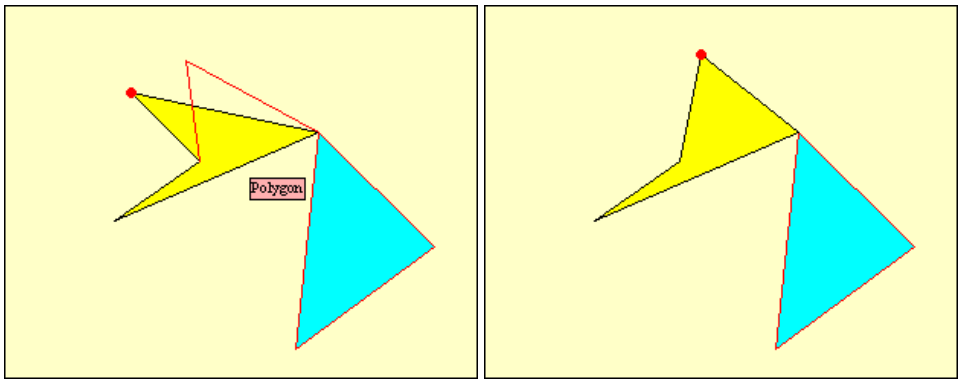

Figure 2.8. Vertex dragging in a polygon (deformation).

This operation is implemented as follows* :

```
public class DrawingEditor extends Applet
      implements Constants, MouseListener, MouseMotionListener,
            KeyListener, ActionListener, UndoEntity
{
. . . . . .
//    Variable used for vertex dragging : boolean flag
boolean vertexMouseDrag = false;

public void mousePressed(MouseEvent evt)
{
s
//    If a vertex is selected, start dragging it.
      clearSelectedVertices();
      if ((!shift) && (control) && (!alt))
      {     // Select a vertex
            int i = 0;
            while (i<target.shapes.size())
            {
                  if ((comp = (Shape)target.shapes.elementAt(i)).
                              selectVertex(x,y))
                  {
                        selectedVertexShape = comp;
                        // Start dragging selected vertex
                        setVertexMouseDrag();
                        prev_x = x; prev_y = y;
                        comp.startVertexDragging();
                        clearFitData();
                        repaint(); return;
                  }
                  i++;
            }
      }
      . . . . . .
}

public void mouseReleased(MouseEvent evt)
{
      . . . . . .
```

```
      if ((!shapeMouseDrag) && (!vertexMouseDrag)
                        && (!rubberbandMouseDrag))
            return;
      if (vertexMouseDrag)
      {
            resetVertexMouseDrag();
            if (!target.includes(x,y)) return;
            comp = selectedVertexShape;
            if (comp.vertexHasBeenDragged())
            {
                  comp.dragVertexBy(x-prev_x,y-prev_y);
                  comp.confirmVertexDragged(target);
            }
            prev_x = -1; prev_y = -1;
      }
 }

public void mouseDragged(MouseEvent evt)
{
      . . . . . .
      if ((!shapeMouseDrag) && (!vertexMouseDrag)
                        && (!rubberbandMouseDrag))
            return;
      if (vertexMouseDrag)
      {
            selectedVertexShape.dragVertexBy(x-prev_x,y-prev_y);
            prev_x = x; prev_y = y;
      }
}

private void setVertexMouseDrag() {vertexMouseDrag = true;}
private void resetVertexMouseDrag() {vertexMouseDrag = false;}

public void paint(Graphics g)
{
      . . . . . .
      if (vertexMouseDrag)
            selectedVertexShape.drawVertexDragged(g);
}

}     // End DrawingEditor

class Shape implements Constants, UndoEntity
{
. . . . . .
Point vertexDragPosition = new Point(0,0);

public void dragVertexBy(int x, int y)
      {vertexDragPosition.x +=x; vertexDragPosition.y +=y;}

public void startVertexDragging()
{
      vertexDragPosition.x = xpoints[selectedVertexIndex];
      vertexDragPosition.y = ypoints[selectedVertexIndex];
}
```

```
public boolean vertexHasBeenDragged()
{
      Point pos = getPosition();
      return ((vertexDragPosition.x != xpoints[selectedVertexIndex]) ||
                  (vertexDragPosition.y !=
ypoints[selectedVertexIndex]));
}

public void confirmVertexDragged(Drawing target)
{
      if (target.includes(vertexDragPosition.x,vertexDragPosition.y))
      {
            xpoints[selectedVertexIndex] = vertexDragPosition.x;
            ypoints[selectedVertexIndex] = vertexDragPosition.y;
      }
      if (type == CIRCLE) validateCircleSize();
}

public void validateCircleSize()
{
      validateCircle();
      int   cx = xpoints[1] - xpoints[0], cy = ypoints[1] - ypoints[0];
      if (cx < cy) xpoints[1] += (cy-cx);
      if (cy < cx) ypoints[1] += (cx-cy);
}

void drawVertexDragged(Graphics g)
{
      Color pen = g.getColor();
      g.setColor(Color.red);

      switch (type)
      {
            case POLYGON:
                  if (selectedVertexIndex == 0)
                  {
                        g.drawLine(xpoints[size-1],ypoints[size-1],
                              vertexDragPosition.x,
                              vertexDragPosition.y);
                        g.drawLine(vertexDragPosition.x,
                              vertexDragPosition.y,
                              xpoints[1],ypoints[1]);
                  }
                  else if (selectedVertexIndex == (size-1))
                  {
                        g.drawLine(xpoints[size-2],ypoints[size-2],
                              vertexDragPosition.x,
                              vertexDragPosition.y);
                        g.drawLine(vertexDragPosition.x,
                              vertexDragPosition.y,
                              xpoints[0],ypoints[0]);
                  }
                  else
                  {
                        g.drawLine(xpoints[selectedVertexIndex-1],
                              ypoints[selectedVertexIndex-1],
```

```
                              vertexDragPosition.x,
                              vertexDragPosition.y);
                        g.drawLine(vertexDragPosition.x,
                              vertexDragPosition.y,
                              xpoints[selectedVertexIndex+1],
                              ypoints[selectedVertexIndex+1]);
                  }
                  break;
            case CIRCLE:
                  if (selectedVertexIndex == 0)
                        validateAndDrawCircle(g,
                              vertexDragPosition.x,vertexDragPosition.y,
                              xpoints[1],ypoints[1]);
                  else validateAndDrawCircle(g,
                              xpoints[0],ypoints[0],
                        vertexDragPosition.x,vertexDragPosition.y);
                  break;
            case LINE:
                  if (selectedVertexIndex == 0)
                        g.drawLine(vertexDragPosition.x,
                                    vertexDragPosition.y,
                                    xpoints[1],ypoints[1]);
                  else g.drawLine(vertexDragPosition.x,
                                    vertexDragPosition.y,
                                    xpoints[0],ypoints[0]);
                  break;
      }
      g.setColor(pen);
}

void validateAndDrawCircle(Graphics g, int x1, int y1, int x2, int y2)
{
      int temp;
      if (x1 > x2)
      {     // Swap x1 and x2
            temp = x1; x1 = x2; x2 = temp;
      }
      if (y1 > y2)
      {     // Swap y1 and y2
            temp = y1; y1 = y2; y2 = temp;
      }
      g.drawOval(x1,y1,x2-x1,y2-y1);
}

}     // End Shape
```

- Single polygon splitting or multiple polygons joining. Splitting is achieved by selecting two non consecutive vertices with Shift+Control+click and then pressing Control-d † (see figure 2.9). Joining is achieved by selecting four vertices (the first and the fourth one belonging to the first polygon, the other two to the second one) and Control-j †. Pairs of vertices for each of the two polygons must be consecutive. Joining is performed by translating the second polygon so that the second vertex coincides with the first and then deforming so that the third vertex coincides with the fourth. Vertices selected for joining or splitting are visualized as large blue crosses.

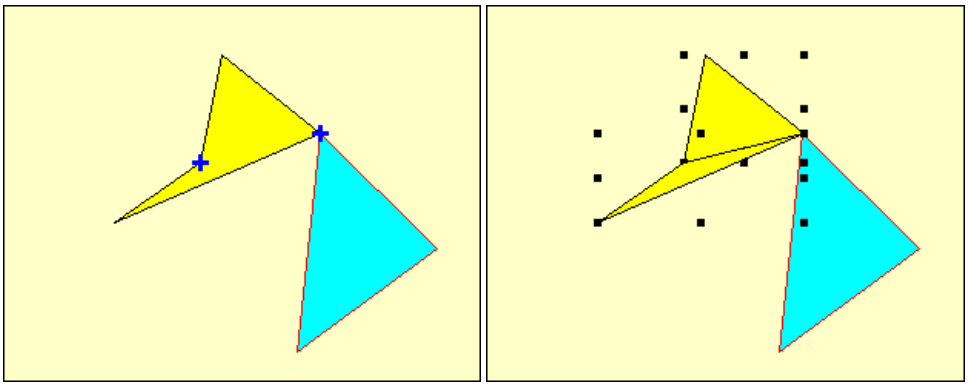

Figure 2.9. Splitting of a polygon.

