

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

© Michael Richard Palmer, 1982

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

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

Abstract	ii
Table of Contents	iii
List of Figures	v
Acknowledgement	vi
1.0 Introduction	1
1.1 Motivation	2
1.2 Terminology	4
1.3 Scope	7
2.0 Synthetic Orthographic Stereo	8
2.1 Theory	8
2.2 Implementation	13
3.0 Synthetic Perspective Stereo	17
3.1 Theory	17
3.2 Implementation	29
4.0 Synthetic Airborne Scanner Imagery	35
4.1 Theory	39
4.1.1 Radar Considerations	45
4.2 Implementation	51
5.0 Conclusion	55
5.1 Future Work	56

Bibliography	57
Appendix A	59
Appendix B	76
Appendix C	112

List of Figures

Figure 1.	Orthographic stereo pair	9
Figure 2.	Geometry of an orthographic stereo pair ...	11
Figure 3.	Perspective stereo pair	18
Figure 4.	Perspective stereo visibility map	20
Figure 5.	Geometry of a vertical aerial photograph ..	22
Figure 6.	Terrain visibility geometry (perspective) .	25
Figure 7.	Terrain visibility - perspective	27
Figure 8.	Direction of relief displacement	31
Figure 9.	Airborne scanner image	36
Figure 10.	Airborne scanner visibility map	37
Figure 11.	Airborne scanner pixel layover map	38
Figure 12.	Operation of an airborne scanner system ...	40
Figure 13.	Geometry of an airborne scanner system	42
Figure 14.	Terrain visibility - airborne scanner	44
Figure 15.	Geometry of a radar imaging system	46
Figure 16.	Geometry of a side-looking radar system ...	48
Figure 17.	Pixel layover - radar	50

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 overlaid 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. Similarly, 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.

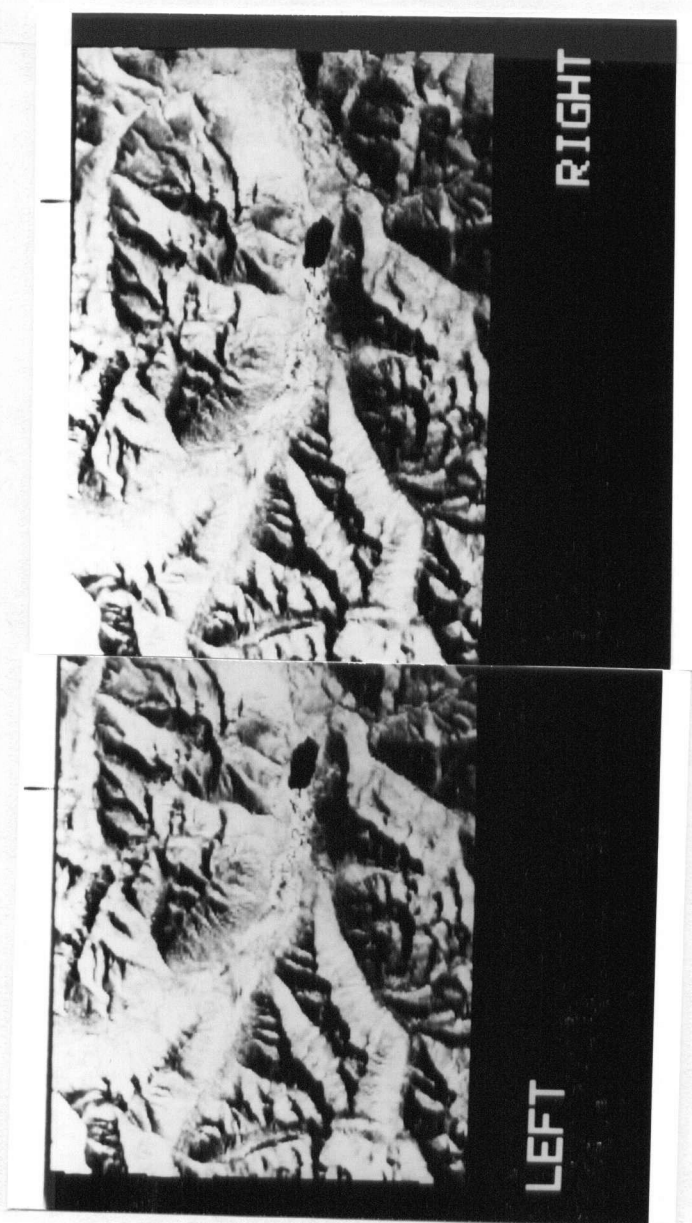


Figure 1. Orthographic stereo pair (St. Mary Lake region of southeast British Columbia).

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 digital terrain model (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 (X_c, Y_c) on the ground datum, forming an angle of convergence of 2θ . From this geometry it is clear that

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

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

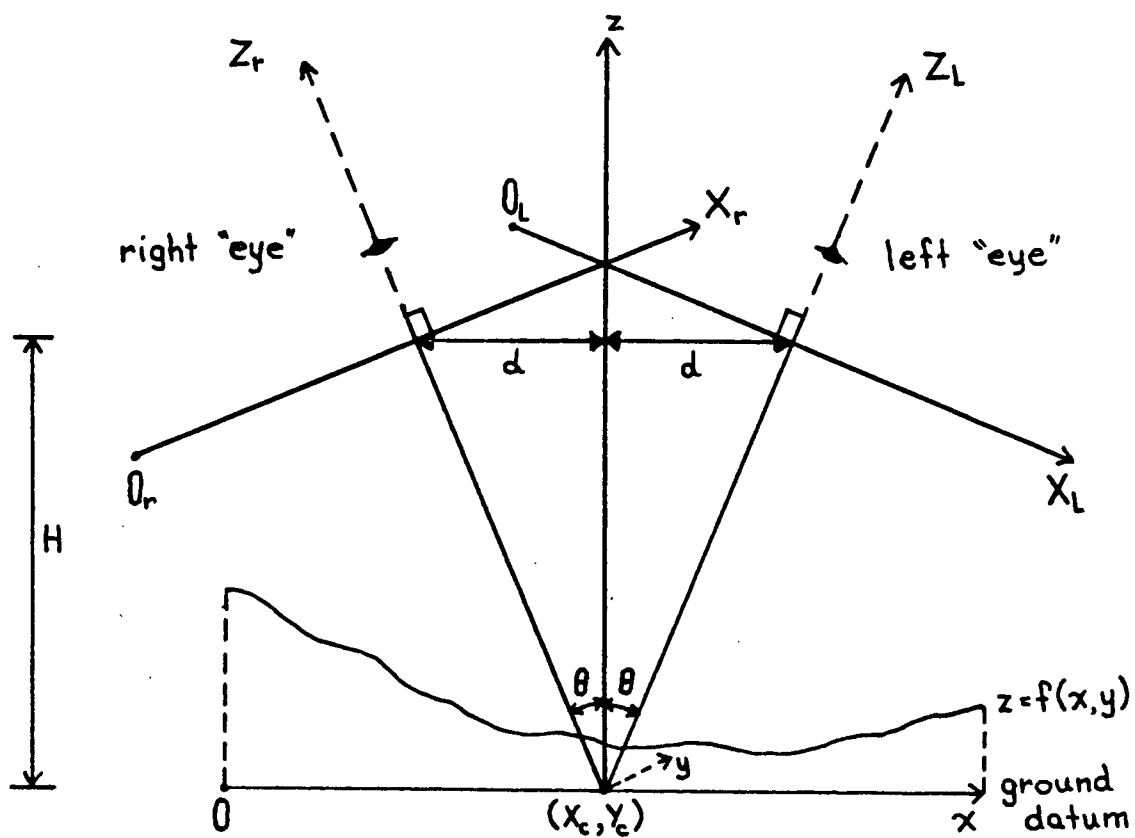


Figure 2. Geometry of an orthographic stereo pair.

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 (X_l,Y_l,Z_l) and the right eye coordinate system (X_r,Y_r,Z_r) . (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 X_l axis and the X_r axis intercept the x axis of the surface model at coordinate (X_c,Y_c) .) One obtains:

$$X_l = X_c + (X - X_c) \cdot \cos\theta + Z \cdot \sin\theta \quad (2.2)$$

$$Y_l = Y \quad (2.3)$$

$$Z_l = Z \cdot \cos\theta - (X - X_c) \cdot \sin\theta \quad (2.4)$$

$$X_r = X_c + (X - X_c) \cdot \cos\theta - Z \cdot \sin\theta \quad (2.5)$$

$$Y_r = Y \quad (2.6)$$

$$Z_r = Z \cdot \cos\theta + (X - X_c) \cdot \sin\theta \quad (2.7)$$

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, $(X_l - X_r)$, 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

$$2 \cdot Z \cdot (d/H) = Z \cdot S/H \quad (2.9)$$

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 X_1 , then the (X,Z) pair with the greatest Z_1 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 X_l and an X_r are calculated for the two new lines. The pixel value corresponding to position X of the original image line is assigned to position X_l of the left image line and to position X_r of the right image line. When this displacement is introduced, it is likely that X_l and X_r 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 X_l and X_r) 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 X_c 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 X_c , 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 X_c . 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 X_l (or X_r) rarely equals another X_l (or X_r) 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

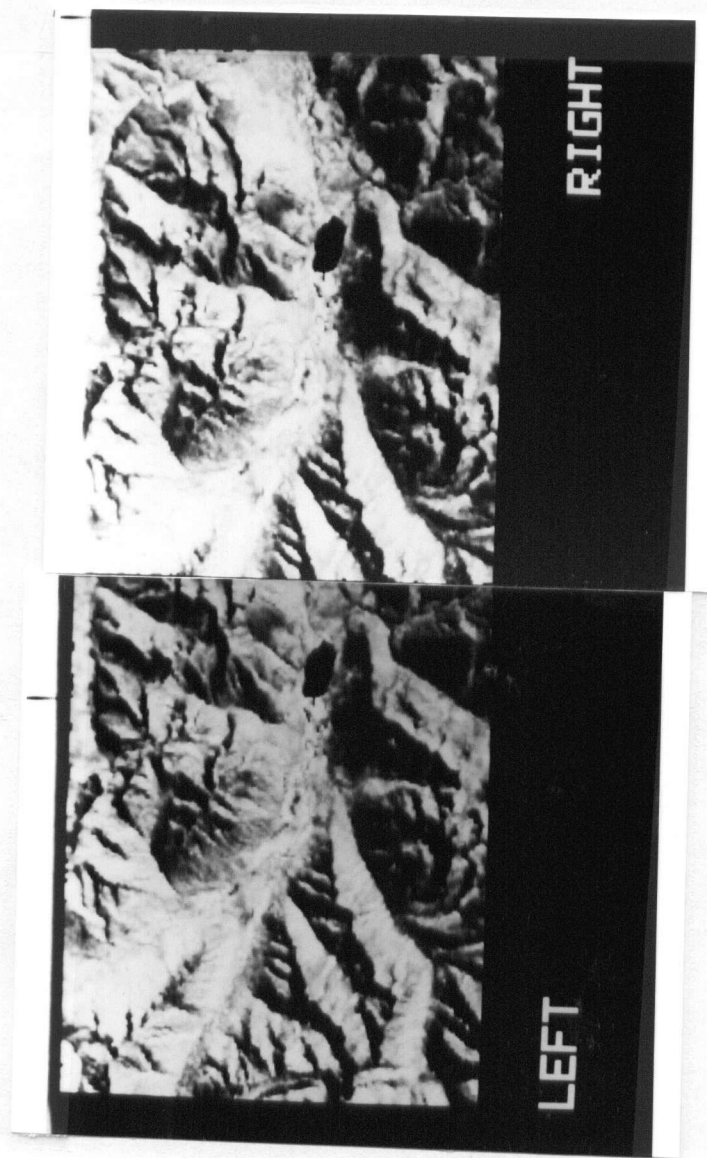


Figure 3. Perspective stereo pair (St. Mary Lake region of southeast British Columbia).

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 (X_p, Y_p) . (The coordinate (X_p, Y_p) , the centre of the vertical aerial photograph, is referred to as the principal point.) Further suppose that the terrain at coordinate

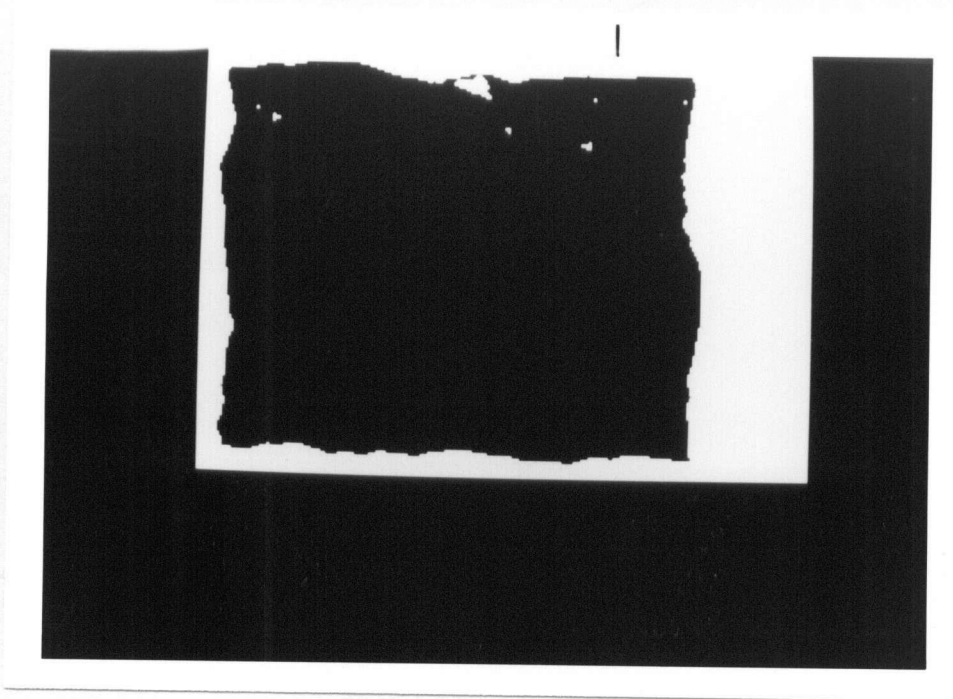


Figure 4. Perspective stereo visibility map.

(X,Y) is at elevation Z above the ground datum and that the radial distance from (X_p,Y_p) 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

$$a / f = R / H \quad (3.1)$$

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

$$a' / f = R / (H - Z) \quad (3.2)$$

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

$$a = Rf / H \quad (3.3)$$

$$a' = Rf / (H - Z) \quad (3.4)$$

An equation for the relief displacement, commonly referred

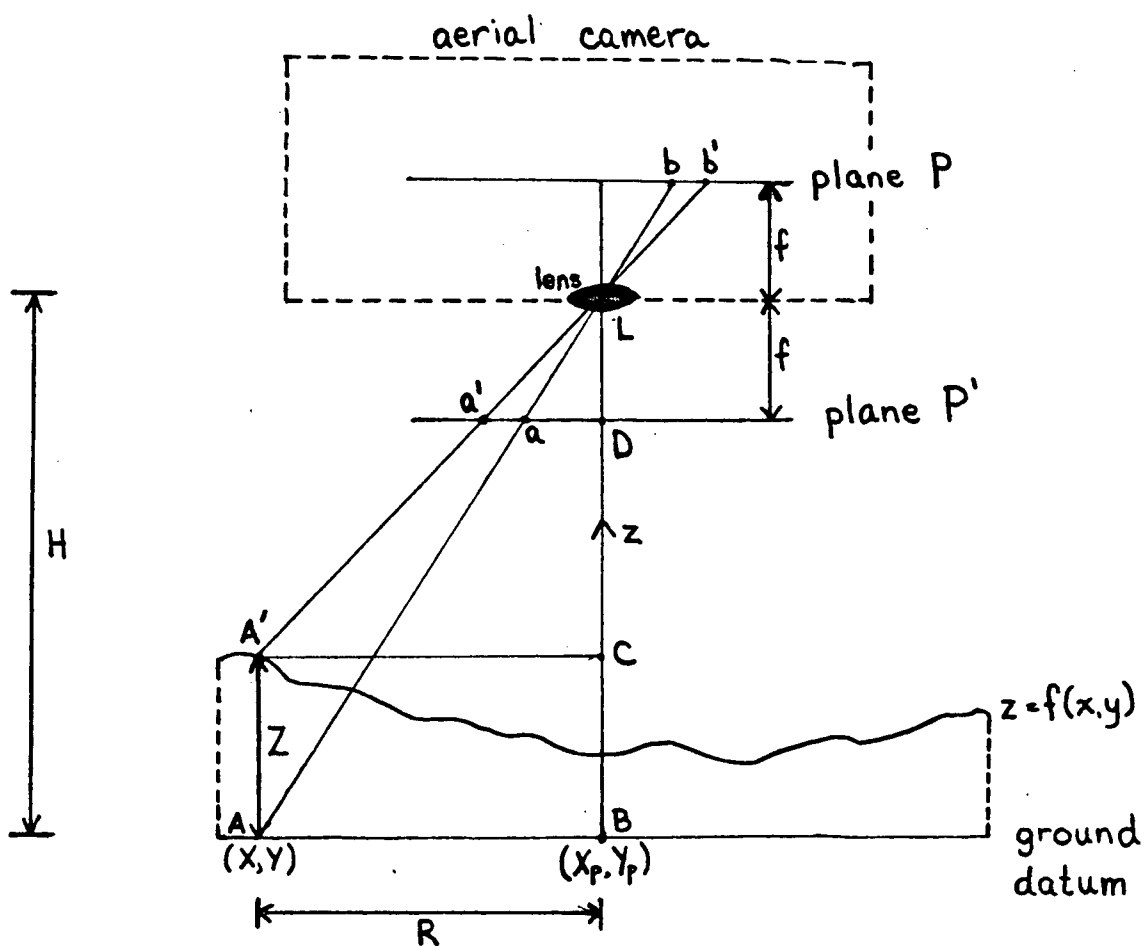


Figure 5. Geometry of a vertical aerial photograph (adapted from [SAB178], figure 2.9A).

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

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

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 (X_p,Y_p) , 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 (X_v,Y_v) has elevation Z_v above the ground datum, that the radial distance from (X_v,Y_v) to (X_p,Y_p) is R_v , 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 (X_v,Y_v) and (X_p,Y_p) will block the visibility of the terrain at (X,Y) if

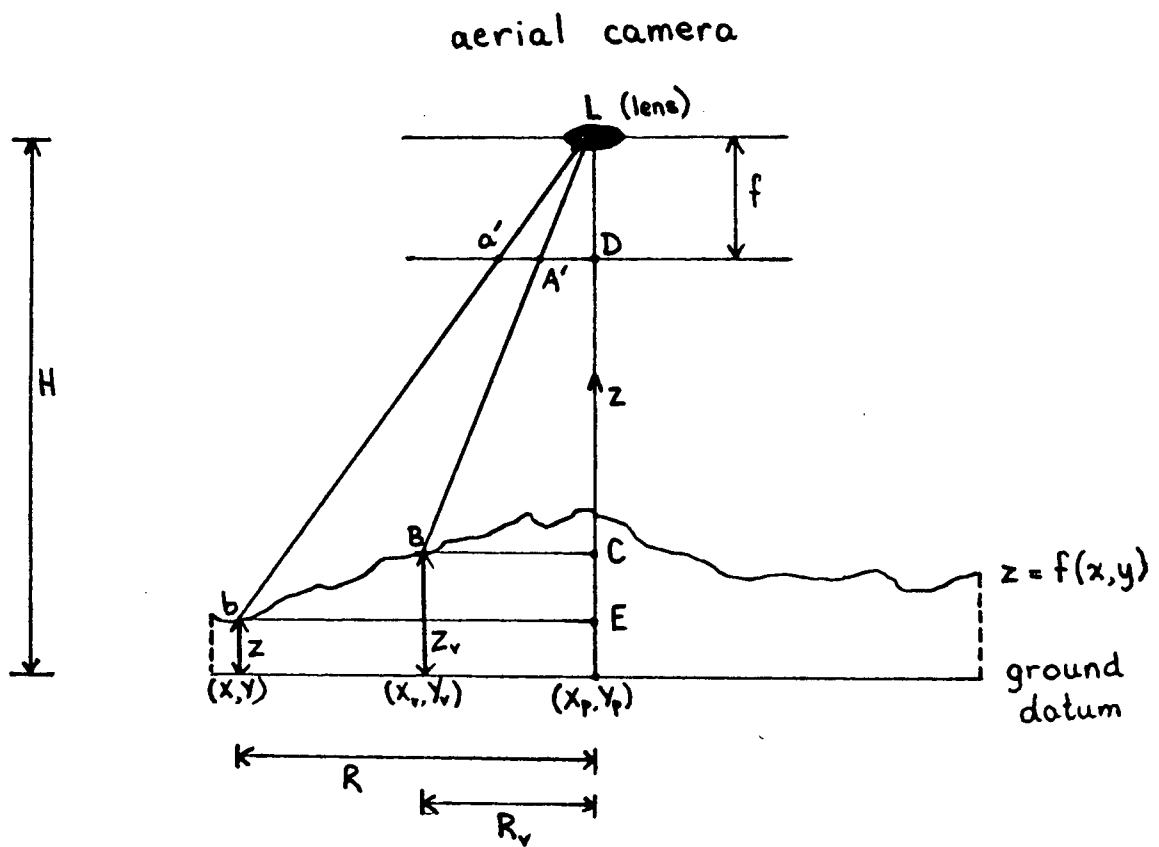


Figure 6. Terrain visibility geometry (perspective).

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

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

$$A' / f = R_v / (H - Z_v) \quad (3.7)$$

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

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

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

$$A' = R_v f / (H - Z_v) \quad (3.9)$$

$$a' = R f / (H - Z) \quad (3.10)$$

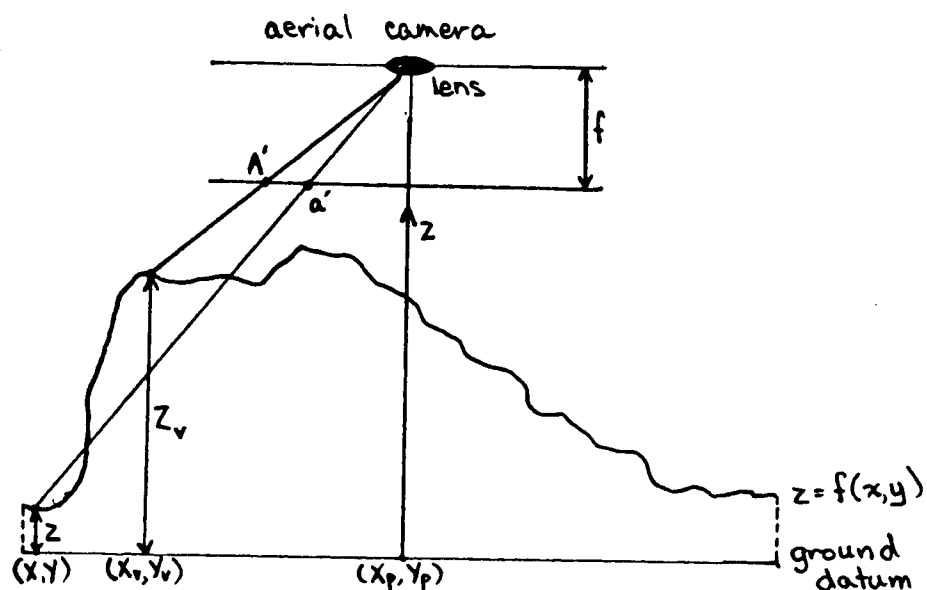
Therefore, substituting into formula (3.6), terrain between coordinates (X_v,Y_v) and (X_p,Y_p) will block the visibility of the terrain at coordinate (X,Y) if

$$R_v f / (H - Z_v) \geq R f / (H - Z) \quad (3.11)$$

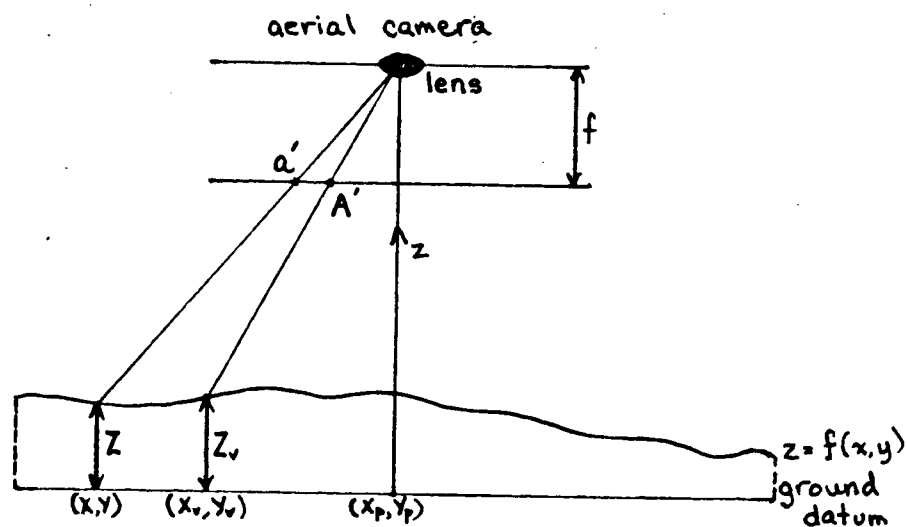
and hence, if

$$R_v / (H - Z_v) \geq R / (H - Z) \quad (3.12)$$

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 (X_v,Y_v) and (X_p,Y_p) 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 (X_v,Y_v) and (X_p,Y_p) .

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 $(R+d,\theta)$, 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.

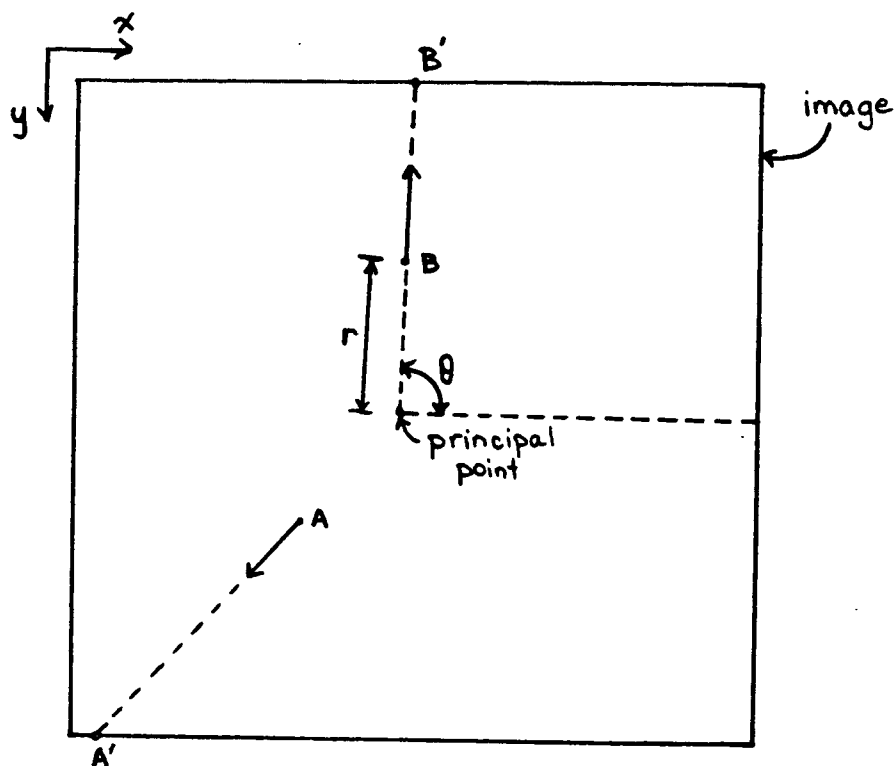


Figure 8. Direction of relief displacement.

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 (X_p,Y_p) 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 (X_p,Y_p) , 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 (X_p,Y_p) . 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 (X_p,Y_p)). 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 R_v and Z_v 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.

