COSC 414/519I: Computer Graphics

2023W2

Shan Du

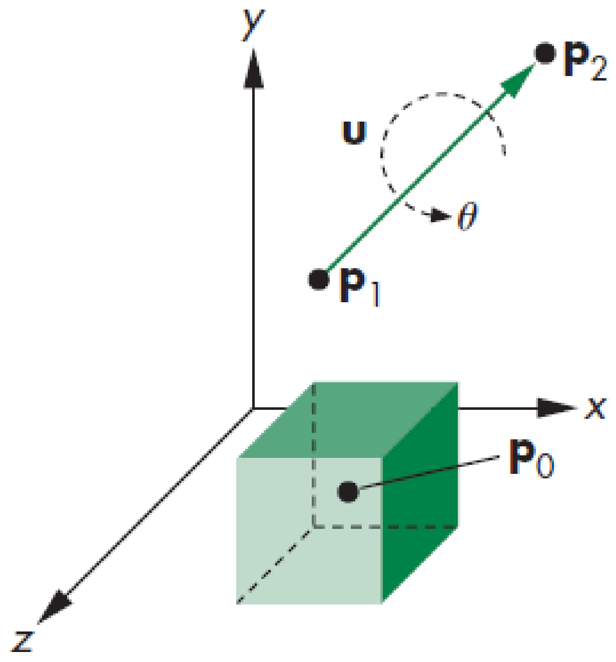
Rotation About an Arbitrary Axis

- How can we use direction angles to specify orientation?
- Define the vector using two points p$_1$ and p$_2$,

$$u = p_2 - p_1$$

- Replace *u* with a unit-length vector

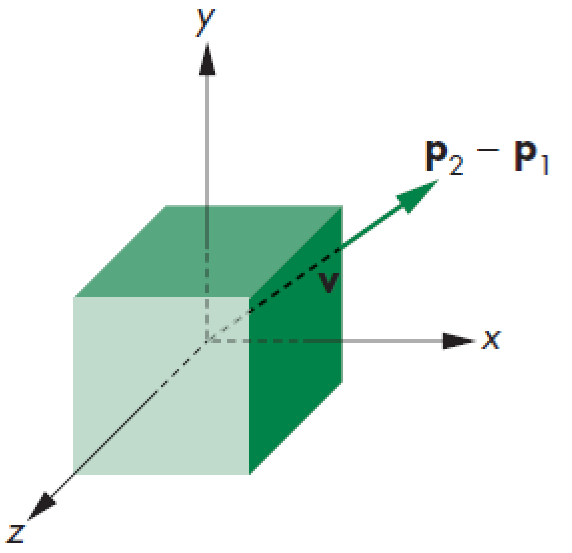
$$v = \frac{u}{|u|} = \begin{bmatrix} \alpha_x \\ \alpha_y \\ \alpha_z \end{bmatrix}$$



Rotation of a cube about an arbitrary axis.

Rotation About an Arbitrary Axis

- Move the fixed point to the origin.
- We can get arbitrary rotation from three rotations about the individual axes.
- But what are the individual angles?