The above are implemented as follows* :

```
public class DrawingEditor extends Applet
      implements Constants, MouseListener, MouseMotionListener,
            KeyListener, ActionListener, UndoEntity
{
. . . . . .
//    Variables used for joining/dividing polygonal shapes
Shape joinDivideShapes[] = {null,null,null,null};
int joinDivideVertices[] = {-1,-1,-1,-1};

public void mousePressed(MouseEvent evt)
{
      . . . . . .
//    Select vertices for polygon join/divide operation
      if (shift && control && (!alt))
      {
            int i=0, j, vtx;
            while (i<target.shapes.size())
            {
                  Shape s = (Shape)target.shapes.elementAt(i);
                  if (((vtx=s.indexOfVertexAt(x,y)) != -1)
                        && (s.type == POLYGON))
                  {
                        if (joinDivideShapes[0] == null) j=0;
                        else if (joinDivideShapes[1] == null) j=1;
                        else if (joinDivideShapes[2] == null) j=2;
                        else if (joinDivideShapes[3] == null) j=3;
                        else j = 4;
                        if (j<4)
                        {
                              joinDivideShapes[j]=s;
                              joinDivideVertices[j]=vtx;
                        }
                        else
                        {
                        //    Shift join/divide vertices array
                        //    toward the beginning
                              for (int k=1;k<4;k++)
                              {
```

```
                                    joinDivideShapes[k-1]=
                                          joinDivideShapes[k];
                                    joinDivideVertices[k-1]=
                                          joinDivideVertices[k];
                              }
                              joinDivideShapes[3]=s;
                              joinDivideVertices[3]=vtx;
                        }
                        repaint();
                        return;
                  }
                  i++;
            }
      }
      clearJoinDivideData();
}

public void keyPressed(KeyEvent evt)
{
      . . . . . .
      if (((ch == 'd') || (ch == 'D')) && onlyControl)
      {
            // Join/divide shapes can only be polygons
            if ((joinDivideShapes[0] != null)
                  && (joinDivideShapes[1] != null)
                  && (joinDivideShapes[0] == joinDivideShapes[1]))
            joinDivideShapes[0].split(joinDivideVertices[0],
                                          joinDivideVertices[1],this);
            clearJoinDivideData();
            repaint(); return;
      }
      if (((ch == 'j') || (ch == 'J')) && onlyControl)
      {     // Join/divide shapes can only be polygons
            if ((joinDivideShapes[0] != null)
                  && (joinDivideShapes[1] != null)
                  && (joinDivideShapes[2] != null)
                  && (joinDivideShapes[3] != null)
                  && (joinDivideShapes[0] == joinDivideShapes[3])
                  && (joinDivideShapes[1] == joinDivideShapes[2]))
            joinPolygons(joinDivideShapes[0], joinDivideShapes[1],
                        joinDivideVertices[0],joinDivideVertices[1],
                        joinDivideVertices[2],joinDivideVertices[3]);
            clearJoinDivideData();
            repaint(); return;
      }
}

public void paint(Graphics g)
{
      . . . . . .
      for (int i=0;i<4;i++)
            if (joinDivideShapes[i]!=null)
                  drawJoinDividePoint(g,
                        joinDivideShapes[i].vertexAt(
                                    joinDivideVertices[i]));
}
```

```
void drawJoinDividePoint(Graphics g, Point po)
{     // Draw a blue cross
      Color pen = g.getColor();
      g.setColor(Color.blue);
      g.fillRect(po.x-1,po.y-5,3,11);
      g.fillRect(po.x-5,po.y-1,11,3);
      g.setColor(pen);
}

public void joinPolygons(Shape pol1, Shape pol2, int a1, int b1,
                              int b2, int a2)
{
      if ((pol1.type != POLYGON) || (pol2.type != POLYGON)
                  || (pol1 == pol2))
            return;
      if ((!pol1.adjacentIndexes(a1,a2))
                  || (!pol2.adjacentIndexes(b1,b2)))
            return;

//    Data for the new polygon
      Shape newPolygon;
      int newSize, newXpoints[], newYpoints[];
      Color newFillColor = mixColors(pol1.fillColor,pol2.fillColor),
            newBorderColor = mixColors(pol1.borderColor,
                                       pol2.borderColor);

      // Step 1. Fit vertex b1 to a1 (move polygon 2)
      pol2.fitVertexAt(b1,pol1.xpointFor(a1),pol1.ypointFor(a1),this);

      // Step 2. Set vertex b2 to a2 (deform polygon 2)
      pol2.setVertexAt(b2,pol1.xpointFor(a2),pol1.ypointFor(a2),this);

      // Step 3. Merge polygon 1 and polygon 2
      //         (omit redundant vertices b1 and b2)

      newSize = pol1.size + pol2.size - 2;
      newXpoints = new int[newSize];
      newYpoints = new int[newSize];

      // First part : Take newXpoints and newYpoints
      //in first polygon up to a1
      . . . . . .
      // Second part : Take newXpoints and newYpoints
      //in second polygon from b1 to b2
      . . . . . .
      // Third part: Take newXpoints and newYpoints
      //in first polygon from a2 to end
      . . . . . .
      // Step 4. Delete polygons 1 and 2. Add new polygon.
      newPolygon = new Shape(POLYGON,newXpoints,newYpoints,newSize,
                                    newFillColor,newBorderColor);
      target.remove(pol1);
      target.remove(pol2);
      target.add(newPolygon);
      select(newPolygon);
```

```
}

protected Color mixColors(Color c1, Color c2)
{
      return new Color(
            (c1.getRed()+c2.getRed())/2,
            (c1.getGreen()+c2.getGreen())/2,
            (c1.getBlue()+c2.getBlue())/2);
}

public void clearJoinDivideData()
{
      for (int i=0;i<4;i++)
            {joinDivideShapes[i]=null; joinDivideVertices[i]=-1;}
}

}     // End DrawingEditor

class Shape implements Constants, UndoEntity
{
. . . . . .

public void split(int from, int to, DrawingEditor editor)
{
      int f = from, t = to, i;
      if (type != POLYGON) return;
      if (f==t) return;
      if (f>t)
      {     // Swap f and t
            int x = f;
            f = t;
            t = x;
      }
      if ((t-f)==1) return;

      // First create and add a new polygon
      int pxpoints[] = new int[t-f+1];
      int pypoints[] = new int[t-f+1];
      for (i=f;i<=t;i++)
      {
            pxpoints[i-f] = xpoints[i];
            pypoints[i-f] = ypoints[i];
      }
      Shape newpolygon = new Shape(POLYGON,pxpoints,pypoints,t-f+1,
                                    fillColor,borderColor);
      editor.target.add(newpolygon);
      editor.select(newpolygon);
      if (!selected) editor.select(this);

      // Then, remove extra vertices from this polygon, to
      // create the shortcut from f to t
      for (i=t;i<size;i++)
      {
            xpoints[i-(t-f-1)]=xpoints[i];
            ypoints[i-(t-f-1)]=ypoints[i];
```

```
        }
        size -= (t-f-1);
}
}       // End Shape
```

2.3.2 *Foreground and background*

Our example editor supports operations for moving selected objects to the foreground or to the background with the key combinations Alt-f (front) and Alt-b (back), respectively (see figure 2.10). The implementation of this operation is very simple and relies on the observation that the objects (shapes) are stored in a vector and drawn in order of appearance in it : thus, moving an object to the background corresponds to moving it to the beginning of the vector, while moving it to the foreground corresponds to moving it to the end of the vector.

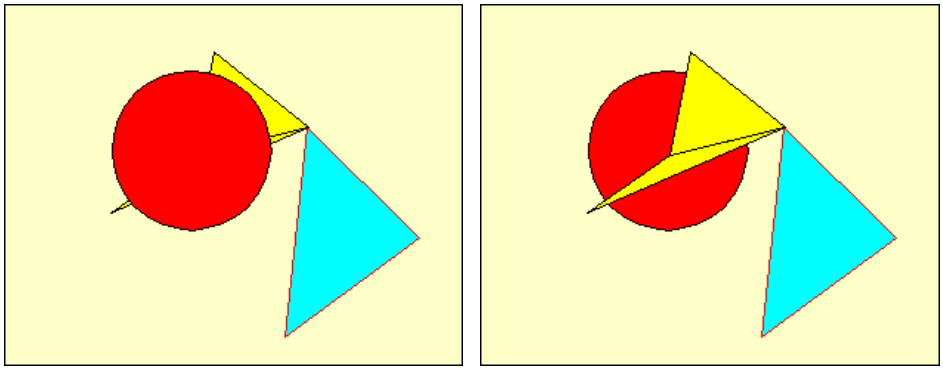

Figure 2.10. Red circle in foreground and background.

