USING SURFACE MODELS TO ALTER THE GEOMETRY OF REAL IMAGES

by

MICHAEL RICHARD PALMER

B.Sc., The University of British Columbia, 1979

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming to the required standard.

THE UNIVERSITY OF BRITISH COLUMBIA

June, 1982

Department of Computer Science

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date Aug. 19, 1982

Abstract

Many applications of image analysis and image processing involve the alteration of the geometry of real images. This thesis presents the theory and implementation of three types of geometrically altered imagery: synthetic orthographic stereo pairs, synthetic perspective stereo pairs, and synthetic airborne scanner imagery. The real image is altered as a function of a corresponding surface model. Included is a determination of surface points which are hidden under the assumed imaging geometry and, in the case of radar systems, a determination of surface points which contribute to pixel layover. In this research, a digital terrain model determines the underlying surface geometry of the real image.

Table of Contents

List of Figures

Acknowledgement

I would like to thank my supervisor, Robert J. Woodham, for his constructive criticism and guidance throughout this research. I would also like to thank Alan K. Mackworth for being the second reader of this thesis and James J. Little for his work in developing the DTM and rectification algorithms that were taken advantage of in my research. Finally, I would like to again thank Bob for his support, financial and otherwise, of this research.

1.0 Introduction

Synthetic images, created via the alteration of the geometry of real images, find many uses in the fields of image analysis and image processing. For example, a synthetic stereo pair of images can be used in the evaluation of the accuracy of an image rectification algorithm [LITT80]. In addition, a synthetic stereo pair of images, can be used as an aid to photo interpretation.

This thesis presents the theory, methods, and results, of creating three specific types of synthetic imagery from real images. The types of imagery produced are:

(a) synthetic orthographic stereo pairs of an area represented by a real image and a corresponding surface model,

(b) synthetic perspective stereo pairs of an area represented by a real image and a corresponding surface model, and

(c) synthetic airborne scanner imagery of an area represented by a real image and a corresponding surface model (including considerations for both passive scanning systems and active radar systems).

The synthesis is performed by altering the geometry of the real image as a function of a corresponding surface model. The three algorithms to produce the three types of synthetic imagery are independent of each other and are therefore presented separately in this thesis.

The motivation for developing these three independent algorithms is now given.

1.1 Motivation

The motivation for each type of imagery considered in this research is summarized as follows:

(1) synthetic orthographic stereo

(a) Stereo viewing is beneficial. Applications include:

- aiding photo interpretation. The interpretation of subtle ground detail is facilitated when an image is viewed in stereo.
- evaluating the accuracy of image rectification [LITT80]. A mismatch between terrain features and ground cover features, as seen in stereo, can delineate errors in image rectification.
- evaluating the accuracy of the underlying surface model. Anomalies in a surface model are made apparent once the model has been registered to a real image and used in the production of a stereo pair.

(b) Synthesis of an orthographic stereo pair is computationally simplest.

(2) synthetic perspective stereo

(a) Stereo viewing is beneficial as detailed above.

(b) Real optical systems perform a perspective transformation. Therefore, even though the production of synthetic perspective stereo is computationally more tedious, it is required to predict the geometry of real imaging systems.

(3) synthetic airborne scanner imagery

Airborne scanner imagery requires additional geometric considerations. Two types of geometric distortions exist: systematic and nonsystematic. Systematic distortions are a function of the scanner itself (eg. mirror velocity, scan skew, and panoramic distortions). Correction of systematic distortions is deterministic [SABI78]. Nonsystematic distortions are either distortions due to parameters which vary (or are unknown) or distortions due to the particular scene in view. Correction of nonsystematic distortions is not deterministic but can be facilitated if a surface model is available.

The synthesis of airborne scanner geometry from a surface model predicts nonsystematic terrain distortions and can include systematic distortions. Applications include:

- evaluating the accuracy of distortion correction routines for real airborne scanner imagery,
- determining surface points which are hidden under a given scanner and line of flight geometry, and

- determining, in the case of radar systems, surface points which contribute anomalous readings due to pixel layover.

1.2 Terminology

Prior to presenting the technical details of the research, it is necessary to define / explain some of the terminology used in the presentation of the material.

A digital image is described as a series of cells or picture elements (pixels) arranged in regular rows and columns. The position of any pixel is determined in a two - dimension cartesian coordinate system with the origin defined at the upper left corner of the image. (This convention is used throughout.) The brightness of each pixel is given an eight bit numerical value. (A value of zero is black and a value of 255 is white.) Images implicitly represent a scene as recorded by a remote sensing imaging system [SABI78].

The platform is the medium which carries the remote sensing imaging system. This medium can range from low-flying aircraft to satellites. In this research, the imagery used was from the Landsat [LILL79] series of satellites.

The angular field of view is the total angle subtended by hypothetical lines from the imaging system to the extreme outer margins of the scene viewed by the system. The instantaneous field of view is the solid angle through which the system is sensitive when determining the value of each pixel. This solid angle, together with other flight parameters, determines the ground resolution of the imaging system (refer to figure 12) [SABI78].

The orthographic projection of a surface onto a plane is one in which the surface and the projection of the surface on the plane, lie on lines perpendicular to the plane [NEWM73] (i.e., all rays from the object to the plane are parallel [WOOD80]). For a perspective projection, the rays converge at a point that is a distance called the focal length from the plane. All optical systems perform a perspective projection.

For objects that are small, compared to their distance from the plane, the perspective projection can be modelled as an orthographic projection [WOOD80]. Images obtained from the Landsat series of satellites can be modelled as an orthographic projection.

Radiometry is the quantitative measurement of electromagnetic radiant energy [REEV75]. This measurement is the basis for remote sensing imaging systems. The value of a pixel is a measurement of the energy arriving at the sensor from the solid angle subtended by its instantaneous field of view.

Stereoscopy is the science that deals with three -

dimensional effects and how these effects are obtained [REEV75]. A stereo pair consists of two overlapping images that may be viewed stereoscopically [SABI78]. A stereo model (or stereoscopic image) is the three - dimensional impression obtained by viewing the left and right images of a stereo pair by the left and right eyes, respectively [SABI78]. Stereoscopic viewing is aided by use of a stereoscope - a binocular optical device. (The stereo pairs presented in this thesis can be viewed with a stereoscope to obtain the appropriate three - dimensional effects.) Synthetic stereo images are a pair of images that have been produced by the digital processing of a single image and a corresponding surface model. Parallax is an object's change in position between the left and right images of a stereo pair, relative to the two centres of the images, as a function of its relative height.

Pixel layover occurs in radar imagery when two or more surface points are equi-distant from the radar device at the time of pixel acquisition. Such surface points contribute to the same pixel in the radar image thus creating false bright targets [REEV75].

A bit map (or simply a map) determines points on the surface model that have a "special status". For example, points on the surface model that are not visible in the altered image are "marked" on a visibility bit map. (The original image can be overlayed by the corresponding bit map so that the "marked" points are easily identified.)

For a stereo pair, each image will have a corresponding visibility bit map. Similarily, a pixel layover bit map is created for those pixels that contribute to layover in a synthesized radar image.

1.3 Scope

Chapter 2 of this thesis presents the theory and implementation of synthesizing orthographic stereo pairs. The corresponding discussion of synthetic perspective stereo pairs is presented in Chapter 3. Chapter 4 details the theory and implementation of synthetic airborne scanner imagery, including considerations related to airborne radar. Chapter 5 presents conclusions, a discussion of the research, and suggestions for future work.

2.0 Synthetic Orthographic Stereo

This chapter discusses the synthesis of orthographic stereo pairs from a single orthographic image. The resulting stereo pair is obtained by introducing relief displacement into the original image. The original image is registered to a corresponding digital terrain model whose elevation information is used in the calculation of stereoscopic parallax. Figure 1 is a synthetic orthographic stereo pair from a portion of a Landsat-1 image of the St. Mary Lake region of southeast British Columbia (frame ID 11514 - 17153, band 7, imaged September 14, 1976).

2.1 Theory

Overlapping vertical aerial photographs can be viewed stereoscopically because relief displacement arises from a perspective projection of the underlying terrain [LILL79]. Images arising from an orthographic projection do not show relief displacement. Nevertheless, stereo viewing can be accomplished if the viewing direction is altered between the left image and the right image of the stereo pair. The assumption of orthographic projection simplifies the procedure for synthesizing stereo pairs, as will now be shown.

Figure 1. Orthographic stereo pair (St. Mary Lake region of southeast British Columbia).

First, consider a digital representation of the surface relief. This can be specified explicitly as the function

$$z = f(x,y)$$

where the x and y axes are in a plane tangent to the earth's surface and the z axis is vertically up (i.e., the xy plane forms a discrete uniform surface grid and the corresponding z values are elevation datum). These three axes are mutually orthogonal. Such a function is referred to as a digital elevation model or more commonly as a <u>digital</u> <u>terrain</u> <u>model</u> (DTM) [MILL58]. It is assumed that f(x,y) is a continuous function with continuous first and second spatial derivatives so that the surface may be considered to be smooth.

Referring to figure 2, assume a viewer to be at elevation H above the ground datum. Let this viewer be looking down with his "eyes" aligned along the x axis (with origin O) and separated by a distance of $2 \cdot d$. Further suppose the viewer's gaze to be fixed on the point (Xc,Yc) on the ground datum, forming an angle of convergence of 2θ. From this geometry it is clear that

$$\tan\theta = d / H \qquad (2.1)$$

A rotation transformation about the y axis and a pair of translations along the x axis, similar to those used in

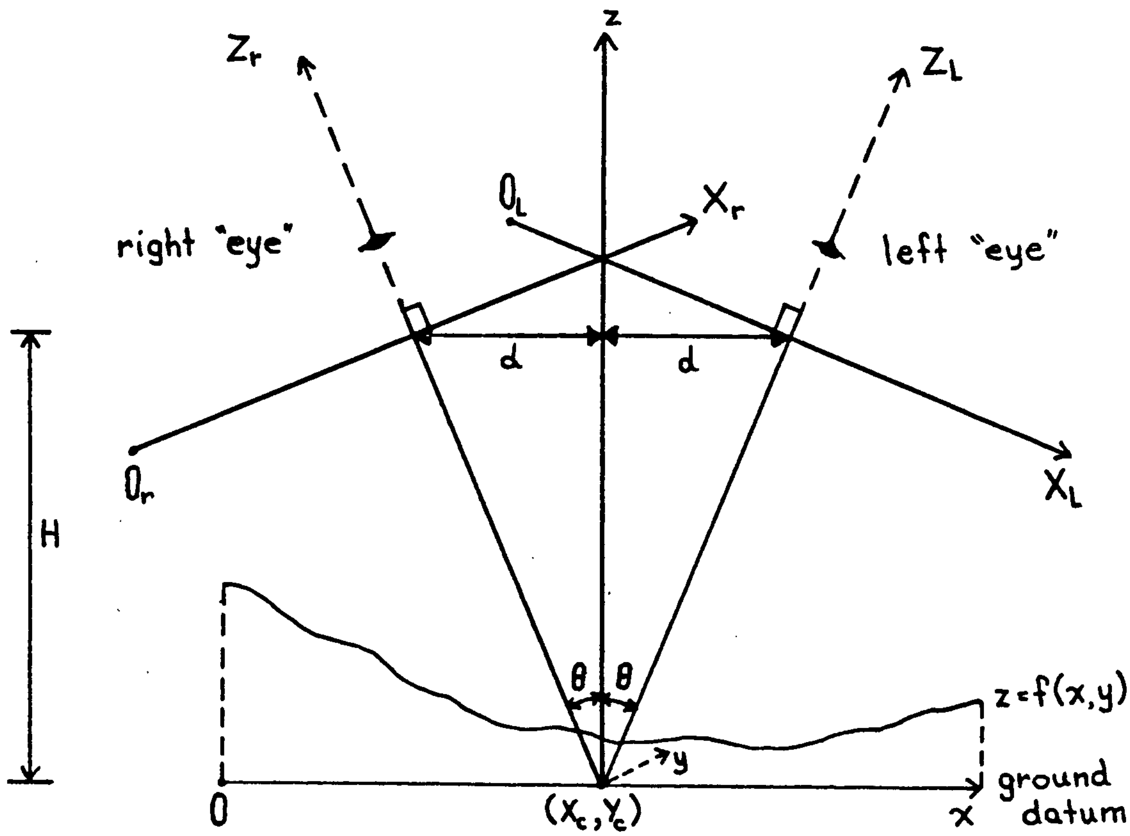


Figure 2. Geometry of an orthographic stereo pair.

computer graphics [NEWM79], can be applied to produce equations to map the coordinates of the original surface (X,Y,Z) into coordinates for the left eye coordinate system (Xl,Yl,Zl) and the right eye coordinate system (Xr,Yr,Zr). (Note: in figure 2, for the purpose of illustration, the left and right eye coordinate systems have been translated along the corresponding z axes, towards the "eyes" of the viewer. In reality, the Xl axis and the Xr axis intercept the x axis of the surface model at coordinate (Xc,Yc).) One obtains:

$$Xl = Xc + (X - Xc)\cdot\cos\theta + Z\cdot\sin\theta \quad (2.2)$$
$$Yl = Y \quad (2.3)$$
$$Zl = Z\cdot\cos\theta - (X - Xc)\cdot\sin\theta \quad (2.4)$$

$$Xr = Xc + (X - Xc)\cdot\cos\theta - Z\cdot\sin\theta \quad (2.5)$$
$$Yr = Y \quad (2.6)$$
$$Zr = Z\cdot\cos\theta + (X - Xc)\cdot\sin\theta \quad (2.7)$$

These equations yield three points of interest:

(a) it is evident from equations (2.3) and (2.6) that the y coordinate is not altered. This illustrates the primary advantage of the orthographic projection, namely, that processing can proceed a row at a time.

(b) the difference in x - position between the left eye coordinate system and the right eye coordinate system, (Xl - Xr), can be calculated from equations (2.2) and (2.5) as:

$$2\cdot Z\cdot\sin\theta \quad (2.8)$$

As expected, this is a function of Z (the elevation) and θ (the angle of convergence). For small θ, $\sin\theta$ can be approximated by $\tan\theta$. Using equation (2.1) and letting $S = 2 \cdot d$, the difference in x - position is thus approximated by

$$2 \cdot Z \cdot (d/H) = Z \cdot S/H \tag{2.9}$$

Equation (2.9) has been used in approximate calculations [BATS76].

(c) when a surface is given as the function $z = f(x,y)$ with the viewer looking down the z axis, the visibility problem has been explicitly solved. However, when the viewing direction is altered, as required here, visibility must be considered. Equations (2.4) and (2.7) determine visibility. If more than one (X,Z) maps into a given Xl, then the (X,Z) pair with the greatest Zl is the point that is visible.

This completes the geometric theory required to synthesize a stereo pair. An algorithm to implement the theory is given in the following section.

2.2 Implementation

The algorithm to synthesize an orthographic stereo pair creates two new images - the left and right images of the stereo pair. The original image must already be registered to a corresponding digital terrain model (i.e., a one-to-one correspondence has been established between the elements of the DTM and the elements of the image) [LITT80]. Thus, the necessary elevation information is provided for each pixel in the image. The algorithm is described as follows (the source code to produce the orthographic stereo pair is presented in Appendix A):

The angle of convergence, 2θ, is constant throughout the procedure so that the values of $\sin\theta$ and $\cos\theta$ are calculated once.

The lines of the image and the registered lines of the DTM are input a pair at a time (i.e., one image line and the corresponding DTM line). For each image and DTM line pair, the corresponding lines of the left image and right image are created using equations (2.2) and (2.5) respectively. For each X position in the original line, an Xl and an Xr are calculated for the two new lines. The pixel value corresponding to position X of the original image line is assigned to position Xl of the left image line and to position Xr of the right image line. When this displacement is introduced, it is likely that Xl and Xr do not align exactly with the uniform grid of the image. A simple one dimensional linear resampling [LILL79] is performed to generate intermediate pixel values.

It is possible that some displaced pixels (i.e., those corresponding to Xl and Xr) are not visible in their

respective new images. This situation is detected and resolved in the following manner:
For a given line in the original image, the pixels closest to Xc are considered first; the ones farthest away last. Employing this technique, if a pixel of the line is mapped to a position that is a distance J from Xc, then another pixel of that line that is subsequently considered is visible if and only if it is mapped to a position that is more than a distance J from Xc. If the distance is less than J, the pixel is not visible and hence it can be ignored. This procedure is done separately for the left and right image.

Another method to test the visibility of each pixel is to explicitly calculate the corresponding elevation using equations (2.4) and (2.7). This approach has several practical drawbacks. One shortcoming is that one Xl (or Xr) rarely equals another Xl (or Xr) since these values are real numbers. This fact leads to extra computations to interpolate intermediate values. A second drawback is the memory required to store an explicit elevation value for each pixel in the new line. For these reasons, the simpler method described above was chosen.

Another common situation arises when the current pixel is displaced more than one grid cell from its predecessor. Gaps appear in the new line because no pixel of the original line is mapped or resampled to that position. In this case, gaps are filled by using linear interpolation to calculate intermediate values.

Once the two new lines have been completely synthesized (displaced, resampled, and interpolated) they are output to the corresponding image file. The process is repeated for each line of the original image.

This algorithm for synthesizing orthographic stereo pairs produces a visually pleasing result. When figure 1 is viewed stereoscopically there are no unnatural terrain features; the radiometry of the stereo model appears smooth.

An earlier algorithm to achieve the same goal has been implemented by the U.S. Geological Survey [BATS76]. This particular algorithm only creates one new image. The original image is used as either the left or right image of the stereo pair. The synthesized image then forms the mate to the original. The new image is created using equation (2.8) or (2.9) depending on whether the calculations are exact or approximate. This method produces similar results as those obtained using the method presented here.

3.0 Synthetic Perspective Stereo

This chapter discusses the synthesis of perspective stereo pairs from a single orthographic image. The stereo pair is generated using a perspective projection of the underlying terrain from two different vertical viewing positions. The terrain is given as a digital terrain model that has been registered to the orthographic image. A perspective projection introduces relief displacement to points of the orthographic image. This displacement is directed radially from the assumed nadir. Methods for determining terrain visibility are also presented in this chapter. Figure 3 is a perspective stereo pair synthesized from the same portion of the Landsat image of St. Mary Lake used in figure 1. Figure 4 is the corresponding visibility map of the left image.

3.1 Theory

Optical systems, such as cameras, perform perspective projections of the underlying terrain [SABI78], [LILL79]. Overlapping vertical aerial photographs can be viewed stereoscopically due to their differing perspective projections of the underlying terrain. Different perspective projections arise when the optical centres of the image moves from one position to another along the line

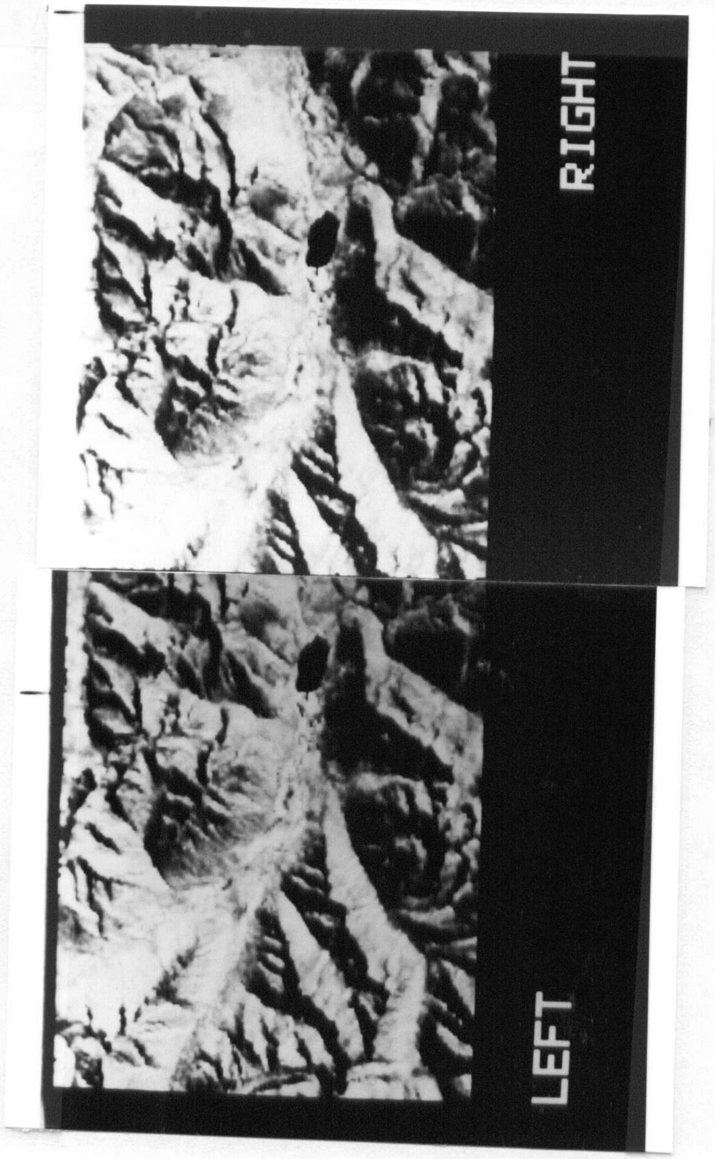


Figure 3. Perspective stereo pair (St. Mary Lake region of southeast British Columbia).

of flight. To synthesize a perspective stereo pair from a single orthographic image, one must provide different perspectives of the terrain depicted by the orthographic image. The theory of synthesizing a perspective image is presented first; that of providing different perspectives second.

On a vertical aerial photograph the position of each terrain point is displaced from its orthographic position. This displacement is directed radially from the optical centre of the aerial photograph. For terrain above the ground datum, displacement is radially outwards. For terrain below the ground datum, displacement is radially inwards [REEV75]. An orthographic image can thus be transformed into a perspective image by introducing relief displacement to points of the original image. The geometry of relief displacement will now be discussed in more detail.

Considering the digital terrain model, figure 5 displays the geometry of an arbitrary cross section of a vertical aerial photography system. This cross section is not, in general, aligned with the x axis or the y axis. With reference to figure 5, assume a hypothetical "camera", with focal length f, to be at elevation H above the ground datum and to be directly over the coordinate (Xp,Yp). (The coordinate (Xp,Yp), the centre of the vertical aerial photograph, is referred to as the principal point.) Further suppose that the terrain at coordinate

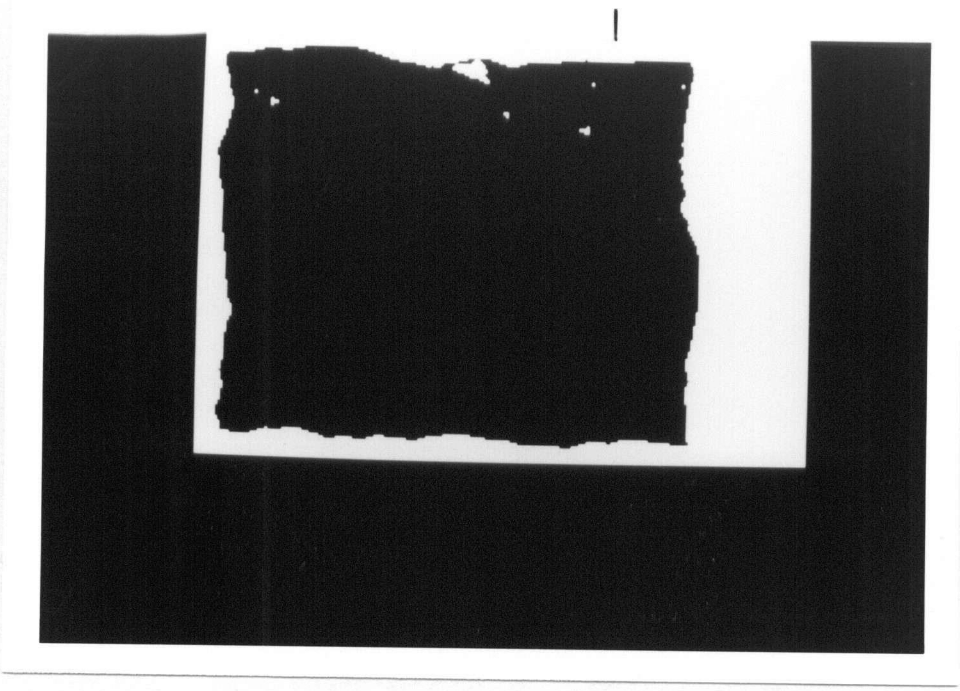


Figure 4. Perspective stereo visibility map.

(X,Y) is at elevation Z above the ground datum and that the radial distance from (Xp,Yp) to (X,Y) is R. The relief displacement for the terrain at coordinate (X,Y), as displayed in figure 5, is then b' - b on plane P, or equivalently, a' - a on plane P'. (For simplicity, calculations are performed on plane P'. Plane P is the actual plane of the film, however, plane P' and plane P have identical geometries as shown in figure 5. Therefore, measurements made on either plane are equivalent [SABI78].) The point a corresponds to the true ground position in the orthographic image and the point a' is the actual position in the vertical aerial photograph. An equation for the amount of relief displacement, a' - a, will now be derived.

From the geometry of figure 5, it is clear that the similar triangles LAB and LaD yield the relationship

$$a / f = R / H \tag{3.1}$$

Similarily, the triangles LA'C and La'D yield the relationship

$$a' / f = R / (H - Z) \tag{3.2}$$

Rearranging equations (3.1) and (3.2), equations for a and a' can be derived:

$$a = Rf / H \tag{3.3}$$
$$a' = Rf / (H - Z) \tag{3.4}$$

An equation for the relief displacement, commonly referred

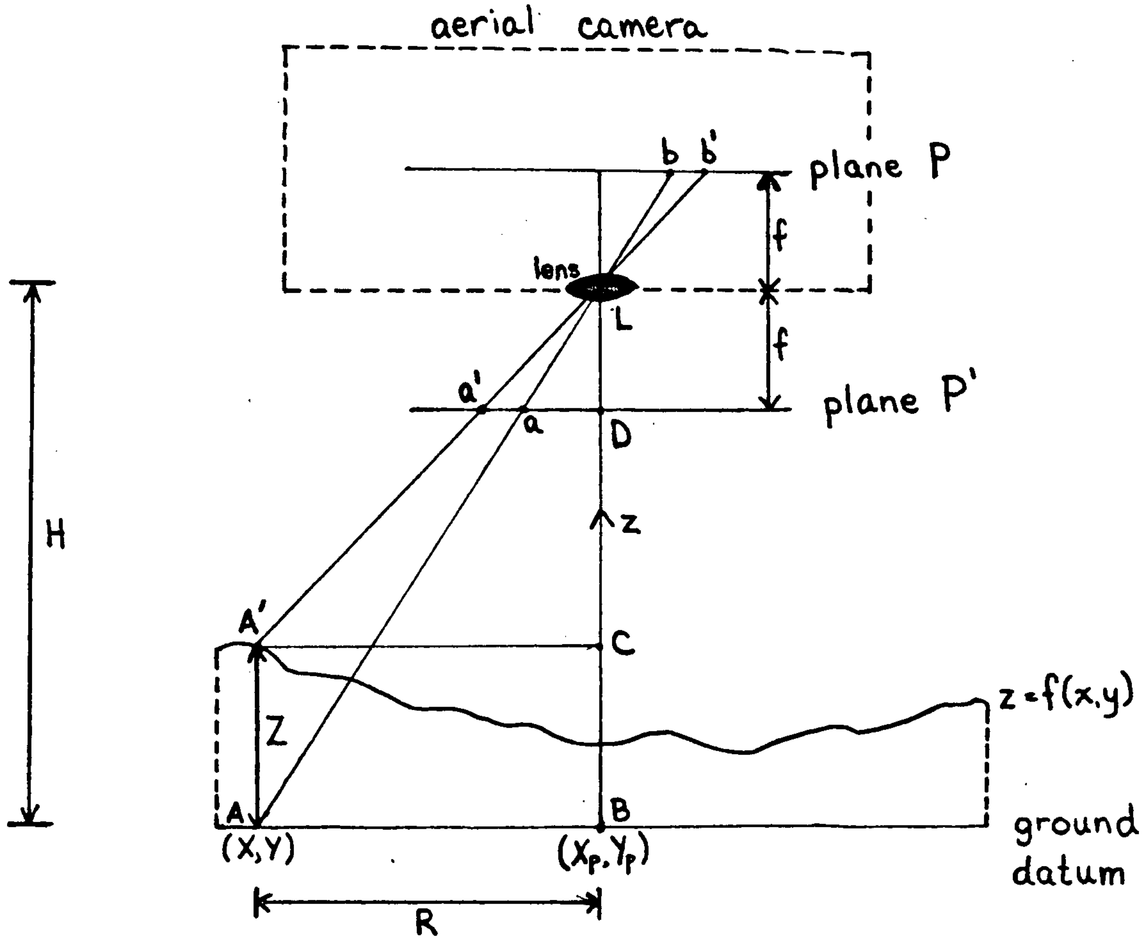


Figure 5. Geometry of a vertical aerial photograph (adapted from [SABI78], figure 2.9A).

to as d, can thus be produced by subtracting equation (3.3) from equation (3.4).

$$d = a' - a = (Rf / (H - Z)) - (Rf / H) \qquad (3.5)$$

Alternatively, as is done for standard photographic manipulations, d can also be derived as follows:
From equation (3.3) and equation (3.4)

$$Rf = aH = a'(H - Z)$$

and thus

$$d = a' - a = a'(Z / H)$$

Substituting with equation (3.4) yields

$$d = (Rf / (H - Z)) \cdot (Z / H)$$

Note that the relief displacement, as calculated by equation (3.5), is positive for terrain above the ground datum ($Z>0$), zero for terrain on the ground datum ($Z=0$), and negative for terrain below the ground datum ($Z<0$). From equation (3.5), it can also be seen that the relief displacement, d, is a function of four variables:

(a) Z, the elevation of the terrain above the ground datum,

(b) H, the elevation of the hypothetical "camera" above the ground datum,

(c) f, the focal length of the hypothetical "camera", and

(d) R, the horizontal radial distance from the principal point to the terrain.

Thus, a perspective image can be synthesized by introducing relief displacement (according to equation (3.5)) relative to a hypothetical principal point.

