Announcement

Homework 2 has been posted online and in dropbox

Due: 1:15 pm, Wednesday, October 5

Both undergraduate and graduate students must answer the first 4 questions. The graduate students must answer the 5th question.

Today’s Agenda

An Example of Using Transformations

Viewing

Frames in OpenGL

4D Homogenous Coordinates

Object (or model) coordinates

World coordinates

Eye (or camera) coordinates

Clip coordinates

Frames for the application

Centered at the camera origin

3D Coordinates

Normalized device coordinates

2D+depth Coordinates

Window (or screen) coordinates

Implementation of the pipeline

Two Important Transformations in OpenGL

Object (or model) coordinates

World coordinates

Eye (or camera) coordinates

Clip coordinates

Normalized device coordinates

Window (or screen) coordinates

Model-view transformation

Projection transformation

Using the Model-view Matrix

In OpenGL the model-view matrix is used to

- Position the camera
  - Can be done by rotations and translations but is often easier to use a LookAt function
- Build models of objects

The projection matrix is used to define the view volume and to select a camera lens

Although these matrices are no longer part of the OpenGL state, it is usually a good strategy to create them in our own applications

An Example of Using Transformations

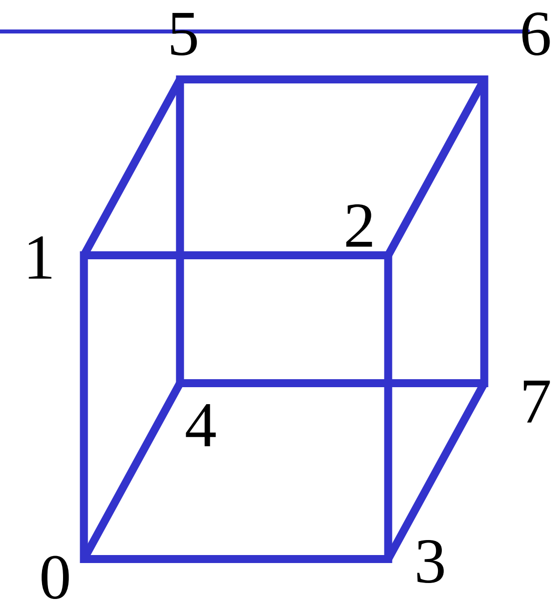
Problem: build a cube and use idle function to rotate a cube and mouse function to change direction of rotation

Start with a program that draws a cube in a standard way

- Centered at origin
- Sides aligned with axes

Representing a Mesh

Consider a mesh



There are 8 vertices,12 edges, and 6 polygons

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 OpenGL code such as

```
vertex[i] = vec3(x1, y1, z1);
vertex[i+1] = vec3(x2, y2, z2);
vertex[i+2] = vec3(x3, y3, z3);
i+=3;
```

Inefficient and unstructured

- Consider moving a vertex to a new location
- Must search for all occurrences

Define a Polygon: Inward and Outward Facing Polygons

The order $\{v_0,v_3,v_2,v_1\}$ and $\{v_3,v_2,v_1,v_0\}$ are equivalent: the same polygon will be rendered by OpenGL but the order $\{v_1,v_2,v_3,v_0\}$ is different

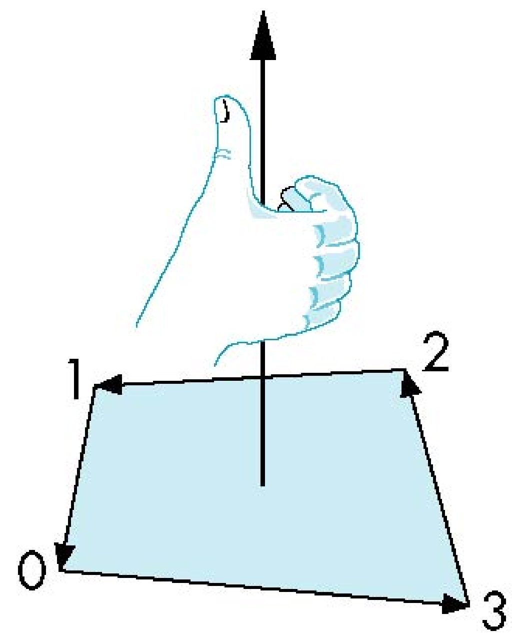
The first two describe ***outward facing*** polygons for the front face using

