# Computer Graphics

## (Affine Transformations: Mathematical Basics)

Thilo Kielmann
Fall 2003
Vrije Universiteit, Amsterdam
kielmann@cs.vu.nl

http://www.cs.vu.nl/~graphics/





## Motivation: Affine Transformations

- Transformations:
  - ⋆ rotation, scaling, translation
  - ⋆ projection
  - ⋆ concatenation (composition)
- Affine = line preserving
  - ⋆ line → line
  - ⋆ polygon → polygon



## Motivation: use of Matrices etc.

```
void myinit(void)
{
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glColor3f(1.0, 0.0, 0.0);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, 500.0, 0.0, 500.0);
    glMatrixMode(GL_MODELVIEW);
}
```

## Outline for today

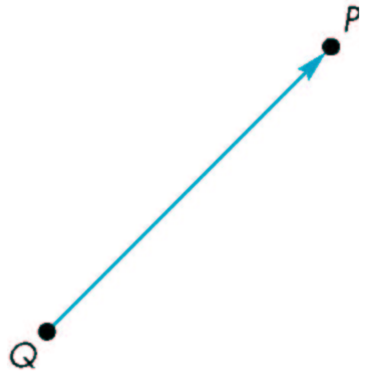
- Scalars, points, and vectors
- Coordinate systems and frames
- Modeling a colored cube
  - ⋆ OpenGL's Vertex arrays

# Scalars, Points, Vectors

- Geometric objects: points, polygons, polyhedra
- Geometric primitives: scalars, points, vectors
- Treatment (views):
  - ⋆ geometric
  - ⋆ mathematical
  - ⋆ computer science



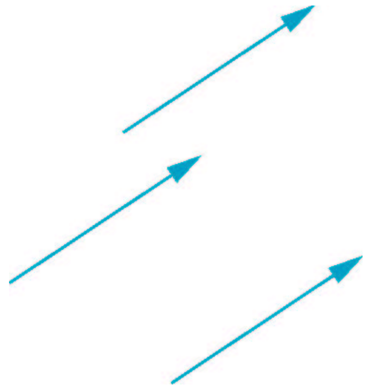
# Geometric View




- no coordinate system (that's just representation)
- directed line segments (between points), $\hat{=}$ vectors



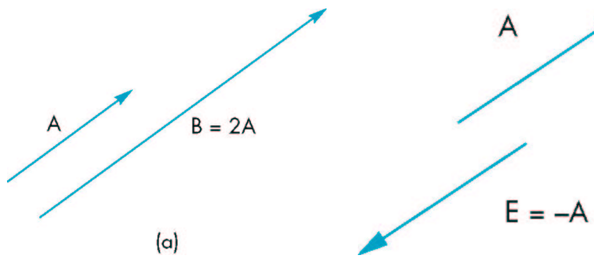
# Vector: Direction and Magnitude



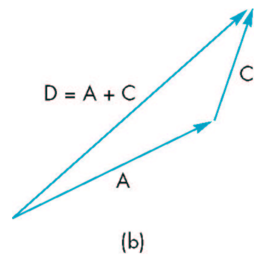
identical vectors



# Combination of Vectors

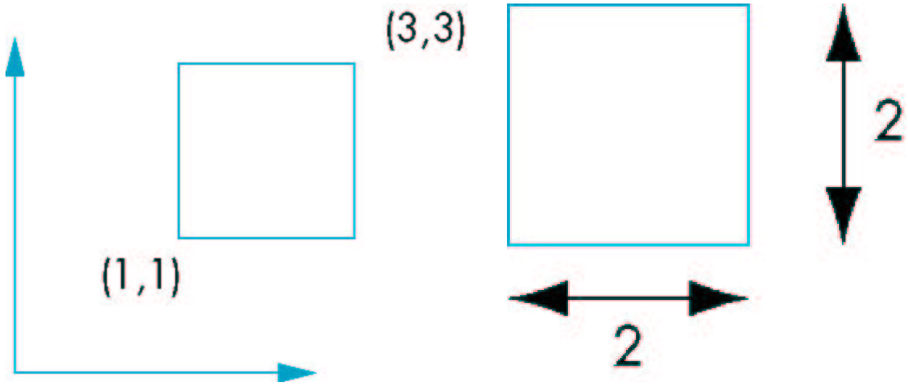



with scalars




head-to-tail rule

# Coordinate-free Geometry




Points and vectors exist without coordinates.
Coordinates just simplify referencing them.



