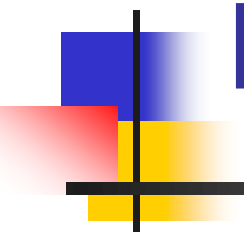
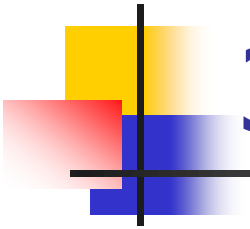
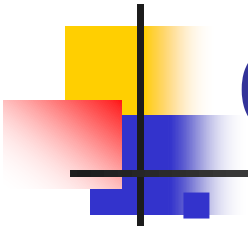


Modeling



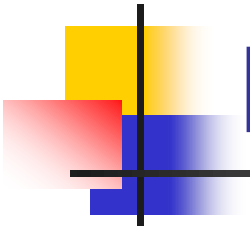
3D Object Representation

- How to model different kinds of objects
  - Trees, clouds, rocks, water, wood, paper, marble, steel, glass, plastic, cloth
- Polygonal approximation
  - Polyhedral objects
  - Curved surfaces
- Physically based modeling



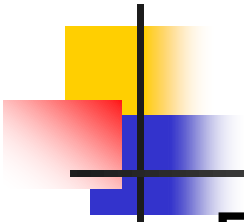
Choosing a Representation

- How well does it represents the objects of interest?
- How easy is it to render (or convert to polygons)?
- How compact is it (how cheap to store and transmit)?
- How easy is it to create?
  - By hand, procedurally, by fitting to measurements, …
- How easy is it to interact with?
  - Modifying it, animating it
- How easy is it to perform geometric computations?
  - Distance, intersection, normal vectors, curvature, …



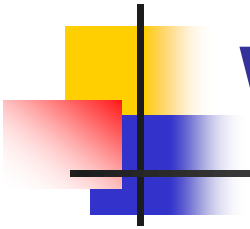
Polygon Modeling

- Polygons are the dominant force in modeling for real-time graphics
- Why?



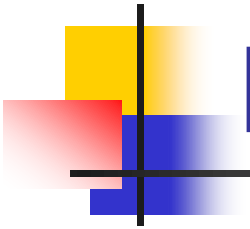
Polygons Dominate

- Everything can be turned into polygons (almost everything)
  - Normally an error associated with the conversion, but with time and space it may be possible to reduce this error
- We know how to render polygons quickly
- Many operations are easy to do with polygons
- Memory and disk space is cheap



What's Bad About Polygons?

- What are some disadvantages of polygonal representations?