right-hand rule = counter-clockwise encirclement of outward-pointing normal

The third one defines an ***inward-facing*** for the back face

OpenGL 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

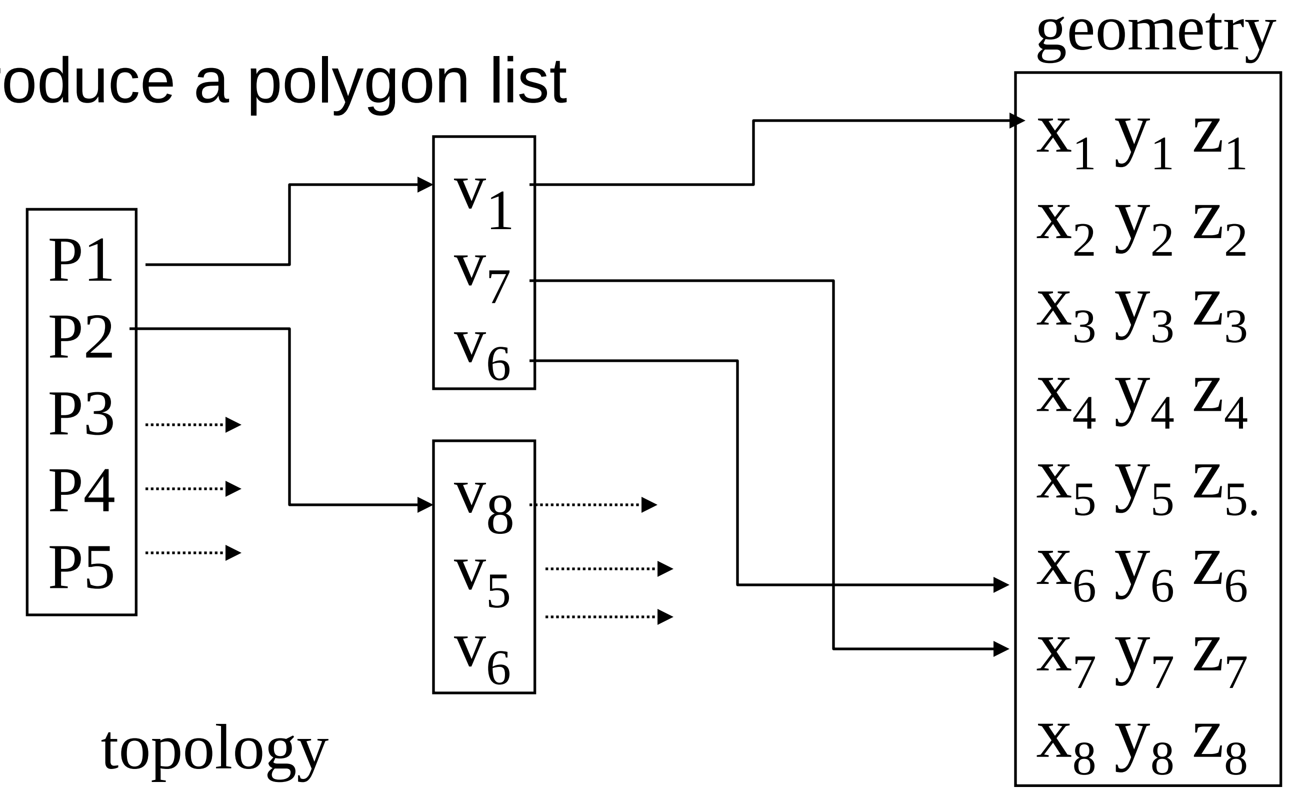
- Geometry: locations of the vertices
- Topology: organization of the vertices and edges
  - a polygon is an ordered list of vertices with an edge connecting successive pairs of vertices and the last to the first
  - For the example of cubic
    - Each vertex is shared by 3 faces
    - Pairs of vertices define edges
    - Each edge is shared by two faces
- Topology holds even if geometry changes

Vertex Lists

Put the geometry in an array

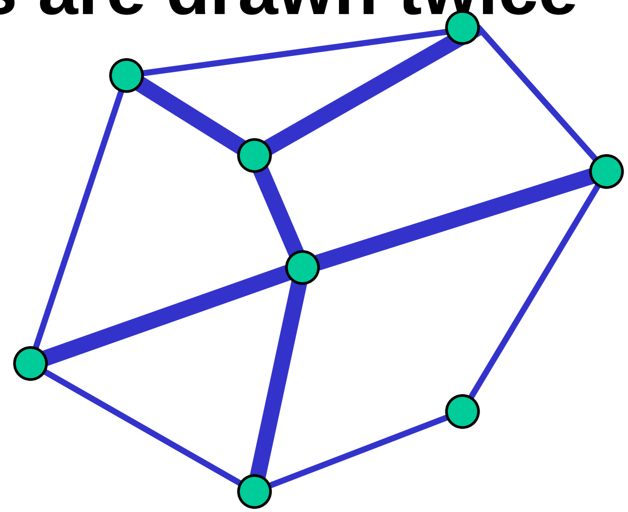
Use pointers from the vertices into this array

Introduce a polygon list



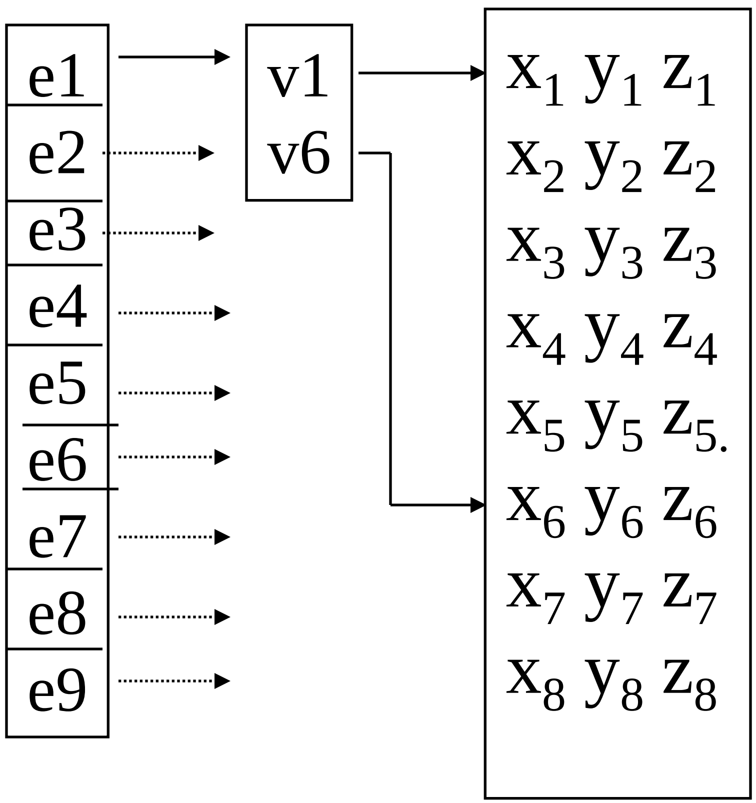
Shared Edges

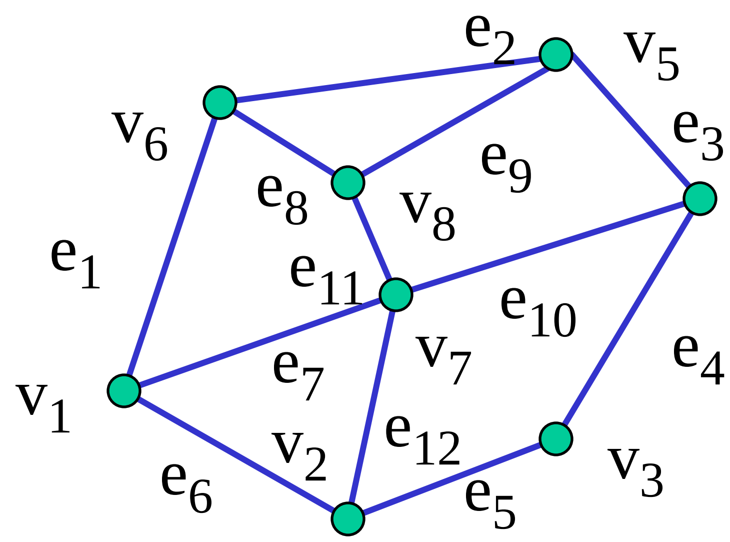
Vertex lists will draw filled polygons correctly but if we draw the polygon by its edges, shared edges are drawn twice



