TerraForm 3D

Status Report on parallelization efforts

TABLE OF CONTENTS

Project Summary

Trent D'Hooge
Deborah Lee Soltesz

1 December 1999

Derive: Creates shaded relief from elevation image data

FILES:

derive.cpp	driver and parallelization
imageManager.*	designed as interface between image file parsing and higher functions ; manages textures and terrain data generically, hiding file formats, parsing, and decompression
image.*	base/parent image class
MIPS_image.*	child image class which holds image parameters and file parsing specific to USGS MIPS format. In the future, there will be child classes for other formats, such as TIFF, JPEG, and BMP.

FUNCTION of NODES:

Mother: is responsible for retrieving image data from file, breaking it into horizontal sections for the child processors, reassembling processed sections, and writing the resulting image to file. The mother node performs file I/O while the child nodes apply a function to the image data.

Child: is responsible for applying the "derive" function to its section of the image. Because the actual function is boiled down to a

single subroutine, multiple types of filters and applications could be available as a command line argument, and the children applying the appropriate filter based on that parameter.

DATA & STRUCTURE:

Required image data is pulled from the file as needed to allow extremely large data sets to be manipulated without straining resources. Data sets are generally in the 30 to 60 megabyte range, but some high resolution regional data sets are in the gigabytes.

The image data is stored in a three-dimensional integer array. While elevation data is only two-dimensional (thus, in the driver, the third array index is always 0), the structure allows for multi-band data to be handled in a single array. An simple example of a multi-band data set would be a standard, full color image composed of the three 8-bit bands: red, green, blue. The capability to hold deeper arrays can be added easily, such as satellite and aerial imagery, with typical depths of seven to 256 bands.

A rasterized digital elevation model (in USGS MIPS format) is composed of evenly spaced pixels, whose locations in the file (x and y) are based on the location on the Earth's surface, and whose values (z) represent the elevation above or below that point on the surface on a given scale. Generally, the data is unsigned word, with 0 equal to sea level, but may be unsigned byte.

DERIVE FILTER:

The DERIVE filter applies artificial sunlight to a terrain model. It is based on the slope value between elevations, which is typically dz/dx, dz/dy, or dz/d(x and y). Because the x and y spacings are even throughout the file, they are dropped from the equation, and only dz is used.

For a two-dimensional array of data, i[x][y] = z, and north is up, the derive filter works as follows:

Sunlight source	equation
west	i[x][y] = i[x + 1][y] - i[x][y] + c
north	i[x][y] = i[x][y + 1] - i[x][y] + c
north-west	i[x][y] = i[x + 1][y + 1] - i[x][y] + c

where c = (upper bound of data + lower bound of data) / 2
  for unsigned byte: c = (0 + 255) / 2 = 127
  for signed word : c = (-32768 + 32768) / 2 = 0

Things to do list:

- Mother should be able to dynamically break data into overlapping pieces based on the filter type. Some filters may require that children have access to several lines of data "belonging" to another child's

section of image.

- Mother should be able to determine what order data is written to the file. Currently, it is written in the order received, but some applications, such as three-dimensional terrain rendering, require that data be written out based on the final distance from "viewer," not necessarily the x-y ordering of the original file.

- Add more filters and applications, possibly using dynamically linked libraries of functions, to provide full image processing functionality. Especially important are terrain visualization and rendering functions.

- Improve the efficiency of the file I/O

REFERENCES:

Antarctica Data Set -- elevation model and imagery applications

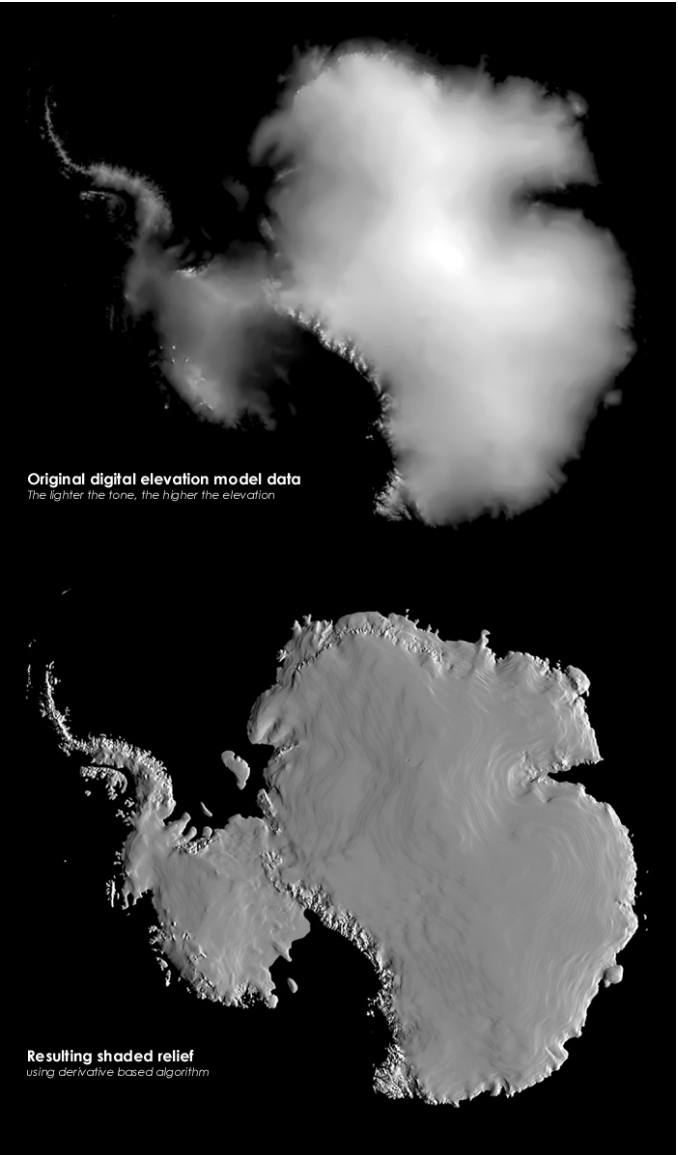
http://TerraWeb.wr.usgs.gov/TRS/projects/Antarctica/AVHRR.html

Monterey Bay Sonar -- an example of an extremely large data set. Incomplete and scaled down to 1 meter per pixel, the data set is approximately 60,000 wide by 20,000 high (in pixels). The actual data is at a resolution of 40 cm per pixel (more than twice the size of the data presented below), and is handled in several files instead of a single file.

http://TerraWeb.wr.usgs.gov/TRS/projects/MontereySonar/

Imagery

The following shows the input data (digital elevation model) and the desired output shaded relief (lit from the east).


