Week 03:
Primitives, Polygon Meshes, and Attributes

CS-537: Interactive Computer Graphics
Dr. Chloe LeGendre
Department of Computer Science

For academic use only.

Some materials from the companion slides of Angel and Shreiner, "Interactive Computer Graphics, A Top-Down Approach with WebGL."

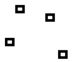
Objectives

- Describe WebGL primitives
- Introduce Polygon meshes (not much of this is in the textbook)
- Describe vertex attributes
- Describe how to use and represent color

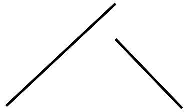
Two classes of Primitives

- In the most general sense, a “primitive” can be thought of as a “building block” of something larger – a basic feature that can be added with other basic features to construct more complicated information
- WebGL has two core types:
  - Geometry Primitives
    - Specified in the problem domain, and include geometry building blocks like points, line segments, polygons.
  - Raster Primitives (or Image Primitives)
    - Arrays of pixels. An image, for example.
- Geometry primitives are all specified through sets of vertices in 2D or 3D space

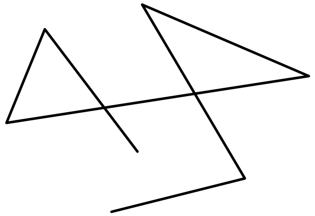
Types of Geometric Primitives in WebGL



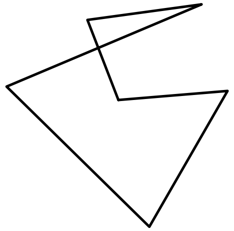
gl.POINTS



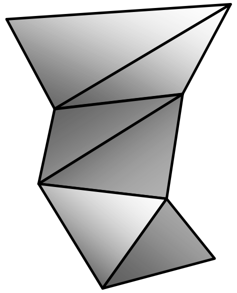
gl.LINES



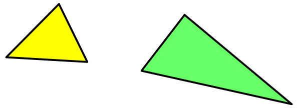
gl.LINE_STRIP



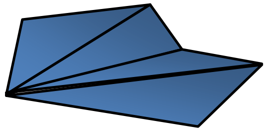
gl.LINE_LOOP



gl.TRIANGLE_STRIP



gl.TRIANGLES



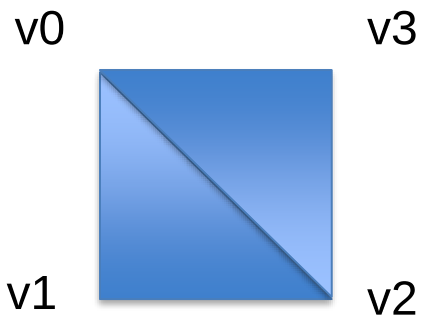
gl.TRIANGLE_FAN

Specified in final draw call during render() function. Example:
gl.drawArrays(gl.TRIANGLES, 0, 3);

Arguments: primitive type, start index in vertex array, number of vertices to render (NOT NUMBER OF PRIMITIVES)

Specifying Vertices for Primitives

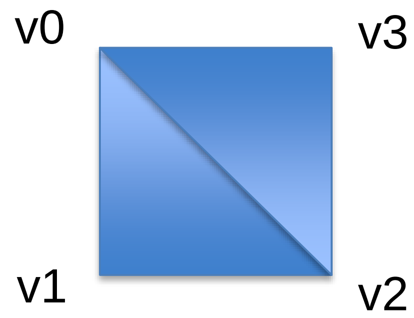
- Different primitives need different number of vertices specified to draw the same shape
- Consider drawing two triangles next to each other on one side using:
  - gl.TRIANGLES



Specify each vertex for each triangle
Order to send to GPU:
v0, v1, v2, v0, v2, v3
(other orders possible too, but all have 6 vertices)

gl.drawArrays(gl.TRIANGLES, 0, 6);

