

Map Transformations

11.1 TRANSFORMATIONS OF OBJECT DIMENSION

In the Chapter 10, we met the transformational view of cartography. Two types of transformations were considered: first, transformations between the types of data and types of map involved in the mapping process; and second, transformations between map scales. A third type of transformation is the transformation of cartographic objects themselves via their dimensions. Because these are transformations of either the locational geometry of the data or of the structure used to represent a cartographic object spatially, they deal directly with the core of map information and are close to the heart of analytical cartography.

We have already noted the contribution of Tobler's transformational view (Tobler, 1979). Tobler saw cartographic object dimensions as states for cartographic data, and he enlarged the set of state transformations beyond the classical point-based cartographic conversions to include the other dimensions with which we specify locations. He envisaged a three-by-three table—of points, lines, and areas by points, lines, and areas—that contained nine possible groups of transforms, with many specific cases and divisions in each category. With volume added as a cartographic measurement dimension, we have 16 possible sets of transformations, which can be arranged in a four-by-four table. Tobler saw this table as a transformation matrix, a set of possible transformations between states at different times. Within this viewpoint, time 0 and time 1 are states, and the cartographic data go through a transformation between states (Figure 11.1).

Changes within this transformation matrix along the diagonal are transformations either between scales or between data structures. The final transformation as far as map production is concerned is that between the cartographic data and a map, or the symbolization transformation. This chapter will focus upon the dimensional transformation, and finish by discussing the symbolization transformation. Chapter 12 follows up specifically with a more in-depth discussion of scale transformations and data structure conversions.





















































		STATE AT TIME ONE			
		 Point	 Line	 Area	 Volume
STATE AT TIME ZERO	Point	  	  	  	  
	Line	  	  	  	  
	Area	  	  	  	  
	Volume	  	  	  	  

Figure 11.1 The 16 possible single step dimensional transformations.

Using matrix algebra to express transformations, we can express a sequence of transformations as matrix multiplication transformations of a vector of (x, y) coordinates, X:

$$X = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_n & y_n \end{bmatrix}$$

$$XT = X' \qquad \text{Normal transformation.}$$

$$X'T^{-1} = X + E \qquad \text{Inverse transformation.}$$

$$TT^{-1} = E \qquad TT^{-1} = I \qquad \text{Transformation both with and without error.}$$

Under normal circumstances, the cartographic transformation is imperfectly invertible, due to error in the transformation process. The transformation, times its own inverse, gives not the ideal identity matrix (*I*), but a vector of errors *E*. Analytical cartography seeks to identify, model, eliminate, or reduce the matrix *E* so that its spatial effects are known and measureable or negligible.

What we would like to be able to do as analytical cartographers is to express a cartographic transformation either as an explicit mathematical operation such as the matrix formulations above, or as an algorithm that allows us to project state 0 onto state 1 in a fully described way. The transformation from state 0 to state 1 has an inverse transformation that transforms from state 1 to state 2, equivalent to state 0. If such a transformation exists, the entire transformation belongs to a special subset of transformations known as invertible transformations. Knowledge of invertible transformations allows prediction of error and its detection and elimination, understanding of the underlying geographic phenomena, and spatial modeling of the real processes behind the cartographic data.

In this chapter, we shall deal with transformations at the low-dimensional end of the transformation matrix, especially point-to-point transformations. Most of these are transformations within a single type but between scales, and of attribute data scaling, usually known as map generalization. Some of them have as their input a cartographic line and as their output a measurement based upon properties of the line. Not all are invertible transformations.

Point-to-point transformations, which Tobler called the classical cartographic transformations, are very central to analytical cartography. An understanding of point-to-point transformations is essential to analytical cartography, more essential than many of the other dimensional transformations. This is because most higher-dimensional cartographic objects can be reduced to a set of locational measurements at points.

Map-based transformations are pivotal to understanding the ability of analytical cartography to make systematic inquiries into how data structures representing cartographic objects encode and represent space. Analytical cartography seeks to determine how the geographic properties of space sharing can be used in analysis, modeling, and prediction.

The transformation matrix shown in Figure 11.1 shows only the single-step transformations. It is possible also to have multiple-step transformations, and these are indeed common. As an example, if we measure elevations in the field, it is most convenient to record them at sets of irregularly distributed points. We can describe these points digitally as point locations with the attribute of elevation.

The data value we have sampled, topography, is volumetric, we have sampled only the upper surface with sets of point data observations. For example, we could take a continuous value, distributed over a volume surface, sample the value using a series of points with attribute values associated with them, and then transform the attributes of the points over space using a TIN data structure (Figure 11.2).

Our motive may be, perhaps, that we want to use an automated contouring technique requiring data in the TIN structure. We transform from state 0 to state 1, and then on to state 2. State 1 to state 2 is a point-to-area transformation, where the first set of points is irregularly spaced and the set maps onto a set of triangles forming a triangulated irregular network. We might take the set of triangles and use it to feed a set of lines through the area to make contour lines. So we have input a series of points, transformed them to areas, and then transformed them to a set of lines. We then draw the lines on the map as contours and use them in the map reader transformation to communicate the impression of topography, the upper surface of a volume.

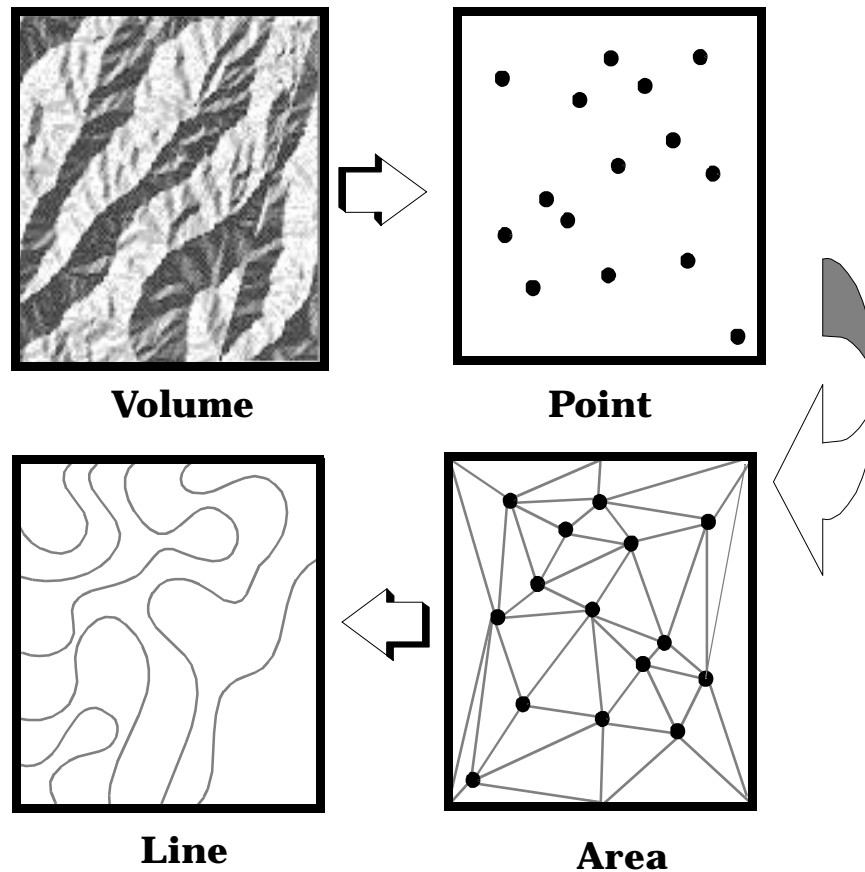


Figure 11.2 Example of a multistep transformation. Volume (terrain) sampled at points, converted to TIN (areas) then contoured (lines).

Using this transformational view of cartography, we have a versatile framework for analytical cartography, and we can place any kind of cartographic data manipulation into this framework. In the remainder of this chapter, we will take one set of transformations and start working through specific examples. These are transformations of the *location measurements* themselves, in other words, coordinate transformations. This type of transformation changes the geographic space of the base map itself. For all maps, but especially small-scale maps, with such transformations we are mainly talking about *map projection* transformations.

11.2 MAP PROJECTION TRANSFORMATIONS

We can have three approaches to transforming the coordinates of small-scale maps. The simplest approach is to ignore the problem of the earth's shape, and this is a frequently employed solution to the problem. If the area of mapping interest is small, such as an area that could be paced or taped out, or surveyed across a construction site, then the simple plane is a reasonable approximation of the geometry of the object to be mapped.

Even at large scales, however, ignoring the projection results in some unexpected, inaccurate spatial statistics, and in poor maps, especially if we take maps that have been prepared on different map projections and then overlay them. Simply mapping 1-degree by 1-degree cells onto a single display square seems to work fine at low and middle latitudes, but as we approach the poles, the north-south distortion becomes sufficient to make computations of areas, directions, and bearings meaningless.

The second approach to coordinate transformations is to use statistical techniques such as rubber sheeting or adjustment. Surveyors have applied adjustments to locational measurements for many years, distributing the random error of measurement proportionately around points and along transects. In remote sensing, images are transformed into map space using the same principle, that of a least squares fit of the image to the map geometry. Under ideal circumstances we would know every geometric property of the map and understand the relationship of these geometric properties to latitude and longitude. This would allow us to know all the equations to perform the space transformation. This is rarely the case.

Rubber sheeting is a statistical and empirical approach. It works, it is frequently used, and it is acceptable cartographically as long as we realize that we have incorporated systematic error into our final map and that we have lost some of the cartographic fidelity. Of course, because this method is empirical the transformation is not directly invertible, although it often comes close to it if we retain the control point locations.

The third approach is to compute the characteristics of the transformation itself geometrically. There are two ways of doing this; we can be precise, or we can be approximate. Which we choose depends largely upon what we want to do with the data. For example, we can choose any one of three geometric forms to model the earth's surface: a sphere, an oblate ellipsoid, or a geoid.

The sphere is most familiar to us as the common globe and is perhaps the easiest to understand. Numerous estimates of the radius of the sphere have been made, and are used in mapping specific areas. The oblate ellipsoid is a more precise model, allowing higher levels of accuracy. An oblate ellipsoid is the volume traced out by an ellipse rotated about its minor axis. Because the earth is about 42 kilometers fatter across the equator than it is pole to pole, the difference from a sphere is small.

A common estimate of the degree of flattening is 1:294.98, an estimate established in 1866 by Alexander Ross Clarke and still normally used for the mapping of the United States, although it will be phased out as the 1983 North American Datum (NAD83) becomes the standard. Because it was established that different figures could be computed for different parts of the globe, a "best-fit" ellipsoid was chosen and is given as the 1980 Geodetic Reference System value of 1:298.257, the basis of the NAD83 (Snyder, 1983).

The ellipsoid is important, because when maps are referenced to it directly they are more accurate and capable of higher precision. The third earth model is even more precise. In this model, we fit an empirical surface to the earth's gravity field, which follows mountain ranges and even dips below and above sea level. How far this deviates from the ellipsoid, or usually the local best-fit ellipsoid, depends on where we are, giving us a lumpy ellipsoid-like figure known as a geoid. How precise we need to get depends on our function. A map of U.S. population clearly does not need to be precise beyond the best-fit sphere, while the guidance system of a spacecraft needs much more accurate and precise calculations.

Cartography traditionally has worked with the sphere because the manual construction of map projections is tedious. The sphere is simplistic, but it makes the calculations and the construction of projections using drafting instruments much easier. Map projections usually can be described as projections from a point source on a sphere to points on a flat plane. Map projections involve two different types of mappings, both of which benefit from use of the computer. Figure 11.3 shows a representation of a sphere and the various types of possible mappings of points on the sphere onto points on a map.