Original digital elevation model data
The lighter the tone, the higher the elevation
Resulting shaded relief
using derivative based algorithm

Source Code

```
/****************************************
DERIVE

file:    derive.cpp

purpose: driver

descrip: create shaded relief from a
         rasterized (non-vector) Digital
         Elevation Model (DEM) using
         MPI on clustered system

authors: Trent D'Hooge
         Deborah Lee Soltesz

date:    30 nov 99

****************************************/


#include "mpi++.h"
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include "imageManager.h"


int main(int argc, char **argv) {

  int myID = 0, numProcs = 1, file = 0, direction = 0,
      length, vert, horiz, imgstatus = 0, range = 0,
      terrainByteSize = 2, linesInBuffer = 0 ;

  int subAreaLines = 0;
  char * filename;
  int *** terrainArr  = NULL ;
  int *** terrainArrBuffer = NULL ;

  int *   msgBuffer ;

  int lines        = 0 ;
  int totalLines   = 0 ;
  int samples      = 0 ;

  int firstTime = 0 ;
  int linesLeft = 0 ;

  FILE * outputFile ;
  char * outputFileName = "AVHRRout.img\0" ;

  // TO DO
  // initialize parallization

  MPI_Status status;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myID);

  int baseImageSubarea = 1024 * 1024 ; // 1 MB = max size of image
```

```
                                        // subarea to work with

  // initialize input file

/*************************************************************
process 0 takes the file name as a command-line argument
*************************************************************/

  if (!myID)
  {
    int i = 1;
    while ( i < argc )
    {

/****************
get the file name
****************/

      if ( !strcmp("-fname", *argv) )
      {
        file = i;
        argv++;
        length = strlen(*argv)+1;
        filename = new char[length-1];
        strcpy(filename, *argv);
        argv--;
      }

/***********************************************
get the direction; horizontal, vertical, diagonal
***********************************************/

      if ( !strcmp("-direction", *argv) )
      {
        argv++;
        direction = atoi(*argv);
        argv--;
      }
      i++;
      argv++;
    }

/*************************************
if do not get direction or file name EXIT
*************************************/

    if ( direction == 0 || file == 0 ) {
      cout << "\n  Usage: %s -fname filename -direction 10/01/11 \n\n";
      MPI_Abort(MPI_COMM_WORLD, 1);
      return 1;
    }

/*****************************
tell all computers the file name
and its length and the direction
*****************************/

    MPI_Bcast(&direction, 1, MPI_INT, 0, MPI_COMM_WORLD);
  }
```

```
/*****************************
receive from computer 0 the info
*****************************/

  else
  {
    MPI_Bcast(&direction, 1, MPI_INT, 0, MPI_COMM_WORLD);
  }

  if ( direction == 10 ) { horiz = 1; vert = 0; } //horizontal
  if ( direction == 1 )  { horiz = 0; vert = 1; } //vertical
  if ( direction == 11 ) { horiz = 1; vert = 1; } //diagonal


/*******************************
Initialize input and output image
*******************************/

  ImageManager * dem ;
  Image * terrain ;

  if (!myID) {
    // Open output file
    outputFile = fopen(outputFileName, "w") ;
    if (!outputFile) {
      cout << "Output file " << outputFileName
           << " failed to initialize" << endl;
      MPI_Abort(MPI_COMM_WORLD, 1);
      return 1 ;
    }

    // Open input file
    cout << "Opening " << filename << endl ;
    dem = new ImageManager (filename)  ;
    terrain = dem -> getTerrainImage () ;

    if (!terrain) {
      cout << "Terrain file failed to initialize" << endl;
      MPI_Abort(MPI_COMM_WORLD, 1);
      return 1 ;
    }

    linesLeft  = terrain -> getFileLines () ;   // total lines in image
    totalLines = linesLeft ;                    // remember num file lines
    samples    = terrain -> getFileSamples () ; // total samples in image

    terrainByteSize = terrain -> getByteSize () ;
  }

  // announce width of the input image
  MPI_Bcast (&samples, 1, MPI_INT, 0, MPI_COMM_WORLD) ;
  MPI_Bcast (&terrainByteSize, 1, MPI_INT, 0, MPI_COMM_WORLD) ;
  msgBuffer = new int[samples] ;

  // mother gets first piece of data
  if (!myID && terrain) {

    firstTime = 1 ;
```

```
    // SUBAREA LINES: num lines per processor
    // lines to work with, evenly divisible by the number of procs
    subAreaLines = (baseImageSubarea / samples) / (numProcs - 1) ;

    // must have at least 2 lines of data per processor to work with
    if (subAreaLines < 2) {
      subAreaLines = 2 ;
    }

    // LINES: num lines handled by mother
    lines = subAreaLines * (numProcs - 1) ; // num total lines
                                            // for mother to grab

    // LINES LEFT: tracks lines remaining in image
    if (lines > linesLeft) {
      lines = linesLeft ;
    }
    linesInBuffer = lines ;

    // load data
    imgstatus = terrain -> getSubArea (0, 0, lines, samples) ;

    // fetch data from image object
    terrainArr = terrain -> getImage() ;

    // buffer for preloading data while children
    // crunch on the current set
    terrainArrBuffer = new int **  ;
  }


/*****************************************
 loop until entire image has been read,
 filtered, and written out
 *****************************************/

    int whileFlag = 1 ;

    while (whileFlag) {

      // if image failed to load, abort
      if (!myID && !imgstatus) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return 1 ;
      }

      // no data left in file, set up to exit
      if (!myID && !linesLeft) {
         whileFlag = 0 ;
      }
      // let everyone know "whileflag" loop status
      MPI_Bcast (&whileFlag, 1, MPI_INT, 0, MPI_COMM_WORLD) ;


/*****************************************
 Send data out to children
*****************************************/
```

```
        // mother divies up data
        if (!myID && whileFlag) {

          // tell each processor how many lines to work on
          for (int i = 1 ; i < numProcs ; i++) {

            // last few lines in the file?
            if (subAreaLines > linesLeft) {
              subAreaLines = linesLeft ;
            }
            MPI_Send (&subAreaLines, 1, MPI_INT, i, 5, MPI_COMM_WORLD) ;

            // send data to children
            for (int j = 0 ; j < subAreaLines && subAreaLines ; j++) {
              for (int c = 0 ; c < samples ; c++) {
