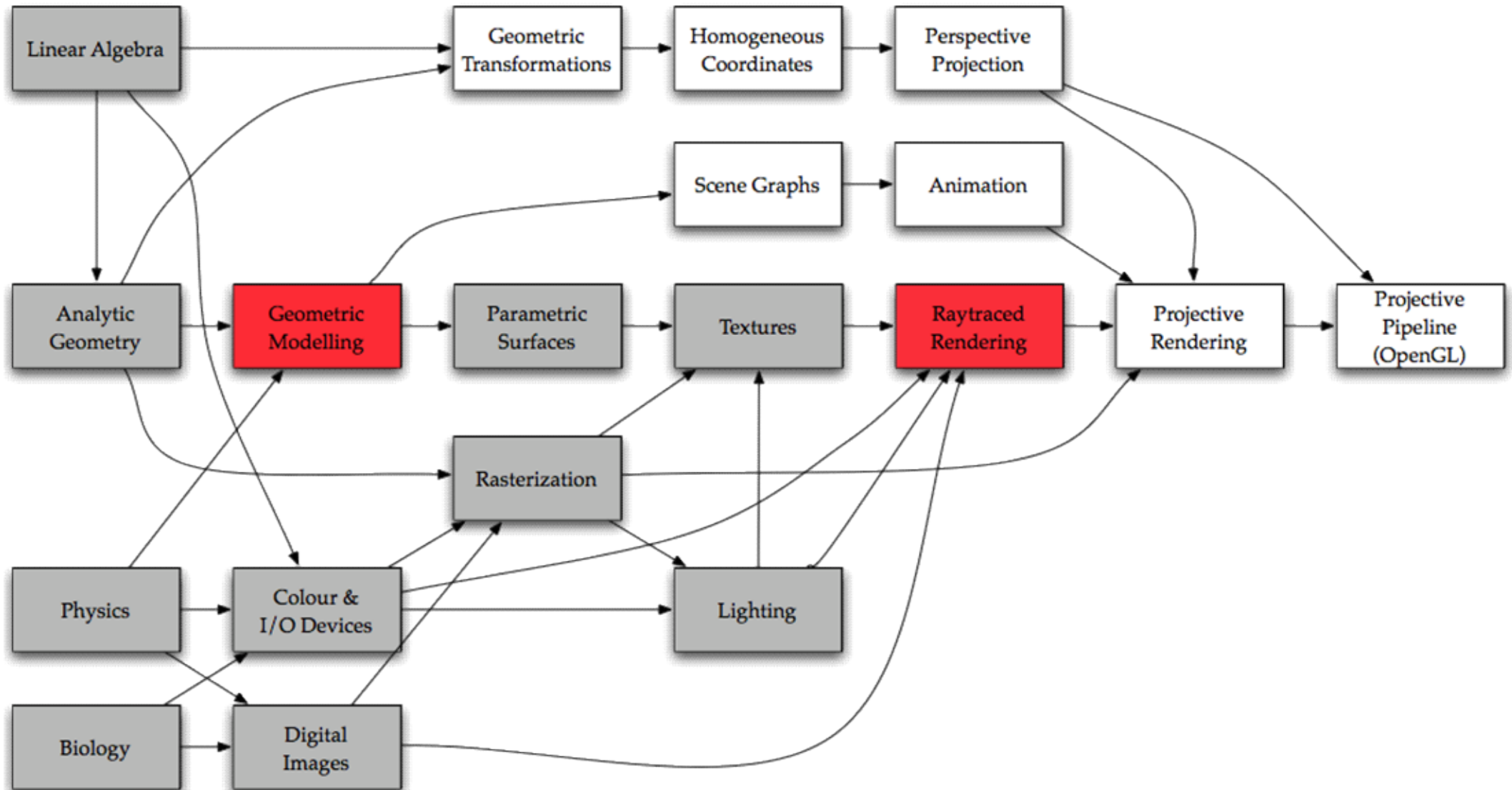


Geometry in 3 Dimensions



Where we Are



Geometric Modelling

- We need to describe the world
- Most important:
 - where an object is
 - what it is
 - how it behaves



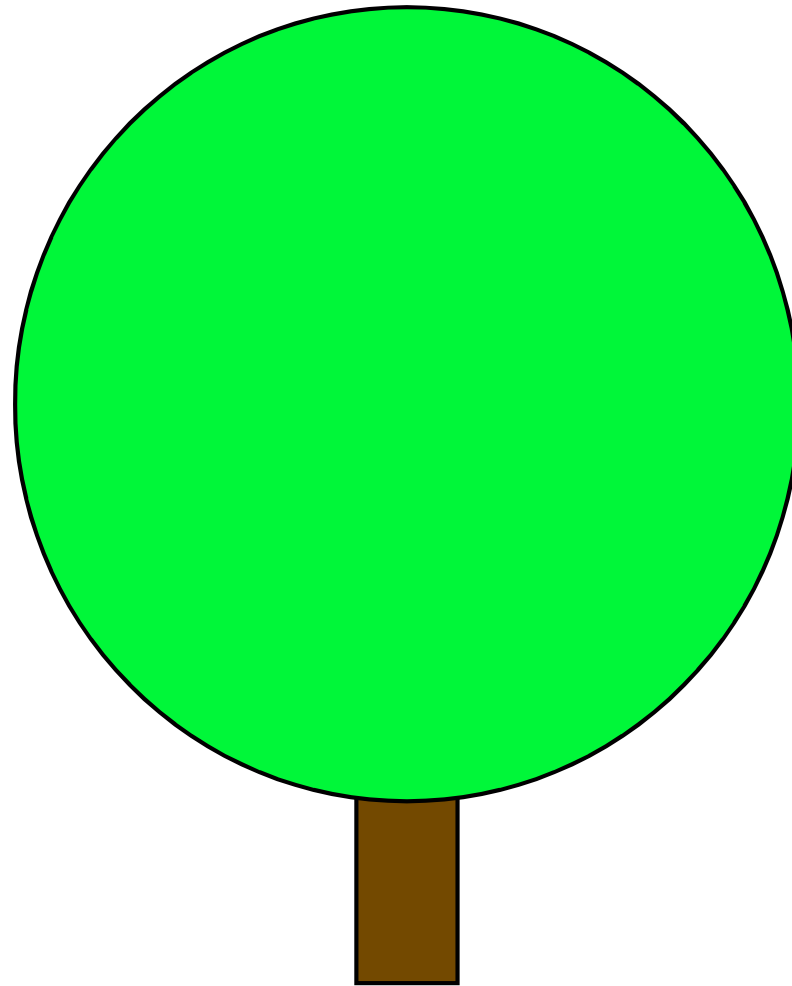
A tree



How can we describe this object?

