Geometry Data Handling: Implementation Issues

Dinesh Shikhare
National Centre for Software Technology, Juhu, Mumbai.
May 1999.

Abstract

In this writeup we report the work carried out during the semester to develop software tools for geometry data manipulation. The scope of manipulation of geometric objects reported here is: parsing of various file formats, conversion between file formats, study and usage of a neutral file format, geometry analysis and cleanup functions and a new approach to compression of database for architectural geometry.

1 Introduction

In order to develop any geometry manipulation application, one needs a set of basic geometry processing tools. The aim of the work reported here is build a set of class libraries capable of handling some basic functionalities. These functions are:

1. Reading from and writing to popular geometry data file formats
2. Development of an extensible data structure for modeling geometry and its attributes
3. Conversion between file formats
4. Geometry cleanup functions
5. Set of traversal functions for moving around in the geometry database at various levels of granularity
   (a) traversal among meshes in a scene
   (b) tranversal among triangles in meshes – these traversal are provided in various different forms
6. Topological analysis utilities

Using these basic tools, a geometry compression tool has been developed which promises to give a high degree of compression for architectural databases.

In the following sections, the work done in various categories of geometric data manipulation has been reported. All the ideas presented below have been implemented as a part of the work in progress. The last section of the report describes the future plans for continuing this work to develop more advanced software tools.

2 3D Object Types, attributes

A typical geometry database for animation, virtual walkthoughs of 3D scenes consists of triangle meshes, associated material properties and texture maps. Many geometry data file formats also include smooth curves, surfaces modeled using NURBS, procedural animation elements such as particle systems, meshes with temporal transformation information, lighting conditions and some preset camera positions.

In the work carried out so far, we have considered the following geometry data types and attributes:

1. Triangle meshes
2. Material properties including ambient, diffuse and specular reflectance, shininess, transparency and material emmission.
3. Texture maps
4. Vertex normals for meshes

3 File formats

As a part of this work, we have concentrated on 3 different file formats: OFF, WAK and Alias Wavefront. From all these we only deal with triangle mesh data, material properties, texture mapping and vertex normals. Support for other typese of entities is planned for the future work. We briefly describe these file formats in the following subsections. All these are ASCII file formats, though binary file formats also exist.

3.1 OFF file format

We describe this file format with an example of dodecahron represented in the format. See Figure 1.

The "OFF" header tells us it's a polylist file. The second line in the file tells us that there are 20 vertices, 12 faces, and 30 edges. The next 20 lines give a list of vertices. The last 12 lines specify the faces: the first number is the number of vertices in that face. Since our polyhedron happens to be regular, all faces have the same number of vertices (in this case, 5). The rest of the numbers on the line are zero based indices into the above list of vertices.

In most of the meshes we have handled, there were only three sided polygons (triangles). A polygon having more sides was first triangulated and then used.

3.2 Alias Wavefront file format

This file format is the native database format for an animation and 3D modeling package called Alias Wavefront. This format is very extensive in terms of different kinds of geo-

```
OFF
20 12 30
1.214124 0.000000 1.589309
0.375185 1.154701 1.589309
-0.982247 0.713644 1.589309
-0.982247 -0.713644 1.589309
0.375185 -1.154701 1.589309
1.964494 0.000000 0.375185
0.607062 1.868345 0.375185
-1.589309 1.154701 0.375185
-1.589309 -1.154701 0.375185
0.607062 -1.868345 0.375185
1.589309 1.154701 -0.375185
-0.607062 1.868345 -0.375185
-1.964494 0.000000 -0.375185
-0.607062 -1.868345 -0.375185
1.589309 -1.154701 -0.375185
0.982247 0.713644 -1.589309
-0.375185 1.154701 -1.589309
-1.214124 0.000000 -1.589309
-0.375185 -1.154701 -1.589309
0.982247 -0.713644 -1.589309
5 0 1 2 3 4
5 0 5 10 6 1
5 1 6 11 7 2
5 2 7 12 8 3
5 3 8 13 9 4
5 4 9 14 5 0
5 15 10 5 14 19
5 16 11 6 10 15
5 17 12 7 11 16
5 18 13 8 12 17
5 19 14 9 13 18
5 19 18 17 16 15
```

Figure 1: Dodecahedron in OFF format

metric entities. The complete specification of the file format is beyond the scope of this report. However, we describe here the features of this file format that were used for our work.

