Internal Use Only
非公開

TR－A－0143 13

Programs used in the Accurate Reconstruction of 3D Scenes from Multiple Imprecise and Uncertain Data

Philippe Quinio

1992. 4. 9

ATR視聴覚機構研究所

〒619-02 京都府相楽郡精華町光台2-2 ☎07749-5-1411

ATR Auditory and Visual Perception Research Laboratories

2-2, Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-02 Japan

Telephone: +81-7749-5-1411
Facsimile: +81-7749-5-1408

Programs used in the Accurate Reconstruction of 3D Scenes from Multiple Imprecise and Uncertain Data

Philippe Quinio

ATR Auditory and Visual Perception Research Laboratories
Advanced Telecommunications Research Institute
2-2 Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-02, Japan

CONTENTS

```
/*********************************************************************/
/*                                                                   */
/*                        AVS module                                 */
/*                                                                   */
/*  Computes the disparity field between 2 (front end) stereo images */
/*                                                                   */
/*                        Ph. Quinio  (03/03/92)                     */
/*                                                                   */
/*********************************************************************/

/* include files */
#include <stdio.h>
#include <math.h>
#include <strings.h>
#include <fcntl.h>
#include <my_functions_and_macros.h>
#include <cm/paris.h>
#include <cm/display.h>
#include <cm/cmsr.h>
#include <cm/cmfb.h>
#include <cm/cm_file.h>
#include <cm/cm_errno.h>
#include <avs/avs.h>
#include <avs/field.h>
#include <avs/flow.h>

/* Used for AVSmessage() */
static char file_version[] = "@(#)avs_stereo.c  Quinio 92/03/05";

/* the following is an abbreviation... */
#define CM_SUBFIELD( field, offset ) CM_add_offset_to_field_id( field, offset )

/* CM safety mode */
#define SAFETY_MODE     NO

/* image constants */
#define IMAGE_SIZE      512
#define IMAGE_SIZE_LEN  9       /* nb of bits in IMAGE_SIZE */
#define BITS_PER_PIXEL  8
#define INTENSITY_MAX   255     /* = power(2, BITS_PER_PIXEL)-1 */
#define SCAN_LINES_NB   478     /* number of scan lines for NTSC */

/* window sizes */
#define DISPARITY_MAX   256 /* must be a power of 2 */
#define DISPARITY_LEN   9   /* must be large enough to store DISPARITY_MAX
                               signed  */
#define ALIGNMENT_MAX   1 /* must be a power of 2 and <= 64 */
#define ALIGNMENT_LEN   2 /* must be large enough to store ALIGNMENT_MAX
                             signed and must be <= 8 */
#define CORRELATION_WINDOW_SIZE  17   /* must be either 3, 5, 9, 17, 33, 65 etc */
#define CORRELATION_WINDOW_LEN   5    /* nb of bits in CORRELATION_WINDOW_SIZE */
#define INV_CORRELATION_SQUARED_SIZE 1.0/(CORRELATION_WINDOW_SIZE*CORRELATION_WINDOW_SIZE)

#define MEDIAN_SIZE 3           /* must be either 3, 5, 9, 17, 33 etc */
#define MEDIAN_SIZE_LEN 2               /* nb of bits in MEDIAN_SIZE */
#define MEDIAN_SIZE_SQUARED  (MEDIAN_SIZE*MEDIAN_SIZE)

#define FPOINT_LEN      FLEN * DIMENSION        /* single precision */
#define DPOINT_LEN      DLEN * DIMENSION        /* double precision */
#define NB_FACES        14
#define OBJECT_LEN      NB_FACES * ( FPOINT_LEN + 2 )

#define DIMENSION       3

#define CENTER_OFFSET_X (IMAGE_SIZE/2)
#define CENTER_OFFSET_Y (IMAGE_SIZE/2)

/* structures and global types */
typedef enum { MORAVEC, SOBEL, REGULAR } feature_extractor_t;

typedef CM_field_id_t coordinates_t[ DIMENSION ];

typedef struct {
```

```
  CM_field_id_t total;
  coordinates_t normal_vector;
  CM_field_id_t sign_bit;
  CM_field_id_t face_bit;
} face_t;

typedef struct {
  CM_field_id_t total;
  face_t        face[ NB_FACES ];
} object_t;

typedef struct {
  float vector[ DIMENSION ];
  float imprecision;
} translation_t;

typedef struct {
  float min;
  float max;
} interval_t;

typedef struct {
  translation_t translation;
  interval_t    yaw;
} displacement_t;

typedef struct {
  interval_t    focal;
  interval_t    baseline;
  interval_t    pitch;
} config_t;

/* global variables */
CM_geometry_id_t stereo_pair_geom;
CM_vp_set_id_t stereo_pair_grid;
CM_geometry_id_t image_geom;
CM_vp_set_id_t image_grid;
CM_geometry_id_t square_geom;
CM_vp_set_id_t square_grid;
CM_geometry_id_t geometry_1D;

CM_vp_set_id_t list;    /* for compact object list */

CM_field_id_t stereo_pair, stereo_pair_mean, stereo_pair_norm;
CM_field_id_t dual_stereo_pair_info, dual_stereo_pair;
CM_field_id_t dual_stereo_pair_mean, dual_stereo_pair_norm;
CM_field_id_t coords, coord[ 3 ];
CM_field_id_t feature_bit;
CM_field_id_t disparity, alignment, correlation;
CM_field_id_t temp, temp_float1, temp_float2, temp_float3;

CM_field_id_t image0, image1, image_bit;

typedef enum { NO_ERROR, NO_object,
               CANNOT_OPEN_DATA,
               CANNOT_OPEN_LEFT_DATA, CANNOT_OPEN_RIGHT_DATA,
               DATA_FILE_ERROR, DATA_WRITING_ERROR,
               CANNOT_OPEN_INFO, INFO_FILE_ERROR,
               CANNOT_OPEN_CONFIG, CONFIG_FILE_ERROR,
               SINGULARITY_ERROR, END_OF_INFO_FILE } status_t;

/* feature extractor type */
feature_extractor_t FEATURE_EXTRACTOR = SOBEL;

#define LEFT_IMAGE_INPUT "Left image"
#define RIGHT_IMAGE_INPUT "Right image"
#define CM_FIELD_OUTPUT "Stereo data DV file"
#define DATA_FILENAME_PARAM "Data filename"
#define DEFAULT_DATA_FILENAME "/data/data"
#define TRANSLATION_X_PARAM "Translation x"
#define TRANSLATION_Y_PARAM "Translation y"
#define TRANSLATION_Z_PARAM "Translation z"
#define POSITIONAL_PRECISION_PARAM "Positional precision"
```

```
#define YAW_MIN_PARAM "Yaw min"
#define YAW_MAX_PARAM "Yaw max"
#define FOCAL_MIN_PARAM "Focal length min"
#define FOCAL_MAX_PARAM "Focal length max"
#define MAX_FOCAL 10000.0 /* in pixels */
#define DEFAULT_FOCAL 1000.0
#define BASELINE_MIN_PARAM "Baseline min"
#define BASELINE_MAX_PARAM "Baseline max"
#define MAX_BASELINE 10.0 /* in meters */
#define DEFAULT_BASELINE .1
#define PITCH_MIN_PARAM "Pitch min"
#define PITCH_MAX_PARAM "Pitch max"
#define DEFAULT_PITCH 0.0
#define CORRELATION_THRESHOLD_PARAM "Correlation th."
#define DEFAULT_CORRELATION_THRESHOLD .98
#define FEATURE_EXTRACTOR_PARAM "Feature extractor"
#define EXTRACTOR_CHOICES "Sobel,Moravec,Regular"
#define DELIMITER ","
#define DEFAULT_EXTRACTOR "Sobel"
#define EXTRACTOR_THRESHOLD_PARAM "Extractor th."
#define DEFAULT_EXTRACTOR_THRESHOLD .001
#define VISUALIZATION_PARAM "Visualization"
#define DEFAULT_VISUALIZATION NO

#define DEFAULT_PATH "."
char *DV_PATH = DEFAULT_PATH;

boolean VISUALIZATION;
boolean DISPLAY_INITIALIZED = NO;

/*****************************************************************************/
AVSinit_modules()
{
    int stereo();

    AVSmodule_from_desc( stereo );
}

/*****************************************************************************/
static
stereo()
{
    int param0, param1, param2, param3, param4, param5;
    int param6, param7, param8, param9, param10, param11, param12;
    int param13, param14, param15, param16;
    int stereo_compute(), stereo_init(), stereo_destroy();
    char *ptr;
    char string[ MISC_BUFFER_SIZE ];

    /* Set the module name */
    AVSset_module_name( "stereo CM", MODULE_MAPPER );

    /* get the path to home directory on DataVault */
    ptr = (char *)getenv( "DVWD" );
    if( ptr != NULL )
      strcpy( DV_PATH, ptr );

    /* Create the input port */
    AVScreate_input_port( LEFT_IMAGE_INPUT,  "field 2D 1-vector byte", REQUIRED );
    AVScreate_input_port( RIGHT_IMAGE_INPUT, "field 2D 1-vector byte", REQUIRED );

    /* Create the output port */
    AVScreate_output_port( CM_FIELD_OUTPUT, "string" );

    /* Add parameter */
    sprintf( string, "%s%s", DV_PATH, DEFAULT_DATA_FILENAME );
    param0 = AVSadd_parameter( DATA_FILENAME_PARAM, "string", string, "", "" );
    AVSconnect_widget(param0, "typein");
    param1 = AVSadd_float_parameter( TRANSLATION_X_PARAM, 0.0,
                                     FLOAT_UNBOUND, FLOAT_UNBOUND );
    AVSconnect_widget(param1, "typein_real");
    param2 = AVSadd_float_parameter( TRANSLATION_Y_PARAM, 0.0,
                                     FLOAT_UNBOUND, FLOAT_UNBOUND );
```

```
    AVSconnect_widget(param2, "typein_real");
    param3 = AVSadd_float_parameter( TRANSLATION_Z_PARAM, 0.0,
                                     FLOAT_UNBOUND, FLOAT_UNBOUND );
    AVSconnect_widget(param3, "typein_real");
    param4 = AVSadd_float_parameter( POSITIONAL_PRECISION_PARAM, 0.0,
                                     0.0, 10.0 );
    AVSconnect_widget(param4, "typein_real");
    param5 = AVSadd_float_parameter( YAW_MIN_PARAM, 0.0,
                                     FLOAT_UNBOUND, FLOAT_UNBOUND );
    AVSconnect_widget(param5, "typein_real");
    param6 = AVSadd_float_parameter( YAW_MAX_PARAM, 0.0,
                                     FLOAT_UNBOUND, FLOAT_UNBOUND );
    AVSconnect_widget(param6, "typein_real");
    param7 = AVSadd_float_parameter( FOCAL_MIN_PARAM, DEFAULT_FOCAL,
                                     0.0, MAX_FOCAL );
    AVSconnect_widget(param7, "typein_real");
    param8 = AVSadd_float_parameter( FOCAL_MAX_PARAM, DEFAULT_FOCAL,
                                     0.0, MAX_FOCAL );
    AVSconnect_widget(param8, "typein_real");
    param9 = AVSadd_float_parameter( BASELINE_MIN_PARAM, DEFAULT_BASELINE,
                                     0.0, MAX_BASELINE );
    AVSconnect_widget(param9, "typein_real");
    param10 = AVSadd_float_parameter( BASELINE_MAX_PARAM, DEFAULT_BASELINE,
                                      0.0, MAX_BASELINE );
    AVSconnect_widget(param10, "typein_real");
    param11 = AVSadd_float_parameter( PITCH_MIN_PARAM, DEFAULT_PITCH,
                                      FLOAT_UNBOUND, FLOAT_UNBOUND );
    AVSconnect_widget(param11, "typein_real");
    param12 = AVSadd_float_parameter( PITCH_MAX_PARAM, DEFAULT_PITCH,
                                      FLOAT_UNBOUND, FLOAT_UNBOUND );
    AVSconnect_widget(param12, "typein_real");
    param13 = AVSadd_float_parameter( CORRELATION_THRESHOLD_PARAM,
                                      DEFAULT_CORRELATION_THRESHOLD,
                                      0.0, 1.0 );
    AVSconnect_widget(param13, "dial");
    param14 = AVSadd_parameter( FEATURE_EXTRACTOR_PARAM, "choice", DEFAULT_EXTRACTOR,
                                EXTRACTOR_CHOICES, DELIMITER );
    AVSconnect_widget(param14, "radio_buttons");
    param15 = AVSadd_float_parameter( EXTRACTOR_THRESHOLD_PARAM,
                                      DEFAULT_EXTRACTOR_THRESHOLD, 0.0, 1.0 );
    AVSconnect_widget(param15, "dial");
    param16 = AVSadd_parameter( VISUALIZATION_PARAM, "boolean", DEFAULT_VISUALIZATION,
                                /* ignored */0, /* ignored */0 );
    AVSconnect_widget(param16, "toggle");

    /* Set the (optional) initialization procedure */
    AVSset_init_proc( stereo_init );

    /* Set the compute procedure */
    AVSset_compute_proc( stereo_compute );

    /* Set the (optional) destroy procedure */
    AVSset_destroy_proc( stereo_destroy );
}

/*****************************************************************************/
status_t
write_to_dv( filename, field, field_length, bit )
     char *filename;
     CM_field_id_t field, bit;
     unsigned field_length;
{
  int output_fd;

  output_fd = CMFS_open( filename,
                         CMFS_O_CREAT | CMFS_O_WRONLY | CMFS_O_TRUNC, 0666 );
  if( output_fd < 0 ) {
    CMFS_perror( filename );
    return( CANNOT_OPEN_DATA );
  }

  printf("\nNow writing %d data objects and the Visual Field into DataVault file '%s'...
           CM_global_count_bit_always( bit )-1, filename ); fflush( stdout );
```

```
  if( CMFS_write_file_always( output_fd, field, field_length ) != field_length ) {
    CMFS_perror( filename );
    CMFS_close( output_fd );
    return( DATA_WRITING_ERROR );
  }
  if( CMFS_write_file_always( output_fd, bit, 1 ) != 1 ) {
    CMFS_perror( filename );
    CMFS_close( output_fd );
    return( DATA_WRITING_ERROR );
  }

  CMFS_close( output_fd );
  printf("done.\n\n");fflush(stdout);

  return( NO_ERROR );
}       /* of write_to_dv */

/*****************************************************************************/
status_t
read_fe_image( filename, field, length )
     char filename[ MISC_BUFFER_SIZE ];
     CM_field_id_t field;
     unsigned length;   /* length of field */
{
  int input_fd;
  char buffer[ IMAGE_SIZE*IMAGE_SIZE ];
  char image_buffer[ IMAGE_SIZE ][ IMAGE_SIZE ];
  int k;
  register i, j;

  /* open the image file on the front end */
  printf("opening file '%s'... ", filename);fflush(stdout);
  input_fd = open( filename, O_RDONLY );

  /* did it open ? */
  if( input_fd < 0 )
    return( CANNOT_OPEN_DATA );

  printf("done.\n");fflush(stdout);

  /*  read the image into memory */
  read( input_fd, buffer, IMAGE_SIZE*IMAGE_SIZE );

  printf("closing...");
  close( input_fd );
  printf(" done.\n");fflush(stdout);

  /* transpose 1D array to 2D array */
  k = 0;
  for( j = 0; j < IMAGE_SIZE; j++ )
    for( i = 0; i < IMAGE_SIZE; i++ ) {
      image_buffer[i][j] = buffer[k];
      k++;
    }

  /* transfer the front end data to the CM */
  put_square_array_on_CM( image_buffer, field, length, IMAGE_SIZE );

  return( NO_ERROR );
} /* of read_fe_image() */

/*****************************************************************************/
put_square_array_on_CM( fe_array, dest, length, size )
     char *fe_array;
     CM_field_id_t dest;
     unsigned int length;
     unsigned int size;
{
  int fe_offset[2];
  unsigned int cm_start[2];
  unsigned int cm_end[2];
  unsigned int cm_axis[2];
```

```
  unsigned int fe_dim[2];

  fe_offset[0] = 0;
  fe_offset[1] = 0;

  cm_start[0] = 0;
  cm_start[1] = 0;

  cm_end[0] = size;
  cm_end[1] = size;

  cm_axis[0] = 0;
  cm_axis[1] = 1;

  fe_dim[0] = size;
  fe_dim[1] = size;

  CM_u_write_to_news_array_1L( fe_array, fe_offset, cm_start,
                               cm_end, cm_axis, dest, length,
                               2, fe_dim, CM_8_bit );
}   /* End of put_square_array_on_CM() */

/*****************************************************************************/
/* read image pair from front end files */
status_t
read_image_pair( image1, image2, image_pair )
     char *image1, *image2;
     CM_field_id_t image_pair;
{
  CM_field_id_t send_address, square_coord[2], square_image, temp_u_pair;

  CM_set_vp_set( stereo_pair_grid );
  temp_u_pair = CM_allocate_stack_field( BITS_PER_PIXEL );

  CM_set_vp_set( square_grid );
  square_image = CM_allocate_stack_field( BITS_PER_PIXEL );
  send_address = CM_allocate_stack_field( CM_geometry_send_address_length(
                                          stereo_pair_geom ) );
  square_coord[0] = CM_allocate_stack_field( 2*IMAGE_SIZE_LEN );
  square_coord[1] = CM_SUBFIELD( square_coord[0], IMAGE_SIZE_LEN );

  CM_set_context();
  CM_my_news_coordinate_1L( square_coord[0], 0, IMAGE_SIZE_LEN );
  CM_my_news_coordinate_1L( square_coord[1], 1, IMAGE_SIZE_LEN );

  if( read_fe_image( image1, square_image, BITS_PER_PIXEL ) == CANNOT_OPEN_DATA )
    return( CANNOT_OPEN_LEFT_DATA );

  CM_make_news_coordinate_1L( stereo_pair_geom, send_address, 0,
                              square_coord[0], IMAGE_SIZE_LEN );
  CM_deposit_news_coordinate_1L( stereo_pair_geom, send_address, 1,
                                 square_coord[1], IMAGE_SIZE_LEN );
  CM_deposit_news_constant_1L( stereo_pair_geom, send_address, 2, /* depth */ 0 );

  CM_send_1L( temp_u_pair, send_address, square_image, BITS_PER_PIXEL, CM_no_field );

  if( read_fe_image( image2, square_image, BITS_PER_PIXEL ) == CANNOT_OPEN_DATA )
    return( CANNOT_OPEN_RIGHT_DATA );

  CM_make_news_coordinate_1L( stereo_pair_geom, send_address, 0,
                              square_coord[0], IMAGE_SIZE_LEN );
  CM_deposit_news_coordinate_1L( stereo_pair_geom, send_address, 1,
                                 square_coord[1], IMAGE_SIZE_LEN );
  CM_deposit_news_constant_1L( stereo_pair_geom, send_address, 2, /* depth */ 1 );

  CM_send_1L( temp_u_pair, send_address, square_image, BITS_PER_PIXEL, CM_no_field );

  CM_set_vp_set( stereo_pair_grid );
  CM_set_context();
  CM_f_u_float_2_2L( image_pair, temp_u_pair, BITS_PER_PIXEL, FLENS );

  CM_deallocate_stack_through( square_image );
```

```
  /* get rid of the bottom lines */
  CM_set_context();
  CM_u_gt_constant_1L( coord[ 1 ], SCAN_LINES_NB, IMAGE_SIZE_LEN );
  CM_logand_context_with_test();
  CM_f_move_zero_1L( image_pair, FLENS );

  CM_set_context();
  return( NO_ERROR );
}       /* of read_image_pair */

/*****************************************************************************/
status_t
load_avs_image( avs_field, cm_field )
     AVSfield_char *avs_field;
     CM_field_id_t cm_field;
{
  char image_buffer[ IMAGE_SIZE ][ IMAGE_SIZE ];
  int i,j;

  for( j = 0; j < IMAGE_SIZE; j++ )
    for( i = 0; i < IMAGE_SIZE; i++ ) {
      image_buffer[i][j] = I2D( avs_field, i, j );
    }

  /* transfer the front end data to the CM */
  put_square_array_on_CM( image_buffer, cm_field, BITS_PER_PIXEL, IMAGE_SIZE );

  return( NO_ERROR );
} /* of load_avs_image() */

/*****************************************************************************/
/* read image pair from front end files */
status_t
load_avs_image_pair( image1, image2, image_pair )
     AVSfield_char *image1, *image2;
     CM_field_id_t image_pair;
{
  CM_field_id_t send_address, square_coord[2], square_image, temp_u_pair;

  CM_set_vp_set( stereo_pair_grid );
  temp_u_pair = CM_allocate_stack_field( BITS_PER_PIXEL );

  CM_set_vp_set( square_grid );
  square_image = CM_allocate_stack_field( BITS_PER_PIXEL );
  send_address = CM_allocate_stack_field( CM_geometry_send_address_length(
                                          stereo_pair_geom ) );
  square_coord[0] = CM_allocate_stack_field( 2*IMAGE_SIZE_LEN );
  square_coord[1] = CM_SUBFIELD( square_coord[0], IMAGE_SIZE_LEN );

  CM_set_context();
  CM_my_news_coordinate_1L( square_coord[0], 0, IMAGE_SIZE_LEN );
  CM_my_news_coordinate_1L( square_coord[1], 1, IMAGE_SIZE_LEN );

  load_avs_image( image1, square_image );

  CM_make_news_coordinate_1L( stereo_pair_geom, send_address, 0,
                              square_coord[0], IMAGE_SIZE_LEN );
  CM_deposit_news_coordinate_1L( stereo_pair_geom, send_address, 1,
                                 square_coord[1], IMAGE_SIZE_LEN );