Can store mesh by *edge list*

Edge List





Note polygons are not represented

Modeling a Cube

Define global arrays for vertices and colors

```
typedef vex4 point4;
point4 vertices[8] = {point4(-1.0,-1.0,-1.0,1.0),
  point4(1.0,-1.0,-1.0,1.0), point4(1.0,1.0,-1.0,1.0),
  point4(-1.0,1.0,-1.0,1.0), point4(-1.0,-1.0,1.0,1.0),
  point4(1.0,-1.0,1.0,1.0), point4(1.0,1.0,1.0,1.0),
  point4(-1.0,1.0,1.0,1.0)};

  typedef vec4 color4;
  color4 colors[8] = {color4(0.0,0.0,0.0,1.0),
    color4(1.0,0.0,0.0,1.0), color4(1.0,1.0,0.0,1.0),
    color4(0.0,1.0,0.0,1.0), color4(0.0,0.0,1.0,1.0),
    color4(1.0,0.0,1.0,1.0), color4(1.0,1.0,1.0,1.0),
    color4(0.0,1.0,1.0,1.0});
```

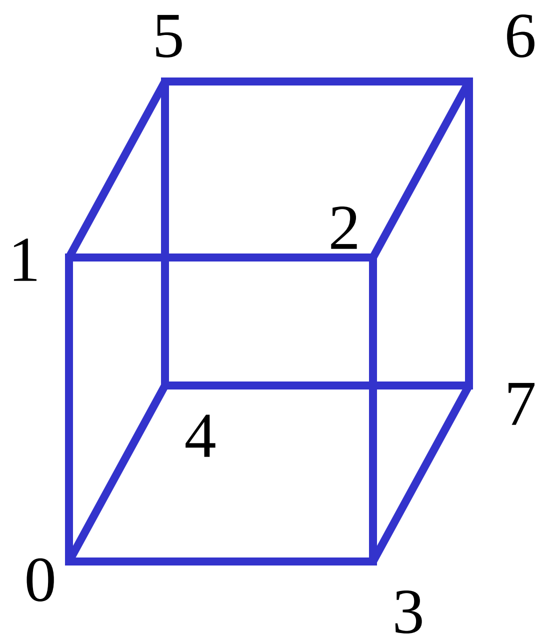
Drawing a triangle from a list of indices

Draw two triangles from a list of indices `vertices` for each face and assign color to each index

```
int Index = 0;
void quad( int a, int b, int c, int d )
{
    colors[Index] = vertex_colors[a]; points[Index] = vertices[a]; Index++;
    colors[Index] = vertex_colors[b]; points[Index] = vertices[b]; Index++;
    colors[Index] = vertex_colors[c]; points[Index] = vertices[c]; Index++;
    colors[Index] = vertex_colors[a]; points[Index] = vertices[a]; Index++;
    colors[Index] = vertex_colors[c]; points[Index] = vertices[c]; Index++;
    colors[Index] = vertex_colors[d]; points[Index] = vertices[d]; Index++;
}
```

Draw cube from faces

```
void 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

Efficiency

The weakness of our approach is that we are building the model in the application and must do many function calls to draw the cube

Drawing a cube by its faces in the most straight forward way used to require

- 6 **`glBegin`**, 6 **`glEnd`**
- 6 **`glColor`**
- 24 **`glVertex`**
- More if we use texture and lighting

Spinning the Cube

```
void main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |
       GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow("colorcube");
    glutReshapeFunc(myReshape);
    glutDisplayFunc(display);
    glutIdleFunc(spinCube);
    glutMouseFunc(mouse);
    glEnable(GL_DEPTH_TEST);
    glutMainLoop();
}
```

Idle and Mouse callbacks

```
void spinCube()
{
      theta[axis] += 2.0;
      if( theta[axis] > 360.0 ) theta[axis] -=
360.0;
      glutPostRedisplay();
}

void mouse(int btn, int state, int x, int y)
{
   if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN)
          axis = 0;
   if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN)
          axis = 1;
   if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
          axis = 2;
}
```

Display callback

We can form matrix in application and send to shader and let shader do the rotation or we can send the angle and axis to the shader and let the shader form the transformation matrix and then do the rotation

More efficient than transforming data in application and resending the data

```
void display()
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glUniform3fv(theta, 1, Theta ); //or glUniformMatrix
   glDrawArrays(GL_TRIANGLES, 0, NumVertices );
   glutSwapBuffers();
}
```

Vertex Shader

```
#version 150
in  vec4 vPosition;
in  vec4 vColor;
out vec4 color;
uniform vec3 theta;

void main()
{
    // Compute the sines and cosines of theta for each of
    //   the three axes in one computation.
    vec3 angles = radians( theta );
    vec3 c = cos( angles );
    vec3 s = sin( angles );
      mat4 rx = mat4( 1.0,  0.0,  0.0, 0.0,
                      0.0,  c.x,  s.x, 0.0,
                      0.0, -s.x,  c.x, 0.0,
                      0.0,  0.0,  0.0, 1.0 );
…
```

Vertex Shader

```
…
     mat4 ry = mat4( c.y, 0.0, -s.y, 0.0,
                     0.0, 1.0,  0.0, 0.0,
                     s.y, 0.0,  c.y, 0.0,
                     0.0, 0.0,  0.0, 1.0 );

     mat4 rz = mat4( c.z, -s.z, 0.0, 0.0,
                     s.z,  c.z, 0.0, 0.0,
                     0.0,  0.0, 1.0, 0.0,
                     0.0,  0.0, 0.0, 1.0 );


   color = vColor;
   gl_Position = rz * ry * rx * vPosition;
}
```

Fragment Shader