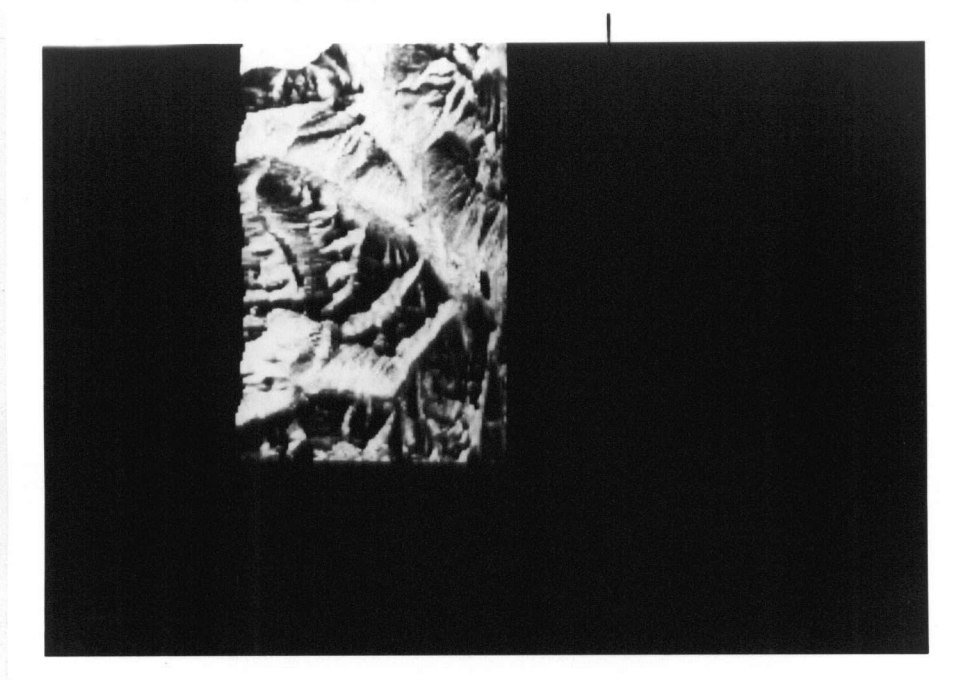


Figure 9. Airborne scanner image (St. Mary Lake region of southeast British Columbia).

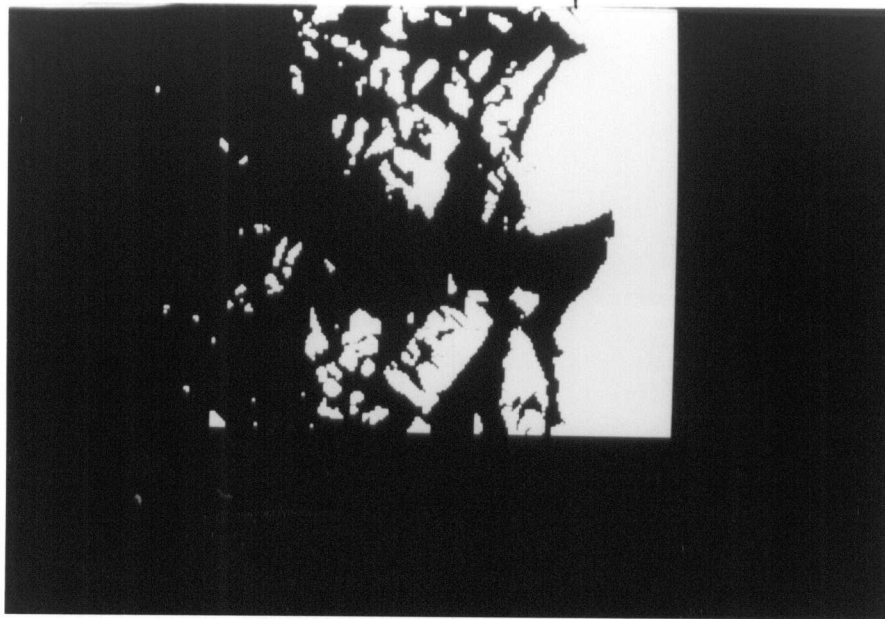


Figure 10. Airborne scanner visibility map.

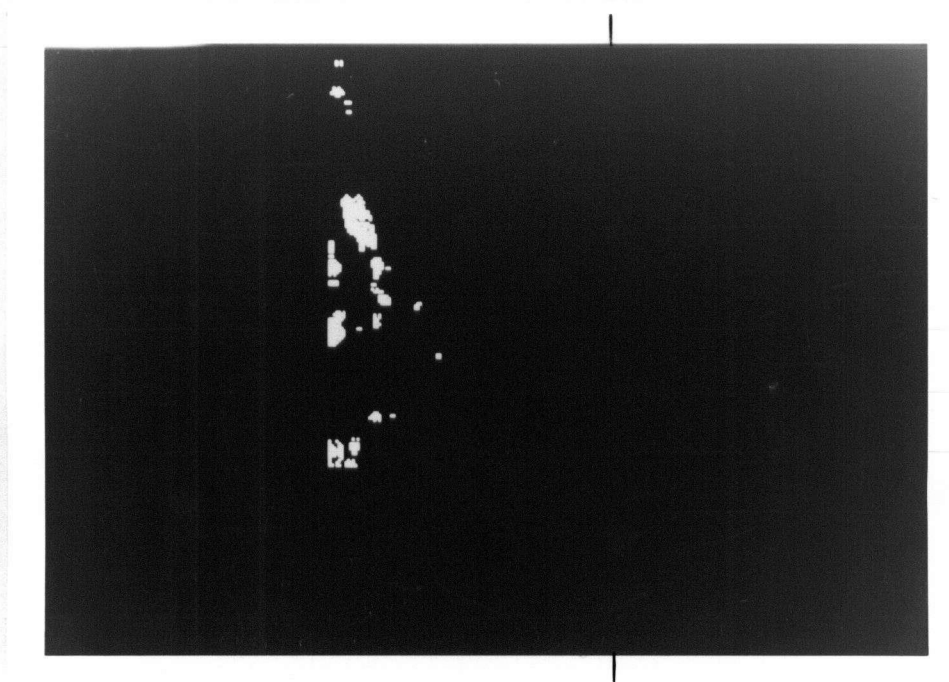


Figure 11. Airborne scanner pixel layover map.

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.)

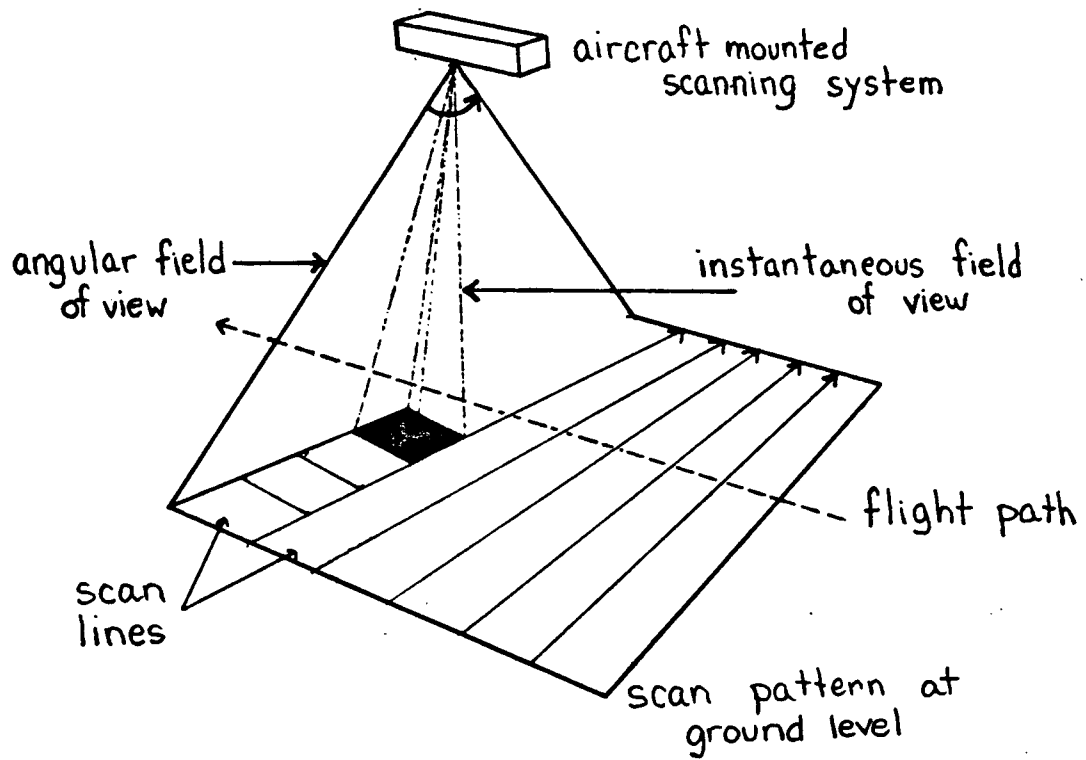


Figure 12. Operation of an airborne scanner system.
(Adapted from [SAB169], figure 2.)

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 (X_p, 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,

$$\tan\theta = (X - X_p) / (H - Z) \quad (4.1)$$

and hence

$$\theta = \arctan\{ (X - X_p) / (H - Z) \} \quad (4.2)$$

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

$$\theta = \arctan\{ \text{abs}\{ X - X_p \} / (H - Z) \} \quad (4.3)$$

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) X_p , the flight path of the airborne scanner system.

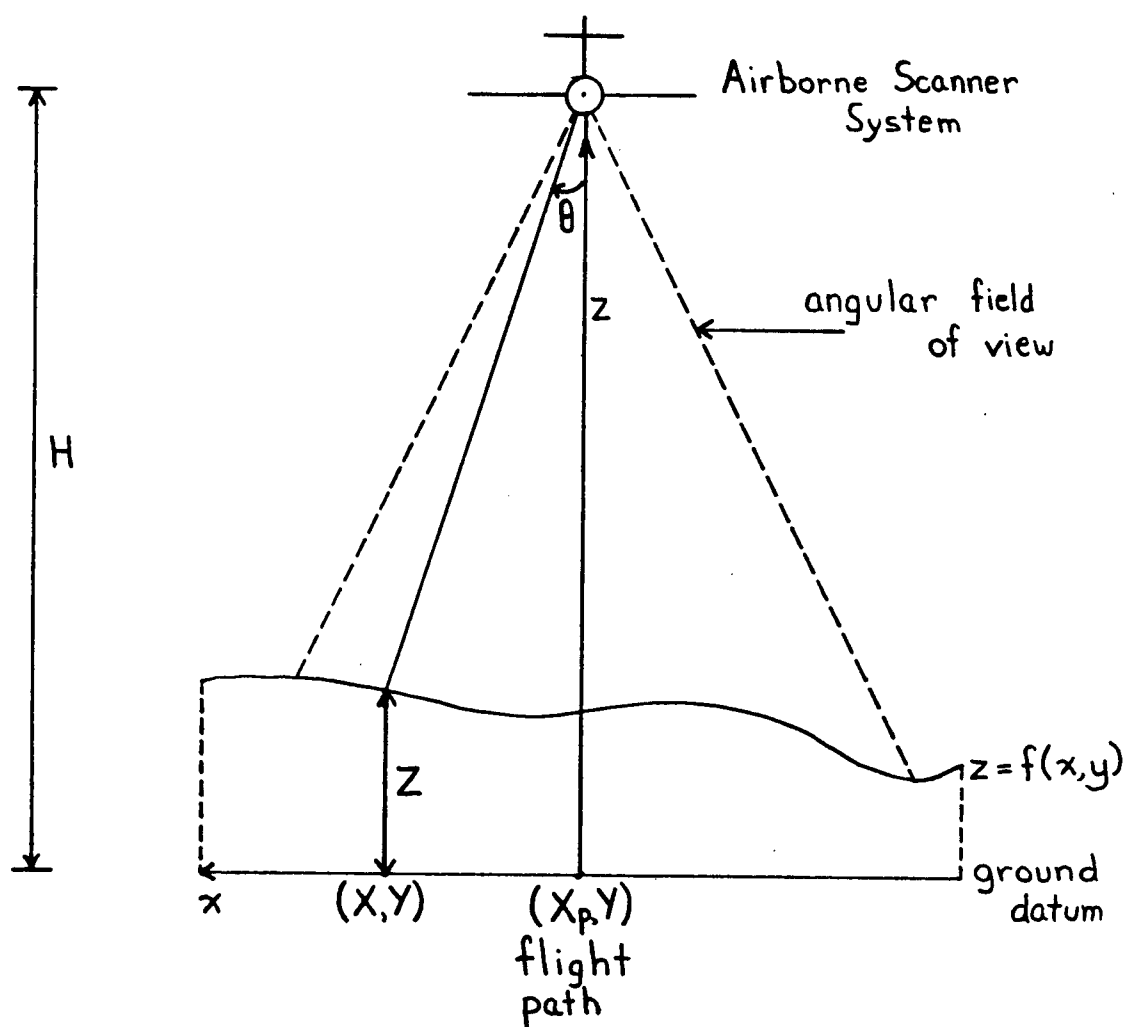


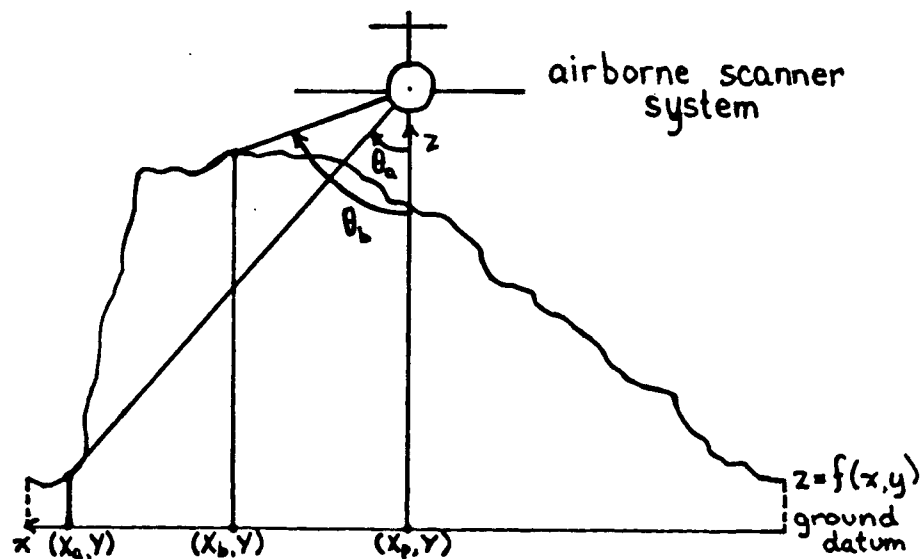
Figure 13. Geometry of an airborne scanner system.

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

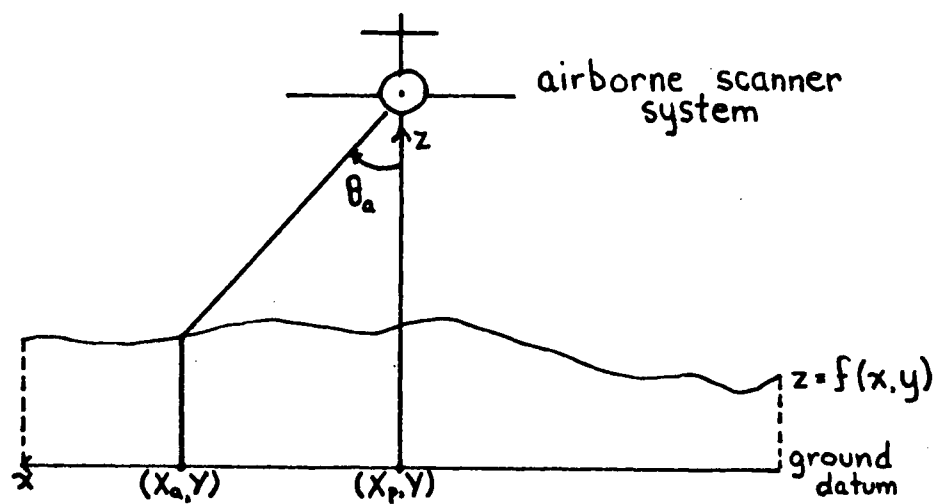
Since the imaging equation (4.3) is not a function of the y coordinate of the terrain, a scan line is imaged independently of other scan lines; the calculations to synthesize an airborne scanner image need not explicitly consider the y coordinate (assuming scan lines to be parallel to the x axis and 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 (X_p,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 X_b is between X_a and the flight path. Therefore, by the preceding visibility rule, terrain at ground coordinate (X_a,Y) is not visible, as is shown. In figure 14(b), there is no terrain between ground coordinates (X_a,Y) and (X_p,Y) that is imaged at an RIA angle larger than θ_a . Hence, terrain at ground coordinate (X_a,Y) is visible.



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



(b) Terrain at (X_a, 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 (X_p, 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 $2 \cdot D$ (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].

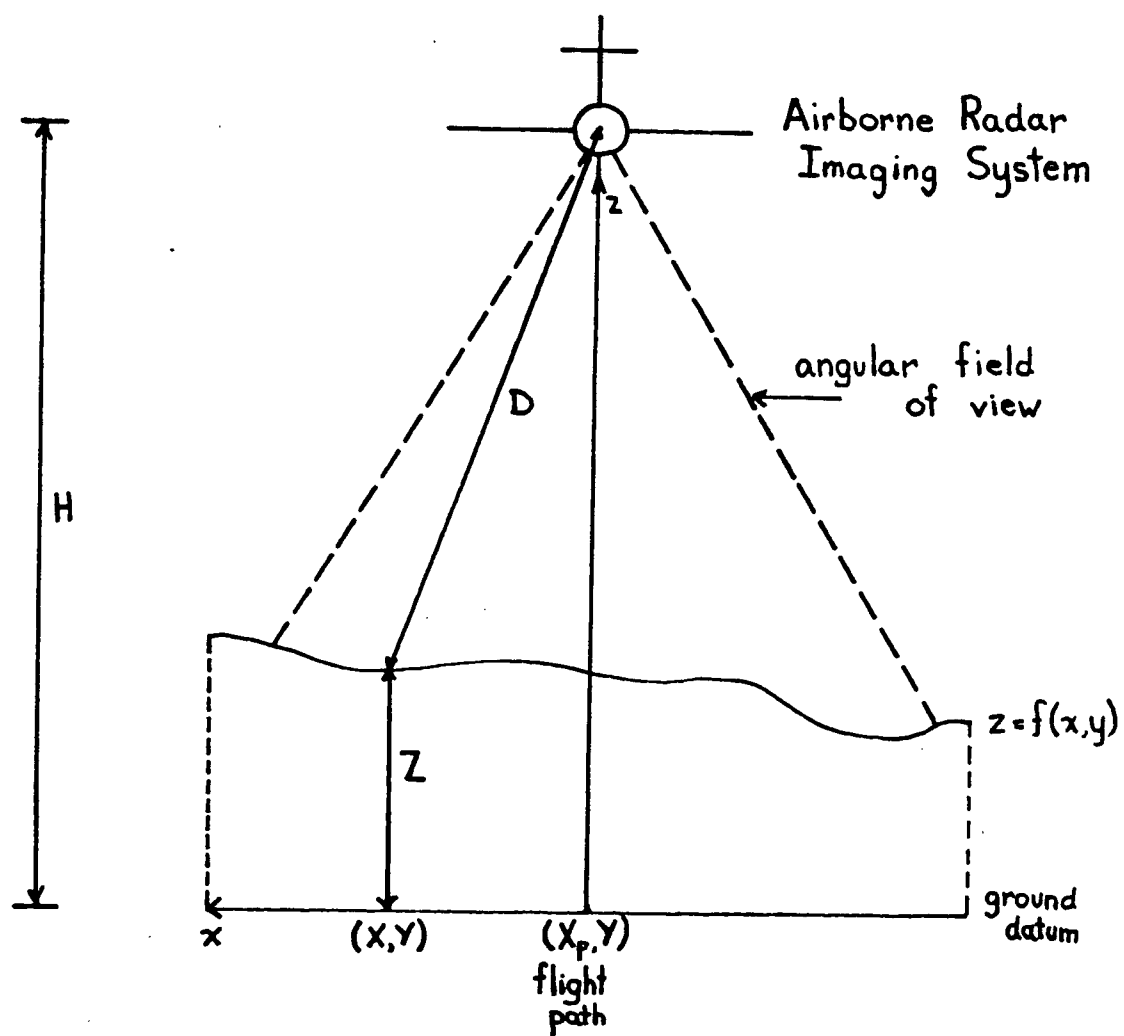


Figure 15. Geometry of a radar imaging system.

equation:

$$D = \sqrt{(X_p - X)^2 + (H - Z)^2} \quad (4.4)$$

Therefore, the distance the radar pulse must travel is:

$$2 \cdot D = 2 \cdot \sqrt{(X_p - X)^2 + (H - Z)^2} \quad (4.5)$$

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, X_p . 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

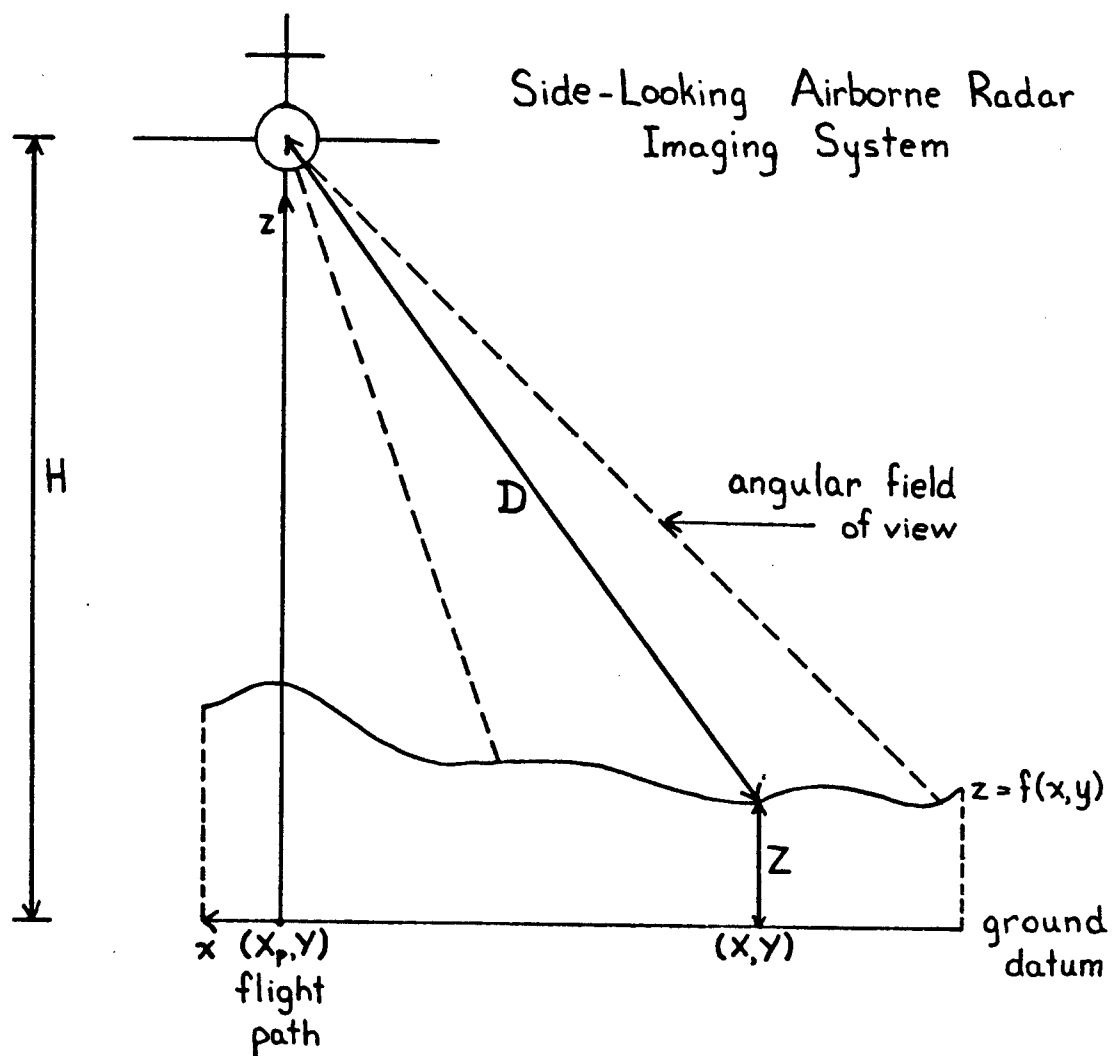


Figure 16. Geometry of a side-looking radar system.

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 (X_a, Y) and (X_b, Y) , assume terrain elevations and pulse travel distances of $Z_a, 2 \cdot D_a$ and $Z_b, 2 \cdot D_b$ respectively. Pixel layover therefore occurs when

$$D_a = D_b \quad (4.6)$$

or, in more detail, when

$$\sqrt{(X_p - X_a)^2 + (H - Z_a)^2} = \sqrt{(X_p - X_b)^2 + (H - Z_b)^2} \quad (4.7)$$

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

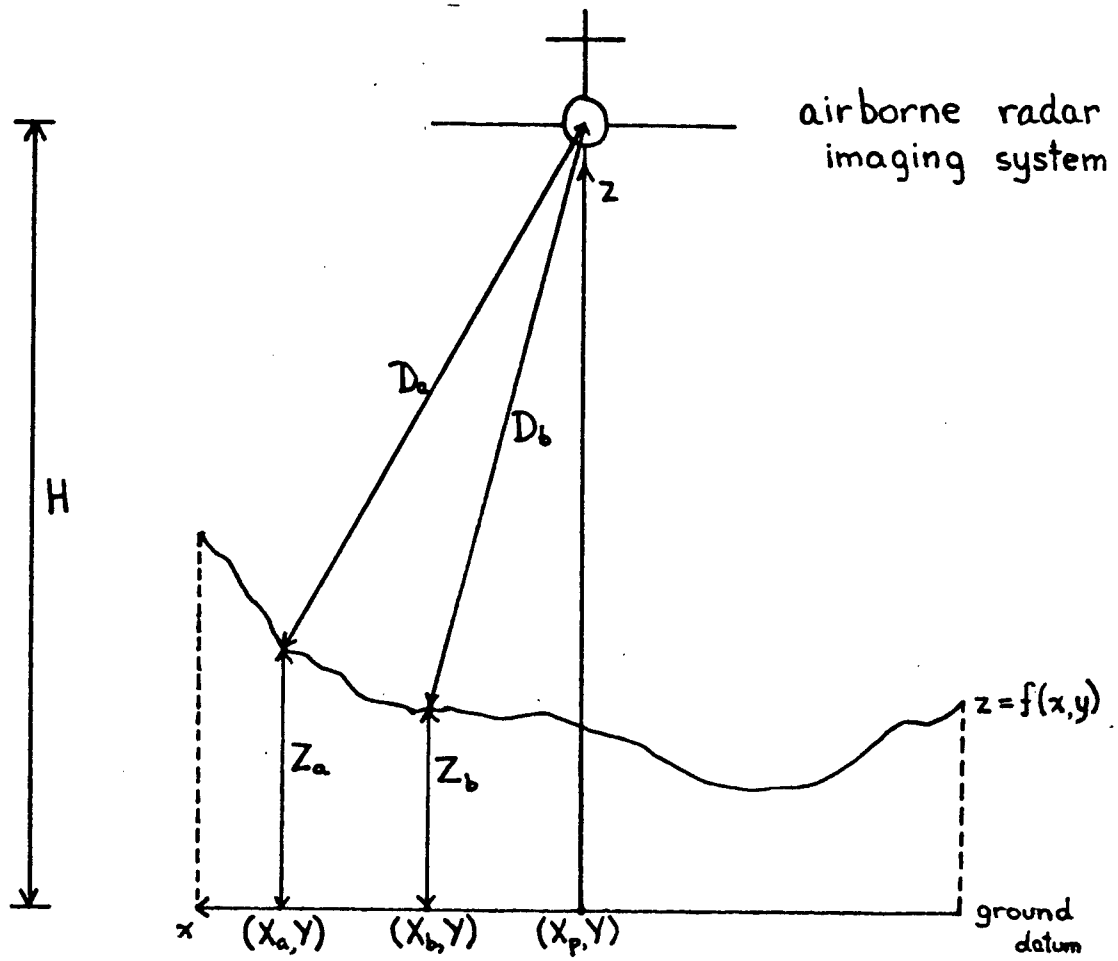


Figure 17. Pixel layover - radar.