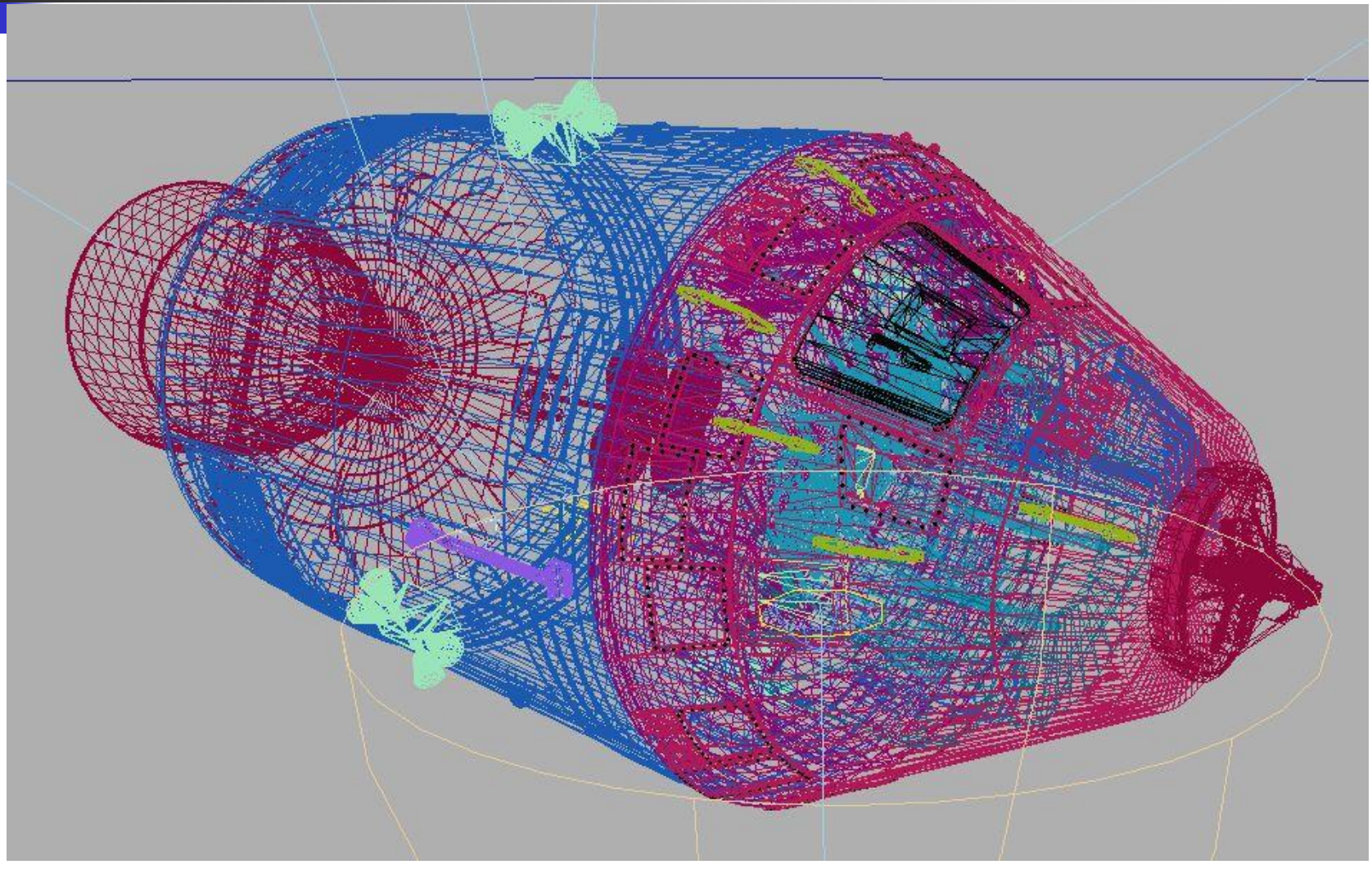


Polygons Aren't Great

- They are always an approximation to curved surfaces
  - But can be as good as you want, if you are willing to pay in size
  - Normal vectors are approximate
  - They throw away information
  - Most real-world surfaces are curved, particularly natural surfaces
- They can be very unstructured