```
#version 150
in  vec4 color;
out vec4 fColor;


void main()
{
   fColor = color;
}
```

Reading Assignment

Angel and Shreiner, Chapter 3.13 and 3.14

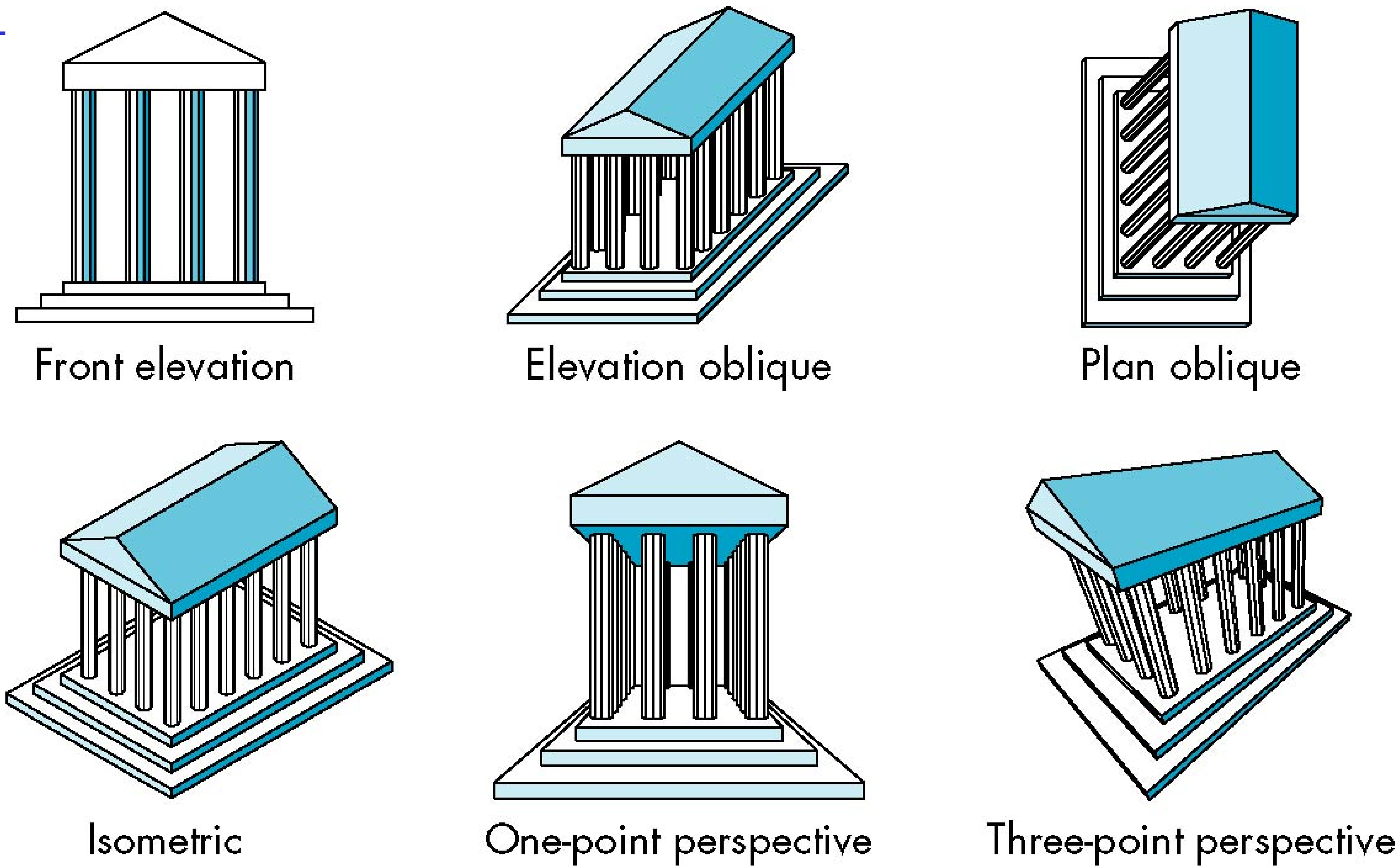
Classical Viewing vs Computer Viewing

Classical viewing: images formed by architects, artists, and engineers, e.g.,

- Isometrics, elevations, etc.
- Each object is assumed to constructed from flat *principal faces*
  - Buildings, polyhedra, manufactured objects
  - Many of them have three orthogonal directions

Computer viewing: image generated by a computer graphics system

Classical Projections



Three Basic Elements in Viewing

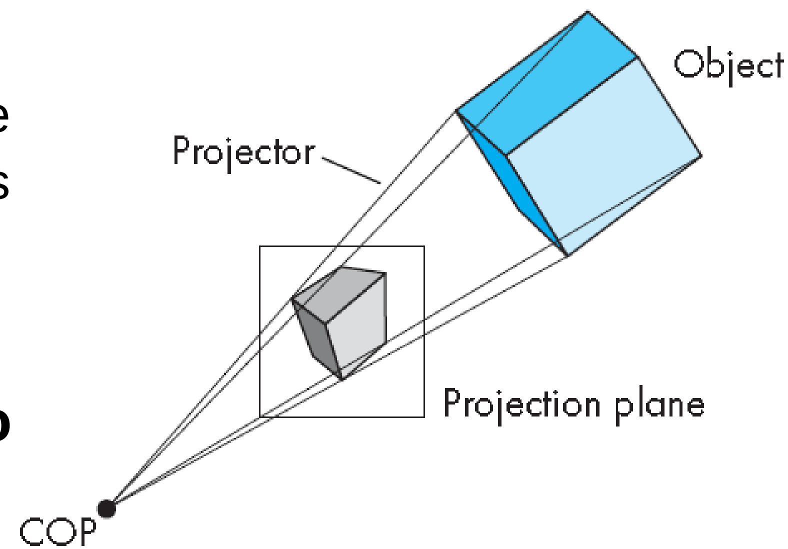
One or more objects

A viewer with a projection surface

- Planar geometric projections
  - standard projections project onto a plane
  - preserve lines but not necessarily angles
- Nonplanar projections are needed for applications such as map construction