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 x_p 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

$$(X_p - X_a)^2 + (H - Z_a)^2 = (X_p - X_b)^2 + (H - Z_b)^2 \quad (4.8)$$

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 in UBC PASCAL [JOLL79].

```

1  (* ##### *)
2  (* *)
3  (* SYNTHETIC ORTHOGRAPHIC STEREO PAIRS *)
4  (* *)
5  (* With input of a DTM and a corresponding registered image, an *)
6  (* orthographic stereo pair of that image is created. Resampling is *)
7  (* done to remove quantization effects. *)
8  (* *)
9  (* ##### *)
10
11  const
12      LINE = 1;      PIXEL = 2;
13      MAX_LINE_LEN = 255;
14      PROMPT = ':';
15      COMPRESSED = true;
16      INTEGRAL = true;
17      READ_REQD = true;
18      MIN_INTENSITY = 0;
19      MAX_INTENSITY = 255;
20      MAX_IMAGE_SIZE = 256;
21      INIT_VALUE = -1;
22      DEFAULT = -1;
23      DEG_TO_RAD = 0.0174532925;
24
25
26  type
27      MSG_TYPE = ( LINE_PROMPT, PIXEL_PROMPT, UNEXPECTED_BLANK,
28                  UNEXPECTED_EOF, UNREC_CMD, NULL_NO, NON_INTEGRAL_VALUE,
29                  UNIT_0_UNASSIGNED, INVALID_FILE, UNEXPECTED_INPUT_EOF,
30                  INVALID_NO, EOF_NO, EOLN_MSG, NOT_IN_RANGE,
31                  ENTER_GRID_SPACING, ENTER_CNTR_COL,
32                  UNIT_1_UNASSIGNED, ENTER_ANG_CONV );
33      INPUT_LINE = array ( 1..MAX_LINE_LEN+1 ) of char;
34      INT_LINE = array ( 0..MAX_LINE_LEN+2 ) of int;
35      REAL_LINE = array ( 0..MAX_LINE_LEN+2 ) of real;
36
37
38      VALUE_RANGE = ( POSITIVE, NON_ZERO, INTENSITY, LINE_LENGTH,
39                     IMAGE_SIZE, NON_NEGATIVE, PERCENT );
40
41      STR_3 = array ( 1..3 ) of char;
42      STR_4 = array ( 1..4 ) of char;
43      STR_6 = array ( 1..6 ) of char;
44      STR_8 = array ( 1..8 ) of char;
45      STR_44 = array ( 1..44 ) of char;
46      STR_50 = array ( 1..50 ) of char;
47
48      IMAGEFILE = file of INPUT_LINE;
49      DTMPFILE = file of array ( 1..(MAX_LINE_LEN+1)*2 ) of char;
50
51
52
53
54  var
55      DIM : array ( 1..2 ) of integer;
56      IMAGE_FILE,
57      LFILE, RFILE : IMAGEFILE;
58      DTM_FILE : DTMPFILE;

```

```

59      GRID_SPAC,
60      CONVERGENCE      : real;
61      CENTER_COL      : integer;
62
63
64
65

```

```

66      (* ##### FORWARD declarations for the Procedures of the system ##### *)
67      (* *)
68      (* *)
69      (* *)
70      (* ##### *)
71      procedure DISPLACE;
72      procedure FILES;
73      procedure GET_DIMENSIONS;
74      procedure INTERPOLATE( var ILINE : INT_LINE;
75                             var FACT : REAL_LINE );
76      procedure PRINT_MSG( MSG : MSG_TYPE; STRING : INPUT_LINE;
77                          SIZE : integer );
78      procedure RESAMPLE( X : real; I : integer;
79                          var ILINE : INT_LINE;
80                          var FACT : REAL_LINE );
81      procedure SYS_INIT;
82      procedure SYSTEM_ERROR( MESSAGE : STR_50 );
83      procedure WRITE_IMAGE_LINE( var IMAGE : IMAGEFILE;
84                                 LINE : INT_LINE;
85                                 START : integer );
86      procedure WRITE_STRING( STR : INPUT_LINE; SIZE : integer );
87      (* *)
88

```

```

89      (* ##### FORWARD declarations for the Functions of the system ##### *)
90      (* *)
91      (* *)
92      (* *)
93      (* ##### *)
94      function GET_EOLN( var STR : INPUT_LINE; COMPRESS : boolean )
95      : integer;
96      function GET_NUM( RANGE : VALUE_RANGE; INTEGRAL_VALUE : boolean )
97      : real;
98      function READ_DTM_LINE( var DTM_FILE : DTMFILE; READ_REQD : boolean )
99      : INT_LINE;
100     function READ_IMAGE_LINE( var IMAGE_FILE : IMAGEFILE;
101                              READ_REQD : boolean ) : INT_LINE;
102     function READ_STRING : INPUT_LINE;
103
104

```

```

105      (* ##### Declarations for FORTRAN routines used by the system ##### *)
106      (* *)
107      (* *)
108      (* *)
109      (* ##### *)
110      procedure CALL_MTS( STR : INPUT_LINE; I : integer );
111      procedure DEFAULT_FNAME( CMD_STR : INPUT_LINE );
112      procedure GET_FNAME( CMD_STR : INPUT_LINE; LEN : integer;
113                          var FNAME : INPUT_LINE );
114      procedure SET_PREFIX( NEW : char; LENGTH : integer );
115      (* *)
116

```



```

117
118 (* ##### *)
119 (* *)
120 (* Introduce displacment into the original image to create two new *)
121 (* images - the left and right images of the stereo pair. The amount *)
122 (* of displacment introduced at a particular pixel is a function of *)
123 (* the relative elevation at that pixel, the location of the hypo- *)
124 (* thetical scanning device, and the angle of convergence of that *)
125 (* device. A one dimensional bilinear resampling process is performed *)
126 (* to create the new images. *)
127 (* *)
128 (* ##### *)
129 procedure DISPLACE;
130
131 var
132     DLINE, ILINE,
133     LLINE, RLINE      : INT_LINE;
134     LFACT, RFACT      : REAL_LINE;
135     I, J, X, XDIFF    : integer;
136     COS_THETA, SIN_THETA,
137     XLPREV, XRPREV,
138     Z, XR, XL         : real;
139
140
141 begin
142
143     COS_THETA := cos( CONVERGENCE );
144     SIN_THETA := sin( CONVERGENCE );
145
146     DLINE := READ_DTM_LINE ( DTM_FILE, not READ_REQD );
147     ILINE := READ_IMAGE_LINE( IMAGE_FILE, not READ_REQD );
148
149     for I := 1 to DIM(LINE) do
150         begin
151
152             for J := 1 to DIM(PIXEL) do
153                 begin
154
155                     LFACT(J) := 0.0;
156                     RFACT(J) := 0.0;
157                     LLINE(J) := MIN_INTENSITY;
158                     RLINE(J) := MIN_INTENSITY;
159
160                 end (* for J *);
161
162
163                 XLPREV := CENTER_COL + SIN_THETA * DLINE(CENTER_COL) / GRID_SPAC;
164                 XRPREV := CENTER_COL - SIN_THETA * DLINE(CENTER_COL) / GRID_SPAC;
165
166                 for X := (CENTER_COL-1) downto 1 do
167                     begin
168
169                         XDIFF := X - CENTER_COL;
170                         Z := DLINE(X) / GRID_SPAC;
171                         XL := CENTER_COL + COS_THETA * XDIFF +
172                             SIN_THETA * Z;
173                         XR := CENTER_COL + COS_THETA * XDIFF -
174                             SIN_THETA * Z;

```

```

175
176      if (XL > 0) and (XL < (DIM(PIXEL)+1)) and (XL < XLPREV) then
177      begin
178          RESAMPLE( XL, ILINE(X), LLINE, LFACT );
179          XLPREV := XL;
180      end (* if (XL > 0) ... *);
181
182      if (XR > 0) and (XR < (DIM(PIXEL)+1)) and (XR < XRPREV) then
183      begin
184          RESAMPLE( XR, ILINE(X), RLINE, RFACT );
185          XRPREV := XR;
186      end (* if (XR > 0) ... *);
187
188
189      end (* for X *);
190
191
192      XLPREV := CENTER_COL - COS_THETA + SIN_THETA * DLINE(CENTER_COL-1) / GRID_SPAC;
193      XRPREV := CENTER_COL - COS_THETA - SIN_THETA * DLINE(CENTER_COL-1) / GRID_SPAC;
194
195      for X := CENTER_COL to DIM(PIXEL) do
196      begin
197
198          XDIFF := X - CENTER_COL;
199          Z      := DLINE(X) / GRID_SPAC;
200          XL     := CENTER_COL + COS_THETA * XDIFF +
201                  SIN_THETA * Z;
202          XR     := CENTER_COL + COS_THETA * XDIFF -
203                  SIN_THETA * Z;
204
205          if (XL > 0) and (XL < (DIM(PIXEL)+1)) and (XL > XLPREV) then
206          begin
207              RESAMPLE( XL, ILINE(X), LLINE, LFACT );
208              XLPREV := XL;
209          end (* if (XL > 0) ... *);
210
211          if (XR > 0) and (XR < (DIM(PIXEL)+1)) and (XR > XRPREV) then
212          begin
213              RESAMPLE( XR, ILINE(X), RLINE, RFACT );
214              XRPREV := XR;
215          end (* if (XR > 0) ... *);
216
217
218          end (* for X *);
219
220
221      INTERPOLATE( LLINE, LFACT );
222      INTERPOLATE( RLINE, RFACT );
223
224      WRITE_IMAGE_LINE( LFILE, LLINE, 1 );
225      WRITE_IMAGE_LINE( RFILE, RLINE, 1 );
226
227      if I < DIM(PIXEL) then
228      begin
229          DLINE := READ_DTM_LINE ( DTM_FILE, READ_REQD );
230          ILINE := READ_IMAGE_LINE( IMAGE_FILE, READ_REQD );
231      end (* if I < DIM(PIXEL) *)
232

```

```

233         end (* for I *);
234
235
236     end (* DISPLACE *);
237
238
239
240 (* ##### *)
241 (* *)
242 (* Set up and attach the files necessary for the system. Unit 0 is the *)
243 (* assumed input for the original DTM. If it is not assigned on the MTS *)
244 (* RUN command, the file name will be prompted for. Unit 1 is assumed *)
245 (* input for the original registered image. Its file name will also be *)
246 (* prompted for if not assigned. The file assigned to *)
247 (* to unit 10 will be used for the output of the left synthetic image and *)
248 (* the file assigned to unit 11 will be used for the output of the right *)
249 (* synthetic image. If these units are not attached, they will default *)
250 (* to MTS temporary files -LEFT# and -RIGHT# respectively. *)
251 (* *)
252 (* ##### *)
253 procedure FILES;
254
255     var
256         DTMFNAME, IMGFNAME : INPUT_LINE;
257
258
259     begin
260
261
262         DTMFNAME := ' ';
263         DEFAULT_FNAME( 'DEFAULT 0=*DUMMY*;' );
264         GET_FNAME( 'QUERY FDNAME 0;', 0, DTMFNAME );
265
266         if DTMFNAME = '*DUMMY*' then
267             begin
268
269                 PRINT_MSG( UNIT_0_UNASSIGNED, '', 0 );
270                 DTMFNAME := READ_STRING;
271                 while DTMFNAME = ' ' do
272                     begin
273                         PRINT_MSG( INVALID_FILE, '', 0 );
274                         DTMFNAME := READ_STRING;
275                         if DTMFNAME = 'CANCEL' or DTMFNAME = 'HALT' then
276                             halt;
277                         end (* while DTMFNAME = ' ' *);
278
279                     end (* if DTMFNAME = '*DUMMY*' *);
280
281
282                 IMGFNAME := ' ';
283                 DEFAULT_FNAME( 'DEFAULT 1=*DUMMY*;' );
284                 GET_FNAME( 'QUERY FDNAME 1;', 0, IMGFNAME );
285
286                 if IMGFNAME = '*DUMMY*' then
287                     begin
288
289                         PRINT_MSG( UNIT_1_UNASSIGNED, '', 0 );
290                         IMGFNAME := READ_STRING;

```

```

291.         while IMGFILENAME = ' ' do
292.             begin
293.                 PRINT_MSG( INVALID_FILE, ' ', 0 );
294.                 IMGFILENAME := READ_STRING;
295.                 if IMGFILENAME = 'CANCEL' or IMGFILENAME = 'HALT' then
296.                     halt;
297.                 end (* while IMGFILENAME = ' ' *);
298.
299.             end (* if IMGFILENAME = '*DUMMY*' *);
300.
301.
302.             DEFAULT_FNAME( 'DEFAULT 10=-LEFT#;' );
303.             DEFAULT_FNAME( 'DEFAULT 11=-RIGHT#;' );
304.             reset( DTM_FILE, DTMFILENAME );
305.             reset( IMAGE_FILE, IMGFILENAME );
306.             rewrite( LFILE, 10 );
307.             rewrite( RFILE, 11 );
308.
309.
310.         end (* FILES *);
311.
312.
313.
314.         (* ##### *)
315.         (* *)
316.         (* Prompt for and get the dimensions of the DTM. *)
317.         (* *)
318.         (* ##### *)
319.         procedure GET_DIMENSIONS;
320.
321.         begin
322.
323.             PRINT_MSG( LINE_PROMPT, ' ', 0 );
324.             DIM(LINE) := trunc( GET_NUM( IMAGE_SIZE, INTEGRAL ) );
325.
326.             PRINT_MSG( PIXEL_PROMPT, ' ', 0 );
327.             DIM(PIXEL) := trunc( GET_NUM( IMAGE_SIZE, INTEGRAL ) );
328.
329.         end (* GET_DIMENSIONS *);
330.
331.
332.
333.         (* ##### *)
334.         (* *)
335.         (* Read and return the end of an input line. If COMPRESS is true then *)
336.         (* remove multiple embedded blanks, otherwise return the actual input *)
337.         (* line. The length of the input line is also returned. *)
338.         (* *)
339.         (* ##### *)
340.         function GET_EOLN( var STR:INPUT_LINE; COMPRESS:boolean ) : integer;
341.
342.         var
343.             I                : integer;
344.             LAST_CHAR_BLANK  : boolean;
345.             CHR               : char;
346.
347.
348.         begin

```

```

349      I := 0;
350      LAST_CHAR_BLANK := false;
351
352      while not eof( INPUT ) and not eoln( INPUT ) do
353      begin
354          read( CHR );
355          if CHR = ' ' or not COMPRESS then
356          begin
357              if LAST_CHAR_BLANK and I>0 then
358              begin
359                  I := I + 1;
360                  STR(I) := ' ';
361              end;
362              I := I + 1;
363              STR(I) := CHR;
364              LAST_CHAR_BLANK := false
365          end;
366          else
367              LAST_CHAR_BLANK := true
368          end;
369
370          if not COMPRESS and I>0 then
371              GET_EOLN := I - 1;
372          else
373              GET_EOLN := I
374          end
375      end (* GET_EOLN *);
376
377
378
379
380      (* ##### *)
381      (* *)
382      (* Return the numerical value of a string in the input stream. If *)
383      (* INTEGRAL_VALUE is true then the string must represent an integer, *)
384      (* otherwise a real number. The numerical value must also be in the *)
385      (* range specified by RANGE. *)
386      (* *)
387      (* ##### *)
388      function GET_NUM( RANGE : VALUE_RANGE;
389                      INTEGRAL_VALUE : boolean ) : real;
390
391      var      STR      : INPUT_LINE;
392              VALID     : boolean;
393              VALU      : real;
394              I, J, MULT : integer;
395
396      begin
397          repeat
398              VALID := true;
399              read( STR );
400              if eof( INPUT ) then      (* CHECK FOR EOF *)
401              begin
402                  VALID := false;
403                  PRINT_MSG( EOF_NO, ' ', 0 )
404              end;
405          else
406              begin

```

```

407      VALU := 0.0;          (* GET INTEGRAL PART          *)
408      I := 1;
409      MULT := 1;
410      while I <= MAX_LINE_LEN and STR(I) = ' ' do
411          I := I + 1;
412      if STR(I) = '+' or STR(I) = '-' then
413          begin
414              if STR(I) = '-' then
415                  MULT := -1;
416              I := I + 1
417          end;
418      while STR(I) in ( '0'..'9' ) do
419          begin
420              VALU := 10.0 * VALU + ord( STR(I) ) - ord( '0' );
421              I := I + 1
422          end;
423      if STR(I) = '.' then
424          begin
425              (* GET FRACTIONAL PART          *)
426              J := 10;
427              I := I + 1;
428              while STR(I) in ( '0'..'9' ) do
429                  begin
430                      VALU := VALU + ( (ord(STR(I)) - ord('0'))/J);
431                      I := I + 1;
432                      J := J * 10
433                  end
434              end;
435          if STR(I) = ' ' or ( eoln(INPUT) and I = MAX_LINE_LEN+1 ) then
436              begin
437                  (* INVALID NUMBER          *)
438                  VALID := false;
439                  if STR(I) = ' ' then
440                      PRINT_MSG( INVALID_NO, STR(I), 1 )
441                  else
442                      if RANGE in ( PERCENT ) then
443                          begin
444                              VALID := true;
445                              MULT := 1;
446                              VALU := DEFAULT;
447                              end (* if RANGE in *);
448                          else
449                              PRINT_MSG( NULL_NO, ' ', 0 )
450                          end
451                      else if INTEGRAL_VALUE and trunc(VALU) = VALU then
452                          begin
453                              VALID := false;
454                              PRINT_MSG( NON_INTEGRAL_VALUE, ' ', 0 )
455                          end
456                      else if STR(I) = ' ' and ( STR(I-1) = '-' or STR(I-1) = '+' ) then
457                          begin
458                              VALID := false;
459                              PRINT_MSG( UNEXPECTED_BLANK, ' ', 0 )
460                          end
461                      else if ( RANGE = POSITIVE and (MULT < 0 or VALU = 0.0)) or
462                          ( RANGE = IMAGE_SIZE and (MULT < 0 or
463                          VALU < 1 or VALU > MAX_IMAGE_SIZE)) or
464                          ( RANGE = LINE_LENGTH and (MULT < 0 or
465                          VALU < 1 or VALU > DIM(PIXEL))) or
466                          ( RANGE = INTENSITY and (MULT < 0 or

```

```

465 VALU < MIN_INTENSITY or
466 VALU > MAX_INTENSITY)) or
467 (RANGE = NON_NEGATIVE and MULT < 0) or
468 (RANGE = PERCENT and (MULT < 0 or
469 VALU < 0 or VALU > 100)) or
470 (RANGE = NON_ZERO and VALU = 0.0) then
471 begin
472 VALID := false;
473 PRINT_MSG( NOT_IN_RANGE, '', 0 )
474 end;
475 end;
476 until VALID;
477 GET_NUM := VALU * MULT
478 end (* GET_NUM *);
479
480
481
482 (* ##### *)
483 (* *)
484 (* Interpolate all undefined pixel values between the first defined *)
485 (* pixel of the image line - ILINE - and the last defined pixel of the *)
486 (* image line. The new values are calculated as a function of the *)
487 (* values of the nearest defined pixels (i.e. the ones on either side *)
488 (* of the undefined pixel ). *)
489 (* *)
490 (* ##### *)
491 procedure INTERPOLATE( var ILINE : INT_LINE; var FACT : REAL_LINE );
492
493 var
494 X, XX, XEND, START,
495 STOP, INCREMENT, I : Integer;
496
497
498 begin
499
500 X := 1;
501 while FACT(X) = 0 and X < DIM(PIXEL) do
502 incr( X );
503
504 XEND := DIM(PIXEL);
505 while FACT(XEND) = 0 and XEND >= 1 do
506 decr( XEND );
507
508 incr( X );
509 repeat
510
511 if FACT(X) = 0 then
512 begin
513
514 I := ILINE(X-1);
515 START := X;
516 while FACT(X) = 0 do
517 incr( X );
518 STOP := X - 1;
519 INCREMENT := round( (ILINE(X) - ILINE(START-1)) / (X - START + 1) );
520 for XX := START to STOP do
521 begin
522 I := I + INCREMENT;

```

```

523          ILINE(XX) := round( (I + ILINE(XX) * FACT(XX)) /
524                               (1.0 + FACT(XX)) );
525          end (* for XX *);
526
527          end (* if FACT(X) < 1 *);
528          incr( X );
529
530      until X >= XEND;
531
532
533      end (* INTERPOLATE *);
534
535
536
537      (* ##### *)
538      (* *)
539      (* Print the message specified by MSG to the output stream. The *)
540      (* parameters STRING and SIZE may be involved in the details of *)
541      (* that message. *)
542      (* *)
543      (* ##### *)
544      procedure PRINT_MSG( MSG : MSG_TYPE; STRING : INPUT_LINE;
545                          SIZE : integer );
546
547      const
548          MSG_PROMPT = ':';
549
550      var
551          I,J          : integer;
552          STR           : INPUT_LINE;
553
554
555      begin
556
557          SET_PREFIX( MSG_PROMPT, 1 );
558
559          if MSG in ( LINE_PROMPT, UNIT_O_UNASSIGNED,
560                    UNIT_I_UNASSIGNED, ENTER_GRID_SPACING ) then
561              writeln;
562
563          case MSG of
564              UNEXPECTED_EOF :
565                  writeln(' Unexpected EOF -- Ignored');
566              UNREC_CMD      :
567                  begin
568                      writeln(' Invalid command -- Input line ignored');
569                      if not eoln( INPUT ) then
570                          I := GET_EOLN( STR, COMPRESSED )
571                      end;
572              EOLN_MSG       :
573                  begin
574                      I := GET_EOLN( STR, COMPRESSED );
575                      if I /= 0 then
576                          begin
577                              write(' Invalid character(s): ');
578                              WRITE_STRING( STR, I );
579                              writeln(' -- Ignored')
580                          end

```



```

581         end;
582     LINE_PROMPT :
583     writeIn('Enter number of lines of DTM and IMAGE (Integer {1-256})');
584     PIXEL_PROMPT :
585     writeIn('&Enter number of pixels per line          (Integer {1-256})');
586     EOF_NO :
587     writeIn('&Unexpected EOF, Enter number');
588     NULL_NO :
589     writeIn(' &Not expecting null line -- Ignored; Enter number' );
590     NON_INTEGRAL_VALUE:
591     writeIn('&Non-integral value, Re-enter number');
592     UNEXPECTED_BLANK:
593     writeIn('&Unexpected blank, Re-enter number');
594     NOT_IN_RANGE :
595     writeIn('&Out of range, Re-enter number');
596     UNIT_O_UNASSIGNED:
597     writeIn(' Logical unit 0 has not been assigned.',
598             EOL, '& Enter FDname of location of DTM ');
599     UNIT_1_UNASSIGNED:
600     writeIn(' Logical unit 1 has not been assigned.',
601             EOL, '& Enter FDname of location of IMAGE ');
602     INVALID_FILE :
603     writeIn(' Invalid FDname. Enter again or', EOL,
604             '&Enter "CANCEL" to terminate program');
605     UNEXPECTED_INPUT_EOF:
606     writeIn(' Unexpected EOF encountered on input',
607             ' -- DTM size updated');
608     INVALID_NO :
609     begin
610         writeIn('&Invalid character, "', STRING(1), '"', Re-enter number');
611         I := GET_EOLN( STR, COMPRESSED )
612     end;
613     ENTER_GRID_SPACING:
614     writeIn('&Enter DTM grid spacing (positive Real)');
615     ENTER_CNTR_COL :
616     writeIn('&Enter centre column for projection (Integer {1-',
617             DIM(PIXEL):0,'})');
618     ENTER_ANG_CONV :
619     writeIn('&Enter angle of convergence (in positive degrees)');
620
621 end (* case *);
622
623 if MSG in ( LINE_PROMPT, PIXEL_PROMPT, INVALID_NO,
624            NON_INTEGRAL_VALUE, UNEXPECTED_BLANK, NOT_IN_RANGE,
625            UNIT_O_UNASSIGNED, UNIT_1_UNASSIGNED, INVALID_FILE,
626            ENTER_GRID_SPACING, ENTER_CNTR_COL, ENTER_ANG_CONV,
627            EOF_NO, NULL_NO ) then
628     readln;
629
630     SET_PREFIX( PROMPT, 1 )
631
632
633 end (* PRINT_MSG *);
634
635
636
637 (* ##### *)
638 (* ##### *)

```

```

639 (* If READ_READ is true then read in a line from the file DTM_FILE. *)
640 (* otherwise perform the subsequent operations on the present input *)
641 (* buffer. Convert the binary form of the buffer into integer form *)
642 (* assuming two bytes per pixel. *)
643 (* ##### *)
644 (* ##### *)
645 function READ_DTM_LINE( var DTM_FILE : DTMFILE; READ_READ : boolean ) : INT_LINE;
646
647 var
648     LINE      : INT_LINE;
649     I         : integer;
650
651 begin
652
653     if READ_READ then
654         get( DTM_FILE );
655     for I := 1 to DIM(PIXEL) do
656         LINE(I) := int( DTM_FILE@((I-1)*2 + 1) ) * 256 +
657                     int( DTM_FILE@((I-1)*2 + 2) );
658     READ_DTM_LINE := LINE;
659
660
661 end (* READ_DTM_LINE *);
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
(* ##### *)
(* ##### *)
(* If READ_READ 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_READ : boolean ) : INT_LINE;

var
    LINE      : INT_LINE;
    I         : integer;
begin
    if READ_READ 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 *);
(* ##### *)
(* ##### *)

```

```

697      (* Read a string from the input stream removing all leading blanks. *)
698      (* The string is then returned. *)
699      (* *)
700      (* ##### *)
701      function READ_STRING : INPUT_LINE;
702
703      var
704          I          : integer;
705          STR         : INPUT_LINE;
706
707
708      begin
709
710          STR := ' ';
711          I := 0;
712          repeat
713              incr( I );
714              read( STR(I) );
715              if STR(I) = ' ' and I = 1 then
716                  I := 0;
717              until eof(INPUT) or eoln(INPUT) or (I /= 0 and STR(I) = ' ')
718                  or I = MAX_LINE_LEN;
719
720          READ_STRING := STR;
721
722
723      end      (* READ_STRING *);
724
725
726
727
728      (* ##### *)
729      (* *)
730      (* Resample using a simple one dimensional bilinear scheme. The data *)
731      (* to be resampled is intensity I at x-location X. The results are *)
732      (* inserted in the image line ILINE. *)
733      (* *)
734      (* ##### *)
735      procedure RESAMPLE( X : real; I : integer; var ILINE : INT_LINE;
736                          var FACT : REAL_LINE );
737
738      var
739          LO, HI          : integer;
740          LO_FACT, HI_FACT,
741          LO_I, HI_I      : real;
742
743
744      begin
745
746          LO := trunc( X );
747          HI := LO + 1;
748          LO_FACT := HI - X;
749          HI_FACT := X - LO;
750
751          if (LO > 0) and (LO_FACT <> 0) then
752              begin
753
754                  LO_I      := ILINE(LO) * FACT(LO) + I * LO_FACT;

