BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-93-M9

''Monotonic Chain Decomposition in Randomly Generated Terrains''

by

Stephen W. Thamel

Monotonic Chain Decomposition in Randomly Generated Terrains

by

Stephen W. Thamel
B.S., Worcester Polytechnic Institute, 1989

Abstract

This paper presents an efficient and practical implementation of the complete monotonic chain decomposition of randomly generated polyhedral terrains. It discusses the involved data structures, algorithmic techniques and design decisions. The implementation is completely object-oriented in C++, emphasizing data encapsulation, total extendibility, modularity and large-scale computer graphics design. The source code is included.

This research project by Stephen W. Thamel is accepted in its present form by the Department of Computer Science at Brown University in partial fulfillment of the requirements for the Degree of Master of Science.

Date: May 5, 1993 Advisor: Franco P. Preparata

Introduction

Terrains are an effective model for many applications in computer graphics. Precisely, a terrain is a three-dimensional closed polyhedron such that, for each location (x,y) on its base plane, there is zero, one or a closed interval of z values that lie on that polyhedron.

It is the nature of terrains to project a connected planar graph on the base plane. It is this special intrinsic property that is of research interest. It allows a complete monotonic decomposition into more simple convex objects, namely polygonal chains.

The corresponding polyhedral chains, in three space, can then be ordered and processed via the ordering of the projected polygonal chains. Optimal O(**n**log**n**) hidden line elimination has been developed for polyhedral chain representations [Pre92]. Other graphic algorithms, such as depth cueing, shading and ray tracing can be adapted to process these complete polyhedral chains.

This project efficiently generates a random planar graph, extrudes the polyhedral terrain, performs the polyhedral decomposition into a useful hierarchy and finally displays the complete set of monotonic chains representing the terrain.

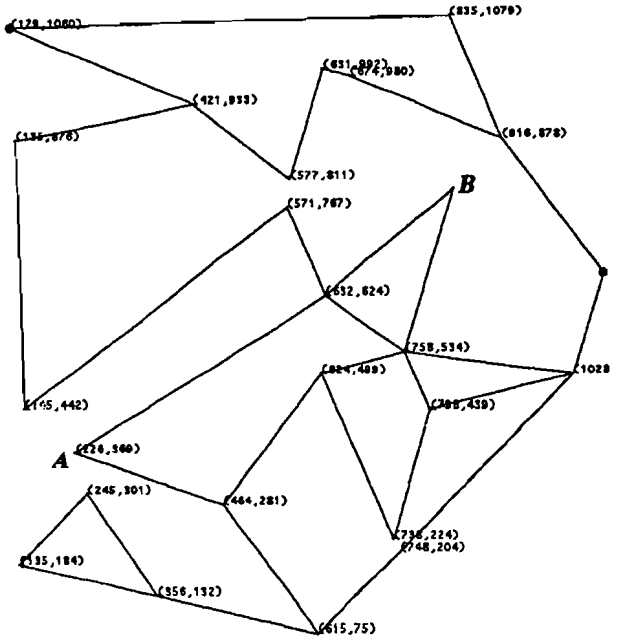

Figure 1 - "An Planar Connected Graph"

Constructing a Planar Connected Graph

The graph shown in figure 1 was constructed using a Delaunay tree structure previously implemented in the programming package LEDA [N90]. A comprehensive discussion of the Delaunay tree and its efficient implementation exists in [De92].

Our **n** nodes of the graph **G** are created using a pseudo-random generator, assigning (x,y) points about a space of (2**n**,2**n**). The distribution period of the random generator is more than adequate to produce excellently distributed data sets.

Every point is then inserted into the Delaunay tree at a cost of O(log**n**) each. After all points are inserted, a traversal of the tree yields the segments that form the complete triangulation of the above graph. As it is based on dual of the Voronoi diagram, the overall cost is O(**n**log**n**).

In an effort to activate the regularization routine, some edges are removed from the complete triangulation. They are removed, if and only if, they do not disconnect the graph. The regularization algorithm presumes a planar connected graph.

Regularization of a Planar Connected Graph

A node is said to be regular if it has both an incoming edge and an outgoing edge. A graph is regular is all its nodes are regular. It is clear, however, that node **A** in figure 1 is not regular. It has no incoming edges. Likewise, node **B** has no outgoing edges. For definitions, the reader is referred to [Pre85].

A forward vertical plane-sweep algorithm is employed to regularize all nodes in **G** lacking incoming edges. The X-structure is maintained as a priority-queue of nodes keyed on their x coordinates. The Y-structure is maintained as a height-balanced tree of edges keyed on their slope. Additionally for each interval of the Y structure, we store the node with max x coordinate yet encountered in this edge's vertical strip.

As we proceed through the priority queue and encounter nodes lacking incoming edges, we decide which neighbor (above or below) has the maximum max node stored. At this point, an edge is added from that special node to our current node. Clearly, this additional edge will not disturb the planarity invariant.

As for nodes lacking outgoing edges, we perform a reverse vertical plane-sweep. We utilize the same structures, except we maintain the node with min x coordinate in the additional structure and decide which neighbor (above or below) has the minimum min node.

The X-structure is built and depleted in O(**n**log**n**), the Y-structure in O(**e**log**e**). Neighbor searches are performed in O(log**e**). Assuming a constant relationship of nodes to edges, we have O(**n**log**n**) overall. Figures 2 and 3 depict the resulting graphs after both regularization passes of figure 1.

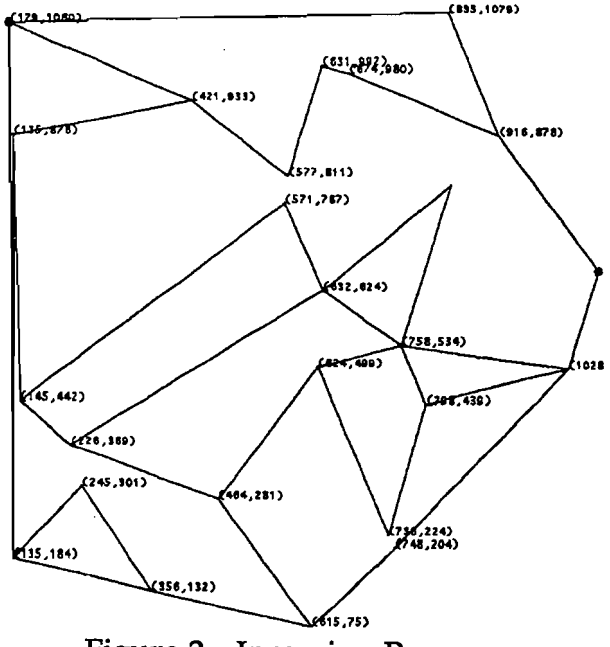

Figure 2 - Incoming Pass

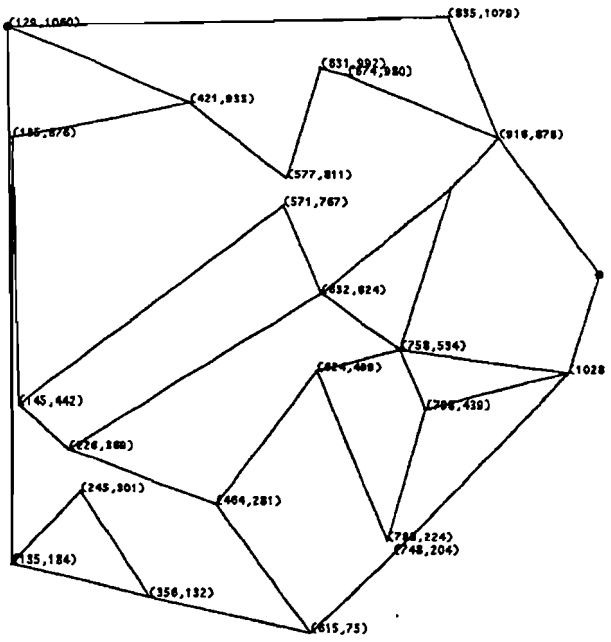

Figure 3 - Outgoing Pass

Weight Balance by Network Flow

After the graph has been completely regularized, we are ready to compute the weight balance of the graph. This is accomplished using a network flow algorithm, see [Pre85]. The algorithm uses a two pass technique on all non-extreme nodes of the graph. The first pass summates the incoming weights of a node and deposits the balance into its uppermost outgoing edge. The second pass is performed in reverse, making deposits into its uppermost incoming edge. After completion of both passes, we have a completely balanced weighted graph, as shown in figure 4. The algorithm clearly runs in time linear with the number of edges and nodes, hence O(**n**+**e**).

