Computer Graphics I

Lecture 3: Coordinate spaces, transformations, projection & rasterization

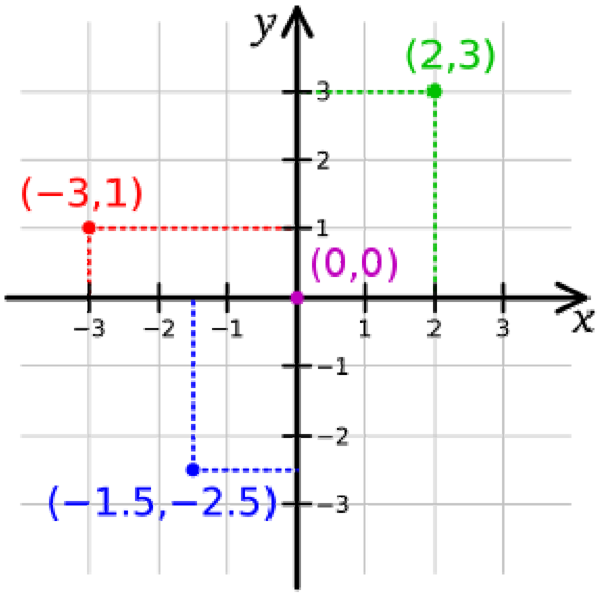
Xiaopei LIU

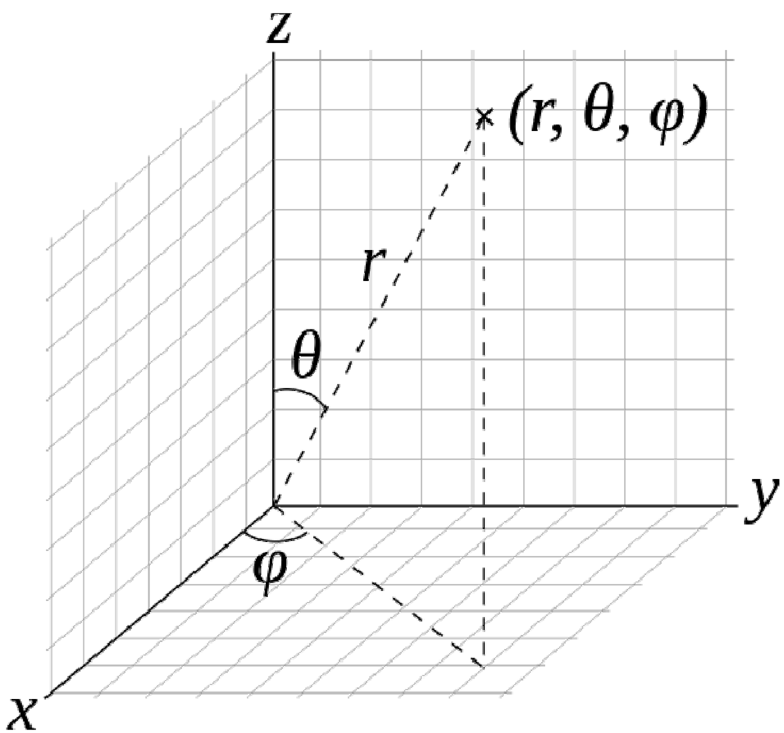
School of Information Science and Technology
ShanghaiTech University

1. Coordinate space

What is a coordinate system?

- **A geometric system**
  - Use one or more numbers, or *coordinates*, to uniquely determine the position of the points



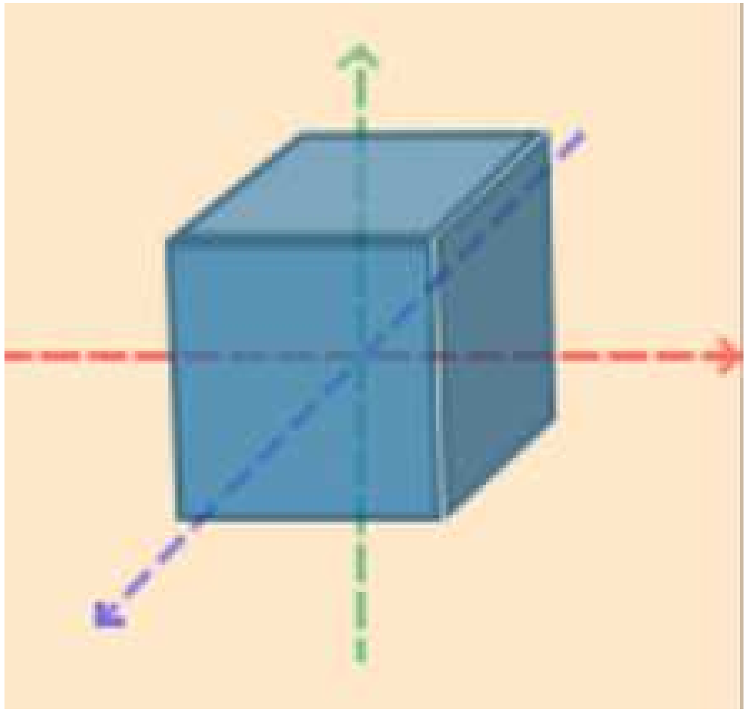


Coordinate spaces

- **Why we need coordinate space?**
  - It tells you where a point in space locates
- **Types of coordinate spaces in graphics**
  - Local coordinate space
  - World coordinate space
  - View coordinate space
  - Clip (including projection) coordinate space
  - Screen (device) coordinate space

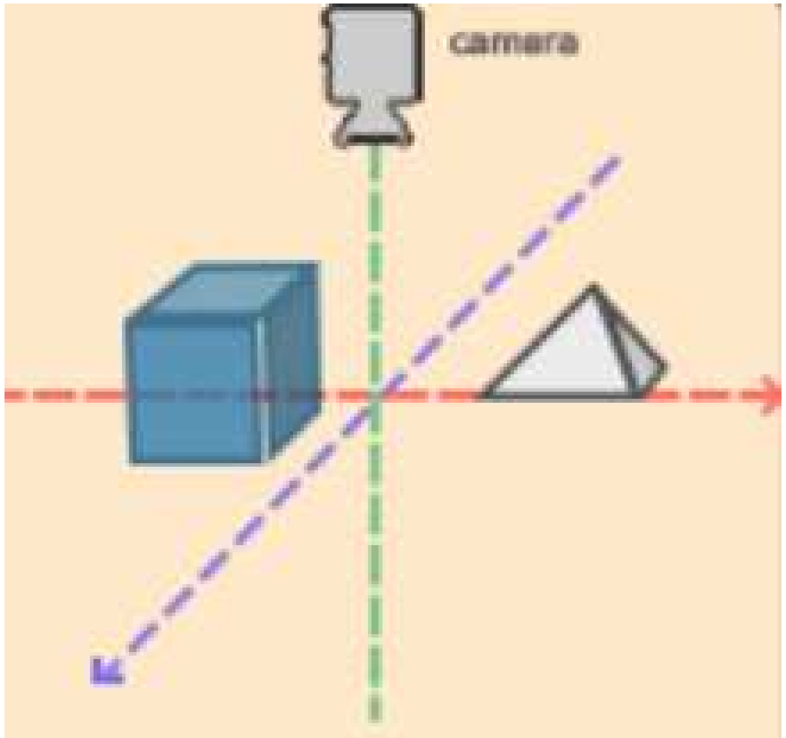
Coordinate spaces

- **Local(object) coordinate space**
  - Local coordinate space is the coordinate space that is local to your object



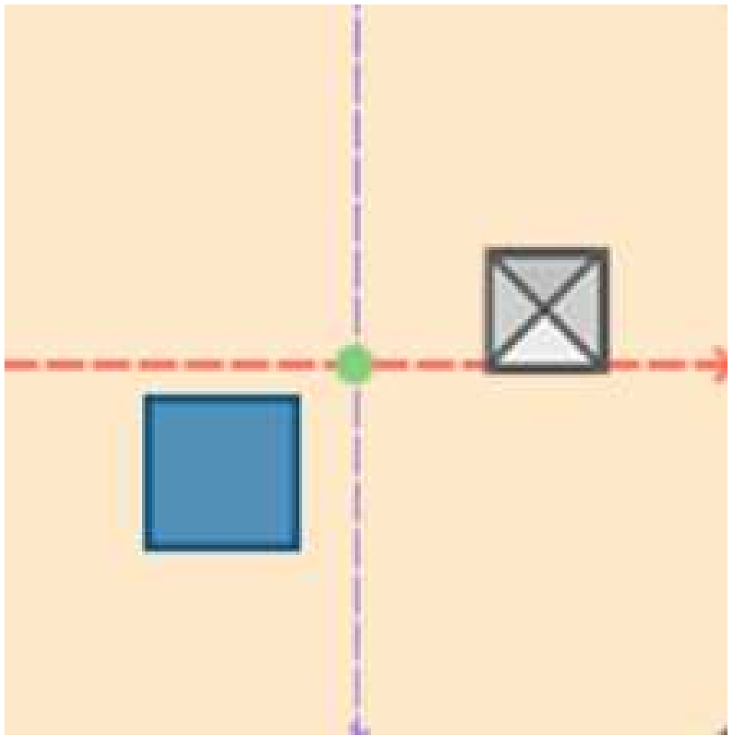
Coordinate spaces

- **World coordinate space**
  - A reference coordinate system that is always fixed
  - Local coordinate can be placed arbitrarily in world coordinate



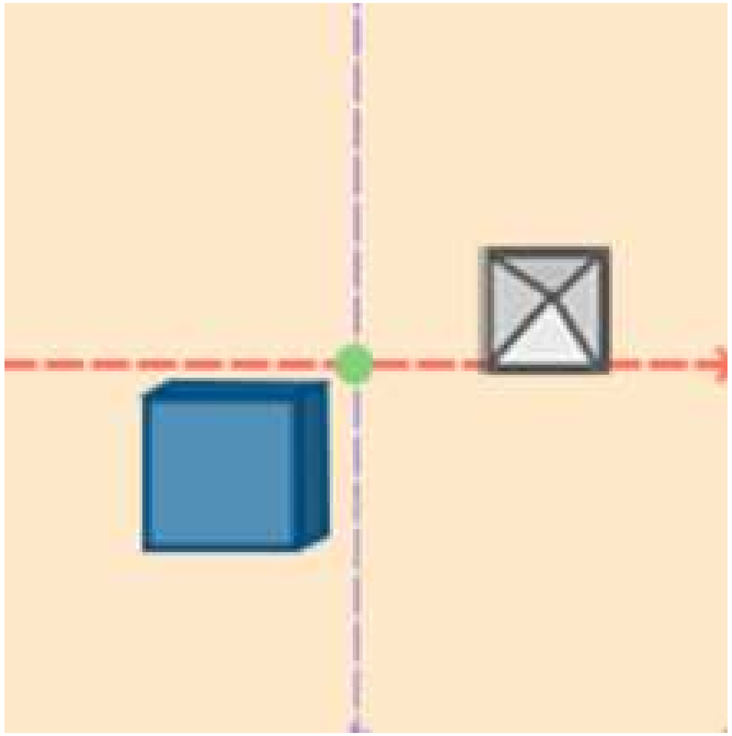
Coordinate spaces

- **View coordinate space**
  - Camera space or eye space
  - Transform world-space coordinates to coordinates that are in front of the user's view (still 3D)



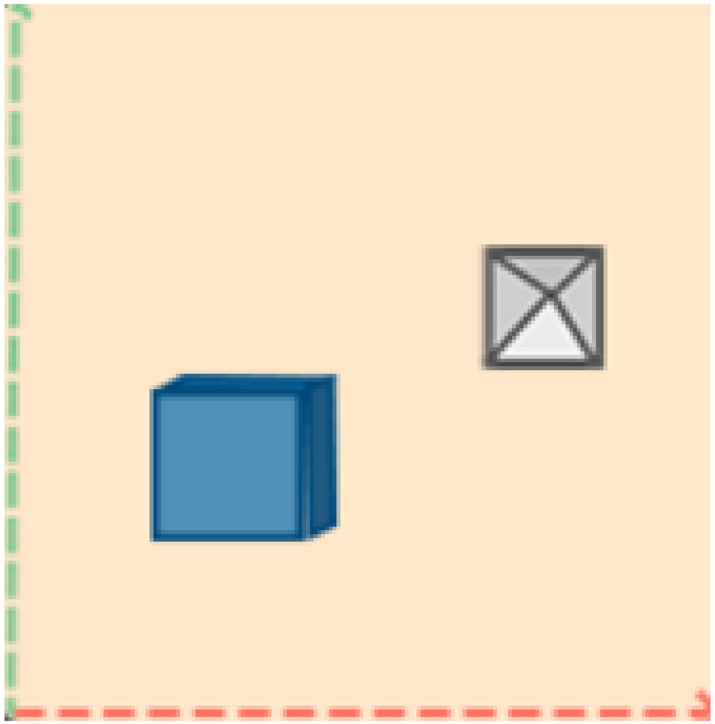
Coordinate spaces

- **Clip coordinate space**
  - Expect the coordinates to be within a specific range
  - Any coordinate that falls outside this range is clipped
  - Projection is done (3D to 2D)



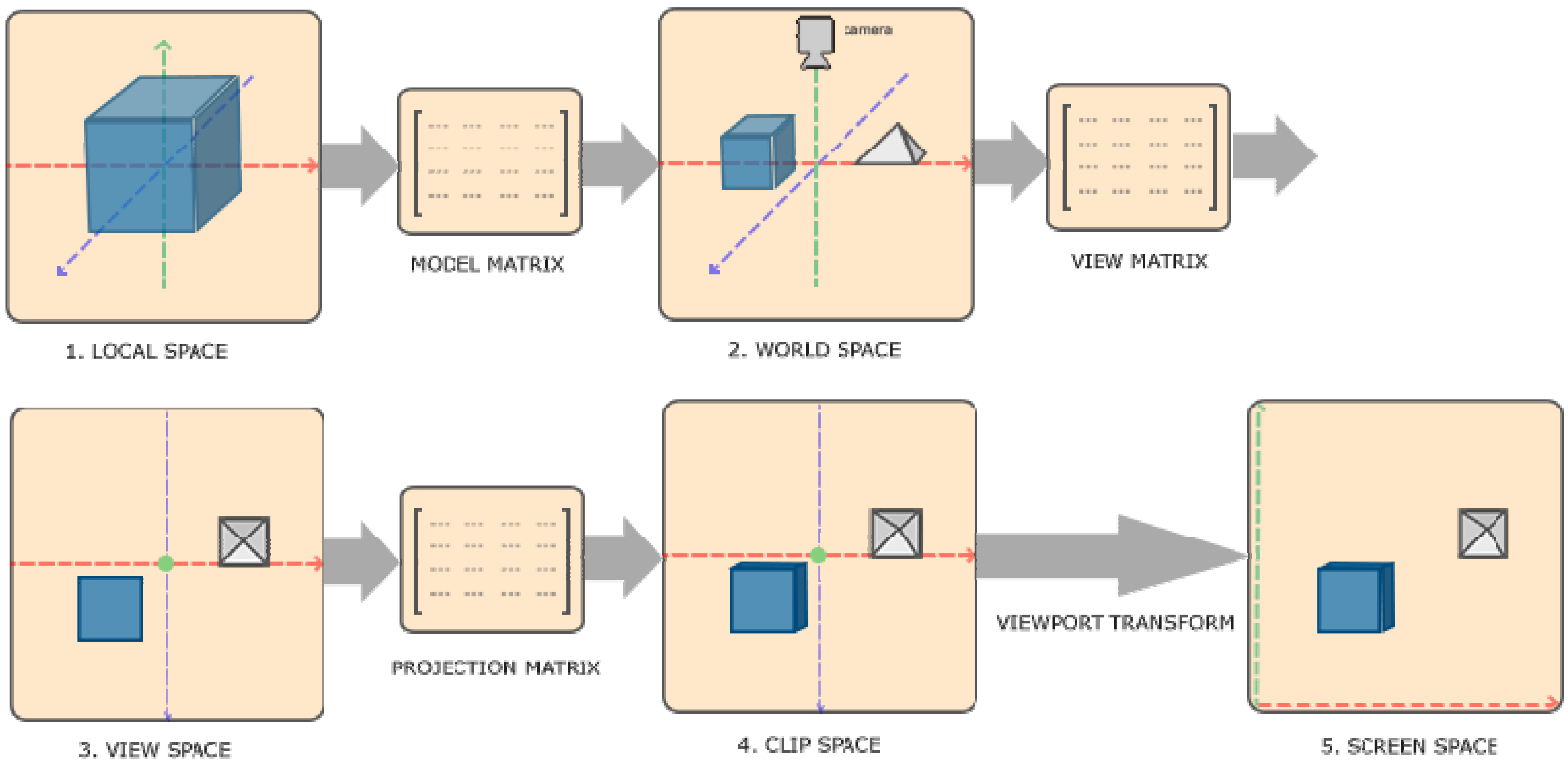
Coordinate spaces

- **Screen coordinate space**
  - The space for display
  - The resulting coordinates are then sent to the rasterizer to turn the continuous representation into fragments/pixels



