Basic Mesh Data Structures

CS 418: Interactive Computer Graphics

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Eric Shaffer
Polygonal Meshes

Rasterization engines typically rely on polygonal meshes (specifically triangle meshes) to represent surfaces.

Modern GPUs are designed specifically to rasterize triangles.

Many advantages to using triangles:
- Simplest 2D primitive...in fact it is a 2-simplex.
- Any 2D polygon can be triangulated.
- Can easily represent sharp surface features.

Any disadvantages you can think of?

Why do we say 2D when we are rendering in 3D?
Surface Mesh Properties

- Single component, closed, triangular, orientable manifold
- With boundaries
- Not orientable
- Multiple components
- Not only triangles
- Non manifold
Surface Mesh Properties

**Manifold:**
1. Every edge connects exactly two faces
2. Vertex neighborhood is “disk-like”

**Orientable:** Consistent normals

**Watertight:** Orientable + Manifold

**Boundary:** Some edges bound only one face

**Ordered:** Vertices in CCW order when viewed from normal
2-Manifold Mesh Examples

Disk-shaped neighborhoods

non-manifolds
Genus

Genus 0  Genus 1  Genus 2  Genus ?
Euler Characteristic

For a closed (no boundary), manifold, connected surface mesh:

\[ V-E+F= 2(1-G) \]

\( V \) = number of vertices
\( E \) = number of edges
\( F \) = number of faces
\( G \) = genus (number of holes in the surface)

A 2-manifold is a surface (locally like a plane)
Euler Characteristic for Closed 2-Manifold Polygonal Meshes

\[ V + F - E = \chi \]

Euler characteristic

Cube:
- \( V = 8 \)
- \( E = 12 \)
- \( F = 6 \)
- \( \chi = 8 + 6 - 12 = 2 \)

Complex shape:
- \( V = 3890 \)
- \( E = 11664 \)
- \( F = 7776 \)
- \( \chi = 2 \)
...and if they are triangle meshes

- *Triangle* mesh statistics
  \[ E \approx 3V \]
  \[ F \approx 2V \]

- Avg. valence \( \approx 6 \)
  *Show using Euler Formula*

Aside from being totally interesting on their own, these formulas are useful for computing memory usage.
Mesh Data Structures

Need to store
- Geometry
- Connectivity

Can be used as file formats or internal formats

Considerations
- Space
- Efficient operations

Mesh processing has different requirements than rendering
- Example: Deforming a mesh when simulating physics...
Mesh Data Structure: Face Set (STL)

- face:
  - 3 positions

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_{11}$</td>
<td>$y_{11}$</td>
<td>$z_{11}$</td>
</tr>
<tr>
<td>$x_{21}$</td>
<td>$y_{21}$</td>
<td>$z_{21}$</td>
</tr>
<tr>
<td>$x_{31}$</td>
<td>$y_{31}$</td>
<td>$z_{31}$</td>
</tr>
<tr>
<td>$x_{12}$</td>
<td>$y_{12}$</td>
<td>$z_{12}$</td>
</tr>
<tr>
<td>$x_{22}$</td>
<td>$y_{22}$</td>
<td>$z_{22}$</td>
</tr>
<tr>
<td>$x_{32}$</td>
<td>$y_{32}$</td>
<td>$z_{32}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x_{13}$</td>
<td>$y_{13}$</td>
<td>$z_{13}$</td>
</tr>
<tr>
<td>$x_{23}$</td>
<td>$y_{23}$</td>
<td>$z_{23}$</td>
</tr>
<tr>
<td>$x_{33}$</td>
<td>$y_{33}$</td>
<td>$z_{33}$</td>
</tr>
</tbody>
</table>

36 B/f = 72 B/v
no connectivity!

Corresponds to structure used for WebGL call
gl.drawArrays(gl.TRIANGLES, 0, vertexPositionBuffer.numberOfItems);
gl.TRIANGLES

vertexPositionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexPositionBuffer);
var triangleVertices = [
    0.0, 0.5, 0.0,
    -0.5, -0.5, 0.0,
    0.5, -0.5, 0.0,
    0.0, 0.5, 0.0,
    1.0, 0.5, 0.0,
    0.5, -0.5, 0.0
];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices), gl.STATIC_DRAW);
vertexPositionBuffer.itemSize = 3;
vertexPositionBuffer.numberOfItems = 6;

...gl.drawArrays(gl.TRIANGLES, 0, vertexPositionBuffer.numberOfItems);

• Assuming you are using gl.drawArrays():
  • Each triangle requires you specify three new vertices
  • Number of triangles = number vertices/3
WebGL supports 3 basic geometric primitives:
1. Triangles
2. Lines
3. Point Sprites

We've already seen one way to send triangles into the pipeline.

