Diss. ETH Nr. 13778

A Generic 2D Graphics API with Object Framework and Applications

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZÜRICH
(ETH Zürich)

for the degree of
Doctor of Technical Sciences

presented by
Erich Oswald
Dipl. Informatik-Ing. ETH
born March 19, 1969
citizen of Winterthur
and Richterswil, Zürich

accepted on the recommendation of
Prof. Dr. Jürg Gutknecht, examiner
Prof. Dr. Markus Gross, co-examiner

2000

Acknowledgements

I am indebted to many people without whose help and support I could never have reached a successful conclusion to this project.

First and foremost, I would like to thank Prof. Jürg Gutknecht for his patient and liberal supervision. Although he never intervened in my work, he would always make sure I was progressing in a meaningful direction. His talent to envision solutions that go beyond current trends is remarkable.

My sincerest thanks also go to Prof. Markus Gross, who readily agreed to co-supervise this thesis and whose advice and remarks considerably helped me consolidate many graphical terms and algorithmic details.

Due to the joint efforts of many current and former members of Prof. Gutknecht's research group, Oberon System 3 has become a very stable and reliable platform for developing software. I would in particular like to thank Emil Zeller, Pieter Muller, Patrik Reali, André Fischer, Hannes Marais, Patrick Saladin, Markus Dätwyler, Andreas Disteli, Ralph Sommerer, and Karl Rege for their contributions to the system, but even more so for their company and the many moments of joy and hilarity they have shared with me.

At the same time, I would like to extend my gratitude to our colleagues on the south side of the elevator. All of them have brought color and good vibes to the countless coffee breaks, lunches, seminar talks, skating expeditions, assistant evenings, fondue parties, and other social events.

Special thanks go to Jürg Bolliger for proofreading the first draft of the manuscript and helping me improve my writing, to Patrik Reali and Pieter Muller for their constant feedback on Leonardo, to Hans Domjan for dragging me into the ACM programming contest business and the resulting trips to San José, Bratislava, and Ulm, and to Erwin Oertli for regularly organizing the "Assistentenabend" and various special events (not to forget our discussions about music, cooking, and computer games).

My dearest thanks go to Marlene Wegmann Oswald for her love, understanding, tolerance, and encouragement, and to my parents, Peter and Doris Oswald, for their everlasting love and support.

Contents

Abstract

Screens and printers have remained the primary output devices of every computer, which is why the field of 2D graphics still plays a central role in today's computer systems. For operating systems, functionality for rendering general graphics in two dimensions has been a matter of course for a long time. Considering innovation, however, it seems that a certain plateau has been reached. Most graphical programming interfaces have been around for many years; they were designed at a time when object-oriented programming was in its infancy. Thus, they offer a fixed set of definitions and procedures, which can only be reused as a whole and which cannot be extended.

In this thesis, we examine the concepts behind common 2D graphics interfaces. We extract those features that we consider essential and combine them with the principles of object-oriented and extensible programming to synthesize a new programming interface. On top of this interface, we implement an object framework that models structure and behavior of graphical scenes. To prove the feasibility of our design, we use programming interface and object framework in two applications, an interactive graphics editor and a graphical description language.

Our contributions to the field include a novel approach for specifying general paths made from lines and curves, an innovative pen concept to paint such paths in various styles, and working implementations that show how these ideas work out in practice. Due to the abstract nature of the graphical contexts in our programming interface, components that integrate graphics within existing environments are light-weight and can be implemented with little effort. In addition, objects from otherwise independent applications can be integrated within another if they rely on these graphical contexts to draw themselves.

Kurzfassung

Bildschirme und Drucker sind die wichtigsten Ausgabegeräte aller Computer geblieben, weshalb das Gebiet der 2D-Grafik auch in heutigen Computersystemen noch eine zentrale Bedeutung innehat. Für Betriebssysteme ist Funktionalität zur Darstellung von allgemeinen Grafiken seit längerem eine Selbstverständlichkeit. Bezüglich Innovation scheint allerdings eine gewisse Abflachung eingetreten zu sein. Die meisten Grafik-Programmierschnittstellen bestehen seit vielen Jahren, wurden also zu einer Zeit entworfen, als die objektorientierte Programmierung noch in ihren Kinderschuhen steckte. Aus diesem Grund bieten sie nur eine beschränkte Menge von Definitionen und Operationen an, welche nur als Ganzes wiederverwendet und nicht erweitert werden kann.

In dieser Arbeit untersuchen wir die Konzepte, auf denen gängige 2D-Grafikschnittstellen aufbauen. Wir reduzieren diese auf diejenigen Eigenschaften, die wir für essentiell halten, und kombinieren sie mit den Prinzipien der objektorientierten und erweiterbaren Programmierung, um daraus eine neue Programmierschnittstelle zu schaffen. Aufbauend auf dieser Schnittstelle implementieren wir ein Objekt-Framework, welches Struktur und Verhalten von grafischen Szenen modelliert. Um zu zeigen, dass unser Entwurf der Problemstellung angemessen ist, verwenden wir Programmierschnittstelle und Framework in zwei Anwendungen, einem interaktiven Grafikeditor und einer grafischen Beschreibungssprache.

Unser Beitrag zum gewählten Gebiet umfasst einen neuartigen Ansatz, um aus Linien und Kurven bestehende Pfade zu spezifizieren, ein innovatives Stift-Konzept, um diese auf verschiedene Arten zu zeichnen, und lauffähige Programme, welche zeigen, wie sich unsere Ideen in der Praxis bewähren. Die grafischen Kontextobjekte in unserer Programmierschnittstelle sind von abstrakter Art; damit werden Komponenten zur Integration von Grafiken in bestehenden Umgebungen leichtgewichtig und mit wenig Aufwand realisierbar. Ausserdem können Objekte von anderweitig unabhängigen Anwendungen gegenseitig ineinander eingebettet werden, sofern sie sich auf obige grafischen Kontextobjekte abstützen, um sich selber zu zeichnen.

CHAPTER 1

Introduction

1.1 Motivation

Most computers today communicate their results to human users by displaying them on a screen or printing them on paper. One of the primary goals of every operating system is therefore to provide application programmers with programming interfaces that allow them to describe graphical objects and paint those on appropriate output devices.

Every major operating system thus includes capabilities for rendering graphics in two dimensions, but with different levels of sophistication. They all offer functionality for rendering basic objects such as lines, rectangles, text, and bitmap images; some of them also incorporate advanced concepts such as a general path model, arbitrary clipping regions, or user-defined coordinate systems.

Most of these systems have been in use for years. As a result, their graphical capabilities display a high degree of maturity and robustness, but also signs of stagnation. In order not to break compatibility with existing applications, further development on these systems is marginal. Besides, replacing or complementing them with new programming interfaces is not easy because customers may be reluctant to adapt to new technology. Most of these interfaces are thus based on a procedural library design and do not take advantage of the benefits that object-oriented and component-oriented software technologies offer. In particular, they cannot be extended by clients, limiting their potential for reuse.

In this thesis, we take advantage of our academic freedom to explore alternatives to the current status quo. We propose concepts and ideas on

which future graphics interfaces could be built and back them up with actual implementations.

1.2 Contributions

With our work, we show how the principles of object-oriented software engineering — in particular, modularity, polymorphism, and extensibility — can be applied to software packages that implement 2D graphics and how this reflects on applications that rely on them.

On a conceptual level, we deliver a thorough examination of what constitutes a graphics interface. We also present an innovative approach, which we call *early rendering*, for specifying general geometric paths that are made from lines and curves. Early rendering allows an implementation to render parts of a complex path before it knows the path completely, whereas traditional approaches require each path to be entirely known before rendering can actually start.

On the system software level, we present a modular application programming interface (API), based on our early rendering model, for drawing advanced 2D graphics on abstract output devices. Graphics can be directed to arbitrary concrete output devices, from physical devices (such as screens and printers) to virtual devices (such as raster images and disk files). We supplement this general programming interface with an extensible shape framework that provides its clients with suitable abstractions for modeling structure, behavior, and graphical appearance of its objects. For the latter, we introduce the novel concept of abstract *pen objects*. These decouple application objects from the potentially limited graphical capabilities of an underlying programming interface. Pen objects can be arranged in persistent, extensible object hierarchies, thereby providing their clients with unmatched flexibility for modeling the graphical appearance of geometric paths.

On the application level, we use our programming interface and our framework within two distinct applications, an interactive graphics editor and a graphical description language, thus proving their feasibility in practice. With these applications, we show that early rendering works well in various environments; we show how an interactive application that relies on models, views, and controllers can deal with extensions in its model; and we show

how abstract graphical contexts let clients integrate graphics as components within different environments.

1.3 Overview

Chapter 2 introduces concepts, terminology, and features of common graphics interfaces. A series of well-known graphics libraries and graphics languages are examined and compared to gain deeper insight into the domain of 2D graphics.

Chapter 3 presents Gfx, a new programming interface for rendering graphics on abstract context objects.

Chapter 4 describes the Leonardo shape framework, which provides applications with an abstract model for representing structure and behavior of graphical scenes.

Chapter 5 discusses Leonardo, an application for editing graphical scenes interactively. Leonardo uses the Gfx API and the Leonardo shape framework that are introduced in Chapters 3 and 4.

Chapter 6 presents Vinci, a graphical description language. Vinci is a second application of Gfx and the Leonardo shape framework. Due to the abstract nature of Gfx contexts, Leonardo objects and Vinci descriptions can be mutually integrated within another.

Chapter 7 draws conclusions about what has been achieved and discusses future perspectives.

Appendix A contains a commented example of a non-trivial Vinci shape.

Appendix B describes the complete syntax of Vinci.

Appendix C lists the modules that make up Gfx, Leonardo, and Vinci, along with their sizes, and analyzes the performance of our software.

Related work is to a major part discussed in Chapter 2. Later chapters supplement this with further citations of related work that specifically apply to the topics that are covered in these chapters.

CHAPTER 2

Concepts of 2D Graphics

Ever since Sutherland's ground-breaking Sketchpad system in 1963 [77], software packages for creating 2D graphics have been relying on a few basic concepts to describe graphical scenes. We first establish some central terms, followed by a closer examination of these concepts in Sections 2.1 and 2.2. We use our findings to evaluate several examples of related work in Section 2.3, including historical landmark systems as well as modern candidates representing state of the art technology.

Graphics Interfaces consist of procedures and data structures for creating graphical output. Most graphics interfaces manifest themselves in the form of application programming interfaces (API). Their executable parts reside in code libraries and are statically or dynamically linked to those of an application. To broaden our perspective on 2D graphics, we extend our definition of a graphics interface to also include special purpose languages. These convert textual input to graphical output by interpreting or compiling their input. The applications and programmers that utilize a graphics interface are called its *clients*.

Output Devices. The majority of physical output devices today are based on the raster paradigm. A raster device comprises an array of individually addressable adjacent raster cells, each of which covers a rectangular area of uniform size. These raster cells are called *pixels* (an abbreviation for "picture elements") and display a uniform shade of color each. A *frame buffer*, which contains the contents of a workstation's screen, typically assigns a multiple of eight bits to each pixel, making each pixel start at a byte address.

In addition to color, per-pixel information such as depth or transparency may also be stored in the frame buffer. Other examples of raster devices include fax modems and various kinds of printers, including matrix, ink-jet, and laser printers. Non-raster devices include vector devices such as pen plotters or older CRT monitors. These have become rare but are still used in a few special places.

In addition to *physical output devices*, many graphics interfaces also support *logical output devices*. A logical output device stores a picture of the produced graphics which the same or another client can later access again. Examples of logical output devices are raster images (in memory or on disk) and disk files that contain raster or vector graphics in a standard file format.

Because the majority of output devices are indeed raster devices, to paint an object usually requires that a graphics interface computes the set of pixels that best approximate the object's geometry (in a process called *scan conversion*) and assigns a color value to them. The graphics interface may choose to hide all details concerning resolution and bit-depth of the underlying hardware from its clients. The same interface can then be used for writing to many physical and logical devices, irrespective of whether they rely on raster or vector technology. Clients only need to be assured that the graphics interface will approximate all graphical objects as exactly as an output device allows.

Graphical Contexts. Graphics interfaces are usually based on an implicit or explicit *context*, where the current output device, visible output area, coordinate system, and graphical attributes are managed (see Section 2.1). Within this context, the graphics interface renders *graphical objects* (see Section 2.2). Normally, to render an object means to paint an appropriate representation on an output device. Other forms of rendering affect the state of the context instead of producing output, for example by restricting future drawing operations to the rendered object's interior.

2.1 Graphical Contexts

Independent of the types of graphical objects that it supports, a graphics interface needs to define and manage a context within which rendering takes place. Such a context is responsible for defining a rendering model

(see Section 2.1.1), a coordinate system (see Section 2.1.2), an imaging model (see Section 2.1.3), and a clip area (see Section 2.1.4).

2.1.1 Rendering Model

The rendering model of a graphics interface defines how graphical objects are described and how they are processed. It determines how clients specify objects and how they pass them to the interface.

Immediate Mode. To start with an example, consider a simple scenario in which a client wants to draw a line from point *A* to point *B*. A typical graphics API will offer an operation called **DrawLine** or similar which takes the *x* and *y* coordinates of points *A* and *B* as arguments and draws a straight line between these points. As soon as **DrawLine** returns, its caller can assume that the line has been completely rendered, which is why this mode of operation is known as *immediate mode*.

In immediate mode, no explicit data structures need to be built. Each graphical object is modeled by calling a procedure and passing it all parameters that are necessary to describe the object's geometry. Despite this low level of abstraction, immediate mode is the standard rendering model used in most graphics interfaces, due to its simplicity and efficiency.

To decrease the number of arguments that must be passed to rendering procedures, various decorative attributes that are common to several kinds of objects, such as color, line width, and dash pattern, are often maintained by the graphical context in a so-called *graphics state*. Clients call procedures to assign new values to these attributes. These values stay in effect until a client changes them once more. The graphics interface can thus be viewed as a state machine on whose internal state all drawing operations depend. As a consequence, clients have to ensure that state variables have appropriate values before they can draw an object. Still, this convention is usually acceptable because most of the time clients use the same attribute values for rendering multiple graphical objects. Besides, a graphics interface may enable its clients to revert all values to a well-defined default state or to a previously stored state to minimize the number of required setup operations.

Internal Data Structures (Display Lists). Some applications may have to redraw a graphical scene at regular intervals or may want to repeatedly

draw simple structures at various locations in the scene. Many graphics interfaces can therefore be put in a special recording mode where procedure invocations do not produce any output. Instead, an internal data structure is created to which all graphical objects are added. This internal data structure, traditionally called *display list*, can later be instantiated repeatedly whenever needed. The term originates from the time when the display processor that controlled the electron beam of a vector display indeed traversed a display list in memory at regular intervals to refresh the image on the screen. Display lists enable a graphics interface to preprocess graphical objects before storing them in the list. Such pre-compilation can improve rendering performance when the list is later instantiated repeatedly. Examples of graphics interfaces that feature display lists are GKS (see Section 2.3.1) and QuickDraw (see Section 2.3.2).

Since several different display lists can exist concurrently, clients use a reference number or pointer to identify the one which they want redrawn. This allows clients to treat possibly complex structures in an abstract manner. Because the same procedural interface as for immediate rendering is used, output from a complex rendering operation can be redirected to a display list merely by first putting the interface in recording mode.

To distinguish immediate rendering models from rendering models that defer actual output to a later, user-defined time, the latter are often said to work in *retained mode*, especially in the domain of 3D graphics.

Exported Data Structures (Object Structures). Taking the idea of display lists even further, a data structure for storing objects may become a public feature of the graphics interface, which is especially attractive within an object-oriented system. There, the graphics system can export an abstract render protocol and an abstract object type, preferably also a few standard object extensions which implement that protocol. The graphics interface renders objects by using the render protocol to query them about their geometry. It decomposes objects into simple standard objects such as lines and arcs; these are then easy to draw. Clients can either create instances of standard object types or of custom type extensions that match their specific needs. An example of such an object framework is Java 2D, to be examined in more detail in Section 2.3.7.

An object-based approach offers a high level of abstraction and flexibility and facilitates the construction of complex object hierarchies. However, a

graphics system that is based on polymorphic object structures alone is slower than one with a fixed number of object types. The main reason is that a general rendering protocol cannot take advantage of many small optimizations that are possible with a fixed set of primitives because details about client-defined objects are not known inside the graphics interface and thus cannot be exploited to speed up rendering.

2.1.2 Coordinate Systems

A coordinate system defines how abstract points, specified by their x and y coordinates, are mapped into geometric locations. While simple graphics interfaces always work in the coordinate system of their output device, advanced interfaces provide a device-independent coordinate system or even let clients define a custom coordinate space.

Device Coordinates. The most basic coordinate system is the *device coordinate system*, whose points correspond to concrete locations in the output plane. For a raster device, these locations are its pixels. For example, the pixel in the bottom left (or top left) corner has coordinates (0, 0), the one to its right (1, 0), and the one above (below) it (0, 1).

Specifying graphical objects directly in device coordinates is simple and efficient because integer coordinates can be used. For example, the Oberon system (see Section 2.3.3) uses device coordinates exclusively. However, the fact that the whole area of a pixel is represented by a single point introduces an inconsistency; namely that the exact position of a point within a pixel is unclear. For drawing thin lines, it seems natural to place its end points at pixel centers and to paint the pixels in between that are closest to the line (for example using Bresenham's famous algorithm [16]), as shown in Figure 2.1. However, if the same method is used to draw a filled rectangle of width w and height h, with corners at pixel centers, the resulting rectangle extends beyond each of its sides by half a pixel, increasing its effective width to $(w + 1)$ and height to $(h + 1)$ pixels, as displayed in Figure 2.2. The solution is in this case to place the rectangle's corners on corners of the pixel grid. However, this introduces the new problem that filled and outlined rectangles no longer have the same size. Stamm examines this problem in more detail in [71] and [72], concluding that all coordinates should be placed on the grid lines between pixels. Stamm's approach works well in

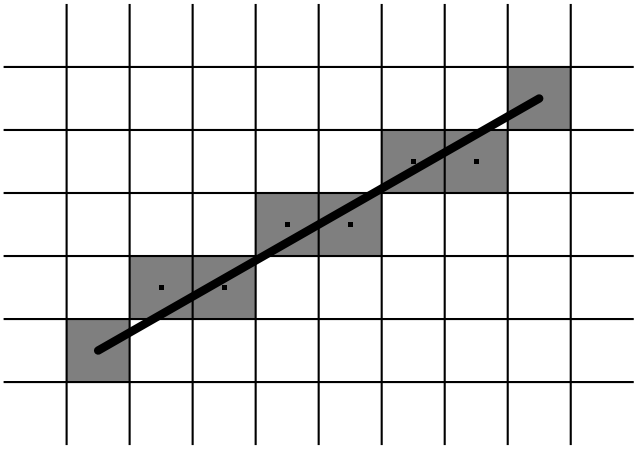

Figure 2.1: Drawing a thin line

the domain of font design, where all graphical objects are part of a closed contour, but is harder to defend in a more general context.

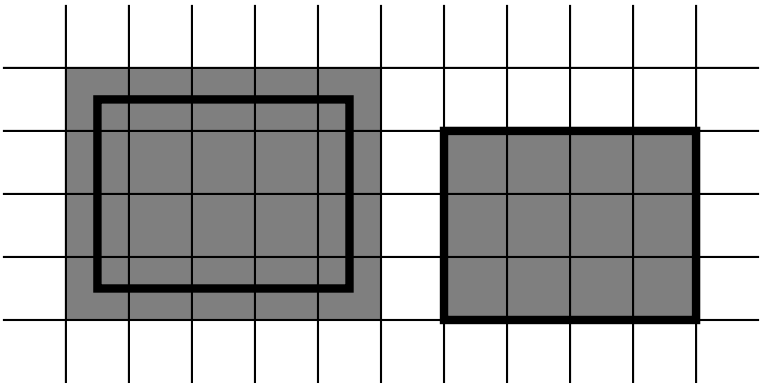

Figure 2.2: Filled rectangle with corners on pixel centers (left) and on pixel grid (right)

Sub-pixel Coordinates. Even if the graphics interface identifies integer coordinates with well-defined and consistent locations in pixel space, there are still situations that cannot be modeled with integer device coordinates at all. Consider the scenario that is depicted in Figure 2.3, which shows a zoomed view of an area where three adjacent polygons meet. Point *A* is supposed to lie on the line between points *B* and *C*. When the exact

coordinates of A are rounded to integer coordinates, it is possible that this condition cannot be met. Depending on the order in which the polygons are drawn and to which grid point A, B, and C are rounded, the resulting picture may have a discontinuity in the line from B to C, or parts of the background may be visible near A. In the displayed scenario, A is rounded towards A' in the upper left corner of the corresponding pixel, whereas B and C are rounded towards B' and C' in the lower right corner of the pixels that contain them. This has the consequence that several pixels along the shared boundary do not get painted even if all three polygons are filled.

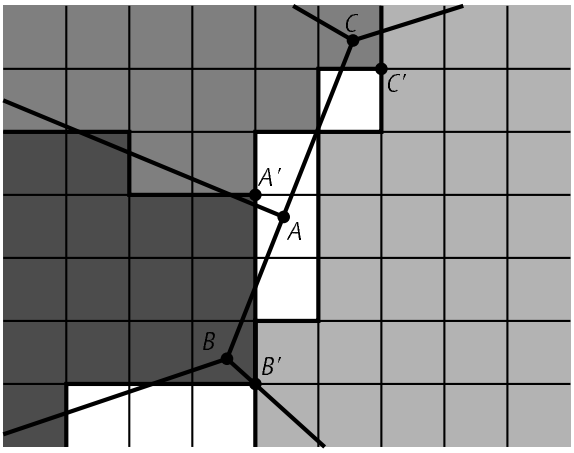

Figure 2.3: Rounding problems due to use of integer coordinates

The only solution for avoiding such artifacts is to introduce sub-pixel precision for point coordinates, either by addressing fractional pixels or by switching to floating point numbers for representing coordinates. Both approaches do not actually solve the problem since even the resolution of floating point numbers is finite and therefore subject to rounding errors. Nevertheless the reduction in scale reduces the chances of visible artifacts significantly. Older graphics interfaces (for example QuickDraw, discussed in Section 2.3.2) avoid using floating point because floating point arithmetic used to be several times slower than integer or fixed point arithmetic. However, with today's super-scalar and pipelined processor architectures this difference is disappearing. On many processors, built-in floating point calculations execute faster than emulated fixed point calculations. This makes floating point the natural choice for representing point coordinates,

at least in a general graphics interface that is not bound to a particular output device.

Normalized Coordinates. If objects are defined in device coordinates, the size of the resulting pictures depends on the pixel size of the corresponding device. For example, a hundred pixels measure about three centimeters on a typical screen, but less than half a centimeter on a standard laser printer. To overcome such device-dependent behavior, a graphics interface can introduce a *normalized coordinate system* with arbitrary but fixed unit size. It can then map graphical objects from normalized to device coordinates (and vice versa) by scaling and translating them accordingly. The required scale factor and translation vector still depend on pixel size and printable area of the output device, but only need to be initialized once and then become an aspect of the graphics context. As a side effect, the interface is able to deal with non-square pixels as well since the pixel aspect ratio can be part of the scale factor that maps normalized to device coordinates. An example of a graphics interface that provides normalized coordinates is GKS (see Section 2.3.1).

User Coordinates. Several graphics interfaces allow clients to define arbitrary *user coordinate systems* on top of the normalized coordinate system. User coordinates are essential to hierarchical modeling since they let clients render parts of a scene at different locations, with different orientations, and in different sizes.

The de facto standard for representing user coordinate systems is based on homogeneous coordinates and transformation matrices. An introduction to homogeneous coordinates can be found in [27], which in turn refers to [52, 53, 69, 12]. In homogeneous coordinates, each point (x, y) is augmented by a third coordinate and becomes (wx, wy, w), one of many points on a line in 3D. Similarly, division by w projects each point in homogeneous space back to two dimensions. What makes homogeneous coordinates interesting is that affine transformations in 2D — consisting of arbitrary combinations of translation, rotation, scale, and shear transformations — can be expressed as linear transformations in 3D and can therefore be stored in 3×3 matrices. A matrix-based transformation is applied to a point by multiplying the matrix with a homogeneous representation of the point (usually with w equal to one) and projecting the result back to two dimensions. For example,

translation by (t_x, t_y) is expressed as follows:

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

Similarly, rotation with angle ϕ, scale with factors s_x and s_y, and shear transformations with factor f are achieved with the following transformation matrices.

rotation	scale	shear along x-axis
$\begin{pmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ f & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

With affine transformations, the third column in any matrix is always equal to $(0, 0, 1)^T$. Unless a graphics interface intends to support perspective transformations as well, it need not store the third column explicitly. Postscript (see Section 2.3.4), for example, therefore uses vectors with six elements to store transformation matrices.

Individual transformations can be combined by concatenating the corresponding transformation matrices. A single matrix is thus sufficient to store arbitrary combinations of transformations. This is of special interest for clients that organize their data as hierarchies of nodes. For those nodes that change the local coordinate system of their descendants, the local node matrix is concatenated with the matrix that represents the current coordinate system before any descendants are visited. Thus, descendants are free to express all their coordinates in local *object coordinates*. The matrices of their parent nodes convert local to global coordinates. To streamline hierarchical modeling even further, some graphics interfaces (e. g. Postscript, see Section 2.3.4) provide their clients with a *matrix stack* onto which they can push the current matrix value before adjusting it and off which they can later pop it again.

2.1.3 Imaging Model

After a graphical object has been reduced to primitive elements, such as pixels or pen strokes, these elements are visualized on the output device

that the graphical context controls. Consider a context that drives a raster device and thus converts graphical objects to sets of pixels. It must combine the color values of these pixels with those of the corresponding pixels of its output device, for example by replacing existing color values with those of the converted object. In general, a context's *imaging model* defines how color values (and possibly other values) from a source (e. g. scan-converted pixels) affect the current color values of a destination (e. g. frame buffer pixels), preferably without explicit reference to the physical representation of these values.

Overdraw. Taking into account that the information provided by a source is primarily its color value, the basic method for reproducing the source consists of copying the source color value to the destination, replacing whatever value was there before. The corresponding imaging model is that of painting objects with opaque color in the output plane. The model works well for a large number of graphical objects and output devices and can be implemented efficiently. An example of a graphics interface that uses overdraw exclusively in its imaging model is Postscript (see Section 2.3.4).

Alpha Blending. Things get more complicated if we tolerate that source and destination need not be fully opaque. Painting an object with partly transparent color lets part of the previous destination color shine through from under the object. In computer graphics, opacity is usually called *alpha*, and the process of applying a partly transparent source to a partly transparent destination is called *alpha blending* or simply *blending*.

A special case of alpha blending is used for painting bitmap patterns, where each pixel in the bitmap is either fully transparent or fully opaque. Such a bitmap mask restricts painting to the area where it is opaque; only there the corresponding destination pixels are overwritten with new color values. Bitmap masks are commonly used for rendering text, where individual character patterns are represented as bitmap masks. Even interfaces that otherwise only support overdraw mode may allow clients to paint bitmap masks.

More general applications of alpha blending in 2D graphics, as supported for example in SVG (see Section 2.3.6) or Java 2D (see Section 2.3.7), include raster image compositing [66] and smoothing jagged edges: when for example a filled polygon is painted in a frame buffer with low reso-

lution, its boundary has a jagged or "staircase" appearance, as displayed in Figure 2.4. This effect is a manifestation of an aliasing problem. Alias-

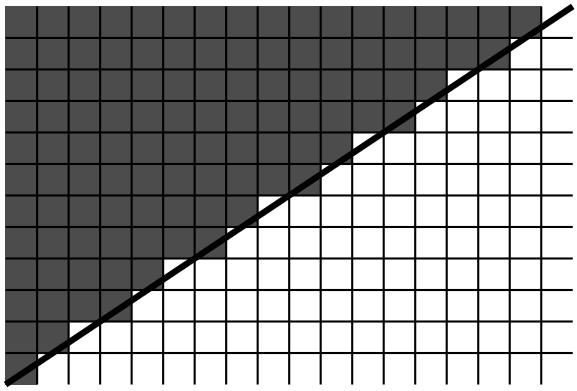

Figure 2.4: Jagged edge on low-resolution display

ing problems frequently appear in computer graphics [19, 20]; they arise because continuous signals (the color value of a horizontal scanline) are sampled at regular intervals (at the center of each pixel). By assigning each source pixel an alpha value that is proportional to how much of the pixel is inside the polygon and blending the resulting pixels with the destination, the resulting edges appear much smoother, as shown in Figure 2.5. In mathematical terms, this corresponds to a convolution of the polygon with a box filter over each pixel.

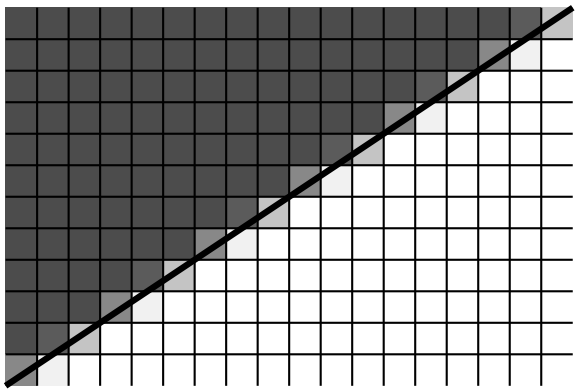

Figure 2.5: Smoothing jagged edges with partly transparent pixels

Note that general blending operations are not possible on all output

devices because not all of them provide read access to destination color and alpha values. For example, consider printing a page, which usually requires that a stream of drawing commands is sent to a printer over a serial or network connection. The graphics interface cannot determine the destination color and alpha value for every rendering command unless it simulates the printing process in an off-screen image itself. However, such a simulation is often not practical because it may consume large amounts of memory and slow down the graphics system.

Invert Mode. Especially for temporarily drawing objects in a frame buffer, another imaging model, called *xor* or *invert* mode, is in widespread use. Its principle is to modify the destination in such a way that the modification can easily be undone again by drawing everything a second time. Its main application is an interactive technique called *rubber-banding*, which simulates animated objects by quickly drawing and erasing them at different locations. Rubber-banding is frequently used for giving feedback to a user who moves or reshapes objects interactively [60, 27]. The traditional implementation of invert mode replaces all bits in the destination by their unary complement since complementing all bits a second time restores their original state.

2.1.4 Clipping

The area where output can be produced on a physical device is limited. It would therefore be useless to spend computing resources for processing graphical objects that fall completely outside the displayable area. Besides, when rendering on a shared device such as the display, output must be restricted to the area that the graphics interface has been assigned by the window manager of the operating system. Otherwise, an application might draw over and hence destroy some other program's screen estate. To avoid such conflicts, graphics interfaces keep track of the area where output is permitted to appear. They discard graphical objects that lie completely outside this area and *clip* objects that overlap its boundary by cutting off and not painting the parts outside it.

The problem of clipping lines and polygons against rectangles or polygons parametrically has been studied by many researchers [60, 78, 21, 49, 13]. However, although the visible area of a physical device is in fact often bounded by a rectangle, clip areas are frequently more complex than simple

rectangles, especially on systems that use a desktop metaphor with overlapping windows in their user interface. If parts of a window are obscured by other windows, its clip area must be shaped accordingly. To deal with such situations, a graphics interface needs to provide clip areas that can either be rectangles or boolean combinations of existing clip areas, possible combinations being union, intersection, and difference.

Some graphics interfaces allow their clients to further customize clip areas, for example to restrict output to the interior of a circle, which is tedious to specify as a combination of rectangles. Therefore, a graphics interface may offer a special rendering mode which intersects the boundary of rendered objects with the existing clip area instead of painting them. Successive output appears only within this user-defined area until the original clip area is restored again. User-defined clip areas were already available with QuickDraw (see Section 2.3.2).

To implement such generic clip areas, a general shape algebra that can apply boolean set operations to all basic shapes, not only rectangles, is required. This raises the question of how generic clip areas can be stored efficiently in a data structure. One approach is to use *regions* [24, 27]. Regions are parametric structures, organized for instance as sets of scanlines, each scanline in turn consisting of a set of horizontal spans of equal height. Figure 2.6 illustrates how a region can be approximated with horizontal

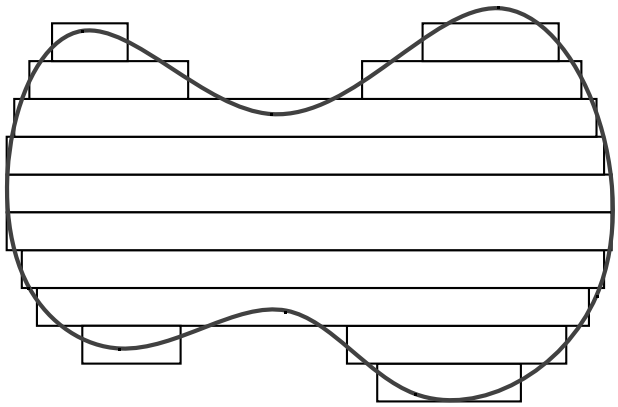

Figure 2.6: Horizontal spans approximating a region

spans. While regions are adequate for representing clip areas that result from combinations of few primitive shapes, they are less efficient for representing

areas with "fuzzy" contours such as irregular objects that have been extracted from photographic images. For objects with such detailed boundaries, the use of *stencil buffers*, which are bitmap masks that store at least one bit of alpha information per pixel, may be preferable. The visible area is then defined by the union of all transparent pixel in the stencil buffer (see Figure 2.7). Stencil buffers are often supported by the display hardware,

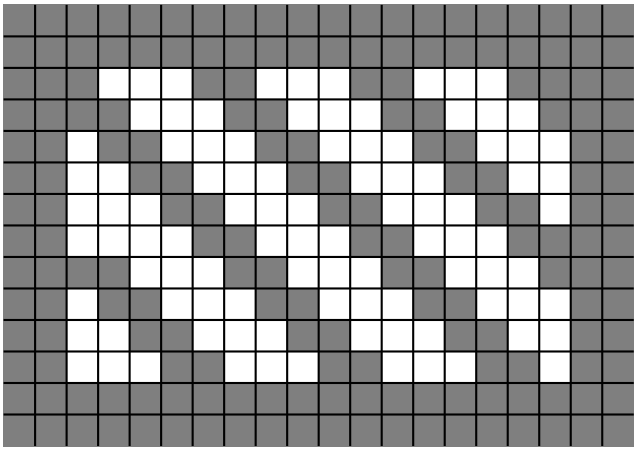

Figure 2.7: Example of a stencil buffer. Only white areas can be painted to

making them faster than purely software-based regions. However, stencil buffers require large (albeit constant) amounts of memory for high output resolutions and take more time to initialize than regions when used for simple areas. Stencil buffers are for example available in SVG descriptions (see Section 2.3.6).

2.2 Graphical Objects

Most graphics interfaces can render various kinds of primitive objects. In this section, we describe common kinds of objects and how they are typically represented. We start with vector graphics primitives in Section 2.2.1; a discussion of their graphical attributes follows in Section 2.2.2. Paths, which combine vector graphics primitives to sequences, are the subject of Section 2.2.3. Finally, we examine raster images in Section 2.2.4 and text objects in Section 2.2.5.

2.2.1 Vector Graphics Primitives

An important class of graphical objects consists of lines and curves exclusively. While such objects may share a common style of appearance, their geometrical shape is highly individual and therefore is their distinctive property. Because of their linear structure, the drawings they help create are often called *vector graphics*.

Most of these objects have their origin in classical geometry and can be viewed as infinite sets of infinitely small points. They do not cover any area, but may enclose one. One way of painting them on a raster device is thus to light all raster cells within that area. The corresponding paint mode is called *filling*. A second paint mode, which traces the lines and curves of an object with an imaginary pen and lights all raster cells that it thus visits, is called *stroking*. Both of these modes are further discussed in Section 2.2.2.

Graphics interfaces that support user-defined clip areas as discussed in Section 2.1.4 may offer another rendering mode, called *clipping*, which intersects the interior of an object with the current clip area instead of painting it. The result of this intersection then becomes the new clip area. Clipping is similar to filling because it also interprets objects as contours that enclose an area.

Depending on the exact specification of the operations in a graphics interface, clients specify the desired rendering mode by calling separate procedures (which could for example be named **StrokeRect** and **FillRect**) or by passing an extra parameter that defines the desired mode to each rendering procedure.

The following paragraphs list several important kinds of vector graphics objects.

Points and Lines. The most basic object is the *point*, defined by its Cartesian coordinates. Points do not play an important role in the final output but are essential for anchoring objects at specific locations. For example, two points define a straight *line*. Lines are essential graphical objects, too, as most shapes can be expressed or at least approximated with lines.

Circles, Ellipses, and Arcs. A point and a radius define a *circle*; a point and a pair of orthogonal radii define an (axis-aligned) *ellipse*. With an additional pair of values which indicate start and end angles, circles and ellipses can

be restricted to circular and elliptical *arcs*.

A convenient way of defining a general ellipse is to specify its bounding parallelogram with three points, one at the center of the ellipse and two others at the end of conjugate diameter pairs [25]. Figure 2.8 displays an ellipse and its conjugate diameter pairs.

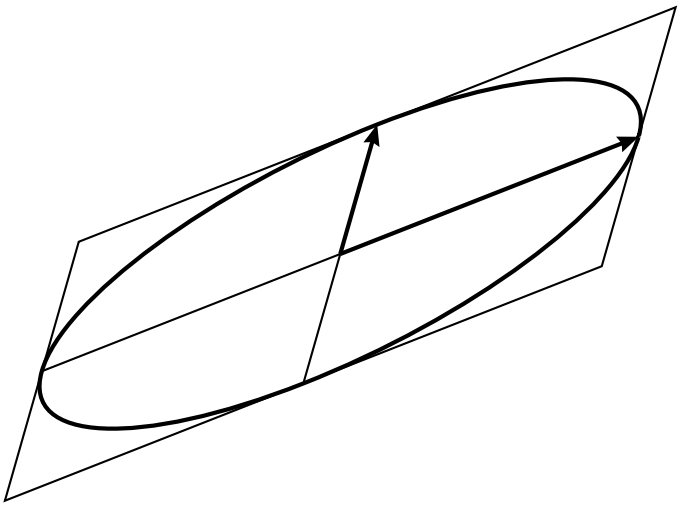

Figure 2.8: A general ellipse and its conjugate diameter pair, displayed as arrows

Rectangles. Two opposing corner points define an axis-aligned *rectangle*. Although rotated and sheared rectangles cannot be specified in this manner, the definition has the advantage that the resulting shapes are guaranteed to be rectangular, which might be difficult to enforce with alternative definitions, as for instance by using four lines or four points.

Polylines and Polygons. If successive points in a list of control points are joined by straight lines, the resulting connected sequence of lines is called a *polyline*. If an additional line that connects the last and the first point is added, the result is a *polygon*.

Curves. In addition to lines, numerous types of polynomial or piecewise polynomial curves can be defined on sequences of control points. So far, only quadratic and cubic Bézier curves [9, 10] have been regularly used in graphics interfaces, for example in Postscript (see Section 2.3.4) or in SVG

(see Section 2.3.6), since they are particularly easy to implement. Detailed discussion of Bézier curves and other polynomial curves is beyond the scope of this thesis; introductions can be found in [27, 65, 35]. A cubic Bézier curve is a parametric curve, defined as

$$C(t) = \sum_{i=0}^{3} B_i^3(t) \cdot \mathbf{P}_i$$

where $\mathbf{P}_i$ is the i-th control point and B_i^3 are Bernstein polynomials, defined as

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}.$$

A cubic Bézier curve can thus also be written as

$$C(t) = (1-t)^3\mathbf{P}_0 + 3t(1-t)^2\mathbf{P}_1 + 3t^2(1-t)\mathbf{P}_2 + t^3\mathbf{P}_3.$$

Cubic Bézier curves pass through their end points $\mathbf{P}_0$ and $\mathbf{P}_3$, but usually do not pass through their other control points $\mathbf{P}_1$ and $\mathbf{P}_2$, as can be seen in Figure 2.9. The curve is always completely within the convex hull of its

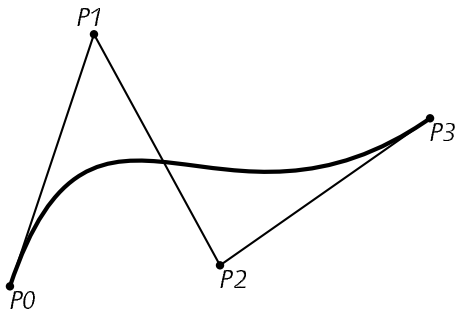

Figure 2.9: A cubic Bézier curve and its control polygon

control polygon. Cubic Bézier curves are so common in 2D graphics because they are easy to draw using recursive subdivision (deCasteljau algorithm) or forward differencing [35].

2.2.2 Object Attributes

Graphical objects based on lines and curves may have a clearly defined geometry, but additional parameters, their *graphical attributes*, are needed

to paint them. We can distinguish two kinds of attributes, geometric and non-geometric.

Geometric Attributes. These tell the graphics interface to slightly adjust the geometry of individual objects, for example by giving lines a non-zero width. Geometrical attributes are not essential and can in principle be emulated outside the graphics interface. Still, they are useful because they offer clients a more abstract view of the rendering process.

Non-geometric Attributes. These include color, pattern, and alpha values. They determine how objects are represented on an output device.

Object Interior

Several graphical objects, for example circles and rectangles, consist of a closed contour that divides the object's interior from its exterior. This is the case for circles, ellipses, polygons, and many others. The following paragraphs describe aspects that are important when filling or clipping the interior areas of closed graphical objects.

Fill Functions. To *fill* the interior of an object, color must be painted on the corresponding area of the output device. In theory, the color value used for each point in an enclosed area is the result of an arbitrary function of the point's coordinates. In practice, to keep specification and implementation simple and efficient, most graphics interfaces only provide a restricted selection of fill functions. Some of the more popular ones are illustrated in Figure 2.10. The *solid color fill* on the left fills the whole area with a con-

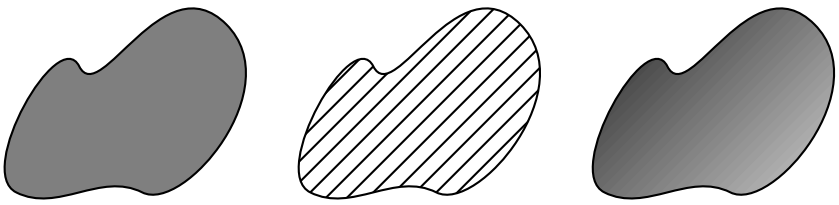

Figure 2.10: Examples of fill functions: solid fill, pattern fill, and linear gradient fill

stant color value. The *pattern fill* in the middle repeatedly paints a pattern (usually a raster image) within the object's boundaries, all instances of the pattern being spaced at regular intervals. With the *linear gradient fill* on the right, clients assign color values to a few specific points and let the graphics engine interpolate color values for all other points inside the object. Apart from linear gradients, radial gradients are also quite common. All three kinds of fill functions are for example supported by SVG (see Section 2.3.6).

Self-intersecting Contours. The interior of objects whose contour intersects itself, such as the star-shaped polygon in Figure 2.11, is not clearly defined. Most graphics interfaces offer two different interpretations of the situation,

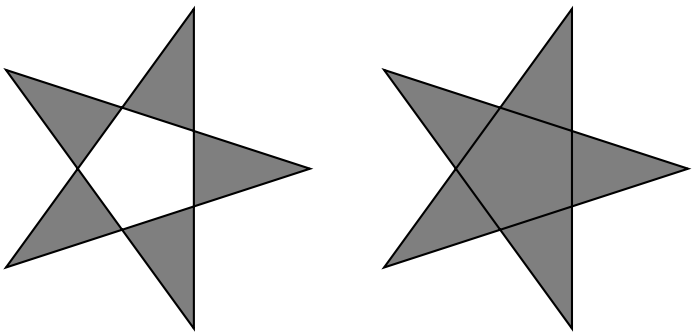

Figure 2.11: A self-intersecting polygon, interpreted as by the even-odd rule on the left and by the non-zero-winding rule on the right

the *even-odd rule* and the *non-zero-winding rule*. In both cases, to find out if a point is inside an object or not, imagine a ray that originates from the point in question and count how often the ray intersects the object's contour. Starting with a sum of zero, add one to the sum each time the contour crosses the ray from left to right and subtract one each time it crosses the ray from right to left. With the even-odd rule, the point is inside the object if the resulting sum is odd; with the non-zero-winding rule, it is inside if the sum is not zero. Graphics interfaces such as Postscript (see Section 2.3.4) allow their clients to choose either rule for each object.