  CM_deposit_news_constant_1L( stereo_pair_geom, send_address, 2, /* depth */ 0 );

  CM_send_1L( temp_u_pair, send_address, square_image, BITS_PER_PIXEL, CM_no_field );

  load_avs_image( image2, square_image );

  CM_make_news_coordinate_1L( stereo_pair_geom, send_address, 0,
                              square_coord[0], IMAGE_SIZE_LEN );
  CM_deposit_news_coordinate_1L( stereo_pair_geom, send_address, 1,
                                 square_coord[1], IMAGE_SIZE_LEN );
  CM_deposit_news_constant_1L( stereo_pair_geom, send_address, 2, /* depth */ 1 );

  CM_send_1L( temp_u_pair, send_address, square_image, BITS_PER_PIXEL, CM_no_field );
```

```
  CM_set_vp_set( stereo_pair_grid );
  CM_set_context();
  CM_f_u_float_2_2L( image_pair, temp_u_pair, BITS_PER_PIXEL, FLENS );

  CM_deallocate_stack_through( square_image );

  CM_set_context();
  return( NO_ERROR );
}       /* of load_avs_image_pair() */

/*****************************************************************************/
/* reads BITS_PER_PIXEL from a DataVault file and converts it into float
   representation (stereo_pair) */
read_dv_file( filename )
     char filename[ MISC_BUFFER_SIZE ];
{
  int input_fd;
  int ok;
  CM_field_id_t temp_u_field;   /* temporary unsigned integer field */

  CM_set_vp_set( stereo_pair_grid );
  CM_set_context();

  temp_u_field = CM_allocate_stack_field( BITS_PER_PIXEL );

  /* open the dv file */
  printf("opening DataVault file '%s'... ", filename);fflush(stdout);
  input_fd = CMFS_open( filename, CMFS_O_RDONLY );

  /* did it open ? */
  if( input_fd < 0 ) {
    printf( "Cannot open '%s'...\n", filename );fflush(stdout);
    exit( EXIT_ERROR );
  }
  printf("done\n");fflush(stdout);

  ok = CMFS_read_file_always( input_fd, temp_u_field, BITS_PER_PIXEL );

  CMFS_close( input_fd );

  CM_f_u_float_2_2L( stereo_pair, temp_u_field, BITS_PER_PIXEL, FLENS );

  CM_deallocate_stack_through( temp_u_field );

  if ( ok < 0 ) {
    CMFS_perror( filename );
    exit( EXIT_ERROR );
  }
  else printf("file '%s' read successfully...\n", filename );fflush(stdout);

}       /* of read_dv_file */

/*****************************************************************************/
initialize_display()
{
  if(VISUALIZATION) {
    CMSR_select_display_menu( BITS_PER_PIXEL, 2*IMAGE_SIZE, IMAGE_SIZE );
    CMSR_display_set_color_map("greyscale");
  }
  DISPLAY_INITIALIZED = YES;
}       /* of initialize_display() */

/*****************************************************************************/
write_to_display( image )
     CM_field_id_t image;
{
  if(VISUALIZATION)
    CMSR_write_to_display( image );

}       /* of write_to_display() */

/*****************************************************************************/
clear_display()
```

```
{
  if(VISUALIZATION)
    CMSR_deallocate_display( CMSR_selected_display() );

}       /* of clear_display() */

/*****************************************************************************/
convert_3D_to_2D( images_3D, image_2D )
     CM_field_id_t images_3D, image_2D;
{
  CM_field_id_t coord_temp, image_2D_temp;

  CM_set_vp_set( image_grid );
  image_2D_temp = CM_allocate_stack_field( BITS_PER_PIXEL );

  CM_set_vp_set( stereo_pair_grid );
  coord_temp = CM_allocate_stack_field( IMAGE_SIZE_LEN+1 );

  CM_set_context();
  CM_u_move_2L( coord_temp, coord[0], IMAGE_SIZE_LEN+1, IMAGE_SIZE_LEN );

  CM_load_context( coord[2] );
  CM_invert_context();
  CM_make_news_coordinate_1L( image_geom, temp, 1, coord[1],
                              IMAGE_SIZE_LEN );
  CM_deposit_news_coordinate_1L( image_geom, temp, 0, coord_temp,
                                 IMAGE_SIZE_LEN+1 );
  CM_send_1L( image_2D_temp, temp, images_3D, BITS_PER_PIXEL, image_bit );

  CM_load_context( coord[2] );
  CM_u_add_constant_2_1L( coord_temp, IMAGE_SIZE, IMAGE_SIZE_LEN+1 );
  CM_make_news_coordinate_1L( image_geom, temp, 1, coord[1],
                              IMAGE_SIZE_LEN );
  CM_deposit_news_coordinate_1L( image_geom, temp, 0, coord_temp,
                                 IMAGE_SIZE_LEN+1 );
  CM_send_1L( image_2D_temp, temp, images_3D, BITS_PER_PIXEL, CM_no_field );

  CM_set_vp_set( image_grid );
  CM_set_context();
  CM_u_max_2_1L( image_2D, image_2D_temp, BITS_PER_PIXEL );

  CM_deallocate_stack_through( image_2D_temp );

}       /* of convert_3D_to_2D() */

/*****************************************************************************/
/* marks the features so that they can be easily visualized */
show_features( bit, image, color_field )
     CM_field_id_t bit, image, color_field;
{
  CM_field_id_t feature_images;

  CM_set_vp_set( stereo_pair_grid );

  feature_images = CM_allocate_stack_field( BITS_PER_PIXEL );
  CM_u_move_zero_always_1L( feature_images, BITS_PER_PIXEL );

  CM_load_context( bit );
  CM_u_move_1L( feature_images, color_field, BITS_PER_PIXEL );

  /* draw small cross around features */
  CM_u_move_1L( temp, feature_images, BITS_PER_PIXEL );
  CM_send_to_news_1L( feature_images, temp, 0, CM_upward, BITS_PER_PIXEL );
  CM_send_to_news_1L( feature_images, temp, 0, CM_downward, BITS_PER_PIXEL );
  CM_send_to_news_1L( feature_images, temp, 1, CM_upward, BITS_PER_PIXEL );
  CM_send_to_news_1L( feature_images, temp, 1, CM_downward, BITS_PER_PIXEL );

  convert_3D_to_2D( feature_images, image );

  CM_deallocate_stack_through( feature_images );

}       /* of show_features() */
```

```
/*****************************************************************************/
/* Median filtering */
median_filter( source, dest )
     CM_field_id_t source, dest;
{
  int i,j;
  CM_field_id_t values, value[ MEDIAN_SIZE_SQUARED ];

  values = CM_allocate_stack_field( MEDIAN_SIZE_SQUARED*FLEN );
  for( i = 0; i < MEDIAN_SIZE_SQUARED; i++ )
      value[i] = CM_SUBFIELD( values, i*FLEN );

  CM_set_context();
  CM_get_from_power_two_always_1L( value[1], source, /* axis */ 0,
                                   MEDIAN_SIZE_LEN-2, CM_upward, FLEN );
  CM_get_from_power_two_always_1L( value[0], value[1], /* axis */ 1,
                                   MEDIAN_SIZE_LEN-2, CM_upward, FLEN );

  for( i = 0; i < MEDIAN_SIZE-1; i++ ) {

    for( j = 0; j < MEDIAN_SIZE-1; j++ )
      CM_send_to_news_always_1L( value[ i*MEDIAN_SIZE + j + 1 ],
                                 value[ i*MEDIAN_SIZE + j ], 0,
                                 CM_upward, FLEN );

    CM_send_to_news_always_1L( value[ (i+1)*MEDIAN_SIZE ],
                               value[ i*MEDIAN_SIZE ], 1,
                               CM_upward, FLEN );
  }
  /* one more time for i = MEDIAN_SIZE-1 */
  for( j = 0; j < MEDIAN_SIZE-1; j++ )
    CM_send_to_news_always_1L( value[ i*MEDIAN_SIZE + j + 1 ],
                               value[ i*MEDIAN_SIZE + j ], 0,
                               CM_upward, FLEN );

  /* bubble sort on values ... */
  for( j = 0; j < MEDIAN_SIZE_SQUARED-1; j++ )
    for( i = 0; i < MEDIAN_SIZE_SQUARED-1; i++ ) {
      CM_set_context();
      CM_f_gt_1L( value[ i ], value[ i+1 ], FLENS );
      CM_logand_context_with_test();
      CM_swap_2_1L( value[ i ], value[ i+1 ], FLEN );
    }
  /* now take the median value in values and assign it to dest */
  CM_f_move_always_1L( dest, value[ (MEDIAN_SIZE_SQUARED-1)/2 ], FLENS );

  CM_deallocate_stack_through( values );

}       /* of median_filter() */

/*****************************************************************************/
/* feature extraction using Moravec interest operator */
sobel_features( image, bit, threshold )
     CM_field_id_t image;
     CM_field_id_t bit;
     float threshold;
{
  int i, j;
  CM_field_id_t filtered_image;
  CM_field_id_t values, value[ 3 ][ 3 ];
  CM_field_id_t gradient;
  static float sobel_x[ 3 ][ 3 ] = { -1.0,  0.0,  1.0,
                                     -2.0,  0.0,  2.0,
                                     -1.0,  0.0,  1.0   };
  static float sobel_y[ 3 ][ 3 ] = { -1.0, -2.0, -1.0,
                                      0.0,  0.0,  0.0,
                                      1.0,  2.0,  1.0   };

  filtered_image = CM_allocate_stack_field( FLEN );

  printf("\nFeature extraction using Sobel (gradient) operator...");
  printf("\n  median filtering (%ux%u window)... ", MEDIAN_SIZE, MEDIAN_SIZE);
  fflush(stdout);
```

```
  median_filter( image, filtered_image );
  printf("done.\n");fflush(stdout);

  values = CM_allocate_stack_field( 9*FLEN );
  gradient = CM_allocate_stack_field( FLEN );
  for( i = 0; i < 3; i++ )
    for( j = 0; j < 3; j++ )
      value[i][j] = CM_SUBFIELD( values, (j+i*3)*FLEN );

  CM_set_context();

  CM_send_to_news_always_1L( value[1][0], filtered_image, 1, CM_upward, FLEN );
  CM_send_to_news_always_1L( value[1][2], filtered_image, 1, CM_downward, FLEN );
  CM_send_to_news_always_1L( value[0][1], filtered_image, 0, CM_upward, FLEN );
  CM_send_to_news_always_1L( value[2][1], filtered_image, 0, CM_downward, FLEN );
  CM_send_to_news_always_1L( value[2][0], value[1][0], 0, CM_downward, FLEN );
  CM_send_to_news_always_1L( value[0][0], value[1][0], 0, CM_upward, FLEN );
  CM_send_to_news_always_1L( value[2][2], value[1][2], 0, CM_downward, FLEN );
  CM_send_to_news_always_1L( value[0][2], value[1][2], 0, CM_upward, FLEN );

  for( i = 0; i < 3; i++ )
    for( j = 0; j < 3; j++ )
      CM_f_multiply_const_always_2_1L( value[i][j], sobel_y[i][j], FLENS );

  CM_f_move_zero_always_1L( gradient, FLENS );
  for( i = 0; i < 3; i++ )
    for( j = 0; j < 3; j++ )
      CM_f_add_always_2_1L( gradient, value[i][j], FLENS );

  CM_f_abs_1_1L( gradient, FLENS );
  CM_f_divide_const_always_2_1L( gradient,
                                 4.0*(power( 2, BITS_PER_PIXEL ) - 1), FLENS );

  /* discard values which are not locally maximum */
  CM_send_to_news_always_1L( value[0][1], gradient, 0, CM_upward, FLEN );
  CM_send_to_news_always_1L( value[2][1], gradient, 0, CM_downward, FLEN );

  CM_f_move_zero_always_1L( value[1][1], FLENS );
  for( i = 0; i < 3; i++ ) {
    CM_f_gt_1L( gradient, value[i][1], FLENS );
    CM_logand_context_with_test();
  }

  printf("  feature extraction: %g <= gradient values <= %g\n",
          CM_global_f_min_1L( gradient, FLENS ),
          CM_global_f_max_1L( gradient, FLENS ) );fflush(stdout);

  printf("  number of features before thresholding: %u\n",
          CM_global_count_context() );fflush(stdout);
  CM_f_ge_constant_1L( gradient, threshold, FLENS );
  CM_logand_context_with_test();
  CM_store_context( bit );

  printf("  number of features with horizontal gradient >= %g : %u\n",
          threshold, CM_global_count_context() );fflush(stdout);

  CM_deallocate_stack_through( values );

}       /* of sobel_features() */

/*****************************************************************************/
/* feature extraction using Moravec interest operator */
moravec_features( image, bit, threshold )
     CM_field_id_t image;
     CM_field_id_t bit;
     float threshold;
{
  int i, j;
  CM_field_id_t filtered_image;
  CM_field_id_t values, value[ 3 ][ 3 ];
  CM_field_id_t interest;
```

```
filtered_image = CM_allocate_stack_field( FLEN );

printf("\nFeature extraction using Moravec interest operator...");
printf("\n  median filtering (%ux%u window)... ", MEDIAN_SIZE, MEDIAN_SIZE);
fflush(stdout);

median_filter( image, filtered_image );
printf("done.\n");fflush(stdout);

values = CM_allocate_stack_field( 9*FLEN );
interest = CM_allocate_stack_field( FLEN );
for( i = 0; i < 3; i++ )
  for( j = 0; j < 3; j++ )
    value[i][j] = CM_SUBFIELD( values, (j+i*3)*FLEN );

CM_set_context();

CM_send_to_news_always_1L( value[1][0], filtered_image, 1, CM_upward, FLEN );
CM_send_to_news_always_1L( value[1][2], filtered_image, 1, CM_downward, FLEN );
CM_send_to_news_always_1L( value[0][1], filtered_image, 0, CM_upward, FLEN );
CM_send_to_news_always_1L( value[2][1], filtered_image, 0, CM_downward, FLEN );
CM_send_to_news_always_1L( value[2][0], value[1][0], 0, CM_downward, FLEN );
CM_send_to_news_always_1L( value[0][0], value[1][0], 0, CM_upward, FLEN );
CM_send_to_news_always_1L( value[2][2], value[1][2], 0, CM_downward, FLEN );
CM_send_to_news_always_1L( value[0][2], value[1][2], 0, CM_upward, FLEN );

for( i = 0; i < 3; i++ )
  for( j = 0; j < 3; j++ ) {
    CM_f_subtract_always_2_1L( value[i][j], filtered_image, FLENS );
    CM_f_multiply_always_2_1L( value[i][j], value[i][j], FLENS );
  }

CM_f_add_always_2_1L( value[1][0], value[1][2], FLENS );
CM_f_add_always_2_1L( value[0][0], value[2][2], FLENS );
CM_f_add_always_2_1L( value[2][1], value[0][1], FLENS );
CM_f_add_always_2_1L( value[0][2], value[2][0], FLENS );

CM_f_min_3_1L( interest, value[1][0], value[0][0], FLENS );
CM_f_min_2_1L( interest, value[2][1], FLENS );
CM_f_min_2_1L( interest, value[0][2], FLENS );

CM_f_divide_const_always_2_1L( interest, 2.0*(power(2, 2*BITS_PER_PIXEL)-1), FLENS );

CM_send_to_news_always_1L( value[1][0], interest, 1, CM_upward, FLEN );
CM_send_to_news_always_1L( value[1][2], interest, 1, CM_downward, FLEN );
CM_send_to_news_always_1L( value[0][1], interest, 0, CM_upward, FLEN );
CM_send_to_news_always_1L( value[2][1], interest, 0, CM_downward, FLEN );
CM_send_to_news_always_1L( value[2][0], value[1][0], 0, CM_downward, FLEN );
CM_send_to_news_always_1L( value[0][0], value[1][0], 0, CM_upward, FLEN );
CM_send_to_news_always_1L( value[2][2], value[1][2], 0, CM_downward, FLEN );
CM_send_to_news_always_1L( value[0][2], value[1][2], 0, CM_upward, FLEN );

CM_f_move_zero_always_1L( value[1][1], FLENS );
for( i = 0; i < 3; i++ )
  for( j = 0; j < 3; j++ ) {
    CM_f_gt_1L( interest, value[i][j], FLENS );
    CM_logand_context_with_test();
  }

printf("  feature extraction: %g <= interest values <= %g\n",
        CM_global_f_min_1L( interest, FLENS ),
        CM_global_f_max_1L( interest, FLENS ) );fflush(stdout);

printf("  number of features before thresholding: %u\n",
        CM_global_count_context() );fflush(stdout);
CM_f_ge_constant_1L( interest, threshold, FLENS );
CM_logand_context_with_test();
CM_store_context( bit );

printf("  number of features with interest >= %g : %u\n",
        threshold, CM_global_count_context() );fflush(stdout);

CM_deallocate_stack_through( values );
```

```
}       /* of moravec_features() */

/*****************************************************************************/
extract_features( image, bit, feature_extractor, threshold )
     CM_field_id_t image;
     CM_field_id_t bit;
     feature_extractor_t feature_extractor;
     float threshold;
{
  unsigned lower_boundary, upper_boundary;

  CM_set_vp_set( stereo_pair_grid );

  switch( feature_extractor ) {
    case MORAVEC: moravec_features( image, bit, threshold ); break;
    case SOBEL:   sobel_features( image, bit, threshold ); break;
    case REGULAR: break;
  }

  /* get rid of the features that are too close to the boundaries */
  lower_boundary = (CORRELATION_WINDOW_SIZE-1)/2 + ALIGNMENT_MAX + 1;
  upper_boundary = IMAGE_SIZE - lower_boundary -1;

  CM_load_context( bit );
  CM_u_ge_constant_1L( coord[0], lower_boundary, IMAGE_SIZE_LEN );
  CM_logand_context_with_test();
  CM_u_le_constant_1L( coord[0], upper_boundary, IMAGE_SIZE_LEN );
  CM_logand_context_with_test();
  CM_u_ge_constant_1L( coord[1], lower_boundary, IMAGE_SIZE_LEN );
  CM_logand_context_with_test();
  upper_boundary = SCAN_LINES_NB - lower_boundary -1;
  CM_u_le_constant_1L( coord[1], upper_boundary, IMAGE_SIZE_LEN );
  CM_logand_context_with_test();
  CM_store_context( bit );

  /* get rid of the features that are too close to the boundaries */
  lower_boundary = (CORRELATION_WINDOW_SIZE-1)/2 + DISPARITY_MAX + 1;
  upper_boundary = IMAGE_SIZE - lower_boundary -1;

  CM_load_context( coord[2] );
  CM_invert_context();
  CM_logand_context( bit );
  CM_u_lt_constant_1L( coord[0], lower_boundary, IMAGE_SIZE_LEN );
  CM_logand_context_with_test();
  CM_clear_bit( bit );

  CM_load_context( coord[2] );
  CM_logand_context( bit );
  CM_u_gt_constant_1L( coord[0], upper_boundary, IMAGE_SIZE_LEN );
  CM_logand_context_with_test();
  CM_clear_bit( bit );

  printf("  number of features far enough from boundaries: %u\n",
          CM_global_count_bit_always( bit ) );fflush(stdout);

}       /* of extract_features() */

/*****************************************************************************/
add_window_values( image, value )
     CM_field_id_t image, value;
{
  CM_set_context();

  CM_scan_with_f_add_1L( temp_float3, image, /* axis */ 0, FLENS, CM_upward,
                         CM_exclusive, CM_none, CM_no_field );
  CM_send_to_news_always_1L( temp_float2, temp_float3,
                             /* axis */ 0, CM_downward, FLEN );
  CM_get_from_power_two_always_1L( temp_float1, temp_float2, /* axis */ 0,
                                   CORRELATION_WINDOW_LEN-2, CM_upward, FLEN );
  CM_get_from_power_two_always_1L( temp_float2, temp_float3, /* axis */ 0,
                                   CORRELATION_WINDOW_LEN-2, CM_downward, FLEN );
  CM_f_subtract_always_2_1L( temp_float1, temp_float2, FLENS );
```

```
  CM_scan_with_f_add_1L( temp_float3, temp_float1, /* axis */ 1, FLENS, CM_upward,
                         CM_exclusive, CM_none, CM_no_field );
  CM_send_to_news_always_1L( temp_float2, temp_float3,
                             /* axis */ 1, CM_downward, FLEN );
  CM_get_from_power_two_always_1L( value, temp_float2, /* axis */ 1,
                                   CORRELATION_WINDOW_LEN-2, CM_upward, FLEN );
  CM_get_from_power_two_always_1L( temp_float2, temp_float3, /* axis */ 1,
                                   CORRELATION_WINDOW_LEN-2, CM_downward, FLEN );
  CM_f_subtract_always_2_1L( value, temp_float2, FLENS );

}      /* of add_window_values() */

/*****************************************************************************/
normalize( function, mean, norm )
     CM_field_id_t function, mean, norm;
{
  CM_field_id_t product_function;

  CM_set_vp_set( stereo_pair_grid );

  product_function = CM_allocate_stack_field( FLEN );

  CM_set_context();
  add_window_values( function, mean );
  CM_f_multiply_const_always_2_1L( mean, INV_CORRELATION_SQUARED_SIZE, FLENS );

  CM_f_move_always_1L( product_function, function, FLENS );
  CM_f_multiply_always_2_1L( product_function, function, FLENS );
  add_window_values( product_function, norm );
  CM_f_multiply_const_always_2_1L( norm, INV_CORRELATION_SQUARED_SIZE, FLENS );
  CM_f_mult_subf_1L( norm, mean, mean, norm, FLENS );
  CM_f_sqrt_1_1L( norm, FLENS );

  CM_deallocate_stack_through( product_function );

}       /* of normalize()