                msgBuffer [c] =  terrainArr[(i - 1) * subAreaLines + j][c][0] ;
              }
              MPI Send (&(msgBuffer[0]), samples, MPI_INT,
                        i, 82, MPI_COMM_WORLD) ;
            }

            // decrement number of lines left to process
            // by the number of lines just passed to a
            // child proc for manipulation
            linesLeft -= subAreaLines ;
          }
        }

        // children receive data
        else if (whileFlag) {
          MPI Recv (&subAreaLines, 1, MPI_INT, 0, 5, MPI_COMM_WORLD, &status) ;
          if (terrainArr) delete terrainArr ;
          terrainArr = new int ** [subAreaLines] ;

          // allocate space for data
          for (int k = 0 ; k < subAreaLines && subAreaLines ; k++ ) {
            terrainArr[k] = new int * [samples] ;
            for (int m = 0 ; m < samples ; m++ ) {
              terrainArr[k][m] = new int[1] ;
            }
          }
          // receive data
          for (int i = 0 ; i < subAreaLines  && subAreaLines ; i++) {
            MPI Recv (&(msgBuffer[0]), samples, MPI_INT,
                      0, 82, MPI_COMM_WORLD,  &status) ;
            for (int c = 0 ; (c < samples) && subAreaLines ; c++) {
              terrainArr[i][c][0] = msgBuffer[c] ;
            }
          }

/*****************************************
CHILDREN: perform calculations on data
*****************************************/

          if (terrainByteSize == 1) {
            range = 127 ;
          }
          else range = 0;
          for ( int line = 0; line < subAreaLines; line++ )
```

```
        {
          for ( int sample = 0; sample < samples; sample++ )
          {
            if ( ( line + vert < subAreaLines ) && ( sample + horiz < samples )  )
            {
              terrainArr[line][sample][0] = terrainArr[line+vert][sample+horiz][0] -
                                            terrainArr[line][sample][0] + range;
            }
          }
        }

/****************************************
 write data to file
****************************************/

      // MOTHER writes terrainArrBuffer to file
      if (!myID) {
        if (firstTime) {
          firstTime = 0 ;
        }
        else {
          cout << "  write " << linesInBuffer << " lines to file " << endl ;
          for (int r = 0 ; r < linesInBuffer ; r++) {
            for (int c = 0 ; c < samples ; c++) {
              fwrite (&(terrainArrBuffer[r][c][0]), terrainByteSize, 1, outputFile)
;
            }
          }
        }

/****************************************
 read next set of data from file
****************************************/

      // MOTHER load data

        if (whileFlag && linesLeft) {
          delete terrainArrBuffer ;
          terrainArrBuffer = new int ** ;

          if (lines > linesLeft) {
            linesInBuffer = linesLeft ;
          }
          imgstatus =
            terrain -> getSubArea (totalLines - linesLeft,
                                   0, linesInBuffer, samples) ;

          // MOTHER fetch data from image object
          terrainArrBuffer = terrain -> getImage() ;
        }
      }


/****************************************
 MOTHER get data from children
****************************************/

      //  children send data
```

```
      if (whileFlag) {
        for (int i = 1 ; i < numProcs ; i++) {
          if (myID == i) {
            MPI_Send (&subAreaLines, 1, MPI_INT, 0, 10, MPI_COMM_WORLD) ;
          }
          else if (!myID) {
            MPI_Recv (&subAreaLines, 1, MPI_INT, i, 10, MPI_COMM_WORLD, &status) ;
          }


          for (int j = 0 ; (j < subAreaLines) && subAreaLines ; j++) {
            for (int c = 0 ; c < samples ; c++) {
              if (myID == i && subAreaLines) {
                msgBuffer[c] = terrainArr[j][c][0] ;
              }
            } // end for c

            if (myID == i && subAreaLines) {
              MPI_Send (&(msgBuffer[0]), samples, MPI_INT, 0, 15, MPI_COMM_WORLD) ;
            }
            if (!myID && subAreaLines) {
              MPI_Recv (&(msgBuffer[0]), samples, MPI_INT, i, 15, MPI_COMM_WORLD, &
status) ;
              for (int c = 0 ; c < samples ; c++) {
                terrainArr[(i - 1) * subAreaLines + j][c][0] = msgBuffer[c] ;
              }
            }
          } // end for j
        } // enf for i
      } // end if whileflag



      // MOTHER swaps terrainArr and terrainArrBuffer
      if (!myID && whileFlag) {
        int *** temp = terrainArr ;
        terrainArr = terrainArrBuffer ;
        terrainArrBuffer = temp ;
        int tempInt = linesInBuffer ;
        linesInBuffer = lines ;
        lines = tempInt ;
      }
//      MPI Barrier(MPI_COMM_WORLD) ;
    }// end big while

/*****************************************
 Bye-bye
*****************************************/

  if (!myID) fclose (outputFile) ;
  if (!myID) cout << "\n\nexiting\n" << endl ;
  MPI Finalize();
  return 0 ;
}
```

```
#ifndef IMAGEMANAGER_H
#define IMAGEMANAGER_H

// NEED: MIPS style headers

/*

file    : imageManager.h

class   : ImageManager
purpose : manage an object and texture via image data

class   : Vertex
purpose : container for holding vertex information

author  :  deborah lee soltesz
date    :  28 september 1999

history :  dls init

*/

#include "image.h"
#include "MIPS_image.h"


// ------------------------------------------------------------
// VERTEX
// container for holding xyz of a vertex and information about
// what texture and where in the texture the vertex maps to

class Vertex {
public:
  float x ;  // x coordinate = image row ; increments as a portion from 0.0 to 1.0
  float y ;  // y coordinate (for terrain, is elevation) ; pixel value in file
  float z ;  // z coordinate = image column ; increments as a portion from 0.0 to 1.
0

  float u ;  // associated row location in texture; increments as a portion from 0.0
to 1.0
  float v ;  // associated column location in texture; increments as a portion from
0.0 to 1.0

  int texture ; // index of associated texture in an array of textures

  Vertex()  { x = y = z = u = v =  0 ; }

  ~Vertex() {;}

} ; // end class VERTEX

// ------------------------------------------------------------
// may need an additional class for grabbing full res, 2 and 3D arrays
// of data to hand off to image renderer



// ------------------------------------------------------------
```

```
class ImageManager {

// MEMBER VARIABLES

private:
  resolution detail          ; // how much detail to work with - compressed for