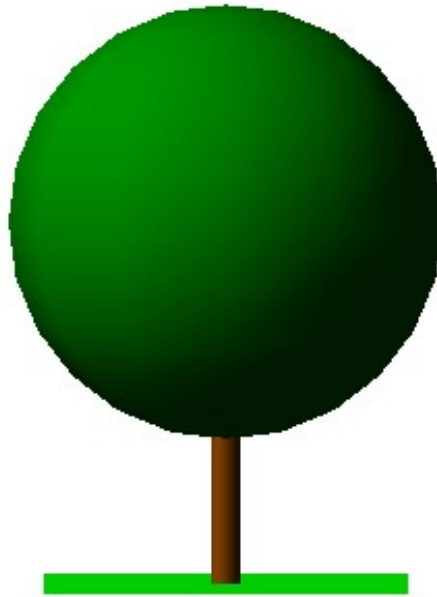


Drawing a Tree



Draw a circle on top of a rectangle

Drawing a 3D Tree



Draw a sphere on top of a cylinder

Complex Geometry

- This isn't a very good tree
- It's too simple geometrically
- A real tree has ?100,000? leaves
- But for now, let's pretend it's OK



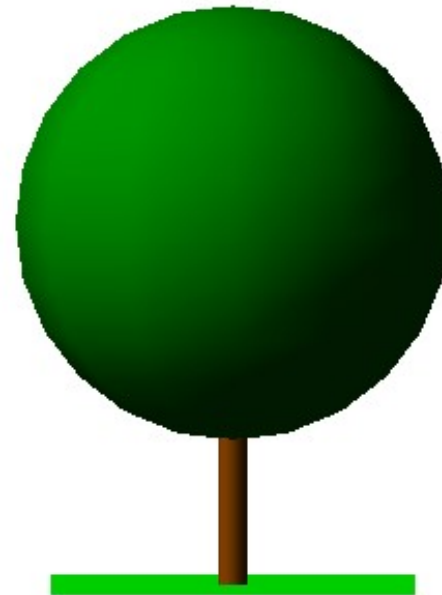
Geometric Description

- How to describe a large complex object:
 - break into smaller *primitives*
 - spheres, cylinders, boxes
 - polygons, polyhedra, prisms
 - render each primitive separately



Description

- Draw a sphere
 - radius 10 m
 - centred 7 m above ground
 - colour light green
- Draw a cylinder
 - radius 2 m, height 15 m
 - bottom face on ground
 - colour brown

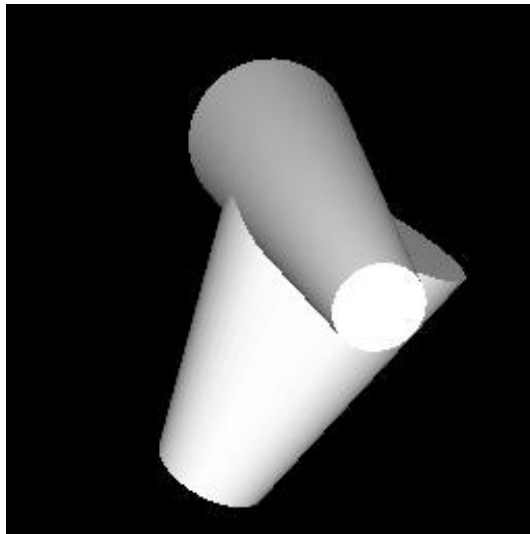


CSG: Constructive Solid Geometry

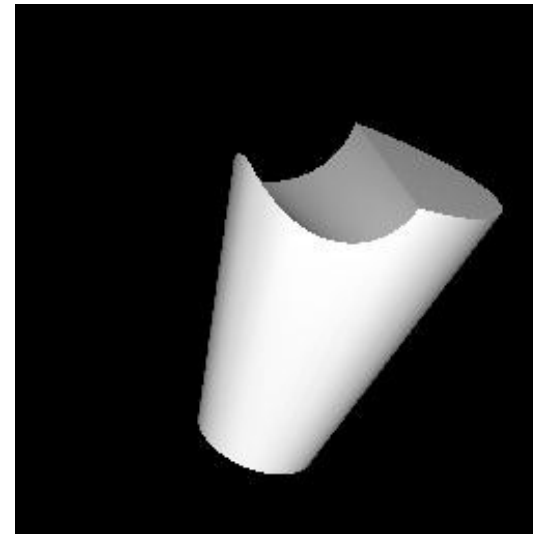
- Tree is built of smaller objects
- Formally, the *union* of them
- Can also do things like:
 - *subtract* objects
 - *xor* objects
- But not with OpenGL



CSG Example



Union



Subtraction

<http://www.cs.unc.edu/~geom/CSG/boole.html>



Building Objects

- Objects are *built up* from primitives
 - points, lines, triangles
- We will build the following:
 - *Platonic* solids
 - *Solids* of revolution



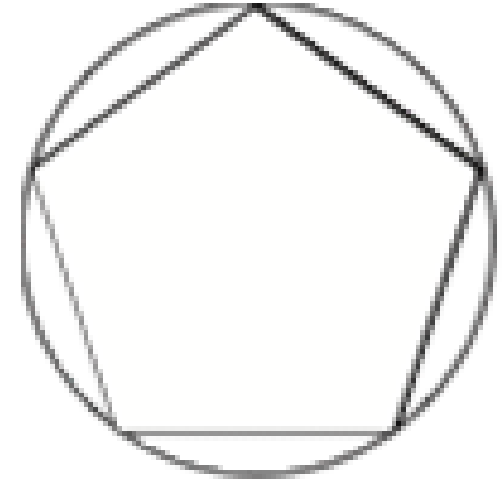
Polygons

- A *polygon* is a 2-D shape
 - triangle (3 sides)
 - square / rectangle / quadrilateral (4)
 - pentagon (5)
 - hexagon (6)
 - circle (∞)



Regular Pentagon

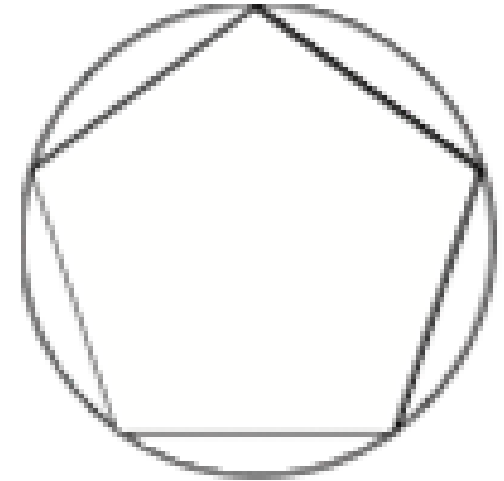
- 5 vertices on a circle
- spaced $360/5 = 72^\circ$ apart
- Edges (lines) between them



```
GL11.glBegin(GL_LINES);
for (vertex = 0; vertex < 5; vertex++)
{ // vertex loop
  theta1 = vertex * 72 / 360 * 2 * PI; // in radians
  theta2 = (vertex + 1) * 72 / 360 * 2 * PI;
  GL11.glVertex3f(sin(theta1), cos(theta1), 0.0);
  GL11.glVertex3f(sin(theta2), cos(theta2), 0.0);
} // vertex loop
GL11.glEnd();
```

