Volume Number: 7
Issue Number: 5
Column Tag: C Workshop

Fractal Mountain Climbing

By Eli Meir, Ben Haller, Ithaca, NY

*Note: **Source code files** accompanying article are located on MacTech CD-ROM or source code disks.*

Fractal Mountains

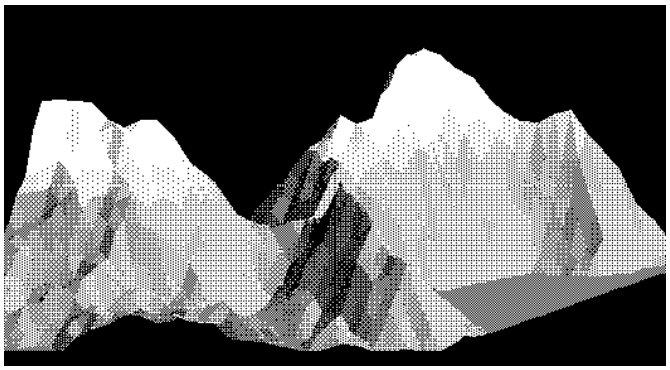
[Eli Meir is a student at Cornell University. He has previously worked in a computer graphics company and is currently programming analog to digital, modelling, and analysis programs on the Macintosh for a neurobiology laboratory. He can be reached at nq5@cornella.bitnet, or at 149 Grandview Ct, Ithaca, NY 14850.

Ben Haller is a student at UC Berkeley. He has done extensive programming on the Macintosh, releasing a good deal of PD and shareware under the name AppleSauce Designs and Stick Software, including the game Solarian II for the Mac II, and doing some consulting work. He can be reached at deadman@garnet.berkeley.edu.]

Introduction

Fractal landscapes are common in movies and animations. They are also fairly easy to generate. In this article, we examine one method of creating and displaying simple fractal mountains which look quite good. Creating and displaying the mountains are two separate problems, so this article and the associated code are divided into two parts: how to create fractal mountains, and a simple method of displaying them.

Before launching into an explanation of the code, let us give a brief description of the program. The 'File' menu contains two commands, Calculate and Quit. Calculate calculates and displays a fractal mountain. The next five menus contain parameters which govern the generation of the mountain. The 'Iterations' menu governs how detailed the mountain will be, with higher numbers indicating more detail. The 'Contour' menu governs the relative steepness at different heights, with higher numbers giving steeper peaks and flatter lowlands. The 'Smoothness' menu governs how jagged the sides of the mountain will be, with higher numbers indicating smoother sides. The 'Height' menu governs both the overall height of the mountains, and also where the treeline will be. Finally, the 'Profile' menu gives four options for the overall shape of the mountain. All these menu items, and how they produce their effects, will be discussed in more detail below.



Generating Fractal Mountains

Fractal mountains are generated by taking an equilateral triangle and recursively subdividing it into smaller and smaller triangles. As each triangle is subdivided, each new midpoint generated is raised or lowered by a random amount. The smaller the subdivisions, the less each new midpoint may be moved, so the overall effect is to get small perturbations of the mountain superimposed on bigger perturbations, which are superimposed on even bigger perturbations. This creates an appearance similar to natural mountains.

The way this works in practice is as follows. Two equilateral triangles are used, set side by side to form a parallelogram, since this makes the code a little easier. The 'Iteration' menu specifies how many times to subdivide each initial triangle. Thus, when the algorithm is done, each side of the initial triangles will have been divided into 2 ^ num_of_iter smaller sides. These smaller sides will have endpoints which still line up if looked at from above (their x and y coordinates haven't been changed), but have been moved up and down (their z coordinates have been changed).

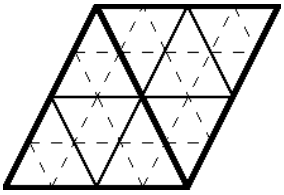


Figure 1: The figure shows the grid of triangles formed after two iterations. The darkest lines represent the initial two triangles, the lighter lines define the triangles formed during the first iteration and the broken lines show the triangles formed during the second iteration. For this example, the points can be stored in a square 5x5 array (2 ^ num_of_iterations + 1 points per side).

As can be seen in figure 1, when all the subdivisions have been made the triangles form a square array of triangle points, 2 ^ num_of_iter + 1 points per side, with each point belonging to up to six triangles. By simply storing the heights of each point in a two-dimensional array, all the triangles are defined. Note that the numbers used to access the array are not identical to the x and y coordinates of the points. Since the array is representing a parallelogram as opposed to a square, the lines of the array must be shifted over by varying amounts to give the x coordinate (the y coordinate is the same as the array indexor).