The calculation of terrain visibility for a perspective image is as follows:

Refer to figure 6. Let a' be the radial distance on the photograph print from the principal point on the print to the position of the terrain at coordinate (X,Y) on the print. The terrain at coordinate (X,Y) is then visible if and only if all visible terrain between itself and the principal point has positions on the print closer to the principal point (i.e., positions less than radial distance a' from the principal point). If terrain at coordinate (X,Y) is not visible then terrain between (X,Y) and (Xp,Yp), the principal point, must block its visibility to the camera. The determination of one point of terrain blocking the visibility of other terrain can be achieved in the following manner:

Refer again to figure 6. Further suppose that terrain at coordinate (Xv,Yv) has elevation Zv above the ground datum, that the radial distance from (Xv,Yv) to (Xp,Yp) is Rv, and that it is visible to the camera with a position on the print of radial distance A' from the principal point. Hence, terrain between (Xv,Yv) and (Xp,Yp) will block the visibility of the terrain at (X,Y) if

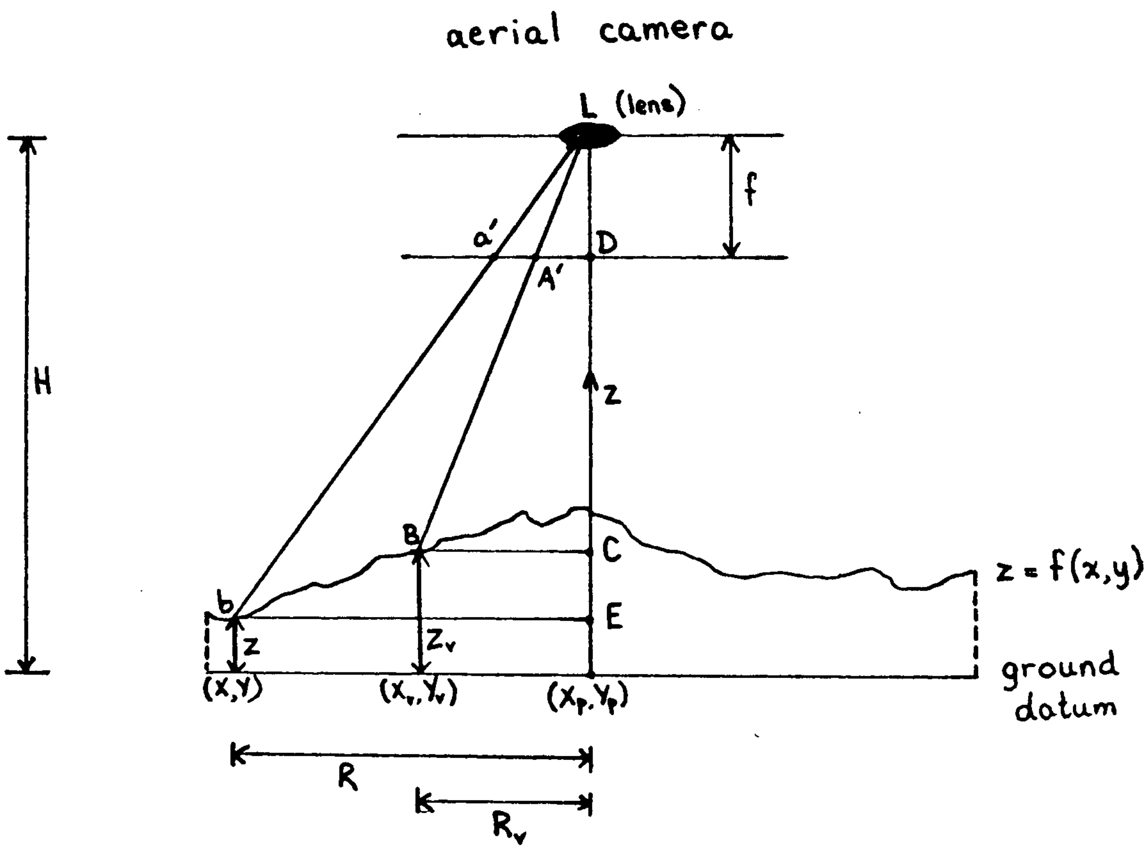


Figure 6. Terrain visibility geometry (perspective).

$$A' \geq a' \qquad (3.6)$$

However, it is clear from the geometry of figure 6 that the similar triangles LA'D and LBC yield the relationship

$$A' / f = Rv / (H - Zv) \qquad (3.7)$$

and similarily, the triangles La'D and LbE yield the relationship

$$a' / f = R / (H - Z) \qquad (3.8)$$

Rearranging equations (3.7) and (3.8), equations for A' and a' are derived:

$$A' = Rvf / (H - Zv) \qquad (3.9)$$
$$a' = Rf / (H - Z) \qquad (3.10)$$

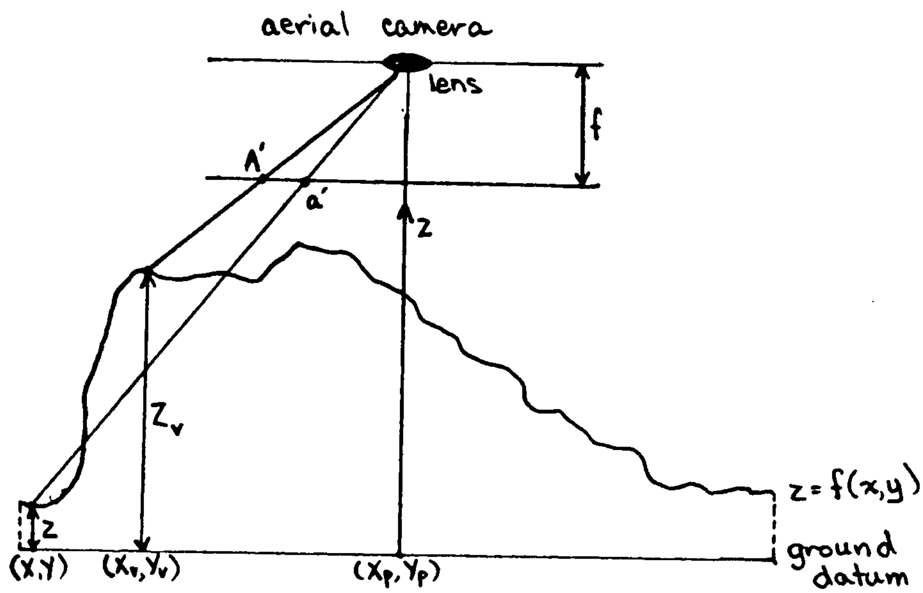
Therefore, substituting into formula (3.6), terrain between coordinates (Xv,Yv) and (Xp,Yp) will block the visibility of the terrain at coordinate (X,Y) if

$$Rvf / (H - Zv) \geq Rf / (H - Z) \qquad (3.11)$$

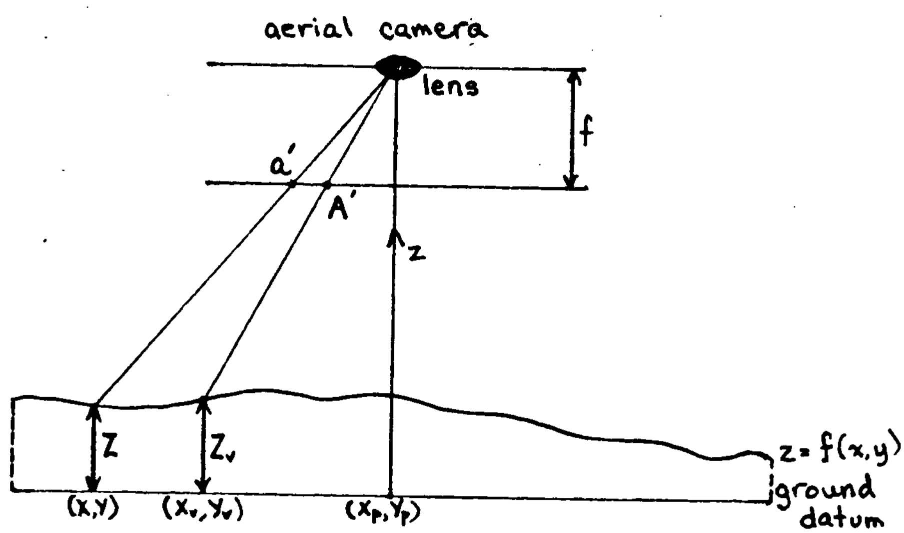
and hence, if

$$Rv / (H - Zv) \geq R / (H - Z) \qquad (3.12)$$

Figure 7 demonstrates the blocked visibility rule as described by equation (3.6). In figure 7(a), A' > a' and therefore, according to equation (3.6), the terrain between coordinates (Xv,Yv) and (Xp,Yp) blocks the visibility of



(a) Terrain at (X,Y,Z) is not visible.



(b) Terrain at (X,Y,Z) is visible.

Figure 7. Terrain visibility - perspective.

the terrain at coordinate (X,Y), as is shown. Figure 7(b) shows the opposite, a' > A', and the visibility of the terrain at coordinate (X,Y) is not blocked by the terrain between coordinates (Xv,Yv) and (Xp,Yp).

The theory of producing a perspective projection from an orthographic projection is now complete. The remainder of this section deals with the problem of providing different perspective views to form the required stereo pair.

Varying perspective views can be accomplished by choosing different principal points for the hypothetical "camera". However, simple as this may seem, caution must be taken such that there is neither too little nor too much overlap between the left and right images of the stereo pair. For a fixed f and H, the amount of overlap and the amount of vertical exaggeration are determined by the distance between principal points. The closer the principal points, the more overlap and the less vertical exaggeration (i.e., the weaker the stereoscopic effect) [SABI78]. Too little overlap reduces the region available for stereoscopic viewing. Fifty-five to sixty-five percent overlap is suggested as an appropriate degree of overlap [LILL79], [SABI78]. The algorithm implemented to synthesize the perspective stereo pair is discussed in the following section.

3.2 Implementation

The algorithm implemented to synthesize a perspective stereo pair creates two perspective images - the left and right images of the stereo pair - plus two corresponding visibility maps. The original, orthographic image must already be registered to the corresponding digital terrain model (i.e., a one-to-one correspondance has been established between the elements of the DTM and the elements of the image) [LITT80]. Thus, the necessary elevation information is provided for each pixel in the orthographic image. The algorithm developed for this research is described as follows (the source code to produce the perspective stereo pair and the corresponding visibility maps is presented in Appendix B):

The left and right perspective images are generated independently; the algorithm described below to synthesize one perspective image is repeated to create the two images. The geometry of the "camera" (i.e., the elevation above the ground datum and the focal length - refer to figure 5) is assumed to be identical for the left and right perspective images; the only variance is geometry between the two images is the location of the principal points which are user input. The procedure to synthesize one perspective image is now described:

The lines of the orthographic image and the registered lines of the DTM are input a pair at a time (i.e., one image line and the corresponding DTM line). The line containing the principal point (the "centre line" is input first; working outwards, the lines at the edge of the image (and DTM) are input last. The perspective projection is thus created by displacing each X position of each DTM and image line pair, radially outwards from the principal point. For example, referring to figure 8, position A is displaced towards A' and, similarly, position B is displaced towards position B'. The amount of relief displacement introduced is calculated by equation (3.5). The introduction of displacement is performed by first converting the cartesian coordinates, (X,Y), of a position in the original image to polar coordinates, (R,θ). The displaced polar coordinates are then (R+d,θ), where d is the relief displacement as calculated by equation (3.5). These new polar coordinates are then converted back to cartesian coordinates, (X',Y'). (The conversions from cartesian to polar coordinates and from polar to cartesian coordinates are both performed via software lookup tables.) The image pixel value corresponding to position (X,Y) of the original image is then assigned to position (X',Y') of the perspective image.

When the relief displacement is introduced, it is likely that the (X',Y') coordinates do not align exactly with the uniform grid of the image. A nearest-neighbour resampling [LILL79] is performed to generate intermediate pixel values.

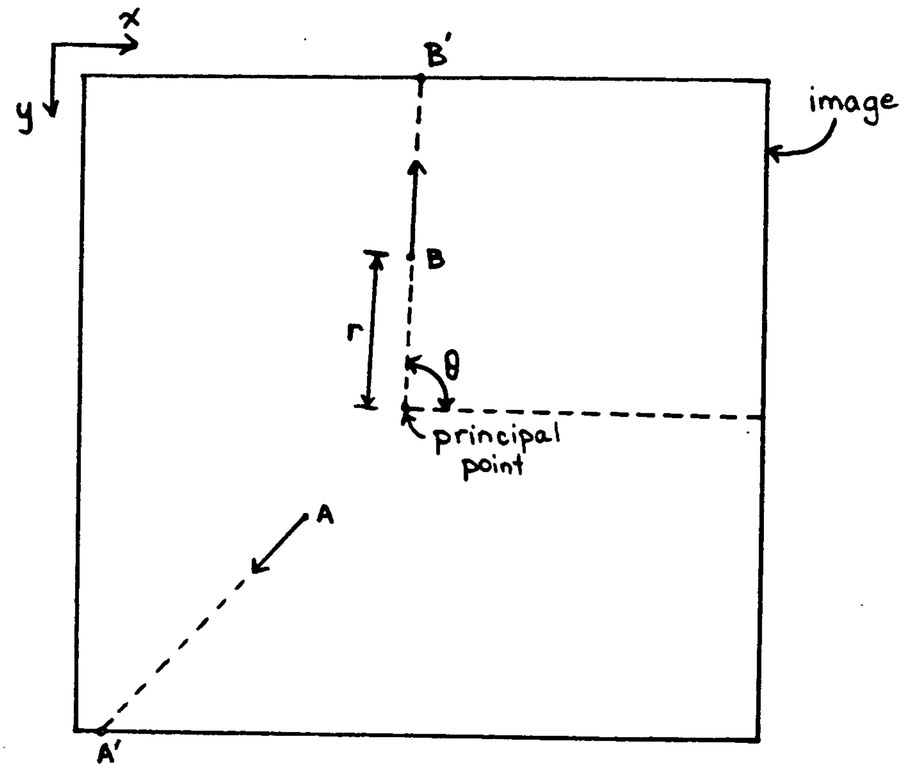


Figure 8. Direction of relief displacement.

A common situation is that a displaced pixel (i.e., one corresponding to position (X',Y')) is not visible in the perspective image. This condition is detected and resolved in the following manner:

For pixels of the orthographic image with the same polar angle θ, the pixels closest to (Xp,Yp) are considered first; the ones farthest away last. Applying this technique, if a pixel with polar coordinate (r,θ) is displaced to a position (i.e., to a polar coordinate (r',θ)) that is a distance J from (Xp,Yp), then another pixel (with polar coordinate (r'',θ)) that is subsequently considered is visible if and only if it is mapped to a position that is more than a distance J from (Xp,Yp). If the distance is less than J, the pixel is not visible and hence can be ignored. The visibility map is created using this procedure. Pixels of the orthographic image that are not visible in the perspective image are "marked" on the map.

In the above procedure, equation (3.12) is used to determine the relative positions of two pixels (as a function of their distances from (Xp,Yp)). Due to the image's uniform grid, it is rare that one pixel's polar angle equals that of another pixel. This problem is alleviated by linearly interpolating Rv and Zv from the coordinates of two visible displaced pixels whose polar angles encompass that of the pixel in question.

Another method to calculate the visibility of each pixel involves the association of radius information with each pixel of the synthetic image. The radius of a pixel in the orthographic image is recorded as a function of the position of that pixel in the perspective image; if two pixels are displaced to the same position, the one with the lesser radius is the visible pixel. The pixels do not have to be displaced in any specific order when using this approach. This method is computationally simpler than the method implemented. However, this alternative method requires drastically more memory to record the radii information. For this reason, the first method described above was chosen.

Another likely situation is that gaps occur in the synthetic image - no pixel of the original image is displaced or resampled to that position. These gaps are filled by using a cubic convolution resampling of the brightness values of the synthetic image [LILL79] to calculate the intermediate values. (Cubic convolution is used, as opposed to bilinear interpolation, to provide superior perceptual radiometric results at a small additional computational overhead.)

The entire synthetic image is stored in memory until all DTM and orthographic line pairs have been inputted and subsequently transformed. The complete image and the corresponding visibility map are then output to the appropriate image file and corresponding map file.

The algorithm implemented for synthesizing perspective stereo pairs produces a visually pleasing result. Viewing figure 3 stereoscopically produces no abnormal terrain features; the radiometry of the stereo model created appears smooth.

4.0 Synthetic Airborne Scanner Imagery

This chapter discusses the synthesis of an airborne scanner image from an orthographic image. Methods for calculating terrain visibility, as viewed by the scanner system, and for calculating pixel layover are also presented.

The synthetic image is obtained by simulating an airborne scanner system, with a given geometry, positioned on a suitable platform, scanning (i.e., sampling or imaging) the terrain below it. The terrain is approximated by a digital terrain model that is registered to the orthographic image. The elevation information is used to introduce displacement into the original image, according to the geometry of the scanning device. This information is also used in the visibility and pixel layover calculations.

Figure 9 is a synthetic side-looking airborne scanner image of the St. Mary Lake region of southeast British Columbia. The orthographic image from which it is synthesized is the same portion of the Landsat image used earlier. Figures 10 and 11 are the corresponding visibility map and pixel layover map, respectively.

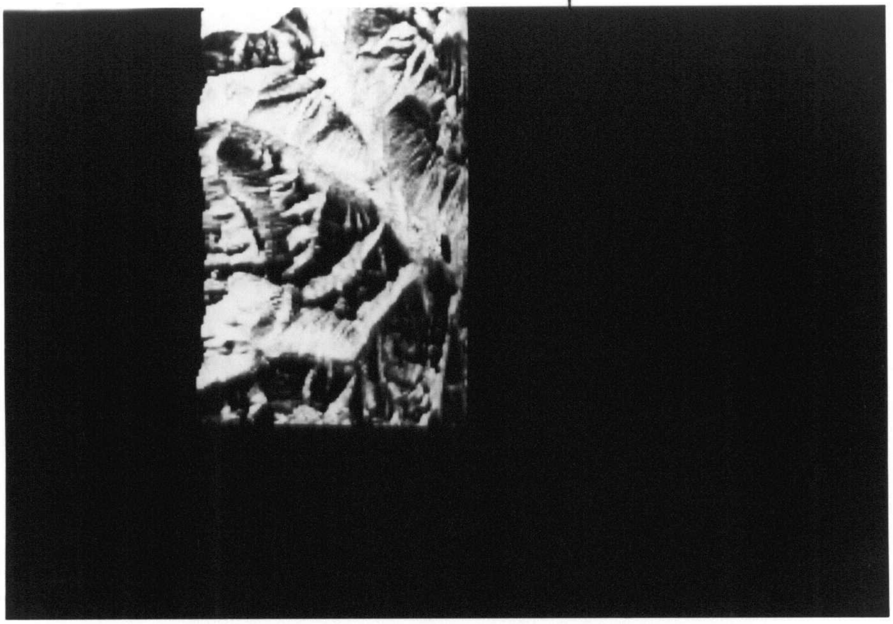


Figure 9. Airborne scanner image (St. Mary Lake region of southeast British Columbia).

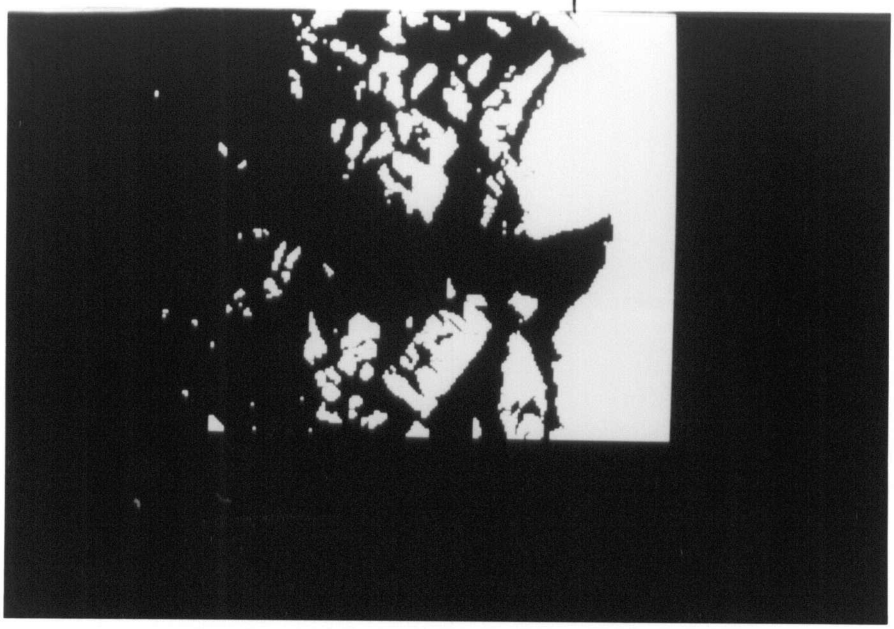


Figure 10. Airborne scanner visibility map.

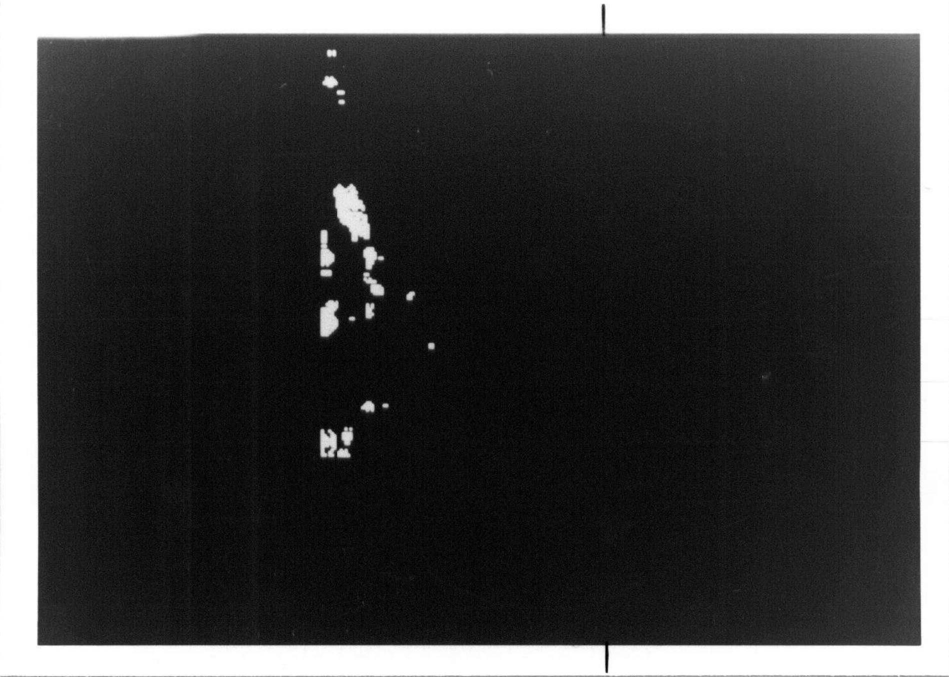


Figure 11. Airborne scanner pixel layover map.

4.1 Theory

The typical airborne scanner system (refer to figure 12) contains a rotating imaging assembly that moves the instantaneous field of view along scan lines. Scan lines run perpendicular to the flight path of the aircraft and extend the length (at ground level) of the angular field of view. An ideal scanner system samples electromagnetic radiation at fixed angular intervals along the scan line; the coverage by the system along the scan line is complete. An image produced by an airborne scanner system consists of measurements of radiation for a series of scan lines; each scan line on the ground is represented by a line on the image [LILL79].

This configuration of an airborne scanner system has two implications:

(a) each scan line is imaged independently of all other scan lines, and

(b) a one-to-one correspondence exists between a pixel of an image line and a sampling (imaging) of an instantaneous field of view of the associated scan line.

With these implications in mind, the geometric theory of imaging for an airborne scanner system is now presented. (Rules for calculating terrain visibility to the scanner and for determining pixel layover are also presented.)

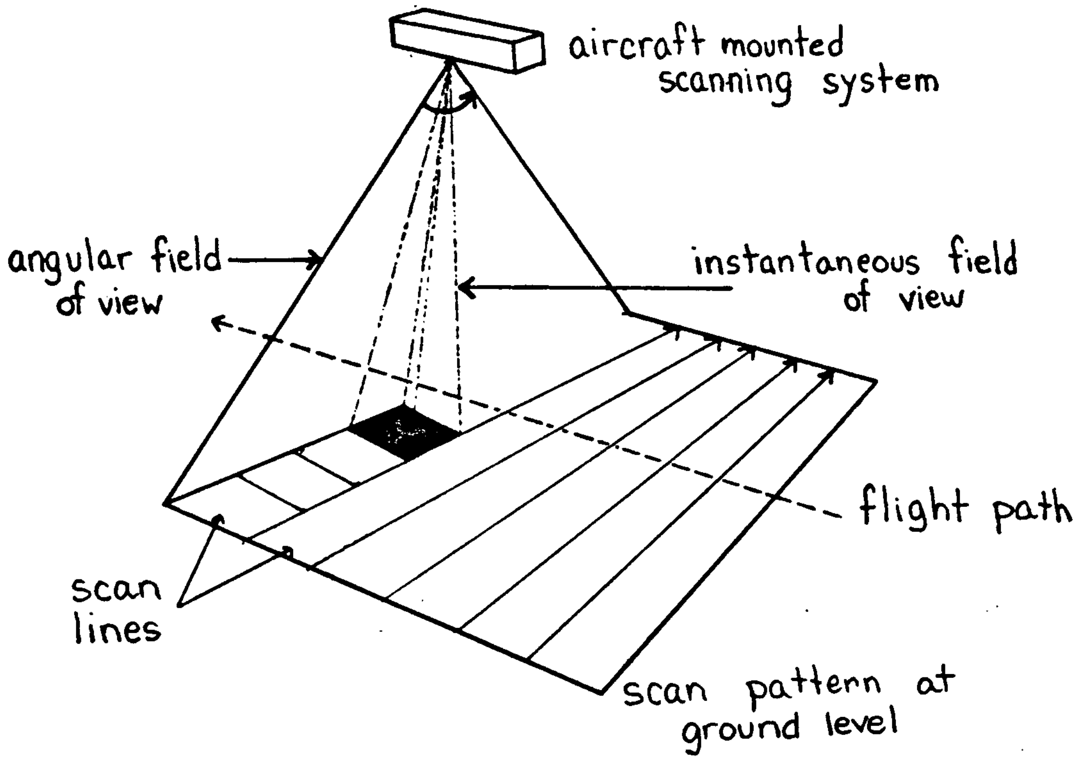


Figure 12. Operation of an airborne scanner system. (Adapted from [SABI69], figure 2.)

First, consider a digital terrain model as explained in section 2.2. Then, referring to figure 13, assume the airborne scanner system to be at elevation H above the ground datum and its flight path to be parallel to the y axis, directly over the coordinates (Xp,Y). Further suppose that the terrain at coordinates (X,Y) has elevation Z and it is imaged by the system with a Rotating Imaging Assembly (RIA) angle of θ. This angle is measured from the vertical (i.e., the vertical is 0 degrees). Clearly, from the geometry displayed in figure 13,

$$\tan\theta = (X - Xp) / (H - Z) \qquad (4.1)$$

and hence

$$\theta = \arctan\{ (X - Xp) / (H - Z) \} \qquad (4.2)$$

Equation (4.2) is the imaging equation for terrain where $X \geq Xp$, as shown in figure 13. In general:

$$\theta = \arctan\{ \mathrm{abs}\{ X - Xp \} / (H - Z) \} \qquad (4.3)$$

From equation (4.3), it can be seen that the RIA angle, θ, at which terrain (ground coordinate (X,Y)) is imaged at is a function of three parameters:

(a) Z, the elevation of the terrain above the ground datum,

(b) H, the elevation of the airborne scanner system above the ground datum, and

(c) Xp, the flight path of the airborne scanner system.

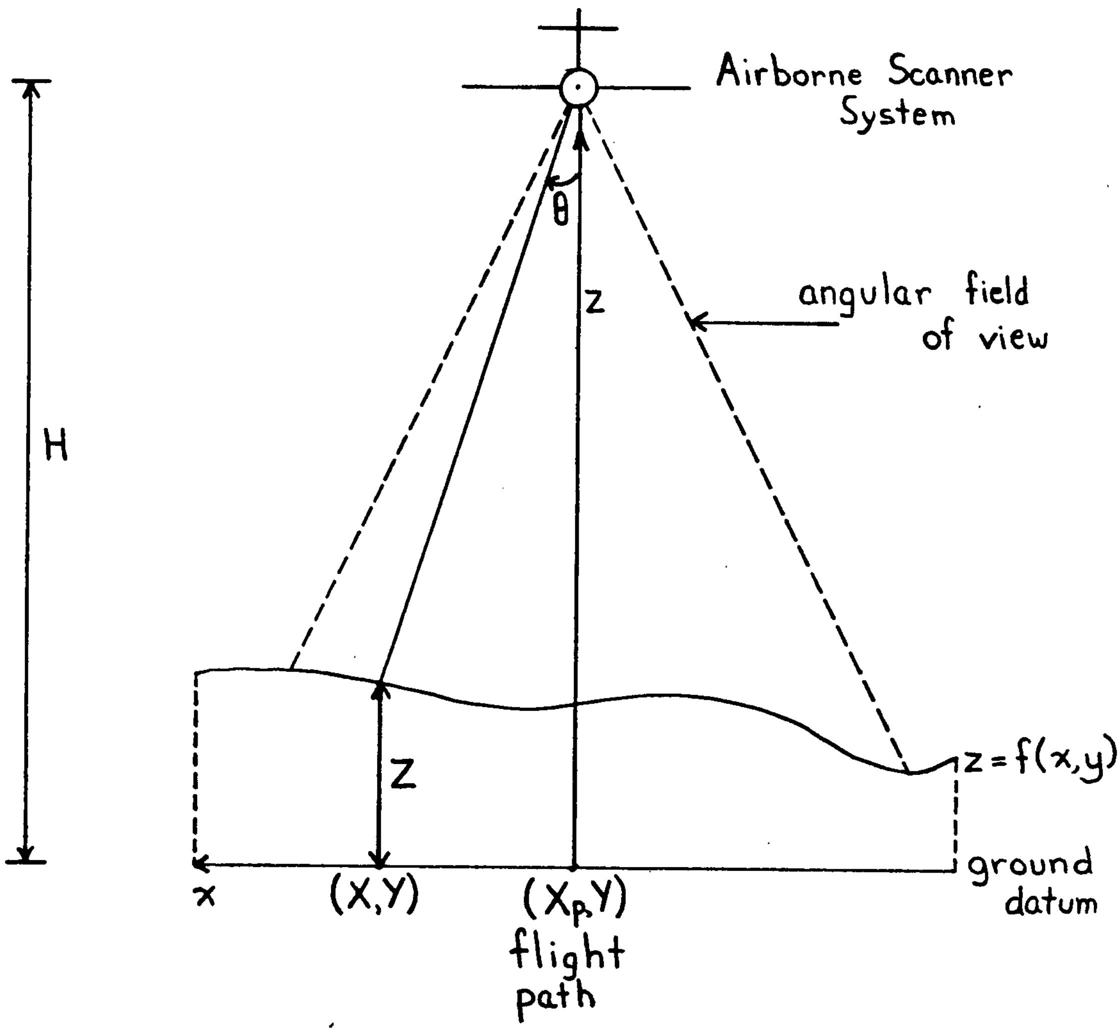


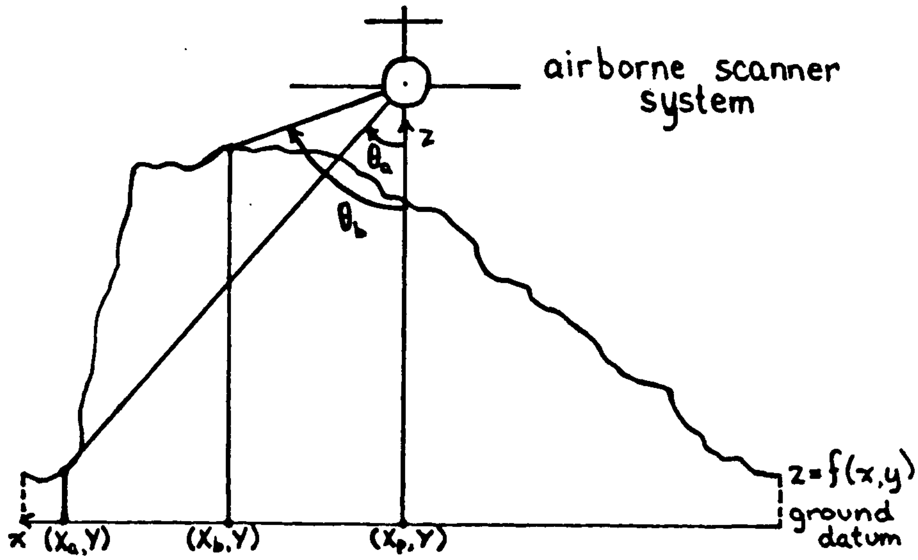
Figure 13. Geometry of an airborne scanner system.