Regular n -gon

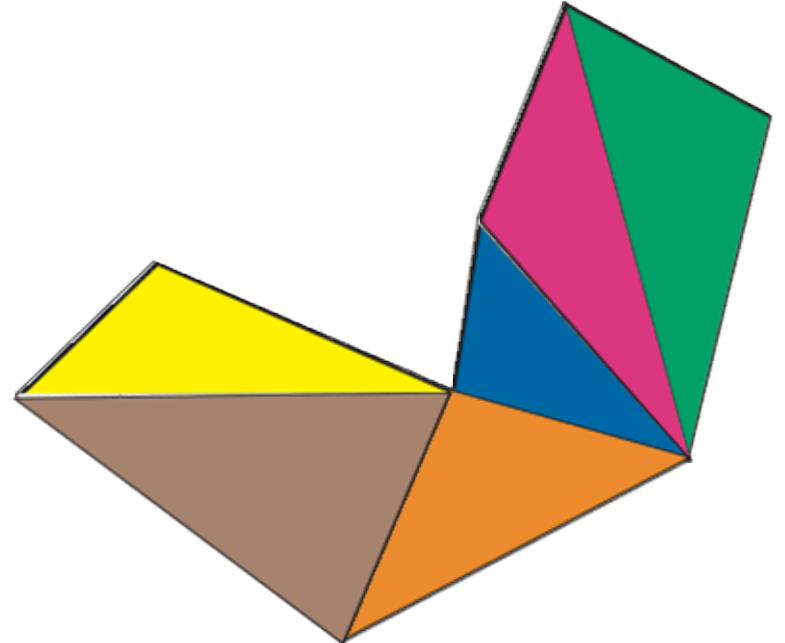
- n vertices on a circle
- spaced $360/n = 72^\circ$ apart
- Edges (lines) between them



```
GL11.glBegin(GL_LINES);
for (vertex = 0; vertex < n; vertex++)
{ // vertex loop
  theta1 = vertex * 2 * PI / n; // in radians
  theta2 = (vertex + 1) * 2 * PI / n;
  GL11.glVertex3f(sin(theta1), cos(theta1), 0.0);
  GL11.glVertex3f(sin(theta2), cos(theta2), 0.0);
} // vertex loop
GL11.glEnd();
```

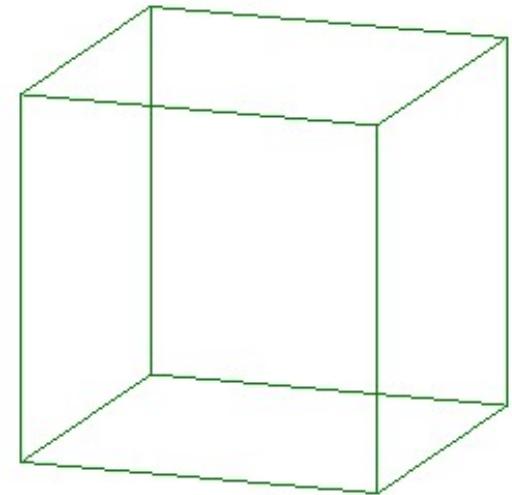
Filling Polygons

- We know how to *rasterize* triangles
 - i.e. how to draw *filled* triangles
- Any polygon can be turned into triangles
 - by cutting vertices off



Polyhedra

- A *polyhedron* is a 3-D shape
 - of *vertices*
 - connected with *edges*
- Can be rendered as lines
 - this is called *wireframe*



Platonic Solids

- Regular *convex* polyhedra
 - All faces are same size and shape
 - Each face is a regular polygon
 - Each edge is same length
- Convex = no indentations

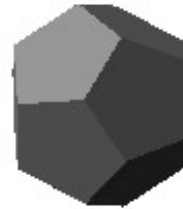


5 Platonic Solids

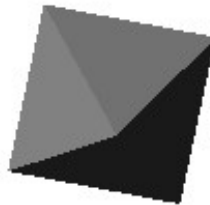
Icosahedron (20 sides)



Dodecahedron (12 sides)



Octahedron (8 sides)



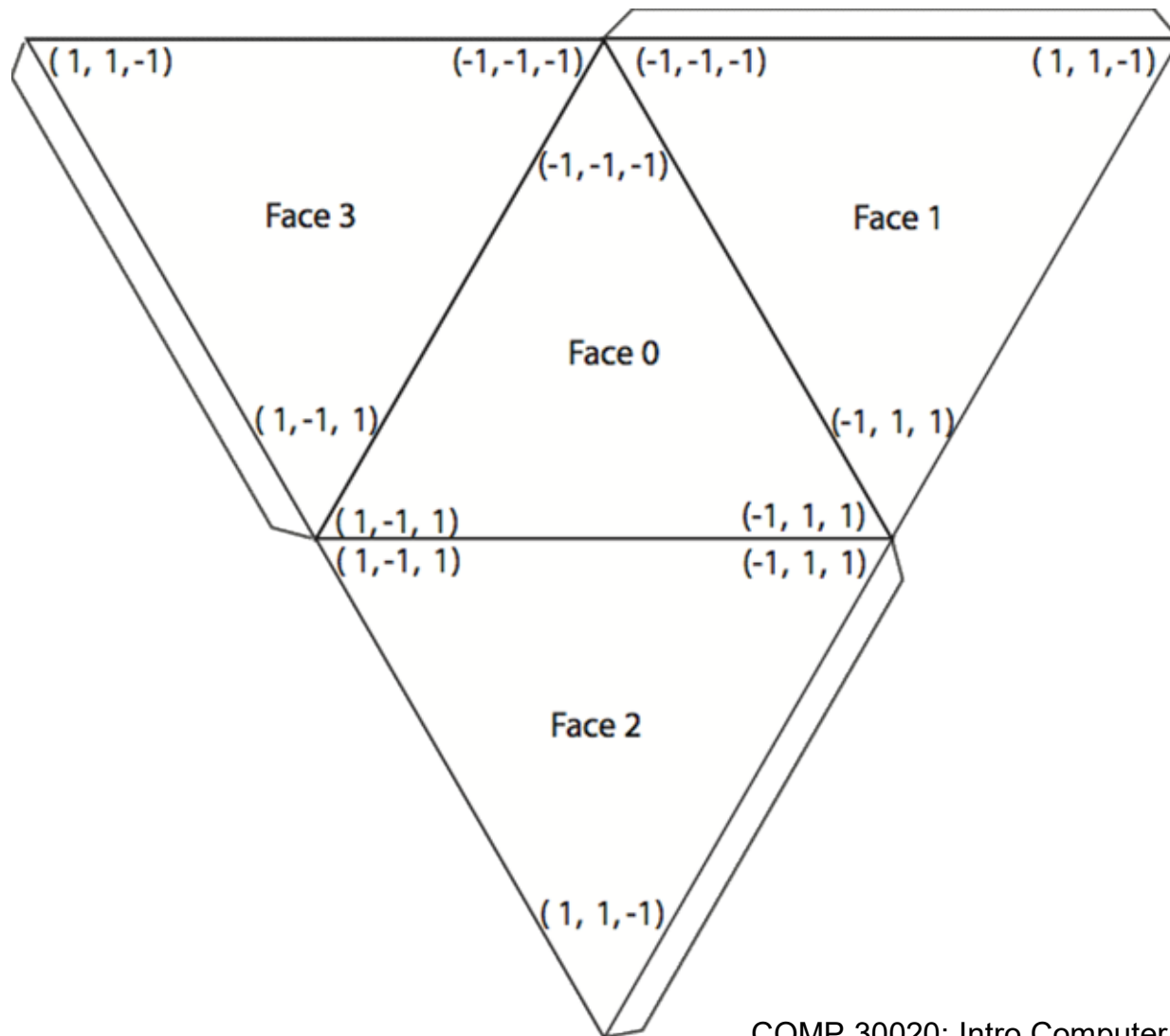
Hexahedron (6 sides)