The above are implemented as follows * :

```
public class DrawingEditor extends Applet
        implements Constants, MouseListener, MouseMotionListener,
                KeyListener, ActionListener, UndoEntity
{
. . . . . .
public void keyPressed(KeyEvent evt)
{
        . . . . . .
        if (((ch == 'b') || (ch == 'B')) && onlyControl)
        {       // Send selections to back
                sendSelectionsToBack();
                repaint(); return;
        }

        if (((ch == 'f') || (ch == 'F')) && onlyControl)
        {       // Bring selections to front
                bringSelectionsToFront();
                repaint(); return;
        }
}
```

```
public void sendSelectionsToBack()
{
      Vector ordered_selections = target.order(selections);
      for (int i=0;i<ordered_selections.size();i++)
            Shape comp = (Shape)ordered_selections.elementAt(i);
      target.sendToBack(ordered_selections);
}

public void bringSelectionsToFront()
{
      Vector ordered_selections = target.order(selections);
      for (int i=ordered_selections.size()-1;i>=0;i--)
            Shape comp = (Shape)ordered_selections.elementAt(i);
      target.bringToFront(ordered_selections);
}

}     // End DrawingEditor

class Drawing implements Constants
{
. . . . . .
public void sendToBack(Vector someShapes)
{
//    Although it does not matter for an ordered vector of shapes,
//    the operation works using the vector in inverse order.
      for (int i=someShapes.size()-1;i>=0;i--)
      {
            Shape aShape = (Shape)someShapes.elementAt(i);
            shapes.removeElement(aShape);
            shapes.insertElementAt(aShape,0);
      }
}

public void bringToFront(Vector someShapes)
{
      for (int i=0;i<someShapes.size();i++)
      {
            Shape aShape = (Shape)someShapes.elementAt(i);
            shapes.removeElement(aShape);
            shapes.addElement(aShape);
      }
}

public void setOrder(Shape s, int i)
{
      shapes.removeElement(s);
      shapes.insertElementAt(s,i);
}

}     // End Drawing
```

2.3.3 Tip drawing

Our example editor supports a tip drawing operation, whereby a small explanatory message is drawn for an object (see figure 2.8). This operation is performed whenever the cursor is placed over an object, provided that there is no selected object and the mouse is not being dragged (it is supposed that the user "explores" the various objects before selecting one of them).

This operation is implemented as follows * :

```
public class DrawingEditor extends Applet
      implements Constants, MouseListener, MouseMotionListener,
            KeyListener, ActionListener
{
. . . . . .
// Variable used for graphical elements tips
Shape tipElement = null;

public void mousePressed(MouseEvent evt)
{
      . . . . . .
      // Selection with dragging, tip drawing and color edition etc.
      . . . . . .
            int i = target.shapes.size()-1;
            while (i >= 0)
            {
                  comp = (Shape)target.shapes.elementAt(i);
                  if (comp.includes(evt.x,evt.y))
                  {
                        . . . . . .
                        // Start dragging selections
                        . . . . . .
                        // Tip stuff
                        resetTipElement();
                        posX = x; posY = y;
                        // If double-clicked, edit the selection
                        . . . . . .
                        repaint(); return;
                  }
                  i--;
            }
            clearSelections();
            repaint();
}

public void mouseMoved(MouseEvent evt)
{
      Shape el;
      . . . . . .

      if (shapeMouseDrag || vertexMouseDrag || (selections.size() > 0))
            {resetTipElement(); repaint(); return;}

      int i = target.shapes.size()-1;
```

```
      while (i >= 0)
      {
            if ( (el=(Shape)target.shapes.elementAt(i)).includes(x,y) )
            {
                  tipElement=el;
                  // Changing the cursor to a hand
                  setCursor(Cursor.getPredefinedCursor(
                                    Cursor.HAND_CURSOR));
                  repaint();
                  return;
            }
            i--;
      }
      if (tipElement != null) { resetTipElement(); repaint(); }
}

public void paint(Graphics g)
{
      . . . . . .
      if (tipElement != null) tipElement.drawTip(g);
}

private void resetTipElement()
{
      tipElement = null;
      setCursor(Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR));
}

}     // End DrawingEditor

class Shape implements Constants, UndoEntity
{
. . . . . .
public void drawTip(Graphics g)
{
String tip = name();
Color old = g.getColor();
Font oldFont = g.getFont();
FontMetrics metrics;

g.setFont(tipFont);
metrics = g.getFontMetrics();
Point stringOffset = new Point(1,1);
Dimension tipSize = new Dimension(
      metrics.stringWidth(tip)+stringOffset.x+stringOffset.x,
      metrics.getHeight()+stringOffset.y+stringOffset.y);
Point tipPosition = new Point(getPosition().x+((width()+width())/3),
                              getPosition().y+((height()+height())/3));

g.setColor(tipColor);
g.fillRect(tipPosition.x,tipPosition.y,tipSize.width,tipSize.height);
g.setColor(old);
g.drawRect(tipPosition.x,tipPosition.y,tipSize.width,tipSize.height);
g.drawString(tip,tipPosition.x+stringOffset.x,
                  tipPosition.y+stringOffset.y+metrics.getAscent());
g.setFont(oldFont);
```

```
}

public String name()
{
      switch (type)
      {
            case LINE: return "Line";
            case CIRCLE: return "Circle";
            case POLYGON: return "Polygon";
            default: return "Shape";
      }
}

}     // End Shape
```

2.3.4 F8 through selections

Our example editor supports the operation of "exploration" of consecutive objects by iterating through them with the F8 key. Objects are selected in the corresponding order.

This operation is implemented as follows* :

```
public class DrawingEditor extends Applet
      implements Constants, MouseListener, MouseMotionListener,
            KeyListener, ActionListener, UndoEntity
{
. . . . . .
// Variable used for looping around components with the F8 key
int selection_index = -1;

public void keyPressed(KeyEvent evt)
{
      . . . . . .
      boolean tab=false;
      if (key == KeyEvent.VK_F8) tab=true;

      if (tab)
      {
            if (selections.size() > 0)
            {
            // Mark (last) selection's index
                  selection_index = target.findComponentIndexFor(
                  (Shape)selections.elementAt(selections.size()-1));
// Loop around target components, repeatedly updating last selection
// 1. First continue from current selection index to end of components
                  for (int i=selection_index+1;
                              i<target.shapes.size();i++)
                  {
                        comp = (Shape)target.shapes.elementAt(i);
                        selection_index = i;
                        deselect((Shape)selections.elementAt(
                                    selections.size()-1));
                        select(comp);
```

```
                        repaint(); return;
                    }
        // 2. Then restart from the beginning up to the current selection
                    for (int i=0;i<=selection_index;i++)
                    {
                        comp = (Shape)target.shapes.elementAt(i);
                        selection_index = i;
                        deselect((Shape)selections.elementAt(
                                    selections.size()-1));
                        select(comp);
                        repaint(); return;
                    }
                }
        }
}

}       // End DrawingEditor

class Drawing implements Constants
{
. . . . . .
public int findComponentIndexFor(Shape arg)
{
        for (int i=0;i<shapes.size();i++)
              if ((Shape)shapes.elementAt(i) == arg) return i;
        return -1;
}

}       // End Drawing
```

2.3.5 Colors and object transparency

Beside editing the fill and border colors of a selected object, our example editor supports the reversal of each one of them with the key combinations Alt+Shift-i, Alt-i respectively. Furthermore, an object may be defined to become opaque or transparent with the key combination Control-t [†] (see figure 2.11) in which case the fill color is shown or not shown, respectively. In the same manner, the background color of the whole drawing area may be defined or reversed.

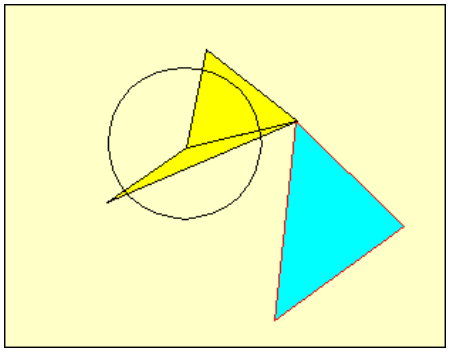

Figure 2.11. Circle in the foreground and transparent (red in figure 2.10).

The above are implemented as follows [*] :

```
public class DrawingEditor extends Applet
      implements Constants, MouseListener, MouseMotionListener,
            KeyListener, ActionListener, UndoEntity
{
. . . . . .

public void keyPressed(KeyEvent evt)
{
      . . . . . .
      if ((ch == 'i') || (ch == 'I'))
      {
            if ((!shift) && (!control) && alt)
            {     // Invert selections fill colors
                  for (int i=0;i<selections.size();i++)
                  {
                        comp = (Shape)selections.elementAt(i);
                        comp.invertFillColor();
                  }
                  repaint(); return;
            }
            else if ((shift) && (!control) && alt)
            {     // Invert selections border colors
                  for (int i=0;i<selections.size();i++)
                  {
                        comp = (Shape)selections.elementAt(i);
                        comp.invertBorderColor();
                  }
                  repaint(); return;
            }
      }
      if (((ch == 't') || (ch == 'T')) && onlyControl)
      {
            // Switch selections to transparent/opaque mode
            for (int i=0;i<selections.size();i++)
            {
                  comp = (Shape)selections.elementAt(i);
                  comp.switchTransparency();
            }
            repaint(); return;
      }
}

}     // End DrawingEditor

class Shape implements Constants, UndoEntity
{
private boolean opaque = true;

public void invertFillColor()
{
      Color c = fillColor;
      fillColor = new Color(255-c.getRed(),
                              255-c.getGreen(),255-c.getBlue());
}
```

```
public void invertBorderColor()
{
      Color c = borderColor;
      borderColor = new Color(255-c.getRed(),
                              255-c.getGreen(),255-c.getBlue());
}

public void switchTransparency()
{
      opaque = (!opaque);
}

void drawUnselected(Graphics g)
{
      . . . . . .

      switch (type)
      {
            case LINE:
                  . . . . . .
            case CIRCLE:
                  . . . . . .
                  if (opaque)
                  {
                        g.setColor(fillColor);
                        g.fillOval(up.x,up.y,diameter,diameter);
                  }
                  g.setColor(borderColor);
                  g.drawOval(up.x,up.y,diameter,diameter);
                  g.setColor(pen);
                  break;
            case POLYGON:
                  if (opaque)
                  {
                        g.setColor(fillColor);
                        g.fillPolygon(xpoints,ypoints,size);
                  }
                  g.setColor(borderColor);
                  g.drawPolygon(xpoints,ypoints,size);
                  g.setColor(pen);
                  break;
      }
      . . . . . .
}

}     // End Shape
```

2.3.6 Drawing grid

In many graphical editors it is convenient to define a grid of horizontal and vertical lines over the drawing area to allow high precision drawing (see figure 2.12). The grid may be

made visible or invisible online with the key combination Alt-g and its dimension and color may be edited with the key combination Alt+Shift-g.