Object Outline

All vector graphics objects can be stroked, meaning that the graphics engine lights the pixels that their contour touches. As with filled objects, we examine some aspects of the stroking process in more detail in the following paragraphs.

Hairlines. Stroking an object is a priori a paradoxical operation; for lines and curves are one-dimensional and continuous, yet they are to be converted to a set of two-dimensional discrete raster cells. The simplest approach is to use the *hairline* model, which approximates a line or curve by painting only a minimal set of adjacent pixels in its proximity. The disadvantage of the hairline model is that the perceived curve thickness depends on the pixel size of the output device and slightly varies with the direction of a curve. (Diagonal lines appear either thinner or thicker than horizontal and vertical lines, depending on how many neighbors each pixel has.)

Thick Lines. Several hairlines must be drawn one pixel apart from each other to make lines appear thicker. To overcome this weakness of the hairline model, most graphics interfaces associate a *line width* or *thickness* attribute with all stroked objects. To draw thick lines, each line is converted to a rectangle of corresponding length whose interior can then be filled [27]. Curves are treated analogously. If the line width is below some threshold, typically related to pixel size, the interface may still fall back to using hairlines instead of thick lines.

Some interfaces, for example QuickDraw (see Section 2.3.2), provide an alternative method for stroking thick lines. Instead of geometrically converting lines and curves to areas and filling these, they lead a virtual *brush* or *pen* along the contour to stroke. The brush has a fixed shape, which is often rectangular or circular. Brushes can be implemented efficiently [72], but do not offer the same flexibility with regard to line caps and line joins (see below) as geometrically constructed thick lines.

Line Caps and Joins. The introduction of thick lines brings up further issues. One of them is how line ends should be rendered. Depending on the desired effect, a round or square cap can be appended to an otherwise bare-ended line, as shown in Figure 2.12. Although other types of line

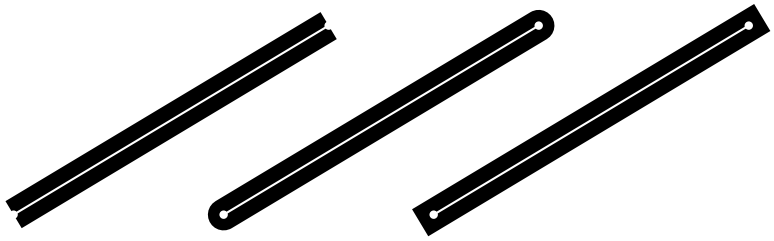

Figure 2.12: From left to right: butt caps, round caps, and square caps

caps, especially arrow heads, often appear in 2D drawings, most interfaces leave it to their clients to implement them. It seems that the additional expressiveness does not justify the additional complexity in this case.

Another question concerns the shape of the join between two lines that meet at a non-zero angle, for example at the corners of a polygon. A natural solution is to extend the outer contours of the adjacent lines until they intersect, leading to so-called *miter joins*. However, as the angle between the lines becomes more pointed, this intersection moves farther away from the original corner. In practice, graphics interfaces impose a limit on the extent of miter joins; if this limit is exceeded, a miter join is replaced by another type of join. The most common kinds of line joins, as for example supported by Postscript (see Section 2.3.4), are displayed in Figure 2.13.

Figure 2.13: From left to right: miter joins, round joins, and bevel joins

If a brush model is used, the shape of the brush automatically determines the shape of line caps and line joins. A consistent look can only be achieved

with round brushes, resulting in round caps and joins. For other types of brushes, the shape of caps and joins depends on individual line orientation and brush shape.

Dash Pattern. Some output devices can only paint objects in black or white. Applications may therefore need attributes other than color for making lines and curves discernible from each other. A common method is to draw dashed and dotted lines instead of continuous ones (see Figure 2.14). Thus, many graphics interfaces allow clients to associate a dash pattern with lines and curves. In the simplest case, the number of available patterns is limited to a fixed set of symbolic constants (for example in GKS, see Section 2.3.1), whereas advanced models accept arbitrary sequences of distances for visible and invisible parts (for example Postscript, see Section 2.3.4).

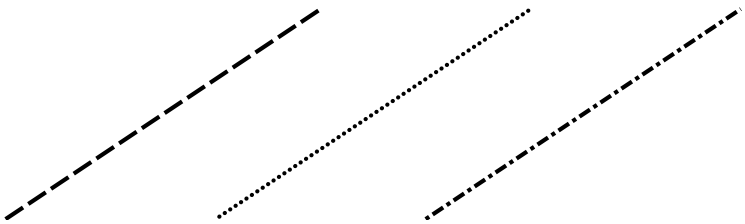

Figure 2.14: Dashed and dotted lines

2.2.3 Paths

A general path model connects primitive objects such as lines and curves to create inhomogeneous sequences. Because the contour of many shapes can indeed be described using only sequences of lines and curves, paths serve well as general representations of graphical objects. In particular, paths which support lines, elliptical arcs, and cubic polynomial curves are sufficient for modeling a wide variety of shapes reasonably well. Even the contours of letters do not require curves of higher degree than cubic curves; in fact, the popular TrueType font format [7] uses only quadratic Bézier curves. As a consequence, it is perfectly acceptable for a graphics interface not to provide any other vector graphics primitives if it supports a general path model; an example of this is Postscript (see Section 2.3.4). For an interface that models graphical objects explicitly such as Java 2D

(see Section 2.3.7), paths are a natural foundation for a general rendering protocol.

Although paths in principle have the same graphical attributes as other vector graphics objects, they have some unique aspects as well.

Subpaths. We assume in the following that each *path* consists of at least one *subpath*; subpaths in turn consist of a non-empty sequence of connected lines and curves. This definition allows paths to contain more than one subpath, which is essential for modeling shapes that consist of multiple disjoint parts or shapes that contain holes. Figure 2.15 displays two examples of paths with multiple subpaths. As with self-intersecting polygons, the interior of a path can be defined according to the even-odd rule or the non-zero-winding rule.

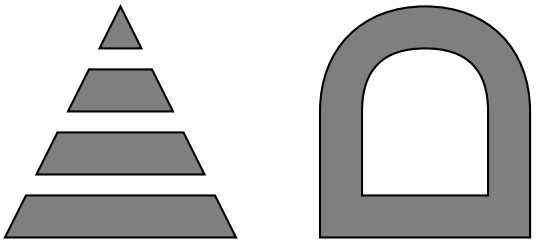

Figure 2.15: Non-trivial path objects

Path Specification. Assuming that the graphics interface uses an immediate rendering model, a general path model has the drawback that the specification of a path is more complicated than that of a primitive curve with a fixed number of parameters such as a line or a circle. Path specification typically takes place over several procedure invocations. One procedure starts a new subpath, others append lines and curves, yet another may close the current subpath, and one ends the path. Thus, the graphics interface is put into special "within path" and "within subpath" modes while the specification of paths and subpaths is in progress. If a client fails to terminate a path or subpath and thus leaves the graphics interface in path mode, the next drawing operation might result in inconsistent or erroneous output.

When the specification of a path is thus spread over several procedures, one of these procedures must allow its caller to decide whether the resulting

path should be filled, stroked, or otherwise processed. Most graphics interfaces delay this decision to the point when a final procedure call completes the specification of the current path. Since rendering can only start when the entire path is known, we call this approach *late rendering*. On the other hand, if the rendering mode is already known when the specification of a path begins, we speak of *early rendering*. Both approaches have their advantages and disadvantages, which we look at in the following paragraphs.

Late Rendering has the advantage that the geometry of the path is known to the graphics interface at the time it starts painting the path. This is of special importance for *closed* subpaths, which are subpaths where the last curve leads back to the starting point of the first curve. Because the tangent vectors of the curves which meet at that point are known, the line join between the last and the first curve can be properly rendered. Figure 2.16 shows the difference between a properly closed path and a path without proper line join.

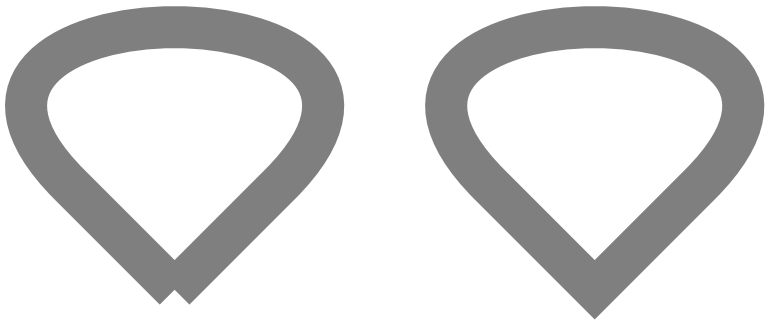

Figure 2.16: Difference between open and closed subpaths

The main drawback of late rendering is that the complete path must be stored in an internal data structure until its construction is complete. Such an internal data structure is similar to a display list. Unlike display lists, however, it only exists until the path has been rendered and cannot be reused at a later time. Although the immediate rendering mode is internally replaced by a retained mode, this fact remains hidden from clients. When

path specification ends, the path structure must be traversed again to render it. Information that was known at the time the path was specified, e. g. the kind of curve to draw and how many control points to consume, must be reconstructed from the data structure. Using late rendering thus inherently involves a small overhead in both space and time. Nevertheless, the late rendering approach is used in all known graphics interfaces that use an immediate rendering model, for example Postscript (see Section 2.3.4) or the Microsoft Windows GDI [56].

Early Rendering permits a graphics interface to paint path segments already during path specification. In the common case where clients stroke hairlines and hairline curves, the graphics interface can immediately paint primitives at the time the corresponding procedures are called. Yet the system is still free to store the complete path in a data structure and to defer rendering to a later time if a client requests it. Early rendering can thus emulate late rendering.

The correct treatment of closed subpaths is more difficult with early rendering than with late rendering because it is not clear whether to paint a line cap or a line join when a client begins a subpath, even if it is understood that the path should be stroked. If clients are allowed to close a subpath whose specification is in progress, output of the area around the first point of the subpath has to be delayed until enough information is available to decide between a line cap and a line join at that point. This slightly complicates the path renderer, which must remember coordinates and tangent vector of the subpath's starting point.

Another solution to the same problem is not to allow clients to close an initially open subpath. An eventual **Close** operation is replaced by two new operations, **Enter** for starting a subpath and **Exit** for ending it. Both operations expect a vector argument; **Enter** expects the final direction of the last subpath element (called d_1 in Figure 2.17) and **Exit** the initial direction of the first subpath element (called d_0 in Figure 2.17). With the help of these direction vectors, proper line joins can be rendered when starting or ending a subpath. Zero length vectors can be used to indicate that a subpath is not closed and that a line cap should be rendered instead of a join.

The enter/exit model is an interesting generalization since, in addition to being useful for specifying closed subpaths, it can also be used to specify partial subpaths. With **Enter** and **Exit**, any part of a subpath can be rendered

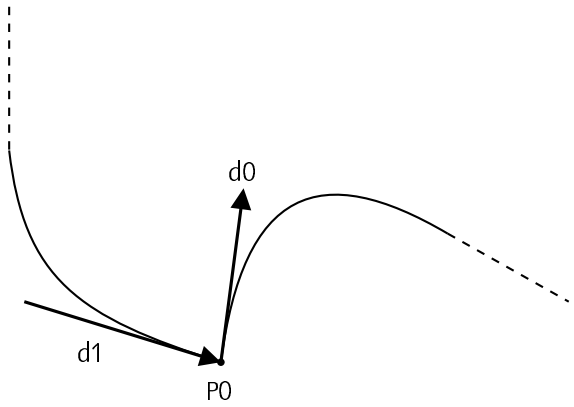

Figure 2.17: Relevant direction vectors for **Enter** and **Exit**

with correct line joins at its ends. On the other hand, a client's duty to provide direction vectors for the first and last curve of a subpath may require it to perform additional calculations. Surprisingly enough, direction vectors can easily be deduced for most graphical objects, especially if their geometry is already captured in a client data structure, as for example in the Leonardo shape framework that is presented in Chapter 4.

As far as we are aware, the Gfx library that we present in Chapter 3 is currently the only graphics interface that combines immediate mode with a path model that is based on early rendering.

2.2.4 Raster Images

A *raster image* is a rectangular array of pixels, each of which contains a color or alpha value. Raster images can not only be used as logical output devices but also as graphical objects. Painting a raster image maps each pixel to the corresponding area on the output device and paints the area with the pixel's color. If the source image is used as a mask and only contains alpha information, clients must provide a color value with which non-transparent pixels are painted.

There are several methods for creating raster images. As already mentioned, any program can use an image as a logical output device and render graphical objects on it. Moreover, special paint programs let users create

images interactively. Last but not least, images can be created using scanners and digital cameras. These are devices that sample a signal (e. g. a photographic image) at regular intervals and assign the sampled color value to the corresponding pixel.

Raster images are superior to vector graphics for describing scenes with lots of small details and many different shades of color, which makes them the preferred model for storing scanned photographs or pictures. Furthermore, the time that is needed to render them on a raster display is constant and does not grow with scene complexity. On the other hand, raster images suffer from their fixed resolution. Rendering a raster image on a raster device with a different pixel size requires that the image is scaled first. However, scaling or otherwise transforming an image is a non-trivial operation (see Section 3.4.5). Simple algorithms are prone to aliasing and other artifacts whereas sophisticated ones require significant amounts of computation [88].

Raster images can be considered a counterpart to vector graphics. Which to use is highly dependent on the application, and both may be necessary at the same time. Any graphics interface has to support both to be complete. However, fine-grained manipulations that involve modifications to individual pixels are often not part of the primary graphics interface and are left to other instances. An example of such a dedicated image processing library is ImageMagick [43]. It not only allows reading and writing individual pixels, but also supports loading and storing images in various file formats, image filtering, image transformations, and color reduction.

2.2.5 Text

A third essential kind of graphical objects manages text. In European and American culture, text consists of paragraphs, paragraphs consist of lines, lines consist of words, and words consist of characters. To make matters more complicated, the graphical representation of a character is called a *glyph*. The distinction between characters and glyphs is necessary because some languages use different glyphs for the same character (depending on the position of the character within a word) or combine several characters into one glyph, which is then called a ligature. For example, typographers often replace an "f" followed by an "i" with a dedicated "fi" glyph.

Of special interest to a graphics interface is the observation that only a

small set of glyphs is needed to display any text. Glyphs that are derived from a consistent design and thus share a common look are collected in a *font*. Once the font and the desired font size, which defines the height of a line of text, are known, the graphics system can map any string of characters into an appropriate series of glyphs.

Glyphs can be available in the form of raster image masks, outline curves, or both. For raster displays with low resolution and small font sizes, image masks are hard to beat in terms of rendering speed, memory requirements, and output quality. However, as output resolution and font size increase, the memory requirements of glyph images can no longer be neglected. It may become more efficient to derive glyph images from a geometrical description of their outline when needed. Besides, type designers typically work with outlines when they create a font, making glyph outlines readily available. However, the conversion of outlines to glyph images is a non-trivial task which either requires sophisticated *hinting* mechanisms (to align points in the glyph outline with the grid lines of the raster image before scan-conversion, [2, 7, 72]) or manual post-processing to deliver acceptable results, at least for low resolutions and small font sizes.

Graphics interfaces that support a general path model often allow their clients to use glyph outlines exactly like other paths, implicating that glyph outlines can be stroked, filled, and used for clipping. With a path model that uses early rendering, the rendering mode is already known at the time when the subpaths of a glyph outline are appended to the current path. Hence, if the path would later be filled anyway, the path renderer may choose to immediately paint corresponding glyph bitmaps instead of appending their outlines to the current path. The graphics interface can thus benefit from the superior quality and rendering speed of glyph bitmaps. This is not possible with late rendering because the rendering mode will only be known when the specification of a path is complete. To nevertheless be able to use glyph bitmaps, graphics interfaces such as Postscript (see Section 2.3.4) provide two different operations for rendering text: one for drawing filled glyphs and another for appending glyph outlines to the current path.

In addition to glyph descriptions, fonts also contain metric information, which is needed to position individual glyphs in the coordinate system of the output page. When typesetting text from left to right, an imaginary horizontal line, called baseline, serves as a reference for aligning glyphs vertically. Each glyph is positioned relative to a current point on the baseline. Before moving

on to the next glyph, the current point is moved along the baseline by a distance which is called the glyph's *advance width* (see Figure 2.18). The spaces to the left and right of a glyph are called its sidebearings. Advanced

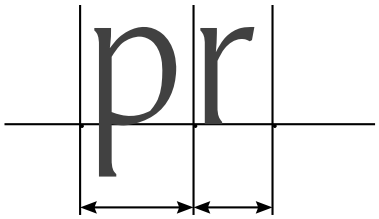

Figure 2.18: Glyphs and advance widths in relation to reference points on the baseline

font models adapt the advance width for specific combinations of adjacent glyphs, for instance by moving "A" and "v" closer together than the advance width of "A" would suggest. This process, illustrated in Figure 2.19, is called *kerning*.

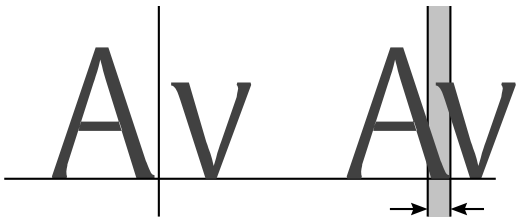

Figure 2.19: Left: unkerned; right: kerned

2.3 Selected Examples

In this section, we examine several graphics interfaces in more detail. The selection is not intended to list all existing graphics systems but to give a representative overview of related work in the field of 2D graphics programming. Most examples have been picked either because of historical importance or because of their unique approach. The examples we have chosen to investigate are GKS (see Section 2.3.1), QuickDraw (see Section 2.3.2), Oberon (see Section 2.3.3), Postscript (see Section 2.3.4), MetaPost (see Section 2.3.5), Scalar Vector Graphics (see Section 2.3.6), and Java 2D (see Section 2.3.7).

Criteria. While we look at the chosen examples of existing graphics interfaces, we are especially interested in the following criteria.

- What output devices are supported? Can clients add new output devices?
- Which rendering models are supported? What kind of internal state is maintained?
- Are user coordinate systems provided? Or at least normalized coordinates? Is there support for affine transformations?
- What can be said about the imaging model?
- Are user-defined clip areas available? How are they specified?
- What kinds of graphical objects are supported? Is there a general path model? Can glyph outlines be used as paths?
- What kinds of attributes can be used with vector graphics and text objects?

A comparison of all chosen examples according to these criteria follows in Section 2.3.8. From this comparison, we draw our conclusions and motivate our research in Section 2.3.9.

Categories. Our examples can be roughly divided into three distinct categories: application programming interfaces (API), graphics languages, and graphical object frameworks.

API are usually provided as part of or in close connection with an operating system. Their goal is to assist application programs with creating graphical output.

Graphics languages are often used where portability between platforms is needed. They decouple the creator of a graphical scene, be it a human or a machine, from the renderer that turns the program into graphical output. Although graphics languages are hardly ever used in interactive environments, they often follow the same basic principles as API.

Similarly, object frameworks also put an intermediate structure between scene creation and rendering, but in the form of an object graph instead of a textual description. They lie in the middle between straightforward API and graphics languages, both in terms of rendering speed and abstraction level.

Sample Figure. To give a small impression of how these graphics interfaces are used in practice, we try to create the sample figure that is displayed in Figure 2.20 with each of them. The sample figure comprises a solid gray circle, a triangle which is stroked with thick dashed lines, a solid white rectangle with a black frame, and a caption which is centered over that rectangle. Some systems may not provide enough functionality to draw the figure, in which case suitable approximations are used. The resulting programs are shown in Figures 2.21 up to 2.27. Note that setup code is not displayed in these examples, only those parts that actually render graphics.

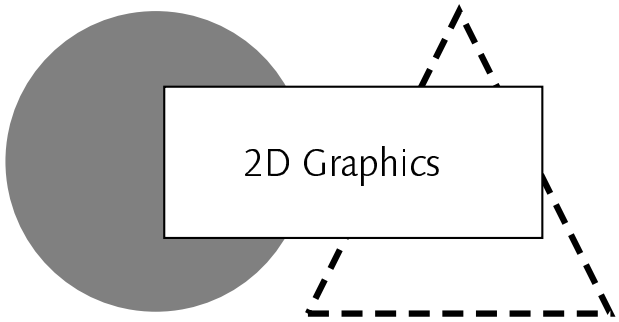

Figure 2.20: Sample figure

2.3.1 Graphical Kernel System

While it was not the first graphics API ever, the *Graphical Kernel System* (GKS), which is based on earlier work on the *3D Core Graphics System* (the Core) in the second half of the seventies [33, 34], was the first to become an international standard [44, 5]. It addressed the problems of earlier graphics packages by providing device-independence and multi-platform portability. In particular, the GKS specification lists the following points as major differences to earlier graphics interfaces:

- GKS has no notion of a current point; at the time, its authors regarded current points as a relict from the time when most output devices were pen plotters. Ironically enough, more recent interfaces again include a current point for managing paths and text.
- It defines a set of logical output devices called *workstations*, several of which can be active at any time. Earlier systems were usually bound to

a single device.

- Input devices are modeled using six different classes of *logical devices*, which are mapped to real devices such as keyboards and light pens.
- Groups of output commands can be stored in so-called *segments* for later use.
- Applications can *inquire* state and capabilities of output devices.

Most GKS implementations are available as procedural libraries with language bindings for at least Fortran and C.

```
C coordinate declarations
      REAL TX(4), TY(4), RX(5), RY(5)
      DATA TX /200.0, 400.0, 500.0, 200.0/
      DATA TY /0.0, 0.0, 100.0, 0.0/
      DATA RX /100.0, 350.0, 350.0, 100.0, 100.0/
      DATA RY /50.0, 50.0, 150.0, 150.0, 50.0/
C
C circle (framed instead of filled)
      GUCIR2(100.0, 100.0, 100.0)   ! center and radius
C
C triangle
      GSLN(GLDASH)   ! set line style to dashed
      GSLWSF(3.0)   ! set line width scale factor
      GPL(4, TX, TY)   ! triangle polygon
C
C rectangle
      GSLN(GLSOLI)   ! line style back to solid
      GSLWSF(1.0)
      GSFAIS(GSOLID)   ! fill area style
      GSFACI(0)   ! use background color for fill area
      GFA(5, RX, RY)   ! filled rectangle
      GPL(5, RX, RY)   ! stroked rectangle
C
C caption (not accurately centered)
      GSTXFP(1, GLSTRKP)   ! choose font 1, stroke precision
      GSCHH(20.0)   ! character height
      GTX(200.0, 90.0, "2D Graphics")
```

Figure 2.21: Sample figure in GKS

Workstations. The first operations of a program are to open and activate a workstation. GKS implementations may offer many different kinds of workstations, corresponding to existing or new windows in a frame buffer or to various virtual output devices that write to files. However, users cannot add custom output devices to the available pool of workstation types.

Graphical Objects. The basic drawing primitives in GKS are *polylines*, *polymarkers* (special marker symbols painted at point locations), text, *fill areas* (filled polygons), *cell arrays* (raster images), and *generalized drawing primitives* (GDP). The latter are "a standard way to be non-standard" and allow implementations of the standard to provide additional primitives, such as circles and ellipses. GDP can (or could) supposedly be registered with the International Standards Organization (ISO) for further revisions of the standard.

Attributes and Bundles. The visual appearance of each drawing primitive is influenced by a matching set of attributes. These attributes are stored in the graphics state of each workstation. Polylines, for instance, have a line style, a line width, and a line color associated with them. Fill areas have a fill color, a fill style (hollow or solid), and a fill pattern. Text output is influenced by font, character spacing, character expansion, text alignment, and character-up vector. The latter allows vector-based fonts to be typeset in any direction. An interesting point is that attributes can not only be set individually, but also by selecting an *attribute bundle*. Attribute bundles allow applications to use abstract styles instead of concrete attribute combinations. Applications must select for each attribute whether it can be set individually or only in bundles.

Windows and Viewports. Device independence is a central part of GKS. GKS relies on two coordinate mappings to provide device independence, one (the *normalization transformation)* from world coordinates to normalized device coordinates in the range from 0 to 1, and another (the *workstation transformation*) from normalized device coordinates to device coordinates. Both transformations map a rectangle in the source coordinate system (the *window*) to a rectangle in the target coordinate system (the *viewport*). Applications can store a number of different normalization transformations for every logical output device in one of a predefined set of transformation slots.

Depending on which transformation is active, partial images can be drawn at different locations and in different sizes. Both the normalization transformation and the workstation transformation can also be used to restrict the area where output is produced. Thus, GKS allows clients to customize the current clip area to an arbitrary rectangle.

Segments. To simplify repeated output of the same graphical object, any series of output primitives can be stored in a *segment* and rendered directly from the segment store later. Segments are thus equivalent to what we called display lists in Section 2.1.1. Depending on the kind of output device, GKS may actually store the segments in the local memory of a graphics terminal instead of the memory where the host application runs. In addition, the normalization transformation is applied to all primitives before storing them in a segment. Because of these two aspects, segments can usually not only be rendered with less programming effort but also faster than by drawing single output primitives repeatedly.

Conclusion. Because at the time when GKS was introduced it was the first of its kind and was clearly superior to everything else in the workstation market, it found widespread use and inspired many further developments. Although it has in the meantime been replaced by more powerful interfaces, GKS was the first to include many of the features that are still relevant for more recent graphics interfaces.

2.3.2 QuickDraw

QuickDraw [6] is the native graphics interface of the Apple Macintosh computer. Although it was developed in the first half of the 1980s, it is still in use today, almost twenty years later, with relatively few extensions added since then.

Ports. Rendering in QuickDraw is based on multiple independent drawing environments, called *ports*. Ports are data structures that manage a current graphics state and produce output on an underlying bitmap, either the screen or an off-screen buffer. Many ports can share a bitmap, each managing a separate state and coordinate system. Several ports can be open at the same time, but only one is active.

```
PROCEDURE Paint;
VAR r: Rect; w: INTEGER;
BEGIN
  (* circle *)
  SetRect(r, 0, 200, 200, 0);
  FillOval(r, gray);

  (* triangle *)
  PenSize(3, 3);
  PenPat(dkgray);   (* pen pattern for lack of dash pattern *)
  ShowPen;
  MoveTo(200, 0); LineTo(400, 0); LineTo(300, 200); LineTo(200, 0);
  HidePen;

  (* rectangle *)
  SetRect(r, 100, 150, 350, 50);
  FillRect(r, white);
  PenSize(1, 1); PenPat(black);
  PaintRect(r);

  (* caption *)
  TextSize(20);
  w := StringWidth("2D Graphics");
  MoveTo(100 + (250-w) DIV 2, 90);
  DrawString("2D Graphics");
END Paint;
```

Figure 2.22: Sample figure in QuickDraw

Coordinate Grid. In QuickDraw, integer coordinates always denote points where horizontal and vertical lines of a regular grid intersect. Between adjacent grid lines, pixels are embedded. To map a grid point into device coordinates, its coordinates are added to those of the port's origin. Neither sub-pixel precision nor user-defined coordinates are supported.

Current Pen. QuickDraw supports a current pen (in the sense of a brush; see Section 2.2.2), which is a rectangle of user-defined dimensions with a user-defined fill pattern. The reference point of a pen is at its upper left corner; when a line between two grid points is drawn, the pen affects the pixels below and to the right of the line. Width and height of the pen, together with the direction of the line, define the resulting line width, which may thus vary considerably, especially for non-square pens. This approach allows painting thick lines fast; however, it does not lend itself well to drawing line caps and line joins in a consistent manner. QuickDraw uses pens to render lines, rectangles (also with rounded corners), ovals (axis-aligned ellipses), arcs, wedges, and polygons.

Text. QuickDraw uses character bitmaps for rendering text. It lets its clients select a font family, a style (including outlined and shadowed variants), and a point size. If a bitmap font is not available in the requested size, the Macintosh font manager creates it dynamically by scaling bitmaps from an existing size of the same font.

Pictures, Polygons, and Regions. Render directives in QuickDraw are usually executed immediately. However, a port can also be put in a special recording mode. It then either records a *picture*, which corresponds to a display list in our terms but can be scaled when it is rendered, a *polygon*, which is a connected sequence of lines, or a *region*, which is an arbitrary area constructed from boolean combinations of primitive shapes and other regions. One region always defines the current clip area. Besides, a region can be filled with a pattern, or the current pen can be led along the interior side of its contour. QuickDraw's powerful and efficient region implementation was one of its major achievements at the time of its introduction.

Imaging. In the original implementation of QuickDraw, only monochromatic bitmaps were supported. Hence, QuickDraw's imaging model merely

consisted of a transfer mode which specified the boolean operation that was used to combine source and destination pixels. Nevertheless, QuickDraw could paint bitmap masks with binary *or* and supported rubber-banding with binary *xor*. More recent versions of QuickDraw also deal with colored images and additional transfer modes.

Customization. Although designed before object-oriented programming became popular, QuickDraw allows customization of rendering procedures by exchanging procedure variables in the method table that is associated with each port. QuickDraw itself uses this mechanism for printing graphics.

Conclusion. Like GKS, QuickDraw is one of the pioneering 2D graphics interfaces. Many of the concepts it introduced, such as regions or pens, resurface in later graphics interfaces. More recent versions of QuickDraw provide additional features, notably support for multiple monitors, color, and pixmaps (raster images that contain colored pixels instead of black and white only). An alternative object-based approach (QuickDraw GX [8]) was not received well by programmers and is discontinued. Other well-known API such as the GDI of Microsoft Windows [56] or the X-Windows library for UNIX workstations [61] are comparable to QuickDraw in terms of software architecture, but offer additional and refined functionality.

2.3.3 Oberon System 3

The main design goal for the Oberon System [85, 86] was to implement a complete operating system with minimal effort and thus to keep everything as simple as possible. Its graphical features were indeed limited: it only rendered pixels, rectangles, and bitmap patterns (especially glyph bitmaps). Besides, it did not feature a unified interface for rendering on displays and printers. Printing required that a separate module with similar, but not equal functionality was imported. Rendering into raster images was not supported at all.

Oberon System 3 [36, 51] is derived from the original Oberon System, but extends it with a library concept for building persistent object hierarchies. It includes a component-based graphical user interface, called Gadgets, which adds additional primitives for drawing lines, circles, ellipses, and polygons with attributes such as line width and fill pattern. Oberon System 3 still

```
IMPORT Display, Display3, Fonts;   (* required modules *)

PROCEDURE Paint (mask: Display3.Mask; x, y: INTEGER);
VAR
  col: Display.Color;
  px, py: ARRAY 3 OF INTEGER;
  font: Fonts.Font;
BEGIN
  (* circle *)
  col := Display.RGB(128, 128, 128);
  Display3.Circle(mask, col,
    Display.solid,   (* pattern *)
    x+100, y+100, 100, 0,   (* center, radius, and width *)
    {Display3.filled},   (* paint mode *)
    Display.replace);   (* imaging model *)

  (* triangle *)
  px[0] := x+200; py[0] := y;
  px[1] := x+400; py[1] := y;
  px[2] := x+300; py[2] := y+200;
  Display3.Poly(mask, Display3.black,
    Display.grey0,   (* use bitmap pattern for lack of dash pattern *)
    px, py, 3, 3,    (* pinpoint coordinates, 3 corners, width 3 *)
    {}, Display.replace);

  (* rectangle *)
  Display3.ReplConst(mask, Display3.white,
    x+100, y+50, 250, 100,   (* lower left corner, width, and height *)
    Display.replace);
  Display3.Rect(mask, Display3.black, Display.solid,
    x+100, y+50, 250, 100, 1, Display.replace);

  (* caption *)
  font := Fonts.This("Syntax20.Scn.Fnt");
  Display3.CenterString(mask, Display3.black,
    x+100, y+50, 250, 100,
    font, "2D Graphics",
    Display.paint);
END Paint;
```

Figure 2.23: Sample figure in Oberon System 3

requires a separate interface for printing, but at least provides equivalent procedures in the printer and the screen interfaces.

Coordinates are always specified in the coordinate system of the corresponding output devices. Thus, clients are themselves responsible for mapping display coordinates to printer coordinates. System 3 supports overlapping frames on the screen and stores their visibility in *mask* structures (which correspond to the regions from Section 2.1.4). Output is clipped to the interior of these masks.

Unlike other graphics interfaces, Oberon's display system is stateless and does not need a graphics context structure or a graphics state; all graphical attributes are passed as parameters to individual drawing procedures. In spite of this simplification, Oberon offers a functionality which, apart from the lack of display lists, is not far from that of the original QuickDraw. However, such a stateless interface is only practical as long as the number of parameters per drawing operation stays small. Otherwise, the overhead that results from having to push parameter values onto the stack and checking whether parameter values are valid in the callee may negatively affect overall performance.

2.3.4 Postscript and Display Postscript

Postscript is a page description language developed by Adobe Systems [3]. In a typical setting, applications create a Postscript source file, which they can later send to a printer with built-in Postscript support. Because of its high level of abstraction and its device-independence, Postscript has become the de facto standard for creating professional quality print. Display Postscript (DPS) is an extension of the standard Postscript language that is specifically designed to run within a window system. DPS therefore adds a few additional operators to the language which allow it to manage multiple graphical contexts in a multi-processing environment.

Input. Postscript is a general programming language. It reads characters from an input stream and interprets them. Its syntax and its stack-based evaluation of expressions are simple, but its support for user-defined procedures and its use of arrays and dictionaries for building complex data structures make it very powerful. In addition to character streams, DPS systems also accept input in the form of binarily encoded operations. This

```
% circle
100 100 100 0 360 arc
0.5 setgray fill

% triangle
200 0 moveto 400 0 lineto 300 200 lineto closepath
[10 6] 0 setdash 3 setlinewidth 0 setgray stroke

% rectangle
1 setgray 100 50 200 100 rectfill
0 setgray 1 setlinewidth 100 50 200 100 rectstroke

% caption
/Helvetica 20 selectfont
200 (2D Graphics) stringwidth pop sub 2 div
150 add 90 moveto (2D Graphics) show
```

Figure 2.24: Sample figure in Postscript

permits them to achieve better performance since the parsing stage of the interpreter can be skipped. A standard component of a DPS system is therefore a pre-processor which replaces Postscript code that is embedded for example in a C program by direct calls to the DPS interpreter.

Graphics State. Rendering in Postscript is based on a graphics state where attribute values are stored. The graphics state includes a current user coordinate system in the form of an affine transformation matrix and a current clipping region. In the case of DPS, multiple graphics states can operate on the same physical screen. All values of the graphics state, including clip area and current path, can be pushed on a stack and later restored, enabling clients to efficiently traverse hierarchical structures in Postscript programs.

Path Model. Postscript features a rich set of built-in operators for creating graphics. Its vector graphics are based on a general path model, where arbitrary sequences of lines and cubic Bézier curves are appended to a current path. Postscript uses late rendering for its path model; when the path is complete, it can be stroked using the current attribute values, its interior can be filled using the current color and pattern, or the current clip area can be intersected with the path's outline, restricting output to the resulting new clip area for future operations. The current path can also be modified before painting it, for example by approximating all its curves with short line segments or by replacing it with the contour of a thick curve along the visited path.

User Paths. A special recording mode is not necessary because frequently used building blocks can be placed in Postscript procedures and invoked whenever needed. However, to avoid re-interpreting a procedure several times to draw a path, the current path can be stored in a *user path*. Like procedures, user paths can be invoked repeatedly, but need not be processed by the interpreter. Although user paths are in some ways similar to display lists, they are less powerful than general display lists because they cannot accommodate raster image data.

Fonts. Text is usually rendered by filling glyph outlines. To speed up text output, glyph outlines are scan-converted to raster images the first time a glyph is rendered. The resulting raster images are stored in an internal cache.

Whenever cached glyphs are painted again, the corresponding raster image can be copied from the cache. To improve output quality on low-resolution devices, Postscript outline font descriptions may contain *hints*. A rasterizer may consult these to slightly adjust glyph outlines to the underlying pixel grid prior to scan-conversion. Glyph outlines can also be appended to the current path. They can therefore be filled with a pattern or used to clip other objects. In addition to using outline fonts, Postscript programs can define their own fonts by associating a custom procedure with each glyph. In general, a font is thus a vector of graphical objects, indexed by character code.

Images. Raster images are commonly embedded into Postscript code by encoding them as character streams. When the interpreter encounters an image operator, it suspends normal execution and starts consuming characters from the embedded stream that immediately follows the **image** operator. After all characters of the embedded stream have been read, the interpreter resumes parsing.

Image data can be specified in a variety of formats; its exact layout and the information that is stored in each pixel are defined by the creator of the Postscript file. As a special case, an image can be interpreted as a bitmap mask. To ensure that binary image data can be transmitted over ASCII channels, image data is usually encoded in hexadecimal ASCII streams.

Conclusion. The complete set of Postscript operators is rather large and includes support for different kinds of color spaces, control over half-toning (which is the approximation of a shade of color or gray with a pattern of dots), device-specific settings, and procedural patterns in addition to the features described above. This wide range of functionality and the fact that Postscript descriptions are dynamic (further functionality may be included in the form of user-defined procedures) make it difficult to import Postscript files into other applications and have kept Postscript from becoming a generally accepted format for exchanging graphics. The Portable Document Format (PDF), also developed by Adobe [4], addresses that weakness. PDF uses the same model for rendering graphics but does not support Postscript's programming language constructs. It has therefore become the format of choice for distributing page-oriented documents on the Web.

2.3.5 MetaPost

MetaPost [42, 41] is a programming language for describing 2D graphics. It processes input files and generates corresponding Postscript files, which are then embedded within other documents. To a large part, the syntax and functionality of MetaPost are identical to that of Metafont, Knuth's glyph description language [47]. Unlike Postscript, which is a general programming language with a lot of built-in operators for producing graphics, MetaPost is a special purpose language that relies on dedicated syntax and specialized data types, such as paths, transforms, and colors, to describe graphical scenes.

```
beginfig(1);
  path c, r;
  c = fullcircle scaled 200 shifted (100, 100);
  fill c withcolor .5white;

  pickup pencircle scaled 3;
  draw (200, 0)--(400, 0)--(300, 200)--cycle
    dashed dashpattern(on 10 off 6);

  r = (100, 50)--(350, 50)--(350, 150)--(100, 150)--cycle;
  fill r withcolor white;
  pickup pencircle scaled 1;
  draw r;

  label("2D Graphics", (225, 100));
endfig;
```

Figure 2.25: Sample figure in MetaPost

Language. The MetaPost language supports local scoping, parameterized macros, if-statements, and for-loops. Furthermore, macro definitions can be loaded from input files. In fact, many of MetaPost's predefined operations are implemented as macros that are themselves implemented in terms of more primitive operators. (The same is true in the case of Knuth's TEX system for typesetting [46].) This scheme allows clients to extend MetaPost's functionality by new macros; however, new primitives cannot be added

without recompiling the MetaPost source code.

Paths and Pens. As in Postscript, paths in MetaPost consist of lines and cubic Bézier curves, but specification of Bézier curves in MetaPost is simpler because MetaPost can calculate the position of off-curve control points automatically. If no additional parameters are given, the control points are positioned such that the resulting curve has continuous slope and approximately continuous curvature. Alternatively, the curve specification may contain directives to follow a user-defined direction or curvature at some point. As with other systems, paths can be stroked, filled, or used for clipping. Stroking leads a virtual pen along the visited path. The shape of this pen is by default a unit circle that can be scaled to the desired size. However, the current pen can also be derived from an arbitrary closed path to simplify the creation of calligraphic effects. Furthermore, paths may be dashed and can be drawn with one of several line cap and join styles or with arrow heads at end points.

Labels. Text is integrated in MetaPost descriptions by placing *labels* relative to an anchor point. Labels are not restricted to literal string constants; they may also include TEX formatting instructions for typesetting labels. However, this requires that the TEX typesetting package is available on the same machine.

Images. Unlike other interfaces, MetaPost and Metafont do not support raster images. This is not surprising, considering that Metafont's goal is to generate glyph bitmaps from a description of their outlines.

Linear Equation Solver. A distinctive property of the MetaPost language is its strong support for numerical calculations. It is able to solve linear equation systems, facilitating for example the placement of objects at points where two lines intersect. Moreover, it can intersect arbitrary paths and build new paths from the resulting pieces.

Conclusion. Although MetaPost does not directly render output and merely translates its input to another graphics language, it is an interesting example of how graphical functionality can be presented to clients. Its focus on graphics-specific syntax was a major source of inspiration for Vinci, our own

graphical description language (see Chapter 6).

2.3.6 Scalable Vector Graphics

Scalable Vector Graphics (SVG, [90]) is the name of a specific extension to the eXtensible Markup Language (XML, [89]) for structured information exchange. Thanks to the World Wide Web (WWW), XML is becoming more and more popular for embedding arbitrary data into distributed documents. SVG are the most recent of several proposals to include vector graphics in Web-based documents, but appear to become the solution that is endorsed by the W3 consortium.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG December 1999//EN"
  "http://www.w3.org/Graphics/SVG/SVG-19991203.dtd">
<svg width="405" height="205">
  <desc>graphics example</desc>
  <circle cx="100" cy="100" r="100"
    style="fill:gray" />
  <polygon style="fill:none; stroke:black; stroke-width:3"
    points="200,0  400,0  300,200" />
  <rect x="100" y="50" width="200" height="100"
    style="fill:white; stroke:black;
      stroke-width:1; stroke-dasharray:10 6" />
  <text x="225" y="100" text-anchor="middle"
      style="font-family:Verdana; font-size:20pt; fill:black">
    2D Graphics
  </text>
</svg>
```

Figure 2.26: Sample figure in SVG

XML. XML is not a programming language and does not support the definition of macros or procedures. Instead, XML and SVG descriptions contain a sequence of object definitions. Objects can be named and repeatedly referenced within a document, or they can be imported from another (remote) document by specifying a corresponding universal resource locator (URL).

A SVG description is thus a textual representation of a potentially complex object graph.

Graphical Objects and Attributes. Concerning graphical capabilities, SVG offer a rich set of basic elements from which complex graphics can be constructed. In addition to primitive shapes, including lines, rectangles, ellipses, polylines, and polygons, SVG provide general path objects built from lines, elliptical arcs, and quadratic and cubic Bézier curves.

As with other graphics interfaces, all graphical objects can be stroked, filled, or used for clipping. In addition to solid color and raster image patterns, SVG support color gradients for painting shapes. Moreover, marker objects can automatically be placed at all control points of a path. Markers can also be rotated to match the curve direction, making them suitable for drawing arrow heads.

All painting attributes also apply to text objects; text is thus a special case of a general path. Furthermore, text can be typeset along an arbitrary path object. To ensure that the fonts which are used in the description are available on the system where the SVG are viewed, font outlines may be embedded directly in the corresponding descriptions.

Coordinates. Coordinates are specified within a current user coordinate system, which is stored in an affine transformation matrix. The user coordinate system is initially equal to the viewport coordinate system, which is in turn initialized by the context where the SVG description is embedded. Both the user coordinate system and the viewport coordinate system can be changed within the description.

Imaging Model. SVG are strongly oriented towards raster devices. Not only can raster image objects be included anywhere in a SVG description, but the complete object graph is itself rendered into a raster image first and copied to the output device from there. It is even possible to render a subset of the graph into a raster image by grouping the corresponding set of objects. SVG allow raster images to include opacity information and support alpha blending for smoothly integrating raster images with the background. An image can also be used as a mask. It then restricts output to those areas where it is opaque. This corresponds to our concept of a stencil buffer (see Section 2.1.4).

Special Features. Several features in SVG are specifically targeted at creating embedded graphics for Web pages. To simplify the simulation of shadow effects and embossed structures, images can be sent through a pipeline of effect filters. In this manner, various image processing filters can transform an image that depicts a (partial) SVG description before it is rendered on the output device. In addition, SVG include animation capabilities. These extensions intend to make SVG a vector-based alternative to the widely used GIF raster image format.

As with other XML extensions, scripted events can be attached to SVG descriptions. SVG descriptions can thus react if a user moves the mouse pointer over them or clicks on them. Moreover, each SVG object can contain a hint which indicates to a SVG renderer whether rendering speed, legibility, or geometric precision should be strived for.

Conclusion. The main purpose of SVG is the exchange of graphical information within the framework established by XML. It therefore inherits syntax, structure, object model, and style sheet support from XML. This ensures that embedded SVG can easily be integrated into XML viewers, even if only to ignore them. On the other hand, their verbose and unwieldy syntax makes SVG descriptions tedious to write for humans, suggesting that SVG will primarily be generated by applications. In fact, SVG descriptions can be regarded as an open file format for storing vector-graphics.

2.3.7 Java 2D

Java 2D [74, 39, 50] is a recent API that substantially extends the graphical capabilities of the original Java Abstract Windowing Toolkit (AWT). As many other API for Java, Java 2D is based on an exclusively object-oriented architecture.

Context Objects. The original Java AWT specification features a **Graphics** class for rendering lines, rectangles, ellipses, and text on abstract output devices. By substituting a corresponding extension of **Graphics**, graphical output can be produced on any physical or logical device. However, the interface of **Graphics** only offers a limited amount of functionality. Java 2D therefore introduces **Graphics2D**, an extension of **Graphics** which features additional functionality.