  - gl.TRIANGLE_STRIP

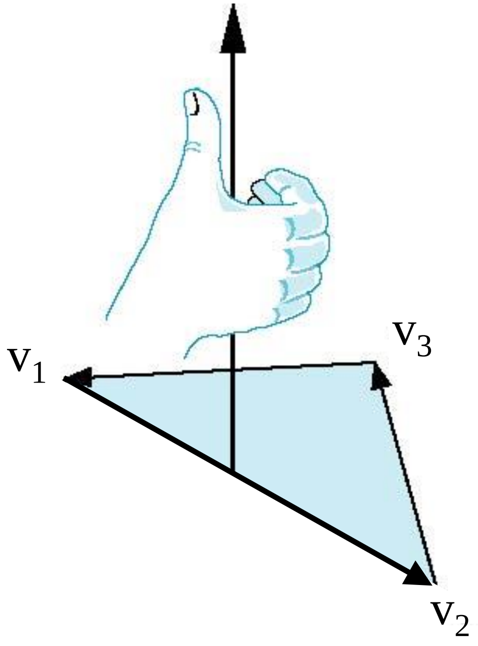


Order to send to GPU:
v1, v0, v2, v3
[Or: v1, v2, v0, v3]

gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);

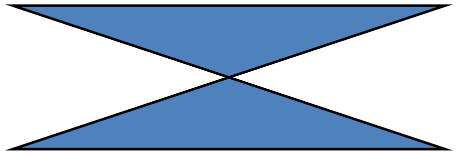
Interlude: Inward and Outward Facing Polygons

- The order $\{v_1, v_2, v_3\}$ and $\{v_2, v_3, v_1\}$ are equivalent in that the same triangle will be rendered by WebGL but the order $\{v_1, v_3, v_2\}$ is different
- The first two describes a polygon facing a certain direction; the third describes a triangle facing the opposite direction
- Use the *right-hand rule* = counter-clockwise encirclement of outward-pointing normal
- WebGL can treat inward and outward facing polygons differently
- You can think of this as being a triangle described by the same three points, but with the surface normal vector in the positive or negative direction



Using Primitives

- WebGL will only display triangles, not other types of polygons. Why?
- Triangles (compared with other polygons) are:
  - Simple. Edges cannot cross each other.
  - Convex. All points on a line segment formed between two points in a triangle are also in the triangle.
  - Flat. All vertices must be in the same plane (3 points fully specify a plane).
- This means that other polygons need to be “tessellated” into triangles in the application program, a process also called “triangulation”
- WebGL does not contain a tessellator (some versions of desktop OpenGL do)



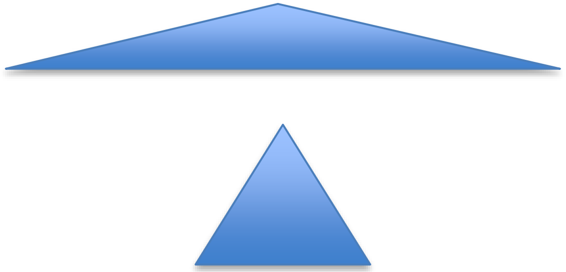
non-simple polygon

nonconvex polygon

Polygon Testing

- Conceptually simple to test for simplicity and convexity, but time-consuming
- Early versions of OpenGL assumed that polygons were always convex and simple, and left polygon testing to the application
- WebGL decides to avoid the issue altogether by only rendering triangles, which are guaranteed to be simple, convex, and planar
  - We need algorithms to triangulate an arbitrary polygon