The ASCII version of Alias Wavefront format (which typically has .obj extension) consists of single-line statements which define components of the 3D objects. We have used only the geometry and static attributes of triangle mesh objects. The geometry is defined with vertex data and surface element data. Grouping mechanisms are used to represent multiple 3D objects in the same file.

Vertex data:

Vertex data provides coordinates for:

1. Geometric vertices (v):
2. Texture vertices (vt):
3. Vertex normals (vn):

The vertex data is represented by three vertex lists; one for each type of vertex coordinate. A right-hand coordinate system is used to specify the coordinate locations.

The following sample is a portion of an .obj file that contains the three types of vertex information. Sometimes, the geometric vertices may have a fourth coordinate indicating the *weight* of the vertex.

```
v       -5.000000       5.000000       0.000000
v       -5.000000      -5.000000       0.000000
v        5.000000      -5.000000       0.000000
v        5.000000       5.000000       0.000000
vt      -5.000000       5.000000       0.000000
vt      -5.000000      -5.000000       0.000000
vt       5.000000      -5.000000       0.000000
vt       5.000000       5.000000       0.000000
vn       0.000000       0.000000       1.000000
vn       0.000000       0.000000       1.000000
vn       0.000000       0.000000       1.000000
vn       0.000000       0.000000       1.000000
```

When vertices are loaded into the Advanced Visualizer, they are sequentially numbered, starting with 1. These reference numbers are used in element statements.

Elements:

For polygonal geometry, the element types available in the .obj file are:

1. Lines (l): This is a polygonal geometry statement. It specifies a line and its vertex reference numbers. You can optionally include the texture vertex reference numbers. Although lines cannot be shaded or rendered, they are used by other Advanced Visualizer programs. The reference numbers for the vertices and texture vertices must be separated by a slash (/). There is no space between the number and the slash.

```
l  v1/vt1   v2/vt2   v3/vt3 . . .
```

v is a reference number for a vertex on the line. A minimum of two vertex numbers are required. There is no limit on the maximum. Positive values indicate absolute vertex numbers. Negative values indicate relative vertex numbers. vt is an optional argument. vt is the reference number for a texture vertex in the line element. It must always follow the first slash.

2. Face (f): This is a solid polygonal geometry statement. It specifies a face element and its vertex reference number. You can optionally include the texture vertex and vertex normal reference numbers. The reference numbers for the vertices, texture vertices, and vertex normals must be separated by slashes (/). There is no space between the number and the slash.

```
f  v1/vt1/vn1   v2/vt2/vn2   v3/vt3/vn3 . . .
```

v is the reference number for a vertex in the face element. A minimum of three vertices are required. vt is an optional argument. vt is the reference number for a texture vertex in the face element. It always follows the first slash. vn is an optional argument. vn is the reference number for a vertex normal in the face element. It must always follow the second slash. Face elements use surface normals to indicate their orientation. If vertices are ordered counterclockwise around the face, both the face and the normal will point toward the viewer. If the vertex ordering is clockwise, both will point away from the viewer. If vertex normals are assigned, they should point in the general direction of the surface normal, otherwise unpredictable results may occur.

If a face has a texture map assigned to it and no texture vertices are assigned in the f statement, the texture map is ignored when the element is rendered.

Examples of geometry:

This example shows a square that measures two units on each side and faces in the positive direction (toward the camera). Note that the ordering of the vertices is counterclockwise. This ordering determines that the square is facing forward.

```
v 0.000000 2.000000 0.000000
v 0.000000 0.000000 0.000000
v 2.000000 0.000000 0.000000
v 2.000000 2.000000 0.000000
f 1 2 3 4
```

The next example is a a cube that measures two units on each side. Each vertex is shared by three different faces.

```
v 0.000000 2.000000 2.000000
v 0.000000 0.000000 2.000000
v 2.000000 0.000000 2.000000
v 2.000000 2.000000 2.000000
v 0.000000 2.000000 0.000000
v 0.000000 0.000000 0.000000
v 2.000000 0.000000 0.000000
v 2.000000 2.000000 0.000000
f 1 2 3 4
f 8 7 6 5
f 4 3 7 8
f 5 1 4 8
f 5 6 2 1
f 2 6 7 3
```

Grouping:

There are four statements in the .obj file to help you manipulate groups of elements:

1. Group name statements are used to organize collections of elements and simplify data manipulation for operations in Model.