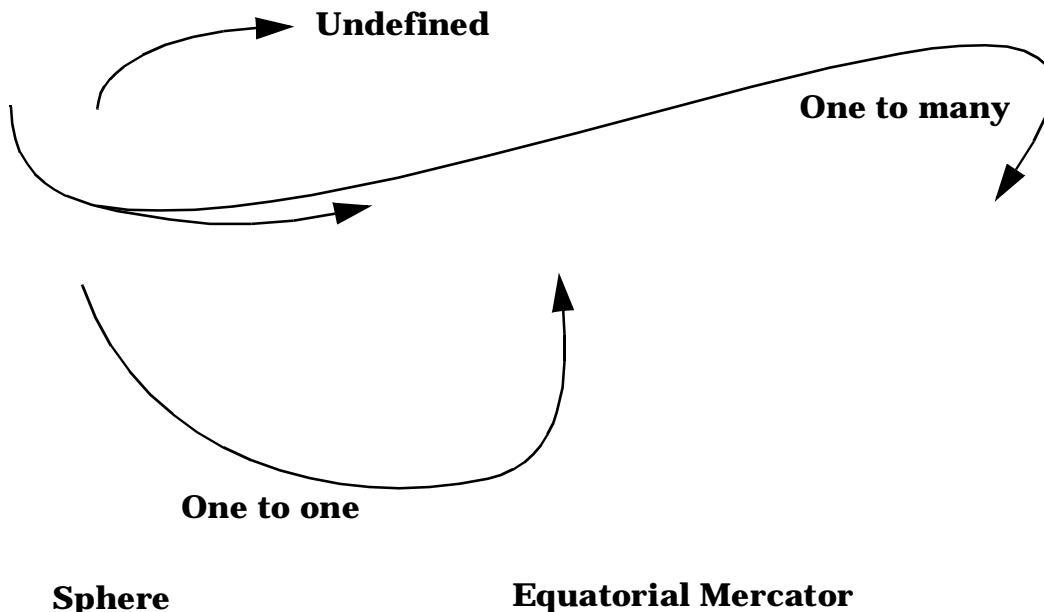


Figure 11.3 Mapping points on a sphere to points on a plane.

Each point on the earth's surface maps onto a single point on the map. Geometry actually uses the term *mapping* to describe this process, borrowed from cartography's ancient Greek roots. This is usually called a *one-to-one mapping*, and it is preferred because it is well behaved and simply invertible. Imagine an old favorite, Gerhardus Mercator's projection of the earth, one we are all familiar with from the wall of our elementary school classroom.

Think about the usual dividing line on the equatorial Mercator projection, 180 degrees east (or west). This single line does not map onto a line; it maps onto two lines and becomes undefined as we approach the poles. This is an example of a *one-to-many* mapping, and an *undefined* mapping, and these are the kinds of mapping transformations that are extremely difficult to invert.

Even when the projection is one-to-one, the cartographer is faced with the dilemma of portraying a spherical or ellipsoidal earth on a flat surface. In general, projections are either *conformal*, in which they preserve local shape and direction at locations on the map, or *equivalent*, in which they preserve area. To achieve one or the other, projections are sometimes cut or interrupted, usually along parallels and meridians. Often a projection is neither conformal nor equivalent, but a compromise between these properties. The projection should therefore be taken into account carefully when one is using cartographic algorithms such as area computation or angle calculation on the geographic coordinates. Different geometries apply in the spherical and planar cases. For example, most people are familiar with the geometry of plane triangles (Figure 11.4). We know things like the law of sines. If a triangle has sides A , B , and C , with opposite angles a , b , and c , then we know that

$$\frac{\sin A}{a} = \frac{\sin B}{b} = \frac{\sin C}{c}$$

We often use this and other formulas to solve general-purpose triangles. It is a little different on a sphere where we use the spherical triangle. For example, if you are flying an airplane from New York to Chicago, you may want to minimize the amount of fuel you use by flying along the shortest path. To fly along the shortest, or *great circle*, route, you would have to fly along the shortest distance on the surface of a sphere between two points. This is a different geometry from the regular triangular case. We have a spherical

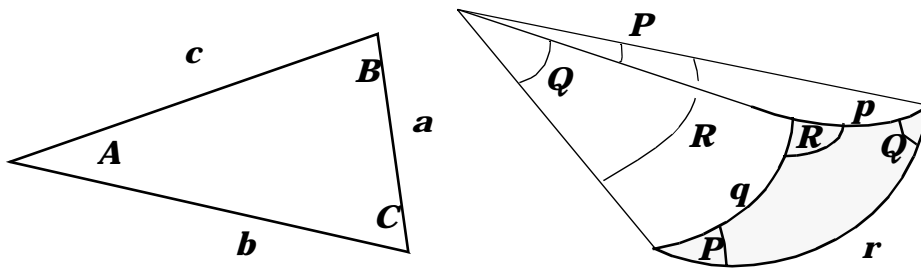


Figure 11.4 Planar and spherical triangles.

rule of sines, because the lengths of our sides are angles, too. If lengths are angles, they

$$\frac{\sin p}{\sin P} = \frac{\sin q}{\sin Q} = \frac{\sin r}{\sin R}$$

are usually expressed in radians rather than in degrees, because radians are used by the computational functions involved. We usually express latitudes and longitudes in degrees minutes and seconds, but in most spherical geometry we deal with radians, and most of the angles are angles at the center of the earth. In particular, computer programs and calculators often require angles in radians. Under the Spatial Data Transfer Standard, the required use is the decimal degrees format. Among existing software, the degrees, minutes, seconds format, that is, DD.MMSS, is frequently encountered.

To convert angles in degrees to radians, use the following formula:

$$\text{radians} = \frac{\pi}{180} \times \left(\text{degrees} + \frac{\text{minutes}}{60} + \frac{\text{seconds}}{60 \times 60} \right)$$

Geodesy works with several alternative means for representing earth coordinates, including three-dimensional (x, y, z) , polar coordinates, three angle coseries, and the familiar latitude and longitude.

Let us now set up a spherical angular referencing system based on latitude and longitude. You will notice that this involves going through a transformation so that you can get a range of values. Figure 11.5 shows a longitude that is an x value going from negative 180 at the international date line (or more strictly, the 180th meridian) to positive 180 at the international date line. Notice that at the dividing line we have a one-to-many mapping. We have chosen a central meridian, which in this case happens to be the origin of

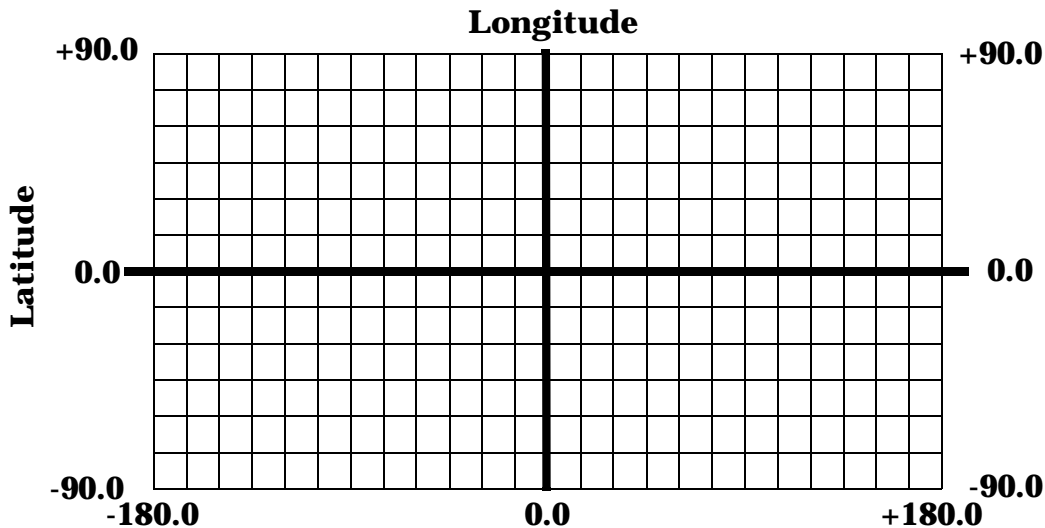


Figure 11.5 Generalized world coordinate system..

zero at the prime meridian. Going up and down, we choose the equator as zero; we make the North Pole 90 and the South Pole -90 , that is, 90 degrees north and 90 degrees south. Notice that here is another one-to-many mapping; that is, that the coordinate system as shown is inherently a cylindrical projection, or more precisely, an equirectangular projection. On this map we have one-to-many mappings around the edge, and one-to-ones within the map itself, where fortunately we can handle most of the transformations without too much trouble.

Any map projection transformation starts with latitude and longitude pairs and yields Euclidean coordinates (x, y) . With the correct scale factor, the x and y values are measurements in map millimeters which we can use in a graphics program when we plot a map. λ and ϕ , longitude and latitude, respectively, are given in radians.

Map projection transformations deserve a closer look here before we move on to planar-only transformations. The Mercator projection, which strictly we should call the equatorial Mercator projection, is a cylindrical projection. It is based on a projection of the Earth's surface onto a cylinder wrapped around the earth so that it just touches at the equator and has a central axis which passes through both poles. The formulas for Mercator's projection are

$$x' = Rs(\lambda - \lambda_0)$$

$$y' = Rs \log \left(\tan \left[\frac{\pi}{4} + \frac{\phi}{2} \right] \right)$$

where R is the radius of the sphere on the map, s is the scale factor (representative fraction of the map), and λ_0 is the longitude in radians of the central meridian. Because the radius of the earth at the equator (under the Geodetic Reference System of 1980) is 6,378,137 meters, scale factors have to reduce that to some reasonable value so that the projected map will fit onto a computer screen or a sheet of plotter paper. At 1:42,000,000 the earth can be represented by a globe about as big as a basketball. With s set to 0.00000001, the map will have an equatorial diameter of about 0.4 meter.

In most Mercator projections the origin is the prime meridian (longitude zero) so most Mercator projections appear with Europe in the middle, or at least with Greenwich in the middle, as this is the origin of the coordinate system. There is nothing that says we have to focus the projection there, but it is rather convenient, because the international date line becomes the space and time dividing line for the map. Also, the one-to-many mapping at the date line is least inconvenient; it passes mostly through water. Strictly speaking, the 180th meridian is not the international date line, because the line is moved to avoid land. We can plot world maps on the equatorial Mercator projection using the transformation equation. The spacings on the Mercator projection latitude graticule get closer and closer as we approach the equator, the effect of taking the logarithm.

Map projection is a part of analytical cartography where the use of computers is quite advantageous because the repetitive computations are very tedious. Most people can work through the formulas here, but it is indeed humbling to realize that Gerhardus Mercator presented the projection as a graphic construction in 1569 and that the formulas were published in 1599 by Edward Wright. This was a significant moment in the history

of cartography, showing the two approaches of a technology-dependent graphic (using Mercator's pen, paper, compasses, and dividers) and Wright's analytical cartographic approach (using mathematics). One wonders whether the mathematical approach presupposes and requires the visualization inherent in the graphic.

Function 11.1 is a C language function designed to perform the equatorial Mercator transformation for all the points in a string stored in the structure `STRING`, introduced in Chapter 7. Note that when the longitude of the origin is not zero, the programmer must rotate the data between -180 and $+180$.