Good and Bad Triangles for Rendering



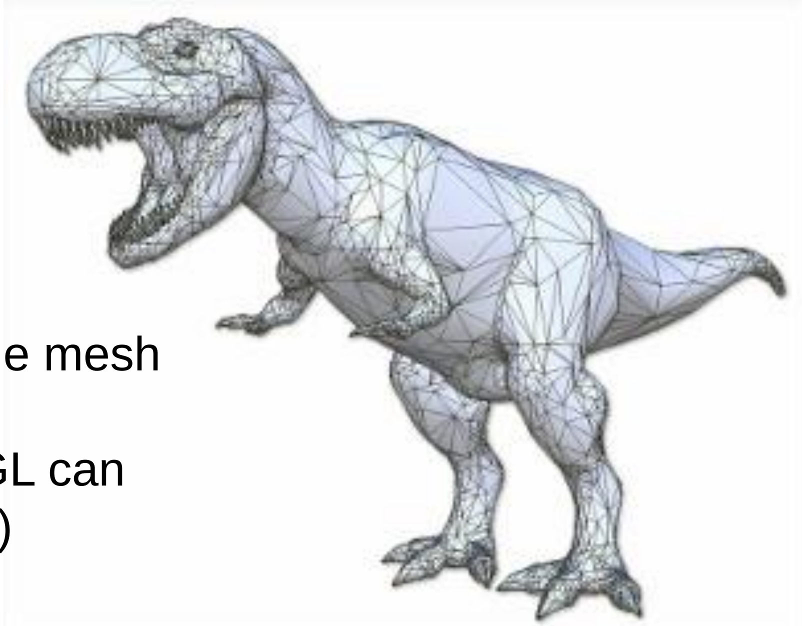
- Long and thin triangles render badly
- Equilateral triangles render well
- Algorithms for triangulation should minimize the minimum angle
- One example: a Delaunay triangulation for unstructured points
  - (Optional reading: Ch 11.13)

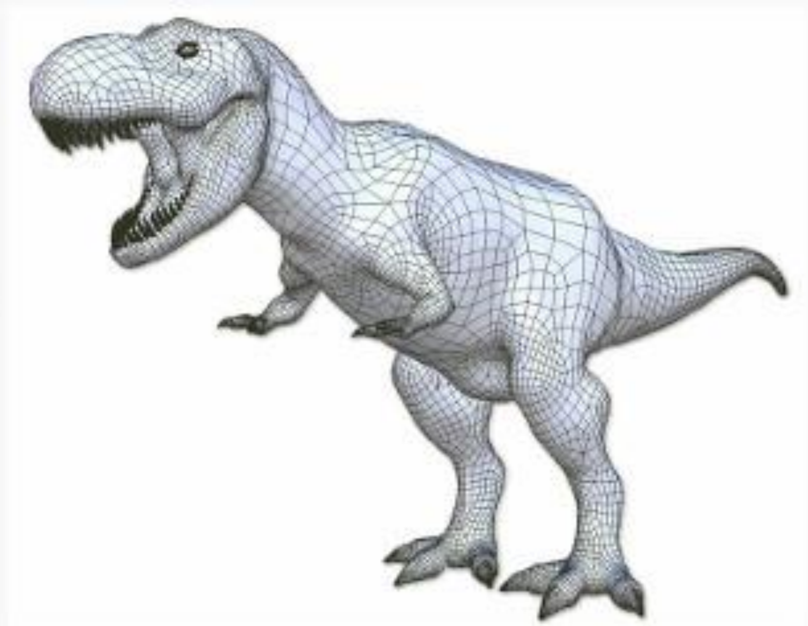
Polygon Meshes (I)

- All 2D or 3D geometry to be rendered should be comprised of the primitives available in WebGL, most commonly triangles.
- One way to represent complex 3D shapes is through a “polygon mesh” or, for WebGL, a “triangle mesh”
- Any shape can be modeled out of polygons if you use enough of them!

Triangle mesh

(WebGL can render)





Quad mesh

(WebGL cannot render; we need to triangulate)

Mesh Level of Detail (LOD)

- Fewer triangles can be used to generate a simpler mesh, that is less likely to well-represent non-planar surfaces. Fewer triangles = less detail = less time to render
- Real-time applications may be limited by the number of polygons that can be rendered per unit of time. If you've heard the expression "low poly count": this is the origin. (Fewer polygons are used to render a scene more quickly but with less fidelity)
- One solution: use models with different LOD depending on their distance to the camera. If the virtual camera is close to the object, use a high LOD model, and if its far away, use a low LOD model
- Algorithms exist for mesh simplification for this use case