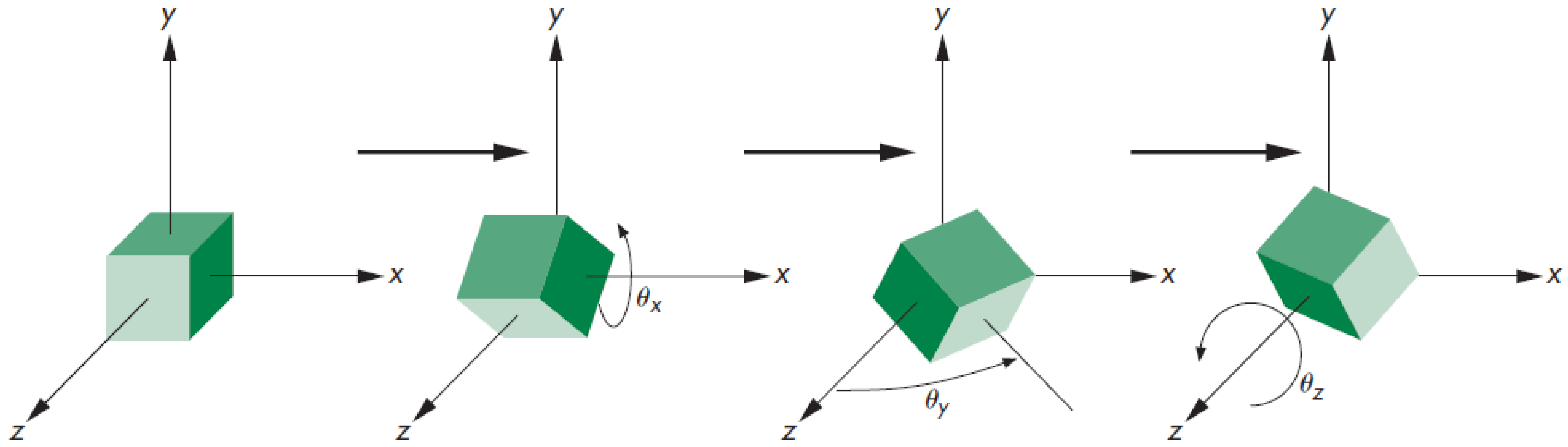
Movement of the fixed point to the origin.

Rotation About an Arbitrary Axis

- Strategy:
  - Carry out two rotations to align the axis of the rotation, with the *z*-axis.
  - Then rotate by θ about the *z*-axis.
  - Undo the two rotations that did the aligning.
  - The final rotation matrix is

$$R = R_x(-\theta_x)R_y(-\theta_y)R_z(\theta)R_y(\theta_y)R_x(\theta_x)$$

Rotation About an Arbitrary Axis

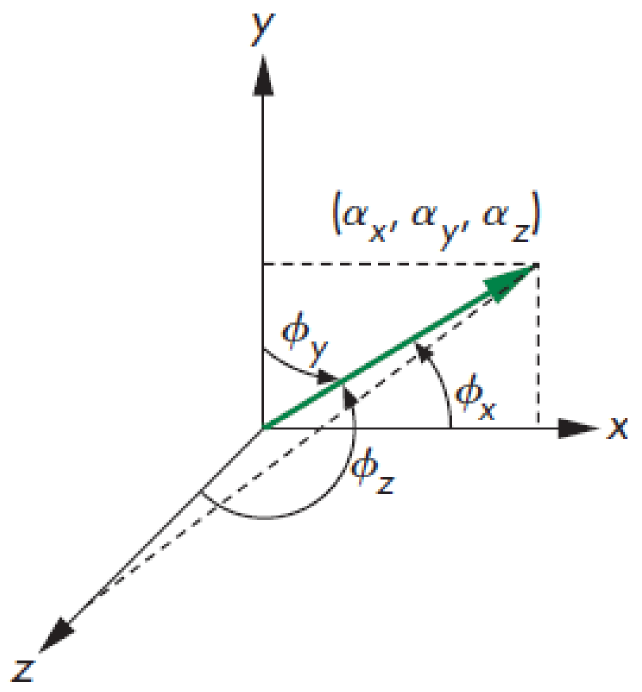


Sequence of rotations.

How to determine θ_x and θ_y?

Rotation About an Arbitrary Axis

- Looking at the components of *v*, since *v* is a unit-length vector, $\alpha_x^2 + \alpha_y^2 + \alpha_z^2 = 1$
- Draw the perpendiculars from point $(\alpha_x, \alpha_y, \alpha_z)$ to the coordinate axes.
- The three direction angles ϕ_x, ϕ_y, ϕ_z are the angles between *v* and the axes.



Direction angles.