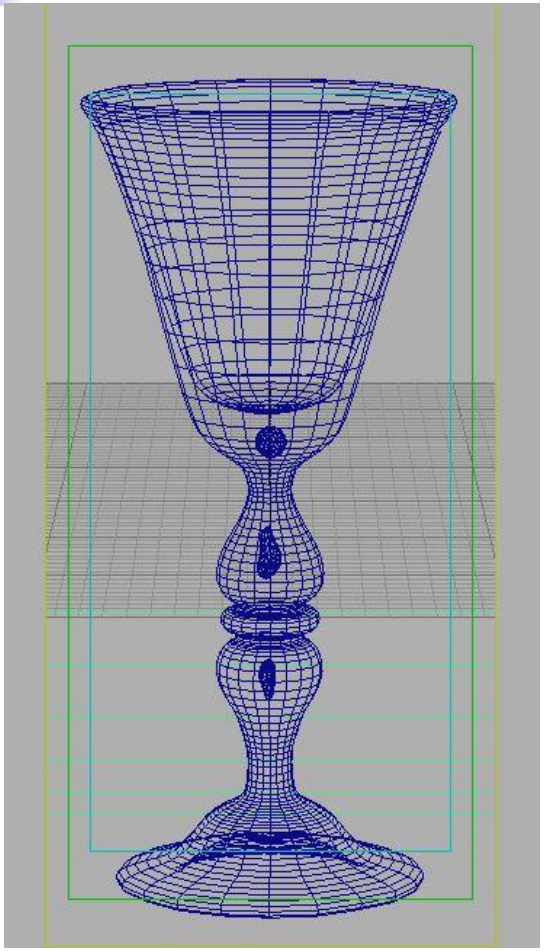
Modeling objects

Wireframe Model

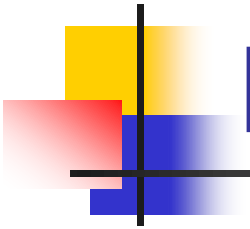


Rendered Image

Symmetric Model



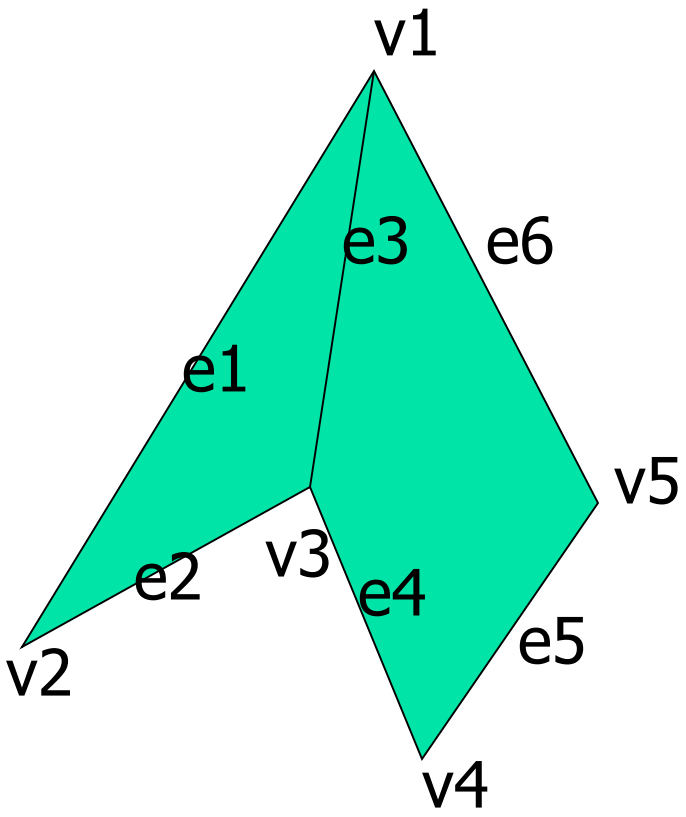
- Define vertices in x-y plane
- Rotate around y axis



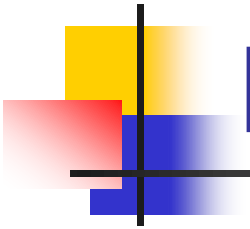
Polygonal Surfaces

- Polygonal surface
  - Boundary representation
  - Simplicity
  - Linear equations
- Surface features
  - Vertex
  - Edge
  - Face

Polygon Tables



- Vertex Table
  - V1: x1, y1, z1
- Edge table
  - E1: v1, v2
- Face table
  - F1: e1, e2, e3



Plane equations

- For transformation, viewing, visible surface identification, and rendering we need spatial information
  - Position
  - Orientation
  - Surface normal vector