```
public void paint (Graphics g) {
  Graphics2D g2 = (Graphics2D) g;

  // filled circle
  g2.setColor(Color.gray);
  Ellipse2D e = new Ellipse2D.Float(0, 0, 200, 200);
  g2.fill(e);

  // dashed triangle
  g2.setColor(Color.black);
  float[] dash = {10.0f, 6.0f};
  BasicStroke s = new BasicStroke(3.0f, BasicStroke.CAP_BUTT,
    BasicStroke.JOIN_MITER, 10.0f, dash, 0.0f);
  g2.setStroke(s);
  GeneralPath p = new GeneralPath(GeneralPath.NON_ZERO);
  p.moveTo(200, 0); p.lineTo(400, 0); p.lineTo(300, 200);
  p.closePath();
  g2.draw(p);

  // framed rectangle
  g2.setColor(Color.white);
  g2.fillRect(100, 50, 250, 100);
  g2.setColor(Color.black);
  g2.setStroke(new BasicStroke(1.0f));
  g2.drawRect(100, 50, 250, 100);

  // centered string
  Font font = new Font("Arial", Font.PLAIN, 20);
  g2.setFont(font);
  FontMetrics fm = g2.getFontMetrics();
  int w = fm.stringWidth("2D Graphics");
  int dy = -fm.getHeight()/2 + fm.getDescent();
  g2.drawString("2D Graphics", 100 + (250-w)/2, 100+dy);
}
```

Figure 2.27: Sample figure in Java 2D

Rendering Model. Any Java object that implements the **Shape** interface is a graphical object that can be rendered on a **Graphics2D** object. The **Shape** interface lets any object (in particular **Graphics2D** objects) iterate over the contour of a shape object. Contours are modeled as general paths that consist of lines, quadratic curves, and cubic curves. Examples of objects that implement the **Shape** interface are lines, rectangles, ellipses, and general paths. Shape contours can be painted (stroked or filled) or combined with the current clip region. Special **Area** shapes perform boolean operations, thereby supporting what Java 2D calls "constructive area geometry" (CAG) or — in our terms — a shape algebra (see Section 2.1.4).

Stroke Objects. When a shape is stroked, the **Graphics2D** visits the shape's contour and asks the current **Stroke** object of the **Graphics2D** object to convert the resulting path into a new path along the outline of the shape path; this new path is then filled. Any class can in theory implement the **Stroke** interface, but Java 2D itself implements it only in the **BasicStroke** class, which converts the path to an area using the same attributes as other graphics interfaces, namely line width, line cap and join styles, and dash pattern. An interesting point is that a **Stroke** object must convert each shape to an area, thereby forfeiting any possibility of stroking objects with hairlines.

Paint Objects. For filling a shape, the interior of its contour is passed to the current **Paint** object. Examples of objects that implement the **Paint** interface are **Color**, **GradientPaint**, and **TexturePaint**, filling a shape's interior with solid color, a color gradient, or an image pattern, respectively.

Coordinate System. Before shapes are passed to a **Paint** object, Java 2D applies the current transformation to them. Transformations are implemented with affine matrices. Java 2D therefore supports the usual set of affine transformations.

Imaging Model. When two graphical objects overlap, the current **Composite** object computes the color that should appear in that location. The most common operation is to paint objects on top of all previously rendered shapes. When alpha values are involved, however, objects may instead be blended with previously rendered graphics. The necessary alpha values are

derived from the current **Paint** object or from a raster image.

Text. To render text, Java 2D converts a string into a series of glyphs, automatically substituting ligatures for corresponding glyph sequences. The series of glyphs is then treated in the same way as all other shapes. Their contour may be stroked, filled, or added to the clip path. Additional Java 2D classes can assist clients with the display of insertion marks (carets) and with text layout.

Rendering Quality. Like SVG, Java 2D gives its clients some control over the quality of its output with so-called rendering hints. However, these are indeed just hints, as an implementation is free to ignore them.

Raster Images. Java 2D supports three different models for rendering raster images. With the *push model*, an image producer sends image data to a consumer, possibly over a network connection. With the *immediate mode model*, images are stored in and displayed from a memory buffer. With the *pull model*, an image consumer asks a producer to deliver image data when it needs to render them.

For buffered images, several interacting classes manage the pixel buffer and define what information is stored in each pixel and how it is stored. Additional classes implement filtering operations, such as color conversions, affine transformations, or amplitude scaling. A buffered image can be rendered on any **Graphics2D** object. Alternatively, a new **Graphics2D** object can be created on top of an image, using the image as a virtual output device.

Conclusion. Java 2D is a complex graphical object framework that lets its clients customize almost every aspect of 2D rendering. However, its many interacting classes and abstract protocols compromise its efficiency. For example, because all stroked objects must be converted to an area that can be filled, efficient algorithms for drawing hairlines cannot be used. Moreover, since only explicit objects can be rendered, many temporary objects may have to be allocated; this can unnecessarily strain Java's garbage collector.

2.3.8 Comparison

In Table 2.1, we list the criteria from the beginning of Section 2.3 and how each of our examples implements them. An entry of – means that the graphics interface does not implement a feature. ++ denotes full support of a feature, whereas + stands for partially supported features.

Output Devices. While most graphics interfaces can address more than one device, only QuickDraw and Java 2D allow their clients to dynamically customize the set of available output devices (and QuickDraw only in an inelegant manner). There is no technical reason for this restriction since the additional overhead of indirect procedure calls is neglectable compared to the overall number of processor cycles that are needed to draw graphical objects.

Rendering Model. Due to the wide spectrum of our examples, the rendering model is one of the aspects where the differences among graphics interfaces are the most pronounced.

Traditional API such as GKS, QuickDraw, or Oberon are very efficient compared to graphics languages and graphical objects structures, but offer less flexibility. They may also depend on specific output devices, hardware platforms, and programming languages.

Graphics languages offer a high level of abstraction and are independent of output devices and hardware platforms. Although DPS systems prove that it is possible to integrate graphics languages with general purpose programming languages and pre-compile the former to bypass lexical and syntactical analysis, their interpretation remains a complex task, which degrades the performance of the resulting solution. We therefore consider dedicated graphics languages not to be suitable as the primary graphics interface of a system.

Object structures, as in SVG or Java 2D, lie somewhere between traditional API and graphics languages. In the case of SVG, their fixed set of graphical objects and attributes could just as well be mapped onto a matching procedural interface. The extensible object framework of Java 2D, however, allows its clients to customize many aspects, particularly how objects are painted. The price to be paid is performance. Because of its many abstract interfaces and its distributed functionality, Java 2D simply cannot

	GKS	Quick-Draw	Obe-ron	Post-script	Meta-Post	SVG	Java 2D
Output devices							
multiple	++	+	−	++	++	++	++
customizable	−	+	−	−	−	−	++
Rendering							
immediate	++	++	++	+	+	−	+
display list	++	++	−	+	++	−	−
object based	−	−	−	−	−	++	+
graphics state	++	++	−	++	++	−	++
Coordinates							
device	−	−	++	+	−	−	+
normalized	++	+	−	++	++	++	++
user	+	−	−	++	−	++	++
affine	−	−	−	++	++	++	++
Imaging model							
overdraw	++	++	++	++	++	++	++
xor/or	−	++	++	−	+	++	++
blend	−	−	−	−	−	++	++
Clipping							
set operations	−	++	+	−	−	−	++
path	−	−	−	++	++	++	++
Path model							
late rendering	−	−	−	++	++	−	−
object based	−	−	−	−	−	++	++
glyph outlines	−	−	−	++	−	++	++
Fill attributes							
color	++	++	++	++	++	++	++
pattern	+	++	++	++	−	++	++
gradient	−	−	−	−	−	++	++
Stroke attributes							
color	++	++	++	++	++	++	++
pattern	−	++	−	−	−	++	++
dashed	+	−	−	++	++	++	++
caps and joins	−	−	−	++	++	++	++

Table 2.1: Feature comparison

render a black filled rectangle as fast as a monolithic API.

Graphics State. As soon as the number of object and context attributes exceeds about half a dozen, it becomes reasonable to maintain a graphics state where all these values are stored. The only exceptions are Oberon, where attribute values are passed as parameters to render procedures, and SVG, where attribute values are defined per object or inherited from parent nodes in the object graph.

Coordinates and Transformations. Only Oberon operates in device coordinates, all other systems either use a normalized coordinate system or let clients choose a custom coordinate system, usually based on combinations of affine transformations. MetaPost does not permit its coordinate system to be modified, but can apply affine transformations to point coordinates.

Imaging Model. A simple imaging model where new objects completely hide previously painted objects sufficiently deals with most situations. To implement rubber-banding and to paint bitmap masks, API such as QuickDraw and Oberon provide additional *xor* and *or* modes. Recently developed graphics interfaces such as SVG and Java 2D also provide alpha blending, which is especially useful for displaying raster images.

Clipping. Almost all interfaces allow complex clip areas to be specified, either as boolean combinations of other primitive shapes or by intersecting the interior of a path with the existing clip area.

Paths. More recent graphics interfaces tend to include a generic path model, where those that rely on immediate mode always use late rendering for drawing paths. The primitive segments from which paths are built always include lines and cubic curves, in some cases (SVG, Java 2D) also quadratic curves. Moreover, graphics interfaces that support general paths usually allow glyph outlines to be treated as subpaths.

Attributes. Most interfaces offer at least color and pattern values for stroking and filling objects. For stroking only, the standard set of attributes includes dash patterns, cap styles, and join styles. Even Java 2D, which in principle allows fill and stroke attributes to be customized, only pro-

vides one default stroke object with exactly those attributes. Java 2D and SVG are the only interfaces that directly support gradient fills; others (e. g. Postscript) must emulate gradient fills by restricting the clip area to the shape in question and drawing the color gradient explicitly.

2.3.9 Conclusions

According to the above comparison, it seems that there is a canonical set of graphical objects and attributes which is sufficient for generating any graphical output that applications or programmers need to create. Innovation therefore can hardly be achieved by adding new graphical functionality to existing interfaces. Instead, we intend to focus on how graphical capabilities can be accessed and reused.

Depending on the environment where they are used, all of the presented approaches are potentially useful since different user requirements favor different solutions. Some applications need an efficient API, whereas others benefit from being able to specify graphics on an abstract object level for storing them in a platform independent manner. Our conclusion is that different kinds of graphics interfaces are indeed necessary.

Since there seems to be a wide agreement on the set of reasonable features in a graphics interface, we believe that it is possible to choose a few selected concepts and incorporate them in an efficient API, an extensible object framework, and a dedicated graphics language. We conclude this chapter with a list of goals that we seek to achieve in the remainder of this thesis:

- Provide clients with an architecture that offers not one but several graphics interfaces, all of which rely on similar concepts but differ in their level of abstraction, their performance, and their potential for customization.
- Prove the feasibility of a path model that is based on early rendering, which to our knowledge has so far never been implemented.
- Strive for extensibility wherever appropriate because the resulting potential for customization makes it easier for clients to reuse and smoothly integrate existing systems and applications with each other.

CHAPTER 3

Design and Implementation of a Graphics Library

At the core of our 2D graphics architecture lies a powerful API, called *Gfx*, which offers its clients many of the features that are discussed in Chapter 2. It is a traditional API in the sense that it uses a fixed interface and an immediate rendering model. However, unlike other traditional API, it can be customized to support new physical and logical output devices with little effort. Although Gfx is based on a comparably simple rendering model, it provides advanced graphical abstractions such as general paths, custom clip areas, and user coordinate systems.

Figure 3.1 shows the overall structure of Gfx. The central entity where the entire Gfx architecture is anchored is the **Gfx** module. It defines an abstract graphical context type and the operations that such contexts provide. The Gfx interface is discussed in detail in Section 3.2.

Gfx is structured in a modular fashion; much of the functionality that it offers to its clients is implemented in one of several building blocks (subsystems). As illustrated in Figure 3.1, Gfx relies on four different subsystems; the *path subsystem* (see Section 3.3), the *image subsystem* (see Section 3.4), the *font subsystem* (see Section 3.5), and the *region subsystem* (see Section 3.6). These subsystems are for the most part independent of each other and can be reused in isolation. This for example allows an application to only import the image subsystem, but not the entire Gfx library.

Because the Gfx interface is based on an abstract context type, the **Gfx** module itself cannot render output on any particular device. Instead, several concrete context extensions which render on specific physical and logical

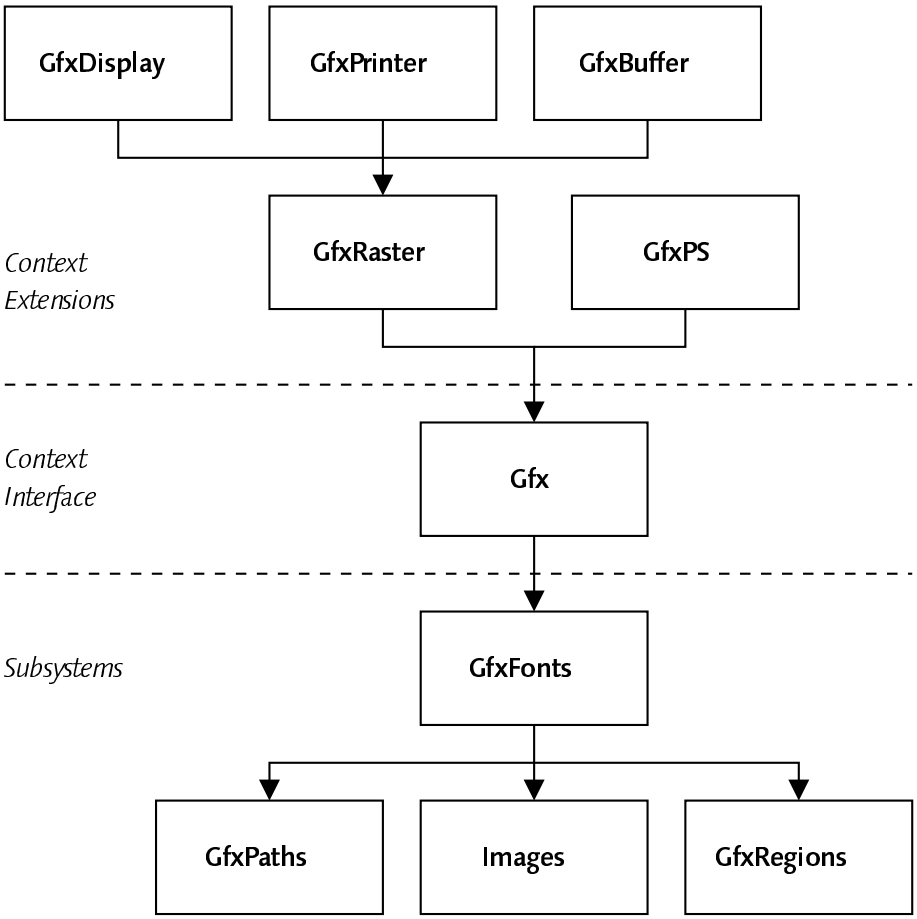

Figure 3.1: Overview of Gfx

devices are derived from this abstract interface. Those extensions that drive raster devices are usually derived from another abstract context, which is defined in module **GfxRaster** (see Section 3.7). The set of concrete contexts that Gfx currently offers includes contexts for drawing on the screen, on printers, into bitmap images, and into Postscript files.

Gfx is built on top of Oberon System 3. Although its concepts are of a general nature and independent of the underlying operating system, we shall frequently motivate design decisions and implementation strategies by referring to equivalent or similar features in Oberon System 3.

3.1 Making a System Extensible

Before delving into the design and implementation aspects of Gfx, a few general remarks on our notion of extensibility are in order. The ultimate goal of an extensible system must be that new components can be dynamically added to an already running system. This requires that the underlying operating system supports dynamic loading of separately compiled modules.

Another requirement is that objects created by these dynamically loaded modules can be integrated within existing data structures in a type-safe manner. Today's object-oriented programming languages achieve this with *type extension* (inclusion polymorphism) and *late binding*. Type extension allows programmers to extend a base type with derived types. Instances of derived types are then accepted wherever instances of the base type are. Late binding ensures that, for dynamically bound procedures, the code of the derived type is executed instead of the code of the base type. Szyperski in [79] calls this combination of object-orientation, separate compilation, and type safety *extensible object-orientation*. Gfx uses extensible object-orientation for integrating new output devices.

3.1.1 Dynamic Loading

If components can be loaded dynamically, we face the question of who is responsible for loading new components. The Oberon system solves the problem with user commands. A *command* is defined as a combination of a module name and the name of a parameterless procedure that is exported from that module. Users activate a command by clicking on a command

name in any displayed text document or by clicking on a graphical control which represents a command. The system loads the module if necessary and transfers control to the command procedure. Hence, new components can be imported into a running system by executing a command of the corresponding module. Programs can start with a small set of core modules and load additional functionality on demand, minimizing application startup time and memory fingerprint.

However, dynamic loading of modules is sometimes desirable or even necessary in circumstances that are not under user control. For instance, users can ask Oberon System 3 to open an arbitrary document by merely supplying the document's file name. If the module that is responsible for loading that specific kind of document cannot be derived from the first few bytes of the file, the document manager extracts the file name extension (the last part of the file name) from the document name and uses it as a key into a table that maps key strings to commands. When the command procedure that is associated with the file name extension is executed, it places a new document object of the correct type in a global variable; this object is then requested to load the document. This technique is far more convenient for users than having to choose different commands for different document types. The same principle is also used in other situations; several tables that associate keys with matching commands are stored in a central registry text file.

Another application of this design pattern [28] can be found in the Netscape Navigator, a well-known Web browser, which uses it for loading "plug-ins". The key is in that case the so-called MIME type of the component to display, whereas the command name is replaced by the name of a separately loadable program library. Gfx uses a similar approach for loading and storing raster image files (see Section 3.4.4) and for loading fonts (see Section 3.5).

3.1.2 Enumeration Procedures

A second important design pattern concerns the use of enumeration procedures for iterating over complex data structures. When enumerating data structures, clients pass a custom procedure to an operation **Enumerate**, which in turn calls the supplied procedure for each element of the structure, as illustrated in the following definitions:

```
TYPE Enumerator = PROCEDURE(element parameters);

PROCEDURE Enumerate(this: complex type; enum: Enumerator);
```

The Oberon system uses enumeration procedures in several places, for example for enumerating files in a directory. Our only gripe with the above scheme is that any additional data that is needed during the traversal must be stored in global variables because most Oberon compilers do not permit nested procedures to be passed as procedure parameters. The natural solution of passing the enumeration procedure a local procedure as the enumerator and storing additional parameters in the local variable space of the calling procedure is thus not feasible. We therefore refine the enumeration principle as follows:

```
TYPE
  Data = RECORD END;
  Enumerator = PROCEDURE(parameters; VAR data: Data);

PROCEDURE Enumerate(this: complex type; enum: Enumerator; VAR data: Data);
```

The advantage of this approach is that clients are free to pass arbitrary extensions of the **Data** type to **Enumerate**, which in turn get passed to the **Enumerator** procedure. Additional data can be stored in fields of the extended **Data** structure instead of in global variables. The resulting pattern is a simplified version of the *Visitor* pattern that is discussed in [28].

Enumeration is not the only way in which a client can iterate over complex data structures. Other approaches usually rely on explicit *iterator* types [28, 73]. Still, enumeration is often easier to implement because the called enumeration procedure determines the flow of control and because the visited data structure can be considered immutable during the traversal. Thus, no state information about which nodes have been visited and where to proceed are necessary.

3.2 Contexts

Most of Gfx's functionality is accessible from the **Gfx** module. **Gfx** exports an abstract type **Context** which defines what Szyperski in [79] calls a *bottleneck interface*: a bottleneck interface binds its clients and implementors to a static interface that cannot be augmented at will. (Implementors of extended types

are free to add additional functionality, but clients of the abstract interface will not be able to take advantage of it.) However, it is because of this restriction to a fixed interface that arbitrary clients (in the role of producers) can render graphics on arbitrary output devices (in the role of consumers). We have encountered the same principle in our discussion of Java 2D with its **Graphics2D** objects in Section 2.3.7. Later chapters will show that Gfx's bottleneck interface is essential for integrating graphical components within component frameworks and mutually within another.

Different parts of the Gfx context interface are described in the following Sections: basic object rendering in 3.2.1, context attributes in 3.2.2, path rendering in 3.2.3, user coordinates in 3.2.4, images in 3.2.5, and temporarily saving and restoring state in 3.2.6.

Context Type. Each context instance contains

- graphical attribute values for rendering graphical objects
- a user coordinate system
- a clip area
- a current path, position, and render state
- a table of procedures

Apart from exporting an abstract interface, **Gfx** implements a substantial amount of default functionality. However, it cannot produce graphics on any output device; this is the responsibility of concrete *context extensions*. **Gfx** thus serves mainly as a facade [28] that conveniently bundles the functionality of several subsystems and concrete output drivers in one place.

Late binding is used to execute procedures of concrete context extensions. For this purpose, each context has a table of procedures associated with it.

```
TYPE
  Context = POINTER TO RECORD
    do: Methods;   (* method table *)
    ...
  END;

  Methods = POINTER TO RECORD
    reset: PROCEDURE (ctxt: Context);
    ...
  END;
```

The method table is explicitly modeled because the original Oberon language [84, 87] does not support type-bound procedures (also known as *methods* or *member functions*), and Oberon dialects that do [58, 57, 37] are not available on as many platforms. Besides, how the context's methods are modeled is of secondary importance because clients are encouraged to use *wrapper procedures* in the **Gfx** module for invoking context methods. One example of such a wrapper procedure is the **Reset** procedure for (re-)initializing a context object.

```
(* reset context to default state *)
PROCEDURE Reset(ctxt: Context);
BEGIN ctxt.do.reset(ctxt)    (* call context method *)
END Reset;
```

Wrapper procedures perform general sanity checks. For example, they test whether attribute values fall within valid ranges or whether a path has been started when a client attempts to append path elements. While most wrapper procedures call a single corresponding procedure in the method table, others have no single matching method and call several methods in sequence to fulfill their purpose (for example the **DrawArc** procedure in the following section).

3.2.1 Standard Objects

The following wrapper procedures render standard objects:

```
PROCEDURE DrawLine(ctxt: Context; x0, y0, x1, y1: REAL; mode: SET);
PROCEDURE DrawArc(ctxt: Context; x, y, r, start, end: REAL; mode: SET);
PROCEDURE DrawRect(ctxt: Context; x0, y0, x1, y1: REAL; mode: SET);
PROCEDURE DrawCircle(ctxt: Context; x, y, r: REAL; mode: SET);
PROCEDURE DrawEllipse(ctxt: Context; x, y, rx, ry: REAL; mode: SET);
PROCEDURE DrawStringAt(ctxt: Context; x, y: REAL; str: ARRAY OF CHAR);
PROCEDURE DrawString(ctxt: Context; str: ARRAY OF CHAR);
```

The **mode** parameter defines the render operation and holds a logical combination of several simple render modes:

```
CONST Record = 0; Fill = 1; Clip = 2; Stroke = 3;
```

The elements of the **mode** parameter are processed in the following order:

1. If **Record** is an element of the render mode, Gfx converts the object to an explicit path structure and stores it in the context's current path. This recorded path can further be processed as discussed in Section 3.2.3.
2. If the render mode includes **Clip**, the context's clip area is reduced to the boolean intersection of the previous clip area with the object's interior. Further output will only appear inside the new clip area.
3. If **Fill** is part of the render mode, the object's interior is filled according to the current fill attribute values. For **DrawString** and **DrawStringAt**, the render mode is implicitly set to {**Fill**}.
4. Finally, if the render mode contains **Stroke**, the object's outline is stroked according to the current stroke attribute values.

3.2.2 Attributes

Each context stores several graphical attribute values, as introduced in Section 2.2.2, which affect the way objects are painted. Clients may read current attribute values from exported fields in the context record. To set new values they invoke corresponding wrapper procedures.

Colors and Patterns. Gfx fills and strokes objects using a combination of colors and raster image patterns:

```
TYPE
  Color = RECORD r, g, b: INTEGER END;   (* red, green, blue in range 0..255 *)

  Pattern = POINTER TO RECORD
    img: Images.Image;   (* raster image *)
    px, py: REAL;   (* pattern pinpoint *)
  END;

VAR Black, White, Red, Green, Blue, Cyan,
  Magenta, Yellow, LGrey, MGrey, DGrey: Color;   (* standard colors *)

PROCEDURE NewPattern(ctxt: Context; img: Images.Image; px, py: REAL): Pattern;
```

NewPattern is responsible for converting the supplied image and pinpoint to a device-specific pattern and returning an appropriate pattern structure. The coordinates of the pinpoint and the pixel size of the pattern image are taken to be in the default coordinate system of the context, not in the current

user coordinate system. A context may scale the pattern image to fit it to the resolution of its output device.

A separate pair of color and pattern values is maintained for stroking and filling since both **Stroke** and **Fill** can be part of the render mode concurrently.

```
TYPE
  Context = POINTER TO RECORD
    ...
    strokeCol, fillCol: Color;   (* default: black *)
    strokePat, fillPat: Pattern;   (* default: none (solid) *)
    ...
  END;

PROCEDURE SetStrokeColor(ctxt: Context; color: Color);
PROCEDURE SetStrokePattern(ctxt: Context; pat: Pattern);
PROCEDURE SetFillColor(ctxt: Context; color: Color);
PROCEDURE SetFillPattern(ctxt: Context; pat: Pattern);
```

Stroke Attributes. Stroking is subject to several additional attributes:

```
CONST
  ButtCap = 1; SquareCap = 2; RoundCap = 3;
  MiterJoin = 1; BevelJoin = 2; RoundJoin = 3;

TYPE
  Context = POINTER TO RECORD
    ...
    lineWidth: REAL;   (* default: 1 *)
    dashPatLen: INTEGER;   (* default: 0 (solid) *)
    dashPatOn, dashPatOff: ARRAY MaxDashPatSize OF REAL;
    dashPhase: REAL;
    capStyle, joinStyle: SHORTINT;  (* default: butt caps, miter joins *)
    styleLimit: REAL;  (* default: 5 (times the line width) *)
    ...
  END;

PROCEDURE SetLineWidth(ctxt: Context; lw: REAL);
PROCEDURE SetDashPattern(ctxt: Context; on, off: ARRAY OF REAL;
  len: INTEGER; phase: REAL);
PROCEDURE SetCapStyle(ctxt: Context; style: SHORTINT);
PROCEDURE SetJoinStyle(ctxt: Context; style: SHORTINT);
PROCEDURE SetStyleLimit(ctxt: Context; limit: REAL);
```

Line width and dash pattern are interpreted in the same coordinate system in which graphical objects are specified. If a client modifies the current user

coordinate system (see Section 3.2.4), line width and dash pattern are thus transformed as well.

Lines and curves are by default one display unit wide, independent of the context's output device. If the line width is set to zero, lines are painted as hairlines, meaning they are as thin as the device allows.

Dashes. Dash patterns consist of up to **MaxDashPatSize** pairs of lengths. The first number of each pair (the *on* length) is the length of the next visible part, the second (the *off* length) the distance from the end of the last visible part to the start of the next. The dash phase serves to offset the starting point within the dash pattern. For example, a pattern of length one, with on and off lengths equal to ten and a dash phase of five, first paints a dash of length five, followed by an indefinite number of dashes of length ten. The space between each dash is of length ten. This is illustrated in Figure 3.2. The corresponding code looks as follows:

```
on[0] := 10.0; off[0] := 10.0;
Gfx.SetDashPattern(ctxt, on, off, 1, 5.0);
Gfx.DrawLine(ctxt, 10, 10, 100, 10);
```

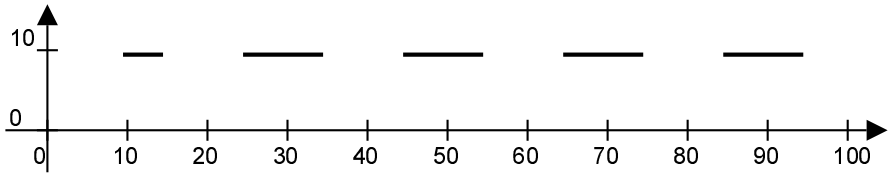

Figure 3.2: Example of a dash pattern with dash phase

Cap and Join Styles. As with many other graphics interfaces, available line cap and line join styles are limited to a fixed set of common styles. As an experiment, Gfx once modeled cap and join styles in procedure variables to make them customizable. Although this variant was later abandoned because the additional complexity did not seem justified, the concept of a general *style limit* in place of a dedicated miter limit has been retained from this earlier version. The style limit restricts the maximal distance of any part on a style's outline to the original line to a multiple of the line width. In contrast, miter limits (see Section 2.2.2) are typically expressed

as the minimally tolerable angle between two lines with which miter joins may still be used. Because the maximal distance of any painted pixel to the original line or curve is the product of the current style limit and line width, calculating the bounding box of the area that gets painted when a thick line is rendered is made simple. Besides, the style limit concept is general enough to accommodate additional cap or join styles should a future Gfx version support them.

Text Attributes. The main attribute for dealing with text is the current font:

```
TYPE
  Context = POINTER TO RECORD
    ...
    font: GfxFonts.Font;
    ...
  END;

PROCEDURE SetFont(ctxt: Context; font: GfxFonts.Font);
PROCEDURE SetFontName(ctxt: Context; name: ARRAY OF CHAR;
  size: INTEGER);
PROCEDURE GetStringWidth(ctxt: Context; str: ARRAY OF CHAR;
  VAR dx, dy: REAL);
```

A typeface is defined by its family and style name (e. g. "Oberon-Bold"), and must be instanced at the desired point size. Gfx's font subsystem, discussed in Section 3.5, is responsible for providing bitmaps and outlines to paint glyphs within the current user coordinate system on the attached output device.

The procedure **GetStringWidth** returns the vector by which the current point would be advanced if the supplied string were drawn in the current context. Its *dy* component is usually zero because text is normally typeset horizontally from left to right. However, when the instance matrix of the current font includes rotations or shear transforms, the advance vector may no longer be horizontal.

Flatness. The last graphical attribute of a context is its current flatness value:

```
TYPE
  Context = POINTER TO RECORD
    ...
```

```
    flatness: REAL;   (* default: 1 device pixel *)
    ...
  END;

PROCEDURE SetFlatness(ctxt: Context; flatness: REAL);
```

Flatness is an error measure that defines the maximally tolerable distance of any painted pixel from the true outline of a graphical object when the object is drawn as a hairline. Flatness primarily plays a role when curves are approximated with lines (see Section 3.3.3). The smaller the flatness, the smoother approximated curves appear. Unlike other attributes, flatness is specified in device pixels.

3.2.3 Paths

A feature which distinguishes Gfx from other graphics interfaces is its use of the early rendering model that we introduced in Section 2.2.3 for drawing arbitrary paths. With early rendering, clients define how a path will be painted when they start its specification, whereas with late rendering the render operation becomes known when the specification of a path ends. While the numbers that are listed in Appendix C.2 indicate that the gain in speed that results from using early rendering is marginal in most cases, early rendering is never slower than late rendering. Besides, late rendering uses a fixed amount of temporary memory, whereas the memory requirements of late rendering grow with the length of stored paths.

The internal structure of a path must follow certain rules, which are summarized in the following EBNF grammar.

```
Path = Begin Subpath {Subpath} End.
Subpath = (Enter | MoveTo) Segment {Segment} [Close | Exit].
Segment = LineTo | ArcTo | BezierTo.
```

The following paragraphs explain the semantics of these path elements.

Begin and End. Gfx offers no single wrapper procedure for specifying general paths. Instead, path definitions start with **Begin** and end with **End**:

```
CONST InPath = 5;

TYPE
  Context = POINTER TO RECORD
```

```
    ...
    mode: SET;   (* current render mode *)
    ...
  END

PROCEDURE Begin(ctxt: Context; mode: SET);
PROCEDURE End(ctxt: Context);
```

Calling **Begin** has two effects. First, the context's **mode** field is set according to the supplied **mode** parameter. This render mode is a combination of the simple render operations from Section 3.2.1. Second, the **InPath** element is included in the **mode** field. This allows subpaths to be started, but at the same time locks the current graphics state and prevents further changes to the context's graphical attributes (except for the current font). The wrapper procedures for changing context attributes will reject any such attempt. This is necessary because — due to early rendering — each context is free to render path elements whenever it sees fit. If changes to graphical attributes were permitted within a path, some but not all path elements might already have been painted using the old values, leading to inconsistent output. While late rendering does not have this consistency problem and may therefore allow its clients to modify graphical attributes during path specifications, it may confuse its clients because only the final attribute values will be used for the entire path, not the attribute values that are in effect when a path element is specified.

After rendering all path elements or before switching to a new path altogether, clients must terminate the current path by calling **End**. This resets the render mode and flushes any pending output.

Subpath Construction with *MoveTo* and *Close*.

```
CONST InSubpath = 6;

TYPE
  Context = POINTER TO RECORD
    ...
    cpx, cpy: REAL;   (* location of current point *)
    ...
  END;

PROCEDURE MoveTo(ctxt: Context; x, y: REAL);
PROCEDURE LineTo(ctxt: Context; x, y: REAL);
```

```
PROCEDURE ArcTo(ctxt: Context; x, y, x0, y0, x1, y1, x2, y2: REAL);
PROCEDURE BezierTo(ctxt: Context; x, y, x1, y1, x2, y2: REAL);
PROCEDURE Close(ctxt: Context);
```

A new subpath is started by calling **MoveTo**. This includes the **InSubpath** element in the context's render mode and moves the *current point*, whose coordinates are stored in the **cpx** and **cpy** fields, to the corresponding position. Lines, elliptical arcs, and cubic Bézier curves are subsequently appended to the current subpath with **LineTo**, **ArcTo**, and **BezierTo**. Each such path segment originates at the current point and moves the current point to the end point of the new path segment.

The current subpath is terminated either by starting a new one with **MoveTo**, by ending the current path with **End**, or with **Close**. Closing a subpath means to append a straight line from its current end point back to its starting point if necessary. Besides, it signals Gfx to render a line join between the last and first subpath segment instead of line caps at both ends.

Path Segments. For all path segments, the first pair of parameters designates the coordinates of their end point. Bézier curves are specified with two additional points that determine their off-curve control points. For elliptical arcs, a general ellipse must be specified by supplying the coordinates of its center and two points at the end of its conjugate diameter vectors (as in Figure 2.8). For example, the first quarter of an ellipse with center (30, 20), radius 20 in *x* direction, and radius 15 in *y* direction is rendered with

```
cx := 30; cy := 20; rx := 20; ry := 15;
Gfx.MoveTo(ctxt, cx+rx, cy);  (* start at end of horizontal diameter *)
Gfx.ArcTo(ctxt,
  cx, cy+ry,   (* end point of arc *)
  cx, cy,   (* center of ellipse *)
  cx+rx, cy,   (* end point of first conjugate diameter vector *)
  cx, cy+ry);   (* end point of second conjugate diameter vector *)
```

Depending on the sign of the cross product between the two conjugate diameter vectors, the arc is rendered clockwise or counter-clockwise. If the current point is not located on the ellipse, a straight line to the nearest point on the ellipse is drawn first. Likewise, a straight line segment is appended after the arc if the target point is not located on the ellipse. The following example exploits this to outline the intersection of two concentric circles and a sector, as shown in Figure 3.3:

```
Gfx.MoveTo(c, 50, 0);
Gfx.ArcTo(c, 50/sqrt2, 50/sqrt2,  0, 0,  100, 0,  0, 100);
Gfx.ArcTo(c, 50, 0,  0, 0,  50, 0,  0, 50);
Gfx.Close(c);
```

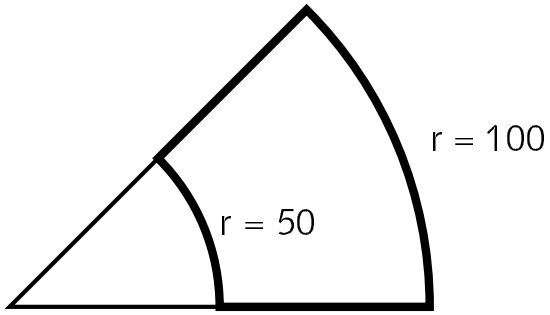

Figure 3.3: Output of arc example program (thick path)

The first arc is drawn counter-clockwise and also draws the straight line segments because neither start nor end point is located on the arc. The second arc runs clockwise and leads back to the starting point. The final **Close** is only necessary to make Gfx draw a line join at that point.

General arcs have many parameters and are inherently complex to specify. Other graphics interfaces face the same problem. For example, SVG descriptions (see Section 2.3.6) expect two scalar radii and a rotation angle for drawing general ellipses (not arcs). Postscript only has primitives for painting circular arcs, relying on transformations of the coordinate system to render elliptical arcs. Gfx provides the **DrawArc**, **DrawCircle**, and **DrawEllipse** wrapper procedures (see Section 3.2.1) to simplify the most common applications of elliptical arcs.

Example. With the features discussed so far, we are able to show how the sample figure from Section 2.3 can be rendered with Gfx.

```
PROCEDURE Paint (c: Context);
VAR on, off: ARRAY 1 OF REAL; dx, dy: REAL;
BEGIN
  (* circle *)
  Gfx.SetFillColor(c, Gfx.MGrey);
  Gfx.DrawCircle(c, 100, 100, 100, {Gfx.Fill});
```

```
    (* triangle *)
    Gfx.SetLineWidth(c, 3);
    on[0] := 10; off[0] := 6;
    Gfx.SetDashPattern(c, on, off, 1, 0);
    Gfx.Begin(c, {Gfx.Stroke});   (* start path *)
    Gfx.MoveTo(c, 200, 0);   (* start subpath *)
    Gfx.LineTo(c, 400, 0); Gfx.LineTo(c, 300, 200);   (* append two triangle sides *)
    Gfx.Close(c);   (* close and end subpath *)
    Gfx.End(c);   (* end path and flush output *)

    (* rectangle *)
    Gfx.SetFillColor(c, Gfx.White);
    Gfx.SetLineWidth(c, 1);
    Gfx.DrawRect(c, 100, 50, 350, 150, {Gfx.Fill, Gfx.Stroke});

    (* caption *)
    Gfx.SetFontName(c, "Oberon", 20);
    Gfx.SetFillColor(c, Gfx.Black);
    Gfx.GetStringWidth(c, "2D Graphics", dx, dy);
    Gfx.DrawStringAt(c, 225-dx/2, 90, "2D Graphics");
END Paint;
```

Enter and Exit. As discussed in Section 2.2.3, closing a previously unclosed path violates the spirit of a path model with early rendering because the decision whether to render a line cap or a line join at the start point must be deferred until the graphics interface knows whether the subpath is closed or left open. Gfx thus provides procedures **Enter** and **Exit** as an alternative for starting and ending subpaths.

```
PROCEDURE Enter(ctxt: Context; x, y, dx, dy: REAL);
PROCEDURE Exit(ctxt: Context; dx, dy: REAL);
```

Enter and **Exit** require an additional pair of coordinates to define the direction of the line or curve that logically precedes the path segment after **Enter** and of the one that logically succeeds the segment before **Exit**. For instance, the triangle from the previous example is alternatively rendered as follows. The direction used in **Enter** is the vector from the third to the first point; the one in **Exit** is the vector from the first to the second point.

```
Gfx.Begin(c, {Gfx.Stroke});   (* start path *)
Gfx.Enter(c, 200, 0,  -100, -200);   (* enter subpath at lower left corner *)
Gfx.LineTo(c, 400, 0); Gfx.LineTo(c, 300, 200); Gfx.LineTo(c, 200, 0);
(* must draw all three sides since we cannot use Close *)
```

```
Gfx.Exit(c, 200, 0);   (* exit subpath *)
Gfx.End(c);   (* end path and flush output *)
```

Although **Enter** and **Exit** at first glance merely seem to be a more clumsy version of **MoveTo** and **Close**, they not only emphasize the symmetrical nature of subpath ends, but are also convenient for storing paths explicitly (see Section 3.3) and for driving pen objects (to be discussed in Section 4.3). In fact, earlier Gfx versions used the **Enter**/**Exit** approach exclusively; only recently has the slightly more convenient **MoveTo**/**Close** combination been added.

Glyph Outlines. Paths naturally integrate text if text is interpreted as a series of subpaths that outline glyph objects. While a context is in path mode (but not in subpath mode), text can be rendered at the current point or at any given point with one of the following procedures.

```
PROCEDURE ShowAt(ctxt: Context; x, y: REAL; str: ARRAY OF CHAR);
PROCEDURE Show(ctxt: Context; str: ARRAY OF CHAR);
```

The difference between **Show** and **ShowAt** is that the former places the first glyph at the position of the current point. Both advance the current point to the point after the last glyph.

On a conceptual level, the **Show** and **ShowAt** operations append glyph outlines to the current path and leave it to the path engine to render them. However, because the render mode is known at the time **Show** and **ShowAt** are invoked, Gfx is free to immediately paint glyph raster images instead of later filling glyph outlines if the current render mode indicates that the path will later be filled anyway. This is an advantage of early rendering over late rendering. Gfx can render text efficiently and in high quality without distinguishing between filled text shapes and explicit outlines as for example Postscript does. (The figures in Appendix C.2 suggest that rendering glyph bitmaps is about ten times faster than filling glyph outlines.) The wrapper procedure **DrawStringAt** can thus be implemented as follows.

```
PROCEDURE DrawStringAt(ctxt: Context; x, y: REAL; str: ARRAY OF CHAR);
BEGIN
  ASSERT(~(InPath IN ctxt.mode), 100);   (* sanity check *)
  ctxt.do.begin(ctxt, {Fill});
  ctxt.do.show(ctxt, x, y, str);
  ctxt.do.end(ctxt)
END DrawStringAt;
```

Other Path Operations. The remaining Gfx procedures deal with explicit paths. If the context's render mode includes the **Record** element, the visited path is stored in a **Path** structure (explained in Section 3.3).