Rotation About an Arbitrary Axis

- The direction cosines are given by

$$\cos \phi_x = \alpha_x,$$

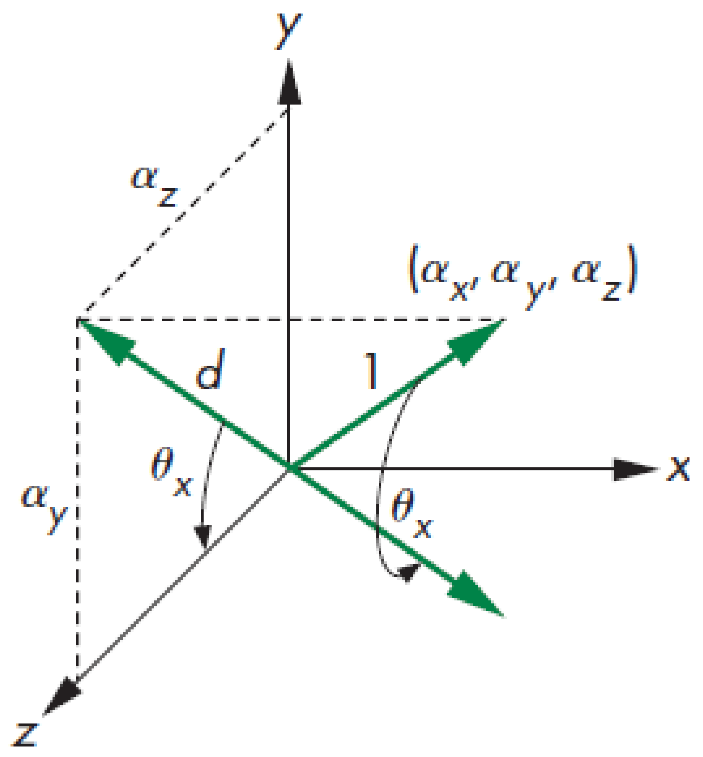
$$\cos \phi_y = \alpha_y,$$

$$\cos \phi_z = \alpha_z.$$

- Only two of the direction angles are independent, because

$$\cos^2 \phi_x + \cos^2 \phi_y + \cos^2 \phi_z = 1.$$

Rotation About an Arbitrary Axis

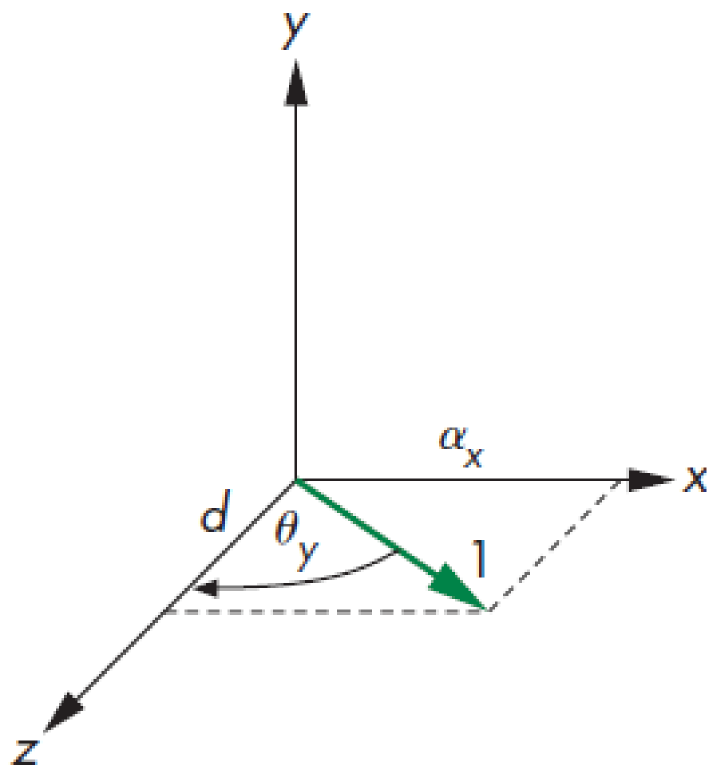


Computation of the x rotation.

$$R_x(\theta_x) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \alpha_z/d & -\alpha_y/d & 0 \\ 0 & \alpha_y/d & \alpha_z/d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

$$d = \sqrt{\alpha_y^2 + \alpha_z^2}$$

Rotation About an Arbitrary Axis



Computation of the *y* rotation.

$$\mathbf{R}_y(\theta_y) = \begin{bmatrix} d & 0 & -\alpha_x & 0 \\ 0 & 1 & 0 & 0 \\ \alpha_x & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Rotation About an Arbitrary Axis