/*****************************************************************************/
match_features()
{
  int i,j;
  CM_field_id_t x_shift_dual_stereo_pair_info;
  CM_field_id_t x_shift_dual_stereo_pair_mean;
  CM_field_id_t x_shift_dual_stereo_pair_norm;
  CM_field_id_t x_shift_dual_stereo_pair;
  CM_field_id_t y_shift_dual_stereo_pair_info;
  CM_field_id_t current_correlation;
  CM_field_id_t info_temp;
  CM_field_id_t product_stereo_pair;

  CM_set_vp_set( stereo_pair_grid );

  current_correlation = CM_allocate_stack_field( FLEN );
  info_temp = CM_allocate_stack_field( 3*FLEN );
  x_shift_dual_stereo_pair_info = CM_allocate_stack_field( 3*FLEN );
  x_shift_dual_stereo_pair = CM_SUBFIELD( x_shift_dual_stereo_pair_info, 0 );
  x_shift_dual_stereo_pair_mean = CM_SUBFIELD( x_shift_dual_stereo_pair_info, FLEN );
  x_shift_dual_stereo_pair_norm = CM_SUBFIELD( x_shift_dual_stereo_pair_info, 2*FLEN );
  y_shift_dual_stereo_pair_info = CM_allocate_stack_field( 3*FLEN );
  product_stereo_pair = CM_allocate_stack_field( FLEN );

  printf("\nNow matching the features (0 <= disparity <= %d)... ",
          DISPARITY_MAX ); fflush(stdout);

  CM_set_context();

  CM_f_move_zero_always_1L( correlation, FLENS );

  CM_get_from_power_two_always_1L( info_temp, dual_stereo_pair_info, /* axis */ 1,
                                   ALIGNMENT_LEN-2, CM_downward, 3*FLEN );
  CM_u_move_always_1L( y_shift_dual_stereo_pair_info, info_temp, 3*FLEN );
```

```
  for( j = -ALIGNMENT_MAX; j <= ALIGNMENT_MAX; j++ ) {

    CM_u_move_always_1L( x_shift_dual_stereo_pair_info,
                         y_shift_dual_stereo_pair_info, 3*FLEN );

    for( i = 0; i >= -DISPARITY_MAX; i-- ) {

      CM_f_multiply_always_3_1L( product_stereo_pair, stereo_pair,
                                 x_shift_dual_stereo_pair, FLENS );
      add_window_values( product_stereo_pair, current_correlation );
      CM_f_multiply_const_always_2_1L( current_correlation,
                                       INV_CORRELATION_SQUARED_SIZE, FLENS );
      CM_set_context();
      CM_f_mult_subf_1L( current_correlation, stereo_pair_mean,
                         x_shift_dual_stereo_pair_mean, current_correlation,
                         FLENS );
      CM_f_multiply_always_3_1L( temp_float1, stereo_pair_norm,
                                 x_shift_dual_stereo_pair_norm, FLENS );

      CM_f_ne_zero_1L( temp_float1, FLENS );
      CM_logand_context_with_test();
      CM_f_divide_2_1L( current_correlation, temp_float1, FLENS );

      CM_load_context( coord[2] );
      CM_invert_context();
      CM_f_gt_1L( current_correlation, correlation, FLENS );
      CM_logand_context_with_test();
      CM_f_move_1L( correlation, current_correlation, FLENS );
      /* store current shift */
      CM_s_move_constant_1L( disparity, i, DISPARITY_LEN );
      CM_s_move_constant_1L( alignment, j, ALIGNMENT_LEN );
      CM_send_to_news_always_1L( info_temp, x_shift_dual_stereo_pair_info,
                                 /* axis */ 0, CM_upward, 3*FLEN );
      CM_u_move_always_1L( x_shift_dual_stereo_pair_info, info_temp, 3*FLEN );

    }

    CM_u_move_always_1L( x_shift_dual_stereo_pair_info,
                         y_shift_dual_stereo_pair_info, 3*FLEN );

    for( i = 0; i <= DISPARITY_MAX; i++ ) {

      CM_f_multiply_always_3_1L( product_stereo_pair, stereo_pair,
                                 x_shift_dual_stereo_pair, FLENS );
      add_window_values( product_stereo_pair, current_correlation );
      CM_f_multiply_const_always_2_1L( current_correlation,
                                       INV_CORRELATION_SQUARED_SIZE, FLENS );
      CM_set_context();
      CM_f_mult_subf_1L( current_correlation, stereo_pair_mean,
                         x_shift_dual_stereo_pair_mean, current_correlation,
                         FLENS );
      CM_f_multiply_always_3_1L( temp_float1, stereo_pair_norm,
                                 x_shift_dual_stereo_pair_norm, FLENS );

      CM_f_ne_zero_1L( temp_float1, FLENS );
      CM_logand_context_with_test();
      CM_f_divide_2_1L( current_correlation, temp_float1, FLENS );

      CM_load_context( coord[2] );
      CM_f_gt_1L( current_correlation, correlation, FLENS );
      CM_logand_context_with_test();
      CM_f_move_1L( correlation, current_correlation, FLENS );
      /* store current shift */
      CM_s_move_constant_1L( disparity, i, DISPARITY_LEN );
      CM_s_move_constant_1L( alignment, j, ALIGNMENT_LEN );

      CM_send_to_news_always_1L( info_temp, x_shift_dual_stereo_pair_info,
                                 /* axis */ 0, CM_downward, 3*FLEN );
      CM_u_move_always_1L( x_shift_dual_stereo_pair_info, info_temp, 3*FLEN );

    }

    CM_send_to_news_always_1L( info_temp, y_shift_dual_stereo_pair_info,
```

```
                                /* axis */ 1, CM_downward, 3*FLEN );
    CM_u_move_always_1L( y_shift_dual_stereo_pair_info, info_temp, 3*FLEN );

  }

  CM_deallocate_stack_through( current_correlation );

  printf("done.\n");fflush(stdout);

}       /* of match_features() */
/*****************************************************************************/
threshold_correlation( threshold )
     float threshold;
{
  CM_load_context( feature_bit );
  CM_f_ge_constant_1L( correlation, threshold, FLENS );
  CM_logand_context_with_test();
  CM_store_context( feature_bit );

  printf("\nTotal number of features with a correlation value >= %g : %u\n",
          threshold, CM_global_count_context() );fflush(stdout);

}       /* of threshold_correlation */

/*****************************************************************************/
merge()
{
  CM_field_id_t address;
  CM_field_id_t temp_coord[2];
  CM_field_id_t temp_bit1, temp_bit2;
  CM_field_id_t my_send_address, start_send_address;
  CM_field_id_t dual_disparity;
  CM_field_id_t dual_alignment;
  unsigned ADDRESS_LEN;

  ADDRESS_LEN = CM_geometry_send_address_length( stereo_pair_geom );

  CM_set_vp_set( stereo_pair_grid );

  temp_bit1 = CM_allocate_stack_field( 1 );
  temp_bit2 = CM_allocate_stack_field( 1 );
  temp_coord[0] = CM_allocate_stack_field( IMAGE_SIZE_LEN+1 );
  temp_coord[1] = CM_allocate_stack_field( IMAGE_SIZE_LEN+1 );
  address = CM_allocate_stack_field( ADDRESS_LEN );
  my_send_address = CM_allocate_stack_field( ADDRESS_LEN );
  start_send_address = CM_allocate_stack_field( ADDRESS_LEN );
  dual_disparity = CM_allocate_stack_field( DISPARITY_LEN );
  dual_alignment = CM_allocate_stack_field( ALIGNMENT_LEN );

  printf("\nNow checking for consistency and merging the 2 disparity maps...\n");
  fflush(stdout);

  CM_set_context();
  CM_my_send_address( my_send_address );
  CM_f_move_zero_always_1L( temp_float1, FLENS );

  CM_load_context( coord[2] );
  CM_invert_context();
  CM_store_context( temp_bit1 );

  CM_load_context( feature_bit );

  /* make room for sign bit */
  CM_u_move_2L( temp_coord[0], coord[0], IMAGE_SIZE_LEN+1, IMAGE_SIZE_LEN );
  CM_u_move_2L( temp_coord[1], coord[1], IMAGE_SIZE_LEN+1, IMAGE_SIZE_LEN );

  CM_s_add_3_3L( temp_coord[0], temp_coord[0], disparity,
                 IMAGE_SIZE_LEN+1, IMAGE_SIZE_LEN+1, DISPARITY_LEN );
  CM_s_add_3_3L( temp_coord[1], temp_coord[1], alignment,
                 IMAGE_SIZE_LEN+1, IMAGE_SIZE_LEN+1, ALIGNMENT_LEN );

  CM_make_news_coordinate_1L( stereo_pair_geom, address, /* axis */ 0,
                              temp_coord[0], IMAGE_SIZE_LEN );
```

```
CM_deposit_news_coordinate_1L( stereo_pair_geom, address, /* axis */ 1,
                               temp_coord[1], IMAGE_SIZE_LEN );
CM_deposit_news_coordinate_1L( stereo_pair_geom, address, /* axis */ 2,
                               temp_bit1, 1 );

CM_my_send_address( my_send_address );

/* keep only the vector with maximum correlation among multiple vectors
   of each image */
CM_send_with_f_max_1L( temp_float1, address, correlation,
                       FLENS, CM_no_field );
CM_get_1L( temp_float2, address, temp_float1, FLEN );
CM_f_eq_1L( temp_float2, correlation, FLENS );
CM_logand_context_with_test();
CM_store_context( feature_bit );
printf("  number of features after multiple vector check: %u\n",
        CM_global_count_context() );fflush(stdout);

CM_clear_bit_always( temp_bit1 );
CM_clear_bit_always( temp_bit2 );
CM_send_1L( start_send_address, address, my_send_address, ADDRESS_LEN,
            temp_bit1 );
CM_send_1L( dual_disparity, address, disparity, DISPARITY_LEN,
            CM_no_field );
CM_send_1L( dual_alignment, address, alignment, ALIGNMENT_LEN,
            CM_no_field );

/* select all 'consistent double vectors' */
CM_load_context( feature_bit );
CM_logand_context( temp_bit1 );
CM_u_eq_1L( start_send_address, address, ADDRESS_LEN );
CM_logand_context_with_test();
CM_store_context( temp_bit2 );        /* temp_bit2 marks the consistent
                                         double matches */
printf("  number of consistent double matches : %u\n",
        CM_global_count_context() );fflush(stdout);
/* arbitrarily clear the vectors of image 1, since they already exist in
   image 0 */
CM_logand_context( coord[2] );
CM_clear_bit( feature_bit );
CM_load_context( coord[2] );
CM_invert_context();
CM_logand_context( temp_bit2 );
CM_clear_bit( temp_bit1 );

/* select all other 'double vectors' */
CM_load_context( feature_bit );
CM_logand_context( temp_bit1 );
printf("  number of inconsistent double matches : %u\n",
        CM_global_count_context() );fflush(stdout);

CM_f_lt_1L( temp_float1, correlation, FLENS );
CM_logand_context_with_test();
CM_clear_bit( temp_bit1 );
CM_send_1L( feature_bit, start_send_address, temp_bit1, 1, CM_no_field );

CM_load_context( feature_bit );
CM_logand_context( temp_bit1 );
CM_clear_bit( feature_bit );

CM_load_context( feature_bit );
CM_clear_bit_always( temp_bit1 );
CM_send_1L( temp_bit1, address, feature_bit, 1, CM_no_field );
CM_logand_context( temp_bit1 );

CM_load_context( temp_bit1 );
CM_set_bit( feature_bit );
CM_f_move_1L( correlation, temp_float1, FLENS );
CM_s_negate_2_1L( disparity, dual_disparity, DISPARITY_LEN );
CM_s_negate_2_1L( alignment, dual_alignment, ALIGNMENT_LEN );

CM_load_context( coord[2] );
printf("  number of vectors in final disparity field: %u\n",
```

```
          CM_global_count_bit( feature_bit ) );fflush(stdout);

  CM_deallocate_stack_through( temp_bit1 );

}       /* of merge  */

/*****************************************************************************/
show_disparity_vectors( image, color_field )
     CM_field_id_t image, color_field;
{
  CM_field_id_t start_point[2], end_point[2];

  CM_set_vp_set( stereo_pair_grid );

  start_point[0] = CM_allocate_stack_field( IMAGE_SIZE_LEN+2 );
  start_point[1] = CM_allocate_stack_field( IMAGE_SIZE_LEN+2 );
  end_point[0] = CM_allocate_stack_field( IMAGE_SIZE_LEN+2 );
  end_point[1] = CM_allocate_stack_field( IMAGE_SIZE_LEN+2 );

  CM_load_context( coord[2] );
  CM_u_move_2L( start_point[0], coord[0], IMAGE_SIZE_LEN+2, IMAGE_SIZE_LEN );
  CM_u_add_constant_2_1L( start_point[0], IMAGE_SIZE, IMAGE_SIZE_LEN+2 );
  CM_u_move_2L( start_point[1], coord[1], IMAGE_SIZE_LEN+2, IMAGE_SIZE_LEN );

  CM_invert_context();
  CM_u_move_2L( start_point[0], coord[0], IMAGE_SIZE_LEN+2, IMAGE_SIZE_LEN );
  CM_u_move_2L( start_point[1], coord[1], IMAGE_SIZE_LEN+2, IMAGE_SIZE_LEN );

  CM_load_context( feature_bit );

  CM_s_add_3_3L( end_point[0], start_point[0], disparity,
                 IMAGE_SIZE_LEN+2, IMAGE_SIZE_LEN+2, DISPARITY_LEN );
  CM_s_add_3_3L( end_point[1], start_point[1], alignment,
                 IMAGE_SIZE_LEN+2, IMAGE_SIZE_LEN+2, ALIGNMENT_LEN );

  CMSR_s_draw_line( image, start_point[0], start_point[1],
                    end_point[0], end_point[1],
                    color_field, IMAGE_SIZE_LEN+2, BITS_PER_PIXEL,
                    CMSR_overwrite,
                    /* draw end point ? */ YES, /* clip line ? */ YES );

  CM_deallocate_stack_through( start_point[0] );

}       /* of show_disparity_vectors */

/*****************************************************************************/
allocate_object( object_p )
     object_t *object_p;
{
  int i, j;
  unsigned offset;

  (*object_p).total = CM_allocate_heap_field( OBJECT_LEN );
  offset = 0;
  for( i = 0; i < NB_FACES; i++ ) {
    (*object_p).face[i].total = CM_SUBFIELD( (*object_p).total, offset );
    for( j = 0; j < DIMENSION; j++ )
      (*object_p).face[i].normal_vector[j] =
        CM_SUBFIELD( (*object_p).face[i].total, j*FLEN );
    (*object_p).face[i].sign_bit =
      CM_SUBFIELD( (*object_p).face[i].total, j*FLEN );
    (*object_p).face[i].face_bit =
      CM_SUBFIELD( (*object_p).face[i].total, j*FLEN + 1 );
    offset += j*FLEN + 2;
  }
}       /* of allocate_object() */

/*****************************************************************************/
deallocate_object( object )
     object_t object;
{
  CM_deallocate_heap_field( object.total );
}       /* of deallocate_object() */
```

```
/*****************************************************************************/
my_cross_product( dest, vector1, vector2 )
     coordinates_t dest, vector1, vector2;
{
  CM_f_multiply_3_1L( dest[ 0 ], vector1[ 2 ], vector2[ 1 ], FLENS );
  CM_f_mult_sub_1L( dest[ 0 ], vector1[ 1 ], vector2[ 2 ],
                    dest[ 0 ], FLENS );
  CM_f_multiply_3_1L( dest[ 1 ], vector1[ 0 ], vector2[ 2 ], FLENS );
  CM_f_mult_sub_1L( dest[ 1 ], vector1[ 2 ], vector2[ 0 ],
                    dest[ 1 ], FLENS );
  CM_f_multiply_3_1L( dest[ 2 ], vector1[ 1 ], vector2[ 0 ], FLENS );
  CM_f_mult_sub_1L( dest[ 2 ], vector1[ 0 ], vector2[ 1 ],
                    dest[ 2 ], FLENS );
}       /* of my_cross_product() */

/*****************************************************************************/
scalar_product( dest, vector1, vector2 )
     CM_field_id_t dest;
     coordinates_t vector1, vector2;
{
  int i;

  CM_f_multiply_3_1L( dest, vector1[ 0 ], vector2[ 0 ], FLENS );
  for( i = 1; i < DIMENSION; i++ )
    CM_f_mult_add_1L( dest, vector1[ i ], vector2[ i ], dest, FLENS );
}       /* of scalar_product() */

/*****************************************************************************/
rotate( dest, source, angle_x, angle_y )
     coordinates_t dest, source;
     float angle_x, angle_y;
{
  float cos_angle_x, sin_angle_x;
  float cos_angle_y, sin_angle_y;
  cos_angle_x = cos( angle_x * PI/180 );
  sin_angle_x = sin( angle_x * PI/180 );
  cos_angle_y = cos( angle_y * PI/180 );
  sin_angle_y = sin( angle_y * PI/180 );

  CM_f_multiply_constant_3_1L( dest[ 0 ], source[ 0 ], cos_angle_y, FLENS );
  CM_f_mult_const_add_1L( dest[ 0 ], source[ 1 ], -sin_angle_y*sin_angle_x,
                          dest[ 0 ], FLENS );
  CM_f_mult_const_add_1L( dest[ 0 ], source[ 2 ], -sin_angle_y*cos_angle_x,
                          dest[ 0 ], FLENS );

  CM_f_multiply_constant_3_1L( dest[ 1 ], source[ 0 ], 0.0, FLENS );
  CM_f_mult_const_add_1L( dest[ 1 ], source[ 1 ], cos_angle_x,
                          dest[ 1 ], FLENS );
  CM_f_mult_const_add_1L( dest[ 1 ], source[ 2 ], -sin_angle_x,
                          dest[ 1 ], FLENS );

  CM_f_multiply_constant_3_1L( dest[ 2 ], source[ 0 ], sin_angle_y, FLENS );
  CM_f_mult_const_add_1L( dest[ 2 ], source[ 1 ], cos_angle_y*sin_angle_x,
                          dest[ 2 ], FLENS );
  CM_f_mult_const_add_1L( dest[ 2 ], source[ 2 ], cos_angle_y*cos_angle_x,
                          dest[ 2 ], FLENS );

}       /* of rotate() */

/*****************************************************************************/
multiply_vector( dest, source, scalar )
     coordinates_t dest, source;
     CM_field_id_t scalar;
{
  int i;

  for( i = 0; i < DIMENSION; i++ )
    CM_f_multiply_3_1L( dest[ i ], source[ i ], scalar, FLENS );
}     /* of multiply_vector() */

/*****************************************************************************/
add_vector( dest, vector )
```

```
     coordinates_t dest;
     float         vector[ DIMENSION ];
{
  int i;

  for( i = 0; i < DIMENSION; i++ )
    CM_f_add_constant_2_1L( dest[ i ], vector[ i ], FLENS );
}     /* of add_vector() */

/*****************************************************************************/
status_t
compact_face( face, point, source_vector, temp_float1, temp_float2 )
     face_t  face;
     coordinates_t point, source_vector;
     CM_field_id_t temp_float1, temp_float2;
{
  CM_logand_context( face.face_bit );
  /* compute scalar product of source_vector with vector defined by origin
     and point */
  scalar_product( temp_float1, source_vector, point );
  /* compute square norm of source_vector */
  scalar_product( temp_float2, source_vector, source_vector );
  /* and normalize */
  CM_f_divide_2_1L( temp_float1, temp_float2, FLENS );

  multiply_vector( face.normal_vector, source_vector, temp_float1 );

  /* the sign of temp_float1 determines whether we must invert sign_bit */
  CM_f_eq_zero_1L( temp_float1, FLENS );
  if( CM_global_count_test() )
    return( SINGULARITY_ERROR );

  CM_f_lt_zero_1L( temp_float1, FLENS );
  CM_logand_context_with_test();
  /* invert the sign bit of face if scalar product is negative */
  CM_load_overflow( face.sign_bit );
  CM_invert_overflow();
  CM_store_overflow( face.sign_bit );

  return( NO_ERROR );
}       /* of compact_face() */

/*****************************************************************************/
status_t
make_object( object, feature, disp, config, displacement )
     object_t            object;
     CM_field_id_t       feature, disp;
     config_t            config;
     displacement_t      displacement;
{
#define SIGNED_LEN       (IMAGE_SIZE_LEN + 1)

  status_t      status;
  int i, j;
  CM_field_id_t match;
  CM_field_id_t temp_float1, temp_float2, temp_signed;
  CM_field_id_t delta_x, delta_x_plus_one, delta_x_minus_one;
  CM_field_id_t index;
  CM_field_id_t coords, coord[ 2 ];
  coordinates_t temp_vector1, temp_vector2;
  coordinates_t point1, point2, vector1, vector2;

  /* allocate and initialize coord field */
  coords = CM_allocate_stack_field( 2*SIGNED_LEN );
  coord[ 0 ] = CM_SUBFIELD( coords, 0 );
  coord[ 1 ] = CM_SUBFIELD( coords, SIGNED_LEN );
  CM_set_context();
  CM_my_news_coordinate_1L( coord[ 0 ], /* axis */ 0, SIGNED_LEN );
  CM_s_subtract_constant_2_1L( coord[ 0 ], CENTER_OFFSET_X, SIGNED_LEN );
  CM_my_news_coordinate_1L( coord[ 1 ], /* axis */ 1, SIGNED_LEN );
  CM_s_subfrom_constant_2_1L( coord[ 1 ], IMAGE_SIZE-1, IMAGE_SIZE_LEN );
  CM_s_subtract_constant_2_1L( coord[ 1 ], CENTER_OFFSET_Y, SIGNED_LEN );
```

```