```
TYPE
  Context = POINTER TO RECORD
    ...
    path: GfxPaths.Path;   (* current path *)
    ...
  END;

PROCEDURE Flatten(ctxt: Context);
PROCEDURE Outline(ctxt: Context);
PROCEDURE Render(ctxt: Context; mode: SET);
PROCEDURE DrawPath(ctxt: Context; path: GfxPaths.Path; mode: SET);
```

Flatten replaces all arcs and Bézier curves in the recorded path with a series of straight lines. It relies on the **EnumFlattened** procedure from module **GfxPaths** (see Section 3.3.3) to approximate these curves with lines according to the current flatness value. **Flatten** is equivalent to the Postscript operator **flattenpath**.

The **Outline** procedure replaces the current path with the outline of the shape that would be painted if the path were stroked. The current line width, dash pattern, cap style, and join style all influence the shape of the resulting outline. A line width of zero is treated as a special case: the path is then replaced with a series of dashes according to the current dash pattern but must still be stroked to paint it. **Outline** is similar to the Postscript operator **strokepath**.

Clients are of course free to apply other modifications to recorded paths. To render the current path with the render mode of their choice, clients can call **Render**. Although **DrawPath** can also be used to render explicitly stored path structures, it interprets paths as being specified in user coordinates and therefore transforms them to device space first. This is not the case with **Render** since the context's current path is always stored in device coordinates.

3.2.4 Coordinate System

Gfx stores a context's current user coordinate system as a 3×2 matrix (the *current transformation matrix*) in the context's **ctm** field.

```
TYPE
  Context = POINTER TO RECORD
    ...
    ctm: GfxMatrix.Matrix;
    ...
  END;
```

Once coordinates have been transformed to device space, the grid lines between pixels are located at integer coordinates. The point (0, 0) is therefore at the lower left corner of the lower left pixel, which has its center at coordinates (0.5, 0.5).

The following procedures modify the current user coordinate system.

```
PROCEDURE Reset(ctxt: Context);

PROCEDURE ResetCTM(ctxt: Context);
PROCEDURE SetCTM(ctxt: Context; VAR mat: GfxMatrix.Matrix);
PROCEDURE Translate(ctxt: Context; dx, dy: REAL);
PROCEDURE Scale(ctxt: Context; sx, sy: REAL);
PROCEDURE ScaleAt(ctxt: Context; sx, sy, x, y: REAL);
PROCEDURE Rotate(ctxt: Context; sin, cos: REAL);
PROCEDURE RotateAt(ctxt: Context; sin, cos, x, y: REAL);
PROCEDURE Concat(ctxt: Context; VAR mat: GfxMatrix.Matrix);
```

When the **ctm** is reset with **Reset** or **ResetCTM**, Gfx initializes it to a normalized default coordinate system, which has its origin at the lower left corner of the paintable area of the output device and is scaled such that one unit corresponds to $\frac{1}{91.44}$ of an inch. This was the display resolution of the machine on which the first versions of the Oberon system were developed; hence, all character bitmaps of the Oberon default font were manually tuned to that resolution. For reasons of convenience, current Oberon implementations frequently assume the same resolution in their display drivers, even if the actual resolution is different. Gfx follows this convention to achieve a higher performance when rendering glyphs. If a different default unit were chosen, all glyph patterns of the default font would have to be scaled.

SetCTM is often necessary for restoring a previously saved **ctm** after temporarily modifying it. Another use is within operations that paint in device space and thus need to set the **ctm** to the identity matrix.

The remaining procedures initialize affine transformation matrices and prepend them to the **ctm**. The effect is that any future point will be transformed according to the new transformation before being subject to the

previous **ctm**. An equivalent interpretation is that the operation transforms the current coordinate system in which future points will be evaluated. For example, the following program produces the two squares which are displayed in Figure 3.4.

```
Gfx.Translate(c, 100, 50);
Gfx.Rotate(c, sqrt(3)/2, 1/2);   (* 30 degrees *)
Gfx.DrawRect(c, 0, 0, 20, 20, Gfx.Stroke);
Gfx.Translate(c, 10, 10);
Gfx.Scale(c, 2, 2);
Gfx.DrawRect(c, 0, 0, 20, 20, Gfx.Stroke);
```

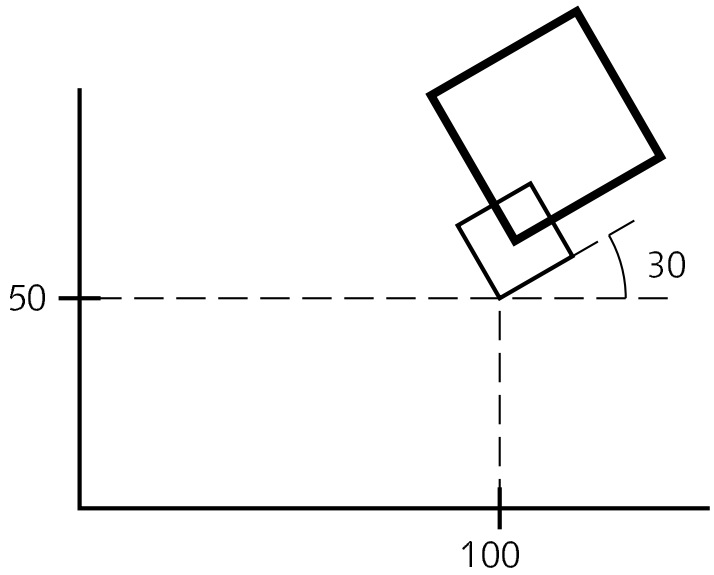

Figure 3.4: Coordinate transformations

Rotations and scale transformations have an immutable point of reference, which is located at the origin of the current coordinate system. To use another reference point, the coordinate system has to be shifted to make the reference coincide with the origin. Then the rotation or scale matrix can be prepended. Finally, the resulting transformation must be shifted back to its previous origin. To simplify the process, the **ScaleAt** and **RotateAt** procedures accept an arbitrary reference point and include the necessary translations.

Current Attribute Matrix. To make path specification as flexible as possible, operations that affect the **ctm** are also permitted while a context is processing a path. However, changes to the **ctm** not only affect coordinates, but also graphical attributes such as line width and dash pattern. Since a Gfx context is free to paint path segments whenever it sees fit, this could have the same consequences as allowing changes to graphical attributes while inside a path. Gfx therefore saves the value of the **ctm** in a second field called **cam** (the *current attribute matrix*) when **Begin** is invoked to start a path.

```
TYPE
  Context = POINTER TO RECORD
    ...
    cam: GfxMatrix.Matrix;
    ...
  END;
```

The current attribute matrix is used for mapping attribute values to device coordinates until the complete path has been painted. Only then is the current attribute matrix synchronized again with the current transformation matrix.

To illustrate how transformations can be used while a path is being specified, the following program draws a star with five arms. It does so by repeatedly translating the origin to one corner of the star, rotating the whole coordinate system, and appending a line that leads to the next corner. The output of this program is displayed in Figure 3.5.

```
Gfx.Begin(c, {Gfx.Stroke});
Gfx.MoveTo(c, 200, 200);
Gfx.Translate(c, c.cpx, c.cpy);
FOR i := 1 TO 5 DO
  Gfx.LineTo(c, 100, 0);
  Gfx.Translate(c, c.cpx, c.cpy);
  Gfx.Rotate(c, Math.sin(-4*Math.pi/5), Math.cos(-4*Math.pi/5))
END;
Gfx.Close(c);
Gfx.End(c)
```

3.2.5 Images

Although Gfx is oriented towards arbitrarily scalable vector graphics, it also renders raster images.

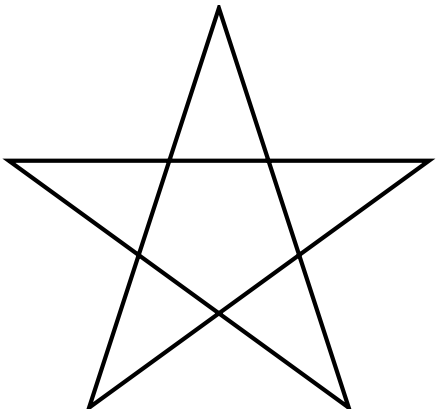

Figure 3.5: Example of changing the current transformation within a path

```
PROCEDURE DrawImageAt(ctxt: Context; x, y: REAL; img: Images.Image;
  VAR filter: GfxImages.Filter);
```

Each pixel in the image corresponds to one unit of the current user coordinate system. If the current user space has been transformed or if the current device has a resolution that differs from the default resolution, the raster image is accordingly transformed. How the image is blended with the destination plane and how transformed images are filtered is defined by the **filter** parameter. Images and filters are implemented as a separate subsystem and are further discussed in Section 3.4.

3.2.6 Saving and Restoring State

Complex graphical scenes are likely to be modeled as hierarchies, be it explicitly in the form of a data structure or implicitly in the form of a procedure call graph. Often a model establishes the convention that any node in the hierarchy may change attributes of the graphics state as long as it restores the previous state before it returns control. Since all graphical attributes and the current coordinate system are available as public fields of the **Context** structure and can be set with a single procedure call, saving and restoring single attributes merely involves assigning their current values to temporary variables and later establishing them again, as outlined in the following example.

```
VAR col: Gfx.Color;
```

```
...
col := ctxt.strokeCol;   (* remember old value *)
Gfx.SetStrokeColor(ctxt, Gfx.Red);   (* set new value *)
...   (* stroke something *)
Gfx.SetStrokeColor(ctxt, col);   (* restore old value *)
```

Clip Area. The current clip area, although managed by concrete extensions in a device-dependent manner, can also be saved and later restored.

```
TYPE
  ClipArea = POINTER TO RECORD END;

PROCEDURE GetClip(ctxt: Context): ClipArea;
PROCEDURE SetClip(ctxt: Context; clip: ClipArea);

PROCEDURE GetClipRect(ctxt: Context; VAR llx, lly, urx, ury: REAL);
```

ClipArea is an abstract type and does not contain any information. Moreover, there is no corresponding field in the **Context** structure because the data structure for storing clip areas is potentially dynamic and more complex than the scalar values that are used for other attributes. If clients were allowed to obtain a reference to the clip area structure without the context being notified, contexts would not be permitted to reuse the existing data structure when they modify the clip area, which would force them to always allocate a new structure. To at least get an idea about the extent of the current clip area, clients can retrieve its bounding rectangle with **GetClipRect**.

Save and Restore. If child nodes in a graph structure cannot be trusted to revert the changes they apply to a context, a parent node may need to store (and later reestablish) the whole graphics state before passing control to its children. Similarly, if a client needs to change several attributes at once, it has to assign each of them to a temporary variable and later restore them. If many attributes need to be saved, this may enlarge the code of the client disproportionately. To save and restore several attributes as a group, Gfx provides procedures **Save** and **Restore**. These save and later restore an arbitrary subset of graphical attributes, current transformation matrix, and current clip area.

```
CONST
  fillColPat = 0; strokeColPat = 1; lineWidth = 2; dashPat = 3;
  capStyle = 4; joinStyle = 5; styleLimit = 6;
```

```
    flatness = 7; font = 8; ctm = 9; clip = 10;
    strokeAttr = {strokeColPat..styleLimit};
    attr = {fillColPat..font};
    all = attr + {ctm, clip};

  TYPE
    State = RECORD END;

  PROCEDURE Save(ctxt: Context; elems: SET; VAR state: State);
  PROCEDURE Restore(ctxt: Context; state: State);
```

The following code example stores all graphical attributes, including clip area and current transformation matrix, in a local variable and later restores them.

```
VAR state: Gfx.State;
...
Gfx.Save(ctxt, Gfx.all, state);
...   (* modify attributes of ctxt at will *)
Gfx.Restore(ctxt, state);
```

3.3 Path Subsystem

When graphical objects are immediately scan-converted and painted, no explicit path data structures are needed. However, when the render mode includes the **Record** element, the current path is stored in a **Path** structure, which is defined in a separate module **GfxPaths**. Gfx also needs to store the visited path if the render mode includes **Stroke** together with either **Fill** or **Clip**, as discussed in Section 3.7.2.

We discuss the path data structure in detail in Section 3.3.1, path construction in Section 3.3.2, path iterators in Section 3.3.3, path queries in Section 3.3.4, and miscellaneous path operations in Section 3.3.5.

3.3.1 Path Data Structure

Paths are inhomogeneous sequences of primitive path elements. Since path operations usually require that all path elements are visited in order, the intuitive way to store paths is as a list of path elements, which in turn contain information about their control points. There are, however, two minor points of concern with this approach. First, it allocates many

small blocks of memory for individual path elements, which may strain the memory manager of the underlying operating system. Second, one segment's end point is the starting point of its successor and should not be stored twice.

The path representation we have chosen uses two dynamically growing lists: one for storing path element types and another for storing the corresponding control point coordinates. Each list node contains a fixed number of element types or coordinates. This results in less separate memory blocks being allocated, but retains the possibility of dynamic growth.

```
CONST Stop = 0; Enter = 1; Line = 2; Arc = 3; Bezier = 4; Exit = 5;

TYPE
  ElemBlock = POINTER TO RECORD   (private)
    next: ElemBlock;
    elem: ARRAY ElemBlockSize OF SHORTINT;
    coords: INTEGER;
  END;

  CoordBlock = POINTER TO RECORD   (private)
    next: CoordBlock;
    x, y: ARRAY CoordBlockSize OF REAL;
  END;

  Path = POINTER TO RECORD
    elems, coords: INTEGER;
    firstEB, lastEB: ElemBlock;   (private)
    firstCB, lastCB: CoordBlock;   (private)
  END;

VAR Coords: ARRAY Exit+1 OF SHORTINT;   (private)
```

By keeping a reference to the last element and coordinate blocks, new blocks can quickly be appended when the path grows. Alternatively, we could have used two dynamic arrays for storing elements and coordinates, but would have to reallocate them and copy their old contents to a new block whenever we run out of space. Note that element blocks and coordinate blocks grow independent of each other since the number of coordinates per element depends on individual element types and is not constant, as displayed in the following table.

Element	Coordinate Pairs	Comment
Enter	2	direction, start point
Line	1	end point
Arc	4	center, 2 diameter end points, end point
Bezier	3	2 control points, end point
Exit	1	direction

Element sequences are required to conform to the following EBNF rules.

```
Sequence = {Enter Segment {Segment} Exit}.
Segment = Line | Arc | Bezier.
```

3.3.2 Path Construction

Path construction is straightforward and follows the same paradigm as path specification within Gfx contexts. As soon as a path is initialized with **Clear**, subpaths can be added to it. Each subpath must be started with **AddEnter**, followed by at least one curve segment, and terminated with **AddExit**.

```
PROCEDURE Clear(path: Path);

PROCEDURE AddEnter(path: Path; x, y, dx, dy: REAL);
PROCEDURE AddLine(path: Path; x, y: REAL);
PROCEDURE AddArc(path: Path; x, y, x0, y0, x1, y1, x2, y2: REAL);
PROCEDURE AddBezier(path: Path; x, y, x1, y1, x2, y2: REAL);
PROCEDURE AddExit(path: Path; dx, dy: REAL);
PROCEDURE AddRect(path: Path; llx, lly, urx, ury: REAL);

PROCEDURE Close(path: Path);

PROCEDURE Append(to, from: Path);
PROCEDURE Copy(src, dst: Path);
```

Note that there is no operation **AddMoveTo(p, x, y)** that would match the **MoveTo** operation in module **Gfx** (see Section 3.2.3). However, an equivalent effect is achieved if an **Enter** element with zero direction is appended, as for example with **AddEnter(p, x, y, 0, 0)**. Because rectangles are so common, **GfxPaths** provides a procedure **AddRect**, which appends a closed rectangular subpath to a path. A complete path can also be appended to another with **Append**, and path contents can be copied with **Copy**.

3.3.3 Iterators

Paths play such an important role in Gfx that **GfxPaths** provides two different methods for traversing them: *scanners* and *enumerators*. The principle of enumerating elements of a data structure is discussed in Section 3.1.2. Path enumeration is for example used when paths are flattened (see page 86). Scanners, on the other hand, give clients explicit control over how and when a *focus* is moved from one element to the next. They are for example used in **Gfx.Outline**, which converts the current path of a Gfx context to a corresponding outline (see Section 3.2.3). The following paragraphs discuss scanners and enumerators in more detail.

Scanners. A scanner is connected to a path with **Open**, which places the scanner's focus on any path element, zero being the first position and **path.elems** − 1 being the last. The path can then be inspected element by element. The type of the current path element is available from the scanner's **elem** field; its coordinates and direction vectors are stored in various other **Scanner** fields. Once a path element has been processed, calling **Scan** moves the focus to the next element. **elem** is set to **Stop** when no more elements are available.

```
TYPE
  Scanner = RECORD
    path: Path;
    pos: INTEGER;
    elem: INTEGER;
    x, y: REAL;
    dx, dy: REAL;
    x0, y0, x1, y1, x2, y2: REAL;
  END;

PROCEDURE Open(VAR s: Scanner; path: Path; pos: INTEGER);
PROCEDURE Scan(VAR s: Scanner);
```

Enumerators. When a path is enumerated, path traversal is under the control of the **GfxPaths** module. For each path element, **Enumerate** calls the passed **Enumerator**, which is the procedure that a client provides for handling path elements.

Enumerate stores all information about the current path element in an **EnumData** structure. Because clients can pass **Enumerate** an arbitrary

type extension of **EnumData**, they are free to store additional data they need to access during the traversal there. For example, when enumerating a (flattened) path to calculate its length, the supplied enumeration data parameter might contain a field that represents the accumulated length of all path segments that have already been visited.

```
TYPE
  EnumData = RECORD
    elem: INTEGER;
    x, y, dx, dy, x0, y0, x1, y1, x2, y2: REAL;
  END;
  Enumerator = PROCEDURE(VAR data: EnumData);

PROCEDURE Enumerate(path: Path; enum: Enumerator; VAR data: EnumData);
PROCEDURE EnumFlattened(path: Path; flatness: REAL;
  enum: Enumerator; VAR data: EnumData);

PROCEDURE EnumArc(x0, y0, x1, y1, x2, y2, x, y, flatness: REAL;
  enum: Enumerator; VAR data: EnumData);
PROCEDURE EnumBezier(x1, y1, x2, y2, x, y, flatness: REAL;
  enum: Enumerator; VAR data: EnumData);
PROCEDURE EnumSpline(VAR x, y: ARRAY OF REAL; n: LONGINT;
  closed: BOOLEAN; enum: Enumerator; VAR data: EnumData);
```

Although enumeration gives clients less control over the traversal than using a scanner, it has the advantage that **GfxPaths** can modify the visited path on the fly. For instance, calling **EnumFlattened** instead of **Enumerate** automatically approximates arcs and Bézier curves with straight lines by in turn calling **EnumArc** and **EnumBezier** upon encountering arc or Bézier elements. Furthermore, enumeration allows clients to visit other kinds of curves by converting them to path segments as they traverse them. This conversion can be done without explicitly storing the resulting segments in a path structure, as illustrated by procedure **EnumSpline**, which enumerates a natural spline, passed as an array of control points, as a sequence of path elements by converting it to cubic Bézier curves. (Natural splines are for example used in Oberon outline font files.) In summary, using scanners for traversing the elements of a path is often more convenient for clients, but offers less options than using enumerators.

Details of Flattening. For flattening elliptical arcs, we use a variant of Fellner and Helmberg's algorithm [25], which traces circles by developing

the parametric circle formulation

$$x(t) = r \cdot \cos t$$
$$y(t) = r \cdot \sin t$$

into a series of points on the circle. The original algorithm limits the increment Δt as a function of the maximal ellipse radius to guarantee that the distance from one point to the next never exceeds one pixel. To improve the visual appearance of diagonal steps, a post-processing step replaces pairs that comprise a horizontal and a vertical step by diagonal steps.

Although path coordinates are not necessarily related to pixels, we can use the **flatness** argument, which denotes the maximally permitted distance between enumerated points and the true ellipse, as a virtual pixel size from which we can derive a conservative estimate of the required increment Δt. To reduce the number of lines that are generated, we add a post-processing step in which we skip ellipse points until the distance between the ellipse and its tangent vector at the last accepted point exceeds the flatness distance.

EnumBezier flattens cubic Bézier curves by recursively subdividing them until the distance between their midpoint (at $t = \frac{1}{2}$) and the line between their end points is smaller than the desired flatness, as illustrated by the following algorithm.

```
PROCEDURE subdiv(t, x0, x1, x2, x3: REAL; VAR a1, a2, m, b1, b2: REAL);
  VAR s, x12: REAL;
BEGIN
  (* recursive subdivision (deCasteljau) *)
  s := 1 - t;
  a1 := s*x0 + t*x1; b2 := s*x2 + t*x3; x12 := s*x1 + t*x2;
  a2 := s*a1 + t*x12; b1 := s*x12 + t*b2;
  m := s*a2 + t*b1
END subdiv;

PROCEDURE draw (x0, y0, x1, y1, x2, y2, x, y: REAL);
  VAR x01, x11, x12, x22, x23, y01, y11, y12, y22, y23, dx, dy, ex, ey, cp: REAL;
BEGIN
  (*
    Draws a normalized Bézier curve.
    Normalized curves have no points of inflection and no local
    extrema in x or y
  *)
  subdiv(0.5, x0, x1, x2, x, x01, x11, x12, x22, x23);
  subdiv(0.5, y0, y1, y2, y, y01, y11, y12, y22, y23);
```

```
    dx := x12 - x0; dy := y12 - y0;
    ex := x - x0; ey := y - y0;
    cp := dx*ey - dy*ex;  (* = |d| * |e| * sin(angle(d, e)) *)
    IF cp*cp <= (flatness*flatness) * (ex*ex + ey*ey) THEN
      (* |d| * sin(angle(d, e)) <= flatness *)
      (* => distance from (x12, y12) to line (x0, y0, x, y) not greater than flatness *)
      enumerate line from (x0, y0) to (x, y)
    ELSE
      draw(x0, y0, x01, y01, x11, y11, x12, y12);
      draw(x12, y12, x22, y22, x23, y23, x, y)
    END
  END draw;
```

Unfortunately, the curve's control points can be arranged in such a manner that the resulting curve gets folded onto itself, contains cusps (singularities in its curvature), or has points of inflection. To treat these cases correctly, the curve must first be normalized by splitting it at points of inflection and at interior local extrema in *x* and *y*. The corresponding values of the curve parameter *t* can be computed by solving a pair of linear and quadratic equations in *x* and *y*. The curve is subdivided if these values are within the visible range of the curve.

3.3.4 Path Queries

Apart from constructors and iterators, a third class of path procedures helps clients determine simple path properties, for example the length of a path or whether a path intersects a rectangle.

```
PROCEDURE Empty(path: Path): BOOLEAN;
```

When a path is cleared, **Empty** is true until the first path element is appended.

```
PROCEDURE InPath(llx, lly, urx, ury: REAL; path: Path;
  evenOdd: BOOLEAN): BOOLEAN;
PROCEDURE OnPath(llx, lly, urx, ury: REAL; path: Path): BOOLEAN;
PROCEDURE GetBox(path: Path; VAR llx, lly, urx, ury: REAL);
```

InPath returns whether a rectangle is completely inside a path, whereas **OnPath** returns if at least one path element passes through a given rectangle. To find out if a path is completely contained within a rectangle, its bounding box can be retrieved with **GetBox**.

```
PROCEDURE LineLength(x0, y0, x1, y1: REAL): REAL;
PROCEDURE ArcLength(sx, sy, ex, ey, x0, y0, x1, y1, x2, y2, flatness: REAL): REAL;
PROCEDURE BezierLength(x0, y0, x1, y1, x2, y2, x3, y3, flatness: REAL): REAL;
PROCEDURE Length (path: Path; flatness: REAL): REAL;
```

These procedures return the length of single path elements or of whole paths. To gain consistency between length calculation and enumeration, all elements are first flattened. The same flatness parameter should therefore be used for computing path lengths as for enumerating them.

3.3.5 Other Path Operations

The remaining operations of **GfxPaths** deal with a few other frequent needs.

```
PROCEDURE Reverse(src, dst: Path);
PROCEDURE Apply(path: Path; VAR mat: GfxMatrix.Matrix);
PROCEDURE Split(path: Path; offset: REAL; head, tail: Path);
```

Reverse stores a reverted copy of a path in another path, **Apply** applies a transformation matrix to all path elements, and **Split** divides a path into a head and a tail at a given distance from its start.

3.4 Imaging Subsystem

Gfx contexts treat raster images in an abstract manner. Although they allow their clients to paint images at arbitrary locations in user space, they offer no operations for allocating images, for reading and modifying pixel values, or for reading images from file. Moreover, Gfx does not address the needs of its concrete context extensions, which require additional functionality for transforming images from client-defined user coordinates to device coordinates. All these responsibilities are delegated to an independently reusable imaging subsystem.

The ability to scale and rotate raster images poses some particular demands. All but the simplest image transformation algorithms compute the color of a destination pixel as a combination of multiple source pixel values. It is therefore mandatory that color values can efficiently be extracted from source pixels and stored in destination pixels. Furthermore, to smoothly blend transformed images with a destination device, an intermediate representation that includes alpha values is required, for instance to represent

the partly opaque pixels along the boundary of a rotated image. By permitting pixels to include alpha values, our image model can also accommodate image masks, i. e. raster images with one bit of binary alpha information per pixel.

Raster image support in Gfx goes beyond the built-in functionality of the Oberon System, which only handles one-bit image masks ("patterns") and eight-bit raster images ("pictures"). Pictures use an indirect color model, which means that each pixel stores an index into an associated color lookup table ("palette"). Unfortunately, *inverse color lookup*, i. e. to find an appropriate index into a table of arbitrary colors for a given color value, is an operation whose complexity cannot be neglected. Thus, the lack of direct color and alpha information make Oberon pictures unsuitable for fulfilling Gfx's requirements. Besides, we felt that eight-bit images with only 256 distinct colors per image did not fit our goal of providing an advanced graphics library, considering that current display hardware supports frame buffers with depths of up to 24 or 32 bits and that corresponding image files are widely available (e. g. on the Internet).

When dealing with raster images, the usual trade-off between abstraction and efficiency is even more pronounced than for other data structures. Transferring pixel data from one image to another or from an image to a frame buffer can only be implemented efficiently if the transferrer (for example a Gfx context that renders on the display) has direct access to pixel memory and knows the internal pixel format of source and destination. However, this contradicts the wish for abstract interfaces that treat all different combinations of source and destination pixel formats alike.

Our solution comprises a general module **Images** for representing, loading, storing, and modifying raster images in a potentially unlimited number of formats, along with a module **GfxImages** for applying affine transformations to these images. **Images** offers clients abstract operations as well as detailed information about internal memory layout. We discuss the internal representation of image pixels in Section 3.4.1, how images arrange pixels in memory in Section 3.4.2, image compositing in Section 3.4.3, how the **Images** module can be extended by new file import and export routines in Section 3.4.4, and image transformations in Section 3.4.5.

3.4.1 Pixel Formats

According to the model that module **Images** provides, pixels in a raster image contain a combination of direct color values, alpha values, and indirect color values. How these values are stored within pixels is determined by a *pixel format* record.

```
CONST
  color = 0; alpha = 1; index = 2;   (* components *)
  b = 0; g = 1; r = 2; a = 3;  (* component index within Pixel type *)

TYPE
  Format = RECORD
    bpp: SHORTINT;   (* bits per pixel: 1, 2, 4, 8, 16, 24, or 32 *)
    components: SET;   (* kind of information stored in pixels *)
    pack, unpack: PackProc;   (* conversion to and from pixels *)
    ...
  END;

  PackProc = PROCEDURE(VAR fmt: Format; adr, bit: LONGINT; VAR pix: Pixel);
  Pixel = ARRAY 4 OF CHAR;
```

Instead of using symbolic constants for all pixel formats that one can possibly imagine or generic bit fields that describe offsets and lengths of individual components within pixels, format records contain a pair of procedures to convert pixel data to general **Pixel** values and vice versa. General pixels contain red, green, blue, and alpha values in the range from 0 to 255. The advantage of this scheme is its simplicity and extensibility, since new pixel formats are easily added by choosing a valid number of bits per pixel and implementing a pair of matching pack and unpack procedures. Its disadvantage is that it hides all information about the internal structure of pixels and thus makes optimized transfers impossible.

To counter this disadvantage, module **Images** defines a few common pixel formats and marks them with symbolic codes.

```
CONST custom = 0;   (* code for generic pixel format *)

TYPE
  Format = RECORD
    ...
    code: SHORTINT;
    ...
  END;
```

The **code** field marks a format as having an exactly defined internal format. Clients can thus bypass the **pack** and **unpack** procedures if they know how to handle the corresponding pixel format. Most built-in formats are directly available to clients as exported global variables. The different kinds of built-in pixel formats are listed in the following paragraphs.

Direct Color Formats. Pixel formats which directly store color values within pixels are called direct color formats. They usually occupy two, three, or four bytes per pixel and assign each color component a fixed number of bits at a fixed bit offset within the memory that the pixel occupies. Formats that occupy 16 bits per pixel are often called *hi-color* formats; those that occupy 24 or 32 bits per pixel are often called *true-color* formats.

```
CONST bgr555 = 5; bgr565 = 6; bgr466 = 7; bgr888 = 8; bgra8888 = 9;

VAR
  BGR555, BGR565, BGR466,    (* hi-color formats *)
  BGR888, BGRA8888: Format;   (* true-color formats *)

PROCEDURE SetRGB(VAR pix: Pixel; red, green, blue: INTEGER);
PROCEDURE SetRGBA(VAR pix: Pixel; red, green, blue, alpha: INTEGER);
PROCEDURE GetRGBA(pix: Pixel; VAR red, green, blue, alpha: INTEGER);
```

For example, the **BGR565** format occupies two bytes per pixel and stores five bits of blue, six bits of green, and five bits of red in each pixel. Similarly, the **BGRA8888** format occupies four bytes per pixel and stores eight bits of blue, green, red, and alpha in each pixel. It is therefore especially suitable for storing partly transparent images that are later blended into other images or frame buffers.

The internal representation of these pixel formats is compatible with that of common display hardware. This enables efficient block transfer between image and frame buffer memory.

Alpha values within pixels are *pre-multiplied*, which means that color values are stored scaled by $\alpha/255$. This simplifies compositing (see Section 3.4.3) but sacrifices numerical resolution. **SetRGB** and **SetRGBA** convert color and alpha values to pixels with pre-multiplied components; **GetRGBA** extracts the original values again from pre-multiplied pixel values.

Indirect Color Formats. Indirect color formats store an index into a color lookup table in each pixel. They occupy up to eight bits per pixel and can

thus contain up to 256 distinct colors.

```
CONST d8 = 3; p8 = 4;

TYPE
  Palette = POINTER TO RECORD
    col: ARRAY 256 OF Pixel;   (* color table *)
    used: INTEGER;   (* number of valid entries in color table *)
  END;

  Format = RECORD
    ...
    pal: Palette;
    ...
  END;

VAR D8: Format;

PROCEDURE InitPaletteFormat(VAR fmt: Format; pal: Palette);

PROCEDURE PaletteIndex(pal: Palette; red, green, blue: INTEGER): INTEGER;
PROCEDURE InitPalette(pal: Palette; used, bits: INTEGER);
PROCEDURE ComputePalette(img: Image; pal: Palette;
  reserved, max, bits: INTEGER);
```

The simplest indirect color format is called **D8**. Its pixels contain eight-bit indices into the internal color table of the underlying Oberon display module. The associated palette is implicitly defined by the one that is used in module **Display**.

For formats with explicit palettes, there can be no common format variable because different palettes must be stored in different format records. However, a format with code **p8** (signaling the presence of a palette and eight-bit pixels) for storing up to 256 different colors can be created by supplying an initialized palette structure to **InitPaletteFormat**.

Palette structures must be initialized with **InitPalette** to associate them with an internal data structure that speeds up inverse color lookup. This data structure consists of a three-dimensional array that maps RGB-triples to indices according to [81]. This data structure is necessary for **PaletteIndex**, which finds a matching palette index for a given color.

The process of reducing the number of distinct colors in an image is known as *color quantization*. As a special case, **ComputePalette** uses an octree-based algorithm [29] to choose a matching palette for any given

image. The computed palette will represent the color distribution in the source image reasonably well, favoring often used colors and merging similar colors into one. Another well-known solution to the same problem is Heckbert's median cut algorithm [40].

Pure Alpha Formats. Pure alpha formats store alpha values, but no color. They are therefore exclusively used for image masks.

```
CONST a1 = 1; a8 = 2;

VAR A1, A8: Format;
```

With **A1**, every pixel uses exactly one bit and is either fully transparent or fully opaque. **A8** offers 256 levels of transparency per pixel and is often used for storing scaled or rotated glyph bitmaps.

Special Formats. Two pixel formats serve special purposes.

```
VAR PixelFormat, DisplayFormat: Format;
```

The **PixelFormat** variable contains the format description of general **Pixel** values. In the current implementation, it is equal to **BGRA8888**. **DisplayFormat** is the pixel format of the system frame buffer. It is primarily used for off-screen images whose contents are to be copied to the visible display area. If **DisplayFormat** is used for such images, their contents can be copied to the frame buffer with a simple memory block transfer, without further conversions.

3.4.2 Images

Module **Images** defines raster images as follows.

```
TYPE
  Format = RECORD
    ...
    align: SHORTINT;
    ...
  END;

  Image = POINTER TO RECORD (Objects.ObjDesc)
    width, height: INTEGER;   (* image dimensions *)
    fmt: Format;   (* pixel format *)
```

```
    bpr: LONGINT;   (* number of bytes per row (may be negative) *)
    adr: LONGINT;   (* address of lower left pixel *)
    mem: POINTER TO ARRAY OF CHAR;   (* pixel storage *)
  END;

PROCEDURE Create(img: Image; w, h: INTEGER; VAR fmt: Format);
PROCEDURE InitBuf(img: Image; w, h: INTEGER; VAR fmt: Format;
  bpr, offset: LONGINT; VAR buf: ARRAY OF CHAR);
PROCEDURE InitRect(img, base: Image; x, y, w, h: INTEGER);
```

An image is basically a combination of a pixel format and a block of memory where pixels are stored. The pixel width of an image is multiplied by the number of bits that its format associates with each pixel and rounded up to the next byte boundary, which is in turn rounded up to the next multiple of the format's **align** field. This value is the number of bytes per image row and is stored in the **bpr** field. The product of **bpr** and the image height in pixels determines the actual amount of memory that is needed to store the image.

The **adr** field holds the address of the pixel in the lower left corner of the image. If images are initialized with **Create**, a heap block of suitable size is allocated in **mem** and the address of its first element is stored in **adr**. To export an address in a type-safe system is rather uncommon. We justify its inclusion in the image interface by observing that some applications benefit from being able to use memory that resides in locations other than the heap (e. g. on the stack) for storing images. Therefore, an image can be initialized on an arbitrary array of bytes with **InitBuf**. Furthermore, a client can reuse a rectangular part of an existing base image in a new image with **InitRect**, in which case the address of the sub-image is correspondingly offset from that of the base image. As a third benefit, arbitrary application data that resides somewhere in memory can be interpreted as image data if a corresponding pixel format can be defined, i. e. if the data can be addressed with a base address (**adr**) and a row stride (**bpr**). If pixel rows are stored upside down, the **bpr** field is negative.

We would like to point out that exported addresses are not unsafe per se since clients still need to import the inherently unsafe **SYSTEM** module to read and write memory at an address. (A module that imports **SYSTEM** clearly signals that it may violate type-safety.) Rather, exported addresses are unsafe because any client can change their value, which to forbid the standard Oberon language is not expressive enough. An elegant way to solve

this problem would be a read-only export mark as provided by the Oberon-2 language [58].

3.4.3 Compositing

The presence of alpha values in raster images entails the need for a method to combine source and destination images. The **Blend** procedure combines two pixels according to one of the twelve different compositing operations that are defined in [27].

```
CONST
   clear = 0; srcCopy = 1; dstCopy = 2; srcOverDst = 3; dstOverSrc = 4;
   srcInDst = 5; dstInSrc = 6; srcWithoutDst = 7; dstWithoutSrc = 8;
   srcAtopDst = 9; dstAtopSrc = 10; srcXorDst = 11;

PROCEDURE Blend(op: INTEGER; VAR src, dst: Pixel);
```

The following table lists all compositing operations and defines their effect with two factors F_S and F_D. When **Blend** is called with a source pixel S and a destination pixel D, it places the result $F_S \cdot S + F_D \cdot D$ in the destination pixel.

Operation	F_S	F_D	Operation	F_S	F_D
srcCopy	1	0	dstCopy	0	1
srcOverDst	1	$1 - \alpha_S$	dstOverSrc	$1 - \alpha_D$	1
srcInDst	α_D	0	dstInSrc	0	α_S
srcWithoutDst	$1 - \alpha_D$	0	dstWithoutSrc	0	$1 - \alpha_S$
srcAtopDst	α_D	$1 - \alpha_S$	dstAtopSrc	$1 - \alpha_D$	α_S
clear	0	0	srcXorDst	$1 - \alpha_D$	$1 - \alpha_S$

Unfortunately, to compose images by first unpacking them pixel by pixel, calling **Blend** for each pair of source and destination pixels, and packing the result again in destination pixel format is too inefficient for frequently used operations such as **srcCopy** (replacing destination by source) and **srcOverDst** (replacing destination according to source alpha). The reason is that the room for optimization that many combinations of pixel formats offer is not taken into account.

The **Images** module solves this problem with what it calls *transfer modes*.

```
TYPE
  TransferProc =
    PROCEDURE(VAR mode: Mode; sadr, sbit, dadr, dbit, len: LONGINT);
```

```
  Mode = RECORD
    src, dst: Format;   (* source and destination format *)
    op: INTEGER;   (* compositing operation *)
    col: Pixel;   (* substitute color for pure alpha sources *)
    transfer: TransferProc;
   END;

VAR SrcCopy, SrcOverDst: Mode;

PROCEDURE InitMode(VAR mode: Mode; op: INTEGER);
PROCEDURE InitModeColor(VAR mode: Mode; op, red, green, blue: INTEGER);
PROCEDURE Bind(VAR mode: Mode; VAR src, dst: Format);
```

Transfer modes are initialized with one of the available compositing operations. For pure alpha sources, transfer modes must also be assigned a color value which determines the color that will be associated with source alpha values. Whenever **Bind** is called, it chooses an appropriate *transfer procedure*, based on compositing operation, source format, and destination format. It stores this procedure in the mode's **transfer** field. As of this writing, module **Images** implements almost 70 different procedures for optimized transfers among its built-in pixel formats, especially for **clear**, **srcCopy**, and **srcOverDst** operations. Modes that are initialized with **srcCopy** and **srcOverDst** are even exported as global variables for convenience.

The following primitive procedures for reading and writing pixel values and for copying rectangular blocks from one image to another utilize transfer modes. These procedures bind the supplied transfer mode to the specific combination of source and destination formats with **Bind** and then use the correspondingly optimized transfer procedure to move pixels from source to destination.

```
PROCEDURE Get(img: Image; x, y: INTEGER; VAR pix: Pixel; VAR mode: Mode);
PROCEDURE Put(img: Image; x, y: INTEGER; pix: Pixel; VAR mode: Mode);

PROCEDURE Fill(img: Image; llx, lly, urx, ury: INTEGER;
  pix: Pixel; VAR mode: Mode);
PROCEDURE Clear(img: Image);

PROCEDURE GetPixels(img: Image; x, y, w: INTEGER; VAR fmt: Format;
  VAR buf: ARRAY OF CHAR; VAR mode: Mode);
PROCEDURE PutPixels(img: Image; x, y, w: INTEGER; VAR fmt: Format;
  VAR buf: ARRAY OF CHAR; VAR mode: Mode);
```

```
PROCEDURE Copy(src, dst: Image; llx, lly, urx, ury, dx, dy: INTEGER;
  VAR mode: Mode);
PROCEDURE FillPattern(pat, dst: Image; llx, lly, urx, ury, px, py: INTEGER;
  VAR mode: Mode);

PROCEDURE Dither(src, dst: Image);
```

It should now be clear why clients are advised to use built-in formats whenever possible. Although all image operations correctly deal with custom formats, optimized transfers are only possible for built-in formats. For custom formats, **Bind** will always fall back to a generic transfer procedure that is based on pack and unpack procedures and cannot read and write pixel memory directly. The additional overhead of invoking pack and unpack procedures and of converting pixel values to and from generic **Pixel** variables seriously lessens overall performance. A small evaluation shows that custom formats operate from about ten to sixty times slower than equivalent built-in formats (see Appendix C.2).

3.4.4 File Formats

Most applications use only raster images that they import from files. Module **Images** offers its clients procedures for loading and storing images in arbitrary file formats using the plug-in mechanism that was described in Section 3.1.1.

```
VAR
  LoadProc, StoreProc:
    PROCEDURE(img: Image; VAR fname: ARRAY OF CHAR; VAR done: BOOLEAN);

PROCEDURE Load(img: Image; name: ARRAY OF CHAR; VAR done: BOOLEAN);
PROCEDURE Store(img: Image; name: ARRAY OF CHAR; VAR done: BOOLEAN);
```

For loading and storing, **Images** searches a matching command procedure by interpreting the file name extension as a key into a table of commands. A matching command procedure, when executed, is expected to initialize the global **LoadProc** and **StoreProc** variables with references to procedures for loading and storing images in the corresponding format. The number of supported image file formats can thus be augmented without recompiling or even reloading the **Images** module itself. In its current implementation, the

Images package includes extensions for loading and storing Oberon pictures and for loading BMP, GIF, and JPEG files.

3.4.5 Transformations

Section 2.2.4 briefly touched the problem of applying affine transformations to raster images, which we discuss in more detail here. More thorough introductions to the field of image reconstruction, resampling, and image transformations can be found in [30] and [88].

Image Reconstruction. A raster image can be regarded as a representation of a two-dimensional analog signal of which only the values at regularly spaced samples, located at pixel centers, are known. When a pixel center in the destination image of an image transformation is mapped back to the source image (using the inverse of the transformation), it will rarely coincide with the center of a source pixel. Hence, the correct signal amplitude at that location in the source image must be *reconstructed* from nearby samples. The reconstructed signal is then *resampled* at the same location to find the correct value for the destination pixel.

The visual quality of a transformed image primarily depends on the quality of the source image, which must be sampled at a frequency that is high enough not to miss high-frequent details, and the quality of the reconstruction. Reconstruction can be expressed as a convolution of the sampled signal with a reconstruction filter. According to the sampling theorem, a signal whose spectrum is limited to a maximal frequency can be perfectly reconstructed from its samples by convoluting it with a sinc function, which is the inverse Fourier transform of a perfect low-pass filter (a box function) in the frequency domain. However, the sinc function, which is defined as

$$\mathrm{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$$

has infinite extent in the signal domain and therefore cannot be used in practice. The filter functions that are applied instead are always prone to introduce reconstruction artifacts because they are not limited in the frequency domain and thus introduce other frequencies of the sampled signal in the reconstructed signal.

Filter Interface. Higher quality reconstruction filters are computationally more expensive than those of lower quality. Since Gfx cannot a priori know the needs of its clients, it lets them decide on the balance between performance and aesthetics by relying on an extensible filter structure:

```
TYPE
  ShiftProc = PROCEDURE(VAR filter: Images.Mode; src, dst: Image;
                sadr, sbit, dadr, dbit, len: LONGINT; t: REAL);
  ScaleProc = PROCEDURE(VAR filter: Images.Mode; src, dst: Image;
                sadr, sbit, dadr, dbit, len: LONGINT; xy, dxy: REAL);

  Filter = RECORD (Images.Mode)
    hshift, vshift: ShiftProc;
    hscale, vscale: ScaleProc;
  END;

VAR NoFilter, LinearFilter: Filter;

PROCEDURE InitFilter(VAR filter: Filter; op: SHORTINT; hsh, vsh: ShiftProc;
  hsc, vsc: ScaleProc);
```

A **Filter** is derived from **Images.Mode**, inheriting the transfer mode's compositing operation and transfer procedure. It augments the transfer mode by procedures for shifting and scaling rows and columns of pixels. This restricts possible filter functions to separable filters, which are combinations of independent one-dimensional filters in the horizontal and vertical axis.

GfxImages exports two built-in filters, **NoFilter** and **LinearFilter**. **NoFilter** implements a simple box filter, which returns the sampled value of the nearest pixel center. It is called **NoFilter** because it achieves the same result as a naive algorithm that does not take filtering into account at all. **LinearFilter** implements a tent filter function, which reconstructs the analog signal as a linear combination of the two nearest sample values. Although by no means an advanced filter, **LinearFilter** is already noticeably slower than **NoFilter**.

Transformation Procedures. With these filters, transformation procedures can apply simple and compound transformations to a source image and store the result in a destination image.

