20
Graphics Standards

Standards are useful in many aspects of everyday life. A notable example is the automobile. Early cars had controls whose placement and function were dictated by engineers, often without much thought of the driver. As car technology matured, a consensus slowly emerged, and virtually all current passenger cars are driven in the same way and have the same set of controls. The development of bicycles, motorcycles, and airplanes followed similar trends. In the early 1970s, computer graphics researchers, users, programmers, and manufacturers felt a similar need for standardization, with the result that CORE, the first graphics standard, was developed in the period 1977–1979. In 1984–85, the next standard, GKS, added three-dimensional capabilities, and at about the same time, its competitor, PHIGS, added more powerful three-dimensional functions. The year 1987 saw the introduction of the X-Windows system, followed quickly by the present standard, OpenGL. The term "application programming interface" (API) is often used to indicate these standards, because they behave more like libraries of routines and functions than like programming languages. It is as if the API is an interface to an existing programming language.

The last sections of this chapter describe several important standard graphics file formats.

20.1 GKS

GKS, the *Graphical Kernel System*, was an early attempt to create a graphics standard. GKS was an international effort, published in 1985 by the ISO (International Standards Organization) as standard #7942. It has also been adopted by ANSI (the American National Standards Institute) as their standard X3.124–1985 [ANSI 85]. Two detailed references are [Enderle et al. 87] and [Hopgood et al. 86]. Following is a short description of GKS, attempting to convey its "flavor" to the reader.

D. Salomon, *The Computer Graphics Manual*, Texts in Computer Science,
DOI 10.1007/978-0-85729-886-7_20, © Springer-Verlag London Limited 2011

GKS is used inside a program and its operation depends on the programming language. The idea is to write a program to calculate the individual components of an image, then employ GKS commands inside the program to actually draw these components. The original GKS definition was developed with Fortran 77 in mind, which is why the GKS commands have a Fortran "flavor."

GKS provides machine-independent notation to specify and manipulate images. An image is constructed from simple building blocks (primitives). GKS offers five primitives plus commands that specify additional attributes for the primitives. A primitive is similar to a procedure call in a typical programming language in the sense that when a primitive is used, the user has to specify the values of parameters. In addition to the parameters, the primitives have *attributes*, which are features that don't change very often. They are specified by additional GKS commands. An example of a primitive is text. Each time text is to be displayed, the user has to specify the text itself (the string of characters to be displayed) and its location as parameters, but the font and the text size are attributes, because we can expect several consecutive strings of text to have the same font and size. The five primitives are as follows:

1. *Polyline*: to draw a sequence of straight, connected line segments.
2. *Polymarker*: to draw a sequence of points, all with the same symbol.
3. *Fill area*: to fill a polygon with a pattern and draw it.
4. *Text*: to display text in different fonts, sizes, and orientations.
5. *Cell array*: to display an image in grayscale or color.

Polyline: The general format of this primitive is `POLYLINE(N, XPTS, YPTS)`, where `N` is the number of points to be connected with straight segments, and `XPTS` and `YPTS` are two arrays with the x and y coordinates of the points. Notice that the polyline consists of $N-1$ segments. The polyline is an important primitive since it can be used to draw curves by drawing a large number of short straight segments. Each polyline has attributes that are specified by means of the command

```
SET POLYLINE REPRESENTATION(WS,PLI,LT,LW,PLCI)
```

where `WS` is the platform id (`WS` stands for WorkStation) and `PLI` is the polyline index. The idea is to predefine several types of polylines, to be used on different platforms, and to assign each an identifying number (an index). For example, after executing the commands

```
SET POLYLINE REPRESENTATION(1,3,...solid....)
SET POLYLINE REPRESENTATION(2,3,...dashed...)
SET POLYLINE REPRESENTATION(3,3,...dotted...)
```

the program can draw a polyline of index 3, which will be plotted in either solid, dashed, or dotted style depending on whether the current platform has id 1, 2, or 3.

The `LT` parameter specifies the *line type*, which can be solid (`LT`=1), dashed (2), dotted (3), or dashed-dotted (4). Other values may be added, but they are nonstandard.

Parameter `LW` specifies the *linewidth scale factor*. This is a real quantity, giving the width of the polyline segments relative to the width of a standard line on the particular platform used (the width of a standard line is typically one pixel).

`PLCI` is the color parameter. Instead of specifying the color itself, this parameter is a pointer to an RGB color table.

After setting the different polyline representations at the start of the program, a particular representation is selected by the command `SET POLYLINE INDEX(N)`. This

command is normally followed by several polylines, following which, another `SET POLYLINE INDEX` may be used, to select another representation.

Polymarker: The general format of this primitive is `POLYMARKER(N, XPTS, YPTS)`, where `N` is the number of points to be drawn, and `XPTS` and `YPTS` are two arrays with the x and y coordinates of the points. The symbol actually drawn at each points is selected by the command

```
SET POLYMARKER REPRESENTATION(WS,INDX,MT,MS,PMCI)
```

where `WS` and `INDX` are the workstation and index numbers, and `PMCI` is the color specification. Parameter `MS` is the marker-size scale factor, a real number specifying the size of the marker relative to the standard marker size on the platform being used. The `MT` parameter specifies the actual marker symbol to be used. The five standard values of this parameter are

1 ".", 2 "+", 3 "*", 4 "0", and 5 "×",

and any platform may have its own nonstandard values.

After setting the different marker styles at the start of the program, a particular style is selected by the command `SET POLYMARKER INDEX(N)`.

Fill Area: The general format of this primitive is `FILL AREA(N, XPTS, YPTS)`. There is a `SET FILL AREA INDEX(N)` command as well as a

```
SET FILL AREA REPRESENTATION(WS,INDX,IS,SI,FACI)
```

command. The `WS` and `INDX` parameters should be familiar by now. `FACI` is a pointer to a color table. Parameter `IS` specifies the interior style of the fill area. It can be one of `HOLLOW`, `SOLID`, `PATTERN`, or `HATCH`, where the last three require more specifications.

Text: The general format of this primitive is `TEXT(X,Y,STRING)`. It draws the string of text with its bottom-left corner at point (X,Y). There are additional commands to specify the font, size, and orientation of the text.

GKS is a large system, including hundreds of commands and specifications, but the discussion above gives an idea of what it is like to use GKS. A three-dimensional GKS standard, called GKS-3D, was also developed. However, developments in computer graphics in the late 1980s and during the 1990s rendered GKS obsolete, and today it is rarely used.

20.2 IGES

The Initial Graphics Exchange Specification (IGES, pronounced eye-jess) is the standard for the interchange of design information between CAD/CAM software systems. Its official definition is in [IGES 86] and reference [IGES-NIST 10] is a source of much information on this topic. IGES defines a neutral data format that allows the digital exchange of information among CAD systems. Figure 20.1 is the IGES logo.

IGES was originally published in January, 1980 by the United States National Bureau of Standards as NBSIR 80-1978. Once a CAD user has constructed an object (two-dimensional or three-dimensional), IGES makes it easy to save a complete specification of the object or send it to other users. The object itself may be in the form of a circuit diagram, wireframe, freeform surface, or a surface built from patches as described in Part III of this book.

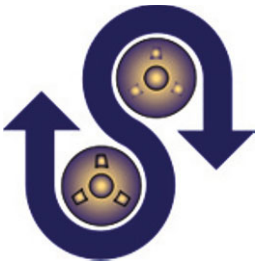

Figure 20.1: The IGES Logo.

As is common in cases of innovation and standardization, the IGES project was not the brainchild of a government agency, but was initiated in 1979 by a group of graphics users and manufacturers. Once IGES development had reached maturity, the National Bureau of Standards, a United States agency, lent it its support. After a few years of experience with IGES, several U.S. government departments decided to accept CAD contracts from private vendors only in IGES, which gave this standard a tremendous boost.

As a result of the popularity of IGES, the specifications for many parts are currently available in this standard. Parts for the automotive, shipbuilding, aerospace, and weapons industries can be made from these specifications even if the original makers are no longer in business.