```

```

755      FACT(LO) := FACT(LO) + LO_FACT;
756      ILINE(LO) := round( LO_I / FACT(LO) );
757
758      end (* if (LO > 0) ... *);
759
760      if (HI <= DIM(PIXEL)) and (HI_FACT <> 0) then
761      begin
762
763          HI_I := ILINE(HI) * FACT(HI) + I * HI_FACT;
764          FACT(HI) := FACT(HI) + HI_FACT;
765          ILINE(HI) := round( HI_I / FACT(HI) );
766
767          end (* if (HI <= DIM(PIXEL)) ... *);
768
769
770      end (* RESAMPLE *);
771
772
773
774      (* ##### *)
775      (* *)
776      (* Initialize the system by setting up the files, getting the dimensions *)
777      (* of the DTM, and reading the necessary system parameters. *)
778      (* *)
779      (* ##### *)
780      procedure SYS_INIT;
781
782      var
783          I, J : integer;
784
785
786      begin
787
788          SET_PREFIX( PROMPT, 1 );
789          FILES;
790          GET_DIMENSIONS;
791
792          PRINT_MSG( ENTER_GRID_SPACING, '', 0 );
793          GRID_SPAC := GET_NUM( POSITIVE, not INTEGRAL );
794
795          PRINT_MSG( ENTER_CNTR_COL, '', 0 );
796          CENTER_COL := trunc( GET_NUM( LINE_LENGTH, INTEGRAL ) );
797
798          PRINT_MSG( ENTER_ANG_CONV, '', 0 );
799          CONVERGENCE := GET_NUM( NON_NEGATIVE, not INTEGRAL ) *
800              DEG_TO_RAD;
801
802
803      end (* SYSTEM_INITIALIZATION *);
804
805
806      (* ##### *)
807      (* *)
808      (* If a system error occurs, output the message MESSAGE and halt the *)
809      (* execution of the system. *)
810      (* *)
811      (* *)
812      (* ##### *)

```

```

813      procedure SYSTEM_ERROR( MESSAGE : STR_50 );
814
815          begin
816
817              writeln( ' ==> SYSTEM ERROR ==> ', MESSAGE );
818              halt;
819
820
821          end (* SYSTEM_ERROR *);
822
823
824
825
826      (* ##### *)
827      (* *)
828      (* Write the line LINE to the file IMAGE starting at pixel START. *)
829      (* It is assumed that each pixel has a value between 0 and 255. *)
830      (* *)
831      (* ##### *)
832      procedure WRITE_IMAGE_LINE( var IMAGE : IMAGEFILE;
833                                  LINE : INT_LINE; START : integer );
834
835          var
836              I          : integer;
837
838
839          begin
840
841              IMAGE@ := ' ';
842
843              for I := 1 to (START-1) do
844                  IMAGE@ (I) := char( MIN_INTENSITY );
845
846              for I := START to DIM( PIXEL ) do
847                  IMAGE@ (I) := char( LINE(I) );
848
849              put( IMAGE );
850
851
852          end (* WRITE_IMAGE_LINE *);
853
854
855
856      (* ##### *)
857      (* *)
858      (* Write the string STR to the present output stream. The string is of *)
859      (* length SIZE. If the length is less than one, the operation does not *)
860      (* take place. *)
861      (* *)
862      (* ##### *)
863      procedure WRITE_STRING( STR : INPUT_LINE; SIZE : integer );
864
865          var
866              I          : integer;
867
868
869          begin
870

```

```
871         if SIZE > 0 then
872             for I := 1 to SIZE do
873                 write( STR(I) )
874
875
876         end      (* WRITE_STRING *);
877
878
879
880     (* ##### *)
881     (* *)
882     (* Initialize the system and then proceed to displace the original *)
883     (* image as a function of its relative height (as specified by the DTM). *)
884     (* *)
885     (* ##### *)
886     begin      (* MAIN ROUTINE *)
887
888         SYS_INIT;
889         DISPLACE;
890
891
892     end      (* MAIN ROUTINE *).
```

Appendix B

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

```

1  (* ##### *)
2  (* *)
3  (* 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 *)
4  (* *)
5  (* With input of a DTM and a corresponding registered image, a *)
6  (* stereo pair of perspective images of the DTM, according to the *)
7  (* intensity values of the inputted image, are created as well as two *)
8  (* corresponding visibility maps for the DTM. *)
9  (* *)
10 (* ##### *)
11
12     const
13         LINE   = 1;    PIXEL = 2;
14         OCT_1  = 1;    OCT_2 = 2;
15         OCT_3  = 3;    OCT_4 = 4;
16         OCT_5  = 5;    OCT_6 = 6;
17         OCT_7  = 7;    OCT_8 = 8;
18         RDS    = 1;    THT   = 2;
19         X       = 1;    Y     = 2;
20         LEFT   = 1;    RIGHT = 2;
21         DTM    = 1;    IMGE  = 2;
22
23         MAX_LINE_LEN      = 255;
24         PROMPT             = ':';
25         COMPRESSED         = true;
26         INTEGRAL           = true;
27         READ_REQD         = true;
28         RESET_REQD        = true;
29         MIN_INTENSITY      = 0;
30         MAX_INTENSITY      = 255;
31         MAX_IMAGE_SIZE     = 256;
32         MAX_HORZN_SIZE     = max( 360, MAX_IMAGE_SIZE+1 );
33         WORDS_PER_ENTRY    = 1;
34         BYTES_PER_WORD     = 4;
35         UNDEFINED          = -1;
36         DEFAULT            = -1;
37         DEG_TO_RAD         = 0.0174532925;
38
39
40     type
41         MSG_TYPE = ( LINE_PROMPT, PIXEL_PROMPT, UNEXPECTED_BLANK,
42                     UNEXPECTED_EOF, UNREC_CMD, NULL_NO, NON_INTEGRAL_VALUE,
43                     UNIT_0_UNASSIGNED, INVALID_FILE, UNEXPECTED_INPUT_EOF,
44                     INVALID_NO, EOF_NO, EOLN_MSG, NOT_IN_RANGE,
45                     ENTER_ELEVATION, ENTER_CENTER_X_COORD,
46                     ENTER_GRID_SPACING, ENTER_FOCAL_LENGTH,
47                     ENTER_PRINT_SIZE, PRINT_PRINT_SIZE,
48                     UNIT_1_UNASSIGNED, ENTER_CENTER_Y_COORD );
49
50         INPUT_LINE = array( 1..MAX_LINE_LEN+1 ) of char;
51         INT_LINE   = array( 0..MAX_LINE_LEN+2 ) of int;
52         REAL_LINE  = array( 0..MAX_LINE_LEN+2 ) of real;
53         SHORT_LINE = array( 0..MAX_LINE_LEN+2 ) of short;
54
55         INT_ARRAY  = array( 0..MAX_LINE_LEN+2 ) of INT_LINE;
56         SHORT_ARRAY = array( 0..MAX_LINE_LEN+2 ) of SHORT_LINE;
57
58         HORZN_LINE = array( 0..MAX_HORZN_SIZE ) of short;

```



```

59
60     VALUE_RANGE = ( POSITIVE, NON_ZERO, INTENSITY, LINE_LENGTH,
61                     IMAGE_SIZE, NON_NEGATIVE, PERCENT, LINES );
62     PIXEL_RANGE = 1..MAX_IMAGE_SIZE;
63     MAPSET      = set of PIXEL_RANGE;
64
65     STR_3       = array ( . 1..3 .) of char;
66     STR_4       = array ( . 1..4 .) of char;
67     STR_6       = array ( . 1..6 .) of char;
68     STR_8       = array ( . 1..8 .) of char;
69     STR_44      = array ( . 1..44 .) of char;
70     STR_50      = array ( . 1..50 .) of char;
71
72     IMAGEFILE   = file of INPUT_LINE;
73     DTMFILE     = file of array( 1..(MAX_LINE_LEN+1)*2 ) of char;
74     MAPFILE     = file of MAPSET;
75
76     POINTER     = @ short;
77     POLAR_COORD = array(. 1..2 .) of short;
78     XY_COORD    = array(. 1..2 .) of short;
79
80
81
82
83     var
84         FULL_IMAGE_SIZE,
85         HALF_IMAGE_SIZE      : integer;
86         MAX_HALF             : int;
87         PRINT_SIZE,
88         FOCAL_LENGTH,
89         H, GRID_SPAC         : short;
90         DIM                  : array(. 1..2 .) of integer;
91         CENTER               : array(. LEFT..RIGHT, X..Y .) of int;
92         XY_LOOKUP            : array(. 0..360, X..Y .) of short;
93         LFILE, RFILE,
94         IMAGE_FILE           : IMAGEFILE;
95         DTM_FILE             : DTMFILE;
96         LMFILE, RMFILE       : MAPFILE;
97         THETA_LOOKUP,
98         RADIUS_LOOKUP        : POINTER;
99         IMAGE                 : INT_ARRAY;
100
101
102
103
104     (* ##### *)
105     (* ##### *)
106     (* FORWARD declarations for the Procedures of the system *)
107     (* ##### *)
108     (* ##### *)
109     procedure CIRCULAR_DISPLACE( Y_COORD, X_CENTER, Y_CENTER,
110                                 START, STOP      : integer;
111                                 var IMAGE         : INT_ARRAY;
112                                 var CURR_ILINE, CURR_DLINE,
113                                 PREV_DLINE        : INT_LINE;
114                                 var MAP_LINE      : MAPSET;
115                                 var HORI_RAD, HORI_ALT : HORZN_LINE );
116                                 forward;

```

```

117 procedure CUBIC_CONVOLUTION( var IMAGE : INT_ARRAY;
118                               X, Y : integer ); forward;
119 procedure DISPLACE( var IMAGE : INT_ARRAY;
120                     var MFILE : MAPFILE;
121                     X_CENTER, Y_CENTER : integer ); forward;
122 procedure DISPLCE_CENTER_LINE( X_CENTER, Y_CENTER : integer;
123                               var IMAGE : INT_ARRAY;
124                               var ILINE, DLINE : INT_LINE;
125                               var MLINE : MAPSET;
126                               var HORI_RAD, HORI_ALT : HORZN_LINE );
127                               forward;
128 procedure EXPANDING_DISPLACE( Y_COORD,
129                               X_CENTER, Y_CENTER : integer;
130                               var IMAGE : INT_ARRAY;
131                               var ILINE, DLINE : INT_LINE;
132                               var MLINE : MAPSET;
133                               var HORI_RAD, HORI_ALT : HORZN_LINE );
134                               forward;
135 procedure EXPNDNG_HORIZON_UPDATE( Y, X_CENTER,
136                                   PREV_START, PREV_STOP,
137                                   PREV_Y : integer;
138                                   var HORI_RAD, HORI_ALT : HORZN_LINE );
139                                   forward;
140 procedure FILES; forward;
141 procedure GET_DIMENSIONS; forward;
142 procedure HORIZON_UPDATE( CURR_POLAR, PREV_POLAR : POLAR_COORD;
143                           CURR_ALT, PREV_ALT : int;
144                           var HORI_RAD, HORI_ALT : HORZN_LINE ); forward;
145 procedure INTERPOLATE( var IMAGE : INT_ARRAY ); forward;
146 procedure INSERT_PIXEL( var IMAGE : INT_ARRAY;
147                          Y, X : short;
148                          INTEN : integer ); forward;
149 procedure OCT_1_TRANSFORM( var X, Y, OCT : integer ); forward;
150 procedure POLAR_INIT( SIZE : integer ); forward;
151 procedure PRINT_MSG( MSG : MSG_TYPE; STRING : INPUT_LINE;
152                     SIZE : integer ); forward;
153 procedure PUT_ARRAY( ARR : POINTER; X, Y : integer;
154                     VALU : short ); forward;
155 procedure SYS_INIT; forward;
156 procedure SYSTEM_ERROR( MESSAGE : STR_50 ); forward;
157 procedure WRITE_IMAGE_LINE( var IMAGE : IMAGEFILE;
158                             LINE : INT_LINE;
159                             START : integer ); forward;
160 procedure WRITE_MAP_LINE( var MAP : MAPFILE;
161                           LINE : MAPSET;
162                           LINE_NO : integer ); forward;
163 procedure WRITE_STRING( STR : INPUT_LINE; SIZE : integer ); forward;
164 procedure WRT_IMAGE( var IMAGE_FILE : IMAGEFILE;
165                     var IMAGE : INT_ARRAY ); forward;
166 procedure XY_INIT; forward;
167
168
169 (* ##### *)
170 (* ##### *)
171 (* FORWARD declarations for the Functions of the system *)
172 (* ##### *)
173 (* ##### *)
174 function ALLOCATE_SPACE( SIZE : integer ) : POINTER; forward;

```

LISTING OF FILE LIFE:persp.ster.s 12:54 P.M. JUNE 12, 1982 ID=LIFE

```

175 function GET_ARRAY( ARR : POINTER; X, Y : integer ) : short; forward;
176 function GET_EOLN( var STR : INPUT_LINE; COMPRESS : boolean )
177       : integer;
178       forward;
179 function GET_NUM( RANGE : VALUE_RANGE; INTEGRAL_VALUE : boolean )
180       : real;
181       forward;
182 function POLAR_TRANSFORM( X, Y : integer ) : POLAR_COORD;
183       forward;
184 function READ_DTM_LINE( var DTM_FILE : DTMFILE; READ_REQD : boolean )
185       : integer;
186       forward;
187 function READ_IMAGE_LINE( var IMAGE_FILE : IMAGEFILE;
188       READ_REQD : boolean ) : INT_LINE;
189       forward;
190 function READ_LINE( TYP, LINE_NO : integer; RESET_REQD : boolean )
191       : integer;
192       forward;
193 function READ_STRING : INPUT_LINE;
194       forward;
195 function VISIBLE( ALT1, ALT2, RAD1, RAD2 : short ) : boolean;
196       forward;
197 function XY_TRANSFORM( RADIUS, THETA : short ) : XY_COORD;
198       forward;
199
200 *****
201 ***** Declarations for FORTRAN routines used by the system *****
202 *****
203
204 *****
205 *****
206 *****
207 *****
208 *****
209 *****
210 ***** Allocate space of SIZE words of memory and return starting location. *****
211 *****
212 *****
213 ***** function ALLOCATE_SPACE( SIZE : integer ) : POINTER; *****
214 *****
215
216 var
217     PTR          : POINTER;
218
219 begin
220
221     getspace( 0, WORDS_PER_ENTRY * BYTES_PER_WORD * SIZE,
222             0, PTR );
223     if PTR = nil then
224         SYSTEM_ERROR( ' In "ALLOCATE_SPACE": Unsuccessful allocate' );
225     ALLOCATE_SPACE := PTR;
226
227
228
229
230
231     end (* ALLOCATE_SPACE *);
232
233 *****
234 *****
235 *****
236 *****
237 *****
238 *****
239 *****
240 *****
241 *****
242 *****
243 *****
244 *****
245 *****
246 *****
247 *****
248 *****
249 *****
250 *****
251 *****
252 *****
253 *****
254 *****
255 *****
256 *****
257 *****
258 *****
259 *****
260 *****
261 *****
262 *****
263 *****
264 *****
265 *****
266 *****
267 *****
268 *****
269 *****
270 *****
271 *****
272 *****
273 *****
274 *****
275 *****
276 *****
277 *****
278 *****
279 *****
280 *****
281 *****
282 *****
283 *****
284 *****
285 *****
286 *****
287 *****
288 *****
289 *****
290 *****
291 *****
292 *****
293 *****
294 *****
295 *****
296 *****
297 *****
298 *****
299 *****
300 *****
301 *****
302 *****
303 *****
304 *****
305 *****
306 *****
307 *****
308 *****
309 *****
310 *****
311 *****
312 *****
313 *****
314 *****
315 *****
316 *****
317 *****
318 *****
319 *****
320 *****
321 *****
322 *****
323 *****
324 *****
325 *****
326 *****
327 *****
328 *****
329 *****
330 *****
331 *****
332 *****
333 *****
334 *****
335 *****
336 *****
337 *****
338 *****
339 *****
340 *****
341 *****
342 *****
343 *****
344 *****
345 *****
346 *****
347 *****
348 *****
349 *****
350 *****
351 *****
352 *****
353 *****
354 *****
355 *****
356 *****
357 *****
358 *****
359 *****
360 *****
361 *****
362 *****
363 *****
364 *****
365 *****
366 *****
367 *****
368 *****
369 *****
370 *****
371 *****
372 *****
373 *****
374 *****
375 *****
376 *****
377 *****
378 *****
379 *****
380 *****
381 *****
382 *****
383 *****
384 *****
385 *****
386 *****
387 *****
388 *****
389 *****
390 *****
391 *****
392 *****
393 *****
394 *****
395 *****
396 *****
397 *****
398 *****
399 *****
400 *****
401 *****
402 *****
403 *****
404 *****
405 *****
406 *****
407 *****
408 *****
409 *****
410 *****
411 *****
412 *****
413 *****
414 *****
415 *****
416 *****
417 *****
418 *****
419 *****
420 *****
421 *****
422 *****
423 *****
424 *****
425 *****
426 *****
427 *****
428 *****
429 *****
430 *****
431 *****
432 *****
433 *****
434 *****
435 *****
436 *****
437 *****
438 *****
439 *****
440 *****
441 *****
442 *****
443 *****
444 *****
445 *****
446 *****
447 *****
448 *****
449 *****
450 *****
451 *****
452 *****
453 *****
454 *****
455 *****
456 *****
457 *****
458 *****
459 *****
460 *****
461 *****
462 *****
463 *****
464 *****
465 *****
466 *****
467 *****
468 *****
469 *****
470 *****
471 *****
472 *****
473 *****
474 *****
475 *****
476 *****
477 *****
478 *****
479 *****
480 *****
481 *****
482 *****
483 *****
484 *****
485 *****
486 *****
487 *****
488 *****
489 *****
490 *****
491 *****
492 *****
493 *****
494 *****
495 *****
496 *****
497 *****
498 *****
499 *****
500 *****
501 *****
502 *****
503 *****
504 *****
505 *****
506 *****
507 *****
508 *****
509 *****
510 *****
511 *****
512 *****
513 *****
514 *****
515 *****
516 *****
517 *****
518 *****
519 *****
520 *****
521 *****
522 *****
523 *****
524 *****
525 *****
526 *****
527 *****
528 *****
529 *****
530 *****
531 *****
532 *****
533 *****
534 *****
535 *****
536 *****
537 *****
538 *****
539 *****
540 *****
541 *****
542 *****
543 *****
544 *****
545 *****
546 *****
547 *****
548 *****
549 *****
550 *****
551 *****
552 *****
553 *****
554 *****
555 *****
556 *****
557 *****
558 *****
559 *****
560 *****
561 *****
562 *****
563 *****
564 *****
565 *****
566 *****
567 *****
568 *****
569 *****
570 *****
571 *****
572 *****
573 *****
574 *****
575 *****
576 *****
577 *****
578 *****
579 *****
580 *****
581 *****
582 *****
583 *****
584 *****
585 *****
586 *****
587 *****
588 *****
589 *****
590 *****
591 *****
592 *****
593 *****
594 *****
595 *****
596 *****
597 *****
598 *****
599 *****
600 *****
601 *****
602 *****
603 *****
604 *****
605 *****
606 *****
607 *****
608 *****
609 *****
610 *****
611 *****
612 *****
613 *****
614 *****
615 *****
616 *****
617 *****
618 *****
619 *****
620 *****
621 *****
622 *****
623 *****
624 *****
625 *****
626 *****
627 *****
628 *****
629 *****
630 *****
631 *****
632 *****
633 *****
634 *****
635 *****
636 *****
637 *****
638 *****
639 *****
640 *****
641 *****
642 *****
643 *****
644 *****
645 *****
646 *****
647 *****
648 *****
649 *****
650 *****
651 *****
652 *****
653 *****
654 *****
655 *****
656 *****
657 *****
658 *****
659 *****
660 *****
661 *****
662 *****
663 *****
664 *****
665 *****
666 *****
667 *****
668 *****
669 *****
670 *****
671 *****
672 *****
673 *****
674 *****
675 *****
676 *****
677 *****
678 *****
679 *****
680 *****
681 *****
682 *****
683 *****
684 *****
685 *****
686 *****
687 *****
688 *****
689 *****
690 *****
691 *****
692 *****
693 *****
694 *****
695 *****
696 *****
697 *****
698 *****
699 *****
700 *****
701 *****
702 *****
703 *****
704 *****
705 *****
706 *****
707 *****
708 *****
709 *****
710 *****
711 *****
712 *****
713 *****
714 *****
715 *****
716 *****
717 *****
718 *****
719 *****
720 *****
721 *****
722 *****
723 *****
724 *****
725 *****
726 *****
727 *****
728 *****
729 *****
730 *****
731 *****
732 *****
733 *****
734 *****
735 *****
736 *****
737 *****
738 *****
739 *****
740 *****
741 *****
742 *****
743 *****
744 *****
745 *****
746 *****
747 *****
748 *****
749 *****
750 *****
751 *****
752 *****
753 *****
754 *****
755 *****
756 *****
757 *****
758 *****
759 *****
760 *****
761 *****
762 *****
763 *****
764 *****
765 *****
766 *****
767 *****
768 *****
769 *****
770 *****
771 *****
772 *****
773 *****
774 *****
775 *****
776 *****
777 *****
778 *****
779 *****
780 *****
781 *****
782 *****
783 *****
784 *****
785 *****
786 *****
787 *****
788 *****
789 *****
790 *****
791 *****
792 *****
793 *****
794 *****
795 *****
796 *****
797 *****
798 *****
799 *****
800 *****
801 *****
802 *****
803 *****
804 *****
805 *****
806 *****
807 *****
808 *****
809 *****
810 *****
811 *****
812 *****
813 *****
814 *****
815 *****
816 *****
817 *****
818 *****
819 *****
820 *****
821 *****
822 *****
823 *****
824 *****
825 *****
826 *****
827 *****
828 *****
829 *****
830 *****
831 *****
832 *****
833 *****
834 *****
835 *****
836 *****
837 *****
838 *****
839 *****
840 *****
841 *****
842 *****
843 *****
844 *****
845 *****
846 *****
847 *****
848 *****
849 *****
850 *****
851 *****
852 *****
853 *****
854 *****
855 *****
856 *****
857 *****
858 *****
859 *****
860 *****
861 *****
862 *****
863 *****
864 *****
865 *****
866 *****
867 *****
868 *****
869 *****
870 *****
871 *****
872 *****
873 *****
874 *****
875 *****
876 *****
877 *****
878 *****
879 *****
880 *****
881 *****
882 *****
883 *****
884 *****
885 *****
886 *****
887 *****
888 *****
889 *****
890 *****
891 *****
892 *****
893 *****
894 *****
895 *****
896 *****
897 *****
898 *****
899 *****
900 *****
901 *****
902 *****
903 *****
904 *****
905 *****
906 *****
907 *****
908 *****
909 *****
910 *****
911 *****
912 *****
913 *****
914 *****
915 *****
916 *****
917 *****
918 *****
919 *****
920 *****
921 *****
922 *****
923 *****
924 *****
925 *****
926 *****
927 *****
928 *****
929 *****
930 *****
931 *****
932 *****
933 *****
934 *****
935 *****
936 *****
937 *****
938 *****
939 *****
940 *****
941 *****
942 *****
943 *****
944 *****
945 *****
946 *****
947 *****
948 *****
949 *****
950 *****
951 *****
952 *****
953 *****
954 *****
955 *****
956 *****
957 *****
958 *****
959 *****
960 *****
961 *****
962 *****
963 *****
964 *****
965 *****
966 *****
967 *****
968 *****
969 *****
970 *****
971 *****
972 *****
973 *****
974 *****
975 *****
976 *****
977 *****
978 *****
979 *****
980 *****
981 *****
982 *****
983 *****
984 *****
985 *****
986 *****
987 *****
988 *****
989 *****
990 *****
991 *****
992 *****
993 *****
994 *****
995 *****
996 *****
997 *****
998 *****
999 *****
1000 *****

```

```

233      (* *)
234      (* Displace the pixels of CURR_ILINE and calculate terrain visibility *)
235      (* utilizing the circular horizon, HORI_RAD and HORI_ALT. The pixel's XY *)
236      (* (cartesian) coordinates are first transformed to polar coordinates *)
237      (* and then the displacement is introduced. The new polar coordinates *)
238      (* are then converted to XY coordinates and the displaced pixel is *)
239      (* inserted in the perspective image at those coordinates. *)
240      (* *)
241      (* ##### *)
242      procedure CIRCULAR_DISPLACE( Y_COORD, X_CENTER, Y_CENTER,
243                                  START, STOP          : integer;
244                                  var IMAGE             : INT_ARRAY;
245                                  var CURR_ILINE, CURR_DLINE,
246                                  PREV_DLINE           : INT_LINE;
247                                  var MAP_LINE          : MAPSET;
248                                  var HORI_RAD, HORI_ALT : HORZN_LINE );
249
250      var
251          HI, LO, I, J      : integer;
252          D,
253          LO_FACT, HI_FACT,
254          H1, H2, R1, R2, RF : short;
255          SWCH               : boolean;
256          PREV_POLAR,
257          CURR_POLAR         : POLAR_COORD;
258          XY                 : XY_COORD;
259
260
261      begin
262
263          if START > STOP then
264              begin
265                  I := START;
266                  START := STOP;
267                  STOP := I;
268                  SWCH := true;
269              end (* if START > STOP *);
270          else
271              SWCH := false;
272
273          for I := START to STOP do
274              begin
275
276                  if SWCH then
277                      J := STOP + START - I;
278                  else
279                      J := I;
280
281                  CURR_POLAR := POLAR_TRANSFORM( J, Y_COORD );
282                  if Y_COORD > 0 then
283                      PREV_POLAR := POLAR_TRANSFORM( J, Y_COORD-1 );
284                  else
285                      PREV_POLAR := POLAR_TRANSFORM( J, Y_COORD+1 );
286
287                  LO := trunc( CURR_POLAR(THT) );
288                  HI := LO + 1;
289                  LO_FACT := HI - CURR_POLAR(THT);
290                  HI_FACT := 1 - LO_FACT;

```

```

291
292      H1      := CURR_DLINE(J+X_CENTER);
293      RF      := CURR_POLAR(RDS) * FOCAL_LENGTH;
294      D      := RF / (H - H1) - RF / H;
295      R1      := CURR_POLAR(RDS) + D * (H / FOCAL_LENGTH);
296      H2      := LO_FACT * HORI_ALT(LO) + HI_FACT * HORI_ALT(HI);
297      R2      := LO_FACT * HORI_RAD(LO) + HI_FACT * HORI_RAD(HI);
298
299      if not VISIBLE( H1, H2, R1, R2 ) then
300      MAP_LINE := MAP_LINE + ( (MAX_IMAGE_SIZE-(J+X_CENTER)) );
301      else
302      begin
303
304          XY      := XY_TRANSFORM( R1, CURR_POLAR(THT) );
305          XY(X)   := XY(X) + X_CENTER;
306          XY(Y)   := XY(Y) + Y_CENTER;
307          if (XY(X) > max( 0, X_CENTER-HALF_IMAGE_SIZE )) and
308             (XY(Y) > max( 0, Y_CENTER-HALF_IMAGE_SIZE )) and
309             (XY(X) <= min( DIM(PIXEL), X_CENTER+HALF_IMAGE_SIZE )) and
310             (XY(Y) <= min( DIM(LINE), Y_CENTER+HALF_IMAGE_SIZE )) then
311              INSERT_PIXEL( IMAGE, XY(Y), XY(X),
312                           CURR_ILINE(J+X_CENTER) );
313          else
314              MAP_LINE := MAP_LINE + ( (MAX_IMAGE_SIZE-(J+X_CENTER)) );
315
316      end (* else VISIBLE *);
317
318      HORIZON_UPDATE( CURR_POLAR, PREV_POLAR,
319                    CURR_DLINE(J+X_CENTER),
320                    PREV_DLINE(J+X_CENTER),
321                    HORI_RAD, HORI_ALT );
322
323      end (* for I *);
324
325
326      end (* CIRCULAR_DISPLACE *);
327
328
329
330      (* ##### *)
331      (* *)
332      (* Perform a cubic convolution at location (X,Y) of the image IMAGE *)
333      (* to define that pixel. *)
334      (* *)
335      (* ##### *)
336      procedure CUBIC_CONVOLUTION( var IMAGE : INT_ARRAY; X, Y : integer );
337
338      var
339          I, J, TOTAL, NUM : integer;
340
341
342      begin
343
344          TOTAL := 0;
345          NUM := 0;
346
347          for I := X-1 to X+1 do
348              for J := Y-1 to Y+1 do

