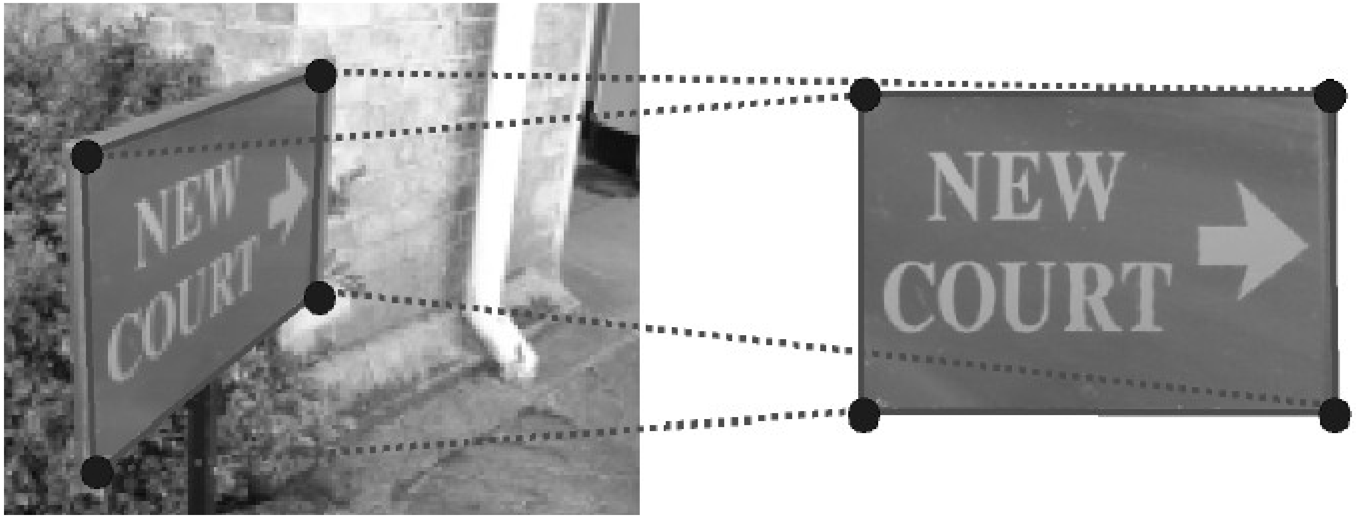
INF555



Fundamentals of 3D

Lecture 4:

Debriefing: ICP (kD-trees)
Homography
Graphics pipeline

Frank Nielsen
5 octobre 2011
nielsen@lix.polytechnique.fr

ICP: Iterative Closest Point Algorithm at a glance

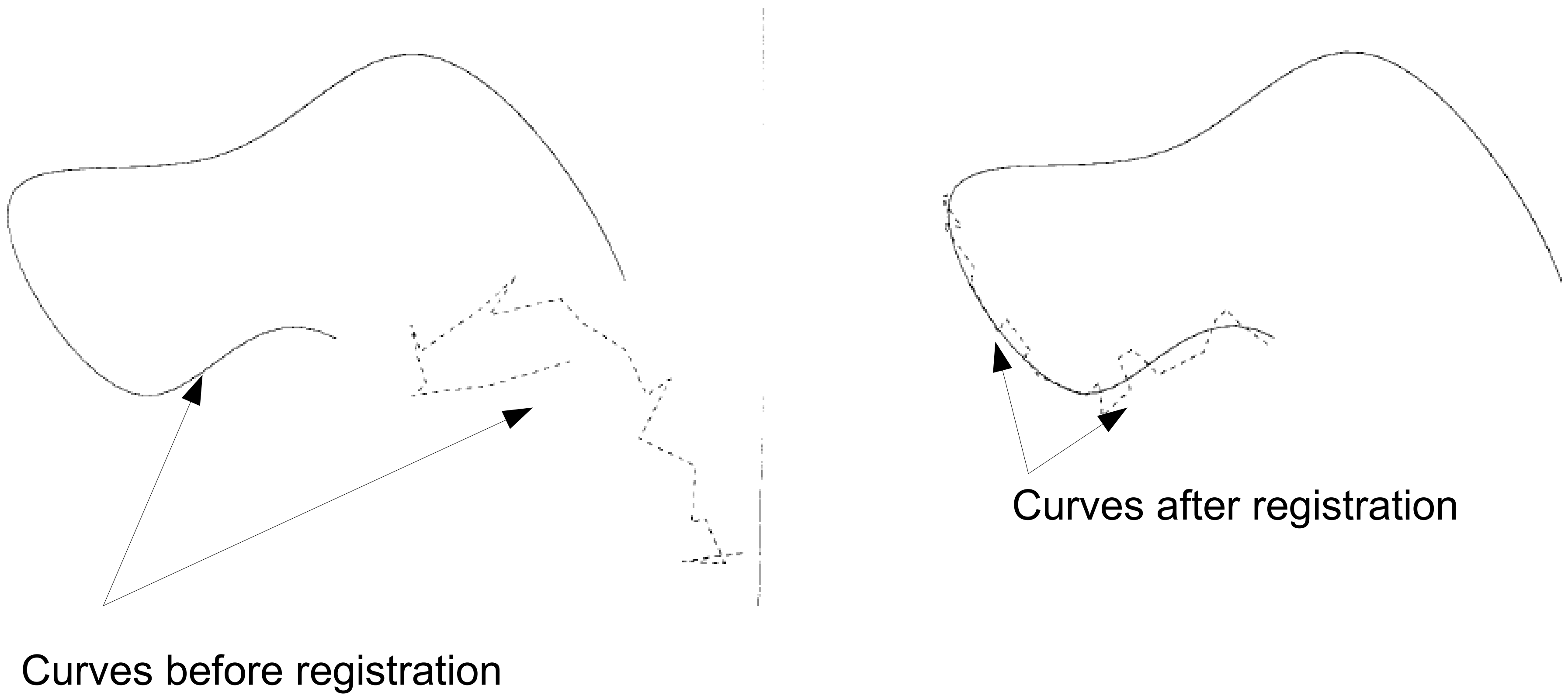
- Start from a not too far initial transformation

Do iterations until the mismatch error goes below a threshold:
- Match the point of the target to the source
- Compute the best transformation from point correspondence

In practice, this is a **very fast** registration method...

A Method for Registration of 3-D Shapes.Paul J *Besl*, Neil D Mckay.
IEEE Trans. Pattern Anal. Mach. Intell., Vol. 14, No. 2. (February 1992

ICP is a generic method
Example for curve registrations:



ICP: Finding the best rigid transformation

Given point correspondences, find the best rigid transformation.

$$X = \{x_1, ..., x_n\}$$

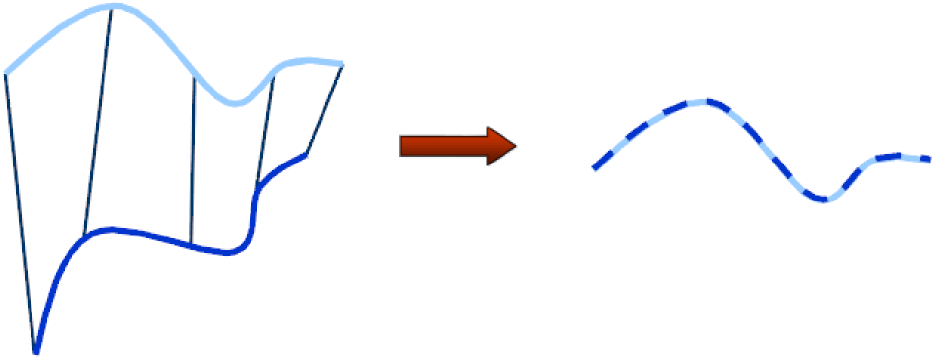
Observation/Target

$$P = \{p_1, ..., p_n\}$$

Source/Model

Find (R,t) that minimizes the <u>squared euclidean error</u>:

$$E(R, t) = \frac{1}{N_p} \sum_{i=1}^{N_p} ||x_i - Rp_i - t||^2$$



Align the center of mass of sets:

$$\mu_x = \frac{1}{N_x}\sum_{i=1}^{N_x} x_i \quad \text{and} \quad \mu_p = \frac{1}{N_p}\sum_{i=1}^{N_p} p_i$$

$$X = \{x_1, \ldots, x_n\}$$
$$P = \{p_1, \ldots, p_n\}$$

$$\Rightarrow$$

$$X' = \{x_i - \mu_x\} = \{x'_i\}$$
$$P' = \{p_i - \mu_p\} = \{p'_i\}$$

Finding the rotation matrix:

Cross-covariance matrix:

$$W = \sum_{i=1}^{N_p} x'_i p'^T_i$$

Compute the singular value decomposition

$$W = U \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{bmatrix} V^T$$

$$\sigma_1 \geq \sigma_2 \geq \sigma_3$$

Optimal transformation:

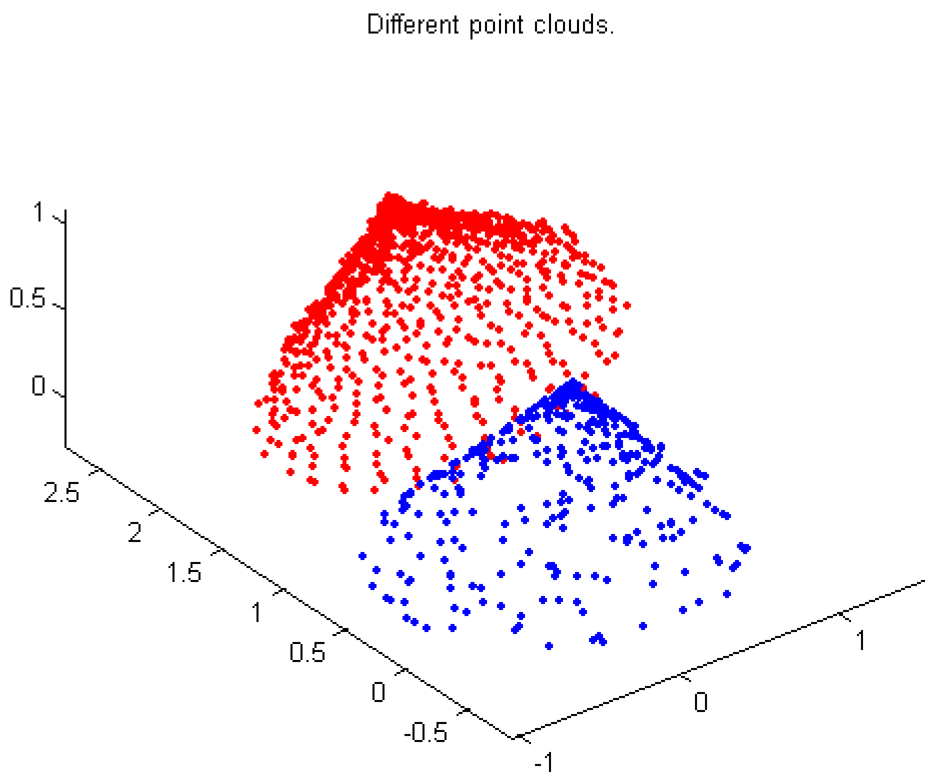
$$R = UV^T$$
$$t = \mu_x - R\mu_p$$

Recover the translation
once optimal rotation is found

ICP: Monotonicity and convergence

The average squared Euclidean distance decreases **<u>monotonously (→ convergence)</u>**

In fact (stronger result):
Each correspondence pair distance decreases



Drawback:
When does the local minimum is global?

Difficult to handle symmetry

(use texture, etc. to disambiguate)

Best 3D transformation (with quaternions)

With respect to least squares...