With the advent of other CAD standards in the mid 1990s, interest in IGES had dwindled and this once-important standard, while still being used, is no longer developed.

A part (an object) described in IGES may be either (1) two-dimensional, described by a three-view drawing or (2) three-dimensional, described by any number of drawing views and dimensions and generated by CAD software. Once described by IGES, the part becomes a text file that can be stored, transmitted, or converted back to drawings and printed. The file can also be input by a computer-controlled machine in order to actually make the part.

An IGES file consists of entities. A part is described in IGES in terms of geometric and non-geometric entities. The former type includes physical shapes such as points, lines, arcs, curves, surface patches, and solids, while the latter consists of elements of annotation, definition, and organization. The annotations include dimensions, drafting notation, and text. These enhance the geometry information. The definitions specify elements that should be grouped and evaluated and operated on together.

Each entity in an IGES file consists of a directory entry and a parameter data entry. The directory entry includes an index and attributes describing the data. These are organized in fixed-length fields and are consistent for all entities in the file to provide simple access to frequently-used descriptive data. The parameter data consists of the definition of the entity. Here, fields are entity-specific and have variable lengths and different formats. The directory and parameter data of an entity are located in separate sections of the IGES file, and bi-directional pointers link them. The file itself consists

of five sections, Start, Global, Directory Entry, Parameter Data, and Terminate. There is also an optional Flag section. Each record is 80 ASCII characters (a format inherited from the old punched cards).

The Start section consists of a prologue. It describes the content of the file and is intended for users. It is skipped by software

The Global section has information describing the preprocessor as well as data needed by any post-processors.

The Directory entry section provides an index for the file and contains attribute information for each entity. This section contains a directory entry for each entity in the file (the order of these entries is arbitrary). A directory entry consists of 20 fields of eight characters each (for a total of 160 characters or two records)

The Parameter data section contains parameter values for each entity in the file. The list of parameters for an entity starts with the entity type and a pointer to the entity in the Directory entry section. The rest of the parameter data for the entry is entry dependent.

The Terminate section consists of one record with the letter `T` in column 73 as an identifier and with four fields indicating the lengths of the four preceding sections.

Here is an example of an entity. A surface of revolution (Section 16.2) is fully specified by (1) its axis of rotation, (2) start and end rotation angles, and (3) the generating curve (directrix). Thus, an entity for a surface of revolution (type 120) consists of a pointer to the directory entry of the axis (a straight segment, entity type 110), two real parameters (the rotation angles) and a pointer to the directory entry of the directrix (that can be any type of curve).

20.3 PHIGS

PHIGS (Programmer's Hierarchical Interactive Graphics Standard) is a sophisticated graphics standard [Hopgood and Duce 91], that offers commands for modeling as well as for drawing. It also features hierarchical structure of images and makes it possible to edit individual parts of an image. PHIGS also has capabilities for color specifications and surface rendering. An extension, called PHIGS+, was later developed [Howard et al. 91] to include shading of surfaces. PHIGS and PHIGS+ were merged into a single international standard, PHIGS, in 1997.

PHIGS was first proposed by ANSI in 1985, and after an intense, three-year development process, was adopted by ISO in 1988 as an international standard. The designers of PHIGS tried to preserve the concepts and terms of GKS whenever possible, but PHIGS supports many new commands and processes.

An important guideline in developing PHIGS was the distinction between physical and logical devices. Graphics devices are being invented, built, and used all the time, but it is very expensive to update existing software each time a new device (tablet, mouse, scanner, etc.) is added to a computer installation. PHIGS handles this problem by introducing the concepts of physical and logical devices. A physical device actually exists, while the software recognizes only logical devices. In PHIGS, a user can associate any physical device P with logical device L, and then any action assigned to L (for example, print) will be carried out by P.

Another important feature of PHIGS is hierarchical graphics data structures. A data structure may store graphics, text, and transformations and may point to another data structure and be pointed at by yet another data structure. Hierarchical graphics data structures are most important in computer animation, where each structure may contain an element of the object being animated (such as arm, hand, and fingers). The valid movements and orientations of an element may be restricted by the positions of other elements, and such restrictions can be implemented in a natural way in a hierarchical graphics data structure.

In the late 1980s and early 1990s, the capabilities of graphics hardware have increased rapidly and users felt that PHIGS had to be extended. The result was the PHIGS+ standard. After a few years of use, PHIGS and PHIGS+ were merged into a new PHIGS standard.

PHIGS provides routines and commands for three-dimensional transformations, while PHIGS+ includes features for rendering. Tasks such as lighting, shading, transparency of objects, and visible-surface determination are addressed by special routines that are part of PHIGS+.

However, even the extended PHIGS became outdated very quickly because of the rapid progress in graphics hardware and algorithms, and because PHIGS does not support features such as ray tracing, radiosity, shadows, texture mapping, and a large number of light sources. Today we say that PHIGS is ideal for most engineering and industrial graphics applications, but may not be adequate for the advanced visualization techniques that users have come to expect. These limitations convinced graphics vendors and users that a new standard was needed, and this standard, which became known as OpenGL, appeared in 1992.

The following is a list of the main features supported by PHIGS:

- Color
  Color Models: RGB, CIE LUV, HSV, HLS
  Lights: Ambient, Directional, Spot, and Positional
  Shading: Flat, Gouraud, and Phong Shading
  Depth Cueing: How color changes with distance
  Color Mapping: How the workstation approximates colors
  Data Mapping: The conversion of application data to color
- Area Primitives
  Fill Area
  Triangle Strips
  Quadrilateral Mesh
  B-spline Surface
- Text
  Text plane determined by text direction vectors
  Slanted and Rotated Text
  Text attributes, alignments, and character spacing
  Apply perspective to text
  Text Clipping
- Modeling

Local and Global Transforms
Scaling and Positioning the entire scene
Scaling, Positioning, and Orienting individual objects
Animating Models
Model Clipping: clipping plane, model-clipping volumes

- Viewing
  Setting the view coordinate system
  Placing the camera
  Parallel and Perspective Projections
  Panning and Zooming

Figure 20.2 shows several PHIGS output primitive operations, a polyline, polymarker, a filled polygon, a fill area set, and cell array.

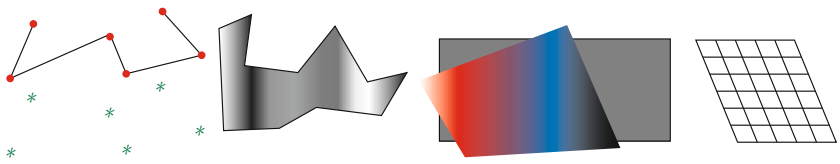

Figure 20.2: PHIGS Output Primitives.

We conclude with a short description of a few PHIGS functions, just to illustrate the syntax and capabilities of this software.

- Polyline. This important function was inherited from GKS. The following example illustrates how it is used in PHIGS.

```
DATA XPL /2, 4, 6, 8, 10 /
DATA YPL /1, 5, 2, 6,  4/
POLYLINE(5, XPL, YPL)
```

- Polymarker. The syntax is identical to polyline, but this function draws just the points, without the segments connecting them.

- Fill an area. The syntax is `FILL AREA(N, XA, YA)`, where `XA` and `YA` are lists with the coordinates of points.

- A character string is displayed by `TEXT(1, 3, 'Example String')`.

20.4 OpenGL

OpenGL (Open Graphics Library) is a graphics standard for writing applications that generate and manipulate two-dimensional and three-dimensional graphics objects. More precisely, OpenGL is an application programming interface (API). It is perhaps best to think of OpenGL as a library of graphics routines that are fast and also portable. An OpenGL user writes a program in C, C++, or another language and embeds in it OpenGL commands and function calls to quickly and easily construct and manipulate graphics objects and scenes. The algorithms employed by OpenGL have been carefully designed, implemented, tested, and improved by many users and implementors over the years in order to produce the best and fastest results.