Plane equation

$$Ax + By + Cz + D = 0$$

$$for\ 3\ vetices\ \ k = 1..3$$

$$\frac{A}{D} x_k + \frac{B}{D} y_k + \frac{C}{D} z_k = -1$$

$$Solve\ for\ A, B, C, D$$

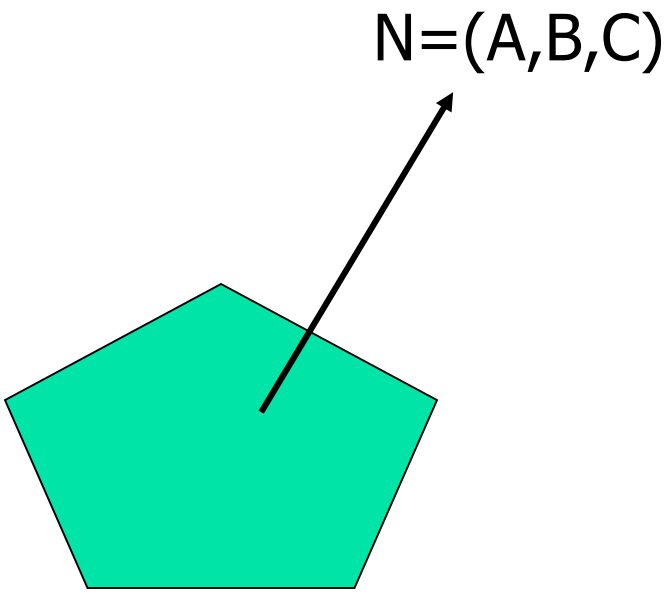
$$A = y_1(z_2 - z_3) + y_2(z_3 - z_1) + y_3(z_1 - z_2)$$

$$B = z_1(x_2 - x_3) + z_2(x_3 - x_1) + z_3(x_1 - x_2)$$

$$C = x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)$$

$$D = -x_1(y_2 z_3 - y_3 z_2) + x_2(y_3 z_1 - y_1 z_3) + x_3(y_1 z_2 - y_2 z_1)$$

Plane equations and normal vectors



Ax+By+Cz+D=0

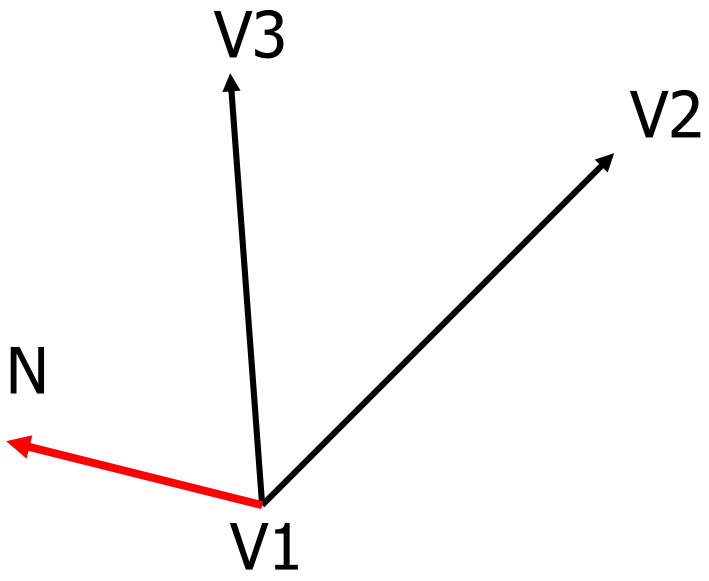
Normal vector can be obtained directly from the coefficient of the plane equations