The function CalcMountains() is the top level function to generate the heights. CalcMountains() first calls InitMountains(), which allocates and initializes the array 'map' where the mountains will be stored and sets the values of some variables based on values selected in the menus. CalcMountains() then sets up some initial heights and calls IterCalc() to start the recursive calculation of heights. We will talk about the initial

heights later on. Notice that since we are starting with two triangles, IterCalc() must be called twice.

IterCalc() is the main calculating function. Given as input the three points of a triangle and an iteration number which indicates the depth of recursion, IterCalc() chops the triangle into four smaller triangles, figures out the heights for the new points which were created, and then, if the recursion hasn't reached the bottom level, IterCalc() calls itself with each of the newly created triangles.

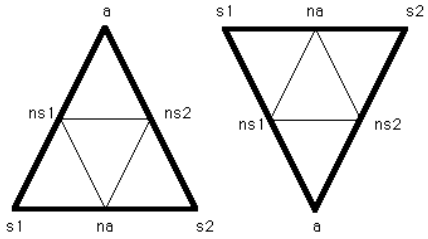


Figure 2: The points s1, s2 and a are passed to IterCalc(), which then finds the points ns1, ns2, and na. These six points then define four smaller triangles as shown, which are recursively passed to IterCalc().

The cutting is done by finding the midpoint of each line segment of the triangle, yielding three new points, and then using the three new midpoints plus the old endpoints to construct four new, smaller triangles (see fig. 2). The three old points are passed to IterCalc() not as x,y,z coordinates, but rather as offsets into the array 'map'. The array is originally set up to be exactly the right size for the number of iterations being done. Thus to find the midpoint between two of the old points, with both midpoint and endpoints being given as offsets into the array, IterCalc() simply adds the offsets of the endpoints and divides by two.

Once the midpoints are found, their heights have to be set. First IterCalc() checks to see if the height was previously set (since each point can be shared by more than one triangle but should only be set once). Initially, all the points in the array are set to 0x80000000, so if the height is different from that then the height was previously set. If the height hasn't already been set, IterCalc() sets it to the average of the heights of the two endpoints, and then passes it to DeviatePoint().

DeviatePoint() is the function which introduces randomness into the picture, and thus gives the mountains their texture. It randomly raises or lowers the height of a point by a number which falls within a certain range. The range of possible deviations is determined by MaxDeviation(), and is based upon what depth the recursion is currently at. The deeper the level of recursion, the smaller the range of possible deviations.

For a given point, the range of possible deviations is calculated using the iteration number and the roughness variable. The iteration number starts out at the maximum number of iterations possible (currently 9) plus one, and is decremented every time IterCalc() call itself. The roughness variable is set in the 'Smoothness' menu, and is a number between 0 and 5.0. The iteration variable times 8 is raised to the power of the roughness variable, and this is returned from MaxDeviation() as the maximum deviation possible for this point. DeviatePoint() then feeds this number into Rand(), which returns a random number between -maximum and maximum, and that number is then added to the point's height. Since the iteration number is decremented by one at each recursion, smaller triangles end up with smaller deviations, at a rate governed by the roughness variable.

The depth of recursion is given by the menu 'Iterations'. IterCalc() must bottom out (not recurse again) when it has recursed this many times. The traditional way to bottom out would be to have a number representing the number of iterations left to go, decrement that number every recursion, and bottom out when that number reaches zero. IterCalc() does pass down a number similar to this, the iteration number used above in MaxDeviation(), but this number always starts at the maximum number of iterations possible so that all mountain ranges will be the same height, regardless of how detailed they are. Rather than passing down an additional number, IterCalc() uses a little trick. When the bottom of the recursion has been reached, the length of each side of any triangle will be one. IterCalc() tests to see if the length of the horizontal side of one of the subtriangles equals one, and if it does IterCalc() bottoms out.

The last step in calculating the mountains is a call to NormalizeMap(), which scales the height of the mountain to be the height selected in the 'Height' menu, and also applies the contour transformation. The contour transformation works by raising each height to the power of the number selected in the 'Contour' menu. Thus, if contour is less than one, the mountains are made more concave, while if contour is more than one, the mountains are made more convex.

While the shape of the mountains is determined by the random deviations of the biggest triangles, the overall outline is determined by the initial heights given to the corners of the first two triangles. For instance, if the top right corner is given a very high value, and the bottom left corner is given a low value, with the other corners given intermediate values, a mountain will be generated which slopes down from the right into the sea on the left. The menu 'Profile' allows selection between four combinations of raising some corners and lowering others. CalcMountains() uses this menu selection to give the corners their initial values.

The outcome of this whole process is an array of points, each with a certain height, and the whole array defines the triangles which make up our mountain range. Next we'll give the drawing techniques necessary to turn these numbers into a pretty picture on your screen.