```

```

349         if IMAGE(I,J) <> UNDEFINED then
350             begin
351                 TOTAL := TOTAL + IMAGE(I,J);
352                 incr( NUM );
353             end (* if IMAGE(I,J) <> UNDEFINED *);
354     if NUM = 0 then
355         IMAGE(X,Y) := MIN_INTENSITY;
356     else
357         IMAGE(X,Y) := round( TOTAL / NUM );
358
359
360     end (* CUBIC_CONVOLUTION *);
361
362
363
364     (* ##### *)
365     (* *)
366     (* Displace the elements of the orthographic image by introducing *)
367     (* relief displacement to all pixels of that image. (This is *)
368     (* accomplished by three routines: DISPLCE_CENTER_LINE which displaces *)
369     (* the center image line (i.e. the line containing the principle point; *)
370     (* CIRCULAR_DISPLACE for pixels whose cartesian coordinates are such *)
371     (* that abs(X) >= abs(Y), assuming the principle point to be the origin; *)
372     (* and EXPANDING_DISPLACE for all other pixels i.e. pixels with *)
373     (* abs(X) < abs(Y).) *)
374     (* *)
375     (* ##### *)
376     procedure DISPLACE( var IMAGE : INT_ARRAY;
377                         var MFILE : MAPFILE;
378                         X_CENTER, Y_CENTER : integer );
379
380     const
381         CURR = 0; PREV = 1;
382
383     var
384         I, J : integer;
385         DLINE : array( CURR..PREV ) of INT_LINE;
386         ILINE_CURR : INT_LINE;
387         MLINE_CURR : MAPSET;
388         EXPD_HORI_RAD,
389         EXPD_HORI_ALT,
390         CIRC_HORI_RAD,
391         CIRC_HORI_ALT : HORZN_LINE;
392
393
394     begin
395
396         for I := 0 to MAX_LINE_LEN+2 do
397             for J := 0 to MAX_LINE_LEN+2 do
398                 IMAGE(I,J) := UNDEFINED;
399
400             for I := 0 to MAX_HORZN_SIZE do
401                 begin
402                     EXPD_HORI_RAD(I) := 0;
403                     EXPD_HORI_ALT(I) := 0;
404                     CIRC_HORI_RAD(I) := 0;
405                     CIRC_HORI_ALT(I) := 0;
406                 end (* for I *);

```

```

407
408     for I := Y_CENTER+1 to DIM(LINE) do
409         begin
410
411             DLINE(PREV) := DLINE(CURR);
412             DLINE(CURR) := READ_LINE( DTM, DIM(LINE)-I+1, RESET_REQD );
413             ILINE_CURR := READ_LINE( IMGE, DIM(LINE)-I+1, RESET_REQD );
414             MLINE_CURR := ( . );
415
416             if (Y_CENTER-I) >= (-X_CENTER+1) then
417                 CIRCULAR_DISPLACE( I-Y_CENTER, X_CENTER, Y_CENTER,
418                                     Y_CENTER-I, -X_CENTER+1, IMAGE,
419                                     ILINE_CURR, DLINE(CURR), DLINE(PREV),
420                                     MLINE_CURR, CIRC_HORI_RAD, CIRC_HORI_ALT );
421             if (I-Y_CENTER) <= (DIM(PIXEL)-X_CENTER) then
422                 CIRCULAR_DISPLACE( I-Y_CENTER, X_CENTER, Y_CENTER,
423                                     I-Y_CENTER, DIM(PIXEL)-X_CENTER, IMAGE,
424                                     ILINE_CURR, DLINE(CURR), DLINE(PREV),
425                                     MLINE_CURR, CIRC_HORI_RAD, CIRC_HORI_ALT );
426             EXPANDING_DISPLACE( I-Y_CENTER, X_CENTER, Y_CENTER,
427                                 IMAGE, ILINE_CURR, DLINE(CURR),
428                                 MLINE_CURR, EXPD_HORI_RAD, EXPD_HORI_ALT );
429
430             WRITE_MAP_LINE( MFILE, MLINE_CURR, DIM(LINE)-I+1 );
431
432         end (* for I *);
433
434     for I := 0 to MAX_HORZN_SIZE do
435         begin
436             EXPD_HORI_RAD(I) := 0;
437             EXPD_HORI_ALT(I) := 0;
438         end (* for I *);
439
440     DLINE(CURR) := READ_LINE( DTM, DIM(LINE)-Y_CENTER+1, RESET_REQD );
441     ILINE_CURR := READ_LINE( IMGE, DIM(LINE)-Y_CENTER+1, RESET_REQD );
442     for I := Y_CENTER-1 downto 1 do
443         begin
444
445             DLINE(PREV) := DLINE(CURR);
446             DLINE(CURR) := READ_LINE( DTM, DIM(LINE)-I+1, not RESET_REQD );
447             ILINE_CURR := READ_LINE( IMGE, DIM(LINE)-I+1, not RESET_REQD );
448             MLINE_CURR := ( . );
449
450             if (I-Y_CENTER) >= (-X_CENTER+1) then
451                 CIRCULAR_DISPLACE( I-Y_CENTER, X_CENTER, Y_CENTER,
452                                     I-Y_CENTER, -X_CENTER+1, IMAGE,
453                                     ILINE_CURR, DLINE(CURR), DLINE(PREV),
454                                     MLINE_CURR, CIRC_HORI_RAD, CIRC_HORI_ALT );
455             if (Y_CENTER-I) <= (DIM(PIXEL)-X_CENTER) then
456                 CIRCULAR_DISPLACE( I-Y_CENTER, X_CENTER, Y_CENTER,
457                                     Y_CENTER-I, DIM(PIXEL)-X_CENTER, IMAGE,
458                                     ILINE_CURR, DLINE(CURR), DLINE(PREV),
459                                     MLINE_CURR, CIRC_HORI_RAD, CIRC_HORI_ALT );
460             EXPANDING_DISPLACE( I-Y_CENTER, X_CENTER, Y_CENTER,
461                                 IMAGE, ILINE_CURR, DLINE(CURR),
462                                 MLINE_CURR, EXPD_HORI_RAD, EXPD_HORI_ALT );
463
464

```

```

465      WRITE_MAP_LINE( MFILE, MLINE_CURR, DIM(LINE)-I+1 );
466
467      end (* for I *);
468
469      DLINE(CURR) := READ_LINE( DTM, DIM(LINE)-Y_CENTER+1, RESET_REQD );
470      ILINE_CURR := READ_LINE( IMGE, DIM(LINE)-Y_CENTER+1, RESET_REQD );
471      MLINE_CURR := ( . . );
472      CIRC_HORI_RAD(0) := 0;
473      CIRC_HORI_ALT(0) := 0;
474      CIRC_HORI_RAD(180) := 0;
475      CIRC_HORI_ALT(180) := 0;
476
477      DISPLCE_CENTER_LINE( X_CENTER, Y_CENTER, IMAGE,
478                          ILINE_CURR, DLINE(CURR), MLINE_CURR,
479                          CIRC_HORI_RAD, CIRC_HORI_ALT
480                          );
481      WRITE_MAP_LINE( MFILE, MLINE_CURR, DIM(LINE)-Y_CENTER+1 );
482
483      end (* DISPLACE *);
484
485
486
487      (* ##### *)
488      (* *)
489      (* Displace the center line of the image (i.e. the line containing the *)
490      (* principle point) utilizing the circular horizon, HORI_RAD and *)
491      (* HORI_ALT, to calculate terrain visibility. *)
492      (* *)
493      (* ##### *)
494      procedure DISPLCE_CENTER_LINE( X_CENTER, Y_CENTER : integer;
495                                   var IMAGE : INT_ARRAY;
496                                   var ILINE, DLINE : INT_LINE;
497                                   var MLINE : MAPSET;
498                                   var HORI_RAD, HORI_ALT : HORZN_LINE );
499
500      var
501          I : integer;
502          RAD, RF, D : short;
503
504
505      begin
506
507          for I := X_CENTER-1 downto 1 do
508              begin
509
510                  RAD := abs( I - X_CENTER );
511                  RF := RAD * FOCAL_LENGTH;
512                  D := RF / (H - DLINE(I)) - RF / H;
513                  RAD := RAD + D * (H / FOCAL_LENGTH);
514                  if VISIBLE( DLINE(I), HORI_ALT(180),
515                             RAD, HORI_RAD(180) ) then
516                      begin
517                          HORI_RAD(180) := RAD;
518                          HORI_ALT(180) := DLINE(I);
519                          if X_CENTER - RAD > max( 0, X_CENTER-HALF_IMAGE_SIZE ) then
520                              INSERT_PIXEL( IMAGE, Y_CENTER, X_CENTER-RAD, ILINE(I) );
521                          else
522                              MLINE := MLINE + ( . (MAX_IMAGE_SIZE-I) . );

```



```

523         end (* if RAD > HORI_RAD(180) and ... *);
524     else
525         MLINE := MLINE + (. (MAX_IMAGE_SIZE-I) .);
526
527     end (* for I *);
528
529     INSERT_PIXEL( IMAGE, Y_CENTER, X_CENTER, ILINE(X_CENTER) );
530
531     for I := X_CENTER+1 to DIM(PIXEL) do
532     begin
533
534         RAD := abs( I - X_CENTER );
535         RF  := RAD * FOCAL_LENGTH;
536         D   := RF / (H - DLINE(I)) - RF / H;
537         RAD := RAD + D * (H / FOCAL_LENGTH);
538         if VISIBLE( DLINE(I), HORI_ALT(O),
539             RAD, HORI_RAD(O) ) then
540         begin
541             HORI_RAD(O) := RAD;
542             HORI_ALT(O) := DLINE(I);
543             if X_CENTER + RAD <= min( DIM(PIXEL), X_CENTER+HALF_IMAGE_SIZE ) then
544                 INSERT_PIXEL( IMAGE, Y_CENTER, X_CENTER+RAD, ILINE(I) );
545             else
546                 MLINE := MLINE + (. (MAX_IMAGE_SIZE-I) .);
547             end (* if RAD > HORI_RAD(O) and ... *);
548         else
549             MLINE := MLINE + (. (MAX_IMAGE_SIZE-I) .);
550         end
551     end (* for I *);
552
553
554
555     end (* DISPLCE_CENTER_LINE *);
556
557
558
559     (* ##### *)
560     (* *)
561     (* Displace the pixels of ILINE utilizing the linear expanding *)
562     (* horizon, HORI_RAD and HORI_ALT, to calculate terrain visibility. *)
563     (* This horizon must be expanded prior to these calculations and this *)
564     (* expansion is accomplished by EXPNDNG_HORIZON_UPDATE. The image *)
565     (* pixel's XY (cartesian) coordinates are first transformed to polar *)
566     (* coordinates and then the displacement is introduced. The new polar *)
567     (* coordinates are then converted to XY coordinates and the displaced *)
568     (* pixel is inserted in the perspective image at these new coordinates. *)
569     (* *)
570     (* ##### *)
571     procedure EXPANDING_DISPLACE( Y_COORD,
572         X_CENTER, Y_CENTER : integer;
573         var IMAGE          : INT_ARRAY;
574         var ILINE, DLINE   : INT_LINE;
575         var MLINE          : MAPSET;
576         var HORI_RAD, HORI_ALT : HORZN_LINE );
577
578     var
579         I, PREV_Y, PREV_START,
580         PREV_STOP          : integer;

```

```

581      H1, R1, RF, D      : short;
582      POLAR              : POLAR_COORD;
583      XY                 : XY_COORD;
584
585
586      begin
587
588          if Y_COORD > 0 then
589              PREV_Y := Y_COORD - 1;
590          else
591              PREV_Y := Y_COORD + 1;
592
593          PREV_START := max( -abs( PREV_Y ), -X_CENTER+1 );
594          PREV_STOP  := min( +abs( PREV_Y ), DIM(PIXEL)-X_CENTER );
595
596          EXPNDNG_HORIZON_UPDATE( Y_COORD, X_CENTER, PREV_START, PREV_STOP,
597                                PREV_Y,  HORI_RAD, HORI_ALT );
598
599          for I := PREV_START to PREV_STOP do
600              begin
601
602                  POLAR := POLAR_TRANSFORM( I, Y_COORD );
603                  H1     := DLINE(I+X_CENTER);
604                  RF      := POLAR(RDS) * FOCAL_LENGTH;
605                  D       := RF / (H - H1) - RF / H;
606                  R1      := POLAR(RDS) + D * (H / FOCAL_LENGTH);
607                  if not VISIBLE( H1, HORI_ALT(I+X_CENTER),
608                                R1, HORI_RAD(I+X_CENTER) ) then
609                      MLINE := MLINE + ( (MAX_IMAGE_SIZE-(I+X_CENTER)) );
610                  else
611                      begin
612
613                          XY := XY_TRANSFORM( R1, POLAR(THT) );
614                          XY(X) := XY(X) + X_CENTER;
615                          XY(Y) := XY(Y) + Y_CENTER;
616                          if (XY(X) > max( 0, X_CENTER-HALF_IMAGE_SIZE )) and
617                             (XY(Y) > max( 0, Y_CENTER-HALF_IMAGE_SIZE )) and
618                             (XY(X) <= min( DIM(PIXEL), X_CENTER+HALF_IMAGE_SIZE )) and
619                             (XY(Y) <= min( DIM(LINE), Y_CENTER+HALF_IMAGE_SIZE )) then
620                              begin
621                                  INSERT_PIXEL( IMAGE, XY(Y), XY(X), ILINE(I+X_CENTER) );
622                                  HORI_ALT(I+X_CENTER) := H1;
623                                  HORI_RAD(I+X_CENTER) := R1;
624                              end (* if (XY(X) > 0) ... *);
625                          else
626                              MLINE := MLINE + ( (MAX_IMAGE_SIZE-(I+X_CENTER)) );
627
628                      end (* else VISIBLE *);
629
630              end (* for I *);
631
632          end (* EXPANDING_DISPLACE *);
633
634      end (* EXPANDING_DISPLACE *);
635
636      (* ***** *)
637      (* ***** *)
638

```

```

639      (* Update the linear expanding horizon, HORI_RAD and HORI_ALT, by *)
640      (* projecting it from a horizon of N elements to a horizon of (N+2) *)
641      (* elements. *)
642      (* *)
643      (* ##### *)
644      procedure EXPNDNG_HORIZON_UPDATE( Y, X_CENTER,
645                                         PREV_START, PREV_STOP,
646                                         PREV_Y
647                                         var HORI_RAD, HORI_ALT : integer;
648                                         : HORZN_LINE );
649
650      var
651      LO, HI, X, START, STOP : integer;
652      LO_FACT, HI_FACT,
653      LO_ALT, HI_ALT,
654      LO_RAD, HI_RAD, NEW_X : short;
655      NEW_HORI_ALT,
656      NEW_HORI_RAD, FACTOR : HORZN_LINE;
657
658      begin
659
660      START := max( -abs( Y ), -X_CENTER+1 );
661      STOP := min( +abs( Y ), DIM(PIXEL)-X_CENTER );
662
663      for X := START+X_CENTER-1 to STOP+X_CENTER+1 do
664      begin
665      NEW_HORI_ALT(X) := 0;
666      NEW_HORI_RAD(X) := 0;
667      FACTOR(X) := 0;
668      end (* for X *);
669
670      if PREV_Y /= 0 then
671      begin
672
673      for X := PREV_START to PREV_STOP do
674      begin
675
676      if Y > 0 then
677      NEW_X := X + X / PREV_Y;
678      else
679      NEW_X := X - X / PREV_Y;
680      LO := trunc( NEW_X );
681      if NEW_X < 0 then
682      HI := LO - 1;
683      else
684      HI := LO + 1;
685      LO_FACT := abs( HI - NEW_X );
686      HI_FACT := 1 - LO_FACT;
687
688      if LO_FACT /= 0 then
689      begin
690
691      LO := LO + X_CENTER;
692      LO_ALT := NEW_HORI_ALT(LO) * FACTOR(LO) +
693      HORI_ALT(X+X_CENTER) * LO_FACT;
694      LO_RAD := NEW_HORI_RAD(LO) * FACTOR(LO) +
695      HORI_RAD(X+X_CENTER) * LO_FACT;
696      FACTOR(LO) := FACTOR(LO) + LO_FACT;

```

```

697      NEW_HORI_ALT(LO) := LO_ALT / FACTOR(LO);
698      NEW_HORI_RAD(LO) := LO_RAD / FACTOR(LO);
699
700      end (* if LO_FACT /= 0 *);
701
702      if HI_FACT /= 0 then
703      begin
704
705          HI      := HI + X_CENTER;
706          HI_ALT   := NEW_HORI_ALT(HI) * FACTOR(HI) +
707                  HORI_ALT(X+X_CENTER) * HI_FACT;
708          HI_RAD   := NEW_HORI_RAD(HI) * FACTOR(HI) +
709                  HORI_RAD(X+X_CENTER) * HI_FACT;
710          FACTOR(HI) := FACTOR(HI) + HI_FACT;
711          NEW_HORI_ALT(HI) := HI_ALT / FACTOR(HI);
712          NEW_HORI_RAD(HI) := HI_RAD / FACTOR(HI);
713
714      end (* if HI_FACT /= 0 *);
715
716      end (* for X *);
717
718      for X := START+X_CENTER to STOP+X_CENTER do
719      if FACTOR(X) = 0 then
720      begin
721          NEW_HORI_ALT(X) := (NEW_HORI_ALT(X+1) - NEW_HORI_ALT(X-1)) /
722                          2.0s + NEW_HORI_ALT(X-1);
723          NEW_HORI_RAD(X) := (NEW_HORI_RAD(X+1) - NEW_HORI_RAD(X-1)) /
724                          2.0s + NEW_HORI_RAD(X-1);
725      end (* for X *);
726
727      HORI_ALT := NEW_HORI_ALT;
728      HORI_RAD := NEW_HORI_RAD;
729
730      end (* if PREV_Y /= 0 *);
731
732      end (* EXPNDNG_HORIZON_UPDATE *);
733
734
735
736
737      (* ##### *)
738      (* *)
739      (* Set up and attach the files necessary for the system. Unit 0 is the *)
740      (* assumed input for the original DTM. If it is not assigned on the MTS *)
741      (* RUN command, the file name will be prompted for. Unit 1 is assumed *)
742      (* input for the original registered image. Its file name will also be *)
743      (* prompted for if not assigned. The file assigned to unit 10 will be *)
744      (* used for the output of the left perspective image; unit 11 for the *)
745      (* output of the right perspective image. The file assigned to *)
746      (* unit 12 will be used for the output of the left visibility map of the *)
747      (* DTM; unit 13 for the output of the right visibility map of the *)
748      (* DTM. If these units are not attached, they will default to MTS *)
749      (* temporary files -LEFT#, -RIGHT#, -LMAP#, and -RMAP# respectively. *)
750      (* *)
751      (* ##### *)
752      procedure FILES;
753
754      var

```

```

755          DTMFNAME, IMGNAME : INPUT_LINE;
756
757
758      begin
759
760
761          DTMFNAME := ' ';
762          DEFAULT_FNAME( 'DEFAULT 0=*DUMMY*;' );
763          GET_FNAME( 'QUERY FNAME 0:', O, DTMFNAME );
764
765          if DTMFNAME = '*DUMMY*' then
766              begin
767
768                  PRINT_MSG( UNIT_0_UNASSIGNED, '', O );
769                  DTMFNAME := READ_STRING;
770                  while DTMFNAME = ' ' do
771                      begin
772                          PRINT_MSG( INVALID_FILE, '', O );
773                          DTMFNAME := READ_STRING;
774                          if DTMFNAME = 'CANCEL' or DTMFNAME = 'HALT' then
775                              halt;
776                          end (* while DTMFNAME = ' ' *);
777
778                      end (* if DTMFNAME = '*DUMMY*' *);
779
780
781          IMGNAME := ' ';
782          DEFAULT_FNAME( 'DEFAULT 1=*DUMMY*;' );
783          GET_FNAME( 'QUERY FNAME 1:', O, IMGNAME );
784
785          if IMGNAME = '*DUMMY*' then
786              begin
787
788                  PRINT_MSG( UNIT_1_UNASSIGNED, '', O );
789                  IMGNAME := READ_STRING;
790                  while IMGNAME = ' ' do
791                      begin
792                          PRINT_MSG( INVALID_FILE, '', O );
793                          IMGNAME := READ_STRING;
794                          if IMGNAME = 'CANCEL' or IMGNAME = 'HALT' then
795                              halt;
796                          end (* while IMGNAME = ' ' *);
797
798                      end (* if IMGNAME = '*DUMMY*' *);
799
800
801          DEFAULT_FNAME( 'DEFAULT 10=-LEFT#;' );
802          DEFAULT_FNAME( 'DEFAULT 11=-RIGHT#;' );
803          DEFAULT_FNAME( 'DEFAULT 12=-LMAP#;' );
804          DEFAULT_FNAME( 'DEFAULT 13=-RMAP#;' );
805
806          reset ( DTM_FILE, DTMFNAME );
807          reset ( IMAGE_FILE, IMGNAME );
808          rewrite( LFILE, 10 );
809          rewrite( RFILE, 11 );
810          rewrite( LMFILE, 12 );
811          rewrite( RMFILE, 13 );
812

```

```

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

```

```

813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870

```

```

871.      I      : integer;
872.      LAST_CHAR_BLANK : boolean;
873.      CHR      : char;
874.
875.
876.      begin
877.
878.          I := 0;
879.          LAST_CHAR_BLANK := false;
880.
881.          while not eof( INPUT ) and not eoln( INPUT ) do
882.              begin
883.                  read( CHR );
884.                  if CHR = ' ' or not COMPRESS then
885.                      begin
886.                          if LAST_CHAR_BLANK and I>0 then
887.                              begin
888.                                  I := I + 1;
889.                                  STR(I) := ' ';
890.                              end;
891.                          I := I + 1;
892.                          STR(I) := CHR;
893.                          LAST_CHAR_BLANK := false;
894.                      end;
895.                  else
896.                      LAST_CHAR_BLANK := true;
897.                  end;
898.
899.                  if not COMPRESS and I>0 then
900.                      GET_EOLN := I - 1;
901.                  else
902.                      GET_EOLN := I
903.
904.
905.          end (* GET_EOLN *);
906.
907.
908.
909.
910.      (* ##### *)
911.      (* *)
912.      (* Return the numerical value of a string in the input stream. If *)
913.      (* INTEGRAL_VALUE is true then the string must represent an integer, *)
914.      (* otherwise a real number. The numerical value must also be in the *)
915.      (* range specified by RANGE. *)
916.      (* *)
917.      (* ##### *)
918.      function GET_NUM( RANGE : VALUE_RANGE;
919.                      INTEGRAL_VALUE : boolean ) : real;
920.
921.      var
922.          STR      : INPUT_LINE;
923.          VALID    : boolean;
924.          VALU     : real;
925.          I, J, MULT : integer;
926.
927.
928.      begin

```

```

929      repeat
930      VALID := true;
931      read( STR );
932      if eof( INPUT ) then      (* CHECK FOR EOF *)
933      begin
934      VALID := false;
935      PRINT_MSG( EOF_NO, ' ', 0 )
936      end;
937      else
938      begin
939      VALU := 0.0;      (* GET INTEGRAL PART *)
940      I := 1;
941      MULT := 1;
942      while I <= MAX_LINE_LEN and STR(I) = ' ' do
943      I := I + 1;
944      if STR(I) = '+' or STR(I) = '-' then
945      begin
946      if STR(I) = '-' then
947      MULT := -1;
948      I := I + 1
949      end;
950      while STR(I) in ( '0'..'9' ) do
951      begin
952      VALU := 10.0 * VALU + ord( STR(I) ) - ord('0');
953      I := I + 1
954      end;
955      if STR(I) = '.' then
956      begin      (* GET FRACTIONAL PART *)
957      J := 10;
958      I := I + 1;
959      while STR(I) in ( '0'..'9' ) do
960      begin
961      VALU := VALU + ( (ord(STR(I)) - ord('0'))/J);
962      I := I + 1;
963      J := J * 10
964      end;
965      end;
966      if STR(I) <= ' ' or ( eofln(INPUT) and I = MAX_LINE_LEN+1 ) then
967      begin      (* INVALID NUMBER *)
968      VALID := false;
969      if STR(I) <= ' ' then
970      PRINT_MSG( INVALID_NO, STR(I), 1 )
971      else
972      if RANGE in ( . PERCENT . ) then
973      begin
974      VALID := true;
975      MULT := 1;
976      VALU := DEFAULT;
977      end (* if RANGE in *);
978      else
979      PRINT_MSG( NULL_NO, ' ', 0 )
980      end
981      else if INTEGRAL_VALUE and trunc(VALU) <= VALU then
982      begin
983      VALID := false;
984      PRINT_MSG( NON_INTEGRAL_VALUE, ' ', 0 )
985      end
986      else if STR(I) = ' ' and ( STR(I-1)='-' or STR(I-1)='+' ) then

```



```

987         begin
988             VALID := false;
989             PRINT_MSG( UNEXPECTED_BLANK, '', 0 )
990         end;
991     else if ( RANGE = POSITIVE and (MULT < 0 or VALU = 0.0)) or
992            ( RANGE = IMAGE_SIZE and (MULT < 0 or
993            VALU < 1 or VALU > MAX_IMAGE_SIZE)) or
994            ( RANGE = LINES and (MULT < 0 or
995            VALU < 1 or VALU > DIM(LINE))) or
996            ( RANGE = LINE_LENGTH and (MULT < 0 or
997            VALU < 1 or VALU > DIM(PIXEL))) or
998            ( RANGE = INTENSITY and (MULT < 0 or
999            VALU < MIN_INTENSITY or
1000            VALU > MAX_INTENSITY)) or
1001            ( RANGE = NON_NEGATIVE and MULT < 0) or
1002            ( RANGE = PERCENT and (MULT < 0 or
1003            VALU < 0 or VALU > 100)) or
1004            ( RANGE = NON_ZERO and VALU = 0.0) then
1005         begin
1006             VALID := false;
1007             PRINT_MSG( NOT_IN_RANGE, '', 0 )
1008         end;
1009     end;
1010     until VALID;
1011     GET_NUM := VALU * MULT
1012
1013
1014     end ( * GET_NUM * );
1015
1016
1017
1018     ( * ##### *)
1019     ( * *)
1020     ( * Update the circular horizon, HORI_RAD and HORI_ALT, by updating the *)
1021     ( * information in the horizon between the CURRENT THETA and the *)
1022     ( * PREVIOUS THETA. *)
1023     ( * *)
1024     ( * ##### *)
1025     procedure HORIZON_UPDATE( CURR_POLAR, PREV_POLAR : POLAR_COORD;
1026                             CURR_ALT, PREV_ALT : int;
1027                             var HORI_RAD, HORI_ALT : HORZN_LINE );
1028
1029     var
1030         START_ALT,
1031         STOP_ALT : int;
1032         I, START, STOP : integer;
1033         ALT, RAD, RF, D : short;
1034         START_POLAR,
1035         STOP_POLAR : POLAR_COORD;
1036
1037
1038     begin
1039
1040         if CURR_POLAR(THT) > 270 and PREV_POLAR(THT) = 0 then
1041             PREV_POLAR(THT) := 360;
1042
1043         if CURR_POLAR(THT) < PREV_POLAR(THT) then
1044             begin