```
PROCEDURE Translate(src, dst: Image; tx, ty: REAL; VAR filter: Filter);
PROCEDURE Scale(src, dst: Image; sx, sy, tx, ty: REAL; VAR filter: Filter);
PROCEDURE Rotate(src, dst: Image; sin, cos, tx, ty: REAL; VAR filter: Filter);
```

```
PROCEDURE ShearRows(src, dst: Image; sx, tx: REAL; VAR filter: Filter);
PROCEDURE ShearCols(src, dst: Image; sy, ty: REAL; VAR filter: Filter);
PROCEDURE Transform(src, dst: Image; m: GfxMatrix.Matrix; VAR filter: Filter);
```

Because filters only implement shifting and scaling for rows and columns, all transformations have to be decomposed into a series of shifting and scaling transformations. An affine 3×3 matrix is split into a rotate-scale-shear and a translation part as

$$\begin{pmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{pmatrix} = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ e & f & 1 \end{pmatrix}$$

The first matrix on the right side of the equation can be further decomposed according to the following identity

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} a & 0 \\ 0 & \frac{ad-bc}{a} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ \frac{ca}{ad-bc} & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & \frac{b}{a} \\ 0 & 1 \end{pmatrix}$$

In this decomposition, the first matrix is a scaling transformation, whereas the latter two are shearing transformations along the *x* and *y* axes, which are implemented as a series of shifts with variable displacements. Since the translation part can be incorporated into the last call of the filter's **shift** or **scale** procedures, any transformation can be performed in at most three steps. However, an additional pre-processing step, in which the whole image is possibly rotated by multiples of 90 degrees or mirrored along a coordinate axis, is necessary to avoid the bottleneck problem that is discussed in [88]. Otherwise, information could be lost in intermediate stages because many pixels could be combined into one. This pre-processing step at the same time keeps the denominators of the fractions in the above decomposition from assuming values in the vicinity of zero and thus avoids problems with numeric overflow.

While clients are responsible for supplying a destination image that provides enough space for the transformed image, filter procedures are responsible for not accessing pixels outside the boundaries of their source images. Depending on the extent of the implemented filter, filter procedures must replace non-existent source pixels beyond the boundary of the source image with reasonable estimates, for example by duplicating boundary pixels or by adapting the corresponding filter functions.

3.5 Font Subsystem

While the Oberon system is able to efficiently display the glyphs of its bitmap fonts, it only provides a fixed set of point sizes and resolutions. Gfx, however, supports a user coordinate system that can be arbitrarily scaled and rotated and thus has to provide glyph images for non-existent font sizes and orientations as well.

Oberon bitmap font files have been scan-converted from outline descriptions and – since the outline format does not contain any hinting information – manually tuned to improve visual appearance and legibility. The outlines of these fonts, however, are not available to Oberon applications unless they decode the corresponding font files themselves. One goal of the font subsystem is therefore to make these outlines available as path structures.

To craft a well-designed font is not only a work of love and labor but also an art [18]. The creators of the Oberon System were lucky enough to gain the support of professional type designer Hans Ed. Meier, who used their tools to create a digital adaptation of his well-known Syntax family of typefaces, which have since been used in all variants of the system. To provide its clients with a broader choice of quality fonts, the Gfx font subsystem is extensible and can load fonts in other formats, too.

3.5.1 Font Interface

The **GfxFonts** module lets its client demand image masks, outlines, and metrics for each glyph.

```
PROCEDURE GetWidth(font: Font; ch: CHAR; VAR dx, dy: REAL);
PROCEDURE GetMap(font: Font; ch: CHAR; VAR x, y, dx, dy: REAL;
  VAR map: Images.Image);
PROCEDURE GetOutline(font: Font; ch: CHAR; x, y: REAL; path: GfxPaths.Path);

PROCEDURE GetStringWidth(font: Font; str: ARRAY OF CHAR; VAR dx, dy: REAL);
```

For each character, **GetWidth** returns its advance width, **GetMap** additionally returns a matching image mask, and **GetOutline** stores the character's contour in the supplied path. **GetStringWidth** computes the advance width of an entire string. Advance width vectors contain *dx* and *dy* components

because fonts can be rotated, requiring that text is set in other directions than from left to right.

The above procedures for getting width, image, and outline of a glyph are again wrapper procedures which invoke corresponding entries in the font method table.

```
TYPE
  Font = POINTER TO RECORD
    class: Methods;
    ...
  END;

  Methods = POINTER TO RECORD
    ...
    getwidth: PROCEDURE(font: Font; ch: CHAR; VAR dx, dy: REAL);
    getmap: PROCEDURE(font: Font; ch: CHAR; VAR x, y, dx, dy: REAL;
                      VAR map: Images.Image);
    getoutline: PROCEDURE(font: Font; ch: CHAR; x, y: REAL; path: GfxPaths.Path);
   END;
```

Depending on the class of a font, the corresponding **getwidth** and **getmap** procedures load glyph metrics and bitmaps from a file or create them dynamically by converting font outlines or by transforming existing glyph metrics and bitmaps. To avoid reconstructing the same bitmap over and over again, the resulting image masks are stored in an internal glyph cache and returned directly from the cache the next time they are requested.

3.5.2 Finding Fonts

A font is defined by its name, its point size, and a matrix that incorporates the device scaling factor and other transformations.

```
TYPE
  Font = POINTER TO RECORD
    class: Methods;
    name: FontName;
    ptsize: INTEGER;
    mat: GfxMatrix.Matrix;
    xmin, ymin, xmax, ymax: INTEGER;   (* union of glyph bounding boxes *)
    rfont: Fonts.Font;   (* corresponding system font, if appropriate *)
    niceMaps: BOOLEAN;   (* true if bitmaps are tuned or grid-fitted *)
  END;
```

```
    Methods = POINTER TO RECORD
      derive: PROCEDURE(font: Font; size: INTEGER; VAR mat: GfxMatrix.Matrix): Font;
      ...
    END;

  VAR
    Default: Font;
    OpenProc: PROCEDURE (VAR family, style: ARRAY OF CHAR;
                         size: INTEGER; VAR mat: GfxMatrix.Matrix): Font;

  PROCEDURE Open (name: ARRAY OF CHAR; ptsize: INTEGER;
                  mat: GfxMatrix.Matrix): Font;
  PROCEDURE OpenSize (name: ARRAY OF CHAR; ptsize: INTEGER): Font;
```

Upon receiving a request to open a font, the following steps are tried in order until a matching font can be returned or all of them have failed.

1. Search a matching font in the internal font cache of module **GfxFonts**.
2. If a font from the same family, but with different point size or instance matrix is found in the cache, call the **derive** method of that instance. For outline-based fonts, **derive** can return a font instance which shares outline data with other fonts of the same family.
3. Try to open a matching Oberon bitmap or outline font file.
4. Search a registered command, using the name of the font family in question as a key into a dictionary (see Section 3.1.1). The command procedure, when called, is expected to initialize the global **OpenProc** variable. That procedure is then called with all necessary parameters and may return a reference to an extension of type **Font**.

Due to the dynamic loading scheme, new font file formats can be added whenever necessary without any change in **GfxFonts**. Special wildcard keys that match all font families are supported as well. Thus, fonts can even be loaded when they have not been registered. However, this results in all registered extensions being loaded at once instead of only the one that would actually be required. At the moment, TrueType fonts [7] are fully supported, including grid-fitted raster images and outlines. Metafont [47] fonts are supported as well, but only in the form of pre-rendered bitmaps.

When a font is requested for which Oberon bitmap or outline font files exist, the following heuristic is used. If a bitmap with matching point size and resolution is available, **GfxFonts** opens it as a standard Oberon

font with the system call **Fonts.This** and keeps a reference of this system font in **rfont**. Because clients can check and access this reference, they can take advantage of the efficient font rendering code that Oberon's low-level **Fonts** and **Display** modules provide. If no exactly matching bitmap font is available, but one which is similar in scale is, **GfxFonts** loads the latter and transforms its bitmaps when an image mask is requested. If no bitmap font at all can be found, but a corresponding outline font is available, image masks are generated by scan-converting these outlines.

The **niceMaps** field is a hint to clients of **GfxFonts** that indicates whether the image masks that **GfxFonts** provides will generally look better than filled outlines, as is for example the case with fonts whose bitmaps were manually tuned or whose outlines contain hints. For instance, the Gfx context extension that module **GfxPS** provides uses this information to decide on whether to include fonts in the generated Postscript file as arbitrarily scalable Type-1 (outline) fonts or as Type-3 fonts (with embedded bitmaps) [3].

3.6 Region Subsystem

The module **GfxRegions** manages *regions*, i. e. coherent areas in two dimensions. Gfx uses regions in its font subsystem (see Section 3.5) for scan-converting glyph outlines and in its raster device contexts (see Section 3.7) for storing the current clip area and for scan-converting closed paths to fill or clip them. Figure 3.6 shows a simple example of a region. This region could either have been constructed as a boolean combination of three rectangles, for example to compute the initial clip area for a partially obscured window in a windowed display, or as the interior of the path along its outline, which is a concave polygon with ten sides. To meet the requirements of Gfx, regions must be structured in a way that supports either approach.

Region representation and type are discussed in Section 3.6.1; boolean operations with region arguments are the topic of Section 3.6.2, and region construction from contour points follows in Section 3.6.3.

3.6.1 Region Representation

A proven method for representing regions is to decompose them into horizontal slices [27]. By intersecting a region with a set of equidistant scanlines,

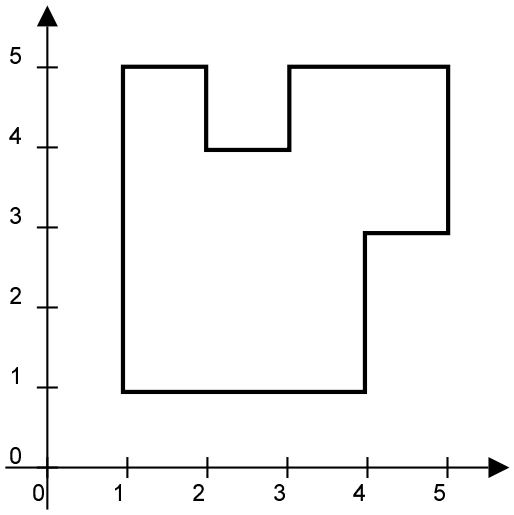

Figure 3.6: Example of a simple region

we get a set of horizontal intervals, as displayed in Figure 3.7. These scanlines typically correspond to rows of pixels, which is why regions are usually stored in device coordinates.

Masks. We can see in Figure 3.7 that the two bottommost scanlines of the displayed region are identical. Instead of storing such redundant scanlines in the region structure, we can merge them into one. This requires that we give each scanline a height. Oberon System 3 implements such a data structure, which it calls *mask*, in its Gadgets component framework [51]. Masks allow Gadgets to efficiently store the visible areas of overlapping rectangular frames (windows). However, since masks place each scanline interval in a separate block of heap memory, complex regions with curved boundaries are spread over many small memory blocks. This strains memory management and increases the likelihood of cache misses during region traversals. Another reason why masks do not suit the needs of Gfx is that regions can only be constructed with boolean operations, but not from region outlines.

Point Vectors. In the Gfx region representation, scanlines and intervals are not stored explicitly. Instead, a region consists of a sequence of points. Each interval contributes two points to the structure, one at its left end and one

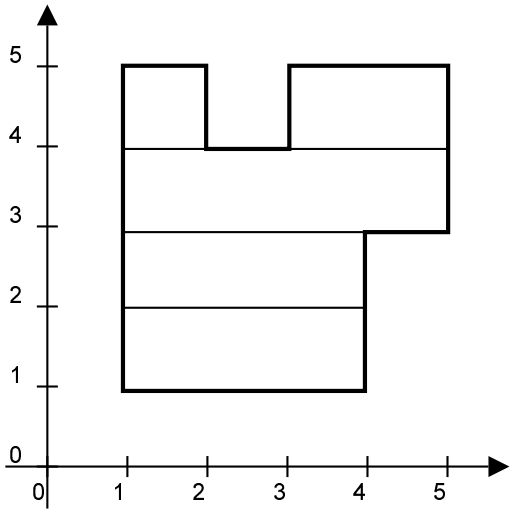

Figure 3.7: Region decomposed into horizontal intervals

at its right end. The region that is displayed in Figure 3.7 thus consists of the following points, ordered from bottom left to top right:

(1, 4)	(2, 4)	(3, 4)		(5, 4)
(1, 3)				(5, 3)
(1, 2)			(4, 2)	
(1, 1)			(4, 1)	

To construct a region from its outline, a client traverses the corresponding path and appends all intersections of the path with any scanline to the region. The resulting sequence lists points in the order they appear on the path, which is not suitable for traversing the interior of the path as a region. However, these points can be brought into a suitable order by sorting the entire point sequence according to point coordinates. A point P comes before a point Q in the sorted order if $P_y < Q_y$ or $P_y = Q_y \wedge P_x < Q_x$.

When region outlines intersect themselves, intervals on a scanline may overlap each other. Such situations can be resolved according to the even-odd rule or to the non-zero-winding rule if we associate a direction (up or down) with each point. The points on a scanline then correspond exactly to the intersections of a virtual ray with the region outline that we used to define these rules on page 23 in Section 2.2.2.

Fillers. As with masks, redundant scanlines in a point vector can be merged. When points (1, 2) and (4, 2) are removed from the above list, the intervals in the scanline just below are automatically assumed to apply to all scanlines above up to the next listed scanline (which in our example starts with point (1, 3)). However, when regions consist of several disconnected parts, as illustrated in Figure 3.8, we can no longer distinguish between scanlines that are empty and scanlines that have been removed because they were redundant. To overcome this problem, we introduce *fillers*, which are special

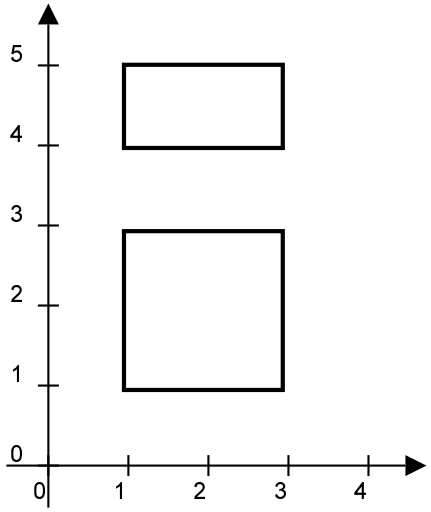

Figure 3.8: Example of disconnected region

intervals that start and end at $x = \infty$. An embedded filler interval implicitly limits the height of the scanline below it. Besides, by adding special fillers at the bottom and at the top of the region, we ensure that each scanline has a valid scanline below and above itself. This simplifies several algorithms that operate on regions because boundary problems are eliminated. The final set of points for the region from Figure 3.8 is thus as follows.

		$(\infty, \infty, +)$	$(\infty,\infty,-)$
$(1, 4, +)$	$(3, 4, -)$		
		$(\infty, 3, +)$	$(\infty, 3, -)$
$(1, 1, +)$	$(3, 1, -)$		
		$(\infty, -\infty, +)$	$(\infty, -\infty, -)$

The third component of each tuple is the direction of the corresponding scanline intersection. It defines whether an interval starts $(+)$ or ends $(-)$

at that point.

Interface. The region data structure is defined as follows:

```
CONST
  Winding = 0; EvenOdd = 1; (* region modes *)

TYPE
  RegionData = POINTER TO ARRAY OF LONGINT;  (private)

  Region = POINTER TO RECORD
    llx, lly, urx, ury: INTEGER;  (* bounding box *)
    mode: INTEGER;   (* rule defining path interior *)

    private fields:
    valid: BOOLEAN;   (* set if points in data array are sorted and compacted *)
    data: RegionData;   (* encoded points defining scanline intervals *)
    points: LONGINT;    (* number of data points actually used *)
  END;
```

To speed up the sorting step that is required to convert point sequences to proper regions, point coordinates are stored in encoded form. Each element in a **RegionData** array contains coordinates and direction of one point. The encoding is such that integer comparisons of data elements lead to the desired sorting order.

When an encoded point is added to an already full block, the block's contents are copied to a larger block, which then replaces the old one.

As an optimization, **points** may be zero. This indicates that the region is rectangular and that the bounding box alone defines the entire region.

3.6.2 Region Algebra

Two regions can be combined with each other to form a new region. When two regions are *joined*, the result encloses all areas that were part of at least one region. When two regions are *intersected*, the result encloses all areas that were part of both regions. When one region is *subtracted* from another, the result encloses all areas that were part of the original region, but not of the one that was subtracted.

Generic Algorithm. More technically, a boolean operation must combine the point vectors from both regions. Depending on the operation, existing

points must be removed or new points must be added. An outline of the generic algorithm that is used for these operations looks as follows.

1. Validate both regions, i. e. sort and compact their point vectors if necessary.
2. Traverse both point vectors simultaneously and build a new sequence of points. Because the result will be stored in the first region, only those points that are not already present in that region are appended to the new sequence. To remove existing points, append new points at the same coordinates, but with opposite direction. This creates an empty interval at that point.
3. Merge old and new point sequences. A generic sorting step is not necessary because both sequences are already in the required order.
4. Compact the resulting point sequence by eliminating empty intervals (from removed points) and redundant scanlines.

Example. To illustrate the generic algorithm, we present how a rectangle with corners (2, 2) and (4, 4) is subtracted from a rectangle with corners (0, 0) and (4, 4) (see Figure 3.9). The initial point sequences of both rectangles look as follows:

Destination		Argument	
$(\infty, \infty, +)$	$(\infty, \infty, -)$	$(\infty, \infty, +)$	$(\infty, \infty, -)$
$(\infty, 4, +)$	$(\infty, 4, -)$	$(\infty, 4, +)$	$(\infty, 4, -)$
$(0, 0, +)$	$(4, 0, -)$	$(2, 2, +)$	$(4, 2, -)$
$(\infty, -\infty, +)$	$(\infty, -\infty, -)$	$(\infty, -\infty, +)$	$(\infty, -\infty, -)$

To subtract the argument from the destination region, its non-filler points are appended with inverted directions to a new point sequence. In addition, the original intervals of the destination region that are in effect on the corresponding scanlines must be duplicated.

Destination		New Points			
$(\infty, \infty, +)$	$(\infty, \infty, -)$				
$(\infty, 4, +)$	$(\infty, 4, -)$				
$(0, 0, +)$	$(4, 0, -)$	$(0, 2, +)$	$(2, 2, -)$	$(4, 2, +)$	$(4, 2, -)$
$(\infty, -\infty, +)$	$(\infty, -\infty, -)$				

When these two point sequences are merged and compacted, the final region contains the following points:

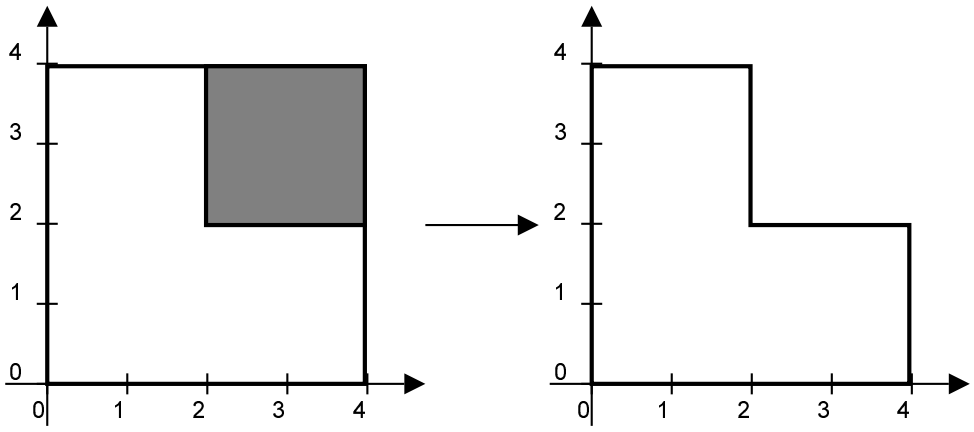

Figure 3.9: Subtract one region from another

$(\infty, \infty, +)$ $(\infty, \infty, -)$
$(\infty, 4, +)$ $(\infty, 4, -)$
$(0, 2, +)$ $(2, 2, -)$
$(0, 0, +)$ $(4, 0, -)$
$(\infty, -\infty, +)$ $(\infty, -\infty, -)$

3.6.3 Construction from Outlines

As mentioned in Section 3.6.1, clients convert closed paths to regions by intersecting them with scanlines and appending the resulting points to the point vector of a region. However, these intersections with scanlines include some special cases that must be properly handled.

1. Paths may contain horizontal lines. Since horizontal lines cannot be intersected with scanlines, they do not contribute points to the region. However, all other conditions must still be met.
2. The path may have corners that fall exactly on a scanline. If such a corner is a local vertical extremum, care must be taken to ensure that either two points or no points are appended to the region, corresponding to a proper interval or no interval at all. Similarly, if the path continues in the same direction as it arrived at the scanline, exactly one point must be appended, but not two.
3. Closed paths return to their starting point. Nevertheless, this starting point must only be entered once into the region.

To cope with these conditions, we slightly shift our focus. Instead of appending points to the region, we append (vertical) *transitions*. Thus, horizontal lines are ignored because they include no vertical transition. Points on scanlines either result in a up-down or down-up transition for local extrema or in two successive up or down transitions for non-extremal points. Last but not least, shared points of a closed path pose no problem because only the first and last transitions count.

AddPoint Procedure. The following procedure appends transitions to a region.

```
PROCEDURE AddPoint(reg: Region; x, y, dy: INTEGER);
BEGIN
  IF (dy # 0) & (y >= -∞) & (y <= ∞) THEN
    IF x < -∞ THEN x := -∞
    ELSIF x > ∞ THEN x := ∞
    END;

    (* convert rectangular region to point sequence *)
    MakeData(reg);

    (* update bounding box *)
    IncludePoint(reg.llx, reg.lly, reg.urx, reg.ury, x, y);

    Append(reg, x, y + (-dy) DIV 2, dy);   (* dy=-1 => y, dy=1 => y-1 *)
    reg.valid := FALSE
  END
END AddPoint;
```

When clients add a transition to a region, they supply the new end point in *x* and *y*. The current vertical direction is captured in *dy*. To ensure that each horizontal interval has proper lower and upper bounds, new points are always positioned on the scanline with the lower *y* coordinate. The **valid** flag is cleared to indicate that the data points of the region must be sorted before the next region operation can be performed.

Alternative Approaches. Instead of a single **AddPoint** procedure, we could have chosen to provide two procedures **Start** and **Step**. Vertical direction could then be calculated internally and would not have to be specified by clients. However, some algorithms for constructing regions benefit from the current approach because they can add points in any order, not just in

the sequence they appear during the traversal of a path. For example, the symmetry in a circle allows an algorithm for drawing circles to only compute the first octant and draw the remaining octants by mirroring computed points. With **AddPoint**, such an algorithm can add several points at once with each step.

A standard method for filling concave polygons places all edges of the polygon into an *edge table* and advances from scanline to scanline, from the bottom of the polygon to the top. Those edges that intersect the current scanline are called *active* and are therefore part of the *active edge table* (AET). For each scanline, its intersections with the edges in the AET are sorted according to *x* coordinate and used to determine the set of visible intervals on that scanline. Because the members of the AET and their relative horizontal order are likely to be the same on the next scanline as on the current scanline (a property that is called *edge coherence*), the algorithm can advance from scanline to scanline with little effort. The complete algorithm is for example explained in detail in [35] or [27].

Scanline algorithms that are based on edge tables can at the same time solve the hidden-surface problem in 3D (e. g. [38]). They only need to keep the edges of all active polygons (including depth information for each edge) in the AET to determine visible intervals of each polygon on the current scanline. For our purposes, however, such an algorithm would not be satisfactory. Even if we ignore that curved path segments have to be approximated with many small line segments, we still need to know the complete set of edges before we can build a sorted edge table. This would defeat our intention of rendering path segments without first recording the entire path in a path data structure. Edge tables might be a viable solution for scan-converting paths in a context that uses late rendering, but are not as attractive when early rendering is used.

3.7 Implementation on Raster Devices

Raster based pixel grids are the most important class of physical and logical output devices. This section gives an overview of a Gfx context extension that targets abstract raster devices. It is in turn the foundation for concrete contexts that render on the display and into raster images in memory.

```
CONST In = 0; Out = 1; InOut = 2;   (* clip states *)
```

```
TYPE
  Context = POINTER TO RECORD (Gfx.Context)
    clipReg: GfxRegions.Region;   (* region inside clip path *)
    clipState: SHORTINT;   (* current clip state *)
    dot: PROCEDURE(rc: Context; x, y: LONGINT);
    rect: PROCEDURE(rc: Context; lx, ly, rx, uy: LONGINT);
    setColPat: PROCEDURE(rc: Context; col: Gfx.Color; pat: Gfx.Pattern);
    col: Gfx.Color;   (* current color *)
    pat: Gfx.Pattern;   (* current pattern *)
    ...
  END
```

The module **GfxRaster** extends the basic context structure that module **Gfx** exports by a clip region, a current color and pattern, and procedures for setting current color and pattern, for painting single pixels, and for painting rectangles. The current color and pattern are equal to either the corresponding stroke or fill attributes, depending on the current render mode.

Raster contexts do not algorithmically clip path elements. They keep track of whether the current path element is completely visible or completely invisible in the **clipState** field, but leave it up to the **dot** and **rect** procedures to correctly handle partially visible elements.

3.7.1 Filling and Clipping

When the current rendering mode includes **Gfx.Fill** or **Gfx.Clip**, the visited path is converted to a region, which is then intersected with the current clip region or enumerated and filled with the current fill color and pattern using the **rect** procedure. When a path is started with **Begin**, a current path region is initialized. Dedicated algorithms based on the classic work by Bresenham [16, 17] are used to scan-convert lines, circles, and axis-aligned ellipses. These algorithms add the resulting points to the current path region. Arcs and Bézier curves are first approximated with lines using the enumeration procedures from **GfxPaths** (see Section 3.3.3) before they are added to the path region. When the region is finally combined with the current clip region or enumerated for filling, **GfxRegions** automatically compacts the set of added points and converts it to a valid region, as discussed in Section 3.6.

3.7.2 Stroking

Stroking a path is slightly more complicated. When the render mode combines stroking with either filling or clipping, the whole path is stored in a separate path structure and stroked in a second pass after all other operations have been executed. Otherwise the rules from Section 3.2.1 that define rendering order might be violated. Since a path can only be filled when its specification is complete, pixels that were affected by stroking operations during the traversal of the path may be overdrawn. Similarly, clipping would wrongfully not affect the stroke operation.

Hairlines. To prepare the context for stroking when a path is started, the current color and pattern are set to the corresponding stroke attributes, and the current line width is transformed to device space. If the line width is less than $1\frac{1}{2}$ pixels, path elements are stroked as hairlines. The same algorithms are used as for filling and clipping, but the calculated points are now painted with the context's **dot** procedure. Horizontal or vertical lines are stroked with a single invocation of **rect** instead of multiple **dot** calls.

Thick Lines. When starting to stroke with a line width greater than $1\frac{1}{2}$ pixels, path elements are first flattened. The resulting thick lines are converted to rectangles; the rectangles' sides are then scan-converted and appended to the current path region. To correctly paint line joins, output for a line is always delayed until the next line segment is appended because only then the geometry of the join is fully specified. Likewise, line caps can only be rendered when a subpath is complete because a client can close the current subpath anytime. Raster contexts therefore contain additional fields for storing the starting point and the initial direction of the subpath plus the start and end points of the previous line. Finally, the path region is filled with the current stroke color and pattern.

Line Joins. The geometry of a thick path's outline can become rather complex, especially when a short line segment meets another line at a sharp angle. Lines that are shorter than the line width itself are difficult to handle and may confuse **GfxRaster**. To minimize artifacts, it trims line joins by not drawing them beyond the midpoints of involved line segments. A more sophisticated algorithm would delay painting joins until more information

about the geometry of nearby line segments is available. Unfortunately, this would also require that an indefinite number of path elements are temporarily stored. Such an algorithm would therefore lose some of the simplicity that early rendering provides.

Dashes. Dashed subpaths are stroked by first flattening arcs and Bézier curves. The length of every traversed line segment is added to the current offset into the subpath, which in turn determines whether the subpath is currently visible or not. Depending on the current line width, single dashes are drawn as hairlines or as thick lines with line caps at their ends.

3.7.3 Text and Images

When text is rendered on a raster context, **GfxRaster** concatenates the current transformation matrix with the instance matrix of the current font. This new matrix is the instance matrix of the current font in the device coordinate system. If the current render mode only demands that the current path is filled, the corresponding font is requested to provide matching image masks, which are then output in the current fill color using the context's **image** method. For all other render modes, outlines are requested and appended to the current path. Concrete context extensions are encouraged to supersede this generic text rendering method and take advantage of the built-in Oberon mechanisms for rendering glyph bitmaps if applicable. The **GfxDisplay** module for generating Gfx output on the Oberon display does so, improving performance when rendering text for which a corresponding Oberon bitmap font is available.

Raster contexts do not render images at all, forcing their concrete extensions to do so. In theory, a raster image could be painted by painting a filled rectangle for every image pixel. In practice, however, this approach would be much too inefficient.

3.8 Summary

This chapter presented Gfx, an API for rendering vector and bitmap graphics on a variety of physical and logical output devices. Its rich functionality is comparable to that of Postscript; however, unlike Postscript and traditional

API (e. g. QuickDraw or GDI), Gfx fulfills the goals that were formulated at the end of Chapter 2. It has a modular structure and consists of several independently reusable subsystems which manage raster images, paths, regions, and fonts. Furthermore, although many of its interfaces are bottleneck interfaces that are not intended to be augmented by new functionality, support for new output devices, new image file formats, and new font formats can be added without having to recompile existing modules. Examples of existing extensions are

- a context type for creating Postscript and encapsulated Postscript files
- image loaders for importing images in BMP, GIF, and JPEG format
- font loaders for importing TrueType fonts and Metafont bitmaps

Last but not least, Gfx successfully implements a path model based on the idea of rendering path elements as early as possible. It can paint arbitrary paths in a number of render modes without first having to store visited paths explicitly, which is not the case for other graphics interfaces that support general path models.

CHAPTER 4

The Leonardo Shape Framework

Chapter 3 showed that the set of graphical object types and object attributes that Gfx handles is limited and cannot be extended. Individual objects are rendered in isolation, independent of the objects that were rendered earlier. Besides, Gfx has no memory of what objects it has rendered and cannot be requested to paint a set of objects a second time. Briefly, although all the tools for producing graphical output are present, there is no abstract model giving structure or persistence to a set of related graphical objects.

This lack of structure and persistence is not necessarily a flaw since many applications already implement a model of their own and may appreciate the direct access to abstract rendering devices that Gfx offers them with its immediate rendering model. However, other applications (especially interactive editors for creating and modifying graphical documents) would benefit from an additional software layer if it provided them with a more abstract view of a graphical scene. This additional layer should allow its clients to organize their objects in an appropriate data structure. Furthermore, it should allow them to store objects in a persistent store and to retrieve stored objects. To be useful to as many applications as possible, this structural layer should be organized as a *framework* [23, 45]. A framework primarily defines abstract objects and the interactions between them. Clients of the framework then extend these abstract objects with domain specific objects to customize the framework to their needs.

In this chapter, we present such a graphical object framework. It models graphical objects, called *shapes*, in a hierarchical, extensible, and persistent data structure. It includes protocols for constructing, rendering, and manipulating scenes (called *figures*) that are built from such objects. Several

of these protocols (e. g. for locating, selecting, or moving shapes) concern tasks that serve interactive editor applications. Thus, the framework's figure objects can directly be used as document models in the Leonardo graphics editor, which is described in Chapter 5. Nevertheless, at least the core part, which defines graphical objects, their appearance, and their structure, could also be reused within other environments. The figures of the shape framework correspond to scene graphs in 3D, as for example provided by PHIGS [64], OpenInventor [83], or Java 3D [75].

An annotated example of a figure is shown in Figure 4.1. Other examples include the figures in this thesis, all of which have either been constructed interactively with Leonardo (see Chapter 5) or descriptively with Vinci (see Chapter 6).

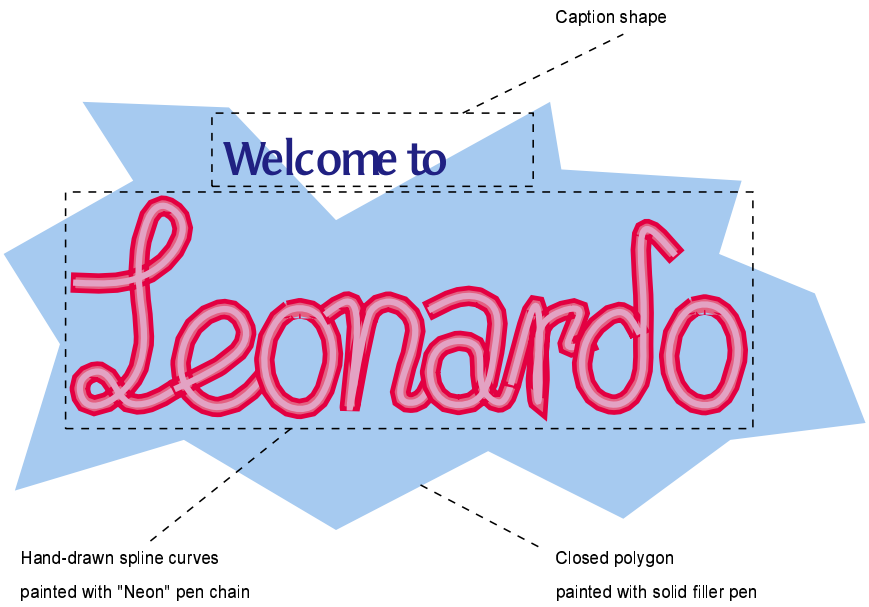

Figure 4.1: Annotated example of a figure

Whereas Gfx on occasion limits extensibility in favor of efficiency and simplicity, the shape framework focuses on extensibility and generalization.

To achieve the necessary flexibility, the shape framework exports several independent type hierarchies. From Oberon System 3's standard **Object** and **ObjMsg** types, whose properties are summarized in Section 4.1, two of these hierarchies are derived; the shape type hierarchy (see Section 4.2) and the shape message hierarchy (see Section 4.4). Their versatility is illustrated by an explanation of how the framework implements geometrical constraints in Section 4.5. A third type hierarchy, which allows clients to customize the visual appearance of existing shapes by delegating render requests to so-called *pen objects*, is discussed in Section 4.3.

4.1 Foundations: Oberon Objects and Libraries

The main difference between Oberon System 3 [36] and earlier versions of the Oberon System was the addition of a module **Objects** which defines root types **Object** for arbitrary objects and **ObjMsg** for arbitrary object messages:

```
TYPE
  ObjMsg = RECORD   (* base type of all messages sent to objects *)
    stamp: LONGINT;   (* message time stamp *)
    dlink: Object;   (* sender of the message *)
  END;

  Handler = PROCEDURE (obj: Object; VAR M: ObjMsg);

  Object = POINTER TO RECORD   (* base type of all objects *)
    handle: Handler   (* message handler *)
    stamp: LONGINT;   (* time stamp of last message processed by object *)
    dlink: Object;   (* next object in the message thread *)
    slink: Object;   (* next object in a list of objects *)
    lib: Library; ref: INTEGER;   (* library and reference number of object *)
  END;
```

Handler. An object's most important field is its *handler* procedure, which dynamically dispatches all message records that it receives. Any derivation of the general **ObjMsg** root type is accepted by the handler procedure of any derivation of **Object**; Marais in [51] calls this an *open message interface*. The open message interface allows messages to be *broadcast* within a complex object graph even if not all objects in the graph know how to handle the message. If an object that receives an unknown message contains other

objects, it will forward the message to them in the hope that they know how to handle it. Otherwise, unknown messages are simply ignored. When the object graph contains a large number of objects, the resulting overhead may become significant, in which case suitable strategies for pruning the broadcast tree must be considered.

For all message types, message handling can also be delegated to the handler of an object's base type, inheriting the corresponding default behavior if appropriate. Because handlers are standard record fields, they can be exchanged at run-time (see Section 5.4). Furthermore, because each object has its own handler, behavior is not bound to type, but rather to each instance of a type. Even if two objects have the same type, they may still handle messages differently. Consequently, polymorphism is *instance-based* rather than *class-based* as in C++ or Java. We can also make an explicit distinction between types and classes and say that the handler of an object defines its class. Thus, many different classes can implement the same type [79].

Message Families. Dynamic message dispatch has the slight disadvantage that – within a handler procedure – the exact type of a given message record must first be evaluated using run-time type information, which is not as efficient as statically dispatching method procedures. However, by gathering related messages into a *message family* and deriving them from a common family base type instead of the general **ObjMsg**, the number of type tests at run-time can greatly be reduced.

One such family of messages, based on **Display.FrameMsg**, serves to broadcast events within the *display space*, which includes all currently visible *frames* in a running Oberon system. Frames may be nested, even within more than one parent frame, which results in a directed acyclic graph (DAG) that represents the entire display hierarchy. An example of an event that needs to be broadcast is a change to a model object of which the frames displaying the model must be notified. Thus, the display space simplifies implementing the Model-View-Controller (MVC) strategy that is used in applications with graphical user interfaces [48] because no explicit list of views and controllers must be maintained for each model. The role of frames in Oberon's compound document architecture is further expanded on in Section 5.2.

Message Variants. Similar actions usually require similar kinds of information to be exchanged between senders and receivers of messages. Related actions are therefore often merged into a single message type. An additional field, typically called **id**, determines the exact *variant* of the message. For example, the standard **AttrMsg** has variants for retrieving attribute values from objects, for storing new values, and for enumerating all an object's attributes.

Object Messages. Some concrete messages that each object is expected to handle are already defined in module **Objects**. They include the **AttrMsg** and **LinkMsg** for querying, getting, and setting named attribute values and references to other objects. The **CopyMsg** requests an object to return an identical copy of itself. The **BindMsg** and **FileMsg** provide an efficient and simple solution for making arbitrary object graphs persistent by binding all objects to a *library* and storing the library as a whole.

Object Libraries. A library manages the mapping from pointer-based object references to serializable reference numbers and vice versa. To be able to recreate objects upon loading, each object has a unique *generator* attribute, which contains the name of a command procedure (see Chapter 3) that, when called, stores a new object of the correct type and with the proper handler procedure in a global variable **Objects.NewObj**.

Libraries appear in two variants: *private* and *public*. Private libraries are typically used for storing object graphs in the manner described above. The connection between an object and a private library is temporary; it often only stays in effect until the library has been stored. A public library, on the other hand, stores publicly shared objects and is stored in a separate file. It retains its objects until they are explicitly removed or until the library file is deleted. When objects are bound to libraries, references to public objects remain in the public library; when such a reference is stored, the name of the public library is stored in addition to the object's reference number. When a reference to a public object is loaded, the corresponding library is automatically loaded from file if it is not already in memory.

4.2 Shape Type Hierarchy

Module **Leonardo** exports the necessary types for structuring graphical objects into hierarchies of shapes. The shape type hierarchy is visualized in Figure 4.2. The corresponding types are discussed in the remainder of this section.

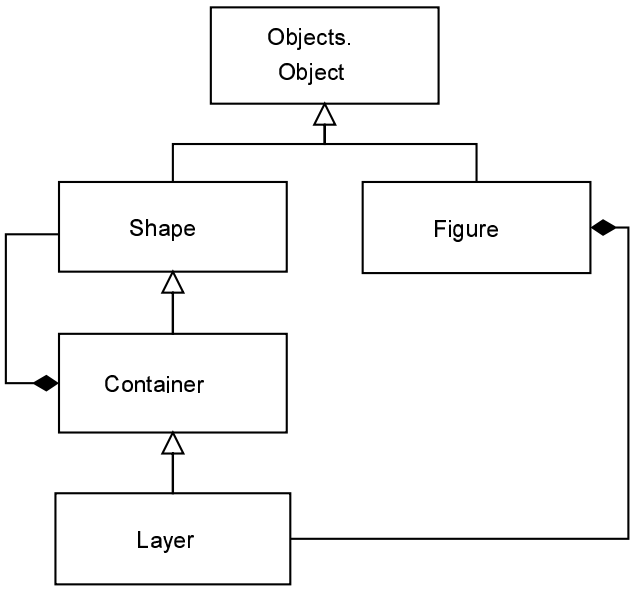

Figure 4.2: The shape type hierarchy (UML style notation)

4.2.1 Abstract Shape Base Type

All graphical objects in the Leonardo shape framework are derived from an abstract type **Shape**.

```
TYPE
  Shape = POINTER TO RECORD (Objects.Object)
    llx, lly, urx, ury: REAL;   (* bounding box in global coordinates *)
    bw: REAL;   (* border width *)
    sel: BOOLEAN;   (* set if shape or one of its components is selected *)
    marked: BOOLEAN;  (* set if shape is temporarily marked *)
    ...
  END;
```

Because **Shape** is in turn derived from **Objects.Object**, all shapes automatically gain the capability to be bound to libraries and thus to be made persistent. They also inherit the open message interface of general objects, which is used to implement shape behavior.

Except from a couple of state flags, the only information an abstract shape conveys is its bounding box, which is an axis-aligned rectangle that tightly encloses the shape's geometry. If rendering a shape affects areas outside its bounding box, for example because it is rendered with non-zero line width, the corresponding extra space is stored in **bw**, the shape's border width.

4.2.2 Container Shapes

Shapes that contain other shapes are called containers:

```
TYPE
  Shape = POINTER TO RECORD (Objects.Object)
    ...
    cont: Shape;   (* shape containing this shape *)
    up, down: Shape;   (* predecessor and successor within containing shape *)
    ...
  END;

  Container = POINTER TO RECORD (Shape)
    bottom, top: Shape;   (* components *)
    subsel: BOOLEAN;   (* set if at least one component is selected *)
  END;
```

Each container references both ends of a doubly linked shape list, with shapes' **up** and **down** fields leading from one shape to the next in both directions. Because shapes refer back to the shape that contains them with **cont**, each shape can only be part of exactly one containing shape, resulting in a tree structure. Figure 4.3 illustrates how shapes and containers interact. Gamma et al. call this structural design pattern a *Composite* [28].

By organizing shapes in trees, containers add structure to a set of related shapes. Besides, they speed up message broadcasts by not forwarding messages to their components when it is obvious that the message cannot affect them. In addition, containers influence shape behavior by exerting *parental control* (see [51]) when messages are broadcast within shape graphs. Based on information in the message record, in its own fields, and in its compo-

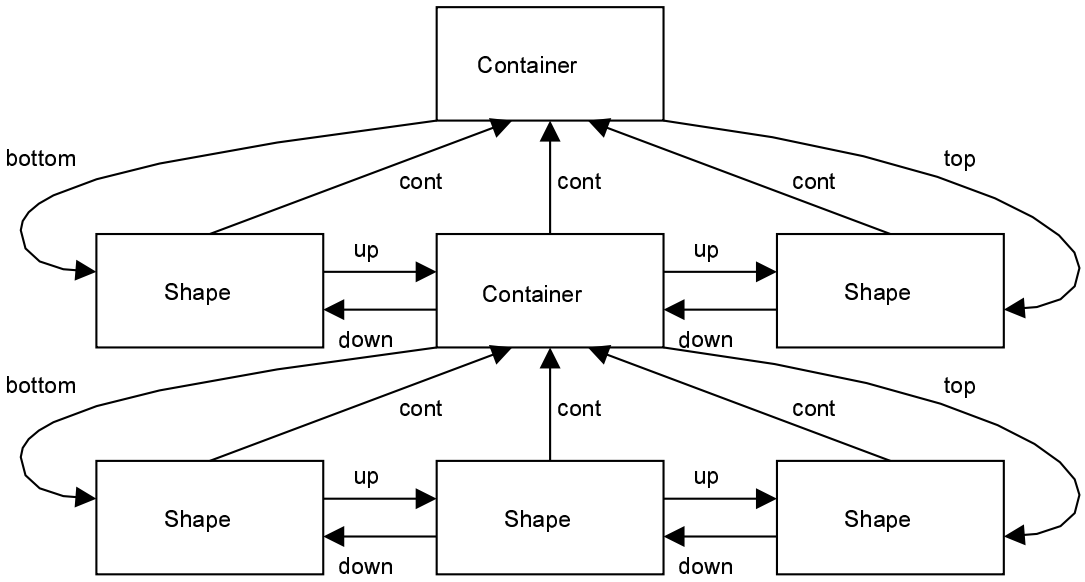

Figure 4.3: Links between containers and shapes

nents, a container may decide to handle a message itself or forward it to its components, possibly also modifying it beforehand. For example, the **Group** shapes from module **LeoBasic** do not allow their components to be selected or located individually. Thus, they hide their container nature and appear like regular leaf shapes.

4.2.3 Layers as Top-Level Containers

Layers are special containers at the top of the shape hierarchy:

```
TYPE
  Layer = POINTER TO RECORD (Container)
    fig: Figure;   (* figure that layer is part of *)
    name: ARRAY 16 OF CHAR;   (* layer name *)
    display, print, align: BOOLEAN;   (* filter flags *)
  END;
```

A layer has no containing shape; instead, it refers to the containing figure. Layers can be configured to filter out specific messages depending on a set of flags. Display on the screen or on the printer can be suppressed, as can

requests to align a point with a shape be. For example, guiding lines that aid interactive users in aligning shapes do not need to be printed and may belong to a layer that only appears on the screen. On the other hand, a watermark image in the background is likely to belong to a layer that only appears when printing.

4.2.4 Figures as Graphical Models

A figure is a single object which represents all shapes in a graphical scene. It manages a set of layers, serves as a model object for clients dealing with shape graphs, updates its clients about invalidated areas, and gathers modifications in undoable commands.

