Preface

Early computer graphics started as a research and application field that was the domain of only a few experts, for instance in the area of computer aided design (CAD). Nowadays, any person using a personal computer benefits from the developments in computer graphics. Operating systems and application programs with graphical user interfaces (GUIs) belong to the simplest applications of computer graphics. Visualisation techniques, ranging from simple histograms to dynamic 3D animations showing changes of winds or currents over time, use computer graphics in the same manner as popular computer games. Even those who do not use a personal computer might see the results of computer graphics on TV or in cinemas where parts of scenes or even a whole movie might be produced by computer graphics techniques.

Without powerful hardware in the form of fast processors, sufficiently large memory and special graphics cards, most of these applications would not have been possible. In addition to these hardware requirements efficient algorithms as well as programming tools that are easy to use and flexible at the time are required. Nowadays, a standard personal computer is sufficient to generate impressive graphics and animations using freely available programming platforms like OpenGL or Java 3D. In addition to at least an elementary understanding of programming, the use of such platforms also requires basic knowledge about the underlying background, concepts and methods of computer graphics.

Aims of the book

The aim of this book is to explain the necessary background and principles of computer graphics combined with direct applications in concrete and simple examples. Coupling the theory with the practical examples enables the reader to apply the technical concepts directly and to visually understand what they mean.

Java 2D and Java 3D build the basis for the practical examples. Wherever possible, the introduced concepts and theory of computer graphics are immediately followed by their counterparts in Java 2D and Java 3D. However, the intention of this book is not to provide a complete introduction to Java 2D or Java 3D, which would both need a multivolume edition themselves without even touching the underlying theoretical concepts of computer graphics.

In order to directly apply computer graphics concepts introduced in this book, the book focusses on the parts of Java 2D and Java 3D that are absolutely relevant for these concepts. Sometimes a simple solution is preferred over the most general one so that not all possible options and additional parameters for an implementation will be discussed. The example programs are kept as simple as possible in order to concentrate on the important concepts and not to disguise them in complex, but more impressive scenes.

There are some selected additional topics—for instance the computation of shadows within computer graphics—that are introduced in the book, although Java 3D does not provide such techniques yet.

Why Java?

There are various reasons for using Java 2D and Java 3D as application platforms. The programming language Java becomes more and more popular in applications and teaching so that extensions like Java 2D/3D seem to be the most obvious choice. Many universities use Java as the introductory programming language, not only in computer science, but also in other areas so that students with a basic knowledge in Java can immediately start to work with Java 2D/3D. Specifically, for multimedia applications Java is very often the language of first choice.

Overview

The first chapters of the book focus on aspects of two-dimensional computer graphics like how to create and draw lines, curves and geometric shapes, handling of colours and techniques for animated graphics.

Chapter 5 and all following chapters cover topics of three-dimensional computer graphics. This includes modelling of 3D objects and scenes, producing images from virtual 3D scenes, animation, interaction, illumination and shading. The last chapter introduces selected special topics, for example special effects like fog, sound effects and stereoscopic viewing.

Guidelines for the reader

In order to be able to apply the computer graphics concepts introduced in this book, the reader will need only very elementary knowledge of the programming language Java. The example programs in this book use Java 3D but also Java 2D in the first chapters, since two-dimensional representations are essential for computer graphics and the geometrical concepts are easier to understand in two dimensions than in three. The necessary background of Java 2D and Java 3D is included as application sections in this book.

Although the coupling of theory and practice was a main guideline for writing this book, the book can also be used as an introduction to the general concepts of computer graphics without focussing on specific platforms or learning how to use Java 2D or Java 3D. Skipping all sections and subsections containing the word "Java" in their headlines, the book will remain completely self-contained in the sense of a more theoretical basic introduction to computer graphics. For some of the computer graphics concepts introduced in this book it is assumed that the reader has basic knowledge about vectors, matrices and elementary calculus.

Supplemental resources

Including the complete source code of all mentioned example programs would have led to a thicker, but less readable book. In addition, no one would like to take the burden of typing the source code again in order to run the examples. Therefore, the book itself only contains those relevant excerpts of the source code that are referred to in the text. The complete source code of all example programs and additional programs can be downloaded from the book web site at

http://public.rz.fh-wolfenbuettel.de/∼klawonn/computergraphics

This online service also provides additional exercises concerning the theoretical background as well programming tasks including sketches of solutions, teaching material in the form of slides and some files that are needed for the example programs. The links mentioned in the appendix and further links to some interesting web sites can also be found at the online service of this book.

Acknowledgements

Over the years, the questions, remarks and proposals of my students had a great influence on how this book was written. I cannot list all of them by name, but I would like to mention at least Daniel Beier, Thomas Weber, Jana Volkmer and especially Dave Bahr for reading the manuscript and their extremely helpful

comments. I also would like to thank Katharina Tschumitschew and Gerry Gehrmann for designing the online service of the book and for some 3D models that I could use in my programs. The book was first published in German and without the encouragement and support of Catherine Brett from Springer Verlag in London this English version would have been impossible. Thanks also to Frank Ganz from Springer, who seems to know everything about LaTeX. My very personal thanks go to my parents and my wife Keiko for their love and for always accepting my sometimes extremely heavy overload of work.

Wolfenbüttel Frank Klawonn
September 2007

2
Basic principles of two-dimensional graphics

This chapter introduces basic concepts that are required for the understanding of two-dimensional graphics. Almost all output devices for graphics like computer monitors or printers are pixel-oriented. Therefore, it is crucial to distinguish between the representation of images on these devices and the model of the image itself which is usually not pixel-oriented, but defined as scalable vector graphics, i.e., floating point values are used for coordinates.

2.1 Raster versus vector graphics

Before an object can be shown on a computer monitor or a printer, a model describing the object's geometry is required, unless the object is an image itself. Modelling of geometrical objects is usually done in the framework of *vector-oriented* or *vector graphics*. A more complex object is modelled as a combination of elementary objects like lines, rectangles, circles, ellipses or arcs. Each of these elementary objects can be defined by a few coordinates, describing the location of the object, and some parameters like the radius for a circle. A very simple description of the house in figure 2.1(a) in terms of vector graphics is shown in figure 2.1(b). The house can be defined as a sequence of points or vectors. It must also be specified within the sequence of points whether two neighbouring points should be connected by a line or not. Dotted lines in figure

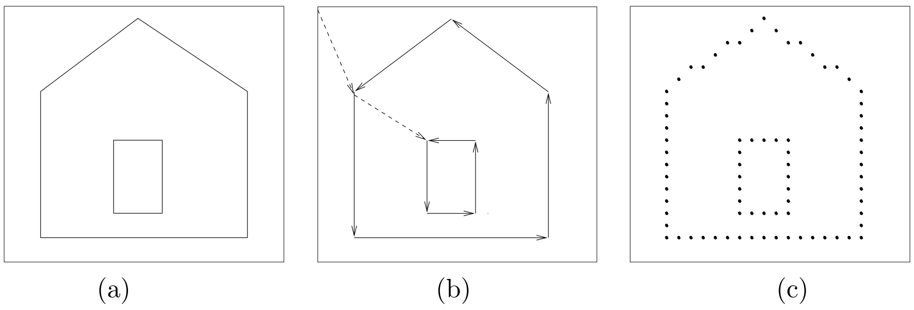


Figure 2.1 Original image, vector and pixel graphics

2.1(b) refer to points in the sequence that should not be connected by a line.

The vector graphics-oriented description of objects is not directly suitable for the representation on a purely pixel-oriented device like an LCD monitor or printer. From a theoretical point of view, it would be possible to display vector graphics directly on a CRT[1] monitor by running the cathode ray—or, in case of colour display, the three cathode rays—along the lines defined by the sequence of points and switch the ray on or off, depending on whether the corresponding connecting line should be drawn. In this case, the monitor might not be flicker free anymore since the cathode ray might take too long to refresh the screen for a more complex image in vector graphics, so that fluorescent spots on the screen might fade out, before the cathode ray returns. Flicker-free monitors should have a refresh rate of 60 Hz. If a cathode ray were to run along the contour lines of objects represented in vector graphics, the refresh rate would depend on how many lines the objects contain, so that a sufficiently fast refresh rate could not be guaranteed in this operational mode. Therefore, the cathode ray scans the screen line by line leading to a guaranteed and constant refresh rate, independent of the image to be drawn.

Computer monitors, printers and also various formats for storing images like bitmaps or JPEG are based on *raster* or *raster-oriented graphics*, also called *pixel* or *pixel-oriented graphics*. Raster graphics uses a pixel matrix of fixed size. A colour can be assigned to each pixel of the raster. In the simplest case of a black-and-white image a pixel takes one of the two values black or white.

In order to display vector-oriented graphics in the form of raster graphics, all geometrical shapes must be converted into pixels. This procedure is called *scan conversion*. On the one hand, this can lead to high computational efforts. A standard monitor has more than one million pixels. For each of them, it must be decided which colour to assign to it for each image. On the other hand, undesired *aliasing effects* occur in the form of jagged edges, known as

[1] Cathode ray tube.

jaggies or *staircasing*. The term *aliasing effect* originates from the field of signal processing and refers to artifacts, i.e., superficial undesired effects that can occur, when a discrete sampling rate is used to measure a continuous signal. A grey-scale image can be viewed as a two-dimensional signal. In this sense, a coloured image based on the three colours red, green and blue, is nothing else than three two-dimensional signals, one for each colour.

Even if an image will be displayed in terms of raster-oriented graphics, it still has advantages to model and store it in a vector-oriented format. Raster graphics is bound to a specific resolution. Once the resolution is fixed, the full information contained in the vector-oriented image cannot be recovered anymore, leading to serious disadvantages, when the image is displayed on a device with a different resolution or when the image needs to be enlarged or scaled down. Figure 2.2 shows the tip of an arrow and its representation in the form of raster graphics for two different resolutions. If only the more coarse pixel image in the middle is stored, it is impossible to reconstruct the refined pixel image on the right-hand side without additional information. One could only produce an image appearing in the same form as the one in the middle by simply identifying four pixels of the refined image with one pixel in the coarser image. If the quotient of the pixel resolution is not an integer number, the transformation from a raster graphics with one resolution to a raster graphics with another resolution becomes even more complicated and will lead to new aliasing effects, even if the new resolution is higher than the original one.

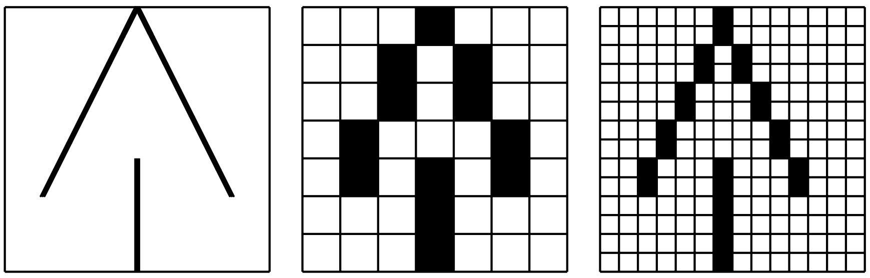


Figure 2.2 The tip of an arrow drawn as raster graphics in two different resolutions

In most cases, when a pixel matrix is considered in this book, each pixel is represented by a square between the lines of a grid as shown in figure 2.2. However, sometimes another representation is more convenient where pixels are illustrated as circles on the points where the lines of the grid cross. Figure 2.3 shows the pixel with the grid coordinates (5,3).

