

7

Spatial Data Structures for Computer Cartography

7.1 C LANGUAGE STRUCTURES FOR CARTOGRAPHIC DATA

The Spatial Data Transfer Standard has provided cartographers with a consistent set of terminology and concepts, known as a *data model*, around which data structures can be developed. Although most data structures are expressed in published form as logical models of data organization, in computer cartographic software they find their actual implementation as computer programming data storage mechanisms, what we will call here *program data structures*. These program data structures must relate to the goals of geocoding discussed in Chapter 4,; they must relate physically to the actual methods of storage and representation of digital data discussed in Chapter 5, and they also ideally should be simple and effective to use in any programming environment.

Many of the nonstructured computer programming languages did not support sophisticated data structures. More recent languages support these structures, especially the languages most likely to be used in most contemporary computing environments: Pascal and C. In Pascal, the “record” is similar to the “structure” in C, and the advanced student will be able to translate most of the computer programs discussed here between the languages with relative ease.

In the C programming language, the “structure” is declared in the following way. We take advantage of the C construct `typedef`, which allows us to name a structure as a type of storage object or data structure (Function 7.1).

Function 7.1

```
/* Example of a C language structure declaration
/* kcc 5-93 */
typedef struct {
    float first;
    int second;
} EXAMPLE;
```

To use this structure in a program, first a variable must be declared to have this particular structure as its type (Function 7.2).

Function 7.2

```
/* Declaration of a variable to be of type EXAMPLE
/* kcc 5-93
*/
EXAMPLE hold_data;
```

Each structure element is then used by linking subelements with periods. For example, to assign a value to the first element and print the value of the second, assuming they exist, the following program lines in Function 7.3 could be used.

Function 7.3

```
/* Example use of data stored in a structure (kcc 6-93) */
EXAMPLE hold_data;
hold_data.first = 3.1415927;
hold_data.second = 1;
printf("The first element of the structure is %f\n",
      hold_data.first);
printf("The second element of the structure is %d\n",
      hold_data.second);
```

A structure can be multidimensional, as in the example Function 7.4.

Function 7.4

```
/* Example declaration of a multidimensional variable to be
of type structure kcc 6-93 */
typedef struct
{
    float first; int second;
} EXAMPLE;
EXAMPLE hold_data[5];
for (i = 0; i < 5; i++)
{
    printf("Structure element one sub %d = %f", i,
          hold_data[i].first);
    printf("Structure element two sub %d = %d", i,
          hold_data[i].second);
}
```

FORTRAN programmers should note that C language arrays all start at a count of zero. The final concept for C language structures is that a structure can be part of the declaration of another structure. An example of this is shown in Function 7.5.

Function 7.5

```
/* Example use of the POINT data structure kcc 6-93
/* The first statement states that structure definitions
/* are in cart_obj.h in this directory */
#include "cart_obj.h"
/* Now declare a structure point, with 50 elements of type
    POINT */
POINT Point[50];
/* Print, for example, the 30th elements of point (indexed
    by 29) */
printf("(x,y) of 30th element is (%f,%f)\n",Point[29].x,
        Point[29].y);
```

The use of structures permits a high degree of organization in computer programs, especially because structures can be passed as arguments between functions or can be declared outside the function definitions to make them available anywhere within a program. To achieve this characteristic, somewhat similar to using COMMON blocks in FORTRAN, the structure definition can be placed together in a separate file. C supports such declarations, using the term *header file* for these files.

In the following declarations and functions, it is assumed that there exists a file called `cart_obj.h` that consists of declarations of the C language data structures to be used in this book. In each case, the structures are given uppercase names, so that when variables are declared to have this type of structure, they can use the same name in lowercase without confusion.

For example, the first declaration defines the zero-dimensional primitive cartographic object POINT. To declare a variable to be of type POINT, we can still call it `Point` (mixedcase, because GKS reserves some lowercase only words such as `link`) and perhaps give it multiple instances.

Thus a collection of points, a maximum of 50 for example, could be stored and used in a C language structure `Point` as shown in Function 7.5. Because the digital cartographic data standards divide the major cartographic object definitions into zero, one, and two dimensions, the same subdivision will be followed for the C language structures.

The following sections outline C language structures suitable for storing cartographic or spatial objects of zero, one and two dimensions. The purpose of these functions is to encourage the use in computer programs of data stored in a manner consistent with the SDTS data model. In each case, the objects are presented along with programming data structures which will hold data, and in some cases, example programs are given that use the structures to perform geographic operations.