```
TYPE
  Figure = POINTER TO RECORD (Objects.Object)
    bottom, top: Layer;   (* layers containing shapes *)
    active: Layer;  (* currently editable layer *)
    ...
  END;
```

Each figure has exactly one *active layer*. Only the active layer integrates new shapes and receives messages that allow a client to select and manipulate existing shapes.

Figure Updates. When a shape is modified, all clients of a figure must be notified of the change, which is achieved by calling **UpdateRect** or **UpdateShape**, which in turn broadcast a corresponding update message within the Oberon display space.

```
TYPE
  UpdateMsg = RECORD (Display.FrameMsg)
    fig: Figure;   (* affected figure *)
    reg: GfxRegions.Region;   (* affected area *)
    bw: REAL;   (* border around affected area *)
  END;

PROCEDURE DisableUpdate(fig: Figure);
PROCEDURE EnableUpdate(fig: Figure);
PROCEDURE UpdateRect(fig: Figure; llx, lly, urx, ury, bw: REAL);
PROCEDURE UpdateShape(fig: Figure; shape: Shape);
```

To limit the number of update events, multiple updates can be merged into one by temporarily disabling updates with **DisableUpdate**. While updates are disabled, a figure accumulates individual rectangles that need to be redrawn in a *damage* region. Later, when updates are re-enabled with **EnableUpdate**, the figure broadcasts a single update message and resets its damage region. Because figures keep track of how often updates have been disabled, disabling calls can be nested.

Undoable Actions and Commands. Figures also maintain a list of undoable commands. The corresponding design pattern is known as a *Command* [28].

```
TYPE
  Command = POINTER TO RECORD   (private)
    next, prev: Command;   (* links to next and previous command *)
    actions, done, last: Action;   (* list of actions *)
  END;

  ActionProc = PROCEDURE(fig: Figure; action: Action);

  Action = POINTER TO RECORD
    do, undo: ActionProc;
  END;

PROCEDURE BeginCommand(fig: Figure);
PROCEDURE AddAction(fig: Figure; action: Action);
PROCEDURE CancelCommand(fig: Figure);
PROCEDURE EndCommand(fig: Figure);

PROCEDURE Undo(fig: Figure);
PROCEDURE Redo(fig: Figure);
```

Most operations that modify shapes call **BeginCommand** to create a *current command* before they send messages to affected shapes. Affected shapes handle these messages by adding undoable actions to the current command with **AddAction** instead of applying the resulting changes directly. Later, when the current command is committed with **EndCommand**, these actions are executed by calling their **do** procedure; the command itself is appended to a list of undoable commands in the figure. Clients can return to a previous state by calling **Undo**, which invokes the **undo** procedure of all actions in reverse order, and can execute an undone command again with **Redo**.

Commands are comparable to transactions in a database system. The actions in a command can only be executed together or not at all. Commands can be canceled with **CancelCommand** instead of committed with **EndCommand**. Besides, they can be nested, with each **EndCommand** or **CancelCommand** only affecting actions that were added since the most recent **BeginCommand**.

Figures vs. Display Space. Each figure forms a message broadcast space of its own, similar to the display space in Oberon. However, figures are models, i. e. they can only be viewed on a screen by visualizing them in a frame (see Section 5.3). Moreover, figures' undoable commands and accumulated updates have no equivalent in the display space. Thus, while many shape messages directly correspond to similar display messages, others are unique to figures.

4.3 Implementing Abstract Styles with Pen Objects

Gfx and other graphics interfaces render graphical objects with regard to a fixed set of graphical attributes. Instead of keeping a value for each attribute in each graphical object, applications usually manage a set of unique attribute combinations, typically called *styles*, and let each shape refer to such a style. The advantages are clear: memory requirements per shape are reduced to a single reference, and several shapes can share the same style. However, these style objects are not extensible as long as they only manage a fixed set of attributes. In this section, we describe how style objects can be given a more active role and become truly extensible.

4.3.1 Pen Interface

In [62], we showed how simple style objects can be converted to powerful *pen* objects by augmenting them with painting functionality. Pens are abstract objects that clients connect with a Gfx context and whose imaginary tip they lead along a path.

```
TYPE
  Pen = POINTER TO RECORD (Objects.Object)
    do: Methods;   (* pen methods *)
```

```
    ctxt: Gfx.Context;   (* graphic context the pen is connected to *)
  END;
```

Each pen decides on its own what graphical attributes it must maintain to achieve its desired output. For example, a pen for solid filling only needs a color value with which it fills the interior of its input path. Other pens may need to store more attribute values. Because pens are derived from **Objects.Object**, their properties can be inspected and manipulated with the standard **AttrMsg** and **LinkMsg** protocols of module **Objects**. Similarly, a **BindMsg** binds them to a library to persistently store them. New pen types can be implemented whenever needed, leading to an extensible pen type hierarchy.

The pen method interface is fixed, binding both implementors and clients to a well-defined interface.

```
  Methods = POINTER TO RECORD
    connect: PROCEDURE(pen: Pen; ctxt: Gfx.Context);
    disconnect: PROCEDURE(pen: Pen);
    enter: PROCEDURE(pen: Pen; x, y, dxi, dyi, bdist: REAL);
    exit: PROCEDURE(pen: Pen; dxo, dyo, edist: REAL);
    line: PROCEDURE(pen: Pen; x, y: REAL);
    arc: PROCEDURE(pen: Pen; x, y, x0, y0, x1, y1, x2, y2: REAL);
    bezier: PROCEDURE(pen: Pen; x, y, x1, y1, x2, y2: REAL);
    render: PROCEDURE(pen: Pen; ctxt: Gfx.Context;
          VAR bdist, edist: ARRAY OF REAL; n: LONGINT);
  END;
```

Comparing the choice of pen methods to the path methods of the Gfx context interface reveals that they are almost identical. Still, a few crucial differences exist.

Beginning and Ending Paths. When a new path is started with **connect**, the render mode is implicitly defined by the pen itself; each pen decides on its own whether to stroke, fill, or clip the resulting path. Instead of a **mode** parameter, a **ctxt** parameter tells the pen on which context it must produce output.

Subpath Offsets. Unlike Gfx contexts, pens cannot automatically close a subpath whose specification is already in progress. This forces its clients to use the enter/exit model for drawing closed subpaths (see Sections 2.2.3

and 3.2.3). The additional parameters **bdist** in **enter** and **edist** in **exit** (and their vector versions in **render**) are called *subpath offsets*. Their applications are discussed in Section 4.3.3.

4.3.2 Pen Chains

Some pens do not directly generate output. Instead, they modify their input path and forward the new path to another pen, using the same abstract pen interface that they implement themselves. An example is the **Dasher** pen from module **LeoPens**. It decomposes its input path into dashes according to its dash pattern and forwards these dashes to a second pen, which may in turn modify and forward its input path again. The last pen in such a chain finally renders its input path on a Gfx context. The dashing pen in this situation is the *master* pen that drives a *slave* or *base* pen. The corresponding design pattern is well-known and often used for filtering events [28] or for processing data streams [3]. However, it has to our knowledge never before been used for processing path specifications. Another example is the **Forker** pen, which forwards its input path to two slave pens at once, turning the pen chain into a tree.

4.3.3 Subpath Offsets

When a subpath is entered with **enter**, its **bdist** parameter contains the accumulated length of all segments between the logical starting point of the subpath and the current entry point. To further clarify this principle, Figure 4.4 displays two paths, one in the upper half and one in the lower half. Both paths consist of two connected lines that are rendered as separate

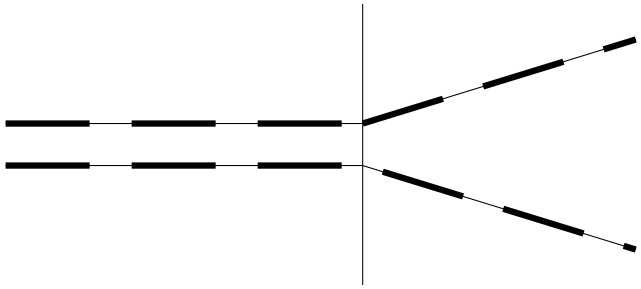

Figure 4.4: Effect of subpath offset in **enter**

subpaths from left to right. In the top half, both lines have been rendered with a **bdist** of zero. Because the dasher pen that was used to render them calculates the appropriate dash phase from the supplied **bdist** value, there is a discontinuity in the dash pattern at the point where the two lines meet. In the lower half, the second line has been given a **bdist** value equal to the length of the first line. This allows the dasher pen to compute the correct dash phase and results in a continuous dash pattern.

The applications of subpath offsets are not restricted to dash patterns. Arrow pens, for example, outline their input path and modulate its width in the vicinity of subpath ends to make them appear like arrow heads. When an arrow pen enters or exits a subpath, it uses the subpath offset value to calculate the proper width. If a dasher pen forwards its dashes to an arrow pen and uses continuous subpath offsets for its dashes (instead of giving each dash subpath offsets of zero), effects like the one in Figure 4.5 can be achieved.

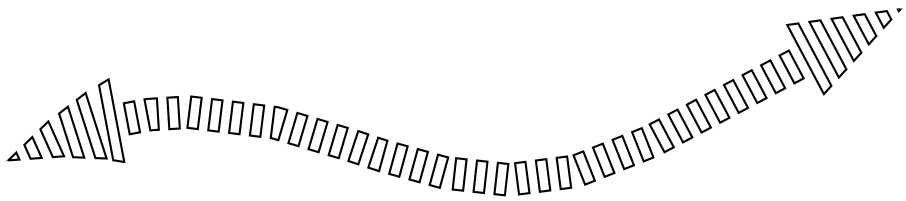

Figure 4.5: Subpath offsets influencing arrow width

4.4 The Shape Message Hierarchy

The Leonardo shape framework heavily relies on the open message interface that it inherits from Oberon's standard objects to implement shape behavior. The main attraction of an open message interface is its potential for extensibility. By introducing new message types, new behavior can be added whenever needed without any changes in an existing code base, circumventing the *syntactic fragile base class problem* that approaches which rely on method procedures exhibit [80]. This section presents several important kinds of messages and discusses their role within the shape framework. The inheritance structure of shape messages is illustrated in Figure 4.6. **Objects.ObjMsg** and **Display.FrameMsg** are standard messages

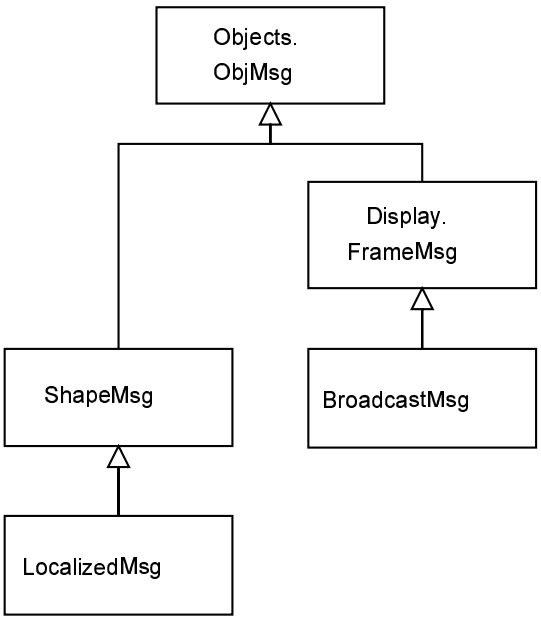

Figure 4.6: The abstract shape message hierarchy

of the the Oberon system, whereas **ShapeMsg**, **LocalizedMsg**, and **BroadcastMsg** are abstract messages of the shape framework, which are discussed in the following subsections.

4.4.1 The Shape Message Family

Shapes are requested to perform an operation by sending a message record of the appropriate type to their handler. To allow recipients to discern quickly between messages that are specifically targeted at shapes and other kinds of messages, shape messages form their own message family and are derived from a common root type **ShapeMsg**.

```
TYPE
  ShapeMsg = RECORD (Objects.ObjMsg)
    fig: Figure;   (* containing figure *)
  END;
```

Containers by default forward unknown shape messages to their components. This ensures that future extensions to the family of shape messages

always reach their destination shapes.

Only few messages are immediately derived from **ShapeMsg**. They mainly include messages that deal with shape state and the structure of the shape graph, but not with shape geometry. The following paragraphs describe examples of such messages.

Controlling the Shape Hierarchy. The two primary operations for modifying the hierarchical structure of a figure are *integration* of new shapes and *deletion* of existing shapes, both of which are achieved with the **ControlMsg**. Similarly, the **OrderMsg** allows existing shapes to be moved up or down within their containers.

When integrating new shapes, the message's fields exactly specify which container should consume the list of new shapes and between which of its components they should be inserted. When the container at which an **integrate** message is targeted receives the message, it adds an undoable action to the figure's current command.

Deleting or reordering shapes requires that the set of shapes that is affected by these operations can quickly be identified. Therefore, each shape possesses a field (**marked**) that may indicate whether the current operation affects it. When all actions of a command have been executed, shapes are automatically unmarked in the accompanying validation step (see Section 4.4.2).

Current Selection. A figure's current selection is a special subset of all its shapes, which is usually determined by interactive users and subject to frequent change. Selected shapes paint additional hints when they are rendered to visually distinguish them from their unselected counterparts. The current selection is managed with the **SelectMsg**. It includes variants that allow clients to build an ad hoc list of all selected shapes or to deselect all shapes in a figure at once.

Since the current selection is a volatile matter that is only of interest to interactive applications, our design would be cleaner if all issues regarding selection were delegated to these interactive applications themselves. However, because shapes are responsible for drawing their graphical appearance, we argue that they should also be responsible for drawing their selection marks. Consequently, if shapes must know about selection marks, they may as well maintain their own selection state to simplify selection management.

Still, selection management has a special status in the shape framework. For example, selection state is not persistently stored, and changes in selection state cannot be undone.

4.4.2 Localized Messages

An important aspect of hierarchical shape trees is that any container in the tree can change the coordinate system within which its components reside. Thus, messages that include geometrical information must carry along the description of a local coordinate system that is subject to change as a message is propagated through the shape tree. The base of all such messages is called **LocalizedMsg**:

```
TYPE
  LocalizedMsg = RECORD (ShapeMsg)
    lgm: GfxMatrix.Matrix;   (* conversion from local to global coordinates *)
  END;
```

A localized message's **lgm** field holds a matrix value that maps coordinates in the current *local* coordinate space into the *global* coordinate space of the containing figure. Providing a distinguished base type for all localized messages is crucial for achieving extensibility because it allows containers to recognize the localized nature of new messages and to properly adapt the **lgm** field even if they do not know the specific message type. The following example illustrates how a container that stores a local coordinate system in its **lcs** field forwards unknown localized messages to its components:

```
PROCEDURE Handle(obj: Objects.Object; VAR msg: Objects.ObjMsg);
BEGIN
  cont := obj(LocalizedContainer);
  IF msg IS ShapeMsg THEN
    ...
  ELSIF msg IS LocalizedMsg THEN
    WITH msg: LocalizedMsg DO
      lgm := msg.lgm;
      GfxMatrix.Concat(cont.lcs, lgm, msg.lgm);
      s := cont.bottom;
      WHILE s # NIL DO
        s.handle(s, msg); s := s.up
      END;
      msg.lgm := lgm
    END
```

```
    ELSE
    ...
END Handle;
```

In the remainder of this section, we discuss additional aspects of localized coordinates and examples of localized messages.

Shape Bounding Box. Although shapes store their geometry in the local coordinate system that is defined by their containers, their bounding box (see Section 4.2.1) is always specified in global figure coordinates to allow quick comparisons involving shape bounds. This for example permits containers to ignore and thus not forward messages that only affect a limited rectangular area if the message rectangle (in global coordinates) does not overlap the container's bounds.

Validation. A **ValidateMsg** is broadcast within a figure after all actions of an undoable command have been executed (or undone). Marked shapes which receive the message unmark themselves, recalculate their bounding boxes, update the figure at their old and new positions, and in turn mark their containers. The message has to be localized because bounding boxes are stored in global coordinates.

Rendering. The **RenderMsg** requests receiving shapes to paint themselves on a Gfx context. Depending on the exact variant of the message, selection marks are also painted or only marked shapes render themselves. In addition, the message contains a bounding rectangle of the Gfx context's clipping area to allow shapes that are completely outside this area to skip rendering altogether. Besides, a matrix which maps global figure coordinates to the default Gfx coordinate system allows shapes to paint graphical features (especially selection marks) with uniform size, irrespective of the current zoom level.

Locating. The **LocateMsg** defines a rectangle and requests receiving shapes to append themselves to an ad hoc shape list if they overlap the rectangle (**overlap** variant) or lie completely inside it (**inside** variant), thus supporting *point queries* as well as *range queries*. The **LocateMsg** can also be used to project a point to a nearby shape, which aids interactive users in aligning shapes. The projected point is expected to lie inside the rectangle that the

message defines. Thus, the distance over which projection is performed can be limited.

4.4.3 Broadcast Messages

Messages that are derived from **ShapeMsg** can be broadcast within a single figure because containers are required to forward shape messages to their components. However, events that have to be propagated to *all* existing figures have to be broadcast in the entire display space and must be derived from **Display.FrameMsg**. The Leonardo shape framework therefore defines a second message family based on **BroadcastMsg**:

```
TYPE
  BroadcastMsg = RECORD (Display.FrameMsg)
  END;
```

As with messages which are derived from **ShapeMsg**, containers by default forward even unknown broadcast messages to their components to ensure that they reach all shapes. Unlike regular shape messages, however, broadcast messages can be propagated to every component of the display space. Unfortunately, visual components by default do not forward messages that are based on **Display.FrameMsg** to their non-visual model objects. Thus, visual components that display figures must recognize and explicitly forward all messages that are derived from **BroadcastMsg** to their model.

A concrete extension of a **BroadcastMsg** is implemented by the **LeoPens** module to broadcast pen updates to all figures. Unlike shapes, which belong to at most one figure, pens can be referred to from shapes in different figures. Thus, when a pen is modified, all figures in the display space must be notified.

4.4.4 Using Optional Messages for Implementing Protocols

The grouping mechanism that container shapes implement is often sufficient for combining simple graphical objects, such as lines, arcs, rectangles, and circles, into complex shapes and complete graphical scenes. There are, however, situations that cannot be modeled accurately without introducing further dependencies between shapes. For example, sometimes two points from different containers should be kept aligned. Where such additional relationships between shapes are necessary, new shape messages are introduced. Shapes handle these messages if they are interested in and capable

of entering the corresponding relationships. However, they are not required to do so. If a shape handles such a message, we say that it *conforms to* or *implements* the corresponding *protocol*.

With our open message interface, such optional protocols are easy to define. Apart from an additional message type definition, there is no overhead whatsoever on shapes that do not conform to the protocol. Only shapes that do implement the protocol need to handle the message in their handler procedure. With a traditional object-oriented design that relies on method procedures, a similar solution can be achieved if the implementation language supports run-time type information and either multiple inheritance (like C++) or an interface mechanism (like Java). Programs can then check with a run-time test if a candidate object is derived from a protocol type or implements a protocol interface and appropriately type-cast it. Otherwise, corresponding methods have to be defined in the abstract base type, bloating its interface with methods that by default do nothing.

Contour Protocol. An example of an optional protocol is the contour protocol, which queries shapes whether they can render themselves with an abstract pen (see Section 4.3).

```
TYPE
  ContourMsg = RECORD (Leonardo.ShapeMsg)
    done: BOOLEAN;   (* set by receiver if it can render itself with a pen *)
  END;

  RenderMsg = RECORD (Leonardo.ShapeMsg)
    pen: LeoPens.Pen;   (* pen to use *)
  END;
```

If upon receiving a **ContourMsg** a shape sets the **done** field of the message, it is expected to also handle the **RenderMsg** and paint itself with the supplied pen object.

Leonardo's **LeoPaths.Path** shapes use the contour protocol to determine if they should integrate other shapes as components. When path shapes are requested to render themselves, they send all their components a **RenderMsg** with the path's pen object.

Vector graphics shapes (e. g. lines, arcs, rectangles, etc.) obviously tend to conform to the contour protocol. However, the same applies for example to shapes that display text captions. When they receive a **RenderMsg**, they

draw the outlines of all characters in the caption with the supplied pen. Thus, the contour protocol is a valuable tool for treating arbitrary shapes as paths. Another immediate benefit is that any shape that conforms to the contour protocol can be converted to a series of line, arc, and Bézier segments which can then be manipulated individually. Figure 4.7 shows an example of such a "pathified" caption after further editing.

Figure 4.7: Edited caption outlines

4.5 Transformations and Constraints

Depending on how individual shapes maintain their geometry, to transform a shape may imply that an affine transformation matrix is applied to one or several pairs of coordinates (e. g. the control points of a line or curve) or that it is concatenated with an existing matrix which describes a local coordinate system within which a shapes renders itself (e. g. for rectangles, ellipses, and captions). A corresponding shape message needs to hold a matrix value that describes the transformation and must be localized in order to be correctly handled within nested coordinate systems.

However, when there are additional relationships between shapes, the transformation of one shape may require that another is transformed as well. For example, when one end point of a line has been attached to the outline of a rectangle, it should move together with the rectangle when the rectangle is moved. In this section, we describe how the Leonardo shape framework implements geometric constraints and give examples of their application.

4.5.1 Geometric Constraints

Ever since interactive graphics editing has become possible with Sketchpad [77], corresponding programs have attempted to give users additional control over position and orientation of graphical objects by relating them to other objects. A number of different approaches have been devised for specifying and resolving the resulting geometric *constraints*. We present some of these approaches and their origins in the following paragraphs.

Gravity. The simplest method to assist users with positioning objects relies on an *alignment grid*. Such a grid restricts valid point coordinates to regularly spaced grid points. To a user it appears as if these points had immense *gravity* and attracted points to grid positions. Gravity can also be used to attract point coordinates to "interesting" areas of existing shapes, for example the corners of a rectangle. Indeed, the **project** variant of the **LocateMsg** can be used for exactly that purpose. However, such alignment relations between shapes are not persistent. If for example a line end point is moved to the outline of a circle, artificial gravity may perfectly align the point to the circle, but does not drag it along if the circle is later moved.

Snap Dragging. A logical extension to alignment grids is the *snap-dragging* technique (implemented in Gargoyle [11]), which adds temporary alignment shapes, such as oriented lines and circles with fixed radii, to a figure while a user is moving a shape. These temporary shapes guide shape alignment and make it easy to construct parallels and points with fixed distances to other points. However, snap-dragging does not keep shapes persistently aligned either.

Algebraic Solvers. Approaches that keep shapes constrained at all times are usually based on algebraic solvers. With these approaches, shapes are manually drawn, and constraints between shapes are gradually added, for example by telling the system that two lines should be parallel or that the distance between two points should be constant. These constraints are established by numerically solving a corresponding system of nonlinear equations. Because the constraints are part of the drawing, they can be reestablished whenever shapes are moved. Systems that use this approach (Sketchpad [77]; Thinglab [14]; Juno [59]) suffer from the problem that

nonlinear equation systems are hard to solve in the general case [67] and that the resulting solutions may not correspond to what a user had in mind, especially when the resulting system of equations is over-constrained or under-constrained. Better results have been achieved by restricting constraint categories to those that result in linear equation systems (Metafont [47]) or by differentially constraining initially legal shape configurations (Briar [31, 32]).

Kepler. Our inspiration has been a much simpler scheme that was implemented in the *Kepler* graphics editor for Oberon V4. Kepler is based on stars, constellations, and planets. Constellations connect stars with lines and other curves, but only stars can be dragged with the mouse. Planets depend on stars or other planets; their position is reevaluated when stars are moved. The asymmetry of stars and planets makes constraint solving trivial, but excludes constellations where points mutually depend on each other. By not fixing the roles of constrained and constraining shape, we overcome this restriction.

4.5.2 Constraint Propagation

We add a simple constraint mechanism to the Leonardo shape framework by handling transformations in multiple phases. In the *apply* phase, a transformation matrix to be applied to all marked shapes is broadcast within the figure. Because all shapes get to see the message, they are able to notice when a shape they observe is transformed. If a constraint is violated, but could be reestablished by transforming additional shapes, the receiver of the transformation request notifies its originator of this by setting a corresponding flag in the message.

The apply phase is then followed by an indefinite number of *notify* phases, in each of which a transformation notification is broadcast. Notified shapes are allowed to transform unmarked shapes to reestablish constraints on the premise that they mark them and again set the message's notification flag. This forces another notification round and gives further shapes a chance to react to the new situation. In this manner, transformations may result in more and more shapes being constrained and marked until a stable state is reached.

Example. Figure 4.8 illustrates how constraints can be used in practice. Its upper half displays two rectangles that are connected with a curve. Its lower half depicts the same three shapes after one rectangle has been moved. The end point of the curve retains its relative position on one side of the rectangle.

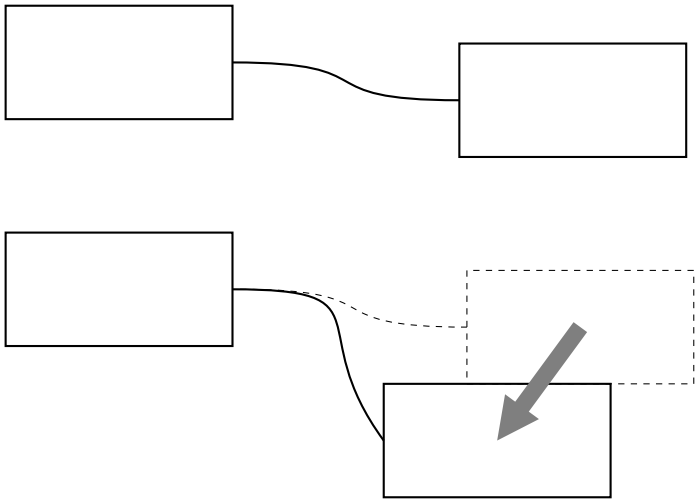

Figure 4.8: Constrained movement

Comparison. Compared to approaches that rely on numerical constraint solvers, our scheme is simple. Nevertheless, it is powerful enough to support several common usages of constraints, for example collinear Bézier control points that remain aligned when one of them is moved to achieve continuous slope between adjacent Bézier curves. However, it cannot deal with situations where multiple constraints affect a shape concurrently, even if a valid solution exists. For example, a line may constrain a point to lie on its trajectory. If a second line similarly constrains the same point, the point is not automatically moved to the point where both lines intersect, and at least one constraint will be violated.

4.5.3 Example: Points and the Link Protocol

Points are elementary shapes that anchor their containers at specific locations. The corresponding *link* protocol keeps geometrical locations aligned.

```
TYPE
  Point = POINTER TO RECORD (Leonardo.Shape)
    x, y: REAL;   (* point coordinates *)
    link: Leonardo.Shape;   (* shape controlled by point *)
  END;

  LinkMsg = RECORD (Leonardo.LocalizedMsg)
    id: SHORTINT;   (* get/set *)
    done: BOOLEAN;   (* to be set by receiver *)
    x, y: REAL;   (* coordinates in global space (in/out) *)
  END;
```

Each point is able to control at most one linked shape. The link is often another point, but can in principle be any shape that conforms to the protocol that the link message defines. The link message asks its receiver to return its own coordinates or to set them to the values that are delivered in the message.

The following procedures illustrate how transformations and linking are implemented for point shapes.

```
PROCEDURE TransformPoint(p: Point; VAR msg: Leonardo.TransformMsg);
  VAR x, y: REAL; lm: LinkMsg;
BEGIN
  IF msg.id = Leonardo.apply THEN   (* apply phase *)
    IF ~p.marked THEN
      IF (p.link # NIL) & p.link.marked THEN
        (* adapt point later when link's coordinates have been committed *)
        msg.notify := TRUE
      END
    ELSIF msg.stamp # p.stamp THEN   (* not yet handled *)
      p.stamp := msg.stamp;

      (* convert transformation matrix to local coordinate system *)
      GfxMatrix.Apply(msg.lgm, p.x, p.y, x, y);
      GfxMatrix.Apply(msg.mat, x, y, x, y);
      GfxMatrix.Solve(msg.lgm, x, y, x, y);

      (* use undoable actions to set new values *)
      Leonardo.SetReal(msg.fig, p, "X", x);
      Leonardo.SetReal(msg.fig, p, "Y", y);
      IF p.link # NIL THEN
        msg.notify := TRUE   (* adapt link in notify phase *)
      END
    END
```

```
  ELSIF (msg.id = Leonardo.notify) & (p.link # NIL) THEN   (* notify phase *)
    IF p.marked & ~p.link.marked THEN
      (* link must follow point *)
      lm.stamp := msg.stamp; lm.fig := msg.fig; lm.id := set; lm.done := FALSE;
      Leonardo.GetCoordSystem(p.link, lm.lgm);
      GfxMatrix.Apply(msg.lgm, p.x, p.y, lm.x, lm.y);
      p.link.handle(p.link, lm);
      IF lm.done THEN p.link.marked := TRUE; msg.notify := TRUE
      ELSE Unlink(msg.fig, p)   (* break link relationship if not fulfilled *)
      END
    ELSIF ~p.marked & p.link.marked THEN
      (* point must follow link *)
      lm.stamp := msg.stamp; lm.fig := msg.fig; lm.id := get; lm.done := FALSE;
      Leonardo.GetCoordSystem(p.link, lm.lgm);
      p.link.handle(p.link, lm);
      IF lm.done THEN
        GfxMatrix.Solve(msg.lgm, lm.x, lm.y, x, y);
        Leonardo.SetReal(msg.fig, p, "X", x);
        Leonardo.SetReal(msg.fig, p, "Y", y);
        p.marked := TRUE; msg.notify := TRUE
      ELSE
        Unlink(msg.fig, p)
      END
    END
  END
END TransformPoint;

PROCEDURE LinkPoint(p: Point; VAR msg: LinkMsg);
  VAR x, y: REAL;
BEGIN
  IF msg.id = get THEN
    (* convert point coordinates from local to global space *)
    GfxMatrix.Apply(msg.lgm, p.x, p.y, msg.x, msg.y);
    msg.done := TRUE   (* signal that message was handled *)
  ELSIF msg.id = set THEN
    GfxMatrix.Solve(msg.lgm, msg.x, msg.y, x, y);
    Leonardo.SetReal(msg.fig, p, "X", x);
    Leonardo.SetReal(msg.fig, p, "Y", y);
    msg.done := TRUE
  END
END LinkPoint;
```

With the link protocol, points (and other shapes that implement it) can for example be attached to rectangles and ellipses and retain their relative position when these shapes are transformed. A similar protocol exists for

establishing relationships between path segments. This *connect* protocol allows adjacent path segments to retain continuous slope at the points where they meet.

4.6 Summary

The Leonardo shape framework provides its clients with the foundations for modeling graphical scenes in a hierarchical, persistent, and extensible manner. Persistence and extensibility result from deriving all relevant object types from the standard Oberon **Objects.Object** and inheriting its open message interface; shape hierarchies are created with the help of container shapes, which manage a set of shape components.

Instead of only one extensible type hierarchy, the framework exports *four* different type hierarchies, all of which add another dimension to its potential for extensibility. Extending the **Shape** hierarchy integrates new graphical objects, extending the **ShapeMsg** hierarchy enables new shape behavior, extending the **Action** hierarchy makes new editing operations undoable, and extending the **Pen** hierarchy gives existing shapes new graphical capabilities.

Unlike object-based graphics interfaces such as Java 2D, the Leonardo framework is not limited to the display of graphical objects. Shape structure and standard shape behavior are also part of the framework; the resulting scenes are dynamic and can be modified using well-defined message protocols. This focus on dynamically manipulated scenes makes the framework less general than an object-based graphics interface. However, the latter is in turn less general than an API such as Gfx, which uses an immediate rendering model and does not require its clients to model graphical objects explicitly.

The framework distinguishes itself from other editor frameworks by its extensible hierarchy of abstract pen objects. We have not been able to find a similarly powerful concept in any other existing commercial or academic graphics software. Also, due to its use of an open message interface and its extensible actions, the framework can be adapted to domain specific applications (a declared goal of Unidraw [82]) by adding domain specific shape behavior.

CHAPTER 5

Application I
The Leonardo Figure Editor

The Leonardo figure editor [63] is the first of two applications that use the Gfx graphics API and the Leonardo shape framework, validating the feasibility of their design by using them in practice. The second application, a graphical description language called Vinci, is covered in Chapter 6.

Leonardo's intended usage patterns are manifold: on one hand, it should enable users to edit graphical documents that can be printed, stored, migrated, and loaded again. On the other hand, it should provide a wrapper around figure objects to integrate them within other documents, for example as embedded figures in a text document (such as the figures in this thesis) or as graphical decorations in a user interface. Thus, Leonardo not only provides a builder tool for creating figures, but also elevates abstract figure models to universally reusable graphical components.

Leonardo's design follows the Model-View-Controller (MVC) paradigm that the Smalltalk system introduced [48]. Its model components are the Leonardo shape framework's figure objects that were described in Chapter 4. The current chapter focuses on view and controller aspects. Our first goal is to allow editing functionality to grow when new objects are added to the model without re-compilation of existing code being necessary. Our second goal is to integrate graphics as light-weight components in an existing visual component framework. These goals can only be achieved if code is dynamically loaded when needed, not as part of a monolithic application.

After an overview of Leonardo's overall architecture in Section 5.1 and a short summary of the the relevant aspects of the Gadgets component frame-

work in Section 5.2, we describe Leonardo's main subsystems in Sections 5.3 (Views), 5.4 (Controllers), 5.5 (Documents), and 5.6 (Editor Panels).

5.1 Architecture

As displayed in Figure 5.1, Leonardo consists of several architectural software layers. At the bottom is the *model layer* with the Leonardo shape framework,

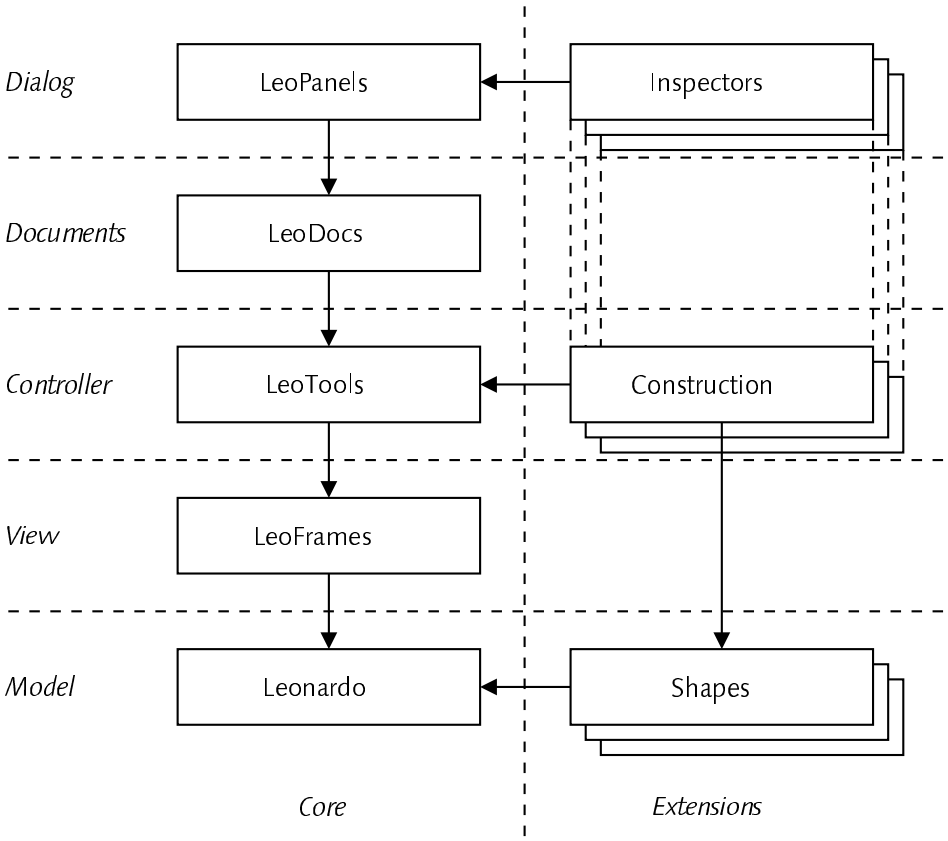

Figure 5.1: Overview of Leonardo's architecture

above which the *view* and *controller layers* reside. The two topmost layers are the *document layer* and the *dialog layer*. Each layer depends on the layers below it, but can be loaded without loading those above it. This vertical decomposition is complemented by a horizontal division into *core* and *extension* modules. The core contains modules that implement the basic

functionality of each layer. An arbitrary number of extension modules adds specific functionality to that core. With only few exceptions, extensions are mutually independent and can be loaded in arbitrary order (or not at all).

To separate mere viewing from editing functionality, extensions are usually split into a model part, which implements a shape type, a controller part, which allows users to interactively integrate new instances of that type in a figure, and a dialog part for inspecting and manipulating the properties of such instances. As is hinted in Figure 5.1, shape specific controller and dialog extensions are often merged within a single module because they are typically used in conjunction. However, they are kept separate from the module that implements the corresponding shape model. Thus, merely viewing a shape only requires its model to be loaded, whereas instancing or inspecting it also requires its controller or dialog module to be loaded.

5.2 The Gadgets Component Framework

Leonardo utilizes the *Gadgets* component framework for Oberon System 3 [51]. Gadgets itself is built on and extends the classic textual user interface known from previous Oberon implementations. The Gadgets framework comprises a multitude of visual and non-visual components, called *gadgets*, from which graphical user interfaces (GUI) can be constructed. The standard set of gadget types includes – among others – standard control elements such as buttons, check boxes, text fields, and sliders, but also complex controls such as text boxes and hierarchical lists.

Gadgets. Module **Gadgets** extends the standard **Objects.Object** for non-visual and **Display.Frame** types for visual components.

```
TYPE
  (* model components *)
  Object = POINTER TO RECORD (Objects.Object)   (* non-visual *)
    attr: Attributes.Attr;   (* dynamic attribute list *)
    link: Links.Link;   (* dynamic link list *)
  END;

  (* visual components *)
  Frame = POINTER TO RECORD (Display.Frame)   (* visual *)
    attr: Attributes.Attr;   (* dynamic attribute list *)
    link: Links.Link;   (* dynamic link list *)
```

```
    state: SET;   (* e.g. transparent *)
    mask: Display3.Mask;   (* visible area *)
    obj: Objects.Object;   (* model object, if any *)
  END;
```

Both types contain dynamic lists of named attribute and link values. Frames also contain references to a model object and to a mask structure that manages the area where they are visible, i.e. not occluded by other frames. Gadgets implement the message protocols that are defined in modules **Objects**, **Display**, and **Gadgets**. This allows them to be integrated within arbitrary container gadgets and therefore to serve as visual components.

Documents. Although all objects which are derived from **Objects.Object** can be made persistent by binding them to a library and storing the library in a file, there has to be some instance at the top of an object hierarchy which manages this process. In the Gadgets framework, this controlling instance is often a *document*. A document embeds gadgets in a wrapper frame and enhances its contents with a menu bar and a document file name. The menu bar contains a gadget displaying the document's name and buttons for storing and closing it. Documents are thus also responsible for embedding gadgets in Oberon's display structure, either in a tiled viewer system or in a desktop viewer which hosts overlapping frames.

The document type is defined in module **Documents**:

```
TYPE
  Document = POINTER TO RECORD (Gadgets.Frame)
    name: ARRAY 128 OF CHAR;   (* document name *)
    Load: PROCEDURE(D: Document);   (* load document contents from disk *)
    Store: PROCEDURE(D: Document);   (* store document contents to disk *)
  END;
```

All documents are instances of **Documents.Document**. They only differ in their **handle**, **Load**, and **Store** procedures, which are initialized by the document's generator procedure. The name of the generator procedure for creating a document is either stored in the document file or is derived from its file name extension. Attempting to load a document with the name of a non-existing file creates a new empty document.

5.3 Generic Views

The Leonardo view layer is responsible for visualizing and integrating figures and the shapes they contain within a graphical user interface, as visual components within compound documents. Because shapes know how to render themselves on abstract graphical contexts, Leonardo's views can be kept remarkably simple.

5.3.1 Extensible Models

Most model objects in the Gadgets framework manage simple scalar values, such as booleans, numbers, or strings. Even the few models that have moderate complexity, most notably text objects, are hardly ever extended. However, the figure models that Leonardo manages are designed to be enhanced by new shape and pen types all the time, leading to the central question of how to design a viewer component for visualizing an ever-growing model.

Unidraw, a framework for building domain specific graphical editors [82], is similar in scope to the Leonardo framework. It solves the above problem by splitting graphical objects into *subjects*, which are abstract representations of domain specific objects, and *views*, which display a graphical representation of a subject. While this approach allows attaching semantical information to subjects and having multiple view types per subject, it has the disadvantage that it doubles the structure of the shape graph. When subjects are added or removed, the resulting structural changes to the subject graph must be reflected in the view graph to retain a consistent state. For each graphical object, two distinct instances must therefore be maintained, one in the model, and one in the view domain.

By exploiting the ability of shapes to render themselves on abstract graphical contexts, Leonardo avoids such parallel type and object hierarchies. Due to shapes' generic rendering capabilities, a single view component suffices to visualize any figure model, irrespective of the complexity of the underlying model.

5.3.2 Leonardo Frames

Leonardo extends **Gadgets.Frame** in module **LeoFrames**.

```
TYPE
  Frame = POINTER TO RECORD (Gadgets.Frame)
    ox, oy: INTEGER;   (* vector to figure origin in screen space *)
    scale: REAL;   (* viewport scale factor *)
    col: Display.Color;   (* background color (unless transparent) *)
    framed: BOOLEAN;   (* whether to draw a 3D border around the frame *)
  END;
```

Each Leonardo frame refers to a figure model in its **obj** field. It renders this figure at its current location whenever it is requested to redraw itself with a **Display.DisplayMsg**. The **ox**, **oy**, and **scale** fields allow the definition of an arbitrary window into the figure. By default, the figure origin is aligned with the top left corner of the frame, and the scale factor is equal to one. Other fields influence whether and how a solid background and a border around the frame are painted.

Restore. To restore a frame on the screen, a Gfx context must be initialized with the frame's display mask and its local coordinate system. Almost everything else can be delegated to the frame's figure and its shapes.