Projectors that go from the object(s) to the projection surface

- Projectors are lines that either
  - converge at a center of projection
  - are parallel



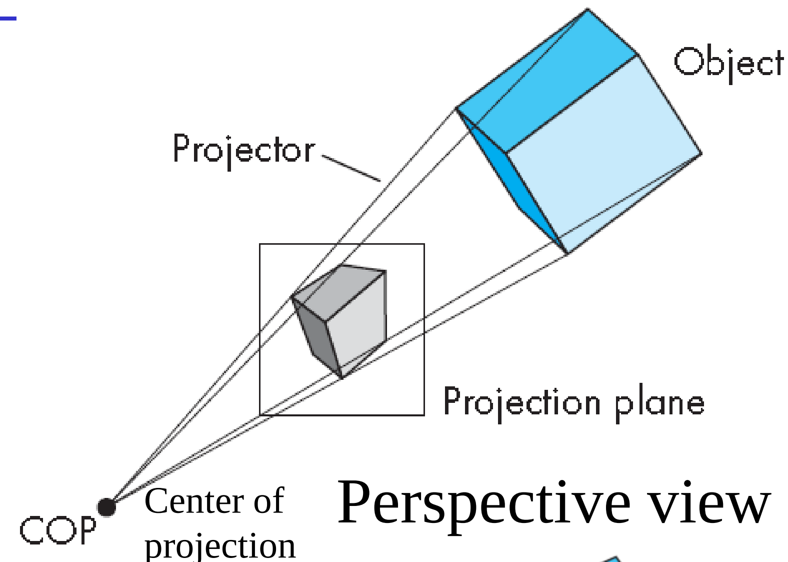
Perspective vs Parallel

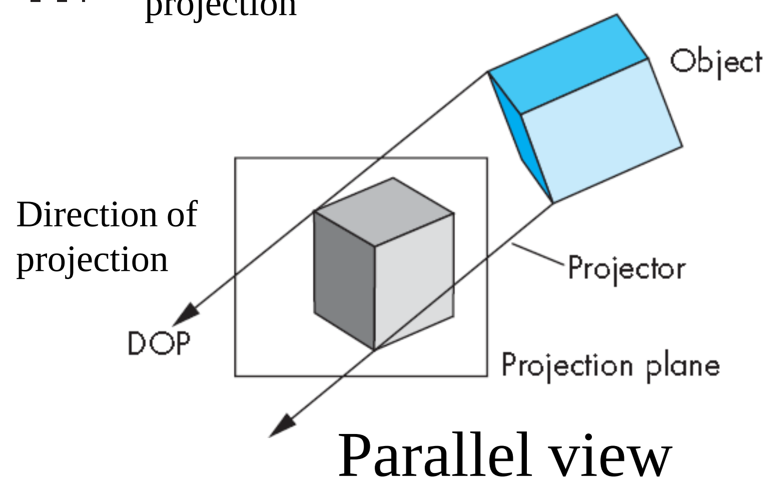
Mathematically parallel viewing is the limit of perspective viewing

- Parallel viewing does not look real because far objects are scaled the same as near objects

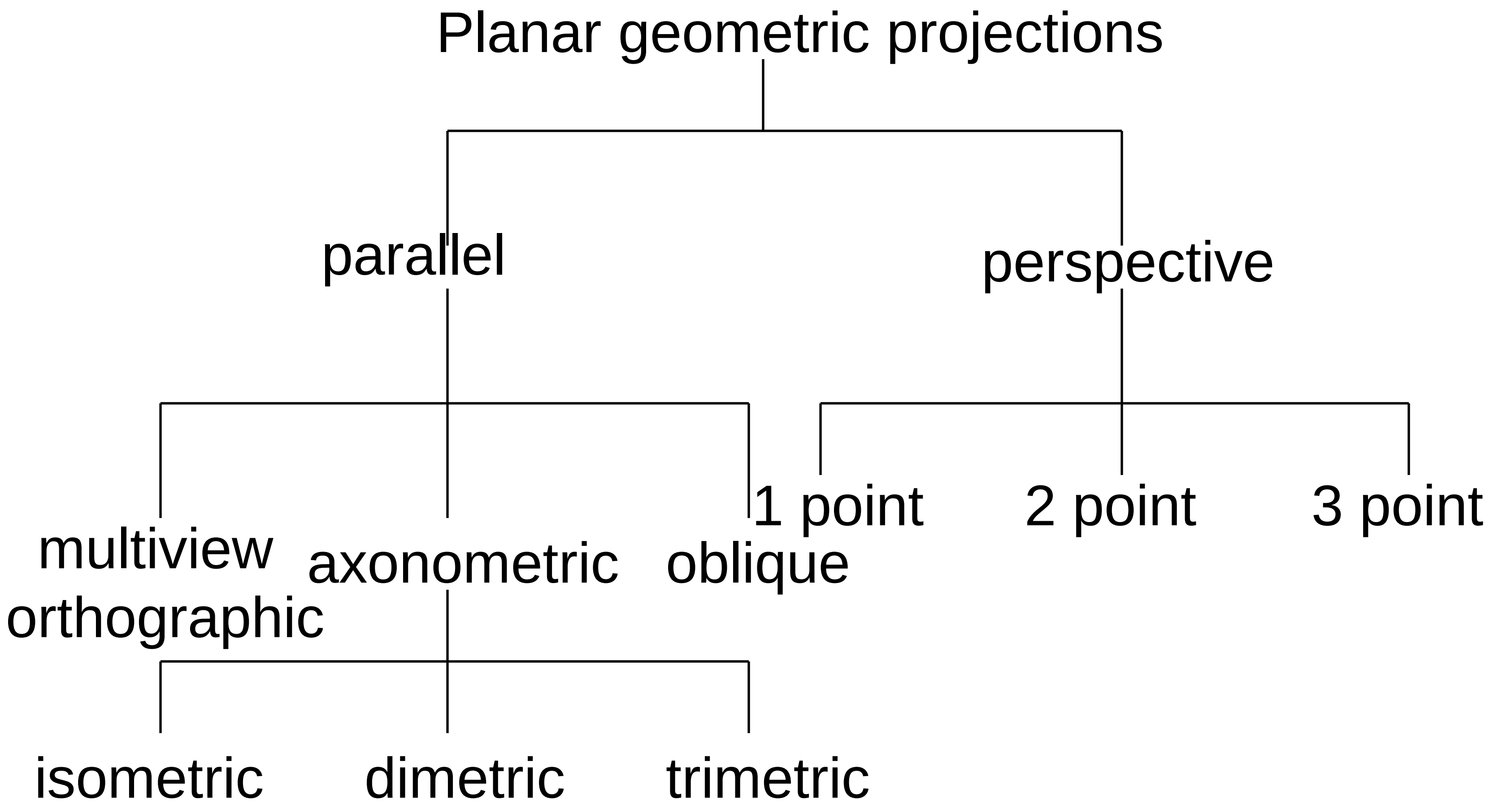
Fundamental distinction is between parallel and perspective viewing

- Classical viewing developed different techniques for drawing each type of projection
- Computer viewing employed two different type of views via the same pipeline



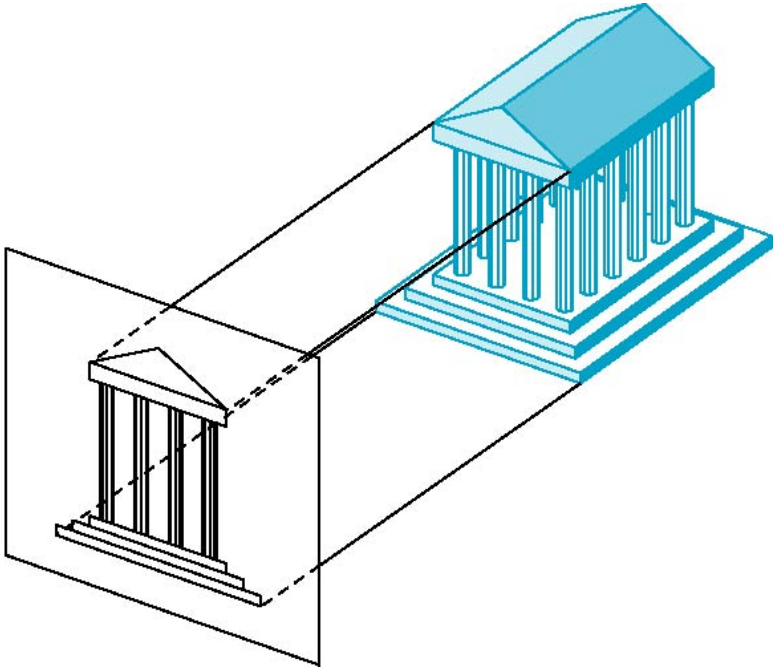


Taxonomy of Planar Geometric Projections



Orthographic Projection

Projectors are orthogonal (perpendicular) to projection plane



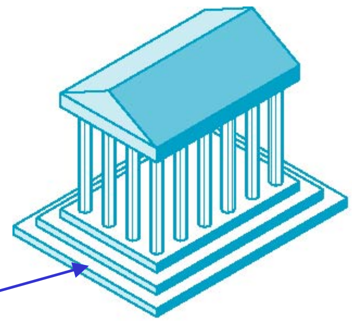