OpenGL consists of over 250 functions and commands that can be employed (sometimes embedded in long programs) to construct and render complex three-dimensional objects, operations, and complete scenes, starting from simple primitives. OpenGL was developed by Silicon Graphics Inc. (SGI) in 1992 and is currently very popular with graphics programmers in many fields that require graphics output and graphics presentations. OpenGL was originally implemented in the C programming language. It is managed by the non-profit technology consortium Khronos Group, it currently (late 2010) stands at release 4.1, and its official website is `opengl.org`.

Like many other standards, OpenGL exists on two planes, the OpenGL specification and the actual implementations of this specification. The specification is very detailed and complete. It covers all aspects of the standard and specifies precisely how each feature works. Any actual implementation may be incomplete, may extend the specification, and may deviate from the official standard in various aspects. The implementation may be in the form of hardware (a graphics card) and software (a device driver), and its deviation from the standard should be well documented.

Most programming languages are procedural. A program in such a language describes steps whose execution leads to the desired result. OpenGL is also procedural. The programmer writes the individual steps needed to produce the final graphics. Some users/programmers feel more comfortable with a descriptive language, where the final scene with all its objects is described in detail, without specifying any steps.

History. The late 1980s witnessed rapid progress in computer hardware in general and in graphics hardware in particular. The range of graphics devices became wider almost by the week, and this presented a challenge to software developers. The effort to implement interfaces and drivers for each new graphics device became as expensive as the devices themselves. This was why Silicon Graphics (SGI) decided, in the early 1990s, to develop a new graphics standard, based on their existing IRIS GL programming language, that would supersede PHIGS and would become a general graphics standard and software system for years to come.

The guiding principle in the development of OpenGL was to standardize access to graphics hardware. By having a modern standard, it was hoped that hardware makers would write and debug device drivers for their new devices, and would thereby allow programmers to access and use new, powerful graphics devices from a high-level language.

Initial efforts in this direction were successful and attracted the attention of several vendors. As a result, SGI founded the OpenGL architectural review board in 1992. A

group of several companies participates in this body, which is responsible for maintaining and extending OpenGL.

References. Some detailed references to OpenGL are [OpenGL 99], [OpenGL 09], and [OpenGL 10]. It is easy to search and find many more sources, including textbooks, class notes, and code examples.

Design. An OpenGL application is independent of the particular computer on which it is executed. This feature, of being platform independent, is considered important, but computers have (and will continue to have) such different architectures, that it proved impossible to design and implement a complete software system that would be 100% platform independent. As a result, several important programming features are missing from OpenGL. In particular, this standard does not offer commands for performing windowing tasks or obtaining user input. When writing graphics software with OpenGL, the programmer must use whatever features are available in the particular operating system in order to control windows on the screen and input data from the keyboard, mouse, and other input devices.

Another desirable feature missing from OpenGL is high-level commands describing three-dimensional objects. Instead, a user must start from simple graphics primitives and use them to build up a complex object. To correct this situation somehow, users have written an OpenGL Utility Library (GLU), which helps in constructing many complex objects such as surfaces of revolution and NURBS curves and surfaces.

A typical OpenGL program specifies the objects to be rendered and how they should be rendered (as wireframe only, as simple solids, with multiple light sources, in a specific color space, in perspective or parallel, with fog or casting shadows). The objects are specified by starting from points, using their coordinates to specify primitive graphics objects, and combining those (mostly polygons) to obtain more complex objects.

Command syntax. An OpenGL command starts with `gl` (except for GLU commands, which employ the prefix `glu`), followed by one or several words, each with an initial uppercase letter. This may be followed by a list of arguments. OpenGL constants start with `GL_`. Following is a list of examples:

- `glClearColor(0.0, 0.0, 0.0, 0.0)` sets the clearing color to RGBA black.
- `glClearDepth(1.0)` specifies the value (floating-point 1) to which every pixel of the depth buffer is to be set.
- `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)` clears the entire window to the current clearing color. The first argument `GL_COLOR_BUFFER_BIT` indicates the color buffer (the buffer where the current image is stored) and the second argument refers to the depth buffer.
- `glColor3f(1.0, 0.0, 0.0)` sets the current color to red.
- `glViewport(0, 0, 100, 110)` sets the bottom-left corner of the current viewport to $(0, 0)$ and the top-right corner to $(100, 110)$.
- `glBegin(GL_POLYGON)` starts a group of commands that define an object.
- `glEnd()` ends such a group.
- `glVertex3f(0.75, 0.75, 0.0)` defines a point. Notice the `3`, which indicates that this is a three-dimensional point, and the `f`, which means that the coordinates are

floating-point numbers. Thus, a command such as `glVertex2i(1, 3)` indicates a two-dimensional point with integer coordinates. If the final letter of a command is `v`, the command takes a pointer, not data values, as its argument.

- `UpdateTheWindowAndCheckForEvents()` is an example of a long command.

Examples: Here are two simple examples of OpenGL groups.

To draw two points

```
glBegin(GL_POINTS);
glVertex2f( 1.0, 1.0 );
glVertex2f( 2.0, 1.0 );
glEnd();
```

To draw a triangle

```
glBegin(GL_TRIANGLES);
glColor3f( 1.0, 0.0, 0.0 );
glVertex3f( 0.3, 1.0, 0.5 );
glVertex3f( 2.7, 0.85, 0.0 );
glVertex3f( 2.7, 1.15, 0.0 );
glEnd();
```

20.5 PostScript

PostScript is a page-description language designed by Adobe Inc. in the mid-1980s to serve as a device-independent language where any desired page can be described independently of any output device. The page can then be displayed or printed on any graphics output device that has a PostScript interpreter. The main reference is [Adobe Systems Inc. 90] but the excellent tutorial [Adobe Systems Inc. 85] is perhaps a better place to start.

Describing a page of text is easy, because each character can be described by its ASCII (or UNICODE) code. Given a page with 40 lines, each with 120 characters, it takes 4,800 bytes to describe it. If the text is positioned irregularly on the page, as in the case of mathematical expressions, then each character can be represented by a triplet of the form (x, y, ASCII code). The amount of space required may be 3–5 times 4,800 bytes; still a small amount of space. However, if the page may contain images in addition to text, a complete description of the page is more complex and requires much more space. To quote the poet Zbigniew Herbert [Herbert 93]: "Language must go to great lengths to accomplish a mere replica of what painting does in an instant. Arranging sentences to describe a canvas is like hauling heavy furniture around a room." This task is accomplished by PostScript. A PostScript file is a text file containing a complete description of one or more pages of text and images.

The interpreter is device dependent (i.e., any PostScript output device has to have its own interpreter). It reads a plain text file that includes the PostScript description of the page, and converts the PostScript commands (which are also called operators) to printer-specific commands. This is how the page can be printed on different printers. The results produced by the printers are not identical since they depend on the printer's resolution, quality, and number of colors. A high-resolution image printed on a low resolution printer will come out in low resolution regardless of how it is described to the printer. Similarly, a color image printed on a black and white printer will come out in black and white.

Many current laser printers have a built-in PostScript interpreter. The StyleScript software package, from strydent software [strydent 11], is a PostScript interpreter for some inkjet printers. GhostScript, by L. Peter Deutsch [Ghostscript 98] is a free PostScript interpreter that can display an image on the computer screen and on dozens of other devices. It has been ported to various platforms.

PostScript includes commands that make it possible to specify graphics elements and place them at precise locations on the page. There are three major types of graphics elements: text, geometric figures, and digitized images.

Text: Any string of characters, from any font, can be specified and placed on the page at any location and in any orientation. The text may also follow a curve.

Geometric Figures: Straight segments, curves, and areas can be defined and placed on the page. The areas can be filled with any pattern or color and an area can be clipped to the boundary of any other area.

Sampled Images: Such images are produced by digital cameras and can also be obtained by scanning (digitizing) any drawing, painting, photograph, or any other image. The sampled image can be placed on the page in any orientation and can also be scaled (which normally reduces its quality).

The PostScript language uses the following three important terms:

Current Page: This starts blank and becomes filled with graphics elements as more and more PostScript *painting operators* are being executed. However, the page is not printed until the `showpage` command is executed. Any graphics object placed on the page obscures anything behind it, since PostScript assumes that all colors are opaque. Nothing is transparent or translucent.