The same transformation can also be expressed using matrix notation. If we start with longitude and latitude and apply a transformation to them, we get one from the other:

$$\begin{bmatrix} x \\ y \end{bmatrix} = T \begin{bmatrix} \lambda \\ \phi \end{bmatrix}$$

It also works the other way around:

$$\begin{bmatrix} \lambda \\ \phi \end{bmatrix} = T^{-1} \begin{bmatrix} x \\ y \end{bmatrix}$$

If the purpose is to get longitude and latitude, we can start with their cartesian coordinates taken by measurement from a map on some map projection and we can apply the inverse of this transformation to yield longitude and latitude.

Computer cartographic software to perform the transformations between map projections is increasingly part of the more sophisticated computer mapping packages. Especially important is the ability to transform between coordinate systems based on specific map projections, such as the Universal Transverse Mercator and the state plane systems. An excellent reference on map projections is Snyder's *Map Projections—A Working Manual* (Snyder, 1987). This work includes references to computer programs to perform the map projections listed and discussed. The reader should also check Snyder and Voxland's (1989) *Album of Map Projections*, which shows plots of each of the map projections along with their formulas.

The MicroCAM software package is an excellent tool for investigating the characteristics of map projections. MicroCAM computes and plots maps on IBM-PC compatible microcomputers for any of some 27 map projections, including graticules, coastlines, rivers, political boundaries, and United States state and county outlines (Anderson, 1994). The package, written by Scott Loomer, is available from the Microcomputer Specialty Group of the Association of American Geographers at a very low cost.

Also of interest is the NOAA *General Cartographic Transformations Package*, a set of FORTRAN programs and subroutines, along with data files containing all the necessary constants for the various standards and datums as well as constants for the map projections. Twenty map projections, with forward and inverse calculations, are included in the public domain package (NOAA, 1988).

Function 11.1

```

/* Function to transform a string to Mercator Projection */
string #include <math.h>
#define PI 3.141592654
#define R 6378137
#define S 0.00000005
#define LAMBDA_SUB_ZERO 0
STRING mercator(line, max_latitude)
    STRING line;
    int max_latitude;
{
    STRING project;
    double two_pi_over_360, pi_over_four, r_times_s;
    int i, k;
    two_pi_over_360 = 2.0 * PI / 360.0;
    pi_over_four = (double) PI / 4.0;
    r_times_s = (double) R * S; k = 0;
    for (i = 0; i < line.number_of_points; i++) {
        /* Convert to radians */
        line.point[i].x = two_pi_over_360 * line.point[i].x;
        line.point[i].y = two_pi_over_360 * line.point[i].y;
        /* test to exclude points outside range of latitude */
        if((line.point[i].y <= max_latitude) &&
            (line.point[i].y >= -1.0 * max_latitude)) {
            project.point[k].x = line.point[i].x;
            project.point[k].y = line.point[i].y;
            k++;}
    }
    project.number_of_points = k;
    for (i = 0; i < project.number_of_points; i++) {
        project.point[i].x = r_times_s *
            (project.point[i].x - LAMBDA_SUB_ZERO);
        project.point[i].y = r_times_s * log(tan(pi_over_four
            + (project.point[i].y / 2.0)));
    }
    return (project);
}

```

11.3 PLANAR MAP TRANSFORMATIONS

In the remainder of this chapter, we will consider some simple geometric transformations, still using map coordinates. Even though some of these transformations are simple, they are nevertheless powerful. Generally, we will be dealing here with transformations between point cartographic objects and objects with the same or higher dimensions. In most cases, we will discuss a problem, note the mathematical expression required for a solution, and then provide a computer program to yield the answer. Two types of results are derived: (1) transformations that produce a graphic object, and (2) transformations that produce a scalar measure. The latter can be thought of as a space-collapsing transformation, or a geographic measurement.

11.3.1 Transformations Based on Points

Distance between Two Points

Starting with something point-based and simple, most elementary is computation of the distance between two points. For example, consider two locations in space; call the first (x_1, y_1) and the second (x_2, y_2) . Given their locational attributes, otherwise known as their coordinates in space, and assuming planar and not spherical geometry (although spherical distance is no less difficult to compute), what is the distance between them? This is not a puzzling problem, having been figured out over 2,400 years ago by Pythagoras. The distance between those two points is equal to the square root of the sum of the squared lengths of the sides of the triangle made between these two points and the orthogonal axes. In other words, the distance is the square root of the difference between the two x values squared, plus the difference between the two y values squared. Pythagoras actually would have said that the areas of squares formed on the sides on the triangle made by the two points and a right angle with the axes are such that the largest triangle has an area equal to that of the sum of the smaller two.

To compute the distance between two points, which is simply a derived measure:

$$d_{2to1} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Analytically, the cartographer may use the computed distance to achieve an objective, for example to minimize distance (Function 11.2).

Function 11.2

```
/* Function distance: distance between two points */
#include <math.h>
float distance(one, two)
POINT one, two;
{
    return ((float) pow((two.x - one.x) * (two.x - one.x)
        + (two.y - one.y) * (two.y - one.y), 0.5));
}
```

We may have a whole series of points, and we want to find which two points are closest together. We have to calculate this value for all pairs of points and then sort the values to find the lowest, which will give us the nearest neighbors. We have already formulated an algorithm for distance computation. Distance minimization is a potential application of this algorithm. We can think of a matrix of points by points, with matrix values as the distances between the points, as being a useful transformation of the map space. In this case, the transformation is invertible, because the surveying practice of trilateration allows us to recompute relative locations from distances.

Length of a Line

Length is very similar to distance. A long line in computer cartography usually consists of a string of joined line segments, each consisting in turn of pairs of points with a connecting straight line. The length of each of these segments can be computed using the algorithm above. We simply have to add the lengths of all the segments to compute the length:

$$\text{length} = \sum_{i=1}^{npts} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}$$

Function 11.3 uses a line stored in our predefined STRING structure (Chapter 7). Notice that if the line has a number of points in the structure of type STRING given by the value `line.number_of_points` points, we have `line.number_of_points - 1` segments.

Function 11.3

```
/* Function line_length: Return length of a line */
#include "cart_obj.h"
float line_length (line)
STRING line;
{
    float distance(), sigma = 0.0;
    int i;
    for (i = 1; i < line.number_of_points; i++)
        sigma += distance(&line.point[i], &line.point[i-1]);
    return (sigma);
}
```

Centroids

One transformation of point data that yields a point rather than a scalar measurement is the computation of the centroid. Let us say that every city in the United States has a location, (x, y) , and also a population. For cartographic reasons, we may wish to associate the population number with the point, to draw a circle in proportional circle mapping, for example. A fairly simple concept is the center of gravity of the population. This single point is representative of a whole scatter of other points, so this is a points-to-point transformation. The Census Bureau usually publishes maps showing how this centroid has moved over time in the United States. How do we calculate this average or descriptive location? Locations in geographic space, perhaps the United States, have eastings (x), northings (y), and values, perhaps populations P . Each instance of the data will be called the n th, and there are a total of $npts$ of them. Assuming planar geometry, we can calculate the average x , \bar{x} , the average y , \bar{y} , weighted by their populations.

$$\bar{x} = \frac{\sum_{i=1}^{npts} P_i x_i}{\sum_{i=1}^{npts} P_i} \quad \bar{y} = \frac{\sum_{i=1}^{npts} P_i y_i}{\sum_{i=1}^{npts} P_i}$$

For example, if a city has a population of 12,000, we treat this as 12,000 cities at the same place with a population of only one person.

Standard Distance

Above we computed the mean x and y as an aggregate measure of location, so we can similarly compute the standard deviations in x and y as a measure of dispersal. The standard deviations in x and y can be used to compute the standard distance, a measure of the degree of scattering of point distributions around a focal point, the mean center, in both x and y as s_x and s_y .

$$s_x = \sqrt{\frac{\sum_{i=1}^{npts} (x_i - \bar{x})^2}{npts}} \quad s_y = \sqrt{\frac{\sum_{i=1}^{npts} (y_i - \bar{y})^2}{npts}}$$

The root of the sums of these values squared gives a value called the standard distance s :

$$s = \sqrt{(s_x^2 + s_y^2)}$$

This value is a measure of dispersal for point distributions. Now we have a simple measure of aggregate location and dispersal for point distributions. These values can be used analytically, for example, to examine the assumptions surrounding interpolation to a grid.

Nearest-Neighbor Statistic

Another descriptive statistic of point distributions is the nearest-neighbor statistic (*NNS*). For $npts$ points (x, y) scattered over an area A ,

$$NNS = 2 \times \frac{\sum_{i=1}^{npts} d_i}{npts \times \sqrt{\frac{A}{npts}}}$$

where d is the distance from each point to its closest neighbor. In this case, the point distribution has an empirically derived scalar statistic that measures the resemblance of the point distribution to a clustered, a random, a grid, or a hexagonal point distribution, although the value is critically sensitive to the area used (Figure 11.6).

A C language function to compute this statistic, therefore, must also choose how to compute the area that encloses the points. This can be the area of an enclosing polygon, such as a county or field boundary, or a bounding rectangle can be used. In Function 11.4 code function, the area is passed from the calling program

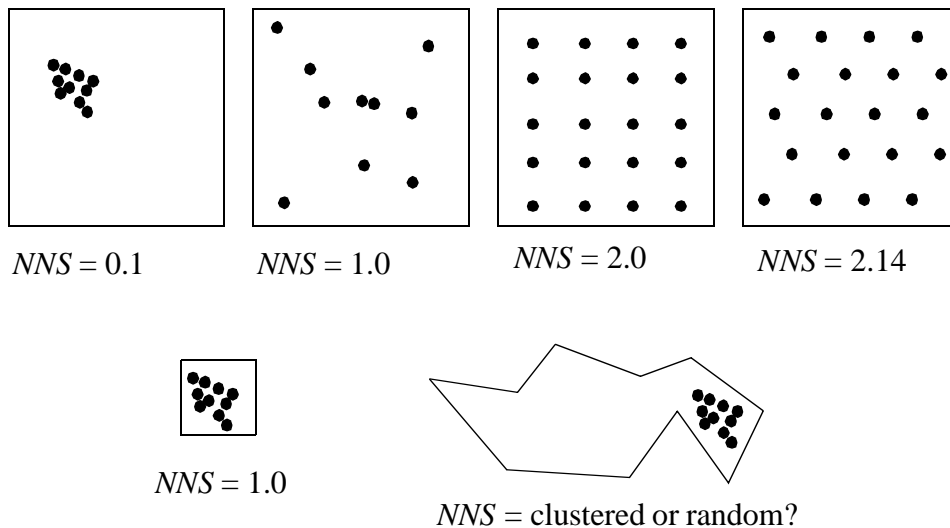


Figure 11.6 Nearest neighbor values and area sensitivity.

Function 11.4

```

/* Function nearest: Nearest neighbor statistic
 * assumes data are in structure POINT.
 * Calling function should provide area.
 * kcc 10-88 revised 8-93 */
#include "cart_obj.h"
#include <math.h>
float      nearest(n_point, point, area)
int        n_point;
POINT      point[];
float      area;
{
    float distance(), this_distance, shortest_distance,
          sigma = 0.0;
    int    i, j;
    for (i = 1; i < n_point; i++) {
        shortest_distance = distance(point[0], point[i]);
        for (j = 0; j < n_point; j++) {
            if (i != j) {
                this_distance =
                    distance(point[i], point[j]);
                if (this_distance < shortest_distance)
                    shortest_distance = this_distance;
            }
        }
        sigma += shortest_distance;
    }
    return ((2.0 * sigma / (n_point * sqrt(area / (double)
        n_point))));
}

```

11.3.2 Transformations Based on Lines

Intersection Point of Two Lines

Finding the intersection point of two lines is an example of an algorithm that performs a cartographic transformation yielding a point. A critical use of this algorithm in analytical and computer cartography is in clipping and computing the overlap between sets of chain encoded areas or the intersection of line features. Line intersection computations form an essential component of the extra computations necessary to structure data topologically in mapping and GIS systems. In these systems, the significance of the chain as a basic object is foremost.