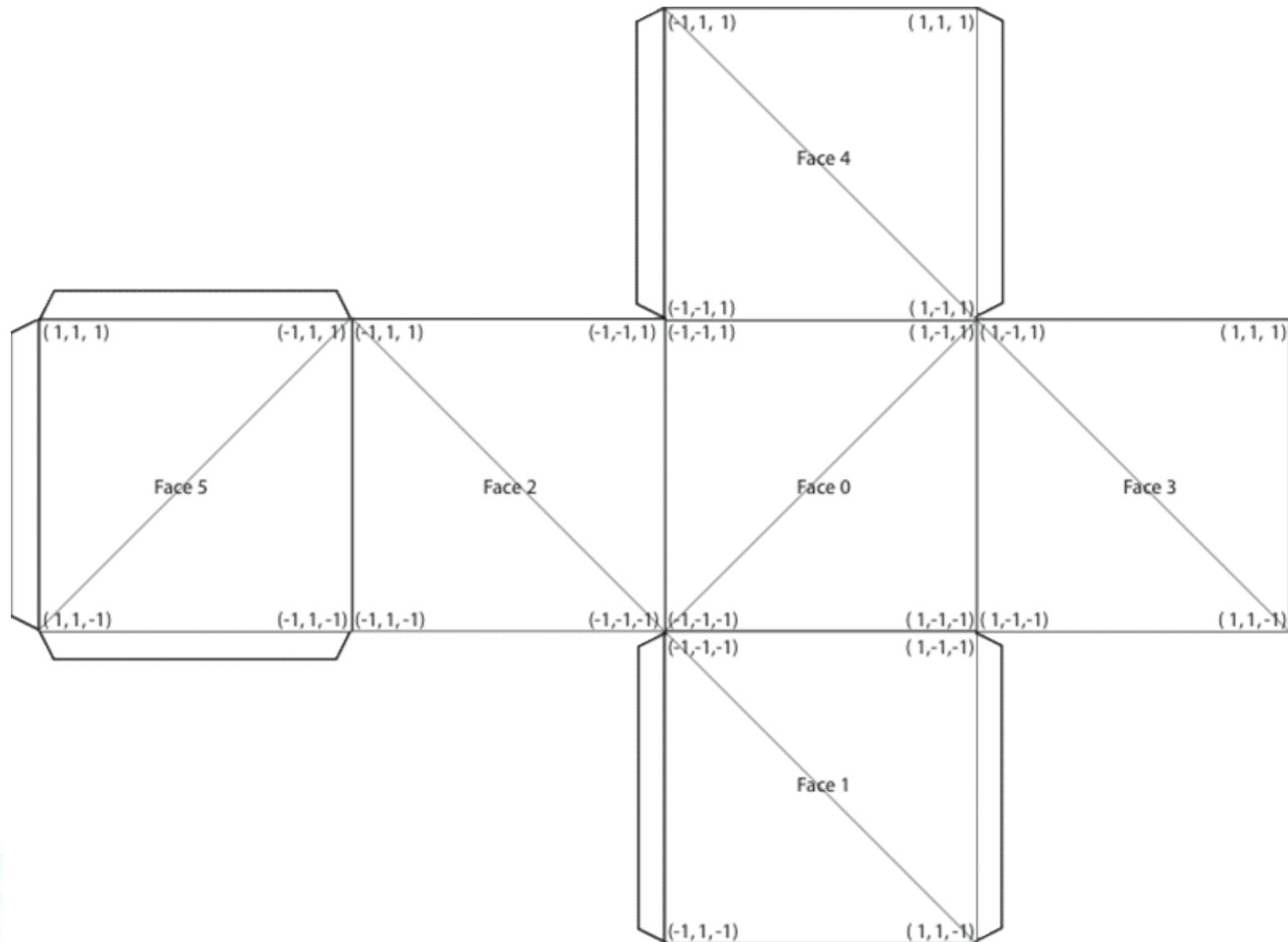
Tetrahedron (4 sides)