Drawing Fractal Mountains

In drawing the fractal mountains, the way we color each triangle is critical, as this is what will give a sense of height and shape to the picture. Without proper coloring, we will just display a bunch of triangles on the screen. We also need to rotate the mountains so that we are looking at them from near the ground, instead of from the top in a birds-eye view (since the mountain's x-y plane is initially the same as the plane of the screen). In addition, we need to use a hidden surface algorithm, so that triangles which are hidden by other triangles are not visible.

DrawMountains() is the top level function of the drawing routines. After setting a couple variables which will be discussed later, DrawMountains() proceeds to call DrawTriangle() with each triangle in the array. The order that it draws these triangles is important. We are tipping the mountains backwards to give us a more interesting view, and this tipping means that some parts of the mountain are going to obscure other parts. We need to draw the triangles so that the obscured parts are in fact hidden from view. DrawMountains() does this by simply drawing the triangles in back (those with the smallest y's) first. Then, any triangles farther forward that hide those triangles will be drawn over them, and thus the hidden surface problem is solved.

DrawTriangle() and _DrawTriangle() are the functions which do the actual coloring and drawing. DrawTriangle() is given the x and y coordinates of the three points of a triangle. It then looks up the z coordinate of each of the points, and passes all the points to _DrawTriangle(), which does the coloring and drawing. The main job of DrawTriangle() is to perform some clipping which will be discussed a little later.

We use a fairly simple coloring algorithm. The mountains are divided into three main colors based on height, with the brightness of the color used being determined by the angle of each triangle with a light source. Thus all triangles below a height of 0 are colored blue for ocean, all triangles with medium heights are colored green, and all triangles above a certain height are colored grey (on a computer with less than 256 colors, these are different black and white patterns).

After deciding on the main color, the three points of the triangle are scaled and rotated to find the points used in drawing the triangle. The scaling is done to account for the size of the window and the fact that the triangles must be drawn smaller as the number of iterations gets

larger, and the points are rotated backwards for the reasons discussed above. The routine which does this rotating and scaling is CalcPoint3(). It does this by multiplying the point by some scaling variables and by a 3x3 rotation matrix. The formula for rotating points in 3-space can be found in any book on graphics (or in a textbook on vector geometry). The rotation in this example is a -70 degree (-1.396 radian) rotation around the x-axis. The constant xTh holds the angle of rotation in radians, and this can be changed, but if any rotation other than a 0 to -90 degree rotation around the x-axis were done, the hidden surface routine would have to be changed, since the back to front ordering of the triangles would be different.

With the proper drawing coordinates known, a triangle is built using Quickdraw's polygon routines. The base color of each triangle has already been figured out using its height. What remains is to decide how bright the color should be. This involves a bit of vector arithmetic, which is beyond the scope of this article. We will briefly summarize the way brightness is calculated in the following paragraph. For those of you not wishing to try to figure out vectors, the whole calculation of brightness can be safely thought of as a black box, without much loss of understanding. For those of you who would like a deeper understanding, we have included a couple references at the end of the article on vectors and computer graphics.

Light comes in at a certain angle, given by a vector, and the cosine of the angle between the light vector and the normal vector of a triangle (a vector which points straight out from the triangle) is used to determine the shading of the triangle. If the cosine is 1, the light is hitting the triangle straight on, so the triangle is colored the brightest shade of the appropriate color. If the cosine is less than one, the light is hitting the triangle at an angle, so the triangle is colored with a darker shade, down to half brightness for a cosine of 0. The only case where the shading is not done is for triangles which are below 0 in height, which are drawn as part of the ocean and are all colored the same shade of blue.

For black and white drawings the process is somewhat different. All the same calculations are made, but instead of ending up with a color, the calculated brightness is used to pick a pattern out of a possible 17 patterns set up in bwPat. The patterns go between dark gray and light gray to give the desired shading.

One final trick is used in coloring triangles. Some triangles will be partially under and partially above water. These triangles are cut at the water line by DrawTriangle(), and the two pieces are colored differently. This gives a smooth water line, which makes the picture look more realistic.

Final Notes

There are many parts of this program which could be played with to give a different, and perhaps better looking, picture. First try playing with the parameters given in the menus. Try adding to the Profile menu, using different initial corner heights. You could also experiment with the height raising and lowering algorithm. If you add more iterations, the pictures will be nicer. However, be aware that each new iteration quadruples the amount of memory necessary and the time taken to calculate and draw the picture. One way around the increased memory usage would be to draw the triangles as they are being figured, a simple change. The coloring scheme could be changed, either by just changing the colors and patterns, or by changing the whole method by which coloring is done. One more thing some of you may want to try is to output all the triangles into a ray-tracing program, to get much better shading and probably much more realistic mountains. However, even with this simple program, you can get very nice looking fractal mountains.