This algorithm falls into the category of a test, because in computer cartography we frequently have to test lines for intersection. If two lines intersect, where do they intersect? This computation of line intersection points is an essential part of many vector mode systems and lies at the basis of all clipping algorithms. For the two line segments shown in figure 11.7, we wish to compute the intersection point (p, q) . We can find the intersection point by solving two simultaneous equations for the line $y = a + bx$, with two known values. If (x_1, y_1) and (x_2, y_2) lie on the same line, then

$$y_1 = a_1 + b_1 x_1$$

$$y_2 = a_1 + b_1 x_2$$

Similarly, if (x_3, y_3) and (x_4, y_4) lie on the same line, then

$$y_3 = a_2 + b_2 x_3$$

$$y_4 = a_2 + b_2 x_4$$

If there exists an intersection point, (p, q) that lies on both lines, then

$$q = a_1 + b_1 p$$

$$q = a_2 + b_2 p$$

By subtracting the former from the latter,

$$q - q = a_1 - a_2 + p(b_1 - b_2)$$

and rearranging, we obtain

$$a_1 - a_2 = p(b_2 - b_1)$$

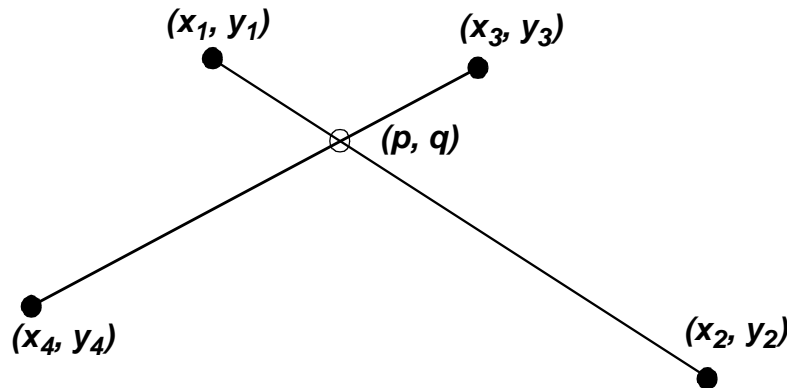


Figure 11.7 Terminology for line intersection.

If an intersection exists, this implies that

$$p = \frac{a_1 - a_2}{b_2 - b_1}$$

and by substitution

$$q = a_1 + b_1 \frac{(a_1 - a_2)}{(b_2 - b_1)}$$

The final equation will always provide a value, except in the case when the gradients (b) of the two lines are equal, in which case the fraction goes to infinity. This is the case of parallel lines, which includes the cases of coincident and collinear lines. Second, the point (p, q) may not lie on the line segments in question, but on the extension of the segment beyond its end points. To test for this, we merely have to test to see if p lies between x_1 and x_2 , and if q lies between y_1 and y_2 . If the coincidence is close to exact, the intersection is at one of the end points. Function 11.5 implements this algorithm.

Note that Function 11.5 uses two different structures, LINK and POINT. A LINK is a one-dimensional cartographic object connecting two nodes. The POINT structure has been used previously in this chapter. In this function, because the point is to be returned to the calling function, p is a pointer to a POINT structure. This requires the “->” structure member syntax rather than the “.” Both LINK and POINT structures were introduced in Chapter 5.

Finally, it should be noted that the intersection algorithm depends upon a value SMALL, which is a tolerance. Setting SMALL to some arbitrary value is not a good idea, and repeated use of the algorithm with different values is suggested as a way to understand the relationship between scale and precision. An alternative tolerance could be one-half of the length of the shortest line segment or the step size of the resolution determined from the data quality report.

Function 11.5 actually uses two different data structures to store the information contained in the line segments. Representation one is with the points of the line stored as a string in the LINK structure. In representation two, the line segments are represented by the constants of the linear equation for the line segment. In our representation, it is still necessary to store the points in the LINK to contain the segment along its line equation.

Saalfeld (1987) in responding to Douglas (1974) listed six ways in which a line can be represented in intersection algorithms. These include that used above, the point-slope form, and the slope-intercept form, plus the two-point form, an alternative two-point form, a linear equation form, and a point-vector form.

Thus a line segment starting at (x_1, y_1) and ending at (x_2, y_2) as in Figure 11.7 can be stored and used by intersection algorithms in any of the following ways:

1. Point-slope form

$$y - y_1 = b(x - x_1)$$

Function 11.5

```

/* intersect: Find the point of intersection between
 * two lines, if it exists. Returns 0 for no intersection,
 * else 1 and point x,y. kcc 10-88 revised 8-93 */
#include "cart_obj.h"
#define SMALL 0.000001
int intersect(line1, line2, p)
    LINK line1, line2; POINT    *p;
{
    int vertical_case = 0, collinear_case = 0;
    float xdif, a1, a2, b1, b2, hi1x, hi2x, hily;
    float hi2y, lo1x, lo2x, lo1y, lo2y;
    xdif = line1.end.x - line1.start.x;
    if ((xdif * xdif) < SMALL) vertical_case = 1;
    else { b1 = (line1.end.y - line1.start.y) / xdif;
          a1 = line1.start.y - (b1 * line1.start.x); }
    xdif = line2.end.x - line2.start.x;
    if ((xdif * xdif) < SMALL) vertical_case += 2;
    else { b2 = (line2.end.y - line2.start.y) / xdif;
          a2 = line2.start.y - (b2 * line2.start.x); }
    switch (vertical_case) {
        case 0: /* Neither link is vertical */
            if ((b1 - b2) >= SMALL)
                { p->x = (a2 - a1) / (b1 - b2);
                  p->y = a1 + (b1 * p->x); }
            else { /* The two links have equal slopes */
                    if ((a1 - a2) >= SMALL)
                        /* Collinear links */ collinear_case = 1;
                    /* The links are parallel and not collinear */
                    else return (0); }
            break;
        case 1: /* First link is vertical */
            p->x = line1.start.x; p->y = a1 + b2 * p->x; break;

        case 2: /* Second link is vertical */
            p->x = line2.start.x; p->y = a1 + b1 * p->x; break;
        case 3: /* Both lines are vertical */
            if ((line1.start.x - line2.start.x) >= SMALL) return (0);
            else /* Lines are vertical and collinear */
                collinear_case = 1; break;
        default: printf("Error, links do not match case set\n");
    }
}

```

Function 11.5 (continued)

```

/* Store x ranges of first link */
hi1x = line1.end.x; lo1x = line1.start.x;
if (line1.end.x < line1.start.x)
    { hi1x = line1.start.x; lo1x = line1.end.x; }
/* Store x ranges of second link */
hi2x = line2.end.x; lo2x = line2.start.x;
if (line2.end.x < line2.start.x) {
    hi2x = line2.start.x; lo2x = line2.end.x; }
/* Store y ranges of first link */
hi1y = line1.end.y; lo1y = line1.start.y;
if (line1.end.y < line1.start.y)
    { hi1y = line1.start.y; lo1y = line1.end.y; }
/* Store y ranges of second link */
hi2y = line2.end.y; lo2y = line2.start.y;
if (line2.end.y < line2.start.y)
    { hi2y = line2.start.y; lo2y = line2.end.y; }
if (collinear_case) { p->y = lo2y; p->x = lo2x; }
/* Test to see if intersection point falls on both links */
if ((p->x <= hi1x) && (p->x >= lo1x) && (p->y <= hi1y) &&
    (p->y >= lo1y) && (p->x <= hi2x) && (p->x >= lo2x) &&
    (p->y <= hi2y) && (p->y >= lo2y)) return (1);
else return (0);
}

```

2. Slope-intercept form

$$y = a + bx$$

3. Two-point form

$$\frac{(y - y_1)}{(x - x_1)} = \frac{(y_2 - y_1)}{(x_2 - x_1)}$$

4. Two-point form (avoids dividing by zero)

$$(y - y_1)(x_2 - x_1) = (x - x_1)(y_2 - y_1)$$

5. Linear equation, using floating point constants a , b , and c .

$$ax + by + c = 0$$

6. Point-vector form, for vector (v_1, v_2) and floating point number r

$$\begin{bmatrix} x & y \end{bmatrix} = \begin{bmatrix} x_1 & y_1 \end{bmatrix} + \begin{bmatrix} rv_1 & rv_2 \end{bmatrix}$$

As in Function 11.5, Saalfeld noted that line segment intersection algorithms mostly use a two-stage approach.

First, a point of intersection is determined, and second the point is tested for inclusion on the two test segments. Computational drawbacks of the algorithms are dealing with parallel or vertical lines, dealing with lines which are almost parallel or vertical, and rounding error. The six representations above all handle the parallel and vertical cases differently and so have computational limitations and costs accordingly.

The rounding and nearly parallel/vertical cases can be solved by using integer coordinates rather than real numbers and by avoiding large numbers. This may mean rounding to meters in UTM, multiplying temporarily by a scaling factor, and truncating large UTM or other coordinates, effectively using a false origin for the computations. Saalfeld pointed out the advantages of the point-vector form, primarily that each value of r corresponds to a unique locations along the line and that r can be made to vary from zero to one.

Distance of a Point to a Line

Yet another critical point-related transformation in automated cartography is establishing the relationship between a point and higher-dimension objects, such as lines and areas. To establish the distance between a point and a line, it is necessary to determine which link or line segment within the line is closest to the point. This can be done by using the point-to-point distance algorithm above, but may not result in the correct segment (Figure 11.8). A better test is to choose the point-to-segment triangle with the minimum area.

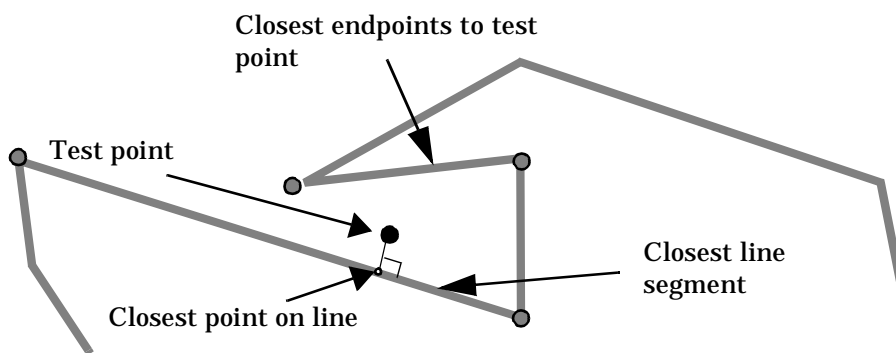


Figure 11.8 Computing the closest point on a line.

Next, the closest links (and there may be more than one, depending on the line) should be tested to determine the closest point along the line to the test point. An easy way to find the intersection point on the line is to mirror the point about the line and then to use the line intersection algorithm above to yield the point (Figure 11.9). Note that the result may be a point that is not on the line segment in question, in which case it should be discarded.

No program for this algorithm is presented here, because the more common cartographic problem is to generate a second line that is at some distance from the first, the so-called line-buffer problem (Schwarz, 1986).