```
PROCEDURE Restore(frame: Frame; x, y, w, h, fx, fy: INTEGER; mask: Display3.Mask);
VAR clip: GfxRegions.Region; ctxt: Gfx.Context;
BEGIN
  Oberon.RemoveMarks(fx+x, fy+y, w, h);   (* remove system markers *)
  IF ~(Gadgets.transparent IN frame.state) THEN   (* restore background *)
    Display3.ReplConst(mask, Color(frame), fx+x, fy+y, w, h, Display.replace)
  END;
  clip := RegionFromMask(mask);
  ctxt := DisplayContext(frame, fx, fy, fx+frame.W, fy+frame.H, clip);

  (* render on shapes on context *)
  Leonardo.Render(frame.obj(Leonardo.Figure), Leonardo.passive, ctxt);

  IF frame.framed THEN   (* restore border *)
    Display3.Rect3D(mask, Display3.topC, Display3.bottomC,
      fx, fy, frame.W, frame.H, 1, Display.replace)
  END;
  IF Gadgets.selected IN frame.state THEN
    Display3.FillPattern(mask, Display3.white, Display3.selectpat,
      fx, fy, fx+x, fy+y, w, h, Display.paint)
  END
END Restore;
```

Implications of generic views. Delegating rendering requests to individual shapes actually leads to a conflict with the MVC paradigm since MVC postulates that a model is completely decoupled from its views. For a graphical model, however, the visual appearance of individual shapes *is* indeed a property of the underlying model. By using abstract rendering contexts whose interface is not coupled with any concrete output device, figure models retain their independence from the environment within which their views are embedded. We thus argue that the MVC principle, although technically violated, is still followed in spirit. Another effect of a generic view component is that viewing functionality makes for a continuously smaller fraction of the total amount of program code as the number of shape extensions grows, almost reducing the MVC to an MC architecture.

5.4 Controllers

The difference between a *passive* frame that only displays a figure and an *active* frame that allows modifications to its model is that an active frame processes mouse and keyboard events and maps them to appropriate shape messages that it sends to its figure model. An active frame is also likely to display additional feedback, such as selection marks, exact mouse coordinates, or alignment hints.

An intuitive solution for adding active frames to Leonardo would be to derive a new frame type from **LeoFrames.Frame**, adding fields for storing new properties and including editing functionality in its handler. However, since the type of an embedded frame cannot be changed, this would complicate "in place" editing of already visible figures. Instead, Leonardo activates embedded passive frames by replacing their handler with one that implements editing capabilities. Moreover, it appends additional frame properties as dynamic object attributes. Thus, Leonardo frames can appear in two different roles – active and passive – between which they can arbitrarily switch at run-time.

The code which implements a frame's active role does not have to reside in the same module as the code for its passive role. Thus, the module that contains the code of active frames only needs to be loaded when the first active frame is opened or when an existing passive frame becomes active. This reduces the amount of code that has to be loaded when a document

with embedded figures is opened.

Tools. The principle of switching handlers to exchange appearance and behavior can be further generalized. Most graphics editors, including Leonardo, follow the notion of a *current tool* which defines what action is taken when a user presses a key or a mouse button inside an editor frame. Depending on the current tool, existing shapes under the mouse cursor are selected or transformed, or new shapes are integrated into the frame's figure. By identifying each such tool with a matching frame handler, new tools can be added by implementing a new handler procedure which incorporates the corresponding behavior. Again, tool code is only loaded the first time a user activates a tool. Thus, editor code is loaded in a fine-grained manner the first time it is needed. In principle, Leonardo can be extended at run-time, since new tool handlers can be written, compiled, and activated while figures are already being viewed or edited.

5.4.1 Tool Structure

Although tool handlers are free to handle all messages individually, most of them rely on the standard tool handler that module **LeoTools** exports for handling most messages and only adjust its behavior to handle mouse input events.

When a frame becomes active, additional data, stored in a *tool object*, is attached to it using a dynamic object link.

```
TYPE
  Tool = POINTER TO RECORD (Gadgets.Object)
    frame: LeoFrames.Frame;   (* frame that the tool object is linked to *)
    unit: REAL;   (* current unit of measurement (in figure coords) *)
    zx, zy: REAL;   (* vector from figure to ruler origin (in figure coords) *)
    grid: RECORD
      information about spacing and visibility of grid
    END;
    hints: RECORD
      information about alignment hints
    END
  END;
```

```
(* return frame's tool; create one if necessary *)
PROCEDURE Current(frame: LeoFrames.Frame): Tool;
VAR obj: Objects.Object; tool: Tool;
BEGIN
  Links.GetLink(frame, "Tool", obj);
  IF (obj # NIL) & (obj IS Tool) THEN tool := obj(Tool)
  ELSE NEW(tool); Init(tool); Links.SetLink(frame, "Tool", tool)
  END;
  tool.frame := frame;
  RETURN tool
END Current;
```

To activate a different tool, a **ToolMsg** is broadcast, asking all receiving tool frames to switch to the handler that is specified in the message.

```
TYPE
  ToolMsg = RECORD (Display.FrameMsg)
    handle: Objects.Handler;   (* new handler *)
  END;

VAR ToolHandler: Objects.Handler;  (* current tool handler *)

PROCEDURE Activate(handle: Objects.Handler);
VAR tm: ToolMsg;
BEGIN
  ToolHandler := handle;
  tm.F := NIL; tm.handle := handle;
  Display.Broadcast(tm)
END Activate;
```

Only frames that are already active handle **ToolMsg**, whereas passive frames ignore it. To activate a passive frame, its handler must be set to the current tool handler explicitly.

5.4.2 Rulers and Alignment

When a frame is active, part of its area is occupied by additional control areas that do not belong to the displayed figure, as shown in Figure 5.2. Horizontal and vertical *rulers* display a user coordinate system, defined by **unit**, **zx**, and **zy** in the **Tool** structure. In addition to providing users with an independent coordinate space, the ruler areas react to mouse clicks by scrolling the visible figure area. The top-left corner is used to shift or reset

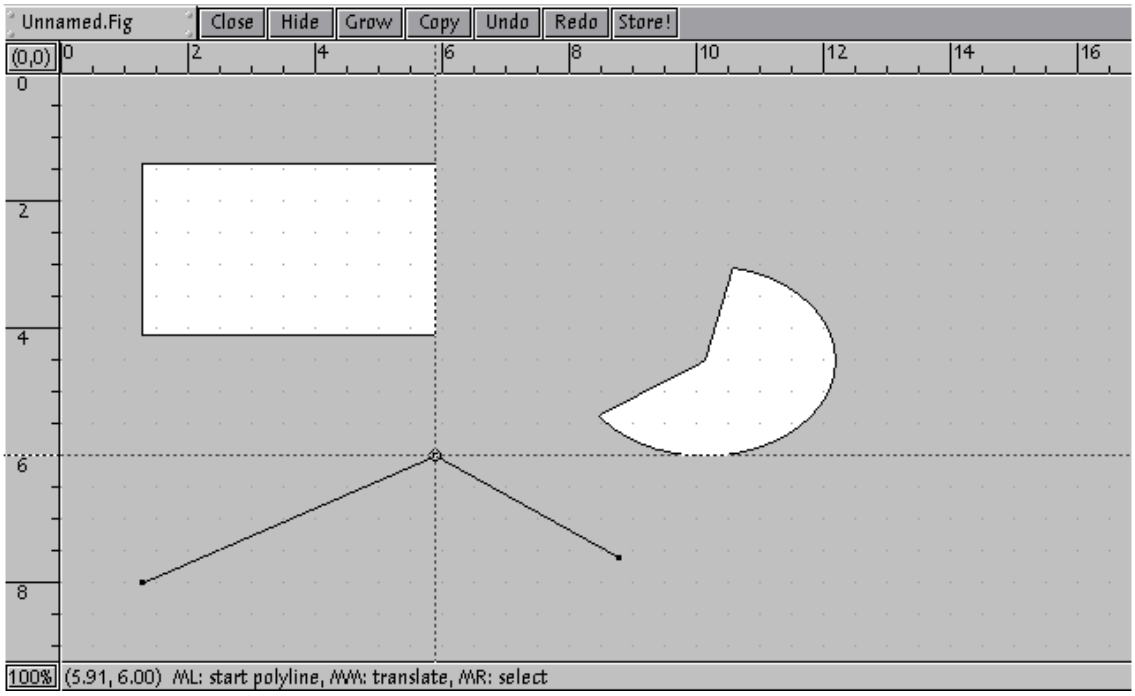

Figure 5.2: Active frame features

the origin of the ruler coordinate system, and the bottom-left corner zooms into or out of the displayed figure.

Along the bottom border of the frame, a status line displays the current mouse coordinates in ruler space and what action will be taken when a mouse button is pressed, notifying users about which tool is active.

The default tool handler in **LeoTools** implements grid alignment and shape specific gravity, as described in section 4.5, to assist users with positioning shapes accurately. By holding the CTRL modifier key on the keyboard during mouse movements, users can horizontally and vertically align the current position of the mouse pointer with the nearest shape specific alignment points.

To give users feedback on shape alignment, horizontal and vertical guiding lines are drawn while CTRL remains pressed. These guiding lines (called *alignment hints*) allow users to determine with which gravity source of which shape a dragged point is being aligned. For example in Figure 5.2, the CTRL key is being pressed. As a result, the guiding lines show that the point that is being moved is horizontally aligned with the right edge of the

rectangle above it and vertically with the bottom tangent of the pie shape to its right.

5.4.3 Mouse Tracking

The Oberon user interface is traditionally controlled with a three-button mouse, all three buttons being associated with a standard action: the left mouse button places insertion marks, the middle button activates commands and moves or reshapes gadgets, and the right button selects objects. Leonardo tries to match these standard assignments as closely as possible. It uses the right mouse button to select shapes and the middle button for transforming them. The action associated with the left button is specific to the current tool and usually integrates new shapes in a figure.

Drag Contexts. While the middle mouse button is kept pressed to transform selected shapes, affected shapes are drawn at intermediate positions to provide users with precise feedback of where they are moving them. As soon as the user moves the mouse, the frame's contents must be adjusted to reflect the new situation. Thus, dragged shapes are potentially painted at dozens of intermediate positions. Also, before painting them at a new location, the previous frame contents must be restored.

To efficiently deal with this situation, **LeoTools** implements a *drag context*, which extends the **GfxRaster.Context** type that was introduced in section 3.7. Drag contexts paint everything in *invert* mode. Hence, when the same output is rendered a second time, all changes are automatically canceled. Another feature of drag contexts is that they only approximate many painting operations. For example, they ignore dash patterns and line widths and always stroke paths, even if the actual render mode would request them to be filled.

Handle Transformations. By default, shapes are transformed by translating them according to mouse movements. However, if the shape located at the initial mouse position responds to a special shape message, called **Leonardo.MatrixMsg**, it can itself determine the transformation that should be applied. For example, when a user clicks on the corner of a rectangle, the rectangle returns a scaling matrix relative to the opposite corner of the corner that was clicked. To the user this appears as a resizing operation. Most

shapes that support such context-sensitive transformations paint special selection marks, called *handles*, at points where transformation behavior is customized.

Transformation Focus. Other transformations require the *focus* tool to be active. Users can then place *focus points* by clicking in a frame with the left mouse button. With every additional mouse click at the same location, the transformation type changes, going from translation to scale with the first click, from scale to rotation with the second, from rotation to reflection with the third, and from reflection to translation again with the fourth. The focus point is used as the origin of the appropriate transformation, except in the case of translation, which needs no origin.

If the mouse is moved while placing the focus point, the focus point becomes a focus axis and the type of the transformation changes to an appropriate directional transformation. Uniform scaling is thus replaced by directional scaling, rotation by a shear transformation, point reflection by directional reflection, and undirected translation by directional translation. The effects of these transformations and the corresponding focus point symbols are illustrated in Figure 5.3.

5.5 Documents

Although Leonardo frames can be integrated in arbitrary gadget containers and handler switching allows embedded figures to be activated and passivated at will, it is often preferable to treat figures as stand-alone documents. By surrounding figure frames with a document wrapper, figures can be stored in files and imported from files. Furthermore, document frames can be arbitrarily resized without disturbing the layout of an embedding container gadget.

The **LeoDocs** module implements a simple document wrapper for Leonardo frames. In addition to opening new documents or loading existing documents from file, **LeoDocs.Open** is also capable of opening any figure that is currently part of the display space in a document. The content frame of the new document is always active and shares its model with the already visible frame; subsequent modifications will thus affect both.

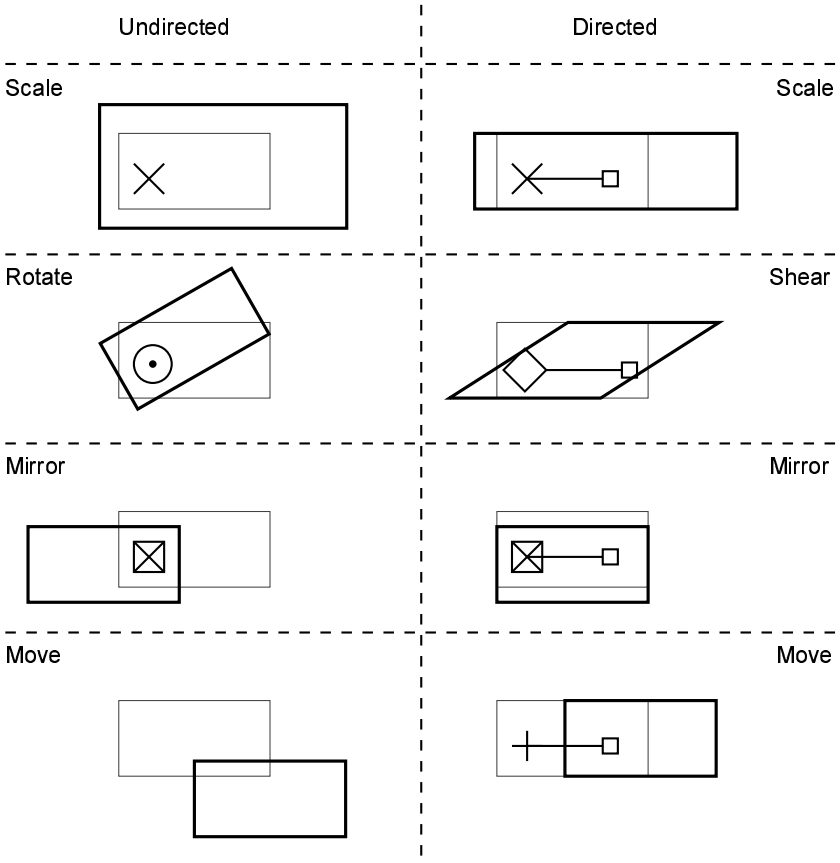

Figure 5.3: Affine transformations using focus points

5.6 Editor Dialogs

In this section, we describe how Leonardo's dialog structure is organized. Furthermore, we address the question of how customized dialogs for inspecting and modifying the properties of new shape extensions and new tool handlers are integrated with existing dialogs.

5.6.1 Application User Interfaces

All manipulations described so far were instances of *direct manipulation* [70], where the movements of a pointing device, such as a mouse, are directly translated to geometric actions. However, direct manipulation is not effective for modifying non-geometrical properties. Changing the font of a caption, for example, involves selecting the caption, choosing a font name and size, and invoking the appropriate command for assigning the new font to the caption. One of the main tasks of a GUI framework is thus to let application developers create and arrange interactive controls that allow users to choose options, quantify values, and invoke operations. Corresponding GUI elements are usually gathered within dialog windows. These may only temporarily be visible and be dismissed after the task they help accomplish has been achieved.

In the Oberon system, the traditional concept of a monolithic application where code and user interface components are stored together in a big executable file has been replaced with command procedures, dynamically loaded modules, and a document-based user interface. Application documents and application user interfaces are unified with a single document concept.

In practice this means that while a figure is displayed in a Leonardo document, the GUI elements that are necessary to open new Leonardo documents and edit the shapes they contain are part of a separate *panel* document. (Panels are container gadgets that allow their components to be arbitrarily placed.) Instead of "starting up" Leonardo, an Oberon user opens the panel document that contains the GUI elements with which Leonardo's functionality is invoked. With each activated command, additional modules that implement the requested operation are potentially loaded. Thus, code is only loaded when it is actually needed.

5.6.2 Leonardo Panel Hierarchy

Many Oberon applications are small enough to have all gadgets that are needed to control them fit within a single panel document. When the number of gadgets reaches a certain limit, however, a single panel does not provide enough space anymore. Consequently, subsets of related gadgets must either be managed more economically (e.g. by putting them in a special gadget that allows its contents to be scrolled or one that hides them until activated) or distributed to multiple panels.

Leonardo favors the second approach and distributes user interface components over a hierarchy of small panel documents. From an initial application panel, users open additional panels that contain exactly the GUI elements that they need for a specific task. To maximize the visible area of nested panels, Leonardo places them right below the title bar of their parent, as shown in Figure 5.4. When they are dismissed again, the previously visible

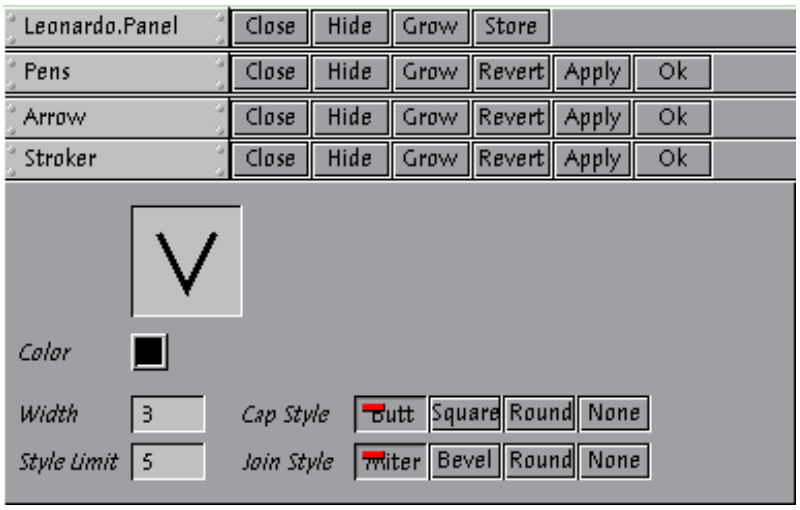

Figure 5.4: Leonardo stacking nested panels

panel reappears. While this automatic placement is usually adequate and reflects users' intentions, Leonardo panels are regular gadgets and can be moved around and resized like any other.

When new functionality is added, the normal action is to create a new panel document with the necessary GUI elements. To grant users immediate access to this new panel, a button for making it visible should be added to one of the existing application panels. The required manual intervention

may not be fully satisfactory, but seems unavoidable. Fortunately, dialog panels are regular documents. Oberon users are accustomed to the fact that they are allowed and even encouraged to edit existing user interfaces to adapt them to their own needs. The only scenario where there are potential conflicts is when independently developed extensions are merged.

To facilitate the integration of new dialog panels and to maintain a common look among Leonardo's panels, the GUI layout language *LayLa* [22] has been used to generate all Leonardo panels rather than designing them interactively. LayLa takes a textual description of visual and non-visual objects, including their dimensions, attributes, and links, and arranges them in rows, columns, and tables according to its built-in layout rules. To avoid cluttering the flat name space of Oberon's file system with dozens of different panel documents, Leonardo panels are gathered in a public object library and copied from there when needed.

5.6.3 Generic Object Inspection

Users are often interested in viewing and editing the properties of an object that they have selected. An application should then be able to make a matching dialog window appear. The Gadgets framework offers generic *inspector* panels for that purpose. Inspectors display an object's attribute and link values and let users change them. They rely on the basic **AttrMsg** and **LinkMsg** object messages from module **Objects** to determine name, type, and value of an object's attributes and links. An attribute's type determines the kind of gadget that is used for visualizing its value. For example, a string attribute is visualized with a text field gadget, or a boolean attribute with a check box gadget.

However, the number of distinct attribute types that **AttrMsg** is able to express is limited. For example, color values and line cap styles are both modeled as integer attributes. A generic inspector therefore models both as integer text fields although the color value would best be represented by a color chooser gadget and the line cap style by a set of radio buttons. Other approaches achieve this kind of customization by utilizing a generic mechanism for finding customized editor dialogs. For example with ActiveX controls [55], inspected objects are asked to return a list of class ids (which correspond to generators in Oberon) with instances of which their properties can be inspected. This closely binds components to their property

inspectors, which has the disadvantage that components must be recompiled to associate new inspectors with them. The JavaBeans API [76] that was introduced with Java 1.1 solves the problem of inspecting arbitrary components ("beans") with a dedicated naming scheme. For a class **Abc**, the class **AbcBeanInfo**, if it exists, is expected to contain additional information about the properties of **Abc** objects. Moreover, a custom dialog window for editing **Abc** components may be requested from a class **AbcPropertyEditor** if such a class file exists. Thus, property editors for Java beans are not as tightly coupled to the corresponding components as property pages for ActiveX controls are.

Leonardo uses a similar approach, but does not rely on a naming scheme to determine custom editor panels. Instead, it again uses the central Oberon registry for associating an object's unique **Gen** attribute with the name of a generator command that creates a matching custom panel. For example, the registry contains an entry that associates **LeoBasic.NewRect**, which is the generator command of rectangle shapes, with **LeoBasicEditors.NewRect**, which is the generator of a new panel for inspecting rectangle shapes. As with JavaBeans, objects and their editor panels are completely separated, but the explicit mapping between them is more flexible than a rigid naming scheme.

With the above mapping scheme, a single command can bring up a matching inspector panel for any selected shape. Similarly, Leonardo allows users to inspect the properties of the current tool. Tool inspectors display default attributes of the new shape objects that the tool integrates in a figure. The current tool is expected to return a matching inspector panel when asked for its **Editor** link.

5.6.4 Editor Objects

When inspecting objects and modifying their values, it is convenient if modifications are not immediately applied to the inspected object, but only when a user commits them explicitly. Most custom inspector panels therefore have a special *editor* model attached. Editor objects buffer the inspected object's properties. A generic editor object type is defined in module **LeoPanels**.

```
TYPE
  Editor = POINTER TO RECORD (Gadgets.Object)
```

```
    apply, revert: PROCEDURE(editor: Editor);
    fig: Leonardo.Figure;   (* associated figure *)
    frame: Gadgets.Frame;   (* associated frame *)
  END;
```

Concrete custom inspectors usually do not extend the generic editor type. Instead, they use the editor's dynamic attributes and links (as provided by **Gadgets.Object**) for storing the inspected object's properties. For example, most editor objects refer to the object that they represent with a dynamic **Model** link. In the **revert** procedure, the inspected object's properties are copied to the editor object, whereas in the **apply** procedure edited values are copied back to the inspected object, usually by adding undoable commands with corresponding actions to the figure to which the inspected object belongs.

Menu Bar. To allow users to revert to the original values of the inspected object or commit their changes, **Revert** and **Apply** buttons (invoking the corresponding **Editor** procedures) must be placed on custom inspector panels. Because these buttons are used in so many Leonardo panels, the Leonardo public library includes a prefabricated menu bar object that already contains those buttons. Panels are displayed with this menu bar if they attach a corresponding **Menu** link to their editor objects. When Leonardo opens a panel, it embeds it within a custom document class that retrieves its menu from the panel's editor object. It also retrieves the document name that is displayed in the menu bar from the editor's **Title** attribute. In a LayLa description, this is implemented as follows.

```
LayLa.AddToLibrary Leonardo.FillerPanel
(CONFIG
  (DEF editor (NEW LeoPanels.NewEditor
    (ATTR Title="Filler") (LINKS Menu=Leonardo.EditorMenu)))
  (VLIST Panel (border=8 w=384 vdist=4) (ATTR Locked=TRUE) (LINKS Model=editor)
    description of panel components
  )
)
```

Referrer Links. Editor objects are also used to connect nested panels to the panel that opened them. For example, when a user clicks on a gadget that displays a color value, a new panel with controls for editing that value is opened. This situation is depicted in Figure 5.5. The editor object of the

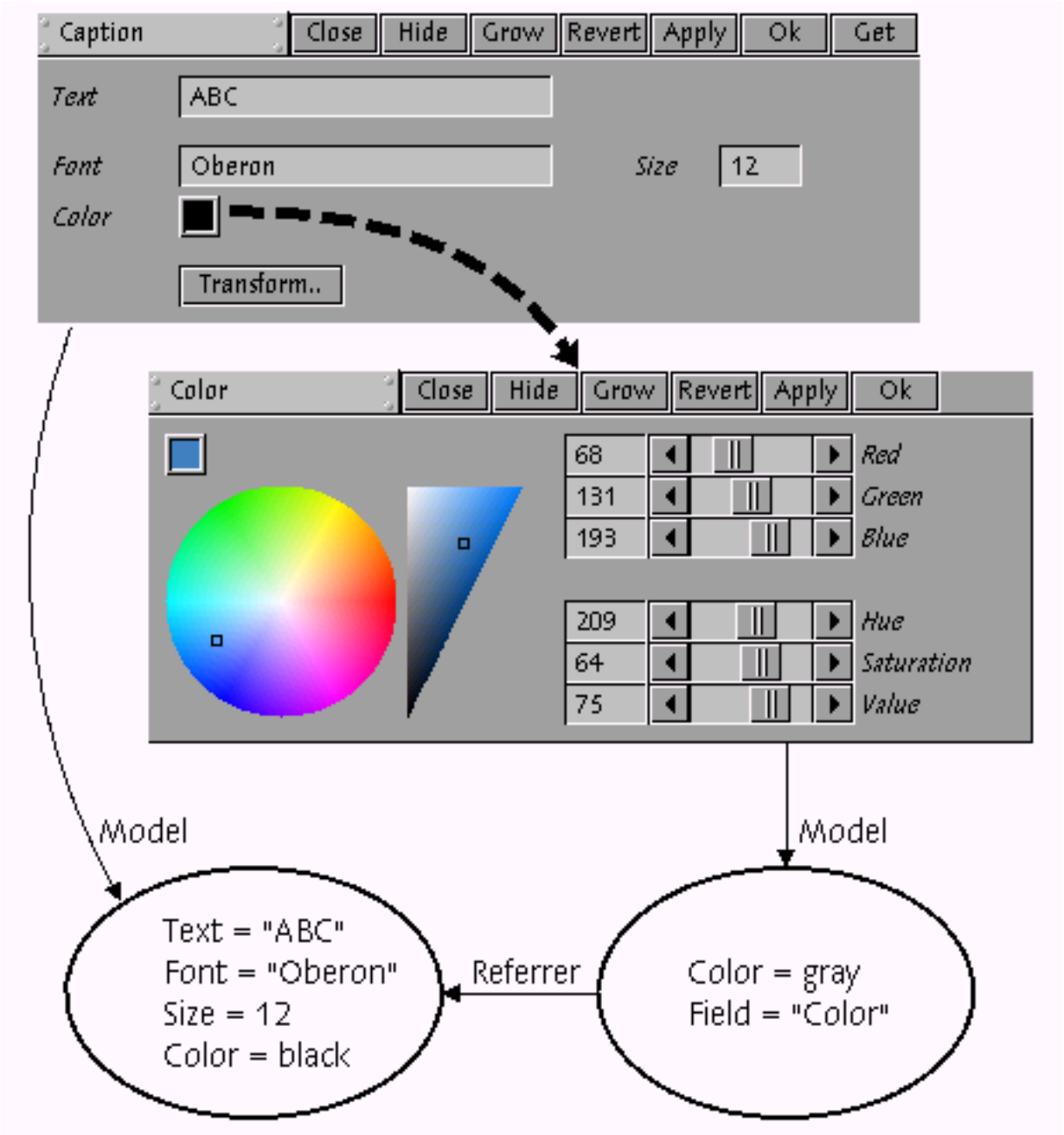

Figure 5.5: Editor object with **Referrer** link

new panel gets a **Referrer** link that points to the editor object of the original panel; the corresponding attribute name is stored in an attribute named **Field**. When a new color value is later confirmed by the user, the color editor follows the **Referrer** link and changes the referrer's **Field** attribute accordingly.

5.7 Summary

The Leonardo graphics editor exemplifies how the MVC paradigm can successfully be applied to complex, extensible model components. Due to the special characteristics of the underlying model, in which objects know how to render themselves, a single generic view component is sufficient for integrating graphics within arbitrary documents. Views can be propagated to full controller components at run-time by replacing their handler with a tool handler. Because functionality is spread over several layers and divided between mandatory core modules and optional extensions, code is loaded in small units and only when it is first needed. In addition to various editing tools for directly manipulating objects, Leonardo offers a hierarchy of editor panels and support for generic object inspection. Its design facilitates integration of new shape and pen types into its user interface with minimal effort.

CHAPTER 6

Application II
The Vinci Graphical Description Language

Vinci is a programming language for describing graphics, based on the graphics contexts of the Gfx API. Describing a shape or figure in textual form is often less intuitive than interactively constructing it with a graphics editor. In some situations, however, textual descriptions are so far superior to interactive construction that the cost of learning the corresponding language is well justified. Whenever coordinates must algorithmically be placed, describing their positions with exact numbers and precise calculations is essential. As the name suggests, we do not regard Vinci as a strict alternative to Leonardo, but rather as a complement. Together, they are more powerful and expressive than either of them on their own.

Like Leonardo figures, Vinci descriptions not only describe stand-alone documents, but can also be embedded as graphical components in other documents. In addition, Leonardo shapes and pens can be used within Vinci programs, and Vinci programs can be imported as shapes within Leonardo figures. This mutual integration of interactive and descriptive objects (see Section 6.2) is what makes the combination of Leonardo and Vinci unique.

Goals. From the intended usage of Vinci descriptions, we derive the following goals:

– Vinci programs are to be written and read by humans. The syntax of Vinci should therefore seem natural to both authors and readers. If no existing syntax can be reused, the syntax of Vinci should at least be easy to acquire.

- To support repeating structures, complex computations, and algorithmically generated graphics, Vinci should support general programming language constructs such as loops, alternatives, and recursive procedure definitions.
- Like Leonardo, Vinci should be highly extensible. This can on the one hand be achieved by allowing Vinci descriptions to import Vinci definitions from other files, on the other by letting programmers extend the set of built-in Vinci operators.
- Vinci should make full use of the features that Gfx and Leonardo pen objects offer, especially the early rendering model that both provide.
- Users should be able to mutually integrate interactive and descriptive graphics.

We have refrained from using one of the languages the we presented in Section 2.3 because they fail to fully satisfy the above goals. For example, SVG (as any other XML extension) only describe static object structures and do not have executable parts. Besides, their syntax is too verbose to be considered easy to read and write. Postscript and MetaPost would be acceptable in these respects and also allow programmers to import custom procedures. However, their set of built-in operators (for tasks that cannot be achieved with custom procedures) cannot be extended, nor do they support early rendering or facilitate integration with interactive graphics. For these reasons, we chose to design a new language, which we describe in the following section.

6.1 Language

Vinci programs are expected to primarily consist of operations that generate graphical output. These graphical operations are often complex and amount for the majority of the overall processing time. Thus, it is only of secondary importance how fast language constructs are executed, justifying the use of an interpreted language in favor of a compiled one.

The focus of Vinci descriptions lies not on maintaining internal data structures, but rather on invoking a series of graphical functions that, as a side effect, result in external changes that fall outside the scope of the language itself. Language constructs that enable general programming are

available, but only to make the task of describing graphical scenes easier.

The descriptive nature of Vinci programs encourages a functional programming style. We therefore decided to base Vinci on the semantics of an existing functional programming language that we augment with a custom syntax. In the current implementation, Vinci programs are parsed according to that custom syntax and then translated to corresponding *Scheme* programs. Scheme [68, 1] is itself a dialect of *Lisp* [54]. While Vinci uses Scheme mainly because of its clean semantics and flexibility and does not require Vinci programmers to know about Scheme, it does not artificially prevent them from using Scheme features either.

The complete Vinci syntax can be found in Appendix B. In this chapter, only those parts that are essential for illustrating the concepts behind the language are presented.

6.1.1 Program Representation

Vinci accesses its input files as standard Oberon texts. It converts the stream of input characters into a series of lexical tokens, which it analyzes in a recursive descent parser. The set of lexical tokens includes numeric constants (e. g. **23**, **0.125**), string literals (e. g. **"Hello, world!"**), identifiers (e. g. **abc**, **positive?**), and operators (e. g. **+**, **=**). Except within string constants, uppercase and lowercase characters are considered equivalent. Comments start with a double minus character (**--**) and include all remaining characters on the same input line.

Unlike other languages, Vinci has no dedicated keywords. Instead, its parser interprets normal identifiers as keywords at points where it has to determine how it should proceed. While this can make legal statements confusing (consider **if repeat=if then const repeat=else end**), it simplifies the parser. Individual statements in a statement sequence are separated by semicolons.

6.1.2 Data Types

Vinci uses dynamic typing, which means that the type of the values that a variable may contain are not fixed and do not have to be declared. Vinci inherits the Scheme type system, including pairs for building lists, vectors (arrays), and input/output ports. However, while list and vector literals can

be specified in a Scheme program, Vinci only accepts numeric and string constants in its input. Other types become available through functions that return corresponding values (e. g. the built-in Scheme function **vector**) and predefined constants (e. g. **true** and **false**).

In addition to the types mentioned in the Scheme report [68] (booleans, numbers, characters, strings, symbols, procedures, pairs, vectors, and ports), our Scheme implementation for Oberon includes an *object* type for embedding Oberon objects within Scheme programs. In particular, Vinci uses this object type to access shape, pen, and image objects.

6.1.3 Expressions

Unlike Scheme, which uses fully parenthesized prefix notation (e. g. (+ 3 x)) for expressing all operations, Vinci uses infix operators for building expressions. Its expression syntax is thus closer to that of imperative programming languages like Oberon or C. Indeed, Vinci's expression syntax is almost identical to that of Oberon. The only differences are that it has no relational operators **IN** and **IS** and that it supplies an additional addition operator **++** for concatenating strings.

6.1.4 Control Structures

In spite of being targeted at describing graphics, Vinci supports a minimal amount of control structures.

Conditions. The Vinci **if** statement evaluates a condition to decide among a set of alternative control paths. It is equivalent in form and semantics to the corresponding Oberon statement and thus comparable to the **cond** statement in Scheme.

```
if a < b then ...
elsif a > b then ...
else ...
end
```

Vinci merely transforms **if** statements to expressions involving the matching Scheme function (or rather special form) **if**. This function can also be called directly when the Vinci parser is not expecting an **if** statement and thus turns it into a normal function call:

```
const min = if(a < b, a, b)   -- in Scheme: (define min (if (< a b) a b))
```

Loops. Looping constructs in Scheme are modeled with anonymous functions (also called λ or *lambda* functions). Because Scheme implementations are required to support tail recursion, even loops with many iterations can be processed without running out of stack space. Vinci defines two looping constructs that are implemented in this manner with lambda functions, called **repeat** and **for**. Both run through a fixed number of iterations, but **for** defines an index variable within its body that contains the current loop index. More general looping constructs, corresponding to the **WHILE**, **REPEAT**, and **LOOP** statements in Oberon, have no place in a description language since a description is supposed to be self-contained and not to rely on external input. Therefore, the number of iterations of each loop can be calculated in advance.

6.1.5 Definitions

Each name in a Vinci program must be defined before it can be used. When a name is defined, a new *binding*, which associates the name with a value, is inserted in the current *environment*. Each function has its own environment, including "hidden" functions such as loop bodies. Since function definitions can be nested, so can environments be. When a function returns, its environment and with it all its bindings disappear. Bindings are only visible within their defining environment and in all environments nested therein. Because function bindings are inserted in the environment where they are defined (not where they are invoked), Vinci follows *lexical scoping* rules.

Vinci supports three binding constructs, called **const**, **define**, and **let**. The purpose of **const** is to define values that are frequently used in the corresponding environment.

```
const pi=3.1415, lblue=colors.rgb(0.8, 0.8, 1);
const avg=(min+max)/2;
```

Vinci cannot guarantee that names that are defined with **const** indeed remain constant because any program can call the built-in Scheme function **set!** to assign it a new value. A second **const** in the same environment also redefines the existing binding. However, the intended use of identifiers that are defined with **const** is indeed as constants.

Because Vinci does not support lambda functions, user functions cannot be defined with **const**. Function definitions must instead be made with **define**.

```
define length(dx, dy) as
  sqrt(dx*dx + dy*dy)
end length;
```

The function can later be called by writing its name followed by its arguments in parentheses.

```
const c=length(a, b);
```

All functions return the value of their last statement or expression, but callers are allowed to ignore function values.

Vinci allows its programs to take advantage of the definitions in the standard Scheme library. It therefore accepts several letters within identifiers that are regularly used in standard Scheme names. For example, **set!** and **vector?** are legal Vinci identifiers. However, some letters that are legal in Scheme names cannot be tolerated in Vinci names because they would break its expression syntax. This affects all conversion functions, for example **number->string**, and most string and vector functions. To deal with these names, the Vinci scanner accepts the otherwise unused backquote character (‘) as an escape character for quoting identifiers. Thus, **number->string** must be written as **‘number->string‘** and **make-vector** as **‘make-vector‘**.

Like Scheme, Vinci includes a **let** statement for creating temporary bindings.

```
let x=(x0+x1)/2, y=(y0+y1)/2 in
  stroke from (x-5, y-5) to (x+5, y+5) end
end
```

The new variables **x** and **y** are only defined within the **let** body, but make an explicit definition unnecessary. Within each variable definition in a **let** clause, the definitions to its left are already instantiated. Thus, the **let** statement in Vinci corresponds to **let*** in Scheme.

6.1.6 Graphics

Considering that Vinci aims to take advantage of Gfx and pen objects, it is no surprise that its graphical operations use the Gfx path model. Although

these operations could all be modeled with regular functions, Vinci offers syntactical constructs for specifying paths and text labels.

Path statements start with **fill**, **stroke**, **clip**, or **draw** (where **draw** instructs Vinci to render with the current pen), followed by a path body and ending with **end**. The body of a path primarily contains a series of subpath statements, but may contain control structures and function calls as well.

Subpaths in turn start with **from** or **enter..at**, followed by a sequence of subpath elements. If they are not explicitly terminated with **close** or **exit**, they end with the start of the next subpath or at the end of the entire path. Point coordinates are specified as pairs of numeric expressions, separated with a comma and embedded in parentheses. Using **enter** and **exit** is mandatory when drawing paths with **draw** since pens only support the enter/exit subpath model (see Section 4.3).

Line and curve elements end with **to**. Unless an **arc** or **curve** directive precedes it, each **to** appends a line to the current subpath. In addition, to make path specification look more natural, the semicolon between statements is optional between subpath elements. A line can thus be stroked with

```
stroke from (x0, y0) to (x1, y1) end
```

and a rectangle be filled with either

```
fill from (0, 0) to (w, 0) to (w, h) to (0, h) close end
```

or

```
fill
  enter(0, -h) at (0, 0);
  to (w, 0) to (w, h) to (0, h) to (0, 0);
  exit(w, 0)
end
```

or simply using the predefined **rect** function

```
fill rect(0, 0, w, h) end
```

By inserting **curve** or **arc** and additional coordinate pairs, Bézier curves and arcs can be drawn instead of lines. A special **corner** element that draws a rounded corner with a given radius is also available.

Within a path body, character outlines can be drawn with **text**. In most situations, however, the **label** statement is more convenient. **label** always

fills its label text with the current fill color, but offers more options for placing its text relative to an anchor point. The following example draws one label to the upper left of a reference point and another centered inside a rectangle (*x*, *y*, *w*, *h*).

```
label "first" to upper left of (rx, ry);
label "second" over (x+w/2, y+h/2);
```

The **with** statement allows changing the current stroke and fill attributes or setting a new current pen. These changes apply to all statements in the **with** statement's body and are undone when **end** is reached. For example, the following line is drawn in blue, with a width of three units, and with a simple dash pattern with on and off lengths equal to five units:

```
with color=colors.blue, width=3, dash=vector(5, 5) do
  stroke from (0, 0) to (100, 0) end
end
```

To be consistent with the Gfx model, attribute changes are not allowed within path bodies. However, transformations to the current coordinate system are perfectly feasible. The corresponding functions are called **translate**, **scale**, **rotate**, and **concat**. They are implemented as standard functions and do not need extra syntax. However, to restore the current transformation matrix after a series of transformations, a statement sequence can be embedded between **savectm** and **restore**. Similarly, the current clip area can be saved and later restored by placing **clip** directives between **saveclip** and **restore**. Graphical attributes are already saved automatically when they are changed using **with** and need no additional **save**/**restore** pair.

6.1.7 Packages

It is easy to extend the set of built-in procedures: it suffices to add a binding that associates a symbol with a new primitive procedure to the environment where all global definitions are stored. However, when these new primitives are implemented in a separate Oberon module, a Vinci program that relies on them must be able to request the Vinci interpreter to load that module. Similarly, useful functions may be implemented in Vinci instead of natively in Oberon. Unless Vinci allows programs to import other Vinci source files, the corresponding function definitions must be repeated in each input file.

To deal with both issues, a Vinci program may contain an **import** directive as its first statement, listing the names of all *packages* that it requires. For each imported package, Vinci first tries to open a file whose name is taken to be the package name followed by '.Pack'. If it finds such a file, it parses the definitions it finds therein and makes them accessible as **P.N**, where **P** stands for the package name and **N** for the name of the defined identifier. It is an error for the imported file to contain any statements except **import**, **const**, **define**, and **module**, where **module** asks Vinci to load an Oberon module. If no package file is found, Vinci consults the Oberon registry and searches a module name for the given package name. If this fails too, it prepends "Vinci" to the package name and attempts to load the module with the corresponding name. Dynamically loaded modules can add new packages to Vinci's package pool or augment existing ones with new functions.

Although Vinci's import mechanism allows extending the set of known Vinci definitions with little effort, it is far from being as sophisticated as for example the import mechanism of an Oberon compiler. For instance, there is no version control whatsoever. Furthermore, there is no check whether an input file only accesses the packages it has imported. If packages have been defined by previously executed programs, a program can access these packages without listing them in its import directive. For example, the Vinci interpreter itself already defines a few basic packages that any program can access without explicitly importing them. Still, Vinci's package scheme achieves its goal of providing Vinci with a generic extension mechanism.

6.2 Integration Aspects

To be of practical use, Vinci descriptions must be displayed as components within an existing environment (unless we restrict its use to pure page descriptions, similar to Postscript programs). Examples of such environments are the Oberon display space or Leonardo figures. On the other hand, integration of other graphical components within Vinci programs should be possible as well. What we desire is the ability to *mutually integrate* Vinci descriptions and Leonardo shapes.

When Vinci descriptions are integrated within Leonardo figures, they should appear to a user like any other shape. This means that users must be able to select them, inspect their attributes, and transform them using

the standard tools of the Leonardo editor. In fact, our solution goes beyond simple embedding and supports script parameterization and fine-grained manipulation of script geometry (see Section 6.2.3).

6.2.1 Shapes inside Vinci Descriptions

Leonardo shapes are seamlessly integrated within Vinci because they are standard Oberon objects. We have augmented the standard Scheme library by functions for creating objects from a generator string, for accessing and copying public objects, and for reading and writing attribute and link values. The following example creates an ellipse shape and changes its dimensions by setting the corresponding fields of its local coordinate matrix. Following that, it allocates a new dasher pen and installs it as the ellipse shape's pen. Finally, the shape is rendered on the graphics context onto which the script should be rendered by passing that context to the library function **shape**.

```
const s = new("LeoBasic.NewEllipse");   -- create shape
'set-attr!'(s, "M00", 100.0, "M11", 80.0);   -- set ellipse radii
const pen = new("LeoPens.NewDasher");   -- create dasher pen
'set-link!'(s, "Pen", pen);   -- attach pen to shape
shape(s);  -- render shape
```

The disadvantage of this example is that it allocates a new shape each time it is executed. Often used shape objects should thus be stored in public libraries and retrieved from there, as illustrated in the following example.