Listing: MountainForm.h

```
/* Header file for Mountains.  Could be
        precompiled if you like... */
#include "Color.h"
#include "ColorToolbox.h"
#include "EventMgr.h"
#include "MacTypes.h"
#include "MenuMgr.h"
#include "Quickdraw.h"
```

Listing: MountainsToolForm.c

```
/*
 MountainsToolbox.c
 The toolbox interface to front-end  MountainsAlgorithm.c
 By Ben Haller 1/1/90
 ©1990 Ben Haller
*/

#include "Mountains.h"

char done=FALSE; /* FALSE as long as user has not quit */
MenuHandle menu[6];  /* Our menus */
EventRecord evt;  /* Our eventrecord */
WindowPtr wp;   /* Our window */
CursHandle watch; /* Watch cursor for long waits */
char colorF;/* TRUE if 8-bit color */
int wx,wy,wd;   /* Size of window (x and y dimensions) */
Rect r; /* scratch rectangle for general use */
char menuCk[5]={3,6,5,5,1}; /* number of
 checked item in each menu */
 /*(File menu not included) */

/*****************************************/
/* Routines in MountainsAlgorithm.c */
void DrawMountains(void);
void CalcMountains(void);

/*****************************************/
/* Initializes the toolbox and all important variables, */
/* loads resources, etc. */
void Init()
{
 int n;

 InitGraf(&thePort);
 InitFonts();
 InitWindows();
 InitMenus();
 TEInit();
```

```
 InitDialogs(0L);
 InitCursor();
 FlushEvents(-1, 0);

 /* Reseed the random number generator.      Note: You can force generation
of a  specific series of random numbers (and       thus a specific picture,
all parameters   being equal) by replacing this statement      with "randSeed=<any
integer>" */
 randSeed=Ticks;

 /* Find out if we're running in 8-bit color and set colorF accordingly
*/
 n=(*((*GetGDevice())->gdPMap))>pixelSize;
 colorF=(n>=8);

 /* Make menus */
 ClearMenuBar();
 menu[0]=NewMenu(5000,"\pFile");
 AppendMenu(menu[0],"\pCalculate;(-;Quit");
 InsertMenu(menu[0],0);
 menu[1]=NewMenu(5001,"\pIterations");
 AppendMenu(menu[1],"\p3;4;5;6;7;8;9");
 InsertMenu(menu[1],0);
 menu[2]=NewMenu(5002,"\pContour");
 AppendMenu(menu[2],"\p0.25;0.50;0.75;1.00;1.25;1.50;1.75;2.00;3.00;5.00");
 InsertMenu(menu[2],0);
 menu[3]=NewMenu(5003,"\pSmoothness");
 AppendMenu(menu[3],"\p1.00;1.25;1.50;1.75;2.00;2.25;2.75;3.50;5.00; ");
 InsertMenu(menu[3],0);
 menu[4]=NewMenu(5004,"\pHeight");
 AppendMenu(menu[4],"\p1;2;3;4;5;6;7;8;9;10");
 InsertMenu(menu[4],0);
 menu[5]=NewMenu(5005,"\pProfile");
 AppendMenu(menu[5],"\p#1;#2;#3;#4");
 InsertMenu(menu[5],0);
 DrawMenuBar();

 /* Check all the menu items appropriate*/
 for (n=1; n<6; n++)
 CheckItem(menu[n],menuCk[n-1],TRUE);

 /* Get watch  */
 watch=GetResource('CURS',4);
 HLock(watch);

 /* Make our window, full screen inset 5 pixels */
 r=screenBits.bounds;
 r.top+=MBarHeight;
 InsetRect(&r,5,5);
 wx=r.right-r.left;
 wy=r.bottom-r.top;
 if (wy>wx/2) wy=wx/2;
 else if (wx>wy*2) wx=wy*2;
 wd=wx;
 r.bottom=r.top+wy;
 r.right=r.left+wx;
 if (colorF)
 wp=NewCWindow(0L, &r, "\p", TRUE,
 plainDBox, -1L, FALSE, 0L);
 else
 wp=NewWindow(0L, &r, "\p", TRUE,
 plainDBox, -1L, FALSE, 0L);
}

/**************************************/
/* Handle the selection of a menu item */
void HandleMenu(m,i)
int m,i;/* m=menu id, i=item number */
{
 m-=5000;
 switch (m) {
 case 0:
 if (i==1) {
 /* Recalculate and update */
 CalcMountains();
 InvalRect(&(wp->portRect));
 } else done=TRUE; /* Quit */
 break;
 case 1:
 case 2:
 case 3:
 case 4:
 case 5:
 /* Move the check */
 CheckItem(menu[m],menuCk[m-1],FALSE);
 menuCk[m-1]=i;
```

```
 CheckItem(menu[m],menuCk[m-1],TRUE);
 break;
 }
 if (!done) HiliteMenu(0);
}

/*****************************************/
/* Handle updating our window */
void HandleUpdate(wp)
WindowPtr wp;
{
 SetPort(wp);
 BeginUpdate(wp);
 if (colorF) {
   RGBColor blackc;

   blackc.red=0; blackc.green=0;
   blackc.blue=0;
 RGBForeColor(&blackc);
 }
 FillRect(&(wp->portRect),black);
 DrawMountains();
 EndUpdate(wp);
}

/*****************************************/
/* Handle a mouse click  */
void HandleMouse()
{
 WindowPtr wW;
 long msresult;

 switch (FindWindow(evt.where, &wW)) {
 case inMenuBar:
 msresult=MenuSelect(evt.where);
 HandleMenu(HiWord(msresult), LoWord(msresult));
 break;
 default: break;
 }
}

/*****************************************/
/* Initialize, get events until done. */
int main()
{
 Init();
 while (!done) {
 SystemTask();
 if (GetNextEvent(everyEvent, &evt)) {
 switch (evt.what) {
 case mouseDown:
 HandleMouse();
 break;
 case updateEvt:
 HandleUpdate(evt.message);
 break;
 }
 }
 }
}
```