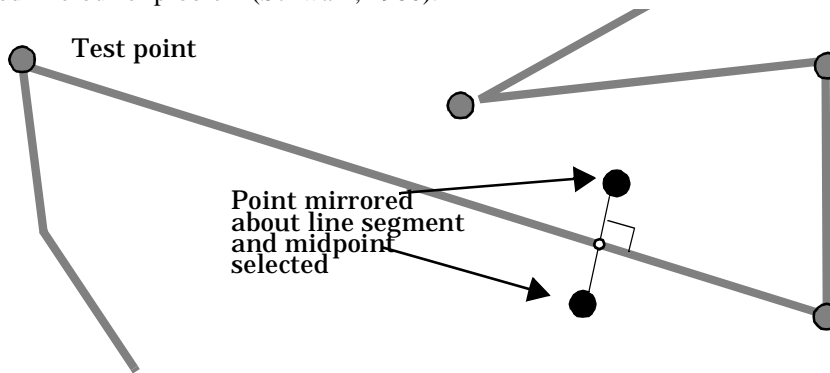


Figure 11.9 Computing the closest point on a line by mirroring.

11.3.3 Transformations Based on Areas

Area of a Polygon

How do we take a closed line (or G-ring) and calculate area? This is an area to a scalar transformation or simply a measurement. Say that we have a closed polygon with coordinates (x_1, y_1) to (x_n, y_n) and we repeat (x_1, y_1) to close the polygon, let's give the formula and then prove empirically that it works:

$$A = \frac{1}{2} \left| \sum_{i=1}^{npts+1} (x_i y_{i-1}) - (x_{i-1} y_i) \right|$$

Function 11.6 is a computer program to compute the area of a closed polygon stored in our predefined GRING structure (Chapter 7). The returned value will be negative if the polygon is defined counterclockwise and the absolute operator is ignored. Figure 11.8 and Table 11.1 show a simple polygon and the computation of the area by two methods, an intuitive proof and use of the equation. For simplicity, let the products of consecutive x and y values be as follows:

$$A = x_i y_{i-1} \text{ and } B = x_{i-1} y_i$$

Table 11.1 Area of A Polygon

Vertex	x	y	A	B	Difference (A-B)
1	1	1	—	—	—
2	1	6	1	6	-5
3	3	6	18	6	12
4	3	5	18	15	3
5	2	5	10	15	-5
6	2	4	10	8	2
7	3	4	12	8	4
8	3	3	12	9	3
9	2	3	6	9	-3
10	2	2	6	4	2
11	3	2	6	4	2
12	3	1	6	3	3
13	1	1	1	3	-2
					16

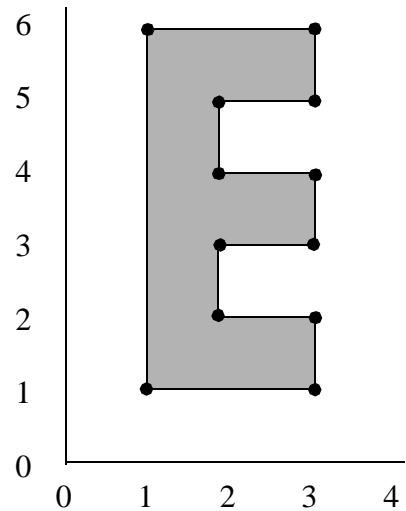


Figure 11.10 Sample polygon for the worked example.

Notice that it is easy to accumulate area as we go, so the computational values can be accumulated while we digitize a polygon. By simple inspection of Figure 11.10 we can see that the polygon consists of a letter E. This letter can be made up from eight unit-square building blocks, five forming a vertical column and three attached to the right in alternate rows.

Clearly, by inspection, the area of the polygon is eight units. From Table 11.1 we can see that the sum of the difference column is 16. If we take half this value, as required in the formula, we get the correct area of eight units. Also note that the value of the area comes out negative if the polygon is specified counterclockwise.

One way we can use this is to include holes within a polygon as rings within the polygon, except specified in the other direction. Then to find the total area, we simply add the area to the area of its enclosed holes to get the actual area of the polygon. Holes within the polygon under the Spatial Data Transfer Standard are considered as rings, with no re-traced lines.

Function 11.6

```
/* Function polygon area : Function 11.6
/* Return clockwise area enclosed by RING structure
/* kcc 10-88 revised 8-93 */

#include "cart_obj.h"

float area_of_ring (ring)
GRING ring;
{
    double sigma = 0.0;
    int i;
    for (i=1;i<ring.string.number_of_points;i++)
    {
        sigma +=(ring.string.point[i].x * \
        ring.string.point[i-1].y) - \
        (ring.string.point[i-1].x * ring.string.point[i].y);
    }
    return (0.5 * sigma);
}
```

When the primary data structure is the chain, such as in a topological data structure, the sum of the cross products terms of the segments within each chain can be stored with the chain as a partial solution, to be aggregated when chains are connected together to form a polygon. One problem with this, however, is that the accumulated sums can be very large numbers, especially if the coordinates are values such as UTM coordinates.

Point-in-Polygon Test

The next highest dimension object for which point relations can be established is the area or polygon. There are many algorithms for testing whether or not a point falls within a polygon. Many of the algorithms use area computations, and many others work only for convex polygons or polygons without concavities. These algorithms are virtually useless in analytical cartography, for the shapes of real-world cartographic entities are far from geometric and are almost never entirely convex in their boundaries.

Probably the simplest algorithm for point-in-polygon testing is the Jordan arc theorem. This simply states that a line between a point known to be outside a polygon will cross the polygon boundary an even number of times if the point is outside the polygon and an odd number of times if the point is inside the polygon (Figure 11.11)

This theorem provides solutions in all cases except when the lines either touch a vertex or run parallel to an edge. The parallel problem is often significant, because the outside point can simply be chosen as vertically above or below any given test point, and because many map lines run parallel to the axes.

Function 11.7 works directly for points on the interior area of the polygon. It establishes a point that is outside the polygon by finding the maximum x value for the polygon and multiplying it by 2. .

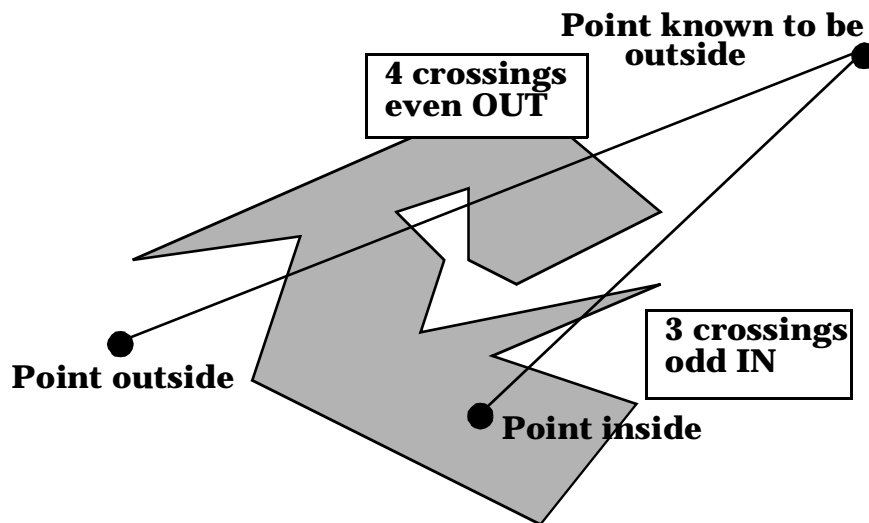


Figure 11.11 The Jordan Arc theorem.

Function 11.7

```

/* Function to perform point in polygon test using
 * Jordan arc theorem    kcc 5-88 revised 8-93
Function 11.07 */
#include "cart_obj.h"
int point_in_polygon (point, polygon)
POINT point;
RING polygon;
{
    POINT pt;
    LINE_SEGMENT test_segment, edge;
    int i, intersections = 0;
    test_segment.point[0].x = point.x;
    test_segment.point[0].y = point.y;
    /* Establish a point outside the polygon */
    test_segment.point[1].y = polygon.string.point[0].y;
    test_segment.point[1].x = polygon.string.point[0].x;
    for (i=1; i < polygon.string.number_of_points;i++)
        if (polygon.string.point[i].x > test_segment.point[1].x)
            test_segment.point[1].x = polygon.string.point[i].x;
    }
    test_segment.point[1].x *= 2.0;
    /* Now check the link from the test point to the outside
    * point against each of the polygon's edge segments */
    for (i=1; i < polygon.string.number_of_points;i++)
        {
            edge.point[0].x = polygon.string.point[i-1].x;
            edge.point[0].y = polygon.string.point[i-1].y;
            edge.point[1].x = polygon.string.point[i].x;
            edge.point[1].y = polygon.string.point[i].y;
            if (intersect(test_segment, edge, pt))
                intersections++;
        }
    /* Return odd/even, 1 means inside, 0 outside */
    return (intersections % 2);
}

```

Other implementations of the algorithm simply choose a point with the same *y* as the test point but a value of *x* greater than the largest *x* value in the ring. This makes the line intersection test somewhat easier, because no intersections have vertical line segments that would have to be treated as a special case in the line intersection routine in Function 11.5.

These algorithms can then deal with the delinquent cases mentioned previously. Sedgewick (1983, p. 317) for example, sorts the points in the ring so that they start with the point with the lowest x value among the points with the lowest y value. Both ends of the test segment are then tested to see if they are on opposite sides of the test segment. Only when this is the case is the intersection counted as the edge of the ring is traversed.

Thiessen Polygons

The final transformation between a point and a higher-dimension object to be considered in this section is a point-to-polygon transformation. The earliest computer cartographic program to perform this transformation was the SYMAP package, which called the polygons created *proximal regions*. These regions are called *Thiessen polygons* in geography after climatologist A. H. Thiessen. In computer science and mathematics the term *Voronoi diagram* is used. Thiessen devised these regions to assist in the interpolation of climatic data from unevenly distributed weather stations.

The method is used wherever data have been collected at points and the cartographer desires to use area-based analytical techniques or symbolization methods. As such, this is a space-partitioning transformation, because it divides the regions surrounding a set of points exclusively into a set of polygons.

The algorithms for finding the Thiessen polygons are too numerous and too lengthy to be included here; the reader is referred to a recent review by Tsai (1993). One noteworthy side effect of computing these areas is that the Delaunay triangulation is also performed because it is the dual of the Voronoi diagram. This transformation partitions the space using the points as nodes in the network and is often used to produce a TIN (triangulated irregular network) structure. Other applications are to make choropleth type maps of point data and to assist in interpolation over space.

11.4 AFFINE TRANSFORMATIONS

Map projections are the only coordinate transformations covered so far. It is possible, if we assume plane geometry, to provide a set of transformations for conversion between any two bounded planar Euclidean coordinate spaces. These transformations, known as *affine* transformations, consist of three distinct transformations applied in sequence.

The sequence is, first, a *translation*, in which the coordinate system origin is moved to a new location; second, a *rotation*, in which the axes of the coordinate system are rotated to match the new system; and finally, a *scaling*, in which the numbers along the axes are scaled to represent the new space scale. It is possible to accumulate the effects of these three transformations into a single step, which can then be applied on an object-by-object basis.

First, a transformation to move the origin can be stated as

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_0 & -y_0 & 1 \end{bmatrix} = \begin{bmatrix} x - x_0 & y - y_0 & 1 \end{bmatrix}$$

Note the similarity between the translation and the movement of the origin for a map projection. Map projections usually start by subtracting from the longitude the longitude at which the projection is to be centered, equivalent to rotating the earth to a new center of view.