Coordinate spaces

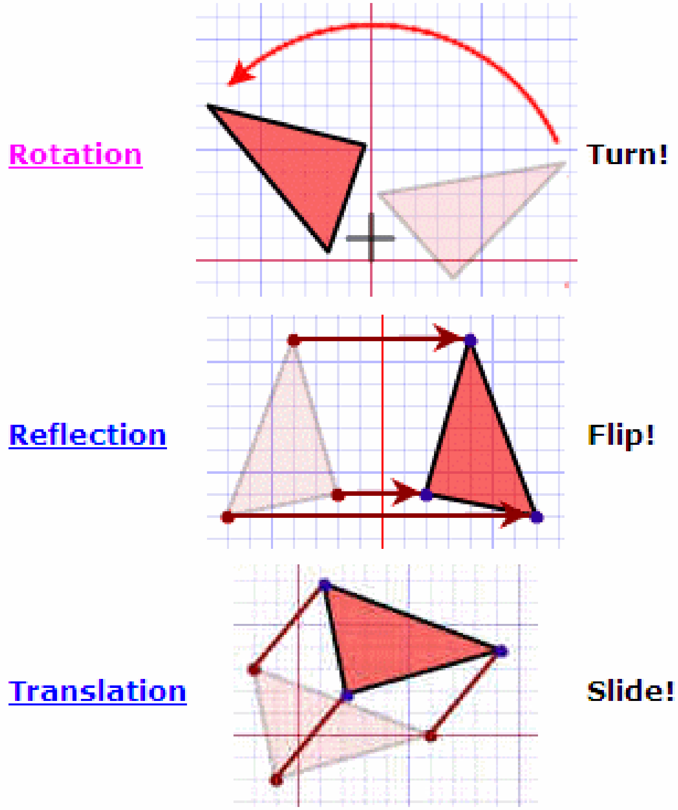
- **The global picture**
  - Space transformations using matrices



2. Model transformations

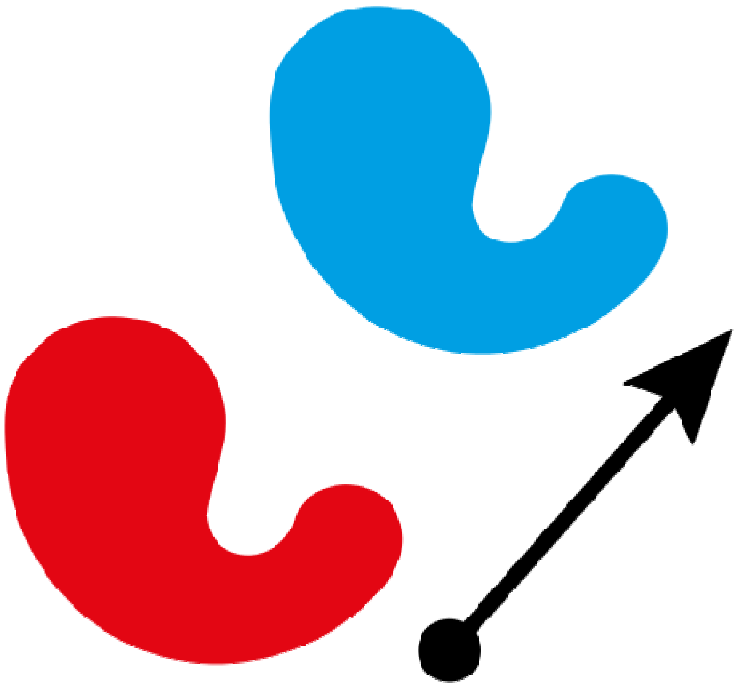
Geometric model transformations

- **A function whose domain and range are point sets**
  - Typical transformations
    - Translation
    - Rotation
    - Scaling
    - Reflection
    - Projective
    - etc.



Translation

- **Move every point in a space by the same distance in a given direction**



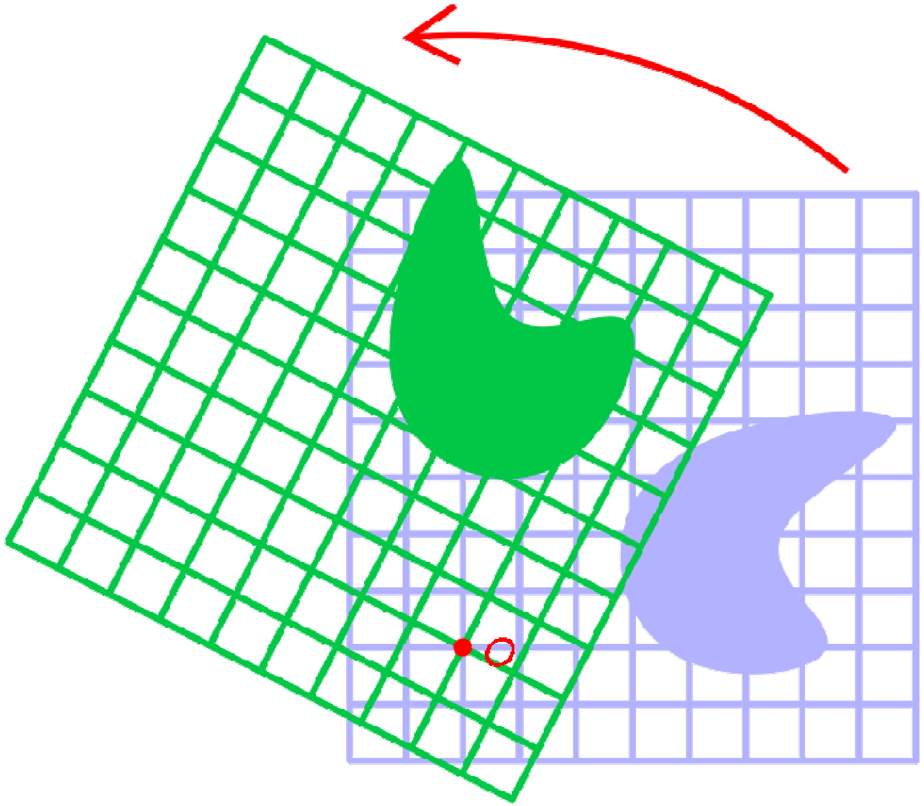
$$x' = x + t_x, \qquad y' = y + t_y$$

$$\mathbf{P} = \begin{bmatrix} x \\ y \end{bmatrix}, \qquad \mathbf{P}' = \begin{bmatrix} x' \\ y' \end{bmatrix}, \qquad \mathbf{T} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

$$\mathbf{P}' = \mathbf{P} + \mathbf{T}$$

Rotation

- **It leaves the distance between any two points unchanged after the transformation**

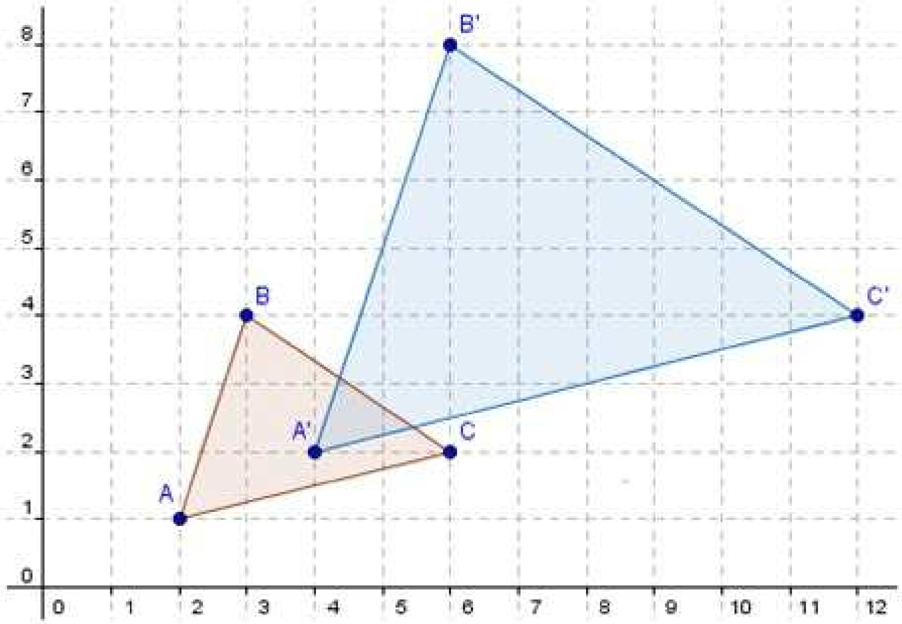


$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta.$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Scaling

- **A separate scale factor for each axis direction**
  - Isotropic/uniform: scale factor is the same for all axis directions
  - Anisotropic: scale factor is different for different axis directions



$$S_v p = \begin{bmatrix} v_x & 0 & 0 \\ 0 & v_y & 0 \\ 0 & 0 & v_z \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} = \begin{bmatrix} v_x p_x \\ v_y p_y \\ v_z p_z \end{bmatrix}$$

All in matrix form?

- **How can we represent these basic transforms with the same matrix operation?**
  - Extend the transformation matrix by one dimension
    - Translation in 3D

$$T_{\mathbf{v}}\mathbf{p} = \begin{bmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x + v_x \\ p_y + v_y \\ p_z + v_z \\ 1 \end{bmatrix} = \mathbf{p} + \mathbf{v}$$

All in matrix form?

- **How can we represent these basic transforms with the same matrix operations?**
  - Extend the transformation matrix by one dimension
    - Rotation in 3D along x-dimension

$$R_x(\theta)\,\mathbf{p} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

All in matrix form?

- **How can we represent these basic transforms with the same matrix operations?**
  - Extend the transformation matrix by one dimension
    - Scaling in 3D

$$S_v p = \begin{bmatrix} v_x & 0 & 0 & 0 \\ 0 & v_y & 0 & 0 \\ 0 & 0 & v_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} v_x p_x \\ v_y p_y \\ v_z p_z \\ 1 \end{bmatrix}$$

All in matrix form?

- **How can we represent these basic transforms with the same matrix operations?**
  - Combine all transformations together to form the final transformation

$$T = \begin{bmatrix} v_x & 0 & 0 & 0 \\ 0 & v_y & 0 & 0 \\ 0 & 0 & v_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling Rotation (x) Translation

Homogenous coordinates

- **Given a coordinate frame**
  - Ambiguity between the representations of a point $\mathbf{p}=[\mathbf{p}_x, \mathbf{p}_y, \mathbf{p}_z]^T$ and a vector $\mathbf{v}=[\mathbf{v}_x, \mathbf{v}_y, \mathbf{v}_z]^T$
  - We can write any point as the inner product $[s_1, s_2, s_3, 1][\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{p}_o]^T$

$$\mathbf{v}=[\mathbf{v}_x, \mathbf{v}_y, \mathbf{v}_z, 0]^T \qquad \mathbf{p}=[\mathbf{p}_x, \mathbf{p}_y, \mathbf{p}_z, 1]^T$$

  - We can write any vector as the inner product $[s'_1, s'_2, s'_3, 0][\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{p}_o]^T$
  - These four vectors of three s_i values and a zero or one are called the *homogeneous coordinates* of the point and vector

Homogeneous coordinates

- **In general, homogeneous points obey the identity**

$$(x,\, y,\, z,\, w) = \left(\frac{x}{w},\, \frac{y}{w},\, \frac{z}{w}\right)$$

  - **Homogenous coordinates can be used to see**
    - How a transformation matrix can describe how points and vectors in one frame can be mapped to another frame
  - **For more information**
    - https://www.tomdalling.com/blog/modern-opengl/explaining-homogenous-coordinates-and-projective-geometry/

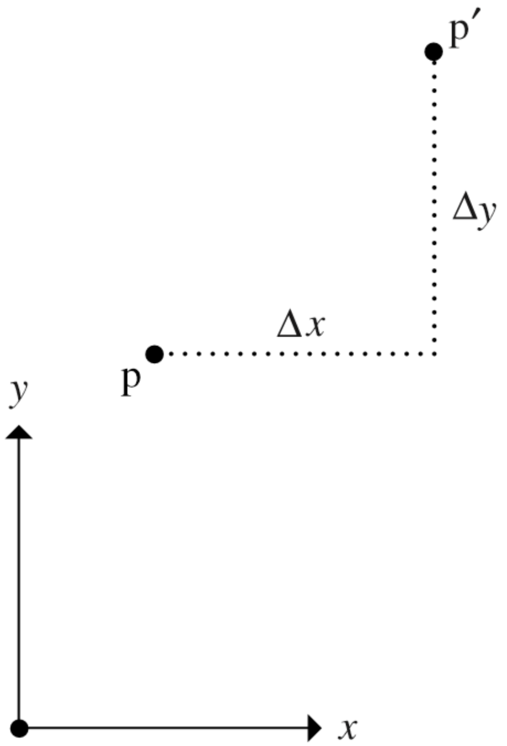
Coordinate transformation

- **Identity transformation**
  - This transformation is represented by the identity matrix
  - It maps each point and each vector to itself

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Coordinate transformation

- **Translation transformation**
  - When applied to a point **p**, it translates **p**'s coordinates
  - Translation only affects points, leaving vectors unchanged



Coordinate transformation

- **Translation transformation**
  - In homogeneous matrix form, the translation transformation is

$$T(\Delta x, \Delta y, \Delta z) = \begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Coordinate transformation

- **Translation transformation**
  - When we consider the operation of a translation matrix on a point

$$\begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x+\Delta x \\ y+\Delta y \\ z+\Delta z \\ 1 \end{pmatrix}$$

  - When we consider the operation of a translation matrix on a vector: unchanged as expected

$$\begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix}$$

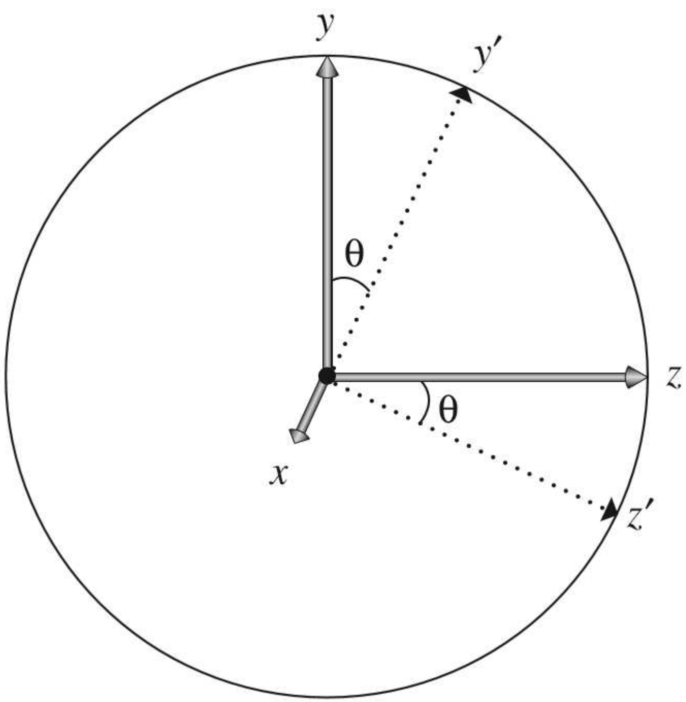
Coordinate transformation

- **Scaling transformation**
  - Take a point or vector and multiply its components by scale factors in x, y, and z
  - Differentiate between uniform scaling and non-uniform scaling

$$S(x, y, z) = \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Coordinate transformation

