```
// m44.cpp - M44 SIMULATION
//
// MODULE INDEX
// NAME                         CONTENTS
// LoadProfile                  Load a profile from azimuth, elevation, height
//                              and width
// Intersect                    Determine whether two profiles intersect
// GetSectorSet                 Convert a profile into a sector set
// AppendToBucket               Append a reference to a bucket
// main                         Main line
//
// MAINTENANCE HISTORY
// DATE         PROGRAMMER AND DETAILS
// 16-10-03     JS      Original
//
//---------------------------------------------------------------------------

#include <string.h>             // String manipulation functions
#include <stdio.h>              // Standard input/output
#include <math.h>               // Mathematical functions
#include <time.h>               // System time functions
#include <stdlib.h>             // Standard library

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

// DEFINITIONS

#define HORIZ_SEC_CNT   1
                                // Number of horizontal sectors
#define VERT_SEC_CNT    1
                                // Number of vertical sectors
#define FRIEND_CNT      100000
                                // Number of friends
#define TRAJ_CNT        1000
                                // Number of trajectories
#define MIN_RADIUS      5.0
                                // Minimum radius
#define MAX_RADIUS      25000.0
                                // Maximum radius
#define MIN_AZIMUTH     0.0
                                // Minimum azimuth
#define MAX_AZIMUTH     360.0
                                // Maximum azimuth
#define AZIMUTH_RANGE   (MAX_AZIMUTH-MIN_AZIMUTH)
                                // Azimuth range
#define SECTOR_WIDTH    (AZIMUTH_RANGE/HORIZ_SEC_CNT)
                                // Sector width
#define MIN_ELEV        (-45.0)
                                // Minimum elevation
#define MAX_ELEV        45.0
                                // Maximum elevation
#define ELEV_RANGE      (MAX_ELEV - MIN_ELEV)
                                // Elevation range
#define SECTOR_HEIGHT   (ELEV_RANGE / VERT_SEC_CNT)
                                // Sector height
#define MIN_FRIEND_ELEV (-30.0)
                                // Minimum friend elevation
#define MAX_FRIEND_ELEV 30.0
                                // Maximum friend elevation
#define FRIEND_WIDTH    0.5
                                // Friend's width in metres
#define FRIEND_HEIGHT   1.8
                                // Friend's height in metres
```

```
#define MIN_TRAJ_ELEV   (-40.0)
                                // Minimum trajectory elevation
#define MAX_TRAJ_ELEV   40.0
                                // Maximum trajectory elevation
#define TRAJ_WIDTH      0.2
                                // Trajectory width in degrees
#define TRAJ_HEIGHT     1
                                // Trajectory height in degrees

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

// PROFILE STRUCTURE

struct Profile_t {
    double      proLeft;        // Left edge of the profile
    double      proBottom;      // Bottom edge of the profile
    double      proRight;       // Right edge of the profile
    double      proTop;         // Top edge of the profile
};

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

// SECTOR SET STRUCTURE

struct SectorSet_t {
    int         ssLeft;         // Left most sector the profile intersects
    int         ssBottom;       // Lowest sector the profile intersects
    int         ssWidth;        // Number of sectors in the horizontal direct
    int         ssHeight;       // Number of sectors in the vertical direct
};

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

// REFERENCE STRUCTURE

struct Reference_t {
    Reference_t *refNext;       // Next reference in the chain
    int         refFriendId;    // Friend identifier
};

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

// BUCKET HEADER STRUCTURE

struct Bucket_t {
    Reference_t *bucFirst;      // First reference in the bucket
    Reference_t *bucLast;       // Last reference in the bucket
};

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

// GLOBAL DATA

Profile_t       friendProArr[FRIEND_CNT];
                                // Friend profile array
Bucket_t        bucketArr[HORIZ_SEC_CNT][VERT_SEC_CNT];
                                // Bucket array
Profile_t       trajProArr[TRAJ_CNT];
                                // Trajectory profile array
Bucket_t        trajInterArr[TRAJ_CNT];
                                // Trajectory/friend intersection array

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

```
// LOAD A PROFILE FROM AZIMUTH, ELEVATION, HEIGHT AND WIDTH

inline void
LoadProfile (
    Profile_t           *pro,           // Profile to load
    double              tx,             // Azimuth
    double              ty,             // Elevation
    double              th,             // Height
    double              tw)             // Width
{
    pro->proLeft = tx - tw/2;
    pro->proRight = pro->proLeft + tw;
    pro->proBottom = ty - th/2;
    pro->proTop = pro->proBottom + th;
}