# Mathematical View

Vector space:

- Entities: vectors and scalars
- Operations:

$$\begin{array}{cccc} \text{scalar} & + & \text{scalar} & \longrightarrow & \text{scalar} \\ \text{scalar} & \times & \text{scalar} & \longrightarrow & \text{scalar} \\ \text{scalar} & \times & \text{vector} & \longrightarrow & \text{vector} \\ \text{vector} & + & \text{vector} & \longrightarrow & \text{vector} \end{array}$$



# Mathematical View

Affine space:

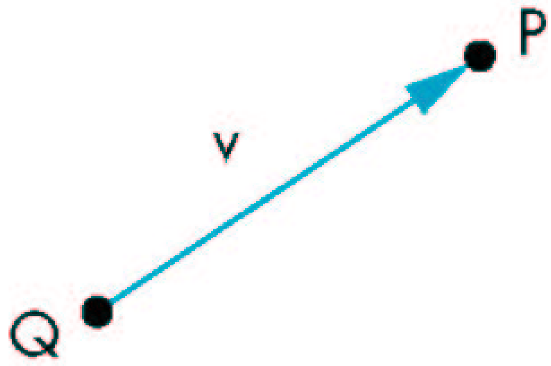
- Entities: vectors, scalars, **and points**
- Operations:

$$\begin{array}{ccccc} \text{scalar} & + & \text{scalar} & \longrightarrow & \text{scalar} \\ \text{scalar} & \times & \text{scalar} & \longrightarrow & \text{scalar} \\ \text{scalar} & \times & \text{vector} & \longrightarrow & \text{vector} \\ \text{vector} & + & \text{vector} & \longrightarrow & \text{vector} \\ \textbf{point} & + & \textbf{vector} & \longrightarrow & \textbf{point} \\ \textbf{point} & - & \textbf{point} & \longrightarrow & \textbf{vector} \end{array}$$

Euclidian space: add a measure for distance



# Vector/Point Operations




$P = Q + v$ or: $v = P - Q$

# Computer Science View

- abstract data types
- separate interface from implementation
- example: OpenGL internally represents points etc. in a four-dimensional system
- separate geometric/mathematical properties from representation (e.g. in a coordinate system)



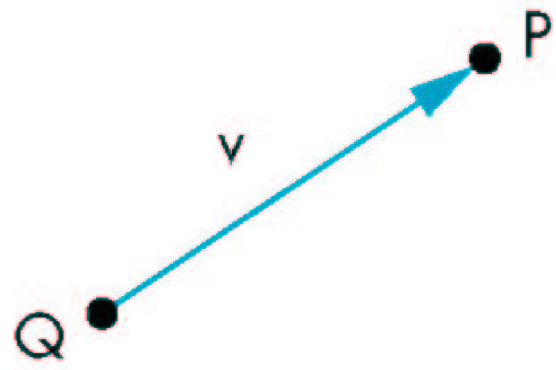
# Notation

- scalars: $\alpha, \beta, \gamma, \ldots$
- points: $P, Q, R, \ldots$
- vectors: $u, v, w, \ldots$
- **magnitude** of a vector: $|v|$