- **Rotation transformation**
  - Rotation about x-coordinate
    - Rotation by an angle θ about the x axis leaves the x coordinate unchanged



$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Positive angle follows the left-hand-side rule from y to z

Coordinate transformation

- **Rotation transformation**
  - Rotation about y- and z-axes

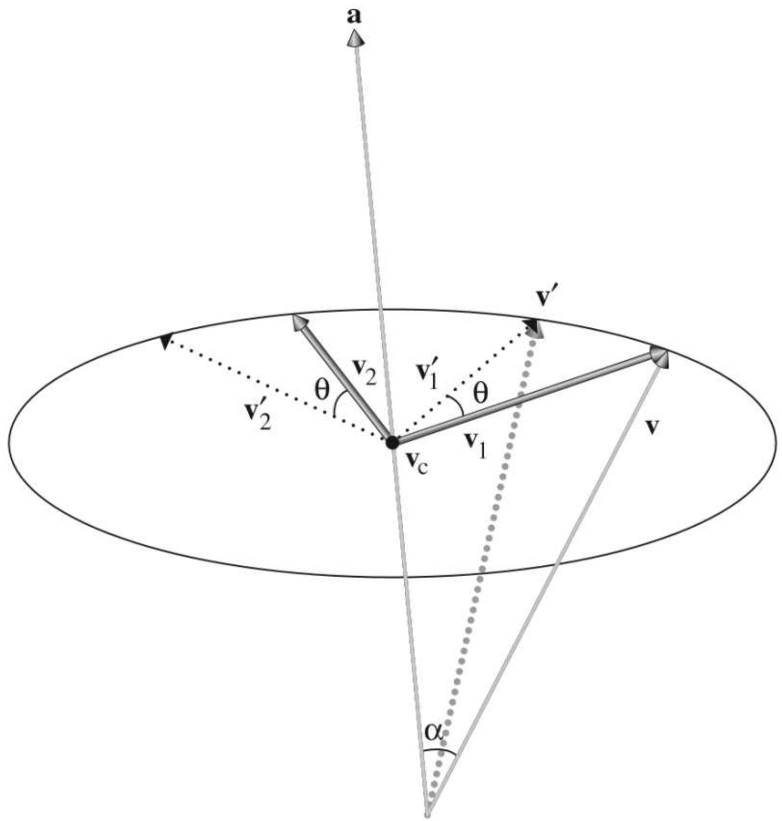
$$\mathrm{R}_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \mathrm{R}_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

  - An arbitrary rotation can be decomposed into rotations about x-, y- and z-axes

$$\mathrm{R}(\theta) = \mathrm{R}_z(\theta)\,\mathrm{R}_y(\theta)\,\mathrm{R}_x(\theta)$$

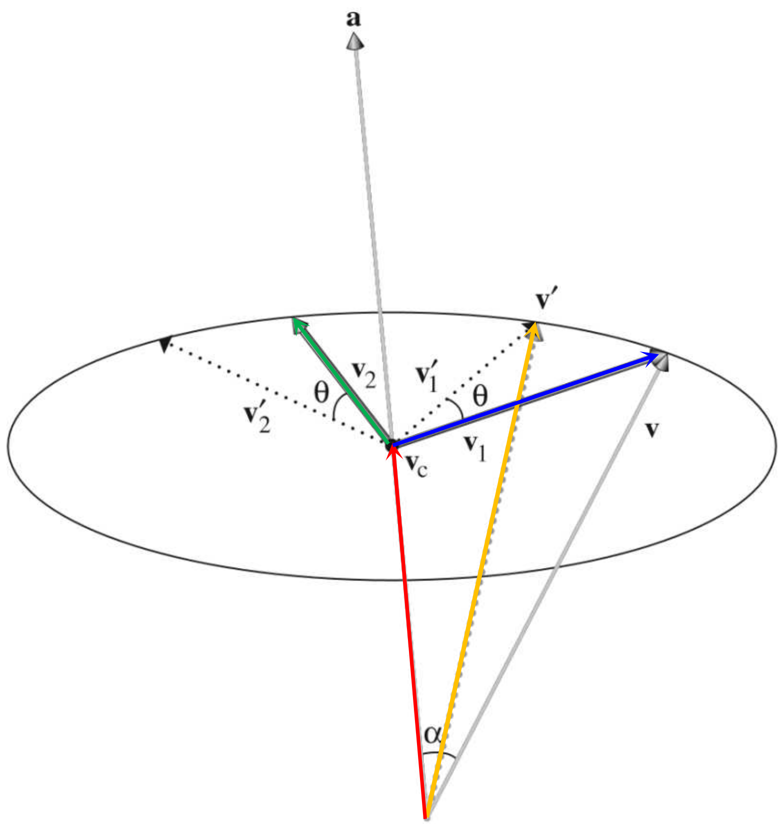
Coordinate transformation

- **Rotation about an arbitrary axis**
  – Consider a normalized direction vector **a** that gives the axis to rotate around by angle θ and a vector **v** to be rotated, <u>how to calculate the rotated vector **v**′?</u>



Coordinate transformation

- **Rotation about an arbitrary axis**
  - How to compute efficiently?



Project **v** onto **a**

$$\mathbf{v_c} = \mathbf{a}\,\|\mathbf{v}\|\cos\alpha = \mathbf{a}(\mathbf{v}\cdot\mathbf{a})$$

Compute basis $\mathbf{v_1}$

$$\mathbf{v_1} = \mathbf{v} - \mathbf{v_c}$$

Compute basis $\mathbf{v_2}$

$$\mathbf{v_2} = (\mathbf{v_1} \times \mathbf{a})$$

Use planar rotation formula

$$\mathbf{v}' = \mathbf{v_c} + \mathbf{v_1}\cos\theta + \mathbf{v_2}\sin\theta$$

Coordinate transformation

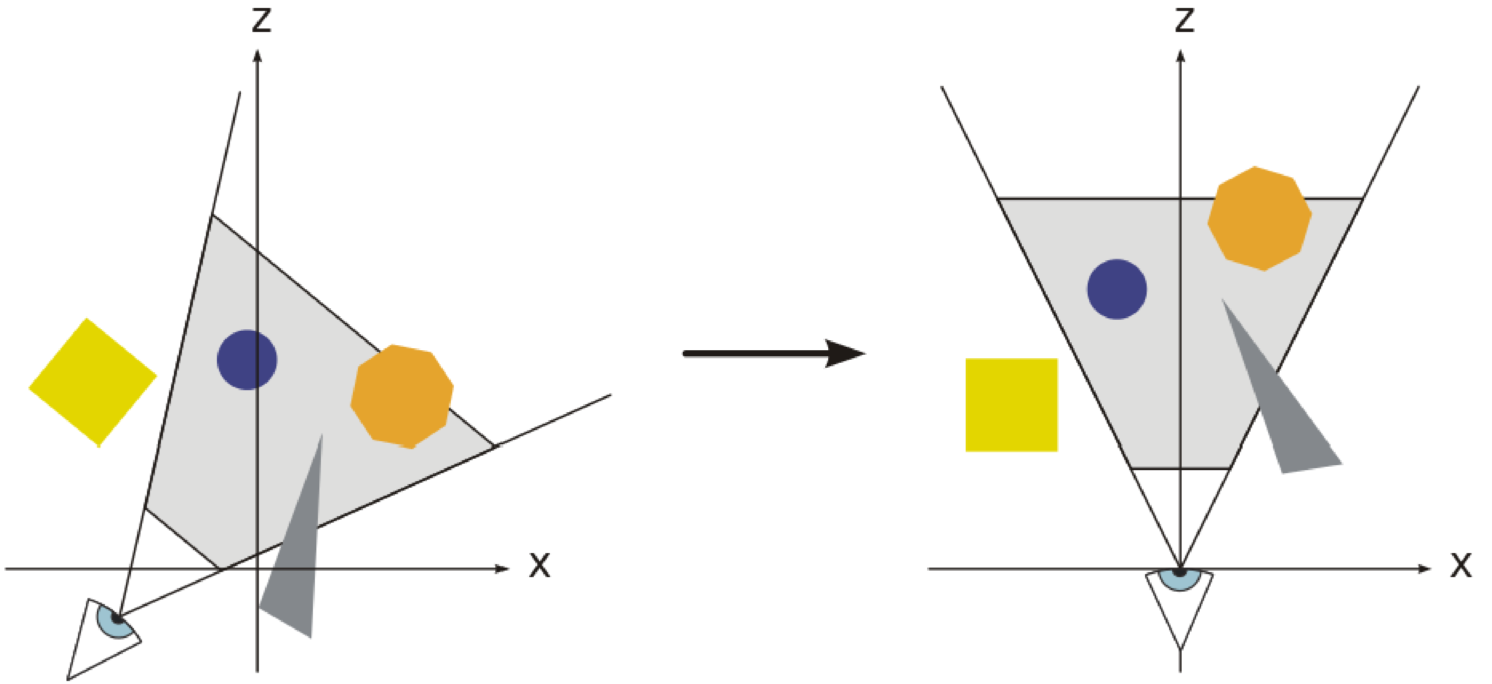
- **Object transformation**
  - Can be decomposed into a series of translations, rotations and scalings
  - All these transformations are ordered series, and based on the previous transformation results
  - For example

$$\mathbf{M} = \ldots\ \mathbf{S}_4\mathbf{T}_3\mathbf{R}_3\mathbf{S}_2\mathbf{T}_2\mathbf{S}_1\ \mathbf{R}_2\mathbf{R}_1\mathbf{T}_1$$

3. View transformation

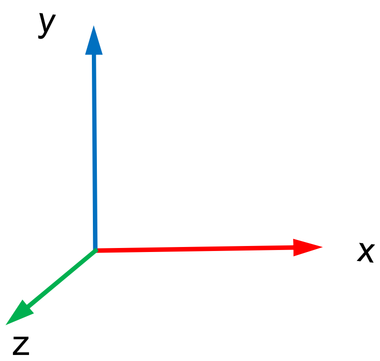
View transformation

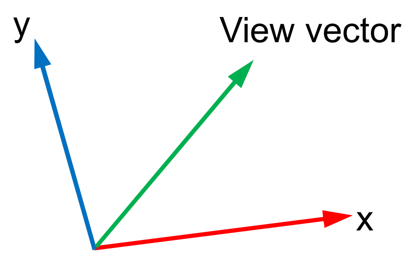
- **What is a view transformation?**
  - Transform the world coordinates into the view (camera/eye) coordinates



View transformation

- **How to compute the view transform?**
  - Translation + rotation from world coordinate system
  - World coordinate system forms an identity matrix
  - Thus, view matrix is formed by camera coordinate system + camera translation in world coordinates
- **View transformation matrix**



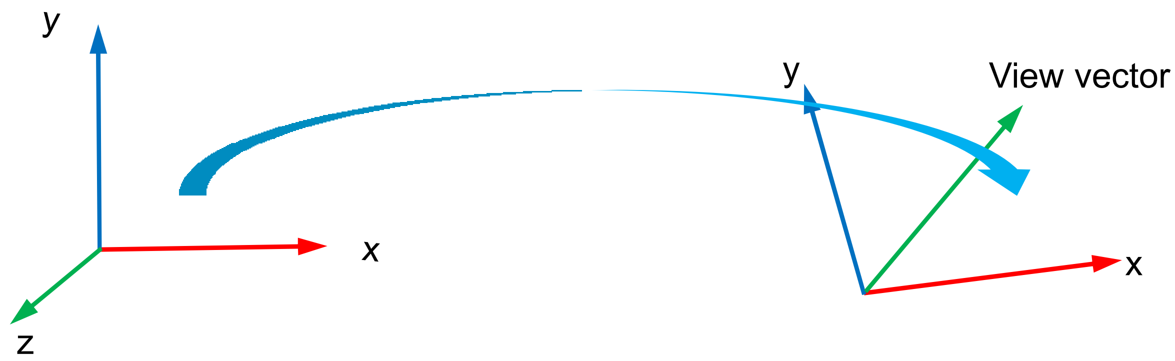


$$I = RTB_v$$

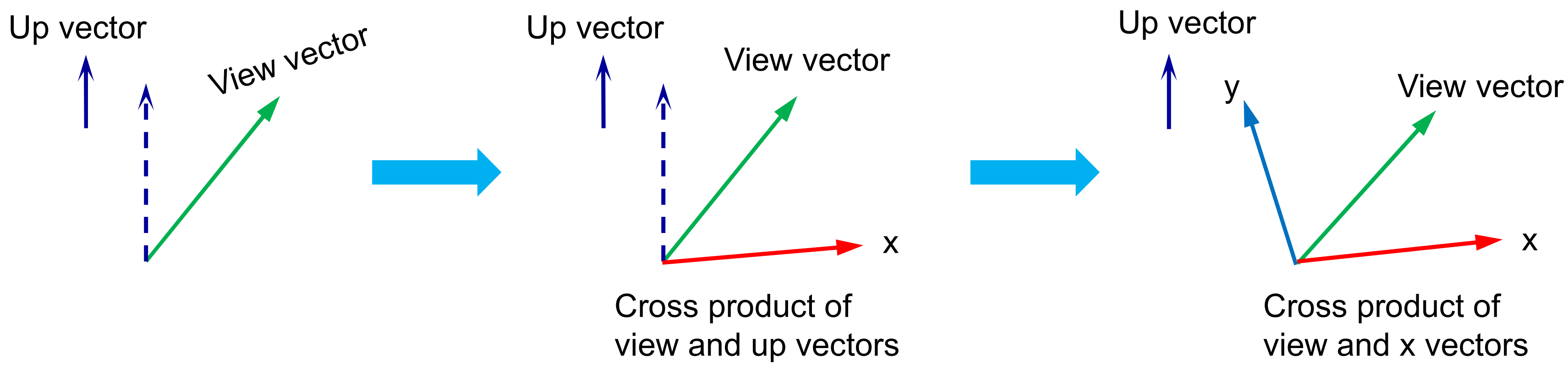
$$B_v = T^{-1}R^T$$

View transformation

- **How to compute the view matrix?**
  - The Gram-Schmidt orthogonalization process