2. Smoothing group statements let you identify elements over which normals are to be interpolated to give those elements a smooth, non-faceted appearance. This is a quick way to specify vertex normals.

3. Merging group statements are used to identify free-form elements that should be inspected for adjacency detection. You can also use merging groups to exclude surfaces which are close enough to be considered adjacent but should not be merged.

4. Object name statements let you assign a name to an entire object in a single file.

All grouping statements are state-setting. This means that once a group statement is set, it applies to all elements that follow until the next group statement.

Syntax:

```
g group_name1 group_name2 . . .
```

Specifies the group name for the elements that follow it. You can have multiple group names. If there are multiple groups on one line, the data that follows belong to all groups. Group information is optional. group_name is the name for the group. Letters, numbers, and combinations of letters and numbers are accepted for group names. The default group name is “default.”

```
s group_number
```

Sets the smoothing group for the elements that follow it. If you do not want to use a smoothing group, specify off or a value of 0. To smooth polygonal geometry for rendering with Image, it is sufficient to put elements in some smoothing group. However, vertex normals override smoothing information for Image.

group_number is the smoothing group number. To turn off smoothing groups, use a value of 0 or off. Polygonal elements use group numbers to put elements in different smoothing groups. For free-form surfaces, smoothing groups are either turned on or off; here is no difference between values greater than 0.

```
mg group_number res
```

Sets the merging group and merge resolution for the free-form surfaces that follow it. If you do not want to use a merging group, specify off or a value of 0. Adjacency detection is performed only within groups, never between groups. Connectivity between surfaces in different merging groups is not allowed. Surfaces in the same merging group are merged together along edges that are within the distance res apart.

group_number is the merging group number. To turn off adjacency detection, use a value of 0 or off. res is the maximum distance between two surfaces that will be merged together. The resolution must be a value greater than 0. This is a required argument only when using merging groups.

```
o object_name
```

Optional statement; it is not processed by any Wavefront programs. It specifies a user-defined object name for the elements defined after this statement. object_name is the user-defined object name. There is no default.

Example:

The following example is a cube with each of its faces placed in a separate group. In addition, all elements belong to the group cube.

```
v 0.000000 2.000000 2.000000
v 0.000000 0.000000 2.000000
v 2.000000 0.000000 2.000000
v 2.000000 2.000000 2.000000
v 0.000000 2.000000 0.000000
v 0.000000 0.000000 0.000000
v 2.000000 0.000000 0.000000
v 2.000000 2.000000 0.000000
# 8 vertices
g front cube
f 1 2 3 4
g back cube
f 8 7 6 5
g right cube
```

```
f 4 3 7 8
g top cube
f 5 1 4 8
g left cube
f 5 6 2 1
g bottom cube
f 2 6 7 3
# 6 elements
```

3.3 WAK file format

This is a neutral file format that has been used at NCST for a virtual walkthough application in a very large digital model of a heritage site. This format consists of nodes which describe various aspects of 3D geometric scene. These nodes are:

1. **Material count:** This node specifies the total number of material definitions to follow in the scene file.
2. **Material node:** This node describes material properties of bundled in a structure. It has the following components in it:
   (a) material name (a string)
   (b) diffuse reflectity (r, g, b components)
   (c) ambient reflectivity (r, g, b components)
   (d) specular reflectivity (r, g, b components)
   (e) shininess (a scalar)
   (f) emmission (a scalar)
   (g) light cone (a scalar)
   (h) direction (a vector with 3 components)
3. **Mesh count:** This node specifies the total number of meshe definitions to follow in the scene file.
4. **Mesh node:** A mesh node specifies a mesh with all its geometric definition and association with its attributes. It has the following components:
   (a) mesh name (a string)
   (b) mesh material (a string, refering to a material node)
   (c) texture map file name (a string)
   (d) vertex list (a sequence of x-, y-, z- coordinates of vertices)
   (e) vertex normals (a sequence of vertex normals, each having 3 components)
   (f) triangle list (list of triangles, each defined using 3 integer references to the vertex list and implicit reference to vertex normal)

(g) bounding sphere defined using 4 scalars - three position vector elements and a radius.