/* allocate temporary vectors */
point1[0] = CM_allocate_stack_field( FPOINT_LEN );
for( i = 1; i < DIMENSION; i++ )
  point1[i] = CM_SUBFIELD( point1[i-1], FLEN );
point2[0] = CM_allocate_stack_field( FPOINT_LEN );
for( i = 1; i < DIMENSION; i++ )
  point2[i] = CM_SUBFIELD( point2[i-1], FLEN );
vector1[0] = CM_allocate_stack_field( FPOINT_LEN );
for( i = 1; i < DIMENSION; i++ )
  vector1[i] = CM_SUBFIELD( vector1[i-1], FLEN );
vector2[0] = CM_allocate_stack_field( FPOINT_LEN );
for( i = 1; i < DIMENSION; i++ )
  vector2[i] = CM_SUBFIELD( vector2[i-1], FLEN );
temp_vector1[0] = CM_allocate_stack_field( FPOINT_LEN );
for( i = 1; i < DIMENSION; i++ )
  temp_vector1[i] = CM_SUBFIELD( temp_vector1[i-1], FLEN );
temp_vector2[0] = CM_allocate_stack_field( FPOINT_LEN );
for( i = 1; i < DIMENSION; i++ )
  temp_vector2[i] = CM_SUBFIELD( temp_vector2[i-1], FLEN );

/* allocate temporary fields */
temp_signed = CM_allocate_stack_field( SIGNED_LEN );
temp_float1 = CM_allocate_stack_field( 2*FLEN );
temp_float2 = CM_SUBFIELD( temp_float1, FLEN );
delta_x = CM_allocate_stack_field( 3*FLEN );
delta_x_plus_one = CM_SUBFIELD( delta_x, FLEN );
delta_x_minus_one = CM_SUBFIELD( delta_x, 2*FLEN );

/* negate disparity so that we get Left-Right, a positive value */
CM_s_negate_1_1L( disp, DISPARITY_LEN );

/* determine match from disp and coord */
match = CM_allocate_stack_field( SIGNED_LEN );
CM_load_context( feature );
CM_s_subtract_3_3L( match,      coord[0],   disp,
                    SIGNED_LEN, SIGNED_LEN, DISPARITY_LEN );

CM_f_s_float_2_2L( delta_x, disp, DISPARITY_LEN, FLENS );
CM_f_add_constant_3_1L( delta_x_plus_one, delta_x, /* here */ 1.0, FLENS );
CM_f_add_constant_3_1L( delta_x_minus_one, delta_x, -/* here */ 1.0, FLENS );

/* now initialize object */
/* note: angles are COUNTER-clockwise (trigonometrical) when viewed
   from a positive coordinate along the axis */

/* object.face[ 0 ] */
/* furthest left vertical plane */
/* keep this face only where delta_x > 0 */
CM_load_context( feature );
CM_f_gt_zero_1L( delta_x, FLENS );
CM_store_test( object.face[ 0 ].face_bit );

CM_set_bit( object.face[ 0 ].sign_bit );

CM_f_move_constant_1L( vector1[ 1 ], 0.0, FLENS );
CM_f_s_float_2_2L( vector1[ 2 ], coord[ 0 ], SIGNED_LEN, FLENS );
CM_f_negate_1_1L( vector1[ 2 ], FLENS );
CM_f_move_constant_1L( point1[ 0 ], -config.baseline.max/2, FLENS );
CM_f_move_constant_1L( point1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 2 ], 0.0, FLENS );

CM_s_gt_zero_1L( coord[ 0 ], SIGNED_LEN );
CM_logand_context_with_test();
CM_f_move_constant_1L( vector1[ 0 ], config.focal.max, FLENS );
CM_invert_context();
CM_f_move_constant_1L( vector1[ 0 ], config.focal.min, FLENS );
CM_load_context( feature );
rotate( vector2, vector1, config.pitch.min, displacement.yaw.max );
rotate( point2,  point1,  config.pitch.min, displacement.yaw.max );
add_vector( point2, displacement.translation.vector );
status = compact_face( object.face[ 0 ], point2, vector2, temp_float1, temp_float2 );

if( status != NO_ERROR ) return( status );
```

```
/* object.face[ 1 ] */
/* closest right vertical plane */
/* always keep this face */
CM_load_context( feature );
CM_set_bit( object.face[ 1 ].face_bit );

CM_clear_bit( object.face[ 1 ].sign_bit );

CM_f_move_constant_1L( vector1[ 1 ], 0.0, FLENS );
CM_f_s_float_2_2L( vector1[ 2 ], coord[ 0 ], SIGNED_LEN, FLENS );
CM_f_add_constant_2_1L( vector1[ 2 ], /* here */ 1.0, FLENS );
CM_f_negate_1_1L( vector1[ 2 ], FLENS );
CM_f_move_constant_1L( point1[ 0 ], -config.baseline.min/2, FLENS );
CM_f_move_constant_1L( point1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 2 ], 0.0, FLENS );

CM_s_gt_constant_1L( coord[ 0 ], -1, SIGNED_LEN );
CM_logand_context_with_test();
CM_f_move_constant_1L( vector1[ 0 ], config.focal.min, FLENS );
CM_invert_context();
CM_f_move_constant_1L( vector1[ 0 ], config.focal.max, FLENS );
CM_load_context( feature );
rotate( vector2, vector1, config.pitch.min, displacement.yaw.min );
rotate( point2,  point1,  config.pitch.min, displacement.yaw.min );
add_vector( point2, displacement.translation.vector );
status = compact_face( object.face[ 1 ], point2, vector2, temp_float1, temp_float2 );

if( status != NO_ERROR ) return( status );

/* object.face[ 2 ] */
/* closest left vertical plane */
/* always keep this face */
CM_load_context( feature );
CM_set_bit( object.face[ 2 ].face_bit );
CM_set_bit( object.face[ 2 ].sign_bit );

CM_f_move_constant_1L( vector1[ 1 ], 0.0, FLENS );
CM_f_s_float_2_2L( vector1[ 2 ], match, SIGNED_LEN, FLENS );
CM_f_negate_1_1L( vector1[ 2 ], FLENS );
CM_f_move_constant_1L( point1[ 0 ], config.baseline.min/2, FLENS );
CM_f_move_constant_1L( point1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 2 ], 0.0, FLENS );

CM_s_gt_zero_1L( match, SIGNED_LEN );
CM_logand_context_with_test();
CM_f_move_constant_1L( vector1[ 0 ], config.focal.max, FLENS );
CM_invert_context();
CM_f_move_constant_1L( vector1[ 0 ], config.focal.min, FLENS );
CM_load_context( feature );
rotate( vector2, vector1, config.pitch.min, displacement.yaw.max );
rotate( point2,  point1,  config.pitch.min, displacement.yaw.max );
add_vector( point2, displacement.translation.vector );
status = compact_face( object.face[ 2 ], point2, vector2, temp_float1, temp_float2 );

if( status != NO_ERROR ) return( status );

/* object.face[ 3 ] */
/* furthest right vertical plane */
/* keep this face only where delta_x > 0 */
CM_load_context( feature );
CM_f_gt_zero_1L( delta_x, FLENS );
CM_store_test( object.face[ 3 ].face_bit );

CM_clear_bit( object.face[ 3 ].sign_bit );

CM_f_move_constant_1L( vector1[ 1 ], 0.0, FLENS );
CM_f_s_float_2_2L( vector1[ 2 ], match, SIGNED_LEN, FLENS );
CM_f_add_constant_2_1L( vector1[ 2 ], /* here */ 1.0, FLENS );
CM_f_negate_1_1L( vector1[ 2 ], FLENS );
CM_f_move_constant_1L( point1[ 0 ], config.baseline.max/2, FLENS );
CM_f_move_constant_1L( point1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 2 ], 0.0, FLENS );
```

```
CM_s_gt_constant_1L( match, -1, SIGNED_LEN );
CM_logand_context_with_test();
CM_f_move_constant_1L( vector1[ 0 ], config.focal.min, FLENS );
CM_invert_context();
CM_f_move_constant_1L( vector1[ 0 ], config.focal.max, FLENS );
CM_load_context( feature );
rotate( vector2, vector1, config.pitch.min, displacement.yaw.min );
rotate( point2,  point1,  config.pitch.min, displacement.yaw.min );
add_vector( point2, displacement.translation.vector );
status = compact_face( object.face[ 3 ], point2, vector2, temp_float1, temp_float2 );

if( status != NO_ERROR ) return( status );

/* object.face[ 4 ] */
/* plane below set passing through origin (approximation) */
/* always keep this face */
CM_load_context( feature );
CM_set_bit( object.face[ 4 ].face_bit );

CM_clear_bit( object.face[ 4 ].sign_bit );

CM_f_move_constant_1L( point1[ 0 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 2 ], 0.0, FLENS );
/* no need for rotation ... */

/* note: (xl+xr)/2 = Delta_x*( xl/Delta_x - 1/2 ) */
CM_f_s_float_2_2L( temp_float1, coord[ 0 ], SIGNED_LEN, FLENS );
CM_f_s_float_2_2L( vector1[ 0 ], match, SIGNED_LEN, FLENS );
CM_f_add_2_1L( vector1[ 0 ], temp_float1, FLENS );
CM_f_divide_constant_2_1L( vector1[ 0 ], 2.0, FLENS );
CM_f_s_float_2_2L( vector1[ 1 ], coord[ 1 ], SIGNED_LEN, FLENS );

CM_s_gt_zero_1L( coord[ 1 ], SIGNED_LEN );
CM_logand_context_with_test();
CM_f_move_constant_1L( vector1[ 2 ], config.focal.max, FLENS );
CM_invert_context();
CM_f_move_constant_1L( vector1[ 2 ], config.focal.min, FLENS );

CM_load_context( feature );
/* compute vector2 = vector1 + ( 1.0, 0, 0) */
for( i = 0; i < DIMENSION; i++ )
  CM_f_move_1L( vector2[ i ], vector1[ i ], FLENS );
CM_f_add_constant_2_1L( vector2[ 0 ], /* here */ 1.0, FLENS );

rotate( temp_vector1, vector1, config.pitch.min, displacement.yaw.max );
rotate( temp_vector2, vector2, config.pitch.min, displacement.yaw.min );
/* now compute cross product of temp_vector1 and temp_vector2 */
my_cross_product( vector1, temp_vector2, temp_vector1 );

add_vector( point1, displacement.translation.vector );
status = compact_face( object.face[ 4 ], point1, vector1, temp_float1, temp_float2 );

if( status != NO_ERROR ) return( status );

/* object.face[ 5 ] */
/* plane above set passing through origin (approximation) */
/* always keep this face */
CM_load_context( feature );
CM_set_bit( object.face[ 5 ].face_bit );

CM_set_bit( object.face[ 5 ].sign_bit );

CM_f_move_constant_1L( point1[ 0 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 2 ], 0.0, FLENS );
/* no need for rotation ... */

/* note: (xl+xr)/2 = Delta_x*( xl/Delta_x - 1/2 ) */
CM_f_s_float_2_2L( temp_float1, coord[ 0 ], SIGNED_LEN, FLENS );
CM_f_s_float_2_2L( vector1[ 0 ], match, SIGNED_LEN, FLENS );
CM_f_add_2_1L( vector1[ 0 ], temp_float1, FLENS );
```

```
CM_f_divide_constant_2_1L( vector1[ 0 ], 2.0, FLENS );
CM_f_s_float_2_2L( vector1[ 1 ], coord[ 1 ], SIGNED_LEN, FLENS );
CM_f_add_constant_2_1L( vector1[ 1 ], /* here */ 1.0, FLENS );

CM_s_gt_constant_1L( coord[ 1 ], -1, SIGNED_LEN );
CM_logand_context_with_test();
CM_f_move_constant_1L( vector1[ 2 ], config.focal.min, FLENS );
CM_invert_context();
CM_f_move_constant_1L( vector1[ 2 ], config.focal.max, FLENS );

CM_load_context( feature );
/* vector2 = vector1 + ( 1.0, 0, 0) */
for( i = 0; i < DIMENSION; i++ )
  CM_f_move_1L( vector2[ i ], vector1[ i ], FLENS );
CM_f_add_constant_2_1L( vector2[ 0 ], /* here */ 1.0, FLENS );

rotate( temp_vector1, vector1, config.pitch.min, displacement.yaw.max );
rotate( temp_vector2, vector2, config.pitch.min, displacement.yaw.min );
/* now compute cross product of temp_vector1 and temp_vector2 */
my_cross_product( vector1, temp_vector2, temp_vector1 );

add_vector( point1, displacement.translation.vector );
status = compact_face( object.face[ 5 ], point1, vector1, temp_float1, temp_float2 );

if( status != NO_ERROR ) return( status );

/* object.face[ 6 ] */
/* vertical plane perpendicular to (O,x) at the left of the set */
CM_load_context( feature );
CM_set_bit( object.face[ 6 ].sign_bit );

CM_f_move_constant_1L( vector1[ 0 ], 1.0, FLENS );
CM_f_move_constant_1L( vector1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( vector1[ 2 ], 0.0, FLENS );
rotate( vector2, vector1, config.pitch.min, displacement.yaw.max );
CM_f_move_constant_1L( point1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 2 ], 0.0, FLENS );

CM_s_gt_zero_1L( match, SIGNED_LEN );
CM_logand_context_with_test();
CM_f_s_float_2_2L( point1[ 0 ], coord[ 0 ], SIGNED_LEN, FLENS );
CM_f_add_constant_2_1L( point1[ 0 ], /* here */ 1.0, FLENS );
CM_f_divide_2_1L( point1[ 0 ], delta_x_plus_one, FLENS );
/* always keep this face */
CM_set_bit( object.face[ 6 ].face_bit );

CM_load_context( feature );
CM_s_le_zero_1L( match, SIGNED_LEN );
CM_logand_context_with_test();
CM_s_gt_zero_1L( coord[ 0 ], SIGNED_LEN );
CM_logand_context_with_test();
CM_f_s_float_2_2L( point1[ 0 ], coord[ 0 ], SIGNED_LEN, FLENS );
CM_f_divide_2_1L( point1[ 0 ], delta_x, FLENS );
/* keep this face only where delta_x > 0 */
CM_f_gt_zero_1L( delta_x, FLENS );
CM_store_test( object.face[ 6 ].face_bit );

CM_load_context( feature );
CM_s_le_zero_1L( match, SIGNED_LEN );
CM_logand_context_with_test();
CM_s_le_zero_1L( coord[ 0 ], SIGNED_LEN );
CM_logand_context_with_test();
CM_f_s_float_2_2L( point1[ 0 ], coord[ 0 ], SIGNED_LEN, FLENS );
CM_f_divide_2_1L( point1[ 0 ], delta_x_minus_one, FLENS );
/* keep this face only where delta_x_minus_one > 0 */
CM_f_gt_zero_1L( delta_x_minus_one, FLENS );
CM_store_test( object.face[ 6 ].face_bit );

CM_load_context( feature );
CM_f_add_constant_2_1L( point1[ 0 ], -.5, FLENS );
CM_s_add_3_1L( temp_signed, coord[ 0 ], match, SIGNED_LEN );
CM_s_gt_zero_1L( temp_signed, SIGNED_LEN );
CM_logand_context_with_test();
```

```
CM_f_multiply_constant_2_1L( point1[ 0 ], config.baseline.min, FLENS );
CM_invert_context();
CM_f_multiply_constant_2_1L( point1[ 0 ], config.baseline.max, FLENS );

CM_load_context( feature );
rotate( point2, point1, config.pitch.min, displacement.yaw.max );
add_vector( point2, displacement.translation.vector );
status = compact_face( object.face[ 6 ], point2, vector2, temp_float1, temp_float2 );

if( status != NO_ERROR ) return( status );

/* object.face[ 7 ] */
/* vertical plane perpendicular to (O,x) at the right of the set */
CM_load_context( feature );
CM_clear_bit( object.face[ 7 ].sign_bit );

CM_f_move_constant_1L( vector1[ 0 ], 1.0, FLENS );
CM_f_move_constant_1L( vector1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( vector1[ 2 ], 0.0, FLENS );
rotate( vector2, vector1, config.pitch.min, displacement.yaw.min );
CM_f_move_constant_1L( point1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 2 ], 0.0, FLENS );

CM_s_gt_constant_1L( match, -1, SIGNED_LEN );
CM_logand_context_with_test();
CM_f_s_float_2_2L( point1[ 0 ], coord[ 0 ], SIGNED_LEN, FLENS );
CM_f_divide_2_1L( point1[ 0 ], delta_x_minus_one, FLENS );
/* keep this face only where delta_x_minus_one > 0 */
CM_f_gt_zero_1L( delta_x_minus_one, FLENS );
CM_store_test( object.face[ 7 ].face_bit );

CM_load_context( feature );
CM_s_le_constant_1L( match, -1, SIGNED_LEN );
CM_logand_context_with_test();
CM_s_gt_constant_1L( coord[ 0 ], -1, SIGNED_LEN );
CM_logand_context_with_test();
CM_f_s_float_2_2L( point1[ 0 ], coord[ 0 ], SIGNED_LEN, FLENS );
CM_f_add_constant_2_1L( point1[ 0 ], /* here */ 1.0, FLENS );
CM_f_divide_2_1L( point1[ 0 ], delta_x, FLENS );
/* keep this face only where delta_x > 0 */
CM_f_gt_zero_1L( delta_x, FLENS );
CM_store_test( object.face[ 7 ].face_bit );

CM_load_context( feature );
CM_s_le_constant_1L( match, -1, SIGNED_LEN );
CM_logand_context_with_test();
CM_s_le_constant_1L( coord[ 0 ], -1, SIGNED_LEN );
CM_logand_context_with_test();
CM_f_s_float_2_2L( point1[ 0 ], coord[ 0 ], SIGNED_LEN, FLENS );
CM_f_add_constant_2_1L( point1[ 0 ], /* here */ 1.0, FLENS );
CM_f_divide_2_1L( point1[ 0 ], delta_x_plus_one, FLENS );
/* always keep this face */
CM_set_bit( object.face[ 7 ].face_bit );

CM_load_context( feature );
CM_f_add_constant_2_1L( point1[ 0 ], -.5, FLENS );
CM_s_add_3_1L( temp_signed, coord[ 0 ], match, SIGNED_LEN );
CM_s_add_constant_2_1L( temp_signed, 2, SIGNED_LEN );
CM_s_gt_zero_1L( temp_signed, SIGNED_LEN );
CM_logand_context_with_test();
CM_f_multiply_constant_2_1L( point1[ 0 ], config.baseline.max, FLENS );
CM_invert_context();
CM_f_multiply_constant_2_1L( point1[ 0 ], config.baseline.min, FLENS );

CM_load_context( feature );
rotate( point2, point1, config.pitch.min, displacement.yaw.min );
add_vector( point2, displacement.translation.vector );
status = compact_face( object.face[ 7 ], point2, vector2, temp_float1, temp_float2 );
if( status != NO_ERROR ) return( status );

/* object.face[ 8 ] */
/* horizontal plane below set */
CM_load_context( feature );
```

```
CM_set_bit( object.face[ 8 ].sign_bit );

CM_f_move_constant_1L( vector1[ 0 ], 0.0, FLENS );
CM_f_move_constant_1L( vector1[ 1 ], 1.0, FLENS );
CM_f_move_constant_1L( vector1[ 2 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 0 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 2 ], 0.0, FLENS );
CM_f_s_float_2_2L( point1[ 1 ], coord[ 1 ], SIGNED_LEN, FLENS );

CM_s_gt_zero_1L( coord[ 1 ], SIGNED_LEN );
CM_logand_context_with_test();
CM_f_divide_2_1L( point1[ 1 ], delta_x_plus_one, FLENS );
CM_f_multiply_constant_2_1L( point1[ 1 ], config.baseline.min , FLENS );
/* always keep this face */
CM_set_bit( object.face[ 8 ].face_bit );

CM_invert_context();
CM_f_divide_2_1L( point1[ 1 ], delta_x_minus_one, FLENS );
CM_f_multiply_constant_2_1L( point1[ 1 ], config.baseline.max , FLENS );
/* keep this face only where delta_x_minus_one > 0 */
CM_f_gt_zero_1L( delta_x_minus_one, FLENS );
CM_store_test( object.face[ 8 ].face_bit );

CM_load_context( feature );
rotate( point2, point1, config.pitch.min, displacement.yaw.min );
rotate( vector2, vector1, config.pitch.min, displacement.yaw.min );
add_vector( point2, displacement.translation.vector );
status = compact_face( object.face[ 8 ], point2, vector2, temp_float1, temp_float2 );
if( status != NO_ERROR ) return( status );

/* object.face[ 9 ] */
/* horizontal plane above set */
CM_load_context( feature );
CM_clear_bit( object.face[ 9 ].sign_bit );

CM_f_move_constant_1L( vector1[ 0 ], 0.0, FLENS );
CM_f_move_constant_1L( vector1[ 1 ], 1.0, FLENS );
CM_f_move_constant_1L( vector1[ 2 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 0 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 2 ], 0.0, FLENS );
CM_f_s_float_2_2L( point1[ 1 ], coord[ 1 ], SIGNED_LEN, FLENS );
CM_f_add_constant_2_1L( point1[ 1 ], /* here */ 1.0, FLENS );

CM_s_gt_constant_1L( coord[ 1 ], -1, SIGNED_LEN );
CM_logand_context_with_test();
CM_f_divide_2_1L( point1[ 1 ], delta_x_minus_one, FLENS );
CM_f_multiply_constant_2_1L( point1[ 1 ], config.baseline.max , FLENS );
/* keep this face only where delta_x_minus_one > 0 */
CM_f_gt_zero_1L( delta_x_minus_one, FLENS );
CM_store_test( object.face[ 9 ].face_bit );

CM_invert_context();
CM_f_divide_2_1L( point1[ 1 ], delta_x_plus_one, FLENS );
CM_f_multiply_constant_2_1L( point1[ 1 ], config.baseline.min , FLENS );
/* always keep this face */
CM_set_bit( object.face[ 9 ].face_bit );

CM_load_context( feature );
rotate( point2, point1, config.pitch.min, displacement.yaw.min );
rotate( vector2, vector1, config.pitch.min, displacement.yaw.min );
add_vector( point2, displacement.translation.vector );
status = compact_face( object.face[ 9 ], point2, vector2, temp_float1, temp_float2 );
if( status != NO_ERROR ) return( status );

/* object.face[ 10 ] */
/* vertical plane at left of set passing through origin */
/* keep this face only where delta_x > 0 */
CM_load_context( feature );
CM_f_gt_zero_1L( delta_x, FLENS );
CM_store_test( object.face[ 10 ].face_bit );

CM_set_bit( object.face[ 10 ].sign_bit );
```

```
CM_f_move_constant_1L( point1[ 0 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 2 ], 0.0, FLENS );
/* no need for rotation */

CM_s_gt_zero_1L( match, SIGNED_LEN );
CM_logand_context_with_test();
CM_f_move_constant_1L( vector1[ 0 ], config.focal.max, FLENS );
CM_invert_context();
CM_f_move_constant_1L( vector1[ 0 ], config.focal.min, FLENS );
CM_load_context( feature );

CM_f_move_constant_1L( vector1[ 1 ], 0.0, FLENS );
CM_f_s_float_2_2L( temp_float1, coord[ 0 ], SIGNED_LEN, FLENS );
CM_f_s_float_2_2L( vector1[ 2 ], match, SIGNED_LEN, FLENS );
CM_f_add_2_1L( vector1[ 2 ], temp_float1, FLENS );
CM_f_divide_constant_2_1L( vector1[ 2 ], 2.0, FLENS );