Determining Normal Vectors

Determining normal from 3 vertices V1, V2, V3 reside on a plane

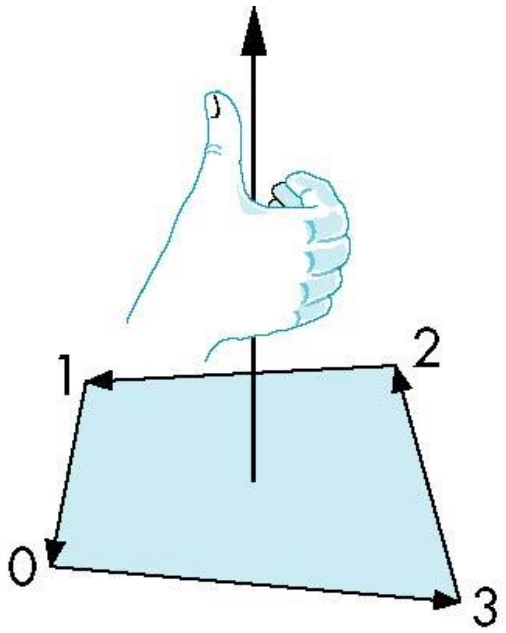
$$N = (V_2 - V_1) \times (V_3 - V_1)$$

$$N \cdot P = -D$$



Inward and Outward Facing Polygons

- The order $\{v_3, v_2, v_1\}$ and $\{v_2, v_1, v_3\}$ 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