Example:

```
# WALKER FILE FORMAT v2.0 (c) NCST

materials 1

material "dik-floor" {
    ambient 0.898039 0.898039 0.898039
    diffuse 0.898039 0.898039 0.898039
    specular 0 0 0
    shininess 0
    transparency 100
    emmission 1
    lightcone 0 0 0 0
}

meshes 1

mesh dik-gflr-f000 {
    material "dik-floor"
    texture 0 "dik grou.bmp" 8
        2.49795 2.49795
        -0.49795 2.49795
        -0.497951 -0.49795
        2.49795 -0.497951
        2.49795 2.49795
        2.49795 -0.497951
        -0.497951 -0.49795
        -0.49795 2.49795
    vertices 8
        -1182.92 -2807.53 62.7297
        -1522.92 -2807.53 62.7297
        -1522.92 -2467.53 62.7297
        -1182.92 -2467.53 62.7297
        -1182.92 -2807.53 62.9797
        -1182.92 -2467.53 62.9797
        -1522.92 -2467.53 62.9797
        -1522.92 -2807.53 62.9797
    normals 8
        0.000367647 -0.000735294 -1
        -0.00147059 -0.000735293 -0.999999
        -0.000367647 0.000735294 -1
```

```
        0.00147059 0.000735293 -0.999999
        0.000735294 -0.000367647 1
        0.000735293 0.00147059 0.999999
        -0.000735294 0.000367647 1
        -0.000735293 -0.00147059 0.999999
    trilist 12
        0 1 2
        2 3 0
        4 5 6
        6 7 4
        5 3 2
        2 6 5
        4 0 3
        3 5 4
        7 1 0
        0 4 7
        6 2 1
        1 7 6
    sphere -1352.92 -2637.53 62.8547 240.416
}
```

4 Parsing of geometry files

For reading in geometry data, we have developed parsers using Lex and YACC as parser generator tools. These tools help in defining a higher level definition of the grammar of the data defined in the file format and generate a parser automatically. Lex (lexical analyser) generates a piece of code that scans the input stream and detects tokens corresponding to keywords, identifiers and numeric data (integers and real numbers). The tagged token are passed on to the parser generated by YACC (Yet Another Compiler Compiler) which looks for syntactic structures in the stream of tokens and invokes user supplied code for processing each syntactic structure.

The input to `lex` program is a set of regular expressions corresponding to tokens for reserved words, identifiers and numeric data. The output of `lex` is C functions which carry out lexical analysis of the input file. The input to YACC is a description file which describes the grammar of the file format in BNF-like notation. For terminal nodes in the grammar definition, it is possible to specify functions to be invoked for actions to translate the input data into a data structure or into another output format. The ouput of YACC is a set of C functions which carry out grammar matching and invocation of corresponding user supplied action functions.

Below we list the lex and yacc a part of the specifications for the WAK file format.

4.1 Lex specs for WAK file format:

```
integer     ([+-]?[0-9]+)
dreal       ([+-]?[0-9]*"."[0-9]+)
ereal       ([+-]?[0-9]*"."[0-9]+[eE][+-]?[0-9]+)
real        {dreal}|{ereal}
nl          \n

%%
[ \t]           ;

{integer}       {
                    sscanf(yytext, "%d", &yylval.integer);
                    return INTEGER;
                }

{real}          {
                    sscanf(yytext, "%lf", &yylval.real);
                    return REAL;
                }

{nl}            {
                    extern int lineNo;
                    lineNo++;
                }

#.*             ;

meshes          return MESHES;
mesh            return MESH;
materials       return MATERIALS;
material        return MATERIAL;
texture         return TEXTURE;
vertices        return VERTICES;
normals         return NORMALS;
trilist         return TRILIST;
sphere          return SPHERE;
ambient         return AMBIENT;
diffuse         return DIFFUSE;
specular        return SPECULAR;
shininess       return SHININESS;
transparency    return TRANSPARENCY;
emmission       return EMMISION;
lightcone       return LIGHTCONE;
```

```
[a-z][-a-zA-Z0-9\_\./]* {
                    strcpy(yylval.id,yytext);
                    return ID;
                }

.               return yytext[0];
%%
```