```

```

1045
1046      START := trunc( CURR_POLAR(THT) );
1047      STOP  := trunc( PREV_POLAR(THT) );
1048      if START <> CURR_POLAR(THT) then
1049          incr( START );
1050      START_POLAR := CURR_POLAR;
1051      STOP_POLAR  := PREV_POLAR;
1052      START_ALT   := CURR_ALT;
1053      STOP_ALT    := PREV_ALT;
1054
1055      end (* if CURR_POLAR(THT) < PREV_POLAR(THT) *);
1056  else
1057      begin
1058
1059          START := trunc( PREV_POLAR(THT) );
1060          STOP  := trunc( CURR_POLAR(THT) );
1061          if STOP <> CURR_POLAR(THT) then
1062              decr( STOP );
1063          START_POLAR := PREV_POLAR;
1064          STOP_POLAR  := CURR_POLAR;
1065          START_ALT   := PREV_ALT;
1066          STOP_ALT    := CURR_ALT;
1067
1068          end (* else CURR_POLAR(THT) >= PREV_POLAR(THT) *);
1069
1070      for I := START to STOP do
1071          begin
1072
1073              ALT := (START_ALT * (STOP_POLAR(THT) - I) +
1074                     STOP_ALT * (I - START_POLAR(THT))) /
1075                     (STOP_POLAR(THT) - START_POLAR(THT));
1076
1077              RAD := (START_POLAR(RDS) * (STOP_POLAR(THT) - I) +
1078                     STOP_POLAR(RDS) * (I - START_POLAR(THT))) /
1079                     (STOP_POLAR(THT) - START_POLAR(THT));
1080
1081              RF := RAD * FOCAL_LENGTH;
1082              D  := RF / (H - ALT) - RF / H;
1083              RAD := RAD + D * (H / FOCAL_LENGTH);
1084
1085              if VISIBLE( ALT, HORI_ALT(I), RAD, HORI_RAD(I) ) then
1086                  begin
1087                      HORI_RAD(I) := RAD;
1088                      HORI_ALT(I) := ALT;
1089                      end (* if RAD > HORI_RAD(I) and ... *);
1090
1091              end (* for I *);
1092
1093          end (* HORIZON_UPDATE *);
1094
1095      end (* HORIZON_UPDATE *);
1096
1097
1098      (* ##### *)
1099      (* *)
1100      (* Insert the pixel with intensity INTEN into the IMAGE at position *)
1101      (* (X,Y). *)
1102      (* *)

```

```

1103      (* ##### *)
1104      procedure INSERT_PIXEL( var IMAGE : INT_ARRAY; Y, X : short;
1105                             INTEN : integer;
1106                             );
1107      var
1108          X_INT, Y_INT      : integer;
1109
1110      (* Perform the actual insertion. *)
1111      procedure PERFORM_INSERT( Y, X, I : integer );
1112      begin
1113          if IMAGE(Y,X) = UNDEFINED then
1114              IMAGE(Y,X) := I;
1115          else
1116              IMAGE(Y,X) := round( (I + IMAGE(Y,X)) / 2 );
1117          end (* PERFORM_INSERT *);
1118
1119      begin
1120          X_INT := trunc( X );
1121          Y_INT := trunc( Y );
1122          PERFORM_INSERT( Y_INT, X_INT, INTEN );
1123
1124          if Y = Y_INT then
1125              begin
1126                  if X <> X_INT then
1127                      PERFORM_INSERT( Y_INT, X_INT+1, INTEN );
1128                  end;
1129              else
1130                  if X = X_INT then
1131                      PERFORM_INSERT( Y_INT+1, X_INT, INTEN );
1132                  else
1133                      begin
1134                          PERFORM_INSERT( Y_INT, X_INT+1, INTEN );
1135                          PERFORM_INSERT( Y_INT+1, X_INT, INTEN );
1136                          PERFORM_INSERT( Y_INT+1, X_INT+1, INTEN );
1137                      end;
1138                  end;
1139              end;
1140          end (* INSERT_PIXEL *);
1141
1142      (* ##### *)
1143      (* For undefined pixels of the image array IMAGE, perform a cubic
1144      (* convolution to define them.
1145      (* ##### *)
1146      procedure INTERPOLATE( var IMAGE : INT_ARRAY );
1147      var
1148          X_CENT, Y_CENT,
1149          X, Y, X_MAX, Y_MAX      : integer;
1150          DEFINED_FOUND           : boolean;

```

```
1161
1162      (* Perform a cubic convolution if undefined *)
1163      (* and in the interior of the image. *)
1164      procedure PERFORM_INTER( X, Y : integer );
1165
1166      begin
1167
1168          if IMAGE(X,Y) <> UNDEFINED then
1169              DEFINED_FOUND := true;
1170          else
1171              if DEFINED_FOUND then
1172                  CUBIC_CONVOLUTION( IMAGE, X, Y );
1173              else
1174                  IMAGE(X,Y) := MIN_INTENSITY;
1175
1176          end (* PERFORM_INTER *);
1177
1178      begin
1179
1180          X_MAX := DIM(LINE );
1181          Y_MAX := DIM(PIXEL);
1182          X_CENT := trunc( (X_MAX+1) / 2 );
1183          Y_CENT := trunc( (Y_MAX+1) / 2 );
1184
1185          for X := 1 to X_CENT do
1186              begin
1187
1188                  DEFINED_FOUND := false;
1189                  for Y := 1 to min( X, Y_CENT ) do
1190                      PERFORM_INTER( X, Y );
1191
1192                  DEFINED_FOUND := false;
1193                  for Y := Y_MAX downto max( Y_MAX-X+1, Y_CENT ) do
1194                      PERFORM_INTER( X, Y );
1195
1196              end (* for X *);
1197
1198          for X := X_CENT+1 to X_MAX do
1199              begin
1200
1201                  DEFINED_FOUND := false;
1202                  for Y := 1 to min( X_MAX-X+1, Y_CENT ) do
1203                      PERFORM_INTER( X, Y );
1204
1205                  DEFINED_FOUND := false;
1206                  for Y := Y_MAX downto max( Y_MAX-X_MAX+X, Y_CENT ) do
1207                      PERFORM_INTER( X, Y );
1208
1209              end (* for X *);
1210
1211          for Y := 1 to Y_CENT do
1212              begin
1213
1214                  DEFINED_FOUND := false;
1215                  for X := 1 to min( Y, X_CENT ) do
1216                      PERFORM_INTER( X, Y );
```

```

1219
1220     DEFINED_FOUND := false;
1221     for X := X_MAX downto max( X_MAX-Y+1, X_CENT ) do
1222         PERFORM_INTER( X, Y );
1223
1224     end (* for Y *);
1225
1226     for Y := Y_CENT+1 to Y_MAX do
1227         begin
1228
1229             DEFINED_FOUND := false;
1230             for X := 1 to min( Y_MAX-Y+1, X_CENT ) do
1231                 PERFORM_INTER( X, Y );
1232
1233             DEFINED_FOUND := false;
1234             for X := X_MAX downto max( X_MAX-Y_MAX+Y, X_CENT ) do
1235                 PERFORM_INTER( X, Y );
1236
1237             end (* for Y *);
1238
1239
1240         end (* INTERPOLATE *);
1241
1242
1243
1244
1245     (* ##### *)
1246     (* *)
1247     (* Calculate which octant of a circle the cartesian coordinates (X,Y) *)
1248     (* reside in. The first octant is where X>0, Y>0, and X>=Y, assuming *)
1249     (* the principle point to be the origin. Move counterclockwise to *)
1250     (* the second octant. *)
1251     (* *)
1252     (* ##### *)
1253     procedure OCT_1_TRANSFORM( var X, Y, OCT : integer );
1254
1255     var
1256         X_TEMP          : integer;
1257
1258
1259     begin
1260
1261         if X > 0 then
1262             if Y >= 0 then
1263                 if abs( Y ) > abs( X ) then
1264                     OCT := OCT_2;
1265                 else (* abs( Y ) <= abs( X ) *)
1266                     OCT := OCT_1;
1267                 else (* Y < 0 *)
1268                     if abs( Y ) >= abs( X ) then
1269                         OCT := OCT_7;
1270                     else (* abs( Y ) < abs( X ) *)
1271                         OCT := OCT_8;
1272
1273                     else if X < 0 then
1274                         if Y > 0 then
1275                             if abs( Y ) >= abs( X ) then
1276                                 OCT := OCT_3;

```

```

1277         else (* abs( Y ) < abs( X ) *)
1278             OCT := OCT_4;
1279         else if Y < 0 then
1280             if abs( Y ) > abs( X ) then
1281                 OCT := OCT_6;
1282             else (* abs( Y ) <= abs( X ) *)
1283                 OCT := OCT_5;
1284             else (* Y = 0 *)
1285                 OCT := OCT_5;
1286
1287         else (* X = 0 *)
1288             if Y > 0 then
1289                 OCT := OCT_3;
1290             else if Y < 0 then
1291                 OCT := OCT_7;
1292             else (* Y = 0 *)
1293                 OCT := OCT_1;
1294
1295
1296         case OCT of
1297
1298             OCT_1, OCT_4, OCT_5, OCT_8 :
1299                 begin
1300                     X := abs( X );
1301                     Y := abs( Y );
1302                     end (* OCT_1, ... *);
1303
1304             OCT_2, OCT_3, OCT_6, OCT_7 :
1305                 begin
1306                     X_TEMP := X;
1307                     X := abs( Y );
1308                     Y := abs( X_TEMP );
1309                     end (* OCT_2, ... *);
1310
1311             end (* case OCT of *);
1312
1313
1314         end (* OCT_1_TRANSFORM *);
1315
1316
1317
1318         (* ##### *)
1319         (* ##### *)
1320         (* Initialize the lookup tables to transform cartesian coordinates *)
1321         (* into polar coordinates. *)
1322         (* ##### *)
1323         (* ##### *)
1324         procedure POLAR_INIT( SIZE : integer );
1325
1326         var
1327             X, Y : integer;
1328
1329
1330         begin
1331
1332             RADIUS_LOOKUP := ALLOCATE_SPACE( round( SIZE * (SIZE + 1) / 2 ) );
1333             THETA_LOOKUP := ALLOCATE_SPACE( round( SIZE * (SIZE + 1) / 2 ) );
1334

```

```

1335     for X := 0 to SIZE-1 do
1336         for Y := 0 to X do
1337             begin
1338
1339                 PUT_ARRAY( RADIUS_LOOKUP, X, Y, roundtoshort( sqrt( sqr( X ) + sqr( Y ) ) ) );
1340                 if X = 0 then
1341                     PUT_ARRAY( THETA_LOOKUP, X, Y, UNDEFINED );
1342                 else
1343                     PUT_ARRAY( THETA_LOOKUP, X, Y, short( arctan( Y/X ) ) );
1344
1345             end (* for Y *);
1346
1347         end (* POLAR_INIT *);
1348
1349
1350
1351
1352     (* ##### *)
1353     (*
1354     (*   Via lookup tables, transform the cartesian coordinates (X,Y) into
1355     (*   polar coordinates. Return the polar form.
1356     (*
1357     (* ##### *)
1358     function POLAR_TRANSFORM( X, Y : integer ) : POLAR_COORD;
1359
1360     var
1361         OCT          : integer;
1362         OCT_1_THETA  : short;
1363         POLAR        : POLAR_COORD;
1364
1365     begin
1366
1367         OCT_1_TRANSFORM( X, Y, OCT );
1368
1369         POLAR( RDS ) := GET_ARRAY( RADIUS_LOOKUP, X, Y );
1370
1371         OCT_1_THETA := roundtoshort( GET_ARRAY( THETA_LOOKUP, X, Y ) / DEG_TO_RAD );
1372         case OCT of
1373
1374             OCT_1 :
1375                 POLAR( THT ) := OCT_1_THETA;
1376             OCT_2 :
1377                 POLAR( THT ) := 90 - OCT_1_THETA;
1378             OCT_3 :
1379                 POLAR( THT ) := 90 + OCT_1_THETA;
1380             OCT_4 :
1381                 POLAR( THT ) := 180 - OCT_1_THETA;
1382             OCT_5 :
1383                 POLAR( THT ) := 180 + OCT_1_THETA;
1384             OCT_6 :
1385                 POLAR( THT ) := 270 - OCT_1_THETA;
1386             OCT_7 :
1387                 POLAR( THT ) := 270 + OCT_1_THETA;
1388             OCT_8 :
1389                 POLAR( THT ) := 360 - OCT_1_THETA;
1390
1391         end (* case OCT of *);
1392

```

```

1393
1394     POLAR_TRANSFORM := POLAR;
1395
1396
1397     end (* POLAR_TRANSFORM *);
1398
1399
1400
1401     (* ##### *)
1402     (* *)
1403     (* Print the message specified by MSG to the output stream. The *)
1404     (* parameters STRING and SIZE may be involved in the details of *)
1405     (* that message. *)
1406     (* *)
1407     (* ##### *)
1408     procedure PRINT_MSG( MSG : MSG_TYPE; STRING : INPUT_LINE;
1409                          SIZE : integer );
1410
1411     const
1412         MSG_PROMPT = ':';
1413
1414     var
1415         I,J          : integer;
1416         STR           : INPUT_LINE;
1417
1418     begin
1419
1420         SET_PREFIX( MSG_PROMPT, 1 );
1421
1422         if MSG in ( LINE_PROMPT, UNIT_O_UNASSIGNED, INVALID_FILE,
1423                   ENTER_CENTER_X_COORD, ENTER_GRID_SPACING,
1424                   ENTER_PRINT_SIZE,
1425                   UNIT_I_UNASSIGNED, ENTER_ELEVATION ) then
1426             writeln;
1427
1428         case MSG of
1429             UNEXPECTED_EOF :
1430                 writeln(' Unexpected EOF -- Ignored');
1431             UNREC_CMD      :
1432                 begin
1433                     writeln(' Invalid command -- Input line ignored');
1434                     if not eoln( INPUT ) then
1435                         I := GET_EOLN( STR, COMPRESSED )
1436                     end;
1437                 end;
1438             EOLN_MSG       :
1439                 begin
1440                     I := GET_EOLN( STR, COMPRESSED );
1441                     if I /= 0 then
1442                         begin
1443                             write(' Invalid character(s): ');
1444                             WRITE_STRING( STR, I );
1445                             writeln(' -- Ignored')
1446                         end
1447                     end;
1448                 end;
1449             LINE_PROMPT    :
1450                 writeln('&Enter number of lines of DTM and IMAGE (Integer {1-256}):');
1451             PIXEL_PROMPT  :

```



```

1451         writeln('Enter number of pixels per line          (Integer {1-256}):');
1452     EOF_NO :
1453         writeln('Unexpected EOF, Enter number:');
1454     NULL_NO :
1455         writeln(' &Not expecting null line -- Ignored; Enter number:');
1456     NON_INTEGRAL_VALUE:
1457         writeln(' &Non-integral value, Re-enter number:');
1458     UNEXPECTED_BLANK:
1459         writeln('Unexpected blank, Re-enter number:');
1460     NOT_IN_RANGE :
1461         writeln(' &Out of range, Re-enter number:');
1462     UNIT_O_UNASSIGNED:
1463         writeln(' Logical unit 0 has not been assigned.',
1464             EOL, ' & Enter FDname of location of DTM :');
1465     UNIT_1_UNASSIGNED:
1466         writeln(' Logical unit 1 has not been assigned.',
1467             EOL, ' & Enter FDname of location of IMAGE :');
1468     INVALID_FILE :
1469         writeln(' Invalid FDname. Enter again or', EOL,
1470             ' &Enter "CANCEL" to terminate program:');
1471     UNEXPECTED_INPUT_EOF:
1472         writeln(' Unexpected EOF encountered on input',
1473             ' -- DTM size updated');
1474     INVALID_NO :
1475         begin
1476             writeln(' &Invalid character. "', STRING(1), '"', Re-enter number:');
1477             I := GET_EOLN( STR, COMPRESSED )
1478         end;
1479     ENTER_CENTER_X_COORD:
1480         begin
1481             write ( ' Enter the coordinates of the',
1482                 ' principal', EOL, ' point of the ');
1483             WRITE_STRING( STRING, SIZE );
1484             writeln(' image:', EOL,
1485                 ' & X (pixel - INTEGER {1-', DIM(PIXEL):0, '}):');
1486         end;
1487     ENTER_CENTER_Y_COORD:
1488         writeln(' & Y (line - INTEGER {1-', DIM(LINE ):0, '}):');
1489     ENTER_GRID_SPACING:
1490         writeln(' &Enter DTM grid spacing          (in DTM units):');
1491     ENTER_ELEVATION :
1492         writeln(' &Enter elevation of imaging device    (in DTM units):');
1493     ENTER_FOCAL_LENGTH:
1494         writeln(' &Enter focal length of imaging device (in DTM units):');
1495     ENTER_PRINT_SIZE:
1496         writeln(' &Enter print size of imaging device    (in DTM units):');
1497     PRINT_PRINT_SIZE:
1498         writeln( ' Print size, in DTM pixels, is approximately ',
1499             SIZE:0, ' X ', SIZE:0, ' pixels.' );
1500
1501     end (* case *);
1502
1503     if MSG in ( LINE_PROMPT, PIXEL_PROMPT, INVALID_NO,
1504         NON_INTEGRAL_VALUE, UNEXPECTED_BLANK, NOT_IN_RANGE,
1505         UNIT_O_UNASSIGNED, UNIT_1_UNASSIGNED, INVALID_FILE,
1506         ENTER_ELEVATION, ENTER_CENTER_X_COORD,
1507         ENTER_GRID_SPACING, ENTER_FOCAL_LENGTH,
1508         ENTER_CENTER_Y_COORD, ENTER_PRINT_SIZE,

```

```

1509             EOF_NO, NULL_NO .) then
1510                 readln;
1511
1512             SET_PREFIX( PROMPT, 1 )
1513
1514
1515         end  (* PRINT_MSG *);
1516
1517
1518
1519         (* ##### *)
1520         (* *)
1521         (* Insert the value VALU into ARR(X,Y). *)
1522         (* *)
1523         (* ##### *)
1524         procedure PUT_ARRAY( ARR : POINTER; X, Y : Integer; VALU : short );
1525
1526             var
1527                 PTR          : POINTER;
1528
1529
1530             begin
1531
1532                 if Y > X then
1533                     SYSTEM_ERROR( ' In "PUT_ARRAY"; Subscript error' );
1534                     PTR := POINTER( adrof( ARR@ ) + (BYTES_PER_WORD *
1535                                     WORDS_PER_ENTRY * (X * (X + 1) / 2 + Y)) );
1536                     PTR@ := VALU;
1537
1538
1539                 end  (* PUT_ARRAY *);
1540
1541
1542
1543         (* ##### *)
1544         (* *)
1545         (* If READ_REQD is true then read in a line from the file DTM_FILE, *)
1546         (* otherwise perform the subsequent operations on the present input *)
1547         (* buffer. Convert the binary form of the buffer into integer form *)
1548         (* assuming two bytes per pixel. *)
1549         (* *)
1550         (* ##### *)
1551         function READ_DTM_LINE( var DTM_FILE : DTMFILE; READ_REQD : boolean ) : INT_LINE;
1552
1553             var
1554                 LINE          : INT_LINE;
1555                 I              : Integer;
1556
1557
1558             begin
1559
1560                 if READ_REQD then
1561                     get( DTM_FILE );
1562                     for I := 1 to DIM(PIXEL) do
1563                         LINE(I) := int( DTM_FILE@((I-1)*2 + 1) ) * 256 +
1564                                     int( DTM_FILE@((I-1)*2 + 2) );
1565                     READ_DTM_LINE := LINE;
1566

```

```

1567.
1568      end      (* READ_DTM_LINE *);
1569
1570
1571
1572
1573      (* ##### *)
1574      (* *)
1575      (* If READ_REQD is true then read in a line from the file IMAGE_FILE. *)
1576      (* otherwise perform the subsequent operations on the present input *)
1577      (* buffer. Convert the binary form of the buffer into integer form *)
1578      (* assuming one byte per pixel. *)
1579      (* *)
1580      (* ##### *)
1581      function READ_IMAGE_LINE( var IMAGE_FILE : IMAGEFILE; READ_REQD : boolean ) : INT_LINE;
1582
1583      var
1584          LINE          : INT_LINE;
1585          I              : integer;
1586
1587
1588      begin
1589
1590          if READ_REQD then
1591              get( IMAGE_FILE );
1592              for I := 1 to DIM(PIXEL) do
1593                  LINE(I) := int( IMAGE_FILE@I );
1594              READ_IMAGE_LINE := LINE;
1595
1596
1597          end      (* READ_IMAGE_LINE *);
1598
1599
1600
1601      (* ##### *)
1602      (* *)
1603      (* Read the LINE_NOth line of either the DTM_FILE or the IMAGE_FILE *)
1604      (* depending on the value of TYP. The reads are performed by either *)
1605      (* READ_DTM_LINE or READ_IMAGE_LINE. The result is returned in INT_LINE. *)
1606      (* *)
1607      (* ##### *)
1608      function READ_LINE( TYP, LINE_NO : integer; RESET_REQD : boolean )
1609          : INT_LINE;
1610
1611      var
1612          I              : integer;
1613
1614
1615      begin
1616
1617          case TYP of
1618
1619              DTM          :
1620                  begin
1621                      if RESET_REQD then
1622                          begin
1623                              reset( DTM_FILE );
1624                              for I := 1 to LINE_NO-1 do

```

```

1625         get( DTM_FILE );
1626         READ_LINE := READ_DTM_LINE( DTM_FILE,
1627                                     not READ_REQD );
1628     end (* if RESET_REQD *);
1629 else
1630     READ_LINE := READ_DTM_LINE( DTM_FILE,
1631                                 READ_REQD );
1632 end (* DTM *);
1633
1634
1635     IMGE :
1636     begin
1637         if RESET_REQD then
1638             begin
1639                 reset( IMAGE_FILE );
1640                 for I := 1 to LINE_NO-1 do
1641                     get( IMAGE_FILE );
1642                     READ_LINE := READ_IMAGE_LINE( IMAGE_FILE,
1643                                                     not READ_REQD );
1644                 end (* if RESET_REQD *);
1645             else
1646                 READ_LINE := READ_IMAGE_LINE( IMAGE_FILE,
1647                                                 READ_REQD );
1648             end (* IMGE *);
1649         end (* case TYP of *);
1650     end (* READ_LINE *);
1651
1652
1653 end (* READ_LINE *);
1654
1655
1656
1657 (* ##### *)
1658 (* *)
1659 (* Read a string from the input stream removing all leading blanks. *)
1660 (* The string is then returned. *)
1661 (* *)
1662 (* ##### *)
1663 function READ_STRING : INPUT_LINE;
1664
1665     var
1666         I           : integer;
1667         STR         : INPUT_LINE;
1668
1669
1670     begin
1671
1672         STR := ' ';
1673         I := 0;
1674         repeat
1675             incr( I );
1676             read( STR(I) );
1677             if STR(I) = ' ' and I = 1 then
1678                 I := 0;
1679             until eof(INPUT) or eoln(INPUT) or (I >= 0 and STR(I) = ' ')
1680                 or I = MAX_LINE_LEN;
1681
1682         READ_STRING := STR;

```

```

1683
1684
1685       end      (* READ_STRING *);
1686
1687
1688
1689
1690 (* ##### *)
1691 (* ##### *)
1692 (* Initialize the system by setting up the files, getting the dimensions *)
1693 (* of the DTM, and reading the necessary system parameters. *)
1694 (* ##### *)
1695 (* ##### *)
1696 procedure SYS_INIT;
1697
1698     var
1699         MAX_LEFT, MAX_RIGHT : int;
1700         I, J                 : integer;
1701
1702
1703     begin
1704
1705         SET_PREFIX( PROMPT, 1 );
1706         FILES;
1707         GET_DIMENSIONS;
1708
1709         PRINT_MSG( ENTER_GRID_SPACING, '', 0 );
1710         GRID_SPAC := roundtoshort( GET_NUM( POSITIVE, NOT INTEGRAL ) );
1711
1712         PRINT_MSG( ENTER_ELEVATION, '', 0 );
1713         H := roundtoshort( GET_NUM( POSITIVE, NOT INTEGRAL ) );
1714
1715         PRINT_MSG( ENTER_FOCAL_LENGTH, '', 0 );
1716         FOCAL_LENGTH := roundtoshort( GET_NUM( NON_NEGATIVE, NOT INTEGRAL ) );
1717
1718         PRINT_MSG( ENTER_PRINT_SIZE, '', 0 );
1719         PRINT_SIZE := roundtoshort( GET_NUM( POSITIVE, NOT INTEGRAL ) );
1720         FULL_IMAGE_SIZE :=
1721             round( PRINT_SIZE * H / ( FOCAL_LENGTH * GRID_SPAC ) );
1722         HALF_IMAGE_SIZE :=
1723             round( PRINT_SIZE * H / ( 2 * FOCAL_LENGTH * GRID_SPAC ) );
1724         PRINT_MSG( PRINT_PRINT_SIZE, '', FULL_IMAGE_SIZE );
1725
1726
1727         PRINT_MSG( ENTER_CENTER_X_COORD, 'left', 5 );
1728         CENTER(LEFT,X) := trunc( GET_NUM( LINE_LENGTH, INTEGRAL ) );
1729         PRINT_MSG( ENTER_CENTER_Y_COORD, '', 0 );
1730         CENTER(LEFT,Y) := trunc( GET_NUM( LINES, INTEGRAL ) );
1731
1732         PRINT_MSG( ENTER_CENTER_X_COORD, 'right', 5 );
1733         CENTER(RIGHT,X) := trunc( GET_NUM( LINE_LENGTH, INTEGRAL ) );
1734         PRINT_MSG( ENTER_CENTER_Y_COORD, '', 0 );
1735         CENTER(RIGHT,Y) := trunc( GET_NUM( LINES, INTEGRAL ) );
1736
1737
1738         MAX_LEFT := max( CENTER(LEFT,X), max( CENTER(LEFT,Y),
1739             max( DIM(PIXEL) - CENTER(LEFT,X),
1740                 DIM(LINE) - CENTER(LEFT,Y) ) ) );

```

```

1741      MAX_RIGHT := max( CENTER(RIGHT,X), max( CENTER(RIGHT,Y),
1742                max( DIM(PIXEL) - CENTER(RIGHT,X),
1743                DIM(LINE) - CENTER(RIGHT,Y) ) ) );
1744      MAX_HALF  := max( MAX_LEFT, MAX_RIGHT ) + 1;
1745
1746      POLAR_INIT( MAX_HALF );
1747      XY_INIT;
1748
1749
1750      end  (* SYSTEM_INITIALIZATION *);
1751
1752
1753
1754      (* ##### *)
1755      (* *)
1756      (* If a system error occurs, output the message MESSAGE and halt the *)
1757      (* execution of the system. *)
1758      (* *)
1759      (* ##### *)
1760      procedure SYSTEM_ERROR( MESSAGE : STR_50 );
1761
1762      begin
1763
1764          writeln( ' ==> SYSTEM ERROR ==> ', MESSAGE );
1765          halt;
1766
1767
1768      end  (* SYSTEM_ERROR *);
1769
1770
1771
1772      (* ##### *)
1773      (* *)
1774      (* Return true if terrain at radius RAD1 from the principle point (with *)
1775      (* elevation ALT1) has its visibility to the imaging system blocked by *)
1776      (* terrain at radius RAD2 from the principle point (with elevation *)
1777      (* ALT2). Otherwise return false. *)
1778      (* *)
1779      (* ##### *)
1780      function VISIBLE( ALT1, ALT2, RAD1, RAD2 : short ) : boolean;
1781
1782
1783
1784      begin
1785
1786          if (RAD1 <= RAD2) or
1787             ((RAD2 / (H - ALT2)) >= (RAD1 / (H - ALT1))) then
1788              VISIBLE := false;
1789          else
1790              VISIBLE := true;
1791
1792
1793      end  (* VISIBLE *);
1794
1795
1796
1797      (* ##### *)
1798      (* *)

```

```

1799.  (* Write the line LINE to the file IMAGE starting at pixel START.      *)
1800.  (* It is assumed that each pixel has a value between 0 and 255.        *)
1801.  (* *)                                                                    *)
1802.  (* ##### *)
1803.  procedure WRITE_IMAGE_LINE( var IMAGE : IMAGEFILE;
1804.                             LINE : INT_LINE; START : integer );
1805.
1806.  var
1807.      I          : integer;
1808.
1809.
1810.  begin
1811.
1812.      IMAGE@ := ' ';
1813.
1814.      for I := 1 to (START-1) do
1815.          IMAGE@ (I) := char( MIN_INTENSITY );
1816.
1817.      for I := START to DIM(PIXEL) do
1818.          IMAGE@ (I) := char( LINE(I) );
1819.
1820.      for I := (DIM(PIXEL)+1) to MAX_IMAGE_SIZE do
1821.          IMAGE@ (I) := char( MIN_INTENSITY );
1822.
1823.      put( IMAGE );
1824.
1825.
1826.  end  (* WRITE_IMAGE_LINE *);
1827.
1828.
1829.
1830.  (* ##### *)
1831.  (* *)                                                                    *)
1832.  (* Write the map line LINE to the file MAPFILE at MTS line.            *)
1833.  (* number LINE_NO. *)
1834.  (* *)                                                                    *)
1835.  (* ##### *)
1836.  procedure WRITE_MAP_LINE( var MAP : MAPFILE; LINE : MAPSET;
1837.                           LINE_NO : integer );
1838.
1839.
1840.  begin
1841.
1842.      position( MAP, LINE_NO * 1000 );
1843.      writeln ( MAP, LINE );
1844.
1845.
1846.  end  (* WRITE_MAP_LINE *);
1847.
1848.
1849.
1850.  (* ##### *)
1851.  (* *)                                                                    *)
1852.  (* Write the string STR to the present output stream. The string is of *)
1853.  (* length SIZE. If the length is less than one, the operation does not *)
1854.  (* take place. *)
1855.  (* *)                                                                    *)
1856.  (* ##### *)