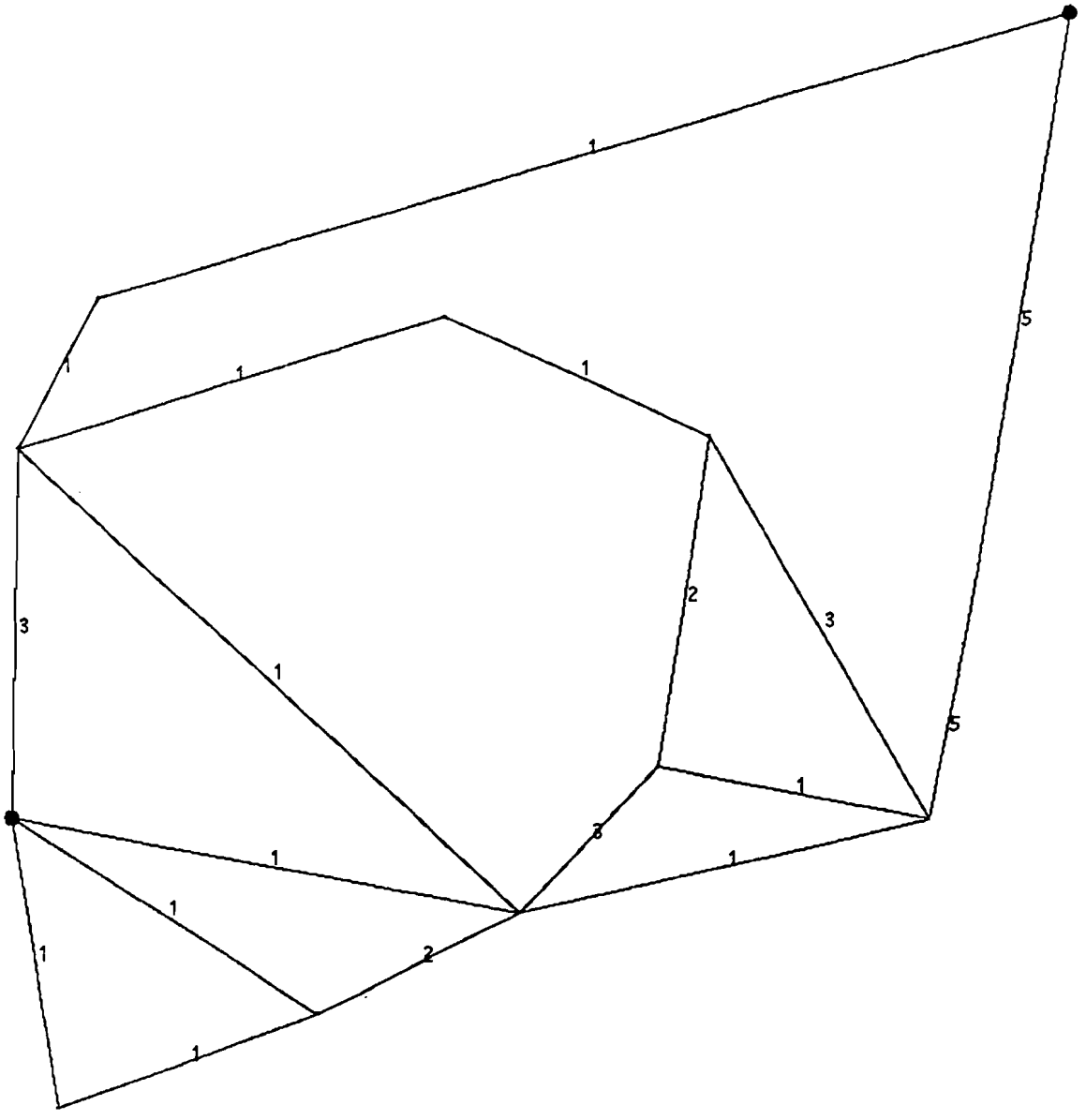

Figure 4 - "Simple Weight Balanced Graph"

Facet Constrained Z Value Assignments

Although we have (x,y) values for the projected graph, we do not have z values for every node to extrude the terrain. When assigning a z value to any node on the graph, we must make sure the incident face of that node lies completely on some plane in three space. Polygons of degree greater than three present a problem to this constraint. The following algorithm solves this assignment.

First, we construct another graph called the dual. A node is added to the dual for every face in the projected graph. Edges are added to the dual for all adjacent faces in the projected graph. Now, we perform a breadth-first search of this dual to establish a visiting order of the projected graph's faces. This ordering is crucial to avoid assignment conflicts.

We now visit those faces in the previously established order. At each face, we visit all adjacent nodes. We obtain, if possible, (3) previously assigned z values. If (3) do not exist, we randomly assign values until we have (3). The remaining nodes of this face are then mathematically assigned z values that lie on the plane determined from the (3) obtained or randomized points.

The above algorithm will produce an extruded planar graph, hereon called the terrain. The facets of this terrain will be completely coplanar and uniformly connected. Figures 4, 5 & 6 are example outputs.

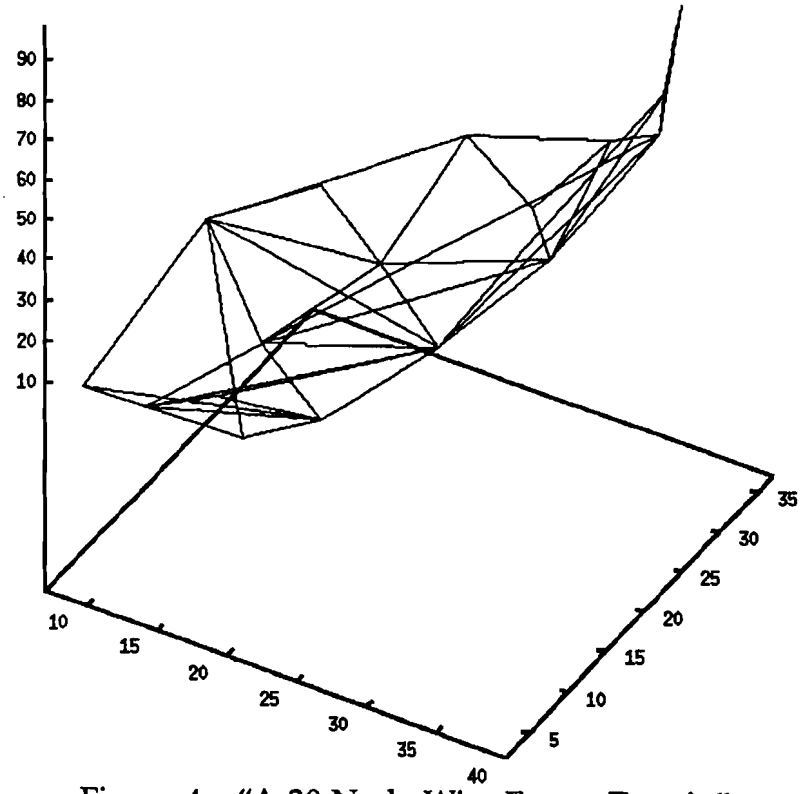

Figure 4 - "A 20 Node Wire-Frame Terrain"

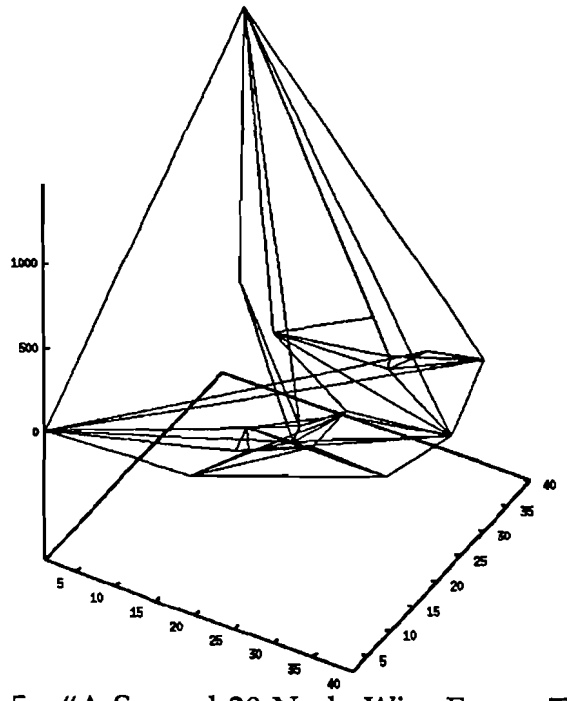

Figure 5 - "A Second 20 Node Wire-Frame Terrain"

