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
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

- Consider 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 WebGL code such as

```javascript
vertex.push(vec3(x1, y1, z1));
vertex.push(vec3(x6, y6, z6));
vertex.push(vec3(x7, y7, z7));
```

• Inefficient and unstructured
  - Consider moving a vertex to a new location
  - Must search for all occurrences
Inward and Outward Facing Polygons

• The order \( \{v_1, v_6, v_7\} \) and \( \{v_6, v_7, v_1\} \) are equivalent in that the same polygon will be rendered by WebGL but the order \( \{v_1, v_7, v_6\} \) is different.
• The first two describe *outwardly facing* polygons.
• Use the *right-hand rule* = counter-clockwise encirclement of outward-pointing normal.
• WebGL 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

```
<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3</td>
<td>P4</td>
</tr>
<tr>
<td>P5</td>
<td></td>
</tr>
</tbody>
</table>

v1 v7 v6

v8 v5 v6

x1 y1 z1
x2 y2 z2
x3 y3 z3
x4 y4 z4
x5 y5 z5
x6 y6 z6
x7 y7 z7
x8 y8 z8

```

topology

geometry
Simple Mesh Format (SMF)

- Michael Garland
  http://graphics.cs.uiuc.edu/~garland/

- Triangle data

- Vertex indices begin at 1

```plaintext
#$SMF 1.0
#$vertices 5
#$faces 6
v 2.0 0.0 2.0
v 2.0 0.0 -2.0
v -2.0 0.0 -2.0
v -2.0 0.0 2.0
v 0.0 5.0 0.0
f 1 3 2
f 1 4 3
f 3 5 2
f 2 5 1
f 1 5 4
f 4 5 3
```

Vertex array

```
[2,0,2]
[-2,0,-2]
[2,0,-2]
[2,0,2]
[-2,0,2]
[-2,0,-2]
[-2,0,-2]
[0,5,0]
[2,0,-2]
[2,0,-2]
[0,5,0]
[2,0,2]
[2,0,2]
[0,5,0]
[-2,0,2]
[-2,0,2]
[0,5,0]
[-2,0,-2]
```
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
Face/Edge/Vertex List

topology

graph

geometry

x1 y1 z1
x2 y2 z2
x3 y3 z3
x4 y4 z4
x5 y5 z5
x6 y6 z6
x7 y7 z7
x8 y8 z8
Reading an SMF File

```javascript
var smf_file = loadFileAJAX(fname); // in initShaders2.js
var lines = smf_file.split('\n');

for(var line = 0; line < lines.length; line++){
    var strings = lines[line].trimRight().split(' ');  
    switch(strings[0]){
        case('v'):
            # Process vertices
            break;
        case('f'):
            # Process faces
            break;
    }
}
```
The Rotating Cube
Objectives

• Put everything together to display rotating cube
• Two methods of display
  - by arrays
  - by elements
Clip Space

• Left-handed!
Modeling a Cube

Define global array for vertices

```javascript
var vertices = [
    vec3( -0.5, -0.5, -0.5 ),
    vec3( -0.5,  0.5, -0.5 ),
    vec3(  0.5,  0.5, -0.5 ),
    vec3(  0.5, -0.5, -0.5 ),
    vec3( -0.5, -0.5,  0.5 ),
    vec3( -0.5,  0.5,  0.5 ),
    vec3(  0.5,  0.5,  0.5 ),
    vec3(  0.5, -0.5,  0.5 )
];
```
Define global array for colors

```javascript
var vertexColors = [
    [1.0, 0.0, 0.0, 1.0],  // red
    [1.0, 0.0, 1.0, 1.0],  // magenta
    [1.0, 1.0, 1.0, 1.0],  // white
    [1.0, 1.0, 0.0, 1.0],  // yellow
    [0.0, 0.0, 0.0, 1.0],  // black
    [0.0, 0.0, 1.0, 1.0],  // blue
    [0.0, 1.0, 1.0, 1.0],  // cyan
    [0.0, 1.0, 0.0, 1.0]   // green
];
```
Draw cube from faces

```plaintext
function colorCube( )
{
    quad(0,3,2,1);
    quad(2,3,7,6);
    quad(0,4,7,3);
    quad(1,2,6,5);
    quad(4,5,6,7);
    quad(0,1,5,4);
}
```