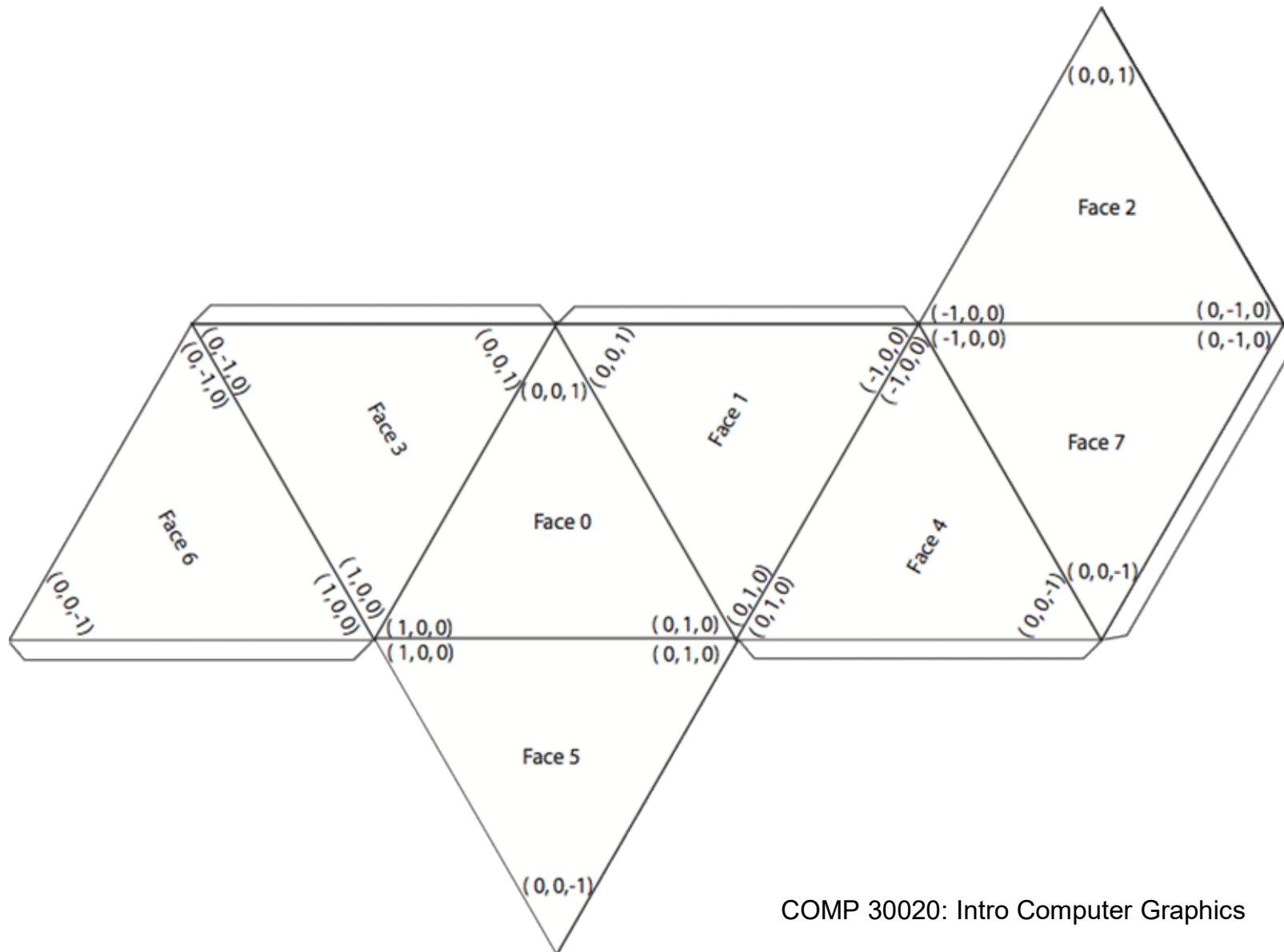
Tetrahedron



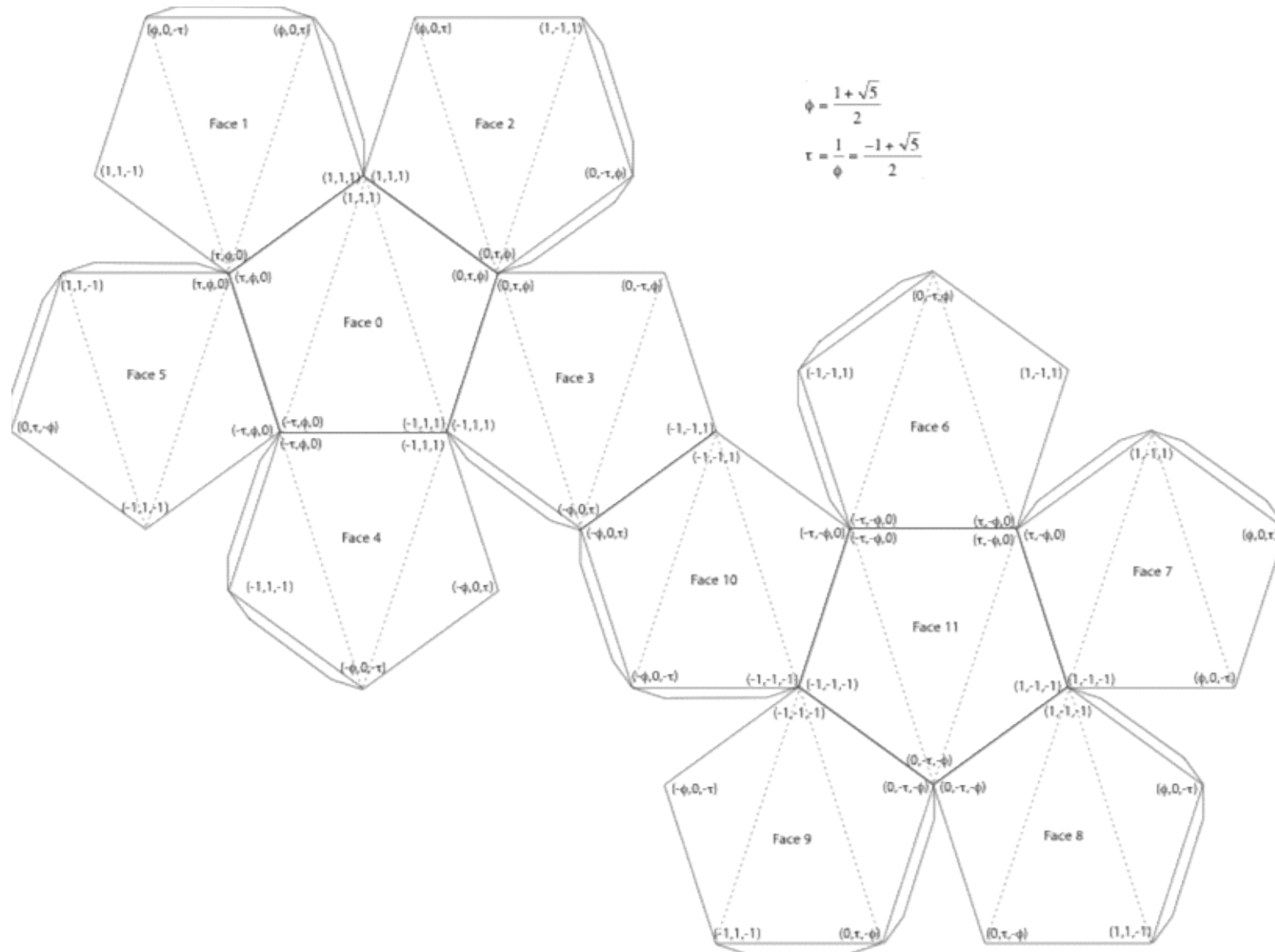
Hexahedron (Cube)