The next stage is a rotation of the axes about an angle θ

$$\begin{bmatrix} x - x_0 & y - y_0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} \cos \theta(x - x_0) - \sin \theta(y - y_0) & \sin \theta(x - x_0) + \cos \theta(y - y_0) & 1 \end{bmatrix}$$

Third, the point is subject to a scaling of the axes:

$$\begin{bmatrix} \cos \theta(x - x_0) - \sin \theta(y - y_0) & \sin \theta(x - x_0) + \cos \theta(y - y_0) & 1 \end{bmatrix} \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} S_x[\cos \theta(x - x_0) - \sin \theta(y - y_0)] & S_y[\sin \theta(x - x_0) + \cos \theta(y - y_0)] & 1 \end{bmatrix}$$

Note that the scaling also looks much like the scale transform in a map projection. This is precisely what the affine transformation above is doing. The sequence of these transformations is translation, rotation, and scaling. Often order is important; in other words, if we use some other sequence, we will not end up with the same final location. That has implications for inverting this transformation. Because the transformation is multistep and we have to perform the steps in the correct sequence, we also have to undo them in the correct sequence.

We can multiply these 3 by 3 matrices by each other and produce a single transformation which summarizes the entire set of transformations:

$$\begin{bmatrix} x & y & 1 \end{bmatrix} TRS = \begin{bmatrix} x' & y' & 1 \end{bmatrix}$$

In nonmatrix algebra, this yields:

$$x' = S_x[\cos \theta(x - x_0) - \sin \theta(y - y_0)]$$

$$y' = S_y[\sin \theta(x - x_0) + \cos \theta(y - y_0)]$$

Figure 11.12 shows these three transformations graphically.

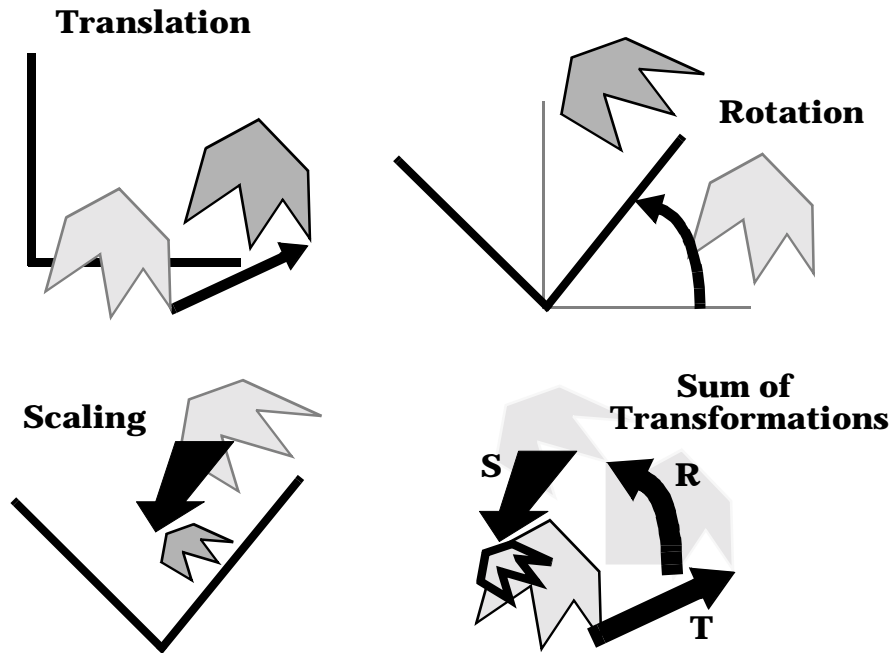


Figure 11.12 Affine transformations.

Note that the x and y of the origin are the x and y of the new origin in the old coordinate system. Consider the following example: A point in the old coordinate system is at $(2, 2)$. What is the coordinate of the point in a system that has a new origin at point $(0, 2)$ in the old coordinate system, a 30-degree clockwise rotation of the two sets of axes, and new axes that have twice the spacing? Using the formulas, we find that

$$x' = 2[\cos 30(2 - 0) - \sin 30(2 - 2)]$$

$$y' = 2[\sin 30(2 - 0) + \cos 30(2 - 2)]$$

This reduces to

$$x' = 2[0.866(2) - 0.5(0)] = 3.464$$

$$y' = 2[0.5(2) + 0.866(0)] = 2.000$$

Typically, these expressions are computed during the setup and registration phase of digitizing, when the affine constants are computed once and stored from a set of control points using the least squares solution given below. Then, successive points can be transformed as they are collected, or as a single step after data collection.

11.5 STATISTICAL SPACE TRANSFORMATIONS

11.5.1 Rubber Sheeting

When data have been captured from a flat map in cartesian coordinates, the affine transformations can be used for transforming data coordinates. In many cases, however, including those in which direct measurement of a cartographic entity has taken place, the precise map geometry is undetermined. This is especially true with remotely sensed images and air photos, when the viewing geometry is unknown. In both cases, the image is distorted for a number of reasons, including the flight geometry, the earth's curvature, the portion of the earth covered, the view angle, the topography, the altitude of the observation platform, the characteristics of the lens, and a host of other reasons. Also, the atmosphere itself is far from uniform, making its own "lens" effects, depending on pressure, moisture content, time of day, solar heating, and so forth.

Even under the best of circumstances, aircraft and even satellites pitch, yaw, and even "wobble." As a result, the earth's surface as seen in an image is an approximate map. In this map the actual cartographic objects and features can be plainly seen, but the precise relationships between scale, area, shape, and direction are imprecise.

An alternative way to state the relationship between such a distorted image and an accurate cartographic map base is to state that the precise map projection transformation is unknown. Instead, we have numerous actual examples of the transformation having been applied. To describe the projection characteristics fully, which is necessary if we wish to invert its effects and convert the image back to actual map space, we have to study the projection empirically.

One analogy is to imagine the map printed on a rubber sheet. When the air photo or satellite image arrives, we should be able to recognize precisely the locations of a key set of places on the image. These locations are the *ground control points*. A ground control point is a physical feature in a scene that is clearly detectable and whose precise coordinates in a coordinate system are known. For some purposes, merely the location of ground control points is enough, but for others we should know the location and elevation quite precisely and with a high degree of accuracy.

Bernstein (1983) noted that typical ground control points are airports, highway intersections, land-water interfaces, and geological and field patterns. Campbell (1987) also noted that the ground control points should ideally be a single pixel in size on the image and should be easily identifiable against their background. They should be dispersed throughout the image, particularly at the edges, not all concentrated in a single region.

To follow the analogy, the ground control points can be seen as holes through which the rubber sheet map is pinned to the correct map and stretched to fit correctly, that is, the pins are at their correct locations on both the map and the image. Given this, we can measure the "stresses" in the rubber sheet to figure out how to stretch it into the map's geometry. We then apply these "stresses" to the image, and as a result the entire image is stretched into the map space. Strictly, the locations of the control points themselves should not be modified by the algorithm, although this is not the case with the function

used below (Figure 11.13). Also, the model of error assumed is that the error is uniformly random over the area in question. This is rarely the case when data have come from scanners, instruments, or field measurements.

Mathematically, we have sets of points that match in the two spaces. These points have n locations (x, y) on the map and (u, v) in the image.

$$\begin{bmatrix} x \\ y \end{bmatrix} = T \begin{bmatrix} u \\ v \end{bmatrix}$$

The trick is to use the n multiple cases of (x, y) and (u, v) , for which we have a set of

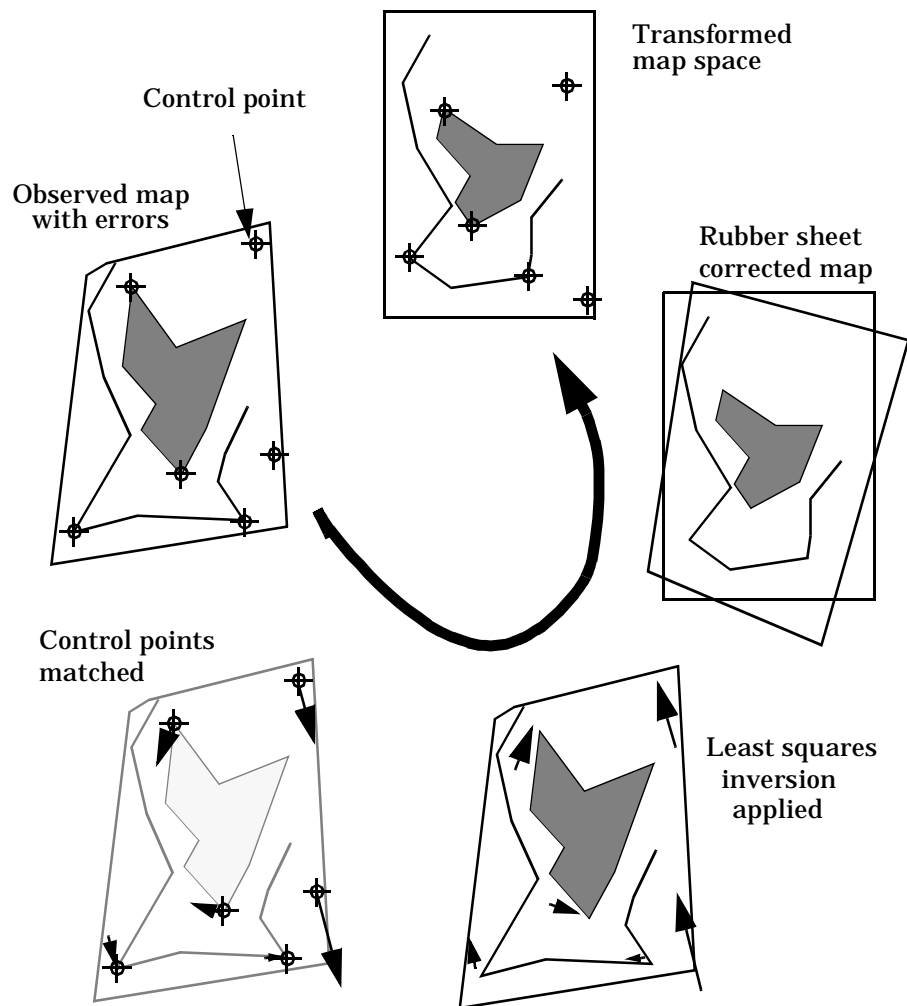


Figure 11.13 Rubber sheet transformations by least squares affine.

ground control points, to estimate T . The process of estimation proceeds by using what Sprinsky (1987) called the six-parameter affine, that is,

$$u = \beta_0 x + \beta_1 y + \beta_2 \quad v = \beta_3 x + \beta_4 y + \beta_5$$

The β coefficients can be computed by a sort of least squares fit. If we let

$$p = \frac{\sum_{i=1}^n [(x_i - \bar{x})(y_i - \bar{y})]}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

and

$$q = \frac{\sum_{i=1}^n [(x_i - \bar{x})(y_i - \bar{y})]}{\sum_{i=1}^n (x_i - \bar{x})^2}$$

then the following sets of equations allow the solution of the six unknown β values from n control points.