```

```

1857     procedure WRITE_STRING( STR : INPUT_LINE; SIZE : integer );
1858
1859     var
1860         I           : integer;
1861
1862
1863     begin
1864
1865         if SIZE > 0 then
1866             for I := 1 to SIZE do
1867                 write( STR(I) );
1868
1869
1870     end   (* WRITE_STRING *);
1871
1872
1873
1874     (* ##### *)
1875     (* *)
1876     (* Write the image array IMAGE to the image file IMAGE_FILE. The *)
1877     (* utility function WRITE_IMAGE_LINE is utilized to perform the writes. *)
1878     (* *)
1879     (* ##### *)
1880     procedure WRT_IMAGE( var IMAGE_FILE : IMAGEFILE;
1881                         var IMAGE       : INT_ARRAY );
1882
1883     var
1884         I           : integer;
1885
1886
1887     begin
1888
1889         rewrite( IMAGE_FILE );
1890         for I := DIM(LINE) downto 1 do
1891             WRITE_IMAGE_LINE( IMAGE_FILE, IMAGE(I), 1 );
1892
1893
1894     end   (* WRT_IMAGE *);
1895
1896
1897
1898     (* ##### *)
1899     (* *)
1900     (* Initialize the lookup tables to convert from polar coordinates *)
1901     (* to cartesian coordinates. *)
1902     (* *)
1903     (* ##### *)
1904     procedure XY_INIT;
1905
1906     var
1907         DEG           : int;
1908
1909
1910     begin
1911
1912         for DEG := 0 to 360 do
1913             begin
1914

```



```

1915 XY_LOOKUP(DEG, X) := roundtoshort( cos( DEG * DEG_TO_RAD ) );
1916 XY_LOOKUP(DEG, Y) := roundtoshort( sin( DEG * DEG_TO_RAD ) );
1917
1918 end (* for DEG *);
1919
1920
1921 end (* XY_INIT *);
1922
1923
1924
1925 (* #####
1926 (*
1927 (* Via lookup tables, transform the polar coordinates (RADIUS, THETA)
1928 (* into cartesian coordinates. Return the cartesian form.
1929 (*
1930 (* #####
1931 function XY_TRANSFORM( RADIUS, THETA : short ) : XY_COORD;
1932
1933 var
1934   LO_THETA, HI_THETA      : int;
1935   COORD                   : XY_COORD;
1936   XY                      : XY_COORD;
1937
1938
1939 begin
1940
1941   LO_THETA := trunc( THETA );
1942   HI_THETA := LO_THETA + 1;
1943
1944   if LO_THETA < 0 or HI_THETA > 360 then
1945     SYSTEM_ERROR('In "XY_TRANSFORM": Illegal theta spec. ');
1946   for COORD := X to Y do
1947     XY(COORD) := ( RADIUS *
1948                   ( XY_LOOKUP(LO_THETA, COORD) *
1949                     ( HI_THETA - THETA ) +
1950                     XY_LOOKUP(HI_THETA, COORD) *
1951                     ( THETA - LO_THETA ) ) );
1952
1953   XY_TRANSFORM := XY;
1954
1955 end (* XY_TRANSFORM *);
1956
1957
1958
1959
1960
1961 (* #####
1962 (*
1963 (* Initialize the system and then proceed to displace the original
1964 (* image as a function of its relative height (as specified by the DIM).
1965 (*
1966 (* #####
1967 begin (* MAIN ROUTINE *)
1968
1969   SYS_INIT;
1970
1971   DISPLACE( IMAGE, LMFILE, CENTER(LEFT, X), CENTER(LEFT, Y) );
1972   INTERPOLATE( IMAGE );
1973   WRT_IMAGE ( LFILE, IMAGE );

```

```
1973
1974      DISPLACE( IMAGE, RMFILE, CENTER(RIGHT,X), CENTER(RIGHT,Y) );
1975      INTERPOLATE( IMAGE );
1976      WRT_IMAGE ( RFILE, IMAGE );
1977
1978      end  (* MAIN ROUTINE *).
```

Appendix C

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

```

1  (* ##### *)
2  (* ##### *)
3  (*      A I R B O R N E   S C A N N E R   I M A G E      *)
4  (* ##### *)
5  (* With input of a DTM and a corresponding registered image, a *)
6  (* scanner image of the DTM, according to the intensity values of the *)
7  (* inputted image, is created as well as a visibility map for the *)
8  (* DTM (as seen from a hypothetical airborne scanner device) and a *)
9  (* pixel layover map (as sensed by a hypothetical radar imaging *)
10 (* device). *)
11 (* ##### *)
12 (* ##### *)
13
14  const
15      LINE = 1;    PIXEL = 2;
16      MAX_LINE_LEN = 255;
17      PROMPT = ':';
18      COMPRESSED = true;
19      INTEGRAL = true;
20      READ_REQD = true;
21      RIGHT_HALF = true;
22      LEFT_HALF = false;
23      MIN_INTENSITY = 0;
24      MAX_INTENSITY = 255;
25      MAX_IMAGE_SIZE = 256;
26      MIN_SCAN_ANGLE = 0;
27      MAX_SCAN_ANGLE = 89;
28      UNDEFINED = -1;
29      DEFAULT = -1;
30      DEG_TO_RAD = 0.0174532925;
31      OUT_OF_RANGE = -(maxInt-1);
32
33
34  type
35      MSG_TYPE = ( LINE_PROMPT, PIXEL_PROMPT, UNEXPECTED_BLANK,
36                  UNEXPECTED_EOF, UNREC_CMD, NULL_NO, NON_INTEGRAL_VALUE,
37                  UNIT_0_UNASSIGNED, INVALID_FILE, UNEXPECTED_INPUT_EOF,
38                  INVALID_NO, EOF_NO, EOLN_MSG, NOT_IN_RANGE,
39                  ENTER_ELEVATION, ENTER_NADIR_COORD,
40                  MAP_DESIRE, ASSUMING_NO, ENTER_GRID_SPACING,
41                  ENTER_DEPRESSION_ANGLE, SIDELOOK_TRANSFORM,
42                  UNIT_1_UNASSIGNED, ENTER_SCAN_ANGLE );
43
44      INPUT_LINE = array( 1..MAX_LINE_LEN+1 ) of char;
45      INT_LINE = array( 0..MAX_LINE_LEN+2 ) of int;
46      REAL_LINE = array( 0..MAX_LINE_LEN+2 ) of real;
47      SHORT_LINE = array( 0..MAX_LINE_LEN+2 ) of short;
48
49      INT_ARRAY = array( 0..MAX_LINE_LEN+2 ) of INT_LINE;
50      SHORT_ARRAY = array( 0..MAX_LINE_LEN+2 ) of SHORT_LINE;
51
52
53      VALUE_RANGE = ( POSITIVE, NON_ZERO, INTENSITY, LINE_LENGTH,
54                      IMAGE_SIZE, NON_NEGATIVE, PERCENT, LINES,
55                      SCAN_RANGE );
56      PIXEL_RANGE = 1..MAX_IMAGE_SIZE;
57      MAPSET = set of PIXEL_RANGE;
58

```

```

59      STR_3      = array ( . 1..3 .) of char;
60      STR_4      = array ( . 1..4 .) of char;
61      STR_6      = array ( . 1..6 .) of char;
62      STR_8      = array ( . 1..8 .) of char;
63      STR_44     = array ( . 1..44 .) of char;
64      STR_50     = array ( . 1..50 .) of char;
65
66      IMAGEFILE  = file of INPUT_LINE;
67      DTMFILE    = file of array( 1..(MAX_LINE_LEN+1)*2 ) of char;
68      MAPFILE    = file of MAPSET;
69
70
71
72
73      var
74      IMAGE_CENTER,
75      NADIR_COORD,
76      MAX_ANGLE      : integer;
77      MAX_DEP_ANGLE,
78      MIN_DEP_ANGLE,
79      GRID_SPAC,
80      SCAN_ANGLE,
81      INTERVAL,
82      ELEVATION      : short;
83      SIDE_LOOK,
84      VISIBILITY_MAP,
85      PIXEL_LAYOVER_MAP : boolean;
86      DIM             : array( . 1..2 .) of integer;
87      TAN_LOOKUP      : array( . 0..MAX_SCAN_ANGLE .) of short;
88      IMAGE_FILE, IFILE : IMAGEFILE;
89      DTM_FILE        : DTMFILE;
90      VMFILE, PLMFILE : MAPFILE;
91
92
93
94
95      (* ##### *)
96      (* *)
97      (* FORWARD declarations for the Procedures of the system *)
98      (* *)
99      (* ##### *)
100     procedure BUFFER_INSERT( INTENSITY : integer;
101                             DEGREE      : short;
102                             var ILINE   : INT_LINE;
103                             var FLINE   : SHORT_LINE;
104                             RIGHT_HALF : boolean ); forward;
105     procedure FILES; forward;
106     procedure GET_DIMENSIONS; forward;
107     procedure INTERPOLATE( var ILINE : INT_LINE;
108                           var FACT : SHORT_LINE ); forward;
109     procedure PRINT_MSG( MSG : MSG_TYPE; STRING : INPUT_LINE;
110                        SIZE : integer ); forward;
111     procedure SCANNER_TRANSFORM; forward;
112     procedure SIDE_LOOK_TRANSFORM; forward;
113     procedure SYS_INIT; forward;
114     procedure SYSTEM_ERROR( MESSAGE : STR_50 ); forward;
115     procedure TAN_INIT( SIZE : integer ); forward;
116     procedure WRITE_IMAGE_LINE( var IMAGE : IMAGEFILE;

```

```

117.          LINE : INT_LINE;
118.          START : integer );          forward;
119.      procedure WRITE_MAP_LINE( var MAP : MAPFILE;
120.          LINE : MAPSET );          forward;
121.      procedure WRITE_STRING( STR : INPUT_LINE; SIZE : integer ); forward;
122.
123.
124.      (* ##### *)
125.      (* ##### *)
126.      (*      FORWARD declarations for the Functions of the system      *)
127.      (* ##### *)
128.      (* ##### *)
129.      function ATAN( ARG : short ) : short;          forward;
130.      function CALC_INTERVAL( MAX_ANGLE : integer ) : short;          forward;
131.      function CALC_PIXEL_LAYOVER( DLINE:INT_LINE; VMLINE:MAPSET )
132.          : MAPSET;          forward;
133.      function GET_EOLN( var STR : INPUT_LINE; COMPRESS : boolean)
134.          : integer;          forward;
135.      function GET_NUM( RANGE : VALUE_RANGE; INTEGRAL_VALUE : boolean )
136.          : real;          forward;
137.      function GET_REPLY : boolean;          forward;
138.      function READ_DTM_LINE( var DTM_FILE : DTMFILE; READ_REQD : boolean )
139.          : INT_LINE;          forward;
140.      function READ_IMAGE_LINE( var IMAGE_FILE : IMAGEFILE;
141.          READ_REQD : boolean ) : INT_LINE;          forward;
142.      function READ_STRING : INPUT_LINE;          forward;
143.
144.
145.      (* ##### *)
146.      (* ##### *)
147.      (*      Declarations for FORTRAN routines used by the system      *)
148.      (* ##### *)
149.      (* ##### *)
150.      procedure CALL_MTS( STR : INPUT_LINE; I : integer ); fortran 'CMDNOE';
151.      procedure DEFAULT_FNAME( CMD_STR : INPUT_LINE ); fortran 'FTNCMD';
152.      procedure GET_FNAME( CMD_STR : INPUT_LINE; LEN : integer;
153.          var FNAME : INPUT_LINE ); fortran 'FTNCMD';
154.      procedure SET_PREFIX( NEW : char; LENGTH : integer ); fortran 'SETPFX';
155.
156.
157.
158.
159.
160.      (* ##### *)
161.      (* ##### *)
162.      (*      Return the arctangent of argument ARG. This is accomplished by      *)
163.      (*      utilizing the tangent lookup table TAN_LOOKUP.      *)
164.      (* ##### *)
165.      (* ##### *)
166.      function ATAN( ARG : short ) : short;
167.
168.      var
169.          LO_DEG, HI_DEG, DEG : integer;
170.          LO_TAN, HI_TAN,
171.          LO_FACT, HI_FACT : short;
172.
173.      begin
174.

```

```

175
176     if ARG < 0 then
177         SYSTEM_ERROR( 'In "ATAN"; Invalid argument' );
178
179     DEG := -1;
180     repeat
181         incr( DEG );
182     until (DEG > MAX_ANGLE) or (TAN_LOOKUP(DEG) >= ARG)
183
184     if DEG > MAX_ANGLE then
185         ATAN := OUT_OF_RANGE;
186     else
187         if DEG = 0 then
188             ATAN := 0;
189         else
190             begin
191
192                 LO_DEG := DEG - 1;
193                 HI_DEG := DEG;
194                 LO_TAN := TAN_LOOKUP(LO_DEG);
195                 HI_TAN := TAN_LOOKUP(HI_DEG);
196                 LO_FACT := (HI_TAN - ARG) / (HI_TAN - LO_TAN);
197                 HI_FACT := 1 - LO_FACT;
198                 ATAN := LO_FACT * LO_DEG + HI_FACT * HI_DEG;
199
200             end (* else DEG <= MAX_ANGLE *);
201
202     end (* ATAN *);
203
204
205
206
207     (* ##### *)
208     (* *)
209     (* Insert into the output buffer a pixel with value INTENSITY that is *)
210     (* sensed by the hypothetical airborne scanner device at angle DEGREE *)
211     (* of the scanner sweep. *)
212     (* *)
213     (* ##### *)
214     procedure BUFFER_INSERT( INTENSITY : integer;
215                             DEGREE : short;
216                             var ILINE : INT_LINE;
217                             var FLINE : SHORT_LINE;
218                             RIGHT_HALF : boolean );
219
220     var
221         LO_INDEX, HI_INDEX : integer;
222         LO_INTEN, HI_INTEN,
223         LO_FACT, HI_FACT : short;
224
225
226     begin
227
228         LO_INDEX := trunc( DEGREE / INTERVAL );
229         if (LO_INDEX * INTERVAL < 0) or (LO_INDEX * INTERVAL > MAX_ANGLE) then
230             SYSTEM_ERROR( 'In "BUFFER_INSERT"; Invalid angle specified' );
231
232         HI_INDEX := LO_INDEX + 1;

```

```

233      LO_FACT := HI_INDEX - DEGREE / INTERVAL;
234      HI_FACT := 1 - LO_FACT;
235
236      if RIGHT_HALF then
237      begin
238          LO_INDEX := IMAGE_CENTER + LO_INDEX;
239          HI_INDEX := IMAGE_CENTER + HI_INDEX;
240      end (* if RIGHT_HALF *);
241      else (* LEFT_HALF *)
242      begin
243          LO_INDEX := IMAGE_CENTER - LO_INDEX;
244          HI_INDEX := IMAGE_CENTER - HI_INDEX;
245      end (* else LEFT_HALF *);
246
247      if LO_FACT = 0 then
248      begin
249
250          LO_INTEN := ILINE(LO_INDEX) * FLIN(LO_INDEX) +
251                      INTENSITY * LO_FACT;
252          FLIN(LO_INDEX) := FLIN(LO_INDEX) + LO_FACT;
253          ILINE(LO_INDEX) := round( LO_INTEN / FLIN(LO_INDEX) );
254
255      end (* if LO_FACT = 0 *);
256
257      if HI_FACT = 0 then
258      begin
259
260          HI_INTEN := ILINE(HI_INDEX) * FLIN(HI_INDEX) +
261                      INTENSITY * HI_FACT;
262          FLIN(HI_INDEX) := FLIN(HI_INDEX) + HI_FACT;
263          ILINE(HI_INDEX) := round( HI_INTEN / FLIN(HI_INDEX) );
264
265      end (* if HI_FACT = 0 *);
266
267
268      end (* BUFFER_INSERT *);
269
270
271
272      (* ##### *)
273      (* *)
274      (* Based on the maximum scan angle of the hypothetical airborne scanner *)
275      (* device, calculate the "optimal" interval (in degrees) between *)
276      (* pixels of the output buffer. *)
277      (* *)
278      (* ##### *)
279      function CALC_INTERVAL( MAX_ANGLE : integer ) : short;
280
281      const
282          ANGLE_000 = MIN_SCAN_ANGLE; (* Minimum angle permitted *)
283          ANGLE_010 = 10; (* Max angle for .10 intervals *)
284          ANGLE_025 = 30; (* Max angle for .25 intervals *)
285          ANGLE_050 = 60; (* Max angle for .50 intervals *)
286          ANGLE_100 = MAX_SCAN_ANGLE; (* Max angle for 1.0 intervals *)
287
288
289      begin
290

```



```

291      if (MAX_ANGLE >= ANGLE_000)      and (MAX_ANGLE <= ANGLE_010) then
292          CALC_INTERVAL := 0.10s;
293      else if (MAX_ANGLE > ANGLE_010) and (MAX_ANGLE <= ANGLE_025) then
294          CALC_INTERVAL := 0.25s;
295      else if (MAX_ANGLE > ANGLE_025) and (MAX_ANGLE <= ANGLE_050) then
296          CALC_INTERVAL := 0.50s;
297      else if (MAX_ANGLE > ANGLE_050) and (MAX_ANGLE <= ANGLE_100) then
298          CALC_INTERVAL := 1.00s;
299      else
300          SYSTEM_ERROR( 'In "CALC_INTERVAL"; Invalid angle specified' );
301
302
303      end      (* CALC_INTERVAL *);
304
305
306
307      (* ##### *)
308      (* *)
309      (* Calculate any pixel layovers (i.e. pixels of same distance from the *)
310      (* radar imaging device) in the DTM line DLINE. Return a map line *)
311      (* indicating where such pixels are located in DLINE. *)
312      (* *)
313      (* ##### *)
314      function CALC_PIXEL_LAYOVER( DLINE : INT_LINE; VMLINE : MAPSET ) : MAPSET;
315
316      var
317          I, J, K, MAP_COORD      : integer;
318          PIXEL_LAYOVER           : boolean;
319          DISTANCE                 : SHORT_LINE;
320          PLMLINE                  : MAPSET;
321
322
323      begin
324
325          for I := 1 to DIM(PIXEL) do
326              if (MAX_IMAGE_SIZE - I) in VMLINE then
327                  DISTANCE(I) := UNDEFINED;
328              else
329                  DISTANCE(I) := sqr( (NADIR_COORD - I) * GRID_SPAC ) +
330                                  sqr( ELEVATION - DLINE(I) );
331
332          PLMLINE := ( . );
333          for I := 1 to DIM(PIXEL) do
334              begin
335
336                  MAP_COORD := MAX_IMAGE_SIZE - I;
337                  if (MAP_COORD not in VMLINE) then
338                      begin
339
340                          PIXEL_LAYOVER := false;
341                          for J := I+1 to DIM(PIXEL)-1 do
342                              if DISTANCE(J) <> UNDEFINED then
343                                  begin
344
345                                      K := J + 1;
346                                      while (K <= DIM(PIXEL)) and
347                                          (DISTANCE(K) = UNDEFINED) do
348                                          incr( K );

```

```

349         if (K <= DIM(PIXEL)) and
350             (DISTANCE(I) <= max( DISTANCE(J), DISTANCE(K) )) and
351             (DISTANCE(I) >= min( DISTANCE(J), DISTANCE(K) )) then
352             begin
353                 PIXEL_LAYOVER := true;
354                 if abs( DISTANCE(I) - DISTANCE(J) ) >
355                     abs( DISTANCE(I) - DISTANCE(K) ) then
356                     PLMLINE := PLMLINE + ( . (MAX_IMAGE_SIZE-K) . );
357                 else
358                     PLMLINE := PLMLINE + ( . (MAX_IMAGE_SIZE-J) . );
359             end (* if (K <= DIM(PIXEL)) ... *);
360         end (* if DISTANCE(J) <> UNDEFINED *);
361     if PIXEL_LAYOVER then
362         PLMLINE := PLMLINE + ( . MAP_COORD . );
363
364     end (* if (MAP_COORD not in VMLINE) *);
365
366 end (* for I *);
367
368 CALC_PIXEL_LAYOVER := PLMLINE;
369
370
371
372 end (* CALC_PIXEL_LAYOVER *);
373
374
375
376 (* ##### *)
377 (* *)
378 (* Set up and attach the files necessary for the system. Unit 0 is the *)
379 (* assumed input for the original DTM. If it is not assigned on the MTS *)
380 (* RUN command, the file name will be prompted for. Unit 1 is assumed *)
381 (* input for the original registered image. Its file name will also be *)
382 (* prompted for if not assigned. The file assigned to unit 10 will be *)
383 (* used for the output of the scanner image. The file assigned to unit *)
384 (* 12 will be used for the output of the visibility map of the DTM, if *)
385 (* requested. The file assigned to unit 13 will be used for output of *)
386 (* the pixel layover map of the DTM, if requested. If these units are *)
387 (* not attached, they will default to MTS temporary files -IMAGE#, *)
388 (* -VMAP#, and -PLMAP# respectively. *)
389 (* *)
390 (* ##### *)
391 procedure FILES;
392
393     var
394         DTMFNAME, IMGFNAME : INPUT_LINE;
395
396
397     begin
398
399
400         DTMFNAME := ' ';
401         DEFAULT_FNAME( 'DEFAULT O=*DUMMY*;' );
402         GET_FNAME( 'QUERY FNAME O;', O, DTMFNAME );
403
404         if DTMFNAME = '*DUMMY*' then
405             begin
406

```

```

407      PRINT_MSG( UNIT_0_UNASSIGNED, '', 0 );
408      DTMFNAME := READ_STRING;
409      while DTMFNAME = ' ' do
410          begin
411              PRINT_MSG( INVALID_FILE, '', 0 );
412              DTMFNAME := READ_STRING;
413              if DTMFNAME = 'CANCEL' or DTMFNAME = 'HALT' then
414                  halt;
415              end (* while DTMFNAME = ' ' *);
416
417      end (* if DTMFNAME = '*DUMMY*' *);
418
419
420      IMGFNAME := ' ';
421      DEFAULT_FNAME( 'DEFAULT 1=*DUMMY*;' );
422      GET_FNAME( 'QUERY FDNAME 1;', 0, IMGFNAME );
423
424      if IMGFNAME = '*DUMMY*' then
425          begin
426
427              PRINT_MSG( UNIT_1_UNASSIGNED, '', 0 );
428              IMGFNAME := READ_STRING;
429              while IMGFNAME = ' ' do
430                  begin
431                      PRINT_MSG( INVALID_FILE, '', 0 );
432                      IMGFNAME := READ_STRING;
433                      if IMGFNAME = 'CANCEL' or IMGFNAME = 'HALT' then
434                          halt;
435                      end (* while IMGFNAME = ' ' *);
436
437              end (* if IMGFNAME = '*DUMMY*' *);
438
439
440      if VISIBILITY_MAP then
441          begin
442              DEFAULT_FNAME( 'DEFAULT 12=-VMAP#;' );
443              rewrite( VMFILE, 12 );
444              end (* if VISIBILITY_MAP *);
445
446      if PIXEL_LAYOVER_MAP then
447          begin
448              DEFAULT_FNAME( 'DEFAULT 13=-PLMAP#;' );
449              rewrite( PLMFILE, 13 );
450              end (* if PIXEL_LAYOVER_MAP *);
451
452      DEFAULT_FNAME( 'DEFAULT 10=-IMAGE#;' );
453      reset ( DTM_FILE, DTMFNAME );
454      reset ( IMAGE_FILE, IMGFNAME );
455      rewrite( IFILE, 10 );
456
457
458      end (* FILES *);
459
460
461
462      (* ##### *)
463      (* *)
464      (* Prompt for and get the dimensions of the DTM. *)

```

```

465      (*
466      (* #####
467      procedure GET_DIMENSIONS;
468
469      begin
470
471          PRINT_MSG( LINE_PROMPT, ' ', 0 );
472          DIM(LINE) := trunc( GET_NUM( IMAGE_SIZE, INTEGRAL ) );
473
474          PRINT_MSG( PIXEL_PROMPT, ' ', 0 );
475          DIM(PIXEL) := trunc( GET_NUM( IMAGE_SIZE, INTEGRAL ) );
476
477      end      (* GET_DIMENSIONS *);
478
479
480
481      (* #####
482      (*
483      (* Read and return the end of an input line. If COMPRESS is true then
484      (* remove multiple embedded blanks, otherwise return the actual input
485      (* line. The length of the input line is also returned.
486      (*
487      (* #####
488      function GET_EOLN( var STR:INPUT_LINE; COMPRESS:boolean ) : integer;
489
490      var
491          I          : integer;
492          LAST_CHAR_BLANK : boolean;
493          CHR         : char;
494
495
496      begin
497          I := 0;
498          LAST_CHAR_BLANK := false;
499
500          while not eof( INPUT ) and not eoln( INPUT ) do
501              begin
502                  read( CHR );
503                  if CHR = ' ' or not COMPRESS then
504                      begin
505                          if LAST_CHAR_BLANK and I>0 then
506                              begin
507                                  I := I + 1;
508                                  STR(I) := ' ';
509                              end;
510                                  I := I + 1;
511                                  STR(I) := CHR;
512                                  LAST_CHAR_BLANK := false;
513                              end;
514                          else
515                              LAST_CHAR_BLANK := true;
516                          end;
517
518                  if not COMPRESS and I>0 then
519                      GET_EOLN := I - 1;
520                  else
521                      GET_EOLN := I
522

```

```

523       end      (* GET_EOLN  *);
524
525
526
527
528  (* ##### *)
529  (* ##### *)
530  (* Return the numerical value of a string in the input stream. If      *)
531  (* INTEGRAL_VALUE is true then the string must represent an integer,    *)
532  (* otherwise a real number. The numerical value must also be in the     *)
533  (* range specified by RANGE.                                           *)
534  (* ##### *)
535  (* ##### *)
536  function GET_NUM( RANGE : VALUE_RANGE;
537                  INTEGRAL_VALUE : boolean ) : real;
538
539      var
540          STR          : INPUT_LINE;
541          VALID        : boolean;
542          VALU         : real;
543          I, J, MULT   : integer;
544
545      begin
546          repeat
547              VALID := true;
548              read( STR );
549              if eof( INPUT ) then      (* CHECK FOR EOF *)
550                  begin
551                      VALID := false;
552                      PRINT_MSG( EOF_NO, ' ', 0 )
553                  end;
554              else
555                  begin
556                      VALU := 0.0;      (* GET INTEGRAL PART *)
557                      I := 1;
558                      MULT := 1;
559                      while I <= MAX_LINE_LEN and STR(I) = ' ' do
560                          I := I + 1;
561                      if STR(I) = '+' or STR(I) = '-' then
562                          begin
563                              if STR(I) = '-' then
564                                  MULT := -1;
565                              I := I + 1;
566                          end;
567                      while STR(I) in ( '0'..'9' ) do
568                          begin
569                              VALU := 10.0 * VALU + ord( STR(I) ) - ord('0');
570                              I := I + 1;
571                          end;
572                      if STR(I) = '.' then
573                          begin
574                              J := 10;
575                              I := I + 1;
576                              while STR(I) in ( '0'..'9' ) do
577                                  begin
578                                      VALU := VALU + ( (ord(STR(I)) - ord('0'))/J);
579                                      I := I + 1;