SVD take into account reflections...

$$\vec{q}_R = [q_0 q_1 q_2 q_3]^t \qquad \vec{q}_T = [q_4 q_5 q_6]^t \qquad \vec{q} = [\vec{q}_R | \vec{q}_T]^t$$

$$\boldsymbol{R} = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1 q_2 - q_0 q_3) & 2(q_1 q_3 + q_0 q_2) \\ 2(q_1 q_2 + q_0 q_3) & q_0^2 + q_2^2 - q_1^2 - q_3^2 & 2(q_2 q_3 - q_0 q_1) \\ 2(q_1 q_3 - q_0 q_2) & 2(q_2 q_3 + q_0 q_1) & q_0^2 + q_3^2 - q_1^2 - q_2^2 \end{bmatrix}$$

Minimize:

$$f(\vec{q}) = \frac{1}{N_p} \sum_{i=1}^{N_p} \|\vec{x}_i - \boldsymbol{R}(\vec{q}_R)\vec{p}_i - \vec{q}_T\|^2$$

Best 3D transformation (with quaternions)

$$\vec{\mu}_p = \frac{1}{N_p} \sum_{i=1}^{N_p} \vec{p}_i \quad \text{and} \quad \vec{\mu}_x = \frac{1}{N_x} \sum_{i=1}^{N_x} \vec{x}_i$$

Cross-covariance matrix:

$$\Sigma_{px} = \frac{1}{N_p} \sum_{i=1}^{N_p} [(\vec{p}_i - \vec{\mu}_p)(\vec{x}_i - \vec{\mu}_x)^t] = \frac{1}{N_p} \sum_{i=1}^{N_p} [\vec{p}_i \vec{x}_i^t] - \vec{\mu}_p \vec{\mu}_x^{\,t}.$$

$$A_{ij} = (\Sigma_{px} - \Sigma_{px}^T)_{ij}$$

Anti-symmetric matrix

$$\Delta = [A_{23} \quad A_{31} \quad A_{12}]^T$$

$$Q(\Sigma_{px}) = \begin{bmatrix} \operatorname{tr}(\Sigma_{px}) & \Delta^T \\ \Delta & \Sigma_{px} + \Sigma_{px}^T - \operatorname{tr}(\Sigma_{px}) \boldsymbol{I}_3 \end{bmatrix}$$

Best 3D transformation (with quaternions)

$$Q(\Sigma_{px}) = \begin{bmatrix} \mathrm{tr}(\Sigma_{px}) & \Delta^T \\ \Delta & \Sigma_{px} + \Sigma_{px}^T - \mathrm{tr}(\Sigma_{px}) \boldsymbol{I}_3 \end{bmatrix}$$

Take the unit eigenvector corresponding to the maximal eigenvalue:

$$\vec{q_R} = [q_0 \quad q_1 \quad q_2 \quad q_3]^t$$

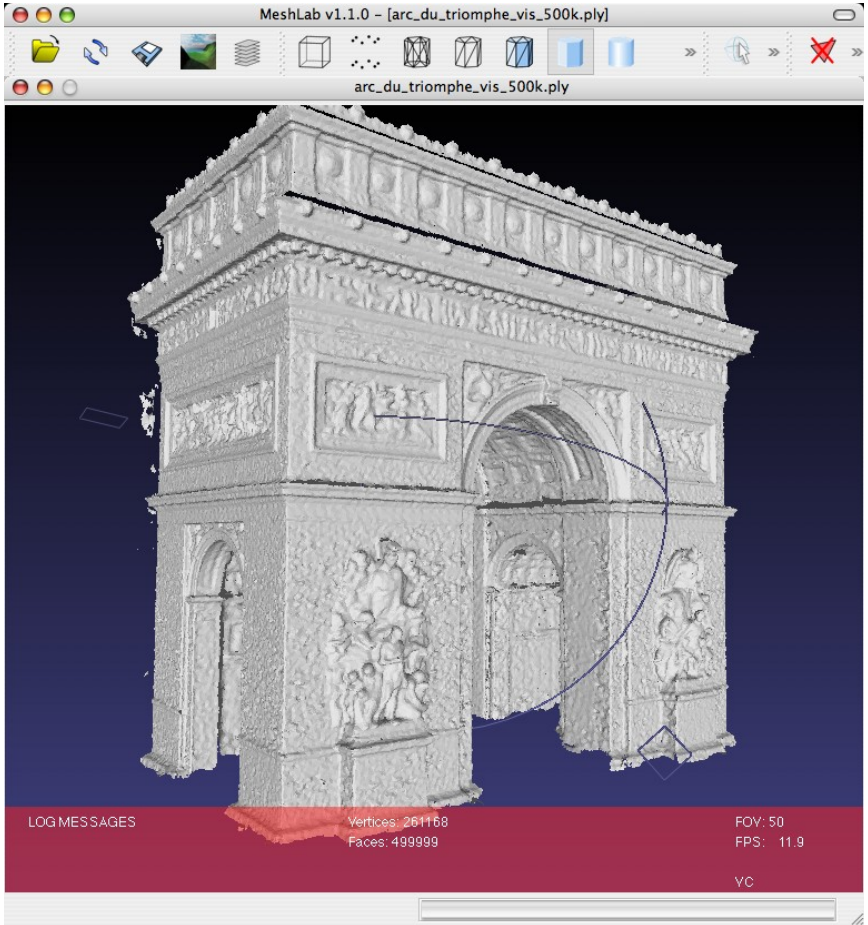
Get the remaining translation (easy) as:

$$\vec{q}_T = \vec{\mu}_x - \boldsymbol{R}(\vec{q}_R)\vec{\mu}_p.$$

Time complexity of ICP

Linear (fixed dimension) to find least square transformation
At each iteration, perform n nearest neighbor queries

Naive implementation: O(I*n*n) => **slow for large n**



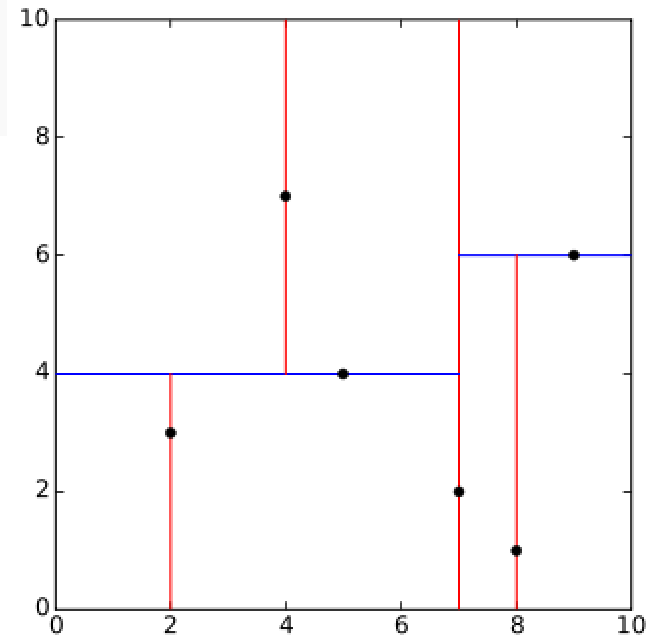
http://meshlab.sourceforge.net/

kD-trees for fast NN queries

```
function kdtree (list of points pointList, int depth)
{
    if pointList is empty
        return nil;
    else
    {
        // Select axis based on depth so that axis cycles through all valid values
        var int axis := depth mod k;

        // Sort point list and choose median as pivot element
        select median from pointList;

        // Create node and construct subtrees
        var tree_node node;
        node.location := median;
        node.leftChild := kdtree(points in pointList before median, depth+1);
        node.rightChild := kdtree(points in pointList after median, depth+1);
        return node;
    }
}
```

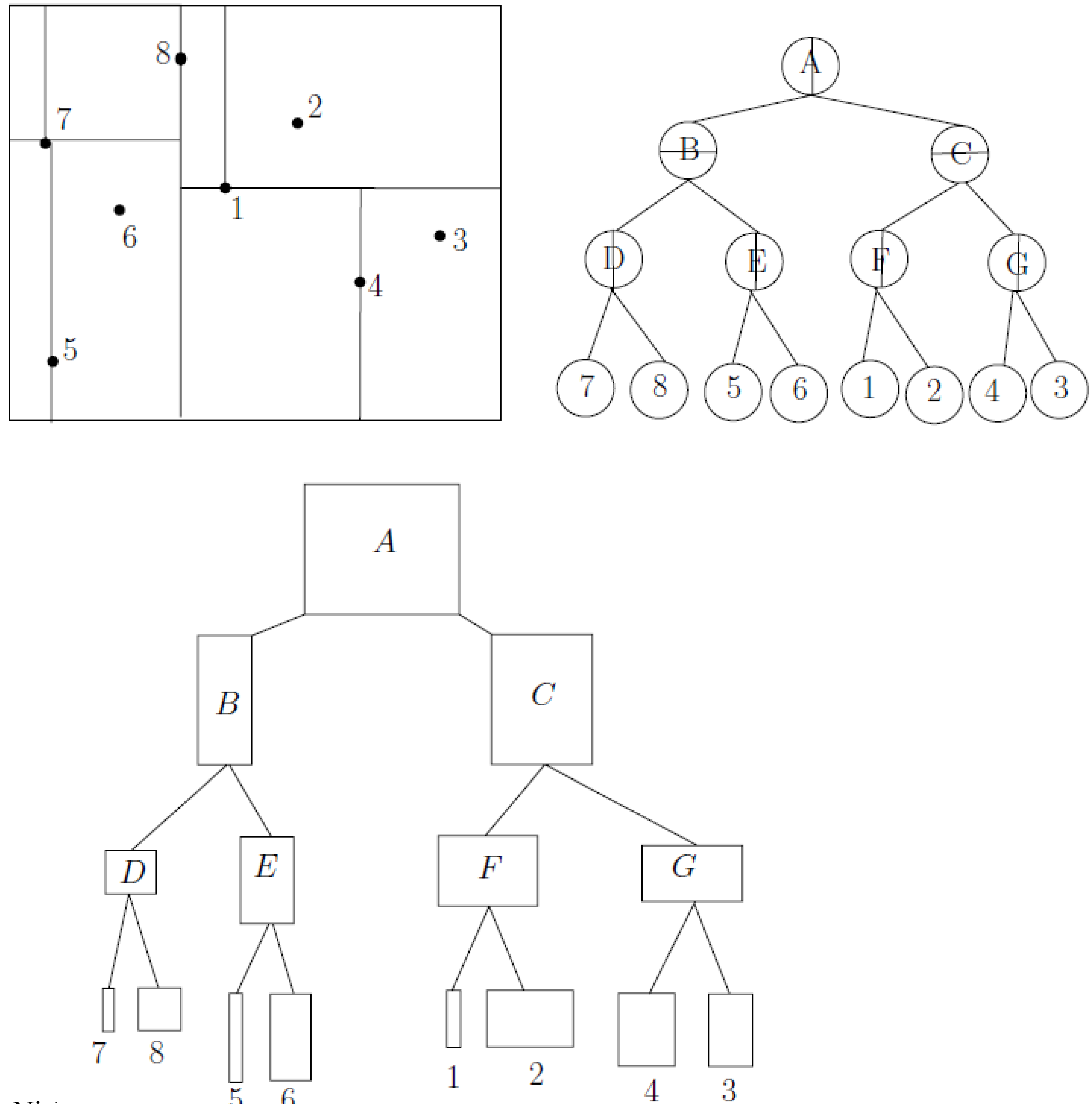


Nearest neighbor (NN) queries in small dimensions...