- Finally, we concatenate all the matrices to find

$$\mathbf{M} = \mathbf{T}(\mathbf{p}_0)\mathbf{R}_x(-\theta_x)\mathbf{R}_y(-\theta_y)\mathbf{R}_z(\theta)\mathbf{R}_y(\theta_y)\mathbf{R}_x(\theta_x)\mathbf{T}(-\mathbf{p}_0)$$

Transformation Matrices in WebGL

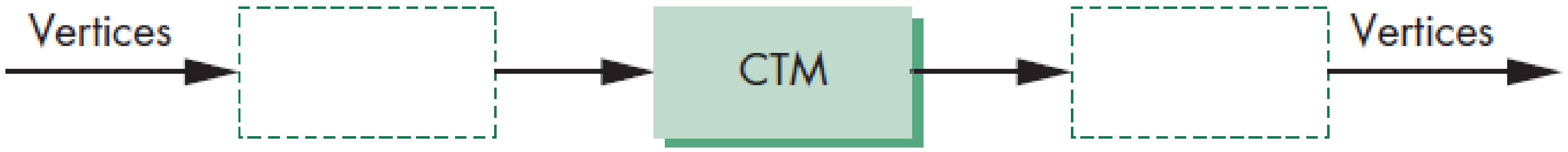
- The two transformations that we will use most often are the model-view transformation and the projection transformation.
- The model-view transformation brings representations of geometric objects from the application or object frame to the camera frame.
- The projection transformation will both carry out the desired projection and change the representation to clip coordinates.

Transformation Matrices in WebGL

- The model-view matrix normally is an affine transformation matrix and has only 12 degrees of freedom.
- The projection matrix is also a 4×4 matrix but is not affine.

Transformation Matrices in WebGL

- Current Transformation Matrices



Current transformation matrix (CTM).

  - If p is a vertex specified in the application, then the pipeline produces Cp.
- The application programmer can decide which frames to use and where to carry out the transformations between frames.

Transformation Matrices in WebGL

- Current Transformation Matrices
  - We can use a simple set of functions to form and manipulate 4×4 affine transformation matrices.
  - Let *C* denote the CTM (or any other matrix).
  - Initially, we set it to the 4×4 identity matrix; it can be reinitialized as needed.
  - We use the symbol ← to denote replacement.

Transformation Matrices in WebGL

- Current Transformation Matrices
  - The initialization operation is $C \leftarrow I$.
  - The functions that alter *C* are of three forms: those that load it with some matrix and those that modify it by pre-multiplication or post-multiplication by a matrix.
  - The three transformations supported in most systems are translation, scaling with a fixed point of the origin, and rotation with a fixed point of the origin.

Transformation Matrices in WebGL

- Current Transformation Matrices
  - We can write these operations in post-multiplication form as $C \leftarrow CT, C \leftarrow CS, C \leftarrow CR$ and in load form $C \leftarrow T, C \leftarrow S, C \leftarrow R$.
  - Most systems allow us to load the CTM with an arbitrary matrix M, $C \leftarrow M$, or to post-multiply by an arbitrary matrix M, $C \leftarrow CM$.

Transformation Matrices in WebGL

- Basic Matrix Functions
  - Create an identity matrix by

```
var a = mat4();
```

  - Or fill it with components by

```
var a = mat4(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15);
```

  - Or by vectors

```
var a = mat4(
  vec4(0, 1, 2, 3),
  vec4(4, 5, 6, 7),
  vec4(8, 9, 10, 11),
  vec4(12, 13, 14, 15)
);
```

Transformation Matrices in WebGL

- Basic Matrix Functions
  - Copy an existing matrix

```
b = mat4(a);
```

  - Find the determinant, inverse, and transpose of a matrix

```
var det = determinant(a);
b = inverse(a);   // 'b' becomes inverse of 'a'
b = transpose(a); // 'b' becomes transpose of 'a'
```

Transformation Matrices in WebGL

- Basic Matrix Functions
  - Multiply two matrices by

```
c = mult(a b); // c = a * b
```

