Building Models

CS 432 Interactive Computer Graphics
Prof. David E. Breen
Department of Computer Science

Objectives

- Introduce simple data structures for building polygonal models
  - Vertex lists
  - Edge lists
- Deprecated OpenGL vertex arrays

Representation of 3D Transformations

- Z axis represents depth
- Right Handed System
  - When looking “down” at the origin, positive rotation is CCW
- Left Handed System
  - When looking “down”, positive rotation is in CW
  - More natural interpretation for displays, big z means “far”

Representing a Mesh

- There are 8 nodes and 12 edges
  - 5 interior polygons
  - 6 interior (shared) edges
- Each vertex has a location \( v_i = (x_i, y_i, z_i) \)

Simple Representation

- Define each polygon by the geometric locations of its vertices
- Leads to OpenGL code such as
  ```cpp
  vertex[i] = vec3(x1, x1, x1);
  vertex[i+1] = vec3(x6, x6, x6);
  vertex[i+2] = vec3(x7, x7, x7);
  i+=3;
  ```
- Inefficient and unstructured
  - Consider moving a vertex to a new location
  - Must search for all occurrences

Inward and Outward Facing Polygons

- The order \( \{v_1, v_2, v_3\} \) and \( \{v_6, v_7, v_1\} \) are equivalent in that the same polygon will be rendered by OpenGL but the order \( \{v_1, v_2, v_3\} \) is different
- The first two describe outwardly facing polygons
- Use the right-hand rule = counter-clockwise encirclement of outward-pointing normal
- OpenGL can treat inward and outward facing polygons differently
Geometry vs Topology

- Generally it is a good idea to look for data structures that separate the geometry from the topology
  - Geometry: locations of the vertices
  - Topology: organization of the vertices and edges
- Example: a polygon is an ordered list of vertices with an edge connecting successive pairs of vertices and the last to the first
  - Topology holds even if geometry changes

Vertex Lists

- Put the geometry in an array
- Use pointers from the vertices into this array
- Introduce a polygon list

Shared Edges

- Vertex lists will draw filled polygons correctly but if we draw the polygon by its edges, shared edges are drawn twice
  - Can store mesh by edge list

Edge List

- Note polygons are not represented

Rotating Cube

- Full example
- Model Colored Cube
- Use 3 button mouse to change direction of rotation
- Use idle function to increment angle of rotation

Draw cube from faces

```
void colorcube()
{
  quad( 1, 0, 3, 2 );
  quad( 2, 3, 7, 6 );
  quad( 3, 0, 4, 7 );
  quad( 6, 5, 1, 2 );
  quad( 4, 5, 6, 7 );
  quad( 5, 4, 0, 1 );
}
```

Note that vertices are ordered so that we obtain correct outward facing normals
Cube Vertices

// Vertices of a unit cube centered at origin
// sides aligned with axes
point4 vertices[8] = {
    point4(-0.5, -0.5,  0.5, 1.0),
    point4(-0.5,  0.5,  0.5, 1.0),
    point4( 0.5,  0.5,  0.5, 1.0),
    point4( 0.5, -0.5,  0.5, 1.0),
    point4(-0.5, -0.5, -0.5, 1.0),
    point4(-0.5,  0.5, -0.5, 1.0),
    point4( 0.5,  0.5, -0.5, 1.0),
    point4( 0.5, -0.5, -0.5, 1.0)
};

Colors

// RGBA colors
color4 vertex_colors[8] = {
    color4(0.0, 0.0, 0.0, 1.0), // black
    color4(1.0, 0.0, 0.0, 1.0), // red
    color4(0.0, 1.0, 0.0, 1.0), // yellow
    color4(0.0, 0.0, 1.0, 1.0), // blue
    color4(1.0, 0.0, 1.0, 1.0), // magenta
    color4(1.0, 1.0, 0.0, 1.0), // white
    color4(0.0, 1.0, 1.0, 1.0), // cyan
};

Quad Function

// quad generates two triangles for each face and assigns colors
// to the vertices
int Index = 0;
void quad(int a, int b, int c, int d)
{
    colors[Index] = vertex_colors[a]; points[Index] = vertices[a]; Index++;
    colors[Index] = vertex_colors[b]; points[Index] = vertices[b]; Index++;
    colors[Index] = vertex_colors[c]; points[Index] = vertices[c]; Index++;
    colors[Index] = vertex_colors[d]; points[Index] = vertices[d]; Index++;
}

Color Cube

// generate 12 triangles: 36 vertices and 36 colors
void colorcube()
{
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}

Initialization I

void init()
{
    colorcube();
    // Create a vertex array object
    GLuint vao;
    glGenVertexArrays( 1, &vao );
    glBindVertexArray( vao );
}

Initialization II

// Create and initialize a buffer object
GLuint buffer;
glGenBuffers( 1, &buffer );
glBindBuffer( GL_ARRAY_BUFFER, buffer );
glBufferData( GL_ARRAY_BUFFER, sizeof(points) + sizeof(colors), NULL, GL_STATIC_DRAW );
glBufferSubData( GL_ARRAY_BUFFER, 0, sizeof(points), points );
glBufferSubData( GL_ARRAY_BUFFER, sizeof(points), sizeof(colors), colors );
// Load shaders and use the resulting shader program
GLuint program = InitShader( "vshdrcube.glsl", "fshdrcube.glsl" );
gUseProgram( program );
Initialization III