(Equivalently, this third parameter may be referred to as abs{ X - Xp }, the horizontal distance between the terrain and the flight path.)

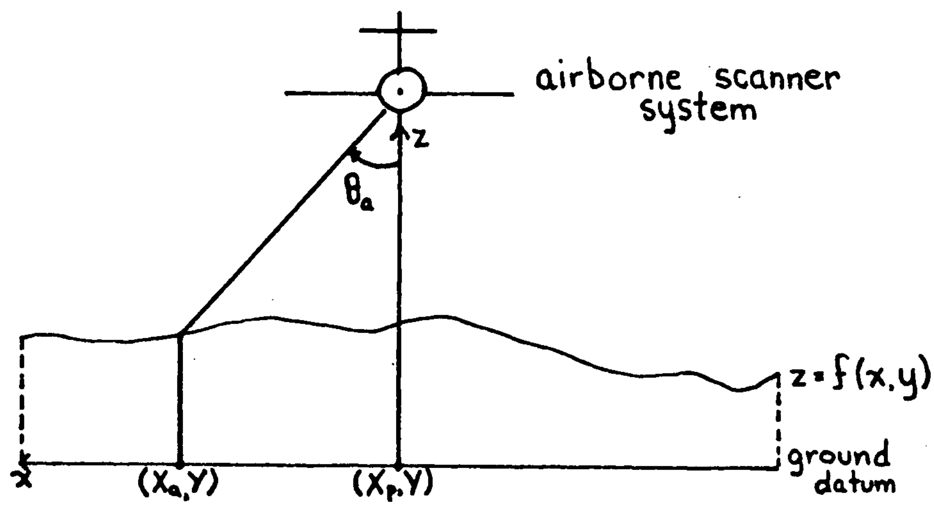
Since the imaging equation (4.3) is not a function of the y coordinate of the terrain, a scan line is imaged independently of other scan lines; the calculations to synthesize an airborne scanner image need not explicitly consider the y coordinate (assuming scan lines to be parallel to the x axis and hence the flight path to be parallel to the y axis; i.e., two points of terrain imaged on the same scan line will have the same y coordinate).

The calculation of terrain visibility, for an airborne scanner system with equation (4.3) as its imaging equation, is as follows:

Referring to figure 13, terrain at ground coordinate (X,Y), imaged at an RIA angle θ, is visible if and only if all visible terrain between itself and the flight path (ground coordinate (Xp,Y)) is imaged at an RIA angle of less than θ. Figure 14 demonstrates this rule. In figure 14(a), $\theta a < \theta b$ and Xb is between Xa and the flight path. Therefore, by the preceding visibility rule, terrain at ground coordinate (Xa,Y) is not visible, as is shown. In figure 14(b), there is no terrain between ground coordinates (Xa,Y) and (Xp,Y) that is imaged at an RIA angle larger than θa. Hence, terrain at ground coordinate (Xa,Y) is visible.



(a) Terrain at (Xa,Y) is not visible.



(b) Terrain at (Xa,Y) is visible.

Figure 14. Terrain visibility - airborne scanner.

4.1.1 Radar Considerations

The geometry of an airborne radar imaging system is very similar to that defined for the airborne scanner system (refer to figure 13). The difference in the two geometries is the position of terrain in the image produced. For the airborne scanner system, the terrain position is a function of the RIA angle. However, the position of terrain in a radar image is a function of the time required for a pulse of microwave energy to travel from the radar imaging system, to the terrain, and back to the system.[1] This time is directly proportional to the distance, D, the radar pulse travelled [JENS77]. Referring to figure 15, this distance will now be calculated. As for the airborne scanner system, assume the radar imaging system to be at elevation H above the ground datum and its flight path to be parallel to the y axis, directly over the coordinate (Xp,Y). Further assume that the terrain at coordinate (X,Y) has elevation Z above the ground datum and that the radar pulse must travel a distance of 2·D (i.e., D is the distance between the radar system and the terrain at coordinate (X,Y)). Clearly from the geometry shown in figure 15, the distance D is given by the following

----

[1] For a more detailed description of radar technology and radiometry, the reader is directed to [JENS77], [LILL79], [REEV75], and [SABI78].

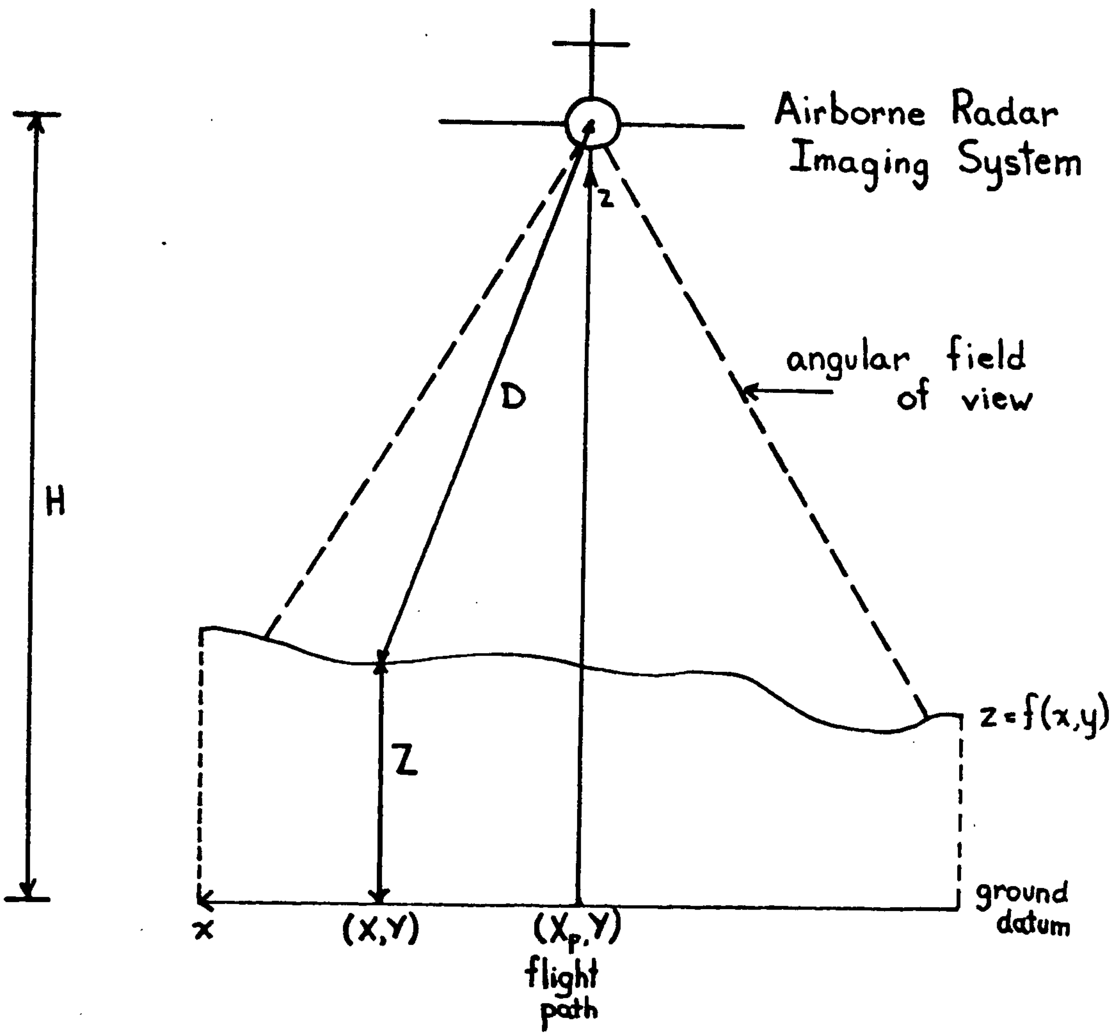


Figure 15. Geometry of a radar imaging system.

equation:

$$D = \mathrm{sqrt}\{ (Xp - X)^2 + (H - Z)^2 \} \qquad (4.4)$$

Therefore, the distance the radar pulse must travel is:

$$2 \cdot D = 2 \cdot \mathrm{sqrt}\{ (Xp - X)^2 + (H - Z)^2 \} \qquad (4.5)$$

Equation (4.5) is the imaging equation for an airborne radar imaging system whose geometry is displayed in figure 15. This equation for the distance the radar pulse travels and the equation for the RIA angle (equation (4.3)) are functions of the same parameters - the elevations of both the terrain and the airborne system as well as the x coordinate of the flight path, Xp. As with the airborne scanner system, the y coordinate is not explicitly considered in the imaging equation; i.e., a scan line is independent of other scan lines when it is imaged by the airborne radar imaging system. (Note: the angular field of view as shown in figure 15 is not typical of all radar imaging systems. Side-looking airborne radar [JENS77], for instance, has its field of view on one side of the aircraft only, hence its name. This fact, however, as in the airborne scanner system case, has little or no effect on the derivation of the imaging equation. Refer to figure 16.)

By comparing the geometries of the airborne scanner system and the airborne radar imaging system (figures 13 and 15 respectively), it is evident that terrain is equally

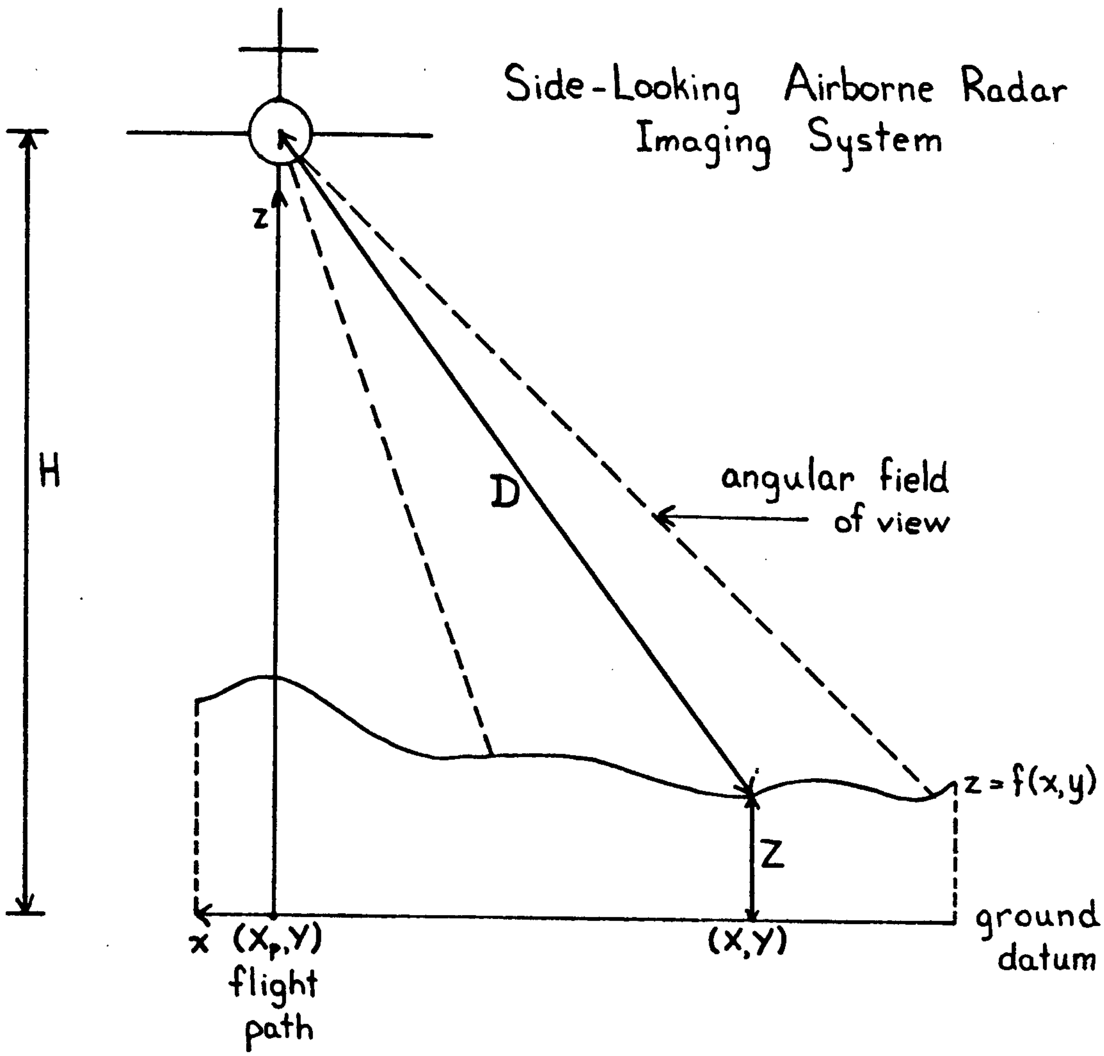


Figure 16. Geometry of a side-looking radar system.

visible, or invisible, to both systems (assuming equal geometric parameters - flight paths and elevations - for both systems). Therefore, terrain visibility for the radar system may be calculated in exactly the same manner as for the scanner system (i.e., by using equation (4.3)).

Pixel layover for the radar imaging system is also easily calculated. Given two points of terrain, both visible, (referring to figure 17) at ground coordinates (Xa,Y) and (Xb,Y), assume terrain elevations and pulse travel distances of Za, 2·Da and Zb, 2·Db respectively. Pixel layover therefore occurs when

$$Da = Db \qquad (4.6)$$

or, in more detail, when

$$\mathrm{sqrt}\{ (Xp - Xa)^2 + (H - Za)^2 \} = \mathrm{sqrt}\{ (Xp - Xb)^2 + (H - Zb)^2 \} \qquad (4.7)$$

While synthesizing the airborne scanner image, the visibility and pixel layover information for an airborne radar system may also be synthesized. The visibility information is equivalent for the two systems and the pixel layover calculation does not require more information than that necessary to synthesize the scanner imagery.

The methods described in this section for synthesizing airborne scanner images and for calculation of terrain visibility and pixel layover are purely geometrical in nature. The algorithms implemented to accomplish this

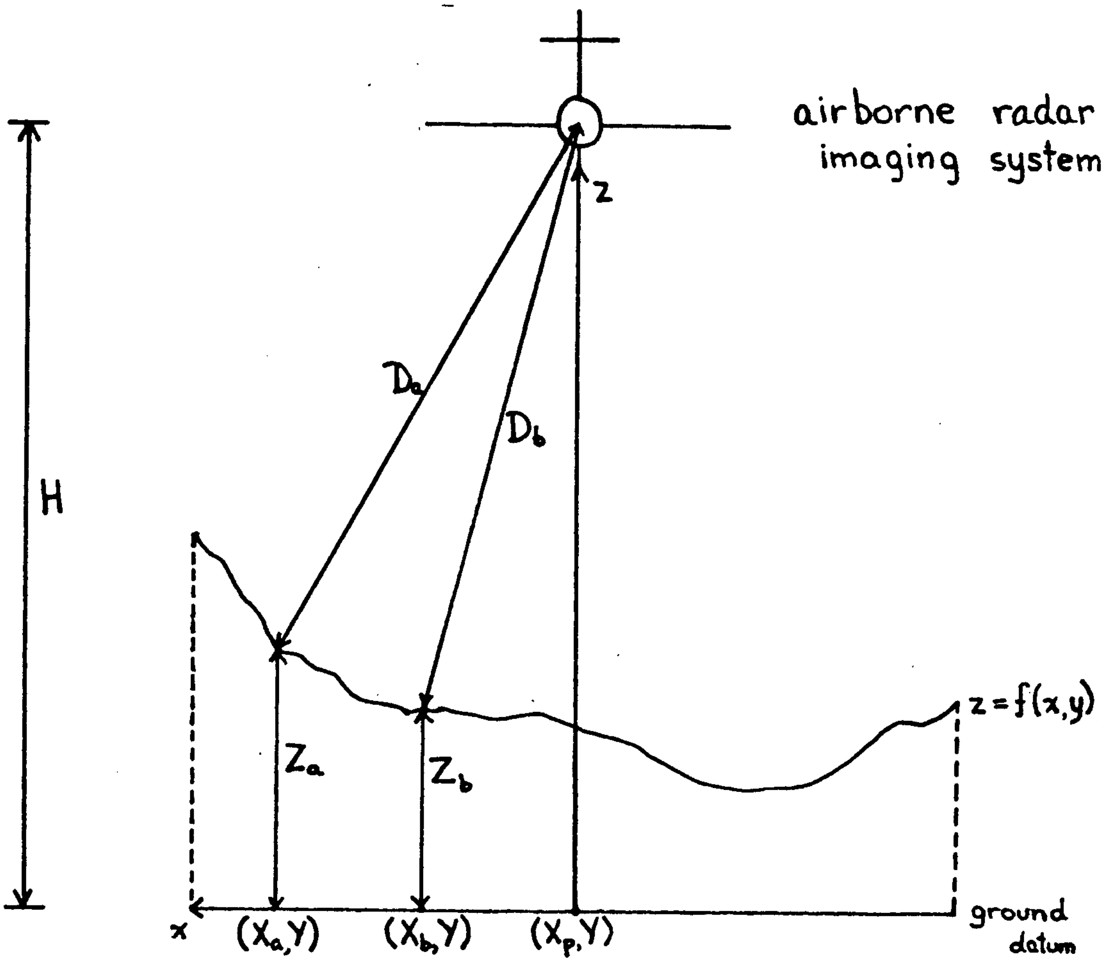


Figure 17. Pixel layover - radar.

synthesis are detailed in the following section.

4.2 Implementation

The algorithm to synthesize an airborne scanner image creates a new image plus corresponding visibility and pixel layover maps (production of the maps is optional). The original orthographic image must already be registered to a corresponding digital terrain model (i.e., a one-to-one correspondence has been established between the elements of the DTM and the elements of the image) [LITT80]. This associates the necessary elevation information to each pixel of the image. Assuming this registration has been successfully performed, the algorithm implemented is as follows (the source code to produce the synthetic airborne scanner imagery and the corresponding visibility and pixel layover maps is presented in Appendix C):

The grid of a synthesized image line is a function of the RIA angle (refer to section 4.1) and the grid interval is a function of the angular field of view of the hypothetical airborne scanner system.

The lines of the orthographic image and the registered lines of the DTM are input a pair at a time (i.e., one image line and the corresponding DTM line). For each image and DTM line pair, the corresponding line of the scanner image is created using equation (4.3); for each X position in the original line, a θ is calculated for the new line. (The arctan function is calculated via a software lookup table.) The image pixel value corresponding to position X of the original image line is assigned to position θ of the synthesized image line. With this transformation, it is likely that θ does not align exactly with the uniform grid of the image. The situation is resolved by performing a simple one dimensional linear resampling [LILL79] to produce the intermediate pixel values.

Another possibility is that a displaced pixel (i.e., one corresponding to position θ) may not be visible in the new airborne scanner image. This condition is detected and resolved in the following manner:
For a given line in the original image, the pixels closest to Xp are considered first; the ones farthest away last. Using this method, if a pixel of the line is hypothetically imaged at the RIA angle θ', then another pixel of that line that is subsequently considered is visible if and only if it is hypothetically imaged at an RIA angle greater than θ'. If the RIA angle is less than or equal to θ', the pixel is not visible and hence can be ignored. This procedure is used to calculate the visibility map. Pixels of the original image that are not visible in the synthesized image are "marked" on the map.

It is also likely that a pixel is displaced more than one grid cell from its predecessor. This produces gaps in the new line as no pixel of the original line is mapped or resampled to that position. In this situation, the gaps are filled by using linear interpolation to calculate the intermediate values.

Pixel layover can be calculated by using equation (4.7). However, the square roots on both sides of the equation are redundant; if equation (4.7) is true then so is

$$(Xp - Xa)^2 + (H - Za)^2 = (Xp - Xb)^2 + (H - Zb)^2 \quad (4.8)$$

The pixel layover map is therefore calculated as follows: For each pixel of the original line that is visible in the synthesized line (visibility calculations are performed before pixel layover calculations), the square of D (i.e., the square of one-half of the radar pulse distance) is calculated for each visible pixel using equation (4.4). These calculations are performed only once for each pixel. Then, employing equation (4.8) and the values just calculated, visible pixels are compared for "equal" pulse distance. Due to the discrete nature of DTMs, this equality will seldom, if ever, occur. This situation is resolved as follows:

If the pulse distance of the visible pixel in question falls between the pulse distances of two other consecutive visible pixels, then the former pixel as well as the latter pixel whose pulse distance is nearer the pulse distance in question are both considered "equal" as calculated by equation (4.8). If two pixels are "equal" as calculated above, then they are both "marked" on the pixel overlay map. If two pixels are not "equal" in pulse distance then the map is not "marked". The final map corresponds to the orthographic image.

Once the new line has been completely synthesized, including resampling and interpolation, and the visibility and pixel layover calculations are completed, the synthesized image line and the associated map lines are output to the corresponding image file and map files. The entire process is repeated for each line of the original image.

As displayed in figure 9, this algorithm produces a visually pleasing result, however a comparison to an actual airborne scanner image of the same geographic area was not possible.

5.0 Conclusion

This thesis has presented three techniques to use a surface model to alter the geometry of a real image. The three techniques are:

(a) synthesizing orthographic stereo pairs,

(b) synthesizing perspective stereo pairs, and

(c) synthesizing airborne scanner imagery.

The surface is determined by a digital terrain model (DTM). The DTM and the real image are registered to one another thus providing the necessary geometry - radiometry correspondance. Using the DTM, the geometry of the real image is altered to produce a synthetic image.

The algorithms implemented for this research have been kept computationally simple. For example, the implementations for synthesizing orthographic stereo pairs and airborne scanner imagery assume the flight path of the hypothetical sensing platform to be parallel to the Y axis of the DTM. Such a constraint can be considered unrealistic (but can be alleviated by a prior rotation of the DTM as required). However, the results of the three implementations appear accurate and, in all cases, produce a visually pleasing result.

5.1 Future Work

As previously stated, the algorithms for this research are simple in nature. These implementations can easily be extended to produce more practical results. For example, the flight path of the hypothetical sensing platform could be depicted as a function of x and y (i.e., platform position = f(x,y)). Alternatively, the flight parameters of the platform (roll, yaw, and pitch) may also be required to be simulated for certain analysis (eg., platform modelling).

These implementations have provided a potentially useful tool. Future research can now be directed towards exploiting these tools for applications including: photo interpretation aids, evaluation of the accuracy of image rectification algorithms, and analysis of distortion correction routines for airborne scanner imagery.

Bibliography

[BATS76] Batson, R. M., Kathleen Edwards, and E. M. Eliason. "Synthetic Stereo and LANDSAT Pictures", Photogrammetric Engineering and Remote Sensing, Vol. 42, No. 10, October 1976. pp. 1279-1284.

[BERN75] Bernstein, Ralph, and Dallam G. Ferneyhough, Jr. "Digital Image Processing", Photogrammetric Engineering and Remote Sensing, Vol. 41, No. 12, December 1975. pp. 1465-1476.

[DOYL78] Doyle, Frederick J. "The Next Decade of Satellite Remote Sensing", Photogrammetric Engineering and Remote Sensing, Vol. 44, No. 2, February 1978. pp. 155-164.

[JENS77] Jenson, H., L. C. Graham, L. J. Porcello, and E. M. Leith. "Side-looking Airborne Radar", Scientific American, Vol. 237, 1977. pp. 84-95.

[JOLL79] Jolliffe, Bruce, and Bary W. Pollack. "UBC PASCAL - Pascal/UBC Programmer's Guide", Computing Centre, University of British Columbia, September 1979.

[LILL79] Lillesand, Thomas M., and Ralph W. Kiefer. Remote Sensing and Image Interpretation. John Wiley & Sons Inc., New York, 1979. pp. 77-85, 443-445, 488-527, 560-561, 577-578, 579-583.

[LITT80] Little, James J. "Automatic Rectification of LANDSAT Images Using Features Derived From Digital Terrain Models", Technical Report 80-10, Department of Computer Science, University of British Columbia, December 1980.

[MILL58] Miller, C. L., and R. A. Laflamme. "The Digital Terrain Model - Theory and Application", Photogrammetric Engineering, Vol. 24, No. 3, June 1958. pp. 433-442.

[NEWM73] Newman, William M., and Robert F. Sproull. Principles of Interactive Computer Graphics, first edition. McGraw-Hill Inc., New York, 1973. pp. 267-268.

[NEWM79] Newman, William M., and Robert F. Sproull. Principles of Interactive Computer Graphics, second edition. McGraw-Hill Inc., New York, 1979. pp. 54-58, 333-336.

[REEV75] Reeves, Robert G., ed. Manual of Remote Sensing. The American Society of Photogrammetry, Falls Church, Virginia, 1975. pp. 399-535, 915-916, 2062-2110.

[SABI69] Sabins, Floyd F. "Thermal Infrared Imagery and its Application to Structural Mapping in Southern California", Geological Society America Bulletin, Vol. 80, 1969. pp. 397-404.

[SABI78] Sabins, Floyd F. Remote Sensing Principles and Interpretation, W. H. Freeman and Company, San Francisco, 1978. pp. 25-29, 100-105, 177-231, 246-247, 405-414.

[WOOD80] Woodham, Robert J. "Photometric Method for Determining Surface Orientation from Multiple Images", Optical Engineering, Vol. 19, No. 1, January/February 1980. pp. 139-149.