Preserve distances and angles

Multiview Orthographic Projection

Multiple projections

- In each projection, the projection plane is parallel to one principal face
- Only show the faces parallel to the projection plane

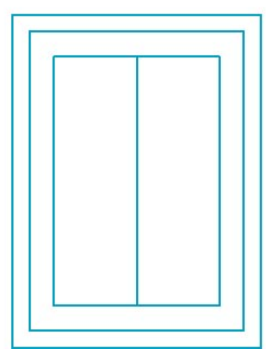
In CAD and architecture, we often display three multiviews plus isometric

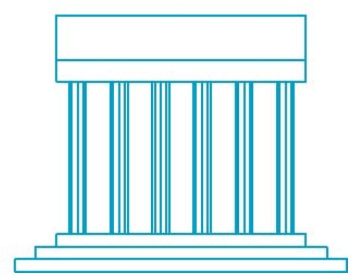


front

Note: isometric is not part of multiview orthographic view)

top





side

Multiview Orthographic Projection

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

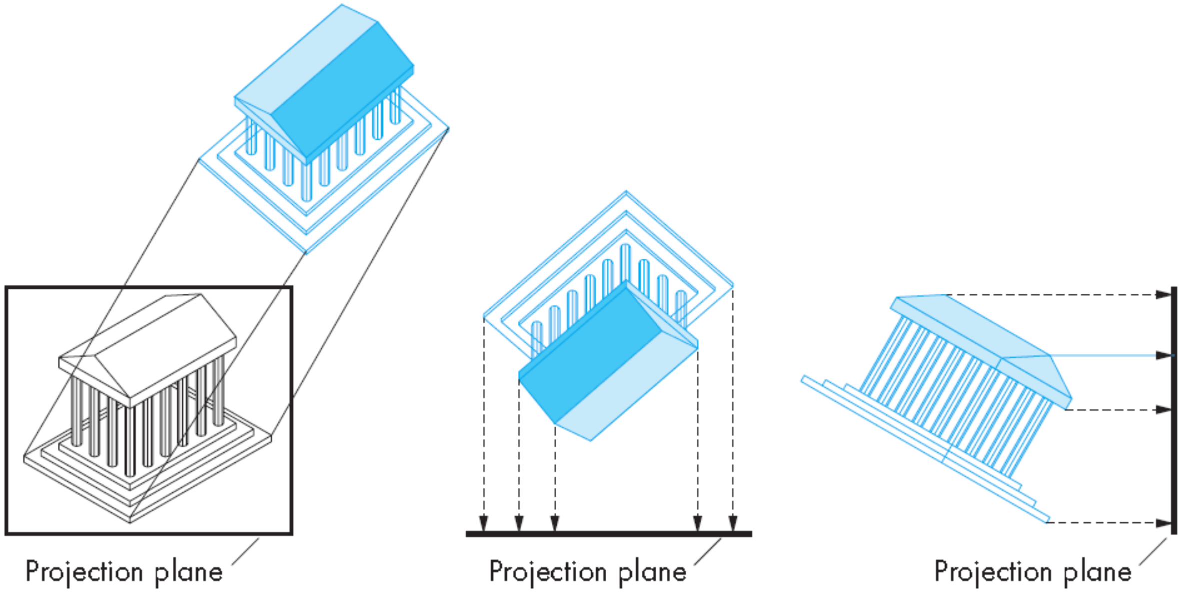
Axonometric Projections

Motivation: Allow the viewer to see more principal faces

Projectors are still orthogonal to the projection plane

Object can move relative to the projection plane

- The projection plane may be not parallel to the principal face

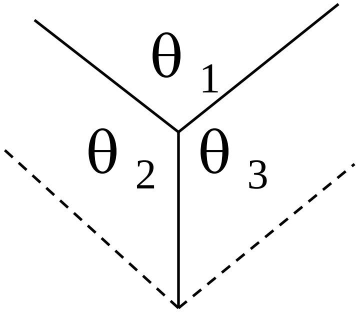


