CSE 528 Project : Sweeping a 3-D Tetrahedral Complex

Ashish Tiwari

28 Nov 1995

Abstract

This report outlines the design and implementation of an algorithm to sweep a 3 dimensional tetrahedral complex in a particular direction. The problem is considered in its most generality, that is, the tetrahedral complex can be concave and have holes inside it which can themselves be concave. This algorithm can be used to efficiently perform ray-tracing of 3-D objects. The main idea in the algorithm comes from the observation that there are few number of boundary vertices than there are internal vertices in a arbitrary tetrahedral complex. Thus, by suitably initializing our possible set of next vertices that can be visited, we get an efficient algorithm to implement the sweep. Another advantage of using this to do ray-tracing is that it uses much less space.

Chapter 1

Introduction

A 3-dimensional object can be modeled using tetrahedral meshes. With each tetrahedron in the mesh, we can store a scalar/vector field value that represents some physical property of the object we are modeling at that point. Foe example, if the object we are modeling is a cloud, we can create a 3 dimensional tetrahedral complex and have the density stored with each tetrahedron. Since the density of a cloud changes across it, each tetrahedron is likely to have a different value. Now, to scan convert such an object, we should be able to efficiently perform ray-tracing and calculate the values required for each pixel (like the color, lightness etc). Clearly, we should be able to traverse through the tetrahedral mesh in the given viewing direction. The need for an efficient sweep algorithm follows immediately from this observation.

1.1 Background

By a *3 D tetrahedral complex* I mean structures similar to polygonal meshes but with tetrahedrons as its smallest unit rather than polygons. The tetrahedral complex can be concave and there can be holes inside it. The holes too can be concave. *Ray tracing* refers to the process used during scan-converting a 3 dimensional object where you shoot a ray from the pixel in the direction of viewing. Depending on the parts of the object the ray strikes (consider translucent objects because for opaque surfaces only the first point where the ray strikes is relevant), we calculate the value of pixel attributes. *Sweep* refers to the process where one considers the intersection of a plane perpendicular to the given sweep direction (called the *sweep plane*) and the given object. Thus, before starting the actual scan-line scan convert algorithm, performing a sweep of the tetrahedral complex in the given direction can save a lot of later effort.

The procedure outlined above is an improvement over what Giertsen [1] proposed for rendering irregular grids. The basic idea there is to turn the complex 3D ray tracing problem into a 2D problem, thus lowering the complexity of all the geometric computations.

Giersten's algorithm works by calculating a complete scanline at a time. Assuming a parallel projection, the plane perpendicular to the screen passing on a given scanline *cuts* the dataset into a 2D arrangement of polygons. Given that one can efficiently find the 3D cells that are intersected by this plane, one is left with the simpler problem of sorting the 2D cells generated by the intersection. The calculation of 3D cells that are intersected by the *scanline-plane* is done by transforming the dataset and then sorting it in the increasing scanline order. The plane is stored in the form of a *buffer-plane*, a fixed resolution 2D piece of memory that serves as a hashing table for calculating the order of the 2D cells. Once this buffer-plane has been initialized, the whole scanline can be scan converted by using this efficiently.

There are 2 main drawbacks of the Giertsen's approach. First is that in order to generate the sweep, the whole dataset needs to be transformed and sorted into a certain direction. Secondly, the buffer-plane is created each time for a new scanline and moreover the buffer-plane is a *fixed resolution* buffer and hence prone to induce more errors.

The algorithm implemented here is a part of a solution to these two drawbacks. Specifically, this takes care of the first drawback pointed out above. The need to transform the dataset and then sort it has been completely eliminated. The idea comes from the fact that a connected 3D cell complex is already (almost) sorted by the underlying connectivity. For example, for a convex 2D cell complex that has no holes, the sorting in any given direction is equivalent to doing a topological sort on the complex with appropriate weights assigned to the edges. The algorithm implemented here is a neat generalization of this idea.

Essentially therefore, this project consists of implementing an algorithm to sweep a given 3 dimensional tetrahedral complex. In order to show something on the screen, a module was added that displays the current intersection of the sweep plane with the object. As the sweep progresses, the intersection changes and this can be seen on the screen.