Appendix A

This appendix contains the source code that was used to produce synthetic orthographic stereo pairs. This source is written is UBC PASCAL [JOLL79].

```
(* ####################################################################### *)
(*                                                                         *)
(*  S Y N T H E T I C   O R T H O G R A P H I C   S T E R E O   P A I R S  *)
(*                                                                         *)
(* With  input  of  a  DTM  and  a  corresponding  registered  image,  an  *)
(* orthographic stereo pair of that image is created.  Resampling is      *)
(* done to remove quantization effects.                                    *)
(*                                                                         *)
(* ####################################################################### *)

     const
        LINE  = 1;    PIXEL = 2;
        MAX_LINE_LEN        = 255;
        PROMPT              = ':';
        COMPRESSED          = true;
        INTEGRAL            = true;
        READ_REQD           = true;
        MIN_INTENSITY       = 0;
        MAX_INTENSITY       = 255;
        MAX_IMAGE_SIZE      = 256;
        INIT_VALUE          = -1;
        DEFAULT             = -1;
        DEG_TO_RAD          = 0.0174532925;


     type
        MSG_TYPE     = ( LINE_PROMPT, PIXEL_PROMPT, UNEXPECTED_BLANK,
                         UNEXPECTED_EOF, UNREC_CMD, NULL_NO, NON_INTEGRAL_VALUE,
                         UNIT_0_UNASSIGNED, INVALID_FILE, UNEXPECTED_INPUT_EOF,
                         INVALID_NO, EOF_NO, EOLN_MSG, NOT_IN_RANGE,
                         ENTER_GRID_SPACING, ENTER_CNTR_COL,
                         UNIT_1_UNASSIGNED, ENTER_ANG_CONV                   );
        INPUT_LINE  = array (. 1..MAX_LINE_LEN+1 .) of char;
        INT_LINE    = array (. 0..MAX_LINE_LEN+2 .) of int;
        REAL_LINE   = array (. 0..MAX_LINE_LEN+2 .) of real;


        VALUE_RANGE = ( POSITIVE, NON_ZERO, INTENSITY, LINE_LENGTH,
                         IMAGE_SIZE, NON_NEGATIVE, PERCENT );

        STR_3       = array (. 1..3  .) of char;
        STR_4       = array (. 1..4  .) of char;
        STR_6       = array (. 1..6  .) of char;
        STR_8       = array (. 1..8  .) of char;
        STR_44      = array (. 1..44 .) of char;
        STR_50      = array (. 1..50 .) of char;

        IMAGEFILE   = file of INPUT_LINE;
        DTMFILE     = file of array ( 1..(MAX_LINE_LEN+1)*2 ) of char;




     var
        DIM                    : array ( 1..2 ) of integer;
        IMAGE_FILE,
        LFILE, RFILE           : IMAGEFILE;
        DTM_FILE               : DTMFILE;
```

```
            GRID_SPAC,
            CONVERGENCE          : real;
            CENTER_COL           : integer;



    (* ####################################################################### *)
    (*                                                                         *)
    (*          FORWARD declarations for the Procedures of the system         *)
    (*                                                                         *)
    (* ####################################################################### *)
         procedure DISPLACE;                                           forward;
         procedure FILES;                                              forward;
         procedure GET_DIMENSIONS;                                     forward;
         procedure INTERPOLATE( var ILINE : INT_LINE;
                                var FACT  : REAL_LINE );               forward;
         procedure PRINT_MSG( MSG : MSG_TYPE; STRING : INPUT_LINE;
                                              SIZE   : integer );      forward;
         procedure RESAMPLE( X : real; I : integer;
                             var ILINE   : INT_LINE;
                             var FACT    : REAL_LINE );                forward;
         procedure SYS_INIT;                                           forward;
         procedure SYSTEM_ERROR( MESSAGE : STR_50 );                   forward;
         procedure WRITE_IMAGE_LINE( var IMAGE : IMAGEFILE;
                                         LINE  : INT_LINE;
                                         START : integer    );         forward;
         procedure WRITE_STRING( STR : INPUT_LINE; SIZE : integer );   forward;


    (* ####################################################################### *)
    (*                                                                         *)
    (*          FORWARD declarations for the Functions of the system          *)
    (*                                                                         *)
    (* ####################################################################### *)
         function GET_EOLN(var STR : INPUT_LINE; COMPRESS : boolean)
                                         : integer;                    forward;
         function GET_NUM( RANGE : VALUE_RANGE; INTEGRAL_VALUE : boolean )
                                         : real;                       forward;
         function READ_DTM_LINE( var DTM_FILE : DTMFILE; READ_REQD : boolean )
                                         : INT_LINE;                   forward;
         function READ_IMAGE_LINE( var IMAGE_FILE : IMAGEFILE;
                                   READ_REQD : boolean ) : INT_LINE;   forward;
         function READ_STRING : INPUT_LINE;                            forward;


    (* ####################################################################### *)
    (*                                                                         *)
    (*           Declarations for FORTRAN routines used by the system         *)
    (*                                                                         *)
    (* ####################################################################### *)
         procedure CALL_MTS( STR : INPUT_LINE; I : integer );  fortran 'CMDNOE';
         procedure DEFAULT_FNAME( CMD_STR : INPUT_LINE );      fortran 'FTNCMD';
         procedure GET_FNAME( CMD_STR : INPUT_LINE; LEN : integer;
                              var FNAME   : INPUT_LINE );      fortran 'FTNCMD';
         procedure SET_PREFIX( NEW : char; LENGTH : integer ); fortran 'SETPFX';
```

```

(* ################################################################## *)
(*                                                                    *)
(*   Introduce displacment into the original image to create two new  *)
(*   images - the left and right images of the stereo pair. The amount *)
(*   of displacment introduced at a particular pixel is a function of *)
(*   the relative elevation at that pixel, the location of the hypo-  *)
(*   thetical scanning device, and the angle of convergence of that   *)
(*   device. A one dimensional bilinear resampling process is performed *)
(*   to create the new images.                                        *)
(*                                                                    *)
(* ################################################################## *)
      procedure DISPLACE;

         var
            DLINE, ILINE,
            LLINE, RLINE            : INT_LINE;
            LFACT, RFACT            : REAL_LINE;
            I, J, X, XDIFF          : integer;
            COS_THETA, SIN_THETA,
            XLPREV, XRPREV,
            Z, XR, XL               : real;


         begin

            COS_THETA := cos( CONVERGENCE );
            SIN_THETA := sin( CONVERGENCE );

            DLINE := READ_DTM_LINE   ( DTM_FILE,   not READ_REQD );
            ILINE := READ_IMAGE_LINE( IMAGE_FILE, not READ_REQD );

            for I := 1 to DIM(LINE) do
               begin

                  for J := 1 to DIM(PIXEL) do
                     begin

                        LFACT(J) := 0.0;
                        RFACT(J) := 0.0;
                        LLINE(J) := MIN_INTENSITY;
                        RLINE(J) := MIN_INTENSITY;

                     end  (* for J *);


                  XLPREV := CENTER_COL + SIN_THETA * DLINE(CENTER_COL) / GRID_SPAC;
                  XRPREV := CENTER_COL - SIN_THETA * DLINE(CENTER_COL) / GRID_SPAC;

                  for X := (CENTER_COL-1) downto 1 do
                     begin

                        XDIFF := X - CENTER_COL;
                        Z     := DLINE(X) / GRID_SPAC;
                        XL    := CENTER_COL + COS_THETA * XDIFF +
                                              SIN_THETA * Z;
                        XR    := CENTER_COL + COS_THETA * XDIFF -
                                              SIN_THETA * Z;
```

```
      if (XL > O) and (XL < (DIM(PIXEL)+1)) and (XL < XLPREV) then
         begin
            RESAMPLE( XL, ILINE(X), LLINE, LFACT );
            XLPREV := XL;
         end  (* if (XL > O) ... *);

      if (XR > O) and (XR < (DIM(PIXEL)+1)) and (XR < XRPREV) then
         begin
            RESAMPLE( XR, ILINE(X), RLINE, RFACT );
            XRPREV := XR;
         end  (* if (XR > O) ... *);


   end  (* for X *);


XLPREV := CENTER_COL - COS_THETA + SIN_THETA * DLINE(CENTER_COL-1) / GRID_SPAC;
XRPREV := CENTER_COL - COS_THETA - SIN_THETA * DLINE(CENTER_COL-1) / GRID_SPAC;

for X := CENTER_COL to DIM(PIXEL) do
   begin

      XDIFF := X - CENTER_COL;
      Z     := DLINE(X) / GRID_SPAC;
      XL    := CENTER_COL + COS_THETA * XDIFF +
                            SIN_THETA * Z;
      XR    := CENTER_COL + COS_THETA * XDIFF -
                            SIN_THETA * Z;

      if (XL > O) and (XL < (DIM(PIXEL)+1)) and (XL > XLPREV) then
         begin
            RESAMPLE( XL, ILINE(X), LLINE, LFACT );
            XLPREV := XL;
         end  (* if (XL > O) ... *);

      if (XR > O) and (XR < (DIM(PIXEL)+1)) and (XR > XRPREV) then
         begin
            RESAMPLE( XR, ILINE(X), RLINE, RFACT );
            XRPREV := XR;
         end  (* if (XR > O) ... *);


   end  (* for X *);


INTERPOLATE( LLINE, LFACT );
INTERPOLATE( RLINE, RFACT );

WRITE_IMAGE_LINE( LFILE, LLINE, 1 );
WRITE_IMAGE_LINE( RFILE, RLINE, 1 );

if I < DIM(PIXEL) then
   begin
      DLINE := READ_DTM_LINE   ( DTM_FILE,   READ_REQD );
      ILINE := READ_IMAGE_LINE( IMAGE_FILE, READ_REQD );
   end  (* if I < DIM(PIXEL) *)
```

```
                  end  (* for I *);


            end   (*  DISPLACE  *);



      (* ###################################################################### *)
      (*                                                                        *)
      (*   Set up and attach the files necessary for the system. Unit 0 is the  *)
      (*   assumed input for the original DTM. If it is not assigned on the MTS *)
      (*   RUN command, the file name will be prompted for. Unit 1 is assumed   *)
      (*   input for the original registered image. Its file name will also be  *)
      (*   prompted for if not assigned. The file assigned to                   *)
      (*   to unit 10 will be used for the output of the left synthetic image and*)
      (*   the file assigned to unit 11 will be used for the output of the right *)
      (*   synthetic image. If these units are not attached, they will default  *)
      (*   to MTS temporary files -LEFT# and -RIGHT# respectively.              *)
      (*                                                                        *)
      (* ###################################################################### *)
         procedure FILES;

            var
               DTMFNAME, IMGFNAME    : INPUT_LINE;


            begin


               DTMFNAME := ' ';
               DEFAULT_FNAME( 'DEFAULT 0=*DUMMY*;' );
               GET_FNAME( 'QUERY FDNAME 0;', 0, DTMFNAME );

               if DTMFNAME = '*DUMMY*' then
                  begin

                     PRINT_MSG( UNIT_0_UNASSIGNED, '', 0 );
                     DTMFNAME := READ_STRING;
                     while DTMFNAME = ' ' do
                        begin
                           PRINT_MSG( INVALID_FILE, '', 0 );
                           DTMFNAME := READ_STRING;
                           if DTMFNAME = 'CANCEL' or DTMFNAME = 'HALT' then
                              halt;
                        end  (* while DTMFNAME = ' ' *);

                  end  (* if DTMFNAME = '*DUMMY*' *);


               IMGFNAME := ' ';
               DEFAULT_FNAME( 'DEFAULT 1=*DUMMY*;' );
               GET_FNAME( 'QUERY FDNAME 1;', 0, IMGFNAME );

               if IMGFNAME = '*DUMMY*' then
                  begin

                     PRINT_MSG( UNIT_1_UNASSIGNED, '', 0 );
                     IMGFNAME := READ_STRING;
```

```
                    while IMGFNAME = ' ' do
                       begin
                          PRINT_MSG( INVALID_FILE, '', 0 );
                          IMGFNAME := READ_STRING;
                          if IMGFNAME = 'CANCEL' or IMGFNAME = 'HALT' then
                             halt;
                       end  (* while IMGFNAME = ' ' *);

                 end  (* if IMGFNAME = '*DUMMY*' *);


              DEFAULT_FNAME( 'DEFAULT 10=-LEFT#;'  );
              DEFAULT_FNAME( 'DEFAULT 11=-RIGHT#;' );
              reset(   DTM_FILE,   DTMFNAME );
              reset(   IMAGE_FILE, IMGFNAME );
              rewrite( LFILE,      10 );
              rewrite( RFILE,      11 );


           end   (*  FILES  *);



     (* ################################################################### *)
     (*                                                                     *)
     (*   Prompt for and get the dimensions of the DTM.                     *)
     (*                                                                     *)
     (* ################################################################### *)
        procedure GET_DIMENSIONS;

           begin

              PRINT_MSG( LINE_PROMPT,  ' ', 0 );
              DIM(LINE)  := trunc( GET_NUM( IMAGE_SIZE, INTEGRAL ) );

              PRINT_MSG( PIXEL_PROMPT, ' ', 0 );
              DIM(PIXEL) := trunc( GET_NUM( IMAGE_SIZE, INTEGRAL ) );

           end   (*  GET_DIMENSIONS  *);



     (* ################################################################### *)
     (*                                                                     *)
     (*   Read and return the end of an input line. If COMPRESS is true then *)
     (*   remove multiple embedded blanks, otherwise return the actual input *)
     (*   line. The length of the input line is also returned.             *)
     (*                                                                     *)
     (* ################################################################### *)
         function GET_EOLN( var STR:INPUT_LINE; COMPRESS:boolean ) : integer;

            var
               I                   : integer;
               LAST_CHAR_BLANK     : boolean;
               CHR                 : char;


            begin
```

```
                I := 0;
                LAST_CHAR_BLANK := false;

                while ¬eof( INPUT ) and ¬eoln( INPUT ) do
                   begin
                      read( CHR );
                      if CHR ¬= ' ' or not COMPRESS then
                         begin
                            if LAST_CHAR_BLANK and I>0 then
                               begin
                                  I := I + 1;
                                  STR(I) := ' '
                               end;
                            I := I + 1;
                            STR(I) := CHR;
                            LAST_CHAR_BLANK := false
                         end;
                      else
                         LAST_CHAR_BLANK := true
                   end;

                if not COMPRESS and I>0 then
                   GET_EOLN := I - 1;
                else
                   GET_EOLN := I

             end   (*  GET_EOLN  *);




   (* ################################################################## *)
   (*                                                                    *)
   (*   Return the numerical value of a string in the input stream. If   *)
   (*   INTEGRAL_VALUE is true then the string must represent an integer, *)
   (*   otherwise a real number. The numerical value must also be in the *)
   (*   range specified by RANGE.                                        *)
   (*                                                                    *)
   (* ################################################################## *)
         function GET_NUM( RANGE : VALUE_RANGE;
                           INTEGRAL_VALUE : boolean ) : real;

             var      STR          : INPUT_LINE;
                      VALID        : boolean;
                      VALU         : real;
                      I, J, MULT   : integer;

          begin
             repeat
                VALID := true;
                read( STR );
                if eof( INPUT ) then         (* CHECK FOR EOF               *)
                   begin
                      VALID := false;
                      PRINT_MSG( EOF_NO, ' ', 0 )
                   end;
                else
                   begin
```

```
VALU := 0.0;         (* GET INTEGRAL PART          *)
I := 1;
MULT := 1;
while I <= MAX_LINE_LEN and STR(I) = ' ' do
   I := I + 1;
if STR(I) = '+' or STR(I) = '-' then
   begin
      if STR(I) = '-' then
         MULT := -1;
      I := I + 1
   end;
while STR(I) in (. '0'..'9' .) do
   begin
      VALU := 10.0 * VALU + ord( STR(I) ) - ord('0');
      I := I + 1
   end;
if STR(I) = '.' then
   begin             (* GET FRACTIONAL PART        *)
      J := 10;
      I := I + 1;
      while STR(I) in (. '0'..'9' .) do
         begin
            VALU := VALU + ( (ord(STR(I)) - ord('0'))/J);
            I := I + 1;
            J := J * 10
         end
   end;
if STR(I) ¬= ' ' or ( eoln(INPUT) and I = MAX_LINE_LEN+1 ) then
   begin             (* INVALID NUMBER             *)
      VALID := false;
      if STR(I) ¬= ' ' then
         PRINT_MSG( INVALID_NO, STR(I), 1  )
      else
         if RANGE in (. PERCENT .) then
            begin
               VALID := true;
               MULT  := 1;
               VALU  := DEFAULT;
            end (* if RANGE in *);
         else
            PRINT_MSG( NULL_NO, ' ', 0 )
   end
else if INTEGRAL_VALUE and trunc(VALU) ¬= VALU then
   begin
      VALID := false;
      PRINT_MSG( NON_INTEGRAL_VALUE, '', 0 )
   end
else if STR(I) = ' ' and ( STR(I-1)='-' or STR(I-1)='+' ) then
   begin
      VALID := false;
      PRINT_MSG( UNEXPECTED_BLANK, '', 0 )
   end
else if (RANGE = POSITIVE and (MULT < 0 or VALU = 0.0)) or
        (RANGE = IMAGE_SIZE and (MULT < 0  or
                  VALU < 1 or VALU > MAX_IMAGE_SIZE))    or
        (RANGE = LINE_LENGTH and (MULT < 0 or
                  VALU < 1 or VALU > DIM(PIXEL)))        or
        (RANGE = INTENSITY and (MULT < 0   or
```

```
                                   VALU < MIN_INTENSITY      or
                                   VALU > MAX_INTENSITY))              or
                             (RANGE = NON_NEGATIVE and MULT < 0)       or
                             (RANGE = PERCENT and (MULT < 0      or
                                   VALU < 0 or VALU > 100))            or
                             (RANGE = NON_ZERO and VALU = 0.0)         then
                         begin
                            VALID := false;
                            PRINT_MSG( NOT_IN_RANGE, '', 0 )
                         end;
                     end;
                  until VALID;
                GET_NUM := VALU * MULT
             end (* GET_NUM *);



      (* ######################################################################## *)
      (*                                                                          *)
      (*   Interpolate all undefined pixel values between the first defined      *)
      (*   pixel of the image line - ILINE - and the last defined pixel of the   *)
      (*   image line. The new values are calculated as a function of the        *)
      (*   values of the nearest defined pixels (i.e. the ones on either side    *)
      (*   of the undefined pixel ).                                              *)
      (*                                                                          *)
      (* ######################################################################## *)
          procedure INTERPOLATE( var ILINE : INT_LINE; var FACT : REAL_LINE );

             var
                X, XX, XEND, START,
                STOP, INCREMENT, I    : integer;


             begin

                X     := 1;
                while FACT(X)     = 0 and X < DIM(PIXEL) do
                   incr( X );

                XEND := DIM(PIXEL);
                while FACT(XEND) = 0 and XEND >= 1 do
                   decr( XEND );

                incr( X );
                repeat

                   if FACT(X) = 0 then
                      begin

                         I          := ILINE(X-1);
                         START      := X;
                         while FACT(X) = 0 do
                            incr( X );
                         STOP       := X - 1;
                         INCREMENT  := round( (ILINE(X) - ILINE(START-1)) / (X - START + 1) );
                         for XX := START to STOP do
                            begin
                               I := I + INCREMENT;
```

```
                              ILINE(XX) := round( (I + ILINE(XX) * FACT(XX)) /
                                                  (1.0 + FACT(XX)) );
                           end  (* for XX *);

                       end  (* if FACT(X) < 1 *);
                     incr( X );

                  until  X >= XEND;


                end   (*  INTERPOLATE  *);



    (* ##################################################################### *)
    (*                                                                       *)
    (*    Print the message specified by MSG to the output stream. The      *)
    (*    parameters STRING and SIZE may be involved in the details of      *)
    (*    that message.                                                      *)
    (*                                                                       *)
    (* ##################################################################### *)
          procedure PRINT_MSG( MSG : MSG_TYPE; STRING : INPUT_LINE;
                               SIZE : integer );

                const
                   MSG_PROMPT = ':';

                var
                   I,J            : integer;
                   STR            : INPUT_LINE;


                begin

                   SET_PREFIX( MSG_PROMPT, 1 );

                   if MSG in (. LINE_PROMPT, UNIT_0_UNASSIGNED,
                                UNIT_1_UNASSIGNED, ENTER_GRID_SPACING .) then
                      writeln;

                   case MSG of
                      UNEXPECTED_EOF   :
                            writeln(' Unexpected EOF -- Ignored');
                      UNREC_CMD        :
                            begin
                               writeln(' Invalid command -- Input line ignored');
                               if ¬eoln( INPUT ) then
                                  I := GET_EOLN( STR, COMPRESSED )
                            end;
                      EOLN_MSG         :
                            begin
                               I := GET_EOLN( STR, COMPRESSED );
                               if I ¬= 0 then
                                  begin
                                     write(' Invalid character(s): "');
                                     WRITE_STRING( STR, I );
                                     writeln('" -- Ignored')
                                  end
```

```
                    end;
                LINE_PROMPT     :
                    writeln('&Enter number of lines of DTM and IMAGE (Integer {1-256})');
                PIXEL_PROMPT    :
                    writeln('&Enter number of pixels per line        (Integer {1-256})');
                EOF_NO          :
                    writeln('&Unexpected EOF, Enter number');
                NULL_NO         :
                    writeln( '&Not expecting null line -- Ignored; Enter number' );
                NON_INTEGRAL_VALUE:
                    writeln('&Non-integral value, Re-enter number');
                UNEXPECTED_BLANK:
                    writeln('&Unexpected blank, Re-enter number');
                NOT_IN_RANGE    :
                    writeln('&Out of range, Re-enter number');
                UNIT_O_UNASSIGNED:
                    writeln(' Logical unit 0 has not been assigned.',
                              EOL, '&   Enter FDname of location of DTM   ');
                UNIT_1_UNASSIGNED:
                    writeln(' Logical unit 1 has not been assigned.',
                              EOL, '&   Enter FDname of location of IMAGE ');
                INVALID_FILE    :
                    writeln(' Invalid FDname. Enter again or', EOL,
                            '&Enter "CANCEL" to terminate program');
                UNEXPECTED_INPUT_EOF:
                    writeln(' Unexpected EOF encountered on input',
                            ' -- DTM size updated');
                INVALID_NO      :
                    begin
                       writeln('&Invalid character, "', STRING(1), '", Re-enter number');
                       I := GET_EOLN( STR, COMPRESSED )
                    end;
                ENTER_GRID_SPACING:
                    writeln('&Enter DTM grid spacing (positive Real)');
                ENTER_CNTR_COL  :
                    writeln('&Enter centre column for projection (Integer {1-',
                            DIM(PIXEL):0,'})');
                ENTER_ANG_CONV  :
                    writeln('&Enter angle of convergence (in positive degrees)');

              end  (*  case  *);

              if MSG in (. LINE_PROMPT, PIXEL_PROMPT, INVALID_NO,
                           NON_INTEGRAL_VALUE, UNEXPECTED_BLANK, NOT_IN_RANGE,
                           UNIT_O_UNASSIGNED, UNIT_1_UNASSIGNED, INVALID_FILE,
                           ENTER_GRID_SPACING, ENTER_CNTR_COL, ENTER_ANG_CONV,
                           EOF_NO, NULL_NO .) then
                 readln;

              SET_PREFIX( PROMPT, 1 )


           end  (*  PRINT_MSG  *);



   (* ####################################################################### *)
   (*                                                                          *)
```

```
(*   If READ_REQD is true then read in a line from the file DTM_FILE,        *)
(*   otherwise perform the subsequent operations on the present input        *)
(*   buffer. Convert the binary form of the buffer into integer form         *)
(*   assuming two bytes per pixel.                                           *)
(*                                                                           *)
(* ######################################################################## *)
     function READ_DTM_LINE( var DTM_FILE : DTMFILE; READ_REQD : boolean ) : INT_LINE;

        var
           LINE                 : INT_LINE;
           I                    : integer;


        begin

           if READ_REQD then
              get( DTM_FILE );
           for I := 1 to DIM(PIXEL) do
              LINE(I) := int( DTM_FILE@((I-1)*2 + 1) ) * 256 +
                         int( DTM_FILE@((I-1)*2 + 2) );
           READ_DTM_LINE := LINE;


        end   (*  READ_DTM_LINE  *);




(* ######################################################################## *)
(*                                                                          *)
(*   If READ_REQD is true then read in a line from the file IMAGE_FILE,     *)
(*   otherwise perform the subsequent operations on the present input       *)
(*   buffer. Convert the binary form of the buffer into integer form        *)
(*   assuming one byte per pixel.                                           *)
(*                                                                          *)
(* ######################################################################## *)
     function READ_IMAGE_LINE( var IMAGE_FILE : IMAGEFILE; READ_REQD : boolean ) : INT_LINE;

        var
           LINE                 : INT_LINE;
           I                    : integer;


        begin

           if READ_REQD then
              get( IMAGE_FILE );
           for I := 1 to DIM(PIXEL) do
              LINE(I) := int( IMAGE_FILE@(I) );
           READ_IMAGE_LINE := LINE;


        end   (*  READ_IMAGE_LINE  *);



(* ######################################################################## *)
(*                                                                          *)
```

```
      (*   Read a string from the input stream removing all leading blanks.      *)
      (*   The string is then returned.                                          *)
      (*                                                                         *)
      (* ###################################################################### *)
           function READ_STRING : INPUT_LINE;

              var
                 I                    : integer;
                 STR                  : INPUT_LINE;


              begin

                 STR := ' ';
                 I := 0;
                 repeat
                    incr( I );
                    read( STR(I) );
                    if STR(I) = ' ' and I = 1 then
                       I := 0;
                 until eof(INPUT) or eoln(INPUT) or (I ¬= 0 and STR(I) = ' ')
                                                   or I = MAX_LINE_LEN;

                 READ_STRING := STR;


              end   (*  READ_STRING  *);




      (* ###################################################################### *)
      (*                                                                         *)
      (*   Resample using a simple one dimensional bilinear scheme. The data     *)
      (*   to be resampled is intensity I at x-location X. The results are       *)
      (*   inserted in the image line ILINE.                                     *)
      (*                                                                         *)
      (* ###################################################################### *)
            procedure RESAMPLE( X : real; I : integer; var ILINE : INT_LINE;
                                                       var FACT  : REAL_LINE );

               var
                  LO, HI                   : integer;
                  LO_FACT, HI_FACT,
                  LO_I, HI_I               : real;


               begin

                  LO      := trunc( X );
                  HI      := LO + 1;
                  LO_FACT := HI - X;
                  HI_FACT := X  - LO;

                  if (LO > 0) and (LO_FACT <> 0) then
                     begin

                        LO_I      := ILINE(LO) * FACT(LO)  +  I * LO_FACT;
```

```
                        FACT(LO)  := FACT(LO) + LO_FACT;
                        ILINE(LO) := round( LO_I / FACT(LO) );

                     end  (* if (LO > 0) ... *);

                  if (HI <= DIM(PIXEL)) and (HI_FACT <> 0) then
                     begin

                        HI_I      := ILINE(HI) * FACT(HI)  +  I * HI_FACT;
                        FACT(HI)  := FACT(HI) + HI_FACT;
                        ILINE(HI) := round( HI_I / FACT(HI) );

                     end  (* if (HI <= DIM(PIXEL)) ... *);


               end   (*  RESAMPLE  *);



      (* ################################################################## *)
      (*                                                                     *)
      (*   Initialize the system by setting up the files, getting the dimensions *)
      (*   of the DTM, and reading the necessary system parameters.          *)
      (*                                                                     *)
      (* ################################################################## *)
            procedure SYS_INIT;

               var
                  I, J              : integer;


               begin

                  SET_PREFIX( PROMPT, 1 );
                  FILES;
                  GET_DIMENSIONS;

                  PRINT_MSG( ENTER_GRID_SPACING, '', 0 );
                  GRID_SPAC    := GET_NUM( POSITIVE, not INTEGRAL );

                  PRINT_MSG( ENTER_CNTR_COL,     '', 0 );
                  CENTER_COL   := trunc( GET_NUM( LINE_LENGTH, INTEGRAL ) );

                  PRINT_MSG( ENTER_ANG_CONV,     '', 0 );
                  CONVERGENCE  := GET_NUM( NON_NEGATIVE, not INTEGRAL ) *
                                  DEG_TO_RAD;


               end   (*  SYSTEM_INITIALIZATION  *);



      (* ################################################################## *)
      (*                                                                     *)
      (*   If a system error occurs, output the message MESSAGE and halt the *)
      (*   execution of the system.                                          *)
      (*                                                                     *)
      (* ################################################################## *)
```

```
         procedure SYSTEM_ERROR( MESSAGE : STR_50 );

            begin

               writeln(' ===> SYSTEM ERROR ===> ',MESSAGE );
               halt;


            end (* SYSTEM_ERROR *);




     (* ###################################################################### *)
     (*                                                                        *)
     (*   Write the line LINE to the file IMAGE starting at pixel START.       *)
     (*   It is assumed that each pixel has a value between 0 and 255.        *)
     (*                                                                        *)
     (* ###################################################################### *)
        procedure WRITE_IMAGE_LINE( var IMAGE : IMAGEFILE;
                                    LINE : INT_LINE; START : integer );

           var
              I                     : integer;


           begin

              IMAGE@ := '.';

              for I := 1 to (START-1) do
                 IMAGE@(I) := char( MIN_INTENSITY );

              for I := START to DIM( PIXEL ) do
                 IMAGE@(I) := char( LINE(I) );

              put( IMAGE );


           end   (*  WRITE_IMAGE_LINE  *);



     (* ###################################################################### *)
     (*                                                                        *)
     (*   Write the string STR to the present output stream. The string is of *)
     (*   length SIZE. If the length is less than one, the operation does not *)
     (*   take place.                                                          *)
     (*                                                                        *)
     (* ###################################################################### *)
         procedure WRITE_STRING( STR : INPUT_LINE; SIZE : integer );

            var
               I                    : integer;


            begin

```

```
              if SIZE > 0 then
                 for I := 1 to SIZE do
                    write( STR(I) )


            end   (*  WRITE_STRING  *);



    (* ############################################################################ *)
    (*                                                                              *)
    (*   Initialize the system and then proceed to displace the original           *)
    (*   image as a function of its relative height (as specified by the DTM).     *)
    (*                                                                              *)
    (* ############################################################################ *)
         begin   (*  MAIN ROUTINE  *)

            SYS_INIT;
            DISPLACE;


         end   (*  MAIN ROUTINE  *).
```