  - We can multiply vector by a matrix

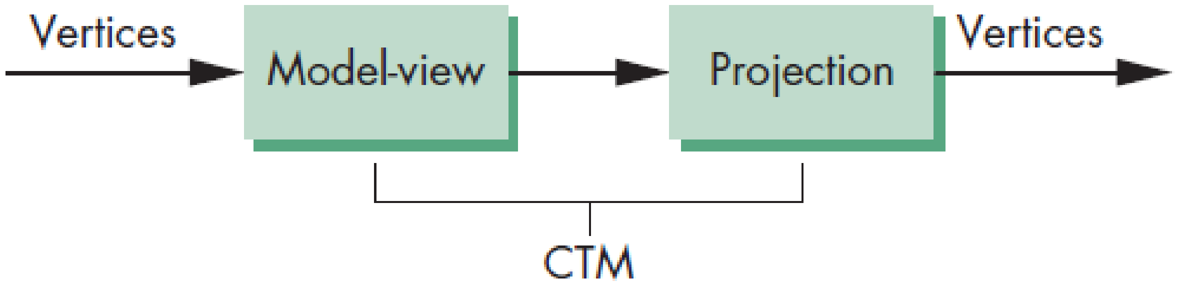
```
e = mult(a, d);
```

  - We can also reference or change individual elements using standard indexing

```
a[1][2] = 0;
var d = vec4(a[2]);
```

Transformation Matrices in WebGL

- Rotation, Translation, and Scaling
  - The matrix that is most often applied to all vertices is the product of the model-view matrix and the projection matrix.
  - We can think of the CTM as the product of these matrices.



Model-view and projection matrices.

Transformation Matrices in WebGL

- Rotation, Translation, and Scaling
  - We can form affine transformation for rotation, translation, and scaling using the following six functions:

```
var a = rotate(angle, direction);
var b = rotateX(angle);
var c = rotateY(angle);
var d = rotateZ(angle);
var e = scale(scaleVector);
var f = translate(translateVector);
```

Transformation Matrices in WebGL

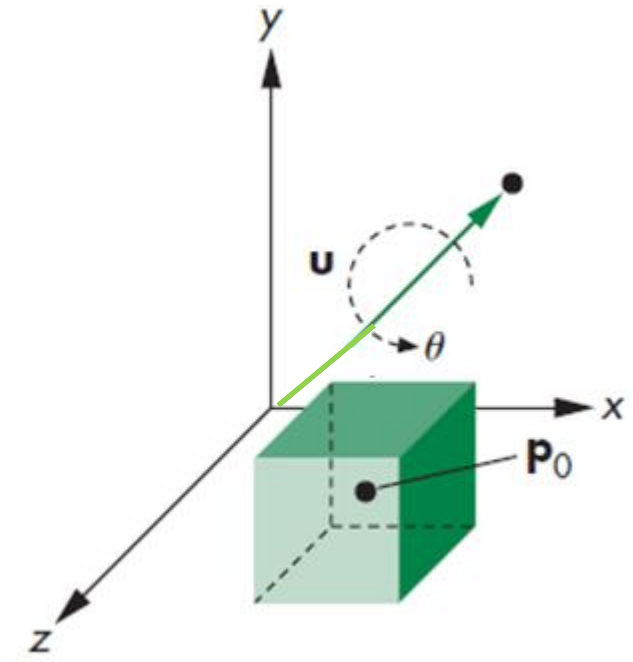
- Rotation, Translation, and Scaling
  - For rotation, the angles are specified in degrees and the rotations are around a fixed point at the origin.
  - In the translation function, the variables are the components of the displacement vector.
  - For scaling, the variables determine the scale factors along the coordinate axes and the fixed point is the origin.

Transformation Matrices in WebGL

- Rotation About a Fixed Point
  - We can perform a rotation about a fixed point, other than the origin, by first moving the fixed point to the origin, then rotating about the origin, and finally moving the fixed point back to its original location.