4.2 YACC specs for WAK file format:

```
%union {
    double  real;
    int     integer;
    char    id[STRLEN];
}

%token COMMENT
%token MESHES
%token MESH
%token MATERIAL
%token TEXTURE
%token VERTICES
%token NORMALS
%token TRILIST
%token SPHERE
%token MATERIALS
%token AMBIENT
%token DIFFUSE
%token SPECULAR
%token SHININESS
%token TRANSPARENCY
%token EMMISION
%token LIGHTCONE

%token <real> REAL
%token <integer> INTEGER
%token <id> ID

%%
wakscene:       /* nothing */
                | nodes
                    { postProcessing(); }

nodes:          /* nothing */
```

```
                | node nodes

node:           /* nothing */
                | materialsLine
                | materialNode
                | meshesLine
                | meshNode

coordValue:     REAL
                    { recordNextCoordValue($1); }
                | INTEGER
                    { recordNextCoordValue($1); }

materialsLine:  MATERIALS INTEGER
                    { addMaterialsNode($2); }

materialNode:   MATERIAL quotedId '{' materialNodeDefs '}'
                    { addMaterialNode(); }

quotedId:       '"' idlist '"'

idlist:         oneId
                | oneId idlist

oneId:          ID
                    { buildCompositeId($1); }

materialNodeDefs:   /* nothing */
                | matNodeDefItem materialNodeDefs

matNodeDefItem: AMBIENT coordValue coordValue coordValue
                    { recordAmbientField(); }
                | DIFFUSE coordValue coordValue coordValue
                    { recordDiffuseField(); }
                | SPECULAR coordValue coordValue coordValue
                    { recordSpecularField(); }
                | SHININESS coordValue
                    { recordShininessField(); }
                | TRANSPARENCY coordValue
                    { recordTransparencyField(); }
                | EMMISION coordValue
                    { recordEmissionField(); }
                | LIGHTCONE
                    coordValue coordValue coordValue coordValue
                    { recordLightconeField(); }
```

```
meshesLine:     MESHES INTEGER
                    { addMeshesNode($2); }

meshNode:       meshNodeBegin '{' meshDetails '}'
                    { addMeshToDatabse(); }

meshNodeBegin:  MESH oneId
                    { registerMeshName(); }

meshDetails:    /* nothing */
                | meshDefItem meshDetails

meshDefItem:    MATERIAL quotedId
                    { registerMeshDetMaterial(); }
                | TEXTURE INTEGER quotedId INTEGER coords
                    { registerMeshDetTexCoord($2, $4); }
                | VERTICES INTEGER coords
                    { registerMeshDetVertexCoord($2); }
                | NORMALS INTEGER coords
                    { registerMeshDetNormals($2); }
                | TRILIST INTEGER coords
                    { registerMeshDetTriIndices($2); }
                | SPHERE
                    coordValue coordValue coordValue coordValue
                    { registerMeshDetBndSphere(); }

coords:         /* nothing */
                | coordValue coords

%%
```

5 Neutral data-structures

The motivation for a neutral data structure for triangle-based geometry database is as follows. There are multiple file formats describing various kinds of attributes for geometry. Catering to all these types of attributes all the time consumes too much memory. Also various algorithms that operate on the geometry require some general purpose mechanism of attaching temporary information in the course of processing. Space for all this can not be allocated statically. So much data may not be needed for some algorithms. An algorithm that will be developed in the future and file formats that will be used in future may have more specialised requirements. Only the related code will know about such requirements.

This led us to design a plug-in architecture for building a data-structure that has a core geometry in the form of a mesh. The basic structure consists of a set of C++ classes for

holding vertices, triangles and meshes. A Mesh consists of a list of vertices and a list of triangles. Each vertex has three real numbers describing the coordinates in 3-space. Each triangle has three integer indices into the list of vertices. A 3D scene may have multiple such meshes. To accomodate attributes at vertex, triangle mesh, and scene levels, a plug-in mechanism is implemented. A generic "Attrib" base-class is developed to hold a list of attributes at each level. This class has a name associated with it. The details of value to be associated with the name is left to the implementor of the derived classes. Specialised classes are derived from Attrib and associated with vertex/triangle/mesh/scene by creating instances at run-time.