7.2 ZERO-DIMENSIONAL CARTOGRAPHIC OBJECTS

The primitive object with one dimension is the *point*. SDTS specifies a point as a zero-dimensional object that specifies geometric location. The location is specified by a coordinate pair (x, y) or by a triplet (x, y, z) . The rules for providing coordinates are discussed in the standards document in Appendix C “Spatial Address Encoding.” A point is inherently a vector data element. Many program applications require the coordinates to be floating point numbers (for example latitude and longitude in decimal degrees). On the other hand, storing two or three floating point values for every location requires significantly more storage and computational power than using integers.

To accommodate both these formats all of the following functions use the C keyword `typedef`, which leaves the choice up to the programmer. It is important to remember in algorithms and programs which type is used. In the program examples in this book, neither type is assumed, and all programs perform type conversions where necessary. Careful attention should also be paid to whether negative values are appropriate, what the maximum values are likely to be, and whether there is a loss of precision involved.

Three special cases of a point are given, allowing the point to store an attribute associated with it, either an *entity*, a *label*, or an *area* identifier. The final zero-dimensional object is the *node*. The node is defined as a zero-dimensional object, that is, a topological junction of two or more links or chains, or an end-point of a link or chain. Examples are shown in Figure 7.1.

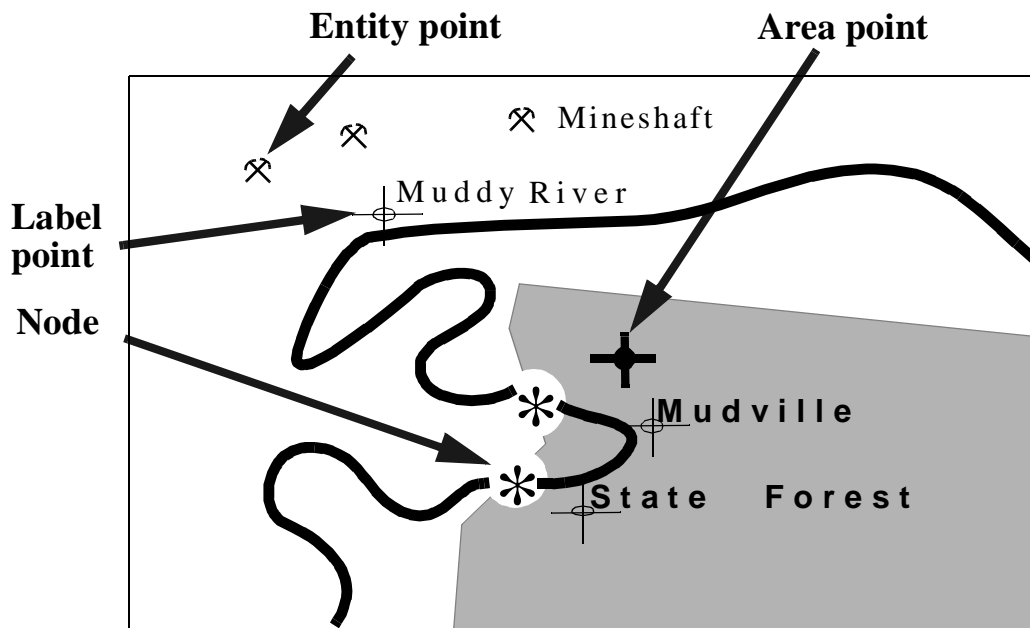


Figure 7.1 Zero-dimensional cartographic and geographic objects.

Function 7.6

```

/* -----
/* C language cartographic data structures
/* Based on FIPS 173 Terminology and data model
/* kcc 3-94 Revised from 5-88
/* -----
/* Define the constants for all objects */
#define MAXPTS 999
#define MAXROW 20
#define MAXCOL 20
#define MAXSTRING 400
#define MAXARC 100
#define MAXLINK 100
#define MAXCHAIN 100
#define MAXLABEL 20
#define MAXHOLE 4

/* Establish the coordinate type */
typedef coordinate {float | int }; /* NOT C, choose one */
/* Zero dimensional objects */
typedef struct {
    coordinate x;
    coordinate y; } POINT ;
typedef struct {
    char entity_type[MAXLABEL];
    char entity_attribute[MAXLABEL];
    char included_term[MAXLABEL];
    POINT Point;
} ENTITY_POINT;
typedef struct {
    char label[MAXLABEL];
    POINT Point;
} LABEL_POINT;
typedef struct {
    int polygon_id;
    POINT Point;
} AREA_POINT;
typedef struct {
    int node_identifier;
    POINT Node;
} NODE ;

```

The normal means by which structure definitions are made in C language programs is by placing them in a header file. The header files can then be included in each function that uses the structure, and the data are available to all functions in the program. A generic header file is built in this section to cover each of the objects defined in the Spatial Data Transfer Standards. This header file (Function 7.6 and continued as Functions 7.10 and 7.12) is called `cart_obj.h`, for the cartographic objects header file.