```

```

581      J := J * 10
582    end;
583  end;
584  if STR(I) = ' ' or ( eoln(INPUT) and I = MAX_LINE_LEN+1 ) then
585    begin
586      (* INVALID NUMBER *)
587      VALID := false;
588      if STR(I) = ' ' then
589        PRINT_MSG( INVALID_NO, STR(I), 1 )
590      else
591        if RANGE in ( . PERCENT . ) then
592          begin
593            VALID := true;
594            MULT := 1;
595            VALU := DEFAULT;
596            end (* if RANGE in *);
597          else
598            PRINT_MSG( NULL_NO, ' ', 0 )
599          end;
600        else if INTEGRAL_VALUE and trunc(VALU) = VALU then
601          begin
602            VALID := false;
603            PRINT_MSG( NON_INTEGRAL_VALUE, ' ', 0 )
604          end;
605        else if STR(I) = ' ' and ( STR(I-1)='-' or STR(I-1)='+' ) then
606          begin
607            VALID := false;
608            PRINT_MSG( UNEXPECTED_BLANK, ' ', 0 )
609          end;
610        else if (RANGE = POSITIVE and (MULT < 0 or VALU = 0.0)) or
611              (RANGE = IMAGE_SIZE and (MULT < 0 or
612              VALU < 1 or VALU > MAX_IMAGE_SIZE)) or
613              (RANGE = LINES and (MULT < 0 or
614              VALU < 1 or VALU > DIM(LINE))) or
615              (RANGE = LINE_LENGTH and (MULT < 0 or
616              VALU < 1 or VALU > DIM(PIXEL))) or
617              (RANGE = INTENSITY and (MULT < 0 or
618              VALU < MIN_INTENSITY or
619              VALU > MAX_INTENSITY)) or
620              (RANGE = SCAN_RANGE and (MULT < 0 or
621              VALU < MIN_SCAN_ANGLE or
622              VALU > (2 * MAX_SCAN_ANGLE))) or
623              (RANGE = NON_NEGATIVE and MULT < 0) or
624              (RANGE = PERCENT and (MULT < 0 or
625              VALU < 0 or VALU > 100)) or
626              (RANGE = NON_ZERO and VALU = 0.0) then
627          begin
628            VALID := false;
629            PRINT_MSG( NOT_IN_RANGE, ' ', 0 )
630          end;
631        end;
632      until VALID;
633      GET_NUM := VALU * MULT
634
635    end (* GET_NUM *);
636
637
638

```

```

639
640  (* ##### *)
641  (* ##### *)
642  (* Read in a requested reply (via GET_EOLN). If first letter of reply *)
643  (* is 'Y' or 'y' then return true, otherwise return false. *)
644  (* ##### *)
645  (* ##### *)
646  function GET_REPLY : boolean;
647
648  const
649      YES_1 = 'Y'; YES_2 = 'y';
650      NO_1  = 'N'; NO_2  = 'n';
651
652  var
653      STR      : INPUT_LINE;
654      I        : integer;
655
656
657  begin
658
659      I := GET_EOLN( STR, COMPRESSED );
660      if (STR(I) = YES_1) and (STR(I) = YES_2) and
661          (STR(I) = NO_1) and (STR(I) = NO_2) then
662          PRINT_MSG( ASSUMING_NO, '', 0 );
663      GET_REPLY := (STR(I) = YES_1) or (STR(I) = YES_2);
664
665
666  end  (* GET_REPLY *);
667
668
669
670  (* ##### *)
671  (* ##### *)
672  (* Interpolate all undefined pixel values between the first defined *)
673  (* pixel of the image line - ILINE - and the last defined pixel of the *)
674  (* image line. The new values are calculated as a function of the *)
675  (* values of the nearest defined pixels (i.e. the ones on either side *)
676  (* of the undefined pixel). *)
677  (* ##### *)
678  (* ##### *)
679  procedure INTERPOLATE( var ILINE : INT_LINE;
680                        var FACT : SHORT_LINE );
681
682  var
683      X, XX, XEND, START,
684      STOP, INCREMENT, I : integer;
685
686
687  begin
688
689      X := 1;
690      while FACT(X) = 0 and X < (MAX_LINE_LEN+2) do
691          incr( X );
692
693      XEND := MAX_LINE_LEN + 2;
694      while FACT(XEND) = 0 and XEND >= 1 do
695          decr( XEND );
696

```

```

697      .incr( X );
698      while X < XEND do
699          begin
700
701              if FACT(X) = 0 then
702                  begin
703
704                      I          := ILINE(X-1);
705                      START      := X;
706                      while FACT(X) = 0 do
707                          incr( X );
708                      STOP      := X - 1;
709                      INCREMENT := round( (ILINE(X) - ILINE(START-1)) / (X - START + 1) );
710                      for XX := START to STOP do
711                          begin
712                              I := I + INCREMENT;
713                              ILINE(XX) := round( (I + ILINE(XX) * FACT(XX)) /
714                                                    (1.0 + FACT(XX)) );
715                          end (* for XX *);
716                      end (* if FACT(X) < 1 *);
717                      incr( X );
718
719              end (* while X < XEND *);
720
721      end (* INTERPOLATE *);
722
723      (* ***** *)
724      (* ***** *)
725      (* Print the message specified by MSG to the output stream. The ***** *)
726      (* parameters STRING and SIZE may be involved in the details of ***** *)
727      (* that message. ***** *)
728      (* ***** *)
729      (* ***** *)
730      procedure PRINT_MSG( MSG : MSG_TYPE; STRING : INPUT_LINE;
731                          SIZE : integer );
732
733      const
734          MSG_PROMPT = ':';
735
736      var
737          I,J          : integer;
738          STR          : INPUT_LINE;
739
740      begin
741          SET_PREFIX( MSG_PROMPT, 1 );
742
743          if MSG in ( LINE_PROMPT, UNIT_O_UNASSIGNED, MAP_DESIRE,
744                    ENTER_GRID_SPACING,
745                    ENTER_NADIR_COORD, ENTER_SCAN_ANGLE,
746                    ENTER_DEPRESSION_ANGLE, SIDELOOK_TRANSFORM,
747                    UNIT_I_UNASSIGNED, ENTER_ELEVATION ) then
748              writeln;

```



```

755
756      case MSG of
757          UNEXPECTED_EOF :
758              writeln(' Unexpected EOF -- Ignored');
759          UNREC_CMD :
760              begin
761                  writeln(' Invalid command -- Input line ignored');
762                  if neoln( INPUT ) then
763                      I := GET_EOLN( STR, COMPRESSED )
764              end;
765          EOLN_MSG :
766              begin
767                  I := GET_EOLN( STR, COMPRESSED );
768                  if I = 0 then
769                      begin
770                          write(' Invalid character(s): ');
771                          WRITE_STRING( STR, I );
772                          writeln(' -- Ignored')
773                      end
774              end;
775          LINE_PROMPT :
776              writeln('&Enter number of lines of DTM and IMAGE (Integer {1-256})');
777          PIXEL_PROMPT :
778              writeln('&Enter number of pixels per line          (Integer {1-256})');
779          EOF_NO :
780              writeln('&Unexpected EOF. Enter number');
781          NULL_NO :
782              writeln('&Not expecting null line -- Ignored; Enter number' );
783          NON_INTEGRAL_VALUE:
784              writeln('&Non-integral value, Re-enter number');
785          UNEXPECTED_BLANK:
786              writeln('&Unexpected blank, Re-enter number');
787          NOT_IN_RANGE :
788              writeln('&Out of range, Re-enter number');
789          UNIT_0_UNASSIGNED:
790              writeln(' Logical unit 0 has not been assigned.',
791                      EOL, '& Enter FDname of location of DTM ');
792          UNIT_1_UNASSIGNED:
793              writeln(' Logical unit 1 has not been assigned.',
794                      EOL, '& Enter FDname of location of IMAGE ');
795          INVALID_FILE :
796              writeln(' Invalid FDname. Enter again or', EOL,
797                      '&Enter "CANCEL" to terminate program');
798          UNEXPECTED_INPUT_EOF:
799              writeln(' Unexpected EOF encountered on input',
800                      ' -- DTM size updated');
801          INVALID_NO :
802              begin
803                  writeln('&Invalid character, ', STRING(I), ', ', Re-enter number');
804                  I := GET_EOLN( STR, COMPRESSED )
805              end;
806          ASSUMING_NO :
807              writeln(' Assuming "NO".');
808          MAP_DESIRE :
809              begin
810                  write('&Do you want a ');
811                  WRITE_STRING( STRING, SIZE );
812                  writeln(' map created? {y/n}');

```

```

813         end;
814     SIDELOOK_TRANSFORM:
815         writeln('&Do you want a side-looking transform? {y/n}');
816     ENTER_DEPRESSION_ANGLE:
817         begin
818             write(' Enter the ');
819             WRITE_STRING( STRING, SIZE );
820             writeln(' depression angle, from the horizontal,', EOL,
821                 '& of the sensing system (degrees {',
822                 MIN_SCAN_ANGLE:0,'-',MAX_SCAN_ANGLE:0,'}')');
823         end;
824     ENTER_NADIR_COORD:
825         writeln(' Enter the X (pixel) coordinate of the nadir',
826             EOL, '& of the airborne scanner system (Integer {1-',
827             DIM(PIXEL):0,'}')');
828     ENTER_SCAN_ANGLE:
829         writeln('&Enter the scan angle of the airborne scanner system (degrees {',
830             MIN_SCAN_ANGLE:0,'-',(2*MAX_SCAN_ANGLE):0,'}')');
831     ENTER_GRID_SPACING:
832         writeln('&Enter DTM grid spacing (in DTM units)');
833     ENTER_ELEVATION :
834         writeln('&Enter elevation of airborne scanner system (in DTM units)');
835
836     end (* case *);
837
838     if MSG in ( LINE_PROMPT, PIXEL_PROMPT, INVALID_NO,
839         NON_INTEGRAL_VALUE, UNEXPECTED_BLANK, NOT_IN_RANGE,
840         UNIT_0_UNASSIGNED, UNIT_1_UNASSIGNED, INVALID_FILE,
841         ENTER_ELEVATION, ENTER_NADIR_COORD,
842         ENTER_SCAN_ANGLE, MAP_DESIRE, ENTER_GRID_SPACING,
843         ENTER_DEPRESSION_ANGLE, SIDELOOK_TRANSFORM,
844         EOF_NO, NULL_NO ) then
845         readln;
846
847         SET_PREFIX( PROMPT, 1 )
848
849
850     end (* PRINT_MSG *);
851
852
853
854     (* ##### *)
855     (* *)
856     (* If READ_REQD is true then read in a line from the file DTM_FILE, *)
857     (* otherwise perform the subsequent operations on the present input *)
858     (* buffer. Convert the binary form of the buffer into integer form *)
859     (* assuming two bytes per pixel. *)
860     (* *)
861     (* ##### *)
862     function READ_DTM_LINE( var DTM_FILE : DTMFILE; READ_REQD : boolean ) : INT_LINE;
863
864     var
865         LINE          : INT_LINE;
866         I              : integer;
867
868
869     begin
870

```

```

871 if READ_REQD then
872   get(_DTM_FILE);
873   for I := 1 to DIM(PIXEL) do
874     LINE(I) := int( DTM_FILE@((I-1)*2 + 1) ) * 256 +
875     int( DTM_FILE@((I-1)*2 + 2) );
876   READ_DTM_LINE := LINE;
877
878
879   end (* READ_DTM_LINE *);
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928

```

```

(* #####
(*
(* 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;

```

```

929      repeat
930          incr( I );
931          read( STR(I) );
932          if STR(I) = ' ' and I = 1 then
933              I := 0;
934          until eof(INPUT) or eoln(INPUT) or (I /= 0 and STR(I) = ' ')
935              or I = MAX_LINE_LEN;
936
937      READ_STRING := STR;
938
939
940  end    (* READ_STRING *);
941
942
943
944
945  (* ##### *)
946  (* *)
947  (* Transform the orthographic image, line by line, into a scanner image *)
948  (* (geometrically, not radiometrically) as seen by a hypothetical *)
949  (* airborne scanner system. Calculate a corresponding visibility map *)
950  (* and a pixel layover map simultaneously, if either is requested. *)
951  (* *)
952  (* ##### *)
953  procedure SCANNER_TRANSFORM;
954
955      var
956          I, J, RADIUS      : integer;
957          ANGLE,
958          PREV_TAN, CURR_TAN : short;
959          VISIBLE           : boolean;
960          DLINE, ILINE,
961          SCANNER_ILINE     : INT_LINE;
962          SCANNER_FLINE     : SHORT_LINE;
963          VMLINE, PLMLINE   : MAPSET;
964
965
966      begin
967
968          ILINE := READ_IMAGE_LINE( IMAGE_FILE, not READ_REQD );
969          DLINE := READ_DTM_LINE ( DTM_FILE,   not READ_REQD );
970
971          for I := 1 to DIM(LINE) do
972              begin
973
974                  VMLINE := ( . );
975                  for J := 0 to MAX_LINE_LEN+2 do
976                      begin
977                          SCANNER_ILINE(J) := MIN_INTENSITY;
978                          SCANNER_FLINE(J) := 0;
979                      end (* for J *);
980
981                  SCANNER_ILINE(IMAGE_CENTER) := ILINE(NADIR_COORD);
982                  SCANNER_FLINE(IMAGE_CENTER) := 1.0s;
983
984
985                  PREV_TAN := 0.0s;
986                  for J := NADIR_COORD-1 downto 1 do

```

```

987      begin
988
989      VISIBLE := true;
990      RADIUS  := abs( NADIR_COORD - J );
991      CURR_TAN := RADIUS * GRID_SPAC / (ELEVATION - DLINE(J));
992      if CURR_TAN > PREV_TAN then
993      begin
994
995          ANGLE := ATAN( CURR_TAN );
996          if (ANGLE /= OUT_OF_RANGE) and (ANGLE <= SCAN_ANGLE) then
997              BUFFER_INSERT( ILINE(J), ANGLE, SCANNER_ILINE,
998                             SCANNER_FLINE, LEFT_HALF );
999          else
1000              VMLINE := VMLINE + ( (MAX_IMAGE_SIZE-J) );
1001              PREV_TAN := CURR_TAN;
1002
1003          end (* if CURR_TAN > PREV_TAN *);
1004      else
1005          VMLINE := VMLINE + ( (MAX_IMAGE_SIZE-J) );
1006
1007      end (* for J *);
1008
1009      PREV_TAN := 0.0s;
1010      for J := NADIR_COORD+1 to DIM(PIXEL) do
1011      begin
1012
1013          VISIBLE := true;
1014          RADIUS  := abs( NADIR_COORD - J );
1015          CURR_TAN := RADIUS * GRID_SPAC / (ELEVATION - DLINE(J));
1016          if CURR_TAN > PREV_TAN then
1017          begin
1018
1019              ANGLE := ATAN( CURR_TAN );
1020              if (ANGLE /= OUT_OF_RANGE) and (ANGLE <= SCAN_ANGLE) then
1021                  BUFFER_INSERT( ILINE(J), ANGLE, SCANNER_ILINE,
1022                                 SCANNER_FLINE, RIGHT_HALF );
1023              else
1024                  VMLINE := VMLINE + ( (MAX_IMAGE_SIZE-J) );
1025                  PREV_TAN := CURR_TAN;
1026
1027              end (* if CURR_TAN > PREV_TAN *);
1028          else
1029              VMLINE := VMLINE + ( (MAX_IMAGE_SIZE-J) );
1030
1031          end (* for J *);
1032
1033          SCANNER_FLINE(IMAGE_CENTER) := 0.0s;
1034          INTERPOLATE( SCANNER_ILINE, SCANNER_FLINE );
1035
1036          WRITE_IMAGE_LINE( IFILE, SCANNER_ILINE, 1 );
1037
1038          if VISIBILITY_MAP then
1039              WRITE_MAP_LINE( VMFILE, VMLINE );
1040
1041          if PIXEL_LAYOVER_MAP then
1042              begin

```

```

1045          PLMLINE := CALC_PIXEL_LAYOVER( DLINE, VMLINE );
1046          WRITE_MAP_LINE( PLMFILE, PLMLINE );
1047      end (* if PIXEL_LAYOVER_MAP *);
1048
1049      if I < DIM(LINE) then
1050      begin
1051          ILINE := READ_IMAGE_LINE( IMAGE_FILE, READ_REQD );
1052          DLINE := READ_DTM_LINE ( DTM_FILE, READ_REQD );
1053      end (* if I < DIM(LINE) *);
1054
1055
1056      end (* for I *);
1057
1058
1059      end (* SCANNER_TRANSFORM *);
1060
1061
1062
1063
1064      (* ##### *)
1065      (* *)
1066      (* Transform the orthographic image, line by line, into a side-looking *)
1067      (* scanner image - as is the type of image produced by a SLAR system - *)
1068      (* (geometrically, not radiometrically) as seen by a hypothetical *)
1069      (* airborne sensing system. Calculate a corresponding visibility map *)
1070      (* and a pixel layover map simultaneously, if either is requested. *)
1071      (* *)
1072      (* ##### *)
1073      procedure SIDE_LOOK_TRANSFORM;
1074
1075      var
1076          I, J, RADIUS      : integer;
1077          ANGLE,
1078          PREV_TAN, CURR_TAN : short;
1079          VISIBLE           : boolean;
1080          DLINE, ILINE,
1081          SL_ILINE          : INT_LINE;
1082          SL_FLINE          : SHORT_LINE;
1083          VMLINE, PLMLINE   : MAPSET;
1084
1085
1086      begin
1087
1088          ILINE := READ_IMAGE_LINE( IMAGE_FILE, not READ_REQD );
1089          DLINE := READ_DTM_LINE ( DTM_FILE, not READ_REQD );
1090
1091          for I := 1 to DIM(LINE) do
1092              begin
1093
1094                  VMLINE := ( . );
1095                  for J := 0 to MAX_LINE_LEN+2 do
1096                      begin
1097                          SL_ILINE(J) := MIN_INTENSITY;
1098                          SL_FLINE(J) := 0;
1099                      end (* for J *);
1100
1101                  PREV_TAN := 0.0s;
1102                  for J := 1 to DIM(PIXEL) do

```

```

1103      begin
1104
1105          VISIBLE := true;
1106          RADIUS  := abs( NADIR_COORD - J );
1107          CURR_TAN := RADIUS * GRID_SPAC / (ELEVATION - DLINE(J));
1108          if CURR_TAN > PREV_TAN then
1109              begin
1110
1111                  ANGLE := ATAN( CURR_TAN );
1112                  if (ANGLE /= OUT_OF_RANGE) and
1113                     (ANGLE <= (90-MIN_DEP_ANGLE)) and
1114                     (ANGLE >= (90-MAX_DEP_ANGLE)) then
1115                      BUFFER_INSERT( ILINE(J), (ANGLE-(90-MAX_DEP_ANGLE)),
1116                                   SL_ILINE, SL_FLINE, RIGHT_HALF );
1117                  else
1118                      VMLINE := VMLINE + (. (MAX_IMAGE_SIZE-J) .);
1119                      PREV_TAN := CURR_TAN;
1120
1121                  end (* if CURR_TAN > PREV_TAN *);
1122              else
1123                  VMLINE := VMLINE + (. (MAX_IMAGE_SIZE-J) .);
1124
1125              end (* for J *);
1126
1127          INTERPOLATE( SL_ILINE, SL_FLINE );
1128
1129          WRITE_IMAGE_LINE( IFILE, SL_ILINE, 1 );
1130
1131          if VISIBILITY_MAP then
1132              WRITE_MAP_LINE( VMFILE, VMLINE );
1133
1134          if PIXEL_LAYOVER_MAP then
1135              begin
1136                  PLMLINE := CALC_PIXEL_LAYOVER( DLINE, VMLINE );
1137                  WRITE_MAP_LINE( PLMFILE, PLMLINE );
1138              end (* if PIXEL_LAYOVER_MAP *);
1139
1140          if I < DIM(LINE) then
1141              begin
1142                  ILINE := READ_IMAGE_LINE( IMAGE_FILE, READ_REQD );
1143                  DLINE := READ_DTM_LINE ( DTM_FILE, READ_REQD );
1144              end (* if I < DIM(LINE) *);
1145
1146          end (* for I *);
1147
1148      end (* SIDE_LOOK_TRANSFORM *);
1149
1150  end (* SIDE_LOOK_TRANSFORM *);
1151
1152  (* ***** *)
1153  (* ***** *)
1154  (* ***** *)
1155  (* ***** *)
1156  (* ***** *)
1157  (* Initialize the system by setting up the files, getting the dimensions *)
1158  (* of the DTM, and reading the necessary system parameters. *)
1159  (* ***** *)
1160  (* ***** *)

```

```

1161.      procedure SYS_INIT;
1162.
1163.      var
1164.          CENTRAL_ANGLE      : real;
1165.          I, J                : integer;
1166.
1167.
1168.      begin
1169.
1170.          SET_PREFIX( PROMPT, 1 );
1171.          GET_DIMENSIONS;
1172.
1173.          PRINT_MSG( ENTER_GRID_SPACING, '', 0 );
1174.          GRID_SPAC      := roundtoshort( GET_NUM( POSITIVE, NOT INTEGRAL ) );
1175.
1176.          PRINT_MSG( ENTER_ELEVATION, '', 0 );
1177.          ELEVATION      := roundtoshort( GET_NUM( POSITIVE, NOT INTEGRAL ) );
1178.
1179.          PRINT_MSG( SIDELOOK_TRANSFORM, '', 0 );
1180.          SIDE_LOOK      := GET_REPLY;
1181.
1182.          if SIDE_LOOK then
1183.              begin
1184.
1185.                  PRINT_MSG( ENTER_DEPRESSION_ANGLE, 'maximum', 7 );
1186.                  MAX_DEP_ANGLE := roundtoshort( GET_NUM( SCAN_RANGE,
1187.                                                                NOT INTEGRAL ) );
1188.
1189.                  PRINT_MSG( ENTER_DEPRESSION_ANGLE, 'minimum', 7 );
1190.                  MIN_DEP_ANGLE := roundtoshort( GET_NUM( SCAN_RANGE,
1191.                                                                NOT INTEGRAL ) );
1192.
1193.                  CENTRAL_ANGLE := (90 - MAX_DEP_ANGLE + ((MAX_DEP_ANGLE -
1194.                                                                MIN_DEP_ANGLE) * 0.80s)) * DEG_TO_RAD;
1195.
1196.                  NADIR_COORD  := trunc( DIM(PIXEL)/2 ) -
1197.                                     round( tan( CENTRAL_ANGLE ) * ELEVATION / GRID_SPAC );
1198.                  MAX_ANGLE    := 90 - trunc( MIN_DEP_ANGLE ) + 1;
1199.                  IMAGE_CENTER := 0;
1200.
1201.              end (* if SIDE_LOOK *);
1202.          else
1203.              begin
1204.
1205.                  PRINT_MSG( ENTER_SCAN_ANGLE, '', 0 );
1206.                  SCAN_ANGLE := roundtoshort( GET_NUM( SCAN_RANGE, NOT INTEGRAL ) / 2 );
1207.
1208.                  PRINT_MSG( ENTER_NADIR_COORD, '', 0 );
1209.                  NADIR_COORD := trunc( GET_NUM( LINE_LENGTH, INTEGRAL ) );
1210.
1211.                  MAX_ANGLE := trunc( SCAN_ANGLE + 1 );
1212.                  IMAGE_CENTER := trunc( MAX_IMAGE_SIZE / 2 );
1213.
1214.              end (* else not SIDE_LOOK *);
1215.
1216.          if MAX_ANGLE > MAX_SCAN_ANGLE then
1217.              MAX_ANGLE := MAX_SCAN_ANGLE;
1218.

```



```

1219      TAN_INIT( MAX_ANGLE );
1220      if SIDE_LOOK then
1221          INTERVAL := CALC_INTERVAL( round( (MAX_DEP_ANGLE-
1222              MIN_DEP_ANGLE)/2 ) );
1223      else
1224          INTERVAL := CALC_INTERVAL( MAX_ANGLE );
1225
1226      PRINT_MSG( MAP_DESIRE, ' visibility ', 13 );
1227      VISIBILITY_MAP := GET_REPLY;
1228
1229      PRINT_MSG( MAP_DESIRE, 'pixel layover', 13 );
1230      PIXEL_LAYOVER_MAP := GET_REPLY;
1231
1232      FILES;
1233
1234
1235      end  (* SYSTEM_INITIALIZATION *);
1236
1237
1238
1239      (* ##### *)
1240      (* ##### *)
1241      (* If a system error occurs, output the message MESSAGE and halt the *)
1242      (* execution of the system. *)
1243      (* ##### *)
1244      (* ##### *)
1245      procedure SYSTEM_ERROR( MESSAGE : STR_50 );
1246
1247          begin
1248
1249              writeln( ' ==> SYSTEM ERROR ==> ', MESSAGE );
1250              halt;
1251
1252
1253          end  (* SYSTEM_ERROR *);
1254
1255
1256
1257
1258      (* ##### *)
1259      (* ##### *)
1260      (* Initialize the tangent lookup table, TAN_LOOKUP. *)
1261      (* ##### *)
1262      (* ##### *)
1263      procedure TAN_INIT( SIZE : integer );
1264
1265          var
1266              DEG          : integer;
1267
1268
1269          begin
1270
1271              if SIZE > MAX_SCAN_ANGLE then
1272                  SYSTEM_ERROR( 'In "TAN_INIT": Invalid size specified' );
1273
1274              for DEG := 0 to SIZE do
1275                  if DEG = 90 then
1276                      TAN_LOOKUP(DEG) := UNDEFINED;

```

```

LISTING OF FILE LIFE:scanner.vfs.s      12:54 P.M.  JUNE 12, 1982  ID=LIFE

1277      else
1278      TAN_LOOKUP(DEG) := roundtoshort( tan( DEG * DEG_TO_RAD ) );
1279
1280
1281      end  ( * TAN_INIT * );
1282
1283
1284
1285      ( * #####
1286      ( * Write the line LINE to the file IMAGE starting at pixel START.
1287      ( * It is assumed that each pixel has a value between 0 and 255.
1288      ( * #####
1289      ( * #####
1290      ( * #####
1291      procedure WRITE_IMAGE_LINE( var IMAGE : IMAGEFILE;
1292      LINE : INT_LINE; START : integer );
1293
1294      var
1295      I      : integer;
1296
1297      begin
1298
1299      IMAGE@ := ' ';
1300
1301      for I := 1 to (START-1) do
1302      IMAGE@(I) := char( MIN_INTENSITY );
1303
1304      for I := START to MAX_IMAGE_SIZE do
1305      IMAGE@(I) := char( LINE(I) );
1306
1307      put( IMAGE );
1308
1309
1310      end  ( * WRITE_IMAGE_LINE * );
1311
1312
1313
1314      ( * #####
1315      ( * Write the map line LINE to the map file MAP.
1316      ( * #####
1317      ( * #####
1318      ( * #####
1319      ( * #####
1320      procedure WRITE_MAP_LINE( var MAP : MAPFILE; LINE : MAPSET );
1321
1322      begin
1323
1324      writeln( MAP, LINE );
1325
1326      end  ( * WRITE_MAP_LINE * );
1327
1328
1329
1330
1331      ( * #####
1332      ( * Write the string STR to the present output stream. The string is of
1333      ( * #####
1334

```

```

1335      (* length SIZE. If the length is less than one, the operation does not *)
1336      (* take place. *)
1337      (* *)
1338      (* ##### *)
1339      procedure WRITE_STRING( STR : INPUT_LINE; SIZE : integer );
1340
1341      var
1342          I          : integer;
1343
1344
1345      begin
1346
1347          if SIZE > 0 then
1348              for I := 1 to SIZE do
1349                  write( STR(I) )
1350
1351
1352      end      (* WRITE_STRING *);
1353
1354
1355      (* ##### *)
1356      (* *)
1357      (* Initialize the system and then proceed to transform the original *)
1358      (* image into a geometrically correct radar image. *)
1359      (* *)
1360      (* ##### *)
1361      (* ##### *)
1362      begin      (* MAIN ROUTINE *)
1363
1364          SYS_INIT;
1365          if SIDE_LOOK then
1366              SIDE_LOOK_TRANSFORM
1367          else
1368              SCANNER_TRANSFORM;
1369
1370      end      (* MAIN ROUTINE *).

```