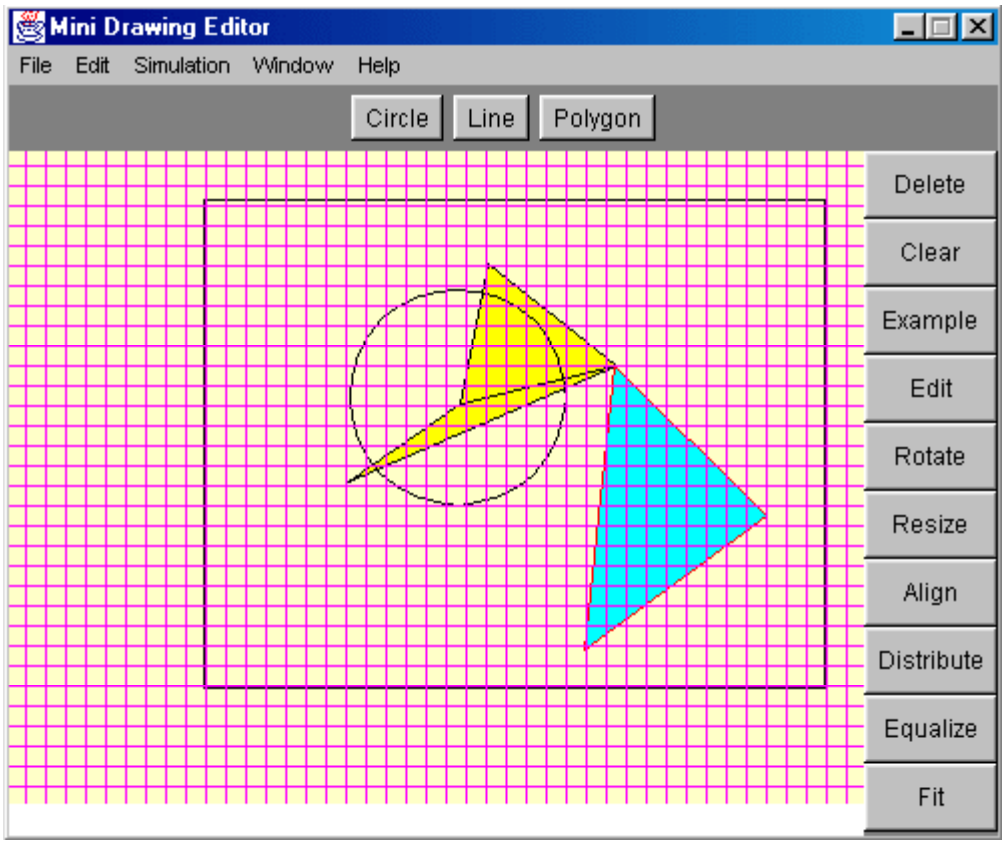

Figure 2.12. Grid visible.

The above are implemented as follows * :

```
public class DrawingEditor extends Applet
      implements Constants, MouseListener, MouseMotionListener,
            KeyListener, ActionListener, UndoEntity
{
. . . . . .
//    Variables used for grid visualization
boolean showGrid = false;
int gridSize = 10;
Color gridColor = Color.lightGray;

public void keyPressed(KeyEvent evt)
{
      . . . . . .
      if ((ch == 'g') || (ch == 'G'))
      {
            if ((!shift) && (!control) && alt)
            {     // Switch grid on/off
                  showGrid = (!showGrid);
                  repaint();
            }
```

```
            else if ((shift) && (!control) && alt)
            {     // Edit grid size and color
                  editGrid();
            }
      }
}

public void paint(Graphics g)
{
      . . . . . .
      // Draw grid before other things
      if (showGrid) showGrid(g,gridSize);

      if (shapeMouseDrag)
      . . . . . .
            etc.

}

public void showGrid(Graphics g, int gs)
{
      Color old = g.getColor();
      g.setColor(gridColor);
      for (int i=1;i<SizeX;i+=gs) g.drawLine(i,0,i,SizeY);
      for (int j=1;j<SizeY;j+=gs) g.drawLine(0,j,SizeX,j);
      g.setColor(old);
}

}     // End DrawingEditor
```

2.3.7 Grouping/ungrouping

To allow for grouping we redesign the Shape class as a hierarchy of 3 classes : an abstract base class Shape implementing all operations common to both simple shapes and groups of shapes, and two subclasses, the SimpleShape class that contains all the functionality of the previous Shape class, and the ShapeGroup class, implementing possibly recursive groups of shapes. Grouping of multiple selecting shapes or ungrouping og a single selected group are performed by pressing Control-u or Control-U [†] (see figure 2.13). The introduction of the grouping possibility introduces subtle intricacies in the final code, mainly in the implementation of operations that are performed recursively.

The above are implemented as follows [*] :

```
public class DrawingEditor extends Applet
      implements Constants, MouseListener, MouseMotionListener,
            KeyListener, ActionListener, UndoEntity
{
. . . . . .
public void keyPressed (KeyEvent evt)
{
      . . . . . .
```

```
        if (((ch == 'u') || (ch == 'U')) && onlyControl)
        {       // Group/ungroup selections
                doGroupUngroupSelections();
                repaint(); return;
        }
}
```