Appendix B

This appendix contains the source code that was used to produce synthetic perspective stereo pairs. This source is written is UBC PASCAL [JOLL79].

```
(* ######################################################################## *)
(*                                                                          *)
(*   P E R S P E C T I V E   I M A G E   &   V I S I B I L I T Y   M A P    *)
(*                                                                          *)
(*   With input of a DTM and a corresponding registered image, a           *)
(*   stereo pair of perspective images of the DTM, according to the        *)
(*   intensity values of the inputted image, are created as well as two    *)
(*   corresponding visibility maps for the DTM.                            *)
(*                                                                          *)
(* ######################################################################## *)

      const
         LINE   = 1;    PIXEL = 2;
         OCT_1  = 1;    OCT_2 = 2;
         OCT_3  = 3;    OCT_4 = 4;
         OCT_5  = 5;    OCT_6 = 6;
         OCT_7  = 7;    OCT_8 = 8;
         RDS    = 1;    THT   = 2;
         X      = 1;    Y     = 2;
         LEFT   = 1;    RIGHT = 2;
         DTM    = 1;    IMGE  = 2;

         MAX_LINE_LEN         = 255;
         PROMPT               = ':';
         COMPRESSED           = true;
         INTEGRAL             = true;
         READ_REQD            = true;
         RESET_REQD           = true;
         MIN_INTENSITY        = 0;
         MAX_INTENSITY        = 255;
         MAX_IMAGE_SIZE       = 256;
         MAX_HORZN_SIZE       = max( 360, MAX_IMAGE_SIZE+1 );
         WORDS_PER_ENTRY      = 1;
         BYTES_PER_WORD       = 4;
         UNDEFINED            = -1;
         DEFAULT              = -1;
         DEG_TO_RAD           = 0.0174532925;


      type
         MSG_TYPE     = ( LINE_PROMPT, PIXEL_PROMPT, UNEXPECTED_BLANK,
                          UNEXPECTED_EOF, UNREC_CMD, NULL_NO, NON_INTEGRAL_VALUE,
                          UNIT_0_UNASSIGNED, INVALID_FILE, UNEXPECTED_INPUT_EOF,
                          INVALID_NO, EOF_NO, EOLN_MSG, NOT_IN_RANGE,
                          ENTER_ELEVATION, ENTER_CENTER_X_COORD,
                          ENTER_GRID_SPACING, ENTER_FOCAL_LENGTH,
                          ENTER_PRINT_SIZE, PRINT_PRINT_SIZE,
                          UNIT_1_UNASSIGNED, ENTER_CENTER_Y_COORD            );

         INPUT_LINE   = array(. 1..MAX_LINE_LEN+1 .) of char;
         INT_LINE     = array(. 0..MAX_LINE_LEN+2 .) of int;
         REAL_LINE    = array(. 0..MAX_LINE_LEN+2 .) of real;
         SHORT_LINE   = array(. 0..MAX_LINE_LEN+2 .) of short;

         INT_ARRAY    = array(. 0..MAX_LINE_LEN+2 .) of INT_LINE;
         SHORT_ARRAY  = array(. 0..MAX_LINE_LEN+2 .) of SHORT_LINE;

         HORZN_LINE   = array(. 0..MAX_HORZN_SIZE .) of short;
```

```
          VALUE_RANGE = ( POSITIVE, NON_ZERO, INTENSITY, LINE_LENGTH,
                          IMAGE_SIZE, NON_NEGATIVE, PERCENT, LINES    );
          PIXEL_RANGE = 1..MAX_IMAGE_SIZE;
          MAPSET      = set of PIXEL_RANGE;

          STR_3       = array (. 1..3  .) of char;
          STR_4       = array (. 1..4  .) of char;
          STR_6       = array (. 1..6  .) of char;
          STR_8       = array (. 1..8  .) of char;
          STR_44      = array (. 1..44 .) of char;
          STR_50      = array (. 1..50 .) of char;

          IMAGEFILE   = file of INPUT_LINE;
          DTMFILE     = file of array( 1..(MAX_LINE_LEN+1)*2 ) of char;
          MAPFILE     = file of MAPSET;

          POINTER     = @ short;
          POLAR_COORD = array(. 1..2 .) of short;
          XY_COORD    = array(. 1..2 .) of short;




       var
          FULL_IMAGE_SIZE,
          HALF_IMAGE_SIZE        : integer;
          MAX_HALF               : int;
          PRINT_SIZE,
          FOCAL_LENGTH,
          H, GRID_SPAC           : short;
          DIM                    : array(. 1..2 .) of integer;
          CENTER                 : array(. LEFT..RIGHT, X..Y .) of int;
          XY_LOOKUP              : array(. 0..360, X..Y .) of short;
          LFILE, RFILE,
          IMAGE_FILE             : IMAGEFILE;
          DTM_FILE               : DTMFILE;
          LMFILE, RMFILE         : MAPFILE;
          THETA_LOOKUP,
          RADIUS_LOOKUP          : POINTER;
          IMAGE                  : INT_ARRAY;




  (* ################################################################## *)
  (*                                                                    *)
  (*        FORWARD declarations for the Procedures of the system       *)
  (*                                                                    *)
  (* ################################################################## *)
       procedure CIRCULAR_DISPLACE( Y_COORD, X_CENTER, Y_CENTER,
                                    START, STOP            : integer;
                                var IMAGE                  : INT_ARRAY;
                                var CURR_ILINE, CURR_DLINE,
                                    PREV_DLINE             : INT_LINE;
                                var MAP_LINE               : MAPSET;
                                var HORI_RAD, HORI_ALT     : HORZN_LINE );
                                                                  forward;
```

```
        procedure CUBIC_CONVOLUTION( var IMAGE : INT_ARRAY;
                                         X, Y  : integer     );       forward;
        procedure DISPLACE( var IMAGE          : INT_ARRAY;
                            var MFILE          : MAPFILE;
                            X_CENTER, Y_CENTER : integer     );       forward;
        procedure DISPLCE_CENTER_LINE( X_CENTER, Y_CENTER : integer;
                                   var IMAGE              : INT_ARRAY;
                                   var ILINE, DLINE       : INT_LINE;
                                   var MLINE              : MAPSET;
                                   var HORI_RAD, HORI_ALT : HORZN_LINE );
                                                                      forward;
        procedure EXPANDING_DISPLACE( Y_COORD,
                                      X_CENTER, Y_CENTER : integer;
                                  var IMAGE              : INT_ARRAY;
                                  var ILINE, DLINE       : INT_LINE;
                                  var MLINE              : MAPSET;
                                  var HORI_RAD, HORI_ALT : HORZN_LINE );
                                                                      forward;
        procedure EXPNDNG_HORIZON_UPDATE( Y, X_CENTER,
                                  PREV_START, PREV_STOP,
                                  PREV_Y                : integer;
                              var HORI_RAD, HORI_ALT    : HORZN_LINE );
                                                                      forward;
        procedure FILES;                                              forward;
        procedure GET_DIMENSIONS;                                     forward;
        procedure HORIZON_UPDATE( CURR_POLAR, PREV_POLAR : POLAR_COORD;
                                  CURR_ALT,   PREV_ALT   : int;
                              var HORI_RAD, HORI_ALT : HORZN_LINE ); forward;
        procedure INTERPOLATE( var IMAGE : INT_ARRAY );               forward;
        procedure INSERT_PIXEL( var IMAGE : INT_ARRAY;
                                    Y, X  : short;
                                    INTEN : integer      );           forward;
        procedure OCT_1_TRANSFORM( var X, Y, OCT : integer );         forward;
        procedure POLAR_INIT( SIZE : integer );                       forward;
        procedure PRINT_MSG( MSG  : MSG_TYPE; STRING : INPUT_LINE;
                             SIZE : integer                    ); forward;
        procedure PUT_ARRAY( ARR : POINTER; X, Y : integer;
                                          VALU : short     );         forward;
        procedure SYS_INIT;                                           forward;
        procedure SYSTEM_ERROR( MESSAGE : STR_50 );                   forward;
        procedure WRITE_IMAGE_LINE( var IMAGE : IMAGEFILE;
                                        LINE  : INT_LINE;
                                        START : integer  );           forward;
        procedure WRITE_MAP_LINE( var MAP     : MAPFILE;
                                      LINE    : MAPSET;
                                      LINE_NO : integer  );           forward;
        procedure WRITE_STRING( STR : INPUT_LINE; SIZE : integer );   forward;
        procedure WRT_IMAGE( var IMAGE_FILE : IMAGEFILE;
                             var IMAGE      : INT_ARRAY  );           forward;
        procedure XY_INIT;                                            forward;


     (* ################################################################### *)
     (*                                                                     *)
     (*          FORWARD declarations for the Functions of the system       *)
     (*                                                                     *)
     (* ################################################################### *)
        function ALLOCATE_SPACE( SIZE : integer ) : POINTER;          forward;
```

```
        function GET_ARRAY( ARR : POINTER; X, Y : integer ) : short; forward;
        function GET_EOLN( var STR : INPUT_LINE; COMPRESS : boolean)
                                      : integer;                forward;
        function GET_NUM( RANGE : VALUE_RANGE; INTEGRAL_VALUE : boolean )
                                      : real;                   forward;
        function POLAR_TRANSFORM( X, Y : integer ) : POLAR_COORD;    forward;
        function READ_DTM_LINE( var DTM_FILE : DTMFILE; READ_REQD : boolean )
                                      : INT_LINE;               forward;
        function READ_IMAGE_LINE( var IMAGE_FILE : IMAGEFILE;
                                 READ_REQD : boolean ) : INT_LINE;  forward;
        function READ_LINE( TYP, LINE_NO : integer; RESET_REQD : boolean )
                                      : INT_LINE;               forward;
        function READ_STRING : INPUT_LINE;                       forward;
        function VISIBLE( ALT1, ALT2, RAD1, RAD2 : short ) : boolean;
                                                                forward;
        function XY_TRANSFORM( RADIUS, THETA : short ) : XY_COORD;    forward;


     (* ################################################################## *)
     (*                                                                    *)
     (*           Declarations for FORTRAN routines used by the system     *)
     (*                                                                    *)
     (* ################################################################## *)
        procedure CALL_MTS( STR : INPUT_LINE; I : integer );  fortran 'CMDNOE';
        procedure DEFAULT_FNAME( CMD_STR : INPUT_LINE );      fortran 'FTNCMD';
        procedure GET_FNAME( CMD_STR : INPUT_LINE; LEN : integer;
                             var FNAME   : INPUT_LINE );      fortran 'FTNCMD';
        procedure SET_PREFIX( NEW : char; LENGTH : integer ); fortran 'SETPFX';





     (* ################################################################## *)
     (*                                                                    *)
     (*   Allocate space of SIZE words of memory and return starting location.  *)
     (*                                                                    *)
     (* ################################################################## *)
        function ALLOCATE_SPACE( SIZE : integer ) : POINTER;

           var
              PTR              : POINTER;


           begin

              getspace( 0, WORDS_PER_ENTRY * BYTES_PER_WORD * SIZE,
                        0, PTR                                    );
              if PTR = nil then
                 SYSTEM_ERROR( ' In "ALLOCATE_SPACE"; Unsuccessful allocate' );
              ALLOCATE_SPACE := PTR;


           end   (*  ALLOCATE_SPACE   *);



     (* ################################################################## *)
```

```
(*                                                                         *)
(*   Displace the pixels of CURR_ILINE and calculate terrain visibility    *)
(*   utilizing the circular horizon, HORI_RAD and HORI_ALT. The pixel's XY *)
(*   (cartesian) coordinates are first transformed to polar coordinates    *)
(*   and then the displacement is introduced. The new polar coordinates    *)
(*   are then converted to XY coordinates and the displaced pixel is       *)
(*   inserted in the perspective image at those coordinates.               *)
(*                                                                         *)
(* ####################################################################### *)
     procedure CIRCULAR_DISPLACE( Y_COORD, X_CENTER, Y_CENTER,
                                  START, STOP           : integer;
                              var IMAGE                 : INT_ARRAY;
                              var CURR_ILINE, CURR_DLINE,
                                  PREV_DLINE            : INT_LINE;
                              var MAP_LINE              : MAPSET;
                              var HORI_RAD, HORI_ALT    : HORZN_LINE );


        var
           HI, LO, I, J        : integer;
           D,
           LO_FACT, HI_FACT,
           H1, H2, R1, R2, RF  : short;
           SWTCH               : boolean;
           PREV_POLAR,
           CURR_POLAR          : POLAR_COORD;
           XY                  : XY_COORD;


        begin

           if START > STOP then
              begin
                 I     := START;
                 START := STOP;
                 STOP  := I;
                 SWTCH := true;
              end  (* if START > STOP *);
           else
                 SWTCH := false;

           for I := START to STOP do
              begin

                 if SWTCH then
                    J := STOP + START - I;
                 else
                    J := I;

                 CURR_POLAR := POLAR_TRANSFORM( J, Y_COORD   );
                 if Y_COORD > 0 then
                    PREV_POLAR := POLAR_TRANSFORM( J, Y_COORD-1 );
                 else
                    PREV_POLAR := POLAR_TRANSFORM( J, Y_COORD+1 );

                 LO      := trunc( CURR_POLAR(THT) );
                 HI      := LO + 1;
                 LO_FACT := HI - CURR_POLAR(THT);
                 HI_FACT := 1 - LO_FACT;
```

```
                    H1       := CURR_DLINE(J+X_CENTER);
                    RF       := CURR_POLAR(RDS) * FOCAL_LENGTH;
                    D        := RF / (H - H1) - RF / H;
                    R1       := CURR_POLAR(RDS) + D * (H / FOCAL_LENGTH);
                    H2       := LO_FACT * HORI_ALT(LO) + HI_FACT * HORI_ALT(HI);
                    R2       := LO_FACT * HORI_RAD(LO) + HI_FACT * HORI_RAD(HI);

                    if not VISIBLE( H1, H2, R1, R2 ) then
                       MAP_LINE := MAP_LINE + (. (MAX_IMAGE_SIZE-(J+X_CENTER)) .);
                    else
                       begin

                          XY    := XY_TRANSFORM( R1, CURR_POLAR(THT) );
                          XY(X) := XY(X) + X_CENTER;
                          XY(Y) := XY(Y) + Y_CENTER;
                          if (XY(X) > max( 0, X_CENTER-HALF_IMAGE_SIZE ))          and
                             (XY(Y) > max( 0, Y_CENTER-HALF_IMAGE_SIZE ))          and
                             (XY(X) <= min( DIM(PIXEL), X_CENTER+HALF_IMAGE_SIZE )) and
                             (XY(Y) <= min( DIM(LINE ), Y_CENTER+HALF_IMAGE_SIZE )) then
                             INSERT_PIXEL( IMAGE, XY(Y), XY(X),
                                           CURR_ILINE(J+X_CENTER) );
                          else
                             MAP_LINE := MAP_LINE + (. (MAX_IMAGE_SIZE-(J+X_CENTER)) .);

                       end  (* else VISIBLE *);

                    HORIZON_UPDATE( CURR_POLAR, PREV_POLAR,
                                    CURR_DLINE(J+X_CENTER),
                                    PREV_DLINE(J+X_CENTER),
                                    HORI_RAD, HORI_ALT       );

                 end  (* for I *);


            end   (*  CIRCULAR_DISPLACE  *);



      (* ################################################################## *)
      (*                                                                    *)
      (*   Perform a cubic convolution at location (X,Y) of the image IMAGE *)
      (*   to define that pixel.                                            *)
      (*                                                                    *)
      (* ################################################################## *)
         procedure CUBIC_CONVOLUTION( var IMAGE : INT_ARRAY;  X, Y : integer );

            var
               I, J, TOTAL, NUM    : integer;


            begin

               TOTAL := 0;
               NUM   := 0;

               for I := X-1 to X+1 do
                  for J := Y-1 to Y+1 do
```

```
                    if IMAGE(I,J) <> UNDEFINED then
                       begin
                          TOTAL := TOTAL + IMAGE(I,J);
                          incr( NUM );
                       end  (* if IMAGE(I,J) <> UNDEFINED *);
                 if NUM = 0 then
                    IMAGE(X,Y) := MIN_INTENSITY;
                 else
                    IMAGE(X,Y) := round( TOTAL / NUM );


           end   (*  CUBIC_CONVOLUTION  *);



     (* ###################################################################### *)
     (*                                                                        *)
     (*   Displace the elements of the orthographic image by introducing       *)
     (*   relief displacement to all pixels of that image. (This is            *)
     (*   accomplished by three routines: DISPLCE_CENTER_LINE which displaces  *)
     (*   the center image line (i.e. the line containing the principle point; *)
     (*   CIRCULAR_DISPLACE for pixels whose cartesian coordinates are such    *)
     (*   that abs(X) >= abs(Y), assuming the principle point to be the origin; *)
     (*   and EXPANDING_DISPLACE for all other pixels   i.e. pixels with       *)
     (*   abs(X) < abs(Y).)                                                    *)
     (*                                                                        *)
     (* ###################################################################### *)
        procedure DISPLACE( var IMAGE            : INT_ARRAY;
                            var MFILE            : MAPFILE;
                            X_CENTER, Y_CENTER : integer    );

           const
              CURR = 0;   PREV = 1;

           var
              I, J                    : integer;
              DLINE                   : array(. CURR..PREV .) of INT_LINE;
              ILINE_CURR              : INT_LINE;
              MLINE_CURR              : MAPSET;
              EXPD_HORI_RAD,
              EXPD_HORI_ALT,
              CIRC_HORI_RAD,
              CIRC_HORI_ALT           : HORZN_LINE;


           begin

              for I := 0 to MAX_LINE_LEN+2 do
                 for J := 0 to MAX_LINE_LEN+2 do
                    IMAGE(I,J) := UNDEFINED;

              for I := 0 to MAX_HORZN_SIZE do
                 begin
                    EXPD_HORI_RAD(I) := 0;
                    EXPD_HORI_ALT(I) := 0;
                    CIRC_HORI_RAD(I) := 0;
                    CIRC_HORI_ALT(I) := 0;
                 end  (* for I *);
```

```
          for I := Y_CENTER+1 to DIM(LINE) do
             begin

                DLINE(PREV) := DLINE(CURR);
                DLINE(CURR) := READ_LINE( DTM,  DIM(LINE)-I+1, RESET_REQD );
                ILINE_CURR  := READ_LINE( IMGE, DIM(LINE)-I+1, RESET_REQD );
                MLINE_CURR  := (. .);

                if (Y_CENTER-I) >= (-X_CENTER+1) then
                   CIRCULAR_DISPLACE( I-Y_CENTER, X_CENTER,       Y_CENTER,
                                      Y_CENTER-I, -X_CENTER+1,    IMAGE,
                                      ILINE_CURR, DLINE(CURR),    DLINE(PREV),
                                      MLINE_CURR, CIRC_HORI_RAD, CIRC_HORI_ALT );
                if (I-Y_CENTER) <= (DIM(PIXEL)-X_CENTER) then
                   CIRCULAR_DISPLACE( I-Y_CENTER, X_CENTER,               Y_CENTER,
                                      I-Y_CENTER, DIM(PIXEL)-X_CENTER, IMAGE,
                                      ILINE_CURR, DLINE(CURR),            DLINE(PREV),
                                      MLINE_CURR, CIRC_HORI_RAD,          CIRC_HORI_ALT );
                EXPANDING_DISPLACE( I-Y_CENTER, X_CENTER,       Y_CENTER,
                                    IMAGE,      ILINE_CURR,     DLINE(CURR),
                                    MLINE_CURR, EXPD_HORI_RAD, EXPD_HORI_ALT );

                WRITE_MAP_LINE( MFILE, MLINE_CURR, DIM(LINE)-I+1 );

             end  (* for I *);


          for I := 0 to MAX_HORZN_SIZE do
             begin
                EXPD_HORI_RAD(I) := 0;
                EXPD_HORI_ALT(I) := 0;
             end  (* for I *);

          DLINE(CURR) := READ_LINE( DTM,  DIM(LINE)-Y_CENTER+1, RESET_REQD );
          ILINE_CURR  := READ_LINE( IMGE, DIM(LINE)-Y_CENTER+1, RESET_REQD );
          for I := Y_CENTER-1 downto 1 do
             begin

                DLINE(PREV) := DLINE(CURR);
                DLINE(CURR) := READ_LINE( DTM,  DIM(LINE)-I+1, not RESET_REQD );
                ILINE_CURR  := READ_LINE( IMGE, DIM(LINE)-I+1, not RESET_REQD );
                MLINE_CURR  := (. .);

                if (I-Y_CENTER) >= (-X_CENTER+1) then
                   CIRCULAR_DISPLACE( I-Y_CENTER, X_CENTER,       Y_CENTER,
                                      I-Y_CENTER, -X_CENTER+1,    IMAGE,
                                      ILINE_CURR, DLINE(CURR),    DLINE(PREV),
                                      MLINE_CURR, CIRC_HORI_RAD, CIRC_HORI_ALT );
                if (Y_CENTER-I) <= (DIM(PIXEL)-X_CENTER) then
                   CIRCULAR_DISPLACE( I-Y_CENTER, X_CENTER,               Y_CENTER,
                                      Y_CENTER-I, DIM(PIXEL)-X_CENTER, IMAGE,
                                      ILINE_CURR, DLINE(CURR),            DLINE(PREV),
                                      MLINE_CURR, CIRC_HORI_RAD,          CIRC_HORI_ALT );
                EXPANDING_DISPLACE( I-Y_CENTER, X_CENTER,       Y_CENTER,
                                    IMAGE,      ILINE_CURR,     DLINE(CURR),
                                    MLINE_CURR, EXPD_HORI_RAD, EXPD_HORI_ALT );