Translation + Rotation



Model-view transformation

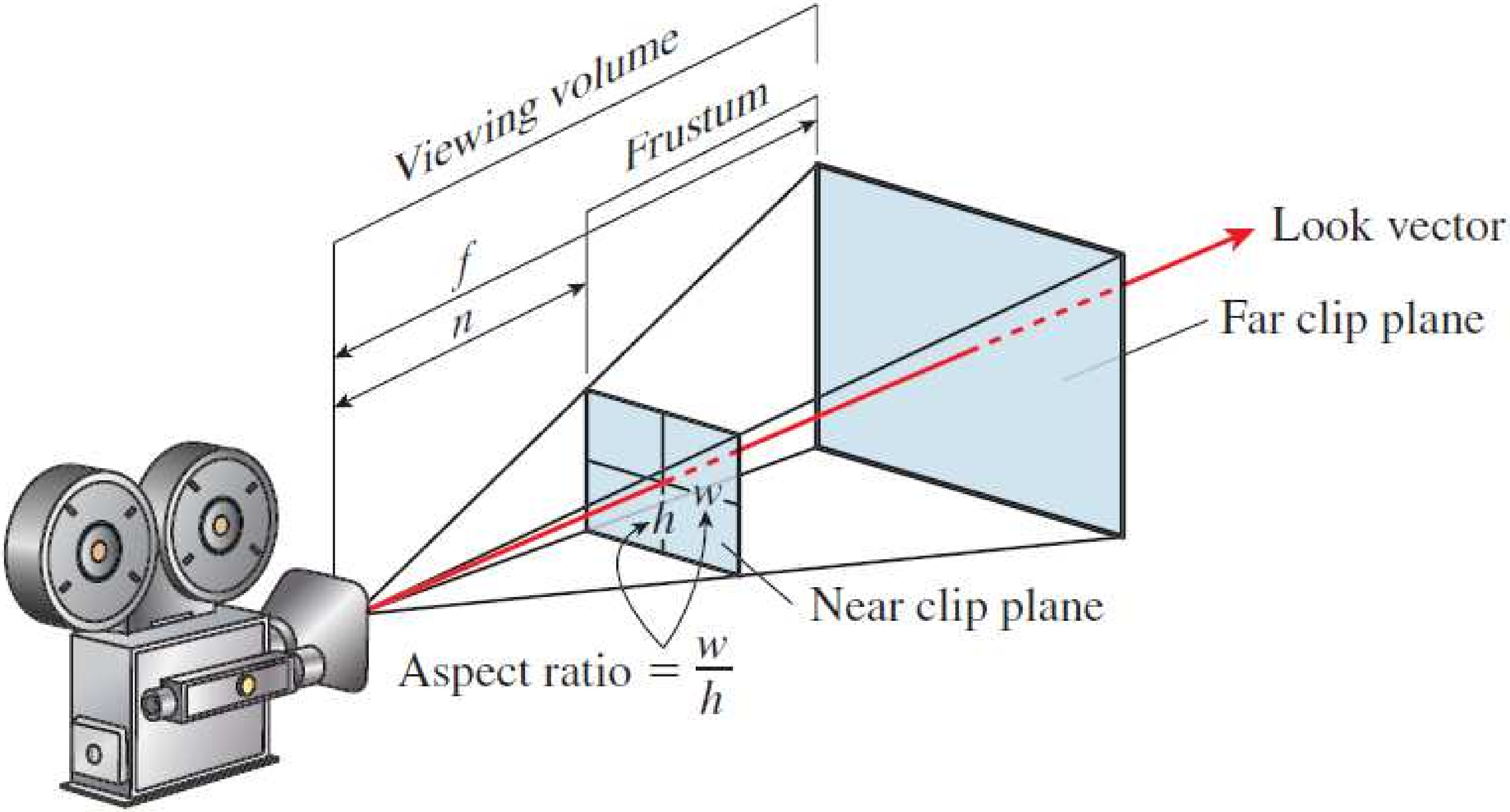
- **In practice, we will combine the model transformation and view transformation**
  - Model transformation: determine the final coordinates in world coordinate system
  - View transformation: transform the final world coordinates to view (camera) coordinates

  - Computation:
    - $\mathbf{M}=\mathbf{M}_{\mathrm{view}}\,\mathbf{M}_{\mathrm{model}}=\mathbf{M}_{\mathrm{view}}\,(\ldots\mathbf{S}_{\mathrm{model}}\,\mathbf{R}_{\mathrm{model}}\,\mathbf{T}_{\mathrm{model}})$

3. Projection

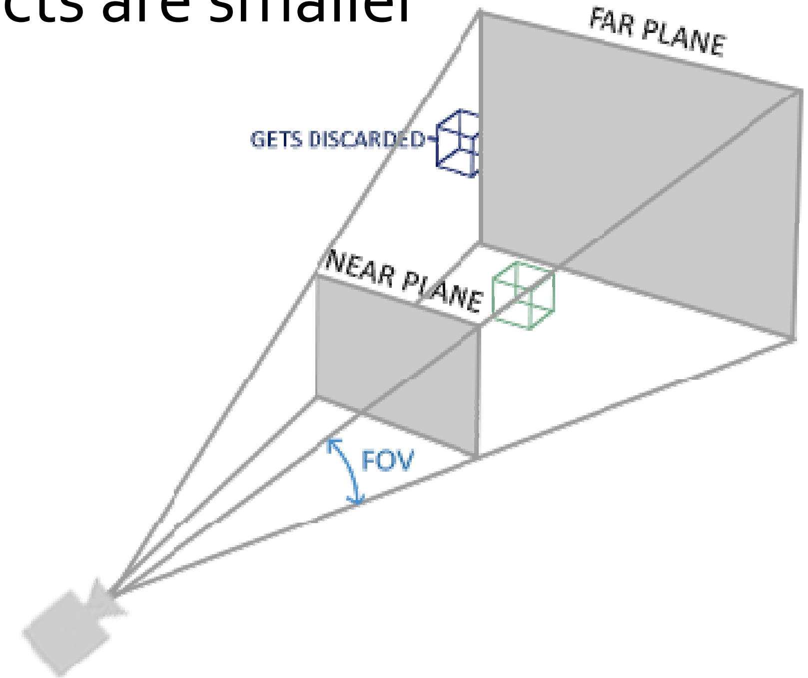
A perspective camera

- **Specify a perspective camera system**



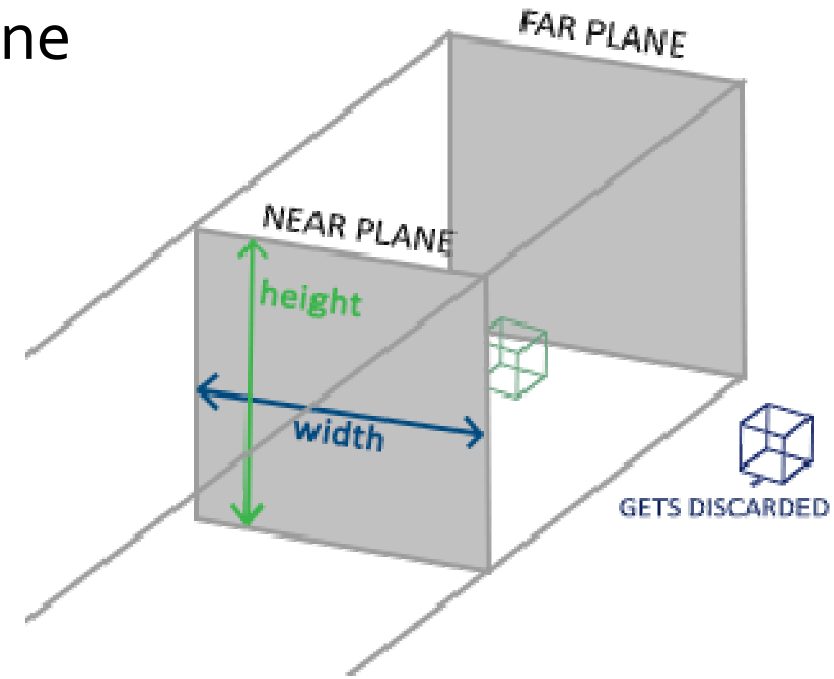
Perspective projection

- **Clipping & projection**
  - A large *frustum* that defines the clipping space
  - All the coordinates inside this frustum is projected along perspective projection line to the projection plane
  - Farther objects are smaller



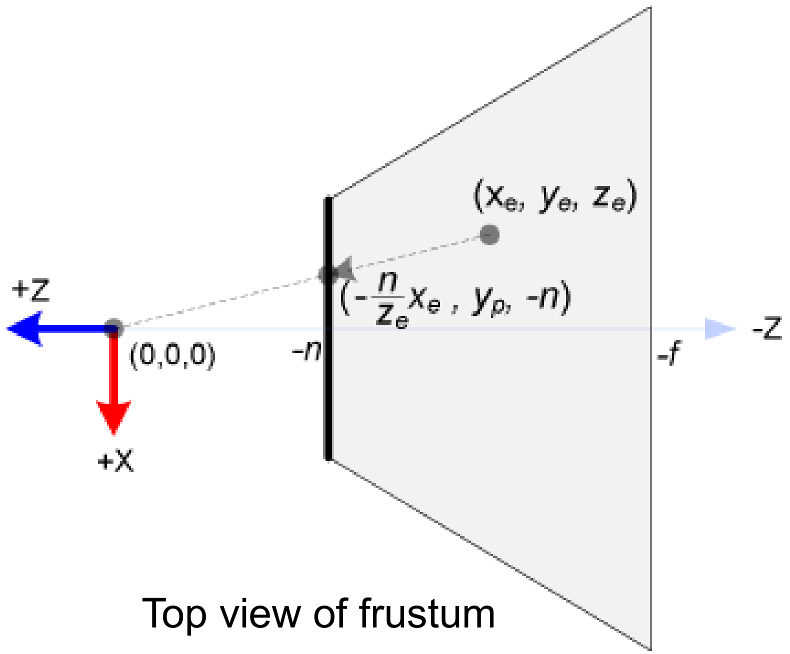
Orthogonal projection

- **Clipping & projection**
  - A cube-like *frustum* that defines the clipping space
  - All the coordinates inside this frustum is projected along the parallel lines to the projection plane
  - Object sizes do not depend on the distance to the projection plane

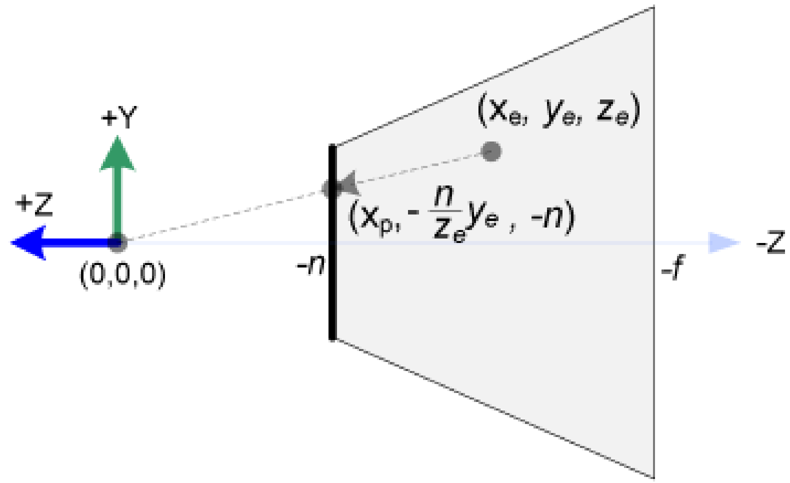


Constructing perspective projection

- **A 3D point in eye space is projected onto the *near* plane (projection plane)**



Top view of frustum



Side view of frustum

$$x_p = \frac{-n \cdot x_e}{z_e} = \frac{n \cdot x_e}{-z_e}$$

$$y_p = \frac{-n \cdot y_e}{z_e} = \frac{n \cdot y_e}{-z_e}$$

Perspective projection representation

- **Look at the perspective projection again**

$$x_p = \frac{-n \cdot x_e}{z_e} = \frac{n \cdot x_e}{-z_e} \qquad y_p = \frac{-n \cdot y_e}{z_e} = \frac{n \cdot y_e}{-z_e}$$

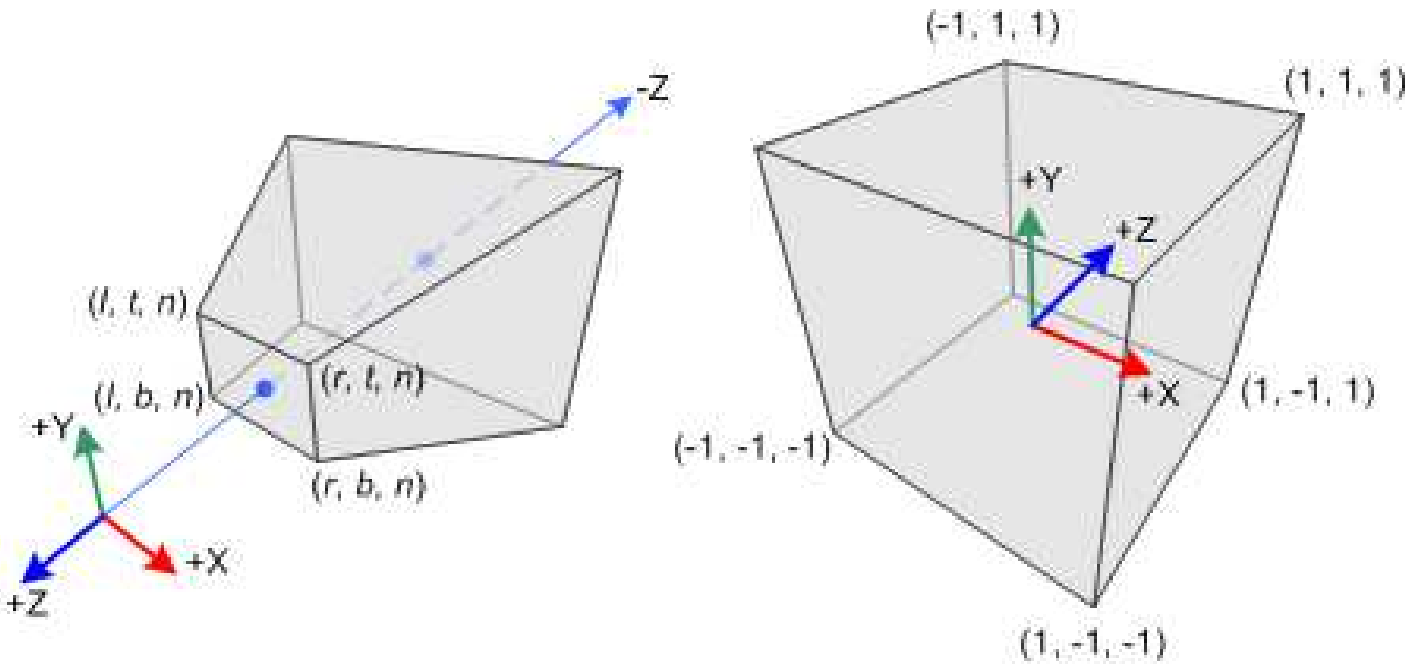
- **Represented as homogeneous coordinates**

$$\begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{pmatrix} = M_{projection} \cdot \begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix} \qquad \begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \end{pmatrix} = \begin{pmatrix} x_{clip}/w_{clip} \\ y_{clip}/w_{clip} \\ z_{clip}/w_{clip} \end{pmatrix}$$

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix}, \qquad \therefore w_c = -z_e$$

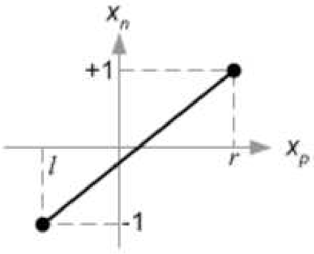
Perspective projection representation

- **Normalized device coordinate (NDC)**
  - Range normalization
    - x-coordinate: [l, r] to [-1, 1]
    - y-coordinate: [b, t] to [-1, 1]
    - z-coordinate: [n, f] to [-1, 1]



Perspective projection representation

- **Mapping to normalized device coordinates**



Mapping from x_p to x_n

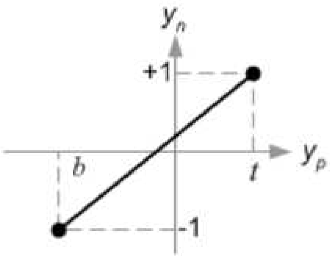
$$x_n = \frac{1-(-1)}{r-l} \cdot x_p + \beta$$

$$1 = \frac{2r}{r-l} + \beta \qquad \text{(substitute } (r,1) \text{ for } (x_p, x_n))$$

$$\beta = 1 - \frac{2r}{r-l} = \frac{r-l}{r-l} - \frac{2r}{r-l}$$

$$= \frac{r-l-2r}{r-l} = \frac{-r-l}{r-l} = -\frac{r+l}{r-l}$$

$$\therefore x_n = \frac{2x_p}{r-l} - \frac{r+l}{r-l}$$



Mapping from y_p to y_n

$$y_n = \frac{1-(-1)}{t-b} \cdot y_p + \beta$$

$$1 = \frac{2t}{t-b} + \beta \qquad \text{(substitute } (t,1) \text{ for } (y_p, y_n))$$

$$\beta = 1 - \frac{2t}{t-b} = \frac{t-b}{t-b} - \frac{2t}{t-b}$$

$$= \frac{t-b-2t}{t-b} = \frac{-t-b}{t-b} = -\frac{t+b}{t-b}$$

$$\therefore y_n = \frac{2y_p}{t-b} - \frac{t+b}{t-b}$$

Perspective projection representation

- **Substitute x_p and y_p with eye space coordinates**

$$
\begin{aligned}
x_n &= \frac{2x_p}{r-l} - \frac{r+l}{r-l} \qquad (x_p = \frac{nx_e}{-z_e}) \\
&= \frac{2 \cdot \frac{n \cdot x_e}{-z_e}}{r-l} - \frac{r+l}{r-l} \\
&= \frac{2n \cdot x_e}{(r-l)(-z_e)} - \frac{r+l}{r-l} \\
&= \frac{\frac{2n}{r-l} \cdot x_e}{-z_e} - \frac{r+l}{r-l} \\
&= \frac{\frac{2n}{r-l} \cdot x_e}{-z_e} + \frac{\frac{r+l}{r-l} \cdot z_e}{-z_e} \\
&= \Big(\underbrace{\frac{2n}{r-l} \cdot x_e + \frac{r+l}{r-l} \cdot z_e}_{x_c} \Big) \Big/ - z_e
\end{aligned}
$$

$$
\begin{aligned}
y_n &= \frac{2y_p}{t-b} - \frac{t+b}{t-b} \qquad (y_p = \frac{ny_e}{-z_e}) \\
&= \frac{2 \cdot \frac{n \cdot y_e}{-z_e}}{t-b} - \frac{t+b}{t-b} \\
&= \frac{2n \cdot y_e}{(t-b)(-z_e)} - \frac{t+b}{t-b} \\
&= \frac{\frac{2n}{t-b} \cdot y_e}{-z_e} - \frac{t+b}{t-b} \\
&= \frac{\frac{2n}{t-b} \cdot y_e}{-z_e} + \frac{\frac{t+b}{t-b} \cdot z_e}{-z_e} \\
&= \Big(\underbrace{\frac{2n}{t-b} \cdot y_e + \frac{t+b}{t-b} \cdot z_e}_{y_c} \Big) \Big/ - z_e
\end{aligned}
$$