Vector-scalar multiplication: $|\alpha v| = |\alpha||v|$



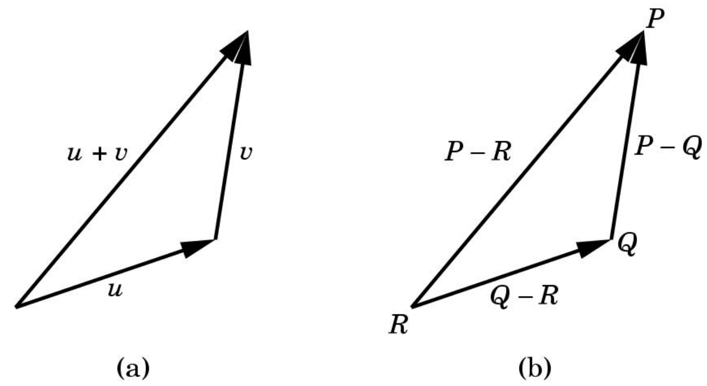
# Vector-Point Addition




Vector-point addition: $P = Q + v$ or: $v = P - Q$

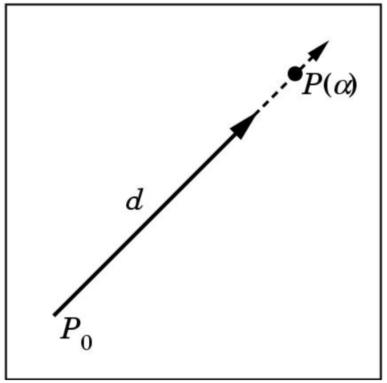


# Vector-Vector Addition




$(P - Q) + (Q - R) = P - R$

# Parametric Form for Lines

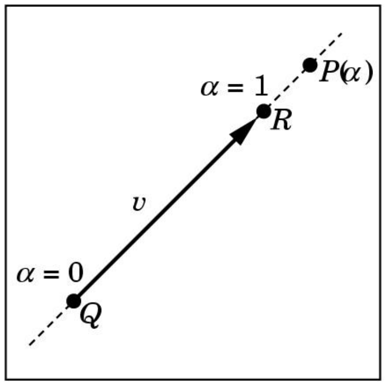



$$P(\alpha) = P_0 + \alpha d$$



# Affine Sums

Affine spaces do **not** have point/point addition or scalar/point multiplication, but:

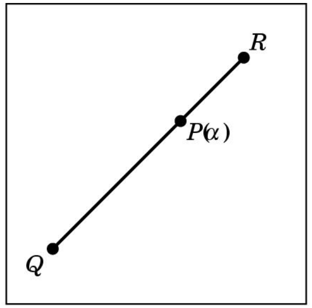



$$\begin{aligned} P &= Q + \alpha v \\ v &= R - Q \\ P &= Q + \alpha(R - Q) \\ &= \alpha R + (1 - \alpha) Q \\ P &= \alpha_1 R + \alpha_2 Q \\ & \quad\ \alpha_1 + \alpha_2 = 1 \\ & \quad\ \text{this looks as if...} \end{aligned}$$



# Convexity

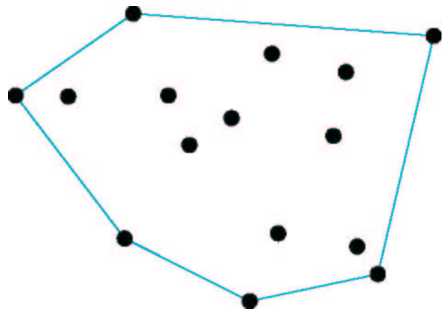
Any point on the line segment between any 2 points of a convex object is inside the object.




line segments are convex:
for $0 \leq \alpha \leq 1$, affine sum defines segment btw. $R$ and $Q$

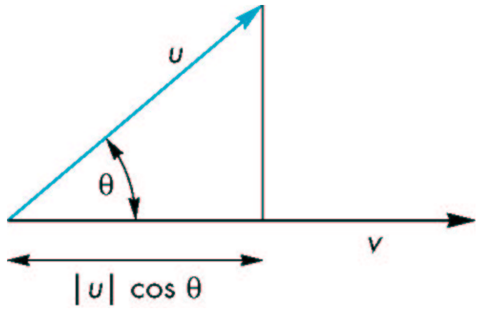


# Convex Hull



Set of points $P$, formed by the affine sums of $P_1 \ldots P_n$:
$P = \alpha_1 P_1 + \alpha_2 P_2 + \ldots + \alpha_n P_n$
$\alpha_1 + \alpha_2 + \ldots + \alpha_n = 1$
$\alpha_i \geq 0, i = 1, 2, \ldots, n$