```

```
                    WRITE_MAP_LINE( MFILE, MLINE_CURR, DIM(LINE)-I+1 );

                 end  (* for I *);

              DLINE(CURR) := READ_LINE( DTM,  DIM(LINE)-Y_CENTER+1, RESET_REQD );
              ILINE_CURR  := READ_LINE( IMGE, DIM(LINE)-Y_CENTER+1, RESET_REQD );
              MLINE_CURR  := (. .);
              CIRC_HORI_RAD(0)   := 0;
              CIRC_HORI_ALT(0)   := 0;
              CIRC_HORI_RAD(180) := 0;
              CIRC_HORI_ALT(180) := 0;

              DISPLCE_CENTER_LINE( X_CENTER,      Y_CENTER,     IMAGE,
                                   ILINE_CURR,    DLINE(CURR),  MLINE_CURR,
                                   CIRC_HORI_RAD, CIRC_HORI_ALT               );
              WRITE_MAP_LINE( MFILE, MLINE_CURR, DIM(LINE)-Y_CENTER+1 );


           end   (*  DISPLACE  *);



     (* ###################################################################### *)
     (*                                                                         *)
     (*   Displace the center line of the image (i.e. the line containing the   *)
     (*   principle point) utilizing the circular horizon, HORI_RAD and         *)
     (*   HORI_ALT, to calculate terrain visibility.                            *)
     (*                                                                         *)
     (* ###################################################################### *)
        procedure DISPLCE_CENTER_LINE( X_CENTER, Y_CENTER  : integer;
                                   var IMAGE               : INT_ARRAY;
                                   var ILINE, DLINE        : INT_LINE;
                                   var MLINE               : MAPSET;
                                   var HORI_RAD, HORI_ALT  : HORZN_LINE );

           var
              I                    : integer;
              RAD, RF, D           : short;


           begin

              for I := X_CENTER-1 downto 1 do
                 begin

                    RAD := abs( I - X_CENTER );
                    RF  := RAD * FOCAL_LENGTH;
                    D   := RF / (H - DLINE(I)) - RF / H;
                    RAD := RAD + D * (H / FOCAL_LENGTH);
                    if VISIBLE( DLINE(I), HORI_ALT(180),
                                RAD,      HORI_RAD(180) ) then
                       begin
                          HORI_RAD(180) := RAD;
                          HORI_ALT(180) := DLINE(I);
                          if X_CENTER - RAD > max( 0, X_CENTER-HALF_IMAGE_SIZE ) then
                             INSERT_PIXEL( IMAGE, Y_CENTER, X_CENTER-RAD, ILINE(I) );
                          else
                             MLINE := MLINE + (. (MAX_IMAGE_SIZE-I) .);
```

```
             end  (* if RAD > HORI_RAD(180) and ... *);
           else
             MLINE := MLINE + (. (MAX_IMAGE_SIZE-I) .);

         end  (* for I *);

       INSERT_PIXEL( IMAGE, Y_CENTER, X_CENTER, ILINE(X_CENTER) );

       for I := X_CENTER+1 to DIM(PIXEL) do
         begin

           RAD := abs( I - X_CENTER );
           RF  := RAD * FOCAL_LENGTH;
           D   := RF / (H - DLINE(I)) - RF / H;
           RAD := RAD + D * (H / FOCAL_LENGTH);
           if VISIBLE( DLINE(I), HORI_ALT(0),
                       RAD,      HORI_RAD(0) ) then
             begin
               HORI_RAD(0) := RAD;
               HORI_ALT(0) := DLINE(I);
               if X_CENTER + RAD <= min( DIM(PIXEL), X_CENTER+HALF_IMAGE_SIZE ) then
                 INSERT_PIXEL( IMAGE, Y_CENTER, X_CENTER+RAD, ILINE(I) );
               else
                 MLINE := MLINE + (. (MAX_IMAGE_SIZE-I) .);
             end  (* if RAD > HORI_RAD(0) and ... *);
           else
             MLINE := MLINE + (. (MAX_IMAGE_SIZE-I) .);

         end  (* for I *);



     end   (*  DISPLCE_CENTER_LINE  *);



(* ################################################################### *)
(*                                                                     *)
(*   Displace the pixels of ILINE utilizing the linear expanding       *)
(*   horizon, HORI_RAD and HORI_ALT, to calculate terrain visibility.  *)
(*   This horizon must be expanded prior to these calculations and this *)
(*   expansion is accomplished by EXPNDNG_HORIZON_UPDATE. The image    *)
(*   pixel's XY (cartesian) coordinates are first transformed to polar *)
(*   coordinates and then the displacement is introduced. The new polar *)
(*   coordinates are then converted to XY coordinates and the displaced *)
(*   pixel is inserted in the perspective image at these new coordinates. *)
(*                                                                     *)
(* ################################################################### *)
   procedure EXPANDING_DISPLACE( Y_COORD,
                       X_CENTER, Y_CENTER : integer;
                     var IMAGE            : INT_ARRAY;
                     var ILINE, DLINE     : INT_LINE;
                     var MLINE            : MAPSET;
                     var HORI_RAD, HORI_ALT : HORZN_LINE );

     var
       I, PREV_Y, PREV_START,
       PREV_STOP               : integer;
```

```
            H1, R1, RF, D           : short;
            POLAR                   : POLAR_COORD;
            XY                      : XY_COORD;


        begin

            if Y_COORD > 0 then
               PREV_Y := Y_COORD - 1;
            else
               PREV_Y := Y_COORD + 1;

            PREV_START := max( -abs( PREV_Y ), -X_CENTER+1          );
            PREV_STOP  := min( +abs( PREV_Y ), DIM(PIXEL)-X_CENTER );

            EXPNDNG_HORIZON_UPDATE( Y_COORD, X_CENTER, PREV_START, PREV_STOP,
                                    PREV_Y,  HORI_RAD, HORI_ALT                  );

            for I := PREV_START to PREV_STOP do
               begin

                  POLAR := POLAR_TRANSFORM( I, Y_COORD );
                  H1    := DLINE(I+X_CENTER);
                  RF    := POLAR(RDS) * FOCAL_LENGTH;
                  D     := RF / (H - H1) - RF / H;
                  R1    := POLAR(RDS) + D * (H / FOCAL_LENGTH);
                  if not VISIBLE( H1, HORI_ALT(I+X_CENTER),
                                  R1, HORI_RAD(I+X_CENTER) ) then
                     MLINE := MLINE + (. (MAX_IMAGE_SIZE-(I+X_CENTER)) .);
                  else
                     begin

                        XY    := XY_TRANSFORM( R1, POLAR(THT) );
                        XY(X) := XY(X) + X_CENTER;
                        XY(Y) := XY(Y) + Y_CENTER;
                        if (XY(X) > max( 0, X_CENTER-HALF_IMAGE_SIZE ))         and
                           (XY(Y) > max( 0, Y_CENTER-HALF_IMAGE_SIZE ))         and
                           (XY(X) <= min( DIM(PIXEL), X_CENTER+HALF_IMAGE_SIZE )) and
                           (XY(Y) <= min( DIM(LINE ), Y_CENTER+HALF_IMAGE_SIZE )) then
                           begin
                              INSERT_PIXEL( IMAGE, XY(Y), XY(X), ILINE(I+X_CENTER) );
                              HORI_ALT(I+X_CENTER) := H1;
                              HORI_RAD(I+X_CENTER) := R1;
                           end  (* if (XY(X) > 0) ... *);
                        else
                           MLINE := MLINE + (. (MAX_IMAGE_SIZE-(I+X_CENTER)) .);

                     end  (* else VISIBLE *);

               end  (* for I *);


        end   (*  EXPANDING_DISPLACE  *);



    (* ################################################################## *)
    (*                                                                    *)
```

```
(*   Update the linear expanding horizon, HORI_RAD and HORI_ALT, by        *)
(*   projecting it from a horizon of N elements to a horizon of (N+2)      *)
(*   elements.                                                             *)
(*                                                                         *)
(* ####################################################################### *)
     procedure EXPNDNG_HORIZON_UPDATE( Y, X_CENTER,
                                       PREV_START, PREV_STOP,
                                       PREV_Y                 : integer;
                                   var HORI_RAD, HORI_ALT     : HORZN_LINE );
        var
           LO, HI, X, START, STOP : integer;
           LO_FACT, HI_FACT,
           LO_ALT,  HI_ALT,
           LO_RAD,  HI_RAD, NEW_X : short;
           NEW_HORI_ALT,
           NEW_HORI_RAD, FACTOR    : HORZN_LINE;
        begin
           START := max( -abs( Y ), -X_CENTER+1          );
           STOP  := min( +abs( Y ), DIM(PIXEL)-X_CENTER );
           for X := START+X_CENTER-1 to STOP+X_CENTER+1 do
              begin
                 NEW_HORI_ALT(X) := 0;
                 NEW_HORI_RAD(X) := 0;
                 FACTOR(X)       := 0;
              end  (* for X *);
           if PREV_Y ¬= 0 then
              begin
                 for X := PREV_START to PREV_STOP do
                    begin
                       if Y > 0 then
                          NEW_X    := X + X / PREV_Y;
                       else
                          NEW_X    := X - X / PREV_Y;
                       LO       := trunc( NEW_X );
                       if NEW_X < 0 then
                          HI    := LO - 1;
                       else
                          HI    := LO + 1;
                       LO_FACT := abs( HI - NEW_X );
                       HI_FACT := 1 - LO_FACT;
                       if LO_FACT ¬= 0 then
                          begin
                             LO       := LO + X_CENTER;
                             LO_ALT   := NEW_HORI_ALT(LO) * FACTOR(LO) +
                                         HORI_ALT(X+X_CENTER) * LO_FACT;
                             LO_RAD   := NEW_HORI_RAD(LO) * FACTOR(LO) +
                                         HORI_RAD(X+X_CENTER) * LO_FACT;
                             FACTOR(LO)        := FACTOR(LO) + LO_FACT;
```

```
                           NEW_HORI_ALT(LO) := LO_ALT / FACTOR(LO);
                           NEW_HORI_RAD(LO) := LO_RAD / FACTOR(LO);

                        end  (* if LO_FACT ¬= 0 *);

                     if HI_FACT ¬= 0 then
                        begin

                           HI      := HI + X_CENTER;
                           HI_ALT  := NEW_HORI_ALT(HI) * FACTOR(HI) +
                                      HORI_ALT(X+X_CENTER) * HI_FACT;
                           HI_RAD  := NEW_HORI_RAD(HI) * FACTOR(HI) +
                                      HORI_RAD(X+X_CENTER) * HI_FACT;
                           FACTOR(HI)       := FACTOR(HI) + HI_FACT;
                           NEW_HORI_ALT(HI) := HI_ALT / FACTOR(HI);
                           NEW_HORI_RAD(HI) := HI_RAD / FACTOR(HI);

                        end  (* if HI_FACT ¬= 0 *);

                  end  (* for X *);

               for X := START+X_CENTER to STOP+X_CENTER do
                  if FACTOR(X) = 0 then
                     begin
                        NEW_HORI_ALT(X) := (NEW_HORI_ALT(X+1) - NEW_HORI_ALT(X-1)) /
                                           2.0s + NEW_HORI_ALT(X-1);
                        NEW_HORI_RAD(X) := (NEW_HORI_RAD(X+1) - NEW_HORI_RAD(X-1)) /
                                           2.0s + NEW_HORI_RAD(X-1);
                     end  (* for X *);

               HORI_ALT := NEW_HORI_ALT;
               HORI_RAD := NEW_HORI_RAD;

            end  (* if PREV_Y ¬= 0 *);


      end   (*  EXPNDNG_HORIZON_UPDATE  *);



(* ###################################################################### *)
(*                                                                        *)
(*   Set up and attach the files necessary for the system. Unit 0 is the  *)
(*   assumed input for the original DTM. If it is not assigned on the MTS *)
(*   RUN command, the file name will be prompted for. Unit 1 is assumed   *)
(*   input for the original registered image. Its file name will also be  *)
(*   prompted for if not assigned. The file assigned to unit 10 will be   *)
(*   used for the output of the left perspective image; unit 11 for the   *)
(*   output of the right perspective image. The file assigned to          *)
(*   unit 12 will be used for the output of the left visibility map of the *)
(*   DTM; unit 13 for the output of the right visibility map of the       *)
(*   DTM. If these units are not attached, they will default to MTS       *)
(*   temporary files -LEFT#, -RIGHT#, -LMAP#, and -RMAP# respectively.    *)
(*                                                                        *)
(* ###################################################################### *)
   procedure FILES;

      var
```

```
      DTMFNAME, IMGFNAME    : INPUT_LINE;


   begin

      DTMFNAME := ' ';
      DEFAULT_FNAME( 'DEFAULT 0=*DUMMY*;' );
      GET_FNAME( 'QUERY FDNAME 0;', 0, DTMFNAME );

      if DTMFNAME = '*DUMMY*' then
         begin

            PRINT_MSG( UNIT_0_UNASSIGNED, '', 0 );
            DTMFNAME := READ_STRING;
            while DTMFNAME = ' ' do
               begin
                  PRINT_MSG( INVALID_FILE, '', 0 );
                  DTMFNAME := READ_STRING;
                  if DTMFNAME = 'CANCEL' or DTMFNAME = 'HALT' then
                     halt;
               end  (* while DTMFNAME = ' ' *);

         end  (* if DTMFNAME = '*DUMMY*' *);


      IMGFNAME := ' ';
      DEFAULT_FNAME( 'DEFAULT 1=*DUMMY*;' );
      GET_FNAME( 'QUERY FDNAME 1;', 0, IMGFNAME );

      if IMGFNAME = '*DUMMY*' then
         begin

            PRINT_MSG( UNIT_1_UNASSIGNED, '', 0 );
            IMGFNAME := READ_STRING;
            while IMGFNAME = ' ' do
               begin
                  PRINT_MSG( INVALID_FILE, '', 0 );
                  IMGFNAME := READ_STRING;
                  if IMGFNAME = 'CANCEL' or IMGFNAME = 'HALT' then
                     halt;
               end  (* while IMGFNAME = ' ' *);

         end  (* if IMGFNAME = '*DUMMY*' *);


      DEFAULT_FNAME( 'DEFAULT 10=-LEFT#;'  );
      DEFAULT_FNAME( 'DEFAULT 11=-RIGHT#;' );
      DEFAULT_FNAME( 'DEFAULT 12=-LMAP#;'  );
      DEFAULT_FNAME( 'DEFAULT 13=-RMAP#;'  );

      reset  ( DTM_FILE,   DTMFNAME );
      reset  ( IMAGE_FILE, IMGFNAME );
      rewrite( LFILE,      10 );
      rewrite( RFILE,      11 );
      rewrite( LMFILE,     12 );
      rewrite( RMFILE,     13 );
```

```
         end   (*  FILES  *);



   (* ####################################################################### *)
   (*                                                                         *)
   (*   Retrieve the value of ARR(X,Y).                                       *)
   (*                                                                         *)
   (* ####################################################################### *)
      function GET_ARRAY( ARR : POINTER; X, Y : integer ) : short;

         var
            PTR               : POINTER;


         begin

            if Y > X then
               SYSTEM_ERROR( ' In "GET_ARRAY"; Subscript error' );
            PTR := POINTER( adrof( ARR@ ) + (BYTES_PER_WORD *
                            WORDS_PER_ENTRY * (X * (X + 1) / 2 + Y)) );
            GET_ARRAY := PTR@;


         end   (*  GET_ARRAY  *);



   (* ####################################################################### *)
   (*                                                                         *)
   (*   Prompt for and get the dimensions of the DTM.                         *)
   (*                                                                         *)
   (* ####################################################################### *)
      procedure GET_DIMENSIONS;

         begin

            PRINT_MSG( LINE_PROMPT,  ' ', 0 );
            DIM(LINE)  := trunc( GET_NUM( IMAGE_SIZE, INTEGRAL ) );

            PRINT_MSG( PIXEL_PROMPT, ' ', 0 );
            DIM(PIXEL) := trunc( GET_NUM( IMAGE_SIZE, INTEGRAL ) );

         end   (*  GET_DIMENSIONS  *);



   (* ####################################################################### *)
   (*                                                                         *)
   (*   Read and return the end of an input line. If COMPRESS is true then    *)
   (*   remove multiple embedded blanks, otherwise return the actual input    *)
   (*   line. The length of the input line is also returned.                  *)
   (*                                                                         *)
   (* ####################################################################### *)
       function GET_EOLN( var STR:INPUT_LINE; COMPRESS:boolean ) : integer;

          var
```

```
            I                  : integer;
            LAST_CHAR_BLANK    : boolean;
            CHR                : char;


         begin

            I := 0;
            LAST_CHAR_BLANK := false;

            while ¬eof( INPUT ) and ¬eoln( INPUT ) do
               begin
                  read( CHR );
                  if CHR ¬= ' ' or not COMPRESS then
                     begin
                        if LAST_CHAR_BLANK and I>0 then
                           begin
                              I := I + 1;
                              STR(I) := ' '
                           end;
                        I := I + 1;
                        STR(I) := CHR;
                        LAST_CHAR_BLANK := false;
                     end;
                  else
                     LAST_CHAR_BLANK := true;
               end;

            if not COMPRESS and I>0 then
               GET_EOLN := I - 1;
            else
               GET_EOLN := I


         end   (*  GET_EOLN  *);




(* ###################################################################### *)
(*                                                                        *)
(*   Return the numerical value of a string in the input stream. If       *)
(*   INTEGRAL_VALUE is true then the string must represent an integer,    *)
(*   otherwise a real number. The numerical value must also be in the     *)
(*   range specified by RANGE.                                            *)
(*                                                                        *)
(* ###################################################################### *)
   function GET_NUM( RANGE : VALUE_RANGE;
                     INTEGRAL_VALUE : boolean ) : real;

      var
         STR         : INPUT_LINE;
         VALID       : boolean;
         VALU        : real;
         I, J, MULT  : integer;


      begin
```

```
      repeat
         VALID := true;
         read( STR );
         if eof( INPUT ) then          (* CHECK FOR EOF                 *)
            begin
               VALID := false;
               PRINT_MSG( EOF_NO, ' ', 0 )
            end;
         else
            begin
               VALU := 0.0;            (* GET INTEGRAL PART             *)
               I := 1;
               MULT := 1;
               while I <= MAX_LINE_LEN and STR(I) = ' ' do
                  I := I + 1;
               if STR(I) = '+' or STR(I) = '-' then
                  begin
                     if STR(I) = '-' then
                        MULT := -1;
                     I := I + 1
                  end;
               while STR(I) in (. '0'..'9' .) do
                  begin
                     VALU := 10.0 * VALU + ord( STR(I) ) - ord('0');
                     I := I + 1
                  end;
               if STR(I) = '.' then
                  begin                (* GET FRACTIONAL PART           *)
                     J := 10;
                     I := I + 1;
                     while STR(I) in (. '0'..'9' .) do
                        begin
                           VALU := VALU + ( (ord(STR(I)) - ord('0'))/J);
                           I := I + 1;
                           J := J * 10
                        end;
                  end;
               if STR(I) ¬= ' ' or ( eoln(INPUT) and I = MAX_LINE_LEN+1 ) then
                  begin                (* INVALID NUMBER                *)
                     VALID := false;
                     if STR(I) ¬= ' ' then
                        PRINT_MSG( INVALID_NO, STR(I), 1  )
                     else
                        if RANGE in (. PERCENT .) then
                           begin
                              VALID := true;
                              MULT  := 1;
                              VALU  := DEFAULT;
                           end (* if RANGE in *);
                        else
                           PRINT_MSG( NULL_NO, ' ', 0 )
                  end
               else if INTEGRAL_VALUE and trunc(VALU) ¬= VALU then
                  begin
                     VALID := false;
                     PRINT_MSG( NON_INTEGRAL_VALUE, '', 0 )
                  end
               else if STR(I) = ' ' and ( STR(I-1)='-' or STR(I-1)='+' ) then
```

```
                              begin
                                 VALID := false;
                                 PRINT_MSG( UNEXPECTED_BLANK, '', 0 )
                              end;
                           else if (RANGE = POSITIVE and (MULT < 0 or VALU = 0.0)) or
                                   (RANGE = IMAGE_SIZE and (MULT < 0  or
                                            VALU < 1 or VALU > MAX_IMAGE_SIZE))    or
                                   (RANGE = LINES and (MULT < 0        or
                                            VALU < 1 or VALU > DIM(LINE)))         or
                                   (RANGE = LINE_LENGTH and (MULT < 0 or
                                            VALU < 1 or VALU > DIM(PIXEL)))        or
                                   (RANGE = INTENSITY and (MULT < 0    or
                                            VALU < MIN_INTENSITY       or
                                            VALU > MAX_INTENSITY))                 or
                                   (RANGE = NON_NEGATIVE and MULT < 0)             or
                                   (RANGE = PERCENT and (MULT < 0      or
                                            VALU < 0 or VALU > 100))               or
                                   (RANGE = NON_ZERO and VALU = 0.0)               then
                              begin
                                 VALID := false;
                                 PRINT_MSG( NOT_IN_RANGE, '', 0 )
                              end;
                        end;
                     until VALID;
                  GET_NUM := VALU * MULT


               end (* GET_NUM *);



       (* ################################################################## *)
       (*                                                                    *)
       (*   Update the circular horizon, HORI_RAD and HORI_ALT, by updating the   *)
       (*   information in the horizon between the CURRent THeTa and the     *)
       (*   PREVious THeTa.                                                  *)
       (*                                                                    *)
       (* ################################################################## *)
          procedure HORIZON_UPDATE( CURR_POLAR, PREV_POLAR : POLAR_COORD;
                                    CURR_ALT,   PREV_ALT   : int;
                                var HORI_RAD,   HORI_ALT   : HORZN_LINE  );

             var
                START_ALT,
                STOP_ALT             : int;
                I, START, STOP       : integer;
                ALT, RAD, RF, D      : short;
                START_POLAR,
                STOP_POLAR           : POLAR_COORD;


             begin

                if CURR_POLAR(THT) > 270  and  PREV_POLAR(THT) = 0  then
                   PREV_POLAR(THT) := 360;

                if CURR_POLAR(THT) < PREV_POLAR(THT) then
                   begin
```

```

                  START := trunc( CURR_POLAR(THT) );
                  STOP  := trunc( PREV_POLAR(THT) );
                  if START <> CURR_POLAR(THT) then
                     incr( START );
                  START_POLAR := CURR_POLAR;
                  STOP_POLAR  := PREV_POLAR;
                  START_ALT   := CURR_ALT;
                  STOP_ALT    := PREV_ALT;

               end  (* if CURR_POLAR(THT) < PREV_POLAR(THT) *);
            else
               begin

                  START := trunc( PREV_POLAR(THT) );
                  STOP  := trunc( CURR_POLAR(THT) );
                  if STOP <> CURR_POLAR(THT) then
                     decr( STOP );
                  START_POLAR := PREV_POLAR;
                  STOP_POLAR  := CURR_POLAR;
                  START_ALT   := PREV_ALT;
                  STOP_ALT    := CURR_ALT;

               end  (* else CURR_POLAR(THT) >= PREV_POLAR(THT) *);

            for I := START to STOP do
               begin

                  ALT := (START_ALT * (STOP_POLAR(THT) - I) +
                          STOP_ALT * (I - START_POLAR(THT))  ) /
                         (STOP_POLAR(THT) - START_POLAR(THT));

                  RAD := (START_POLAR(RDS) * (STOP_POLAR(THT) - I) +
                          STOP_POLAR(RDS) * (I - START_POLAR(THT))  ) /
                         (STOP_POLAR(THT) - START_POLAR(THT));

                  RF  := RAD * FOCAL_LENGTH;
                  D   := RF / (H - ALT) - RF / H;
                  RAD := RAD + D * (H / FOCAL_LENGTH);

                  if VISIBLE( ALT, HORI_ALT(I), RAD, HORI_RAD(I) ) then
                     begin
                        HORI_RAD(I) := RAD;
                        HORI_ALT(I) := ALT;
                     end  (* if RAD > HORI_RAD(I) and ... *);

               end  (* for I *);


         end   (*  HORIZON_UPDATE  *);



   (* ####################################################################### *)
   (*                                                                          *)
   (*   Insert the pixel with intensity INTEN into the IMAGE at position       *)
   (*   (X,Y).                                                                 *)
   (*                                                                          *)
```

```
 (* ##################################################################### *)
      procedure INSERT_PIXEL( var IMAGE : INT_ARRAY; Y, X : short;
                                    INTEN : integer;                );

         var
            X_INT, Y_INT          : integer;


         (*  Perform the actual insertion.  *)
         procedure PERFORM_INSERT( Y, X, I : integer );

            begin
               if IMAGE(Y,X) = UNDEFINED then
                  IMAGE(Y,X) := I;
               else
                  IMAGE(Y,X) := round( (I + IMAGE(Y,X)) / 2 );
            end   (*  PERFORM_INSERT  *);


         begin

            X_INT := trunc( X );
            Y_INT := trunc( Y );
            PERFORM_INSERT( Y_INT, X_INT, INTEN );

            if Y = Y_INT then
               begin
                  if X <> X_INT then
                     PERFORM_INSERT( Y_INT, X_INT+1, INTEN );
               end;
            else
               if X = X_INT then
                  PERFORM_INSERT( Y_INT+1, X_INT, INTEN );
               else
                  begin
                     PERFORM_INSERT( Y_INT,   X_INT+1, INTEN );
                     PERFORM_INSERT( Y_INT+1, X_INT,   INTEN );
                     PERFORM_INSERT( Y_INT+1, X_INT+1, INTEN );
                  end;


         end   (*  INSERT_PIXEL  *);



 (* ##################################################################### *)
 (*                                                                       *)
 (*   For undefined pixels of the image array IMAGE, perform a cubic      *)
 (*   convolution to define them.                                         *)
 (*                                                                       *)
 (* ##################################################################### *)
      procedure INTERPOLATE( var IMAGE : INT_ARRAY );

         var
            X_CENT, Y_CENT,
            X, Y, X_MAX, Y_MAX      : integer;
            DEFINED_FOUND           : boolean;
```

```

       (*  Perform a cubic convolution if undefined  *)
       (*  and in the interior of the image.          *)
       procedure PERFORM_INTER( X, Y : integer );

          begin

             if IMAGE(X,Y) <> UNDEFINED then
                DEFINED_FOUND := true;
             else
                if DEFINED_FOUND then
                   CUBIC_CONVOLUTION( IMAGE, X, Y );
                else
                   IMAGE(X,Y) := MIN_INTENSITY;

          end   (*  PERFORM_INTER  *);



       begin

          X_MAX  := DIM(LINE );
          Y_MAX  := DIM(PIXEL);
          X_CENT := trunc( (X_MAX+1) / 2 );
          Y_CENT := trunc( (Y_MAX+1) / 2 );

          for X := 1 to X_CENT do
             begin

                DEFINED_FOUND := false;
                for Y := 1 to min( X, Y_CENT ) do
                   PERFORM_INTER( X, Y );

                DEFINED_FOUND := false;
                for Y := Y_MAX downto max( Y_MAX-X+1, Y_CENT ) do
                    PERFORM_INTER( X, Y );

             end  (* for X *);

          for X := X_CENT+1 to X_MAX do
             begin

                DEFINED_FOUND := false;
                for Y := 1 to min( X_MAX-X+1, Y_CENT ) do
                   PERFORM_INTER( X, Y );

                DEFINED_FOUND := false;
                for Y := Y_MAX downto max( Y_MAX-X_MAX+X, Y_CENT ) do
                    PERFORM_INTER( X, Y );

             end  (* for X *);

          for Y := 1 to Y_CENT do
             begin

                DEFINED_FOUND := false;
                for X := 1 to min( Y, X_CENT ) do
                   PERFORM_INTER( X, Y );
```

```

                    DEFINED_FOUND := false;
                    for X := X_MAX downto max( X_MAX-Y+1, X_CENT ) do
                        PERFORM_INTER( X, Y );

                 end  (* for Y *);

              for Y := Y_CENT+1 to Y_MAX do
                 begin

                    DEFINED_FOUND := false;
                    for X := 1 to min( Y_MAX-Y+1, X_CENT ) do
                       PERFORM_INTER( X, Y );

                    DEFINED_FOUND := false;
                    for X := X_MAX downto max( X_MAX-Y_MAX+Y, X_CENT ) do
                        PERFORM_INTER( X, Y );

                 end  (* for Y *);



           end   (*  INTERPOLATE  *);



    (* ####################################################################### *)
    (*                                                                         *)
    (*   Calculate which octant of a circle the cartesian coordinates (X,Y)    *)
    (*   reside in. The first octant is where X>0, Y>0, and X>=Y, assuming     *)
    (*   the principle point to be the origin. Move counterclockwise to        *)
    (*   the second octant.                                                    *)
    (*                                                                         *)
    (* ####################################################################### *)
         procedure OCT_1_TRANSFORM( var X, Y, OCT : integer );

            var
               X_TEMP              : integer;


            begin

               if X > 0 then
                  if Y >= 0 then
                     if abs( Y ) > abs( X ) then
                        OCT := OCT_2;
                     else   (* abs( Y ) <= abs( X ) *)
                        OCT := OCT_1;
                  else      (* Y < 0               *)
                     if abs( Y ) >= abs( X ) then
                        OCT := OCT_7;
                     else   (* abs( Y ) < abs( X )  *)
                        OCT := OCT_8;

               else if X < 0 then
                  if Y > 0 then
                     if abs( Y ) >= abs( X ) then
                        OCT := OCT_3;
```

```
                  else    (* abs( Y ) < abs( X )  *)
                     OCT := OCT_4;
               else if Y < 0 then
                  if abs( Y ) > abs( X ) then
                     OCT := OCT_6;
                  else    (* abs( Y ) <= abs( X ) *)
                     OCT := OCT_5;
               else       (* Y = 0                *)
                     OCT := OCT_5;

            else          (* X = 0                *)
               if Y > 0 then
                  OCT := OCT_3;
               else if Y < 0 then
                  OCT := OCT_7;
               else       (* Y = 0                *)
                  OCT := OCT_1;


            case  OCT of

                  OCT_1, OCT_4, OCT_5, OCT_8     :
                     begin
                        X := abs( X );
                        Y := abs( Y );
                     end  (* OCT_1, ... *);

                  OCT_2, OCT_3, OCT_6, OCT_7     :
                     begin
                        X_TEMP := X;
                        X := abs( Y       );
                        Y := abs( X_TEMP );
                     end  (* OCT_2, ... *);

               end  (* case  OCT of *);


         end   (*  OCT_1_TRANSFORM  *);



   (* ################################################################### *)
   (*                                                                     *)
   (*   Initialize the lookup tables to transform cartesian coordinates   *)
   (*   into polar coordinates.                                           *)
   (*                                                                     *)
   (* ################################################################### *)
      procedure POLAR_INIT( SIZE : integer );

         var
            X, Y              : integer;


         begin

            RADIUS_LOOKUP := ALLOCATE_SPACE( round( SIZE * (SIZE + 1) / 2 ) );
            THETA_LOOKUP  := ALLOCATE_SPACE( round( SIZE * (SIZE + 1) / 2 ) );

```

```
            for X := 0 to SIZE-1 do
               for Y := 0 to X do
                  begin

                     PUT_ARRAY(  RADIUS_LOOKUP, X, Y, roundtoshort( sqrt( sqr( X ) + sqr( Y ) ) )  );
                     if X = 0 then
                        PUT_ARRAY( THETA_LOOKUP, X, Y, UNDEFINED );
                     else
                        PUT_ARRAY( THETA_LOOKUP, X, Y, short( arctan( Y/X ) ) );

                  end  (* for Y *);


         end   (*  POLAR_INIT  *);



   (* ################################################################### *)
   (*                                                                     *)
   (*   Via lookup tables, transform the cartesian coordinates (X,Y) into *)
   (*   polar coordinates. Return the polar form.                         *)
   (*                                                                     *)
   (* ################################################################### *)
      function POLAR_TRANSFORM( X, Y : integer ) : POLAR_COORD;

         var
            OCT               : integer;
            OCT_1_THETA       : short;
            POLAR             : POLAR_COORD;


         begin

            OCT_1_TRANSFORM( X, Y, OCT );

            POLAR( RDS ) := GET_ARRAY( RADIUS_LOOKUP, X, Y );

            OCT_1_THETA  := roundtoshort( GET_ARRAY( THETA_LOOKUP,  X, Y ) / DEG_TO_RAD );
            case  OCT  of

                  OCT_1    :
                     POLAR( THT ) :=       OCT_1_THETA;
                  OCT_2    :
                     POLAR( THT ) :=  90 - OCT_1_THETA;
                  OCT_3    :
                     POLAR( THT ) :=  90 + OCT_1_THETA;
                  OCT_4    :
                     POLAR( THT ) := 180 - OCT_1_THETA;
                  OCT_5    :
                     POLAR( THT ) := 180 + OCT_1_THETA;
                  OCT_6    :
                     POLAR( THT ) := 270 - OCT_1_THETA;
                  OCT_7    :
                     POLAR( THT ) := 270 + OCT_1_THETA;
                  OCT_8    :
                     POLAR( THT ) := 360 - OCT_1_THETA;

               end  (* case OCT of *);
```

```
               POLAR_TRANSFORM := POLAR;


            end   (*  POLAR_TRANSFORM  *);



   (* ################################################################## *)
   (*                                                                    *)
   (*   Print the message specified by MSG to the output stream. The    *)
   (*   parameters STRING and SIZE may be involved in the details of    *)
   (*   that message.                                                    *)
   (*                                                                    *)
   (* ################################################################## *)
         procedure PRINT_MSG( MSG : MSG_TYPE; STRING : INPUT_LINE;
                              SIZE : integer );

            const
               MSG_PROMPT = ':';

            var
               I,J            : integer;
               STR            : INPUT_LINE;


            begin

               SET_PREFIX( MSG_PROMPT, 1 );

               if MSG in (. LINE_PROMPT, UNIT_0_UNASSIGNED, INVALID_FILE,
                            ENTER_CENTER_X_COORD, ENTER_GRID_SPACING,
                            ENTER_PRINT_SIZE,
                            UNIT_1_UNASSIGNED, ENTER_ELEVATION .) then
                  writeln;

               case MSG of
                  UNEXPECTED_EOF   :
                        writeln(' Unexpected EOF -- Ignored');
                  UNREC_CMD        :
                        begin
                           writeln(' Invalid command -- Input line ignored');
                           if ¬eoln( INPUT ) then
                              I := GET_EOLN( STR, COMPRESSED )
                        end;
                  EOLN_MSG         :
                        begin
                           I := GET_EOLN( STR, COMPRESSED );
                           if I ¬= 0 then
                              begin
                                 write(' Invalid character(s): "');
                                 WRITE_STRING( STR, I );
                                 writeln('" -- Ignored')
                              end
                        end;
                  LINE_PROMPT      :
                        writeln('&Enter number of lines of DTM and IMAGE (Integer (1-256)):');
                  PIXEL_PROMPT     :
```

```
                    writeln('&Enter number of pixels per line        (Integer {1-256}):');
              EOF_NO             :
                    writeln('&Unexpected EOF, Enter number:');
              NULL_NO            :
                    writeln( '&Not expecting null line -- Ignored; Enter number:' );
              NON_INTEGRAL_VALUE:
                    writeln('&Non-integral value, Re-enter number:');
              UNEXPECTED_BLANK:
                    writeln('&Unexpected blank, Re-enter number:');
              NOT_IN_RANGE       :
                    writeln('&Out of range, Re-enter number:');
              UNIT_0_UNASSIGNED:
                    writeln(' Logical unit 0 has not been assigned.',
                              EOL, '&   Enter FDname of location of DTM   :');
              UNIT_1_UNASSIGNED:
                    writeln(' Logical unit 1 has not been assigned.',
                              EOL, '&   Enter FDname of location of IMAGE :');
              INVALID_FILE       :
                    writeln(' Invalid FDname. Enter again or', EOL,
                            '&Enter "CANCEL" to terminate program:');
              UNEXPECTED_INPUT_EOF:
                    writeln(' Unexpected EOF encountered on input',
                            ' -- DTM size updated');
              INVALID_NO         :
                    begin
                       writeln('&Invalid character, "', STRING(1), '", Re-enter number:');
                       I := GET_EOLN( STR, COMPRESSED )
                    end;
              ENTER_CENTER_X_COORD:
                    begin
                       write  (' Enter the coordinates of the',
                               ' principal', EOL, ' point of the ');
                       WRITE_STRING( STRING, SIZE );
                       writeln(' image:', EOL,
                               '&     X (pixel - INTEGER {1-', DIM(PIXEL):0, '}):');
                    end;
              ENTER_CENTER_Y_COORD:
                    writeln('&     Y (line  - INTEGER {1-', DIM(LINE ):0, '}):');
              ENTER_GRID_SPACING:
                    writeln('&Enter DTM grid spacing                 (in DTM units):');
              ENTER_ELEVATION :
                    writeln('&Enter elevation of imaging device     (in DTM units):');
              ENTER_FOCAL_LENGTH:
                    writeln('&Enter focal length of imaging device (in DTM units):');
              ENTER_PRINT_SIZE:
                    writeln('&Enter print size of imaging device   (in DTM units):');
              PRINT_PRINT_SIZE:
                    writeln( ' Print size, in DTM pixels, is approximately ',
                              SIZE:0,' X ',SIZE:0,' pixels.' );

           end  (*  case  *);

           if MSG in (. LINE_PROMPT, PIXEL_PROMPT, INVALID_NO,
                        NON_INTEGRAL_VALUE, UNEXPECTED_BLANK, NOT_IN_RANGE,
                        UNIT_0_UNASSIGNED, UNIT_1_UNASSIGNED, INVALID_FILE,
                        ENTER_ELEVATION, ENTER_CENTER_X_COORD,
                        ENTER_GRID_SPACING, ENTER_FOCAL_LENGTH,
                        ENTER_CENTER_Y_COORD, ENTER_PRINT_SIZE,
```

```
                        EOF_NO, NULL_NO .) then
                 readln;

               SET_PREFIX( PROMPT, 1 )


           end   (*  PRINT_MSG  *);



    (* ###################################################################### *)
    (*                                                                        *)
    (*   Insert the value VALU into ARR(X,Y).                                 *)
    (*                                                                        *)
    (* ###################################################################### *)
        procedure PUT_ARRAY( ARR : POINTER; X, Y : integer; VALU : short );

           var
              PTR               : POINTER;


           begin

              if Y > X then
                 SYSTEM_ERROR( ' In "PUT_ARRAY"; Subscript error' );
              PTR := POINTER( adrof( ARR@ ) + (BYTES_PER_WORD *
                               WORDS_PER_ENTRY * (X * (X + 1) / 2 + Y)) );
              PTR@ := VALU;


           end   (*  PUT_ARRAY  *);



    (* ###################################################################### *)
    (*                                                                        *)
    (*   If READ_REQD is true then read in a line from the file DTM_FILE,     *)
    (*   otherwise perform the subsequent operations on the present input     *)
    (*   buffer. Convert the binary form of the buffer into integer form      *)
    (*   assuming two bytes per pixel.                                        *)
    (*                                                                        *)
    (* ###################################################################### *)
        function READ_DTM_LINE( var DTM_FILE : DTMFILE; READ_REQD : boolean ) : INT_LINE;

           var
              LINE                 : INT_LINE;
              I                    : integer;


           begin

              if READ_REQD then
                 get( DTM_FILE );
              for I := 1 to DIM(PIXEL) do
                 LINE(I) := int( DTM_FILE@((I-1)*2 + 1) ) * 256 +
                            int( DTM_FILE@((I-1)*2 + 2) );
              READ_DTM_LINE := LINE;