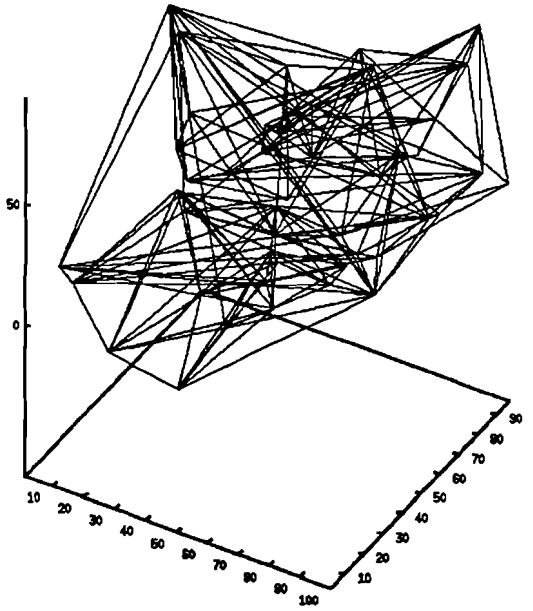

Figure 6 - "A 50 Node Wire-Frame Terrain"

Polygonal Chain Decomposition

The values assigned to the edges during the weight balancing algorithm directly correspond to the number of chains that "flow" through that edge. Therefore, these values are significant to the decomposition process. The simplest technique to decompose the graph involves selecting successive uppermost outgoing edges and decreasing the weight value each time. Once the edge weight is depleted, remove the edge from consideration.

As noted in [Pre85] an edge will belong to a closed interval of chains. It will be stored, however, only in the appropriate chain. Gaps will be placed in the remaining chains of the interval. A binary search structure keyed on chain numbers produces a well balanced distribution of chains. This implementation creates this hierarchy via the recursive functions floor((b-a+1)/2) and floor((b+a+1)/2) on the interval (a,b). Initially, the interval is (1,C), where C is the number of chains in the graph.

Figure 7 is an regular weight balanced graph. It has 10 chains in its polygonal decomposition. Figure 8 is the associated binary search hierarchy of the interval (1,10).

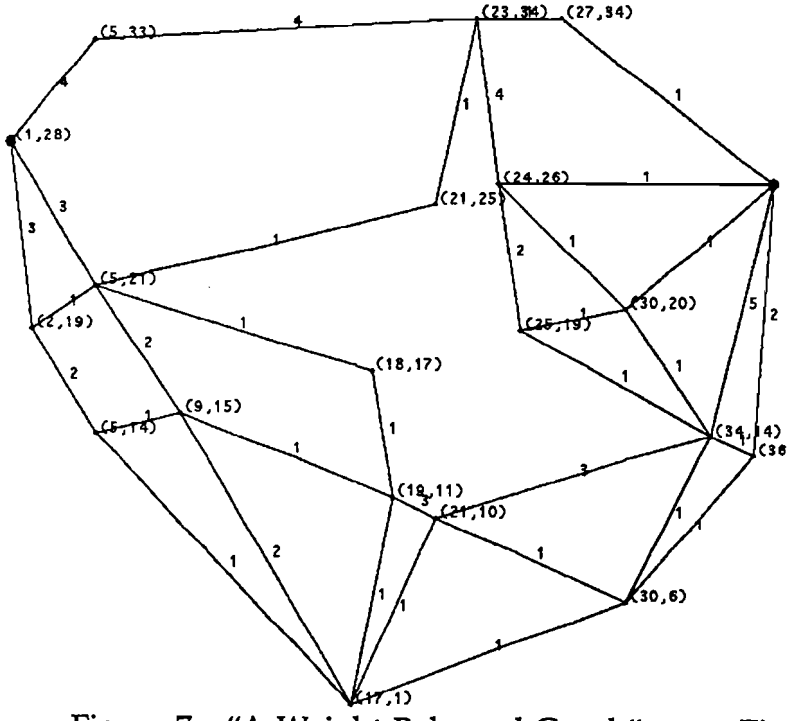

Figure 7 - "A Weight Balanced Graph"

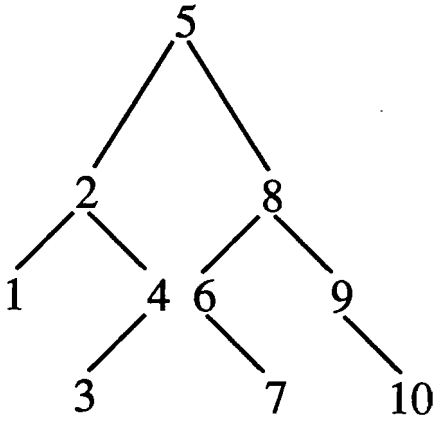

Figure 8 - "Binary Search Hierarchy"

Figure 9 depicts the chain with highest priority stored in the complete chain structure. Figure 10 shows the addition of chains #2 and #8, the two next highest priority chains. The reader will notice edge **A** is stored in chain #5 but gapped in chain #2. Edge **B** too is stored in chain #5 yet gapped in chain #8. These gaps are important to realize the desired $O(n\alpha(n))$ space.

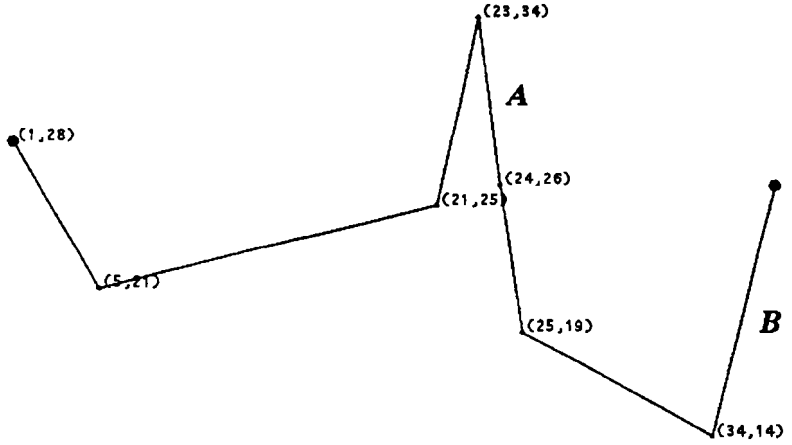

Figure 9 - "Chain #5 of Hierarchy"

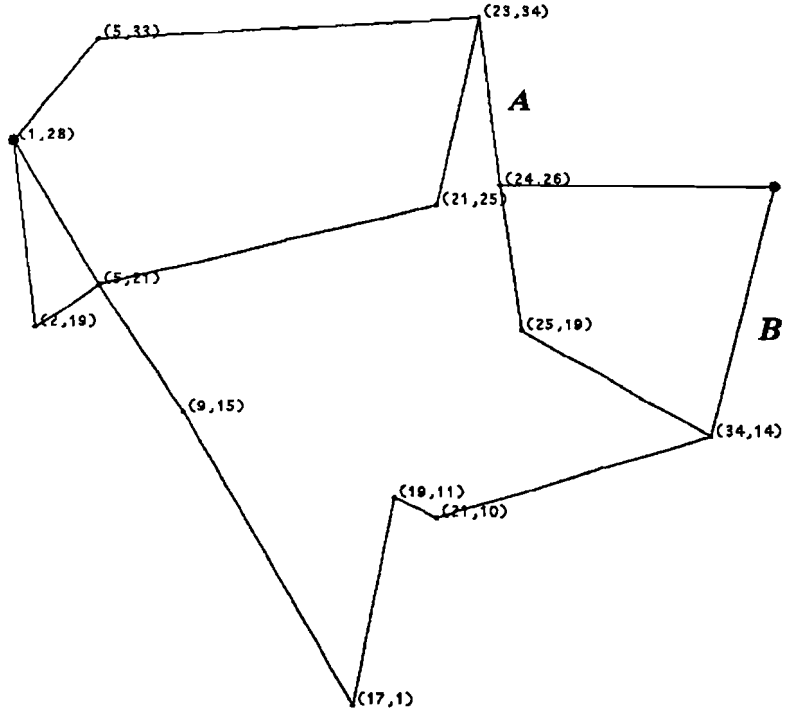

Figure 10 - "Chain #2 and #8 of Hierarchy"

Homogeneous Coordinate System

A common spherical based viewing system is employed in this project. A point, two angles and a projection distance are all the parameters needed to compute the associated homogeneous perspective projection matrix. The following diagrams should assist the reader in understanding the basis of these angles.