In the `cart_obj.h` header file, the initial set of statements (the `#defines`) allow the preprocessor to assign any constants to be used in the structure declarations. For example, instead of allowing a fixed value of 20 as the maximum number of points, the constant `MAXPTS` is used and is simply switched at compile time for the constant 20 by the C preprocessor. Because only those constants and structures which the program will use are loaded at run time, including the whole `cart_obj.h` file incurs no extra overhead.

An example of a point has already been given above. An entity point may contain a feature type of "church", identified in the feature code section of the data standards as entity type `BUILDING` (a permanent walled and roofed construction), included term `Church` (which has the attribute of `EXISTING`), for example at 1 degree 30 minutes East, 55 degrees 20 minutes and 45 seconds North.

Function 7.7a

```
/* Program segment to demonstrate an entity point
/* structure kcc 6-93 Revised from 12-88 funct 7.7a */
#include "cart_obj.h"
(void) show_entity_point()
{
    ENTITY_POINT Feature;
    Feature.entity_type = "BUILDING";
    Feature.entity_attribute = "EXISTING";
    Feature.included_term = "Church";

    Feature.Point.x = (coordinate) 1.3000;
    Feature.Point.y = (coordinate) 55.2045;
    /* Note DD.MMSS Global coord. format */

    (void) printf("Entity type %s Included term %s with
        attributes %s is located at %f %f\n",
        Feature.entity_type, Feature.entity_attributes,
        Feature.included_terms, Feature.Point.x,
        Feature.Point.y);
    return;
}
```

The declaration and assignments shown in Function 7.7a would store this information. Similarly, the same church may have a specific name to be placed next to a symbol for a church on a map, such as “St. Paul’s Church”. In this case, a label point should be used (Function 7.7b).

Function 7.7b

```
/* Program segment to demonstrate a label point
/* structure kcc 4-94 Revised from 12-88 funct 7.7b */
#include "cart_obj.h"
(void) show_label_point()
{

    LABEL_POINT Feature;
    Feature.label = "St. Paul's Church";

    Feature.Point.x = (coordinate) 1.3000;
    Feature.Point.y = (coordinate) 55.2045;
    /* Note DD.MMSS Global coord. format */

    (void) printf("Label Point: Label %s is located \
        at %f %f\n",
        Feature.label, Feature.point.x, Feature.point.y);
    return;
}
```

In the case of the area point, the identifier is a pointer to a polygon which is associated with the point. For example, the pointer could be to a polygon defining the boundaries of the church parish. Similarly, the node simply defines the location and gives an identifier so that the node can be found by other, higher-dimensional objects. The first encounter within a computer program in the C language with these structures is when the data are read from files. We will use ASCII files throughout this discussion, although very much faster input and output to files is possible if binary reads and writes are used. As noted above, ASCII data files can be edited, examined, modified, and corrected with the use of your favorite text editor. A function to read a single file containing a large number of

Function 7.8

```
File:    point_data
Format:  ASCII

1.3000    55.0000    St. Paul's Church
2.1023    56.1012    Water Tower
-1.1213    57.1123    Bench Mark (45.4 meters)
-1.1557    55.5927    Radio Tower
2.1519    54.1314    Radar Reflector
```

point features with their labels follows. Assume that the file called `point_data` as shown in Function 7.8 exists in the users current directory. Function 7.9 will load the data into the structure `label_point`. This function, `read_label_points()`, should be declared in the calling function to be of type `int`. The function returns the number of points read from the file via the `return` statement, plus the structure containing the data through the argument list. The calling function should provide in the argument list the address of a structure ready to receive the data.