Listing: MountainAlgoForm.c

```
/*
 MountainsAlgorithm.c
 A fractal mountain generating algorithm
 By Ben Haller 1/1/90
 Technical Assistance by Eli Meir
 ©1990 Ben Haller
*/

#include <math.h>
#include "Mountains.h"

/* Imported declarations from MountainsToolbox.c*/
extern CursHandle watch;
extern int wx,wy,wd;
extern char colorF;
extern char menuCk[5];

/* Algorithmic declarations */
#define maxIter  9L

/* Black and white patterns for shading */
long bwPat[17][2]={
 {0xFFFFFFFF, 0xFFFFFFFF},
 {0xFFFBFFBF, 0xFFFBFFBF},
```

```
 {0xFFEEFFBB, 0xFFEEFFBB},
 {0xFFEEFFAA, 0xFFEEFFAA},
 {0xBBEEBBEE, 0xBBEEBBEE},
 {0xAA77AAFF, 0xAA77AAFF},
 {0xAA77AADD, 0xAA77AADD},
 {0xAA77AA55, 0xAA77AA55},
 {0xAA55AA55, 0xAA55AA55},
 {0xAA55AA11, 0xAA55AA11},
 {0xAA44AA11, 0xAA44AA11},
 {0xAA44AA00, 0xAA44AA00},
 {0x22882288, 0x22882288},
 {0x2200AA00, 0x2200AA00},
 {0x22008800, 0x22008800},
 {0x80000800, 0x80000800},
 {0x00000000, 0x00000000}
};

/* 3D point - double FP precision */
typedef struct {
 double x,y,z;
} point3;

double xTh=-1.396; /* Angle of tilt (in radians) */
double tm[3][3]; /* Transformation matrix */
long *map;/* A pointer to the depth values of the map */

/* Variables dependant on menu selections */
long nIter;  /* Number of iterations of divide-and-conquer */
long mapS;/* Length of one side of the map */
long mapD;/* Map dimension - equals mapS-1 */

double contour;  /* Power to raise heights
     to, to make contour */
double roughness;/* Base to raise to get scale variance */
long normalHeight; /* Normalized height */
char profile;    /* Profile number */

/* Drawing variables - precalculated coefficients */
double xc,sc;

/* The Map #define lets us access map easily, although it is a variable-dimension
array. */
#define Map(x,y) map[(x)+mapS*(y)]

/*****************************************/
/* Calculation Routines                  */
/*****************************************/

/* Make the mountains a standard height, and simultaneously apply the
contour transform */
void NormalizeMap()
{
 register long n,m,k;
 register long *a;
 register double i,z;

 a=map;
 m=0x80000000;

 /* Find highest point in map */
 for (n=mapS*mapS-1; n>=0; n--)    if (*(a+n)>m)
 m=*(a+n);
 z=(double)m;
 z=pow(z,contour);
 /* Apply contour transform to it */
 z=(double)normalHeight/z;
 /* Find normalizing coefficient such that   the range will be the chosen
height */
 for (n=mapS*mapS-1; n>=0; n--) { /* For each
   point in the map */
 k=*(a+n);
 if (k>0) { /* Special case for positive & neg. heights */
 i=(double)k;
 /* Apply contour transform, normalize */               i=pow(i,contour)*z;

 *(a+n)=(long)i; /* Poke back in map */
 } else {
 i=(double)(-k);
 /* Apply contour transform, normalize */
 i=pow(i,contour)*z;
 *(a+n)=-((long)i); /*Poke back in map,preserving sign*/
 }
 }
}

/* Returns a random long integer.  This routine just generates two integers
```

```
using QuickDraw's random number generator and tacks them together.  I
have no idea how good the numbers thus generated are, but I suspect not
very good at all.  But, they're good enough.*/
long Rand(first, last)
long first, last;
{
 long r;

 do {
 r=(((long)Random())<<16)+Random();