```

```
          end   (*  READ_DTM_LINE  *);




(* ##################################################################### *)
(*                                                                       *)
(*   If READ_REQD is true then read in a line from the file IMAGE_FILE,  *)
(*   otherwise perform the subsequent operations on the present input    *)
(*   buffer. Convert the binary form of the buffer into integer form     *)
(*   assuming one byte per pixel.                                        *)
(*                                                                       *)
(* ##################################################################### *)
     function READ_IMAGE_LINE( var IMAGE_FILE : IMAGEFILE; READ_REQD : boolean ) : INT_LINE;

          var
             LINE                : INT_LINE;
             I                   : integer;


          begin

             if READ_REQD then
                get( IMAGE_FILE );
             for I := 1 to DIM(PIXEL) do
                LINE(I) := int( IMAGE_FILE@(I) );
             READ_IMAGE_LINE := LINE;


          end   (*  READ_IMAGE_LINE  *);



(* ##################################################################### *)
(*                                                                       *)
(*   Read the LINE_NOth line of either the DTM_FILE or the IMAGE_FILE    *)
(*   depending on the value of TYP. The reads are performed by either    *)
(*   READ_DTM_LINE or READ_IMAGE_LINE. The result is returned in INT_LINE. *)
(*                                                                       *)
(* ##################################################################### *)
     function READ_LINE( TYP, LINE_NO : integer; RESET_REQD : boolean )
                       : INT_LINE;

          var
             I          : integer;


          begin

             case TYP of

                DTM        :
                   begin
                      if RESET_REQD then
                         begin
                            reset( DTM_FILE );
                            for I := 1 to LINE_NO-1 do
```

```
                             get( DTM_FILE );
                          READ_LINE := READ_DTM_LINE( DTM_FILE,
                                                   not READ_REQD );
                       end  (* if RESET_REQD *);
                    else
                       READ_LINE := READ_DTM_LINE( DTM_FILE,
                                                   READ_REQD );
                 end  (* DTM *);


              IMGE      :
                 begin
                    if RESET_REQD then
                       begin
                          reset( IMAGE_FILE );
                          for I := 1 to LINE_NO-1 do
                             get( IMAGE_FILE );
                          READ_LINE := READ_IMAGE_LINE( IMAGE_FILE,
                                                     not READ_REQD );
                       end  (* if RESET_REQD *);
                    else
                       READ_LINE := READ_IMAGE_LINE( IMAGE_FILE,
                                                     READ_REQD );
                 end  (* IMGE *);

           end  (* case TYP of *);


        end   (*  READ_LINE  *);



(* ##################################################################### *)
(*                                                                       *)
(*   Read a string from the input stream removing all leading blanks.   *)
(*   The string is then returned.                                       *)
(*                                                                       *)
(* ##################################################################### *)
     function READ_STRING : INPUT_LINE;

        var
           I                   : integer;
           STR                 : INPUT_LINE;


        begin

           STR := ' ';
           I := 0;
           repeat
              incr( I );
              read( STR(I) );
              if STR(I) = ' ' and I = 1 then
                 I := 0;
           until eof(INPUT) or eoln(INPUT) or (I ¬= 0 and STR(I) = ' ')
                                           or I = MAX_LINE_LEN;

           READ_STRING := STR;
```

```


          end   (*  READ_STRING  *);




     (* ############################################################### *)
     (*                                                                  *)
     (*   Initialize the system by setting up the files, getting the dimensions *)
     (*   of the DTM, and reading the necessary system parameters.      *)
     (*                                                                  *)
     (* ############################################################### *)
          procedure SYS_INIT;

             var
                MAX_LEFT, MAX_RIGHT : int;
                I, J                : integer;


             begin

                SET_PREFIX( PROMPT, 1 );
                FILES;
                GET_DIMENSIONS;

                PRINT_MSG( ENTER_GRID_SPACING, '', 0 );
                GRID_SPAC        := roundtoshort( GET_NUM( POSITIVE, NOT INTEGRAL ) );

                PRINT_MSG( ENTER_ELEVATION, '', 0 );
                H                := roundtoshort( GET_NUM( POSITIVE, NOT INTEGRAL ) );

                PRINT_MSG( ENTER_FOCAL_LENGTH, '', 0 );
                FOCAL_LENGTH     := roundtoshort( GET_NUM( NON_NEGATIVE, NOT INTEGRAL ) );

                PRINT_MSG( ENTER_PRINT_SIZE, '', 0 );
                PRINT_SIZE       := roundtoshort( GET_NUM( POSITIVE, NOT INTEGRAL ) );
                FULL_IMAGE_SIZE  :=
                     round( PRINT_SIZE * H / (FOCAL_LENGTH * GRID_SPAC) );
                HALF_IMAGE_SIZE  :=
                     round( PRINT_SIZE * H / (2 * FOCAL_LENGTH * GRID_SPAC) );
                PRINT_MSG( PRINT_PRINT_SIZE, '', FULL_IMAGE_SIZE );


                PRINT_MSG( ENTER_CENTER_X_COORD, 'left ', 5 );
                CENTER(LEFT ,X)  := trunc( GET_NUM( LINE_LENGTH, INTEGRAL ) );
                PRINT_MSG( ENTER_CENTER_Y_COORD, '', 0 );
                CENTER(LEFT ,Y)  := trunc( GET_NUM( LINES,       INTEGRAL ) );

                PRINT_MSG( ENTER_CENTER_X_COORD, 'right', 5 );
                CENTER(RIGHT,X)  := trunc( GET_NUM( LINE_LENGTH, INTEGRAL ) );
                PRINT_MSG( ENTER_CENTER_Y_COORD, '', 0 );
                CENTER(RIGHT,Y)  := trunc( GET_NUM( LINES,       INTEGRAL ) );


                MAX_LEFT  := max( CENTER(LEFT ,X), max( CENTER(LEFT ,Y),
                                  max( DIM(PIXEL) - CENTER(LEFT ,X),
                                       DIM(LINE ) - CENTER(LEFT ,Y) ) ) );
```

```
              MAX_RIGHT := max( CENTER(RIGHT,X), max( CENTER(RIGHT,Y),
                               max( DIM(PIXEL) - CENTER(RIGHT,X),
                                    DIM(LINE ) - CENTER(RIGHT,Y) ) ) );
              MAX_HALF  := max( MAX_LEFT, MAX_RIGHT ) + 1;

              POLAR_INIT( MAX_HALF );
              XY_INIT;


         end   (*  SYSTEM_INITIALIZATION  *);



(* ##################################################################### *)
(*                                                                       *)
(*   If a system error occurs, output the message MESSAGE and halt the   *)
(*   execution of the system.                                            *)
(*                                                                       *)
(* ##################################################################### *)
     procedure SYSTEM_ERROR( MESSAGE : STR_50 );

         begin

              writeln(' ===> SYSTEM ERROR ===> ',MESSAGE );
              halt;


         end   (*  SYSTEM_ERROR  *);



(* ##################################################################### *)
(*                                                                       *)
(*   Return true if terrain at radius RAD1 from the principle point (with *)
(*   elevation ALT1) has its visibility to the imaging system blocked by *)
(*   terrain at radius RAD2 from the principle point (with elevation     *)
(*   ALT2). Otherwise return false.                                      *)
(*                                                                       *)
(* ##################################################################### *)
     function VISIBLE( ALT1, ALT2, RAD1, RAD2 : short ) : boolean;



        begin

            if (RAD1 <= RAD2) or
               ((RAD2 / (H - ALT2)) >= (RAD1 / (H - ALT1))) then
               VISIBLE := false;
            else
               VISIBLE := true;


        end   (*  VISIBLE  *);



(* ##################################################################### *)
(*                                                                       *)
```

```
(*   Write the line LINE to the file IMAGE starting at pixel START.         *)
(*   It is assumed that each pixel has a value between 0 and 255.           *)
(*                                                                          *)
(* ######################################################################## *)
    procedure WRITE_IMAGE_LINE( var IMAGE : IMAGEFILE;
                          LINE : INT_LINE; START : integer );

       var
          I                  : integer;


       begin

          IMAGE@ := ' ';

          for I := 1 to (START-1) do
             IMAGE@(I) := char( MIN_INTENSITY );

          for I := START to DIM(PIXEL) do
             IMAGE@(I) := char( LINE(I) );

          for I := (DIM(PIXEL)+1) to MAX_IMAGE_SIZE do
             IMAGE@(I) := char( MIN_INTENSITY );

          put( IMAGE );


       end   (*  WRITE_IMAGE_LINE  *);



(* ######################################################################## *)
(*                                                                          *)
(*   Write the map line LINE to the file MAPFILE at MTS line                *)
(*   number LINE_NO.                                                        *)
(*                                                                          *)
(* ######################################################################## *)
     procedure WRITE_MAP_LINE( var MAP : MAPFILE; LINE : MAPSET;
                                           LINE_NO : integer );


        begin

           position( MAP, LINE_NO * 1000 );
           writeln ( MAP, LINE );


        end   (*  WRITE_MAP_LINE  *);



(* ######################################################################## *)
(*                                                                          *)
(*   Write the string STR to the present output stream. The string is of   *)
(*   length SIZE. If the length is less than one, the operation does not   *)
(*   take place.                                                            *)
(*                                                                          *)
(* ######################################################################## *)
```

```
         procedure WRITE_STRING( STR : INPUT_LINE; SIZE : integer );

            var
               I                 : integer;


            begin

               if SIZE > 0 then
                  for I := 1 to SIZE do
                     write( STR(I) )


            end   (*  WRITE_STRING  *);



    (* ###################################################################### *)
    (*                                                                        *)
    (*   Write the image array IMAGE to the image file IMAGE_FILE. The       *)
    (*   utility function WRITE_IMAGE_LINE is utilized to perform the writes. *)
    (*                                                                        *)
    (* ###################################################################### *)
         procedure WRT_IMAGE( var IMAGE_FILE : IMAGEFILE;
                              var IMAGE      : INT_ARRAY  );

            var
               I                  : integer;


            begin

               rewrite( IMAGE_FILE );
               for I := DIM(LINE) downto 1 do
                  WRITE_IMAGE_LINE( IMAGE_FILE, IMAGE(I), 1 );


            end   (*  WRT_IMAGE  *);



    (* ###################################################################### *)
    (*                                                                        *)
    (*   Initialize the lookup tables to convert from polar coordinates      *)
    (*   to cartesian coordinates.                                            *)
    (*                                                                        *)
    (* ###################################################################### *)
         procedure XY_INIT;

            var
               DEG                : int;


            begin

               for DEG := 0 to 360 do
                  begin
```

```
                    XY_LOOKUP(DEG, X) := roundtoshort( cos( DEG * DEG_TO_RAD ) );
                    XY_LOOKUP(DEG, Y) := roundtoshort( sin( DEG * DEG_TO_RAD ) );

                 end  (* for DEG *);


             end   (*  XY_INIT  *);



      (* ################################################################### *)
      (*                                                                     *)
      (*   Via lookup tables, transform the polar coordinates (RADIUS,THETA) *)
      (*   into cartesian coordinates. Return the cartesian form.            *)
      (*                                                                     *)
      (* ################################################################### *)
           function XY_TRANSFORM( RADIUS, THETA : short ) : XY_COORD;

             var
               LO_THETA, HI_THETA,
               COORD                 : int;
               XY                    : XY_COORD;


             begin

               LO_THETA := trunc( THETA );
               HI_THETA := LO_THETA  +  1;

               if LO_THETA < 0 or HI_THETA > 360 then
                 SYSTEM_ERROR('In "XY_TRANSFORM"; Illegal theta spec.');
               for COORD := X to Y do
                 XY(COORD) := ( RADIUS *
                                ( XY_LOOKUP(LO_THETA, COORD) *
                                ( HI_THETA - THETA )          +
                                  XY_LOOKUP(HI_THETA, COORD) *
                                ( THETA - LO_THETA )            ) );

               XY_TRANSFORM := XY;


             end   (*  XY_TRANSFORM  *);



      (* ################################################################### *)
      (*                                                                     *)
      (*   Initialize the system and then proceed to displace the original   *)
      (*   image as a function of its relative height (as specified by the DTM). *)
      (*                                                                     *)
      (* ################################################################### *)
          begin   (*  MAIN ROUTINE  *)

            SYS_INIT;

            DISPLACE( IMAGE, LMFILE, CENTER(LEFT ,X), CENTER(LEFT ,Y) );
            INTERPOLATE( IMAGE );
            WRT_IMAGE  ( LFILE, IMAGE );
```

```

            DISPLACE( IMAGE, RMFILE, CENTER(RIGHT,X), CENTER(RIGHT,Y) );
            INTERPOLATE( IMAGE );
            WRT_IMAGE  ( RFILE, IMAGE );

        end   (*  MAIN ROUTINE  *).
```

Appendix C

This appendix contains the source code that was used to produce synthetic airborne scanner imagery. This source is written is UBC PASCAL [JOLL79].

```
(* ######################################################################## *)
(*                                                                          *)
(*              A I R B O R N E   S C A N N E R   I M A G E                 *)
(*                                                                          *)
(*   With input of a DTM and a corresponding registered image, a            *)
(*   scanner image of the DTM, according to the intensity values of the     *)
(*   inputted image, is created as well as a visibility map for the         *)
(*   DTM (as seen from a hypothetical airborne scanner device) and a        *)
(*   pixel layover map (as sensed by a hypothetical radar imaging           *)
(*   device).                                                               *)
(*                                                                          *)
(* ######################################################################## *)

     const
        LINE  = 1;   PIXEL = 2;
        MAX_LINE_LEN         = 255;
        PROMPT               = ':';
        COMPRESSED           = true;
        INTEGRAL             = true;
        READ_REQD            = true;
        RIGHT_HALF           = true;
        LEFT_HALF            = false;
        MIN_INTENSITY        = 0;
        MAX_INTENSITY        = 255;
        MAX_IMAGE_SIZE       = 256;
        MIN_SCAN_ANGLE       = 0;
        MAX_SCAN_ANGLE       = 89;
        UNDEFINED            = -1;
        DEFAULT              = -1;
        DEG_TO_RAD           = 0.0174532925;
        OUT_OF_RANGE         = -(maxint-1);


     type
        MSG_TYPE    = ( LINE_PROMPT, PIXEL_PROMPT, UNEXPECTED_BLANK,
                        UNEXPECTED_EOF, UNREC_CMD, NULL_NO, NON_INTEGRAL_VALUE,
                        UNIT_0_UNASSIGNED, INVALID_FILE, UNEXPECTED_INPUT_EOF,
                        INVALID_NO, EOF_NO, EOLN_MSG, NOT_IN_RANGE,
                        ENTER_ELEVATION, ENTER_NADIR_COORD,
                        MAP_DESIRED, ASSUMING_NO, ENTER_GRID_SPACING,
                        ENTER_DEPRESSION_ANGLE, SIDELOOK_TRANSFORM,
                        UNIT_1_UNASSIGNED, ENTER_SCAN_ANGLE                  );

        INPUT_LINE  = array(. 1..MAX_LINE_LEN+1 .) of char;
        INT_LINE    = array(. 0..MAX_LINE_LEN+2 .) of int;
        REAL_LINE   = array(. 0..MAX_LINE_LEN+2 .) of real;
        SHORT_LINE  = array(. 0..MAX_LINE_LEN+2 .) of short;

        INT_ARRAY   = array(. 0..MAX_LINE_LEN+2 .) of INT_LINE;
        SHORT_ARRAY = array(. 0..MAX_LINE_LEN+2 .) of SHORT_LINE;


        VALUE_RANGE = ( POSITIVE, NON_ZERO, INTENSITY, LINE_LENGTH,
                        IMAGE_SIZE, NON_NEGATIVE, PERCENT, LINES,
                        SCAN_RANGE );
        PIXEL_RANGE = 1..MAX_IMAGE_SIZE;
        MAPSET      = set of PIXEL_RANGE;

```

```
            STR_3        = array (. 1..3  .) of char;
            STR_4        = array (. 1..4  .) of char;
            STR_6        = array (. 1..6  .) of char;
            STR_8        = array (. 1..8  .) of char;
            STR_44       = array (. 1..44 .) of char;
            STR_50       = array (. 1..50 .) of char;

            IMAGEFILE    = file of INPUT_LINE;
            DTMFILE      = file of array( 1..(MAX_LINE_LEN+1)*2 ) of char;
            MAPFILE      = file of MAPSET;




         var
            IMAGE_CENTER,
            NADIR_COORD,
            MAX_ANGLE            : integer;
            MAX_DEP_ANGLE,
            MIN_DEP_ANGLE,
            GRID_SPAC,
            SCAN_ANGLE,
            INTERVAL,
            ELEVATION            : short;
            SIDE_LOOK,
            VISIBILITY_MAP,
            PIXEL_LAYOVER_MAP    : boolean;
            DIM                  : array(. 1..2 .) of integer;
            TAN_LOOKUP           : array(. 0..MAX_SCAN_ANGLE .) of short;
            IMAGE_FILE, IFILE    : IMAGEFILE;
            DTM_FILE             : DTMFILE;
            VMFILE, PLMFILE      : MAPFILE;




      (* ##################################################################### *)
      (*                                                                       *)
      (*         FORWARD declarations for the Procedures of the system         *)
      (*                                                                       *)
      (* ##################################################################### *)
         procedure BUFFER_INSERT( INTENSITY   : integer;
                                  DEGREE      : short;
                              var ILINE       : INT_LINE;
                              var FLINE       : SHORT_LINE;
                                  RIGHT_HALF  : boolean     );          forward;
         procedure FILES;                                               forward;
         procedure GET_DIMENSIONS;                                      forward;
         procedure INTERPOLATE( var ILINE : INT_LINE;
                                var FACT  : SHORT_LINE );               forward;
         procedure PRINT_MSG( MSG  : MSG_TYPE; STRING : INPUT_LINE;
                              SIZE : integer                       ); forward;
         procedure SCANNER_TRANSFORM;                                   forward;
         procedure SIDE_LOOK_TRANSFORM;                                 forward;
         procedure SYS_INIT;                                            forward;
         procedure SYSTEM_ERROR( MESSAGE : STR_50 );                    forward;
         procedure TAN_INIT( SIZE : integer );                          forward;
         procedure WRITE_IMAGE_LINE( var IMAGE : IMAGEFILE;
```

```
                                           LINE  : INT_LINE;
                                           START : integer  );          forward;
      procedure WRITE_MAP_LINE( var MAP   : MAPFILE;
                                         LINE : MAPSET    );            forward;
      procedure WRITE_STRING( STR : INPUT_LINE; SIZE : integer );  forward;


  (* ################################################################## *)
  (*                                                                    *)
  (*          FORWARD declarations for the Functions of the system      *)
  (*                                                                    *)
  (* ################################################################## *)
      function ATAN( ARG : short ) : short;                          forward;
      function CALC_INTERVAL( MAX_ANGLE : integer ) : short;         forward;
      function CALC_PIXEL_LAYOVER( DLINE:INT_LINE; VMLINE:MAPSET )
                                            : MAPSET;                forward;
      function GET_EOLN( var STR : INPUT_LINE; COMPRESS : boolean)
                                            : integer;               forward;
      function GET_NUM( RANGE : VALUE_RANGE; INTEGRAL_VALUE : boolean )
                                            : real;                  forward;
      function GET_REPLY : boolean;                                  forward;
      function READ_DTM_LINE( var DTM_FILE : DTMFILE; READ_REQD : boolean )
                                            : INT_LINE;              forward;
      function READ_IMAGE_LINE( var IMAGE_FILE : IMAGEFILE;
                                   READ_REQD : boolean ) : INT_LINE;  forward;
      function READ_STRING : INPUT_LINE;                             forward;


  (* ################################################################## *)
  (*                                                                    *)
  (*          Declarations for FORTRAN routines used by the system      *)
  (*                                                                    *)
  (* ################################################################## *)
      procedure CALL_MTS( STR : INPUT_LINE; I : integer );  fortran 'CMDNOE';
      procedure DEFAULT_FNAME( CMD_STR : INPUT_LINE );      fortran 'FTNCMD';
      procedure GET_FNAME( CMD_STR : INPUT_LINE; LEN : integer;
                           var FNAME    : INPUT_LINE );     fortran 'FTNCMD';
      procedure SET_PREFIX( NEW : char; LENGTH : integer ); fortran 'SETPFX';





  (* ################################################################## *)
  (*                                                                    *)
  (*   Return the arctangent of argument ARG. This is accomplished by  *)
  (*   utilizing the tangent lookup table TAN_LOOKUP.                   *)
  (*                                                                    *)
  (* ################################################################## *)
      function ATAN( ARG : short ) : short;

         var
            LO_DEG,  HI_DEG, DEG : integer;
            LO_TAN,  HI_TAN,
            LO_FACT, HI_FACT     : short;


         begin
```

```
            if ARG < 0 then
               SYSTEM_ERROR( 'In "ATAN"; Invalid argument' );

            DEG := -1;
            repeat
               incr( DEG );
            until (DEG > MAX_ANGLE) or (TAN_LOOKUP(DEG) >= ARG)

            if DEG > MAX_ANGLE then
               ATAN := OUT_OF_RANGE;
            else
               if DEG = 0 then
                  ATAN := 0;
               else
                  begin

                     LO_DEG  := DEG - 1;
                     HI_DEG  := DEG;
                     LO_TAN  := TAN_LOOKUP(LO_DEG);
                     HI_TAN  := TAN_LOOKUP(HI_DEG);
                     LO_FACT := (HI_TAN - ARG) / (HI_TAN - LO_TAN);
                     HI_FACT := 1 - LO_FACT;
                     ATAN    := LO_FACT * LO_DEG + HI_FACT * HI_DEG;

                  end  (* else DEG <= MAX_ANGLE *);


         end   (*  ATAN  *);



   (* ####################################################################### *)
   (*                                                                         *)
   (*   Insert into the output buffer a pixel with value INTENSITY that is    *)
   (*   sensed by the hypothetical airborne scanner device at angle DEGREE    *)
   (*   of the scanner sweep.                                                 *)
   (*                                                                         *)
   (* ####################################################################### *)
      procedure BUFFER_INSERT( INTENSITY  : integer;
                               DEGREE     : short;
                           var ILINE      : INT_LINE;
                           var FLINE      : SHORT_LINE;
                               RIGHT_HALF : boolean    );

         var
            LO_INDEX, HI_INDEX : integer;
            LO_INTEN, HI_INTEN,
            LO_FACT,  HI_FACT  : short;


         begin

            LO_INDEX := trunc( DEGREE / INTERVAL );
            if (LO_INDEX * INTERVAL < 0) or (LO_INDEX * INTERVAL > MAX_ANGLE) then
               SYSTEM_ERROR( 'In "BUFFER_INSERT"; Invalid angle specified' );

            HI_INDEX := LO_INDEX + 1;
```

```
                LO_FACT  := HI_INDEX - DEGREE / INTERVAL;
                HI_FACT  := 1 - LO_FACT;

                if RIGHT_HALF then
                   begin
                      LO_INDEX := IMAGE_CENTER + LO_INDEX;
                      HI_INDEX := IMAGE_CENTER + HI_INDEX;
                   end  (* if RIGHT_HALF *);
                else    (* LEFT_HALF *)
                   begin
                      LO_INDEX := IMAGE_CENTER - LO_INDEX;
                      HI_INDEX := IMAGE_CENTER - HI_INDEX;
                   end  (* else LEFT_HALF *);

                if LO_FACT ¬= 0 then
                   begin

                      LO_INTEN        := ILINE(LO_INDEX) * FLINE(LO_INDEX) +
                                         INTENSITY * LO_FACT;
                      FLINE(LO_INDEX) := FLINE(LO_INDEX) + LO_FACT;
                      ILINE(LO_INDEX) := round( LO_INTEN / FLINE(LO_INDEX) );

                   end  (* if LO_FACT ¬= 0 *);

                if HI_FACT ¬= 0 then
                   begin

                      HI_INTEN        := ILINE(HI_INDEX) * FLINE(HI_INDEX) +
                                         INTENSITY * HI_FACT;
                      FLINE(HI_INDEX) := FLINE(HI_INDEX) + HI_FACT;
                      ILINE(HI_INDEX) := round( HI_INTEN / FLINE(HI_INDEX) );

                   end  (* if HI_FACT ¬= 0 *);


            end   (*  BUFFER_INSERT  *);



      (* ##################################################################### *)
      (*                                                                       *)
      (*    Based on the maximum scan angle of the hypothetical airborne scanner *)
      (*    device, calculate the "optimal" interval (in degrees) between       *)
      (*    pixels of the output buffer.                                       *)
      (*                                                                       *)
      (* ##################################################################### *)
            function CALC_INTERVAL( MAX_ANGLE : integer ) : short;

               const
                  ANGLE_000 = MIN_SCAN_ANGLE; (* Minimum angle permitted     *)
                  ANGLE_010 = 10;             (* Max angle for .10 intervals *)
                  ANGLE_025 = 30;             (* Max angle for .25 intervals *)
                  ANGLE_050 = 60;             (* Max angle for .50 intervals *)
                  ANGLE_100 = MAX_SCAN_ANGLE; (* Max angle for 1.0 intervals *)


               begin

```

```
                if (MAX_ANGLE >= ANGLE_000)     and (MAX_ANGLE <= ANGLE_010) then
                   CALC_INTERVAL := 0.10s;
                else if (MAX_ANGLE > ANGLE_010) and (MAX_ANGLE <= ANGLE_025) then
                   CALC_INTERVAL := 0.25s;
                else if (MAX_ANGLE > ANGLE_025) and (MAX_ANGLE <= ANGLE_050) then
                   CALC_INTERVAL := 0.50s;
                else if (MAX_ANGLE > ANGLE_050) and (MAX_ANGLE <= ANGLE_100) then
                   CALC_INTERVAL := 1.00s;
                else
                   SYSTEM_ERROR( 'In "CALC_INTERVAL"; Invalid angle specified' );


             end   (*  CALC_INTERVAL  *);



   (* ###################################################################### *)
   (*                                                                        *)
   (*   Calculate any pixel layovers (i.e. pixels of same distance from the  *)
   (*   radar imaging device) in the DTM line DLINE. Return a map line       *)
   (*   indicating where such pixels are located in DLINE.                   *)
   (*                                                                        *)
   (* ###################################################################### *)
        function CALC_PIXEL_LAYOVER( DLINE : INT_LINE; VMLINE : MAPSET ) : MAPSET;

           var
              I, J, K, MAP_COORD       : integer;
              PIXEL_LAYOVER            : boolean;
              DISTANCE                 : SHORT_LINE;
              PLMLINE                  : MAPSET;


           begin

              for I := 1 to DIM(PIXEL) do
                 if (MAX_IMAGE_SIZE - I) in VMLINE then
                    DISTANCE(I) := UNDEFINED;
                 else
                    DISTANCE(I) := sqr( (NADIR_COORD - I) * GRID_SPAC ) +
                                   sqr( ELEVATION - DLINE(I) );

              PLMLINE := (. .);
              for I := 1 to DIM(PIXEL) do
                 begin

                    MAP_COORD := MAX_IMAGE_SIZE - I;
                    if (MAP_COORD notin VMLINE) then
                       begin

                          PIXEL_LAYOVER := false;
                          for J := I+1 to DIM(PIXEL)-1 do
                             if DISTANCE(J) <> UNDEFINED then
                                begin

                                   K := J + 1;
                                   while (K <= DIM(PIXEL)) and
                                         (DISTANCE(K) = UNDEFINED) do
                                      incr( K );
```

```
                                if (K <= DIM(PIXEL)) and
                                   (DISTANCE(I) <= max( DISTANCE(J), DISTANCE(K) )) and
                                   (DISTANCE(I) >= min( DISTANCE(J), DISTANCE(K) )) then
                                   begin
                                      PIXEL_LAYOVER := true;
                                      if abs( DISTANCE(I) - DISTANCE(J) ) >
                                         abs( DISTANCE(I) - DISTANCE(K) ) then
                                         PLMLINE := PLMLINE + (. (MAX_IMAGE_SIZE-K) .);
                                      else
                                         PLMLINE := PLMLINE + (. (MAX_IMAGE_SIZE-J) .);
                                   end  (* if (K <= DIM(PIXEL)) ... *);
                               end  (* if DISTANCE(J) <> UNDEFINED *);
                           if PIXEL_LAYOVER then
                              PLMLINE := PLMLINE + (. MAP_COORD .);

                     end  (* if (MAP_COORD notin VMLINE) *);

                 end  (* for I *);


              CALC_PIXEL_LAYOVER := PLMLINE;


            end   (*  CALC_PIXEL_LAYOVER  *);



   (* ##################################################################### *)
   (*                                                                       *)
   (*   Set up and attach the files necessary for the system. Unit 0 is the *)
   (*   assumed input for the original DTM. If it is not assigned on the MTS *)
   (*   RUN command, the file name will be prompted for. Unit 1 is assumed  *)
   (*   input for the original registered image. Its file name will also be *)
   (*   prompted for if not assigned. The file assigned to unit 10 will be  *)
   (*   used for the output of the scanner image. The file assigned to unit *)
   (*   12 will be used for the output of the visibility map of the DTM, if *)
   (*   requested. The file assigned to unit 13 will be used for output of  *)
   (*   the pixel layover map of the DTM, if requested. If these units are  *)
   (*   not attached, they will default to MTS temporary files -IMAGE#,     *)
   (*   -VMAP#, and -PLMAP# respectively.                                   *)
   (*                                                                       *)
   (* ##################################################################### *)
         procedure FILES;

            var
               DTMFNAME, IMGFNAME    : INPUT_LINE;


            begin


               DTMFNAME := ' ';
               DEFAULT_FNAME( 'DEFAULT 0=*DUMMY*;' );
               GET_FNAME( 'QUERY FDNAME 0;', 0, DTMFNAME );

               if DTMFNAME = '*DUMMY*' then
                  begin