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

// DETERMINE WHETHER TWO PROFILES INTERSECT

int
Intersect (
    const Profile_t     *p1,            // First profile
    const Profile_t     *p2)            // Second profile
{
    return
        (p1->proTop > p2->proBottom && p1->proBottom < p2->proTop) &&
        (
            (p1->proRight > p2->proLeft && p1->proLeft < p2->proRight) ||
            (p1->proRight+360>p2->proLeft && p1->proLeft+360<p2->proRight) ||
            (p1->proRight>p2->proLeft+360 && p1->proLeft<p2->proRight+360)
        );
}

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

// CONVERT A PROFILE INTO A SECTOR SET

void
GetSectorSet (
    SectorSet_t         *ss,            // Sector set
    const Profile_t     *p)             // Profile
{
    int                 unWrapLeft;     // Unwrapped left boundary
    int                 unWrapRight;    // Unwrapped right boundary
    int                 unWrapBottom;   // Unwrapped bottom boundary
    int                 unWrapTop;      // Unwrapped top boundary

    // Calculate unwrapped sector ranges in the horizontal
    // and vertical directions.

    unWrapLeft = (int)((p->proLeft - MIN_AZIMUTH + AZIMUTH_RANGE)
        / SECTOR_WIDTH);
    unWrapRight = (int)((p->proRight - MIN_AZIMUTH + AZIMUTH_RANGE)
        / SECTOR_WIDTH);
    if (unWrapRight*SECTOR_WIDTH != p->proRight-MIN_AZIMUTH+AZIMUTH_RANGE)
        unWrapRight++;
    unWrapBottom = (int)((p->proBottom - MIN_ELEV + ELEV_RANGE)
        / SECTOR_HEIGHT);
    unWrapTop = (int)((p->proTop - MIN_ELEV + ELEV_RANGE)
        / SECTOR_HEIGHT);
    if (unWrapTop*SECTOR_HEIGHT != p->proTop-MIN_ELEV+ELEV_RANGE)
```

```
        unWrapTop++;

    // Convert the unwrapped sector ranges into the sector set components.

    ss->ssLeft = unWrapLeft % HORIZ_SEC_CNT;
    ss->ssWidth = unWrapRight - unWrapLeft;
    ss->ssBottom = unWrapBottom % VERT_SEC_CNT;
    ss->ssHeight = unWrapTop - unWrapBottom;
}

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

// APPEND A REFERENCE TO A BUCKET

void
AppendToBucket (
    Bucket_t            *buc,           // Pointer to bucket structure
    int                 friendId)       // Friend identifier
{
    Reference_t         *ref;           // Pointer to reference structure

    ref = new Reference_t;
    ref->refNext = 0;
    ref->refFriendId = friendId;
    if (buc->bucLast == 0)
        buc->bucFirst = ref;
    else
        buc->bucLast->refNext = ref;
    buc->bucLast = ref;
}

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

// MAIN LINE

int
main ()
{
    int                 i, j;           // General purpose indices
    int                 x, y;           // Sector indices
    double              e, n;           // Easterly and northerly co-ordinates
    double              r;              // Distance from subject to friend
    double              tx, ty;         // Azimuth and elevation
    double              th, tw;         // Angular height and width
    time_t              startTime;      // Starting time
    time_t              endTime;        // Ending time
    SectorSet_t         ss;             // Sector set
    Reference_t         *ref;           // Reference pointer
    Reference_t         *prevRef;       // Previous reference pointer
    int                 friendId;       // Friend identifier
    int                 trajId;         // Trajectory identifier

    // Initialise the buckets

    for (x = 0; x < HORIZ_SEC_CNT; x++) {
        for (y = 0; y < VERT_SEC_CNT; y++) {
            bucketArr[x][y].bucFirst = 0;
            bucketArr[x][y].bucLast = 0;
        }
    }

    // Generate the friend profiles
```

```
friendId = 0;
do {

    // Generate pseudo-random easterly and northerly co-ordinates

    e = (drand48() * 2*MAX_RADIUS) - MAX_RADIUS;
    n = (drand48() * 2*MAX_RADIUS) - MAX_RADIUS;

    // Calculate the radius.  If it is outside the required range
    // generate another pair of co-ordinates.

    r = sqrt (e*e + n*n);
    if (r < MIN_RADIUS || r >= MAX_RADIUS) continue;

    // Calculate the azimuth to the friend

    tx = atan2 (e, n) * 180.0 / M_PI;
    if (tx < 0) tx += 360.0;
    if (tx >= 360) tx = 0; // (get rid of rounding overruns)