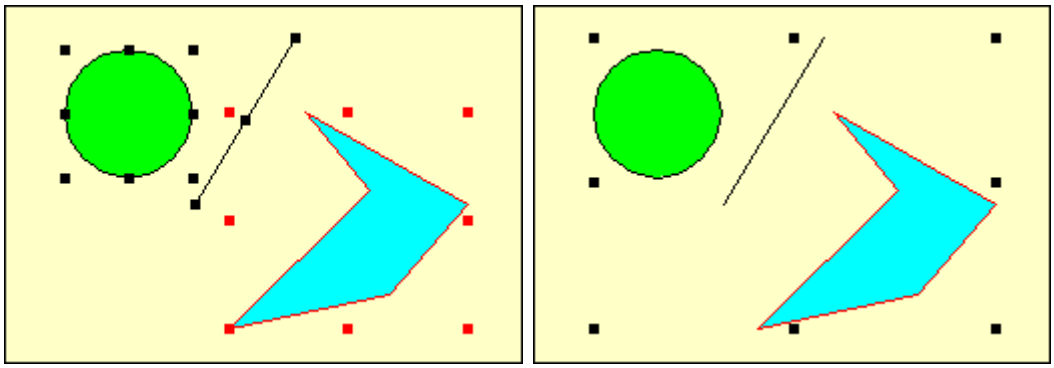

Figure 2.13. Grouping of shapes.

```
public void doGroupUngroupSelections()
{
        if (selections.size() == 1)
        {       // Ungroup selections
                Shape s = (Shape)(selections.elementAt(0));
                if (s instanceof ShapeGroup)
                {
                        target.remove(s);
                        clearSelections();
                        Vector v = ((ShapeGroup)s).ungrouped();
                        for (int i=0;i<v.size();i++)
                        {
                                Shape s1 = (Shape)v.elementAt(i);
                                target.add(s1);
                                select(s1);
                        }
                }
        }
        else if (selections.size() >= 2)
        {       // Group selections
                for (int i=0;i<selections.size();i++)
                {
                        Shape s = (Shape)selections.elementAt(i);
                        target.remove(s);
                        s.deselect();
                }
                ShapeGroup sg = new ShapeGroup(selections);
                clearSelections();
                target.add(sg);
                select(sg);
        }
}

}       // End DrawingEditor
```

```
abstract class Shape implements Constants, UndoEntity
{
protected ShapeGroup group = null;

protected boolean selected = false;
Point dragPosition = new Point(0,0);

static Color tipColor = Color.pink;
static Font tipFont = new Font("Times Roman",Font.PLAIN,9);

public void anchorAt(ShapeGroup sg) { group = sg; }

public ShapeGroup getTopGroup()
{
      if (group == null) return getThisGroup();
      return group.getTopGroup();
}
abstract public ShapeGroup getThisGroup();

public void clearGroup()
{     group = null;     }

protected String name() { return "Shape"; }

public void edit(DrawingEditor ed)
{. . . . . }

public void select() {selected = true;}
public void deselect() {selected = false;}
public boolean isSelected() {return selected;}

public boolean includes(int x, int y)
{
      return rectangleIncludes(getPosition(), width(), height(), x, y);
}

public boolean rectangleIncludes(Point upperleft, int width, int height,
                                 int x, int y)
{. . . . . }

public Dimension getSize()
{
      return new Dimension(width(),height());
}

public void setSize(DrawingEditor ed)
{
      ResizeDialog.open(this,ed);
}

abstract public void setSize(int wi, int hei);

abstract void moveBy(int x, int y);
abstract void increaseBy(int x, int y, DrawingEditor editor);
abstract void decreaseBy(int x, int y, DrawingEditor editor);
```

```
public void rotate(DrawingEditor editor)
{     rotateBy(1,0);    }

abstract void rotateBy(int x, int y);
abstract void rotateAround(int cx, int cy, boolean right);

protected boolean rotateClockwise(int x, int y)
{. . . . . }

abstract void invertFillColor();
abstract void invertBorderColor();
abstract void switchTransparency();

public void dragBy(int x, int y)
      {dragPosition.x +=x; dragPosition.y +=y;}
public void startDragging()
      {     dragPosition.x = getPosition().x;
            dragPosition.y = getPosition().y;   }
public boolean hasBeenDragged()
{
      Point pos = getPosition();
      return ((dragPosition.x != pos.x) || (dragPosition.y != pos.y));
}

public void confirmDragged(Drawing target)
{
      if (target.includes(dragPosition.x,dragPosition.y) &&
            target.includes(dragPosition.x+width(),
                            dragPosition.y+height()))
      setPosition(dragPosition.x,dragPosition.y);
}

public Point getAlignPosition(int pos)
{. . . . . }

public void align(int type, int pos, Point po)
{. . . . . }

abstract void setPosition(int x, int y);

abstract void equalize(int dimension, Dimension po, DrawingEditor
editor);
abstract Shape copy();
abstract Point getPosition();

public Point relativePosition(Point rel)
{. . . . . }

public Point topleft() { return getPosition(); }
abstract Point topright();
abstract Point bottomleft();
abstract Point bottomright();
abstract int width();
abstract int height();

abstract void drawUnselected(Graphics g);
```

```
public void draw(Graphics g)
{
      drawUnselected(g);
      if (selected) drawSelected(g);
}

protected void drawSelected(Graphics g)
{
      Color pen = g.getColor();
      g.setColor(getBorderColor());
      drawSelectedRectangle(g,getPosition(),width(),height());
      g.setColor(pen);
}

protected void drawSelectedRectangle(Graphics g, Point upperleft, int
width, int height)
{}

protected void drawSelectedPoint(Graphics g, int x, int y)
{. . . . . }

void drawDragged(Graphics g)
{ drawDraggedAt(g,dragPosition.x,dragPosition.y); }

abstract void drawDraggedAt(Graphics g, int x, int y);

abstract void execute(DrawingEditor editor);

public void drawTip(Graphics g)
{. . . . . }

public void undo(int operator, int op1, int op2, int op3, DrawingEditor
editor)
{
      . . . . . .
      // ADD, DELETE, MOVE, RESIZE, ROTATE, SET_FILL_COLOR,
      // SET_BORDER_COLOR, SWITCH_TRANSPARENCY, SET_ORDER
}

public void redo(int operator, int[] operands, DrawingEditor editor)
{
      . . . . . .
      // MOVE_BY, INCREASE_BY, DECREASE_BY, ROTATE_BY, DELETE,
      // INVERT_FILL_COLOR, INVERT_BORDER_COLOR, SWITCH_TRANSPARENCY,
      // EXECUTE
}

}     // End Shape

class SimpleShape extends Shape
{
. . . . . .
// Assumes the older Shape class functionality, except the code moved
// to new superclass Shape, by implementing the abstract methods of its
// superclass (for instance, moveBy(…)).
```

```
. . . . . .
}     // End SimpleShape


class ShapeGroup extends Shape
{
//    Group data
private Vector shapes = new Vector(10,5);

ShapeGroup() {}

ShapeGroup(Vector v)
{
      for (int i=0;i<v.size();i++)
      {
            Shape s = (Shape)v.elementAt(i);
            shapes.addElement(s);
            s.anchorAt(this);
      }
}

public ShapeGroup getThisGroup()
{
      return this;
}

public Vector ungrouped()
{
      Vector v = new Vector(shapes.size());
      for (int i=0;i<shapes.size();i++)
      {
            Shape s = (Shape)shapes.elementAt(i);
            s.clearGroup();
            v.addElement(s);
      }
      return v;
}

public void setSize(int wi, int hei)
{
      int oldwi = width();
      int oldhei = height();
      float sx = (float)wi/(float)oldwi;
      float sy = (float)hei/(float)oldhei;
      Point pos = getPosition();
      int i;

      //    Scale all shapes
      for (i=0;i<shapes.size();i++)
      {
            Shape s = (Shape)shapes.elementAt(i);
            Point s_pos = s.getPosition();
            s.setPosition( pos.x + (int)((s_pos.x-pos.x)*sx),
                           pos.y + (int)((s_pos.y-pos.y)*sy) );
            s.setSize((int)(s.width()*sx), (int)(s.height()*sy));
      }
```

```
}

public void setPosition(int x, int y)
{
      Point po = getPosition();
      for (int i=0;i<shapes.size();i++)
            ((Shape)shapes.elementAt(i)).moveBy(x-po.x,y-po.y);
}

public Point getPosition()
{
      int posx = 1000, posy = 1000;
      for (int i=0;i<shapes.size();i++)
      {
            Shape s = (Shape)shapes.elementAt(i);
            Point po = s.getPosition();
            posx = (posx > po.x) ? po.x : posx;
            posy = (posy > po.y) ? po.y : posy;
      }
      return new Point(posx,posy);
}

protected String name() { return "Shapes group"; }
protected int numberOfShapes() { return shapes.size(); }

public void moveBy(int x, int y)
{
      for (int i=0;i<shapes.size();i++)
            ((Shape)shapes.elementAt(i)).moveBy(x,y);
}

public void increaseBy(int x, int y, DrawingEditor editor)
{
      int wi = width(), hei = height(), i, j;
      Point pos = getPosition();

      if (x>0)
      {     // Find rightmost shapes s
            for (i=0;i<shapes.size();i++)
            {
                  Shape s = (Shape)shapes.elementAt(i);
                  if (Math.abs(s.getPosition().x + s.width()
                                    - pos.x - wi) <= 1)
                        s.increaseBy(x,y,editor);
            }
      }
      else if (x<0)
      {     // Find leftmost shapes s
                  . . . . . .
                  // s.increaseBy(x,y,editor);
      }
      if (y>0)
      {     // Find bottommost shapes s
                  . . . . . .
                  // s.increaseBy(x,y,editor);
            }
```

```
        }
        else if (y<0)
        {     // Find topmost shapes s
                    . . . . . .
                    // s.increaseBy(x,y,editor);
              }
        }
}

public void decreaseBy(int x, int y, DrawingEditor editor)
{
        int wi = width(), hei = height(), i, j;
        Point pos = getPosition();

        if (x>0)
        {     // Find leftmost shapes s
                    . . . . . .
                    // s.decreaseBy(x,y,editor);
              }
        }
        else if (x<0)
        {     // Find rightmost shapes s
                    . . . . . .
                    // s.decreaseBy(x,y,editor);
              }
        }
        if (y>0)
        {     // Find topmost shapes s
                    . . . . . .
                    // s.decreaseBy(x,y,editor);
              }
        }
        else if (y<0)
        {     // Find bottommost shapes s
                    . . . . . .
                    // s.decreaseBy(x,y,editor);
              }
        }
}

public void rotateBy(int x, int y)
{
        int i, wi = width(), hei = height();
        Point pos = getPosition();

        int cx = pos.x + wi/2;
        int cy = pos.y + hei/2;
        rotateAround(cx,cy,rotateClockwise(x,y));
}

public void rotateAround(int cx, int cy, boolean right)
{
        for (int i=0;i<shapes.size();i++)
              ((Shape)shapes.elementAt(i)).rotateAround(cx,cy,right);
}
```

```
public void invertFillColor()
{
      for (int i=0;i<shapes.size();i++)
            ((Shape)shapes.elementAt(i)).invertFillColor();
}

public void invertBorderColor()
{
      for (int i=0;i<shapes.size();i++)
            ((Shape)shapes.elementAt(i)).invertBorderColor();
}

public void switchTransparency()
{
      for (int i=0;i<shapes.size();i++)
            ((Shape)shapes.elementAt(i)).switchTransparency();
}

public void equalize(int dim, Dimension po, DrawingEditor editor)
{
      switch (dim)
      {
            case EqualizeConstants.EQUALIZE_WIDTH:
                  equalizeSize(po.width,height(),editor);
                  break;
            case EqualizeConstants.EQUALIZE_HEIGHT:
                  equalizeSize(width(),po.height,editor);
                  break;
            case EqualizeConstants.EQUALIZE_BOTH:
                  equalizeSize(po.width,po.height,editor);
                  break;
      }
}

public void equalizeSize(int wi, int hei, DrawingEditor editor)
{
      int oldwi = width();
      int oldhei = height();
      float sx = (float)wi/(float)oldwi;
      float sy = (float)hei/(float)oldhei;
      Point pos = getPosition();
      int i;

      //    Scale all shapes
      for (i=0;i<shapes.size();i++)
      {
            Shape s = (Shape)shapes.elementAt(i);
            Point s_pos = s.getPosition();
            editor.record(s,MOVE,s_pos.x,s_pos.y);
            editor.record(s,RESIZE,s.width(),s.height());
            s.setPosition( pos.x + (int)((s_pos.x-pos.x)*sx),
                              pos.y + (int)((s_pos.y-pos.y)*sy) );
            s.setSize((int)(s.width()*sx), (int)(s.height()*sy));
      }
}
```

```
public Shape copy()
{
      int i;
      Vector v = new Vector(shapes.size());
      for (i=0;i<shapes.size();i++)
            v.addElement(((Shape)shapes.elementAt(i)).copy());
      ShapeGroup sg = new ShapeGroup(v);
      return sg;
}

public Color getFillColor()
{
      if (shapes.size() > 0)
            return ((Shape)shapes.firstElement()).getFillColor();
      return null;
}

public Color getBorderColor()
{
      if (shapes.size() > 0)
            return ((Shape)shapes.firstElement()).getBorderColor();
      return null;
}

public void setFillColor(Color c)
{
      for (int i=0;i<shapes.size();i++)
            ((Shape)shapes.elementAt(i)).setFillColor(c);
}

public void setBorderColor(Color c)
{
      for (int i=0;i<shapes.size();i++)
            ((Shape)shapes.elementAt(i)).setBorderColor(c);
}

public Point topleft() { return getPosition(); }

public Point topright()
      {. . . . . }
public Point bottomleft()
      {. . . . . }
public Point bottomright()
      {. . . . . }

public int width()
{
      int right = 0;
      for (int i=0;i<shapes.size();i++)
      {
            Shape s = (Shape)shapes.elementAt(i);
            Point pos = s.getPosition();
            right = Math.max(s.width() + pos.x, right);
      }
      return (right - getPosition().x);
}
```

```
public int height()
{
      int bottom = 0;
      for (int i=0;i<shapes.size();i++)
      {
            Shape s = (Shape)shapes.elementAt(i);
            Point pos = s.getPosition();
            bottom = Math.max(s.height() + pos.y, bottom);
      }
      return (bottom - getPosition().y);
}

public void undo(int operator, int op1, int op2, int op3,
                        DrawingEditor editor)
{
      . . . . . .
      // UNGROUP, REGROUP
}

public void drawUnselected(Graphics g)
{
      for (int i=0;i<shapes.size();i++)
            ((Shape)shapes.elementAt(i)).drawUnselected(g);
}

public void drawDraggedAt(Graphics g, int x, int y)
{
      for (int i=0;i<shapes.size();i++)
            ((Shape)shapes.elementAt(i)).drawDraggedAt(g,x,y);
}

public void execute(DrawingEditor editor)
{
      for (int i=0;i<shapes.size();i++)
            ((Shape)shapes.elementAt(i)).execute(editor);
}

public void parse(StringTokenizer tokenizer)
{
      String token=null;

      token = nextToken(tokenizer);
      if (token == null)
            { shapes.removeAllElements(); return; }
      int no = parseNumberOfShapes(token);
      if (no > 0)
      {
            for (int i=0;i<no;i++)
            {
                  token = nextToken(tokenizer);
                  if (token == null)
                        { shapes.removeAllElements(); return; }
                  if (token.equalsIgnoreCase("Group"))
                  {
                        ShapeGroup s = new ShapeGroup();
```

```
                        s.parse(tokenizer);
                        if (validate(s)) shapes.addElement(s);
                  }
                  else
                  {
                        int typ = SimpleShape.parseShapeType(token);
                        if (SimpleShape.validType(typ))
                        {
                              SimpleShape s = new SimpleShape();
                              s.setType(typ);
                              s.parseWithoutType(tokenizer);
                              if (validate(s)) shapes.addElement(s);
                        }
                  }
            }
      }
      token = nextToken(tokenizer);
      if (token == null)
            { shapes.removeAllElements(); return; }
      if (!token.equalsIgnoreCase("End"))
            System.out.println(
                  "WARNING : "+token+" found instead of END.");
}

protected int parseNumberOfShapes(String str)
{
      return Integer.parseInt(str);
}

private boolean validate(Shape s)
{
      if (s == null) return false;
      if (s instanceof SimpleShape)
            return (((SimpleShape)s).size >= 2);
      else
            return (((ShapeGroup)s).numberOfShapes() > 0);
}

public String toString()
{
      String str = new String("GROUP "+shapes.size());
      for (int i=0;i<shapes.size();i++)
      {
            Shape s = (Shape)shapes.elementAt(i);
            str += ("\n"+s.toString());
      }
      str += "\nEND";
      return str;
}

}     // End ShapeGroup
```