 } while (r<0);
 return(r % (last - first + 1) + first);
}

/* Returns the maximum deviation a point could attain given an iteration
depth.  The exact number returned depends on roughness, but this function
is always  strictly decreasing monotonic for given depth ic. */
long MaxDeviation(ic)
long ic;
{
 if (roughness>0)
 return ((long)(pow(roughness, (double)(ic-1L))*8.0));
 else return 100000L;
}

/* This routine deviates a point by a random amount from -m to m, m being
the maximum deviation for the given iteration.  If roughness==0 (symbolic
of infinite smoothness), it is not deviated at all. */
void DeviatePoint(long,long);
void DeviatePoint(o,ic)
long o,ic;
{
 long v;

 if (roughness<0) return;
 v=MaxDeviation(ic);
 map[o]+=Rand(-v,v);
}

/* Passed a triangle with "side" (i.e. horizontal side) (sx1,sy1)-(sx2,sy2)
and "apex" (ax,ay).  It calculates midpoints and recurses.  Parameter
'c' gives a standard iteration count */
void IterCalc(long,long,long,long);
void IterCalc(s1,s2,a,c)
long s1,s2,a,c;
{
 register long ns1,ns2,na;

 /* Decrement iteration count */
 c--;

 /* Find midpoints */
 ns1=(s1+a)>>1;
 ns2=(s2+a)>>1;
 na=(s1+s2)>>1;

/* For each midpoint, if it hasn't already been set, set it to average
of it's endpoints, and deviate by a random amount*/
 if (map[ns1]==0x80000000L) {
 map[ns1]=(map[s1]+map[a])>>1;
 DeviatePoint(ns1,c);
 }
 if (map[ns2]==0x80000000L) {
 map[ns2]=(map[s2]+map[a])>>1;
 DeviatePoint(ns2,c);
 }
 if (map[na]==0x80000000L) {
 map[na]=(map[s1]+map[s2])>>1;
 DeviatePoint(na,c);
 }

 /* Iterate calculations on sub-triangles if we haven't yet reached maximum
resolution of the map */
 if (ns1+1!=ns2) {
 IterCalc(s1,na,ns1,c);
 IterCalc(na,s2,ns2,c);
 IterCalc(ns1,ns2,na,c);
 IterCalc(ns1,ns2,a,c);
 }
}

/* Initialize the entire map to 0x80000000 (<not set> value)*/
void InitMap()
{
 register long n;
 register long *a;
```

```
 a=map;
 for (n=mapS*mapS-1; n>=0; n--)
 *(a+n)=0x80000000L;
}

/* Initialize values from menu choices, allocate map, etc. */
void InitMountains()
{
 nIter=menuCk[0]+2;
 map=0L;
 mapD=1L<<nIter;
 mapS=mapD+1;

 contour=0.25*menuCk[1];
 if (menuCk[1]==9) contour=3.0;
 else if (menuCk[1]==10) contour=5.0;

 roughness=0.75+0.25*menuCk[2];
 if (menuCk[2]==7) roughness=2.75;
 else if (menuCk[2]==8) roughness=3.5;
 else if (menuCk[2]==9) roughness=5.0;
 else if (menuCk[2]==10) roughness=-1.0;

 normalHeight=10000L*menuCk[3];

 profile=menuCk[4];

 if (map==0L) map=NewPtr(mapS*mapS*4L);
 if (map==0L) ExitToShell();/* Out of memory */
 InitMap();
}

/* Calculate mountain range using current
 parameters, profile, etc. */
void CalcMountains()
{
 register long q;

 SetCursor(*watch);
 InitMountains();

 /* Generate starting profile to build on */
 q=MaxDeviation(maxIter+1)/2;
 switch (profile) {
 case 1:
 Map(0,0)=q;
 Map(mapD,0)=0;
 Map(0,mapD)=0;
 Map(mapD,mapD)=-q;
 break;
 case 2:
 Map(0,0)=q;
 Map(mapD,0)=0;
 Map(0,mapD)=0;
 Map(mapD,mapD)=0;
 Map(mapD>>1,mapD>>1)=0;
 Map(mapD>>1,mapD)=0;
 Map(mapD,mapD>>1)=0;
 break;
 case 3:
 Map(0,0)=0;
 Map(mapD,0)=0;
 Map(0,mapD)=0;
 Map(mapD,mapD)=-q;
 break;
 case 4:
 Map(0,0)=0;
 Map(mapD,0)=0;
 Map(0,mapD)=0;
 Map(mapD,mapD)=0;
 Map(mapD>>1,mapD>>1)=q/2;