$$\beta_1 = \frac{\sum_{i=1}^n [(y_i - \bar{y})(u_i - x_i)] - q \sum_{i=1}^n [(x_i - \bar{x})(u_i - x_i)]}{\sum_{i=1}^n (y_i - \bar{y})^2 - q \sum_{i=1}^n [(x_i - \bar{x})(y_i - \bar{y})]}$$

$$\beta_3 = \frac{\sum_{i=1}^n [(x_i - \bar{x})(v_i - y_i)] - p \sum_{i=1}^n [(y_i - \bar{y})(v_i - y_i)]}{\sum_{i=1}^n (x_i - \bar{x})^2 - p \sum_{i=1}^n [(y_i - \bar{y})(x_i - \bar{x})]}$$

$$\beta_4 = 1 - p\beta_3 + \frac{\sum_{i=1}^n [(y_i - \bar{y})(v_i - y_i)]}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

$$\beta_0 = 1 - q\beta_1 + \frac{\sum_{i=1}^n [(x_i - \bar{x})(u_i - x_i)]}{\sum_{i=1}^n (x_i - \bar{x})^2}$$

$$\beta_2 = \frac{1}{n} \sum_{i=1}^n (u_i - x_i) + \bar{x} - \beta_0 \bar{x} - \beta_1 \bar{y}$$

$$\beta_5 = \frac{1}{n} \sum_{i=1}^n (v_i - y_i) + \bar{y} - \beta_4 \bar{y} - \beta_3 \bar{x}$$

The inverse transformation represented by the inversion of these formulas can then be used to allow the image coordinates as input and to provide map coordinates as output:

$$x = \alpha_0 u + \alpha_1 v + \alpha_2 \quad y = \alpha_3 u + \alpha_4 v + \alpha_5$$

where, for convenience, we let

$$k = \beta_4 \beta_0 - \beta_3 \beta_1$$

Then

$$\alpha_0 = \frac{\beta_4}{k} \quad \alpha_1 = -\frac{\beta_1}{k} \quad \alpha_2 = \frac{\beta_1 \beta_5 - \beta_2 \beta_4}{k}$$

$$\alpha_3 = -\frac{\beta_3}{k} \quad \alpha_4 = \frac{\beta_0}{k} \quad \alpha_5 = \frac{\beta_3 \beta_2 - \beta_5 \beta_0}{k}$$

This solution is also frequently used in surveying to adjust errors in measurement over a set of readings, and is frequently developed as a matrix solution (Moffitt and Bouchard, 1982). Several of the terms in the solution above are clearly determinants and transposes. A computer program to illustrate the solution is included on the companion disk. Once more, we have an example of a cartographic transformation. In this case, however, we have had to deduce the exact nature of the transformation from its results, and have inverted the transformation empirically in a “best-fit” solution.

At any point in space, it is possible to compute a vector displacement with an x and a y component from the inverse transformation. This vector can be used to compute the root mean squared error (RMS) at any point in the system. RMS error computation is of-

ten averaged over many points, so that an overall measure of error can be computed. Also, the RMS error at control points can be computed and used to eliminate from the computation those control points that contribute most to the RMS error. Thus the six-parameter affine transformation allows both practical transformation into a known geometry and a means of analyzing the error in doing so.

When there is little or no control over the measurement transformation (as in many air photo applications) or when the transformation is very complex, the statistical affine is the only approach available. Finally, it should be noted that in remote sensing the final result is a grid of points that are not regular. The points are therefore immediately resampled back into a regular grid that is orthogonal to the map coordinate system and has some finite pixel size. Methods for performing this second step transformation are discussed in Bernstein (1983).

11.5.2 Cartograms

Cartograms, also known as value-by-area maps and varivalent projections (Tobler, 1986), are maps drawn so that the areas of their internal enumeration units are proportional to the values they represent (Dent, 1985, p. 326). Generally, they are of two types. First, *non-contiguous* cartograms simply shrink the scale factor of an areal unit in isolation from all others and present the results as a diagram in which geographic space is interrupted. As such, these maps could be regarded as a form of proportional point symbol mapping. Second, *contiguous* cartograms maintain the continuity of geographic space, and as such are really a variation upon map projections. Like all map projections, direction, distance, shape, area, and the graticule are distorted in combination on these maps. Their advantage is that the map base can be distorted to make the mapping of a thematic data item more appropriate, removing the underlying differences in population density when showing per capita income, for example.

Tobler (1986) advocated the following method for the computation of “pseudo-cartograms.” First, the data to be used to “reshape” the map should be converted to the structure of a uniform grid, with equal latitude-longitude spacing north-south and east-west. If this grid is an m by n array, Z , then it can be written as the sum of k arrays, where k is the lesser of m and n . The arrays can then be written as a set of products of each of two k by 1 vectors, U and V :

$$Z = U_1 V_1 + U_2 V_2 + \dots + U_k V_k$$

where U_k is the k th eigenvector of the matrix Z multiplied by its transpose, Z^T and

$$V_k = \frac{Z^T U_k}{U_k^T U_k}$$

The vectors can then be sorted by the amount of variance accounted for by the model, and the first picked to yield the transforms. The U vectors are now functions only of the row

index or y , and the V vectors are functions of the column index or x . These values are then projected using a standard equal-area map projection:

$$y' = RsU_1 \cos y \quad x' = RsV_1$$

Tobler has published extensively on these transformations, including a set of computer programs (Tobler, 1974).

11.6 SYMBOLIZATION TRANSFORMATIONS

As far as producing a map is concerned, the final transformation is the symbolization transformation. In this transformation, the cartographic objects are subjected to all the necessary transformations to use a specific map type, and the map itself is generated. Although the actual look of the map is determined largely by the type of data, the fundamental properties of the data, the type of map, and the design of the map, there are some common or generic symbolization transformations. Among these is the viewing transformation, in which the geometry of the map is established. Also important is the actual plotting of the graphic elements, how they are symbolized, and how the text is added. This section contains information about these final, often critical, symbolization transformations according to the GKS standard.

11.6.1 The Normalization or Viewing Transformation

A point-to-point transformation of great importance in analytical and computer cartography is the transformation between the *world* coordinates, given in the coordinate system required by an application, and the coordinates of a particular display device, such as the screen of a graphics terminal. Typical *world* coordinate systems are either arbitrary (map millimeters, for example), or standardized (such as UTM coordinates). The user should note that world coordinates as defined by the GKS standard are not the same as geographic (latitude and longitude) coordinates but can be any coordinate system into which the objects have been geocoded. An orthogonal planar coordinate system is assumed. The display device also uses arbitrary coordinates known as *device* coordinates, usually the number of pixels that are addressable in the east-west and north-south directions. Because each particular display device has a different number of addressable pixels, it is desirable to have an intermediate coordinate system with a standard size into which we can transform the map.

The intermediate coordinates are known as *normalized device* coordinates. In the language of the Graphical Kernel System, the transformation from the world to the normalized device coordinates is known as the *normalization* transformation, while the transformation to device or screen coordinates is the *workstation* transformation. These transformations are summarized in Figure 11.14.

The normalization transformation is specified by defining the limits of an area in the world coordinate system known as the *world window*. This rectangle or square is mapped

onto a specified part of the normalized device coordinate space called the *world viewport*.

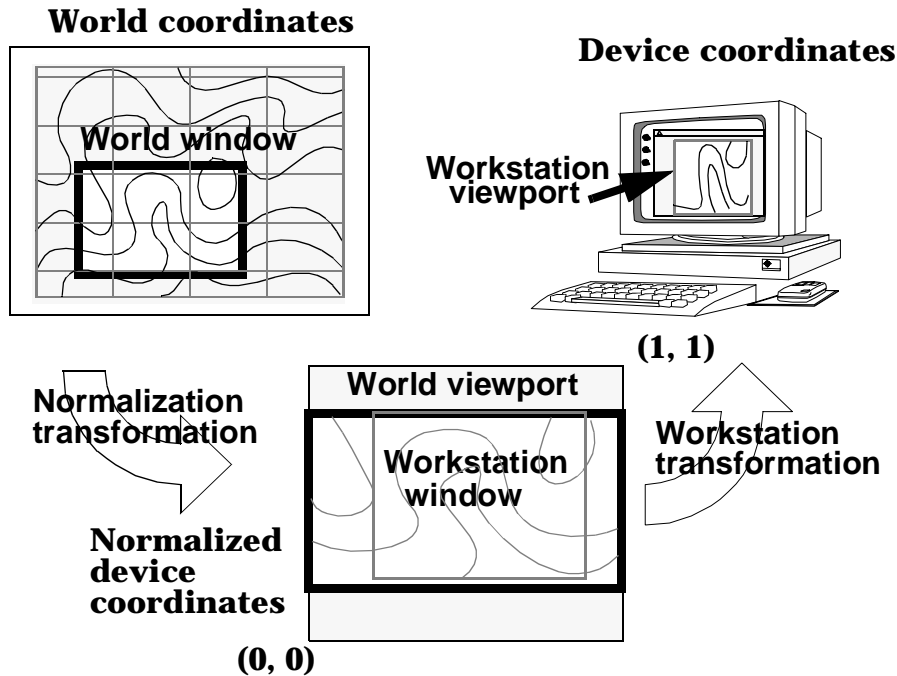


Figure 11.14 Normalization and workstation transformations.

Window and viewport limits are assumed to be parallel to the coordinate axes, so any rotations must be handled by the user, using the rotation algorithms given previously. Normalized device coordinates range from $(0.0, 0.0)$ at the lower left corner to $(1.0, 1.0)$ at the upper right corner. Similarly, we can control how much of the normalized device coordinate space is visible by establishing a workstation transformation. This is done by specifying a workstation window in normalized device coordinates. The actual transformations, the mathematics of which can be derived from the transformational equations given in the previous section on affine transformations, are contained within the Graphical Kernel System functions `set_window` and `set_viewport`.

A small interactive program to establish the transformation is given as Function 11.8. In this example, the goal is to center the map onto the largest possible piece of the device space while maintaining the correct scale relationships between x and y . This is done with the help of the GKS inquiry function `inquire_display_space_size`. This function returns the device coordinate units, the size of the surface of the device, and the number of pixels the device is capable of displaying. Note that your particular implementation of GKS will vary slightly in how C-binding GKS functions are named. SunGKS, for example, uses `gsetwindow`, `gsetviewport` and so forth. In a windowing system or graphical user interface, the dimensions of the device become those of the active window.

Function 11.8

```

/* Establish the normalization transformation for an active
 * display device, using the largest available centered
 * rectangle with appropriate scale.
 * Function 11.08 kcc 5-88 Revised 8-93 */
#include "gks.h"
void normalize()
{
    float llx, lly, urx, ury, xmin, xmax, ymin, ymax;
    float units, xpix, ypix, width, height, aspect_ratio,
          map_size;
    /* Establish the size of the display */
    ginqire_display_surface_size (&units, &xmax, &ymax,
                                  &xpix, &ypix);
    printf(" Enter the world window\n");
    printf(" Enter (xmin, ymin) :");
    scanf ("%f%f%*1c", &llx, &lly);
    printf(" Enter (xmax, ymax) :");
    scanf ("%f%f%*1c", &urx, &ury);
    /* Establish the aspect ratio of the world window */
    width = urx - llx; height = ury - lly;
    aspect_ratio = height / width;
    /* Apply the same aspect ratio to the display device */
    if (aspect_ratio < 1.0) { /* Landscape image */
        map_size = xmax * aspect_ratio;
        ymin = (ymax - map_size) / (2.0 * xmax);
        ymax = (ymin + map_size) / xmax; xmax = 1.0;
    } else { /* Portrait Image */
        map_size = ymax / aspect_ratio;
        xmin = (xmax - map_size) / (2.0 * ymax);
        xmax = (xmin + map_size) / ymax; ymax = 1.0 }
    /* Finally set up the normalization transformation */
    set_window ( llx, urx, lly, ury);
    set_viewport ( xmin, xmax, ymin, ymax);
    return;
}

```

11.6.2 Primitives and Attributes

Under GKS, one of the basic tasks of the system is to generate *pictures*, the standard name for the final rendered or symbolized graphic. In the case of computer cartography, the pictures contain symbolizations of cartographic objects, that is, maps. In the terminology of the Spatial Data Transfer standard, the real-world phenomenon to be represented on a map is called a cartographic entity. The digital representation, usually as a data structure, of the entity is as a cartographic object. In the final cartographic transformation, the *symbolization* transformation, the cartographic object is depicted as a cartographic element, that is, a distinct part of the graphic of a map. As such, the cartographic map element is a picture under GKS terminology and therefore consists of a collection of *primitives* with their associated *attributes*.