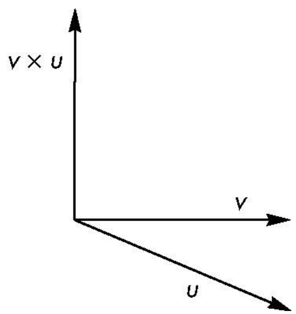
# Dot Product and Projection




Dot product: $u \cdot v$ $\quad u \cdot v = 0$ iff $u, v$ are orthogonal
Euclidian space: $|u|^2 = u \cdot u$
$\cos\,\theta = \frac{u \cdot v}{|u||v|}$
orthogonal projection of $u$ onto $v$: $|u|\cos\,\theta = u \cdot v / |v|$



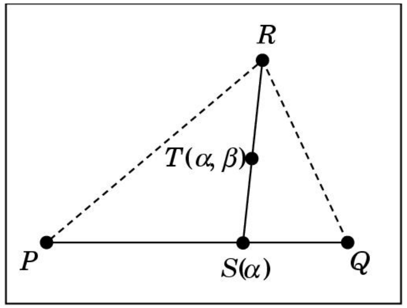
# Cross Product




$u, v$ non-parallel: $n = u \times v$ is orthogonal to $u, v$
$|\sin\,\theta| = \frac{|u \times v|}{|u||v|}$
(right-handed coordinate system)



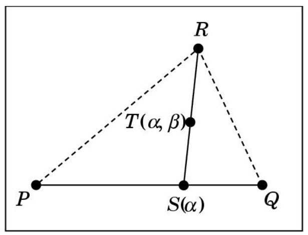
# Planes




$S(\alpha) = \alpha P + (1-\alpha)Q, \quad 0 \leq \alpha \leq 1$
$T(\beta) = \beta S + (1-\beta)R, \quad 0 \leq \beta \leq 1$
$T(\alpha, \beta) = \beta[\alpha P + (1-\alpha)Q] + (1-\beta)R$

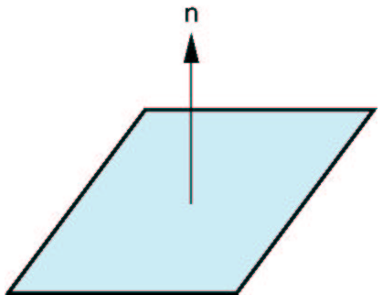


# Planes (2)




$$
\begin{aligned}
T(\alpha, \beta) &= \beta[\alpha P + (1-\alpha)Q] + (1-\beta)R \\
T(\alpha, \beta) &= P + \beta(1-\alpha)(Q-P) + (1-\beta)(R-P) \\
T(\alpha, \beta) &= P_0 + \alpha' u + \beta' v \\
(P - P_0) &= \alpha' u + \beta' v \quad \text{iff P lies in the plane} \\
n \cdot (P - P_0) &= 0 \quad n \text{ is the } \textbf{normal} \text{ to the plane}
\end{aligned}
$$

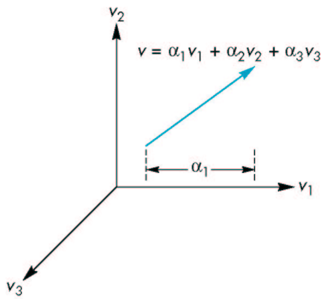
# Planes (3)




$P_0, u, v$ define a plane.
$n = u \times v$ is the **normal** (vector) to the plane.