Octahedron



Dodecahedron



$$\phi = \frac{1+\sqrt{5}}{2}$$

$$\tau = \frac{1-\sqrt{5}}{2}$$

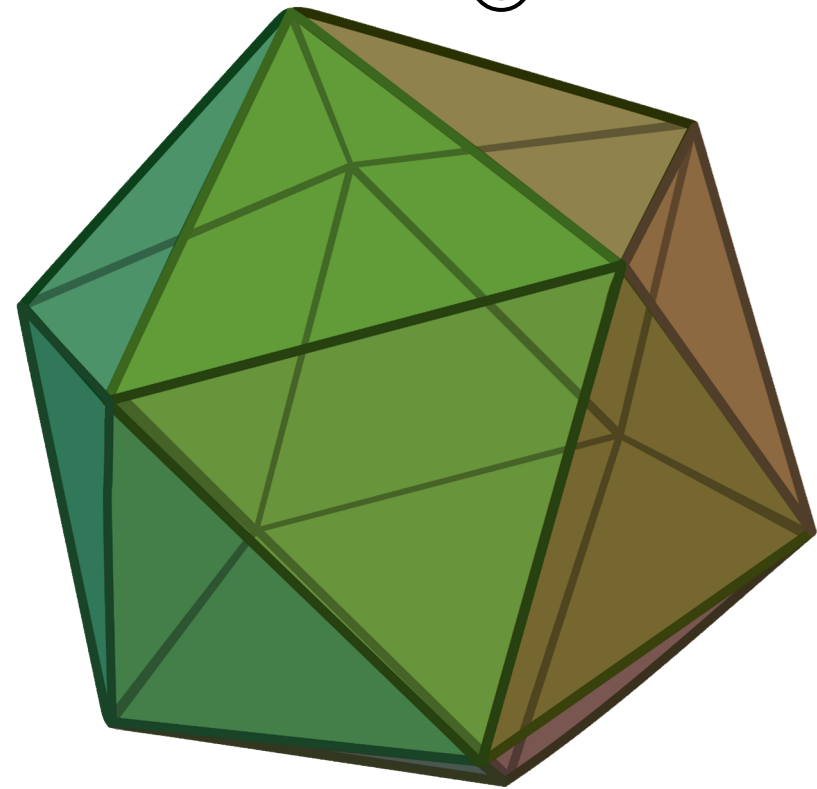
Icosahedron

Gets to difficult to make a
paper model for a D20

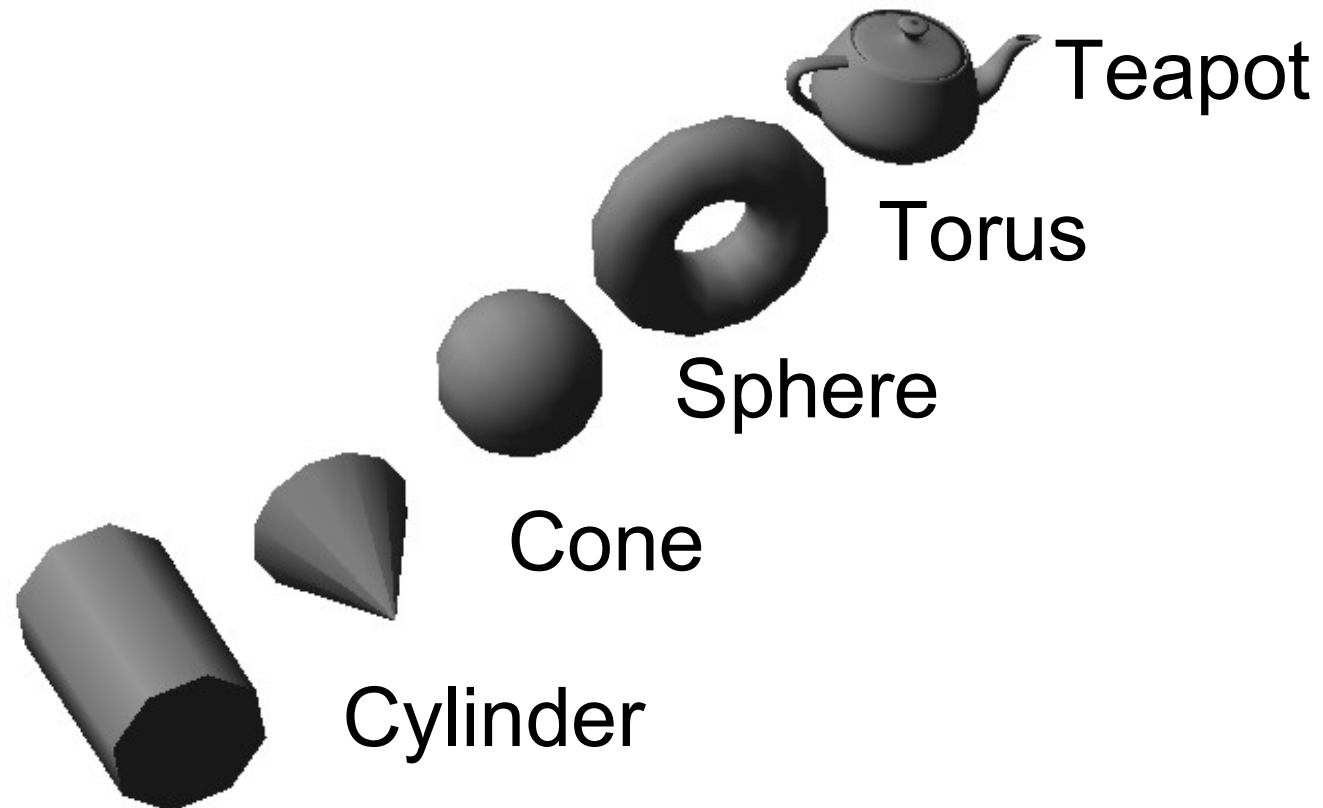


```
Point4f vertices[] = {  (-X, 0.0f, Z ,0.0f) ,   (X, 0.0f, Z,0.0f),  
    (-X, 0.0f, -Z,0.0f),   (X, 0.0f, -Z,0.0f),  
    (0.0f, Z, X,0.0f),   (0.0f, Z, -X,0.0f),  
    (0.0f, -Z, X,0.0f),   (0.0f, -Z, -X,0.0f),  
    (Z, X, 0.0f,0.0f),   (-Z, X, 0.0f,0.0f),  
    (Z, -X, 0.0f,0.0f),   (-Z, -X, 0.0f,0.0f)  
    };
```

```
int faces[][] = { {0,4,1}, {0,9,4},  
    {9,5,4}, {4,5,8},  
    {4,8,1}, {8,10,1},  
    {8,3,10},{5,3,8},  
    {5,2,3}, {2,7,3},  
    {7,10,3}, {7,6,10},  
    {7,11,6}, {11,0,6},  
    {0,1,6},{6,1,10},  
    {9,0,11}, {9,11,2},  
    {9,2,5}, {7,2,11}};
```



Some Round Objects

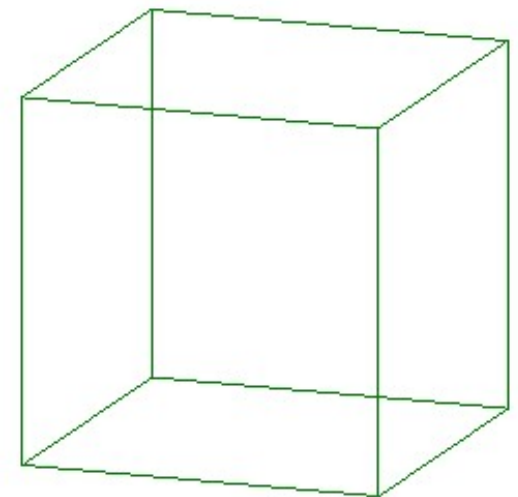


Example: A Cube

```
(-1.0, -1.0, -1.0); (-1.0, -1.0, 1.0);  
(-1.0, -1.0, 1.0); (-1.0, 1.0, 1.0);  
(-1.0, 1.0, 1.0); (-1.0, 1.0, -1.0);  
(-1.0, 1.0, -1.0); (-1.0, -1.0, -1.0);
```