GKS defines and makes available to the programmer six primitives for digital drawing. These consist of one line primitive, one point primitive, one text primitive, two raster primitives, and one general-purpose primitive (Figure 11.15). The final cartographic transformation is to convert the cartographic data into groups of these primitives and to set their attributes. The line primitive is called the *polyline*. In symbolizing a polyline, GKS generates a set of straight line segments connecting a given point sequence. The point primitive is the *polymarker*. To symbolize a polymarker, GKS generates a symbol of a given type at a specified location. Symbols can be points, circles, squares, and so forth. The text primitive in GKS is *text*. Text causes GKS to generate a text character string at a specified location.

The two raster primitives under GKS are *fill area* and *cell array*. Fill area is used to generate polygons and to control the symbolization of their interiors. Cell array generates an array or grid of rectangular cells, each with individual colors. The cell array is an abstraction of the array of pixels on a raster type device. Finally, the *generalized drawing primitive* is designed to make use of special hardware characteristics of a specific workstation. Some workstations can generate circles, ellipses, spline curves, and so forth. These primitives have the workstation geometric transformations applied to them, but actual plotting is left up to the workstation..

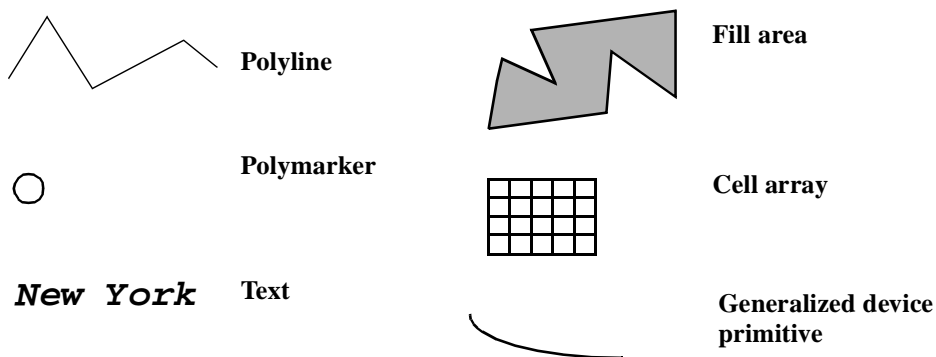


Figure 11.15 Primitives in the GKS standard.

Attributes are a function of the primitive to which they apply. Generally, polylines have the attributes of *line width*, *line type*, and *line color*. Polymarkers have the attributes of *marker size scale factor*, *marker type*, and *color*. Text has by far the most attributes, including *character height*, *character up vector*, *character expansion factor*, *text path*, *character spacing*, *text font*, *text precision*, *color*, and *text alignment*. Fill areas have the attributes of *pattern size*, *pattern reference point*, *pattern array*, *interior style*, *hatch style*, and *color*. The cell array has only *color*, while the attributes of the generalized drawing primitive are dependent on its type. Another important attribute is the *pick identifier*, which controls the way the user may interact with the particular primitive when it is displayed on the workstation, allowing objects to be interactively selected by the user.

Among the large variety of possible settings, line types may be solid, dashed, or dashed-dotted; text path may be left, right, up, or down; character alignment may be left-aligned, centered, or right-aligned to top, normal (middle), or bottom; and interior styles may be hollow, solid, pattern, or hatch. In addition, the attributes may be individual, in which case they apply to all the primitives that follow until they are changed, or bundled. If bundled, the attributes may be collected together as a group and applied collectively to one or more primitives. Such a collection of primitives is stored for an application such as a mapping program in a *bundle table*, a set of linked attribute settings for different but related primitives. *Indexes* named for each primitive allow separate management of these bundle tables.

It should be noted that GKS implementations usually come with library files, which predefine certain attribute parameters so that more intuitive names can be used. These definitions are often in a header file, such as `gksdef.h`, but the actual name depends on the GKS implementation. The pre-processor names, in keeping with the C standard, are in uppercase. So, for example, the first eight colors, which require calls with integers 1 through 8, are instead `#defined` to RED, GREEN, BLUE, MAGENTA, CYAN, YELLOW, BLACK, and WHITE. Similarly, the GKS implementations come with a documented set of supported output devices, such as X-windows terminals, PostScript output and the Computer Graphics Metafile. These values are all fully determined by the attribute `workstation_type`.

11.6.3 Drawing Cartographic Objects

The very final stage in the cartographic transformation from data to map is to actually implement a primitive, after having set its attributes, either individually or in a bundle. Before the primitives can be accessed, a workstation must be open and the sets of geometric space transformations to be applied must be active. Assuming that this is the case and that the computer cartographer has loaded data into the data structures we have outlined in earlier chapters, all that remains is to point to the GKS functions that actually do the display. First, we will cover the actual calls themselves, and following each function will be the relevant attributes that can be changed to manipulate the workstation display.

1. Polyline

To display a polyline, as stored in the previously defined STRING data structure, the

GKS call is

```
polyline(String.number_of_points, &String.point[0].x,
         &String.point[0].y);
```

Attributes are

```
set_linewidth_scale_factor (scale);
set_linetype (type);
```

2. Polymarker

A single polymarker is defined and stored in the previously defined POINT structure. Of course, these are normally handled as arrays of polymarkers. The GKS drawing call is

```
polymarker (1, Point.x, Point.y);
```

Attributes are

```
set_marker_size_scale_factor (scale);
set_marker_type (type);
set_marker_color_index (color);
```

3. Text

Text has not been treated as having a cartographic data structure in this book. In its simplest form, text consists of a string of C type `char`. The GKS drawing call is

```
text (text_string);
```

Attributes are

```
set_text_alignment (horizontal, vertical);
set_text_font_and_precision
    (font_index, precision_index);
set_text_path (path_direction);
set_text_color_index (color);
```

4. Fill area

To fill a polygon, the cartographer should use the previously defined RING structure, into which has been loaded data. The GKS command to plot the polygon is

```
fill_area (Ring.number_of_points, &Ring.point[0].x,
          &Ring.point[0].y);
```

Attributes are

```
set_fillarea_color_index (index);
set_fillarea_interior_style (style);
set_pattern_reference_point (x, y);
```



```
set_pattern_size (size_x, size_y);
```

To fill a complex polygon, the polygon boundary should be filled first, followed by the holes.

5. Cell array

To depict an array, such as a digital elevation model or a hill-shaded grid on a workstation, the cartographer should use the previously defined GRID structure, into which has been loaded data. The GKS command to plot the entire grid cell map is

```
grid_cell (x_p, y_p, x_q, y_q, no_cols, no_rows, col_start,
          row_start, dx, dy, &Grid.z[0][0]);
```

The only attribute is

```
set_color_index (index);
```

which should be called color by color to set the color table.

6. Generalized drawing primitive (GDP)

The depiction of a GDP is specific to a particular workstation, and is rarely used under cartographic applications unless drawing times become too slow.

11.7 SUMMARY

In this chapter we have covered numerous examples of cartographic transformations. We have worked through specific examples of how a transformational point of view allows the organization of the body of material that makes up computer cartography into its constituent parts within analytical cartography. Two major types of transformations have been presented. First, we have seen how cartographic objects can be transformed by dimension. As part of these dimensional transformations, the case of point-to-point transformations was used to show how the locational attributes of cartographic data can be transformed to produce map projections and cartograms. Second, we have examined the symbolization transformation, in which the cartographic objects gain actual instances as GKS pictures, collections of graphics primitives with their associated attributes.

Transformations of the first type noted above are fully within the domain of analytical cartography. The symbolization transformation is the very essence of computer cartography, and we have made the natural step from discussion of algorithms directly to computer standards and functions. The symbolization transformation and the programming theme form Part IV of this book, Producing the Map. In the remainder of Part III, Chapters 12 and 13, we will look specifically at line and area transformations, at data structure transformations, and finally at the transformations possible with volumetric data.

11.8 REFERENCES

- Anderson, P. S. (1994). *Guide for Users of MicroCAM*, February edition, Project MicroCAM, Department of Geography-Geology, Illinois State University, Normal, IL.
- Bernstein, R., ed. (1983). "Image Geometry and Rectification" in *Manual of Remote Sensing. Volume I: Theory, Instruments and Techniques*, edited by D. S. Simonett, American Society of Photogrammetry, Falls Church, VA: Sheridan Press, pp. 873–922.
- Campbell, J. B. (1987). *Introduction to Remote Sensing*. New York: Guilford Press.
- Dent, B. D. (1985). *Principles of Thematic Map Design*. Reading, MA: Addison-Wesley.
- Douglas, D. H. (1974). "It Makes Me So CROSS." Harvard University Laboratory for Computer Graphics and Spatial Analysis, internal memorandum. Reprinted in D. Marble, H. Calkins, and D. Peuquet (1984) *Basic Readings in Geographic Information Systems*. New York: SPAD Systems, Ltd.
- Microcomputer Specialty Group. Association of American Geographers. MicroCAM Software. Contact Dr. Robert P. Sechrist, Dept. of Geography, Indiana University of Pennsylvania, Indiana, PA 15705.
- Moffitt, F. H., and Bouchard, H. (1982). *Surveying*. 7th ed. New York: Harper & Row.
- National Oceanic and Atmospheric Administration (1988). *The General Cartographic Transformation Package*. National Ocean Service, Charting and Geodetic Service, National Geodetic Survey, Rockville, MD.
- Saalfeld, A. (1987). "It Doesn't Make Me Nearly as CROSS. Some Advantages of the Point-Vector Representation of Line Segments in Automated Cartography." *International Journal of Geographical Information Systems*, vol. 1, no. 4, pp. 379–386.
- Schwarz, C. R. (1986). "Algorithms for Constructing Lines Separated by a Fixed Distance." Proceedings of the Second International Symposium on Spatial Data Handling, Seattle.
- Sedgewick, R. (1983). *Algorithms*. Reading, MA: Addison-Wesley.
- Snyder, J. P. (1983). *Map Projections Used by the U.S. Geological Survey*, 2d ed. Geological Survey Bulletin 1532. Washington, DC: U.S. Government Printing Office.

- Snyder, J. P. (1987). *Map Projections—A Working Manual*. Geological Survey Bulletin P-1395. Washington, DC: U.S. Government Printing Office.
- Snyder, J. P., and Voxland, P. M. (1989). *An Album of Map Projections*. Geological Survey Professional Paper 1453. Washington, DC: U.S. Government Printing Office.
- Sprinsky, W. H. (1987). "Transformation of Positional Geographic Data from Paper-based Map Products." *American Cartographer*, vol. 14, no. 4, pp. 359–366.
- Tobler, W. R. (1974). *Cartogram Programs*. Department of Geography, University of Michigan, Ann Arbor.
- Tobler, W. R. (1979). "A Transformational View of Cartography." *American Cartographer*, vol. 6, no. 2, pp. 101–106.
- Tobler, W. R. (1986). "Pseudo-cartograms." *American Cartographer*, vol. 13, no. 1, pp. 43–50.
- Tsai, V. J. D. (1993). "Delaunay Triangulations in TIN Creation: An Overview and a Linear Time Algorithm." *International Journal of Geographical Information Systems*, vol. 7, no. 6, pp. 501–524.