Function 7.9

```

/* Read a file of latitudes and longitudes in DDD.MMSS
/* format with point labels and load the data
/* into a structure of label points
/* kcc      4-94 revised from 12-88 funct 7.9 */
#include <stdio.h>
#include "cart_obj.h"
int read_label_points(label_point)
    LABEL_POINT Label_point[];
    {
        int i=0, number_of_points;
        FILE *file_pointer;
        /* Open the input file and abort if non-existing */
        if ((file_pointer = fopen("point_data", "r")) == NULL)
            { printf("Unable to open file 'point_data'\n");
              exit(); }
        /* Read the point data, counting entries */
        while (fscanf(file_pointer, "%f%f%s",
            &Label_point[i]->Point->x,
            &Label_point[i]->Point->y,
            Label_point[i]->label) != EOF) i++;
        number_of_points = i;
        /* Close the file */ fclose (file_pointer);
        /* Return the number of points */
        return (number_of_points);
    }

```

7.3 ONE-DIMENSIONAL CARTOGRAPHIC OBJECTS

The one-dimensional cartographic objects are more varied, serving a large number of cartographic needs (Figure 7.2). The generic name for a one-dimensional object is a *line*, which can consist either of a locus of points defined by a function such as a polynomial or a b-spline (an *arc*) or as a sequence of connected primitive objects known as line segments. The *line segment* is simply a straight line connecting two points, and we define it to be simply two points stored together.

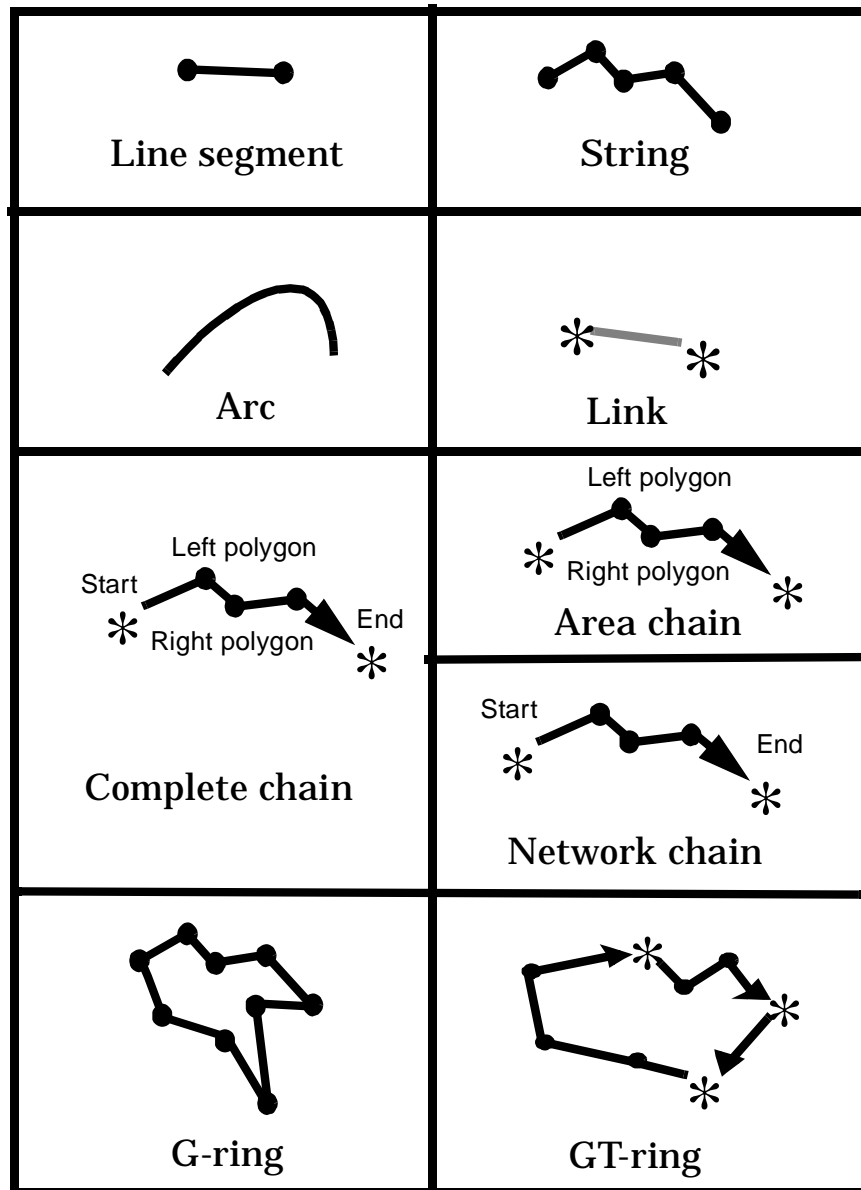


Figure 7.2 One-dimensional cartographic and geographic objects.