CM_f_negate_1_1L( vector1[ 2 ], FLENS );

rotate( vector2, vector1, config.pitch.min, displacement.yaw.max );
add_vector( point1, displacement.translation.vector );
status = compact_face( object.face[ 10 ], point1, vector2, temp_float1, temp_float2 );
if( status != NO_ERROR ) return( status );

/* object.face[ 11 ] */
/* vertical plane at right of set passing through origin */
/* keep this face only where delta_x > 0 */
CM_load_context( feature );
CM_f_gt_zero_1L( delta_x, FLENS );
CM_store_test( object.face[ 11 ].face_bit );

CM_clear_bit( object.face[ 11 ].sign_bit );

CM_f_move_constant_1L( point1[ 0 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 2 ], 0.0, FLENS );
/* no need for rotation */

CM_s_gt_constant_1L( coord[ 0 ], -1, SIGNED_LEN );
CM_logand_context_with_test();
CM_f_move_constant_1L( vector1[ 0 ], config.focal.min, FLENS );
CM_invert_context();
CM_f_move_constant_1L( vector1[ 0 ], config.focal.max, FLENS );
CM_load_context( feature );

CM_f_move_constant_1L( vector1[ 1 ], 0.0, FLENS );
CM_f_s_float_2_2L( temp_float1, coord[ 0 ], SIGNED_LEN, FLENS );
CM_f_s_float_2_2L( vector1[ 2 ], match, SIGNED_LEN, FLENS );
CM_f_add_2_1L( vector1[ 2 ], temp_float1, FLENS );
CM_f_divide_constant_2_1L( vector1[ 2 ], 2.0, FLENS );
CM_f_add_constant_2_1L( vector1[ 2 ], /* here */ 1.0, FLENS );
CM_f_negate_1_1L( vector1[ 2 ], FLENS );

rotate( vector2, vector1, config.pitch.min, displacement.yaw.min );
add_vector( point1, displacement.translation.vector );
status = compact_face( object.face[ 11 ], point1, vector2, temp_float1, temp_float2 );
if( status != NO_ERROR ) return( status );

/* object.face[ 12 ] */
/* closest vertical plane approximating yaw.min-yaw.max curve */
CM_load_context( feature );
if( displacement.yaw.max > displacement.yaw.min ) {
  /* always keep this face */
  CM_set_bit( object.face[ 12 ].face_bit );
  CM_set_bit( object.face[ 12 ].sign_bit );

  CM_f_s_float_2_2L( vector1[ 0 ], coord[ 0 ], SIGNED_LEN, FLENS );
  CM_f_add_constant_2_1L( vector1[ 0 ], /* here */ 1.0, FLENS );
  CM_f_s_float_2_2L( vector1[ 1 ], coord[ 1 ], SIGNED_LEN, FLENS );
  CM_f_move_constant_1L( vector1[ 2 ], config.focal.min, FLENS );
  for( i = 0; i < DIMENSION; i++ )
    CM_f_divide_2_1L( vector1[ i ], delta_x_plus_one, FLENS );
  CM_f_add_constant_2_1L( vector1[ 0 ], -.5, FLENS );
```

```
    for( i = 0; i < DIMENSION; i++ )
      CM_f_multiply_constant_2_1L( vector1[ i ], config.baseline.min, FLENS );
    rotate( temp_vector1, vector1, config.pitch.min, displacement.yaw.max );
    rotate( temp_vector2, vector1, config.pitch.min, displacement.yaw.min );
    for( i = 0; i < DIMENSION; i++ )
      CM_f_subtract_2_1L( temp_vector1[ i ], temp_vector2[ i ], FLENS );

    /* compute cross product vector1 = (0,1,0) x temp_vector1 */
    CM_f_move_1L( vector1[ 0 ], temp_vector1[ 2 ], FLENS );
    CM_f_move_zero_1L( vector1[ 1 ], FLENS );
    CM_f_move_1L( vector1[ 2 ], temp_vector1[ 0 ], FLENS );
    CM_f_negate_1_1L( vector1[ 2 ], FLENS );

    for( i = 0; i < DIMENSION; i++ )
      CM_f_move_1L( point1[ i ], temp_vector2[ i ], FLENS );

    add_vector( point1, displacement.translation.vector );
    status = compact_face( object.face[ 12 ], point1, vector1, temp_float1, temp_float2
    if( status != NO_ERROR ) return( status );

    /* object.face[ 13 ] */
    /* furthest vertical plane approximating yaw.min-yaw.max curve */
    /* keep this face only where delta_x_minus_one > 0 */
    CM_load_context( feature );
    CM_f_gt_zero_1L( delta_x_minus_one, FLENS );
    CM_store_test( object.face[ 13 ].face_bit );

    CM_clear_bit( object.face[ 13 ].sign_bit );

    CM_f_s_float_2_2L( vector1[ 0 ], coord[ 0 ], SIGNED_LEN, FLENS );
    CM_f_s_float_2_2L( vector1[ 1 ], coord[ 1 ], SIGNED_LEN, FLENS );
    CM_f_move_constant_1L( vector1[ 2 ], config.focal.max, FLENS );
    for( i = 0; i < DIMENSION; i++ )
      CM_f_divide_2_1L( vector1[ i ], delta_x_minus_one, FLENS );
    CM_f_add_constant_2_1L( vector1[ 0 ], -.5, FLENS );
    for( i = 0; i < DIMENSION; i++ )
      CM_f_multiply_constant_2_1L( vector1[ i ], config.baseline.max, FLENS );
    rotate( temp_vector1, vector1, config.pitch.min, displacement.yaw.max );
    rotate( temp_vector2, vector1, config.pitch.min, displacement.yaw.min );
    for( i = 0; i < DIMENSION; i++ )
      CM_f_subtract_2_1L( temp_vector1[ i ], temp_vector2[ i ], FLENS );

    /* compute cross product vector1 = (0,1,0) x temp_vector1 */
    CM_f_move_1L( vector1[ 0 ], temp_vector1[ 2 ], FLENS );
    CM_f_move_zero_1L( vector1[ 1 ], FLENS );
    CM_f_move_1L( vector1[ 2 ], temp_vector1[ 0 ], FLENS );
    CM_f_negate_1_1L( vector1[ 2 ], FLENS );

    for( i = 0; i < DIMENSION; i++ )
      CM_f_move_1L( point1[ i ], temp_vector2[ i ], FLENS );

    add_vector( point1, displacement.translation.vector );
    status = compact_face( object.face[ 13 ], point1, vector1, temp_float1, temp_float2
    if( status != NO_ERROR ) return( status );

  }
  else {      /* discard face[ 12 ] and face[ 13 ] */
    CM_clear_bit( object.face[ 12 ].face_bit );
    CM_clear_bit( object.face[ 13 ].face_bit );
  }

  CM_deallocate_stack_through( coords );

  return( NO_ERROR );

}       /* of make_object() */

/*****************************************************************************/
status_t
dilate_object( object, imprecision, feature )
     object_t      object;
     float         imprecision;
     CM_field_id_t feature;
```

```
{
  int i, j;

  /* allocate temporary fields */
  temp = CM_allocate_stack_field( 2*FLEN );
  temp_float1 = CM_SUBFIELD( temp, 0 );
  temp_float2 = CM_SUBFIELD( temp, FLEN );

  for( i = 0; i < NB_FACES; i++ ) {

    CM_load_context( feature );
    CM_logand_context( object.face[ i ].face_bit );
    if( CM_global_count_context() ) {

      /* compute norm of normal vector */
      scalar_product( temp_float1, object.face[ i ].normal_vector,
                      object.face[ i ].normal_vector, FLENS );
      CM_f_sqrt_1_1L( temp_float1, FLENS );

      CM_f_move_constant_1L( temp_float2, imprecision, FLENS );
      /* select faces pointing into object (sign_bit = 1) */
      CM_logand_context( object.face[ i ].sign_bit );
      CM_f_negate_1_1L( temp_float2, FLENS );

      CM_load_context( feature );
      CM_logand_context( object.face[ i ].face_bit );

      CM_f_add_2_1L( temp_float2, temp_float1, FLENS );
      CM_f_divide_2_1L( temp_float2, temp_float1, FLENS );

      /* the sign of temp_float2 determines whether we must invert sign_bit */
      CM_f_eq_zero_1L( temp_float2, FLENS );
      if( CM_global_count_test() )
        return( SINGULARITY_ERROR );
      CM_f_lt_zero_1L( temp_float2, FLENS );
      CM_logand_context_with_test();
      /* invert the sign bit of those faces  */
      CM_load_overflow( object.face[ i ].sign_bit );
      CM_invert_overflow();
      CM_store_overflow( object.face[ i ].sign_bit );

      /* now multiply normal vector by temp_float2 */
      CM_load_context( feature );
      CM_logand_context( object.face[ i ].face_bit );
      multiply_vector( object.face[ i ].normal_vector,
                       object.face[ i ].normal_vector, temp_float2 );
    }
  }

  CM_deallocate_stack_through( temp );

  return( NO_ERROR );
}       /* of dilate_object(); */

/*****************************************************************************/
status_t
erode_object( object, imprecision, feature )
     object_t      object;
     float         imprecision;
     CM_field_id_t feature;
{
  int i, j;
  CM_field_id_t temp, temp_float1, temp_float2;

  /* allocate temporary fields */
  temp = CM_allocate_stack_field( 2*FLEN );
  temp_float1 = CM_SUBFIELD( temp, 0 );
  temp_float2 = CM_SUBFIELD( temp, FLEN );

  for( i = 0; i < NB_FACES; i++ ) {

    CM_load_context( feature );
    CM_logand_context( object.face[ i ].face_bit );
```

```
    if( CM_global_count_context() ) {

      /* compute norm of normal vector */
      scalar_product( temp_float1, object.face[ i ].normal_vector,
                      object.face[ i ].normal_vector, FLENS );
      CM_f_sqrt_1_1L( temp_float1, FLENS );

      CM_f_move_constant_1L( temp_float2, imprecision, FLENS );
      /* select faces pointing into object (sign_bit = 1) */
      CM_logand_context( object.face[ i ].sign_bit );
      CM_f_negate_1_1L( temp_float2, FLENS );

      CM_load_context( feature );

      CM_f_subfrom_2_1L( temp_float2, temp_float1, FLENS );
      CM_f_divide_2_1L( temp_float2, temp_float1, FLENS );

      /* the sign of temp_float2 determines whether we must invert sign_bit */
      CM_f_eq_zero_1L( temp_float2, FLENS );
      if( CM_global_count_test() )
        return( SINGULARITY_ERROR );

      CM_f_lt_zero_1L( temp_float2, FLENS );
      CM_logand_context_with_test();
      /* invert the sign bit of those faces  */
      CM_load_overflow( object.face[ i ].sign_bit );
      CM_invert_overflow();
      CM_store_overflow( object.face[ i ].sign_bit );

      /* now multiply normal vector by temp_float2 */
      CM_load_context( feature );
      multiply_vector( object.face[ i ].normal_vector,
                       object.face[ i ].normal_vector, temp_float2 );
    }
  }

  CM_deallocate_stack_through( temp );

  return( NO_ERROR );
}       /* of erode_object(); */

/*****************************************************************************/
status_t
make_visual_field( object, bit, config, displacement )
     object_t       object;
     CM_field_id_t  bit;
     config_t       config;
     displacement_t displacement;
{
  int i;
  status_t status;
  coordinates_t temp_vector1, temp_vector2;
  coordinates_t point1, point2, vector1, vector2;
  CM_field_id_t temp_float1, temp_float2;

  temp_float1 = CM_allocate_stack_field( 2*FLEN );
  temp_float2 = CM_SUBFIELD( temp_float1, FLEN );

  /* allocate temporary vectors */
  point1[0] = CM_allocate_stack_field( FPOINT_LEN );
  for( i = 1; i < DIMENSION; i++ )
    point1[i] = CM_SUBFIELD( point1[i-1], FLEN );
  point2[0] = CM_allocate_stack_field( FPOINT_LEN );
  for( i = 1; i < DIMENSION; i++ )
    point2[i] = CM_SUBFIELD( point2[i-1], FLEN );
  vector1[0] = CM_allocate_stack_field( FPOINT_LEN );
  for( i = 1; i < DIMENSION; i++ )
    vector1[i] = CM_SUBFIELD( vector1[i-1], FLEN );
  vector2[0] = CM_allocate_stack_field( FPOINT_LEN );
  for( i = 1; i < DIMENSION; i++ )
    vector2[i] = CM_SUBFIELD( vector2[i-1], FLEN );

  /* object.face[ 0 ] */
```

```
/* (closest) right vertical plane */
CM_load_context( bit );
CM_set_bit( object.face[ 0 ].face_bit );
CM_clear_bit( object.face[ 0 ].sign_bit );

CM_f_move_constant_1L( vector1[ 0 ], config.focal.max, FLENS );
CM_f_move_constant_1L( vector1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( vector1[ 2 ], (float)(IMAGE_SIZE-1-CENTER_OFFSET_X), FLENS );
CM_f_add_constant_2_1L( vector1[ 2 ], /* here */ 1.0, FLENS );
CM_f_negate_1_1L( vector1[ 2 ], FLENS );
CM_f_move_constant_1L( point1[ 0 ], -config.baseline.max/2, FLENS );
CM_f_move_constant_1L( point1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 2 ], 0.0, FLENS );

rotate( vector2, vector1, config.pitch.min, displacement.yaw.max );
rotate( point2,  point1,  config.pitch.min, displacement.yaw.max );
add_vector( point2, displacement.translation.vector );
status = compact_face( object.face[ 0 ], point2, vector2, temp_float1, temp_float2 );

if( status != NO_ERROR ) return( status );

/* object.face[ 1 ] */
/* (closest) left vertical plane */
CM_load_context( bit );
CM_set_bit( object.face[ 1 ].face_bit );
CM_set_bit( object.face[ 1 ].sign_bit );

CM_f_move_constant_1L( vector1[ 0 ], config.focal.max, FLENS );
CM_f_move_constant_1L( vector1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( vector1[ 2 ], (float)(CENTER_OFFSET_X-IMAGE_SIZE+1), FLENS );
CM_f_negate_1_1L( vector1[ 2 ], FLENS );
CM_f_move_constant_1L( point1[ 0 ], config.baseline.max/2, FLENS );
CM_f_move_constant_1L( point1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 2 ], 0.0, FLENS );

rotate( vector2, vector1, config.pitch.min, displacement.yaw.min );
rotate( point2,  point1,  config.pitch.min, displacement.yaw.min );
add_vector( point2, displacement.translation.vector );
status = compact_face( object.face[ 1 ], point2, vector2, temp_float1, temp_float2 );

if( status != NO_ERROR ) return( status );

/* object.face[ 2 ] */
/* plane below set passing through origin (yaw yaw.max) */
CM_load_context( bit );
CM_set_bit( object.face[ 2 ].face_bit );
CM_set_bit( object.face[ 2 ].sign_bit );

CM_f_move_constant_1L( point1[ 0 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 2 ], 0.0, FLENS );
/* no need for rotation ... */

CM_f_move_constant_1L( vector1[ 0 ], 0.0, FLENS );
CM_f_move_constant_1L( vector1[ 1 ], config.focal.max, FLENS );
CM_f_move_constant_1L( vector1[ 2 ], (float)(CENTER_OFFSET_Y-IMAGE_SIZE+1), FLENS );
CM_f_negate_1_1L( vector1[ 2 ], FLENS );

rotate( vector2, vector1, config.pitch.min, displacement.yaw.max );
add_vector( point1, displacement.translation.vector );
status = compact_face( object.face[ 2 ], point1, vector2, temp_float1, temp_float2 );

if( status != NO_ERROR ) return( status );

/* object.face[ 3 ] */
/* plane below set passing through origin (yaw yaw.min) */
CM_load_context( bit );
CM_set_bit( object.face[ 3 ].face_bit );
CM_set_bit( object.face[ 3 ].sign_bit );

CM_f_move_constant_1L( point1[ 0 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 2 ], 0.0, FLENS );
```

```
/* no need for rotation ... */

CM_f_move_constant_1L( vector1[ 0 ], 0.0, FLENS );
CM_f_move_constant_1L( vector1[ 1 ], config.focal.max, FLENS );
CM_f_move_constant_1L( vector1[ 2 ], (float)(CENTER_OFFSET_Y-IMAGE_SIZE+1), FLENS );
CM_f_negate_1_1L( vector1[ 2 ], FLENS );

rotate( vector2, vector1, config.pitch.min, displacement.yaw.min );
add_vector( point1, displacement.translation.vector );
status = compact_face( object.face[ 3 ], point1, vector2, temp_float1, temp_float2 );

if( status != NO_ERROR ) return( status );

/* object.face[ 4 ] */
/* plane above set passing through origin (yaw yaw.max) */
CM_load_context( bit );
CM_set_bit( object.face[ 4 ].face_bit );
CM_clear_bit( object.face[ 4 ].sign_bit );

CM_f_move_constant_1L( point1[ 0 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 2 ], 0.0, FLENS );
/* no need for rotation ... */

CM_f_move_constant_1L( vector1[ 0 ], 0.0, FLENS );
CM_f_move_constant_1L( vector1[ 1 ], config.focal.max, FLENS );
CM_f_move_constant_1L( vector1[ 2 ], (float)(IMAGE_SIZE-1-CENTER_OFFSET_Y), FLENS );
CM_f_add_constant_2_1L( vector1[ 2 ], /* here */ 1.0, FLENS );
CM_f_negate_1_1L( vector1[ 2 ], FLENS );

rotate( vector2, vector1, config.pitch.min, displacement.yaw.max );
add_vector( point1, displacement.translation.vector );
status = compact_face( object.face[ 4 ], point1, vector2, temp_float1, temp_float2 );

if( status != NO_ERROR ) return( status );

/* object.face[ 5 ] */
/* plane above set passing through origin (yaw yaw.min) */
CM_load_context( bit );
CM_set_bit( object.face[ 5 ].face_bit );
CM_clear_bit( object.face[ 5 ].sign_bit );

CM_f_move_constant_1L( point1[ 0 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 2 ], 0.0, FLENS );
/* no need for rotation ... */

CM_f_move_constant_1L( vector1[ 0 ], 0.0, FLENS );
CM_f_move_constant_1L( vector1[ 1 ], config.focal.max, FLENS );
CM_f_move_constant_1L( vector1[ 2 ], (float)(IMAGE_SIZE-1-CENTER_OFFSET_Y), FLENS );
CM_f_add_constant_2_1L( vector1[ 2 ], /* here */ 1.0, FLENS );
CM_f_negate_1_1L( vector1[ 2 ], FLENS );

rotate( vector2, vector1, config.pitch.min, displacement.yaw.min );
add_vector( point1, displacement.translation.vector );
status = compact_face( object.face[ 5 ], point1, vector2, temp_float1, temp_float2 );

if( status != NO_ERROR ) return( status );

/* object.face[ 6 ] */
/* closest vertical plane perpendicular to (O,z) (yaw yaw.max) */
CM_load_context( bit );
CM_set_bit( object.face[ 6 ].face_bit );
CM_set_bit( object.face[ 6 ].sign_bit );

CM_f_move_constant_1L( vector1[ 0 ], 0.0, FLENS );
CM_f_move_constant_1L( vector1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( vector1[ 2 ], 1.0, FLENS );

CM_f_move_constant_1L( point1[ 0 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 1 ], 0.0, FLENS );
CM_f_move_constant_1L( point1[ 2 ], config.focal.max * config.baseline.max/
                                    (DISPARITY_MAX+1), FLENS );
```

```
  rotate( vector2, vector1, config.pitch.min, displacement.yaw.max );
  rotate( point2,  point1,  config.pitch.min, displacement.yaw.max );
  add_vector( point2, displacement.translation.vector );
  status = compact_face( object.face[ 6 ], point2, vector2, temp_float1, temp_float2 );

  if( status != NO_ERROR ) return( status );

  /* object.face[ 7 ] */
  /* closest vertical plane perpendicular to (0,z) (yaw yaw.min) */
  CM_load_context( bit );
  CM_set_bit( object.face[ 7 ].face_bit );
  CM_set_bit( object.face[ 7 ].sign_bit );

  CM_f_move_constant_1L( vector1[ 0 ], 0.0, FLENS );
  CM_f_move_constant_1L( vector1[ 1 ], 0.0, FLENS );
  CM_f_move_constant_1L( vector1[ 2 ], 1.0, FLENS );

  CM_f_move_constant_1L( point1[ 0 ], 0.0, FLENS );
  CM_f_move_constant_1L( point1[ 1 ], 0.0, FLENS );
  CM_f_move_constant_1L( point1[ 2 ], config.focal.max * config.baseline.max/
                                      (DISPARITY_MAX+1), FLENS );

  rotate( vector2, vector1, config.pitch.min, displacement.yaw.min );
  rotate( point2,  point1,  config.pitch.min, displacement.yaw.min );
  add_vector( point2, displacement.translation.vector );
  status = compact_face( object.face[ 7 ], point2, vector2, temp_float1, temp_float2 );

  if( status != NO_ERROR ) return( status );

  CM_deallocate_stack_through( temp_float1 );

  /* inactivate the remaining (undefined) faces */
  CM_load_context( bit );
  for( i = 8; i < NB_FACES; i++ )
    CM_clear_bit( object.face[ i ].face_bit );

  return( NO_ERROR );
}     /* of make_visual_field() */

/*****************************************************************************/
status_t
error_message( status )
     status_t status;
{
  char string[ MISC_BUFFER_SIZE ];

  switch ( status ) {
  case NO_ERROR:
    return( status );
  case NO_object:
    sprintf( string, "No object in data file...\n");
    break;
  case CANNOT_OPEN_LEFT_DATA:
    sprintf( string, "Cannot open left data file...\n" );
    break;
  case CANNOT_OPEN_RIGHT_DATA:
    sprintf( string, "Cannot open right data file...\n" );
    break;
  case DATA_WRITING_ERROR:
    sprintf( string, "An error occured when trying to write objects into data file '%s'.