Perspective projection representation

- **The projection matrix becomes**

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix}$$

- **Finding z_n is a little different from others**
  - z_e in eye space is always projected to -n on the near plane

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix}, \qquad z_n = z_c / w_c = \frac{Az_e + Bw_e}{-z_e}$$

Perspective projection representation

- **Establishing relations for A and B**
  - In eye space

$$z_n = \frac{Az_e + B}{-z_e}$$

  - To find the coefficients, *A* and *B*, we use the (z_e, z_n) relation: (-n, -1) and (-f, 1)

$$\begin{cases} \dfrac{-An + B}{n} = -1 \\[2ex] \dfrac{-Af + B}{f} = 1 \end{cases} \rightarrow \begin{cases} -An + B = -n & (1) \\ -Af + B = f & (2) \end{cases}$$

Perspective projection representation

- **Final projection matrix**
  - Perspective projection for a projection frustum
  - http://www.songho.ca/opengl/gl_projectionmatrix.html

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

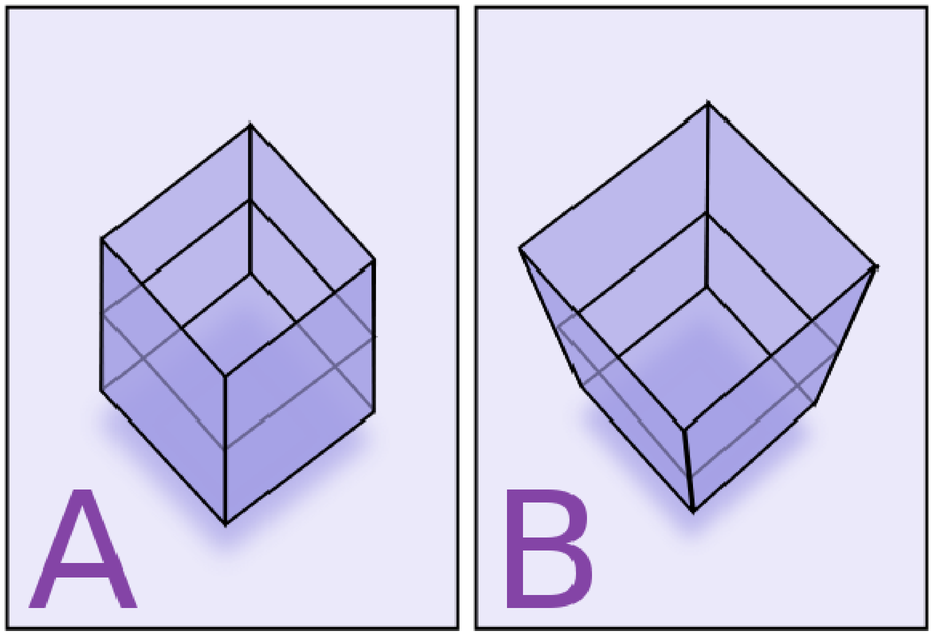
Orthogonal projection representation

- **Similarly, we can obtain the homogeneous representation for orthogonal projection**
  - http://www.songho.ca/opengl/gl_projectionmatrix.html

$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

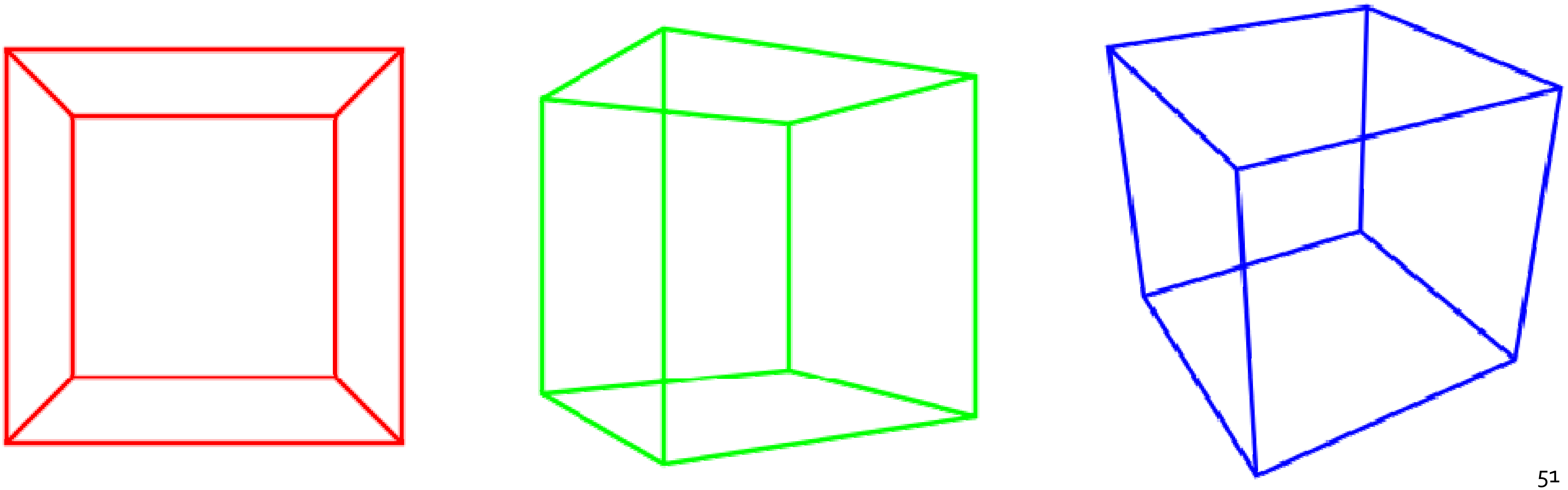
Foreshortening

- **The visual effect or optical illusion from perspective projection**
  - Cause an object or distance to appear shorter than it actually is



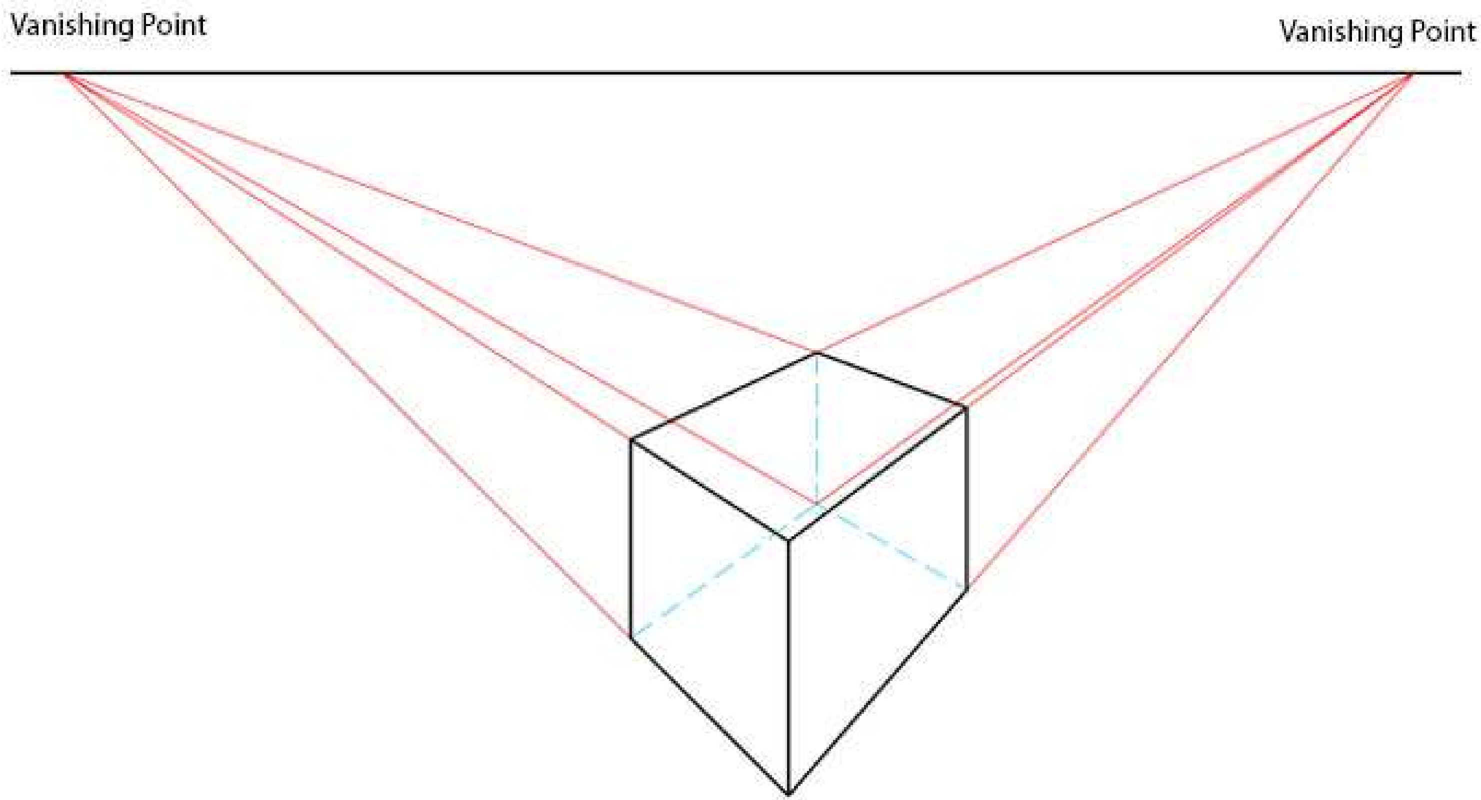
Vanishing points

- **Vanishing points**
  - An abstract point on the image plane
  - 2D projections of a set of parallel lines in 3D space appear to converge
- **One- , two- & three-point perspective**



Vanishing points

- **An example of two-point perspective**



4. Transformations in OpenGL

Transformations in OpenGL

- **Select transformation matrix**
  - Select model-view matrix in OpenGL

```
glMatrixMode(GL_MODELVIEW);
```

  - Select projection matrix in OpenGL

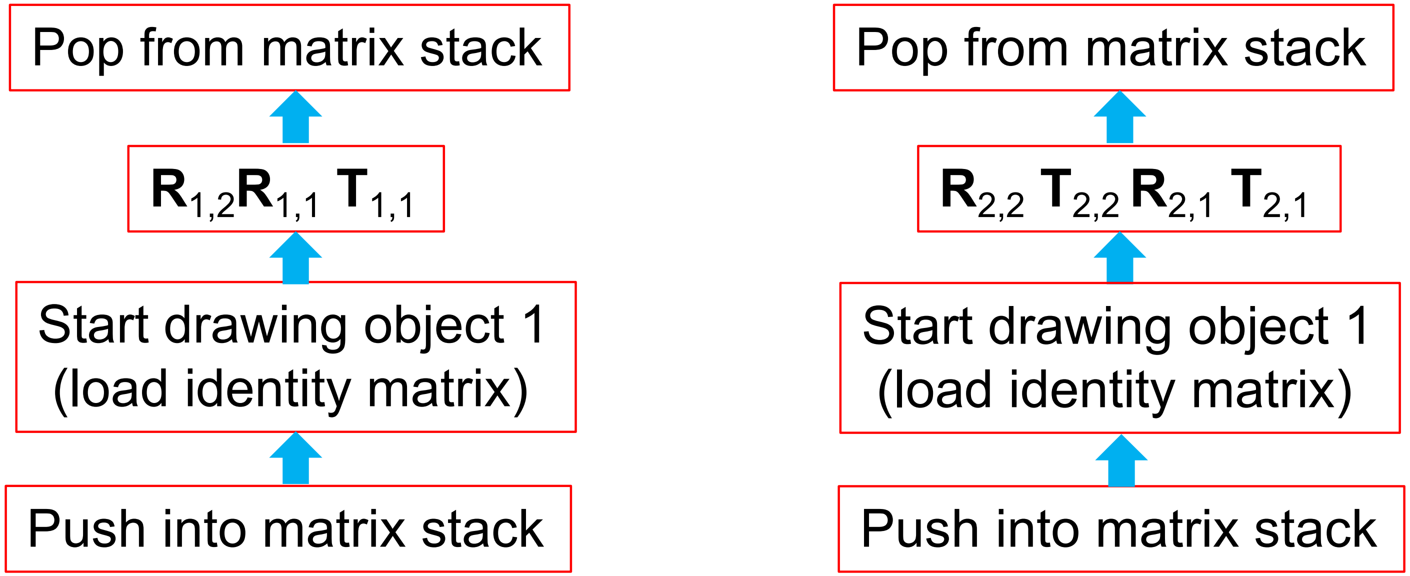
```
glMatrixMode(GL_PROJECTION);
```

Transformations in OpenGL

- **Object transformations**
  - Initial setting: model-view matrix is an identity matrix
  - Translation
    - glTranslatef(): multiply translation matrix to the existing model-view matrix
  - Rotation
    - glRotatef(): multiply rotation matrix to the existing model-view matrix
  - Scaling
    - glScalef(): multiply scaling matrix to the existing model-view matrix

Coordinate transformation in OpenGL

- **Maintaining transformation matrices in a stack**
  - Suppose we want to transform two objects, with different transformations
  - Object 1: $\mathbf{R}_{1,2}\mathbf{R}_{1,1}\ \mathbf{T}_{1,1}$
  - Object 2: $\mathbf{R}_{2,2}\ \mathbf{T}_{2,2}\ \mathbf{R}_{2,1}\ \mathbf{T}_{2,1}$
  - Stack implementation (glPushMatrix/glPopMatrix)