Current Path: The `newpath` operator starts a new current path. A PostScript path is a set of graphics elements such as points, lines, curves, and areas. It is constructed by PostScript's *path operators*. A path does not have to be contiguous on the paper (i.e., it may consist of disconnected parts) and it does not automatically drawn on the paper. To actually place marks on the page, the user has to stroke the path and fill it. The term *stroke* refers to the edge (or outline) of the path. It can be thick or thin, solid or dashed, it can be in any shade of gray or in any color, and it can be a pattern. The path can be filled with white, black, gray, any color, or any pattern. A path can have multiple strokes and fills or none. The latest stroke (or fill) covers its predecessors, but several strokes (or fills) can be visible if they have different widths or opacities. Figure 20.3 shows examples of paths with various strokes and fills. The current path is terminated when the PostScript interpreter encounters the next `newpath` or a `showpage`.

Clipping Path: The initial clipping path is the entire current page. At any time, the user can specify any path as the current clipping path. Following that, when a mark is placed on the page, only those parts of the mark that are inside the clipping path will be drawn and the rest will be clipped.

> I was staring through the cage of those meticulous ink strokes at an absolute beauty.
> —F. Murray Abraham (as Antonio Salieri) in *Amadeus* (1984).

The reason for the name PostScript is that the language is based on a stack and on the *postfix* notation, where an operator follows its operands. The language is also extensible in the sense that once a new procedure is defined, it can be used as any built-in operator. The following example illustrates the use of the PostScript stack.

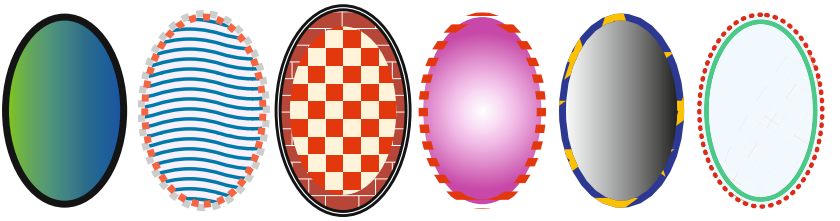

Figure 20.3: Examples of Stroke and Fill.

```
-6
18
add
```

When the PostScript interpreter reads the first number -6, it simply pushes it into the stack since it has nothing else to do with it. The same is true for the second number. When the `add` operator is found and is executed, it pops the top two stack elements, adds them, and places the sum at the top of the stack. In general, the rule is that any numbers being input are placed on the stack and that operators look for their operands in the stack, remove them, and place their results on the stack.

Notice, however, that a stack element does not have to be a number. It can be a string of characters, for example.

◇ **Exercise 20.1:** Guess how the following is executed by the PostScript interpreter "`3.2 11.6 sub`"?

The next example defines a path, strokes it, and prints the page:

```
1 newpath
2 72 144 moveto
3 216 72 lineto
4 stroke
5 showpage
```

Line 1 starts a new path that's terminated by line 5. Line 2 pushes the numbers 72 and 144 into the stack and executes `moveto`. This command pops the top two stack elements, uses them as the (x, y) coordinates of a point on the page, and moves an imaginary pen to that point (without drawing anything). Line 3 is similar, but the `lineto` operator draws a line as it moves the pen. Line 4 strokes the path, which makes it visible. The precise stroke used depends on the values of several PostScript parameters. Finally, line 5 causes the page to be printed (or displayed, if GhostScript or something similar is used). The result is a straight line from $(72, 144)$ to $(216, 72)$. PostScript uses a default coordinate system with an origin at the bottom-left corner of the page and with 72 coordinate units per inch. Expressed in inches, the coordinates of the endpoints of our line will therefore be $(1, 2)$ and $(3, 1)$. The origin, orientation, and units of the coordinate system can, of course, be changed by the user.

There are also `rmoveto` and `rlineto` operators that consider their operands as relative coordinates. A square, for example, is drawn by the (relatively long) PostScript program

```
1 newpath
2 288 288 moveto
3 0 72 rlineto
4 72 0 rlineto
5 0 -72 rlineto
6 -72 0 rlineto
7 4 setlinewidth
8 stroke showpage
```

where lines 3–6 draw four 1-inch segments and line 7 sets the stroke width to four coordinate units (i.e., 4/72 in when the default is used). Since a square is a closed path, it is better to close it automatically. This is done by replacing the last line segment "`-72 0 rlineto`" in line 6 with the operator `closepath`. The square can be filled by saying `fill` instead of (or in addition to) `stroke`. The default fill is black, but a fill of 50% gray can be specified by the commands `.5 setgray fill`.

If a certain image calls for many squares, it is best to define the above program as a *procedure*. Before showing how this is done, we need to discuss the PostScript dictionaries. A common language dictionary is a set of pairs where each pair consists of a word and its definition. Similarly, a PostScript dictionary is a set of pairs, where each pair consists of a *key* and its *value*. At any time, there are at least two dictionaries: the system dictionary and the user dictionary. The former contains the predefined PostScript operators and the latter contains the user-defined procedures and variables.

When the interpreter reads an item from the input file, it searches the user dictionary, then the system dictionary for the item. If it finds the item as a key in one of the dictionaries, it uses the associated value to decide what to do next. If the item is not found, the PostScript interpreter generates an error.

The user may create new user dictionaries and PostScript maintains a dictionary stack. Initially, this stack contains the user dictionary at the top and the system dictionary below it. As more dictionaries are created, they are added at the top of the dictionary stack. The topmost dictionary is called the current dictionary.

If a sequence of PostScript operators is used a lot in a program, it can be defined as a procedure (i.e., it can become a user-defined operator). It is assigned a name and both its name and its definition are stored as the key and value, respectively, of a pair in the current dictionary. Defining a new operator is done by the `def` operator. As an example, we show how to define the sequence of commands for a square as a procedure called `square`. The code is

```
/square
{newpath
moveto
0 72 rlineto
72 0 rlineto
0 -72 rlineto
```

```
closepath}
def
```

where the slash "/" in the first line indicates that the string `square` (that is going to be the procedure's name) is to be pushed into the stack. The curled braces are also pushed, with their contents, into the stack. The stack thus contains two items, but notice that neither is a number. The `def` operator on the last line pops two items from the stack and enters them into the current dictionary. The top item becomes the value and the item below it (the string `square`) becomes the key. Once this is done, the string `square` can be used, since it can be found in the current dictionary (i.e., its value is known). An example is

```
72 144 square stroke
288 288 square 4 setlinewidth stroke
0 288 square .5 setgray fill
showpage
```

Notice how each use of `square` is preceded by pushing two numbers into the stack. They can be considered the parameters of procedure `square` and they specify the location of the square on the page.

Curves can be drawn by the operator `x1 y1 x2 y2 x3 y3 curveto`. This draws a cubic Bézier curve segment from the current position of the pen to point (x_3, y_3), using (x_1, y_1) and (x_2, y_2) as the two intermediate control points.

Adobe Illustrator is an example of a graphics program, available for several platforms, that produces its output in PostScript. Another example is `dvips`, by Tomas Rokicki. It translates a `dvi` file, which is the main output produced by the typesetting program TEX, to PostScript.

20.6 Graphics File Formats

The preceding sections discuss graphics standards, but these are only one aspect of an overall standardization effort. Another aspect is the way a graphics file is organized, and several such formats are described in the remainder of this chapter (the all-important JPEG standard is more a compression method than a file format and is described in detail in Section 24.5).

Section 26.4.5 is concerned with raw image format. In principle, a raw image file may contain only the dimensions of the image, the number of bits per pixel, a code for the color space used (RGB, CMY, or others), and the three color values for each pixel. Such a file is big but has a number of advantages including: (1) It enables the user to change the white balance to the correct value *after* the picture has been taken, (2) it may provide considerably more dynamic range than a JPEG file, and (3) it allows for quick and lossless transformations of the color space. However, most image files that are generated by cameras and scanners are in some compressed form, to save space.