    break;
  case SINGULARITY_ERROR:
    sprintf( string, "%s\n%s\n",
            "WARNING: a singularity was detected when computing the compact representati
            "Please translate the origin to another location.");
    break;
  default:
    sprintf( string, "WARNING: an error occured!\n");
    break;
  }
  AVSmessage( file_version, AVS_Fatal, AVSmodule, "stereo", NULL, "%s", string );
```

```
  return( status );

}       /* end of error_message */

/*****************************************************************************/
static stereo_compute( left_image, right_image, file_output,
                       data_filename,
                       translation_x, translation_y, translation_z,
                       positional_precision, yaw_min, yaw_max, focal_min, focal_max,
                       baseline_min, baseline_max, pitch_min, pitch_max,
                       correlation_threshold, feature_extractor, extractor_threshold,
                       visualization )
     AVSfield_char *left_image, *right_image;
     char **file_output;
     char *data_filename;
     float *translation_x, *translation_y, *translation_z, *positional_precision;
     float *yaw_min, *yaw_max, *focal_min, *focal_max;
     float *baseline_min, *baseline_max, *pitch_min, *pitch_max;
     float *correlation_threshold, *extractor_threshold;
     char *feature_extractor;
     boolean visualization;
{
  int i,j,k,l;
  status_t STATUS = NO_ERROR;
  unsigned image_dim[2];
  unsigned stereo_pair_dim[3];
  unsigned square_dim[2];
  unsigned geom_dim_1D;
  CM_field_id_t color_field;
  CM_field_id_t integer_image_pair, square_image, image_coord[2];
  CM_field_id_t feature, disp;          /* for square_grid in square_geom */
  CM_field_id_t send_address;
  object_t object, object_list;
  CM_field_id_t object_bit;
  displacement_t displacement;
  config_t       config;
  char string[ MISC_BUFFER_SIZE ];

  if( AVSparameter_changed( VISUALIZATION_PARAM ) )
    DISPLAY_INITIALIZED = NO;

  VISUALIZATION = visualization;

  printf("\nWarm booting the CM...");fflush(stdout);
  CM_init();
  printf(" done\n\n");fflush(stdout);

  CM_set_safety_mode( SAFETY_MODE );

  stereo_pair_dim[0] = IMAGE_SIZE;
  stereo_pair_dim[1] = IMAGE_SIZE;
  stereo_pair_dim[2] = 2;
  image_dim[0] = 2*IMAGE_SIZE;
  image_dim[1] = IMAGE_SIZE;
  square_dim[0] = IMAGE_SIZE;
  square_dim[1] = IMAGE_SIZE;

  /* create geometries, allocate vp-set and CM memory fields */
  stereo_pair_geom = CM_create_geometry( stereo_pair_dim, 3 );
  stereo_pair_grid = CM_allocate_vp_set( stereo_pair_geom );
  image_geom = CM_create_geometry( image_dim, 2 );
  image_grid = CM_allocate_vp_set( image_geom );
  square_geom = CM_create_geometry( square_dim, 2 );
  square_grid = CM_allocate_vp_set( square_geom );

  geom_dim_1D = IMAGE_SIZE*IMAGE_SIZE;
  geometry_1D = CM_create_geometry( &geom_dim_1D, 1 );

  list = CM_physical_vp_set();

  /* initialize displacement structure for future use */
  displacement.translation.vector[ 0 ] = *translation_x;
  displacement.translation.vector[ 1 ] = *translation_y;
```

```
  displacement.translation.vector[ 2 ] = *translation_z;
  displacement.translation.imprecision = *positional_precision;
  displacement.yaw.min = *yaw_min;
  displacement.yaw.max = *yaw_max;

  /* initialize the configuration structure for future use */
  config.focal.min = *focal_min;
  config.focal.max = *focal_max;
  config.baseline.min = *baseline_min;
  config.baseline.max = *baseline_max;
  config.pitch.min = *pitch_min;
  config.pitch.max = *pitch_max;

  if( !DISPLAY_INITIALIZED )
    initialize_display();

  CM_set_vp_set( image_grid );
  image0 = CM_allocate_stack_field( BITS_PER_PIXEL );
  image1 = CM_allocate_stack_field( BITS_PER_PIXEL );
  CM_u_move_zero_always_1L( image0, BITS_PER_PIXEL );
  image_bit = CM_allocate_stack_field( 1 );
  CM_clear_bit_always( image_bit );

    CM_set_vp_set( stereo_pair_grid );
    stereo_pair = CM_allocate_stack_field( FLEN );
    stereo_pair_mean = CM_allocate_stack_field( FLEN );
    stereo_pair_norm = CM_allocate_stack_field( FLEN );
    dual_stereo_pair_info = CM_allocate_stack_field( 3*FLEN );
    dual_stereo_pair = CM_SUBFIELD( dual_stereo_pair_info, 0 );
    dual_stereo_pair_mean = CM_SUBFIELD( dual_stereo_pair_info, FLEN );
    dual_stereo_pair_norm = CM_SUBFIELD( dual_stereo_pair_info, 2*FLEN );
    feature_bit = CM_allocate_stack_field( 1 );
    color_field = CM_allocate_stack_field( BITS_PER_PIXEL );
    CM_u_move_const_always_1L( color_field, INTENSITY_MAX, BITS_PER_PIXEL );

    coords = CM_allocate_stack_field( 1+2*IMAGE_SIZE_LEN ) ;
    coord[ 0 ] = CM_SUBFIELD( coords, 0 );
    coord[ 1 ] = CM_SUBFIELD( coords, IMAGE_SIZE_LEN );
    coord[ 2 ] = CM_SUBFIELD( coords, 2*IMAGE_SIZE_LEN );
    CM_set_context();
    CM_my_news_coordinate_1L( coord[ 0 ], /* axis */ 0, IMAGE_SIZE_LEN );
    CM_my_news_coordinate_1L( coord[ 1 ], /* axis */ 1, IMAGE_SIZE_LEN );
    CM_my_news_coordinate_1L( coord[ 2 ], /* axis */ 2, 1 );

    disparity = CM_allocate_stack_field( DISPARITY_LEN );
    correlation = CM_allocate_stack_field( FLEN );

    temp = CM_allocate_stack_field( 3*FLEN );
    temp_float1 = CM_SUBFIELD( temp, 0 );
    temp_float2 = CM_SUBFIELD( temp, FLEN );
    temp_float3 = CM_SUBFIELD( temp, 2*FLEN );

    STATUS = load_avs_image_pair( left_image, right_image, stereo_pair );

  if (error_message( STATUS ) != NO_ERROR ) {
    AVSmark_output_unchanged( CM_FIELD_OUTPUT );
    return( 0 );
  }

    /* show stereo pair images */
    CM_set_vp_set( stereo_pair_grid );
    integer_image_pair = CM_allocate_stack_field( BITS_PER_PIXEL );
    CM_u_f_floor_2_2L( integer_image_pair, stereo_pair, BITS_PER_PIXEL, FLENS );
    convert_3D_to_2D( integer_image_pair, image0 );
    CM_deallocate_stack_through( integer_image_pair );
    write_to_display( image0 );

    extract_features( stereo_pair, feature_bit,
                      (feature_extractor_t)AVSchoice_number( FEATURE_EXTRACTOR_PARAM,
                                                             feature_extractor ),
                      *extractor_threshold );

    CM_set_vp_set( image_grid );
```

```
    CM_u_move_always_1L( image1, image0, BITS_PER_PIXEL );
    show_features( feature_bit, image1, color_field );
    write_to_display( image1 );

    normalize( stereo_pair, stereo_pair_mean, stereo_pair_norm );

    /* make dual image stereo_pair */
    CM_send_to_news_always_1L( dual_stereo_pair, stereo_pair,
                               /* axis */ 2, CM_upward, FLEN );
    CM_send_to_news_always_1L( dual_stereo_pair_mean, stereo_pair_mean,
                               /* axis */ 2, CM_upward, FLEN );
    CM_send_to_news_always_1L( dual_stereo_pair_norm, stereo_pair_norm,
                               /* axis */ 2, CM_upward, FLEN );
    /* TRICKY! : upward or downward have the same effect as axis 2 has only
       2 coordinates */

    alignment = CM_allocate_stack_field( ALIGNMENT_LEN );

    match_features();

    threshold_correlation( *correlation_threshold );

    CM_set_vp_set( image_grid );
    CM_u_move_always_1L( image1, image0, BITS_PER_PIXEL );
    show_features( feature_bit, image1, color_field );
    show_disparity_vectors( image1, color_field );
    write_to_display( image1 );

    merge();

    CM_set_vp_set( stereo_pair_grid );
    CM_set_context();
    CM_f_multiply_const_always_3_1L( temp_float1, correlation,
                                     (float) INTENSITY_MAX, FLENS );
    CM_u_f_round_2_2L( color_field, temp_float1, BITS_PER_PIXEL, FLENS );
    CM_load_context( coord[2] );
    CM_u_move_constant_1L( color_field, INTENSITY_MAX, BITS_PER_PIXEL );

    CM_set_vp_set( image_grid );
    CM_u_move_always_1L( image1, image0, BITS_PER_PIXEL );
    CM_load_context( image_bit );
    CM_u_move_zero_1L( image1, BITS_PER_PIXEL );

    show_features( feature_bit, image1, color_field );
    show_disparity_vectors( image1, color_field );

    CM_deallocate_stack_through( alignment );

    write_to_display( image1 );

    CM_set_vp_set( square_grid );
    feature = CM_allocate_heap_field( 1 );  /* feature bit in square_geom */
    CM_clear_bit_always( feature );
    disp = CM_allocate_heap_field( DISPARITY_LEN );
    CM_set_vp_set( stereo_pair_grid );
    /* send feature and disparity information contained in left image
       to square_grid */
    CM_make_news_coordinate_1L(    square_geom, temp, 0, coord[0], IMAGE_SIZE_LEN );
    CM_deposit_news_coordinate_1L( square_geom, temp, 1, coord[1], IMAGE_SIZE_LEN );
    CM_load_context( coord[2] );
    CM_invert_context();                /* select image 0 (left) */
    CM_logand_context( feature_bit );
    CM_send_1L( disp, temp, disparity, DISPARITY_LEN, feature );

    CM_deallocate_stack_through( temp );

    /* set default vp_set, which will be the only one remaining */
    CM_set_vp_set( square_grid );

    /* now deallocate all the useless fields */
    CM_deallocate_stack_through( image0 );
    /* now only disp and feature are still allocated */
```

```
    allocate_object( &object );
    STATUS = make_object( object, feature, disp, config, displacement );

    /* now dilate the computed object so as to take translation
       imprecision into account */

    if( STATUS == NO_ERROR )
      STATUS = dilate_object( object, displacement.translation.imprecision, feature );

    /* enumerate all active processors */
    CM_set_vp_set_geometry( square_grid, geometry_1D ); /* switch to 1D geometry */
    temp = CM_allocate_stack_field( CM_physical_processors_length );
    CM_load_context( feature );
    CM_enumerate_1L( temp, /* axis */ 0, CM_physical_processors_length,
                     CM_upward, CM_inclusive, CM_none, CM_no_field );

    CM_set_vp_set( list );
    allocate_object( &object_list );
    object_bit = CM_allocate_stack_field( 1 );
    CM_clear_bit_always( object_bit );

    CM_u_write_to_processor( /* address */ 0, object_bit, 1, 1 );

    if( STATUS == NO_ERROR )
      STATUS = make_visual_field( object_list, object_bit, config, displacement );

    if( STATUS == NO_ERROR )
      STATUS = erode_object( object_list, displacement.translation.imprecision,
                             object_bit );

    CM_set_vp_set( square_grid );
    CM_load_context( feature );
    CM_send_1L( object_list.total, temp, object.total, OBJECT_LEN, object_bit );

    CM_set_vp_set( list );

    /* do not forget the visual field stored at address 0 ! */
    CM_u_write_to_processor( /* address */ 0, object_bit, 1, 1 );

    if( STATUS == NO_ERROR )
      STATUS = write_to_dv( data_filename, object_list.total, OBJECT_LEN, object_bit );

    deallocate_object( object_list );

    CM_set_vp_set( square_grid );
    deallocate_object( object );
    CM_deallocate_stack_through( temp );

    CM_set_vp_set_geometry( square_grid, square_geom ); /* switch back to 2D  */

  if (error_message( STATUS ) != NO_ERROR ) {
    AVSmark_output_unchanged( CM_FIELD_OUTPUT );
    return( 0 );
  }

  fflush( stdout );

  sprintf( string, "%s", data_filename );

  *file_output = string;

  /* indicate success */
  return(1);

}       /* of compute() */

/*****************************************************************************/
static
stereo_init()
{
  /* indicate success */
  return(1);
} /* of init() */
```

```
/*****************************************************************************/
static
stereo_destroy()
{
  fflush( stdout );      fflush( stderr );

  if( DISPLAY_INITIALIZED )
    clear_display();

  /* indicate success */
  return(1);
} /* of destroy() */
```

```
#define PROGRAM_NAME "avs_tree"

/*****************************************************************/
/*                                                               */
/*                        AVS module                             */
/*                                                               */
/*  Computes the octree representation of 3D objects (polyhedra) */
/*                 stored in a front end file                    */
/*                                                               */
/*                        Ph. Quinio  (25/02/92)                 */
/*                                                               */
/*****************************************************************/

/* include files */
#include <stdio.h>
#include <math.h>
#include <strings.h>
#include <cm/paris.h>
#include <cm/display.h>
#include <cm/cmsr.h>
#include <cm/cmfb.h>
#include <cm/cm_file.h>
#include <cm/cm_errno.h>
#include <avs/avs.h>
#include <avs/field.h>
#include <avs/flow.h>
#include <my_functions_and_macros.h>
#include <fcntl.h>
#include <signal.h>

/* Used for AVSmessage() */
static char file_version[] = "@(#)avs_tree.c    Quinio 92/03/04";

/* the following is an abbreviation... */
#define CM_SUBFIELD( field, offset ) CM_add_offset_to_field_id( field, offset )

/* CM safety mode */
#define SAFETY_MODE     NO

/* constants */
#define DIMENSION       3
#define POWER_2_DIMENSION 8                     /* = power( 2, DIMENSION ) */

#define COUNTER_LEN     6       /* power( 2, COUNTER_LEN ) is the maximum number of
                                   image pairs that can be processed in the tree */
#define LEVEL_NB_MAX    7
#define NB_FACES        14      /* number of faces in data polyhedra */
#define FACES_PER_NODE  2       /* >= NB_FACES/POWER_2_DIMENSION
                                   This allows more faces to be used ber polyhedron */

#define POINT_LEN       (DIMENSION * FLEN)  /* size of a point on space */
#define COORD_LEN       22
#define ADDRESS_LEN     23
#define CUBE_LEN        (DIMENSION * FLEN + DIMENSION * FLEN)  /* size of a cube */
#define FACE_LEN        (POINT_LEN + 2)  /* size of a face */
#define OBJECT_LEN      NB_FACES * FACE_LEN  /* size of an object */
#define SPLIT_OBJECT_LEN (FACES_PER_NODE * FACE_LEN)
#define NODE_LEN        (LEVEL_NB_MAX + 2*COUNTER_LEN + COORD_LEN + CUBE_LEN)
                        /* total size of a node */

#define INFO_LEN        (2*COUNTER_LEN + COORD_LEN)

/* miscellaneous */
#define TEMP_LEN        3*FLEN

/* Structures and general data types */
typedef enum { NO_ERROR, NO_OBJECT,
               CANNOT_OPEN, CANNOT_OPEN_DATA, DATA_ERROR } Status_t;

typedef CM_field_id_t coordinates_t[ DIMENSION ];

typedef struct {
  CM_field_id_t         total;
```

```
  coordinates_t          origin;
  coordinates_t          size;
} cube_t;

typedef struct {
  CM_field_id_t          total;
  coordinates_t          normal_vector;
  CM_field_id_t          sign_bit;
  CM_field_id_t          face_bit;
} face_t;

typedef struct {
  CM_field_id_t          total;
  face_t                 face[ FACES_PER_NODE ];
} split_object_t;

typedef struct {
  CM_field_id_t          total;
  face_t                 face[ NB_FACES ];
} object_t;

typedef struct {
  CM_field_id_t          total;
  CM_field_id_t          level_mask;
  CM_field_id_t          visibility_count;
  CM_field_id_t          hit_count;
  CM_field_id_t          children;
  CM_field_id_t          parent;
  cube_t                 cube;
} node_t;

typedef struct {
  CM_field_id_t          total;
  CM_field_id_t          hit_count;
  CM_field_id_t          visibility_count;
  CM_field_id_t          children;
} tree_info_t;

/* front_end structures and types */
typedef float fe_coordinates_t[ DIMENSION ];

typedef struct {
  fe_coordinates_t       origin;
  fe_coordinates_t       size;
} fe_cube_t;

typedef struct {
  fe_coordinates_t       normal_vector;
  int                    sign_bit;
  int                    face_bit;
} fe_face_t;

typedef struct {
  fe_face_t              face[ NB_FACES ];
} fe_object_t;

typedef struct {
  fe_coordinates_t       origin;
  fe_coordinates_t       size;
  unsigned               hit_count;
  unsigned               visibility_count;
} fe_node_t;

/* global variables */

CM_geometry_id_t geometry_1D, tree_geom;
CM_geometry_id_t image_geom;

CM_vp_set_id_t tree_grid;
CM_vp_set_id_t image_grid;
CM_vp_set_id_t list;    /* for object list */

unsigned      image_dim[2];
```

```
node_t          node;
object_t        object_list;
split_object_t object, visual;
face_t          working_face;

tree_info_t     tree_info;

CM_field_id_t temp,      /* temporary fields */
               temp_float0, temp_float1,
               temp_bit0, temp_bit1, temp_bit2, temp_bit3, temp_bit4;
CM_field_id_t ImplyBit_visual, HitBit_visual;
CM_field_id_t ImplyBit_object, HitBit_object;
CM_field_id_t coord;     /* 0-coordinate of tree nodes */
CM_field_id_t child;     /* 1-coordinate of tree nodes */

unsigned        LEVEL_MAX_MASK;
unsigned        tree_size;

unsigned        free_proc_addr;      /* 0-coordinate of zone of free processors */

unsigned        a_processor[ LEVEL_NB_MAX+1 ];     /* 0-coordinate of a processor
                                                      at a given level */
unsigned        object_index;    /* index for objects */

#define DEFAULT_PATH "."