screen rendering, full for file rendering
  float      imgCompression  ;

  int        tileRows        ; // rows and cols the model is broken into
  int        tileCols        ;

  Image    * texture         ;
  Image   ** textureTiles    ;
  int        texturesCount   ; // number of texture images
  Image    * terrain         ;

  Vertex  ** vertices        ;

// MEMBER FUNCTIONS

public:

  ImageManager () ;
/*
  ImageManager (char * terrainFile, char * textureFile = NULL) {
    detail = compressed ;
    imgCompression = 1.0 ;
    tileRows = tileCols = 0 ;
    numTextures = 0 ;

    terrain    = setImageFile (terrainFile) ;
    texture    = setImageFile (textureFile) ;
    textureTiles= NULL ;
  }
*/


  Image * getTerrainImage () ;

  ImageManager (char * terrainFile, char * textureFile = NULL, int numTextures = 0)
;
  // ------------------------------------------------------------
  // GET TEXTURES
  //  returns all textures as an array of chars
  //  - last element in the array will be NULL
  //  - single image texture will live in a 2 element array

  Image ** getTextures () ;


  // ------------------------------------------------------------
  // GET VERTEX
  //  returns completed vertex

  Vertex * getVertex (int row = 0, int col = 0) ;
    // if not Vertices [row][col] create Vertex
    // return Vertices [row][col]
```

```
  // ------------------------------------------------------------
  // GET VERTICES
  //  returns all vertices

  Vertex ** getVertices () ;

  // ------------------------------------------------------------
  // SET RESOLUTION
  // sets resolution to work with to full or compressed textures

  void setFullResolution () ;
  void setCompressedResolution () ;
  void setResolutionCompression (float compression) ;
  void setMaxHeightWidth (int height, int width) ;


  // --------------------- ---------------------------------------
  // CHANGE TERRAIN DETAIL
  // changes terrain detail amount

  void changeTerrainDetail (int rows, int cols) ;


  // ------------------------------------------------------------
  // CHANGE <data file>
  // changes terrain or texture

  void changeTerrainFile (char * filename) ;
  void changeTextureFile (char * filename) ;

  // ------------------------------------------------------------
  // ADD TEXTURE FILE
  // put an additional texture file in list

  void addTextureFile (char * filename) ;


private:

  // ------------------------------------------------------------
  // SET IMAGE FILE
  //  creates appropriate image object based on file type

  Image * setImageFile (char * fileName) ;
    // pull extension off of file name
    // determine file type
    // instantiate correct Image object with file name and file type
    // return that object

} ; // end class IMAGE MANAGER

#endif // IMAGEMANAGER_H
```

```
/*

file    : imageManager.cpp

class   : ImageManager
purpose : manage an object and texture via image data

class   : Vertex
purpose : container for holding vertex information

author  :  deborah lee soltesz
date    :  28 september 1999

history :  dls init

*/

#include "imageManager.h"

  ImageManager::ImageManager () {
    detail = compressed ;
    imgCompression = 1.0 ;
    tileRows = tileCols = 0 ;

    texture  = NULL ;
    terrain  = NULL ;
    vertices = NULL ;
  }

  ImageManager::ImageManager (char * terrainFile,
                              char * textureFile/* = NULL */,
                              int numTextures/* = 0*/) {
    detail = compressed ;
    imgCompression = 1.0 ;
    tileRows = tileCols = 0 ;
    texturesCount = numTextures ;

    terrain = setImageFile (terrainFile) ;

    if (texturesCount) {
      textureTiles    = new Image * [texturesCount + 1] ;
      textureTiles[0] = setImageFile (textureFile) ;
      for (int i = 1 ; i < texturesCount + 1 ; i++) {
        textureTiles[i] = NULL ;
      }
    }
    else if (textureFile) {
      texture    = setImageFile (textureFile) ;
      textureTiles= NULL ;
    }

  }


  Image * ImageManager::getTerrainImage () { return terrain ; }


  // ------------------------------------------------------------
  // GET TEXTURES
```

```
//  returns all textures as an array of chars
//  - last element in the array will be NULL
//  - single image texture will live in a 2 element array

Image ** ImageManager::getTextures () { return textureTiles ;}


// ------------------------------------------------------------
// GET VERTEX
//  returns completed vertex

Vertex * ImageManager::getVertex (int row = 0, int col = 0) {
  return &vertices[row][col] ;
}
  // if not Vertices [row][col] create Vertex
  // return Vertices [row][col]


// ------------------------------------------------------------
// GET VERTICES
//  returns all vertices

Vertex ** ImageManager::getVertices () { return vertices ;}

// ------------------------------------------------------------
// SET RESOLUTION
// sets resolution to work with to full or compressed textures

void ImageManager::setFullResolution () { detail = full ;}
void ImageManager::setCompressedResolution () { detail = compressed ;}

void ImageManager::setResolutionCompression (float compression) {
  detail = compressed ;
  imgCompression = compression ;
  if (terrain) {
    terrain -> setCompressedResolution (compression) ;
  }
  if (textureTiles) {
    for (int i = 0 ; i < texturesCount ; i++) {
      textureTiles[i] -> setCompressedResolution(compression) ;
    }
  }
  if (texture) {
    texture -> setCompressedResolution(compression) ;
  }
}

void ImageManager::setMaxHeightWidth (int height, int width) {
  if (terrain) {
    terrain -> setCompressedResolution () ;
  }
}


// -------------------- ---------------------------------------
// CHANGE TERRAIN DETAIL
// changes terrain detail amount

void ImageManager::changeTerrainDetail (int rows, int cols) {;}
```

```
// ------------------------------------------------------------
// CHANGE <data file>
// changes terrain or texture

void ImageManager::changeTerrainFile (char * filename) {;}
void ImageManager::changeTextureFile (char * filename) {;}

// ------------------------------------------------------------
// ADD TEXTURE FILE
// put an additional texture file in list

void ImageManager::addTextureFile (char * filename) {;}

// ------------------------------------------------------------
// SET IMAGE FILE
//  creates appropriate image object based on file type

Image * ImageManager::setImageFile (char * fileName) {

  DEBUG_MAC (cout << "filename: " << fileName << endl ;)

  Image * new_image = NULL ;