There are three different triangle drawing modes depending on how you specify the connectivity:

- `gl.TRIANGLES`
- `gl.TRIANGLE_STRIP`
- `gl.TRIANGLE_FAN`
gl.TRIANGLE_STRIP

- Allows you to reuse vertices when drawing triangles that share vertices.
- Number of triangles = what?
- Notice that per-triangle color is not easy to achieve
- Order of the vertices is important

```javascript
vertexPositionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexPositionBuffer);
var triangleVertices = [
  -0.5, -0.5, 0.0,
  0.5, -0.5, 0.0,
  0.0, 0.5, 0.0,
  1.0, 0.5, 0.0,
];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices), gl.STATIC_DRAW);
vertexPositionBuffer.itemSize = 3;
vertexPositionBuffer.numberOfItems = 4;
....
gl.drawArrays(gl.TRIANGLE_STRIP, 0, vertexPositionBuffer.numberOfItems);
```
Winding Order

• Winding order is determined by the order of the vertices making up a triangle when seen from the viewing direction.

• Equivalently, winding order tells you the direction of the triangle surface normal.

• CCW is traditional and is WebGL default: gl.frontFace(gl.CCW)

• For triangle strips, winding order determines the order in which vertices in the buffer are used to form triangles
gl.TRIANGLE_FAN

- First vertex is the fan center
- Next two vertices specify the first triangle
- Each succeeding vertex forms a triangle with the center and previous vertex
- How many triangles for a given number of vertices?
- Are fans and strips equivalent?

```javascript
vertexPositionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexPositionBuffer);
var triangleVertices = [
    0.5, -0.5, 0.0,
    1.0, 0.5, 0.0,
    0.0, 0.5, 0.0,
    -0.5, -0.5, 0.0,
];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices), gl.STATIC_DRAW);
vertexPositionBuffer.itemSize = 3;
vertexPositionBuffer.numberOfItems = 4;
...
gl.drawArrays(gl.TRIANGLE_FAN, 0, vertexPositionBuffer.numberOfItems);
```
Mesh Data Structure: Indexed Face Set (OBJ)

- **vertex:**
  - position

- **face:**
  - vertex indices

### Vertices

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$y_1$</td>
<td>$z_1$</td>
</tr>
<tr>
<td>$x_v$</td>
<td>$y_v$</td>
<td>$z_v$</td>
</tr>
</tbody>
</table>

### Triangles

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_{11}$</td>
<td>$v_{12}$</td>
<td>$v_{13}$</td>
</tr>
<tr>
<td>$v_{13}$</td>
<td>$v_{17}$</td>
<td>$v_{18}$</td>
</tr>
</tbody>
</table>

\[12\text{ B/v} + 12\text{ B/f} = 36\text{ B/v}\]

no neighborhood info

The OBJ file format is a popular storage format for meshes

Developed by Wavefront Technologies...now part of AutoDesk

Text files with .obj extension

```
# List of geometric vertices
v 0.123 0.234 0.345 1.0
v ...
# List of triangles
f 1 3 4
f 2 4 5
...```
Indexed Face Set

Can be used for offline storage...a file format

Or can be used as an internal data structure

One block of data are the vertices
  • Each vertex is a set of 3 coordinates
  • Often referred to as the geometry of the mesh

Another block of data is the set of triangles
  • Each triangle is set of 3 integers vertex IDs
  • The vertex IDs are indices into the vertex block

What are some advantages of this representation?
Indexed Meshes in WebGL

WebGL supports drawing indexed face meshes
Simply need another buffer for the indexed faces

function draw() {
  ...
  gl.bindBuffer(gl.ARRAY_BUFFER, meshVertexPositionBuffer);
  gl.vertexAttribPointer(
    shaderProgram.vertexPositionAttribute,
    meshVertexPositionBuffer.itemSize, gl.FLOAT, false, 0, 0);

  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, meshIndexBuffer);
  gl.drawElements(gl.TRIANGLES,
                  meshIndexBuffer.numberOfItems, gl.UNSIGNED_SHORT, 0);
}
WebGL: `gl.drawElements()` method

We’ll see how to do this in the next MP.
function setupBuffers() {
  meshVertexPositionBuffer = gl.createBuffer();

  gl.bindBuffer(gl.ARRAY_BUFFER, meshVertexPositionBuffer);
  var meshVertexPositions = [1.0, 5.0, 0.0,...];

  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(meshVertexPositions),
               gl.STATIC_DRAW);
  meshVertexPositionBuffer.itemSize = 3;
  meshVertexPositionBuffer.numberOfItems = 36;
  gl.enableVertexAttribArray(
    shaderProgram.vertexPositionAttribute);
Setting up Indexed Drawing