 Map(mapD>>1,0)=q;
 Map(0,mapD>>1)=q;
 break;
 }

 /* Generate each main triangle recursively */
 IterCalc(0L,mapS-1,mapS*mapS-1,maxIter+1);
 IterCalc(mapS*(mapS-1), mapS*mapS 1,0L,maxIter+1);

 /* Normalize to standard height, apply contour transform */
 NormalizeMap();

 SetCursor(&arrow);
}
```

```
/*****************************************/
/* Drawing Routines                      */
/*****************************************/

/* Transform from map coordinates to screen coordinates*/
void CalcPoint3(p)
point3 *p;
{
 register double xp,yp,zp;

 xp=(p->x*2-p->y+mapD)*xc;
 yp=(p->y*2)*xc;
 zp=p->z*0.00217;

 p->x=xp*sc;
 p->y=(yp*tm[1][1]+zp*tm[2][1]+230)*sc;
 p->z=(yp*tm[1][2]+zp*tm[2][2])*sc;
}

/* This routine actually draws a triangle, given by a set of three (x,y,z)
triplets. It determines the color and shading according to altitude and
lighting, constructs a QuickDraw polygon for the triangle, and draws
it. */
void _DrawTriangle(p)
point3 *p;
{
 double nx,ny,nz,lx,ly,lz,i;
 RGBColor c;
 PolyHandle ph;
 long color,slope;

 /* Figure out what base color our triangle
 will be : blue, green, or gray */
 if ((p[0].z==0.0) && (p[1].z==0.0) && (p[2].z==0.0))
 color=-1;    /* blue */
 else {
 double az,treeline;

 /* Calculate treeline  */
 treeline=.4*normalHeight+10000;   az=(p[0].z+p[1].z+p[2].z)/3.0; /*
Find avg. height*/
 if (az>treeline+9000) color=150;/* gray  */
 else if (az<treeline) color=0;  /* green */
 else color=(az-treeline)/60;  /* intermediate */
 }

 /* Transform into viewing space */
 CalcPoint3(p+0);
 CalcPoint3(p+1);
 CalcPoint3(p+2);

 /* Create a Quickdraw polygon to be the
 triangle in question */
 ph=OpenPoly();
 MoveTo((int)p[0].x,(int)p[0].y);
 LineTo((int)p[1].x,(int)p[1].y);
 LineTo((int)p[2].x,(int)p[2].y);
 LineTo((int)p[0].x,(int)p[0].y);
 ClosePoly();

 /* Calculate the normal vector to our triangle, by means of a cross
product */
 {
 double v1x,v1y,v1z,v2x,v2y,v2z;

 v1x=p[1].x-p[0].x;
 v1y=p[1].y-p[0].y;
 v1z=p[1].z-p[0].z;

 v2x=p[2].x-p[0].x;
 v2y=p[2].y-p[0].y;
 v2z=p[2].z-p[0].z;

 nx=v1y*v2z-v1z*v2y;
 ny=v1z*v2x-v1x*v2z;
 nz=v1y*v2x-v1x*v2y;
 }

 /* Initialize our light source */
 lx=-1.0/sqrt(3.0);
 ly=-lx; lz=-lx;

 /* Figure out what color we want */
 if (color==-1) {
 c.red=0x9FFF; c.green=0x9FFF; c.blue=0xFFFF;/* Water */
```

```
 if (colorF) RGBForeColor(&c);
 else PenPat(bwPat[8]);
 } else {
 unsigned long cv[3];

 /* Start with green */
 cv[0]=0x00005555L; cv[1]=0x0000FFFFL; cv[2]=0x0000AAAAL;

 /* Blend to brown and white over treeline*/
 cv[0]=((150L-color)*(cv[0]0x00002000L))/
       150L+0x00002000L;
 cv[1]=(cv[1]*(150L-color))/150L;
 cv[2]=0x0000EFFFL-((150L color)*
 (0x0000EFFFL-cv[2]))/150L;

 /* Move into an RGBColor structure */
 c.red=cv[0]; c.green=cv[1]; c.blue=cv[2];

 /* Scale brightness according to the incidence of light */
 i=((lx*nx+ly*ny+lz*nz)/ sqrt(nx*nx+ny*ny+nz*nz))*0.5+0.5;
 c.blue*=i;

 if (colorF) {   /* Pick our color */
 HSV2RGB(&c,&c);
 RGBForeColor(&c);
 } else { /* Pick a pattern based on brightness */
 int pat;

 pat=(((unsigned long)c.blue)*36L)/0x00010000L;
 pat-=10;
 if (pat<0) pat=0;
 if (color==150) {
 if (pat>16) pat=16;
 } else if (pat>15) pat=15;