  imgType new_format = imgError ;

  int len = strlen (fileName) ;
  int i = len ;
  int extLen = 0 ;

  for (i ; i >= 0 && fileName[i] != '.' ; i--) {
    extLen++ ;
  }
  i++ ;

  char ext[extLen + 1] ;
  int j = 0 ;
  for (i ; i < len ; i++) {
    ext[j++] = fileName[i] ;
  }
  ext[j] = '\0' ;

  char new_fileNameBase[len - extLen+1] ;

  for (i = 0 ; i < (len - extLen) && fileName[i] != '.' ; i++) {
    new_fileNameBase[i] = fileName[i] ;
    DEBUG_MAC(cout << new_fileNameBase[i];)
  }

  DEBUG_MAC(cout << endl;)

  new_fileNameBase[i] = '\0' ;


  DEBUG MAC( cout << "filename base: " << new_fileNameBase
                  << " extension: " << ext << endl ;)

  if (!strcmp("img", ext)) {
    new_format = MIPS ;
```

```
    DEBUG MAC(cout << " instantiate a MIPS image" << endl ;)
    new_image = new MIPS Image (new_fileNameBase, 1.0) ;
    DEBUG MAC(cout << "return from making mips image, lines "
                   << new_image -> getFileLines()
                   << " samples " << new_image -> getFileSamples ()
              << endl ;)
  }
  else new_format = imgError ;

  DEBUG MAC (cout << "filename base: " << new_fileNameBase
                  << " extension: " << ext << endl ;)

  if (!new_image || (new_image -> getColorDepth () == colorError) ) {
    return NULL ;
  }
  return new_image ;

}
```

```
#ifndef IMAGE_H
#define IMAGE_H


// NEED: MIPS style headers

/*

file    : image.h

class   : Image
purpose : parent class for various image types differentiated by their formats

class   : imgString
purpose : container for holding image data as a string

author  :  deborah lee soltesz
date    :  28 september 1999

history :  dls init

*/

#ifdef  DEBUG
#define DEBUG_MAC(x) x
#else
#define DEBUG_MAC(x)
#endif


#include <string.h>
#include <stdio.h>
#include <iostream.h>

// ------------------------------------------------------------
// ENUMERATED TYPES

enum imgType    { imgError, MIPS } ;
enum resolution { full, compressed } ;
enum colorDepth { colorError, grey, color = 3, alpha } ;

// ------------------------------------------------------------
// IMG STRING
// container for holding an image as a string to send to hardware/renderer

class imgString {
public:
  int    width ;
  int    height ;
  char * texture ;

  imgString () {
    width = height = 0 ;
    texture = NULL ;
  }

  ~imgString () {
    delete texture ;
  }
```

```
} ; // end class IMG STRING


// ------------------------------------------------------------

class Image {

// MEMBER VARIABLES

protected:
  int        lines        ; // number of lines in the image; aka rows, height
  int        samples      ; // number of samples in the image ; aka columns, width

  int        fileLines    ; // number of lines in the image; aka rows, height
  int        fileSamples  ; // number of samples in the image ; aka columns, width

  FILE     * imageFile    ; // pointer to an imagefile

  int    *** image        ; // 3D array of pixel values
  int    *** tempImage    ; // 3D array of pixel values for temporary storage

  char     * fileNameBase ; // file name with extension removed
  imgType    format       ; // file format
  colorDepth colors       ;
  resolution detail       ;
  float      increment    ; // how much to increment through the file (for full,
increment = 1)

  int        byteSize     ;

// MEMBER FUNCTIONS

public:

  Image () {
    fileLines = fileSamples = lines = samples = 0 ;
    image = NULL ;
    tempImage = NULL ;
    fileNameBase = NULL ;
    format = imgError ;
    colors = colorError ;
    detail = compressed ;
    increment = 5.0     ;
  }

  Image (char * new_fileNameBase, imgType new_format, float compression = .2) {
    fileLines = fileSamples = lines = samples = 0 ;
    image        = NULL ;
    tempImage    = NULL ;

    fileNameBase = new char[strlen(new_fileNameBase)+1] ;
    strcpy (fileNameBase, new_fileNameBase) ;
    format       = new_format ;
    colors       = colorError ;
    detail       = compressed ;
    increment    = 1.0 / compression ;
  }

  ~Image () {
```

```
    fclose (imageFile) ;
    delete image ;
    delete tempImage ;
    delete fileNameBase ;
  }

  // ------------------------------------------------------------
  // SET RESOLUTION
  // sets resolution to work with to full or compressed image

  void setFullResolution () ;
  void setCompressedResolution (float compression = .2) ; // defaults to 20%


  // get image attributes

  int getImgLines    () { return lines       ; }
  int getImgSamples  () { return samples     ; }
  int getFileLines   () { return fileLines   ; }
  int getFileSamples () { return fileSamples ; }
  int getByteSize    () { return byteSize    ; }

  colorDepth getColorDepth () { return colors ; }

  int *** getImage       () { return image       ; }

  // ------------------------------------------------------------
  // REDUCE TO CHAR
  // reduces image values with byteSize > 1 to byteSize = 1 sized values

  int *** reduceToChar (int *** imageVals) ;

  // ------------------------------------------------------------
  // GET SUBAREA
  // returns a full resolution subarea of the image

  int     getSubArea        (int startLine, int startSample, int height, int width)
;


  // ------------------------------------------------------------
  // GET SUB-SAMPLING
  // returns a compressed version of the entire image

  int *** getSubSampling    (float compression) ;
  int *** getSubSampling    (int height,  int width) ;


  // ------------------------------------------------------------
  // GET SUBAREA SUB-SAMPling
  // returns a compressed version of a subarea of the iamge

  int *** getSubAreaSubSamp (int startLine, int startSample, int height, int width,
float compression) ;
  int *** getSubAreaSubSamp (int startLine, int startSample, int height, int width,
int destHeight, int destWidth) ;


  // ------------------------------------------------------------
```

```
  // GET IMG RGB STRING
  // returns image as a string of pixel color values in rgb order

  imgString * getImgRGBString   () ; // gets full image
  imgString * getImgRGBString   (int startLine, int startSample, int height, int
width, float compression = 1.0) ;
  imgString * getImgRGBString   (int startLine, int startSample, int height, int
width, int destHeight, int destWidth) ;

  // ------------------------------------------------------------
  // GET IMG RGBA STRING
  // returns image as a string of pixel color values in rgba order

  imgString * getImgRGBAString   () ; // gets full image