Note that vertices are ordered so that we obtain correct outward facing normals
Each quad generates two triangles
The quad Function

Put position and color data for two triangles from a list of indices into the array vertices

```javascript
function quad(a, b, c, d)
{
    var indices = [ a, b, c, a, c, d ];
    for ( var i = 0; i < indices.length; ++i ) {

        points.push( vertices[indices[i]]);
        colors.push( vertexColors[indices[i]] );

        // for solid colored faces use
        //colors.push(vertexColors[a]);

    }
}
```

Angel and Shreiner: Interactive Computer Graphics 7E © Addison-Wesley 2015
Initialization

```javascript
var canvas, gl;
var numVertices = 36;
var points = [];
var colors = [];

window.onload = function init(){
    canvas = document.getElementById( "gl-canvas" );
    gl = canvas.getContext('webgl2');

    colorCube();

    gl.viewport( 0, 0, canvas.width, canvas.height );
    gl clearColor( 1.0, 1.0, 1.0, 1.0 );
    gl.enable(gl.DEPTH_TEST);

    // rest of initialization and html file
    // same as previous examples
```
gl.bufferData(gl.ARRAY_BUFFER, flatten(points), gl.STATIC_DRAW);
gl.bufferData(gl.ARRAY_BUFFER, flatten(colors), gl.STATIC_DRAW);

function render(){
    gl.clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    gl.drawArrays( gl.TRIANGLES, 0, numVertices );
}
 Mapping indices to faces

```javascript
var indices = [
  0, 3, 2,
  0, 2, 1,
  2, 3, 7,
  2, 7, 6,
  0, 4, 7,
  0, 7, 3,
  1, 2, 6,
  1, 6, 5,
  4, 5, 6,
  4, 6, 7,
  0, 1, 5,
  0, 5, 4
];
```

```javascript
var vertices = [
  vec3(-0.5, -0.5, -0.5),
  vec3(-0.5, 0.5, -0.5),
  vec3(0.5, 0.5, -0.5),
  vec3(0.5, -0.5, -0.5),
  vec3(-0.5, -0.5, 0.5),
  vec3(-0.5, 0.5, 0.5),
  vec3(0.5, 0.5, 0.5),
  vec3(0.5, -0.5, 0.5)
];
```
• Just send vertices and vertexColors, then indices
• No redundant data transferred
  - More efficient

```javascript
const flatten = (array) => array.flat();
gl.bufferData(gl.ARRAY_BUFFER, flatten(vertices), gl.STATIC_DRAW);
gl.bufferData(gl.ARRAY_BUFFER, flatten(vertexColors), gl.STATIC_DRAW);
```
Rendering by Elements

• Send indices to GPU, along with vertex data

```javascript
var iBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, iBuffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER,
    new Uint8Array(indices), gl.STATIC_DRAW);
```

• Render by elements

```javascript
    gl.drawElements( gl.TRIANGLES, numVertices,
        gl.UNSIGNED_BYTE, 0 );
```

• Even more efficient if we use triangle strips or triangle fans

Angel and Shreiner: Interactive Computer Graphics 7E © Addison-Wesley 2015
Adding Buttons for Rotation

```
var xAxis = 0;
var yAxis = 1;
var zAxis = 2;
var axis = 0;
var theta = [ 0, 0, 0 ];
var thetaLoc;

document.getElementById( "xButton" ).onclick = function () {axis = xAxis;};
document.getElementById( "yButton" ).onclick = function () {axis = yAxis;};
document.getElementById( "zButton" ).onclick = function () {axis = zAxis;};
```
function render(){
    gl.clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    theta[axis] += 2.0;
    gl.uniform3fv(thetaLoc, theta);
    gl.drawArrays( gl.TRIANGLES, 0, numVertices );
    requestAnimFrame( render );
}
WebGL matrix code