Standards are important in all fields, and image files are no exception. We want to send, receive, and store image files, to process images in many ways, and to display images in websites. Thus, standard formats for image files are a must. These formats can be classified in several ways as follows:

- By the number of dimensions. Most images are two dimensional. A photograph, for example, can be scaled and rotated, but this does not reveal previously-hidden details. Three-dimensional images can be generated by many programs and they are different and more complex than the two-dimensional variety. When a three-dimensional image is rotated, we may see details and objects that were previously hidden from view. Such an image requires a different file format. In addition to images in two and three dimensions, there are also graphics file formats for video (where time can be considered an extra dimension).

- As a bitmap or as vector graphics. A photograph may contain much graphics information, but it is essentially a bitmap (individual pixels). When such an image is saved as a file, it can be saved in raw or compressed format, but it is always the color values of the pixels that are saved in the file. On the other hand, many drawing and illustration programs construct an image from geometrical entities such as points, lines, curves, and circles, each with a stroke and fill (Section 20.5). When such an image is displayed, the software first converts it to a bitmap, but when the image is saved as a file, only the geometrical data, and not the pixels, is written on the file. (A graphics file may include both bitmaps and geometrical data for different parts of an image. Such a file is sometimes referred to as a metadata file.)

- As explicit color or as palette-index. In the former type, the color of each pixel is written on the image file, in either raw or compressed form. A palette-index graphics file, on the other hand, contains an index for each pixel. This is an index to a palette (a table) of colors, which is also part of the file. Thus, if the value 3 is stored in the file for a certain pixel, then the color of the pixel is the color that happens to be stored in entry 3 of the color table (the palette). The chief advantage of this type of graphics file is small size. The image itself may be large, but if the number of colors is small, then the palette is small and its indexes are small numbers. Instead of three color components for each pixel, there is a single small index. The palette is therefore a form of lossless image compression. There is also a secondary advantage. By changing one palette entry, all the pixels with indexes to that entry change their color simultaneously.

Currently, there are dozens, perhaps even hundreds, of image file formats, many of them proprietary. The following sections describe the TIFF, PNG, and GIF formats. The all-important JPEG compression and file format are described in Section 24.5, and PostScript, the popular page description language, is the topic of Section 20.5.

20.7 GIF

The Graphics Interchange Format (GIF) is a bitmap palette-based graphics format for still images and animation. The format is based on a color palette of 256 entries, which allows for only 256 RGB colors. On the other hand, an index to such a table is only one byte, so each pixel of the image occupies one byte in the GIF file instead of the usual three bytes.

GIF was introduced by CompuServe in 1987 (the word GIF can be pronounced with either a soft or a hard `G`). At that time, 256 colors seemed a lot, but in spite of the many colors we can use today, GIF is still popular because it supports animation and

because many images—such as simple diagrams, shapes, logos, sharp-edged line art, and cartoons—require only a small number of colors.

The original GIF format (dubbed GIF87a) had a facility for multiple images. In 1989, CompuServe extended this format to support animation, transparent background colors, and storage of application-specific metadata. This version (see [martinreddy 10] for the full specifications) became known as GIF89a and is the one used today. The first six bytes (the header) of a GIF file identify the version as either `GIF87a` or `GIF89a`.

GIF also uses lossless LZW compression [Salomon 09]. This type of compression is effective for images that have large uniform regions. Continuous-tone (natural) images (Page 1031), such as photographs, compress better with a lossy method, such as JPEG. The LZW algorithm was patented, and for many years its use required a license from Unisys. The controversy surrounding this patent was one of the reasons for the development of the PNG graphics format (Section 20.9).

Another interesting (and optional) GIF feature is interlacing. This permits rows of the image to be stored in the GIF file out of natural order. When the file is read and displayed, the resulting image may often be recognizable after only 10–20% of its rows have been displayed.

As has already been mentioned, the header of the GIF file is one of the strings `GIF87a` and `GIF89a`. This is followed by a fixed-length logical screen descriptor (LSD) with the size and other features of the image. The LSD may also specify the presence and size of an optional global color table, which follows the LSD. Following this, the remainder of the file consists of segments, each preceded by a 1-byte sentinel whose values are $2C_{16}$ for an image, 21_{16} for an extension block, and $3B_{16}$ for the trailer (the last byte of the file).

An image segment (sentinel $2C_{16}$) starts with a fixed-length image descriptor (ID). The ID specifies the presence and length of a local color table. If such a table is present, it follows the ID. Following this is the image data itself, encoded in LZW.

An extension segment (sentinel 21_{16}) extends the original GIF 87a format. The sentinel is followed by blocks with the extension data. (Each block begins with a byte giving its length.) Examples of such data are the animation delay time and the transparent background color.

Another example of an extension block is the Netscape application block, introduced by Netscape in the 1990s. This block indicates that the GIF file contains an animation, a set of images (called frames) to be displayed consecutively. Each frame is LZW-encoded and is stored in short blocks inside a variable-length graphic control extension (GCE) data structure. Each frame is displayed for a certain period of time, measured in hundredths of a second, during which the next frame is decoded and is prepared for display.

The presence of sentinels and length specifications makes is possible for software to read a GIF file even when certain parts of it are not understood (perhaps by old software, unfamiliar with the GIF89a features).

Figure 20.4 is a complete listing of the GIF file of a small, 4×4 color image. Notice the sentinel $2C_{16}$ in locations 45 and 781, and sentinel $3B_{16}$ at the very end of the file. There are no extension segments (sentinel 21_{16}) in this file.

```
000  474946383961040004000F70731FFFFFFFCF305FF6402DD0806F20884
028  4600A50000D402ABEA1FB714006411562C0590713AC0C0C080808040
056  4040000000000000000000000000000000000000000000000000000000
084  0000000000000000000000000000000000000000000000000000000000
112  0000000000000000000000000000000000000000000000000000000000
140  0000000000000000000000000000000000000000000000000000000000
168  0000000000000000000000000000000000000000000000000000000000
196  0000000000000000000000000000000000000000000000000000000000
224  0000000000000000000000000000000000000000000000000000000000
252  0000000000000000000000000000000000000000000000000000000000
280  0000000000000000000000000000000000000000000000000000000000
308  0000000000000000000000000000000000000000000000000000000000
336  0000000000000000000000000000000000000000000000000000000000
364  0000000000000000000000000000000000000000000000000000000000
392  0000000000000000000000000000000000000000000000000000000000
420  0000000000000000000000000000000000000000000000000000000000
448  0000000000000000000000000000000000000000000000000000000000
476  0000000000000000000000000000000000000000000000000000000000
504  0000000000000000000000000000000000000000000000000000000000
532  0000000000000000000000000000000000000000000000000000000000
560  0000000000000000000000000000000000000000000000000000000000
588  0000000000000000000000000000000000000000000000000000000000
616  0000000000000000000000000000000000000000000000000000000000
644  0000000000000000000000000000000000000000000000000000000000
672  0000000000000000000000000000000000000000000000000000000000
700  0000000000000000000000000000000000000000000000000000000000
728  0000000000000000000000000000000000000000000000000000000000
756  0000000000000000000000000000000000000000000000002C0000
784  00000400040000815000F1028F04F8001040A123008D000C0000704
812  0202003B
```

Figure 20.4: A 4×4 GIF Image.

20.8 TIFF

TIFF (Tagged Image File Format) is a file format that can handle various types of data in addition to images. A TIFF file consists of sections where a section may contain a

bitmap image (in lossy or lossless compression or raw), a vector-based image, text, or other type of data. The type of data (and also type of compression) contained in a section is identified by the section's tag. The images within a TIFF file can be monochromatic, grayscale, palette-based, or full color. Each color component of a pixel is saved in the TIFF file as either eight or 16 bits. The LZW compression method [Salomon 09] is used in a TIFF file for lossless compression, while JPEG is used for lossy compression.

The TIFF format was developed in the mid-1980s by Aldus Corp. mostly as a standard format for scanner output. The first published version (version 3, in 1986) could handle only monochromatic images (which was all that scanners could produce in those years). Versions 4 and 5 came out in 1987 and 1988, respectively and included many extensions. Today, the TIFF format is maintained by Adobe (which acquired Aldus), and it has been stable since 1992, when version 6.0 was finalized.