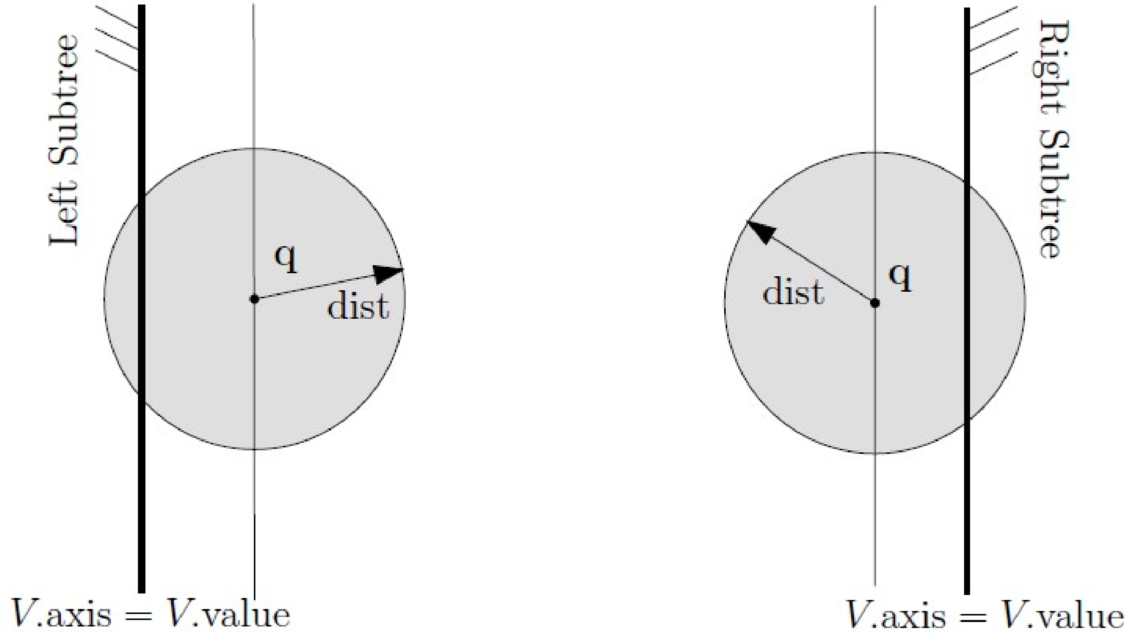
http://en.wikipedia.org/wiki/Kd-tree

```
kDTree(𝒫, l)
  ◁ Build a 2D kD-tree ▷
  ◁ l denote the level. Initially, l = 0 ▷
  if |𝒫| = 1
     then return Leaf(𝒫)
     else  if Even(l)
              then ◁ Compute the median x-abscissa (vertical split) ▷
                   x_l = MedianX(𝒫)
                   𝒫_left = {p ∈ 𝒫 | x(p) ≤ x_l}
                   𝒫_right = {p ∈ 𝒫 | x(p) > x_l}
                   return Tree(x_l, kDTree(𝒫_left, l + 1), kDTree(𝒫_right, l + 1));
              else ◁ Compute the median y-abscissa (horizontal split) ▷
                   y_l = MedianY(𝒫)
                   𝒫_bottom = {p ∈ 𝒫 | y(p) ≤ y_l}
                   𝒫_top = {p ∈ 𝒫 | y(p) > y_l}
                   return Tree(y_l, kDTree(𝒫_bottom, l + 1), kDTree(𝒫_top, l + 1));
```

Build time: O(dn log n) with O(dn) memory

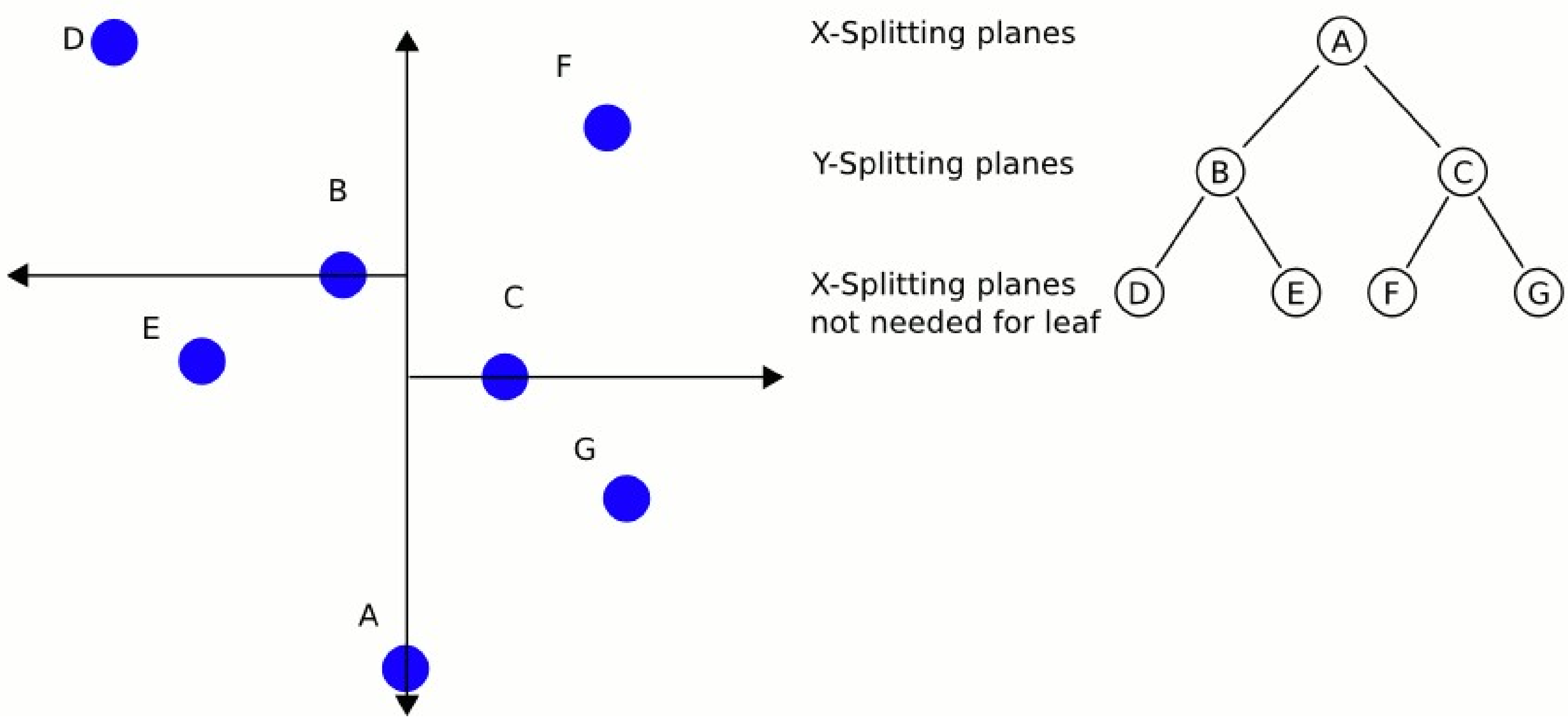

8
7
2
1
6
3
4
5
A
B
C
D
E
F
G
7 8 5 6 1 2 4 3
A
B
C
D
E
F
G
7 8 5 6 1 2 4 3

```
SearchNNinKDTree(q, V; p, dist)
◁ Input: ▷
◁ V: a kD-Tree node ▷
◁ q: a query point ▷
◁ Output: ▷
◁ p: nearest neighbor point ▷
◁ dist: distance to the nearest neighbor ▷
if V.left = V.right = NULL
   then ◁ Leaf of a kD-Tree ▷
        dist' = ||q − V.point||
        if dist' < dist
           then dist = dist'
                p = V.point
   else if q_{V.axis} ≤ V.value
           then ◁ Search on the left subtree first ▷
                SearchNNinKDTree(q, V.left; p, dist)
                if q_{V.axis} + dist > V.value
                   then SearchNNinKDTree(q, V.right; p, dist)
           else ◁ Search on the right subtree first ▷
                SearchNNinKDTree(q, V.right; p, dist)
                if q_{V.axis} − dist ≤ V.value
                   then SearchNNinKDTree(q, V.left; p, dist)
```



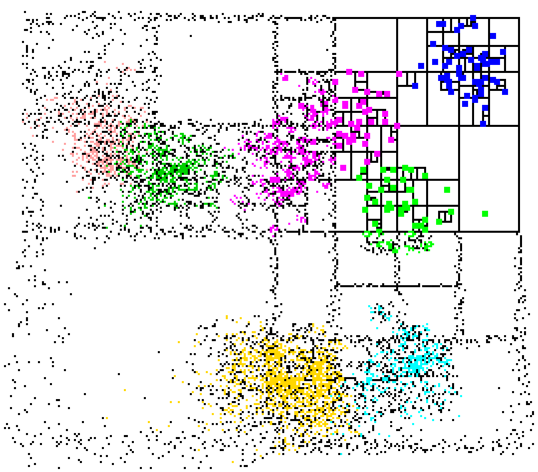
Query complexity: From $O(d\log n)$ to $O(dn)$

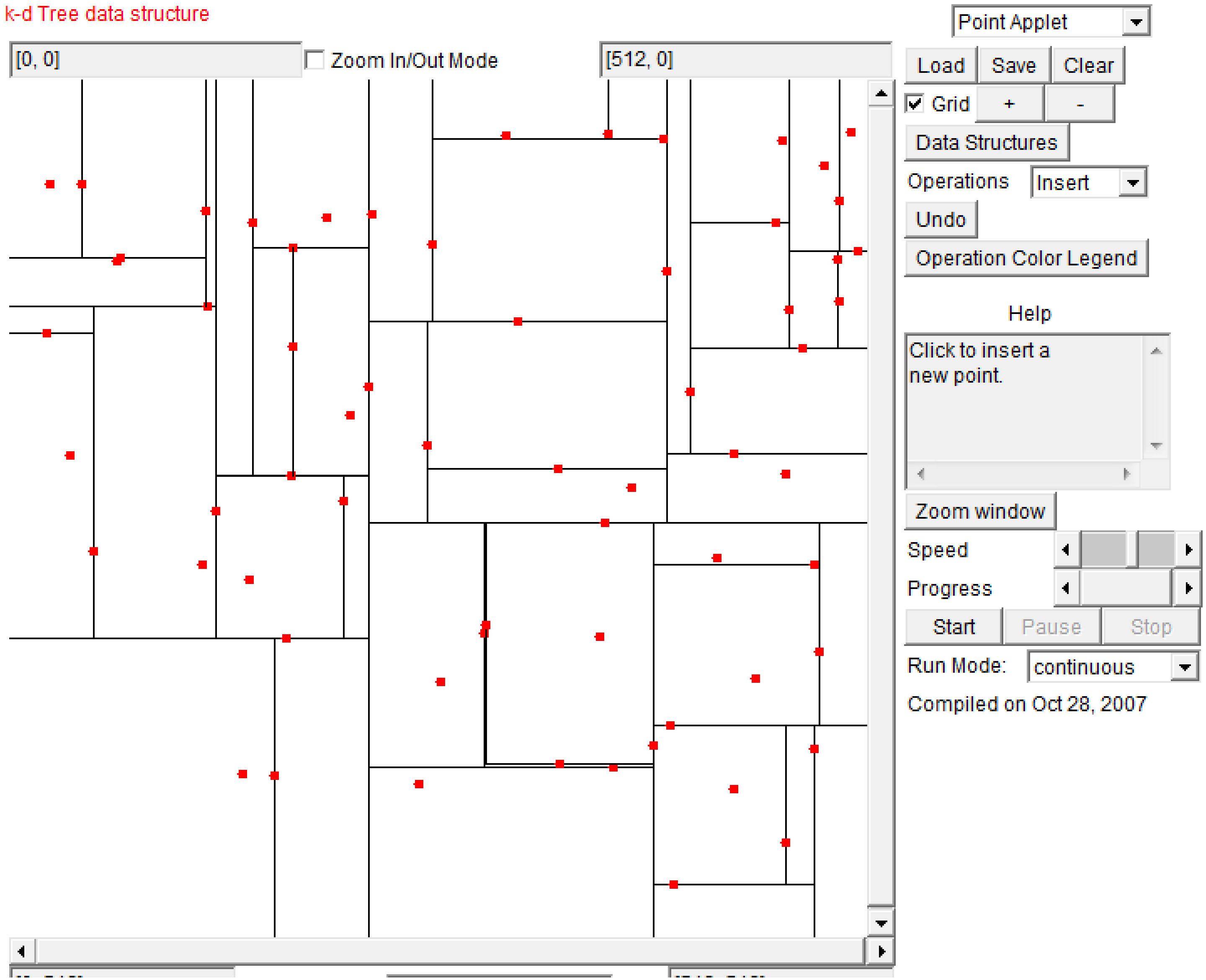
kD-trees for fast NN queries



Kd-Trees are extremely useful data-structures (many applications)