Chapter 3 Adventure games

3.1 Dynamic visualization

Dynamic visualization is the visualization of dynamic systems, i.e. systems whose state evolves with time, either thanks to an internal process or through continuous flow of external data. Dynamic visualization applications comprise ***visualization components*** to show part of the state of the dynamic system either in its entirety or just specific paremeters or views of it. In most cases, such applications also comprise components to ***manipulate and control*** the parameters and the state of the system, together with components to ***gather and present data*** of statistical or other nature.

From a programmer's point of view, such an application uses one or more Threads that operate and execute independently of the user and that implement the behavior of the dynamic system. In some cases, the user may be given some limited possibility to control the dynamic process, for instance to start or stop a simulation.

3.2 Application to adventure games

As far as operational specifications are concerned, most adventure computer games belong to the class of dynamic visualization with minimal requirements for parameter manipulation and data gathering. The user takes active part in the dynamic process by assuming a certain role, and the dynamic nature of the game is evident in the user's interaction with opponents or enemies acting against the user's interests and independently of him. Usually, enemies in such a game chase the player (user) or shoot at him in some way.

In what follows, we will analyze the main features of an adventure game based on a simple example, the "Key game" or "Game of the locked room", in which the user has to grasp a key and unlock a door to escape from an enemy. A snapshot of the game is given in figure 3.1.

3.2.1 World representation

The world (environment) in which the hero (player) and its enemy move is a limited 2D area, that may be represented as a 2D array, having in its position a value that shows the type of object occupying the corresponding place of the world. For visualization purposes a drawing scale is used (here 12x12 pixels), otherwise each object would have to be represented as a single point in space.

In our example game, a few general adventure game classes have been implemented first (classes AdventureGame, AdventureGameAgent and Enemy which is a subclass of AdventureGameAgent), while the actual classes for the Key game subclass the former ones.

The above are implemented as follows * :

```
class AdventureGame extends Applet implements KeyListener, MouseListener
{
static int WIDTH = 30, HEIGHT = 30;
public int world[][] = new int[WIDTH][HEIGHT];
static Point scale = new Point(12,12);
Point offset() {return new Point(0,60);}

final static int OBSTACLE = 20;
```