Version 6.0 consists of several enhancements of version 5, the most important of which are the following:

- CMYK image definition
- A revised RGB Colorimetry section
- YCbCr image definition
- CIE L*a*b* image definition
- Tiled image definition
- JPEG compression

A TIFF file starts with a two-byte indicator of either `II` (for little endian) or `MM` (for big endian). The next two bytes contain the verification constant `42` (itself in little- or big-endian). The next four bytes (long word) contain the 32-bit offset (relative address from the start of the file) of the first image file directory (IFD). This directory may be located anywhere in the file, and may even follow the image it describes.

> The term big endian means that the high-order byte (the big end) of a number or a string is stored in memory at the lowest address (it comes first). For example, given the four-byte number $b_3b_2b_1b_0$, if the most-significant byte b_3 is stored at address A, then the least-significant byte b_0 will be stored at address $A + 3$. Naturally, little endian refers to the reverse byte order.

Notice that offsets in a TIFF file are always 32 bits long and specify a location from the beginning of the file. Thus, the file itself cannot be longer than $2^{32} = 4$ Giga bytes. Back in the 1980s, this file size was considered huge, but today it is not very large. Thus, a new version of TIFF, called BigTIFF, employs 64-bit offsets and can accommodate much larger files.

An Image File Directory (IFD) is an important component of a TIFF file. It consists of a variable number of 12-byte fields, which is why its first two bytes contain the number of fields. The four bytes that follow are the offset of the next IFD (or zero if this is the last IFD). Each IFD field is 12 bytes long and has the following format:

A two-byte tag
A two-byte type

A four-byte length
A four-byte offset for the value of the field

The value is the payload of each IFD field. It is typically an image. The offset and length tell software where to find the value and how long it is. The tag and type specify the type of data stored in the value.

Each IFD is considered a subfile. This term is especially useful when the TIFF file contains an image. In such cases, each subfile may consist of a "subimage" that is somehow related to the main image.

The fields of an IFD must be sorted in ascending order by tag, but a field may point to any type of value.

There can be five types as listed here:

1 = BYTE. An 8-bit unsigned integer.
2 = ASCII. 8-bit bytes that store ASCII codes; the last byte must be null.
3 = SHORT. A 16-bit (2-byte) unsigned integer.
4 = LONG. A 32-bit (4-byte) unsigned integer.
5 = RATIONAL. Two LONG's: the first represents the numerator of a fraction, the second the denominator.

TIFF allows for many tags. Here are a few examples:

ImageWidth
Tag = 256 (100)
Type = SHORT or LONG
N = 1

ImageLength
Tag = 257 (101)
Type = SHORT or LONG
N = 1

BitsPerSample
Tag = 258 (102)
Type = SHORT
N = SamplesPerPixel

ImageDescription
Tag = 270 (10E)
Type = ASCII

SamplesPerPixel
Tag = 277 (115)
Type = SHORT
N = 1

RowsPerStrip
Tag = 278 (116)
Type = SHORT or LONG
N = 1

XResolution
Tag = 282 (11A)
Type = RATIONAL

N = 1

YResolution
Tag = 283 (11B)
Type = RATIONAL
N = 1

TIFF classes. Originally, the TIFF format was designed as a standard for scanner makers. Because of its success, its developers have expanded it until it became very powerful and flexible, but also very complex. In order to simplify the task of creating, opening, and reading TIFF files, the developers of this standard have introduced the concept of classes. The following classes have been defined so far (but more may be added in the future).

- Class B for bilevel (monochromatic) images
- Class G for grayscale images
- Class P for palette color images
- Class R for RGB full color images

The following is a simple example of a TIFF file:

A Sample TIFF B Image
Offset Value
(hex) Name (mostly hex)
Header:
0000 Byte Order 4D4D
0002 Version 002A
0004 1st IFD pointer 00000014
IFD:
0014 Entry Count 000D
0016 NewSubfileType 00FE 0004 00000001 00000000
0022 ImageWidth 0100 0004 00000001 000007D0
002E ImageLength 0101 0004 00000001 00000BB8
003A Compression 0103 0003 00000001 8005 0000
0046 PhotometricInterpretation 0106 0003 00000001 0001 0000
0052 StripOffsets 0111 0004 000000BC 000000B6
005E RowsPerStrip 0116 0004 00000001 00000010
006A StripByteCounts 0117 0003 000000BC 000003A6
0076 XResolution 011A 0005 00000001 00000696
0082 YResolution 011B 0005 00000001 0000069E
008E Software 0131 0002 0000000E 000006A6
009A DateTime 0132 0002 00000014 000006B6
00A6 Next IFD pointer 00000000
Fields pointed to by the tags:
00B6 StripOffsets Offset0, Offset1, ... Offset187
03A6 StripByteCounts Count0, Count1, ... Count187
0696 XResolution 0000012C 00000001
069E YResolution 0000012C 00000001

06A6 Software "PageMaker 3.0"
06B6 DateTime "1988:02:18 13:59:59"
Image Data:
00000700 Compressed data for strip 10
xxxxxxxx Compressed data for strip 179
xxxxxxxx Compressed data for strip 53
xxxxxxxx Compressed data for strip 160
⋮
End of example

Comments on the example

1. The IFD in this example starts at position 14_{16}. As has been mentioned elsewhere, an IFD can be located anywhere in a TIFF file.

2. With 16 rows per strip, there is a total of 188 strips.

3. For illustration purposes, the example has highly fragmented image data; the strips of the image in the example are not in sequential order. The point is that strip offsets must not be disregarded. The software reading a TIFF file should not assume that strip $n + 1$ follows strip n. Also, the image data does not have to follow the IFD information. The software should strictly follow the pointers, whether they be IFD pointers, field pointers, or strip offsets.

20.9 PNG

The portable network graphics (PNG) file format has been developed in the mid-1990s by a group (the PNG development group [PNG 03]) headed by Thomas Boutell. The project was started in response to the legal issues surrounding the GIF file format (Section 20.7). The aim of this project was to develop a sophisticated graphics file format that will be flexible, will support many different types of images, will be easy to transmit over the Internet, and will be unencumbered by patents. The design was finalized in October 1996, and its main features are as follows:

1. It supports images with 1, 2, 4, 8, and 16 bitplanes.
2. Sophisticated color matching.
3. A transparency feature with very fine control provided by an alpha channel.
4. Lossless compression by means of Deflate combined with pixel prediction.
5. Extensibility: New types of meta-information can be added to an image file without creating incompatibility with existing applications.

Currently, PNG is supported by many image viewers and web browsers on various platforms. This section is a general description of the PNG format, followed by the details of the compression method it employs.

A PNG file consists of chunks that can be of various types and sizes. Some chunks are critical. They contain information that's essential for displaying the image, and decoders must be able to recognize and process these chunks. They may enhance the display of the image or may contain meta-data such as the image title, author's name, creation and modification dates and times, etc. (but notice that decoders may choose

not to process such chunks). New, useful types of chunks can also be registered with the PNG development group.

A chunk consists of the following parts: (1) size of the data field, (2) chunk name, (3) data field, and (4) a 32-bit cyclical redundancy code (CRC, Section 20.10). Each chunk has a four-letter name of which (1) the first letter is uppercase for a critical chunk and lowercase for an ancillary chunk, (2) the second letter is uppercase for standard chunks (those defined by or registered with the PNG group) and lowercase for a private chunk (an extension of PNG), (3) the third letter is always uppercase, and (4) the fourth letter is uppercase if the chunk is "unsafe to copy" and lowercase if it is "safe to copy."

Any PNG-aware application will process all the chunks it recognizes. It can safely disregard any ancillary chunk it doesn't recognize, but if it finds a critical chunk it cannot recognize, it has to stop and issue an error message. If a chunk cannot be recognized but its name indicates that it is safe to copy, the application may safely read and rewrite it even if it has altered the image. However, if the application cannot recognize an "unsafe to copy" chunk, it must discard it. Such a chunk should not be rewritten on the new PNG file. Examples of "safe to copy" are chunks with text comments or those indicating the physical size of a pixel. Examples of "unsafe to copy" are chunks with gamma/color correction data or palette histograms.