    // Generate the elevation of the friend

    ty = MIN_FRIEND_ELEV + (drand48() * (MAX_FRIEND_ELEV-MIN_FRIEND_ELEV));

    // Calculate the angular width and height of the friend

    tw = FRIEND_WIDTH * 180.0 / (r * M_PI);
    th = FRIEND_HEIGHT * 180.0 / (r * M_PI);

    // Generate the friend's profile

    LoadProfile (&friendProArr[friendId], tx, ty, tw, th);

    // Append the friend to the buckets his profile intersects

    GetSectorSet (&ss, &friendProArr[friendId]);
    for (
        x = ss.ssLeft, i = 0;
        i < ss.ssWidth;
        x = (x + 1) % HORIZ_SEC_CNT, i++
    ) {
        for (
            y = ss.ssBottom, j = 0;
            j < ss.ssHeight;
            y = (y + 1) % VERT_SEC_CNT, j++
        ) {
            AppendToBucket (&bucketArr[x][y], friendId);
        }
    }

    // Repeat until we have enough friends

} while (++friendId < FRIEND_CNT);

// Record the starting time

startTime = time(0);

// Simulate the trajectory intersection calculations

for (trajId = 0; trajId < TRAJ_CNT; trajId++) {

    // Generate the profile of the trajectory
```

```
    tx = MIN_AZIMUTH + (drand48() * (MAX_AZIMUTH-MIN_AZIMUTH));
    ty = MIN_TRAJ_ELEV + (drand48() * (MAX_TRAJ_ELEV-MIN_TRAJ_ELEV));
    LoadProfile (&trajProArr[trajId], tx, ty, TRAJ_WIDTH, TRAJ_HEIGHT);

    // Determine the sectors the trajectory profile intersects

    GetSectorSet (&ss, &trajProArr[trajId]);

    // Initialise the intersection list

    trajInterArr[trajId].bucFirst = 0;
    trajInterArr[trajId].bucLast = 0;

    // Determine the friends the trajectory intersects

    for (
        x = ss.ssLeft, i = 0;
        i < ss.ssWidth;
        x = (x + 1) % HORIZ_SEC_CNT, i++
    ) {
        for (
            y = ss.ssBottom, j = 0;
            j < ss.ssHeight;
            y = (y + 1) % VERT_SEC_CNT, j++
        ) {
            for (
                ref = bucketArr[x][y].bucFirst;
                ref != 0;
                ref = ref->refNext
            ) {
                if (
                    Intersect (&trajProArr[trajId],
                        &friendProArr[ref->refFriendId])
                ) {
                    // Only add the friend to the intersection list
                    // if the friend is not already in the list.

                    for (
                        prevRef = trajInterArr[trajId].bucFirst;
                        prevRef != 0 &&
                            prevRef->refFriendId != ref->refFriendId;
                        prevRef = prevRef->refNext
                    );
                    if (prevRef == 0)
                        AppendToBucket (&trajInterArr[trajId],
                            ref->refFriendId);
                }
            }
        }
    }
}

// Record the ending time

endTime = time(0);

// Display the trajectories and intersections

for (trajId = 0; trajId < TRAJ_CNT; trajId++) {
    printf (
        "%7.3f %7.3f %7.3f %7.3f\n",
        (trajProArr[trajId].proLeft + trajProArr[trajId].proRight) / 2,
```

```
            (trajProArr[trajId].proBottom + trajProArr[trajId].proTop) / 2,
            trajProArr[trajId].proRight - trajProArr[trajId].proLeft,
            trajProArr[trajId].proTop - trajProArr[trajId].proBottom
        );

        for (
            ref = trajInterArr[trajId].bucFirst;
            ref != 0;
            ref = ref->refNext
        ) {
            printf (
                "        %7.3f %7.3f %7.3f %7.3f\n",
                (friendProArr[ref->refFriendId].proLeft
                    + friendProArr[ref->refFriendId].proRight) / 2,
                (friendProArr[ref->refFriendId].proBottom
                    + friendProArr[ref->refFriendId].proTop) / 2,
                friendProArr[ref->refFriendId].proRight
                    - friendProArr[ref->refFriendId].proLeft,
                friendProArr[ref->refFriendId].proTop
                    - friendProArr[ref->refFriendId].proBottom
            );
        }
    }

    // Display the average time per trajectory

    printf ("Average calculation time per trajectory = %1.9f\n",
        (endTime - startTime) / (double)TRAJ_CNT);

    // Exit gracefully

    return 0;
}
```