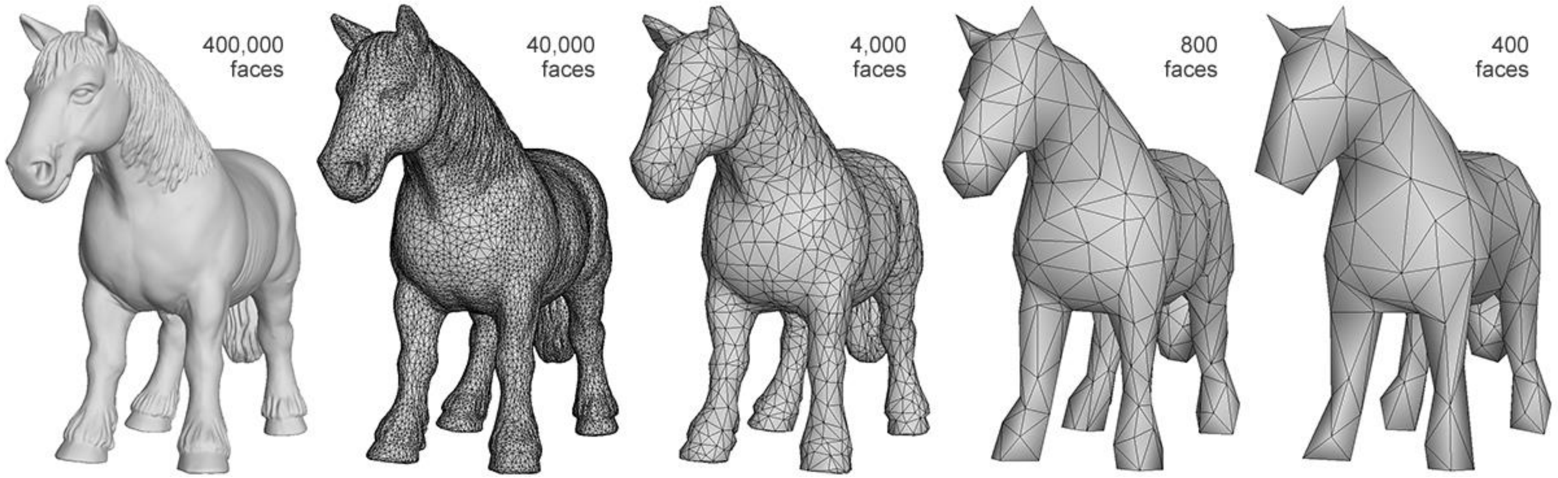
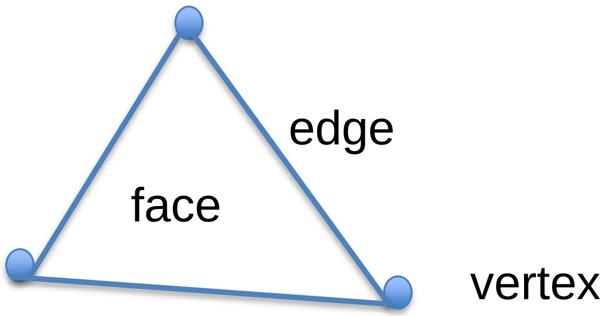


Image credit: Scanifly.readme.io

Polygon Meshes (II)



- A polygon mesh is built from:
  - Vertices (points)
  - Edges (line segments between the vertices)
  - Faces (polygons bounded by the edges)
- A mesh is not just a bucket of vertices (points) in an arbitrary order
  - If we want to render a mesh in WebGL, we need to know which vertices correspond to which triangles since we need to specify vertices in order!
- The question then arises: how to store connectivity information for a mesh?

Mesh Data Structures (I)

- Data structures that store mesh geometry and connectivity (“topology”)
- Goals:
  - Enable compact storage and file formats
  - Enable efficient algorithms on meshes
    - Many steps are time-critical
    - May want to operate on all vertices or all edges of a particular face
      - Example: “Turn this face blue”
    - May want to operate on all neighboring vertices/edges/or faces of a vertex
      - Example: “Turn every face that incorporates this vertex blue”

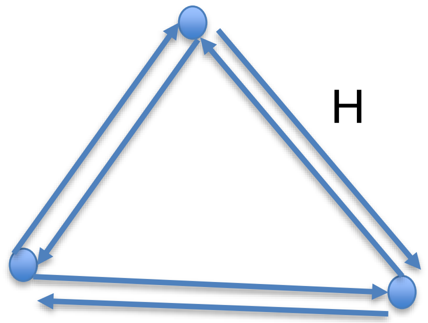
Mesh Data Structures (II)