Function 7.10

```

/* -----
/* C language cartographic data structures
/* Based on FIPS 173 Terminology and data model
/* kcc 6-93 Revised from 5-88 function 5.10 (7.7 Ctd.)
/* ----- */
/* One-dimensional objects */
typedef struct {
    POINT Point[2]; } LINE_SEGMENT;
typedef struct {
    int number_of_points;
    POINT Point[MAXPTS]; } STRING ;
typedef struct {
    int arc_identifier;
    STRING String;
    int *function; } ARC;
typedef struct {
    NODE Node[2]; } LINK;
typedef struct {
    NODE Node[2];
    int left_polygon, right_polygon;
    STRING Chain;} COMPLETE_CHAIN;
typedef struct {
    NODE start;
    NODE end;
    int left_polygon, right_polygon;
    STRING Chain;} AREA_CHAIN;
typedef struct {
    NODE start;
    NODE end;
    STRING Chain;} NETWORK_CHAIN;
typedef struct {
    int number_of_strings, number_of_arcs;
    STRING String[MAX_STRING];
    ARC Arc[MAX_ARC]; } G_RING;
typedef struct {
    int number_of_chains;
    COMPLETE_CHAIN Chain[MAX_CHAIN]; } GT_RING;
typedef struct {
    STRING Ring;} RING;

```

Function 6.11

```

/*
/* Return a selected line segment from a string
/* kcc 6-93 Revised from 5-88 funct 6.11 */
#include "cart_obj.h"
void get_segment_from_string (line_segment, line, seg_num)
struct LINE_SEGMENT *line_segment;
struct STRING *line;
int seg_num;
{
/* Assumes that seg_num is segment connecting n-th with n+1-
th point in string */
    if (seg_num+1 >= MAXPTS)
        {printf("Error in get_segment_from_string(), segment\
is too long for string\n");
        exit();}
    line_segment->point[0]->x = line->point[seg_num]->x;
    line_segment->point[0]->y = line->point[seg_num]->y;
    line_segment->point[1]->x = line->point[seg_num+1]->x;
    line_segment->point[1]->y = line->point[seg_num+1]->y;
    return;
}

```

The *string*, a group of connected segments, without any other topological information, makes up the building block for the more complex one-dimensional objects. When a string closes upon itself, it is termed a *ring*, and rings can be created from strings, arcs, links, or chains. The *link* is a network primitive, consisting of a single edge of a connected graph. Direction within a link is simply by the sequence of points as stored.

This is a complex group of objects, and there is a need to break the list down. A simple division is by those objects that store topology, and those which are designed largely simply to reflect geometry. Traditionally, cartography has been concerned with using geometry-only objects to represent cartographic data which is destined solely for display. Nevertheless, topology is often required for consistency checking, sequenced color fills, and symbolization. Topology is also required for some data structure conversions. The geometric objects of one dimension are the line segment, the string, and rings of arcs and strings.

To read a string, assume a file containing data in the format of one string per line, with the first record being the number of points in the string (*n*), and the next records being the coordinate pairs, as (*x*₁, *y*₁), (*x*₂, *y*₂), (*x*₃, *y*₃) up to (*x*_{*n*}, *y*_{*n*}). Such a file, containing the decimal latitudes and longitudes of the outline of North America, is contained on the companion disk in the file *namerica*. Similarly, a G-ring is identical to the string except that the first and last point *x* and *y* values match exactly and that the string may contain arcs.

Function 7.11 shows the two functions that perform the task of reading a string. The

function `get_pts()` reads a single string into a structure of type `STRING` and assumes that the header file `cart_obj.h` both defines `STRING` and declares `MAXSTRINGS`. Here inclusion comes by being in the same file. The actual structure containing each string, called `line`, is declared here outside the brace defining the function. The function `read_strings` makes successive function calls to `get_pts()`, each time storing an additional `STRING` structure into `line`. When the function finds the `end_of_file` marker, the function `read_strings()` returns the number of successfully stored lines in the structure.

The same function will work for a G-ring, but a check should be implemented to ensure closure and to ensure that no arcs are involved. The primary purpose for a G-ring is for either the computation of an area or the coloring, shading, or filling of the area delimited by the G-ring with a pattern.

To read an arc, the user should first read an identifier for the arc, next read a string in the same way as in function `get_pts()`, and finally, read a pointer to a function. The user must determine how the function's parameters are stored. For example, if we are limited to circular arcs, we could store the (x, y) of the circle center, the circle radius, followed by the starting and ending circular arc bearing, clockwise from north, and the number of points to be generated along the arc. Thus four floating point numbers, one coordinate pair, and one integer would be needed to define the arc. The `STRING` part of the structure can then hold the points that are generated, at any given increment.