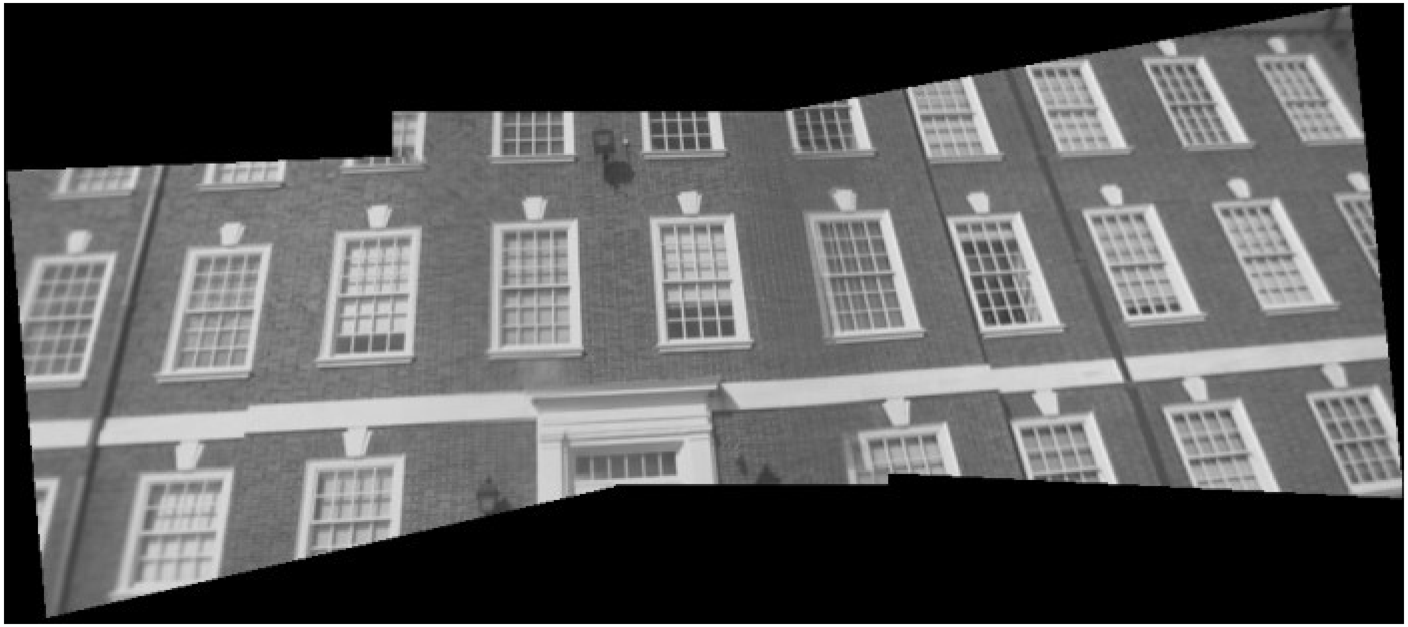
But also: Approximate nearest neighbors
http://www.cs.umd.edu/~mount/ANN/

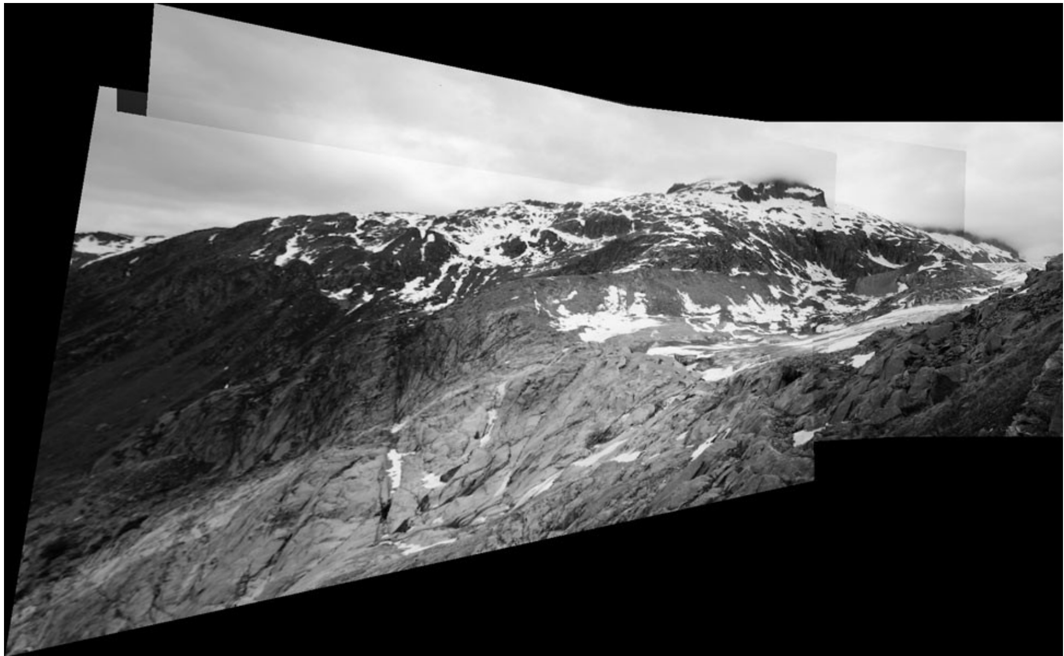




http://donar.umiacs.umd.edu/quadtree/points/kdtree.html

Homography (also called collineation)





Homography (Collineation)

$$\mathbf{r_i} = \begin{bmatrix} x'_i \\ y'_i \\ w'_i \end{bmatrix} = \mathbf{Hl_i} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ w_i \end{bmatrix}$$

$$\mathbf{l_i} = \begin{bmatrix} \frac{x_i}{w_i} & \frac{y_i}{w_i} \end{bmatrix}^T \qquad \mathbf{r}_i = \begin{bmatrix} \frac{x'_i}{w'_i} & \frac{y'_i}{w'_i} \end{bmatrix}^T$$

Assuming h33 is not zero, set it to 1 and get:

$$x'_i = \frac{h_{11}x_i + h_{12}y_i + h_{13}}{h_{31}x_i + h_{32}y_i + 1},$$

$$y'_i = \frac{h_{21}x_i + h_{22}y_i + h_{23}}{h_{31}x_i + h_{32}y_i + 1}.$$

Homography (Collineation)

$$x'_i = \frac{h_{11}x_i + h_{12}y_i + h_{13}}{h_{31}x_i + h_{32}y_i + 1},$$

$$y'_i = \frac{h_{21}x_i + h_{22}y_i + h_{23}}{h_{31}x_i + h_{32}y_i + 1}.$$

$$\Downarrow$$

$$x'_i = h_{11}x_i + h_{12}y_i + h_{13} - x'_i(h_{31}x_i + h_{32}y_i),$$

$$y'_i = h_{21}x_i + h_{22}y_i + h_{23} - y'_i(h_{31}x_i + h_{32}y_i).$$

Homography (Collineation)

From 4 pairs of point correspondences:

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x'_1 & -y_1x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y'_1 & -y_1y'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x'_2 & -y_2x'_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y'_2 & -y_2y'_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x'_3 & -y_3x'_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3y'_3 & -y_3y'_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4x'_4 & -y_4x'_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4y'_4 & -y_4y'_4 \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \end{bmatrix}$$

$$\underbrace{\mathbf{A}}_{8\times 8} \times \underbrace{\mathbf{h}}_{8\times 1} = \underbrace{\mathbf{b}}_{8\times 1}.$$

$$\mathbf{A}\mathbf{h} = \mathbf{b} \Longrightarrow \qquad \mathbf{h} = \mathbf{A}^{-1}\mathbf{b}.$$

Homography (Collineation)

From n pairs of point correspondences:

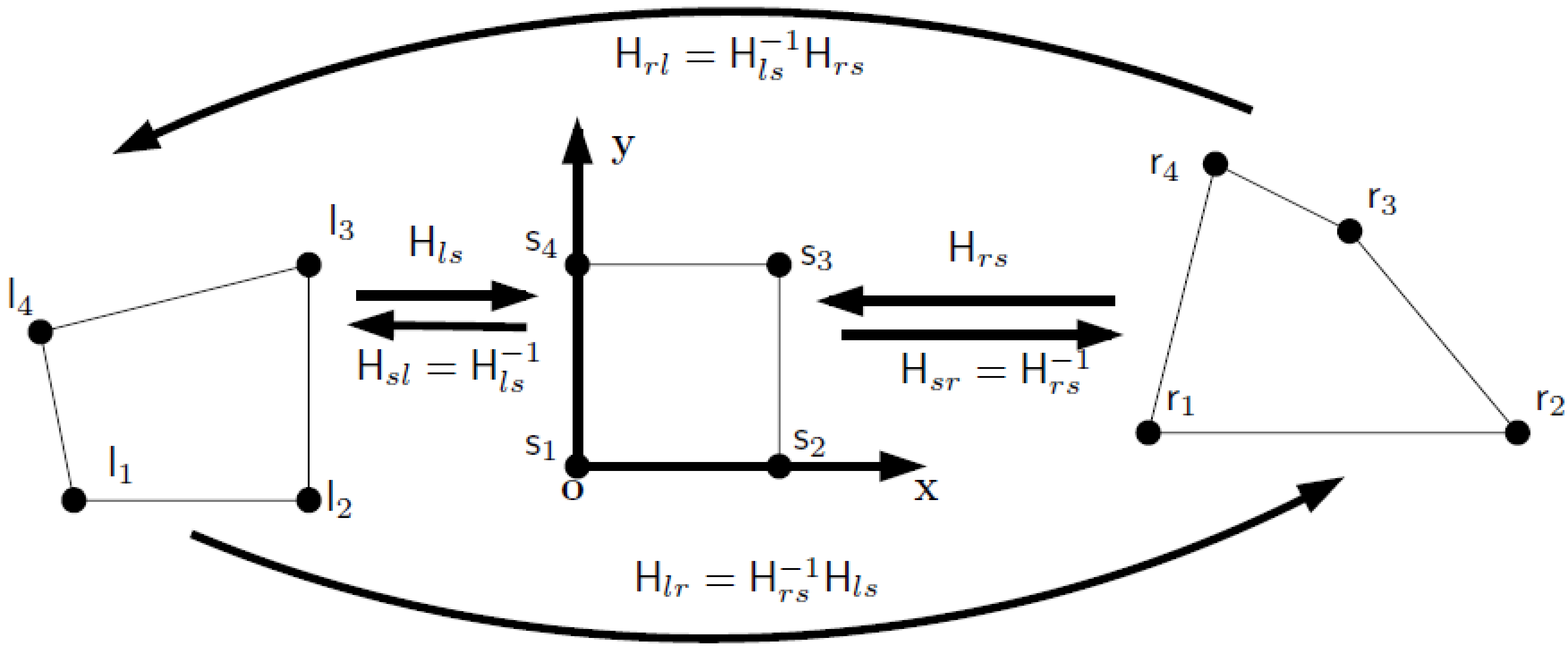
$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x'_1 & -y_1x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y'_1 & -y_1y'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x'_2 & -y_2x'_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y'_2 & -y_2y'_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x'_3 & -y_3x'_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3y'_3 & -y_3y'_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4x'_4 & -y_4x'_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4y'_4 & -y_4y'_4 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n & y_n & 1 & 0 & 0 & 0 & -x_nx'_n & -y_nx'_n \\ 0 & 0 & 0 & x_n & y_n & 1 & -x_ny'_n & -y_ny'_n \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \\ \vdots \\ x'_n \\ y'_n \end{bmatrix},$$