// set up vertex arrays
GLuint vPosition = glGetUniformLocation( program, "vPosition" );
glEnableVertexAttribArray( vPosition );
glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0) );

GLuint vColor = glGetUniformLocation( program, "vColor" );
glEnableVertexAttribArray( vColor );
glVertexAttribPointer( vColor, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(sizeof(points)) );

Glint thetaLoc = glGetUniformLocation( program, "theta" );

Display Callback

void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glUniform3fv( thetaLoc, 1, theta );
    glDrawArrays( GL_TRIANGLES, 0, NumVertices );
    glutSwapBuffers();
}

Mouse Callback

void mouse( int button, int state, int x, int y )
{
    if ( state == GLUT_DOWN )
    {
        switch( button )
        {
            case GLUT_LEFT_BUTTON:    axis = Xaxis;  break;
            case GLUT_MIDDLE_BUTTON:  axis = Yaxis;  break;
            case GLUT_RIGHT_BUTTON:   axis = Zaxis;  break;
        }
    }
}

Idle Callback

void idle( void )
{
    theta[axis] += 0.01;
    if ( theta[axis] > 360.0 )
    {
        theta[axis] -= 360.0;
    }
    glutPostRedisplay();
}

Objectives

• Introduce the classical views
• Compare and contrast image formation by computer with how images have been formed by architects, artists, and engineers
• Learn the benefits and drawbacks of each type of view
Classical Viewing

- Viewing requires three basic elements
  - One or more objects
  - A viewer with a projection surface
  - Projectors that go from the object(s) to the projection surface
- Classical views are based on the relationship among these elements
  - The viewer picks up the object and orients it how she would like to see it
- Each object is assumed to be constructed from flat principal faces
  - Buildings, polyhedra, manufactured objects

Planar Geometric Projections

- Standard projections project onto a plane
- Projectors are lines that either
  - Converge at a center of projection
  - Are parallel
- Such projections preserve lines
  - But not necessarily angles
- Nonplanar projections are needed for applications such as map construction

Classical Projections

- Front elevation
- Elevation oblique
- Plan oblique
- Isometric
- One point perspective
- Two point perspective
- Three point perspective

Perspective vs Parallel

- Computer graphics treats all projections the same and implements them with a single pipeline
- Classical viewing developed different techniques for drawing each type of projection
- Fundamental distinction is between parallel and perspective viewing even though mathematically parallel viewing is the limit of perspective viewing

Taxonomy of Planar Geometric Projections

- Planar geometric projections
  - Parallel
  - Perspective
    - Multiview
      - Axonometric
    - Oblique
      - 1 point
      - 2 point
      - 3 point
  - Orthographic
    - Isometric
    - Dimetric
    - Trimetric

Perspective Projection

- Object
- Projector
- COF
- Projection plane
Parallel Projection

Orthographic Projection

Multiview Orthographic Projection

Advantages and Disadvantages

Axonometric Projections

Types of Axonometric Projections

- Preserves both distances and angles
  - Shapes preserved
  - Can be used for measurements
    - Building plans
    - Manuals
- Cannot see what object really looks like because many surfaces hidden from view
  - Often we add the isometric
Advantages and Disadvantages

- Lines are scaled (foreshortened) but can find scaling factors
- Lines preserved but angles are not preserved
- Projection of a circle in a plane not parallel to the projection plane is an ellipse
- Can see three principal faces of a box-like object
- Some optical illusions possible
- Parallel lines appear to diverge
- Does not look real because far objects are scaled the same as near objects
- Used in CAD applications

Oblique Projection

Arbitrary relationship between projectors and projection plane

Advantages and Disadvantages

- Can pick the angles to emphasize a particular face
- Architecture: plan oblique, elevation oblique
- Angles in faces parallel to projection plane are preserved while we can still see “around” side
- In physical world, cannot create with simple camera; possible with bellows camera or special lens (architectural)

Perspective Projection

Projectors converge at center of projection

Vanishing Points

- Parallel lines (not parallel to the projection plane) on the object converge at a single point in the projection (the vanishing point)
- Drawing simple perspectives by hand uses these vanishing point(s)

Three-Point Perspective

- No principal face parallel to projection plane
- Three vanishing points for cube
Two-Point Perspective

- On principal direction parallel to projection plane
- Two vanishing points for cube

One-Point Perspective

- One principal face parallel to projection plane
- One vanishing point for cube

Advantages and Disadvantages

- Objects further from viewer are projected smaller than the same sized objects closer to the viewer (*diminution*)
  - Looks realistic
- Equal distances along a line are not projected into equal distances (*nonuniform foreshortening*)
- Angles preserved only in planes parallel to the projection plane
- More difficult to construct by hand than parallel projections (but not more difficult by computer)

Taxonomy of Planar Geometric Projections

- Planar geometric projections
  - Parallel
  - Perspective
  - Multiview
    - Axonometric
    - Oblique
  - Orthographic
  - Isometric
  - Dimetric
  - Trimetric
  - 1 point
  - 2 point
  - 3 point