/* define AVS parameter names (titles) and default values */
#define DATA_INPUT "Data file"
#define DV_TREE_OUTPUT "Tree file on DV"
#define FE_TREE_OUTPUT "Tree file on FE"
#define ORIGIN_X_OUTPUT "Origin x output"
#define ORIGIN_Y_OUTPUT "Origin y output"
#define ORIGIN_Z_OUTPUT "Origin z output"
#define SIZE_X_OUTPUT "Size x output"
#define SIZE_Y_OUTPUT "Size y output"
#define SIZE_Z_OUTPUT "Size z output"
#define TREE_DEPTH_PARAM "Tree depth"
#define DEFAULT_TREE_DEPTH 6
#define WRITE_FE_PARAM "Write on front end"
#define DEFAULT_WRITE_FE YES
#define TREE_FILENAME_PARAM "Tree filename"
#define DEFAULT_TREE_FILENAME "/data/trees/tree.tree"
#define DONE_PARAM "Output tree"
#define CLEAR_TREE_PARAM "Clear tree"
#define ORIGIN_X_PARAM "Origin x"
#define ORIGIN_Y_PARAM "Origin y"
#define ORIGIN_Z_PARAM "Origin z"
#define DEFAULT_ORIGIN -0.5
#define SIZE_X_PARAM "Size x"
#define SIZE_Y_PARAM "Size y"
#define SIZE_Z_PARAM "Size z"
#define DEFAULT_SIZE 0.5
#define MAX_SIZE 10000.0
#define VPR_PARAM "VP ratio"
#define DEFAULT_VPR "VP ratio 64"
#define VPR_CHOICES "VP ratio 16,VP ratio 32,VP ratio 64,VP ratio 128"
#define DELIMITER ","

char DV_PATH[ MISC_BUFFER_SIZE ] = DEFAULT_PATH;
char FE_PATH[ MISC_BUFFER_SIZE ] = DEFAULT_PATH;

boolean INITIALIZED = NO;

/*****************************************************************************/
AVSinit_modules()
{
    int tree();

    AVSmodule_from_desc( tree );
}

/*****************************************************************************/
```

```
static
tree()
{
    int param0, param1, param2, param3, param4, param5;
    int param6, param7, param8, param9, param10, param11;
    int output1, output2, output3, output4, output5, output6;
    int tree_compute(), tree_init(), tree_destroy();
    char DEFAULT_FILENAME[ MISC_BUFFER_SIZE ];
    char *ptr;

    /* Set the module name */
    AVSset_module_name( "octree CM", MODULE_MAPPER );

    /* get the path to home directory on DataVault */
    ptr = (char *)getenv( "DVWD" );
    if( ptr != NULL )
      strcpy( DV_PATH, ptr );

    /* get the path to home directory on Front End */
    ptr = (char *)getenv( "HOME" );
    if( ptr != NULL )
      strcpy( FE_PATH, ptr );

    /* Create the input port */
    AVScreate_input_port( DATA_INPUT, "string", REQUIRED );

    /* Create the output port */
    AVScreate_output_port( DV_TREE_OUTPUT, "string" );
    AVScreate_output_port( FE_TREE_OUTPUT, "string" );
    output1 = AVScreate_output_port( ORIGIN_X_OUTPUT, "real" );
    output2 = AVScreate_output_port( ORIGIN_Y_OUTPUT, "real" );
    output3 = AVScreate_output_port( ORIGIN_Z_OUTPUT, "real" );
    output4 = AVScreate_output_port( SIZE_X_OUTPUT, "real" );
    output5 = AVScreate_output_port( SIZE_Y_OUTPUT, "real" );
    output6 = AVScreate_output_port( SIZE_Z_OUTPUT, "real" );

    /* make some output ports invisible by default */
    AVSset_output_flags( output1, INVISIBLE );
    AVSset_output_flags( output2, INVISIBLE );
    AVSset_output_flags( output3, INVISIBLE );
    AVSset_output_flags( output4, INVISIBLE );
    AVSset_output_flags( output5, INVISIBLE );
    AVSset_output_flags( output6, INVISIBLE );

    /* Add parameter */
    param0 = AVSadd_parameter( VPR_PARAM, "choice", DEFAULT_VPR,
                               VPR_CHOICES, DELIMITER );
    AVSconnect_widget(param0, "radio_buttons");
    param1 = AVSadd_parameter( TREE_DEPTH_PARAM, "integer", DEFAULT_TREE_DEPTH,
                               2, LEVEL_NB_MAX );
    AVSconnect_widget(param1, "idial");
    param2 = AVSadd_parameter( TREE_FILENAME_PARAM, "string", DEFAULT_TREE_FILENAME,
                               "", ".tree" );
    AVSconnect_widget(param2, "typein");
    param3 = AVSadd_parameter( WRITE_FE_PARAM, "boolean", DEFAULT_WRITE_FE,
                               /* ignored */0, /* ignored */0 );
    AVSconnect_widget(param3, "toggle");
    param4 = AVSadd_parameter( DONE_PARAM, "boolean", NO,
                               /* ignored */0, /* ignored */0 );
    AVSconnect_widget(param4, "toggle");
    param5 = AVSadd_parameter( CLEAR_TREE_PARAM, "boolean", NO,
                               /* ignored */0, /* ignored */0 );
    AVSconnect_widget(param5, "toggle");
    param6 = AVSadd_float_parameter( ORIGIN_X_PARAM, DEFAULT_ORIGIN,
                                     FLOAT_UNBOUND, FLOAT_UNBOUND );
    AVSconnect_widget(param6, "typein_real");
    param7 = AVSadd_float_parameter( ORIGIN_Y_PARAM, DEFAULT_ORIGIN,
                                     FLOAT_UNBOUND, FLOAT_UNBOUND );
    AVSconnect_widget(param7, "typein_real");
    param8 = AVSadd_float_parameter( ORIGIN_Z_PARAM, DEFAULT_ORIGIN,
                                     FLOAT_UNBOUND, FLOAT_UNBOUND );
    AVSconnect_widget(param8, "typein_real");
    param9 = AVSadd_float_parameter( SIZE_X_PARAM, DEFAULT_SIZE, 0.0, MAX_SIZE );
```

```
    AVSconnect_widget(param9, "typein_real");
    param10 = AVSadd_float_parameter( SIZE_Y_PARAM, DEFAULT_SIZE, 0.0, MAX_SIZE );
    AVSconnect_widget(param10, "typein_real");
    param11 = AVSadd_float_parameter( SIZE_Z_PARAM, DEFAULT_SIZE, 0.0, MAX_SIZE );
    AVSconnect_widget(param11, "typein_real");

    /* Set the (optional) initialization procedure */
    AVSset_init_proc( tree_init );

    /* Set the compute procedure */
    AVSset_compute_proc( tree_compute );

    /* Set the (optional) destroy procedure */
    AVSset_destroy_proc( tree_destroy );
}

/*************************************************************************/
Status_t
read_from_dv( data_filename, object, length, nb_objects )
     char *data_filename;
     object_t object;
     unsigned length;
     int *nb_objects;
{
  int input_fd;
  CM_field_id_t bit;

  input_fd = CMFS_open( data_filename, CMFS_O_RDONLY );
  if( input_fd < 0 ) {
    CMFS_perror( data_filename );
    return( CANNOT_OPEN_DATA );
  }

  printf("\nreading objects from DataVault file '%s'... ", data_filename );
  fflush( stdout );

  if( CMFS_read_file_always( input_fd, object.total, length ) != length ) {
    CMFS_perror( data_filename );
    CMFS_close( input_fd );
    return( DATA_ERROR );
  }

  bit = CM_allocate_stack_field( 1 );

  if( CMFS_read_file_always( input_fd, bit, 1 ) != 1 ) {
    CMFS_perror( data_filename );
    CMFS_close( input_fd );
    return( DATA_ERROR );
  }

  *nb_objects = CM_global_count_bit_always( bit );
  CM_deallocate_stack_through( bit );

  CMFS_close( input_fd );
  printf("done.\n");fflush( stdout );

  return( NO_ERROR );
}       /* of read_from_dv */

/*************************************************************************/
Status_t
read_from_fe( filename, object, nb_objects )
     char *filename;
     object_t object;
     int *nb_objects;
{
#define NB_OBJECTS_MAX 5000

  int input_fd;
  fe_object_t buffer[ NB_OBJECTS_MAX ];
  int i, j, n;
  int fe_offset, cm_start, cm_end, cm_axis, fe_dim;
  float float_buffer[ NB_OBJECTS_MAX ];
```

```
  unsigned unsigned_buffer[ NB_OBJECTS_MAX ];

  input_fd = open( filename, O_RDONLY );
  if( input_fd < 0 ) {
    perror( filename );
    return( CANNOT_OPEN_DATA );
  }

  printf("\nreading objects from file '%s'... ", filename );

  n = read( input_fd, buffer, NB_OBJECTS_MAX * sizeof( fe_object_t ) );

  close( input_fd );

  if( n <= 0 ) {
    perror( filename );
    return( DATA_ERROR );
  }

  n /= sizeof( fe_object_t );

  /* put buffer on CM */
  fe_offset = 0;  cm_start = 0;  cm_end = n;  cm_axis = 0;  fe_dim = n;
  for( i = 0; i < NB_FACES; i++ ) {
    for( j = 0; j < n; j++ ) {
      float_buffer[ j ] = buffer[ j ].face[ i ].normal_vector[ 0 ];
    }
    CM_f_write_to_news_array_1L( float_buffer, &fe_offset, &cm_start,
                                 &cm_end, &cm_axis,
                                 object.face[ i ].normal_vector[ 0 ], FLENS,
                                 1, &fe_dim,
                                 CM_float_single );
    for( j = 0; j < n; j++ ) {
      float_buffer[ j ] = buffer[ j ].face[ i ].normal_vector[ 1 ];
    }
    CM_f_write_to_news_array_1L( float_buffer, &fe_offset, &cm_start,
                                 &cm_end, &cm_axis,
                                 object.face[ i ].normal_vector[ 1 ], FLENS,
                                 1, &fe_dim,
                                 CM_float_single );
    for( j = 0; j < n; j++ ) {
      float_buffer[ j ] = buffer[ j ].face[ i ].normal_vector[ 2 ];
    }
    CM_f_write_to_news_array_1L( float_buffer, &fe_offset, &cm_start,
                                 &cm_end, &cm_axis,
                                 object.face[ i ].normal_vector[ 2 ], FLENS,
                                 1, &fe_dim,
                                 CM_float_single );
    for( j = 0; j < n; j++ ) {
      unsigned_buffer[ j ] = buffer[ j ].face[ i ].face_bit;
    }
    CM_u_write_to_news_array_1L( unsigned_buffer, &fe_offset, &cm_start,
                                 &cm_end, &cm_axis,
                                 object.face[ i ].face_bit, 1,
                                 1, &fe_dim,
                                 CM_32_bit );
    for( j = 0; j < n; j++ ) {
      unsigned_buffer[ j ] = buffer[ j ].face[ i ].sign_bit;
    }
    CM_u_write_to_news_array_1L( unsigned_buffer, &fe_offset, &cm_start,
                                 &cm_end, &cm_axis,
                                 object.face[ i ].sign_bit, 1,
                                 1, &fe_dim,
                                 CM_32_bit );
  }

  printf("done.\n");
  *nb_objects = n;

  return( NO_ERROR );
}       /* of read_from_fe() */

/*****************************************************************************/
```

```
/* broadcast an object to the processors of a given level */
broadcast_object ( object, object_index, level )
     split_object_t object;
     unsigned object_index;      /* object index in list */
     unsigned level;             /* level at which to broadcast */
{
  int i;

  /* broadcast */
  for( i = 0; i < NB_FACES; i++ ) {
    /* select nodes of level */
    CM_load_context( CM_SUBFIELD( node.level_mask, level-1 ) );
    CM_u_eq_constant_1L( child, (unsigned) (i/FACES_PER_NODE), DIMENSION );
    CM_logand_context_with_test();

    CM_spread_from_processor_1L( object.face[ (unsigned) (i%FACES_PER_NODE) ].total,
                                 object_index, object_list.face[ i ].total,
                                 FACE_LEN );
  }
}       /* end of broadcast_object() */

/*****************************************************************************/
imply_hit( the_object, the_HitBit, the_ImplyBit )
     split_object_t the_object;
     CM_field_id_t the_HitBit, the_ImplyBit;
{
  unsigned i,j,k,l;

  CM_set_context();

  CM_set_bit_always( temp_bit2 );
  CM_set_bit_always( temp_bit3 );

  for( k = 0; k < NB_FACES; k++ ) {

    CM_u_move_1L( working_face.total, the_object.face[ k % FACES_PER_NODE ].total,
                  FACE_LEN );
    CM_spread_with_copy_1L( working_face.total, working_face.total,
                            /* axis */ 1, FACE_LEN,
                            (unsigned)(k / FACES_PER_NODE) );

    /* initialize temp_float0 */
    CM_f_multiply_always_3_1L( temp_float0,
                               working_face.normal_vector[0],
                               working_face.normal_vector[0], FLENS );

    /* multiply-add all coordinates of the normal vector of face k
       with themselves - square norm of the normal-  */
    for( j = 1; j < DIMENSION; j++ )
      CM_f_mult_add_always_1L( temp_float0,
                               working_face.normal_vector[j],
                               working_face.normal_vector[j],
                               temp_float0, FLENS );

    /* multiply-subf all coordinates of the normal vector of face k
       with origin of node-cube */
    for( j = 0; j < DIMENSION; j++ )
      CM_f_mult_subf_always_1L( temp_float0,
                                node.cube.origin[j],
                                working_face.normal_vector[j],
                                temp_float0, FLENS );

    CM_clear_bit_always( temp_bit0 );
    CM_set_bit_always( temp_bit1 );

    for( i = 0; i < POWER_2_DIMENSION; i++ ) {

      /* initialize temp_float1 */
      CM_f_move_always_1L( temp_float1, temp_float0, FLENS );

      for( j = 0; j < DIMENSION; j++ )
        if( EXTRACT_BIT( i, j ) )
          CM_f_mult_subf_always_1L( temp_float1,
```

```
                                        working_face.normal_vector[j],
                                        node.cube.size[j], temp_float1, FLENS );

      CM_f_lt_zero_1L( temp_float1, FLENS );
      CM_store_test_always( temp_bit4 );
      CM_logeqv_always_2_1L( temp_bit4, working_face.sign_bit, 1 );

      CM_f_eq_zero_1L( temp_float1, FLENS );
      CM_logior_test_always( temp_bit4 );
      CM_store_test_always( temp_bit4 );

      CM_logand_always_2_1L( temp_bit1, temp_bit4, 1 );
      CM_logior_always_2_1L( temp_bit0, temp_bit4, 1 );

    }                                                   /* loop on i */

    /* get rid of the result if face_bit is 0, else keep result unchanged */
    CM_logorc2_always_2_1L( temp_bit0, working_face.face_bit, 1 );
    CM_logorc2_always_2_1L( temp_bit1, working_face.face_bit, 1 );

    /* combine the results of all faces */
    CM_logand_always_2_1L( temp_bit2, temp_bit0, 1 );
    CM_logand_always_2_1L( temp_bit3, temp_bit1, 1 );

  }                                                     /* loop on face k */

  CM_u_ne_zero_1L( node.level_mask, LEVEL_NB_MAX );
  CM_logand_context_with_test();

  CM_logior_2_1L( the_HitBit, temp_bit2, 1 );
  CM_logior_2_1L( the_ImplyBit, temp_bit3, 1 );

}       /* end of imply_hit() */

/*****************************************************************************/
update_counters( update_level_mask )
     unsigned update_level_mask;
{
  /* select all levels in update_level_mask */
  CM_set_context();
  CM_logand_constant_3_1L( temp, node.level_mask, update_level_mask, LEVEL_NB_MAX );
  CM_u_ne_zero_1L( temp, LEVEL_NB_MAX );
  CM_logand_context_with_test();

  /* update counters and clear bits */
  CM_u_add_3_3L( node.visibility_count, node.visibility_count, ImplyBit_visual,
                 COUNTER_LEN, COUNTER_LEN, 1 );

  CM_logand_2_1L( HitBit_object, ImplyBit_visual, 1 );

  CM_u_add_3_3L( node.hit_count, node.hit_count, HitBit_object,
                 COUNTER_LEN, COUNTER_LEN, 1 );

  CM_clear_bit( ImplyBit_object );
  CM_clear_bit( HitBit_object );
  CM_clear_bit( ImplyBit_visual );
  CM_clear_bit( HitBit_visual );

}       /* end of update_counters */

/*****************************************************************************/
Status_t
process_object( update_level_mask, tree_depth )
     unsigned update_level_mask;
     int tree_depth;

{
  unsigned new_proc_nb;
  int i, j;

  if( update_level_mask ) update_counters( update_level_mask );

  imply_hit( visual, HitBit_visual, ImplyBit_visual );
```

```
imply_hit( object, HitBit_object, ImplyBit_object );

new_proc_nb = tree_size - free_proc_addr;

/* if there are some processors free, then see if we can use them for
   splitting nodes */
if( new_proc_nb ) {
  /* select all the nodes that need to be split:
     (ImplyBit_visual = 1) && (ImplyBit_object = 0) && (HitBit_object = 1) or
     (ImplyBit_visual = 0) && (HitBit_visual = 1)   */

  CM_set_context();
  CM_load_test_always( ImplyBit_object );
  CM_invert_test_always();
  CM_logand_test_always( HitBit_object );
  CM_logand_test_always( ImplyBit_visual );
  CM_invert_test_always();
  CM_logand_context_with_test();
  CM_load_test_always( ImplyBit_visual );
  CM_invert_test_always();
  CM_logand_test_always( HitBit_visual );
  CM_invert_test_always();
  CM_logand_context_with_test();
  CM_invert_context();

  /* keep only the nodes that do not have children yet  */
  CM_u_eq_zero_1L( node.children, COORD_LEN );
  CM_logand_context_with_test();

  /* keep only the nodes whose level < tree_depth */
  CM_load_test( CM_SUBFIELD( node.level_mask, (unsigned)(tree_depth-1) ) );
  CM_invert_test();
  CM_logand_context_with_test();

  printf("[%u] Number of nodes to be split:  %u\n", object_index,
         CM_global_count_context() );fflush(stdout);

  /* if there are some nodes to be split ... */
  if( CM_global_logior_context() ) {

    /* enumerate all active processors and store their number */
    CM_set_vp_set_geometry( tree_grid, geometry_1D ); /* --> 1D geometry */
    CM_enumerate_1L( node.children, /* axis */ 0, COORD_LEN,
                     CM_upward, CM_inclusive, CM_none, CM_no_field );
    new_proc_nb = MIN( new_proc_nb, CM_global_u_max_1L( node.children,
                                                        COORD_LEN ) );
    CM_u_le_constant_1L( node.children, new_proc_nb, COORD_LEN );
    CM_logand_context_with_test();
    CM_set_vp_set_geometry( tree_grid, tree_geom ); /* switch back to the
                                                       original 2D geometry */

    /* compute 0-coordinate of the children */
    CM_u_add_constant_2_1L( node.children, free_proc_addr-1, COORD_LEN );

    /* convert it into a send-adress */
    CM_make_news_coordinate_1L( tree_geom, temp_float0,  /* axis */ 0,
                                node.children, ADDRESS_LEN );

    /* copy field of splitting nodes into 1st processor of children line */
    CM_send_1L( node.total, temp_float0, node.total, NODE_LEN,
                CM_no_field );

    /* select the new nodes */
    CM_set_context();
    CM_u_ge_constant_1L( coord, free_proc_addr, COORD_LEN );
    CM_logand_context_with_test();
    CM_u_lt_constant_1L( coord, free_proc_addr + new_proc_nb, COORD_LEN );
    CM_logand_context_with_test();

    /* spread the field of 1st child to all children of the same parent */
    CM_spread_with_copy_1L( node.total, node.total, /* axis */ 1,
                            NODE_LEN, 0 );
```

```
      /* now all nodes have correct children address fields */

      /* increment level number in the new nodes */
      CM_u_s_shift_constant_3_2L( node.level_mask, node.level_mask,
                                  (int) 1, LEVEL_NB_MAX, LEVEL_NB_MAX );

      /* update size of the new nodes */
      for( i = 0; i < DIMENSION; i++ )
        CM_f_divide_constant_2_1L( node.cube.size[i], 2.0, FLENS );

      /* clear children field */
      CM_u_move_zero_1L( node.children, COORD_LEN );

      /* update origin of the new nodes */
      CM_u_move_zero_always_1L( temp, DIMENSION );
      CM_my_news_coordinate_1L( temp, /* axis */ 1, DIMENSION );
      for( i = 0; i < DIMENSION; i++ ) {
        CM_load_context( CM_SUBFIELD( temp, i ) );
        CM_f_add_2_1L( node.cube.origin[i], node.cube.size[i], FLENS );
      }

      /* update the pointer to the free processor zone */
      free_proc_addr += new_proc_nb;

      printf("   pointer to free zone = %u\n", free_proc_addr );fflush(stdout);

      for( i = 3; i < LEVEL_NB_MAX; i++ ) {
        /* select all nodes of level i */
        CM_load_context( CM_SUBFIELD( node.level_mask, i-1 ) );
        a_processor[ i ] = CM_global_u_min_1L( coord, COORD_LEN );
      } /* NOTE: the values of a_processor[1] and a_processor[2] are already known */

    }

  }
  else printf("CAN'T SPLIT: ran out of nodes in octree...\n");fflush(stdout);

  /* in all cases, move the objects down the pipeline */
  for( i = LEVEL_NB_MAX; i > 1; i-- ) {
    /* select all nodes of level i */
    CM_load_context( CM_SUBFIELD( node.level_mask, i-1 ) );
    if( CM_global_logior_context() ) {

      CM_spread_with_copy_1L( object.total, object.total, /* axis */ 0,
                              SPLIT_OBJECT_LEN, a_processor[ /* level */ i-1 ] );
      CM_spread_with_copy_1L( visual.total, visual.total, /* axis */ 0,
                              SPLIT_OBJECT_LEN, a_processor[ /* level */ i-1 ] );
    }
  }

  return( NO_ERROR );

}       /* end of process_object() */

/*****************************************************************************/
make_tree( space_cube )
     fe_cube_t space_cube;
{
  unsigned i;

  CM_set_context();
  /* clear all processors */
  CM_u_move_zero_always_1L( node.total, NODE_LEN );
  CM_clear_bit_always( ImplyBit_object );
  CM_clear_bit_always( HitBit_object );
  CM_clear_bit_always( ImplyBit_visual );
  CM_clear_bit_always( HitBit_visual );

  /* select 1st line (0-coordinate = 0) of processors */
  CM_u_eq_zero_1L( coord, COORD_LEN );
  CM_logand_context_with_test();

  /* initialize level to 1 */
```

```
  CM_u_move_constant_1L( node.level_mask, 1, LEVEL_NB_MAX );

  /* initialize size */
  for( i = 0; i < DIMENSION; i++ )
    CM_f_move_constant_1L( node.cube.size[i], space_cube.size[i]/2.0, FLENS );

  /* initialize origins of the nodes */
  for( i = 0; i < DIMENSION; i++ )
    CM_f_move_constant_1L( node.cube.origin[i],
                           space_cube.origin[i], FLENS );
  CM_u_move_zero_always_1L( temp, DIMENSION );
  CM_my_news_coordinate_1L( temp, /* axis */ 1, DIMENSION );
  for( i = 0; i < DIMENSION; i++ ) {
    CM_load_context( CM_SUBFIELD( temp, i ) );
    CM_f_add_2_1L( node.cube.origin[i], node.cube.size[i], FLENS );
  }

  free_proc_addr = 1;

}

/*****************************************************************************/