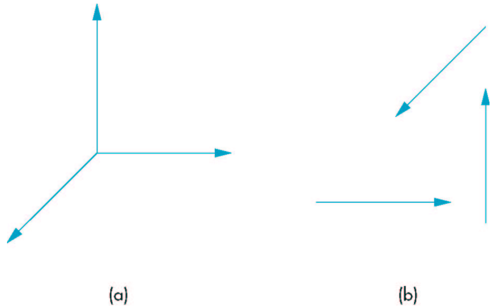
# Coordinate Systems




$v = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$
$\alpha_1, \alpha_2, \alpha_3$ are **components** of $v$ w.r.t. **basis** $v_1, v_2, v_3$.
**representation** $a = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix}$ $\qquad v = a^T \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$



# Frame: Basis Vectors + Reference Point




Vector: $v = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$
Point: $P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3$



# Changes of Coordinate Systems

(All transformations like scaling, rotation, etc. are in fact changes of coordinate systems.)

$\{v_1, v_2, v_3\}$ and $\{u_1, u_2, u_3\}$ are bases

$$\begin{aligned} u_1 &= \gamma_{1,1} v_1 + \gamma_{1,2} v_2 + \gamma_{1,3} v_3 \\ u_2 &= \gamma_{2,1} v_1 + \gamma_{2,2} v_2 + \gamma_{2,3} v_3 \\ u_3 &= \gamma_{3,1} v_1 + \gamma_{3,2} v_2 + \gamma_{3,3} v_3 \end{aligned}$$

$$M = \begin{bmatrix} \gamma_{1,1} & \gamma_{1,2} & \gamma_{1,3} \\ \gamma_{2,1} & \gamma_{2,2} & \gamma_{2,3} \\ \gamma_{3,1} & \gamma_{3,2} & \gamma_{3,3} \end{bmatrix}$$

# Changes of Coordinate Systems(2)

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = M \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

$$w = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = a^T \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = b^T \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}$$

$$w = b^T \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = b^T M \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = a^T \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

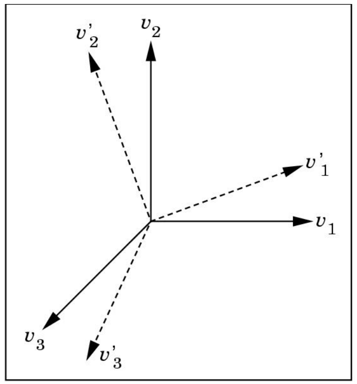
Thus: $M$ translates between coordinate systems!

$$a = M^T b$$

$$b = Aa = (M^T)^{-1} a$$



# Rotation and Scaling (of a Basis)




Applying a matrix $M$ allows us to rotate and scale a coordinate system.



# Example: Change of Representation

Unit basis: $v_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad v_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad v_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

Vector: $a = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \qquad w = v_1 + 2v_2 + 3v_3$

New basis: $\begin{aligned} u_1 &= v_1 \\ u_2 &= v_1 + v_2 \\ u_3 &= v_1 + v_2 + v_3 \end{aligned}$

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$



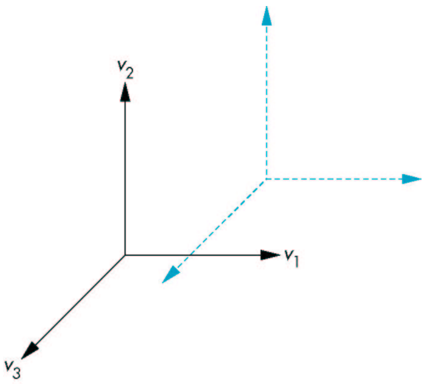
# . . . and now translate

$$A = (M^T)^{-1} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$b = Aa = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ 3 \end{bmatrix}$$

$$w = -u_1 - u_2 + 3u_3$$

# Rotation/Scaling . . . but no Translation




A translation (change of frame — origin) can not be modeled by applying $M$.