The four critical chunks defined by the PNG standard are IHDR (the image header), PLTE (the color palette), IDAT (the image data, as a compressed sequence of filtered samples), and IEND (the image trailer). The standard also defines several ancillary chunks that are deemed to be of general interest. Anyone with a new chunk that may also be of general interest may register it with the PNG development group and have it assigned a public name (a second letter in uppercase).

The PNG file format uses a 32-bit CRC (Section 20.10) as defined by certain international standards. The CRC polynomial is

$$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1.$$

The particular calculation proposed in the PNG standard employs a precomputed table that speeds up the computation significantly.

A PNG file starts with an 8-byte signature that helps software to identify it as PNG. This is immediately followed by an IHDR chunk with the image dimensions, number of bitplanes, color type, and data filtering and interlacing. The remaining chunks must include a PLTE chunk if the color type is palette, and one or more adjacent IDAT chunks with the compressed pixels. The file must end with an IEND chunk. The PNG standard defines the order of the public chunks, whereas private chunks may have their own ordering constraints.

An image in PNG format may have one of the following five color types: RGB with 8 or 16 bitplanes, palette with 1, 2, 4, or 8 bitplanes, grayscale with 1, 2, 4, 8, or 16 bitplanes, RGB with alpha channel (with 8 or 16 bitplanes), and grayscale with alpha channel (also with 8 or 16 bitplanes). An alpha channel implements the concept of transparent color. One color can be designated transparent, and pixels of that color are not displayed or printed. Instead of seeing those pixels, a viewer sees the background behind the image. The alpha channel is a number in the interval $[0, 2^{bp}-1]$, where bp is the number of bitplanes. Assuming that the background color is B, a pixel in

the transparent color P is painted in color $(1-\alpha)B+\alpha P$. This is a mixture of $\alpha\%$ background color and $(1-\alpha)\%$ pixel color.

Perhaps the most intriguing feature of the PNG format is the way it handles interlacing. Interlacing makes it possible to display a rough version of the image on the screen, then improve it gradually until it reaches its final, high-resolution form. The special interlacing used in PNG is called Adam 7 after its developer, Adam M. Costello. PNG divides the image into blocks of 8×8 pixels each, and displays each block in seven steps. In step 1, the entire block is filled up with copies of the top-left pixel (the one marked "1" in Figure 20.5a). In each subsequent step, the block's resolution is doubled by modifying half its pixels according to the next number in Figure 20.5a. This process is easiest to understand with an example, such as the one shown in Figure 20.5b.

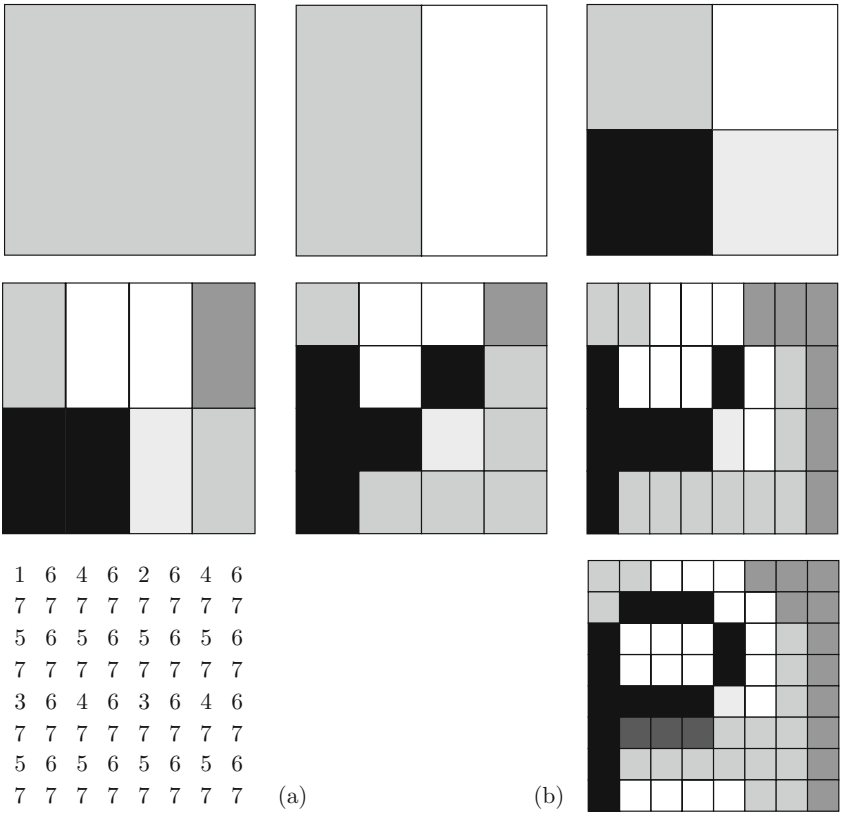

Figure 20.5: Interlacing in PNG.

PNG compression is lossless and is performed in two steps. The first step, termed delta filtering (or just filtering), converts pixel values to numbers by a process similar

to the prediction used in the lossless mode of JPEG (Section 24.5.4). The filtering step calculates a "predicted" value for each pixel and replaces the pixel with the difference between the pixel and its predicted value. The second step employs Deflate (a method described in [Salomon 09]) to encode the differences.

Filtering does not compress the data. It only transforms the pixel data to a format where it is more compressible. Filtering is done separately on each image row, so an intelligent encoder can switch filters from one image row to the next (this is called adaptive filtering). PNG specifies five filtering methods (the first one is simply no filtering) and recommends a simple heuristic for choosing a filtering method for each image row. Each row starts with a byte indicating the filtering method used in it. Filtering is done on bytes, not on complete pixel values. This is easy to understand in cases where a pixel consists of three bytes, specifying the three color components of the pixel. Denoting the three bytes by a, b, and c, we can expect a_i and a_{i+1} to be correlated (and also b_i and b_{i+1}, and c_i and c_{i+1}), but there is no correlation between a_i and b_i. Also, in a grayscale image with 16-bit pixels, it makes sense to compare the most-significant bytes of two adjacent pixels and then the least-significant bytes. Experiments suggest that filtering is ineffective for images with fewer than eight bitplanes, and also for palette images, so PNG recommends no filtering in such cases.

The heuristic recommended by PNG for adaptive filtering is to apply all five filtering types to the row of pixels and select the type that produces the smallest sum of absolute values of outputs. (For the purposes of this test, the filtered bytes should be considered signed differences.)

The five filtering types are described next. The first type (type 0) is no filtering. Filtering type 1 (sub) sets byte $B_{i,j}$ in row i and column j to the difference $B_{i,j} - B_{i-t,j}$, where t is the interval between a byte and its correlated predecessor (the number of bytes in a pixel). Values of t for the various image types and bitplanes are listed in Table 20.6. If $i - t$ is negative, then nothing is subtracted, which is equivalent to having a zero pixel on the left of the row. The subtraction is done modulo 256, and the bytes subtracted are considered unsigned.

Image type	Bit planes	Interval t
Grayscale	1, 2, 4, 8	1
Grayscale	16	2
Grayscale with alpha	8	2
Grayscale with alpha	16	4
Palette	1, 2, 4, 8	1
RGB	8	3
RGB	16	6
RGB with alpha	8	4
RGB with alpha	16	8

Figure 20.6: Interval between Bytes.

Filtering type 2 (up) sets byte $B_{i,j}$ to the difference $B_{i,j} - B_{i,j-1}$. The subtraction is done as in type 1, but if j is the top image row, no subtraction is done.