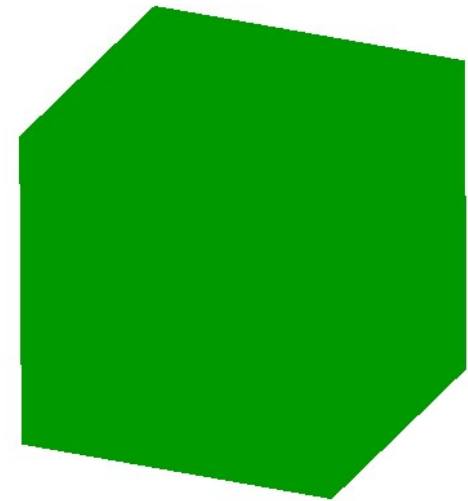
```
(-1.0, -1.0, -1.0); ( 1.0, -1.0, -1.0);  
(-1.0, -1.0, 1.0); ( 1.0, -1.0, 1.0);  
(-1.0, 1.0, 1.0); ( 1.0, 1.0, 1.0);  
(-1.0, 1.0, -1.0); ( 1.0, 1.0, -1.0);
```

```
( 1.0, -1.0, -1.0); ( 1.0, -1.0, 1.0);  
( 1.0, -1.0, 1.0); ( 1.0, 1.0, 1.0);  
( 1.0, 1.0, 1.0); ( 1.0, 1.0, -1.0);  
( 1.0, 1.0, -1.0); ( 1.0, -1.0, -1.0);
```



Solid Polyhedra

- Polyhedra also have *surfaces*
 - each *face* is a polygon
 - broken into triangles
 - specified in CCW order
 - from the outside



Example: A Cube

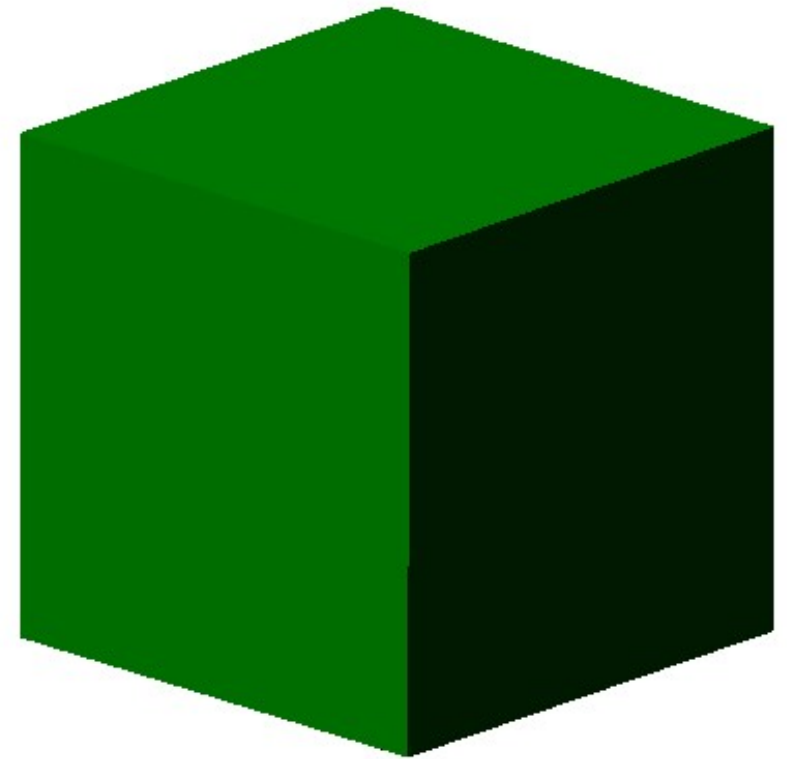
```
int vertices[8][3] =  
    {-1, -1, -1,    -1, -1,  1,    -1,  1, -1,    -1,  1,  1,  
     1, -1, -1,     1, -1,  1,     1,  1, -1,     1,  1,  1};
```

```
int triangles[12][3] = {  
    0,  1,  3,           0,  3,  2,  
    0,  2,  4,           2,  6,  4,  
    0,  1,  4,           1,  4,  5,  
    1,  5,  7,           1,  7,  3,  
    5,  4,  6,           5,  6,  7,  
    2,  3,  7,           2,  7,  6  
};
```



Shaded Polyhedra

- Each *face* lies on a plane
- we can compute a *normal*
- for reflecting light



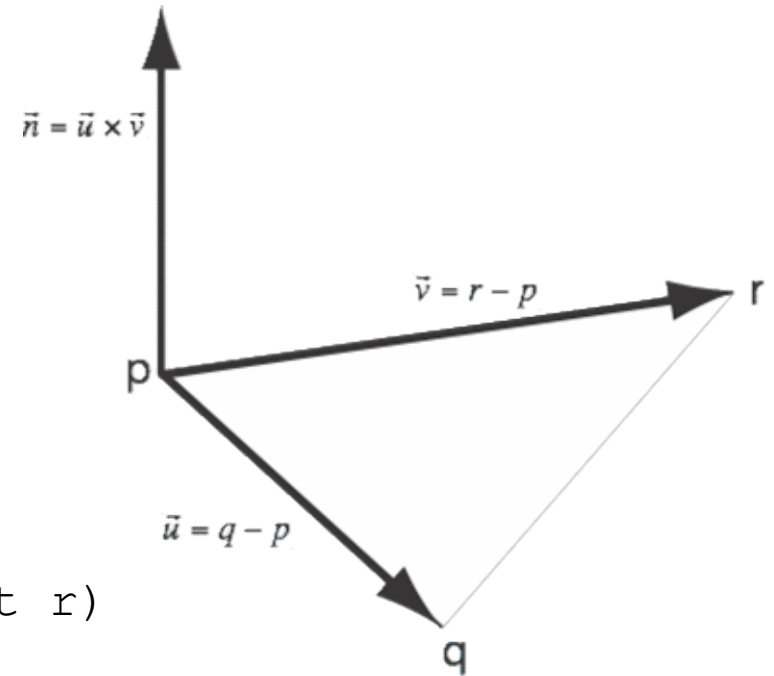
Normal for a Triangle

For each triangle pqr ,

let $\vec{u} = q - p$ and $\vec{v} = r - p$

Then

$\vec{n} = \vec{u} \times \vec{v}$ is normal to pqr



```
Vector Normal(Point p, Point q, Point r)
{ // Normal()
  Vector u = q - p;
  Vector v = r - p;
  Vector n = u.Cross(v);
  // optional - make vector unit length
  n = n / n.Length();

  return n;
} // Normal()
```