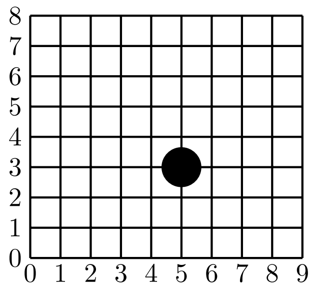


Figure 2.3 An alternative representation for pixels

2.2 The first Java 2D program

Before modelling of two-dimensional objects is discussed in more detail, a short introduction into how Java 2D can be used to generate images in general is provided. The first chapters of this book dealing exclusively with problems and questions of two-dimensional graphics refer to Java 2D. Chapter 5 and the latter chapters will use Java 3D for three-dimensional modelling, animation and representations.

It is not the aim of this book to provide a complete introduction to Java 2D and Java 3D. Instead, the main intention of this book is to enable even those readers with only very basic knowledge in Java to use and apply the more theoretical concepts of computer graphics immediately within the framework of Java 2D and Java 3D. For this reason, the example programs are kept as simple as possible and not all available options and settings will be explained in detail, in order to better focus on the main aspects and concepts. For readers who are already more familiar with Java programming the book provides an introduction to Java 2D and 3D that enables the reader to study the more advanced options and possibilities of these two Application Programming Interfaces (APIs) with the help of specific literature and the API documentations.

Detailed information concerning Java 2D can be found in books like [24, 29], in the API documentation and the Java tutorial, that are available on the Internet (see the appendix).

Java 2D is an API belonging to the kernel classes of the Java 2 (formerly JDK 1.2) and later platforms so that it is not necessary to carry out additional installations to use Java 2D classes, as long as a Java platform is installed on the computer.

Java 2D extends some of the AWT[2] packages of Java by additional classes and also introduces new packages within AWT. Java 2D can be viewed as a

[2] Abstract Windowing Toolkit.

component under Java's graphics components AWT and Swing (see figure 2.4).

Although AWT is seldom used anymore, the introductory examples for Java 2D in this book are based on AWT. The reason is that within AWT it is easily possible to program simple animations without the technique of double buffering that will be used later on in this book.

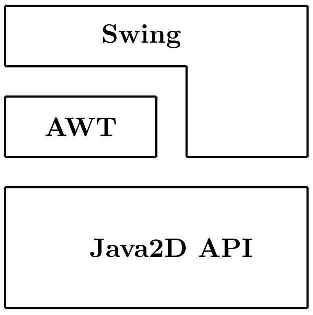
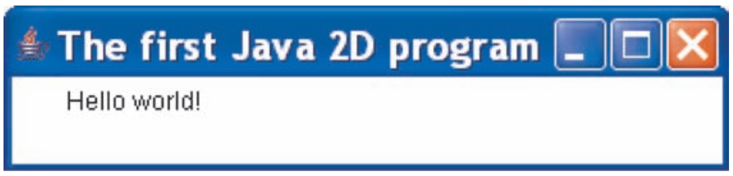


Figure 2.4 The Java 2D API extends AWT

AWT components that are displayed on the computer screen contain a `paint` method with a `Graphics` object as its argument. In order to use the facilities of Java 2D for the corresponding AWT component, it is necessary to cast this `Graphics` object into a `Graphics2D` object. The class `Graphics2D` within Java 2D extends the class `Graphics`. The following simple Java class `SimpleJava2DExample.java` demonstrates this simple casting procedure. In order to keep the printed code examples short, comments are only included in the programs that can be downloaded from the web site of this book, but not in the printed versions. The result of this program is shown in figure 2.5.



```
Hello world!
```

Figure 2.5 The result of the first Java 2D program

```
import java.awt.*;

public class SimpleJava2DExample extends Frame
{
  SimpleJava2DExample()
  {
    addWindowListener(new MyFinishWindow());
  }

  public void paint(Graphics g)
  {
    Graphics2D g2d = (Graphics2D) g;
    g2d.drawString("Hello world!",30,50);
  }

  public static void main(String[] argv)
  {
    SimpleJava2DExample f = new SimpleJava2DExample();
    f.setTitle("The first Java 2D program");
    f.setSize(350,80);
    f.setVisible(true);
  }
}
```

The method `addWindowListener`, called in the constructor, enables the closing of the window by clicking on the cross in the upper right corner. The method uses a simple additional class `MyFinishWindow.java`, that can also be downloaded from the web site of this book. The `main` method generates the corresponding window, defines the title of the window, determines its size by 350 pixels in width and 80 pixels in height and finally displays it. This structure of the `main` method will be used for all Java 2D examples in this book. For other programs, it will only be necessary to replace `SimpleJava2DExample` by the corresponding class name and—if desired—to change the title of the window and its size.

The image or graphics to be displayed is defined within the `paint` method. The first line of this method will always be the same in all examples here: It carries out the casting of the `Graphics` object to a `Graphics2D` object. The remaining code lines in the `paint` method depend on what is to be displayed and will be different for each program. In the example here, only the text "Hello worl" is printed at the window coordinates (30,50).

When specifying window coordinates, the following two aspects should be taken into account.

– The point (0,0) is located in the upper left corner of the window. The window extends to the right (in the example program 350 pixels) and downwards (in the example program 80 pixels). This means that the y-axis of the coordinate system does not point upwards, but downwards since the pixel lines in the window are counted from the top to the bottom. How to avoid this problem of an inverted y-axis will be explained later on.

– The window includes margins on all its four sides. Especially the upper margin, containing the title of the window, is quite broad. It is not possible to draw anything on or outside these margins within the `paint` method. Trying to draw an object on the margin or outside the window will not lead to an error or exception. The clipping procedure will simply make sure that the object or its corresponding part is not drawn. Therefore, when a window of a size of 350×80 pixels is defined as in the example program, a slightly smaller area is available for drawing. The width of the margins depends on the operating system platform. The example programs avoid this problem by defining a window that is large enough and by not drawing objects too close to any of the margins. The exact width of the margins can also be determined within the class, for instance within the `paint` method using

```
Insets ins = this.getInsets();
```

The width of the left, right, upper and lower margin in pixels is given by `ins.left`, `ins.right`, `ins.top` and `ins.bottom`, respectively.

The first example of a Java 2D program did not require any additional computations before the objects—in this case only text—could be drawn. For real graphics it is usually necessary to carry out more or less complicated computations in order to define and position the objects to be displayed. Java 2D distinguishes between the definition of objects and drawing objects. An object that has been defined will not be drawn or shown in the corresponding window, until a `draw`- or `fill` method is called with the corresponding object as argument. Therefore, Java 2D also differentiates between modelling objects based on vector graphics using floating point arithmetics and displaying or drawing objects on the screen based on raster graphics with scan conversion and integer arithmetics.

In order to keep the example programs in this book as simple and understandable as possible, the computations required for defining and positioning the geometric objects are carried out directly in the `paint` method. For more complex animated graphics, i.e., for graphics with moving or changing objects, this can lead to flickering effects and also to the effect that the window might react very slowly, for instance when it should be closed while the animation is still running. Java assigns a high priority to the `paint` method so that other events like closing of the window cannot be carried out immediately. In order to avoid this undesired effect, one can carry out all computations to construct or position objects outside the `paint` method and instead call the `repaint` method only, when objects have to be drawn. The double buffering technique, introduced later on in section 4.2, provides an even better solution.

2.3 Basic geometric objects

The basic geometric objects in computer graphics are usually called *primitives* or *graphics output primitives*. They include geometric entities like points, straight and curved lines and areas as well as character strings. The basic primitives are the following ones.

Points that are uniquely defined by their x- and y-coordinate. Points are usually not drawn themselves. Their main function is the description of other objects like lines that can be defined by their two endpoints.

Lines, *polylines* or *curves* can be defined by two or more points. Whereas for a line two points are needed, curves require additional control points. Polylines are connected sequences of lines.

Areas are usually bounded by *closed polylines* or *polygons*. Areas can be filled with a colour or a texture.

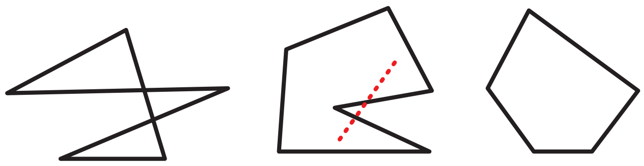


Figure 2.6 A self-overlapping, a nonconvex and a convex polygon

The simplest curve is a line segment or simply a line. A sequence of line where the following line starts where the previous one ends is called a *polyline*. If the last line segment of a polyline ends where the first line segment started, the polyline is called a *polygon*. For various applications—for instance for modelling surfaces—additional properties of polygons are required. One of such properties is that the polygon should not overlap with itself. *Convexity* is another important property that is often needed. A polygon or, more generally, an area or a region is *convex* if whenever two points are within the region the connecting line between these two points lies completely inside the region as well. Figure 2.6 shows a self-overlapping polygon, a nonconvex polygon and a convex polygon. For the nonconvex polygon two points inside the polygon are chosen and connected by a dotted line that lies not completely inside the polygon.

In addition to lines and piecewise linear polylines, curves are also common in computer graphics. In most cases, curves are defined as parametric polynomials that can also be attached to each other like lines in a polyline. The precise definition and computation of these curves will be postponed until chapter 6. Here it is sufficient to understand the principle of how the parameters of a curve influence its shape. In addition to the endpoints of the curve, one or more *control points* have to be specified. Usually, two control points are used leading to a cubic curve or only one control point is used in order to define a quadratic curve. The curve begins and ends in the two specified endpoints. In general, it will not pass through control points. The control points define the direction of the curve in the two endpoints.

In the case of a quadratic curve with one control point one can imagine the lines connecting the control point with the two endpoints. The connecting lines are the tangents of the quadratic curve in the two endpoints. Figure 2.7 illustrates the definition of a quadratic curve on the left-hand side. The quadratic curve is given by two endpoints and one control point through which the curve does not pass. The tangents in the endpoints are also shown here as dotted lines. For a cubic curve as shown on the right-hand side of the figure, the tangents in the two endpoints can be defined independently by the two control points.

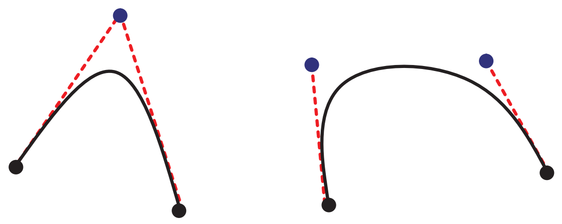


Figure 2.7 Definition of quadratic and cubic curves

When fitting quadratic or cubic curves together in order to form a longer, more complicated curve, it is not sufficient to simply use the endpoint of the previous curve as a starting point for the next curve. The resulting joint curve would be continuous, but not smooth, i.e., sharp bends might occur. In order to avoid sharp bends, the tangent of the endpoint of the previous curve and the following curve must point into the same direction. This means the endpoint, which is equal to the starting point of the next curve, and the two control points defining the two tangents must be collinear. This means they must lie on the same line. Therefore, the first control point of a succeeding curve must be on the line defined by the last control and endpoint of the previous curve.

In the same way a curve can be fitted to a line without causing a sharp bend by locating the first control point on the prolongation of the line. Figure 2.8 illustrates this principle.