```javascript
meshIndexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, meshIndexBuffer);
var meshIndex = [ 0, 1, 2, 2, 1, 3, 2, 3, 4, ... ];
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(meshIndex), gl.STATIC_DRAW);
meshIndexBuffer.itemSize = 1;
meshIndexBuffer.numberOfItems = 150;
}
```
Other WebGL Primitives: Lines

`gl.LINES` draws independent lines \((v_0,v_1), (v_2,v_3), (v_4,v_5)\)

`gl.LINE_STRIP` draws a polyline \((v_0,v_1),(v_1,v_2),(v_2,v_3),(v_3,v_4),(v_4,v_5)\)

`gl.LINE_LOOP` draws a line strip with a line connecting the first and final vertex

Generally have poor visual quality in most browser implementations….people often use triangle strips instead.
Other WebGL Primitives: Point Sprites

Specified with gl.POINTS mode
Renders one point per vertex in the buffer
Uses N pixels in the point is specified using `gl.pointSize(N)`

The star field below uses some extra shader code to achieve its look.
Using Multiple Buffers

You can use more than one vertex buffer. Set them up like you did the first buffer and call gl.drawArrays multiple times in your draw function.

```
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer1);
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
    vertexBuffer1.itemSize, gl.FLOAT, false, 0, 0);

gl.bindBuffer(gl.ARRAY_BUFFER, vertexColorBuffer1);
gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
    vertexColorBuffer1.itemSize, gl.FLOAT, false, 0, 0);

gl.drawArrays(gl.TRIANGLES, 0, vertexBuffer1.numItems);
```

Complex scenes and geometry may require more than one buffer. Is there a way to draw multiple objects from a single buffer?
Minimizing draw calls

• You generally want as few calls to gl.drawArrays as possible
  • Same is true for gl.drawElements...we’ll discuss that later

• For triangle strips, you can insert degenerate triangles into the stream
  • These triangles will have two identical vertices and 0 area

• Can connect strips using a sequence of degenerate triangles

• Better to do this with gl.drawElements
  • Bigger performance hit for gl.drawArrays due to cache effects
Back Face Culling

• Backface culling is an optimization technique
• It drops backfacing polygons from the pipeline.
• Why would backface culling be useful?

• What artifact do you see here?

• Backface culling is not hidden surface removal
Vector Dot Product

The *dot product* or *inner product* of two vectors is

\[ u \cdot v = u_x v_x + u_y v_y + \cdots = ||u|| ||v|| \cos \theta \]

Can think of it as a measure of how aligned the vectors are.
Vector Dot Product

Is a polygon facing away from the viewer? We can decide by using a dot product test.
Back Face Culling

- Decide whether the view vector $V$ runs from the surface to the eyepoint or from the eyepoint to the surface.
  - For this test, we’ll use eyepoint to surface.
- So, if $90 \leq \theta \leq 270$ where $\theta > 0$
  - then dot product is negative and polygon faces viewer
- IF the dot product is positive then polygon does not face viewer
WebGL Back Face Culling

Polygon culling is disabled by default. To enable or disable culling, use the `enable()` and `disable()` methods with the argument `gl.CULL_FACE`.

```javascript
1  gl.enable(gl.CULL_FACE);
2  gl.cullFace(gl.FRONT_AND_BACK);
```

To check the current cull face mode, query the `CULL_FACE_MODE` constant.

```javascript
1  gl.getParameter(gl.CULL_FACE_MODE) === gl.FRONT_AND_BACK;
2  // true
```