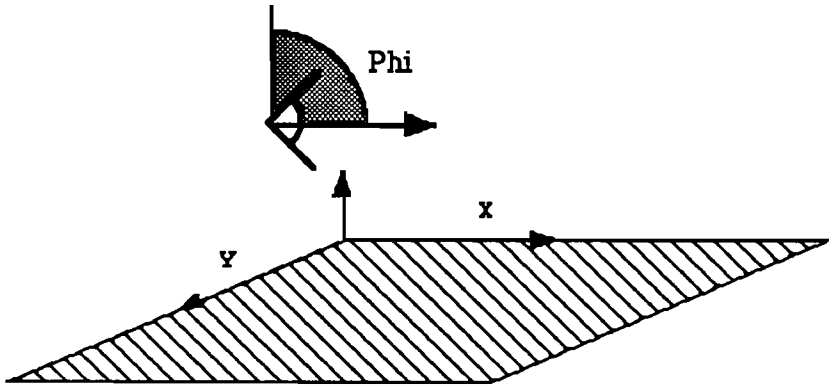

Figure 11 - The angle phi represents the upward angle off the XY plane.

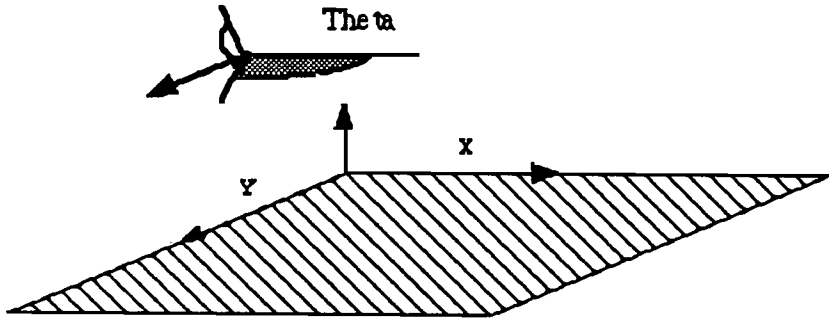

Figure 12 - The angle theta represents the angle the around the z axis.

References

[De92] O. Devillers: "Robust and Efficient Implementation of the Delaunay Tree", Technical Report, INRIA, 1992.

[N90] S. Näher: "LEDA 2.0 User Manual", Technischer Bericht A 17/90, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1990.

[Pre85] F. Preparata & M. Shamos: Computational Geometry: An Introduction, Springer-Verlag, University of Illinois at Urbana-Champaign: Urbana, IL, 1985.