Filtering type 3 (average) sets byte $B_{i,j}$ to the difference $B_{i,j} - \left[B_{i-t,j} + B_{i,j-1}\right] \div 2$. The average of the left neighbor and the top neighbor is computed and subtracted from the byte. Any missing neighbor to the left or above is considered zero. Notice that the sum $B_{i-t,j} + B_{i,j-1}$ may be up to nine bits long. To guarantee that the filtering is lossless and can be reversed by the decoder, the sum must be computed exactly. The division by 2 is equivalent to a right shift and brings the nine-bit sum down to eight bits. Following that, the eight-bit average is subtracted from the current byte $B_{i,j}$ modulo 256 and unsigned.

Example: Assume that the current byte $B_{i,j} = 112$, its left neighbor $B_{i-t,j} = 182$, and its top neighbor $B_{i,j-1} = 195$. The average is $(182 + 195) \div 2 = 188$. Subtracting $(112 - 188)$ mod 256 yields -76 mod 256 or $256 - 76 = 180$. Thus, the encoder sets $B_{i,j}$ to 180. The decoder inputs the value 180 for the current byte, computes the average in the same way as the encoder to end up with 188, and adds $(180 + 188)$ mod 256 to obtain 112.

Filter type 4 (Paeth) sets byte $B_{i,j}$ to $B_{i,j} - \text{PaethPredict}\left[B_{i-t,j}, B_{i,j-1}, B_{i-t,j-1}\right]$. PaethPredict is a function that uses simple rules to select one of its three parameters, then returns that parameter. Those parameters are the left, top, and top-left neighbors. The selected neighbor is then subtracted from the current byte, modulo 256 unsigned. The PaethPredictor function is defined by the following pseudocode:

```
function PaethPredictor (a, b, c)
begin
; a=left, b=above, c=upper left
p:=a+b-c ;initial estimate
pa := abs(p-a) ; compute distances
pb := abs(p-b) ; to a, b, c
pc := abs(p-c)
; return nearest of a,b,c,
; breaking ties in order a,b,c.
if pa<=pb AND pa<=pc then return a
else if pb<=pc then return b
else return c
end
```

PaethPredictor must perform its computations exactly, without overflow. The order in which PaethPredictor breaks ties is important and should not be altered. This order (that's different from the one given in [Paeth 91]) is left neighbor, neighbor above, upper-left neighbor.

PNG is a single-image format, but the PNG development group has also designed an animated companion format named MNG (multiple-image network format), which is a proper superset of PNG.

> Does the world really need yet another graphics format? We believe so. GIF is no longer freely usable,... it would not be all that much easier to implement than a whole new file format. (PNG is designed to be simple to implement, with the exception of the compression engine, which would be needed in any case.) We feel that this is an excellent opportunity to design a new format that fixes some of the known limitations of GIF.
>
> —From the PNG standard, RFC 2083, 1999.

20.10 CRC

The idea of a parity bit is simple, old, and familiar to most computer practitioners. A parity bit is the simplest type of error detecting code. It increases the reliability of a group of bits by making it possible for hardware to detect certain errors that may occur when the group is stored in memory, is written on a disk, or is transmitted over communication lines between computers. A single parity bit does not render the group absolutely reliable. There are certain errors that cannot be detected with a parity bit, but experience shows that even a single parity bit can significantly increase the reliability of data transmissions.

The parity bit is computed from a group of $n-1$ bits, and then is included in the group, making it n bits long. A common example is a 7-bit ASCII code that becomes eight bits long after a parity bit is added. The parity bit p is computed by counting the number of 1's in the original group, and setting p to complete that number to either odd or even. The former is called odd parity, and the latter is called even parity.

Examples: Given the group of seven bits 1010111, the number of 1's is five, an odd number. Assuming odd parity, the value of p should be 0, leaving the total number of 1's odd. Similarly, the group 1010101 has four 1's, so its odd parity bit should also be a 1, bringing the total number of 1's to five.

Imagine a block of data where the most significant bit (MSB) of each byte is an odd parity bit, and the bytes are written vertically (Table 20.7a).

1 01101001	1 01101001	1 01101001	1 01101001
0 00001011	0 00001011	0 00001011	0 00001011
0 11110010	0 11**0**10010	0 11**0**10**1**10	0 11**0**10**1**10
0 01101110	0 01101110	0 01101110	0 01101110
1 11101101	1 11101101	1 11101101	1 11101101
1 01001110	1 01001110	1 01001110	1 01001110
0 11101001	0 11101001	0 11101001	0 11101001
1 11010111	1 11010111	1 11010111	1 11010111
			0 00011100
(a)	(b)	(c)	(d)

Table 20.7: Horizontal and Vertical Parities.

When this block is read from a disk or is received by a computer, it may contain transmission errors, errors that have been caused by imperfect hardware or by electrical interference during transmission. We can think of the parity bits as *horizontal reliability*. When the block is read, the hardware can check every byte, verifying the parity. This is done by simply counting the number of 1's in the byte. If this number is odd, the hardware assumes that the byte is good. This assumption is not always correct, since two bits may get corrupted during transmission (Table 20.7c). A single parity bit is therefore useful (Table 20.7b) but does not provide full error detection capability.

A simple way to increase the reliability of a block of data is to compute vertical parities. The block is considered to be eight vertical columns, and an odd parity bit is computed for each column (Table 20.7d). If two bits in a byte get corrupted, the horizontal parity will not catch it, but two of the vertical parities will. Even the vertical bits do not provide complete error detection capability, but they provide a simple way to significantly improve data reliability.

A CRC is a glorified vertical parity. CRC stands for Cyclical Redundancy Check (or Cyclical Redundancy Code) and it is a rule that shows how to compute the vertical check bits (they are now called check bits, not just simple parity bits) from all the bits of the data. Here is how CRC-32 (one of the many standards developed by the CCITT) is computed. The block of data is written as one long binary number. In our example this will be the 64-bit number

101101001|000001011|011110010|001101110|111101101|101001110|011101001|111010111.

The individual bits are considered the coefficients of a polynomial. In our example, this will be the degree-63 polynomial

$$\begin{aligned} P(x) &= 1\times x^{63}+0\times x^{62}+1\times x^{61}+1\times x^{60}+\cdots+1\times x^2+1\times x^1+1\times x^0 \\ &= x^{63}+x^{61}+x^{60}+\cdots+x^2+x+1. \end{aligned}$$

This polynomial is then divided by the standard CRC-32 *generating polynomial*

$$\mathrm{CRC}_{32}(x)=x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x^1+1.$$

When an integer M is divided by an integer N, the result is a quotient Q (which we disregard) and a remainder R, which is in the interval $[0, N-1]$. Similarly, when a polynomial $P(x)$ is divided by a degree-32 polynomial, the result is two polynomials, a quotient and a remainder. The remainder is a degree-31 polynomial, which means that it has 32 coefficients, each a single bit. Those 32 bits are the CRC-32 code, which is appended to the block of data as four bytes. As an example, the CRC-32 of a recent version of the file with the text of this chapter is $\texttt{586DE4FE}_{16}$.

Selecting a generating polynomial is more an art than science. Page 196 of [Tanenbaum 02] is one of several places where the interested reader can find a clear discussion of this topic.

The CRC is sometimes called the "fingerprint" of the file. Of course, since it is a 32-bit number, there are only 2^{32} different CRCs. This number equals approximately 4.3 billion, so, in theory, there may be different files with the same CRC, but in practice this is rare. The CRC is useful as an error-detecting code because it has the following properties:

1. Every bit in the data block is used to compute the CRC. This means that changing even one bit may produce a different CRC.
2. Even small changes in the data normally produce very different CRCs. Experience with CRC-32 shows that it is very rare that introducing errors in the data does not modify the CRC.
3. Any histogram of CRC-32 values for different data blocks is flat (or very close to flat). For a given data block, the probability of any of the 2^{32} possible CRCs being produced is practically the same.

Other common generating polynomials are $\mathrm{CRC}_{12}(x) = x^{12} + x^3 + x + 1$ and $\mathrm{CRC}_{16}(x) = x^{16} + x^{15} + x^2 + 1$. They generate the common CRC-12 and CRC-16 codes, which are 12 and 16 bits long, respectively.

The way you treat yourself sets the standard for others.
—Sonya Friedman