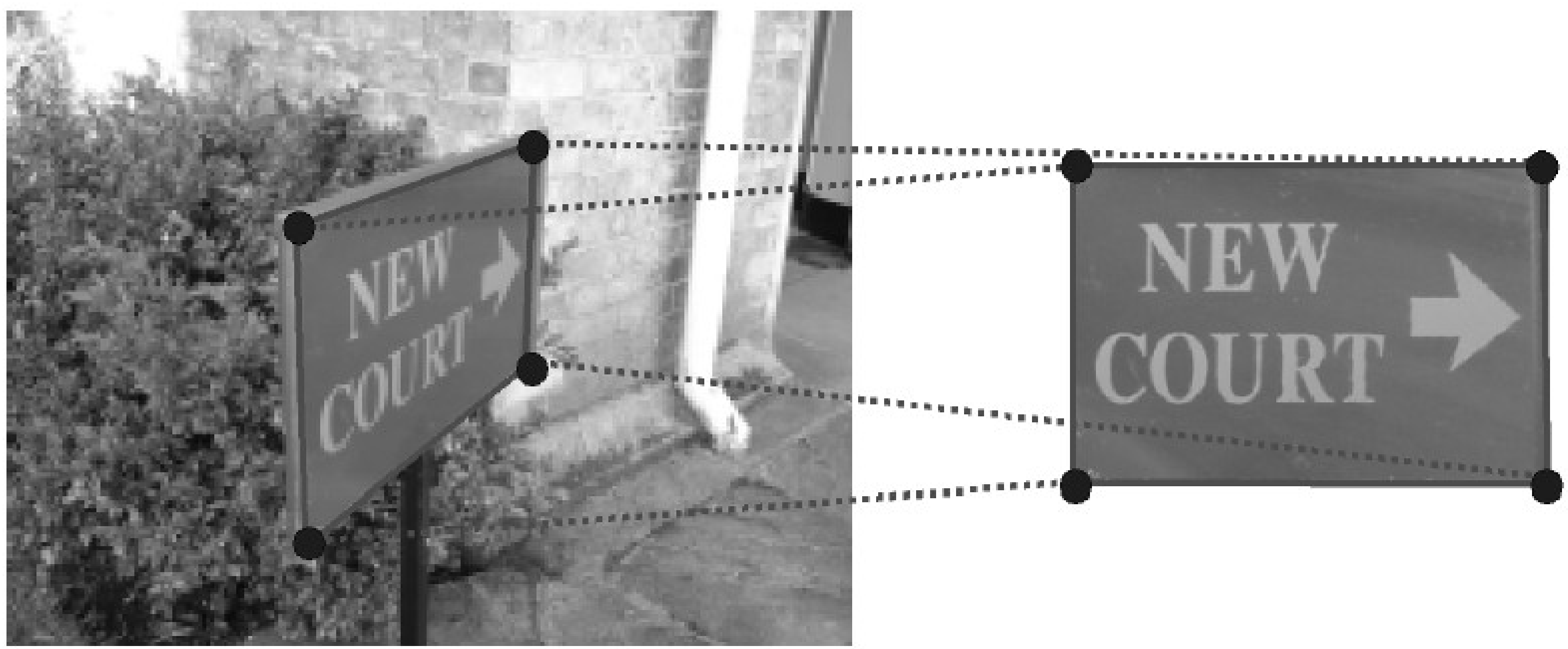
$$\underbrace{\mathbf{A}}_{2n\times 8} \times \underbrace{\mathbf{h}}_{8\times 1} = \underbrace{\mathbf{b}}_{2n\times 1}$$

$$\mathbf{h} = \mathbf{A}^+\mathbf{b}$$

Matrix pseudo-inverse

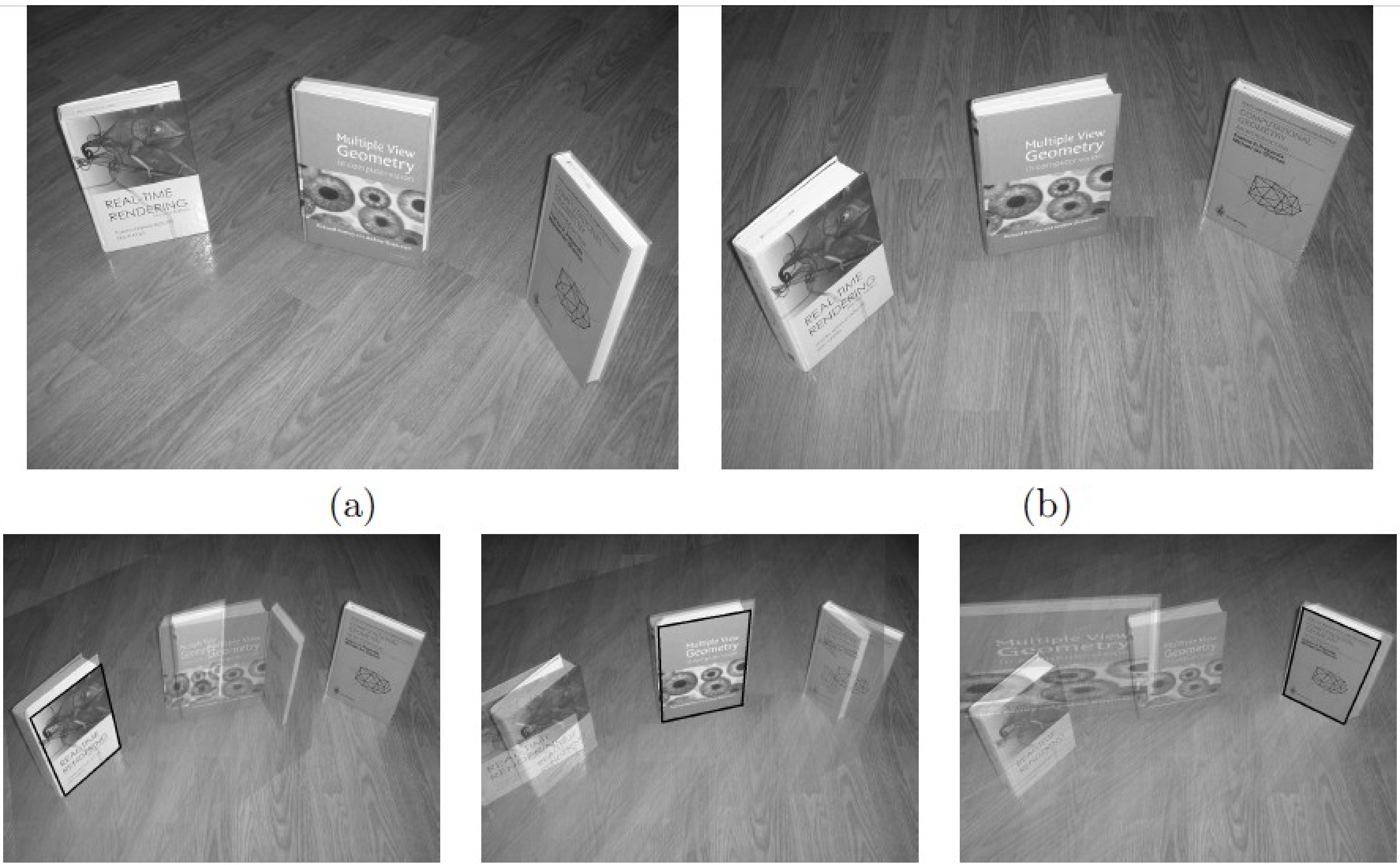
$$\mathbf{A}^+ = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T$$


$H_{rl} = H_{ls}^{-1} H_{rs}$
y
H_{ls}
$H_{sl} = H_{ls}^{-1}$
H_{rs}
$H_{sr} = H_{rs}^{-1}$
l_1 l_2 l_3 l_4
s_1 s_2 s_3 s_4
r_1 r_2 r_3 r_4
o
x
$H_{lr} = H_{rs}^{-1} H_{ls}$


NEW COURT
NEW COURT

Homography (Collineation)

Matching planar surfaces...



(a) (b)

Homography (Collineation)

Matching perspective pictures acquired from the same nodal point

(a) (b) (c) (d)

(e)

Homography (Collineation): Projective geometry

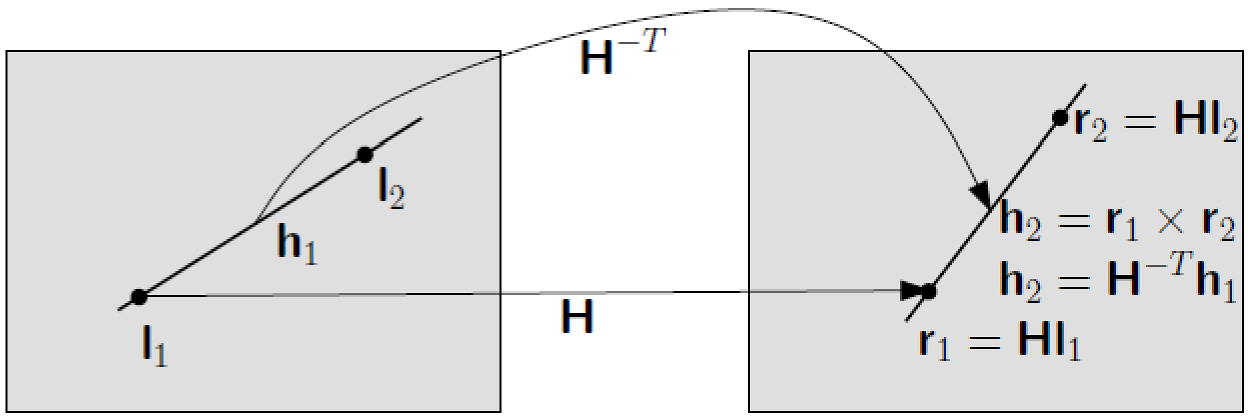
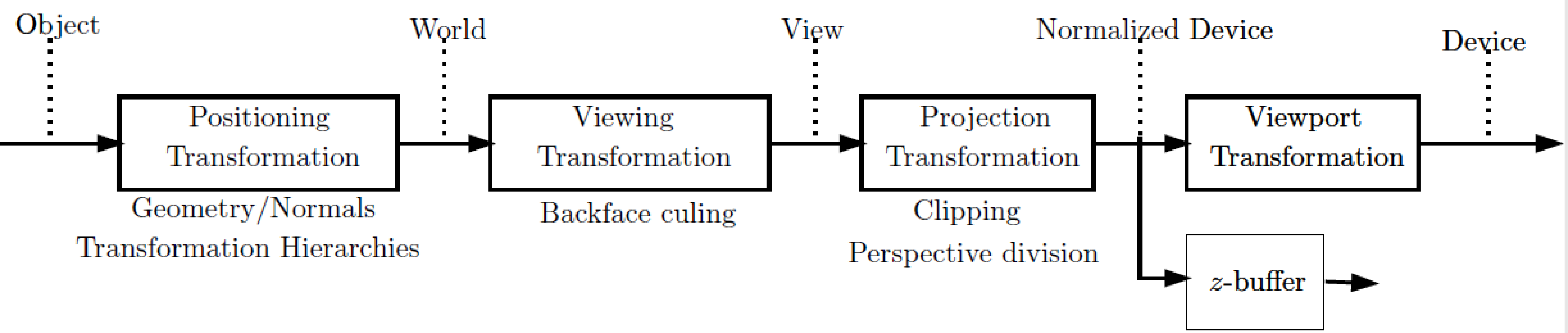


FIGURE 3.33 *Point/line mappings under a homography* $\mathbf{H}$. *Lines map by the transpose of the inverse of the homography mapping points:* $\mathbf{H}^{-T}$.