 PenPat(bwPat[pat]);
 }
 }

 PaintPoly(ph);  /* Draw our triangle */
 KillPoly(ph);   /* Get rid of it */
}

/* Draw a given triangle.  This routine is mainly concerned with the
possibility that a triangle could span the waterline.  If this occurs,
this procedure breaks it up into three smaller triangles, each of which
is either above or below water.  All actual drawing or coloration is
delegated to _DrawTriangle, above. */
void DrawTriangle(x1,y1,x2,y2,x3,y3)
long x1,y1,x2,y2,x3,y3;
{
 long z[3];
 point3 p[3];

 /* Get depths of the three points */
 z[0]=Map(x1,y1);
 z[1]=Map(x2,y2);
 z[2]=Map(x3,y3);
 if ((z[0]<=0) && (z[1]<=0) && (z[2]<=0)) {  /*All underwater */
 p[0].x=x1; p[0].y=y1; p[0].z=0;
 p[1].x=x2; p[1].y=y2; p[1].z=0;
 p[2].x=x3; p[2].y=y3; p[2].z=0;
 _DrawTriangle(p);
 } else if ((z[0]<0) || (z[1]<0) || (z[2]<0))      {     /* Some underwater
- split at waterline*/
 point3 ap,s[2],m[2];
 char w[3];
 int n;
 double f;

 p[0].x=x1; p[0].y=y1; p[0].z=z[0];
 p[1].x=x2; p[1].y=y2; p[1].z=z[1];
 p[2].x=x3; p[2].y=y3; p[2].z=z[2];
 for (n=0; n<3; n++)
 w[n]=(z[n]<0);
 if ((w[0]!=w[1]) && (w[0]!=w[2])) {
 ap=p[0]; s[0]=p[1]; s[1]=p[2];
 } else {
 if (w[1]!=w[0]) {
 s[1]=p[0]; ap=p[1]; s[0]=p[2];
 } else {
 s[0]=p[0]; s[1]=p[1]; ap=p[2];
 }
 }

 /* At this point, ap is the "odd man out" -             either it is
above water and the other two are below, or it is below and the other
```

```
two     are above.  Which corner s[0] is and which             s[1] is
*is* important - if we get the     wrong order, the normal vector used
to find the shading coefficient is the wrong sign.             This is
true whenever we are manipulating corners - the ordering is always important.*/

 /* Find the "midpoints" between ap and       s[0]&s[1] - this is where
we split our big triangle into smaller triangles. Actually it is not
a normal midpoint, but a  weighted midpoint, such that the z  component
is 0 - waterline.*/
 for (n=0; n<2; n++) {
 f=-((double)ap.z)/(double)(s[n].z-ap.z);

 m[n].x=ap.x-(ap.x-s[n].x)*f;
 m[n].y=ap.y-(ap.y-s[n].y)*f;
 m[n].z=0;
 }

 /* Set whichever triangles are below water to 0 altitude */
 if (ap.z<0) ap.z=0;
 else {
 s[0].z=0; s[1].z=0;
 }

 /* Draw our three triangles */
 p[0]=ap; p[1]=m[0]; p[2]=m[1];
 _DrawTriangle(p);
 p[0]=m[0]; p[1]=s[0]; p[2]=s[1];
 _DrawTriangle(p);
 p[0]=m[0]; p[1]=s[1]; p[2]=m[1];
 _DrawTriangle(p);
 } else { /* All above water */
 p[0].x=x1; p[0].y=y1; p[0].z=z[0];
 p[1].x=x2; p[1].y=y2; p[1].z=z[1];
 p[2].x=x3; p[2].y=y3; p[2].z=z[2];
 _DrawTriangle(p);
 }
}

/* Draw the entire mountain range.  Non-recursive. */
void DrawMountains()
{
 long x,y;

 if (map==0L) return;
 SetCursor(*watch);

 /* Set drawing coefficients (used in CalcPoint3) */
 xc=0.4073*(double)(1<<(maxIter-nIter));
 sc=((double)wd)/630.0;

 /* Make transformation matrix */
 tm[0][0]=1;
 tm[1][0]=0;
 tm[2][0]=0;
 tm[0][1]=0;
 tm[1][1]=cos(xTh);
 tm[2][1]=sin(xTh);
 tm[0][2]=0;
 tm[1][2]=-sin(xTh);
 tm[2][2]=cos(xTh);

 /* Go back to front, left to right, and draw
 each triangle */
 for (y=0; y<mapS-1; y++) {
 for (x=0; x<mapS-1; x++) {
 DrawTriangle(x,y,x,y+1,x+1,y+1);
 DrawTriangle(x,y,x+1,y+1,x+1,y);
 }
 }

 SetCursor(&arrow);
}
```