- A few types:
  - Face Set
  - Shared Vertex
  - Face-Based Connectivity
  - Edge-Based Connectivity
  - Half-Edge Based Connectivity
- Design decision: memory vs. speed trade-off, depends on operations you want to perform on the mesh and data storage requirements.
- To figure out space requirements, we can borrow from geometry/topology/mathematics the Euler Characteristic / Euler's polyhedron formula: $V - E + F = 2$ for convex polyhedra.

Mesh Data Structure: Face Set

Triangles		
x_{11} y_{11} z_{11}	x_{12} y_{12} z_{12}	x_{13} y_{13} z_{13}
x_{21} y_{21} z_{21}	x_{22} y_{22} z_{22}	x_{23} y_{23} z_{23}
...	...	...
x_{F1} y_{F1} z_{F1}	x_{F2} y_{F2} z_{F2}	x_{F3} y_{F3} z_{F3}

- Example file format: STL file
- A triangular face is encoded as 3x 3D vertex positions
- If we assume single precision float (4 bytes):
  - 9 floats x 4 bytes = 36 bytes / face
- How many bytes / vertex?
  - First define a “half edge” H: a pair of a directional edge + the face it borders (see diagram)
  - Total number of half edges H in mesh is 2E (each edge has two halve-edges)
  - Total number of half edges H is also 3F (each face has 3 half edges)
  - Therefore 2E = 3F ; E = 3F/2
  - Assume many vertices, so V – E + F ≈ 0
  - By substitution: V - (1/2)F ≈ 0 ; or: F ≈ 2V.
  - ~72 bytes / vertex for Face Set data structure



- But note that this data structure encodes no explicit connectivity!
- Redundant data storage: vertices included more than once!

Mesh Data Structure: Shared Vertex

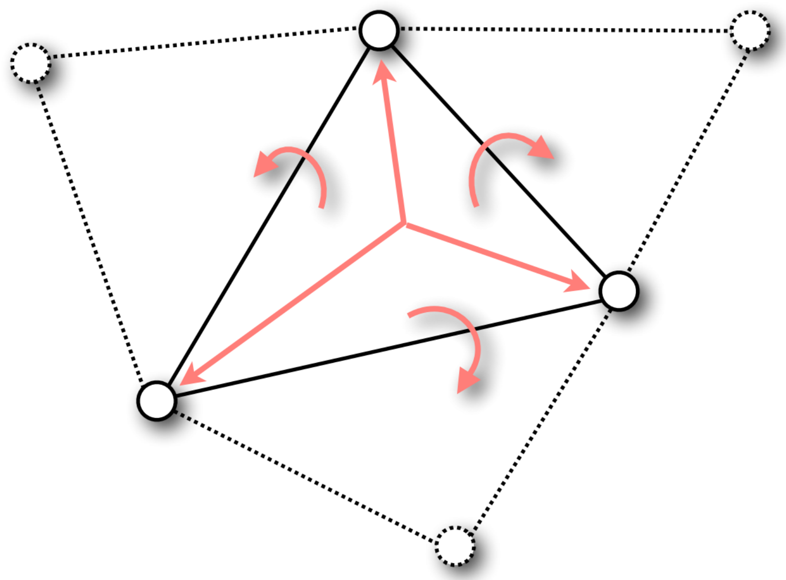
- Used in format OBJ (Assignments 2 & 3)
- Indexed Face List:
  - Vertex: position
  - Face: vertex indices for a face (location IDs in vertex list)
- Bytes per vertex:
  - 3 floats x 4 bytes = 12 bytes/ vertex PLUS
  - 3 floats x 4 bytes = 12 bytes/ face = 24 bytes / vertex
  - Total = 36 bytes/vertex
- No explicit neighborhood information, but saves some storage compared with Face Set.