1.2 Outline

This report is organized into four chapters. The second chapter deals with defining the problem and then stating the algorithm in a very-high level language. This chapter also contains a brief explanation of the algorithm. In the third chapter, issues pertaining to the implementation of the algorithm are considered. Details of the choices of data-structures made and what are the operations that are supported on those structures are discussed there. Chapter four briefly says about how the program has been organized. Following that, there is a conclusion. Appendix A contains information about the executable file created in the form of a *man* page. Appendix B is a programmers manual with implementation details.

Chapter 2

Algorithm

The problem stated formally is :

Given a arbitrary 3 dimensional tetrahedral complex along with an origin and a direction of sweep, output the vertices of the complex in the order they are intersected by the sweep plane.

The algorithm stated in a high level language is:

Algorithm Sweep

1 Get the list of all boundary vertices (v_1, v_2, ..., v_n) of the complex.

2 Initialize the list of active vertices by inserting all the boundary vertices.
AVL := (v_1, v_2, ..., v_n)

3 Calculate the distance *dist[]* of each vertex in the complex from the given origin along the given direction.

4 while (AVL is not empty) do

  4.1 Choose the vertex , say v_0, with minimum *dist[]* value from the AVL.

  4.2 Delete v_0 from AVL.

  4.3 Update AVL by adding to it all inactive vertices in the complex that are adjacent to v_0.

A naive approach to do the sweep would be to just sort all the vertices by the distance measure defined as the distance of the vertex from the origin along the given sweep direction. Clearly the sweep plane will intersect the vertices in that order. But, the problem with this approach is that it takes a lot of time and space. In the algorithm that has been implemented, the sorting is not done explicitly. The connectivity information from the complex is used to keep only a few vertices in the active vertices list rather than all the vertices that are needed in the naive approach in order to decide the next vertex encountered.

2.1 Correctness

How to argue for the *correctness* of the algorithm? We can claim that the active vertices list contains all potential vertices that can be intersected by the sweep plane next. Proving this claim suffices to prove the correctness of the above mentioned algorithm. So, we prove the claim now. Initially, the vertex that will be touched first will clearly be a boundary vertex. At any intermediate stage, suppose vertex v_i is the next vertex that the sweep plane will intersect. If v_i is an internal vertex then it has tetrahedrons all around it. Therefore, in whichever direction the sweep plane is coming from, the sweep plane would have visited a vertex v_j thats adjacent to v_i. In fact v_j will be on one of those tetrahedrons that are in the negative of sweep direction from the vertex v_i. The second case is when v_i is a boundary vertex. The only way in which none of v_i's neighbours would have been visited will be when all neighbours are in the direction of the sweep direction. Clearly, this implies that v_i is a *boundary* vertex where by boundary we mean the exterior boundary *plus* the holes' boundary.

It must be pointed out here that the origin specified in the input should make sure that the whole complex lies on only *one* side of the sweep plane passing through the origin. If this is not the case, then the distances calculated above and the whole sweep algorithm in general don't make much sense. Also, the program that implements the above algorithm does no consistency checks on the given input tetrahedral complex. If these two constraints are satisfied then the above algorithm is guaranteed to produce the correct output.

In order to display the sweep on the screen, a list of active tetrahedrons is maintained too. This list contains all the tetrahedrons that are intersected by the sweep plane. In each iteration, (inside the while loop above) this list is updated too. Another subtle point is the choice of appropriate $x-$ and $y-$ coordinate axes for displaying the intersection. The $z-$ axis is the direction of sweep. In the implementation the choice for $x-$ and $y-$ axes is arbitrary. The only constraint satisfied is that the choice is made so that the system remains a right-hand rectilinear coordinate system.

Chapter 3

Implementation

In this section we consider some of the implementation issues. The program can be divided into three phases. In the first phase, the input is taken and some pre-processing is done. This includes calculating the distances and boundary vertices. In the second phase, we perform the sweep. The third phase consists of displaying the results. The second and third phase are inter-mixed in the sense that we perform one-step of sweep and then display the intersection, and then repeat this whole process as long as the sweep continues.

3.1 Phase 1

The input is read and organized into the following structures -

1 points[0-MAXPOINTS][0-2], which contains the $x-$, $y-$ and $z-$ coordinates of all points in the complex.

2 tetrahedrons[0-MAXTETRAHEDRONS][0-3], which contains the four indices of the four vertices that make up each tetrahedron.