```

```
                  PRINT_MSG( UNIT_O_UNASSIGNED, '', 0 );
                  DTMFNAME := READ_STRING;
                  while DTMFNAME = ' ' do
                     begin
                        PRINT_MSG( INVALID_FILE, '', 0 );
                        DTMFNAME := READ_STRING;
                        if DTMFNAME = 'CANCEL' or DTMFNAME = 'HALT' then
                           halt;
                     end  (* while DTMFNAME = ' ' *);

               end  (* if DTMFNAME = '*DUMMY*' *);


            IMGFNAME := ' ';
            DEFAULT_FNAME( 'DEFAULT 1=*DUMMY*;' );
            GET_FNAME( 'QUERY FDNAME 1;', 0, IMGFNAME );

            if IMGFNAME = '*DUMMY*' then
               begin

                  PRINT_MSG( UNIT_1_UNASSIGNED, '', 0 );
                  IMGFNAME := READ_STRING;
                  while IMGFNAME = ' ' do
                     begin
                        PRINT_MSG( INVALID_FILE, '', 0 );
                        IMGFNAME := READ_STRING;
                        if IMGFNAME = 'CANCEL' or IMGFNAME = 'HALT' then
                           halt;
                     end  (* while IMGFNAME = ' ' *);

               end  (* if IMGFNAME = '*DUMMY*' *);


            if VISIBILITY_MAP    then
               begin
                  DEFAULT_FNAME( 'DEFAULT 12=-VMAP#;'  );
                  rewrite( VMFILE,  12 );
               end  (* if VISIBILITY_MAP *);

            if PIXEL_LAYOVER_MAP then
               begin
                  DEFAULT_FNAME( 'DEFAULT 13=-PLMAP#;' );
                  rewrite( PLMFILE, 13 );
               end  (* if PIXEL_LAYOVER_MAP *);

            DEFAULT_FNAME( 'DEFAULT 10=-IMAGE#;' );
            reset  ( DTM_FILE,   DTMFNAME );
            reset  ( IMAGE_FILE, IMGFNAME );
            rewrite( IFILE,      10 );


        end   (*  FILES  *);



   (* ##################################################################### *)
   (*                                                                       *)
   (*   Prompt for and get the dimensions of the DTM.                       *)
```

```
(*                                                                     *)
(* ################################################################### *)
   procedure GET_DIMENSIONS;

      begin

         PRINT_MSG( LINE_PROMPT,  ' ', 0 );
         DIM(LINE)  := trunc( GET_NUM( IMAGE_SIZE, INTEGRAL ) );

         PRINT_MSG( PIXEL_PROMPT, ' ', 0 );
         DIM(PIXEL) := trunc( GET_NUM( IMAGE_SIZE, INTEGRAL ) );

      end   (*  GET_DIMENSIONS  *);



(* ################################################################### *)
(*                                                                     *)
(*   Read and return the end of an input line. If COMPRESS is true then *)
(*   remove multiple embedded blanks, otherwise return the actual input *)
(*   line. The length of the input line is also returned.              *)
(*                                                                     *)
(* ################################################################### *)
    function GET_EOLN( var STR:INPUT_LINE; COMPRESS:boolean ) : integer;

       var
          I                   : integer;
          LAST_CHAR_BLANK     : boolean;
          CHR                 : char;


       begin
          I := 0;
          LAST_CHAR_BLANK := false;

          while ¬eof( INPUT ) and ¬eoln( INPUT ) do
             begin
                read( CHR );
                if CHR ¬= ' ' or not COMPRESS then
                   begin
                      if LAST_CHAR_BLANK and I>0 then
                         begin
                            I := I + 1;
                            STR(I) := ' '
                         end;
                      I := I + 1;
                      STR(I) := CHR;
                      LAST_CHAR_BLANK := false;
                   end;
                else
                   LAST_CHAR_BLANK := true;
             end;

          if not COMPRESS and I>0 then
             GET_EOLN := I - 1;
          else
             GET_EOLN := I
```

```
         end   (*  GET_EOLN  *);




(* ######################################################################## *)
(*                                                                          *)
(*   Return the numerical value of a string in the input stream. If         *)
(*   INTEGRAL_VALUE is true then the string must represent an integer,      *)
(*   otherwise a real number. The numerical value must also be in the       *)
(*   range specified by RANGE.                                              *)
(*                                                                          *)
(* ######################################################################## *)
   function GET_NUM( RANGE : VALUE_RANGE;
                        INTEGRAL_VALUE : boolean ) : real;

      var
         STR          : INPUT_LINE;
         VALID        : boolean;
         VALU         : real;
         I, J, MULT   : integer;


      begin
         repeat
            VALID := true;
            read( STR );
            if eof( INPUT ) then        (* CHECK FOR EOF                *)
               begin
                  VALID := false;
                  PRINT_MSG( EOF_NO, ' ', 0 )
               end;
            else
               begin
                  VALU := 0.0;          (* GET INTEGRAL PART            *)
                  I := 1;
                  MULT := 1;
                  while I <= MAX_LINE_LEN and STR(I) = ' ' do
                     I := I + 1;
                  if STR(I) = '+' or STR(I) = '-' then
                     begin
                        if STR(I) = '-' then
                           MULT := -1;
                        I := I + 1
                     end;
                  while STR(I) in (. '0'..'9' .) do
                     begin
                        VALU := 10.0 * VALU + ord( STR(I) ) - ord('0');
                        I := I + 1
                     end;
                  if STR(I) = '.' then
                     begin              (* GET FRACTIONAL PART          *)
                        J := 10;
                        I := I + 1;
                        while STR(I) in (. '0'..'9' .) do
                           begin
                              VALU := VALU + ( (ord(STR(I)) - ord('0'))/J);
                              I := I + 1;
```

```
                         J := J * 10
                      end;
                end;
            if STR(I) ¬= ' ' or ( eoln(INPUT) and I = MAX_LINE_LEN+1 ) then
               begin              (* INVALID NUMBER              *)
                  VALID := false;
                  if STR(I) ¬= ' ' then
                     PRINT_MSG( INVALID_NO, STR(I), 1  )
                  else
                     if RANGE in (. PERCENT .) then
                        begin
                           VALID := true;
                           MULT  := 1;
                           VALU  := DEFAULT;
                        end (* if RANGE in *);
                     else
                        PRINT_MSG( NULL_NO, ' ', 0 )
               end;
            else if INTEGRAL_VALUE and trunc(VALU) ¬= VALU then
               begin
                  VALID := false;
                  PRINT_MSG( NON_INTEGRAL_VALUE, '', 0 )
               end;
            else if STR(I) = ' ' and ( STR(I-1)='-' or STR(I-1)='+' ) then
               begin
                  VALID := false;
                  PRINT_MSG( UNEXPECTED_BLANK, '', 0 )
               end;
            else if (RANGE = POSITIVE and (MULT < 0 or VALU = 0.0)) or
                    (RANGE = IMAGE_SIZE and (MULT < 0  or
                             VALU < 1 or VALU > MAX_IMAGE_SIZE))    or
                    (RANGE = LINES and (MULT < 0        or
                             VALU < 1 or VALU > DIM(LINE)))         or
                    (RANGE = LINE_LENGTH and (MULT < 0 or
                             VALU < 1 or VALU > DIM(PIXEL)))        or
                    (RANGE = INTENSITY and (MULT < 0    or
                             VALU < MIN_INTENSITY       or
                             VALU > MAX_INTENSITY))                 or
                    (RANGE = SCAN_RANGE and (MULT < 0  or
                             VALU < MIN_SCAN_ANGLE      or
                             VALU > (2 * MAX_SCAN_ANGLE)))          or
                    (RANGE = NON_NEGATIVE and MULT < 0)             or
                    (RANGE = PERCENT and (MULT < 0      or
                             VALU < 0 or VALU > 100))               or
                    (RANGE = NON_ZERO and VALU = 0.0)               then
               begin
                  VALID := false;
                  PRINT_MSG( NOT_IN_RANGE, '', 0 )
               end;
         end;
      until VALID;

   GET_NUM := VALU * MULT


end (* GET_NUM *);
```

```
(* ###################################################################### *)
(*                                                                        *)
(*   Read in a requested reply (via GET_EOLN). If first letter of reply  *)
(*   is 'Y' or 'y' then return true, otherwise return false.             *)
(*                                                                        *)
(* ###################################################################### *)
    function GET_REPLY : boolean;

       const
          YES_1 = 'Y';  YES_2 = 'y';
          NO_1  = 'N';  NO_2  = 'n';

       var
          STR            : INPUT_LINE;
          I              : integer;


       begin

          I := GET_EOLN( STR, COMPRESSED );
          if (STR(1) ¬= YES_1) and (STR(1) ¬= YES_2) and
             (STR(1) ¬= NO_1)  and (STR(1) ¬= NO_2)     . then
             PRINT_MSG( ASSUMING_NO, '', 0 );
          GET_REPLY := (STR(1) = YES_1) or (STR(1) = YES_2);


       end   (*  GET_REPLY  *);



(* ###################################################################### *)
(*                                                                        *)
(*   Interpolate all undefined pixel values between the first defined   *)
(*   pixel of the image line - ILINE - and the last defined pixel of the *)
(*   image line. The new values are calculated as a function of the     *)
(*   values of the nearest defined pixels (i.e. the ones on either side  *)
(*   of the undefined pixel).                                            *)
(*                                                                        *)
(* ###################################################################### *)
    procedure INTERPOLATE( var ILINE : INT_LINE;
                           var FACT  : SHORT_LINE );

       var
          X, XX, XEND, START,
          STOP, INCREMENT, I          : integer;


       begin

          X     := 1;
          while FACT(X)     = 0 and X < (MAX_LINE_LEN+2) do
             incr( X );

          XEND := MAX_LINE_LEN + 2;
          while FACT(XEND) = 0 and XEND >= 1 do
             decr( XEND );

```

```
               incr( X );
               while X < XEND do
                  begin

                     if FACT(X) = 0 then
                        begin

                           I          := ILINE(X-1);
                           START      := X;
                           while FACT(X) = 0 do
                              incr( X );
                           STOP       := X - 1;
                           INCREMENT  := round( (ILINE(X) - ILINE(START-1)) / (X - START + 1) );
                           for XX := START to STOP do
                              begin
                                 I := I + INCREMENT;
                                 ILINE(XX) := round( (I + ILINE(XX) * FACT(XX)) /
                                                     (1.0 + FACT(XX)) );
                              end  (* for XX *);

                        end  (* if FACT(X) < 1 *);
                     incr( X );

                  end  (* while X < XEND *);


            end   (*  INTERPOLATE  *);



      (* ##################################################################### *)
      (*                                                                       *)
      (*   Print the message specified by MSG to the output stream. The        *)
      (*   parameters STRING and SIZE may be involved in the details of        *)
      (*   that message.                                                       *)
      (*                                                                       *)
      (* ##################################################################### *)
         procedure PRINT_MSG( MSG : MSG_TYPE; STRING : INPUT_LINE;
                              SIZE : integer );

            const
               MSG_PROMPT = ':';

            var
               I,J            : integer;
               STR            : INPUT_LINE;


            begin

               SET_PREFIX( MSG_PROMPT, 1 );

               if MSG in (. LINE_PROMPT, UNIT_0_UNASSIGNED, MAP_DESIRED,
                            ENTER_GRID_SPACING,
                            ENTER_NADIR_COORD, ENTER_SCAN_ANGLE,
                            ENTER_DEPRESSION_ANGLE, SIDELOOK_TRANSFORM,
                            UNIT_1_UNASSIGNED, ENTER_ELEVATION .) then
                  writeln;
```

```

               case MSG of
                  UNEXPECTED_EOF   :
                        writeln(' Unexpected EOF -- Ignored');
                  UNREC_CMD        :
                        begin
                           writeln(' Invalid command -- Input line ignored');
                           if ¬eoln( INPUT ) then
                              I := GET_EOLN( STR, COMPRESSED )
                        end;
                  EOLN_MSG         :
                        begin
                           I := GET_EOLN( STR, COMPRESSED );
                           if I ¬= 0 then
                              begin
                                 write(' Invalid character(s): "');
                                 WRITE_STRING( STR, I );
                                 writeln('" -- Ignored')
                              end
                        end;
                  LINE_PROMPT      :
                        writeln('&Enter number of lines of DTM and IMAGE (Integer {1-256})');
                  PIXEL_PROMPT     :
                        writeln('&Enter number of pixels per line        (Integer {1-256})');
                  EOF_NO           :
                        writeln('&Unexpected EOF, Enter number');
                  NULL_NO          :
                        writeln( '&Not expecting null line -- Ignored; Enter number' );
                  NON_INTEGRAL_VALUE:
                        writeln('&Non-integral value, Re-enter number');
                  UNEXPECTED_BLANK:
                        writeln('&Unexpected blank, Re-enter number');
                  NOT_IN_RANGE     :
                        writeln('&Out of range, Re-enter number');
                  UNIT_0_UNASSIGNED:
                        writeln(' Logical unit 0 has not been assigned.',
                                  EOL, '&   Enter FDname of location of DTM   ');
                  UNIT_1_UNASSIGNED:
                        writeln(' Logical unit 1 has not been assigned.',
                                  EOL, '&   Enter FDname of location of IMAGE ');
                  INVALID_FILE     :
                        writeln(' Invalid FDname. Enter again or', EOL,
                                '&Enter "CANCEL" to terminate program');
                  UNEXPECTED_INPUT_EOF:
                        writeln(' Unexpected EOF encountered on input',
                                ' -- DTM size updated');
                  INVALID_NO       :
                        begin
                           writeln('&Invalid character, "', STRING(1), '", Re-enter number');
                           I := GET_EOLN( STR, COMPRESSED )
                        end;
                  ASSUMING_NO      :
                        writeln(' Assuming "NO".');
                  MAP_DESIRED      :
                        begin
                           write('&Do you want a ');
                           WRITE_STRING( STRING, SIZE );
                           writeln(' map created? {y/n}');
```

```
                        end;
                  SIDELOOK_TRANSFORM:
                        writeln('&Do you want a side-looking transform? {y/n}');
                  ENTER_DEPRESSION_ANGLE:
                        begin
                           write(' Enter the ');
                           WRITE_STRING( STRING, SIZE );
                           writeln(' depression angle, from the horizontal,', EOL,
                                   '&     of the sensing system (degrees {',
                                   MIN_SCAN_ANGLE:0,'-',MAX_SCAN_ANGLE:0, '})');
                        end;
                  ENTER_NADIR_COORD:
                        writeln(' Enter the X (pixel) coordinate of the nadir',
                           EOL, '&     of the airborne scanner system (Integer {1-',
                               DIM(PIXEL):0,'})');
                  ENTER_SCAN_ANGLE:
                        writeln('&Enter the scan angle of the airborne scanner system (degrees {',
                               MIN_SCAN_ANGLE:0,'-',(2*MAX_SCAN_ANGLE):0,'})');
                  ENTER_GRID_SPACING:
                        writeln('&Enter DTM grid spacing (in DTM units)');
                  ENTER_ELEVATION :
                        writeln('&Enter elevation of airborne scanner system (in DTM units)');

               end  (*  case  *);

               if MSG in (. LINE_PROMPT, PIXEL_PROMPT, INVALID_NO,
                            NON_INTEGRAL_VALUE, UNEXPECTED_BLANK, NOT_IN_RANGE,
                            UNIT_0_UNASSIGNED, UNIT_1_UNASSIGNED, INVALID_FILE,
                            ENTER_ELEVATION, ENTER_NADIR_COORD,
                            ENTER_SCAN_ANGLE, MAP_DESIRED, ENTER_GRID_SPACING,
                            ENTER_DEPRESSION_ANGLE, SIDELOOK_TRANSFORM,
                            EOF_NO, NULL_NO .) then
                  readln;

               SET_PREFIX( PROMPT, 1 )


            end  (*  PRINT_MSG  *);



   (* ################################################################## *)
   (*                                                                    *)
   (*   If READ_REQD is true then read in a line from the file DTM_FILE,  *)
   (*   otherwise perform the subsequent operations on the present input *)
   (*   buffer. Convert the binary form of the buffer into integer form  *)
   (*   assuming two bytes per pixel.                                    *)
   (*                                                                    *)
   (* ################################################################## *)
      function READ_DTM_LINE( var DTM_FILE : DTMFILE; READ_REQD : boolean ) : INT_LINE;

         var
            LINE                : INT_LINE;
            I                   : integer;


         begin

```

```
            if READ_REQD then
               get( DTM_FILE );
            for I := 1 to DIM(PIXEL) do
               LINE(I) := int( DTM_FILE@((I-1)*2 + 1) ) * 256 +
                          int( DTM_FILE@((I-1)*2 + 2) );
            READ_DTM_LINE := LINE;


         end   (*  READ_DTM_LINE  *);




   (* ###################################################################### *)
   (*                                                                        *)
   (*   If READ_REQD is true then read in a line from the file IMAGE_FILE,   *)
   (*   otherwise perform the subsequent operations on the present input     *)
   (*   buffer. Convert the binary form of the buffer into integer form,     *)
   (*   assuming one byte per pixel.                                         *)
   (*                                                                        *)
   (* ###################################################################### *)
      function READ_IMAGE_LINE( var IMAGE_FILE : IMAGEFILE; READ_REQD : boolean ) : INT_LINE;

         var
            LINE                 : INT_LINE;
            I                    : integer;


         begin

            if READ_REQD then
               get( IMAGE_FILE );
            for I := 1 to DIM(PIXEL) do
               LINE(I) := int( IMAGE_FILE@(I) );
            READ_IMAGE_LINE := LINE;


         end   (*  READ_IMAGE_LINE  *);



   (* ###################################################################### *)
   (*                                                                        *)
   (*   Read a string from the input stream removing all leading blanks.     *)
   (*   The string is then returned.                                         *)
   (*                                                                        *)
   (* ###################################################################### *)
      function READ_STRING : INPUT_LINE;

         var
            I                    : integer;
            STR                  : INPUT_LINE;


         begin

            STR := ' ';
            I := 0;
```

```
                repeat
                   incr( I );
                   read( STR(I) );
                   if STR(I) = ' ' and I = 1 then
                      I := 0;
                until eof(INPUT) or eoln(INPUT) or (I ¬= 0 and STR(I) = ' ')
                                                  or I = MAX_LINE_LEN;

                READ_STRING := STR;


             end   (*  READ_STRING  *);




      (* ####################################################################### *)
      (*                                                                         *)
      (*   Transform the orthographic image, line by line, into a scanner image  *)
      (*   (geometrically, not radiometrically) as seen by a hypothetical        *)
      (*   airborne scanner system. Calculate a corresponding visibility map     *)
      (*   and a pixel layover map simultaneously, if either is requested.       *)
      (*                                                                         *)
      (* ####################################################################### *)
          procedure SCANNER_TRANSFORM;

             var
                I, J, RADIUS        : integer;
                ANGLE,
                PREV_TAN, CURR_TAN  : short;
                VISIBLE             : boolean;
                DLINE, ILINE,
                SCANNER_ILINE       : INT_LINE;
                SCANNER_FLINE       : SHORT_LINE;
                VMLINE, PLMLINE     : MAPSET;


             begin

                ILINE := READ_IMAGE_LINE( IMAGE_FILE, not READ_REQD );
                DLINE := READ_DTM_LINE  ( DTM_FILE,   not READ_REQD );

                for I := 1 to DIM(LINE) do
                   begin

                      VMLINE := (. .);
                      for J := 0 to MAX_LINE_LEN+2 do
                         begin
                            SCANNER_ILINE(J) := MIN_INTENSITY;
                            SCANNER_FLINE(J) := 0;
                         end  (* for J *);

                      SCANNER_ILINE(IMAGE_CENTER) := ILINE(NADIR_COORD);
                      SCANNER_FLINE(IMAGE_CENTER) := 1.0s;


                      PREV_TAN := 0.0s;
                      for J := NADIR_COORD-1 downto 1 do
```

```
          begin

             VISIBLE   := true;
             RADIUS    := abs( NADIR_COORD - J );
             CURR_TAN := RADIUS * GRID_SPAC / (ELEVATION - DLINE(J));
             if CURR_TAN > PREV_TAN then
                begin

                   ANGLE := ATAN( CURR_TAN );
                   if (ANGLE ¬= OUT_OF_RANGE) and (ANGLE <= SCAN_ANGLE) then
                      BUFFER_INSERT( ILINE(J), ANGLE, SCANNER_ILINE,
                                     SCANNER_FLINE,   LEFT_HALF      );
                   else
                      VMLINE := VMLINE + (. (MAX_IMAGE_SIZE-J) .);
                   PREV_TAN := CURR_TAN;

                end  (* if CURR_TAN > PREV_TAN *);
             else
                VMLINE := VMLINE + (. (MAX_IMAGE_SIZE-J) .);

          end  (* for J *);


       PREV_TAN := 0.0s;
       for J := NADIR_COORD+1 to DIM(PIXEL) do
          begin

             VISIBLE   := true;
             RADIUS    := abs( NADIR_COORD - J );
             CURR_TAN := RADIUS * GRID_SPAC / (ELEVATION - DLINE(J));
             if CURR_TAN > PREV_TAN then
                begin

                   ANGLE := ATAN( CURR_TAN );
                   if (ANGLE ¬= OUT_OF_RANGE) and (ANGLE <= SCAN_ANGLE) then
                      BUFFER_INSERT( ILINE(J), ANGLE, SCANNER_ILINE,
                                     SCANNER_FLINE,   RIGHT_HALF     );
                   else
                      VMLINE := VMLINE + (. (MAX_IMAGE_SIZE-J) .);
                   PREV_TAN := CURR_TAN;

                end  (* if CURR_TAN > PREV_TAN *);
             else
                VMLINE := VMLINE + (. (MAX_IMAGE_SIZE-J) .);

          end  (* for J *);


       SCANNER_FLINE(IMAGE_CENTER) := 0.0s;
       INTERPOLATE( SCANNER_ILINE, SCANNER_FLINE );

       WRITE_IMAGE_LINE( IFILE, SCANNER_ILINE, 1 );

       if VISIBILITY_MAP then
          WRITE_MAP_LINE( VMFILE, VMLINE );

       if PIXEL_LAYOVER_MAP then
          begin
```

```
                        PLMLINE := CALC_PIXEL_LAYOVER( DLINE, VMLINE );
                        WRITE_MAP_LINE( PLMFILE, PLMLINE );
                     end  (* if PIXEL_LAYOVER_MAP *);

                  if I < DIM(LINE) then
                     begin
                        ILINE := READ_IMAGE_LINE( IMAGE_FILE, READ_REQD );
                        DLINE := READ_DTM_LINE  ( DTM_FILE,   READ_REQD );
                     end  (* if I < DIM(LINE) *);


              end  (* for I *);


         end   (*  SCANNER_TRANSFORM  *);




 (* ####################################################################### *)
 (*                                                                         *)
 (*   Transform the orthographic image, line by line, into a side-looking   *)
 (*   scanner image - as is the type of image produced by a SLAR system -   *)
 (*   (geometrically, not radiometrically) as seen by a hypothetical        *)
 (*   airborne sensing system. Calculate a corresponding visibility map     *)
 (*   and a pixel layover map simultaneously, if either is requested.       *)
 (*                                                                         *)
 (* ####################################################################### *)
    procedure SIDE_LOOK_TRANSFORM;

       var
          I, J, RADIUS        : integer;
          ANGLE,
          PREV_TAN, CURR_TAN  : short;
          VISIBLE             : boolean;
          DLINE, ILINE,
          SL_ILINE            : INT_LINE;
          SL_FLINE            : SHORT_LINE;
          VMLINE, PLMLINE     : MAPSET;


       begin

           ILINE := READ_IMAGE_LINE( IMAGE_FILE, not READ_REQD );
           DLINE := READ_DTM_LINE  ( DTM_FILE,   not READ_REQD );

           for I := 1 to DIM(LINE) do
              begin

                 VMLINE := (. .);
                 for J := 0 to MAX_LINE_LEN+2 do
                    begin
                       SL_ILINE(J) := MIN_INTENSITY;
                       SL_FLINE(J) := 0;
                    end  (* for J *);

                 PREV_TAN := 0.0s;
                 for J := 1 to DIM(PIXEL) do
```

```
                        begin

                           VISIBLE  := true;
                           RADIUS   := abs( NADIR_COORD - J );
                           CURR_TAN := RADIUS * GRID_SPAC / (ELEVATION - DLINE(J));
                           if CURR_TAN > PREV_TAN then
                              begin

                                 ANGLE := ATAN( CURR_TAN );
                                 if (ANGLE ¬= OUT_OF_RANGE)         and
                                    (ANGLE <= (90-MIN_DEP_ANGLE)) and
                                    (ANGLE >= (90-MAX_DEP_ANGLE)) then
                                    BUFFER_INSERT( ILINE(J), (ANGLE-(90-MAX_DEP_ANGLE)),
                                                   SL_ILINE, SL_FLINE, RIGHT_HALF        );
                                 else
                                    VMLINE := VMLINE + (. (MAX_IMAGE_SIZE-J) .);
                                 PREV_TAN := CURR_TAN;

                              end  (* if CURR_TAN > PREV_TAN *);
                           else
                              VMLINE := VMLINE + (. (MAX_IMAGE_SIZE-J) .);

                        end  (* for J *);


                     INTERPOLATE( SL_ILINE, SL_FLINE );

                     WRITE_IMAGE_LINE( IFILE, SL_ILINE, 1 );

                     if VISIBILITY_MAP then
                        WRITE_MAP_LINE( VMFILE, VMLINE );

                     if PIXEL_LAYOVER_MAP then
                        begin
                           PLMLINE := CALC_PIXEL_LAYOVER( DLINE, VMLINE );
                           WRITE_MAP_LINE( PLMFILE, PLMLINE );
                        end  (* if PIXEL_LAYOVER_MAP *);

                     if I < DIM(LINE) then
                        begin
                           ILINE := READ_IMAGE_LINE( IMAGE_FILE, READ_REQD );
                           DLINE := READ_DTM_LINE  ( DTM_FILE,   READ_REQD );
                        end  (* if I < DIM(LINE) *);


                  end  (* for I *);


            end   (*  SIDE_LOOK_TRANSFORM  *);



      (* ##################################################################### *)
      (*                                                                       *)
      (*   Initialize the system by setting up the files, getting the dimensions *)
      (*   of the DTM, and reading the necessary system parameters.            *)
      (*                                                                       *)
      (* ##################################################################### *)
```

```
        procedure SYS_INIT;

           var
              CENTRAL_ANGLE        : real;
              I, J                 : integer;


           begin

              SET_PREFIX( PROMPT, 1 );
              GET_DIMENSIONS;

              PRINT_MSG( ENTER_GRID_SPACING, '', 0 );
              GRID_SPAC    := roundtoshort( GET_NUM( POSITIVE,   NOT INTEGRAL )     );

              PRINT_MSG( ENTER_ELEVATION,    '', 0 );
              ELEVATION    := roundtoshort( GET_NUM( POSITIVE,   NOT INTEGRAL )     );

              PRINT_MSG( SIDELOOK_TRANSFORM, '', 0 );
              SIDE_LOOK    := GET_REPLY;

              if SIDE_LOOK then
                 begin

                    PRINT_MSG( ENTER_DEPRESSION_ANGLE, 'maximum', 7 );
                    MAX_DEP_ANGLE := roundtoshort( GET_NUM( SCAN_RANGE,
                                                   NOT INTEGRAL ) );

                    PRINT_MSG( ENTER_DEPRESSION_ANGLE, 'minimum', 7 );
                    MIN_DEP_ANGLE := roundtoshort( GET_NUM( SCAN_RANGE,
                                                   NOT INTEGRAL ) );

                    CENTRAL_ANGLE := (90 - MAX_DEP_ANGLE + ((MAX_DEP_ANGLE -
                                     MIN_DEP_ANGLE) * 0.80s)) * DEG_TO_RAD;

                    NADIR_COORD  := trunc( DIM(PIXEL)/2 ) -
                                    round( tan( CENTRAL_ANGLE )*ELEVATION/GRID_SPAC );
                    MAX_ANGLE    := 90 - trunc( MIN_DEP_ANGLE ) + 1;
                    IMAGE_CENTER := 0;

                 end  (* if SIDE_LOOK *);
              else
                 begin

                    PRINT_MSG( ENTER_SCAN_ANGLE,   '', 0 );
                    SCAN_ANGLE   := roundtoshort( GET_NUM( SCAN_RANGE, NOT INTEGRAL ) / 2 );

                    PRINT_MSG( ENTER_NADIR_COORD,  '', 0 );
                    NADIR_COORD  := trunc( GET_NUM( LINE_LENGTH, INTEGRAL ) );

                    MAX_ANGLE    := trunc( SCAN_ANGLE + 1 );
                    IMAGE_CENTER := trunc( MAX_IMAGE_SIZE/2 );

                 end  (* else not SIDE_LOOK *);

              if MAX_ANGLE > MAX_SCAN_ANGLE then
                 MAX_ANGLE := MAX_SCAN_ANGLE;

```

```
                  TAN_INIT( MAX_ANGLE );
                  if SIDE_LOOK then
                     INTERVAL := CALC_INTERVAL( round( (MAX_DEP_ANGLE-
                                 MIN_DEP_ANGLE)/2 ) );
                  else
                     INTERVAL := CALC_INTERVAL( MAX_ANGLE );

                  PRINT_MSG( MAP_DESIRED, '  visibility ', 13 );
                  VISIBILITY_MAP     := GET_REPLY;

                  PRINT_MSG( MAP_DESIRED, 'pixel layover', 13 );
                  PIXEL_LAYOVER_MAP := GET_REPLY;

                  FILES;


                end   (*  SYSTEM_INITIALIZATION  *);



        (* ################################################################# *)
        (*                                                                   *)
        (*   If a system error occurs, output the message MESSAGE and halt the     *)
        (*   execution of the system.                                        *)
        (*                                                                   *)
        (* ################################################################# *)
             procedure SYSTEM_ERROR( MESSAGE : STR_50 );

                begin

                  writeln(' ===> SYSTEM ERROR ===> ',MESSAGE );
                  halt;


                end   (*  SYSTEM_ERROR  *);




        (* ################################################################# *)
        (*                                                                   *)
        (*   Initialize the tangent lookup table, TAN_LOOKUP.                *)
        (*                                                                   *)
        (* ################################################################# *)
            procedure TAN_INIT( SIZE : integer );

               var
                  DEG                 : integer;


               begin

                  if SIZE > MAX_SCAN_ANGLE then
                     SYSTEM_ERROR( 'In "TAN_INIT"; Invalid size specified' );

                  for DEG := 0 to SIZE do
                     if DEG = 90 then
                        TAN_LOOKUP(DEG) := UNDEFINED;
```

```
                  else
                     TAN_LOOKUP(DEG) := roundtoshort( tan( DEG * DEG_TO_RAD ) );


              end   (*  TAN_INIT  *);



      (* ###################################################################### *)
      (*                                                                         *)
      (*   Write the line LINE to the file IMAGE starting at pixel START.        *)
      (*   It is assumed that each pixel has a value between 0 and 255.          *)
      (*                                                                         *)
      (* ###################################################################### *)
          procedure WRITE_IMAGE_LINE( var IMAGE : IMAGEFILE;
                                  LINE : INT_LINE; START : integer );

              var
                 I                    : integer;


              begin

                 IMAGE@ := ' ';

                 for I := 1 to (START-1) do
                    IMAGE@(I) := char( MIN_INTENSITY );

                 for I := START to MAX_IMAGE_SIZE do
                    IMAGE@(I) := char( LINE(I) );

                 put( IMAGE );


              end   (*  WRITE_IMAGE_LINE  *);



      (* ###################################################################### *)
      (*                                                                         *)
      (*   Write the map line LINE to the map file MAP.                          *)
      (*                                                                         *)
      (* ###################################################################### *)
          procedure WRITE_MAP_LINE( var MAP : MAPFILE; LINE : MAPSET );


              begin

                 writeln( MAP, LINE );


              end   (*  WRITE_MAP_LINE  *);



      (* ###################################################################### *)
      (*                                                                         *)
      (*   Write the string STR to the present output stream. The string is of   *)
```

```
      (*   length SIZE. If the length is less than one, the operation does not   *)
      (*   take place.                                                           *)
      (*                                                                         *)
      (* ###################################################################### *)
           procedure WRITE_STRING( STR : INPUT_LINE; SIZE : integer );

              var
                 I                  : integer;


              begin

                 if SIZE > 0 then
                    for I := 1 to SIZE do
                       write( STR(I) )


              end   (*  WRITE_STRING  *);



      (* ###################################################################### *)
      (*                                                                         *)
      (*   Initialize the system and then proceed to transform the original      *)
      (*   image into a geometrically correct radar image.                       *)
      (*                                                                         *)
      (* ###################################################################### *)
          begin   (*  MAIN ROUTINE  *)

             SYS_INIT;
             if SIDE_LOOK then
                SIDE_LOOK_TRANSFORM
             else
                SCANNER_TRANSFORM;

          end   (*  MAIN ROUTINE  *).
```