Example: A Cube

```
int vertices[8][3] =  
    {-1, -1, -1,    -1, -1,  1,    -1,  1, -1,    -1,  1,  1,  
     1, -1, -1,     1, -1,  1,     1,  1, -1,     1,  1,  1};
```

```
int triangles[12][3] = {  
    0,  1,  3,          0,  3,  2,  
    0,  1,  4,          1,  4,  5,  
    0,  2,  4,          2,  6,  4,  
    5,  4,  6,          5,  6,  7,  
    1,  5,  7,          1,  7,  3,  
    2,  3,  7,          2,  7,  6  
};
```

```
int normals[12][3] = {  
    -1,  0,  0,          -1,  0,  0,  
     0, -1,  0,          0, -1,  0,  
     0,  0, -1,          0,  0, -1,  
     1,  0,  0,          1,  0,  0,  
     0,  1,  0,          0,  1,  0,  
     0,  0,  1,          0,  0,  1  
};
```



Round Objects

- We *approximate* round objects
 - with polyhedra
 - possibly with *lots* of faces
- How can we approximate a cylinder?
 - as an *extruded* polygon
 - turn each edge into a vertical strip



A Vertical Cylinder

```
for (float i = 0.0; i < nSegments; i += 1.0)
{ /* a loop around circumference of a tube */
    float angle = PI * i * 2.0 / nSegments ;
    float nextAngle = PI * (i + 1.0) * 2.0 / nSegments;

    /* compute sin & cosine */
    float x1 = sin(angle), y1 = cos(angle);
    float x2 = sin(nextAngle), y2 = cos(nextAngle);

    /* draw top (green) triangle */
    VertexAt(x1, y1, 0.0);
    VertexAt(x2, y2, 0.0);
    VertexAt(x1, y1, 1.0);

    /* draw bottom (red) triangle */
    VertexAt(x1, y1, 0.0);
    VertexAt(x2, y2, 0.0);
    VertexAt(x2, y2, 1.0);

} /* a loop around circumference of a tube */
```

$(x_1, y_1, 1.0)$

$(x_2, y_2, 1.0)$

$(x_1, y_1, 0.0)$

$(x_2, y_2, 0.0)$

COMP 30020: Intro Computer Graphics



Smoother Cylinders

- How can we make the cylinder *smoother*?
 - provide normals for *each* vertex
 - based on the *tangent plane*
- Cylinder is vertical, so normal is horizontal
 - sticking straight out through each vertex



A Smoother Cylinder

```
for (float i = 0.0; i < nSegments; i += 1.0)
{ /* a loop around circumference of a tube */
    float angle = PI * i * 2.0 / nSegments ;
    float nextAngle = PI * (i + 1.0) * 2.0 / nSegments;

    /* compute sin & cosine */
    float x1 = sin(angle), y1 = cos(angle);
    float x2 = sin(nextAngle), y2 = cos(nextAngle);

    /* draw top (green) triangle */
    NormalIs(x1, y1, 0.0); VertexAt(x1, y1, 0.0);
    NormalIs(x2, y2, 0.0); VertexAt(x2, y2, 1.0);
    NormalIs(x1, y1, 0.0); VertexAt(x1, y1, 1.0);

    /* draw bottom (red) triangle */
    NormalIs(x1, y1, 0.0); VertexAt(x1, y1, 0.0);
    NormalIs(x2, y2, 0.0); VertexAt(x2, y2, 0.0);
    NormalIs(x2, y2, 0.0); VertexAt(x2, y2, 1.0);

} /* a loop around circumference of a tube */
```



Rendering Triangles

- Each triangle lies on a plane
- So compute where ray intersects plane
- Find barycentric coordinates at intersection
- Compute shading
- Interpolate texture coordinates



Point-Plane Intersection

Given a triangle Δabc and a line $\vec{l} = q + \vec{w}t$

Let $\vec{u} = b - a$ and let $\vec{v} = c - a$

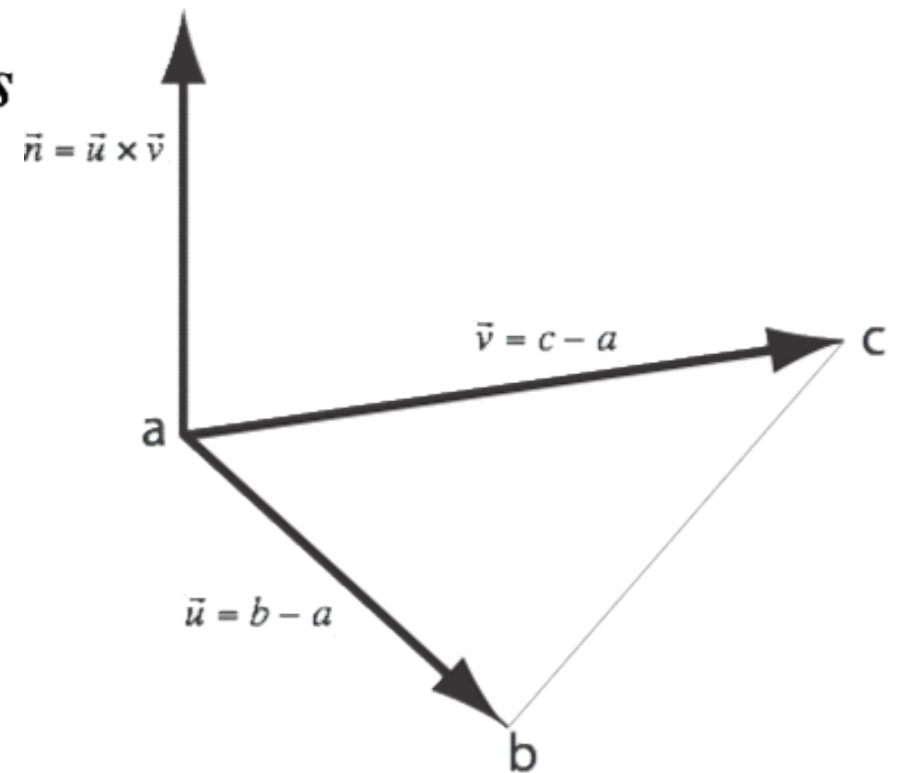
Then Δabc is on plane $\Pi = a + \vec{u}r + \vec{v}s$

Find the intersection of these two:

$$q + \vec{w}t = a + \beta\vec{u} + \gamma\vec{v}$$

Solve for t, β and γ :

$$\begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix} + \begin{bmatrix} w_x \\ w_y \\ w_z \end{bmatrix} t = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} + \beta \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} + \gamma \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$



Continued . . .

$$\beta \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} + \gamma \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} + \begin{bmatrix} w_x \\ w_y \\ w_z \end{bmatrix} t = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} - \begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix}$$

$$\begin{bmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} a_x - q_x \\ a_y - q_y \\ a_z - q_z \end{bmatrix}$$

$$\begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{bmatrix}^{-1} \begin{bmatrix} a_x - q_x \\ a_y - q_y \\ a_z - q_z \end{bmatrix}$$



Uggggh!

- This is bad enough
 - it gets *worse* for curved surfaces
 - and it's slow
- We want a simpler method for rendering
 - projective rendering