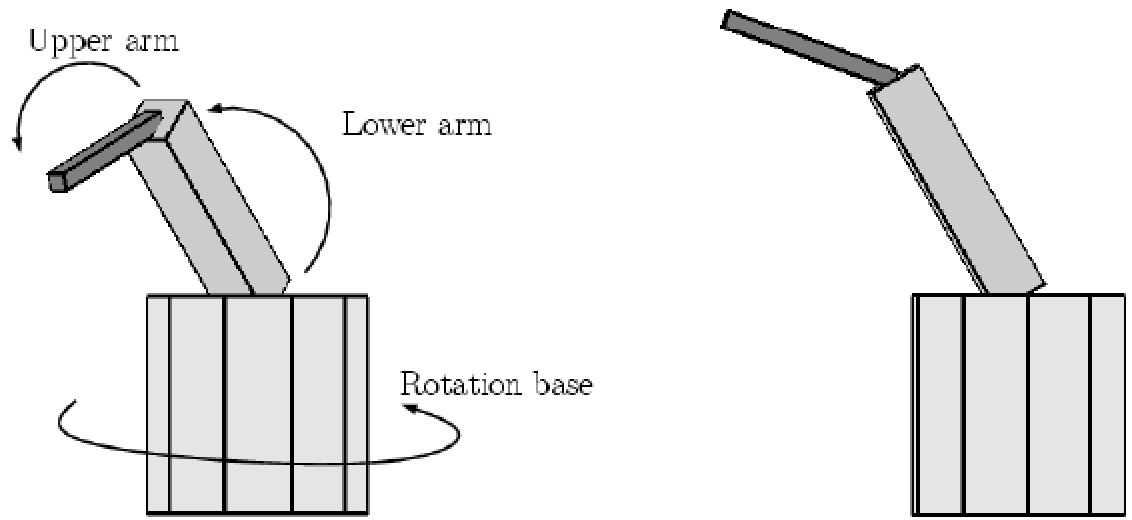
Play with duality point/line in projective space

It is more stable to select four pairs of lines instead of four points

Graphics pipeline



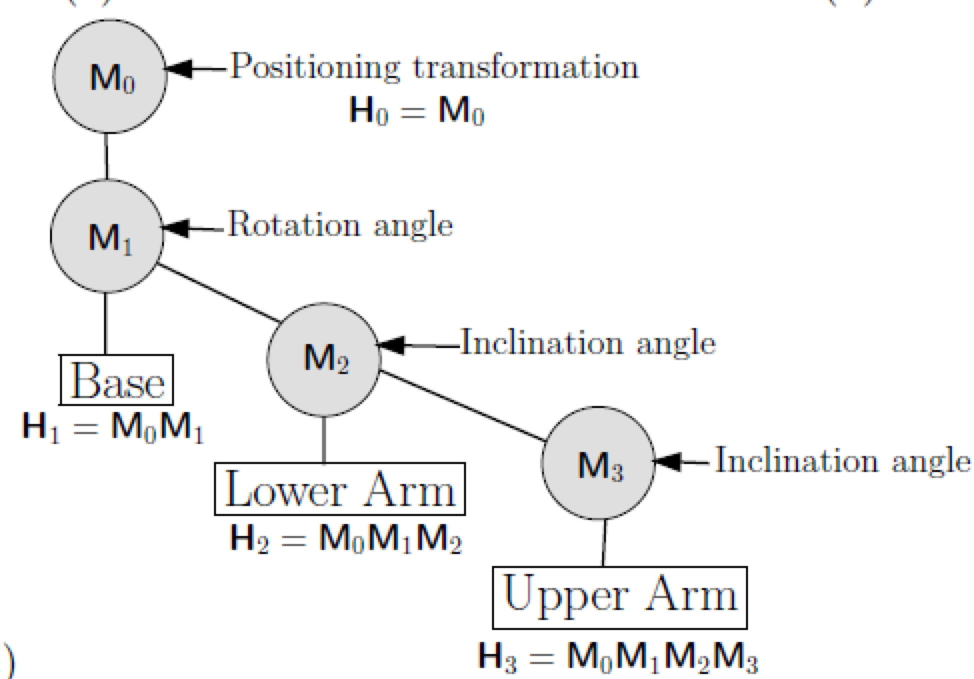
Graphics pipeline: **Scene graph**



(a) (b)

High level API is Java3D:
https://java3d.dev.java.net/

OpenGL in Processing



(c)

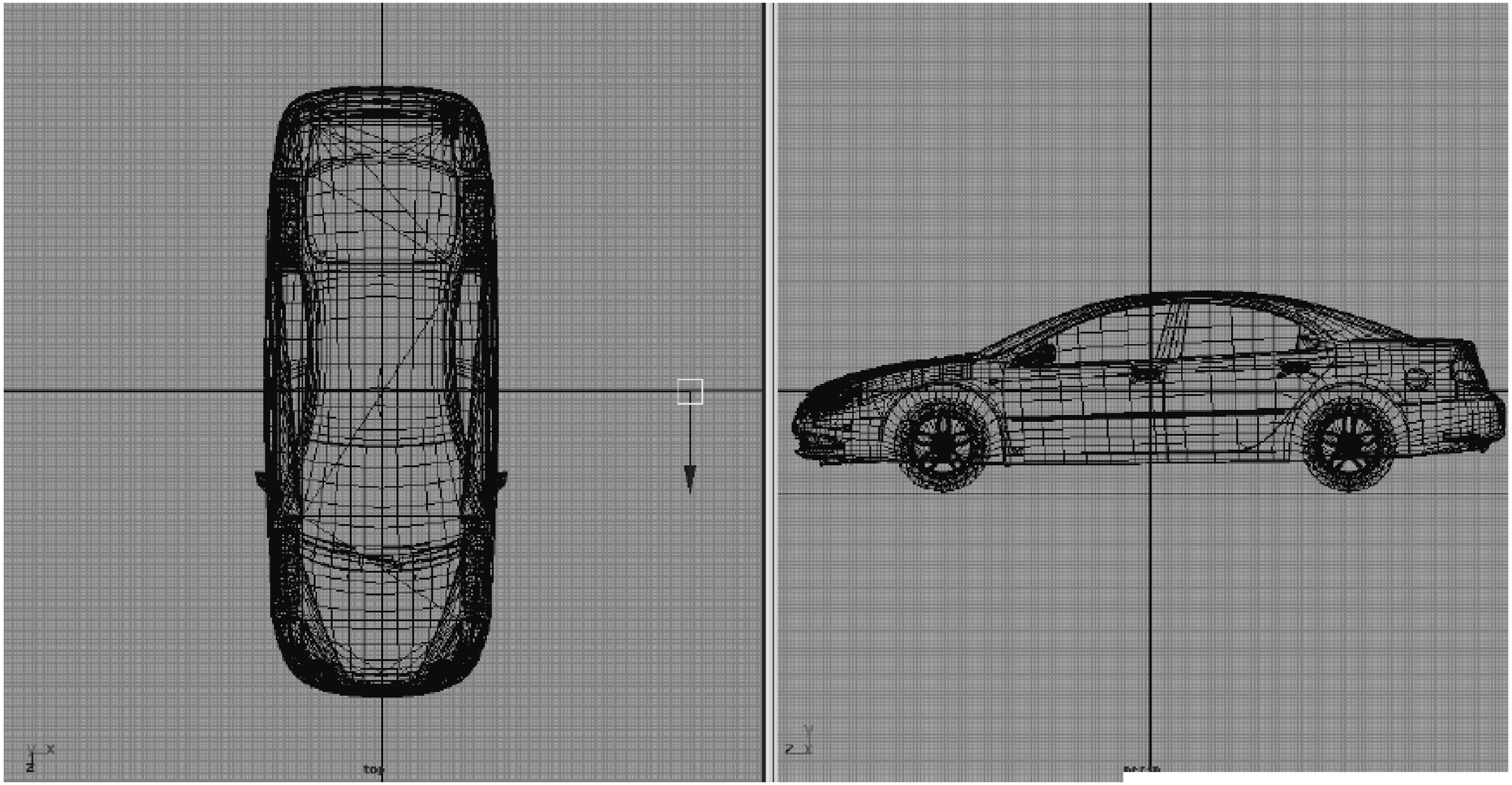
Scaled rigid transformation:

$$\mathbf{M}_k = \underbrace{\mathbf{T}_k}_{\text{Translation}} \underbrace{\mathbf{R}_k}_{\text{Rotation}} \underbrace{\mathbf{S}_k}_{\textit{Scale}}$$

Graphics pipeline: Projection

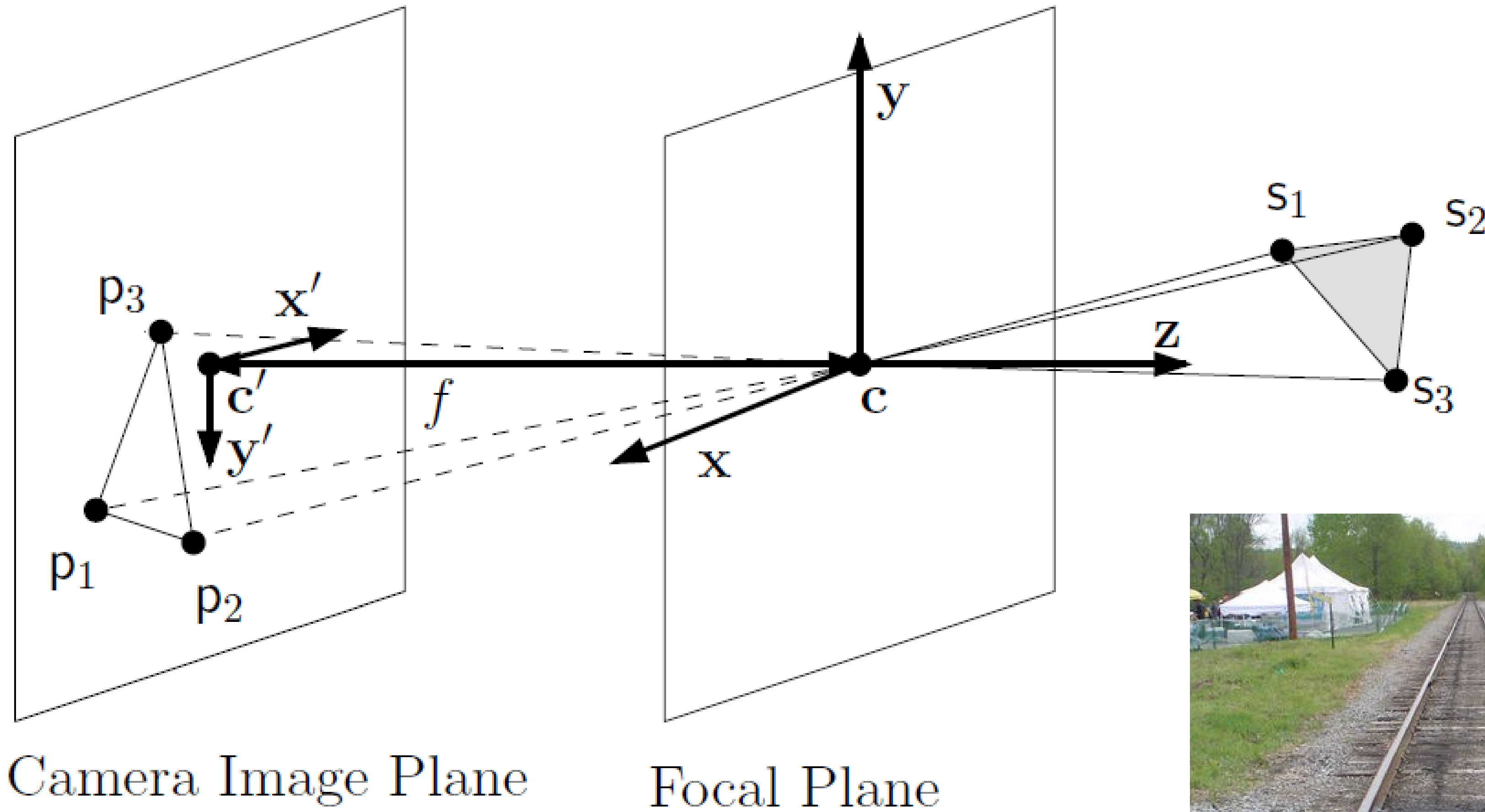
Projections are **irreversible** transformations

Orthographic projection



$$\mathsf{P}_O = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{e}_1{}^T \\ \mathbf{e}_2{}^T \\ \mathbf{0}^T \\ \mathbf{e}_4{}^T \end{bmatrix}$$