Vertices
x_1 y_1 z_1
...
x_V y_V z_V

Triangles
i_{11} i_{12} i_{13}
...
...
...
...
i_{F1} i_{F2} i_{F3}

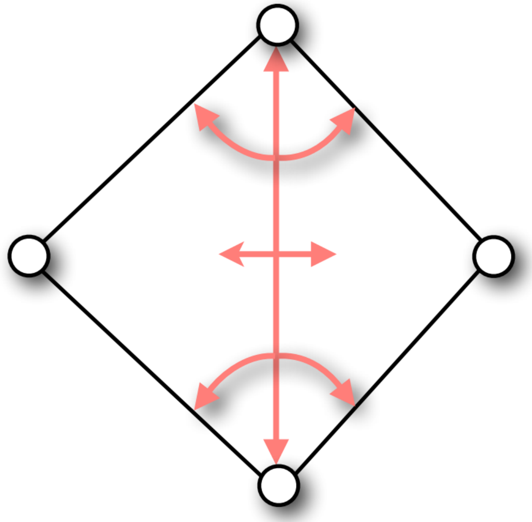
Mesh Data Structure: Face-Based Connectivity

- Vertex:
  - Position (12 bytes/vertex) AND
  - 1 face index (4 bytes)
- Face:
  - 3 vertex indices (12 bytes/face) AND
  - 3 face neighbor indices (12 bytes/face)
  - Face Total: 24 bytes/face = 48 bytes/vertex
- Total: 64 bytes/vertex
- Encodes limited neighborhood information of face adjacency



Mesh Data Structure: Edge-Based Connectivity

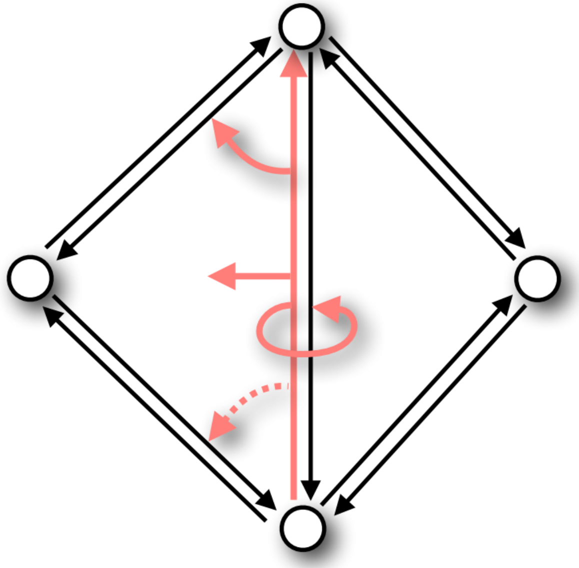
- Vertex:
  - Position (12 bytes/vertex) AND
  - 1 edge index (4 bytes)
- Edge:
  - 2 vertex indices (8 bytes/edge) AND
  - 2 face indices (8 bytes/edge)
  - 4 edge indices (16 bytes/edge)
  - Edge Total: 32 bytes/edge = 96 bytes/vertex
- Face:
  - 1 edge index (4 bytes/face) = 8 byte/vertex
- Total: 120 bytes/vertex
- But edges have no orientation (special case for handling neighbors)



CS-537: Interactive Computer Graphics

Mesh Data Structure: Half-Edge-Based Connectivity

- Uses oriented edges, with 2 half-edges H for each edge E
- Vertex:
  - Position (12 bytes/vertex) AND
  - 1 outgoing half-edge index (4 bytes)
- Half-Edge:
  - 1 vertex index (4 bytes/edge) AND
  - 1 face index (4 bytes/edge)
  - 1, 2 or 3 half-edge indices (4, 8, or 12 bytes/edge)
    - Next, Prev, Opposite (twin)
  - Half-Edge Total: 12 - 20 bytes/halfedge = 72 – 120 bytes/vertex
- Face:
  - 1 half-edge index (4 bytes/face) = 8 byte/vertex
- Total: 96 – 144 bytes/vertex
- Encodes complete connectivity and allows for very easy and fast traversal of mesh



Recap: Data Structures for Polygon Meshes

- Triangle list:
  - Simplest, but dumb.
  - Redundant because each vertex is stored multiple times.
- Vertex list with face list (Shared Vertex):
  - List of vertices, each vertex contains position information only
  - List of triangles, where each includes three vertex IDs
  - Fine for many purposes, but finding adjacent faces is O(F) for a model with F faces.
- Fancier schemes:
  - Store topological information about connectivity so adjacencies can be computed in O(1) time.