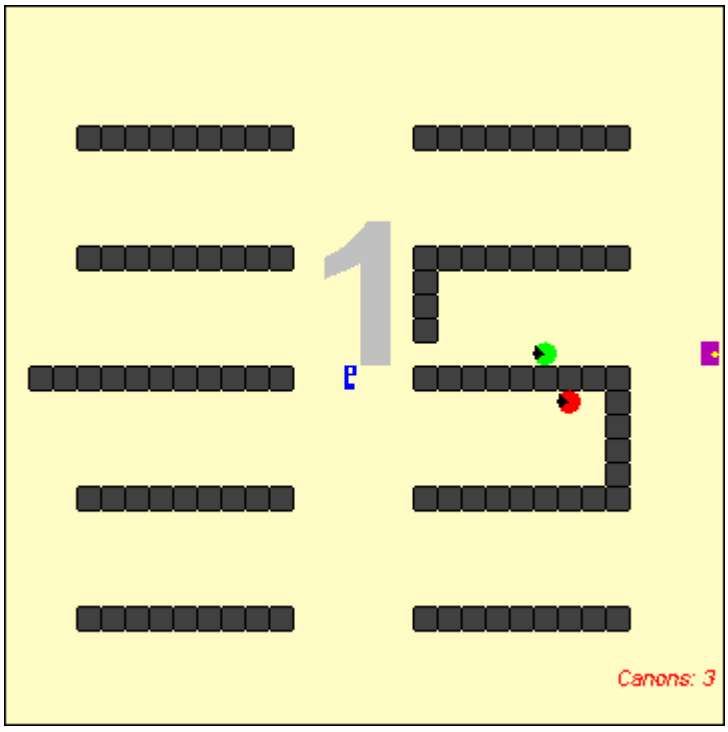

Figure 3.1. A snapshot of the key game.

```
// Special initialization in each subclass (e.g. enemies initialization)
public void initSpecial() {}
// World initialization
public void initWorld() {}

public void init()
{
      initLayout();
      setSize(WIDTH * scale.x + offset().x,
                  HEIGHT * scale.y + offset().y);
      initSpecial();
      initGame();
      addKeyListener(this);
      addMouseListener(this);
      requestFocus();
}
```

```
// Eventually, other graphical components, for instance buttons
public void initLayout() {}

public void paint(Graphics g) {paintDefault(g);}

public void paintDefault(Graphics g)
{
String str;
Color pen = g.getColor();

//    Draw borders
      g.setColor(BackgroundColor());
      g.fillRect(0,0, WIDTH * scale.x, HEIGHT * scale.y);
      g.setColor(pen);
      g.drawRect(0,0, (WIDTH * scale.x) - 1, (HEIGHT * scale.y) - 1);

//    The game terrain
      for (int i=0;i<WIDTH;i++)
      for (int j=0;j<HEIGHT;j++)
            drawPlace(g,i,j);
}

void initGame()
{
      initWorld();
}

void initDefault()
{
      for (int i=0;i<WIDTH;i++)
      for (int j=0;j<HEIGHT;j++)
            world[i][j] = 0;
}

public void drawPlace(Graphics g, int x, int y)
{
// To implement in subclasses
}

boolean freePlace(int x, int y)
{
      if ((x<0) || (x>=WIDTH) || (y<0) || (y>=HEIGHT)) return false;
      return (world[x][y] == 0);
}

boolean validPlace(int x, int y)
{
      if ((x<0) || (x>=WIDTH) || (y<0) || (y>=HEIGHT)) return false;
      return true;
}

}     // End AdventureGame

public class KeyGame extends AdventureGame
{
//    Hero and monster of the game
```

```
KeyHero hero;
KeyMonster monster;

//    Type constants
final static int HERO = 5;
final static int KEY = 6;
final static int DOOR = 7;
final static int MONSTER = 8;

Point offset() {return new Point(0,80);}

public void initSpecial()
{
      hero = new KeyHero(this);
      monster = new KeyMonster(this);
}

public void initWorld()
{
      int i,j;

      initDefault();
//    Init hero
      hero.init(1,29);
      world[1][29] = HERO;
//    Init monster
      monster.init(1,1,120);
      world[1][1] = MONSTER;
//    Init obstacles
      for (j=5;j<=25;j+=5)
      {
            for (i=3;i<12;i++)
            {
                  world[i][j] = OBSTACLE;
                  world[14+i][j] = OBSTACLE;
            }
      }
      for (i=1;i<5;i++) world[i][15] = OBSTACLE;
      for (j=15;j<=20;j++) world[25][j] = OBSTACLE;
      world[17][11] = OBSTACLE; world[17][12] = OBSTACLE;
      world[17][13] = OBSTACLE;
//    Init key
      world[14][15] = KEY;
//    Init door
      world[29][14] = DOOR;
}

void initGame()
{
      super.initGame();
      monster.start();
}

public void paint(Graphics g)
{
Color pen = g.getColor();
```

```
Font font = g.getFont();
FontMetrics metrics;
String str;

      paintDefault(g);
      hero.draw(g);
      monster.draw(g);
}

public void drawPlace(Graphics g, int x, int y)
{
      switch (world[x][y])
      {
            case OBSTACLE: drawObstacle(g,x,y); break;
            case KEY: drawKey(g,x,y); break;
            case DOOR: drawDoor(g,x,y); break;
            default: break;
      }
}

void drawObstacle(Graphics g, int x, int y)
{. . . . . }

void drawKey(Graphics g, int x, int y)
{. . . . . }

void drawDoor(Graphics g, int x, int y)
{. . . . . }

boolean keyAt(int x, int y)
{
      if (validPlace(x,y)) return (world[x][y] == KEY);
      return false;
}

boolean doorAt(int x, int y) { /* Similarly */ }

boolean heroAt(int x, int y) { /* Similarly */ }

boolean monsterAt(int x, int y) { /* Similarly */ }

void putKeyAt(int x, int y)
{
      world[x][y] = KEY;
}

boolean putDoorAt(int x, int y) { /* Similarly */ }

boolean putHeroAt(int x, int y) { /* Similarly */ }

boolean putMonsterAt(int x, int y) { /* Similarly */ }

void removeAt(int x, int y)
{
      world[x][y] = 0;
}
```

```
}     // End KeyGame
```

3.2.2 Motion

To visualize the motion of the player and the enemy, we use a heading variable whose different values (up, down, right, left) correspond to different images for the player and enemy, respectively. Some games use additional animation techniques to animate particular actions, for example in a Pacman game the hero is seen to open and close its mouth quickly (to swallow dots). Moving a player to a certain position in the grid world may have varying consequences, depending on the specificities of the game and the type of the player (hero or enemy), see method doSpecialAt().

The above are implemented as follows [*] :

```
abstract class AdventureGameAgent
{
AdventureGame game;

Point position = new Point(1,1);
int heading = Event.RIGHT;

abstract Color color();
abstract public void putAt(int x, int y);
abstract public void removeFrom(int x, int y);
abstract public void doSpecialAt(int x, int y);
Color headingColor() {return null;}

public void init(int x, int y)
{
      position.x = x;
      position.y = y;
}

public void draw(Graphics g)
{
      Color pen = g.getColor();
      Color c = headingColor();

      g.setColor(color());
      g.fillOval(position.x*game.scale.x, position.y*game.scale.y,
            game.scale.x, game.scale.y);
      if (c != null) g.setColor(c);
            else g.setColor(pen);
      int angle = computeHeadingStartAngle();
      g.fillArc(position.x*game.scale.x, position.y*game.scale.y,
                  game.scale.x, game.scale.y, angle, 70);
      g.setColor(pen);
}

int computeHeadingStartAngle()
{
      switch (heading)
```

```
	{
		case KeyEvent.VK_UP:
			return 55;
		case KeyEvent.VK_DOWN:
			return -125;
		case KeyEvent.VK_LEFT:
			return 145;
		case KeyEvent.VK_RIGHT:
			return -35;
		default:
			return 60;
	}
}

public void moveBy(int direction)
{
	heading = direction;
	game.repaint();
	Point step = getStepForHeading();
	Point targetPosition = new Point(position.x + step.x,
					position.y + step.y);
	if (!game.validPlace(targetPosition.x,targetPosition.y)) return;
	if (game.freePlace(targetPosition.x,targetPosition.y))
	{
		gotoposition(targetPosition.x,targetPosition.y);
	}
	// Do something special when the target place is not empty
	else doSpecialAt(targetPosition.x,targetPosition.y);
}

public void gotoposition(int x, int y)
{
	removeFrom(position.x,position.y);
	setPosition(x,y);
	putAt(x,y);
}

Point getStepForHeading()
{
	switch (heading)
	{
		case KeyEvent.VK_UP: return new Point(0,-1);
		case KeyEvent.VK_DOWN: return new Point(0,1);
		case KeyEvent.VK_RIGHT: return new Point(1,0);
		case KeyEvent.VK_LEFT: return new Point(-1,0);
		default: return new Point(0,-1);
	}
}

}	// End AdventureGameAgent

class KeyHero extends AdventureGameAgent
{
boolean hasKey = false;
boolean alive = true;
boolean won = false;
```

```
public void init(int x, int y)
{
      super.init(x,y);
      hasKey = false;
      alive = true;
      won = false;
}

}     // End KeyHero
```

3.2.3 Enemies

In our example game, the enemy has the goal to chase the hero and when they both coincide in the same place the hero is automatically killed. A very simple chasing algorithm for the enemy consists in moving to the neighboring place closest to the hero with a small probability of moving to a random neighboring place. This statistical component allows the enemy to avoid being trapped forever in some geometrically challenging places, but instead to escape after some time.

The above are implemented as follows [*] :

```
abstract class Enemy extends AdventureGameAgent implements Runnable
{
Thread engine;
int sleepTime = 1000;
boolean threadControlVariable = false;

public void init(int x, int y, int s)
{
      super.init(x,y);
      sleepTime = s;
}

public void run()
{
      while (game.runnable())
      {
            if (threadControlVariable)
            {
                  doRun();
                  game.repaint();
            }
            try {engine.sleep(sleepTime);}
            catch (InterruptedException e) {}
      }
}

abstract public void doRun();
public void start()
{
      if (engine == null)
      {
```

```
            engine = new Thread(this);
            engine.start();
            threadControlVariable = true;
      }
      else if (!engine.isAlive())
      {
            engine = new Thread(this);
            engine.start();
            threadControlVariable = true;
      }
}

public void kill()
{
      if (engine != null)
      {
            if (engine.isAlive()) threadControlVariable = false;
            engine = null;
      }
}

}     // End Enemy


abstract class AdventureGameAgent
{
. . . . . .
int findDirectionTo(int x, int y)
      {return findCorrectDirectionTo(x,y);}

int findPerturbedDirectionTo(int x, int y, double probability)
{
//    Find the direction to a given target point (x,y).
//    With a given probability, return a random direction

      if (Math.random() < probability)
      {
            switch ((int)(Math.round((Math.random() * 4) - 0.5)))
            {
                  case 0: return KeyEvent.VK_DOWN;
                  case 1: return KeyEvent.VK_UP;
                  case 2: return KeyEvent.VK_RIGHT;
                  case 3: return KeyEvent.VK_LEFT;
                  default: return KeyEvent.VK_DOWN;
            }
      }
      else return findCorrectDirectionTo(x,y);
}

int findCorrectDirectionTo(int x, int y)
{
//    Find the direction to a given target point (x,y).

      int dx = x - position.x;
      int dy = y - position.y;
```

```
      if (Math.abs(dx) < Math.abs(dy))
            return ( (dy>0) ? KeyEvent.VK_DOWN : KeyEvent.VK_UP );
      else
            return ( (dx>0) ? KeyEvent.VK_RIGHT : KeyEvent.VK_LEFT );
}

}     // End AdventureGameAgent

class KeyMonster extends Enemy
{
KeyMonster(KeyGame g)
{
      super((AdventureGame)g);
}

public void removeFrom(int x, int y)
      {((KeyGame)game).removeAt(x,y);}

public void putAt(int x, int y)
      {((KeyGame)game).putMonsterAt(x,y);}

public void doSpecialAt(int x, int y)
{
      if (((KeyGame)game).heroAt(x,y))
      {
            gotoposition(x,y);
            ((KeyGame)game).killHero();
      }
}

public void doRun()
{
//    Chase hero.
      int dir = findDirectionTo(
                        ((KeyGame)game).hero.position.x,
                        ((KeyGame)game).hero.position.y);
      moveBy(dir);
}

int findDirectionTo(int x, int y)
      {return findPerturbedDirectionTo(x,y,0.1);}

}     // End KeyMonster


public class KeyGame extends AdventureGame
{
. . . . . .
public boolean runnable()
{
      return (hero.alive && (!hero.won));
}

public void killHero()
{
      hero.alive = false;
```

```
      repaint();
}

}     // End KeyGame
```