Coordinate transformation in OpenGL

- **Setting up 3D projection in OpenGL**
  - **Orthogonal projection**

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

glOrtho(left,right,bottom,top,zNear,zFar);

glMatrixMode(GL_MODELVIEW);
```

  - **Perspective projection**

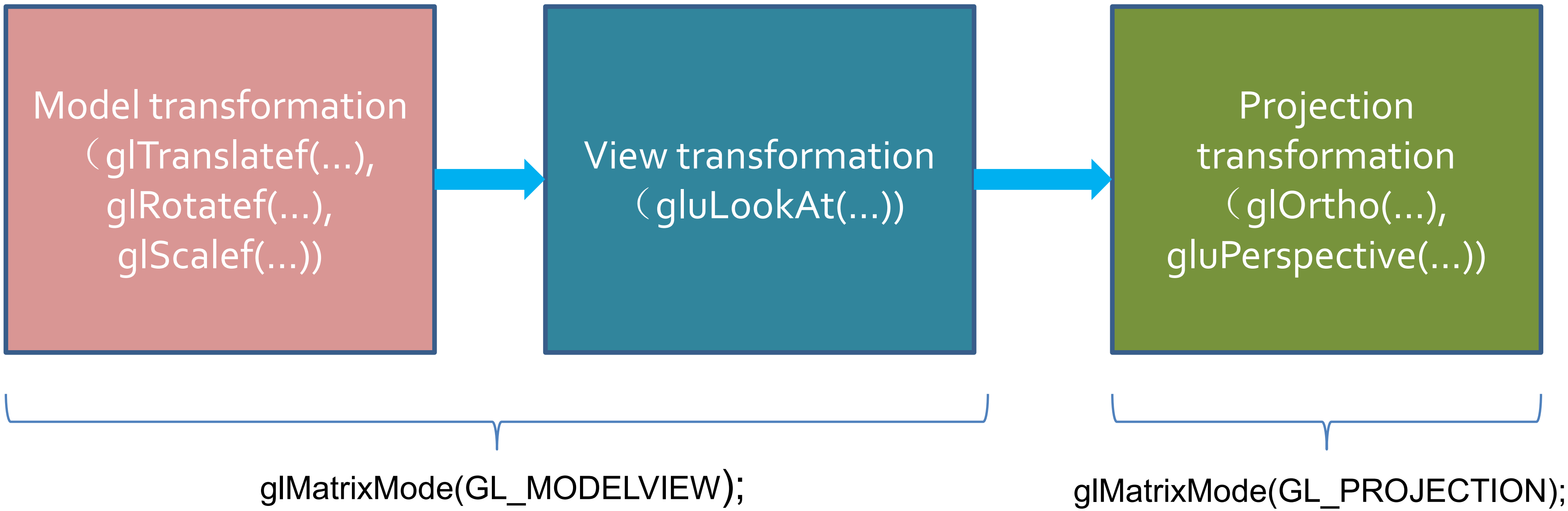
```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

gluPerspective(fovy, aspect, zNear, zFar);

glMatrixMode(GL_MODELVIEW);
```

Coordinate transformation in OpenGL

- **The whole transformation**



Coordinate transformation in OpenGL

- **Customized transformation**
  - You can always multiply your own matrix in OpenGL

```
glMultMatrix (…);
```

    - Provide customized model-view and projection transformations
  - Steps
    - 1. Select corresponding matrix mode

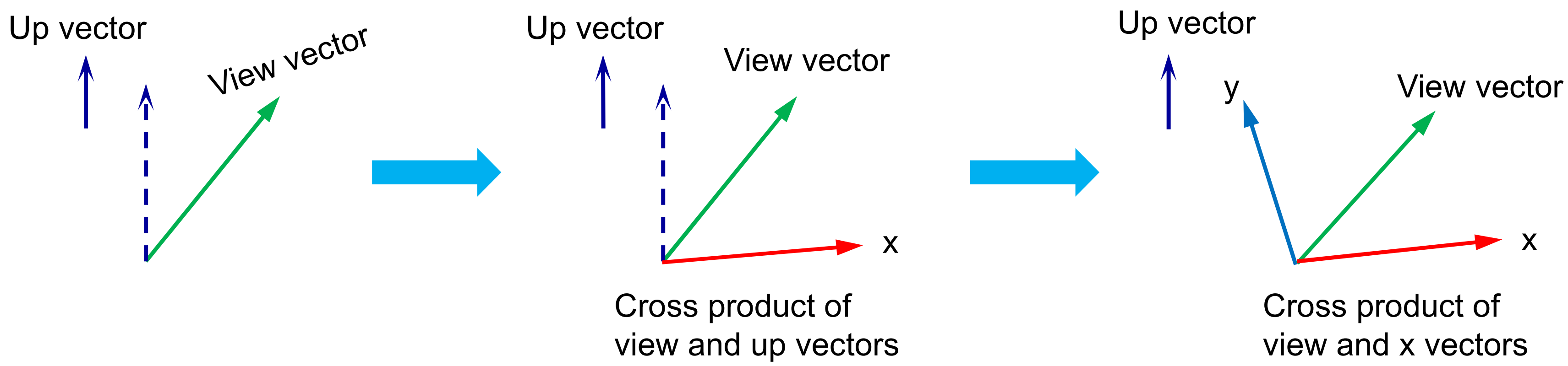
```
glMatrixMode (…);
```

    - 2. Multiply your own transformation matrix