[Pre92] F. Preparata & J. Vitter: "A Simplified Technique for Hidden Line Elimination in Terrains", International Journal of Computational Geometry & Applications, June 2, 1992.

```
// This file contains routines associated with operations on graphs.  That is
// various node, edge and graph functions.  It also contains the associated
// geometry computations to perform these operations.

#include <LEDA/window.h>
#include <LEDA/matrix.h>
#include <LEDA/stream.h>
#include <LEDA/h_array.h>
#include <LEDA/prio.h>
#include <LEDA/d_array.h>
#include <LEDA/graph_alg.h>
#include <LEDA/plane_alg.h>
#include <LEDA/list.h>
#include <LEDA/sortseq.h>
#include <LEDA/stack.h>
#include <LEDA/impl/delaunay_tree.h>
#include "typedefs.h"

// ********************* These Variables have FILE scope. **********************

// The FILE scope variables are needed for the Delaunay function routine.
list<segment>                S; // List of Segments. (Needed by Triangulate Edge Routine)
const double                 eps = 0.00001; // Our Assumed Zero for error tolerance.

// *********** External FILE scope Variables Declared Elsewhere. ***************

extern Ugraph G;
extern double mu,theta,phi,distance;
extern float Time;
extern file_ostream LOG;

// ********************** Comparison & Simple I/O Functions *********************
// It should be noted most (compare) functions are needed for any data-type that is
// to be insertted into a keyed structure.  This is so the tree constructor can discern
// which path to traverse along the insertion.  Furthermore, it establishes the equality of
// any two items -- very important.  Most keyed structures do not allow items with duplicate
// keys - they become clobbered!  For example, two edges are considered equal if this function
// returns zero when they are compared.  So even though two edges may be completely distinct
// to YOU, they are not necessarily to the structure you are asking to insert. The function
// (compare) is the default called for any keyed item, however you can sort objects by any
// comparator function you define.

int compare(segment& A, segment& B) // Segments by (Source,Target)
// This routine compares segments by their start and end points.  This is so the traingle
// tracing function can store then directed properly for when they become edges.
{
    int result;
    if (result = compare(A.start(),B.start())) return result;
    else if (result = compare(A.end(),B.end())) return result;
    else return 0;
}

int compare(node& a, node& b) {return compare(G[a],G[b]);} // Nodes via Points

int compare(edge& a, edge& b)
// Routine used for structures that are keyed on edges.  These comparison functions are
// essential, so as to avoid clobbering other items stored in the structure.
{
    node sa = G.source(a), ta = G.target(a), sb = G.source(b), tb = G.target(b);

    int result;
    if (result = compare(sa,sb)) return result;
    else if (result = compare(ta,tb)) return result;
    else return 0; // You must be comparing to identical edges (same memory location);
}

int Fan_Order(edge& a, edge& b)
{
/* This routine is used for specialized sorting of the edges of the graph.  It is done
   primerally so certain adjacent edges about a node are available in O(1) time.
```

```
   0  -- Indicates edge (a) is not incident to edge (b) on any common node.
   1  -- Indicates both edges are sourced at the same node.
   2  -- Indicates both edges are targetted at the same node.
   3  -- Indicates edge (a) is entering the node while edge (b) exists.
   4  -- Indicates edge (b) is entering the node while edge (a) exists.
   5  -- Indicates edge (a) and edge (b) are not incident to any common node.
   10 -- Indicates edge (a) is a duplicate of edge (b).  (Yes, a may be compared to a!)
*/
    const node sa = G.source(a), ta = G.target(a), sb = G.source(b), tb = G.target(b);
    double ysa=G[sa].ycoord(), ysb=G[sb].ycoord(), yta=G[ta].ycoord(), ytb=G[tb].ycoord();

    int incidence = 1*(sa==sb)+2*(ta==tb)+3*(ta==sb)+4*(tb==sa); // Incidence Cases.

    switch (incidence) // Determine Incidence Type.
        {
          case 00: return 0;
          case 01: return (+compare(yta,ytb)); // Source Incident
          case 02: return (-compare(ysa,ysb)); // Target Incident
          case 03: return -1; // A in with B out.
          case 04: return +1; // B in with A out.
          case 10: return 0;
        }
}

ostream& operator<<(ostream& O, const node& n) {return O << G[n];} // Print point.
ostream& operator<<(ostream& O, const edge& e) {return O << G.source(e)<< "-"<< G.target(e);}

// ****************** Geometric Assistance functions. ****************

inline segment SEGMENT(edge& e) {return (segment(G[G.source(e)],G[G.target(e)]));}

point Slope_Intercept (edge& e)
// This routine returns a point (mb) representing the slope/y intercept of segment(e)
{
    segment S = SEGMENT(e);
    return (point(S.slope(),S.ycoord1()-S.slope()*S.xcoord1())); // Slope/intercept
}

inline vector Plane(vector Q, vector R, vector S)
// Input: Routine is given 3 non-collinear points in space stored as vectors.
// Caveat: Other routines have not guarenteed the non-collinearity contraint.
// Output: The (a,b,c) values of Ax+Dy+Cz stored as a vector.
{
    vector QR = R - Q, RS = S - R; // These vectors must lie on the plane.
    return vector(QR[1]*RS[2]-QR[2]*RS[1],QR[2]*RS[0]-QR[0]*RS[2],QR[0]*RS[1]-QR[1]*RS[0]);
}


inline double Z(vector& plane,vector& P,point& p)
// Input: A plane [a,b,c], a point on that plane P(x,y,z), and a point on the XY plane.
// Output: The appropriate z coordinate for the projected point that lies on the given plane.
// Technique: Since Ax+By+Cz=k, k = plane*P -- using vector multiplication.  Solve for z
// Technique: in the plane equation but use the point on the XY plane for x and y.
// Notes: This method below was hand-derived using the equation and stuffed into vectors
// Notes: so the computer would do the multiplication.
{
    vector V1(plane[0]/plane[2],plane[1]/plane[2],1); // Normalize the Plane
    vector V2(P[0]-p.xcoord(),P[1]-p.ycoord(),P[2]); // Differnece both points
    return (V1*V2); // Solve for z of the unknown point.
}


void Neighbors(node& n, sortseq<edge,point>& Y, seq_item& un, seq_item& ln)
// Input: A node and sorted sequence of edges keyed on (mb) slope-intercept pair.
// Function: To determine the upper and lower neighbors in Y of the node n.
// Output: The upper neighbor and lower neighbor items.
// Technique: Using (XY) of n, and (mb) of each edge in the Y Structure. We compute
// Technique: D = Y-MX-b where (XY) is from n & (mb) is from each edge in the Y Structure.
// Technique: >1 means the node is above the edge. <1 means the node is below the edge.
// Technique: 0 means the node is on this edge somewhere.
```

```
// Technique: UN is the edge with min positive d. LN is the edge with max negative d.
// Technique: d's are not unique, but do represent directed distance.
{
    seq_item sit; // Iterator.
    un=nil,ln=nil; // Initial neighbor assumptions.
    double mpd=MAXDOUBLE,mnd=-MAXDOUBLE; // min-postive and max-negative
    forall_items(sit,Y)
        {
            const double x = G[n].xcoord(), y = G[n].ycoord();
            const double m = Y.inf(sit).xcoord(), b = Y.inf(sit).ycoord(), d = y-m*x-b;
            if ((d<eps) && (d<=mpd)) {mpd=d; ln=sit;} // This upper is closer to n.
            if ((d>-eps) && (d>=mnd)) {mnd=d; un=sit;} // This lower is close to n.
        }
}

void Determine_Triangle_Edge(double a, double b, double c, double d)
// This routine is successive called BY the delaunay triangulator.  The given values
// are the coordinates of a segment.  These segments are added to the segment list
// in proper directed fashion.
{
    segment s(a,b,c,d),t(c,d,a,b);
    if (compare(s.start(),s.end())==-1) S.append(s); else S.append(t);
}

// ********************** Node/Edge Utility Functions **********************

inline node Min_Node(node& A, node& B) {return ((compare(A,B)==-1) ? A:B};)
inline node Max_Node(node& A, node& B) {return ((compare(A,B)==-1) ? B:A);}

inline bool nonextreme(const node& n)
// Function: To determine iff n is an extreme node of G (is. first or last node.)
// Assumption: G is assumed sorted along the node comparator.
{return ((n != G.first_node()) && (n != G.last_node()));}

// ****************************************************************************
// ************************ Major Graph Functions *****************************
// ****************************************************************************

int Regularize (Ugraph& G)
// Input: A planar straight-line connected graph.  Results not guareenteed otherwise.
// Formal: For every edge e in G, Degree(source(e)) > 1 AND Degree(target(e)) > 1.
// Function: To add edges to all nodes that are not regular without violating planarity.
// Restriction: No vertical edges are allowed.  Other routines will not produce any.
// Side Effects: The nodes of the given graph will be sorted along the node comparator.
// Output: The total number of edges added regularize the given graph.
{
    priority_queue<edge,node> X; // 2e nodes ordered by point. (Dupes ok)
    sortseq<edge,point>       Y; // Status (slope,int) stored as a point.
    stack<pq_item>            X_Rev; // The complete X Structure in reverse.
    node_array<int>           IN(G,0),OUT(G,0); // In/Out Weights of a node.
    seq_item                  un,ln; // Items to hold the upper and lower neighbors.

    int added = -G.number_of_edges(); // Current number of edges. (Trick Saves Variable)

    edge e; forall_edges(e,G) // Initialize the X Structure & Node In/Out Weights.
        {
            const node s = G.source(e), t = G.target(e); OUT[s]++; IN[t]++;
            X.insert(e,s); X.insert(e,t); // Init X_Strucure
        }

    G.sort_nodes(compare); // AFTER Priority Q insertion, ie. skewwed. (30% Increase)

    h_array<seq_item,node> MAX_ABOVE,MAX_BELOW; // Maxima nodes within this interval.
    while (!X.empty()) // First Pass -- NonRegular for Incoming Edges.
        {
            pq_item it = X.find_min(); // Next Considered Event