Status_t
error_message( status )
     Status_t status;
{
  char string[ MISC_BUFFER_SIZE ];

  switch ( status ) {
  case NO_ERROR:     return( status );
  case NO_OBJECT:    sprintf( string, "No object in data file...\n");
                     break;
  case CANNOT_OPEN:  sprintf( string, "Cannot open file...\n" );
                     break;
  case CANNOT_OPEN_DATA:
                     sprintf( string, "Cannot open data file...\n" );
                     break;
  case DATA_ERROR:   sprintf( string, "Error in data file...\n" );
                     break;
  default:           sprintf( string, "WARNING: error occured!\n");
  }

  AVSmessage( file_version, AVS_Fatal, AVSmodule,"tree", NULL, "%s\n", string );
  return( status );

}       /* end of error_message */

/*****************************************************************************/
allocate_face( face_p )
     face_t *face_p;
{
  int i;

  (*face_p).total = CM_allocate_heap_field( FACE_LEN );
  for( i = 0; i < DIMENSION; i++ )
    (*face_p).normal_vector[i] =
      CM_SUBFIELD( (*face_p).total, i*FLEN );
  (*face_p).sign_bit =
    CM_SUBFIELD( (*face_p).total, i*FLEN );
  (*face_p).face_bit =
    CM_SUBFIELD( (*face_p).total, i*FLEN + 1 );

}       /* of allocate_face() */

/*****************************************************************************/
allocate_object( object_p )
     object_t *object_p;
{
  int i, j;
  unsigned offset;

  (*object_p).total = CM_allocate_heap_field( OBJECT_LEN );
  CM_u_move_zero_always_1L( (*object_p).total, OBJECT_LEN );
```

```
  offset = 0;
  for( i = 0; i < NB_FACES; i++ ) {
    (*object_p).face[i].total = CM_SUBFIELD( (*object_p).total, offset );
    for( j = 0; j < DIMENSION; j++ )
      (*object_p).face[i].normal_vector[j] =
        CM_SUBFIELD( (*object_p).face[i].total, j*FLEN );
    (*object_p).face[i].sign_bit =
      CM_SUBFIELD( (*object_p).face[i].total, j*FLEN );
    (*object_p).face[i].face_bit =
      CM_SUBFIELD( (*object_p).face[i].total, j*FLEN + 1 );
    offset += j*FLEN + 2;
  }
}       /* of allocate_object() */

/*****************************************************************************/
allocate_split_object( object_p )
     split_object_t *object_p;
{
  int i, j;
  unsigned offset;

  (*object_p).total = CM_allocate_heap_field( SPLIT_OBJECT_LEN );
  offset = 0;
  for( i = 0; i < FACES_PER_NODE; i++ ) {
    (*object_p).face[i].total = CM_SUBFIELD( (*object_p).total, offset );
    for( j = 0; j < DIMENSION; j++ )
      (*object_p).face[i].normal_vector[j] =
        CM_SUBFIELD( (*object_p).face[i].total, j*FLEN );
    (*object_p).face[i].sign_bit =
      CM_SUBFIELD( (*object_p).face[i].total, j*FLEN );
    (*object_p).face[i].face_bit =
      CM_SUBFIELD( (*object_p).face[i].total, j*FLEN + 1 );
    offset += j*FLEN + 2;
  }
}       /* of allocate_split_object() */

/*****************************************************************************/
init_CM_fields( VPR )
     unsigned VPR;
{
  int i, j, offset;
  unsigned geom_dim_1D;

  CM_axis_descriptor_t tree_geom_axes[ 2 ];
  CM_axis_descriptor_t axis0, axis1;

  printf("\nAllocating virtual processor set and CM memory... ");fflush(stdout);

  list = CM_physical_vp_set();
  CM_set_vp_set( list );
  allocate_object( &object_list );

  axis0 = (CM_axis_descriptor_t)malloc( sizeof(struct CM_axis_descriptor) );
  axis1 = (CM_axis_descriptor_t)malloc( sizeof(struct CM_axis_descriptor) );
  bzero( axis0, sizeof( struct CM_axis_descriptor ) );
  bzero( axis1, sizeof( struct CM_axis_descriptor ) );
  tree_size = VPR * CM_physical_processors_limit / POWER_2_DIMENSION;
  axis0->length = tree_size;
  axis1->length = POWER_2_DIMENSION;
  axis0->weight = 1;
  axis1->weight = 1;
  axis0->ordering = CM_news_order;
  axis1->ordering = CM_news_order;

  tree_geom_axes[ 0 ] = axis0;
  tree_geom_axes[ 1 ] = axis1;
  tree_geom = CM_create_detailed_geometry( tree_geom_axes, 2 );

  geom_dim_1D = CM_geometry_total_processors( tree_geom );
  geometry_1D = CM_create_geometry( &geom_dim_1D, 1 );

  tree_grid = CM_allocate_vp_set( tree_geom );
  CM_set_vp_set( tree_grid );
```

```
  node.total              = CM_allocate_stack_field( NODE_LEN );
  node.level_mask         = CM_SUBFIELD( node.total, 0 );
  node.visibility_count = CM_SUBFIELD( node.level_mask, LEVEL_NB_MAX );
  node.hit_count          = CM_SUBFIELD( node.visibility_count, COUNTER_LEN );
  node.children           = CM_SUBFIELD( node.hit_count, COUNTER_LEN );
  node.cube.total         = CM_SUBFIELD( node.children, COORD_LEN );

  for( i = 0; i < DIMENSION; i++ )
    node.cube.origin[i]   = CM_SUBFIELD( node.cube.total, i*FLEN );
  for( i = 0; i < DIMENSION; i++ )
    node.cube.size[i]     = CM_SUBFIELD( node.cube.total, (3+i)*FLEN );

  allocate_split_object( &object );
  allocate_split_object( &visual );

  allocate_face( &working_face );

  /* allocate 4 bits for Hit/Imply computation */
  ImplyBit_object = CM_allocate_stack_field( 1 );
  HitBit_object   = CM_allocate_stack_field( 1 );
  ImplyBit_visual = CM_allocate_stack_field( 1 );
  HitBit_visual   = CM_allocate_stack_field( 1 );

  /* allocate temporary field */
  temp            = CM_allocate_stack_field( SPLIT_OBJECT_LEN );
  temp_float0     = CM_SUBFIELD( temp, 0 );
  temp_float1     = CM_SUBFIELD( temp, FLEN );
  temp_bit0       = CM_SUBFIELD( temp, 2*FLEN );
  temp_bit1       = CM_SUBFIELD( temp, 2*FLEN + 1 );
  temp_bit2       = CM_SUBFIELD( temp, 2*FLEN + 2 );
  temp_bit3       = CM_SUBFIELD( temp, 2*FLEN + 3 );
  temp_bit4       = CM_SUBFIELD( temp, 2*FLEN + 4 );

  coord           = CM_allocate_stack_field( COORD_LEN );
  CM_set_context();
  CM_my_news_coordinate_1L( coord, /* axis */ 0, COORD_LEN );

  child           = CM_allocate_stack_field( DIMENSION );
  CM_set_context();
  CM_my_news_coordinate_1L( child, /* axis */ 1, DIMENSION );

  /* allocate the info field and its subfields. This field is only */
  /* used when writing the tree into a file */
  tree_info.total = CM_allocate_heap_field( INFO_LEN );
  tree_info.hit_count = CM_SUBFIELD( tree_info.total, 0 );
  tree_info.visibility_count = CM_SUBFIELD( tree_info.total, COUNTER_LEN );
  tree_info.children = CM_SUBFIELD( tree_info.total, 2*COUNTER_LEN );

  printf("done.\n\nVP ratio = %u\nMemory per VP = %u bits.\n\n",
         VPR, CM_physical_memory_limit/VPR );fflush(stdout);

}       /* end of init_CM_fields */

/*****************************************************************************/
write_fe_tree( filename )
     char *filename;
{
  int output_fd, ok;
  fe_node_t *node_list;
  unsigned *u_buffer;
  float *f_buffer;
  int i, j, k;
  unsigned geom_dim_1D;
  CM_geometry_id_t   geometry_1D;
  node_t node_bis;
  unsigned nb_nodes;
  unsigned len;
  CM_field_id_t address, index;

  /* create 1D geometry */
  geom_dim_1D = CM_geometry_total_processors( tree_geom );
  geometry_1D = CM_create_geometry( &geom_dim_1D, 1 );
```

```
/* switch to 1D geometry */
CM_set_vp_set_geometry( tree_grid, geometry_1D );

/* allocate node_bis */
node_bis.total = CM_allocate_stack_field( 2*DIMENSION*FLEN + 2*COUNTER_LEN );
j = 0;
for( i = 0; i < DIMENSION; i++, j++ )
  node_bis.cube.origin[ i ] = CM_SUBFIELD( node_bis.total, j*FLEN );
for( i = 0; i < DIMENSION; i++, j++ )
  node_bis.cube.size[ i ] = CM_SUBFIELD( node_bis.total, j*FLEN );
node_bis.hit_count = CM_SUBFIELD( node_bis.total, j*FLEN );
node_bis.visibility_count = CM_SUBFIELD( node_bis.total, j*FLEN+COUNTER_LEN );

CM_set_context();
CM_u_ne_zero_1L( node.level_mask, LEVEL_NB_MAX );
CM_logand_context_with_test();
CM_u_eq_zero_1L( node.children, COORD_LEN );
CM_logand_context_with_test();
CM_u_ne_zero_1L( node.hit_count, COUNTER_LEN );
CM_logand_context_with_test();

nb_nodes = CM_global_count_context();

node_list = (fe_node_t *)calloc( nb_nodes, sizeof(fe_node_t) );

len = CM_geometry_send_address_length( geometry_1D );
address = CM_allocate_stack_field( len );
index = CM_allocate_stack_field( len );

CM_enumerate_1L( index, /* axis */ 0, len,
                 CM_upward, CM_exclusive, CM_none, CM_no_field );
CM_make_news_coordinate_1L( geometry_1D, address, /* axis */ 0, index, len );

CM_deallocate_stack_through( index );

CM_send_1L( node_bis.cube.origin[0], address, node.cube.origin[0],
            3*FLEN, CM_no_field );
f_buffer = (float *)calloc( nb_nodes, sizeof(float) );
for( i = 0; i < DIMENSION; i++ ) {
  get_1D_f_array_from_CM( f_buffer, node_bis.cube.origin[ i ], nb_nodes );
  for( j = 0; j < nb_nodes; j++ )
    (*(node_list+j)).origin[i] = *(f_buffer+j);
}
cfree(f_buffer);

CM_send_1L( node_bis.cube.size[0], address, node.cube.size[0],
            3*FLEN, CM_no_field );
f_buffer = (float *)calloc( nb_nodes, sizeof(float) );
for( i = 0; i < DIMENSION; i++ ) {
  get_1D_f_array_from_CM( f_buffer, node_bis.cube.size[ i ], nb_nodes );
  for( j = 0; j < nb_nodes; j++ )
    (*(node_list+j)).size[i] = *(f_buffer+j);
}
cfree(f_buffer);

CM_send_1L( node_bis.hit_count, address, node.hit_count,
            COUNTER_LEN, CM_no_field );
u_buffer = (unsigned *)calloc( nb_nodes, sizeof(unsigned) );
get_1D_u_array_from_CM( u_buffer, node_bis.hit_count, nb_nodes, COUNTER_LEN );
for( j = 0; j < nb_nodes; j++ )
  (*(node_list+j)).hit_count = *(u_buffer+j);
cfree(u_buffer);

CM_send_1L( node_bis.visibility_count, address, node.visibility_count,
            COUNTER_LEN, CM_no_field );
u_buffer = (unsigned *)calloc( nb_nodes, sizeof(unsigned) );
get_1D_u_array_from_CM( u_buffer, node_bis.visibility_count, nb_nodes, COUNTER_LEN );
for( j = 0; j < nb_nodes; j++ )
  (*(node_list+j)).visibility_count = *(u_buffer+j);
cfree(u_buffer);

CM_deallocate_stack_through( address );
```

```
  printf("\nwriting %u nodes into front-end file '%s'... ", nb_nodes, filename );
  fflush(stdout);

  /* open the image file on the front end */
  output_fd = open( filename, O_CREAT | O_WRONLY | O_TRUNC, 0666 );

  /*  write the image onto disk */
  ok = write( output_fd, node_list, nb_nodes*sizeof( fe_node_t ) );

  close( output_fd );

  cfree( node_list );

  if( ok < 0 ) perror( filename );
  else printf("done.\n");fflush(stdout);

  CM_deallocate_stack_through( node_bis.total );

  /* switch back to original geometry */
  CM_set_vp_set_geometry( tree_grid, tree_geom );
  CM_deallocate_geometry( geometry_1D );

}       /* of write_fe_tree() */

/*************************************************************************/
get_1D_f_array_from_CM( fe_array, source, size )
     float *fe_array;
     CM_field_id_t source;
     int size;
{
  int fe_offset;
  int cm_start, cm_end;
  int cm_axis, fe_dim;

  fe_offset = 0;
  cm_start = 0;
  cm_end = size;
  cm_axis = 0;
  fe_dim = size;

  CM_f_read_from_news_array_1L( fe_array, &fe_offset, &cm_start,
                                &cm_end, &cm_axis, source, FLENS,
                                /* rank */ 1, &fe_dim, CM_float_single );

}   /* of get_1D_f_array_from_CM() */
/*************************************************************************/
get_1D_u_array_from_CM( fe_array, source, size, length )
     unsigned *fe_array;
     CM_field_id_t source;
     int size;
     unsigned int length;
{
  int fe_offset;
  int cm_start, cm_end;
  int cm_axis, fe_dim;

  fe_offset = 0;
  cm_start = 0;
  cm_end = size;
  cm_axis = 0;
  fe_dim = size;

  CM_u_read_from_news_array_1L( fe_array, &fe_offset, &cm_start,
                                &cm_end, &cm_axis, source, length,
                                /* rank */ (int) 1, &fe_dim, CM_32_bit );

}   /* of get_1D_u_array_from_CM() */

/*************************************************************************/
save_tree( filename, field )
     char *filename;
     CM_field_id_t field;
{
```

```
  int output_fd;
  int ok;

  CM_set_vp_set( tree_grid );
  output_fd = CMFS_open( filename,
                         CMFS_O_CREAT | CMFS_O_WRONLY | CMFS_O_TRUNC, 0666 );

  printf("\nwriting DataVault file '%s'... ", filename );fflush(stdout);

  ok = CMFS_write_file_always( output_fd, field, INFO_LEN );
  CMFS_close( output_fd );

  if( ok < 0 ) CMFS_perror( filename );
  else printf("done.\n");fflush(stdout);

}       /* of save_tree */

/*****************************************************************************/
initialize( vpr_string, space_cube, origin_x, origin_y, origin_z,
            size_x, size_y, size_z, mask )
     char *vpr_string;
     fe_cube_t space_cube;
     float origin_x, origin_y, origin_z, size_x, size_y, size_z;
     int *mask;
{
  int VPR;

  /* make VPR param invisible since the user should not modify it */
  AVSparameter_visible( VPR_PARAM, AVS_INVISIBLE );

  /* initialize the (global) space_cube structure for future use */
  space_cube.origin[ 0 ] = origin_x;
  space_cube.origin[ 1 ] = origin_y;
  space_cube.origin[ 2 ] = origin_z;
  space_cube.size[ 0 ] = size_x;
  space_cube.size[ 1 ] = size_y;
  space_cube.size[ 2 ] = size_z;

  printf("\nWarm booting the CM..."); fflush(stdout);
  CM_init();
  printf(" done.\n");fflush(stdout);

  CM_set_safety_mode( SAFETY_MODE );

  /* allocate vp-set and CM memory fields */
  sscanf( vpr_string, "VP ratio %d", &VPR );
  init_CM_fields( VPR );

  /* initialize the tree, ie create root */
  make_tree( space_cube );

  /* nothing to update yet! */
  *mask = 0;

  INITIALIZED = YES;
} /* of initialize() */

/*****************************************************************************/
static
tree_compute( data_input, dv_tree_output, fe_tree_output,
              origin_x_output, origin_y_output, origin_z_output,
              size_x_output, size_y_output, size_z_output,
              vpr_string, tree_depth,
              tree_filename, write_fe, done, clear, origin_x, origin_y, origin_z,
              size_x, size_y, size_z )
     char *data_input;
     char **dv_tree_output, **fe_tree_output;
     float **origin_x_output, **origin_y_output, **origin_z_output;
     float **size_x_output, **size_y_output, **size_z_output;
     char *vpr_string;
     int tree_depth;
     char *tree_filename;
     boolean write_fe, done, clear;
```

```
    float *origin_x, *origin_y, *origin_z;
    float *size_x, *size_y, *size_z;
{
  int i, j, k;
  unsigned update_level_mask;
  unsigned nb_objects;
  char data_filename_header[ MISC_BUFFER_SIZE ];
  char fe_filename[ MISC_BUFFER_SIZE ];
  char dv_filename[ MISC_BUFFER_SIZE ];
  fe_cube_t space_cube;
  Status_t STATUS = NO_ERROR;

  if( clear ) {
    INITIALIZED = NO;
    /* reset parameter */
    AVSmodify_parameter( CLEAR_TREE_PARAM, AVS_VALUE, NO, NULL, NULL );
  }

  if( !INITIALIZED ||
      AVSparameter_changed( ORIGIN_X_PARAM ) ||
      AVSparameter_changed( ORIGIN_Y_PARAM ) ||
      AVSparameter_changed( ORIGIN_Z_PARAM ) ||
      AVSparameter_changed( SIZE_X_PARAM ) ||
      AVSparameter_changed( SIZE_Y_PARAM ) ||
      AVSparameter_changed( SIZE_Z_PARAM ) ||
      AVSparameter_changed( TREE_DEPTH_PARAM ) ||
      AVSparameter_changed( VPR_PARAM ) )
    initialize( vpr_string, space_cube,
                *origin_x, *origin_y, *origin_z,
                *size_x, *size_y, *size_z, &update_level_mask );

  if( AVSinput_changed( DATA_INPUT, 0 ) ||
      AVSparameter_changed( ORIGIN_X_PARAM ) ||
      AVSparameter_changed( ORIGIN_Y_PARAM ) ||
      AVSparameter_changed( ORIGIN_Z_PARAM ) ||
      AVSparameter_changed( SIZE_X_PARAM ) ||
      AVSparameter_changed( SIZE_Y_PARAM ) ||
      AVSparameter_changed( SIZE_Z_PARAM ) ||
      AVSparameter_changed( TREE_DEPTH_PARAM ) ||
      AVSparameter_changed( VPR_PARAM ) ) {

    CM_set_vp_set( list );

    strncpy( data_filename_header, data_input, strlen( DV_PATH ) );
    data_filename_header[ strlen( DV_PATH )] = '\0';
    if( streql( data_filename_header, DV_PATH ) )
      /* read from a DataVault file */
      STATUS = read_from_dv( data_input, object_list, OBJECT_LEN, &nb_objects);
    else
      /* read from a standard Unix file */
      STATUS = read_from_fe( data_input, object_list, &nb_objects );

    if (error_message( STATUS ) != NO_ERROR ) {
      AVSmark_output_unchanged( DV_TREE_OUTPUT );
      AVSmark_output_unchanged( FE_TREE_OUTPUT );
      AVSmark_output_unchanged( ORIGIN_X_OUTPUT );
      AVSmark_output_unchanged( ORIGIN_Y_OUTPUT );
      AVSmark_output_unchanged( ORIGIN_Z_OUTPUT );
      AVSmark_output_unchanged( SIZE_X_OUTPUT );
      AVSmark_output_unchanged( SIZE_Y_OUTPUT );
      AVSmark_output_unchanged( SIZE_Z_OUTPUT );
      return( 0 );
    }

    printf("number of objects (including visual field): %u\n", nb_objects );fflush( stdo

    if( nb_objects == 1 ) { /* when there are no objects besides the visual field */
      nb_objects = 2; /* create an artificial object */
      CM_f_write_to_processor( /* index */ 1, object_list.face[ 0 ].normal_vector[ 0 ],
                               PLUS_INFINITY, FLENS );
      CM_f_write_to_processor( /* index */ 1, object_list.face[ 0 ].normal_vector[ 1 ],
                               0.0, FLENS );
      CM_f_write_to_processor( /* index */ 1, object_list.face[ 0 ].normal_vector[ 2 ],
```

```
                                0.0, FLENS );
      CM_u_write_to_processor( /* index */ 1, object_list.face[ 0 ].sign_bit, 1, 1 );
      CM_u_write_to_processor( /* index */ 1, object_list.face[ 0 ].face_bit, 1, 1 );
  }

    CM_set_vp_set( tree_grid );

    /* broadcast visual field contained at index 0 to all processors */
    object_index = 0;
    broadcast_object( visual, object_index, /* level */ 1 );

    for( object_index = 1; object_index < nb_objects; object_index++ ) {
      broadcast_object( object, object_index, /* level */ 1 );
      STATUS = process_object( update_level_mask, tree_depth );
      update_level_mask <<= 1;  /* shift 1 bit to the left */
    }

    update_level_mask += 1;

  }

  if( done ) {

    /* now, empty the pipeline as long as necessary */
    while( ( STATUS == NO_ERROR ) && (update_level_mask % LEVEL_MAX_MASK) ) {

      /* process the same object at the root of the tree so that all objects
         still in the tree can move down and the counters be updated */
      STATUS = process_object( update_level_mask, tree_depth );

      update_level_mask <<= 1;  /* shift 1 bit to the left */

      printf(" -> emptying the pipeline\n");fflush( stdout );
    }

    if( (STATUS == NO_ERROR) )
      STATUS = process_object( update_level_mask, tree_depth );

    /* write resulting tree into file either on front-end or DataVault, */
    /* or both if required */
    if( (STATUS == NO_ERROR) ) {
      if( write_fe ) {
        sprintf( fe_filename, "%s%s", FE_PATH, tree_filename );
        write_fe_tree( fe_filename );

        /* set module output */
        sprintf( fe_filename, "%s", tree_filename );
        *fe_tree_output = fe_filename;
      }
      else
        AVSmark_output_unchanged( FE_TREE_OUTPUT );

      /* now write DV tree */
      CM_set_context();
      CM_u_move_1L( tree_info.hit_count, node.hit_count, COUNTER_LEN );
      CM_u_move_1L( tree_info.visibility_count, node.visibility_count, COUNTER_LEN );
      CM_u_move_1L( tree_info.children, node.children, COORD_LEN );
      sprintf( dv_filename, "%s%s", DV_PATH, tree_filename );
      save_tree( dv_filename, tree_info.total );

      /* set module output */
      *dv_tree_output = dv_filename;
    }

    CM_set_context();
    printf("Last children pointer: %u.\n",
           CM_global_u_max_1L( node.children, COORD_LEN ) );fflush( stdout );

    /* reset parameter */
    AVSmodify_parameter( DONE_PARAM, AVS_VALUE, NO, NULL, NULL );

  } /* if( done )... */
  else {
```

```
    AVSmark_output_unchanged( FE_TREE_OUTPUT );
    AVSmark_output_unchanged( DV_TREE_OUTPUT );
    AVSmark_output_unchanged( ORIGIN_X_OUTPUT );
    AVSmark_output_unchanged( ORIGIN_Y_OUTPUT );
    AVSmark_output_unchanged( ORIGIN_Z_OUTPUT );
    AVSmark_output_unchanged( SIZE_X_OUTPUT );
    AVSmark_output_unchanged( SIZE_Y_OUTPUT );
    AVSmark_output_unchanged( SIZE_Z_OUTPUT );
 }

/* print error message if appropriate  */
  if (error_message( STATUS ) != NO_ERROR ) {
    AVSmark_output_unchanged( DV_TREE_OUTPUT );
    AVSmark_output_unchanged( FE_TREE_OUTPUT );
    AVSmark_output_unchanged( ORIGIN_X_OUTPUT );
    AVSmark_output_unchanged( ORIGIN_Y_OUTPUT );
    AVSmark_output_unchanged( ORIGIN_Z_OUTPUT );
    AVSmark_output_unchanged( SIZE_X_OUTPUT );
    AVSmark_output_unchanged( SIZE_Y_OUTPUT );
    AVSmark_output_unchanged( SIZE_Z_OUTPUT );
    return( 0 );
  }
  fflush( stdout );      fflush( stderr );

  *origin_x_output = origin_x;
  *origin_y_output = origin_y;
  *origin_z_output = origin_z;
  *size_x_output = size_x;
  *size_y_output = size_y;
  *size_z_output = size_z;

  /* indicate success */
  return(1);
}         /* end of compute() */

/*****************************************************************************/
static
tree_init()
{
  LEVEL_MAX_MASK = power( 2, LEVEL_NB_MAX );

  a_processor[ 1 ] = 0;         /* these are obvious values in a_processor */
  a_processor[ 2 ] = 1;

  /* indicate success */
  return(1);
} /* of init() */

/*****************************************************************************/
static
tree_destroy()
{
  fflush( stdout );      fflush( stderr );

  /* indicate success */
  return(1);
} /* of destroy() */
```