  imgString * getImgRGBAString   (int startLine, int startSample, int height, int
width, float compression = 1.0) ;
  imgString * getImgRGBAString   (int startLine, int startSample, int height, int
width, int destHeight, int destWidth) ;

  // ------------------------------------------------------------
  // GET IMG GREY STRING
  // returns grey image as a string of pixel values

  imgString * getImgGreyString   () ; // gets full image
  imgString * getImgGreyString   (int startLine, int startSample, int height, int
width, float compression = 1.0) ;
  imgString * getImgGreyString   (int startLine, int startSample, int height, int
width, int destHeight, int destWidth) ;


protected:

  virtual int parseImage (int startLine, int startSample, int endLine, int endSample
) = 0 ;

} ; // end class IMAGE


#endif // IMAGE_H
```

```
/*

file    : image.cpp

class   : Image
purpose : parent class for various image types
          differentiated by their formats

class   : imgString
purpose : container for holding image data as a string

author  :  deborah lee soltesz
date    :  28 september 1999

history :  dls init

*/

#include "image.h"

// ------------------------------------------------------------
// ENUMERATED TYPES

// enum imgType    { imgError, MIPS } ;
// enum resolution { full, compressed } ;
// enum colorDepth { colorError, grey, color = 3, alpha } ;



  // ------------------------------------------------------------
  // SET RESOLUTION
  // sets resolution to work with to full or compressed image

  void Image::setFullResolution () {
    detail = full ;
  }

  void Image::setCompressedResolution (float compression /*= .2*/) {
    increment = 1.0 / compression ;
  }


  // ---DEBUG MAC--------------------------------------------------------
  // GET SUBAREA
  // returns a full resolution subarea of the image

  int Image::getSubArea (int startLine, int startSample, int height, int width) {


    DEBUG MAC(cout << "Image::getSubArea starting - SL " << startLine
                   << " SS " << startSample  << " H " << height
                   << " W " << width << endl ;)
    setFullResolution () ;
    int status =
      parseImage (startLine, startSample, startLine + height - 1,
                  startSample + width - 1) ;

    DEBUG_MAC(cout << "Image::getSubArea parseImage returned "
```

```
                << status << " " << image << endl ;)

  return status ;
}


// ------------------------------------------------------------
// GET SUB-SAMPLING
// returns a compressed version of the entire image

int *** Image::getSubSampling    (float compression) {
  setCompressedResolution (compression) ;
  parseImage (0, 0, fileLines, fileSamples) ;
  return image ;
}

int *** Image::getSubSampling    (int height,  int width) {
  setCompressedResolution (height / lines) ;
  parseImage (0, 0, fileLines, fileSamples) ;
  return image ;
}


// ------------------------------------------------------------
// GET SUBAREA SUB-SAMPling
// returns a compressed version of a subarea of the iamge

int *** Image::getSubAreaSubSamp (int startLine, int startSample,
                                  int height, int width, float compression) {
  setCompressedResolution (compression) ;
  parseImage (startLine, startSample, startLine + height, startSample + width) ;
  return image ;
}

int *** Image::getSubAreaSubSamp (int startLine, int startSample,
                                  int height, int width,
                                  int destHeight, int destWidth) {
  setCompressedResolution (height / lines) ;
  parseImage (startLine, startSample, startLine + height, startSample + width) ;
  return image ;
}

// ------------------------------------------------------------
// REDUCE TO CHAR
// reduces image values with byteSize > 1 to byteSize = 1 sized values

int *** Image::reduceToChar (int *** imageVals) {
  int tempVal = 0, maxVal = 256 ;

  for (int line = 0 ; line < lines ; line++) {
    for (int sample = 0 ; sample < samples ; sample++) {

      if (byteSize > 1) {
        for (int pow = 2 ; pow < byteSize ; pow++) {
          maxVal *= maxVal ;
        }
      }
      maxVal -= 1 ;
```

```
        for (int c = 0 ; c < colors ; c++) {
          imageVals[line][sample][c] =
            (int)( ((float)imageVals[line][sample][c] / (float)maxVal + .5) * 255 )
;
        }
      }
    }
    return image ;
  }

  // ------------------------------------------------------------
  // GET IMG RGB STRING
  // returns image as a string of pixel color values in rgb order

  imgString * Image::getImgRGBString   () {
    setFullResolution () ;
    colors = color ;
    int *** imageValues = getSubSampling (1.0) ;
    reduceToChar (imageValues) ;

    imgString * returnString = new imgString ;
    (*returnString).height = lines ;
    (*returnString).width  = samples ;

    char * values = new char [lines * samples] ;

    for (int line = 0 ; line < lines ; line++) {
      for (int sample = 0 ; sample < samples ; sample++) {
        for (int c = 0 ; c < colors ; c++) {
          values [line * sample + 0] = (char)imageValues [line][sample][0] ;
        }
      }
    }
    (*returnString).texture = values ;
    return returnString ;
  }

  imgString * Image::getImgRGBString   (int startLine, int startSample,
                                        int height, int width,
                                        float compression /*= 1.0*/) {

    setCompressedResolution (compression) ;
    colors = color ;
    int *** imageValues =
      getSubAreaSubSamp (startLine, startSample, height, width, compression) ;
    reduceToChar (imageValues) ;

    imgString * returnString = new imgString ;
    (*returnString).height = (int)( (float)lines   * compression + .5) ;
    (*returnString).width  = (int)( (float)samples * compression + .5) ;

    char values [(*returnString).height * (*returnString).width] ;

    for (int line = 0 ; line < (*returnString).height ; line++) {
      for (int sample = 0 ; sample < (*returnString).width ; sample++) {
        for (int c = 0 ; c < colors ; c++) {
          values [line * sample + 0] = (char)imageValues [line][sample][0] ;
        }
      }
```

```
    }
    (*returnString).texture = values ;
    return returnString ;
  }

  imgString * Image::getImgRGBString   (int startLine, int startSample,
                                        int height, int width,
                                        int destHeight, int destWidth) {

    return getImgRGBString (startLine, startSample, height, width, destHeight /
height) ;

  }