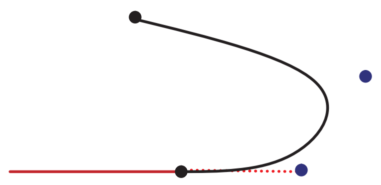


Figure 2.8 Fitting a cubic curve to a line without sharp bends

Other important curves in computer graphics are circles, ellipses and circular and elliptic arcs.

In the same sense as polygons, circles and ellipses define areas. Areas are bounded by a closed curve. When only the shell or margin of the area should be drawn, there is no difference to drawing arbitrary curves. In contrast to lines and simple curves, areas can be filled by colours and textures. From the algorithmic point of view, filling of an area is very different from drawing curves.

Axes-parallel rectangles, whose sides are parallel to the coordinate axes, play an important role in computer graphics. Although they can be understood as special cases of polygons, they are simpler to handle since it is already sufficient to specify two opposing vertices.

Instead of specifying a polygon or the boundary directly in order to define an area, it is sometimes more convenient to construct a more complicated area by combining previously defined areas using set-theoretic operations. The most important operations are union, intersection, difference and symmetric difference. The *union* joins two areas to a larger area whereas their *intersection* consists of the part belonging to both areas. The *difference* of an area with another removes all parts from the first area that also belong to the second area. The *symmetric difference* corresponds to a pointwise exclusive OR-operation applied to the two areas. The symmetric difference is the union of the two areas without their intersection. Figure 2.9 shows the results of applying these operations to two areas in the form of a circle and a rectangle.

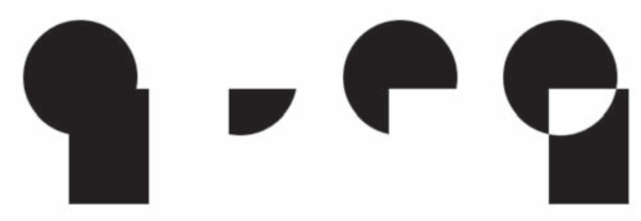


Figure 2.9 Union, intersection, difference and symmetric difference of a circle and a rectangle

Geometric transformations like scalings will be discussed in section 2.5. They provide another way of constructing new areas from already existing ones.

2.4 Basic geometric objects in Java 2D

All methods for generating geometric objects as they were described in the previous section are also available within the Java 2D framework. The abstract class `Shape` with its various subclasses allows the construction of various two-dimensional geometric objects. Vector graphics is used to define `Shape` objects, whose real-valued coordinates can either be given as `float-` or `double`-values. Shapes will not be drawn until the `draw` or the `fill` method is called with the corresponding `Shape` as argument in the form `graphics2d.draw(shape)` or `graphics2d.fill(shape)`, respectively. The `draw` method draws only the margin or circumference of the `Shape` object, whereas the whole area defined by the corresponding `Shape` object is filled, when the `fill` method is called.

The abstract class `Point2D` for points is not a subclass of `Shape`. Points cannot be drawn directly. If one wants to draw a point, i.e., a single pixel, then a line from this point to the same point can be drawn instead. Objects of the class `Point2D` are mainly used to specify coordinates for other geometric objects. In most cases, it is also possible to define these coordinates also directly by two single values determining the x- and the y-coordinate. Therefore, the class `Point2D` will not occur very often in the example programs. The abstract class `Point2D` is extended by the two classes `Point2D.Float` and `Point2D.Double`. When using the abstract class `Point2D` it is not necessary to specify whether coordinates are given as `float-` or `double`-values. The same concept is also used for most of the other geometric objects.

The elementary geometric objects in Java 2D introduced in the following extend the class `Shape`, so that they can be drawn by applying one of the methods `draw` or `fill`.

The abstract class `Line2D` defines lines. One way to define a line from point (x_1, y_1) to point (x_2, y_2) is the following:

```
Line2D.Double line = new Line2D.Double(x1,y1,x2,y2);
```

The parameters `x1`, `y1`, `x2` and `y2` are of type `double`. Similarly, `Line2D.Float` requires the same parameters, but of type `float`. It should be emphasised again that the defined line will not yet be drawn. Only when the method `g2d.draw(line)` is called, will the line appear on the screen.

Analogously to lines, quadratic curves are modelled by the abstract class `QuadCurve2D`. The definition of a quadratic curve requires two endpoints and one control point. The quadratic curve is constructed in such a way that it connects the two endpoints (x_1, y_1) and (x_2, y_2) and the tangents in the endpoints meet in the control point (crtlx,crtly), as illustrated by the left curve in figure 2.7. One way to define quadratic curves in Java 2D is the following:

```
QuadCurve2D.Double qc = new QuadCurve2D.Double(x1,y1,
                                               ctrlx,ctrly,
                                               x2,y2);
```

Cubic curves need two control points instead of one in order to define the tangents in the two endpoints independently as shown by the right curve in figure 2.7. Java 2D provides the abstract class `CubicCurve2D` for modelling cubic curves. Analogously to the cases of lines and quadratic curves, `CubicCurve2D.Double` is a subclass of `CubicCurve2D` allowing to define a cubic curve in the following way:

```
CubicCurve2D.Double cc =
                  new CubicCurve2D.Double(x1,y1,
                                          ctrlx1,ctrly1,
                                          ctrlx2,ctrly2,
                                          x2,y2);
```

The program `CurveDemo.java` demonstrates the usage of the classes `Line2D.Double`, `QuadCurve2D.Double` and `CubicCurve2D.Double`.

The class `GeneralPath` allows the construction not only of polylines, i.e., sequences of lines, but also mixed sequences of lines, quadratic and cubic curves in Java 2D. A `GeneralPath` starts in the origin of the coordinate system, i.e., in the point (0,0). The class `GeneralPath` provides four basic methods for defining a sequence of lines, quadratic and cubic curves. Each method will append a corresponding line or curve to the endpoint of the last element in the sequence of the `GeneralPath`. The methods `lineTo`, `quadTo` and `curveTo` append a line, a quadratic and a cubic curve, respectively, as the next element in the sequence of the `GeneralPath`. These methods are used within `GeneralPath` in the same way as in `Line2D`, `QuadCurve2D` and `CubicCurve2D` except that the definition of the first endpoint of the line or curve is omitted since this point is already determined by the endpoint of the previous line or curve in the `GeneralPath`. The coordinates of the points must be specified as `float`-values. In addition to these three methods for curves and lines, the class `GeneralPath` also contains the method `moveTo` that allows to jump from the endpoint of the previous curve to another point without connecting the points by a line or curve. A `GeneralPath` must always start with the method `moveTo`, defining the starting point of the general path.

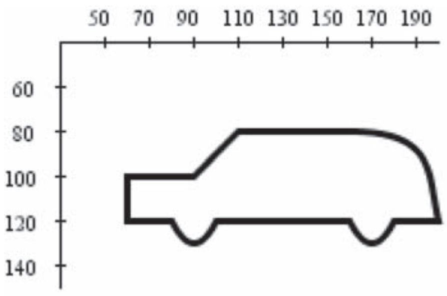


Figure 2.10 An example for a `GeneralPath`

Figure 2.10 shows the outline of a car that was generated by the following `GeneralPath`:

```
GeneralPath gp = new GeneralPath();

//Start at the lower left corner of the car
gp.moveTo(60,120);
gp.lineTo(80,120); //front underbody
gp.quadTo(90,140,100,120); //front wheel
gp.lineTo(160,120); //middle underbody
gp.quadTo(170,140,180,120); //rear wheel
gp.lineTo(200,120); //rear underbody
gp.curveTo(195,100,200,80,160,80); //rear
gp.lineTo(110,80); //roof
gp.lineTo(90,100); //windscreen
gp.lineTo(60,100); //bonnet
gp.lineTo(60,120); //front
```

```
g2d.draw(gp); //Draw the car
```

The coordinate system shown in figure 2.10 refers to the window coordinates, so that the y-axis points downwards. The complete class for drawing the car can be found in the example program `GeneralPathCar.java`.

An area can be defined by its boundary that might be specified as a `GeneralPath` object. In addition to the class `GeneralPath` Java 2D also provides classes for axes-parallel rectangles and ellipses as basic geometric objects.

By the class `Rectangle2D.Double`, extending the abstract class `Rectangle2D`, an axes-parallel rectangle can be defined in the following way:

```
Rectangle2D.Double r2d =
             new Rectangle2D.Double(x,y,width,height);
```

The rectangle is determined by its opposite corners (x, y) and $(x + width, y + height)$ on the diagonal. Taking into account that the y-axis in the window where the rectangle will be drawn points downwards, a rectangle is defined whose upper left corner is located at the position (x, y) and whose lower right corner is at $(x + width, y + height)$. Figure 2.11 shows a rectangle on the left-hand side that was defined by

```
Rectangle2D.Double r2d =
            new Rectangle2D.Double(50,60,150,100);
```

It should be emphasised again that this constructor will only define the rectangle in the same way as for all other `Shape` objects that were introduced so far. It is still necessary to call the method `g2d.draw(r2d)` in order to show the rectangle in the corresponding window.

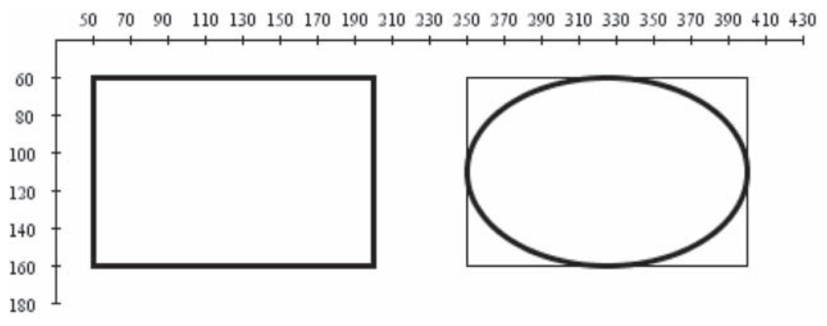


Figure 2.11 An example for a rectangle and an ellipse

In the same way as rectangles, axes-parallel ellipses can be defined in Java 2D. An ellipse is determined by its bounding rectangle which can be specified with the same parameters as `Rectangle2D` objects. The ellipse shown in figure 2.11 on the right-hand side was generated by

```
Ellipse2D.Double elli =
            new Ellipse2D.Double(250,60,150,100);
```

For illustration purposes the bounding rectangle that was used to generate the ellipse is also shown in figure 2.11. The figure was generated by the class `RectangleEllipseExample.java`.

A circle is a special case of an ellipse, where the bounding rectangle is a square. A circle with centre point (x, y) and radius r can be generated by

```
Ellipse2D.Double circle =
            new Ellipse2D.Double(x-r,y-r,2*r,2*r);
```

With the class `Arc2D` elliptic arcs and, of course, circular arcs can be defined.

```
Arc2D.Double arc = new
           Arc2D.Double(rect2D,start,extend,type);
```

– `rect2D` specifies the bounding rectangle of the corresponding ellipse in the form of a `Rectangle2D`.

– `start` is the angle where the arc is supposed to start relative to the bounding rectangle viewed as a square. The angle is given as a `float`-value in terms of degrees.[3] The angle corresponds to the angle with the x-axis only in the special case when a circular arc is defined, i.e., when the bounding rectangle is a square. Otherwise, the angle is determined relative to the rectangle. For example, a starting angle of $45°$ means that the starting point of the arc lies on the connecting line from the centre of the rectangle to its upper right corner.