```
translate(x, y);   -- move coordinate origin to desired location
shape(pubobj("MyShapeLibrary", "MyShape"))   -- render public object
```

In this manner, any graphical component that can be rendered on a Gfx context and that is available as an Oberon object can be integrated within Vinci. All that is needed is a new package function like **shape** that renders such objects.

6.2.2 Vinci Gadgets

Gadgets for displaying Vinci descriptions have the same purpose as Leonardo gadgets have for figures, which is to integrate graphics within the Gadgets framework. The **VinciGadgets** module exports a gadget frame type which accepts a text object that contains Vinci source code as its model.

A corresponding document type is implemented in module **VinciDocs**. Although Vinci descriptions are rarely used in this stand-alone form, Vinci documents provide a suitable testbed for experimenting with the language and for debugging Vinci programs. To simplify the edit/test cycle, Vinci documents have a button in their menu bar for quickly switching between graphical and textual representation of a program.

6.2.3 Vinci Shapes

Integrating Vinci descriptions within Leonardo at least in principle is easy: as Vinci gadgets integrate them in the Gadgets framework, Vinci shapes integrate them within a figure. A basic Vinci shape merely needs to manage a rectangular box within which its description is embedded. The box provides both a coordinate system and a bounding box for its description, and users may interactively transform the box like any other shape. However, this simple scheme does not allow users to customize embedded scripts. If some part of the Vinci program paints a blue line, the only way to have it draw a green line is to edit the program's source code. This is not only tedious when several variants of the same description are needed, it also keeps users who do not know Vinci sufficiently well from reusing a description that somebody else has written.

Our approach for integrating Vinci descriptions within Leonardo therefore follows a slightly different path. A shape description consists of a series of definitions, similar to a package file. When the handler of a Vinci shape receives a standard **RenderMsg**, it delegates rendering to a Vinci function with the special name **render** if such a function has been defined in the corresponding Vinci source code. Similar functions are used for locating shapes (**locate**), for calculating their bounding box (**bbox**), and for calculating the local coordinate system within which **render** and **locate** are executed (**matrix**). In addition, a description may include directives that declare defined values to be freely customizable parameters. The following example contains a minimal shape description to draw a colored rectangle.

```
import colors, rectangles, shapes;

const x = 100, y = 100, w = 300, h = 200;  -- default position and dimensions
const col = colors.blue;  -- default color
```

```
-- render function
define render as
  with color=col do
    fill rect(x, y, x+w, y+h) end
  end
end render;

-- bounding box function
define bbox as
  rectangles.init(x, y, x+w, y+h)
end bbox;

-- locate function
define locate (llx, lly, urx, ury) as
  rectangles.overlap(bbox(), rectangles.init(llx, lly, urx, ury))
end locate;

-- exported shape parameters
shapes.real("x", "X Position");
shapes.real("y", "Y Position");
shapes.real("w", "Width", 10.0);
shapes.real("h", "Height", 10.0);
shapes.color("col", "Color");
```

Of special interest are the last five statements. These declare that the defined names **x**, **y**, **w**, and **h** hold real numbers as their values and that **col** holds a color value. Besides, they associate each parameter with a descriptive name. Width and height parameters **w** and **h** are declared to have a minimal value of 10. Although these declarations are not meaningful to the Vinci shapes themselves (they are more interested in the **render**, **bbox**, and **locate** functions), they allow Leonardo to build a customized inspector panel when a selected script shape is inspected. The Oberon procedure that is registered with Leonardo for creating a custom editor panel for a Vinci shape can determine which definitions are considered parameters and what types they have. Figure 6.1 shows a Leonardo frame that displays the above sample description and the corresponding editor panel.

Coordinate System. A distinctive lack of the above example is that the shape cannot be transformed using direct manipulation; it can only be moved and resized by adjusting its parameters from an inspector panel. One solution would be to add other special functions for handling transform

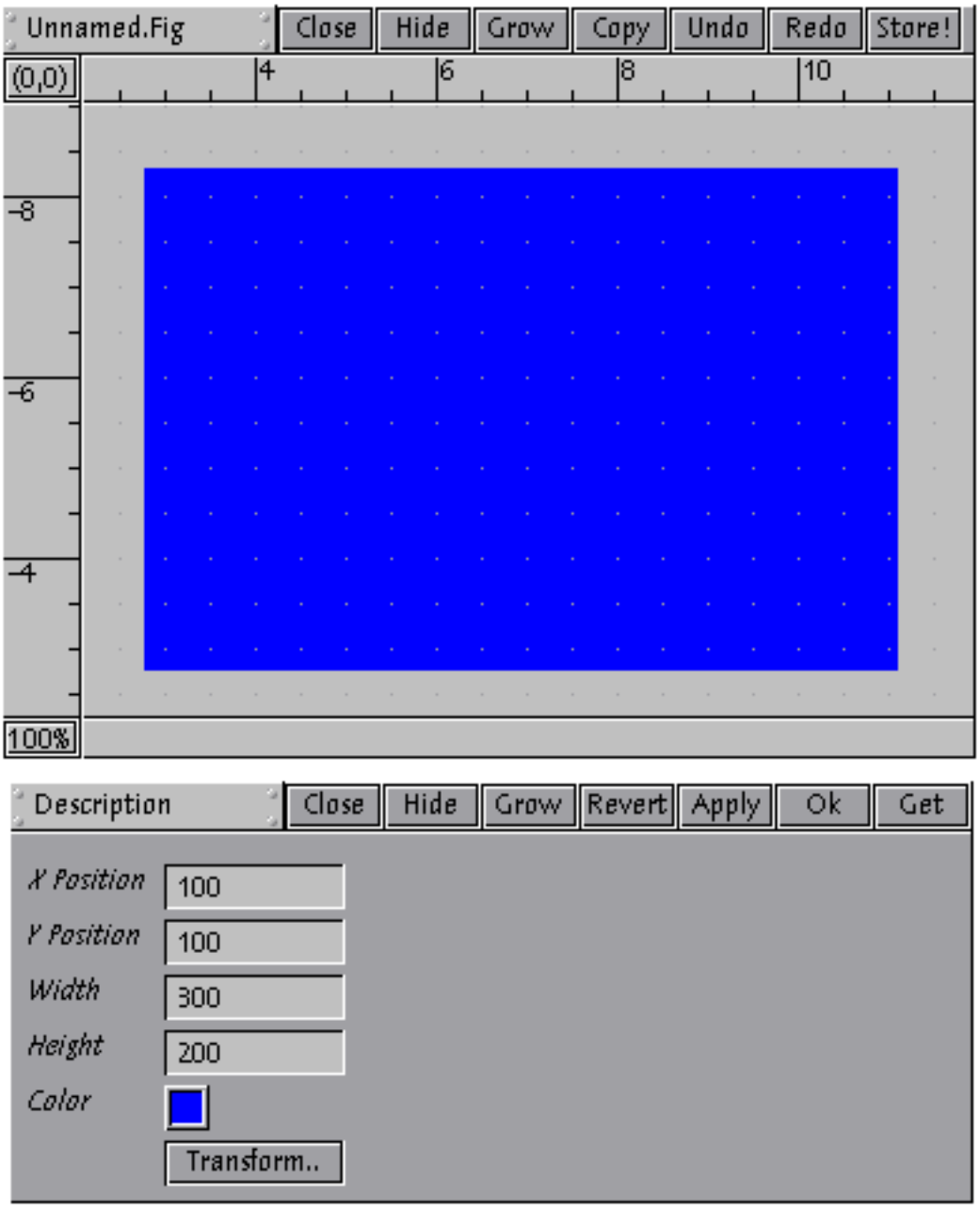

Figure 6.1: Vinci shape and corresponding editor panel

requests and for returning location specific transformation matrices. Such a **transform** function would have to apply the matrix that it receives to the geometric parameters of the description and assign the resulting values to them with undoable actions. Although feasible, this approach could only transform the shape as a whole.

In the approach we have chosen, shape descriptions can define links to other shape objects and declare them to be components of the description. These shape links become components of the Vinci shape, which is therefore derived from **Leonardo.Container**. The scenario that we described earlier in this section, where a description is embedded within a rectangular frame, is achieved by replacing **x**, **y**, **w**, and **h** in our example with a reference to a Vinci *frame*, as shown in the following program.

```
import colors, rectangles, shapes;

const w = 300, h = 200;   -- dimensions of the rectangle
const frame = shapes.frame(w, h);   -- component frame
const col = colors.color("blue");

define render as
  with color=col do
    fill rect(0, 0, w, h) end   -- frame defines coordinate origin
  end
end render;

define matrix as
  shapes.matrix(frame)   -- frame defines local coordinate system
end matrix;

shapes.component("frame");
shapes.color("col", "Color");
```

We observe that **x** and **y** have been completely discarded. **w** and **h** are still present, but are no longer declared to be parameters. Instead, they are reduced to symbolic constants. However, a frame component with the same dimensions as the painted rectangle has been introduced, along with a **matrix** function that returns the frame's local coordinate system, which is retrieved with **shapes.matrix**. **locate** and **bbox** are no longer necessary because the shape by default delegates these tasks to its components if no matching functions are found. When a user drags the component frame around or otherwise transforms it, the frame adjusts its local coordinate

system within which the shape's **render** function is evaluated. The behavior of the integrated Vinci description is thus very close to that of a natively implemented shape extension.

Feasible components of shape descriptions are not restricted to a single frame shape. If a description is based on a set of point links, each point can be transformed independently of all others, allowing fine-grained modifications to the shape's geometry. Furthermore, these points can be consumed within other shapes, subjecting the corresponding Vinci shape to Leonardo's constraint mechanism. An example of this is a description that draws a line between two points and centers a caption over the line which displays the current line length. When either point is moved, the displayed line length is automatically adjusted. Figure 6.2 displays several such lines. Appendix A contains the commented source code for this particular Vinci shape.

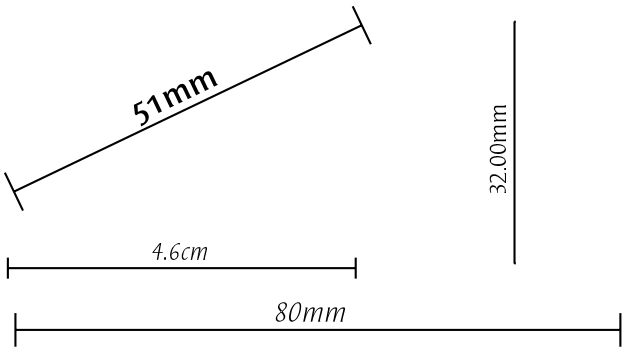

Figure 6.2: Vinci line shapes which measure their own length

6.3 Summary

As a second application for Gfx and the Leonardo shape framework, we have presented Vinci, a graphical description language. Vinci is an interpreted functional language that couples custom syntax with an underlying Scheme engine. Vinci offers syntactical support for specifying paths, placing labels, and modifying graphical attributes. Besides, it implements a package concept that lets it accommodate future extensions.

Because the underlying Scheme interpreter includes a data type whose

instances represent Oberon objects, Oberon objects are ordinary expression values that can be passed to and returned from functions. As a result, arbitrary objects that a library function is able to render on a Gfx context can be rendered by Vinci programs. Pen and image objects are thus available within Vinci as well.

By conforming to a few simple conventions, Vinci descriptions can be integrated as shapes in Leonardo. Vinci shapes delegate tasks such as rendering or locating themselves to correspondingly named Vinci functions. Furthermore, Vinci values can be exported as parameters; users are able to inspect and modify these within Leonardo like the properties of any other shape. Besides, Vinci shapes may rely on component shapes to define a local coordinate system for them. To interactive users, these shapes look like ordinary components and allow them to reshape the geometry of a Vinci shape in a fine-grained manner.

CHAPTER 7

Conclusions

7.1 Evaluation

As we draw conclusions of what we have achieved, we do not wish to merely recapitulate individual contributions here since each previous chapter already includes a summary of relevant points. Instead, we would rather attempt to view them in a bigger context.

7.1.1 Graphics API

Gfx, our graphics programming interface, is evolutionary rather than revolutionary. Like many older API it uses an immediate rendering model and a large interface that cannot be extended. For the following reasons, we nevertheless regard it as a successful combination of graphical functionality and the principles of extensible object-oriented software design:

- It is modular and consists of several subsystems that can be reused on their own. In fact, the imaging subsystem, although originally developed as a part of Gfx, is available as a separate package and no longer part of the regular Gfx distribution. Moreover, concrete context implementations are independent of each other; an application only needs to import those that it actually needs.
- Its interface is general and complete. Gfx integrates many advanced concepts in a natural way; consider its seamless integration of character outlines with general paths or its support for early and late path rendering.
- It can be customized. Not only can support for new image and font file

formats be added; the potential number of available context types is unlimited.

– Its abstract context interface and the exchangeability of concrete context implementations allow graphical content to be integrated in various environments. For example, Leonardo figures and Vinci descriptions can be embedded within a graphical user interface such as the Gadgets component framework. Furthermore, Vinci descriptions can draw Leonardo shapes and Leonardo can integrate Vinci descriptions although Leonardo and Vinci are otherwise independent applications.

Gfx has also proved itself in practice. In addition to Leonardo and Vinci, the applications we have described in Chapters 5 and 6, a couple of student projects have used it to implement an experimental PDF viewer [15] and a viewer for TEX-generated DVI files [26]. Besides, Gfx has been used as a tool in in undergraduate student courses at ETH Zürich and in Belgium.

7.1.2 Shape Framework

As a model for the Leonardo editor, our shape framework is a success, currently including about a dozen shape types and more than half a dozen pen types. However, it has not been used within other environments, which makes an evaluation of its qualities as a reusable framework difficult. Still, we feel confident that – due to its use of an open message interface – its potential for customization within other applications is considerable.

By introducing abstract pens, we have been able to decouple paths and similar shapes from the fixed set of graphical attributes that Gfx (and other graphics interfaces) provide. New pen variations can either be created by specialization, meaning that new concrete extensions of the abstract pen type are implemented, or by aggregation, meaning that instances of existing pens are connected in new master and slave constellations.

7.1.3 Leonardo

Leonardo is an application that uses the MVC paradigm to display and edit an extensible graphical model. This model has the unique property that its shapes represent graphical information and know how to render themselves on abstract graphical contexts. Therefore, a single generic view type can

display arbitrary figures. This is generally not possible with other kinds of applications where view components have to be adapted or extended when new kinds of objects are added to their model.

Leonardo is also an application that adapts to a user's demands. On the lower end of the scale, to load a single module suffices for viewing arbitrary figures and embedding them as (passive) graphical components within arbitrary documents. On the upper end of the scale, several graphical documents can be edited at once with a potentially unlimited set of interactive tools and operations. Important aspects are that components can arbitrarily switch between passive and active roles, that additional code is only loaded the first time it is needed, and that new shapes, tools, and editor panels can easily be integrated within Leonardo, even at run-time.

7.1.4 Vinci

By interpreting textual descriptions, Vinci provides users with an alternative method for creating graphics. Because these descriptions may include precise numeric values, arbitrary calculations, and repetitive structures, figures that would be difficult to construct interactively are easily expressed in Vinci.

Although Vinci descriptions can be treated as stand-alone documents, they are of particular interest when they are integrated as shapes within Leonardo figures. Users can manipulate these Vinci shapes by inspecting them inside customized editor panels or by interactively transforming their component shapes. Thus, even users that would not write or edit a Vinci program themselves are able to use Vinci shapes in their drawings. Experienced programmers may find Vinci descriptions beneficial for implementing simple shapes that are used repeatedly but do not justify the effort that is needed to implement them as a proper shape type.

Because Vinci is a true programming language which allows functions to be defined and built-in names to be hidden within local scopes, interpreting a Vinci description is more complex than parsing purely structural descriptions such as SVG or PDF files, which can directly be translated to an object structure. For example, executing a Vinci description may result in an endless recursion or in a run-time exception (e. g. a division by zero). Vinci may therefore be less suitable than those other formats for distributing graphical content (for example in browsable documents on the Internet). However, with Microsoft's active scripting technology [55] it is possible to

embed Vinci descriptions within a HTML page and have an ActiveX control in the same page interpret and display this description [91].

7.2 Future Perspectives

The following paragraphs discuss aspects that we consider worthy of further exploration.

7.2.1 Transparency

We have deliberately restricted Gfx to only support opaque color for painting graphical objects, except for images, where transparency is essential for representing character bitmaps. The reason for this limitation is that some output devices, especially printers, cannot correctly handle partly transparent objects.

Nevertheless, the current trend to move content to the Internet in preference to publishing it in printed form will probably increase the number of documents that are primarily designed for online viewing. Web designers are accustomed to having transparency effects at their disposal with bitmap graphics. They thus demand them for vector graphics as well, which is why Web-oriented approaches such as SVG support transparency.

It would not be difficult to extend the Gfx context interface by fields and procedures to manage transparency and blend functions. However, since not all intended output devices support them, we would also have to include procedures which let clients inquire about whether a context supports this particular feature and to choose an alternative drawing method if not.

As an additional advantage of having transparency support available, Gfx could be augmented by procedures for rendering anti-aliased primitives. Thus, the rendering quality of Gfx could be further improved.

7.2.2 Context Extensions

So far, Gfx contexts have almost exclusively been used to implement device-specific output drivers. However, the abstract context interface could also be used for other purposes.

If the current **GfxDisplay** context, which uses the generic Oberon display driver for output, were replaced by a hardware-specific context extension, Gfx could utilize hardware-accelerated raster operations. Hardware acceleration could be used for drawing hairlines, for clipping, for painting bitmaps, and for drawing transformed images as texture maps. Thus, performance for specific operations could drastically be improved.

Another class of concrete context extensions could convert graphical input to equivalent Leonardo figures or to standard file formats (such as PDF or SVG). An existing example of such a context is **GfxPS.Context**, which converts its input to a corresponding Postscript file.

7.2.3 Animation

Apart from the addition of new shape and pen types, an interesting extension of the shape framework would be to augment it by animation capabilities, similar to those found in SVG. If animation could be added by merely introducing some kind of animator shapes and new shape messages for synchronizing them with a global timer, the validity of our design could be further confirmed.

APPENDIX A

Sample Vinci Script

The following program implements a Vinci shape that integrates lines within a figure which measure and display their own length. The shape definition declares two points as its components. These points not only define the end points of the corresponding line, but also a rotated coordinate system within which the text label that displays the length of the line is drawn.

```
-- import required packages
import colors, matrix, rectangles, shapes;

-- name to be displayed in inspector panels
shapes.name("Distance Line");

-- unit scale factor (default unit is 1/91.44 inch)
const unit = 91.44/25.4, unitname = "mm";
shapes.real("unit", "Unit", 0.001);
shapes.string("unitname", "Unit Name");

-- line end points
const p1 = shapes.point(0, 0), p2 = shapes.point(100*unit, 0);
shapes.component("p1");
shapes.component("p2");

-- graphical parameters
const offset = 5;   -- length of "ticks" at line ends
shapes.real("offset", "Offset");

const fontname = "Oberon-Italic", fontsize = 12;   -- label font
shapes.string("fontname", "Font Name");
shapes.integer("fontsize", "Font Size");
```

```
const col = colors.black;   -- color with which to paint
shapes.color("col", "Color");

const digits = 0;   -- number of digits after decimal point
shapes.integer("digits", "Digits", 0, 5);

-- return shape bounding box
define bbox as
  let
    x1 = attr(p1, "X"), y1 = attr(p1, "Y"),
    dx = attr(p2, "X") - x1, dy = attr(p2, "Y") - y1,
    mat = matrix.rotate(matrix.init(1, 0, 0, 1, x1, y1), atan(dy, dx)),
    box = rectangles.init(0, 0, sqrt(dx*dx + dy*dy), offset + 1.5*fontsize)
  in
    rectangles.apply(box, mat)
  end
end bbox;

-- draw only line when shape is being interactively dragged
define drag as
  stroke
    from (attr(p1, "X"), attr(p1, "Y")) to (attr(p2, "X"), attr(p2, "Y"))
  end
end drag;

-- draw shape including label
define render as
  let
    x1 = attr(p1, "X"), y1 = attr(p1, "Y"),
    dx = attr(p2, "X") - x1, dy = attr(p2, "Y") - y1,
    len = sqrt(dx*dx + dy*dy)
  in
    -- establish local coordinate system
    translate(x1, y1);
    rotate(atan(dy, dx));
    translate(0, offset);
    with font = fontname, size = fontsize, color = col do
      stroke
        from (0, 0) to (len, 0);   -- line between points
        from (0, -offset) to (0, offset);   -- tick marks at end points
        from (len, -offset) to (len, offset)
      end;
      label realtostrfix(len/unit, digits) ++ unitname above (len/2, 0)
    end
  end
```

```
end render;

-- return if supplied rectangle intersects shape
define locate (llx, lly, urx, ury) as
  let
    x1 = attr(p1, "X"), y1 = attr(p1, "Y"),
    dx = attr(p2, "X") - x1, dy = attr(p2, "Y") - y1,
    mat = matrix.rotate(matrix.init(1, 0, 0, 1, 0, 0), -atan(dy,
    dx)),
    r = rectangles.apply(
      rectangles.init(llx - x1, lly - y1, urx - x1, ury - y1), mat
    );
  in
    rectangles.overlap?(r,
      rectangles.init(0, 0, sqrt(dx*dx + dy*dy), offset + 1.5*fontsize)
    )
  end
end locate;

-- various functions for converting reals to strings
-- with a fixed number of digits after the decimal point
-- (could be made into a library function)
define roundtostr (x) as
  'number->string'('inexact->exact'(round(x)))
end roundtostr;

define realtostrfix (x, d) as
  if d = 0 then roundtostr(x)
  else
    let t = tento(d), x = round(t * x)/t in
      roundtostr(x) ++ "." ++
        fractostr('make-string'(d), x - truncate(x), 0, d - 1)
    end
  end
end realtostrfix;

define fractostr (str, x, pos, d) as
  -- recursively add next digit to string
  if pos = d then
    string-set!'(str, pos, tochar(round(10*x)));
    str   -- return value is str
  else
    let y = 10*x, i = truncate(y) in
      'string-set!'(str, pos, tochar(i));
      fractostr(str, y - i, pos + 1, d)
```

```
      end
    end
end fractostr;

define tochar (x) as
  'integer->char'(48 + 'inexact->exact'(x))
end tochar;

define tento (n) as
  if n = 0 then 1
  else 10*tento(n - 1)
  end
end tento;
```

APPENDIX B

Vinci Syntax

```
Program = ["import" ident {"," ident} ";"] Seq.

Seq = Stat {";" Stat}.
Stat = [Const|Define|With|Save|Draw|Label|DrawStat].

Const = "const" Assign.
Define = "define" ident ["(" [ident {"," ident}] ")"] "as" Seq "end" ident.
Let = "let" Assign "in" Seq "end".
If = "if" Expr "then" Seq {"elsif" Expr "then" Seq} ["else" Seq] "end".
Repeat = "repeat" Expr "times" Seq "end".
For = "for" ident "=" Expr "to" Expr ["by" Expr] "do" Seq "end".
With = "with" Assign "do" Seq "end".
Save = ("saveclip"|"savectm") Seq "restore".
Draw = ("stroke"|"fill"|"clip"|"record"|"draw") DrawSeq "end".
Label = "label" Expr Locator ["rotated" Expr].
Assign = ident "=" Expr {"," ident "=" Expr}.

Locator = ("at"|"over"|"above"|"below"|
  ["to" ["lower"|"upper"]] ("left"|"right") "of") Coord.
DrawSeq = DrawStat {";" DrawStat}.
DrawStat = [If|Repeat|For|Let|Elements|Expr].
Elements = {"from" Coord|Enter|Line|Arc|Curve|Corner|"close"|Exit|Text}.
Enter = "enter" Coord "at" Coord.
Line = "to" Coord.
Arc = "arc" Coord ["," Coord ["," Coord]] "to" Coord.
Curve = "curve" Coord ["," Coord] "to" Coord.
Corner = "corner" Coord "," Expr "to" Coord.
Exit = "exit" Coord.
Coord = "(" Expr "," Expr ")".
Text = "text" Expr ["at" Coord].
```

```
Expr = SimpleExpr [relop SimpleExpr].
SimpleExpr = ["+"|"-"] Term {addop Term}.
Term = Factor {mulop Factor}.
Factor = number|string|"(" Expr ")"|Qualident|Call|"~" Factor.
Call = Qualident "(" [Expr "," Expr] ")".
Qualident = [ident "."] ident.
relop = "="|"#"|"<"|"<="|">"|">=".
addop = "+"|"-"|"++"|"or".
mulop = "*"|"/"|"&"|"div"|"mod".
```

APPENDIX C

Module and Performance Statistics

C.1 Module Sizes

To give an impression of the size of the various software packages that are described in this thesis, tables C.1 up to C.8 list the sizes of involved Oberon modules. For each module, its name, its function, its size in number of statements, and its size in number of lines of source code are given. The number of statements is calculated by the Analyzer tool that is part of Oberon System 3. Few statements per lines of code are typical for modules that contain many definitions.

For some packages, we separate essential *core* modules from optional *extensions*. Note that at least one extension module is usually necessary to use the generic core modules in practice.

Module	Function	Statements	Lines
Colors	Color model and inverse lookup	552	575
ColorGadgets	Gadgets for displaying and choosing color	1022	1186
Images	Image type and image processing	1819	2478
ImageGadgets	Image viewer gadgets	333	405
ImageDocs	Image documents	70	111
Total		3796	4755

Table C.1: Core modules of the Colors and Images packages

Module	Function	Statements	Lines
PictImages	Oberon picture images	316	406
BMPImages	Windows bitmap image loader	358	331
JPEGImages	JPEG image loader	2517	3931
GIFImages	GIF image loader	216	254
Total		3407	4922

Table C.2: Extension modules implementing various image formats

Module	Function	Statements	Lines
GfxMatrix	Affine matrices	217	345
GfxImages	Affine image transformations, filters	792	781
GfxPaths	Path structure and operations	1090	1399
GfxRegions	Region structure and shape algebra	769	1245
GfxFonts0	Platform specific font file directories	56	77
GfxFonts	Oberon bitmap and outline fonts	919	1117
Gfx	Context interface and default methods	629	1349
GfxRaster	Abstract raster contexts	1226	1343
Total		5698	7656

Table C.3: Gfx core modules

Module	Function	Statements	Lines
GfxBuffer	Raster contexts rendering to image buffers	150	206
GfxDisplay	Raster contexts rendering to screen	470	646
GfxPrinter	Oberon printer contexts	307	351
GfxPS	Contexts writing to Postscript files	2030	2036
GfxOType	Bridge between Gfx and OpenType fonts	143	198
GfxPKFonts	PK (TEX) font loader	404	477
Total		3504	3914

Table C.4: Gfx context and font extensions

Module	Function	Statements	Lines
Leonardo	Shape framework with shapes and figures	1322	1884
LeoFrames	View frames (passive)	339	460
LeoTools	Controller frames (active) and tools	1849	2188
LeoDocs	Figure documents	203	307
LeoPanels	Editor panels and standard commands	1212	1560
LeoLists	Shape list gadgets	589	682
Total		5514	7081

Table C.5: Leonardo core modules

Module	Function	Statements	Lines
LeoPens	Stroker, filler, dasher, and forker pens	908	1216
LeoOutliners	Outliner and arrow pens	584	719
LeoPaths	Paths, points, lines, arcs, and Bézier curves	2465	2929
LeoSplines	Natural splines	299	430
LeoCaptions	Text labels	462	574
LeoBasic	Rectangles, ellipses, and groups	624	799
LeoImages	Image shapes	231	281
Total		5573	6948

Table C.6: Leonardo model extensions

Module	Function	Statements	Lines
LeoPenEditors	Pen inspectors	951	1263
LeoPathEditors	Paths, points, and segment tools and inspectors	611	755
LeoSplineEditors	Natural spline tool	101	134
LeoCaptionEditors	Text tool and inspectors	323	438
LeoBasicEditors	Basic shape tools and inspectors	343	472
LeoImageEditors	Image tool and inspectors	119	184
Total		2448	3246

Table C.7: Leonardo editor extensions

Module	Function	Statements	Lines
Scheme	Scheme engine	2302	2603
SchemeOps	Scheme library primitives	745	788
Vinci	Vinci parser and primitives	1848	1732
VinciGadgets	Gadgets for displaying scripts	251	339
VinciDocs	Script documents	245	314
VinciPens	Pen package primitives	234	188
VinciShapes	Shapes package, Vinci and frame shapes	991	1047
VinciEditors	Vinci shape tool and inspectors	286	322
Total		6902	7333

Table C.8: Vinci modules

C.2 Performance Evaluation

Since Gfx is currently the only high-level graphics API that runs on top of the Oberon system, a comparison with other graphics interfaces is not practical. We thus restrict our evaluation to the following comparisons: early rendering versus late rendering, Gfx versus the standard Oberon library, and built-in pixel formats versus custom pixel formats. All tests were performed with Native Oberon running on an Intel-compatible processor at 300MHz and were repeated at least five times.

Early Rendering vs. Late Rendering. We compare early and late rendering with four small benchmarks. These draw a set of 1000 random lines (*lines* benchmark), a line sequence with 1000 corners (*polyline* benchmark), a set of 1000 random rectangles (*rectangles* benchmark), and a set of text labels in different font sizes from 8 to 24 (*glyphs* benchmark). The results of these tests are listed in Table C.9.

Benchmark	Late	Early	Relative
lines	137ms	132ms	- 3.92%
polyline	125ms	124ms	- 0.48%
rectangles	924ms	301ms	-67.42%
glyphs	13ms	1ms	-89.39%

Table C.9: Early rendering vs. late rendering

The first two benchmarks show that – for thin lines – early rendering and late rendering are about equal. Only when rendering filled rectangles or when filling glyphs does early rendering have a significant advantage. In the *rectangle* benchmark, early rendering profits from an optimized method that renders axis-aligned rectangles, whereas late rendering must reconstruct rectangles from their edges. The *glyph* benchmark validates our assertion that glyphs can be rendered faster if corresponding bitmaps are painted than if their outlines are filled. (With early rendering, glyphs are filled instantly, whereas with late rendering, their outlines are appended to the current path.)

Gfx vs. Oberon. To compare Gfx to the standard module **Display3** of Oberon System 3, we repeat the *lines*, *rectangles*, and *glyphs* benchmarks from above. This time, we compare the performance of early rendering with that of the corresponding **Display3** procedures. In the first group of tests, the whole area where rendering takes place is visible. In the second group, only about four percent of the whole area is visible. This lets us compare Gfx and **Display3** when clipping comes into play. Table C.10 shows the corresponding results.

Benchmark	**Display3**	Gfx	Relative
lines	76ms	135ms	+77.75%
rectangles	294ms	297ms	+ 0.95%
glyphs	<1ms	1ms	+45.37%
lines (clipped)	87ms	100ms	+14.42%
rectangles (clipped)	8ms	10ms	+13.64%
glyphs (clipped)	< 1ms	<1ms	-19.55%

Table C.10: Gfx vs. Oberon

As expected, the overheads of the Gfx interface – primarily due to wrapper procedures and application of the current transformation – make it fall behind the standard **Display3** module, especially when rendering lines. At least Gfx seems to implement clipping better than **Display3**, which is slower in the clipped *lines* benchmark than in the unclipped case although it only has to draw a small percentage of all pixels. Similarly, Gfx can even overtake **Display3** in the clipped *glyphs* benchmark. (Although absolute execution times are small, the speedup of about 19 percent was consistent in several runs.)

Built-in Pixel Formats vs. Custom Pixel Formats. For the last group of benchmarks, we filled a set of random rectangles in an image with arbitrary pixel values. The format of these images was either one of the built-in pixel formats **D8**, **BGR565**, **BGR888**, and **BGRA8888** or a corresponding custom format that relied solely on pack and unpack procedures. The results of the corresponding tests are shown in Table C.11. Obviously, speed-critical imaging applications should never rely on custom pixel formats and convert their images to one of the built-in formats whenever possible.

Format	Built-in	Custom	Relative
D8	16ms	965ms	+5733%
BGR565	26ms	490ms	+1750%
BGR888	40ms	592ms	+1348%
BGRA8888	61ms	578ms	+844%

Table C.11: Built-in vs. custom pixel formats

Bibliography

[1] H. Abelson and G.J. Sussman with J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, second edition, 1996.

[2] Adobe Systems, Inc. *Adobe Type 1 Font Format*. Addison–Wesley, 1990.

[3] Adobe Systems, Inc. *PostScript Language Reference Manual*. Addison-Wesley, second edition, 1990.

[4] Adobe Systems, Inc. *Portable Document Format Reference Manual*. Addison–Wesley, 1993.

[5] ANSI (American National Standards Institute). *American National Standard for Information Processing Systems–Computer Graphics–Graphical Kernel System (GKS) Functional Description*. ANSI, 1985. ANSI X3.124-1985.

[6] Apple Computer, Inc. *Inside Macintosh Volume 1*. Addison–Wesley, 1986.

[7] Apple Computer, Inc. *The TrueType Font Format Specification*. Apple Computer, Inc., 1990.

[8] Apple Computer, Inc. *Inside Macintosh: QuickDraw GX Graphics*. Addison–Wesley, 1994.

[9] P. Bézier. *Emploi des Machines à Commande Numérique*. Masson et Cie, 1970.

[10] P. Bézier. Mathematical and practical possibilities of UNISURF. In R. E. Barnhill and R. F. Riesenfeld, editors, *Computer Aided Geometric Design*. Academic Press, 1974.

[11] E. A. Bier and M. C. Stone. Snap-dragging. *Computer Graphics*, 20(4):233-240, 1986. SIGGRAPH'86 Proceedings.

[12] J. F. Blinn. A homogeneous formulation for lines in 3-space. In *SIGGRAPH*, pages 237–241, 1977.

[13] J. F. Blinn. Jim Blinn's corner: A trip down the graphics pipeline: Line clipping. *IEEE Computer Graphics and Applications*, 11(1):98–105, January 1991.

[14] A. Borning. *Thinglab – A Constraint Oriented Simulation Laboratory*. PhD thesis, Stanford University, July 1979.

[15] M. Bösiger. PDF-Viewer für Oberon. Semester project, ETH Zürich, Institut für Computersysteme, 1998.

[16] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[17] J. E. Bresenham. A linear algorithm for incremental digital display of circular arcs. *Communications of the ACM*, 20(2):100–106, February 1977.

[18] R. Bringhurst. *The Elements of Typographic Style*. Hartley & Marks, second edition, January 1997.

[19] F. C. Crow. The aliasing problem in computer-generated shaded images. *Communications of the ACM*, 20(11):799–805, November 1977.

[20] F. C. Crow. A comparison of antialiasing techniques. *Computer Graphics & Applications*, 1(1):40–48, January 1981.

[21] M. Cyrus and J. Beck. Generalized two- and three-dimensional clipping. *Computers and Graphics*, 3:23–37, 1978.

[22] J. Derungs. LayLa – eine Beschreibungssprache für Gadgets. Diploma thesis, ETH Zürich, Institut für Computersysteme, 1996.

[23] L. P. Deutsch. Design reuse and frameworks in the Smalltalk-80 system. In T.J. Biggerstaff and A.J. Perlis, editors, *Applications and Experience*, volume II of *Software Reusability*, pages 57–71. Addison–Wesley, 1989.

[24] N. Donato and R. Rocchetti. Techniques for manipulating arbitrary regions. In *Course Notes 11 for SIGGRAPH 88*, August 1988.

[25] D. W. Fellner and C. Helmberg. Robust rendering of general ellipses and elliptical arcs. *ACM Transactions on Graphics*, 12(3):251–276, July 1993.

[26] K. Fischer. Implementierung eines DVI-Viewers und eines Postscript Type-1 Font-Renderers für Oberon. Semester project, ETH Zürich, Institut für Computersysteme, 2000.

[27] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison–Wesley, 1990.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software*. Addison—Wesley, 1995.

[29] M. Gervautz and W. Purgathofer. A simple method for color quantization: Octree quantization. In A. S. Glassner, editor, *Graphics Gems*, pages 287—293. Academic Press Professional, 1990.

[30] A. S. Glassner. *Principles of Digital Image Synthesis*. Morgan-Kaufmann, 1995.

[31] M. L. Gleicher and A. Witkin. Differential manipulation. In T. Calvert, editor, *Proceedings of Graphics Interface 91*, pages 61—67, June 1991.

[32] M. L. Gleicher and A. Witkin. Drawing with constraints. *The Visual Computer*, 11(1):39—51, 1994.

[33] Graphics Standards Planning Committee. Status report of the graphics standards planning committee. *Computer Graphics*, 11, 1977.

[34] Graphics Standards Planning Committee. Status report of the graphics standards planning committee. *Computer Graphics*, 13(3), August 1979.

[35] M. Gross. Graphische Datenverarbeitung. Lecture Notes on Computer Graphics, ETH Zürich, 1999.

[36] J. Gutknecht. Oberon System 3: A vision of a future software technology. *Software—Concepts and Tools*, 15(1):26—33, 1994.

[37] J. Gutknecht. Do the fish really need remote control? A proposal for self-active objects in Oberon. In H. Mössenböck, editor, *Modular Programming Languages— Proceedings of the JMLC97, Linz, Austria*, pages 207—220. Springer, March 1997. Lecture Notes in Computer Science 1204.

[38] G. Hamlin, Jr. and C.W. Gear. Raster-scan hidden surface algorithm techniques. *Computer Graphics*, 11(2):206—213, 1977.

[39] V.J. Hardy. *Java 2D API Graphics*. Prentice—Hall, 1999.

[40] P. Heckbert. Color image quantization for frame buffer display. *Computer Graphics*, 16(3):297—307, July 1982.

[41] J. D. Hobby. Introduction to MetaPost. In *EuroTEX 92 Proceedings*, pages 21—36, September 1992.

[42] J. D. Hobby. A user's manual for MetaPost. Computing Science Technical Report no. 162, AT&T Bell Laboratories, 1992.

[43] ImageMagick Studio. ImageMagick – image conversion and manipulation software.
http://www.wizards.dupont.com/cristy/ImageMagick.html.

[44] International Organization for Standardization. The graphical kernel system (GKS). Technical Report 7942, ISO Geneva, 1985.

[45] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–25, 1988.

[46] D. E. Knuth. *The TEXbook*. Addison–Wesley, 1984.

[47] D. E. Knuth. *The Metafont Book*. Addison–Wesley, 1985.

[48] G. E. Krasner and S. T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August 1988.

[49] Y. Liang and B. Barsky. A new concept and method for line clipping. *ACM Transactions on Graphics*, 3(1):1–22, January 1984.

[50] W. Loeb. The Java 2D API. *Dr. Dobbs Journal of Software Tools*, 24(2):44, 46–49, February 1999.

[51] J. L. Marais. *Design and Implementation of a Component Architecture for Oberon*. PhD thesis, ETH Zürich, 1996. Diss. ETH No. 11697.

[52] E. A. Maxwell. *Methods of Plane Projective Geometry Based on the Use of General Homogeneous Coordinates*. Cambridge University Press, 1946.

[53] E. A. Maxwell. *General Homogeneous Coordinates in Space of Three Dimensions*. Cambridge University Press, 1951.

[54] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, 1960.

[55] Microsoft. Microsoft Platform SDK.
http://msdn.microsoft.com/downloads/sdks/platform/platform.asp.

[56] Microsoft. *Win32 Programmers Reference Volume 1*. Microsoft Press, 1993.

[57] H. Mössenböck. *Object-Oriented Programming in Oberon-2*. Springer, 1993.

[58] H. Mössenböck and N. Wirth. The programming language Oberon-2. *Structured Programming*, 12(4):179–195, 1991.

[59] G. Nelson. Juno, a constraint based graphics system. *Computer Graphics*, 19(3):235–243, 1985. SIGGRAPH'85 Proceedings.

[60] W. Newman and R. Sproull. *Principles of Interactive Computer Graphics*. McGraw–Hill, 1973.

[61] A. Nye. *Xlib Programming Manual Volume 1*. O'Reilly & Associates, 1988.

[62] E. Oswald. Drawing pens: An extensible approach to rendering vector graphics. In H. Mössenböck, editor, *Joint Modular Languages Conference (Short Presentations)*, pages 28–39, Linz, Austria, March 1997.

[63] E. Oswald. Leonardo: A framework for modeling and editing graphical components. In J. Gutknecht and Wolfgang Weck, editors, *Modular Programming Languages—Joint Modular Languages Conference, JMLC 2000*, pages 47–60, Zürich, Switzerland, September 2000. Springer. Lecture Notes in Computer Science 1897.

[64] PHIGS+ Committee. PHIGS+ functional description, revision 3.0. *Computer Graphics*, 22(3):125–218, July 1988.

[65] L. Piegl and W. Tiller. *The NURBS Book*. Springer, second edition, 1996.

[66] T. Porter and T. Duff. Compositing digital images. In *SIGGRAPH*, pages 253–259, 1984.

[67] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, second edition, 1992.

[68] J. Rees and W. Clinger (eds.). The revised^4 report on the algorithmic language Scheme. *Lisp Pointers*, 4(3), 1991.

[69] L. G. Roberts. Homogeneous matrix representations and manipulation of n-dimensional constructs. Technical Report 1405, Lincoln Laboratory, MIT, 1965. Document MS.

[70] B. Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, August 1983.

[71] B. Stamm. Algorithms for drawing thick lines and curves on raster devices. Technical Report 107, Department of Computer Science, ETH Zürich, 1989.

[72] B. Stamm. *A Hybrid Approach to Medium- and Low-Resolution Font-Scaling and its OOP Style Implementation*. PhD thesis, ETH Zürich, 1994. Diss. ETH No. 10884.

[73] B. Stroustrup. *The C++ Programming Language*. Addison–Wesley, third edition, 1997.

[74] Sun Microsystems, Inc. Java 2D™ API home page. http://java.sun.com/products/java-media/2D/.

[75] Sun Microsystems, Inc. Java 3D™ API home page. http://java.sun.com/products/java-media/3D/.

[76] Sun Microsystems, Inc. JavaBeans: The only component architecture for Java technology. http://java.sun.com/beans/.

[77] I. E. Sutherland. *Sketchpad: A Man-Machine Graphical Communication System*. PhD thesis, Massachusetts Institute of Technology, January 1963.

[78] I. E. Sutherland and G. W. Hodgman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, January 1974.

[79] C. Szyperski. *Insight ETHOS: On Object-Orientation in Operating Systems*. PhD thesis, ETH Zürich, 1992. Diss. ETH Nr. 9884.

[80] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, 1997.

[81] S. W. Thomas. Efficient inverse color map computation. In J. Arvo, editor, *Graphics Gems II*, pages 116–125, 528–535. Academic Press Professional, 1991.

[82] J. M. Vlissides and M. A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268, July 1990.

[83] J. Wernecke. *The Inventor Mentor*. Addison–Wesley, 1994.

[84] N. Wirth. The programming language Oberon. *Software – Practice and Experience*, 18(7):671–690, July 1988.

[85] N. Wirth and J. Gutknecht. The Oberon System. *Software – Practice and Experience*, 19(9):857–893, September 1989.

[86] N. Wirth and J. Gutknecht. *Project Oberon – The Design of an Operating System and Compiler*. Addison–Wesley, 1992.

[87] N. Wirth and M. Reiser. *Programming in Oberon – Steps Beyond Pascal and Modula*. Addison–Wesley, 1992.

[88] G. Wolberg. *Digital Image Warping*. IEEE Computer Society Press, 1990.

[89] WWW Consortium (W3C). Extensible markup language (XML). http://www.w3.org/XML/.

[90] WWW Consortium (W3C). W3C scalable vector graphics (SVG). http://www.w3.org/Graphics/SVG/, December 1999. Seventh working draft.

[91] E.J. Zeller. Personal Demonstration, 2000.

Curriculum Vitae

	Erich Oswald
March 19, 1969	Born in Winterthur, ZH citizen of Winterthur, ZH, and Richterswil, ZH son of Peter and Doris Elisabeth Oswald-Keller
1988	Matura Typus B (Kantonsschule Rychenberg, Winterthur)
1994	Dipl. Informatik-Ing. ETH
1994–2000	Teaching and research assistant at the Institute for Computer Systems at the Swiss Federal Institute of Technology (ETH), Zürich, in the research group of Prof. Jürg Gutknecht