OBJ File Format for Polygon Meshes

- We'll use this for Assignment 2 and 3. A file reader will be provided to you.
- File format that implements the Vertex list with face list (Shared Vertex) structure

bunny.obj

```
# OBJ file format with ext .obj
# vertex count = 2503
# face count = 4968
v -3.4101800e-003 1.3031957e-001 2.1754370e-002
v -8.1719160e-002 1.5250145e-001 2.9656090e-002
v -3.0543480e-002 1.2477885e-001 1.0983400e-003
v -2.4901590e-002 1.1211138e-001 3.7560240e-002
v -1.8405680e-002 1.7843055e-001 -2.4219580e-002
v 1.9067940e-002 1.2144925e-001 3.1968440e-002
v 6.0412000e-003 1.2494359e-001 3.2652890e-002
v -1.3469030e-002 1.6299355e-001 -1.2000020e-002
v -3.4393240e-002 1.7236688e-001 -9.8213000e-004
v -8.4314160e-002 1.0957263e-001 3.7097300e-003
v -4.2233540e-002 1.7211574e-001 -4.1799800e-003
```

Vertex coordinates (x, y, z)
Lines start with v

bunny.obj

```
v -6.8866880e-002 1.4723338e-001 -2.8739870e-002
v -6.0965420e-002 1.7002113e-001 -6.0839390e-002
v -1.3895490e-002 1.6787168e-001 -2.1897230e-002
v -6.9413000e-002 1.5121847e-001 -4.4538540e-002
v -5.5039800e-002 5.7309700e-002 1.6990900e-002
f 1069 1647 1578
f 1058 909 939
f 421 1176 238
f 1055 1101 1042
f 238 1059 1126
f 1254 30 1261
f 1065 1071 1
f 1037 1130 1120
f 1570 2381 1585
f 2434 2502 2473
f 1632 1654 1646
f 1144 1166 669
f 1202 1440 305
f 1071 1090 1
f 1555 1570 1584
```

Faces: vertex indices (three per face for triangle)
Lines start with f
are comments

Drawbacks with Meshes

- Need a lot of polygons to represent smooth curved surfaces or shapes
- Need a lot of polygons to represent highly detailed surfaces or shapes
- Difficult to edit, because you need to move individual vertices
- But still very useful and often used to represent 3D content