– `extend` is the opening angle of the arc, i.e., the arc extends from the start angle `start` to the angle `start` + `extend`. Analogously to the `start` angle, `extend` corresponds to the true angle of the arc only in the case of a circular arc. The angle `start` + `extend` is again interpreted relative to the bounding rectangle in the same way as `start`. `extend` must also be specified as a `float`-value in degrees.

– `type` can take one of the three values `Arc2D.OPEN`, `Arc2D.PIE` and `Arc2D.CHORD`, specifying whether only the arc itself, the corresponding segment or the arc with the chord of the ellipse, respectively, should be constructed.

[3] `Arc2D` is the only exception where angles are specified in the unit radians. Otherwise angles in Java 2D and Java 3D must be specified in radiant.

Figure 2.12 shows from left to right an arc of an ellipse, a segment and an arc together with the corresponding chord. In all cases a starting angle of 45° and an opening angle of 90° were chosen. For illustration purposes the bounding rectangle is also shown in the figure. One can see clearly that the arc starts on the intersection of the ellipse with the line from the centre of the bounding rectangle to its upper right corner, according to the choice of the starting angle of 45°. Obviously, the line defined by the centre point of the rectangle and the starting point of the ellipse meets the x-axis in a smaller angle than 45° since a flat, but long bounding rectangle was chosen. The same applies to the opening angle. The actual opening angle is not 90°, but it corresponds to the angle between the lines from the centre of the bounding rectangle to its upper right and to its upper left corner. An example for using the class `Arc2D` can be found in the file `ArcExample.java`, which was also used to generate figure 2.12.

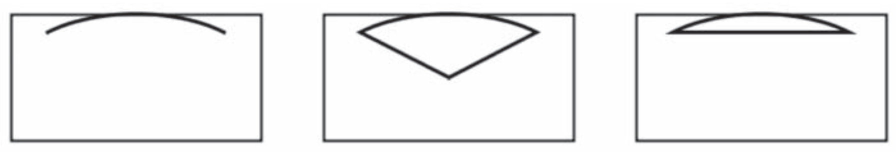


Figure 2.12 An arc of an ellipse, a segment and an arc with its corresponding chord

An area can be defined as a rectangle, an ellipse or in the form of a `GeneralPath`. At the end of section 2.3 a technique for defining areas based on the set-theoretic operations union, intersection, set difference and symmetric difference was explained. Applying these operations to already defined areas, new shapes of areas can be defined. Java 2D offers the class `Area` for this purpose. From a `Shape` object `s`, for instance a `Rectangle2D`, an `Ellipse2D`, a closed `GeneralPath` or an `Arc2D`, representing the segment of an ellipse or an ellipse arc with its chord, an `Area` object with the same outline can be defined by

```
Area a = new Area(Shape s);
```

The above-mentioned set-theoretic operations can be applied to such `Area` objects to generate new areas. Given two `Area` objects `areaA` and `areaB`, the following methods are available, implementing the corresponding set-theoretic operations.

– `areaA.add(areaB)` computes the union of `areaA` and `areaB`.

– `areaA.intersect(areaB)` computes the intersection of `areaA` and `areaB`.

– `areaA.subtract(areaB)` yields `areaA` without the parts lying in `areaB`, i.e., their difference.

– `areaA.exclusiveOr(areaB)` constructs the union of `areaA` and `areaB` without their intersection, i.e., their symmetric difference.

The `Area` object `areaA` contains the result of the application of the corresponding set-theoretic operation. An `Area` object can be used as an argument of the methods `draw`, which will only draw the outline of the area, and `fill`, which will fill the whole area, in the same way as these methods are used for `Shape` objects. The file `AreaExample.java`, which was also used to generate figure 2.9 on page 17, demonstrates the use of `Area` objects.

2.5 Geometric transformations

In addition to geometric objects, geometric transformations play a crucial role in computer graphics. Geometric transformations can be used to position objects, i.e., to shift them to another position or to rotate them, to change the shape of objects, for instance to stretch or shrink them in one direction, or to move objects or change the shape of objects step by step in animated scenes.

Before discussing geometric transformations in more detail, it is necessary to explain some general conventions. In computer graphics, *points* as well as *vectors* are used. From a purely mathematical point of view, both can be represented as elements of the space $\mathbb{R}^n$, i.e., as a tuple of real numbers. Especially in physics, it is very important to distinguish clearly between these two concepts of points and vectors. In the framework of this book and from the viewpoint of computer graphics, it is very common to switch between the interpretations of a tuple of real numbers as a point and as a vector, giving more flexibility in handling certain matters. A tuple $(x_1, \ldots, x_n) \in \mathbb{R}^n$ might occur in one equation as a point and in the next equation it might be interpreted as a vector. Hopefully, physicists will tolerate the abuse of notation in the context of this book. For equations within this book, column vectors will be used consistently. Within the text, points are sometimes written as row vectors in order to avoid stretching of text lines. In those cases where a point is explicitly understood as a column vector, the symbol for transposing vectors will be used, i.e., the point will be written as $(x, y)^\top \in \mathbb{R}^2$ and $(x, y, z)^\top \in \mathbb{R}^3$, respectively.

The *dot product* of two vectors $\mathbf{u}$ and $\mathbf{v}$ will be denoted in the following way, which is also very common in statistics:

$$\mathbf{u}^\top \cdot \mathbf{v} \;=\; (u_1, \ldots, u_n) \cdot \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} \;=\; \sum_{i=1}^{n} u_i \cdot v_i.$$

The most important geometric transformations are scaling, rotation, shearing and translation.

A *scaling* leads to *stretching* or *shrinking* of objects in the direction of the x- and the y-axis. A scaling $S(s_x, s_y)$ maps the point (x, y) to the point (x', y') given by

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x \cdot x \\ s_y \cdot y \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \cdot \begin{pmatrix} x' \\ y' \end{pmatrix}.$$

s_x is the scaling factor in the direction of the x-axis. If $|s_x| > 1$ holds, then a stretching in the direction of the x-axis is carried out. For $|s_x| < 1$ shrinking takes place. If s_x is negative, in addition to stretching or shrinking in the x-direction, a reflection with respect to the y-axis is applied. In the same way, s_y leads to stretching or shrinking in the direction of the y-axis and a negative value of s_y incorporates an additional reflection with respect to the x-axis.

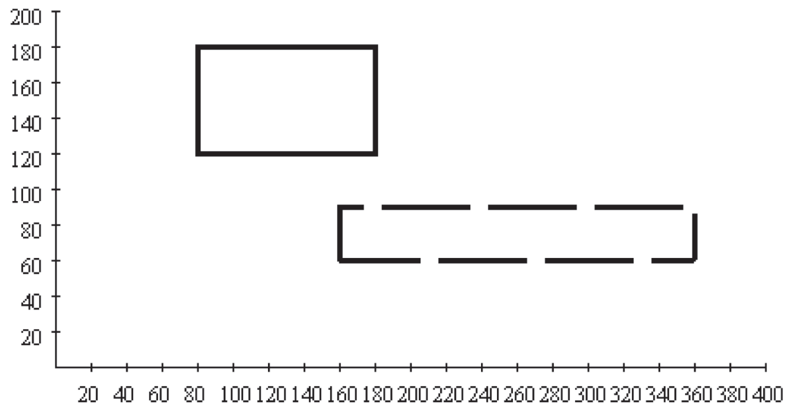


Figure 2.13 Scaling applied to a rectangle

Applying a scaling to an object means that the scaling is carried out pointwise. The same holds for all other geometric transformations. They carry out pointwise transformations of objects. As an example, the translation with the scaling factors $s_x = 2$ and $s_y = 0.5$ is considered, stretching along the x-axis by the factor 2 and shrinking in the direction of the y-axis by the factor 0.5. The application of this scaling to the rectangle whose lower left corner is located at the point (80,120) and whose upper right corner is at (180,180) yields a rectangle whose width has doubled with half the original height. But in addition, the centre of the rectangle is also transformed so that the transformed rectangle is shifted to the lower right compared to the original rectangle. Figure 2.13[4] shows the original rectangle and the rectangle after scaling by dashed lines. A scaling is always carried out with respect to the origin of the coordinate system. Applying a scaling to an object that is not centred around the origin of the coordinate system will lead to a translation of the (centre of the) object in addition to the scaling.

Another important group of geometric transformations are *rotations* that are determined by a single parameter, the rotation angle. The rotation is carried out anticlockwise around the origin of the coordinate system in case of a positive angle. A negative angle means that the rotation is carried out in a clockwise manner. The rotation $R(\theta)$ by the angle θ maps the point (x, y) to the point (x', y') given by

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \cdot \cos(\theta) - y \cdot \sin(\theta) \\ x \cdot \sin(\theta) + y \cdot \cos(\theta) \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}.$$

A rotation is always carried out around the origin of the coordinate system. Therefore, a similar shifting effect as in the case of scalings happens, when an object is not centred around the origin. In figure 2.14 a rotation by an angle of 45° was carried out, mapping the original rectangle to the rectangle drawn with dashed lines.

The *shear transformation* is another elementary geometric transformation that causes a certain deformation of objects. Similar to scalings, the shear transformation requires two parameters, however, not on the main diagonal of the transformation matrix, but on the other two positions. Applying a shear transformation $Sh(s_x, s_y)$ to a point (x, y) yields the point (x', y') with the new coordinates

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x + s_x \cdot y \\ y + s_y \cdot x \end{pmatrix} = \begin{pmatrix} 1 & s_x \\ s_y & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}.$$

As in the case of scalings and rotations, shear transformations are carried out with respect to the origin of the coordinate system, so that an object that is not centred around the origin will not only be deformed by a shear transformation, but also shifted. The dashed rectangle is obtained from the original rectangle in figure 2.15 by applying a shear transformation with the parameters $s_x = 1$ and $s_y = 0$.

[4] The figure is drawn in the usual representation and not in the standard Java 2D window coordinate representation where the y-axis would point downwards.

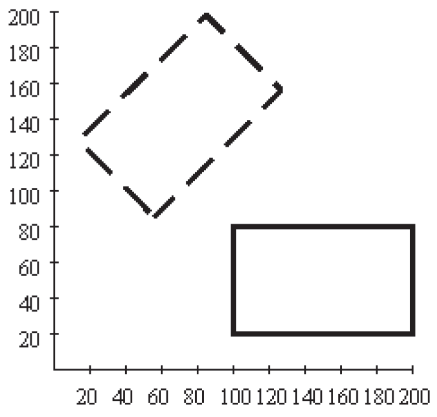


Figure 2.14 A rotation applied to a rectangle

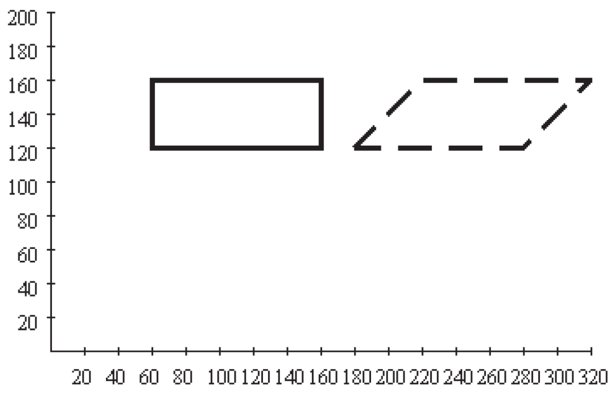


Figure 2.15 A shear transformation applied to a rectangle

Since $s_y = 0$ was chosen for this shear transformation, the shearing takes place in the direction of the y-axis. When the shearing should be carried out in the direction of the x-axis, $s_x = 0$ must hold.