            X_Rev.push (it); // Place this on the stack for the reverse pass.
            edge e = X.key(it); node n = X.inf(it); X.del_item(it); // Edge/Node & Delete
            const point Pair = Slope_Intercept(e); // Compute slope-intercept pair.
            Neighbors(n,Y,un,ln); // Compute associated upper and lower neighbors @ n.
```

```
            if (G.source(e)==n) // n is a left node.
                {
                    seq_item s = Y.insert(e,Pair); // Store slope/intercept pair.
                    MAX_ABOVE[s] = MAX_BELOW[s] = G.source(e); // Initialize these maxes.

                    // Now compare both neighbors appropriate maximum nodes yet encountered.
                    // The one with maximum x coordinate will be chosen as the connector.
                    // If missing neighbor, assume first node - reduces nil checks.
                    node v_un = (un ? MAX_BELOW[un] : G.first_node());
                    node v_ln = (ln ? MAX_ABOVE[ln] : G.first_node());
                    node best = Max_Node(v_un,v_ln);

                    if ((!IN[n]) && (nonextreme(n))) // Only nonregular nonextremes.
                        {color c=blue; G.new_edge(best,n,c); OUT[best]++; IN[n]++;}
                }

            else Y.del(e); // n is a right node.

            // This node may NOW be the appropriate maximum for the neighbors.
            if ((un) && (compare(MAX_BELOW[un],n)==-1)) MAX_BELOW[un] = n;
            if ((ln) && (compare(MAX_ABOVE[ln],n)==-1)) MAX_ABOVE[ln] = n;
        }

    h_array<seq_item,node>    MIN_ABOVE,MIN_BELOW; // Maximuma nodes within this interval.

    while (!X_Rev.empty()) // Second Pass -- NonRegular for Outgoing Edges.
        {
            pq_item it = X_Rev.pop(); // Get largest next event & Delete.
            edge e = X.key(it); node n = X.inf(it); // Edge/Node Informations
            const point Pair = Slope_Intercept(e); // Compute pair.
            Neighbors(n,Y,un,ln); // Compute associated upper and lower neighbors @ n.

            if (G.target(e)==n) // n is a right node.
                {
                    seq_item s = Y.insert(e,Pair); // Store slope/intercept pair.
                    MIN_ABOVE[s] = MIN_BELOW[s] = G.target(e); // Initialize these mins.

                    // Now compare both neighbors appropriate minimum nodes yet encountered.
                    // The one with minimum x coordinate will be chosen as the connector.
                    // If missing neighbor, assume last node - reduces nil checks.
                    node v_un = (un ? MIN_BELOW[un] : G.last_node());
                    node v_ln = (ln ? MIN_ABOVE[ln] : G.last_node());
                    node best = Min_Node(v_un,v_ln);

                    if ((!OUT[n]) && (nonextreme(n))) // Only nonextremes Qualify.
                        {color c=red; G.new_edge(n,best,c); OUT[n]++; IN[best]++;}
                }

            else Y.del(e); // n is a Left Node.

            // This node may NOW be the appropriate minimum for the neighbors.
            if ((un) && (compare(n,MIN_BELOW[un])==-1)) MIN_BELOW[un] = n;
            if ((ln) && (compare(n,MIN_ABOVE[ln])==-1)) MIN_ABOVE[ln] = n;
        }

    added += G.number_of_edges(); // Determine number added from passes.
    double t; LOG << string("%d edges to regularize. CPU=%f & ",added,t=used_time(Time));
    LOG << string("Rate=%d edges per second.",int(G.number_of_edges()/t)) << endl;
    return added; // # of new edges added.
}


edge_array<int> Chain_Flow (Ugraph& G,int& C)
// Input: A regular planar straight-line connected graph.
// Function: To compute the network-flow weighting of every edge in G.
// Side Effects: Edges of the graph are reorganized (nlogn), see comparison function.
// Output: For all edges e, the number of chains that pass thru e.
// Notes: Modified version of "Computational Geometry" page 51.
// Notes: Concurrent Determination of |C| added 2/13/93 by Stephen Thamel.
```

```
{
    node_array<int> W_in(G,0),W_out(G,0); // Assume no ins or outs.
    edge_array<int> CAP(G,1); // Initial Assumption

    G.sort_edges(Fan_Order); // The Highest Outgoing and Highest Incoming are O(1).
    node n = G.first_node(); // Start at First Node.
    C = G.degree(n); // Assume one chain for each outgoing of first node.

    while ((n = G.succ_node(n)) && (nonextreme(n))) // Go Forward
        {
            edge e;
            int ins = 0, outs = 0; // Assume zero
            edge highest_out = G.last_adj_edge(n); // As per our sorting function.
            forall_adj_edges(e,n) if (G.target(e)==n) W_in[n] += CAP[e],ins++; // IN only
            if (W_in[n] > (outs = G.degree(n)-ins)) CAP[highest_out] = W_in[n] - outs + 1;
        }

    n = G.last_node(); // Start at Last Node.
    while ((n = G.pred_node(n)) && (nonextreme(n))) // Go Backward
        {
            edge e;
            // Since we have all reverse edges in the map, we can say:
            edge highest_in = G.first_adj_edge(n); // As per our sorting function.
            forall_adj_edges(e,n) if (G.source(e)==n) W_out[n] += CAP[e]; // OUT only
            if (W_out[n] > W_in[n]) CAP[highest_in] += W_out[n] - W_in[n];
            if (G.source(highest_in) == G.first_node()) C += -1 + CAP[highest_in];
        }

    G.reset(); // Good idea to reset since we used iterators on the graph.

    LOG << string("%d chains were determined, capacities calculated. ",C);
    LOG << string("CPU=%f",used_time(Time)) << endl;
    return CAP; // The edge capacities.
}


int Generate_PSLG (int vertices, Ugraph& G)
// Input: The number of vertices to generate and a graph to hold the results.
// Function: To generate a randomly connected planar straight-line graph.
// Side Effects: None.
// Output: The maximum value assigned to any coordinate. Useful for window matting.
// Notes: Routine performs a complete Delaunay triangulation.  See LEDA implementation.
// Notes: It then removes some non articulation edges, so the graph does not disconnect.
{
    const int VV = 2*vertices; // Spread the vertices over twice the range.

    // The Delaunay triangulation routine will produce a list of segments.  Its
    // unfortunate, no node information can be included - but se la vi.
    DELAUNAY_TREE<int> Tree; // Delaunay Tree Structure (a la LEDA internal)
    for (int i=1; i<=vertices; i++) Tree.insert(point(1+random()%VV,1+random()%VV),i);
    Tree.trace_triang_edges(Determine_Triangle_Edge); // Verified |S|=#edges

    // Since we have only a list of segments.  We must make nodes at only
    // the "newly" encountered endpoints of any segment.  This avoids duplicate
    // nodes stacked atop one another at the same coordinates.
    segment s; d_array<point,node> NODE; // Point to node. (Not h_array!)
    forall(s,S) // All segments in the segment list, properly directed.
        if (!s.vertical()) // Do not permit vertical segment in the terrain.
            {
                color c = black;  point P = s.start(), Q = s.end();
                if (!NODE.defined(P)) NODE[P] = G.new_node(P); // Not a node yet.
                if (!NODE.defined(Q)) NODE[Q] = G.new_node(Q); // Not a node yet.
                G.new_edge(NODE[P],NODE[Q],c);
            }

    // We now have a complete Delaunay triangulated graph.  In an effort, to
    // activate the regularization routine, we must de-regularize some nodes.
    // We must make sure not to disconnect the graph.  See routine header.
    edge e; list<edge> edges = G.all_edges();
    forall(e,edges) // Successive constraints to legally remove an edge from the graph.
```

```
        if ((G.degree(G.source(e))>2) && (G.degree(G.target(e))>2) && (random(1,100)<40))
            G.del_edge(e); // Make sure does not disconnect.

    const int nn = G.number_of_nodes(), ne = G.number_of_edges();
    LOG << string("%d nodes and %d edges randomly constructed. ",nn,ne);
    LOG << string("CPU=%f",used_time(Time)) << endl;

    return VV; // Largest coordinate possible, nice for matte centering.
}

GRAPH<face,int> Facial_Dual(planar_map& M)
// Input: A planar map M of a graph G.
// Output: A graph representing the dual of the given planar map.
// Technique: For each face of M, a node of H is created in the dual.
// Technique: For each adjacent face of every edge surround f, an edge is created
// Technique: in the dual.  This Dual(#edges) = 2*(#interior edges).
{
    h_array<face,node> Face_to_Node(nil); // Important to Init to Something.
    GRAPH<face,int> Dual; // A graph with face information at the nodes.

    face whiteboard = M.adj_face(M.first_adj_edge(M.first_node()));  // Unbounded Region
    list<face> F = M.all_faces(); F.del_item(F.search(whiteboard)); // Remove Unbounded

    // Make a new node storing the face there inform the hashing function of addition.
    face f; forall(f,F) Face_to_Node[f] = Dual.new_node(f);

    forall(f,F)
        {
            edge e; list<edge> E = M.adj_edges(f); // Get all adjacent edges of face.
            forall(e,E)
                {
                    face g = M.adj_face(M.reverse(e)); // adj(Rev(e)) = adj(f);
                    if (f == g) g = M.adj_face(e); // Sometimes not.
                    if (M.adj_face(e)==M.adj_face(M.reverse(e))) LOG<<"** Self Edge **\n";
                    if (g!=whiteboard) Dual.new_edge(Face_to_Node[f],Face_to_Node[g]);
                }
        }

    return Dual;
}


h_array<point,vector> Facet_Face(Ugraph& G)
// Output: An undirected planar graph - no necessarilly regular (but so in our case).
// Output: A hash table indexed by points to return the 3d vector associated.
// Technique: We traverse the faces via a BFS of the facial dual of the given graph.
// Technique: We use a (3) pass approach to obtain, randomly assign, or compute
// Technique: the z coordinates of any given node in the graph.
// Notes: Must use h_array and not node_array as once M is destroyed any DS keyed on its
// Notes: nodes will no longer function, there is a distinction between M nodes and G nodes.
{
    node n; edge e;

    // These steps below are intrinsic to the various graph data structures supported.