• Remember that matrices are column major order in GLSL

In OpenGL we had to transpose your matrices when sending them to the shaders!

```cpp
glUniformMatrix4fv(matrix_loc, 1, GL_TRUE, model_view);
```

• `flatten()` now does it for you!

```cpp
gl.uniformMatrix4fv(matrix_loc, false, flatten(model_view));
```
Transforming Each Vertex

```glsl
in vec4 aPosition, aColor;
out vec4 vColor;
uniform mat4 rot;

void main()
{
    gl_Position = rot * aPosition;
    vColor = aColor;
}
```
Go to Assignment 4
• The default viewing volume is a box centered at the origin with sides of length 2
  • \((-1,-1,-1) \rightarrow (1,1,1)\)
• All geometry in box is parallel-projected into the \(z=0\) plane!
• Then rendered
· Define cube geometry and color in init()
· Specify transformations with 9 values
  - scalex, scaley, scalez, rotx, roty, rotz, dx, dy, dz
  - Default values define no transformations
· They define 5 matrices
  - scale, rotx, roty, rotz, translate
· Keyboard callback
  - Figures out how to change 9 transformation values
  - Different delta values for scale, rotation and translation
  - Change deltas with multiplication, e.g. 1.02 & 0.98
• Render function
  - Composes final transformation matrix from scale, rotx, roty, rotz and translate matrices (in the correct order!)
  - Sends transformation matrix to vertex shader
  - Draws cube
• Vertex shader applies transformation matrix to vertices
Classical Viewing
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 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

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

multiview
orthographic

axonometric
oblique

isometric
dimetric
trimetric

perspective

1 point
2 point
3 point
Perspective Projection
Parallel Projection

Object

Projector

Projection plane

DOP
Orthographic Projection

Projectors are orthogonal to projection surface
Multiview Orthographic Projection

- Projection plane parallel to principal face
- Usually form front, top, side views

Isometric (not multiview orthographic view)

In CAD and architecture, we often display three multiviews plus isometric.
Advantages and Disadvantages

- 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
Axonometric Projections

Allow projection plane to move relative to object

classify by how many angles of a corner of a projected cube are the same

none: trimetric
two: dimetric
three: isometric
Types of Axonometric Projections

Dimetric

Trimetric

Isometric
Advantages and Disadvantages

- Lines are scaled (foreshortened) but can find scaling factors
- Lines preserved but angles are not
  - 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
History of Linear Perspective

• Renaissance artists
  - Alberti (1435)
  - Della Francesca (1470)
  - Da Vinci (1490)
  - Pélerin (1505)
  - Dürer (1525)

Dürer: Measurement Instruction with Compass and Straight Edge

Vanishing Points

• Parallel lines (not parallel to the projection plan) on the object converge at a single point in the projection (the \textit{vanishing point})

• Drawing simple perspectives by hand uses these vanishing point(s)
One-Point Perspective

• One principal face parallel to projection plane
• One vanishing point for cube
Two-Point Perspective

- One principal direction parallel to projection plane
- Two vanishing points for cube
Three-Point Perspective

- No principal face parallel to projection plane
- Three vanishing points for cube
Perspective Projection (Titanic)
Advantages and Disadvantages

- Objects further from viewer are projected smaller than the same sized objects closer to the viewer \((\text{diminution})\)
  - Looks realistic
- Equal distances along a line are not projected into equal distances \((\text{nonuniform foreshortening})\)
- 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

multiview
orthographic

axonometric

1 point

2 point

3 point

perspective

oblique

isometric
dimetric
trimetric