The last elementary or primitive geometric transformation to be considered here is very simple, but differs from the other three types of elementary transformations that were introduced so far in an important aspect. A *translation* $T(d_x, d_y)$ causes a shift by the vector $\mathbf{d} = (d_x, d_y)^\top$. This means the translation maps the point (x, y) to the point

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x + d_x \\ y + d_y \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} d_x \\ d_y \end{pmatrix}.$$

In figure 2.16 a translation defined by the vector $\mathbf{d} = (140, 80)^\top$ is applied to a rectangle, mapping it to the dashed rectangle.

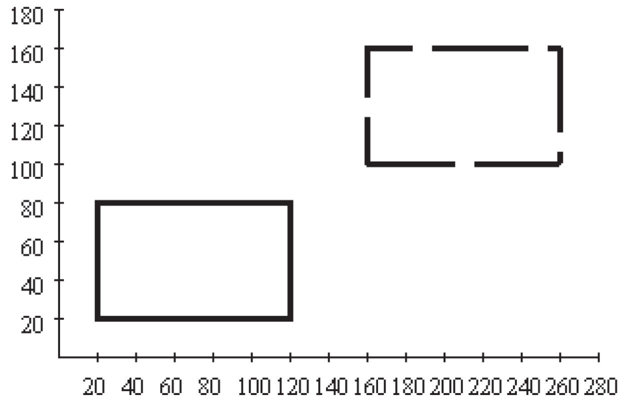


Figure 2.16 Translation of a rectangle

In contrast to the other transformations introduced so far, translations are not linear, so that they cannot be represented in terms of matrix multiplication. A matrix multiplication will always leave the zero vector unchanged, whereas a translation will shift all points including the origin of the coordinate system corresponding to the zero vector. Translations are affine, but not linear mappings.

Within computer graphics, more complex transformations are usually described or generated as compositions of elementary geometric transformations. A transformation composed of scalings, rotations and shear transformations can be specified by a single matrix, obtained as the product of the matrices encoding the corresponding elementary transformations. When also translations are involved, the composition of transformation can no longer be computed by simple matrix multiplication and represented by a single matrix. If all this was possible within matrix calculus, this would be a great advantage in terms of memory—just a single matrix is required to represent a complex transformations—and in terms of computational efficiency since all that would be needed for fast computations are efficient implementations of matrix operations.

In order to represent also translations in matrix form, another representation of the coordinates of points is introduced. The next section will discuss this alternative representation called homogeneous coordinates in more detail.

2.6 Homogeneous coordinates

This section introduces the representation of points in the two-dimensional plane in homogeneous coordinates. The same concept will also be applied later on to points in the three-dimensional space for the same reason, to allow the representation of 3D translations in matrix form. *Homogeneous coordinates* use an additional dimension for the representation of points. The point (x, y, z) in homogeneous coordinates is identified with the point $\left(\frac{x}{z}, \frac{y}{z}\right)$ in Cartesian coordinates. The z-component of a point in homogeneous coordinates must not be zero. When the point (x_0, y_0) in Cartesian coordinates has to be transformed into homogeneous coordinates, the representation $(x_0, y_0, 1)$ can be used. This is, however, not the only way to represent the point (x_0, y_0) in homogeneous coordinates. Any representation of the form $(z \cdot x_0, z \cdot y_0, z)$ where $z \neq 0$ encodes also the same point. The points $\{(x, y, z) \in \mathbb{R}^3 \mid (x, y, z) = (z \cdot x_0, z \cdot y_0, z)\}$ lie on a line in the space $\mathbb{R}^3$ passing through the origin of the coordinate system. The line is given by the system of equations

$$\begin{aligned} x - x_0 \cdot z &= 0, \\ y - y_0 \cdot z &= 0. \end{aligned}$$

Any point on this line, except the origin of the coordinate system, is a representative in homogenous coordinates of the point (x_0, y_0) in Cartesian coordinates. Fixing a value for z for the representation in homogeneous coordinates, for instance $z = 1$, the Cartesian x/y-plane is represented by a parallel plane with the corresponding constant z-value.

Figure 2.17 illustrates these relations. All points on the line shown in figure 2.17 represent the same point in the two-dimensional Cartesian plane $\mathbb{R}^2$. Choosing a constant value for z, for instance one of the planes shown in figure 2.17, the corresponding plane is a homogeneous representative of the Cartesian plane $\mathbb{R}^2$.

The origin of the Cartesian coordinate system corresponds to any point of the form $(0, 0, z)$ $(z \neq 0)$ in homogeneous coordinates. This point is no longer a necessary fixed point of a linear mapping in terms of homogeneous coordinates, i.e., a linear mapping from $\mathbb{R}^3$ to $\mathbb{R}^3$. The linear mapping can map this point to another point in homogeneous coordinates.

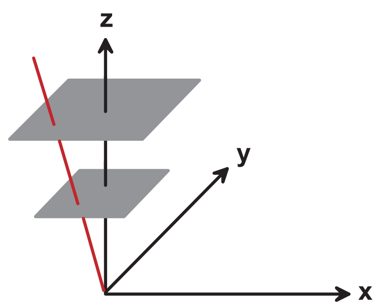


Figure 2.17 Homogeneous coordinates

In homogeneous coordinates a translation can now be written as matrix multiplication:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} x + d_x \\ y + d_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} .$$

The other elementary transformation can be extended to homogeneous coordinates in a straightforward manner, leading to the following set of transformation matrices:

transformation	notation	matrix
translation	$T(d_x, d_y)$	$\begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix}$
scaling	$S(s_x, s_y)$	$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$
rotation	$R(\theta)$	$\begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$
shear transformation	$S(s_x, s_y)$	$\begin{pmatrix} 1 & s_x & 0 \\ s_y & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Rotations and translations preserve lengths and angles. Scalings and shear transformations do not preserve lengths and angles in general, but at least parallel lines will be mapped to parallel lines again.

With this matrix representation in homogeneous coordinates, the composition of geometric transformations can be computed by matrix multiplication.

All matrices, introduced for the elementary geometric transformations are of the form

$$\begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix}. \tag{2.1}$$

It is easy to verify that the product of two such matrices results again in a matrix of the same form. Therefore, geometric transformations are usually represented and stored in this way in computer graphics. This does not only apply to transformations that operate on the two-dimensional plane, but also to transformations in the three-dimensional space that will be discussed in chapter 5. It is now obvious that a graphics card of a computer must—among other things—be able to carry out matrix operations as fast as possible.

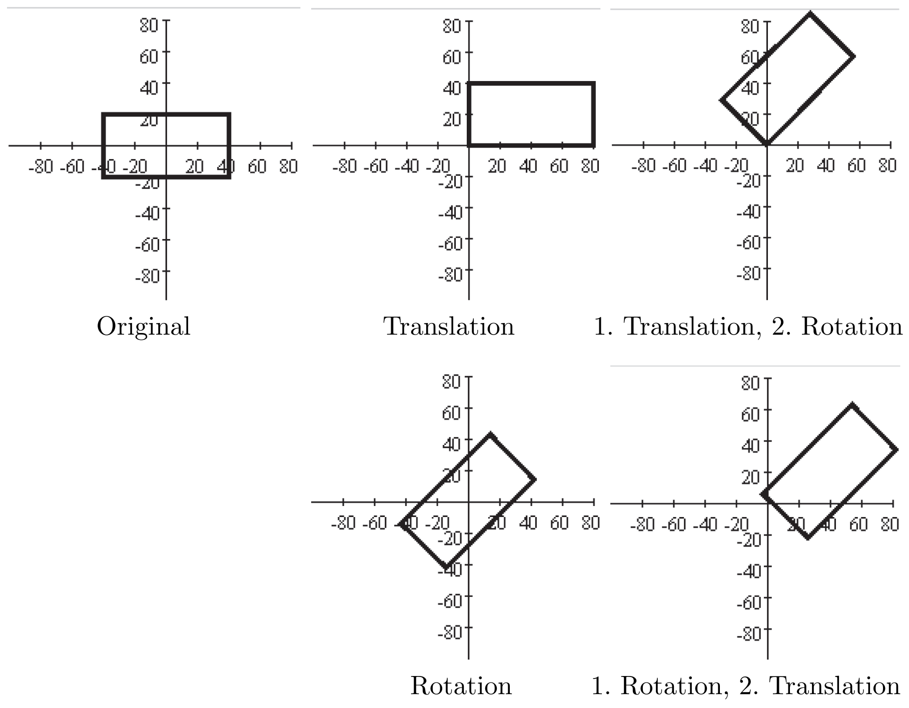


Figure 2.18 Differing results on changing the order for the application of a translation and a rotation

For the composition of transformations it should be taken into account that the order in which the transformations are applied is of importance. Matrix multiplication is a noncommutative operation. The right part of figure 2.18 shows the different results that are obtained, when the order in which translation and rotation are applied is changed. When first a translation with the translation vector $(40, 20)^\top$ and afterwards a rotation by 45° is applied, then the rectangle on the left-hand side of figure 2.18 is mapped to the upper rectangle on the right. When the rotation is carried out first and afterwards the translation, the result is the lower right rectangle. This effect occurs in general in all cases, when geometric transformations of different types are combined. Only when transformations of the same type, i.e., only rotations, only translations, only scalings or only shear transformations, are composed, the order in which the transformations are applied is of no importance.

It should also be taken into account that transformations in matrix notation or as compositions of mappings are carried out from right to left. The transformation

$$(T(d_x, d_y) \circ R(\theta))(\mathbf{v})$$

or in matrix notation

$$\begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \mathbf{v}$$

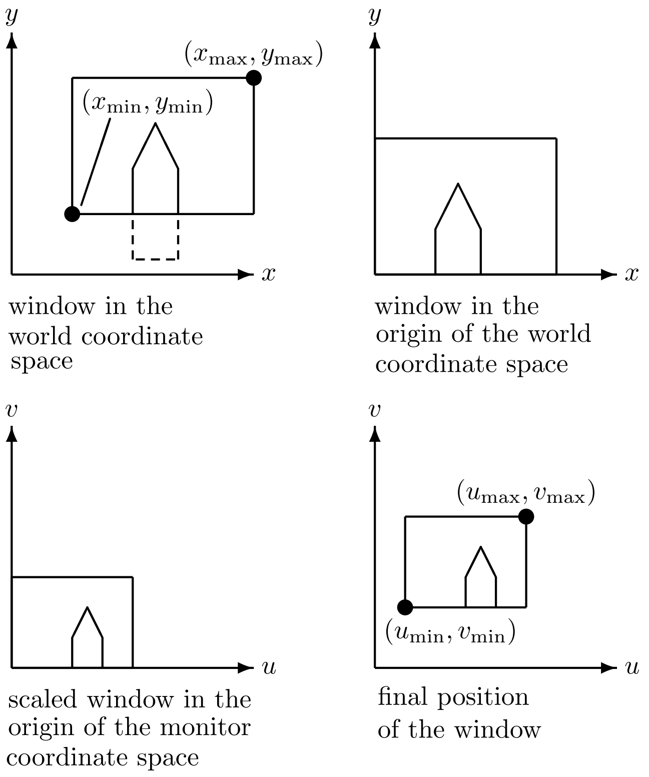
means that first the rotation $R(\theta)$ and then the translation $T(d_x, d_y)$ is applied to the point $\mathbf{v}$.