Function 7.11

```

/*=====
* Read a string from a file into a structure
* kcc 6-93 funct 7.11
*===== */
#include <stdio.h>
#include "cart_obj.h"
#define INPUT_FILE "namerica"
/* read_strings : reads strings into structure */
STRING Line[MAXSTRINGS];
int read_strings()
{
    STRING get_pts();
    int i, j, number_of_lines = 0;
    FILE *fp;
    fp = fopen(INPUT_FILE, "r");
    while (!feof(fp)) {
        Line[number_of_lines++] = get_pts(fp);};
    number_of_lines--;
    (void) printf("%d lines read from %s\n",
        number_of_lines, INPUT_FILE);
    return (number_of_lines);
}

/* get_pts : reads points for one string */
STRING get_pts(fp)
    FILE *fp;
{
    STRING get_line;
    int i, j;
    (void) fscanf(fp, "%d%*1c",&get_line.number_of_points);
    for (i = 0, j = 1; i < get_line.number_of_points; i++) {
        fscanf(fp, "%f%f", &get_line.Point[i].y,
            &get_line.Point[i].x);
        if (j++ == 6) {j = 1; (void) fscanf(fp, "%*1c");}
    }
    if (j != 1) fscanf(fp, "%*1c");
    return (get_line);
}

```

The RING structure in Function 7.10 is not explicitly part of the SDTS definitions, but is included here because by far the majority of simple applications use G_RINGS consisting of only one string to define a topology-less polygon. Storing multiple strings and arcs for every instance of a G_RING would use an unnecessarily large amount of storage in these cases. The burden of ensuring closure is then left to the user. Closure in the RING can be implicit (that is, the software will duplicate the first point as the last) or required.

The remainder of the one-dimensional objects are designed to store both topology and geometry. These are the link, the complete chain (called simply a chain here), the area chain, the network chain, and the GT-ring. The latter objects are included as components of aggregate spatial objects, considered in Section 7.4. The simplest of these is the link.

A link consists simply of two nodes, stored here as a line segment with an implied ordering by the order of storage. A very common application of the link is to show traffic flow. A traffic flow network in Manhattan has a series of links, because each street from intersection to intersection is a straight line segment. Where streets curve, as in Central Park, the network chain would replace the link.

Taken together, the whole street network would be described by the network, one of the aggregate spatial objects defined in the Spatial Data Transfer Standards. For such a network, the geometry is important for computing lengths and drawing the map, but much of the topology can be computed simply from the endpoints of the network chains stored separately as links.

The various versions of the chain, other than the network chain, are usually encountered in applications that plot chain by chain rather than string by string or polygon by polygon. Although it is possible to check for topological continuity using the chain structures, their format is inherently one-dimensional; that is, it says little about the areas enclosed by the chains.

In the case of the GT-ring, it is known that the chains connect together perfectly to produce a ring, a property that differs from the G-ring because a string component of a G-ring does not have to be free of crossovers or intersections by definition. This sort of distinction is often important in digitizing, where endpoints are snapped together to maintain connectivity along chains. Usually, the digitizer operator must separately signal a chain beginning and end, perhaps by pressing different cursor buttons, and must sometimes enter connecting chain information from the keyboard.

7.4 TWO-DIMENSIONAL CARTOGRAPHIC OBJECTS

Two-dimensional objects under the final version of the Spatial Data Transfer Standards differ to the greatest extent from the draft version of the standard in 1988. An *area* is defined as a bounded, continuous, two-dimensional object that may or may not include its boundary. The defined two-dimensional objects are the *interior area*, the *G-polygon*, the *GT-polygon*, two special case polygons (*void* and *universe*), and finally the raster objects the *pixel*, and the *grid cell* (Figure 7.3).

The interior area is an area excluding its boundary. This area is filled when a polygon is colored or shaded. It is also the area within which some algorithms, such as point-in-polygon tests, provide a simple solution. The G-polygon is defined as an area consisting of an interior area, one outer ring, and zero or more nonnested inner rings. The ring