3 vertex-tetrahedron incidence list[0-MAXPOINTS], which contains a pointer to a list containing indices of tetrahedrons that are incident upon each vertex.

The distance calculation is done easily by finding the dot-product of the difference of the given vertex and origin with the direction vector. The boundary vertices are generated by using the simple observation that a vertex is a boundary vertex if and only if it is a part of an external face. An *external* face is a face that belong to 2 tetrahedrons . Thus, we first mark all faces that belong to 2 different tetrahedron and then the vertices of all unmarked faces are boundary vertices.

3.2 Phase 2

The choice of a good data-structure for implementing the *active vertices list* is crucial for performance of the sweep algorithm. Since the operations that need to be supported are *delete-min* and *insert*, a *binary heap* turns out to be just the right choice. It performs the delete-min step and the insert step in $O(lgn)$ time, where n is the size of the heap. The algorithm ensures the heap-size is controlled and is always less than the total number of points in the complex.

After deleting the minimum distance vertex from the heap, we need to update the heap by adding new vertices to it. Thus, we need the vertex-tetrahedron-incidence list for this. The implementation can be changed and made more efficient by using a vertex-vertex incidence array instead. But since we need the vertex-tetrahedron incidence list for updating the active tetrahedron list used in displaying, this change tends to increase the amount of space required.

At each sweep step, we also need to update the *active tetrahedrons list*. This can be efficiently done by using the list of all tetrahedrons that are incident upon the given visited vertex. This information is available from the vertex-tetrahedron incidence list. A plane can intersect a tetrahedron to form either a triangle or a quadrilateral. Along with each tetrahedron in the active tetrahedron list, we store the number of vertices of that tetrahedron that have been visisted by the sweep plane.

The data-structure chosen for implementing the *active tetrahedrons list* is a *hash table*. The reason for this choice is the observation that we need to perform arbitrary inserts to and deletes from the active tetrahedrons list. Also the number of active tetrahedrons doesn't get to large . The only data-structure that gives good running time for these operations is a hash table. Overflow is handled through lists.

3.3 Phase 3

This phase consists of routines for displaying the intersection of the sweep plane with the tetrahedral complex. This is pretty straight forward. The displaying is done after each sweep step.

Chapter 4

Organization

The program is organised into several files. All the major phases appear in different files. Also, there are different files containing codes for different sub-parts of a phase. The use of global variables has been minimized and the interface between different modules has been kept neat and well-defined.

Chapter 5

Conclusion

The algorithm implemented here accepts as input virtually any 3D tetrahedral complex, which can be concave and have concave holes inside it, and an origin and a direction of sweep, and performs sweep on the 3D complex in the given direction starting from the origin. As it sweeps the complex, information of the active tetrahedrons is accumulated. The sweep is also displayed to the user in the form of 2D planar intersection of the sweep plane with the tetrahedral complex. This algorithm is a part of a larger sweep based rendering algorithm for 3D grids which has been suggested as an improvement over Giertsen's algorithm for ray tracing irregular grids.

Bibliography

[1] C. Giertsen, *Volume Visualization of Sparse Irregular Meshes*, IEEE Computer Graphics and Applications, 1992.

[2] J. Foley et al., *Computer Graphics: Principles and Practice*, Addison Wesley, Mass., 1990.

Appendix A

Manual for Sweep

NAME

sweep - sweep the given tetrahedral complex in the given direction.

SYNOPSIS

sweep filename

DESCRIPTION

sweep reads data from the named file assuming its in the following format :

number of vertices in the tetrahedral complex(n)

number of tetrahedrons in the tetrahedral complex(m)

x-coordinate y-coordinate z-coordinate of point 0

x-coordinate y-coordinate z-coordinate of point 1

⋮

x-coordinate y-coordinate z-coordinate of point n-1

vertex0 vertex1 vertex2 vertex3 of tetrahedron 0

vertex0 vertex1 vertex2 vertex3 of tetrahedron 1

⋮

vertex0 vertex1 vertex2 vertex3 of tetrahedron k-1

eof

Next the program prompts the user to input the direction along which he wants to perform the sweep. Next the user is asked to input the origin. After these steps, the program starts the sweep process and opens a new window where it displays the current intersection of the sweep plane with the tetrahedral complex. There are two buttons on the window. If the *quit* button is clicked on, the program quits and exits. If the *next* button is clicked on, the sweep progresses by one step onto the next vertex. It stops here and refreshes the screen. This process continues till the end of sweep is reached.