2.7 Applications of transformations

This section introduces typical applications and problems that can be solved using geometric transformations.

In computer graphics it is common to define objects in an arbitrary coordinate system in floating point arithmetics, the so-called *world coordinates*. In order to generate an image of specific objects on the computer screen or another output device, a rectangular clipping region, called *viewport*, must be specified. The viewport determines which region of the “object world” is visible. Therefore, it is necessary to find a transformation that maps the objects in the viewport, given in world coordinates, to the window coordinates of the computer screen.

Figure 2.19 illustrates the problem and its solution. The rectangle with the lower left corner at $(x_{\min}, y_{\min})$ and the upper right corner at $(x_{\max}, y_{\max})$ in the upper left part of the figure defines the viewport or the clipping region, a window specified in the world coordinate system. This part of the object world



y
$(x_{\max}, y_{\max})$
$(x_{\min}, y_{\min})$
x
window in the world coordinate space
y
x
window in the origin of the world coordinate space
v
u
scaled window in the origin of the monitor coordinate space
v
$(u_{\max}, v_{\max})$
$(u_{\min}, v_{\min})$
u
final position of the window

Figure 2.19 From world to window coordinates

should be shown on the computer screen in a window defined by the rectangle with $(u_{\min}, v_{\min})$ as its lower left and $(u_{\max}, v_{\max})$ as its upper right corner, given in the coordinates of the computer monitor. These two windows can have different sizes. Even the shape of the two rectangles does not have to coincide.

The mapping from the viewport to the computer monitor can be realized as a composition of the following transformations. In the first step, the viewport window is shifted to the origin of the coordinate system by a suitable translation. In the next step, the viewport window is scaled in such a way that it matches the size of the window on the computer screen. Finally, another translation will position the scaled window at the correct location on the computer screen. Altogether, the following sequence of elementary geometric transformations has to be carried out:

$$T(u_{\min}, v_{\min}) \circ S\left(\frac{u_{\max} - u_{\min}}{x_{\max} - x_{\min}}, \frac{v_{\max} - v_{\min}}{y_{\max} - y_{\min}}\right) \circ T(-x_{\min}, -y_{\min}). \qquad (2.2)$$

In equation (2.2) $\circ$ denotes the composition or concatenation of mappings. It should be noted again that the transformations are carried out from right to left in this formula.

As mentioned before, rotations $R(\theta)$ defined by a rotation matrix as described on page 29 will always carry out the rotation around the origin of the coordinate system. When the centre of the rotation is supposed to be another point (x_0, y_0), one can achieve this by applying first a translation shifting the point (x_0, y_0) to the origin of the coordinate system, then carry out the rotation around the origin and afterwards reverse the initial translation. This means that a rotation through the angle θ around the point (x_0, y_0) can be implemented by the following composition of transformations:

$$R(\theta, x_0, y_0) \;=\; T(x_0, y_0) \circ R(\theta) \circ T(-x_0, -y_0). \tag{2.3}$$

In the same way, a scaling can be carried out with respect to the point (x_0, y_0) instead of the origin of the coordinate system, by simply replacing the rotation in (2.3) by the corresponding scaling.

Pixel coordinates within a window on the computer screen are usually specified in such a way that the first component refers to the pixel column, whereas the second component refers to the pixel row, where pixel rows are counted from top to bottom. As a consequence, the x-axis points as usual from left to right, but the y-axis is oriented in a reverse manner, i.e., it points downwards instead of upwards. When the specification in standard Cartesian coordinates is preferred, one can simply apply suitable geometric transformations, before drawing objects in the window coordinate system. In order to reverse the direction of the y-axis, a reflection with respect to the x-axis has to be carried out. After this reflection, the y-axis points upwards, but the origin of the coordinate system of the window still remains in the upper left corner, so that only objects with negative y-components would be visible. Therefore, after the reflection a translation is also carried out. The translation is a shift in y-direction by the height h of the window, measured in pixels. In this way, the origin of the coordinate system is mapped to the lower left corner of the window. The reflection is a scaling with the parameters $s_x = 1$ and $s_y = -1$. Altogether the transformation is given by

$$T(0, h) \circ S(1, -1). \tag{2.4}$$

2.8 Geometric transformations in Java 2D

The class `AffineTransform` is the basis for geometric transformations in Java 2D where geometric transformations are also implemented by matrices in homogeneous coordinates. The most important constructors are:

– `AffineTransform id = new AffineTransform()`
  generates the identical transformation, that is encoded by the unity matrix. This default constructor generates a transformation that maps every point to itself.

– The constructor

```
AffineTransform at = new AffineTransform(a,d,b,e,c,f)
```

  allows the specification of an arbitrary transformation matrix. The arguments $a, \dots, f$ define the six `Double`-parameters of the transformation matrix. The matrix (2.1) on page 30 shows the assignment of the arguments to the corresponding matrix parameters.

The elementary geometric transformations can be generated in the following way:

Rotation:

- For rotations the class `AffineTransform` provides the two methods `affTrans.setToRotation(angle)` that defines the transformation `affTrans` as a rotation through the angle `angle` around the origin of the coordinate system and `affTrans.setToRotation(angle,x,y)` setting the transformation `affTrans` to a rotation through the angle `angle` around the point (x, y), respectively.
- The method `affTrans.rotation(angle)` and the corresponding method `affTrans.rotation(angle,x,y)` extend the transformation `affTrans` by a rotation around the origin of the coordinate system or around the point (x, y). This means that the matrix that encodes the original transformation `affTrans` is multiplied from the right by a matrix for the corresponding rotation. As a consequence, when `affTrans` is applied to an object, the rotation is carried out first and afterwards the original transformation in `affTrans` is applied.

Scaling:

- The method `affTrans.setToScale(sx,sy)` defines the transformation `affTrans` as a scaling with the scaling factors `sx` for the x- and `sy` for the y-axis with respect to the origin of the coordinate system.
- The method `affTrans.scale(sx,sy)` extends the transformation `affTrans` by a corresponding scaling. The extension is to be understood in the same way as in the case of rotations, i.e., as a matrix multiplication from the right.

Shear transformation:

- The method `affTrans.setToShear(sx,sy)` defines the transformation `affTrans` as a shear transformation with the shear values `sx` for the x- and `sy` for the y-axis with respect to the origin of the coordinate system.
- The method `affTrans.shear(sx,sy)` extends the transformation `affTrans` by a corresponding shear transformation, again in terms of matrix multiplication from the right.

Translation:

- The method `affTrans.setToTranslation(dx,dy)` defines the transformation `affTrans` as translation by the vector $(dx, dy)^\top$.
- The method `affTrans.translate(dx,dy)` extends the transformation `affTrans` by a corresponding translation in the same manner as for rotations, scalings and shear transformations, i.e., as matrix multiplication from the right.

The following methods for the composition of such affine transformations are available in the class `AffineTransform`:

- By `at1.concatenate(at2)` the affine transformation `at2` is appended to the affine transformation `at1` in terms of matrix multiplication from the right, so that first `at2` and then the original transformation `at1` is carried out.
- By `at1.preConcatenate(at2)` the affine transformation `at2` is combined with the affine transformation `at1` in the sense of matrix multiplication from the left. This means that first the original transformation `at1` and then `at2` is carried out.

In both cases, the composition of the two transformations is stored in `at1`.

An affine transformation that is defined as an instance `affTrans` of the class `AffineTransform` can be applied to a `Shape` object `s` in the following way:

```
Shape transformedShape = affTrans.createTransformedShape(s);
```

The method `createTransformedShape` returns the transformed object again as an instance of the class `Shape`.

In the same way, affine transformations can be applied to an `Area` object `a`, for instance in the form

```
Area transformedArea = affTrans.createTransformedArea(a);
```

An affine transformation can also be applied to the `Graphics2D` object `g2d` by

```
g2d.transform(affTrans);
```

In this case, the corresponding affine transformation will be applied to all objects before they are drawn.

Figures 2.13–2.16 and the images in figure 2.18 were generated using these methods in the following programs:

– `ScalingExample.java`,
– `RotationExample.java`,
– `ShearingExample.java`,
– `TranslationExample.java`,
– `TransformationOrderExample.java`,
– `TransformationOrderExampleT.java`,
– `TransformationOrderExampleRT.java`,
– `TransformationOrderExampleR.java` and
– `TransformationOrderExampleTR.java`.

In all of these figures a standard Cartesian coordinate system instead of the window coordinate system was used for the representation of the objects, so that the y-axis points upwards in the window. In order to achieve this effect, an affine transformation according to (2.4) was applied to the `Graphics2D` object yielding the desired orientation of the y-axis and the desired location of the origin of the coordinate system.

```
AffineTransform yUp = new AffineTransform();
yUp.setToScale(1,-1);
AffineTransform translate = new AffineTransform();
translate.setToTranslation(xOffset,windowHeight-yOffset);
yUp.preConcatenate(translate);
g2d.transform(yUp);
```

In (2.4) the values `xOffset` and `yOffset` do not occur. This means they are assumed to be zero there. Setting both values to zero means that the origin of the coordinate system is in the lower left corner of the window, a point that is on the margin of the window and therefore, it is not possible to draw this point. Thus, the origin of the coordinate system was slightly shifted to the interior of the window by choosing `xOffset`=140 and `yOffset`=150, so that the origin of the coordinate system is visible inside the window.

2.9 Animation and movements based on transformations

So far, geometric transformations were only applied in a static manner in order to map one coordinate system to another or to describe positioning and deformation of objects. Geometric transformations are also suitable to model moving objects, for instance moving the hands of a clock carrying out a rotation of 6° per second or per minute or 30° per hour. Continuous movements must be decomposed into small stepwise movements which can be described by geometric transformations. The stepwise changes between two images must be small enough and the time between two images in a sequence must be short enough in order to let the movement appear as a continuous or floating movement and not as jumps from one state to another.

Once the movement of an object is modelled by suitable geometric transformations, the object must be drawn, the transformed object has to be computed, the old object must be deleted and the new transformed object has to be drawn again. Deleting the old object causes problems for raster graphics. Deleting means in this case to overwrite the object. In order to overwrite the pixels of the old object a background image must be specified. For more complex objects, overwriting the pixels belonging to the old object would require to render the object again in order to determine which pixels were covered by the old object. Therefore, instead of overwriting single moving objects it is common to write the complete image buffer again, instead of modifying the old one. However, the new image is usually not written directly into the window buffer, but into a virtual buffer which will be copied to the window buffer after the image has been completed. This technique is also called *double buffering* and will be explained in more detail in section 4.2.

As a simple example for a moving object, a moving clock with a single hand for the seconds is considered sliding from the lower left to the upper right of a display window. The clock itself consists of a quadratic frame and the single rectangular hand for the seconds. The hands for minutes and hours could be modelled in the same way, but are not included here for reasons of simplicity. A translation is needed in order to move the quadratic frame of the clock from the lower left to the upper right corner of the window. This translation must also be applied to the hand for the seconds. In addition to the translation, the hand must also rotate. Figure 2.20 shows some snapshots of the moving clock.