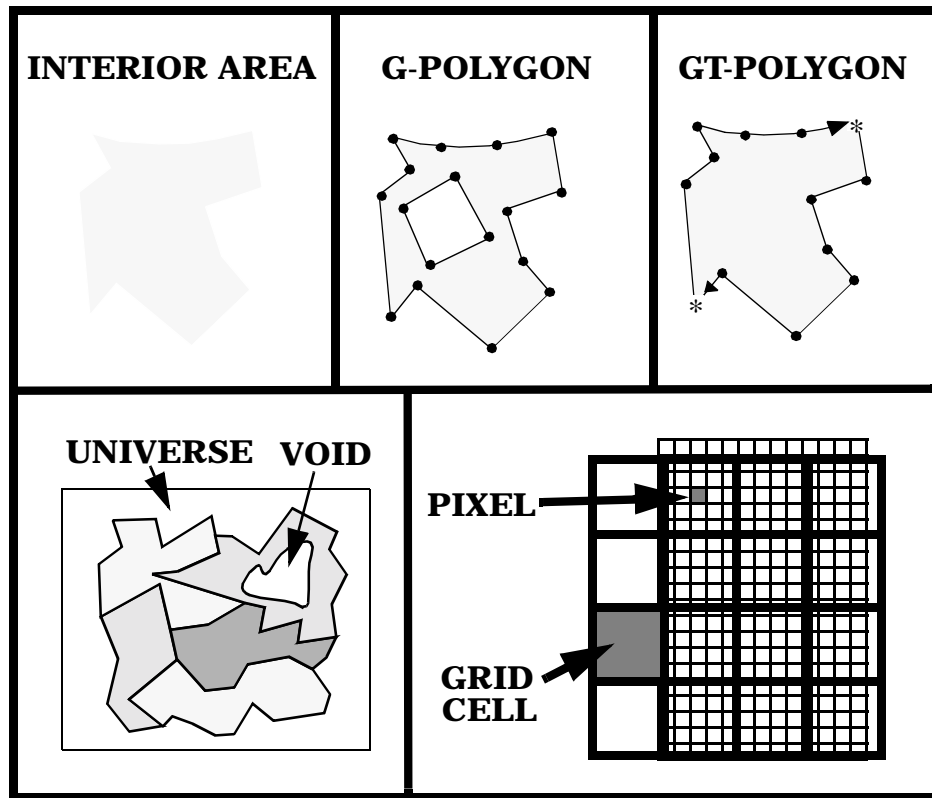


Figure 7.3 Two-dimensional cartographic and geographic objects.

is the standard representation of the polygon for shading, area computation, and filling in most cartographic applications.

The GT-polygon is similar to the G-polygon, with the exception that the boundary is topologically defined as a GT-ring built from chains rather than simply a G-ring. Two degenerate forms of the polygon are the *universe* polygon and the *void* polygon. The universe polygon makes up the area within the bounding area of the map's extent, but it is outside the area for which attributes and objects are defined. On maps, this area is often bounding countries, states, oceans, and so forth. The void polygon is an area without attributes within the figure of the map. Thus a map of North America may consider the Great Lakes as void polygons, especially if the map depicts a land-related variable such as population density or land use.

The two final two-dimensional cartographic objects accord well with the GKS standard for graphics and are designed to facilitate the handling of raster and grid data. The *pixel* is a picture element consisting of the smallest nondivisible element of an aggregate object called a digital image. The grid cell is the final object and is an element of a regular or nearly regular tessellation of a surface, an aggregate object called a grid. Thus for DEM data, for example, the grid cell is an elevation observation spaced 30 meters apart

Function 7.12

```

/* -----
/* C language cartographic data structures
/* Based on FIPS 173 Terminology and data model
/* kcc 6-93 Revised from 5-88
/* function 7.12 (Continues 7.10)
/* ----- */
typedef struct {
    G_RING interior;
} INTERIOR_AREA;
typedef struct {
    int polygon_identifier;
    char polygon_descriptor[MAXLABEL];
    struct G_RING Boundary;
    int number_of_holes;
    struct G_RING Hole[MAXHOLES];
} G_POLYGON;
typedef struct {
    int polygon_identifier;
    char polygon_descriptor[MAXLABEL];
    struct GT_RING Boundary[MAXCHAIN];
} GT_POLYGON;
typedef struct {
    char grid_descriptor[MAXLABEL];
    int nrows, ncols;
    POINT corners[4];
    int datum;
    int z[MAXROW][MAXCOL];
} GRID;
typedef struct {
    int nrows, ncols;
    unsigned char *pixels; /* Assumes 8 bit data */
} IMAGE;

```

from its neighbors on a square tessellation, while multiple pixels could be in use in a digital map to symbolize the grid cell. The two-dimensional objects can be represented by the C language programming data structures shown in Function 7.12, which continues the definition of the `cart_obj.h` header file mentioned previously.

The interior area is represented as a G-ring. Algorithms for computing area and testing a point for inclusion within an area or on the boundary are considered in Chapter 10.

The G-polygon structure represents a normal nontopologically constructed polygon. This structure contains an identifier, a descriptor (such as the label “New York State”), and a single structure of type `G_ring` to contain the boundary. In addition, the number of holes must be stored, and the `G_RING` structure is multidimensional to store the holes.