  // ------------------------------------------------------------
  // GET IMG RGBA STRING
  // returns image as a string of pixel color values in rgba order

  imgString * Image::getImgRGBAString   () {
    setFullResolution () ;
    colors = alpha ;
    int *** imageValues = getSubSampling (1.0) ;
    reduceToChar (imageValues) ;

    imgString * returnString = new imgString ;
    (*returnString).height = lines ;
    (*returnString).width  = samples ;

    char values [lines * samples] ;

    for (int line = 0 ; line < lines ; line++) {
      for (int sample = 0 ; sample < samples ; sample++) {
        for (int c = 0 ; c < colors ; c++) {
          values [line * sample + 0] = (char)imageValues [line][sample][0] ;
        }
      }
    }
    (*returnString).texture = values ;
    return returnString ;
  }

  imgString * Image::getImgRGBAString   (int startLine, int startSample,
                                        int height, int width,
                                        float compression /*= 1.0*/) {

    setCompressedResolution (compression) ;
    colors = alpha ;
    int *** imageValues = getSubAreaSubSamp (startLine, startSample,
                                             height, width, compression) ;
    reduceToChar (imageValues) ;

    imgString * returnString = new imgString ;
    (*returnString).height = (int)( (float)lines   * compression + .5) ;
    (*returnString).width  = (int)( (float)samples * compression + .5) ;

    char values [(*returnString).height * (*returnString).width] ;

    for (int line = 0 ; line < (*returnString).height ; line++) {
      for (int sample = 0 ; sample < (*returnString).width ; sample++) {
```

```
        for (int c = 0 ; c < colors ; c++) {
          values [line * sample + 0] = (char)imageValues [line][sample][0] ;
        }
      }
    }
    (*returnString).texture = values ;
    return returnString ;
  }

  imgString * Image::getImgRGBAString   (int startLine, int startSample,
                                         int height, int width,
                                         int destHeight, int destWidth) {
    return getImgRGBAString (startLine, startSample, height, width, destHeight /
height) ;
  }

  // ------------------------------------------------------------
  // GET IMG GREY STRING
  // returns grey image as a string of pixel values

  imgString * Image::getImgGreyString   () {
    setFullResolution () ;
    colors = grey ;
    int *** imageValues = getSubSampling (1.0) ;
    reduceToChar (imageValues) ;

    imgString * returnString = new imgString ;
    (*returnString).height = lines ;
    (*returnString).width  = samples ;

    char values [lines * samples] ;

    for (int line = 0 ; line < lines ; line++) {
      for (int sample = 0 ; sample < samples ; sample++) {
        for (int c = 0 ; c < colors ; c++) {
          values [line * sample + 0] = (char)imageValues [line][sample][0] ;
        }
      }
    }
    (*returnString).texture = values ;
    return returnString ;
  }

  imgString * Image::getImgGreyString   (int startLine, int startSample,
                                         int height, int width,
                                         float compression /*= 1.0*/) {
    setCompressedResolution (compression) ;
    colors = grey ;
    int *** imageValues = getSubAreaSubSamp (startLine, startSample,
                                             height, width, compression) ;
    reduceToChar (imageValues) ;

    imgString * returnString = new imgString ;
    (*returnString).height = (int)( (float)lines   * compression + .5) ;
    (*returnString).width  = (int)( (float)samples * compression + .5) ;

    char values [(*returnString).height * (*returnString).width] ;

    for (int line = 0 ; line < (*returnString).height ; line++) {
```

```
      for (int sample = 0 ; sample < (*returnString).width ; sample++) {
        for (int c = 0 ; c < colors ; c++) {
          values [line * sample + 0] = (char)imageValues [line][sample][0] ;
        }
      }
    }
    (*returnString).texture = values ;
    return returnString ;
  }

  imgString * Image::getImgGreyString   (int startLine, int startSample,
                                         int height, int width,
                                         int destHeight, int destWidth) {

    return getImgGreyString (startLine, startSample, height, width, destHeight /
height) ;
  }
```

```
#ifndef MIPS_IMAGE_H
#define MIPS_IMAGE_H


// NEED: MIPS style headers

/*

file    : MIPS_image.h

class   : MIPS Image
purpose : child class of Image for reading USGS MIPS images

author  :  deborah lee soltesz
date    :  28 september 1999

history :  dls init

*/

#include "image.h"
#include <fstream.h>

enum MIPS_imageType { ubyte, uword, sword } ;

// ------------------------------------------------------------

class MIPS_Image : public Image {

private:
  char     * fileNameBaseGreen ; // file name for green channel with extension
removed
  char     * fileNameBaseBlue  ; // file name for blue channel with extension
removed
  FILE     * imageFileGreen    ; // pointer to the GREEN imagefile
  FILE     * imageFileBlue     ; // pointer to the BLUE  imagefile

  MIPS_imageType mipsType      ;


// MEMBER FUNCTIONS

public:

  MIPS_Image () : Image () { ; }

  MIPS Image (char * fileNameBase, char * fileNameBaseGreen,
              char * fileNameBaseBlue) : Image (fileNameBase, MIPS) {
    // open label files

    //TO DO: finish label file assembly -- allocate enough space in
    //       label string for base + extension
    colors = color ;
    char * labelFileR ;
    strcpy (labelFileR, fileNameBase) ;
    char * labelFileG ;
    strcpy (labelFileG, fileNameBaseGreen) ;
    char * labelFileB ;
    strcpy (labelFileB, fileNameBaseBlue) ;
```

```
  ifstream redLabel   (strcat (labelFileR, ".lbl"))  ;
  ifstream greenLabel (strcat (labelFileG, ".lbl"))  ;
  ifstream blueLabel  (strcat (labelFileB, ".lbl"))  ;


  if (redLabel.fail() || greenLabel.fail() || blueLabel.fail() ) {
    colors = colorError ;
    return ;
  }

  int rwidth = 0, gwidth = 0, bwidth = 0 ;
  int rheight = 0, gheight = 0, bheight = 0 ;

  int status = 0 ;

  status = readLabel (redLabel, rwidth, rheight) ;
  status = readLabel (greenLabel, rwidth, rheight) ;
  status = readLabel (blueLabel, rwidth, rheight) ;

  // error with mips image label file, exit

  if (!status) {
    // ERROR: one image height or width = 0, error
    colors = colorError ;
    return ;
  }

  // make sure all images are the same size (return error)