# Problem: modeling points

Frame: $(v_1, v_2, v_3, P_0)$, point at $(x, y, z)$

First try: $p = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$

$P = P_0 + xv_1 + yv_2 + zv_3$

But: $w = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 \qquad w = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix}$

"a point is a vector from the origin"
mixing two concepts! :-(



# Homogeneous Coordinates

$P = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 + P_0$

Define point-scalar "multiplication":

$0 \cdot P = 0$
$1 \cdot P = P$

$$P = [\alpha_1\ \alpha_2\ \alpha_3\ 1] \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix} \qquad p = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ 1 \end{bmatrix}$$



# Homogeneous Representation of Vectors and Points

$$P = [\alpha_1\ \alpha_2\ \alpha_3\ 1] \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix} \qquad p = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ 1 \end{bmatrix}$$

$$w = [\delta_1\ \delta_2\ \delta_3\ 0] \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix} \qquad a = \begin{bmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \\ 0 \end{bmatrix}$$

# Matrix Representation

Two frames: $(v_1, v_2, v_3, P_0)$ and $(u_1, u_2, u_3, Q_0)$

$$\begin{aligned}
u_1 &= \gamma_{1,1}v_1 + \gamma_{1,2}v_2 + \gamma_{1,3}v_3 \\
u_2 &= \gamma_{2,1}v_1 + \gamma_{2,2}v_2 + \gamma_{2,3}v_3 \\
u_3 &= \gamma_{3,1}v_1 + \gamma_{3,2}v_2 + \gamma_{3,3}v_3 \\
Q_0 &= \gamma_{4,1}v_1 + \gamma_{4,2}v_2 + \gamma_{4,3}v_3 + P_0
\end{aligned}$$

$$M = \begin{bmatrix} \gamma_{1,1} & \gamma_{1,2} & \gamma_{1,3} & 0 \\ \gamma_{2,1} & \gamma_{2,2} & \gamma_{2,3} & 0 \\ \gamma_{3,1} & \gamma_{3,2} & \gamma_{3,3} & 0 \\ \gamma_{4,1} & \gamma_{4,2} & \gamma_{4,3} & 1 \end{bmatrix}$$



# Matrix Representation (2)

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ Q_0 \end{bmatrix} = M \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix}$$

And again:

$$a = M^T b \qquad b = Aa = (M^T)^{-1} a$$



# Example: Change of Representation

Old example: $$\begin{aligned} u_1 &= v_1 \\ u_2 &= v_1 + v_2 \\ u_3 &= v_1 + v_2 + v_3 \end{aligned}$$

and move $Q_0$ to $(1, 2, 3)$ w.r.t. $P_0$

$Q_0 = v_1 + 2v_2 + 3v_3 + P_0$

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 2 & 3 & 1 \end{bmatrix}$$



# Example (2)

$$A = (M^T)^{-1} = \begin{bmatrix} 1 & -1 & 0 & 1 \\ 0 & 1 & -1 & 1 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

translate the **point** $(1, 2, 3)$: $\quad q = Ap = A \begin{bmatrix} 1 \\ 2 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$

translate the **vector** $(1, 2, 3)$: $\quad b = Aa = A \begin{bmatrix} 1 \\ 2 \\ 3 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ 3 \\ 0 \end{bmatrix}$

# Summary: Frames and Homogeneous Coordinates

- Frames define coordinate systems (three vectors and a reference point)
- Homogeneous coordinates capture the reference point in the fourth dimension.
- 4-dimensional coordinates and matrices allow to deal with frames easily.

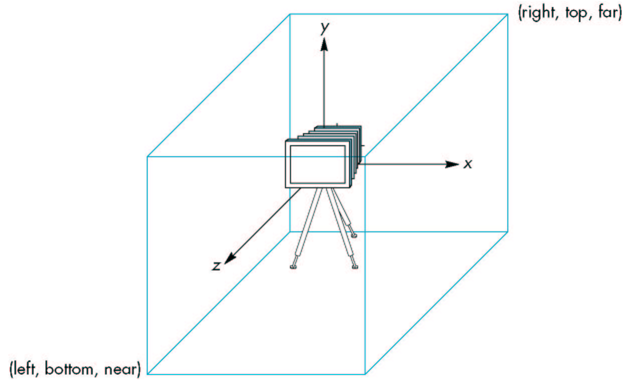


# Frames and Abstract Data Types

- What we want:
  - ⋆ 2-dimensional and 3-dimensional coordinates (for vertices)
- What we need (inside OpenGL):
  - ⋆ 4-dimensional coordinates
- The OpenGL API shields 4-dim from the programmer



# Frames in OpenGL




OpenGL uses a **world frame** and a **camera frame**.