    // In this case, we must copy an undirected graph into a directed graph structure
    // but since directed graphs are not allowed any incoming edges - they must be removed.
    // Then planar maps want directed graphs that have all reverse edges as well and
    // so they must too be added.  Note: Conceptually the reverse(e) may appear like
    // the one just removed from copied graph - but structurally they are stored in
    // different adjacency lists.
    graph H = G; H.make_directed(); H.insert_reverse_edges(); // Required for LEDA maps.

    // We must now sort these graph edges by increasing horizon angles (source,target).
    // Must complete before the planar mapping is formed, or face information will be wrong!
    edge_array<double> angle(H); // Compute the Angle for every edge in Graph.
    forall_edges(e,H) angle[e] = segment(G[H.source(e)],G[H.target(e)]).angle();
    H.sort_edges(angle); // Sort all edges counter-clockwise manner for planar map.

    planar_map M(H); // Directed Planar graph w/ Reverse Edges into a Planar Map.
    GRAPH<face,int> Dual = Facial_Dual(M); // Compute the face dual of the planar map.
```

```
node_array<int> dist(Dual,-1); // The distance array BFS for all nodes in the Dual
list<node> Order = BFS(Dual,Dual.first_node(),dist); // Dual breadth-first ordering.

h_array<point,vector> XYZ(vector(0,0,0)); // Our 3D vector, init to zero.

while (!Order.empty()) // While faces still exist.
    {
        face f = Dual[Order.pop()]; // Top node of order converted to Face information.
        list<node> N = M.adj_nodes(f); // All adjacent nodes of f
        vector Zero(0,0,0),Q,R,S;
        int assigned = 0; // Designates the number of nodes assigned

        // Step #1 - Determine (1,2,3) nodes about f previous assigned by another face.
        forall(n,N)
            {
                vector P = XYZ[G[n]]; // This 3D point @ this node.
                if (P!=Zero) // This node was assigned, store value in Q, R, or S.
                    switch (++assigned) // Keep count.
                        {
                          case 1: Q = P; break;
                          case 2: R = P; break;
                          case 3: S = P; break;
                        }
            }

        // Step #2 - Randomly assign Z values for remainging (1,2,3) nodes.
        forall(n,N)
            {
                const long VV = 2*G.number_of_nodes();
                vector P = XYZ[G[n]]; // This 3D point @ this node.
                if ((P==Zero) && (assigned<3)) // n not assigned, we assign.
                    {
                        double z = 1+random()%VV+1.0/float(random());
                        XYZ[G[n]] = vector(G[n].xcoord(),G[n].ycoord(),z);
                        switch (++assigned)
                            {
                              case 1: Q = XYZ[G[n]]; break;
                              case 2: R = XYZ[G[n]]; break;
                              case 3: S = XYZ[G[n]]; break;
                            }
                    }
            }

        // Step #3 - Traverse any remaining nodes and set Z via plane(1,2,3).
        forall(n,N)
            {
                vector P = XYZ[G[n]]; // This point @ this node.
                if (P==Zero)
                    {
                        vector plane = Plane(Q,R,S);
                        XYZ[G[n]]=vector(G[n].xcoord(),G[n].ycoord(),Z(plane,Q,G[n]));
                    }
            }

        LOG << "Face has N = " << N.size() << " adjacent nodes. ============ \n";
        LOG << "Plane equation Used: " << Plane(Q,R,S) << endl;

        double K = Plane(Q,R,S)*Q; // The k value of this plane.
        forall(n,N)
            {
                vector P = XYZ[G[n]]; // This point @ this node.
                vector plane = Plane(Q,R,S);
                double k = plane*P; // Plane constant.
                vector P1 = vector(G[n].xcoord(),G[n].ycoord(),Z(plane,Q,G[n]));
                LOG << XYZ[G[n]];
                if (XYZ[G[n]]==Q) LOG << " *1*";
                if (XYZ[G[n]]==R) LOG << " *2*";
                if (XYZ[G[n]]==S) LOG << " *3*";
                if (K - k >.5) LOG << "&&&" << P1;
                LOG << endl;
```

```
                }
            }
        return XYZ;
    }
```

```
#include <math.h>
#include <LEDA/list.h>
#include <LEDA/stream.h>
#include "typedefs.h"
#include "graph.h"
#include "view.h"
#include "silloutte.h"

// Our graph is implemented so that the an edge e is accessible on the adjacency
// lists of both source(e) and target(e).  We cannot use a directed graph, since an
// edge will not exist in the adjacency of its target node.
Ugraph                                  G;

// Our viewing parameters for perspective viewing.  We use a four parameter
// homogeneous coordinate system for accurate projections.
double                                  mu,theta,phi,distance; // Viewing
matrix                                  M(4,4);

// Statistical Parameters, Output files and Visual declarations.
ofstream                                OUT,LOG,PLOT; // Output File & Log File Streams
double                                  Time; // CPU clock Value (used by many routines).
window                                  W; // Open Window now for object.

main(int argc, char** argv)
{
    if (argc == 7) // Program n mu theta phi distance outfile
        {
            // Read in Viewing Parameter from Command Line
            mu = atof(argv[2]); theta = atof(argv[3]);
            phi = atof(argv[4]); distance = atof(argv[5]);
            LOG.open(string("%s.log",argv[6]));
            PLOT.open(string("%s.plot",argv[6]));
            Generate_PSLG(atoi(argv[1]),G); // Read #vertices & Generate
            G.write(argv[6]); // Write out terrain.
        }

    else

        {cerr << "TERRAIN: Invalid arguments." << endl; exit(-1);}

    Regularize(G); // Add necessary edges to regularize directed graph.
    int C; edge_array<int> CAP(G); CAP = Chain_Flow(G,C); // Compute edge flow.

    edge e;
    const double VV = 2*G.number_of_nodes();
    W.init(-VV*0.05,VV*1.05,0); // Matted the window for better Visibility.
    forall_edges(e,G)
        {
            if (G.number_of_nodes()<50)
                {
                    point P = G[G.source(e)], Q = G[G.target(e)];
                    string_ostream s; s.clear(); s.precision(4); s << P;
                    W.draw_text(P.translate(.2,.2),s.str());
                    point M((P.xcoord()+Q.xcoord())/2.0,(P.ycoord()+Q.ycoord())/2.0);
                    string_ostream t; t.clear(); t << CAP[e];
                    W.draw_text(M.translate(.2,.3),t.str());
                }
            if (G.source(e)==G.first_node()) W.set_node_width(5); else W.set_node_width(2);
            W.draw_filled_node(G[G.source(e)],red);
            if (G.target(e)==G.last_node()) W.set_node_width(5); else W.set_node_width(2);
            W.draw_filled_node(G[G.target(e)],red);
            W.draw_segment(G[G.source(e)],G[G.target(e)],color(G[e]));
        }

    h_array<point,vector> XYZ = Facet_Face(G); // Determine XYZs via faces.
    list<edge> ** CCL = Complete_Chains(G,CAP,C); // Fill complete chain structure.

    W.read_mouse(); W.clear(); Show_Chains(G,CCL,C);
    M = Transform(mu,theta,phi,distance); W.clear();
    double xmin=0,xmax=0,ymin=0,ymax=0; View_Box(XYZ,M,xmin,xmax,ymin,ymax);
```

```
    W.init(xmin-.2*abs(xmin),xmax+.2*abs(xmax),ymin);
    Show_Projected_Chains(G,CCL,C,XYZ,M);

    memory_clear(); print_statistics(); // Output Memory Statistics.
}
```

```
//  This file contains routines associated with operations on the Silloutte.

#include <LEDA/basic.h>
#include <LEDA/stream.h>
#include <LEDA/array.h>
#include <LEDA/list.h>
#include <LEDA/queue.h>
#include <LEDA/window.h>
#include <LEDA/h_array.h>
#include "typedefs.h"

// ********** External Variables Declared Elsewhere. ****************
// These variables/routines are defined elsewhere.

extern Ugraph G;
extern double mu,theta,phi,distance;
extern float Time;
extern ofstream LOG,PLOT;
extern window W;
extern point Project(const vector&,const matrix&);
extern int compare(node& a, node& b);
extern int Fan_Order(edge& a, edge& b);
extern ostream& operator<<(ostream& O, const node& n);
extern ostream& operator<<(ostream& O, const edge& e);

// ****************** Silloutte Assistance functions. ****************

array<int> hierarchy (int chains)
// This routine computes an array of priorities for all chains.
// Its done via a level-order traversal of a binary search tree, floor(x/2)
// For example 6 chains = 3-2-5-1-4-6.  O(c).  c - # of chains.
{
    array<int> priority(1,chains); // Construct
    queue<int> S,E; S.append(1); E.append(chains); // Starting & Ending chains

    int i = 1;
    while (!S.empty())
        {
            int s = S.pop(), e = E.pop(), mid = (s+e)/2; // Midpoint
            priority[i++] = mid;
            if (mid>s) S.append(s),E.append(mid-1);
            if (mid<e) S.append(mid+1),E.append(e);
        }
    return priority;
}

int maxpriority (const int start,const int weight,array<int>& priority)
// This routine will determine the highest-priority chain in
// the given interval [start,start+capacity].  It does so by
// traversing the array of priorities until it locates a chain
// number that is in the given interval. O(c).  c - # of chains.
{
   if (weight == 1) return start; // Simple case, efficient.

   for (int i=1; i<=priority.high(); i++)
      if ((priority[i] >= start) && (priority[i] < start+weight))
         return priority[i];
}


// ************************************************************************
// ******************** Major Silloutte Functions *************************
// ************************************************************************

list<edge>** Complete_Chains(Ugraph& G, edge_array<int>& CAP, int chains)
// Input: A regular graph, an array of edge weights and the #chains.
// Assumption: None others, edges are sorted as needed - despite duplicating work.
// Note: Deletion of the CCL structure is not performed in this program.
// Note: Since the program ends soon after, the structure is purged anyway.