Transformation Matrices in WebGL

- Rotation About an Arbitrary Axis
  - Form the matrix for a 45-degree rotation about the line through the origin and the point (1,2,3) with a fixed point of (4,5,6)



Rotation of a cube about an arbitrary axis.

$$M = T(p_0)R_x(-\theta_x)R_y(-\theta_y)R_z(\theta)R_y(\theta_y)R_x(\theta_x)T(-p_0)$$

Transformation Matrices in WebGL

- Rotation About an Arbitrary Axis

$$\mathbf{M} = \mathbf{T}(\mathbf{p}_0)\mathbf{R}_x(-\theta_x)\mathbf{R}_y(-\theta_y)\mathbf{R}_z(\theta)\mathbf{R}_y(\theta_y)\mathbf{R}_x(\theta_x)\mathbf{T}(-\mathbf{p}_0)$$

```
var R = mat4();
var ctm = mat4();
var thetaX = Math.acos(3.0/Math.sqrt(14.0));
var thetaY = Math. acos(sqrt(13.0/14.0));
var d = vec3(4.0, 5.0, 6.0);
R = mult(R, rotateX(thetaX));
R = mult(R, rotateY(thetaY));
R = mult(R, rotateZ(45.0));
R = mult(R, rotateY(-thetaY));
R = mult(R, rotateX(-thetaX));
ctm = translate(ctm, d);
ctm = mult(ctm, R);
ctm = translate(ctm, negate(d));
```

Transformation Matrices in WebGL

- Order of Transformations
  - The transformation specified last is the one applied first.
  - The sequences of operations that we specified was

$$\mathbf{C} \leftarrow \mathbf{I}$$

$$\mathbf{C} \leftarrow \mathbf{CT}(4.0, 5.0, 6.0)$$

$$\mathbf{C} \leftarrow \mathbf{CR}(45.0, 1.0, 2.0, 3.0)$$

$$\mathbf{C} \leftarrow \mathbf{CT}(-4.0, -5.0, -6.0).$$

Transformation Matrices in WebGL

- Order of Transformations
  - In each step, we post-multiply at the end of the existing CTM, forming the matrix

$$\mathbf{C} = \mathbf{T}(4.0, 5.0, 6.0)\,\mathbf{R}(45.0, 1.0, 2.0, 3.0)\,\mathbf{T}(-4.0, -5.0, -6.0);$$

Spinning of The Cube

- There are three fundamentally different ways of doing the updates to the display.
  1. Form a new model-view matrix in the application and apply it to the vertex data to get new vertex positions and then send the new data to the GPU.
  2. Compute a new model-view matrix for each rotation and send it to the vertex shader and apply it there.
  3. We send only the angles to the vertex shader and recompute the model-view matrix there.

Spinning of The Cube

- In the first two strategies, we compute a model-view matrix in the application code

```
modelViewMatrix = mat4();
modelViewMatrix = mult(modelViewMatrix, rotateX(thetaArray[xAxis]));
modelViewMatrix = mult(modelViewMatrix, rotateY(thetaArray[yAxis]));
modelViewMatrix = mult(modelViewMatrix, rotateZ(thetaArray[zAxis]));
```

- In the first approach, we use this matrix to change the positions of all points, and then resend the points to the GPU.
- In the second approach, we send this matrix to the vertex shader and apply this model-view matrix in the vertex shader.

Spinning of The Cube

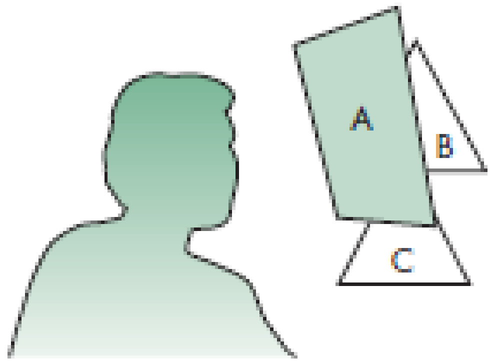
- The third approach is to send only the rotation angles and let the vertex shader compute the model-view matrix.

Hidden-Surface Removal

- The program draws triangles in the order that they are specified in the program.
- This order is determined by the recursion in the program, not the geometric relationships among the triangles.
- Each triangle is drawn (filled) in a solid color and is drawn over those triangles that have been rendered to the display.

Hidden-Surface Removal

- The hidden surface problem:



- Algorithms for ordering objects so that they are drawn correctly are called visible-surface algorithms or hidden-surface-removal algorithms.

Hidden-Surface Removal

- Z-buffer algorithm:

```
gl.enable(gl.DEPTH_TEST);
```

```
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```