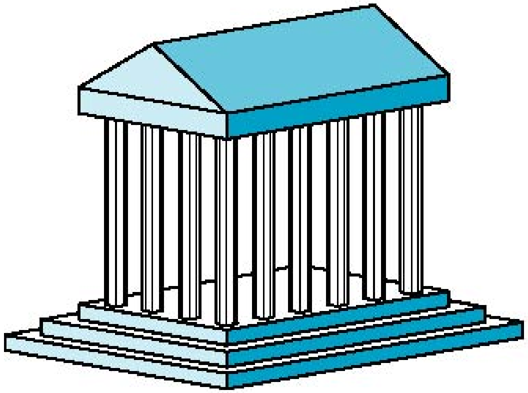
E. Angel and D. Shreiner

Types of Axonometric Projections

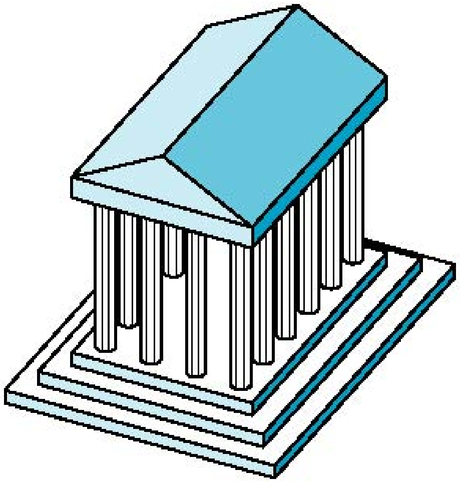
Classify by how many angles of a corner of a projected cube are the same:

- Trimetric: none
- Dimetric: two
- Isometric: three

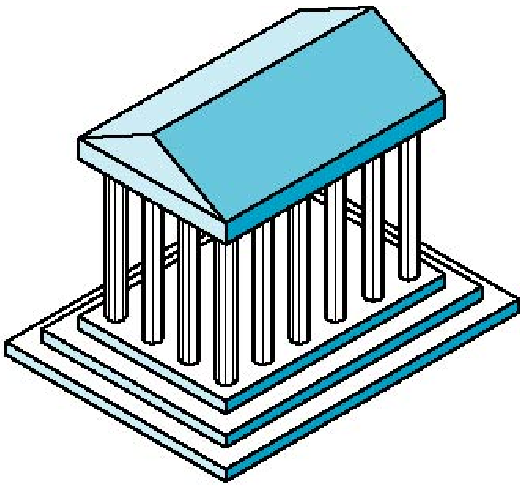




Dimetric



Trimetric



Isometric

Axonometric Projections

Used in CAD applications

Can see three principal faces of a box-like object

Lines are scaled (***foreshortened***) but can find scaling factors

- Isometric view has 1 scaling factor for all directions and allows distance measurements
- Dimetric has 2 scaling factors
- Trimetric has 3 scaling factors

Lines preserved but angles are not

- Projection of a circle in general is an ellipse

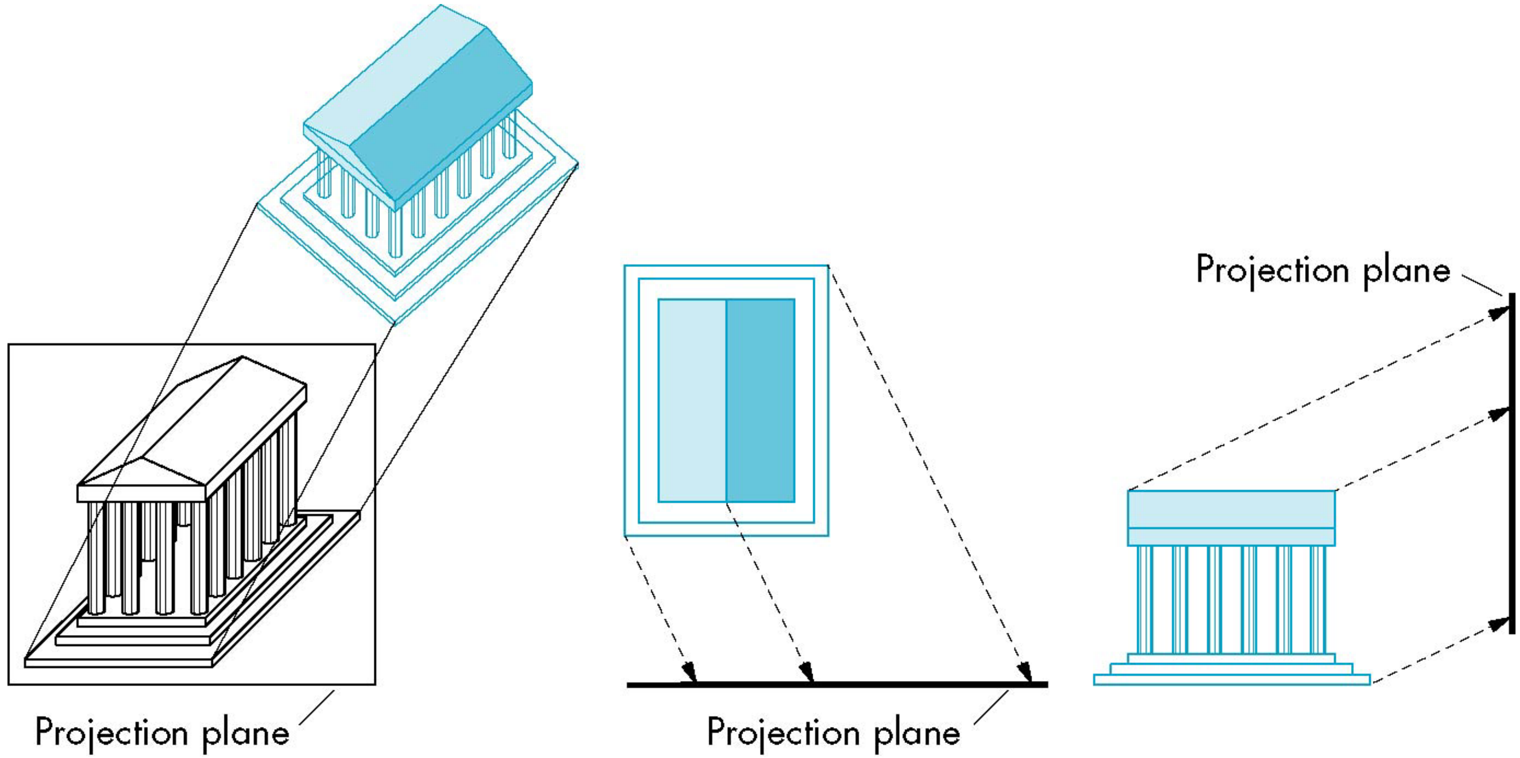
Some optical illusions possible

- Parallel lines appear to diverge

Oblique Projection

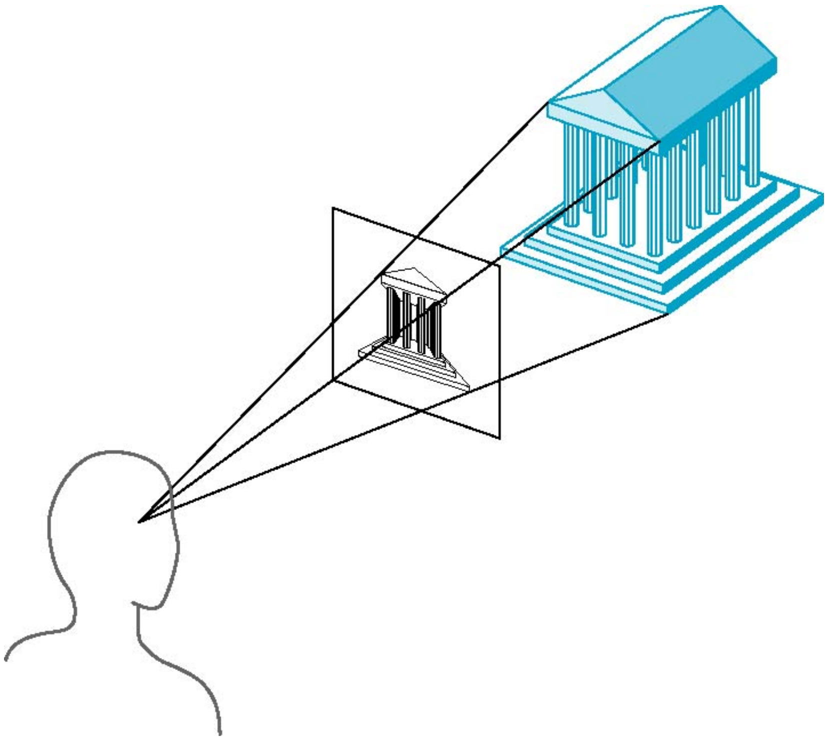