3.2.4 Player actions

The user may move the hero with the arrows (right, left, up or down), grasp the key with ‘g’ or ‘G’, drop the key with ‘d’ or ‘D’ and finally unlock the door with ‘b’ or ‘B’. The hero gets killed when he runs into his enemy. When the game is over, either because the hero won or got killed, the user may restart it with ‘s’ or ‘S’ or with a mouse click.

The above are implemented as follows * :

```
abstract class AdventureGameAgent
{
. . . . . .
Point nextPlace()
{
      switch (heading)
      {
            case KeyEvent.VK_UP: return new Point(position.x,
                                          position.y-1);
            case KeyEvent.VK_DOWN: return new Point(position.x,
                                          position.y+1);
            case KeyEvent.VK_RIGHT: return new Point(position.x+1,
                                          position.y);
            case KeyEvent.VK_LEFT: return new Point(position.x-1,
                                          position.y);
            default: return new Point(position.x,position.y-1);
      }
}

}     // End AdventureGameAgent

public class KeyGame extends AdventureGame
{
. . . . . .
public void doEnd()
{
      hero.won = true;
}

public void keyPressed(KeyEvent evt)
{
      char key = evt.getKeyChar();
      int keycode = evt.getKeyCode();

      if ((!hero.alive) || (hero.won))
      {
      //    Press 's' or 'S' to restart
            if ((key == 's') || (key == 'S'))
            {
                  init();
```

```
                  repaint();
            }
      }

      //    Press 'g' or 'G' to grasp the key
      else if ((key == 'g') || (key == 'G'))
      {
            hero.graspKey();
            repaint();
      }

      //    Press 'd' or 'D' to drop the key
      else if ((key == 'd') || (key == 'D'))
      {
            hero.dropKey();
            repaint();
      }

      //    Press 'b' or 'B' to unlock the door
      else if ((key == 'b') || (key == 'B'))
      {
            hero.openDoor();
            repaint();
      }

      else switch (keycode)
      {
            case KeyEvent.VK_LEFT:
            case KeyEvent.VK_RIGHT:
            case KeyEvent.VK_DOWN:
            case KeyEvent.VK_UP:
                  hero.moveBy(keycode);
                  repaint();
                  break;
            default:
                  break;
      }
      return;
}

public void mousePressed(MouseEvent evt)
{
      if ((!hero.alive) || (hero.won))
      {
            initGame();
            repaint();
      }
      return;
}

}     // End KeyGame

class KeyHero extends AdventureGameAgent
{
. . . . . .
public void graspKey()
```

```
{
      Point target = nextPlace();
      if (((KeyGame)game).keyAt(target.x,target.y))
      {
            hasKey = true;
            ((KeyGame)game).removeAt(target.x,target.y);
      }
}

public void dropKey(){ /* Similarly */ }

public void openDoor()
{
      Point target = nextPlace();
      if (hasKey && (((KeyGame)game).doorAt(target.x,target.y)))
      {
            levelDone = true;
            game.repaint();
            game.gotoNextLevel();
      }
}

public void doSpecialAt(int x, int y)
{
      if (((KeyGame)game).monsterAt(x,y))
      {
            gotoposition(x,y);
            ((KeyGame)game).killHero();
      }
}

}     // End KeyHero
```

3.2.5 Additional representational issues

Additional information usually represented in adventure games is information about level of game difficulty, number of lives ("canons") left to the hero and eventually a score. Moreover, when the game is over, either because the hero won or got killed, a final message is shown, usually a "Game Over" message.

In our example editor the level of difficulty is represented in the middle of the world and the number of canons in the bottom right end. Finally, a "Game Over" or a "Congratulations!" message is shown, according to whether the hero lost or won respectively, as well as the message "Press 's' or click to restart" (see figure 3.2).

The above are implemented as follows * :

```
class AdventureGame extends Applet implements KeyListener, MouseListener
{
. . . . . .
static int MAX_CANONS = 3;
static int MAX_LEVEL = 3;
int level=1, canons=MAX_CANONS;
```

Figure 3.2. When you have lost in the simple Key Game.

```
public void initWorld1() {}
public void initWorld2() {}
public void initWorld3() {}

void initGame()
{
      level = 1; canons = MAX_CANONS;
      initWorld(level);
}
public void initWorld(int lev)
{
      switch (lev)
      {
            case 1: initWorld1(); break;
            case 2: initWorld2(); break;
            case 3: initWorld3(); break;
            default: initWorld3(); break;
      }
}

public void paintDefault(Graphics g)
{
String str;
FontMetrics metrics;

Color pen = g.getColor();
Font font = g.getFont();

//    Draw borders
```

```
      g.setColor(BackgroundColor());
      g.fillRect(0,0, WIDTH * scale.x, HEIGHT * scale.y);
      g.setColor(pen);
      g.drawRect(0,0, (WIDTH * scale.x) - 1, (HEIGHT * scale.y) - 1);

//    Draw level
      g.setColor(LevelColor);
      g.setFont(LevelFont);
      metrics = g.getFontMetrics();
      str = ""+level;
      g.drawString(str,
                  ((WIDTH * scale.x) - metrics.stringWidth(str))/2,
                  ((HEIGHT * scale.y) /*- metrics.getHeight()*/)/2);

//    Draw canons
      g.setColor(CanonsColor);
      g.setFont(CanonsFont);
      metrics = g.getFontMetrics();
      str = "Canons: "+canons;
      g.drawString(str,
                  (WIDTH * scale.x)-metrics.stringWidth(str)-5,
                  (HEIGHT * scale.y)-metrics.getHeight()-5);

      g.setColor(pen);
      g.setFont(font);

//    The game terrain
      for (int i=0;i<WIDTH;i++)
      for (int j=0;j<HEIGHT;j++)
            drawPlace(g,i,j);
}

void gotoNextLevel()
{
      if (level == MAX_LEVEL) doEnd();
      else
      { level++; initWorld(level); }
      repaint();
}

void restartLevel()
{
      initWorld(level); repaint();
}

public boolean runnable() {return (canons > 0);}

}     // End AdventureGame

public class KeyGame extends AdventureGame
{
. . . . . .
//    Colors and fonts used
static Font GameOverFont=new Font("TimesRoman",Font.BOLD,60);
static Color GameOverColor = Color.red;
static Font CongratulationsFont=new Font("TimesRoman",Font.BOLD,24);
```

```
static Color CongratulationsColor = Color.blue;
static Font RestartFont=new Font("TimesRoman",Font.ITALIC,12);

public void paint(Graphics g)
{
Color pen = g.getColor();
Font font = g.getFont();
FontMetrics metrics;
String str;

      paintDefault(g);
      hero.draw(g);
      monster.draw(g);

//    Check hero dead
      if (!hero.alive)
      {
            g.setColor(GameOverColor);
            g.setFont(GameOverFont);
            metrics = g.getFontMetrics();
            str = "Game Over";
            g.drawString(str,
                        ((WIDTH * scale.x)-metrics.stringWidth(str))/2,
                        ((HEIGHT * scale.y)-metrics.getHeight())/2);
            g.setFont(CongratulationsFont);
            g.setFont(RestartFont);
            metrics = g.getFontMetrics();
            str = "Press 'S' or click to restart";
            g.drawString(str,
                        ((WIDTH * scale.x)-metrics.stringWidth(str))/2,
                        (HEIGHT * scale.y)/2+metrics.getHeight());
            g.setColor(pen);
            g.setFont(font);
      }

//    Check hero won
      else if (hero.won)
      {
            g.setColor(CongratulationsColor);
            g.setFont(CongratulationsFont);
            metrics = g.getFontMetrics();
            str = "Congratulations ! You won !";
            g.drawString(str,
                        ((WIDTH * scale.x)-metrics.stringWidth(str))/2,
                        (HEIGHT * scale.y)/2);
            g.setFont(RestartFont);
            metrics = g.getFontMetrics();
            str = "Press 'S' or click to restart";
            g.drawString(str,
                        ((WIDTH * scale.x)-metrics.stringWidth(str))/2,
                        (HEIGHT * scale.y)/2+metrics.getHeight());
            g.setColor(pen);
            g.setFont(font);
      }
}
```

```
}      // End KeyGame
```