OPTIONS

No options are available with this command.

Appendix B

Programmers Manual

B.1 What's in Which File?

The details of the various files and what they contain are as follows:

1 atl.c : This file contains routines for manipulating the *active tetrahedral list*. There are routines for doing *insertion*, *deletion* and *modification* of records in the active tetrahedral list. Note that the list has been implemented as an array of pointers. This is essentially the hash table data-structure.

2 boundary.c : This file contains routines to create the list of all boundary vertices in the given tetrahedral complex.

3 construct-incidence.c : This file initializes the vertex-tetrahedron incidence list given the tetrahedral complex as input.

4 distance.c : This file has routines that calculate the distance of each vertex in the tetrahedral complex from the origin along the given-direction. The calculation is straight forward. First the vector difference of the vertex and origin is found. Then the dot-product of this difference is taken with the direction vector.

5 draw.c : This file has routines that display the intersection of a given plane and a tetrahedral complex on the screen. The *active tetrahedral list* too is given as input so that only those tetrahedrons that are in this list are considered for displaying purposes.

6 heap.c : The routines that perform the standard operations of *delete-min* and *insert* on the heap data-structure reside here. The heap is implemented as an array. If heap[0-n] is the array then the 2 children of node heap[i] are heap[2*i+1] and heap[2*i+2], if they exist. The heap always satisfies the property that the value (of distance) at any node (vertex) is less than the values at both of its children.

7 main.c : This file contains routines that read in the data and the main() function that is called when the program starts execution.

8 sweep.c : This file contains routines that perform one step of the sweep by choosing the next vertex from the heap, deleting it and then updating both the heap and the active tetrahedrons list. The different cases tell us which of the tetrahedrons 4 vertices have been visited and which haven't. The array *vertexflag* has the value 0 if the vertex has not entered the heap at all. The value is 1 if it is in the heap and has a value of 2 of it has entered and then left the heap.

9 transform.c : This file has routines that create the transformation matrix and apply the transformation. The transformation is needed before any point is displayed on the screen because since we want to show the intersection on the sweep plane, our viewing direction is the sweep direction. So suitable changes need to be done. Also note that since just giving the z-direction (the sweep direction) is not sufficient to construct a unique coordinate system, a choice for the x-axis has been made arbitrarily. So its likely that you see figures that are rotated as compared to what you had expected.

10 xtdisplay.c : This file contains the routine that creates the display window and then which calls routines in draw.c to display the intersection. It handles the whole execution once the window has been created.

B.2 Global Variables

Here is a discussion on what the global variables store. It must be pointed out that if the draw / display routines are dispensed with, then all these global variables can be made local to the *main* function. Only the two files, *main.c* and *xtdisplay.c* include these global declarations.

B.2.1 The Complex

The 2D array *p* stores the x,y,z coordinates of all points. p[i][X] gives the x-coordinate of the ith point. Similarly for p[i][Y] and p[i][Z]. The 2D array *t* stores the indices (id's) of the 4 vertices making up each tetrahedron. For eg., t[j][0] stores the id of one of the vertices of tetrahedron j. The array of linked lists *LList* is essentially the vertex-tetradron incidence list. LList[i] is a pointer to the list of tetrahedrons incident upon vertex i.

B.2.2 Active Vertices Table

The variable *heap*, which is a 1D array, stores all *active* vertices, ie the vertices that are potentially next vertices in the sweep. *heapsize* stores the number of vertices that are active. *item* is the vertex id of the vertex last visited by the sweep plane. *vertexflag* is a 1D array that stores the status of each vertex. If vertexflag[i] is 0, it means that the vertex i has not entered the heap (active vertex list) , a value of 1 shows that its in the heap and a value of 2 implies that the vertex has left the heap.

B.2.3 Active Tetrahedrons' Table

The array of linked lists *ATL* is the active tetrahedrons' list. The active tetrahedrons are hashed into the appropriate bucket and inserted in the corresponding list.

B.2.4 Others

The 3 coordinates for the direction vector and the origin are stored in *direction* and *origin* arrays respectively. The *distance* array stores the distance. For eg., distance[i] is the distance of vertex i from the origin along the given direction. The 3 x 3 matrix T is the transformation matrix used before displaying the intersection on the screen.