# Frames in OpenGL

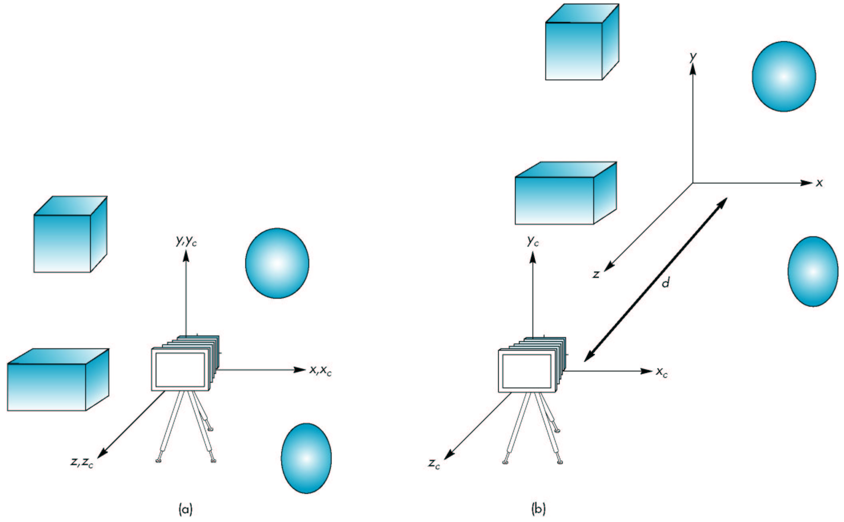
The **model-view matrix** converts world coordinates to camera coordinates.

Example: $$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Model-view matrix $A$ moves the point $(x, y, z)$ in the world frame to $(x, y, z - d)$ in the camera frame.

Interpretation: either moving the objects relative to the camera, or moving the camera relative to the objects.

# Applying the Model-view Matrix



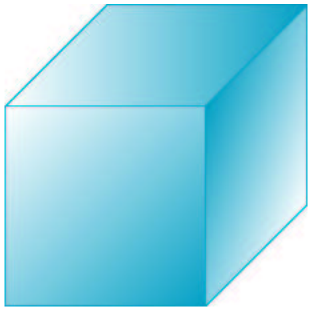



# Setting the Model-view Matrix

- Simple: call `glLoadMatrix` with a parameter array of 16 elements :-)
- Problem: how to compute the right matrix for interesting transformations?
- Solution: OpenGL has predefined operations that “do the right thing” depending on what the programmer really wants to do . . .



# Example: Modeling a Colored Cube



A cube with the colors of the color cube attached.
(see Lecture 2)



# Describe the Cube by its Vertices

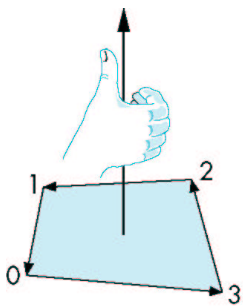
```
GLfloat vertices[8][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
        {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
        {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};

GLfloat colors[8][3] = {{0.0,0.0,0.0},{1.0,0.0,0.0},
        {1.0,1.0,0.0}, {0.0,1.0,0.0}, {0.0,0.0,1.0},
        {1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}};
```

# Draw one side of the Cube as a Polygon

```
glBegin(GL_POLYGON);
    glColor3fv(colors[0]);
    glVertex3fv(vertices[0]);
    glColor3fv(colors[3]);
    glVertex3fv(vertices[3]);
    glColor3fv(colors[2]);
    glVertex3fv(vertices[2]);
    glColor3fv(colors[1]);
    glVertex3fv(vertices[1]);
glEnd();
```