General parallel views

Arbitrary relationship between projectors and projection plane



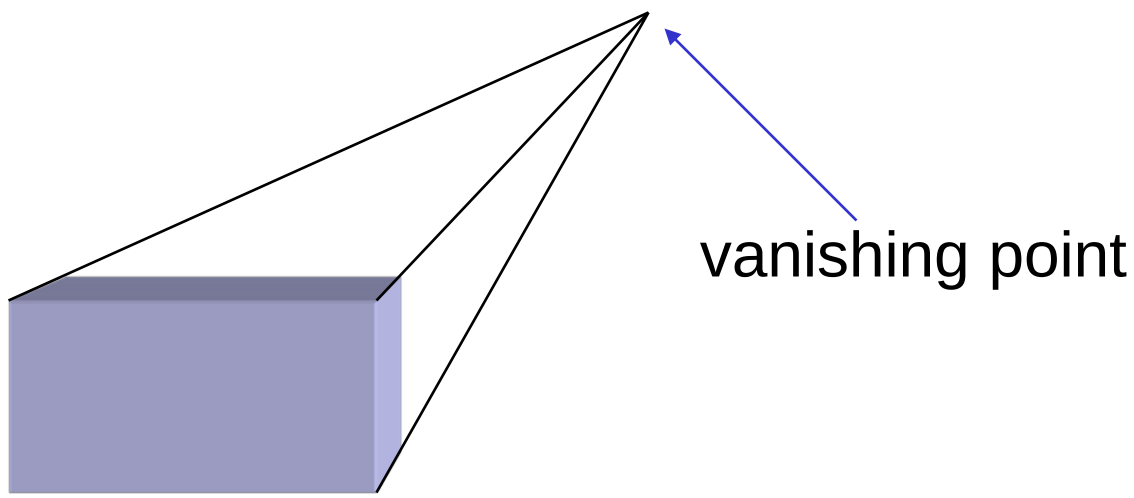
Perspective Projection

Projectors coverge at center of projection



Vanishing Points

Parallel lines (not parallel to the projection plane) on the object converge at a single point in the projection (the *vanishing point*)



Perspective Projection

(a) (b) (c)

FIGURE 4.10 Classical perspective views. (a) Three-point. (b) Two-point. (c) One-point.

Perspective Projection

Objects further from viewer are projected smaller than the same sized objects closer to the viewer (*diminution*)

- Looks realistic

Equal distances along a line may be not projected into equal distances (*nonuniform foreshortening*)

Angles are preserved only in planes parallel to the projection plane

More difficult to construct by hand than parallel projections (but not more difficult by computer)