Identifying the side

$$Ax + By + Cz + D < 0,\ inside$$

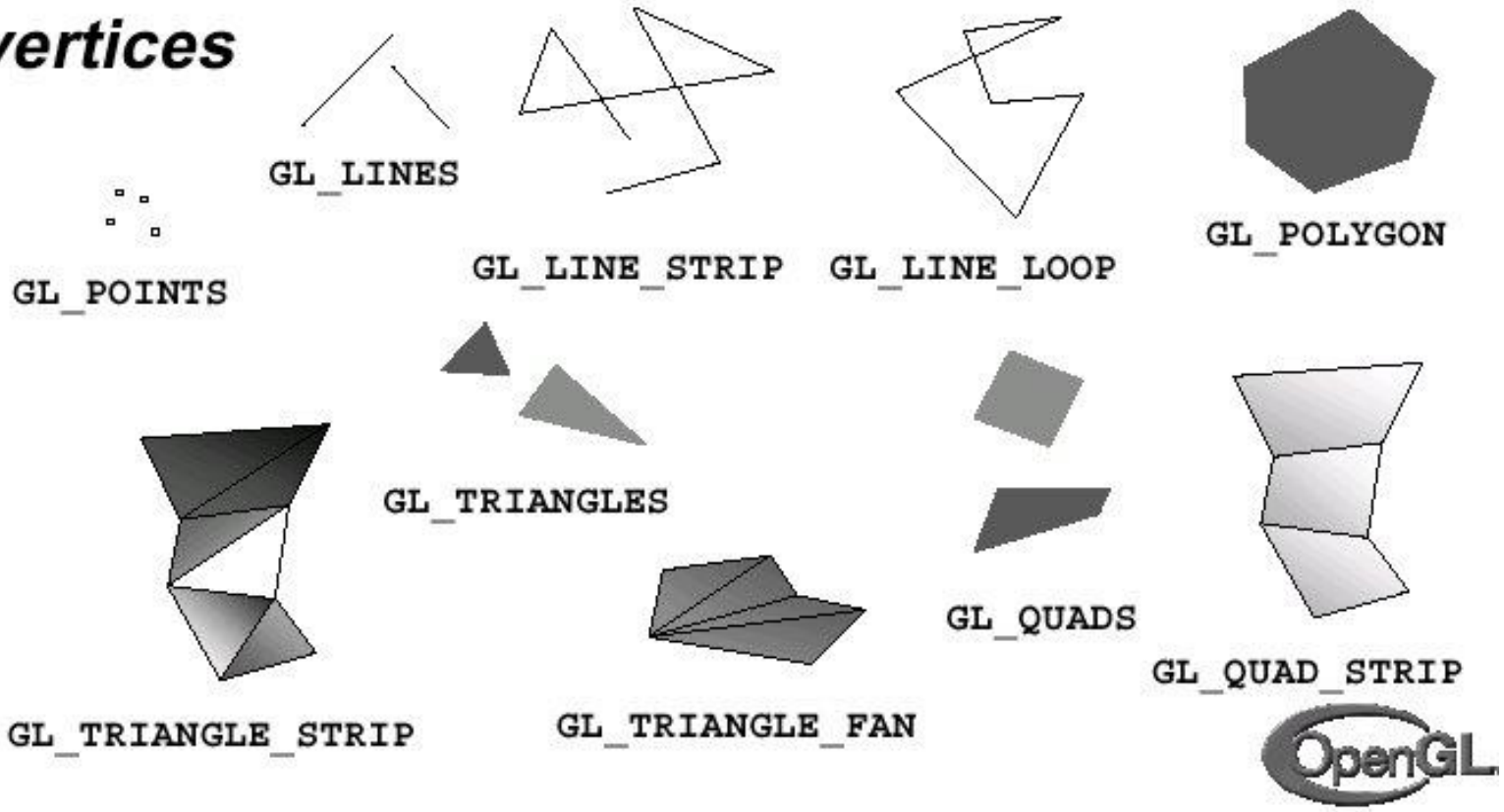
$$Ax + By + Cz + D > 0, outside$$

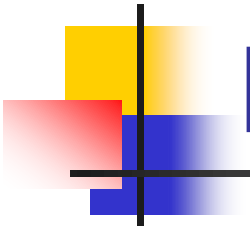
$$Ax + By + Cz + D = 0, on\ the\ plane$$

$$Ax + By + Cz + D \neq 0, Not\ on\ the\ plane$$

Surface modeling in OpenGL

All geometric primitives are specified by vertices

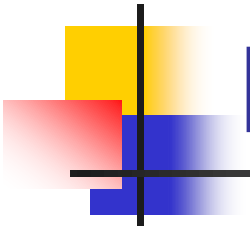




Normal Vectors

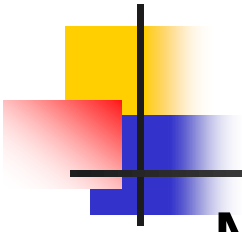
- glNormal3{bsidf}(Type nx, ny, nz)
- glNormal3{fsidf}v(const Type *v)

- Normal vectors define the orientation of its surface
- Required for proper illumination



Example

```
glBegin(GL_POLYGON);
  glNormal3fv(n0);
  glVertex3fv(v0);
  glVertex3fv(v1);
  glVertex3fv(v2);
  glVertex3fv(v3);
glEnd();
```

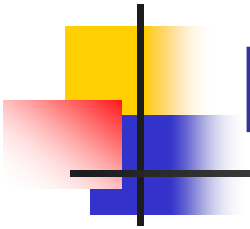


Polygon Soup

- Many polygon models are just lists of polygons

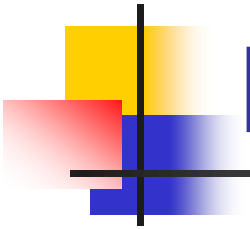
```
struct Vertex {
    float coords[3];
}
struct Triangle {
    struct Vertex verts[3];
}
struct Triangle mesh[n];

glBegin(GL_TRIANGLES)
    for ( i = 0 ; i < n ; i++ )
    {
          glVertex3fv(mesh[i].verts[0]);
          glVertex3fv(mesh[i].verts[1]);
          glVertex3fv(mesh[i].verts[2]);
    }
glEnd();
```

Important Point: OpenGL, and almost everything else, assumes a constant vertex ordering: clockwise or counter-clockwise. Default, and slightly more standard, is counter-clockwise



Polygon Soup Evaluation

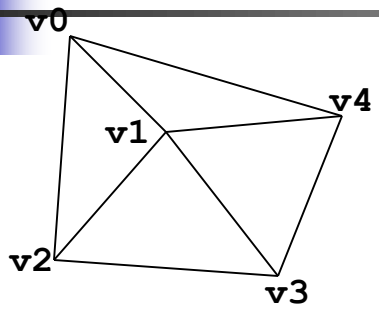
- What are the advantages?
- What are the disadvantages?