  int fileSizeFlag = (rwidth == bwidth == gwidth) &&
                     (rheight == bheight == gheight) ;

  if (!fileSizeFlag) {
  // ERROR: images aren't same lines and samples
    colors = colorError ;
    return ;
  }
}

MIPS Image (char * fileNameBase, double comp = 0.2)
: Image (fileNameBase, MIPS) {
  // open label files
  colors = grey ;
  char labelFile[strlen(fileNameBase) + 5] ;

  strcpy (labelFile, fileNameBase) ;
  strcat (labelFile, ".lbl") ;

  ifstream redLabel (labelFile)  ;

  if (redLabel.fail()) {
    colors = colorError ;
    return ;
  }

  int rwidth  = 0 ;
  int rheight = 0 ;

  int status  = 0 ;
```

```
    status = readLabel (redLabel, fileSamples, fileLines) ;
    cout << "done reading MIPS label: lines " << fileLines
         << " samples " << fileSamples << endl ;

    // error with mips image label file, exit

    if (!status) {
      // ERROR: one image height or width = 0, error
      cout << "Error reading MIPS label " << endl ;
      colors = colorError ;
      return ;
    }
    samples = fileSamples ;
    lines   = fileLines ;
  }

  ~MIPS_Image () {

    delete fileNameBaseGreen ;
    delete fileNameBaseBlue  ;
    fclose (imageFileGreen)  ;
    fclose (imageFileBlue)   ;

  }


protected:

  //virtual
    int parseImage (int startLine, int startSample, int endLine, int endSample) ;

  // parses MIPS label file
    int readLabel (ifstream & labelFile, int & wid, int & hgt) ;

} ; // end class MIPS_Image


#endif // MIPS_IMAGE_H
```

```
/*

file    : imageManager.cpp

class   : ImageManager
purpose : manage an object and texture via image data

class   : Vertex
purpose : container for holding vertex information

author  :  deborah lee soltesz
date    :  28 september 1999

history :  dls init

*/

#include "imageManager.h"

  ImageManager::ImageManager () {
    detail = compressed ;
    imgCompression = 1.0 ;
    tileRows = tileCols = 0 ;

    texture  = NULL ;
    terrain  = NULL ;
    vertices = NULL ;
  }

  ImageManager::ImageManager (char * terrainFile,
                              char * textureFile/* = NULL */,
                              int numTextures/* = 0*/) {
    detail = compressed ;
    imgCompression = 1.0 ;
    tileRows = tileCols = 0 ;
    texturesCount = numTextures ;

    terrain = setImageFile (terrainFile) ;

    if (texturesCount) {
      textureTiles    = new Image * [texturesCount + 1] ;
      textureTiles[0] = setImageFile (textureFile) ;
      for (int i = 1 ; i < texturesCount + 1 ; i++) {
        textureTiles[i] = NULL ;
      }
    }
    else if (textureFile) {
      texture    = setImageFile (textureFile) ;
      textureTiles= NULL ;
    }

  }


  Image * ImageManager::getTerrainImage () { return terrain ; }


  // ------------------------------------------------------------
  // GET TEXTURES
```

```
//  returns all textures as an array of chars
//  - last element in the array will be NULL
//  - single image texture will live in a 2 element array

Image ** ImageManager::getTextures () { return textureTiles ;}


// ------------------------------------------------------------
// GET VERTEX
//  returns completed vertex

Vertex * ImageManager::getVertex (int row = 0, int col = 0) {
  return &vertices[row][col] ;
}
  // if not Vertices [row][col] create Vertex
  // return Vertices [row][col]


// ------------------------------------------------------------
// GET VERTICES
//  returns all vertices

Vertex ** ImageManager::getVertices () { return vertices ;}

// ------------------------------------------------------------
// SET RESOLUTION
// sets resolution to work with to full or compressed textures

void ImageManager::setFullResolution () { detail = full ;}
void ImageManager::setCompressedResolution () { detail = compressed ;}

void ImageManager::setResolutionCompression (float compression) {
  detail = compressed ;
  imgCompression = compression ;
  if (terrain) {
    terrain -> setCompressedResolution (compression) ;
  }
  if (textureTiles) {
    for (int i = 0 ; i < texturesCount ; i++) {
      textureTiles[i] -> setCompressedResolution(compression) ;
    }
  }
  if (texture) {
    texture -> setCompressedResolution(compression) ;
  }
}

void ImageManager::setMaxHeightWidth (int height, int width) {
  if (terrain) {
    terrain -> setCompressedResolution () ;
  }
}


// -------------------- ---------------------------------------
// CHANGE TERRAIN DETAIL
// changes terrain detail amount

void ImageManager::changeTerrainDetail (int rows, int cols) {;}
```

```
// ------------------------------------------------------------
// CHANGE <data file>
// changes terrain or texture

void ImageManager::changeTerrainFile (char * filename) {;}
void ImageManager::changeTextureFile (char * filename) {;}

// ------------------------------------------------------------
// ADD TEXTURE FILE
// put an additional texture file in list

void ImageManager::addTextureFile (char * filename) {;}

// ------------------------------------------------------------
// SET IMAGE FILE
//  creates appropriate image object based on file type

Image * ImageManager::setImageFile (char * fileName) {

  DEBUG_MAC (cout << "filename: " << fileName << endl ;)

  Image * new_image = NULL ;
  imgType new_format = imgError ;

  int len = strlen (fileName) ;
  int i = len ;
  int extLen = 0 ;

  for (i ; i >= 0 && fileName[i] != '.' ; i--) {
    extLen++ ;
  }
  i++ ;

  char ext[extLen + 1] ;
  int j = 0 ;
  for (i ; i < len ; i++) {
    ext[j++] = fileName[i] ;
  }
  ext[j] = '\0' ;

  char new_fileNameBase[len - extLen+1] ;

  for (i = 0 ; i < (len - extLen) && fileName[i] != '.' ; i++) {
    new_fileNameBase[i] = fileName[i] ;
    DEBUG_MAC(cout << new_fileNameBase[i];)
  }

  DEBUG_MAC(cout << endl;)

  new_fileNameBase[i] = '\0' ;


  DEBUG MAC( cout << "filename base: " << new_fileNameBase
                  << " extension: " << ext << endl ;)

  if (!strcmp("img", ext)) {
    new_format = MIPS ;
```

```
    DEBUG MAC(cout << " instantiate a MIPS image" << endl ;)
    new_image = new MIPS Image (new_fileNameBase, 1.0) ;
    DEBUG MAC(cout << "return from making mips image, lines "
                   << new_image -> getFileLines()
                   << " samples " << new_image -> getFileSamples ()
              << endl ;)
  }
  else new_format = imgError ;

  DEBUG MAC (cout << "filename base: " << new_fileNameBase
                  << " extension: " << ext << endl ;)

  if (!new_image || (new_image -> getColorDepth () == colorError) ) {
    return NULL ;
  }
  return new_image ;

}
```