Assuming that the clock should move in each step two units to the right and one unit up, this could be modelled by a translation

$$T_{\text{clock,step}} \;=\; T(2,1).$$

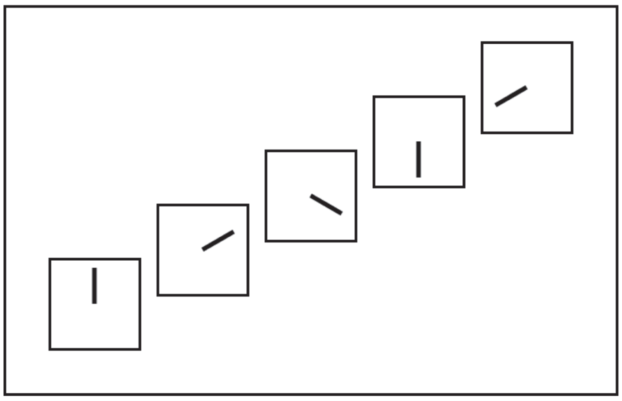


Figure 2.20 A moving clock with a rotating hand

For the hand a rotation of the form

$$T_{\text{hand,step}} = R(-\pi/180)$$

is needed in order to turn the hand by $-\pi/180$, i.e., by 6° clockwise in each step. One end of the hand is fixed in the centre of the clock which is therefore also the centre of the rotation. Even if the clock were initially centred in the origin of the coordinate system, it would move out of the origin after one step already and the centre of rotation for the hand would no longer be the origin.

There are two possible strategies to describe and handle such composed movements as in the case of the hand, where a translation as well as a rotation has to be carried out. One way would be to track the position of the corresponding object—in this case the hand of the clock—and to shift the centre of the rotation accordingly. In the general case, it is not sufficient to track only the translation of an object. If, for instance, the object is also supposed to be scaled along one of its axes, it is also necessary to know the orientation of the object in order to apply the scaling properly. As an example, the hand of the clock could get longer or shorter while it is rotating without changing its width. In the beginning, the corresponding scaling had to be a scaling along the y-axis. But once the hand starts to rotate, the axis of scaling must also be rotated. Otherwise the hand would not only become longer or shorter, but also thicker or thinner.

Although this strategy for modelling continuous movements of objects is applicable, the following second strategy seems to be more convenient and simpler to implement. The principle of this second strategy is to leave the objects in their initial positions and to compute accumulated geometric transformations which are applied to the objects before they are drawn. For the example of the clock one could use three more transformations in addition to the abovementioned two transformations:

$$\begin{aligned} T^{(\text{new})}_{\text{clock,accTrans}} &= T_{\text{clock,step}} \circ T^{(\text{old})}_{\text{clock,accTrans}} \\ T^{(\text{new})}_{\text{hand,accRotation}} &= T_{\text{hand,step}} \circ T^{(\text{old})}_{\text{hand,accRotation}} \\ T_{\text{hand,acc}} &= T_{\text{clock,accTrans}} \circ T_{\text{hand,accRotation}}. \end{aligned}$$

$T_{\text{clock,accTrans}}$ and $T_{\text{hand,accRotation}}$ are initialized by the identical transformations and are then updated in each step according to the specified equations. $T_{\text{clock,accTrans}}$ specifies the translation which has to be carried out in order to shift the clock from the initial position to the actual position. $T_{\text{clock,accTrans}}$ is applied to the initial frame of the clock that is centred in the origin of the coordinate system. $T_{\text{hand,accRotation}}$ describes the rotation around the origin of the coordinate system that must be applied to the hand of the clock in order to reach its actual position within the clock centred around the origin. In addition to this rotation, the hand must also move along with the clock. Therefore, after rotating the hand around the origin, the corresponding translation $T_{\text{clock,accTrans}}$ is also applied to the hand. It is important that the rotation is carried out first and only then the translation is applied.

Scenegraphs as they are introduced in chapter 5 provide a more convenient alternative to this way of modelling movements and animations.

2.10 Movements via transformations in Java 2D

This section explains how to implement the simple example of the moving clock of the previous section in Java 2D. Within the book, only the most essential parts of the source code are shown. The full source code for this example can be found in the class `NonSynchronizedClock.java`.

In order to specify the location of the objects and the transformations in standard coordinates with the y-axis pointing upwards, the transformation `yUp` introduced on page 36 is applied to the `Graphics2D` object.

Initially, the frame of the clock will be centred in the origin of the coordinate system. It is represented by the object `clockFrame` of the class `Rectangle2D`. The single hand of the clock named `clockHand` is also generated as an object from the class `Rectangle2D`. Its initial position is chosen in such a way that it starts in the origin of the coordinate system and points upwards with the y-axis as its centre axis.

The transformations $T_{\text{clock,step}}$, $T_{\text{hand,step}}$, $T_{\text{clock,accTrans}}$, $T_{\text{hand,accRotation}}$ and $T_{\text{hand,acc}}$ as described above are represented by the objects `singleTranslation`, `singleRotation`, `accumulatedTranslation`, `accumulatedRotation` and `handTransform`, respectively, all belonging to the class `AffineTransform`.

The transformation `singleTranslation` is defined as a translation by the vector $(2, 1)^\top$, whereas `singleRotation` is a clockwise rotation by an angle of $6°$. Both transformations remain unchanged, while the program is running.

The transformation `accumulatedRotation` is initialised as the identity. The transformation `accumulatedTranslation` could also be initialised as the identity. But this would lead to the effect that the clock starts its movement centred in the lower left corner of the window, so that in the beginning only the upper left quarter of the clock would be visible. Therefore, a suitable translation is chosen for the initialisation of `accumulatedTranslation` ensuring that the clock is fully visible, when the animation is started.

A loop is used to compute the stepwise changing positions of the clock and the hand. In this loop, the transformations `accumulatedTranslation`, `accumulatedRotation` and `handTransform` are updated according to the equations specified on page 39. This is realised by the following lines of code:

```
accumulatedTranslation.preConcatenate(singleTranslation);
accumulatedRotation.preConcatenate(singleRotation);
handTransform.setTransform(accumulatedRotation);
handTransform.preConcatenate(accumulatedTranslation);
```

The first line corresponds to the first equation, the second line to the second equation and the last two lines implement the last equation.

After the transformations have been updated in this way, the old image must be deleted and afterwards the frame of the clock and its hand have to be drawn again. Before these objects are drawn, the corresponding transformations are applied to them making sure that they are positioned at their updated locations.

```
g2d.draw(accumulatedTranslation.createTransformedShape(
                                                    clockFrame));

g2d.fill(handTransform.createTransformedShape(clockHand));
```

The initial objects `clockFrame` and `clockHand` for the frame of the clock and for its hand, respectively, are not changed in the loop. Only the transformations applied to them change and the updated transformations are used to generate new objects of the class `Shape` that are drawn in each step.

The implementation proposed here has various disadvantages. Since all computations for the animation are carried within the `paint` method, this might lead to flickering and it might also be difficult to stop the animation since the `paint` method has a high priority in order to avoid showing half-ready images.

For MAC computers the animation might not work at all since all computations in the `paint` method will be carried out completely, before anything is drawn on the screen. The program `NonSynchronizedClock.java` also uses a very primitive `sustain` method for the intermediate time between two frames. This method implements active waiting which should be avoided in programming. The double buffering technique introduced in section 4.2 offers a much better solution than the one provided here which was only presented for reasons of simplicity.

2.11 Interpolators for continuous changes

The previous two sections have demonstrated how moving objects can be modelled on the basis of suitable transformations. Single stepwise transformations describe the changes of the objects from one image frame to the next one. The composition of these stepwise transformations determines the complete movement of an object.

But there are also other ways to model movements or changes of objects in animated graphics. An alternative approach is based on the two descriptions of the initial state of the considered object and of its desired final state. The aim is then to find an animation that shows a continuous transition of the object from the initial to its final state. The movement of an object along a line can either be modelled by small stepwise translations to be carried out between two image frames as in the previous two sections or by simply specifying the initial and the end position of the object and then carrying out interpolations between these two positions. In the example of the clock from the previous two sections, one would not define the transformation $T_{\text{clock,step}} = T(2,1)$ to be applied repeatedly to the clock in a sequence of, for instance, 100 images. Instead, it would be sufficient to specify the initial and the end position of the object, say $\mathbf{p}_0 = (0,0)^\top$ and $\mathbf{p}_1 = (200,100)^\top$.

The points $\mathbf{p}_\alpha$ on the connecting line between the points $\mathbf{p}_0$ and $\mathbf{p}_1$ are simply the *convex combinations* of these two points given by

$$\mathbf{p}_\alpha \;=\; (1-\alpha)\cdot\mathbf{p}_0 + \alpha\cdot\mathbf{p}_1, \qquad \alpha \in [0,1].$$

$\alpha = 0$ yields the initial point $\mathbf{p}_0$, $\alpha = 1$ leads to the end point $\mathbf{p}_1$ and $\alpha = 0.5$ defines the point in the middle between $\mathbf{p}_0$ and $\mathbf{p}_1$.

This principle of convex combinations can be applied not only to points or vectors, but also to matrices. Later on, in section 4.7 the principle of convex combinations will also be applied to colours to generate continuous changes from one colour to another.

In order to understand how convex combinations applied to matrices can be used to generate animations, two affine transformations are considered, given by the matrices M_0 and M_1 in homogeneous coordinates. The convex combination M_α of the two matrices is defined as

$$M_\alpha \;=\; (1-\alpha)\cdot M_0 + \alpha \cdot M_1, \qquad\qquad \alpha \in [0,1].$$

Note that M_α is again a matrix describing an affine transformation in homogeneous coordinates. In the simplest case, the two matrices encode translations mapping an object to its initial and its desired final position. The matrices M_α correspond to intermediate translations. M_α places the object on the points on the line connecting the initial and the end position. For $\alpha = 0$, M_α maps the object to its initial position and for $\alpha = 1$ to its final position.

However, convex combinations are not restricted to translations. In principle, the matrices M_0 and M_1 can represent any two affine transformations that do not even have to belong to the same type of transformation. One could be a rotation, the other a scaling combined with a shearing.

In this way, a continuous transformation can be implemented between two objects obtained from the same object by applying two different transformations. Figure 2.21 illustrates this process for two ellipses that were both generated from the same basic object—also an ellipse—by applying different scalings and transformations. The ellipse in the upper left corner is obtained from the basic ellipse by applying the first transformation, whereas the second transformation yields the ellipse in the lower right corner. Applying convex combinations of these two transformations to the basic ellipse leads to the ellipses in between.

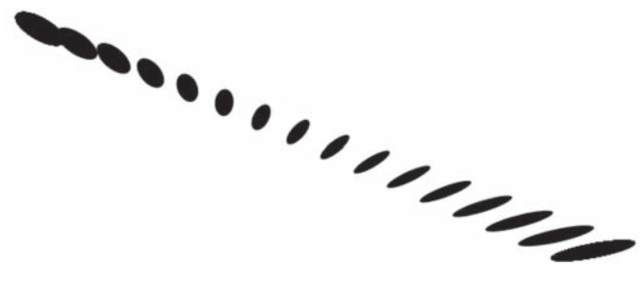


Figure 2.21 Changing one ellipse to another by convex combinations of transformations

Another technique for continuous interpolation between two objects S and S' assumes that both objects are determined by n points $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$ and $P'_1 = (x'_1, y'_1), \ldots, P'_n = (x'_n, y'_n)$, respectively, and by lines or quadratic and cubic curves defined using these points. It is important that the lines or curves in both objects are determined by the corresponding points. This means, if a quadratic curve defined by the points P_1, P_3 and P_8 is part of object S, then the corresponding quadratic curve defined by the points P_1', P_3' and P_8' must be part of object S'.