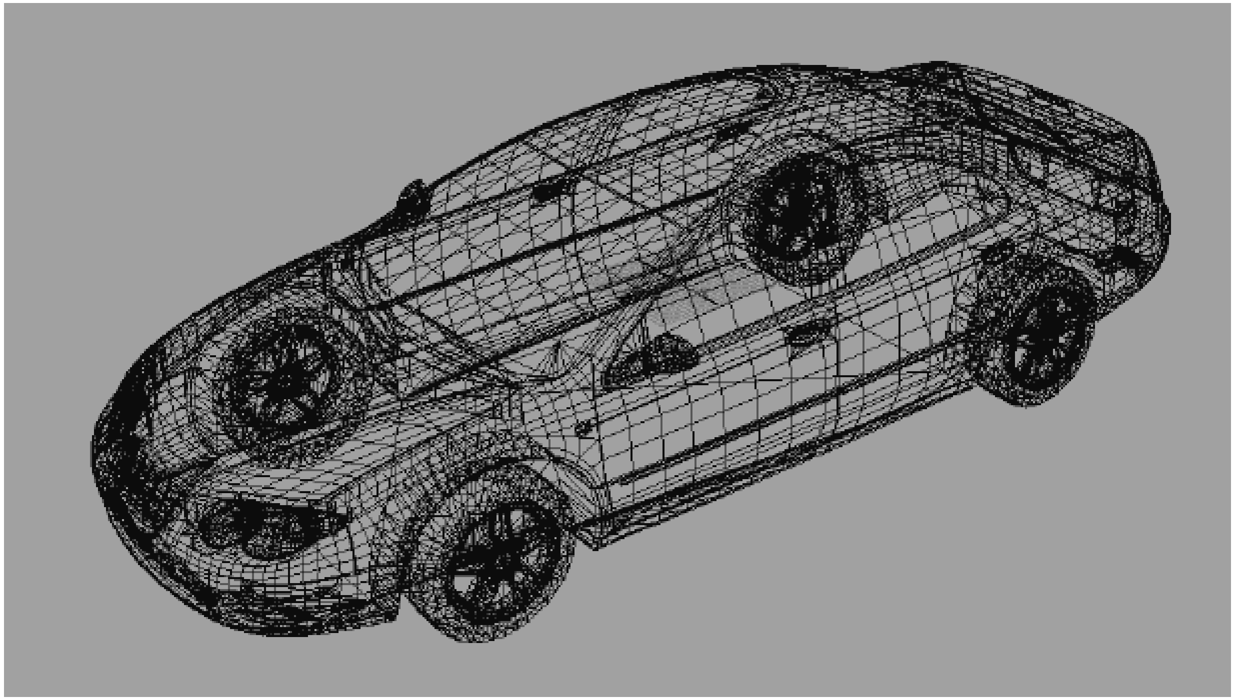
Perspective projection: Pinhole camera



$$\frac{x_p}{x_s} = \frac{y_p}{y_s} = -\frac{f}{z_s}$$

$$\frac{x_s}{x_p} = \frac{y_s}{y_p} = -\frac{z_s}{f}$$

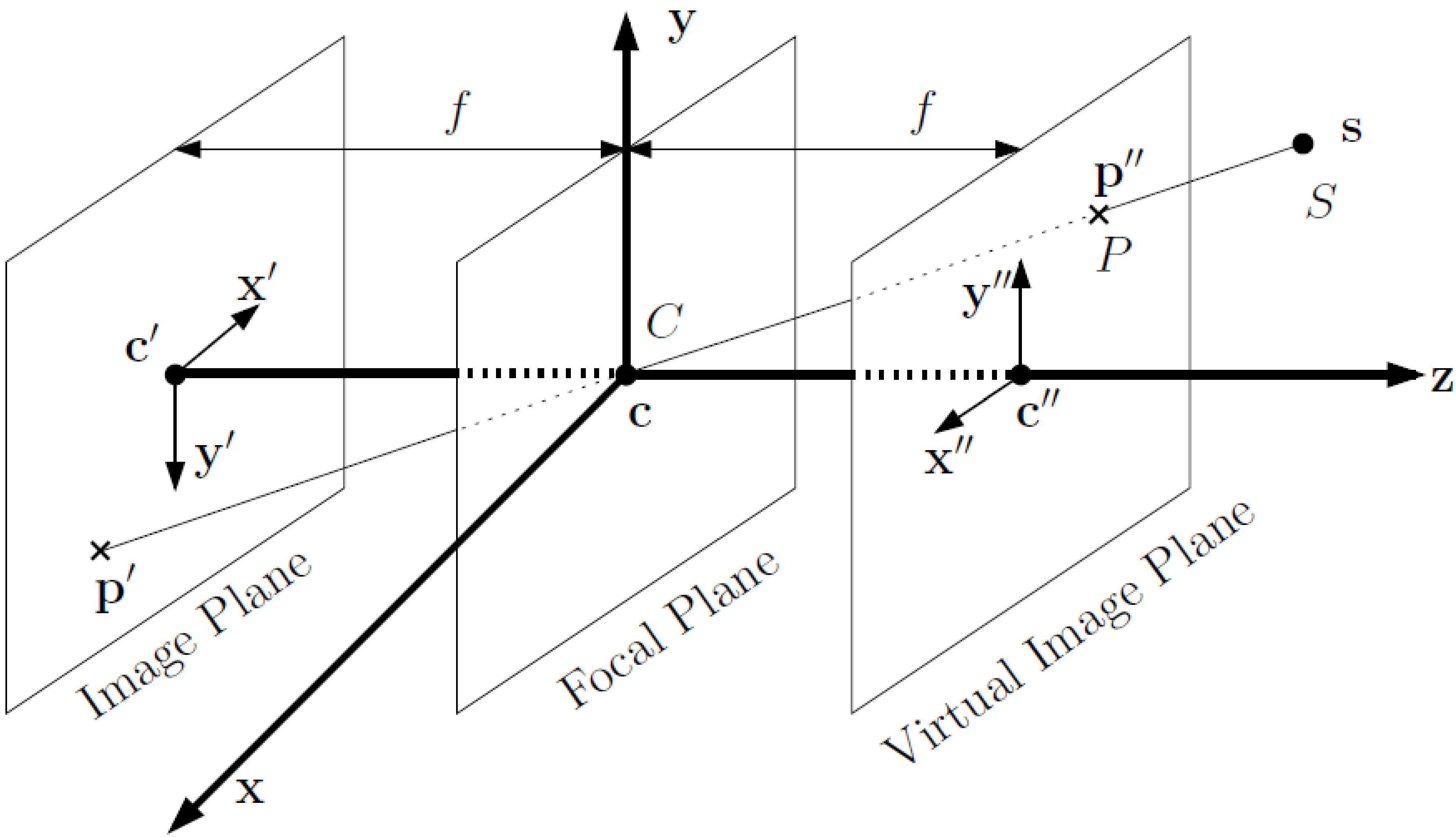
Graphics pipeline: Perspective projection



$$\mathbf{P}_P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{f} & 0 \end{bmatrix}$$

$$\mathbf{P}_P\mathbf{p} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{f} & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \frac{z}{f} \end{bmatrix} \xrightarrow{\text{perspective division}} \begin{bmatrix} \frac{xf}{z} \\ \frac{yf}{z} \\ f \end{bmatrix}$$

Graphics pipeline: Perspective projection

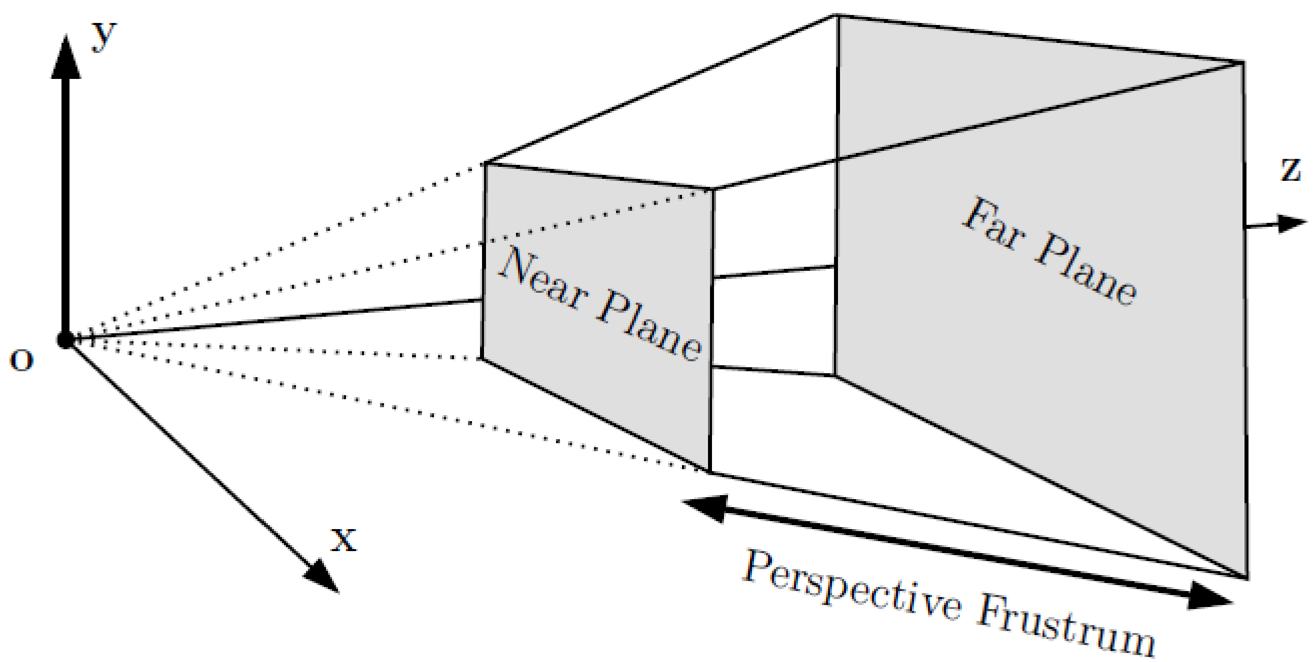


$$x_p = f\frac{x_s}{z_s} \quad \text{and} \quad y_p = f\frac{y_s}{z_s}.$$

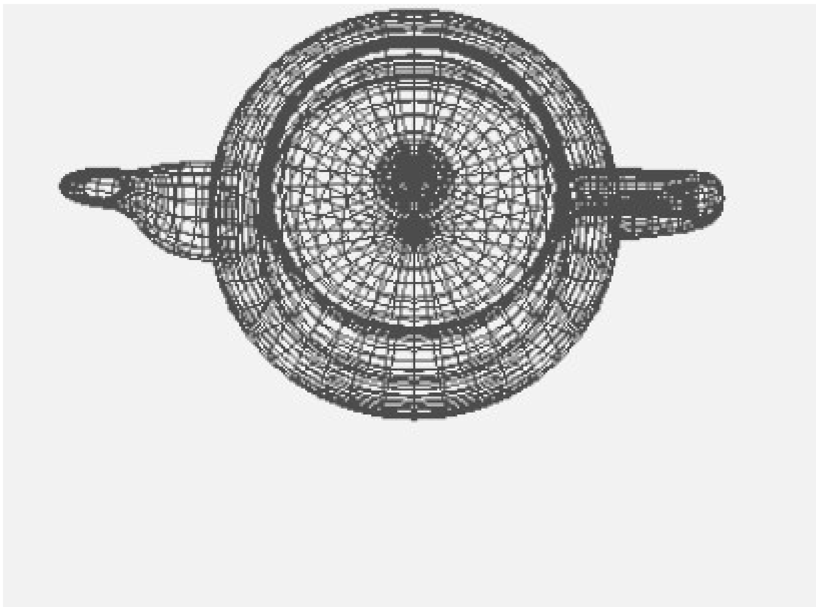
$$\begin{bmatrix} x_s f \\ y_s f \\ z_s f \\ z_s \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & f & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_s \\ y_s \\ z_s \\ 1 \end{bmatrix}$$

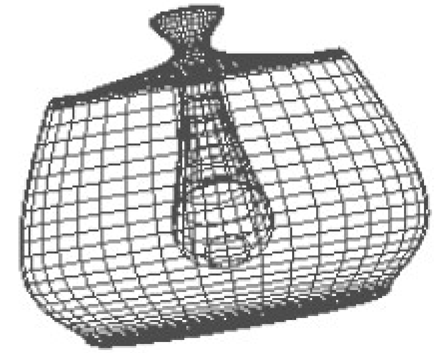
Graphics pipeline: Perspective truncated pyramid Perspective frustrum

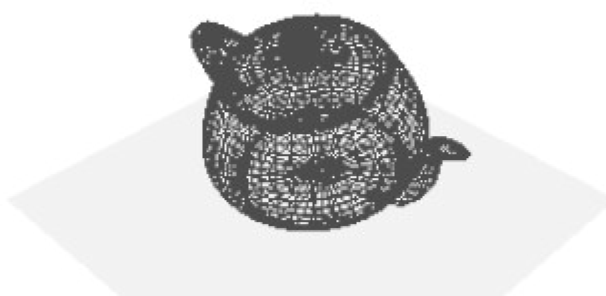
$$\mathbf{C}_P = \begin{bmatrix} \frac{2\text{near}}{\text{right}-\text{left}} & 0 & \frac{\text{right}+\text{left}}{\text{right}-\text{left}} & 0 \\ 0 & \frac{2\text{near}}{\text{top}-\text{bottom}} & \frac{\text{top}+\text{bottom}}{\text{top}-\text{bottom}} & 0 \\ 0 & 0 & -\frac{\text{far}+\text{near}}{\text{far}-\text{near}} & -2\frac{\text{far}\times\text{near}}{\text{far}-\text{near}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

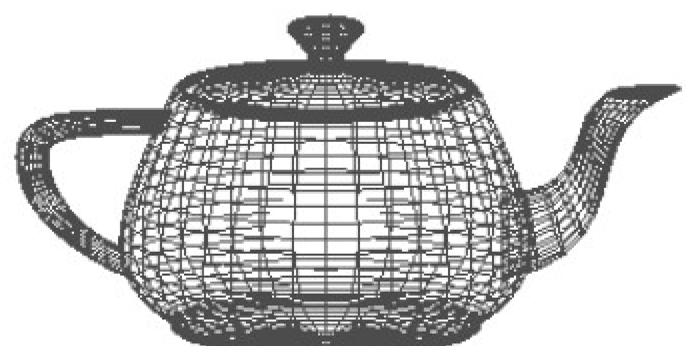


Graphics pipeline: Several viewports









Graphics pipeline (matrix concatenation):

Orthographic projection.