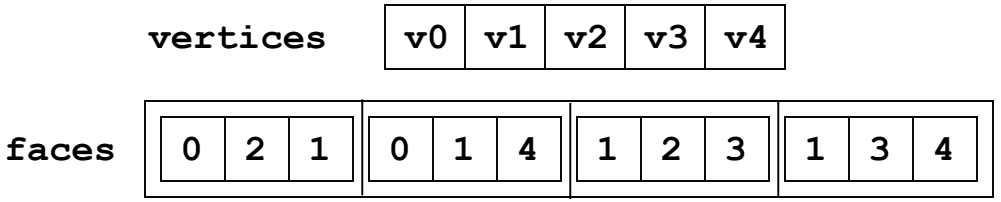


Polygon Soup Evaluation

- What are the advantages?
  - It's very simple to read, write, transmit, etc.
  - A common output format from CAD modelers
  - The format required for OpenGL
- BIG disadvantage: No higher order information
  - No information about neighbors
  - No open/closed information
  - No guarantees on degeneracies

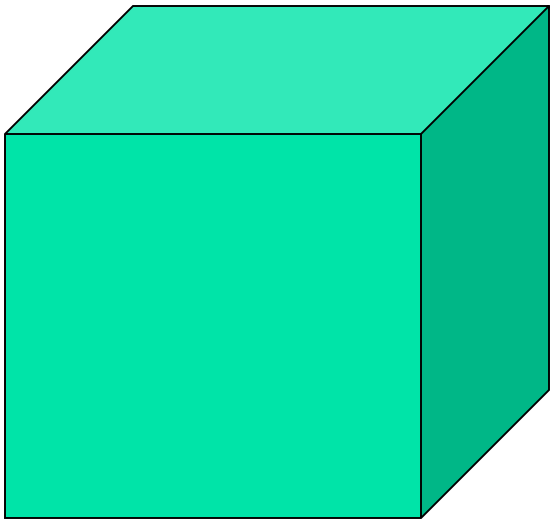
Vertex Indirection



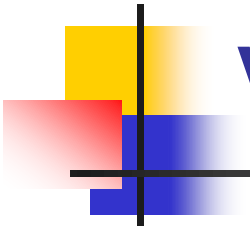


- There are reasons not to store the vertices explicitly at each polygon
  - Wastes memory - each vertex repeated many times
  - Very messy to find neighboring polygons
  - Difficult to ensure that polygons meet correctly
- Solution: Indirection
  - Put all the vertices in a list
  - Each face stores the list indices of its vertices
- Advantages? Disadvantages?

Vertex Array



- Six sides
- Eight shared vertices
- For each face 4 vertices are processed
  - 24 vertices process when there are only 8 vertices



Vertex Array

- If we include normal vectors, colors, index, texture coordinate, edge flag the number of function call increases significantly
- Vertex Array outlines
  - Enabling array
  - Specifying array
  - Accessing array

Specifying Data for the Arrays

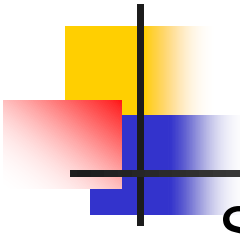
- Using the same color and vertex data, first we enable

```
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
```

- Identify location of arrays

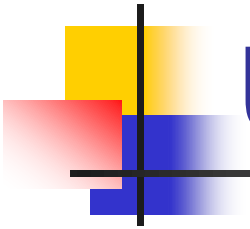
```
glVertexPointer(3, GL_FLOAT, 0, vertices);
```


```
glColorPointer(3, GL_FLOAT, 0, colors);
```