For example, to load WAK file, list of materials is built at scene level, vertex normals, and texture coordinates are attached to the vertices, texture map image is attached to the meshes. This attachment is done in the form of plugging in objects of derived classes of the baseclass Attrib.

Specialised algorithms that need to attach some data with vertices, triangles and meshes, create their own attribute classes and attach instance of the class with a name. The algorithms themselves are expected to do a cleanup operation after the work is done and the auxiliary data is no more needed.

6 Algorithms

Towards the development of a core toolkit for geometry based manipulation, we have implemented a large number of geometric and topological queries that are always needed for many applications. The utility functions are listed below:

1. Given a vertex return the triangles connected to it
2. Given a vertex return the vertices connected to it
3. Return the three neighbours of a given triangle
4. Obtain the boundaries of an open surface
5. Orient a triangulated 2-manifold
6. Flip the orientation of the given triangulated surface
7. Return the list of boundary vertices for a given mesh
8. Determine if the given surface is closed or not
9. Return the normal of the given triangle
10. Return the area of the triangulated surface
11. Merge two surfaces along the boundaries
12. Determine the convex hull of a set of 2D points.

13. Intersect two triangles and determine the line segment of intersection

14. Boolean operations on axis aligned boxes

Note that many of the above algorithms are non-trivial to implement. Typically they are difficult because of the involvement of tolerance based checking at various stages in the algorithm. Some other algorithms have also been developed in the course of this work. We describe them in a little more details.

6.1 Cleaning up of geometry

In many cases, the triangulation carried out by modelers to tessellate smooth surface 3D objects creates some unwanted geometry. This unwanted geometry manifests in the form of unreferenced vertices, degenerate triangles, reapeated vertices, overlapping geometry, and so on. This geometry is in general useless from almost any processing point of view. Also, its being there in the database hogs memory and processing time. Hence it is important to detect and remove it.

6.2 Topological Analysis

Topological characterization of a given mesh is often useful from the point of view of matching shapes. Geometric features that do not contribute to important topological features like corners, edges, loops and faces should be filtered out first. The remaining geometric data is then used to determine the connectivity information between corners. Having obtained vertices and edges, loops (ordered sequence of edges) are obtained. Finally, the faces bounded by loops are determined. Many application are based on such topological analysis of a given triangulated surface. Notably, shape matching, topological simplification of 3D meshes and topological analysis for CSG decomposition.

6.3 Geometry Compression

As a part of the ongoing activity, we have developed an algorithm for compression of 3D scenes consisting of triangulated surfaces. The tranditional byte stream compression algorithms can always be superseded by special algorithms that can use the knowledge about 3D geometry in the compression strategy.

The existing mesh compression algorithms make a naive assumption that a triangle mesh can always be restructured into long triangle strips so that the strips can then be efficiently encoded. In practice, the architectural databases designed in CAD packages, which also have texture maps, cannot usually be converted into long triangle strips. Hence the previously published techniques can not give high compression ratios.

We propose a dictionary-based geometry compression scheme. In this scheme, a dictionary of geometric shapes in constructed from the given database of geometric scene. Only the first instance of a geometry mesh is stored in the dictionary with all its details. All other instances of similar meshes are stored as a reference to the mesh inserted in the

dictionary and the difference is encoded such that during uncompression, the meshes can be reconstructed. The differential coding includes rigid body transformations, name and "minor" topological differences. This scheme can give a high degree of compression for scenes with many repeated shapes.

This scheme can be fine-tuned to look for and match various kinds of geometric and topological features in the geometric database. This is a part of the work in progress.

7 Conclusion and Future work

In this report we listed the work carried out in this semester towards building of a toolkit of geometry handling tools. Concrete implementation of geometry database import and export functionality has been developed. An extensible data-structure is developed to handle current and future development work over triangle-mesh based geometry databases.

The current work concentrates in various schemes for the compression of large and detailed geometry database. In the process, we intend further developing the geometry and topology toolkit's scope and functionality.