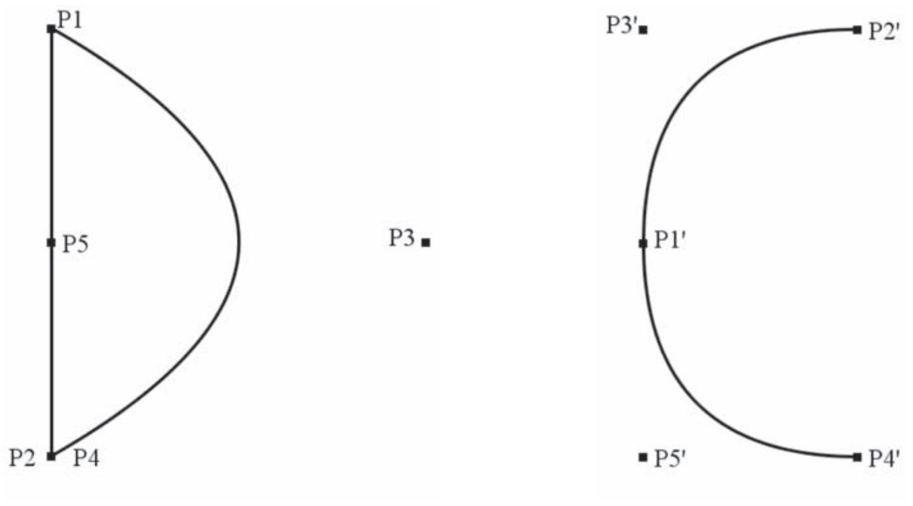


Figure 2.22 Two letters each defined by five points and two quadratic curves

Figure 2.22 shows two simple objects in the form of the two letters D and C. For each of them five control points $P1, \ldots, P5$ and $P1', \ldots, P5'$, respectively, are specified. Both letters are described by two quadratic curves:

– One curve uses the corresponding first and second point as endpoints and the third point as control point. In the case of the letter D the three points are $P1$, $P2$ and $P3$, respectively, for the letter C the corresponding points are $P1'$, $P2'$ and $P3'$, respectively.

– The other quadratic curve of each letter has the corresponding first and fourth point as endpoints and the corresponding fifth point as control point.

In order to continuously transform the two objects—in this case the letters D and C—into each other, convex combinations are applied again. Instead of having convex combinations of transformations as in the previous example of the ellipses, here convex combinations between pairs of corresponding points P_i and P_i' are considered.

$$P_i^{(\alpha)} = (1-\alpha) \cdot P_i + \alpha \cdot P_i'.$$

For an intermediate image $\alpha \in [0,1]$ the corresponding lines or curves are now drawn on the basis of the points $P_i^{(\alpha)}$. In the example of transforming the letter D into the letter C one would draw two quadratic curves, one defined by the points $P_1^{(\alpha)}$, $P_2^{(\alpha)}$ and $P_3^{(\alpha)}$, the other defined by the points $P_4^{(\alpha)}$, $P_5^{(\alpha)}$ and $P_3^{(\alpha)}$.

Figure 2.23 shows intermediate images obtained for the convex combinations with $\alpha = 0, 0.2, 0.4, 0.6, 0.8, 1$ based on the points and the corresponding quadratic curves as illustrated in figure 2.22.

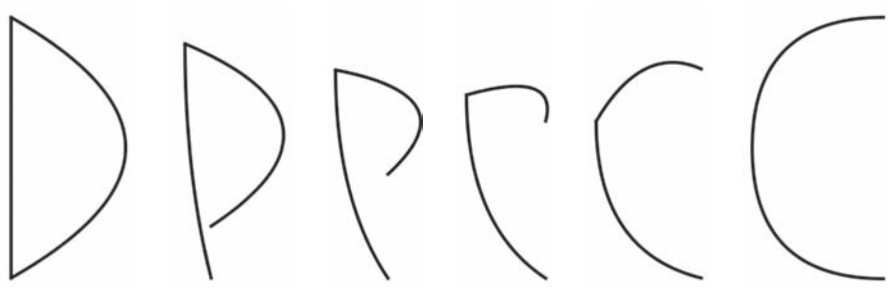


Figure 2.23 Stepwise transformation of two letters into each other

Further applications of interpolators in connection with colours and raster graphics will be introduced in section 4.7.

2.12 Implementation of interpolators in Java 2D

This sections explains in more detail how the two techniques for interpolators introduced in the previous section can be implemented in Java 2D.

The first example of a continuous transition from one ellipse to another as illustrated in figure 2.21 is realised in the class `ConvexCombTransforms.java`. In the first part of the program the basic ellipse `elli` and two affine transformations `initialTransform` and `finalTransform` are defined. The two transformations transform the basic ellipse into the initial ellipse in the beginning of the animation and the final ellipse at the end of the animation. In order to compute the convex combinations of the two transformations the corresponding matrices are required. They are obtained by applying the method `getMatrix` to `initialTransform` and `finalTransform`.

```
double[] initialMatrix = new double[6];
initialTransform.getMatrix(initialMatrix);
```

```
  double[] finalMatrix = new double[6];
  finalTransform.getMatrix(finalMatrix);
```

The coefficients of the two matrices are stored in the one-dimensional arrays `initialMatrix` and `finalMatrix` according to the representation of transformation matrices (2.1) on page 30. The intermediate images are generated in a loop where in each step a new convex combination of the two arrays is computed. The arrays are treated in the same way as vectors[5] so that their convex combination yields an array of the same length. The elements of this new array can again be interpreted as the coefficients of a transformation matrix in homogeneous coordinates. This transformation is then applied to the basic ellipse and in each step of the loop the resulting transformed ellipse is drawn.

The transformation of the letter D into the letter C is implemented in the class `DToCMorphing.java`. Figure 2.22 showing the initial state—the letter D—and the final state—the letter C—was generated by the classes `SimpleLetterD.java` and `SimpleLetterC.java`. For the transformation of the two letters into each other, two arrays are defined for each letter, one array for the x-coordinates of the control points and one array for the y-coordinates. Another two arrays are needed for the computation of the convex combinations of the control points. In each step of the loop the new convex combination is computed and the computed control points are used to draw the corresponding quadratic curves to generate the corresponding intermediate image.

2.13 Single or double precision

For longer animated graphics with moving objects a large number of transformations have to be applied successively. This means that a large number of matrix multiplications must be carried out. Although the roundoff error for a single matrix multiplication might be negligible, roundoff errors can accumulate over time and might lead to undesired effects. In most cases such roundoff errors will be noticeable in the graphics to be drawn since the numerical computations to be carried out in computer graphics are usually not critical from the numerical point of view. Inverting a matrix to reverse a transformation is an example for an exception where roundoff errors might have serious effects on the graphics, when the matrix is badly conditioned. But most of the calculations in computer graphics do not encounter such problems.

For illustration purposes the example of the second hand of a clock is considered. The hand is 100 units or pixels long. The tip of the hand is at the point

[5] Vectors in the mathematical sense, not as the class `Vector` in Java.

time	x	y
	`double`	
1 minute	99.99999999999973	−4.8572257327350600E-14
2 minutes	99.99999999999939	−9.2981178312356860E-14
3 minutes	99.99999999999906	−1.3739009929736312E-13
4 minutes	99.99999999999876	−1.4571677198205180E-13
5 minutes	99.99999999999857	−2.2204460492503130E-13
6 minutes	99.99999999999829	−2.9143354396410360E-13
7 minutes	99.99999999999803	−3.1641356201816960E-13
8 minutes	99.99999999999771	−3.7331249203020890E-13
9 minutes	99.99999999999747	−4.2604808569990380E-13
10 minutes	99.99999999999715	−4.5657921887709560E-13
8 hours	99.99999999986587	−2.9524993561125257E-11
	`float`	
1 minute	100.00008	−1.1175871E-5
2 minutes	100.00020	−1.4901161E-5
3 minutes	100.00032	−1.8626451E-5
4 minutes	100.00044	−1.1920929E-5
5 minutes	100.00056	−8.9406970E-6
6 minutes	100.00068	−3.1292439E-5
7 minutes	100.00085	−5.3644180E-5
8 minutes	100.00100	−7.2270630E-5
9 minutes	100.00108	−8.0466270E-5
10 minutes	100.00113	−8.4191560E-5
8 hours	100.00328	−1.9669533E-4

Table 2.1 Effects of roundoff errors

$(100, 0)$ in the beginning. The hand is rotated clockwise around the origin by 6° per second. This means that the transformation $R(-6°)$ in terms of a rotation matrix is applied each time. After every full minute—after 60 multiplications of the rotations matrix by itself—the hand should return to its original position.

Table 2.1 shows the computed positions of the tip of the hand after various time intervals using double (`double`) and single (`float`) precision. In both cases the roundoff errors are negligible, especially when taking into account that drawing in raster graphics will require rounding to integer values in the end anyway. Even after eight hours demanding 28800 matrix multiplications, single precision will still be sufficient to obtain the exact values in terms of raster graphics. This is only valid if the accumulated rotation is applied to the hand in its initial position or if the new position of the hand is stored in vector graphics, i.e., using floating point arithmetic, and every second a single rotation by $6°$ is applied to the updated position of the hand. If the coordinates of the hand are stored in raster graphics using only integer values and a single rotation by $6°$ is applied to the updated hand in pixel coordinates every second, already after one minute a wrong position of $(95, -2)$ instead of $(100, 0)$ is calculated.

Although computations with double precision values are less error-prone, the accuracy of single precision is sufficient for most applications in computer graphics taking into account that raster graphics will require rounding numbers to integer values in the end, so that numerical errors less than 0.5 are invisible anyway.

Especially for three-dimensional scenes with complex objects a very large number of points is needed to define the objects. In this case the memory requirements very often have a higher priority and single precision is preferred over double precision in order to reduce the amount of memory needed for storing the objects.

2.14 Exercises

Exercise 2.1

Use a `GeneralPath` to draw a rectangle with rounded corners.

Exercise 2.2

A simple two-dimensional solar system model with one sun and one planet should be animated. The centre of the sun is located in the origin of the coordinate system. The spherical planet with a radius of 10 units rotates anticlockwise around the sun on a circular orbit with constant speed. The radius of the planet's orbit (the distance between the centres of the sun and the planet) is 200 units. In the beginning of the animation the centre of the planet is located at the point $(200, 0)$. During one rotation around the sun, the planet rotates 365 times anticlockwise around its own axes. Consider the point on the planet that is closest to the sun in the beginning of the animation. Use geometric transformations to describe where the point will be located after the planet has finished one third of its orbit.

Exercise 2.3

Choose the constant c in the matrix

$$\begin{pmatrix} c & 0 & 6 \\ 0 & c & 4 \\ 0 & 0 & c \end{pmatrix}$$

in such a way that the matrix represents a translation by the vector $(3, 2)^\top$ in homogeneous coordinates.

Exercise 2.4

Use Java 2D for an animation illustrating the movement of the point in exercise 2.2.

Exercise 2.5

Use Java 2D to animate a beating heart that moves along a line in a window on the computer screen.

Exercise 2.6

Apply the technique for transforming one letter into another as illustrated in figure 2.22 for the letters D and C to other letters, for instance your initials.