A maximum number of holes must be specified, stored in `cart_obj.h` as `MAX-HOLES`. We could use C’s pointer capabilities to store holes, that is, to store within the structure a pointer to a structure of type `G_RING` that constitutes a hole. Although such a structure is elegant and allows complex polygons to contain any number of holes without wasting storage, this addition will be left to the more advanced C programmer. The pixel, being largely device dependent, is excluded from the list of structures. This is because the GKS standard provides a means for symbolizing a grid without reference to pixels.

The remaining structure is the *grid*. The grid has as its elements a grid descriptor (which could be the header lines from the USGS DEM data files), `POINT` references to the ground map coordinates of the four grid corners (which could, for example, be UTM values) and a two-dimensional array of integers to store the values. Elevations on land range from 400 meters below sea level on the shore of the Dead Sea to 8,846 meters at the summit of Mount Everest.

In case the range of values required does not fit into the range of an integer, the datum value is provided. This integer value should be added to the grid value to give the actual elevation. For most DEM data, this is zero, but for ocean depths, elevations on other planets, and other geographic variables, other values may be necessary.

Although the grid data structure is a logical construct for the grid, for practicality, the data in a grid are structured independently of the map reference information. Variants on the grid structure that conserve the significant amount of storage necessary when an array is allocated in computer memory are possible. When a grid data set is detached from its spatial frame of reference, it can be called an *image*, a term not used in the Spatial Data Transfer Standards.

An image can be of varying size and needs to be able to store different values in each pixel (note that grids have grid cells; images have pixels). An image with a given type can have its storage size precomputed, and then allocation can be made only for the space it requires. The data structure here is borrowed from Myler and Weeks (1993) to which the interested reader is referred for a cornucopia of image-based algorithms, including the reading and writing of several industry-standard image formats and basic and advanced image-processing algorithms.

The value of using the structure of an image is that allocation of image space can be made dynamically within a program, and that types of different sizes can be accommodated. The type of the pixel need not always be an integer, which often uses far more storage than is necessary. Common types are `short`, `unsigned int` and `unsigned char`. For example, a grid’s elevations can be allocated to an image in the following way (Function 7.13).

Function 7.13

```

/* Function grid_to_image: Allocate a set of grid cells
to an image function 7.13 kcc 4-94 */
IMAGE grid_to_image (grid)
GRID grid;
{
    IMAGE *image;
    int i, j;
    if (image = malloc (2 * sizeof (int) +
        (grid.nrows * grid.ncols *
        sizeof (*image->pixels))) == NULL) {
        printf ("Unable to allocate memory in function \
        grid_to_image()\n");
        return(1)
    }
    /* Image allocated dynamically, now copy grid to image */
    image->nrows = grid.nrows;
    image->ncols = grid.ncols;
    for (i=0; i< grid.nrows;i++ ) {
        for (j=0; j<grid.ncols;j++)
            *(image->pixels++) = grid.z[i][j];
    }
    return (image);
}

```

7.5 DATA STORAGE AND DATA STRUCTURES

At least one major problem with the data standards is the lack of three-dimensional cartographic objects. Fortunately, a small set of cartographic data structures for three-dimensional objects exists, covered in Chapter 13. In CADD, and in computer graphics generally, data structured for three-dimensional objects are a vital part of their symbolization or rendering as pictures. The GKS graphics standard has been extended to three-dimensional graphics (GKS-3D), which may tend to favor particular three-dimensional data structures in the future.

The C language structures introduced in this chapter are used extensively in the chapters that follow. In Chapter 8, they are used to give examples of the data structures in common use, while in Chapters 10, 11, and 12 these basic programming structures are built upon with transformational algorithms and symbolization methods using GKS. Although their understanding is not critical to students of computer cartography, students of analytical cartography are strongly encouraged to use them to experiment with their own software for computer and analytical cartography.

7.6 REFERENCES

- Department of Commerce (1992). *Spatial Data Transfer Standard*. FIPS PUB 173, Federal Information Processing Standards Publication, Computer Systems Laboratory, National Institute of Standards and Technology, Gaithersburg, MD.
- Guptill, S. C. ed. (1990). *An Enhanced Digital Line Graph Design*. Department of the Interior, U.S. Geological Survey Circular 1048. Washington, DC: U.S. Government Printing Office.
- Myler, H. R. and A. R. Weeks (1993). *The Pocket Handbook of Image Processing Algorithms in C*. Englewood Cliffs, NJ: Prentice Hall.
- National Committee for Digital Cartographic Data Standards (1988). "The Proposed Standard for Digital Cartographic Data." *American Cartographer*, vol. 15, no. 1, pp. 9–142.