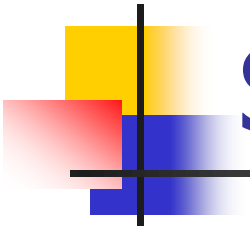
Example

```
Static Glfloat colors[]={1.0, 0.2, 0.2, 0.2, 0.2, 1.0,
                         0.8, 1.0, 0.2,0.75, 0.75, 0.75,
                         0.35, 0.35, 0.35, 0.5, 0.5, 0.5};

Static Glint vertices[]={ 25,  25, 100, 325,
                          175, 25, 175, 325,
                          250, 25, 325, 325};

glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);

glColorPointer(3, GL_FLOAT, 0, colors);
glVertexPointer(2, GL_INT, 0, vertices);
```



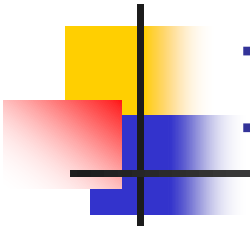
Using Array

```
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
glColorPointer(3, GL_FLOAT, 0, colors);
glVertexPointer(3, GL_INT, 0, vertices);

glBegin(GL_TRIANGLES)
  glArrayElement(2);
  glArrayElement(3);
  glArrayElement(5);
glEnd();
```



Stride Example

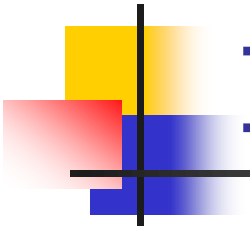
```
Static Glfloat inerwined[] = {
1.0, 0.2, 1.0, 100.0, 100.0, 0.0,
1.0, 1.2, 1.0, 200.0, 100.0, 0.0,
1.0, 1.2, 1.0, 100.0, 200.0, 0.0,
1.2, 0.2, 1.0, 200.0, 200.0, 0.0,
1.2, 0.2, 1.0, 200.0, 100.0, 0.0}

glColorPointer(3, GL_FLOAT, 6*sizeof(Glfloat), &interwined[0]);
glVertexPointer(3, GL_FLOAT, 6*sizeof(Glfloat), &interwined[3]);
```



Indirection Evaluation

- Advantages:
  - Connectivity information is easier to evaluate because vertex equality is obvious
  - Saving in storage:
    - Vertex index might be only 2 bytes, and a vertex is probably 12 bytes
    - Each vertex gets used at least 3 and generally 4-6 times, but is only stored once
  - Normals, texture coordinates, colors etc. can all be stored the same way
- Disadvantages:
  - Connectivity information is not explicit

OpenGL and Vertex Indirection



```
struct Vertex {
   float coords[3];
}
struct Triangle {
   GLuint verts[3];
}
struct Mesh {
   struct Vertex vertices[m];
   struct Triangle triangles[n];
}
```

Continued…

OpenGL and Vertex Indirection (v1)

```
Mesh mesh;
/* fill the mesh information here */

/*drawing the mesh */
glEnableClientState(GL_VERTEX_ARRAY)
glVertexPointer(3, GL_FLOAT, sizeof(struct
  Vertex), mesh.vertices);
glBegin(GL_TRIANGLES)
  for ( i = 0 ; i < n ; i++ )
  {
      glArrayElement(mesh.triangles[i].verts[0]);
      glArrayElement(mesh.triangles[i].verts[1]);
      glArrayElement(mesh.triangles[i].verts[2]);
  }
glEnd();
```

OpenGL and Vertex Indirection (v2)

```
glEnableClientState(GL_VERTEX_ARRAY)
glVertexPointer(3, GL_FLOAT, sizeof(struct Vertex),
                mesh.vertices);
for ( i = 0 ; i < n ; i++ )
   glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_INT,
                  mesh.triangles[i].verts);
```

- Minimizes amount of data sent to the renderer
- Fewer function calls
- Faster!
- Another variant restricts the range of indices that can be used - even faster because vertices may be cached
- Can even interleave arrays to pack more data in a smaller space