```
glMultMatrix (…);
```

Virtual camera in OpenGL

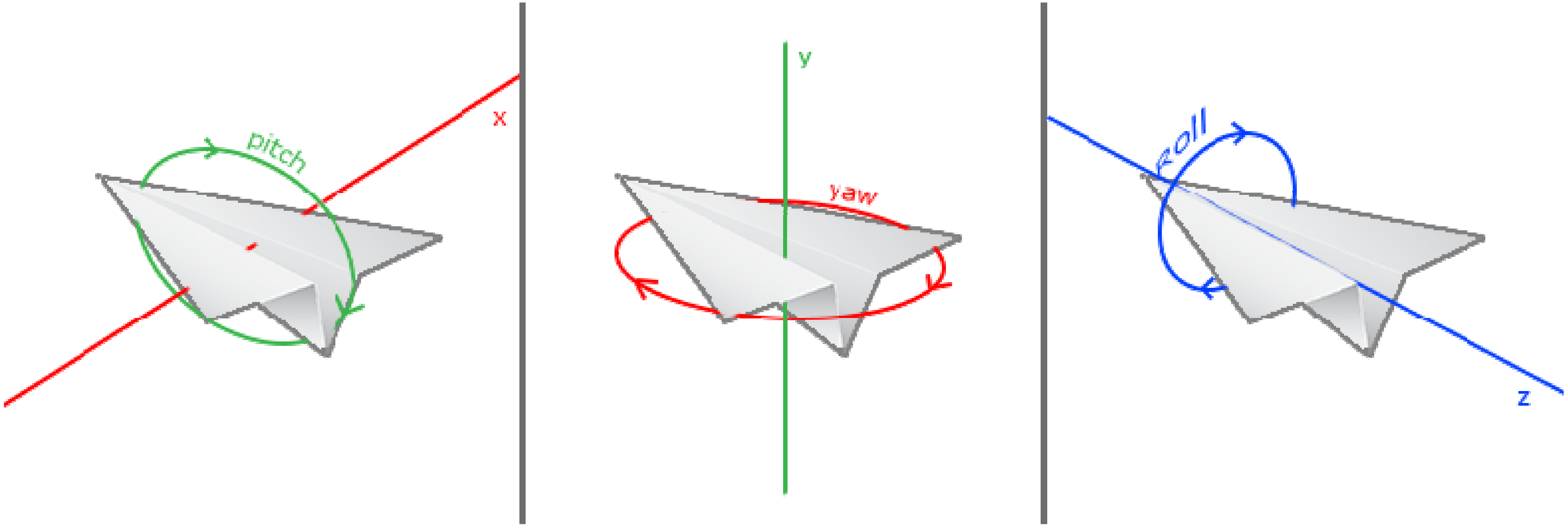
- **Constructing virtual camera**
  - Compute the camera coordinates



  - OpenGL camera function
    - gluLookAt(GLdouble *eyeX*, GLdouble *eyeY*, GLdouble *eyeZ*, GLdouble *centerX*, GLdouble *centerY*, GLdouble *centerZ*, GLdouble *upX*, GLdouble *upY*, GLdouble *upZ*);

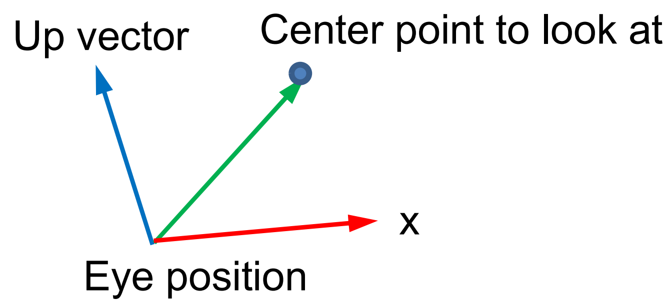
Navigating in virtual world

- **Euler angles**
  – Pitch: rotation around X axis
  – Yaw : rotation around Y axis
  – Roll : rotation around Z axis



Navigating in virtual world

- **Camera translation**
  - Set/translate the eye position
- **Enable "pitch"**
  - Change the center point vertically
- **Enable "roll"**
  - Rotate up vector about the view direction
- **Enable "yaw"**
  - Change the center point horizontally



Vertex shader

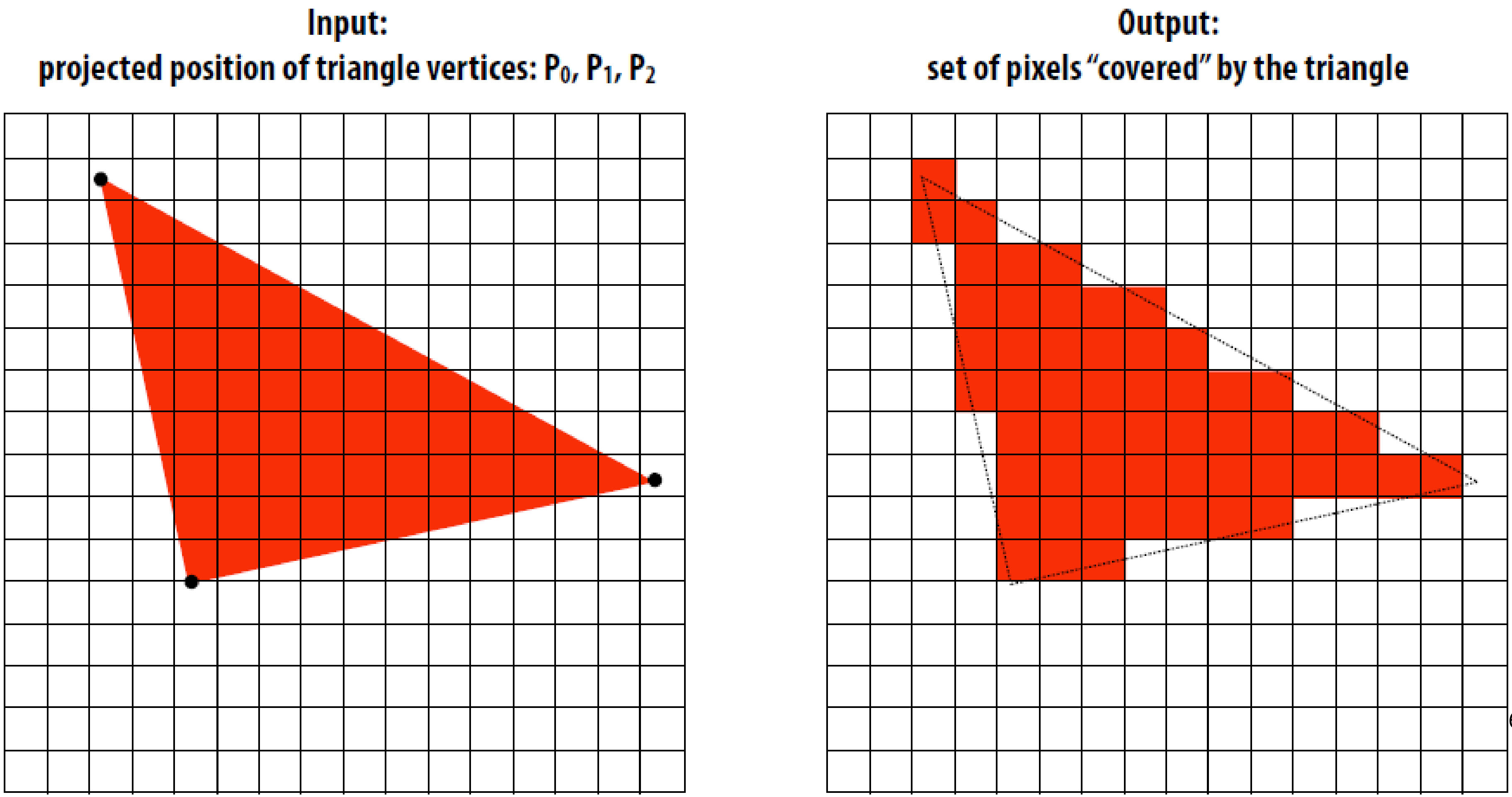
- **Set customized vertex attributes in parallel**
  - vertex position/color/normal/texture coordinates etc.

- **Perform customized transformation and projection**
  - Build-in variables for default transformation/projection
  - Can support customized transformation and projection very freely (even nonlinear)

5. Rasterization

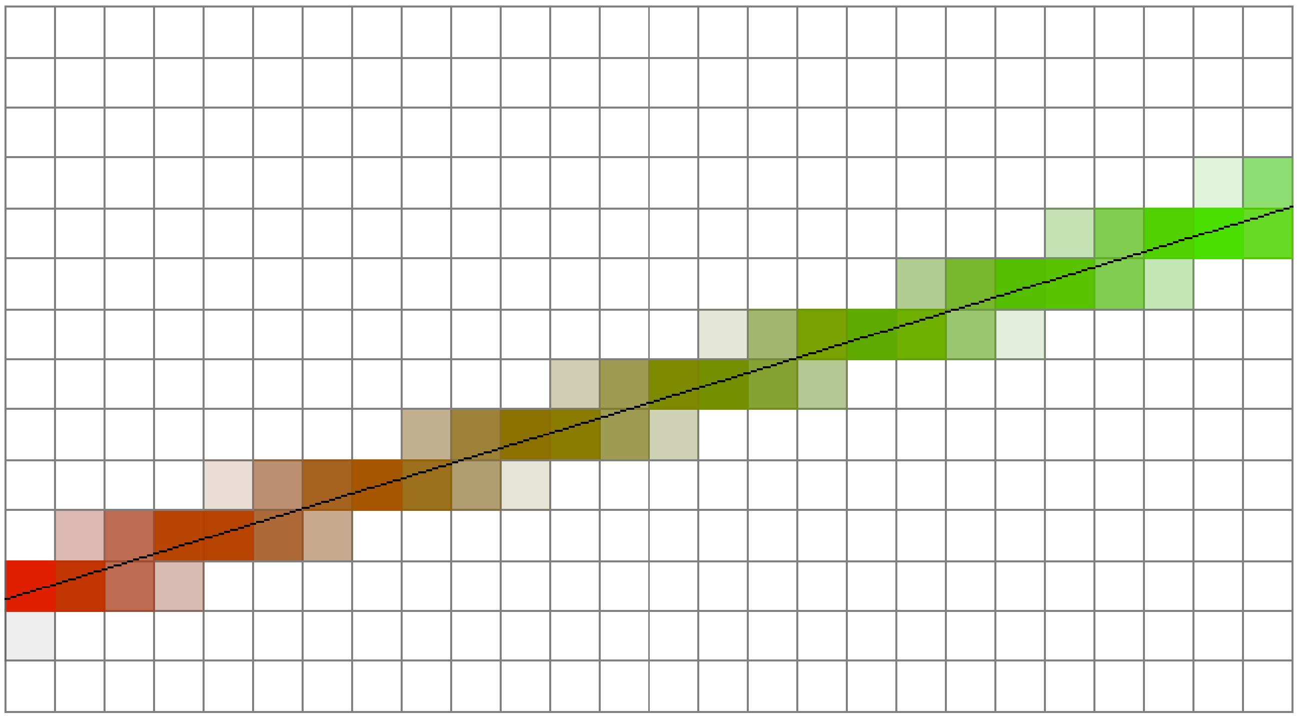
Rasterization

- **Converting continuous representations into discrete pixels (fragments)**



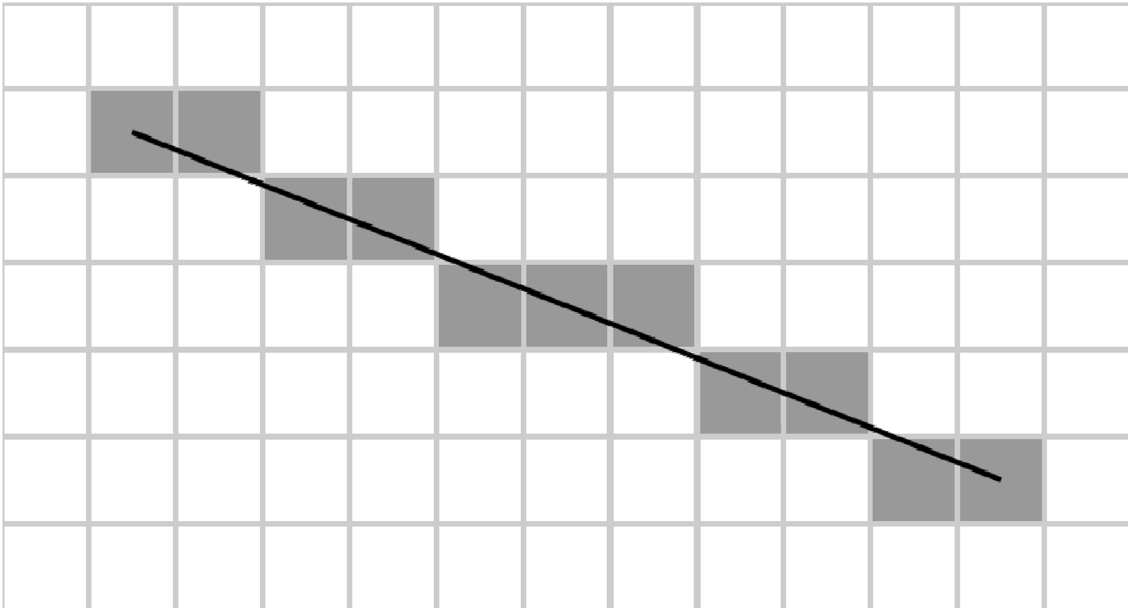
Line rasterization

- **The process of converting continuous lines into the representation by discrete pixels**
  - Determine which pixels are closest to the continuous line
  - Determine the color of the pixels



Line rasterization

- **Bresenham's line algorithm**
  – An algorithm that determines the rasterized points that form a close approximation to a straight line between two end points



Line rasterization

- **Bresenham's line algorithm**
  - Line equation

$$\begin{aligned} y &= mx + b \\ y &= \frac{(\Delta y)}{(\Delta x)}x + b \\ (\Delta x)y &= (\Delta y)x + (\Delta x)b \\ 0 &= (\Delta y)x - (\Delta x)y + (\Delta x)b \end{aligned}$$

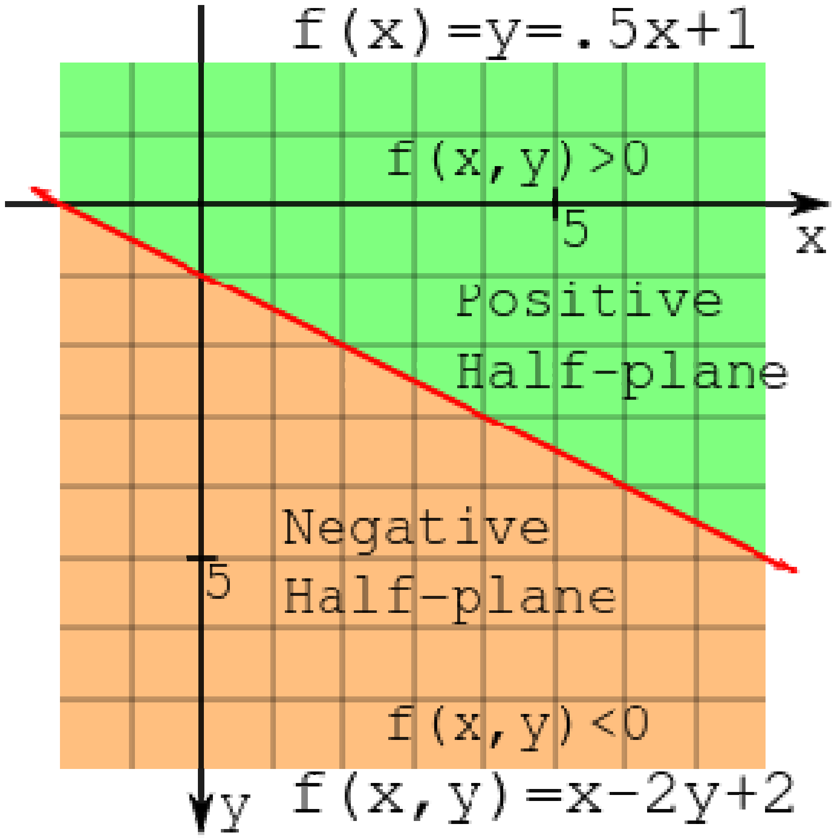
  - Let the last equation be a function of x and y:

$$f(x, y) = 0 = Ax + By + C$$

- $A = \Delta y$
- $B = -\Delta x$
- $C = (\Delta x)b$

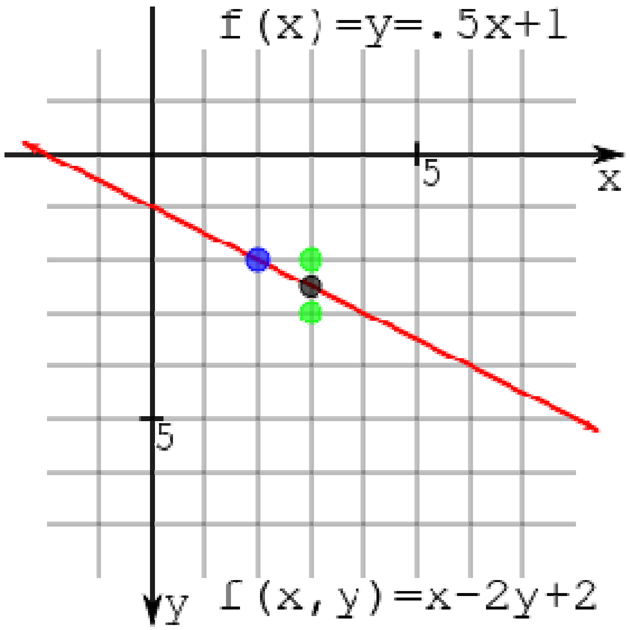
Line rasterization

- **Bresenham's line algorithm**
  – Positive and negative half-planes



Line rasterization

- **Bresenham's line algorithm**
  - Starting from (x_0, y_0), determine the next point to be (x_0+1, y_0) or (x_0+1, y_0+1)
  - Intuition: the point should be chosen based upon which is closer to the line at x_0+1



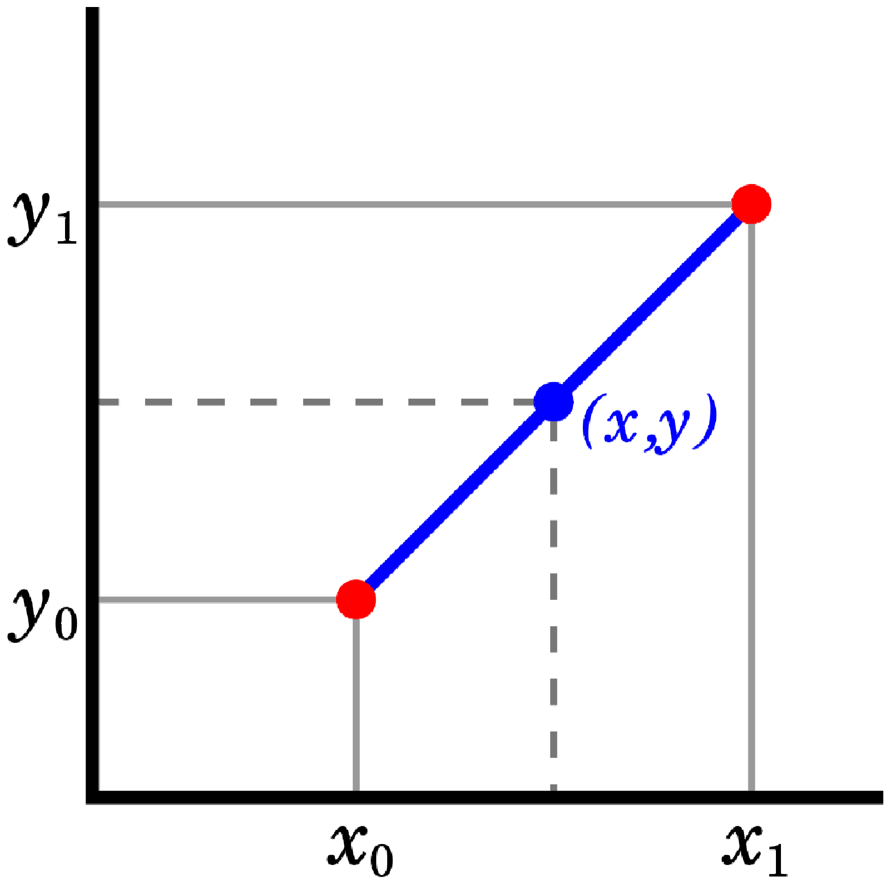
Evaluate the line function at the midpoint