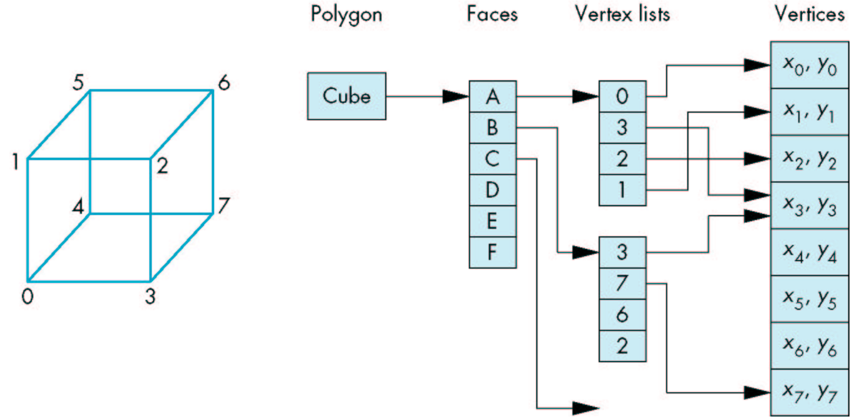
# Inward and Outward Pointing Faces




A face is **outward facing** if vertices are traversed counterclockwise. Also called **right-hand rule**.



# Vertex-list Representation of the Cube




Each vertex is stored exactly once.
The rest of the structure represents topology of the cube.

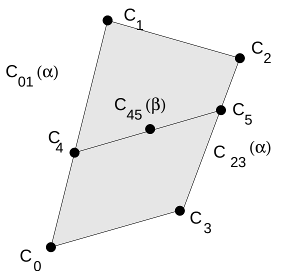


# Drawing the Cube

```
void quad(int a, int b, int c, int d){
glBegin(GL_QUADS);
  glColor3fv(colors[a]);
  glVertex3fv(vertices[a]);
  glColor3fv(colors[b]);
  glVertex3fv(vertices[b]);
  glColor3fv(colors[c]);
  glVertex3fv(vertices[c]);
  glColor3fv(colors[d]);
  glVertex3fv(vertices[d]);
glEnd();
}
```

```
void colorcube(void){
  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);
}
```

# Bilinear Interpolation (of Colors)




$$
\begin{aligned}
C_{01}(\alpha) &= (1-\alpha)C_0 + \alpha C_1 \\
C_{23}(\alpha) &= (1-\alpha)C_2 + \alpha C_3 \\
C_{45}(\beta) &= (1-\beta)C_4 + \beta C_5
\end{aligned}
$$

Interpolation for all three primary colors independently.



# Vertex-lists and Efficiency

Number of calls to OpenGL for drawing the cube once:

6 sides $\times$ (`glBegin` + 4 $\times$ color + 4 $\times$ vertex + `glEnd`) $= 60$ calls.

This comes with 60 times parameter checking, etc. . .

What is the problem?

We pass the vertices (and colors) again and again. . .



# Using Vertex Arrays Instead

E.g., in `myInit`:

```
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);
                // 3-dim float values, no gaps
glColorPointer(3, GL_FLOAT, 0, colors);
```

`colors` and `vertices` are the arrays we already know.



# Drawing with Vertex Arrays

We also need an index array:

```
GLubyte cubeIndices[]={
  0,3,2,1,  2,3,7,6,  0,4,7,3,
  1,2,6,5,  4,5,6,7,  0,1,5,4
};
```

And draw:

```
for (i=0; i<6; i++){
  glDrawElements(GL_POLYGON, 4, GL_UNSIGNED_BYTE,
                 &cubeIndices[4*i]);
}
```

# Drawing the Cube with a Single Call

```
for (i=0; i<6; i++){
  glDrawElements(GL_POLYGON, 4, GL_UNSIGNED_BYTE,
                 &cubeIndices[4*i]);
}


// or simply:

glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE,
               cubeIndices);
```



# Summary

- Scalars, points, and vectors
- Coordinate systems and frames (“weird” 4 dimensions)
- Vertex arrays
- Next week: affine transformations
  - ⋆ rotation, translation, scaling, shear
  - ⋆ “make the cube rotate”

 (show the cube)