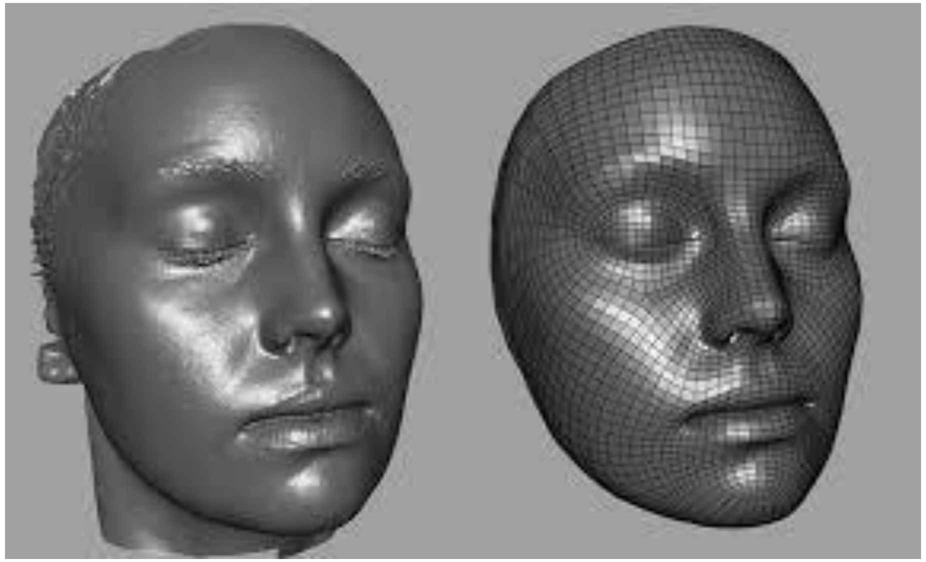


Image: USC ICT Digital Emily Project

Back to WebGL: Attributes

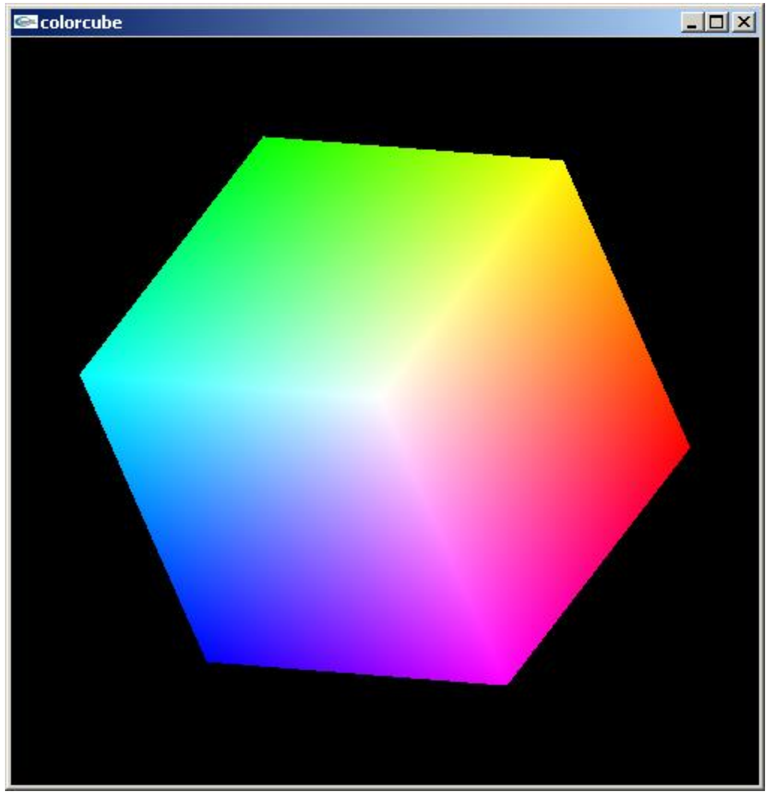
- Attributes determine the appearance of objects
  - Color (points, lines, polygons)
  - Size and width (points and lines)
  - Polygon mode
    - Display edges or vertices
    - Display as filled
- In OpenGL, you could set these with specific API calls
- WebGL only supports a few (gl_PointSize, used within a shader)
- Other attributes we’ll send to GPU as data using “attribute” qualifier

Color in WebGL (I)

- Each color component is stored separately in the frame buffer
- Usually 8 bits per component in the buffer
- Color values range from 0.0 (none) to 1.0 (all) using floats, or over the range from 0 to 255 using unsigned bytes.
- You can set the color of a vertex either by:
  - Sending per-vertex colors to the GPU from application
  - Setting colors in a shader using a uniform
  - Setting colors in a shader using constants in the shader

Color in WebGL (II)

- After you set the vertex colors, you can pass these to the fragment shader using the “varying” qualifier
- The default behavior in WebGL is that the rasterizer will interpolate vertex colors across visible polygons (see the right image)
  - Important to remember this for Assignment 1!
- The alternative is flat shading: the color of the first vertex will determine the fill color. You would need to handle this in your shader in WebGL.



Setting Colors

- Colors are ultimately set in the fragment shader but can be determined in either fragment shader, vertex shader, or in the application
- Application color: pass to vertex shader as a uniform variable or as a vertex attribute
- Vertex shader color: pass to fragment shader as varying variable
- Fragment color: can alter via shader code
- Fragment shader must set: gl_FragColor
- Final color value should be 4D (RGB) plus an alpha value, for transparency.
- For now (Assignment 1) you can set the alpha value to 1.0 to indicate full opacity.