// Note: Further implementation will require a proper deletion, first the lists then
```

```
// Note: the array of the list pointers.
{
    // Nodes are sorted starting 90 degrees thru 270 degrees then -90 degrees thru 90.
    // So, incoming edges are first fanned downward then outgoings fanned upward.
    G.sort_nodes(compare); G.sort_edges(Fan_Order);

    edge_array<bool> Placed(G,false); // Indicated if edge used.
    node_array<edge> Next(G); // Indicates next upper most edge.
    array<int> binary = hierarchy(chains); // Compute level-order traversal.

    // Memory Management Construction.  LEDA was not cooperating with compound data
    // types.  It was necessary to use a bit of C/C++ pointers and such.
    list<edge> **CCL = new list<edge>*[chains+1]; // An Array of List Pointers
    for(int i=0; i<=chains; i++) CCL[i] = new list<edge>; // Create a list for each.

    node n; forall_nodes(n,G) Next[n] = G.last_adj_edge(n); // Set the successors.

    LOG << chains << " chains induce a hierarchial order of: ";
    for(i=1; i<=chains; i++) LOG << binary[i] << " "; LOG << endl;

    for(i=1; i<=chains; i++) // Traverse all chains.
        {
            LOG << "Pass #" << i << endl;
            node n = G.first_node(); // Always start at first node.
            while (n!=G.last_node()) // Not at the end.
                {
                    edge e = Next[n]; // Get the next appropriate edge for this node.
                    if (!Placed[e]) // If this edge was not yet added to a chain.
                        {
                            int j = maxpriority(i,CAP[e],binary); // Use Maximal Chain #
                            LOG << e << " on [" << i << "," << i+CAP[e]-1;
                            LOG << "] insertted to " << j << endl;
                            CCL[j]->append(e); Placed[e] = true; // Add it, note it.
                        }

                    else LOG << e << " GAP" << endl;

                    if (!--CAP[e]) Next[n] = G.adj_pred(Next[n],n); // Used up.
                    n = G.target(e); // Follow edge monotonically.
                }
        }

    LOG << endl;
    edge e; int k = 0;
    for(i=1; i<=chains; i++)
        {
            LOG << "Chain #" << i << endl;
            if (CCL[i]->size())
                {
                    forall(e,*CCL[i]) LOG << e << endl;  k += CCL[i]->size();
                }
            else LOG << "EMPTY!" << endl;
        }

    if (k!=G.number_of_edges()) LOG << "PROBLEM: |CCL| is not equal to #E(G)!!" << endl;

    LOG << string("%d edges stored in the complete chain list. ",G.number_of_edges());
    LOG << string("CPU=%f",used_time(Time)) << endl;
    return CCL;
}

void Show_Chains(Ugraph& G,list<edge>** CCL,int chains)
// Input: A regular graph, a complete monotonic set of chains and the number of chains.
// Output: Individual display of each chain.
{
    array<int> binary = hierarchy(chains); // Compute corresponding hierarchy.

    for(int i=1; i<=chains; i++) // Traverse total number of chains.
        {
            edge e;
```

```
            int j = binary[i]; // Get nextmost appropriate chains.
            list<edge> L = *CCL[j]; // Get this chain list.
            forall(e,L)
                {
                    if (G.number_of_nodes()<50)
                        {
                            point P = G[G.source(e)], Q = G[G.target(e)];
                            string_ostream s; s.clear(); s.precision(4); s << P;
                            W.draw_text(P.translate(.2,.2),s.str());
                            point M((P.xcoord()+Q.xcoord())/2.0,(P.ycoord()+Q.ycoord())/2.0);
                        }
                    if (G.source(e)==G.first_node()) W.set_node_width(5);
                    else W.set_node_width(2);
                    W.draw_filled_node(G[G.source(e)],red);
                    if (G.target(e)==G.last_node()) W.set_node_width(5);
                    else W.set_node_width(2);
                    W.draw_filled_node(G[G.target(e)],red);
                    W.draw_segment(G[G.source(e)],G[G.target(e)],color(G[e]));
                }
            if (chains<20) W.read_mouse();
        }
}


void View_Box(h_array<point,vector>& XYZ, matrix& M, double& xmin, double& xmax,
              double& ymin,double& ymax)
// Input: The XYZ coordinates of a point in WCS, a transformaiton matrix.
// Output: The minimum and maximum x values produced with these points and transformation.
{
    xmin = MAXDOUBLE, ymin = MAXDOUBLE, xmax = -MAXDOUBLE, ymax = -MAXDOUBLE; // Opposites
    point P; forall_defined(P,XYZ)
        {
            point Q = Project(XYZ[P],M);
            if (Q.xcoord()<xmin) xmin = Q.xcoord();
            if (Q.ycoord()<ymin) ymin = Q.ycoord();
            if (Q.xcoord()>xmax) xmax = Q.xcoord();
            if (Q.ycoord()>ymax) ymax = Q.ycoord();
        }
}


void Show_Projected_Chains(Ugraph& G,list<edge>** CCL,int chains, h_array<point,vector>& XYZ,
                           matrix& M)
{
    array<int> binary = hierarchy(chains); // Compute corresponding hierarchy.

    for(int i=1; i<=chains; i++) // Traverse total number of chains.
        {
            edge e;
            int j = chains-i; // Get nextmost appropriate chains.
            list<edge> L = *CCL[j]; // Get this chain list.
            forall(e,L)
                {
                    point  P = G[G.source(e)], Q = G[G.target(e)];
                    vector R = XYZ[P], S = XYZ[Q]; // 3D locations.
                    point  T = Project(R,M), U = Project(S,M);
                    PLOT << R << endl << S << endl << endl;
                    if (G.source(e)==G.first_node()) W.set_node_width(5);
                    else W.set_node_width(2);
                    W.draw_filled_node(T,red);
                    if (G.target(e)==G.last_node()) W.set_node_width(5);
                    else W.set_node_width(2);
                    W.draw_filled_node(U,red); W.draw_segment(T,U,color(G[e]));
                }
            if (chains<20) W.read_mouse();
        }
}
```

```
// This file contains routines associated with graphical viewing operations.

#include <math.h>
#include <LEDA/matrix.h>
#include <LEDA/segment.h>
#include <LEDA/ugraph.h>
#include <LEDA/window.h>
#include <LEDA/stream.h>
#include "typedefs.h"
#include "graph.h"
#include "silloutte.h"

// *********** External Variables Declared Elsewhere. ***************
// These variables are defined elsewhere, most likely in the main source file.

extern Ugraph G;
extern double mu,theta,phi,distance;
extern float Time;
extern file_ostream LOG;

// ***************** Transformation Assistance functions. ***************

matrix Transform(const double& mu, // Distance eye away from origin
                 const double& theta, // XY angle from origin
                 const double& phi, // Z upward angle from XY plane
                 const double& d) // Projection sheet distance fron eye
// This routine will return the homogeneous matrix that will transform
// a given world coordinate to the view coordinate.  Then, a projective
// transformation is performed.
//
// Mu - The distance away from the origin.
// Theta - Angle swing from X axis ontop of the XY plane.
// Phi - Angle swing z-ward off the XY plane.
{
   double t = theta*acos(-1)/180.0, p = phi*acos(-1)/180.0; // Convert Radians
   double st = sin(t), ct = cos(t), sp = sin(p), cp = cos(p);

   matrix V(4,4),P(4,4),T(4,4),RES(4,4); // View & Projection & Translation

   // Seth H. Rosenzweig & I derived this Tranformation on 2/9/92 9:10pm
   V(0,0) = ct*cp; V(0,1) = -st;  V(0,2) = sp*ct;       V(0,3) = 0;
   V(1,0) = cp*st; V(1,1) = ct;   V(1,2) = sp*st;       V(1,3) = 0;
   V(2,0) = sp;    V(2,1) = 0;    V(2,2) = cp;          V(2,3) = 0;
   V(3,0) = 0;     V(3,1) = 0;    V(3,2) = mu;          V(3,3) = 1;

   P(0,0) = 1;     P(1,1) = 1;    P(2,2) = 1;           P(2,3) = 1/d;
   return V*P; // Initial Projection
}


point Project(const vector& P, const matrix& T)
// Routine will project P thru T and return a screen point(x,y)
{
    matrix    V(1,4);

    V(0,0) = P[0]; V(0,1) = P[1]; V(0,2) = P[2]; V(0,3) = 1;
    V = V*T; // Compute this transformation (Homogeneous Coordinates)
    return point(V(0,0)/V(0,3),V(0,1)/V(0,3)); // Normalize the solution.
}

// ************************************************************************
// ********************** Major View Functions ****************************
// ************************************************************************

// None Currently.  Various viewing and transformation code goes here.
```

```
#ifndef GRAPHING
#define GRAPHING

#include <LEDA/h_array.h>
#include "typedefs.h"

// ******************** Accessible Outside ************************
int Generate_PSLG (int,Ugraph&);
int Regularize (Ugraph&);
edge_array<int> Chain_Flow (Ugraph&,int&);
h_array<point,vector> Facet_Face(Ugraph&);

#endif
```

```
#ifndef SILLOUTTE
#define SILLOUTTE

#include <LEDA/ugraph.h>
#include <LEDA/h_array.h>
#include <LEDA/matrix.h>
#include <LEDA/vector.h>
#include "typedefs.h"

// ******************** Accessible Outside ************************
list<edge>** Complete_Chains(Ugraph&,edge_array<int>&,int);
void Show_Chains(Ugraph&,list<edge>**,int);
void View_Box(h_array<point,vector>&,matrix&,double&,double&,double&,double&);
void Show_Projected_Chains(Ugraph&,list<edge>**,int,h_array<point,vector>&,matrix&);

#endif
```

typedefs.h


```
#ifndef TYPEDEFS
#define TYPEDEFS

#include <LEDA/plane.h>
#include <LEDA/graph.h>
#include <LEDA/ugraph.h>
#include <LEDA/planar_map.h>

typedef GRAPH<point,int> Dgraph;
typedef GRAPH<face,int> Face_Graph;
typedef UGRAPH<point,int> Ugraph;

#endif
```

```
#ifndef VIEWING
#define VIEWING

#include <LEDA/point.h>
#include <LEDA/matrix.h>
#include <LEDA/vector.h>
#include <LEDA/h_array.h>
#include <LEDA/window.h>

// ******************** Accessible Outside *************************
void Show_Graph (Ugraph&,matrix&,h_array<point,vector>&,window&);
void Print_Chains ();
void Graph_Chains (matrix&,window&);
void Graph_Image (matrix&,window&);
matrix Transform (const double&,const double&,const double&,const double&);

#endif
```