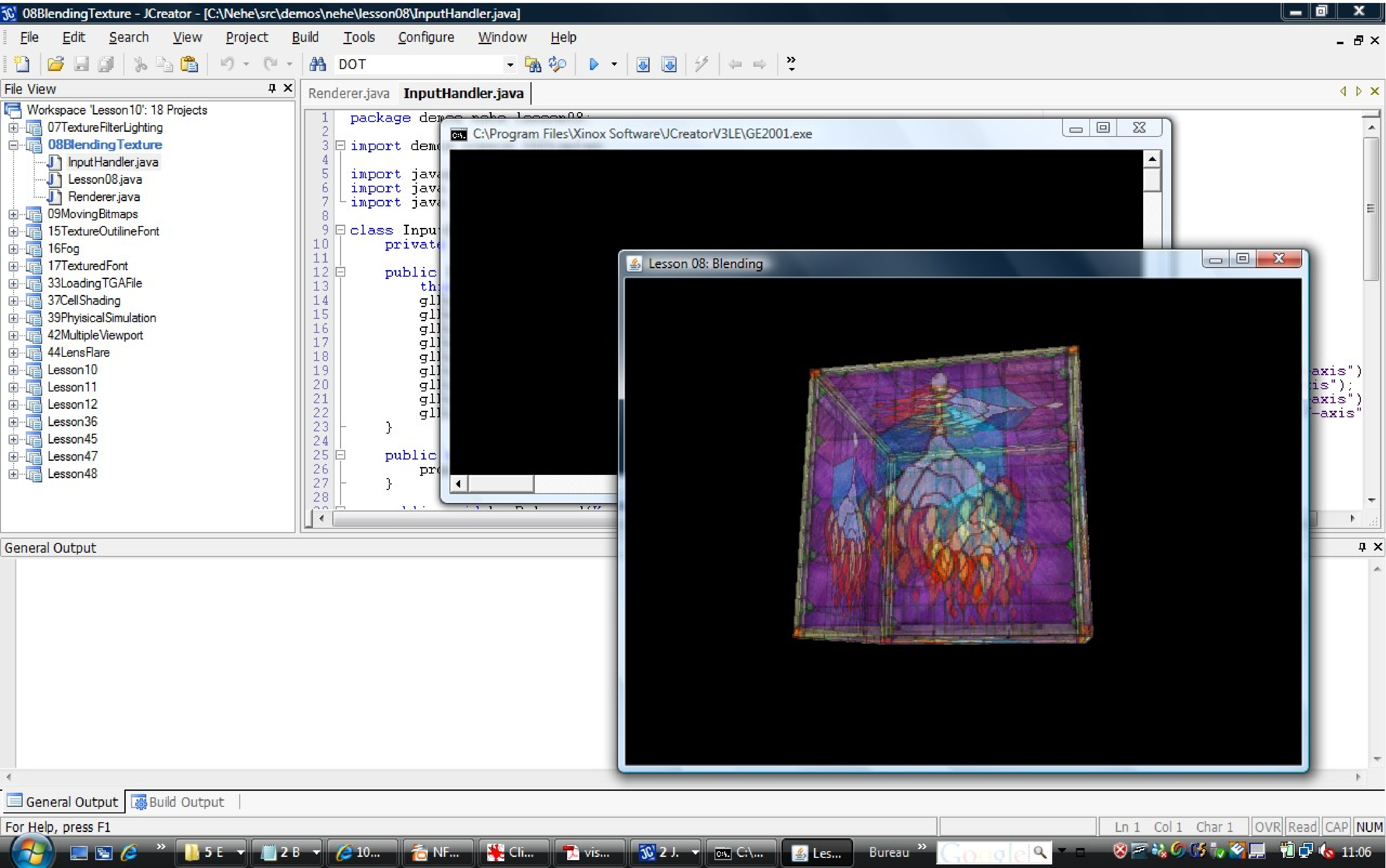
$$\mathrm{p} \longleftarrow \mathsf{M}_V \mathsf{C}_O \mathsf{V} \mathsf{M}_W \mathsf{s}.$$

Perspective projection.

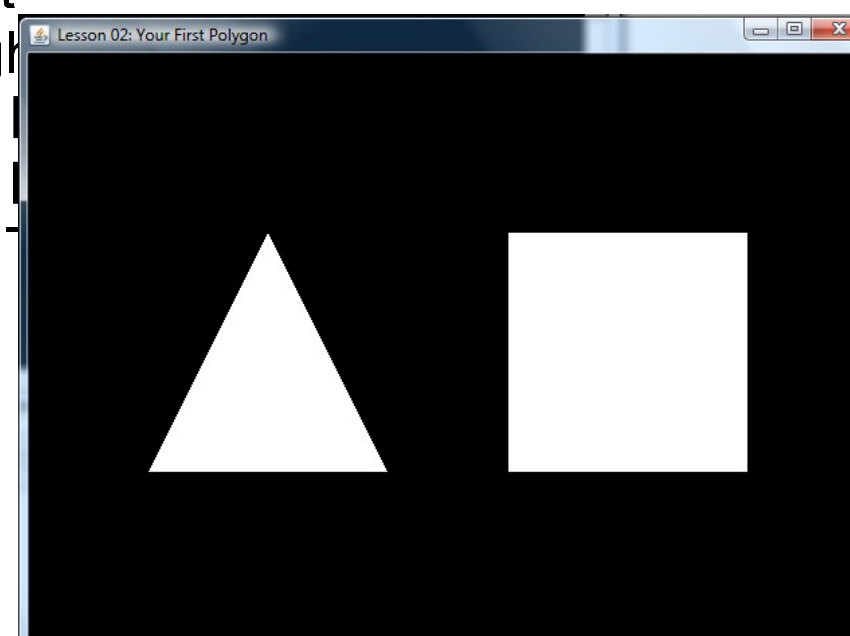
$$\mathrm{p} \longleftarrow \mathsf{M}_V \mathsf{C}_P \mathsf{V} \mathsf{M}_W \mathsf{s}.$$

Mv: positionning transformation
V: viewing transformation
C: clipping transformation
Mv: viewing transformation

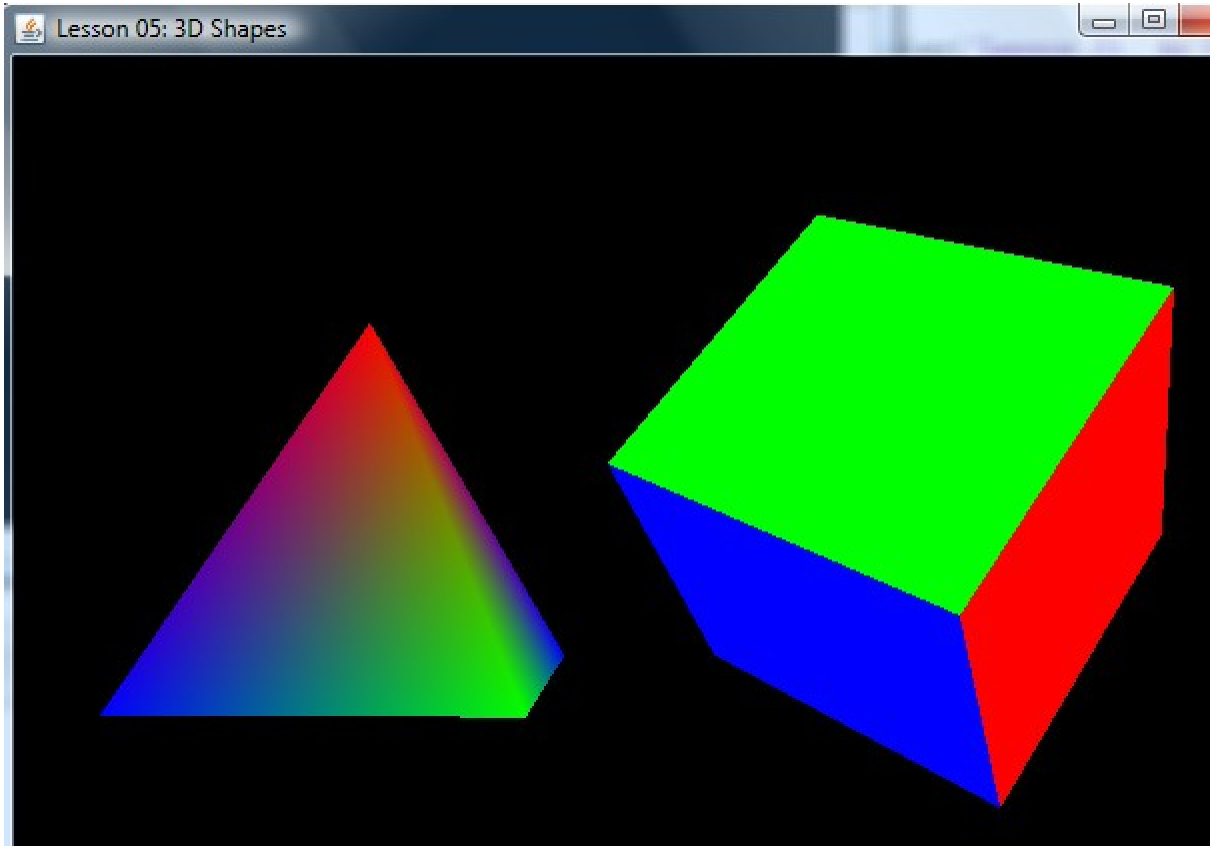
OpenGL in Java: Processing



```
public void display(GLAutoDrawable gLDrawable) {
    final GL gl = gLDrawable.getGL();
    gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);
    gl.glLoadIdentity();
    gl.glTranslatef(-1.5f, 0.0f, -6.0f);
    gl.glBegin(GL.GL_TRIANGLES);        // Drawing Using Triangles
    gl.glVertex3f(0.0f, 1.0f, 0.0f);    // Top
    gl.glVertex3f(-1.0f, -1.0f, 0.0f);  // Bottom Left
    gl.glVertex3f(1.0f, -1.0f, 0.0f);   // Bottom Right
    gl.glEnd();                         // Finished Drawing The Triangle
    gl.glTranslatef(3.0f, 0.0f, 0.0f);
    gl.glBegin(GL.GL_QUADS);            // Draw A Quad
    gl.glVertex3f(-1.0f, 1.0f, 0.0f);   // Top Left
    gl.glVertex3f(1.0f, 1.0f, 0.0f);    // Top Righ
    gl.glVertex3f(1.0f, -1.0f, 0.0f);   // Bottom
    gl.glVertex3f(-1.0f, -1.0f, 0.0f);  // Bottom
    gl.glEnd();                         // Done Drawing
    gl.glFlush();
  }
```



Processing: 3D color shapes



GLU (Utility)
GLUT (Utility Toolkit, including user interfaces.)

http://www.cs.umd.edu/~meesh/kmconroy/JOGLTutorial/