$$f(x_0 + 1, y_0 + 1/2)$$

f<=0: select (x_0+1, y_0)
otherwise
f>0: select (x_0+1, y_0+1)

Line rasterization

- **Color interpolation**
  - Linear interpolation based on x or y value



Point-inside-polygon test

- **Point-in-triangle test**
  - Compute triangle edge equation from projected positions of vertices

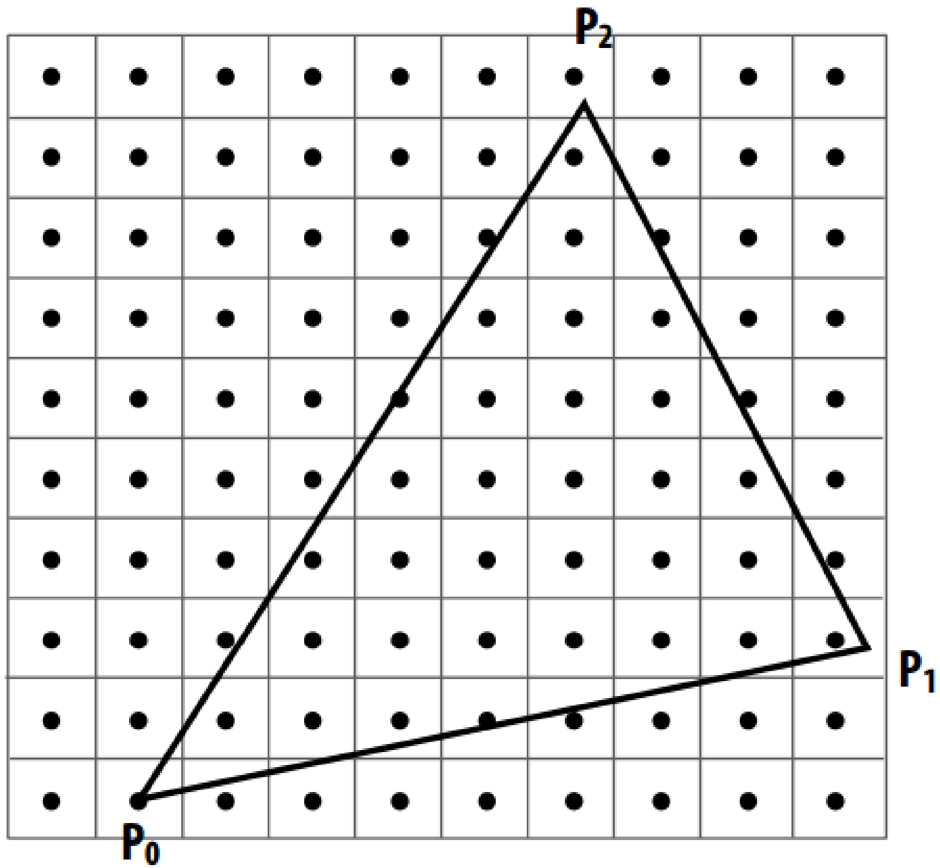
triangle vertices

$$P_i = (X_i, Y_i)$$

$$dX_i = X_{i+1} - X_i$$
$$dY_i = Y_{i+1} - Y_i$$

$$E_i(x, y) = (x - X_i)\, dY_i - (y - Y_i)\, dX_i$$
$$= A_i x + B_i y + C_i$$

$$E_i(x, y) = 0 \text{ : point on edge}$$
$$> 0 \text{ : outside edge}$$
$$< 0 \text{ : inside edge}$$



Point-inside-polygon test

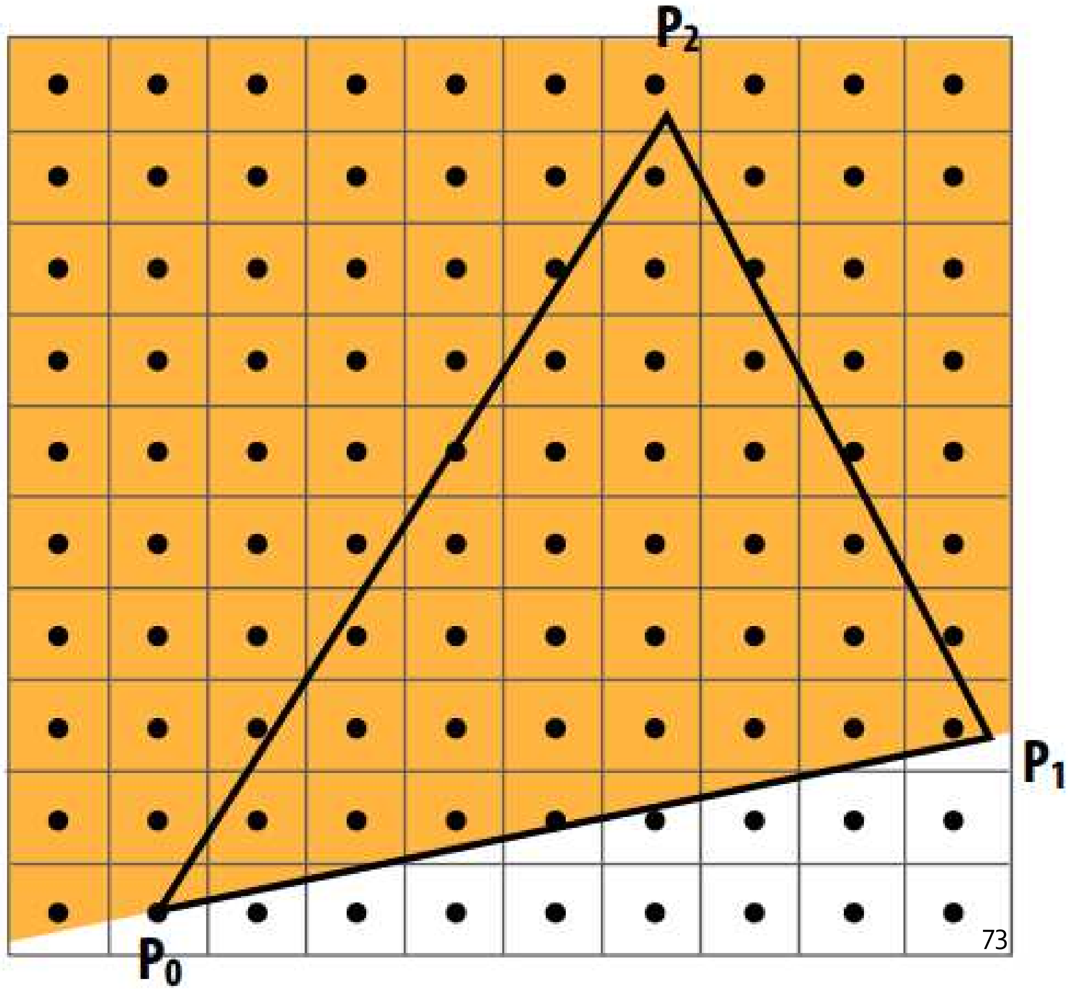
- **Test for whether a point is inside edge P0P1**

$P_i = (X_i, Y_i)$

$dX_i = X_{i+1} - X_i$
$dY_i = Y_{i+1} - Y_i$

$$E_i(x, y) = (x - X_i)\, dY_i - (y - Y_i)\, dX_i$$
$$= A_i x + B_i y + C_i$$

$E_i(x, y) = 0$: point on edge
> 0 : outside edge
< 0 : inside edge



Point-inside-polygon test

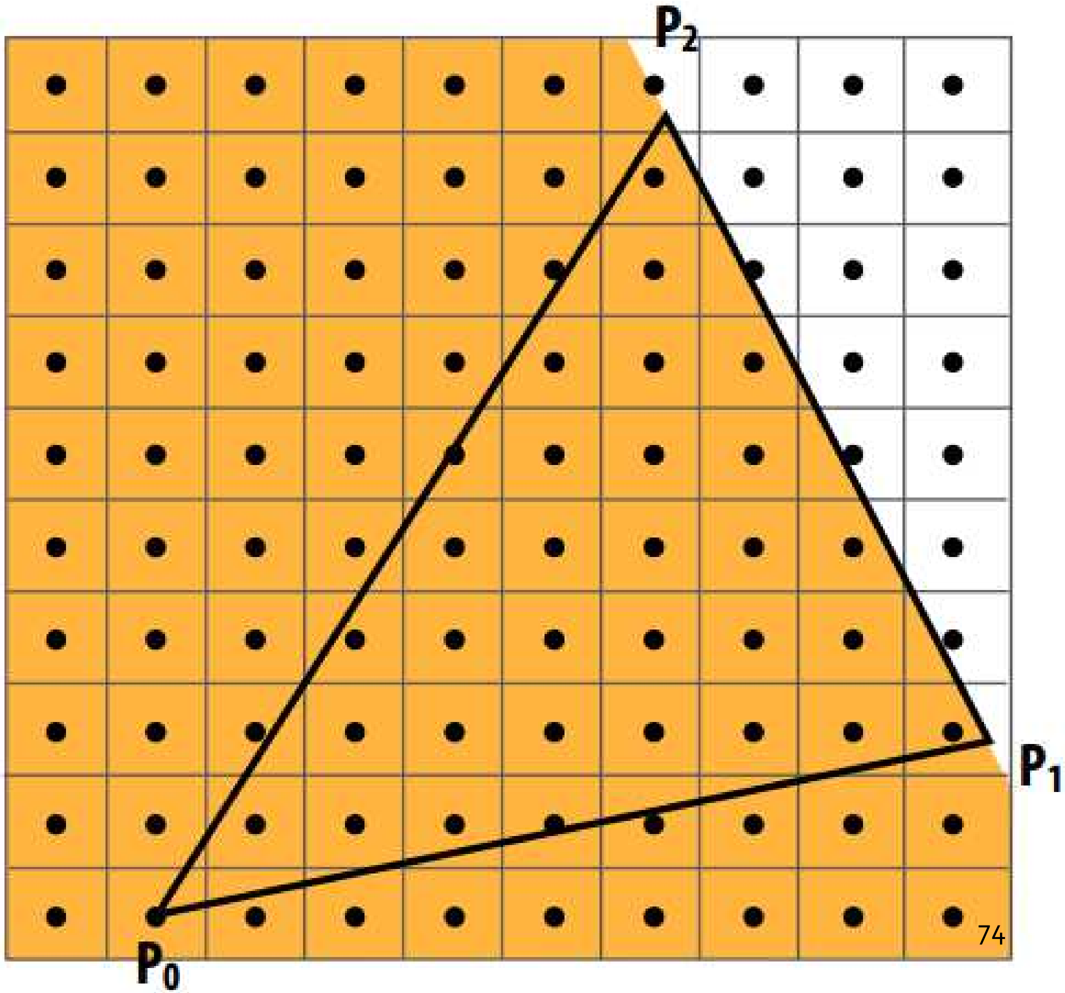
- **Test for whether a point is inside edge P1P2**

$P_i = (X_i, Y_i)$

$dX_i = X_{i+1} - X_i$
$dY_i = Y_{i+1} - Y_i$

$$E_i(x, y) = (x - X_i)\, dY_i - (y - Y_i)\, dX_i$$
$$= A_i x + B_i y + C_i$$

$E_i(x, y) = 0$: point on edge
> 0 : outside edge
< 0 : inside edge



Point-inside-polygon test

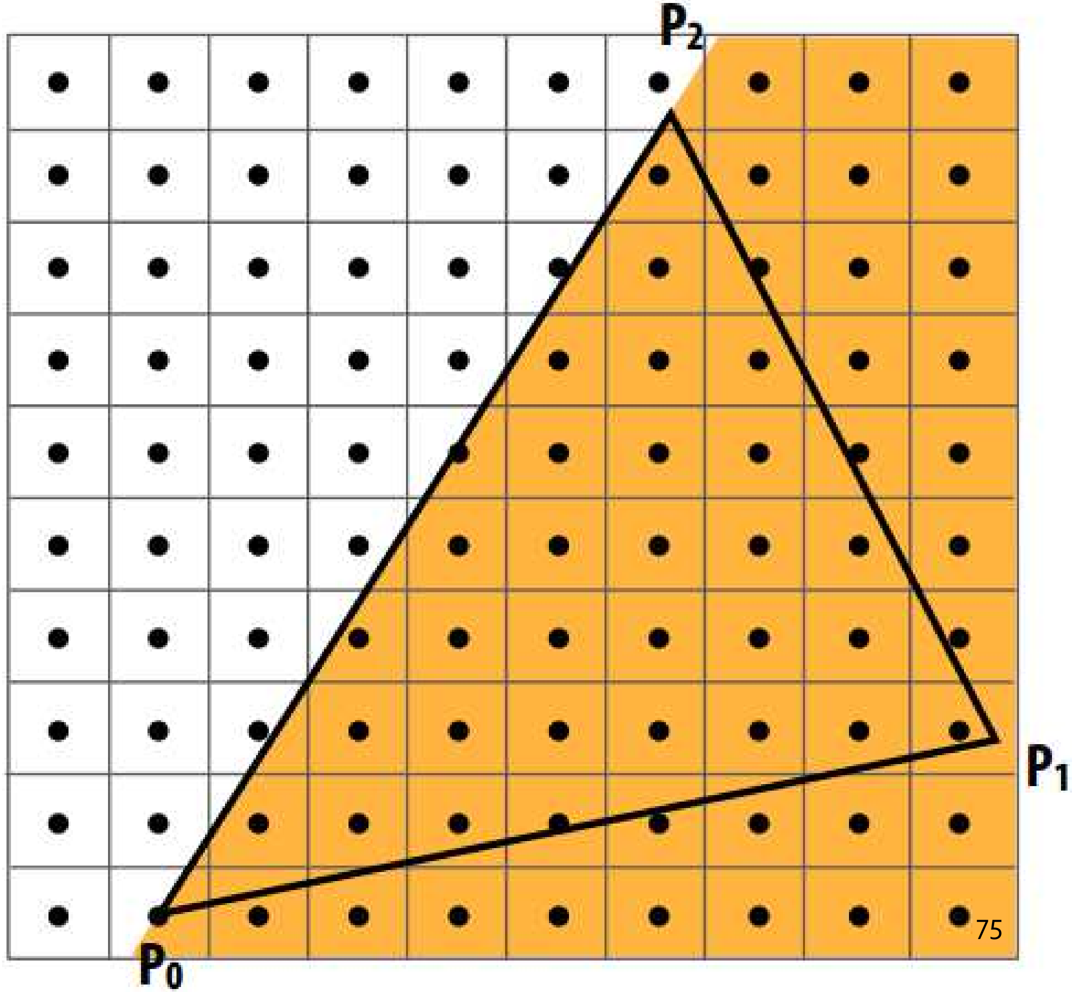
- **Test for whether a point is inside edge P2P0**

$P_i = (X_i, Y_i)$

$dX_i = X_{i+1} - X_i$
$dY_i = Y_{i+1} - Y_i$

$$E_i(x, y) = (x - X_i)\, dY_i - (y - Y_i)\, dX_i$$
$$= A_i x + B_i y + C_i$$

$E_i(x, y) = 0$: point on edge
> 0 : outside edge
< 0 : inside edge

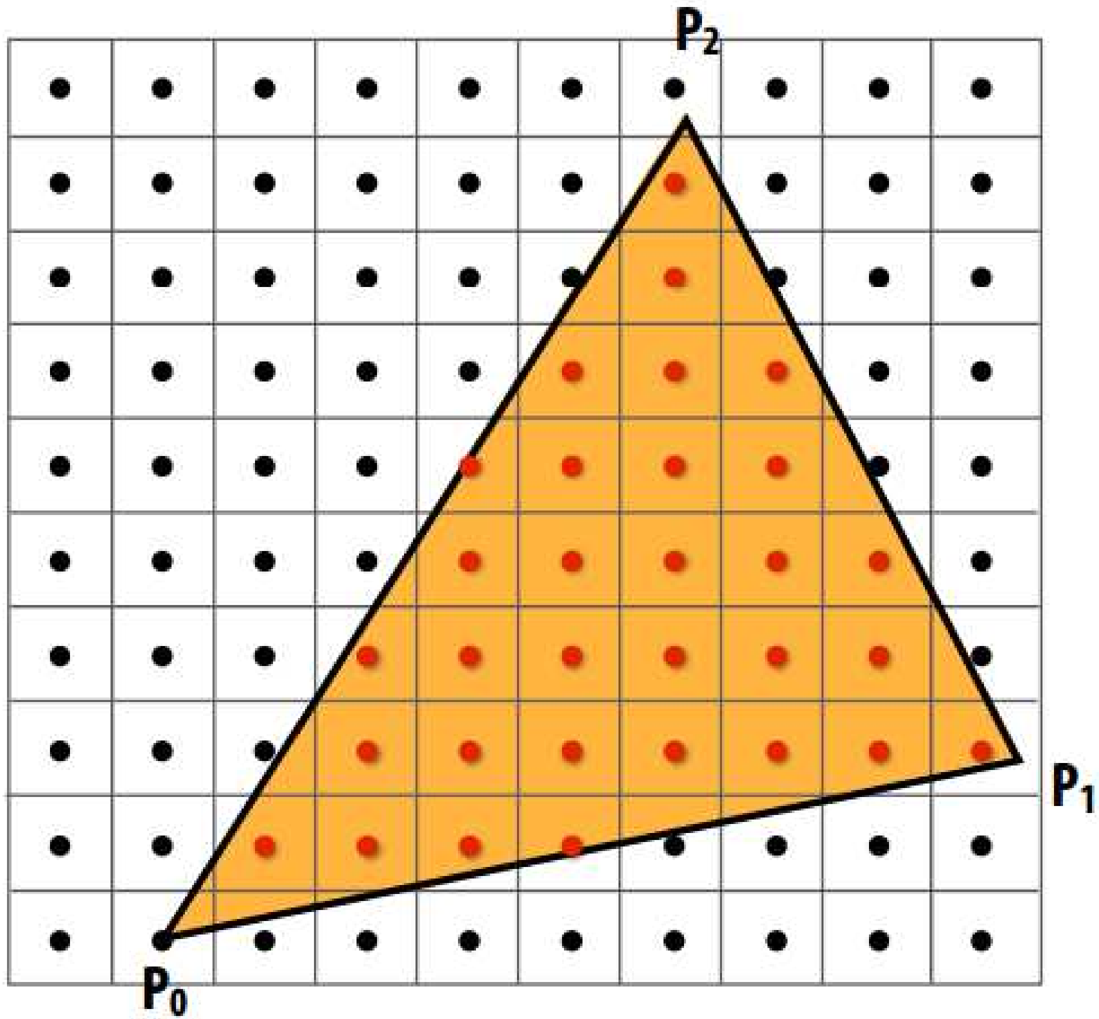


Point-inside-polygon test

Sample point $s = (sx, sy)$ is inside the triangle if it is inside all three edges.

$$\boxed{\begin{aligned} &inside(sx, sy) = \\ &\quad E_0(sx, sy) < 0 \;\&\& \\ &\quad E_1(sx, sy) < 0 \;\&\& \\ &\quad E_2(sx, sy) < 0; \end{aligned}}$$

Note: actual implementation of $inside(sx,sy)$ involves ≤ checks based on the triangle coverage edge rules (see beginning of lecture)



Sample points inside triangle are highlighted red.

Scanline algorithm

- **Incremental triangle traversal**

$P_i = (X_i, Y_i)$

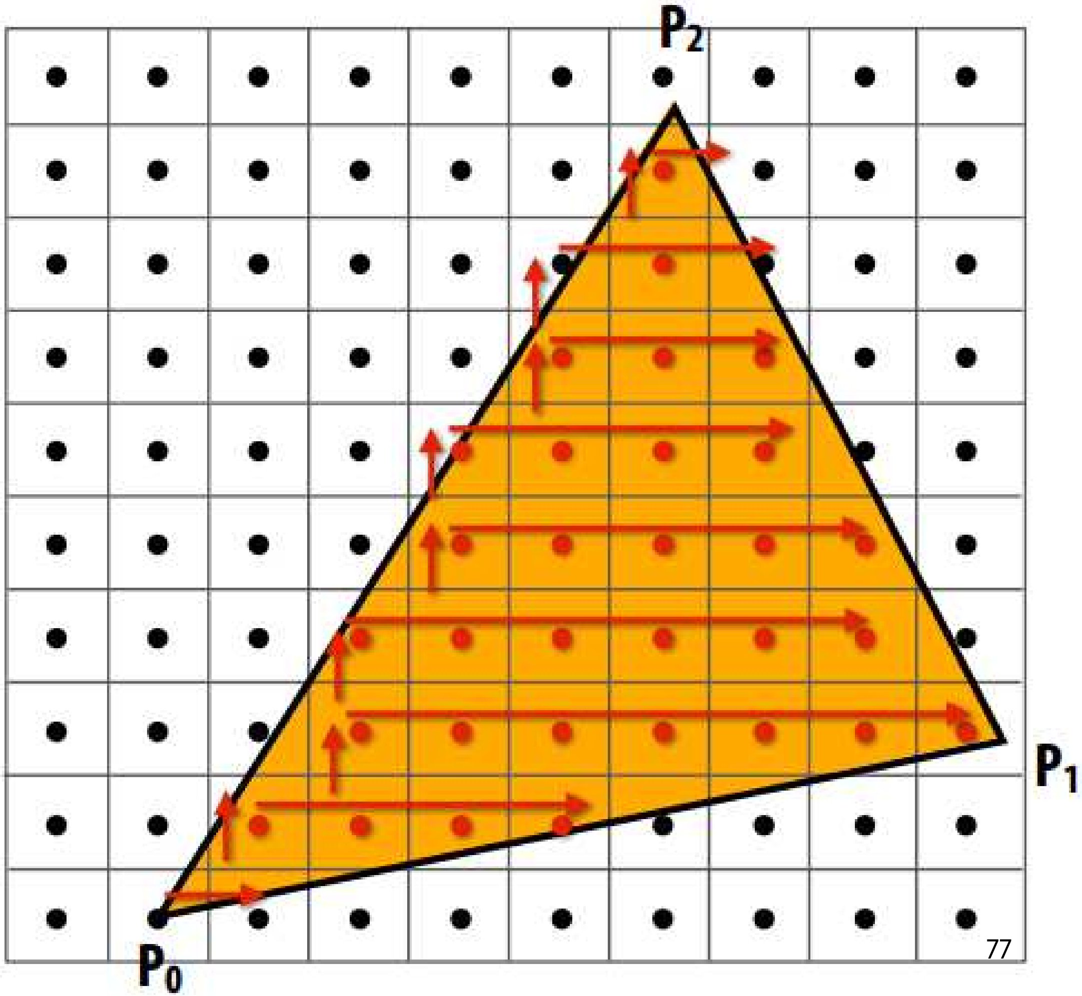
$dX_i = X_{i+1} - X_i$
$dY_i = Y_{i+1} - Y_i$

$$E_i(x, y) = (x - X_i)\, dY_i - (y - Y_i)\, dX_i = A_i x + B_i y + C_i$$

$E_i(x, y) = 0$: point on edge
> 0 : outside edge
< 0 : inside edge

Efficient incremental update:

$dE_i(x+1, y) = E_i(x,y) + dY_i = E_i(x,y) + A_i$
$dE_i(x, y+1) = E_i(x,y) - dX_i = E_i(x,y) + B_i$



Scan line algorithm

- **Modern approach: tiled triangle traversal**

Traverse triangle in blocks

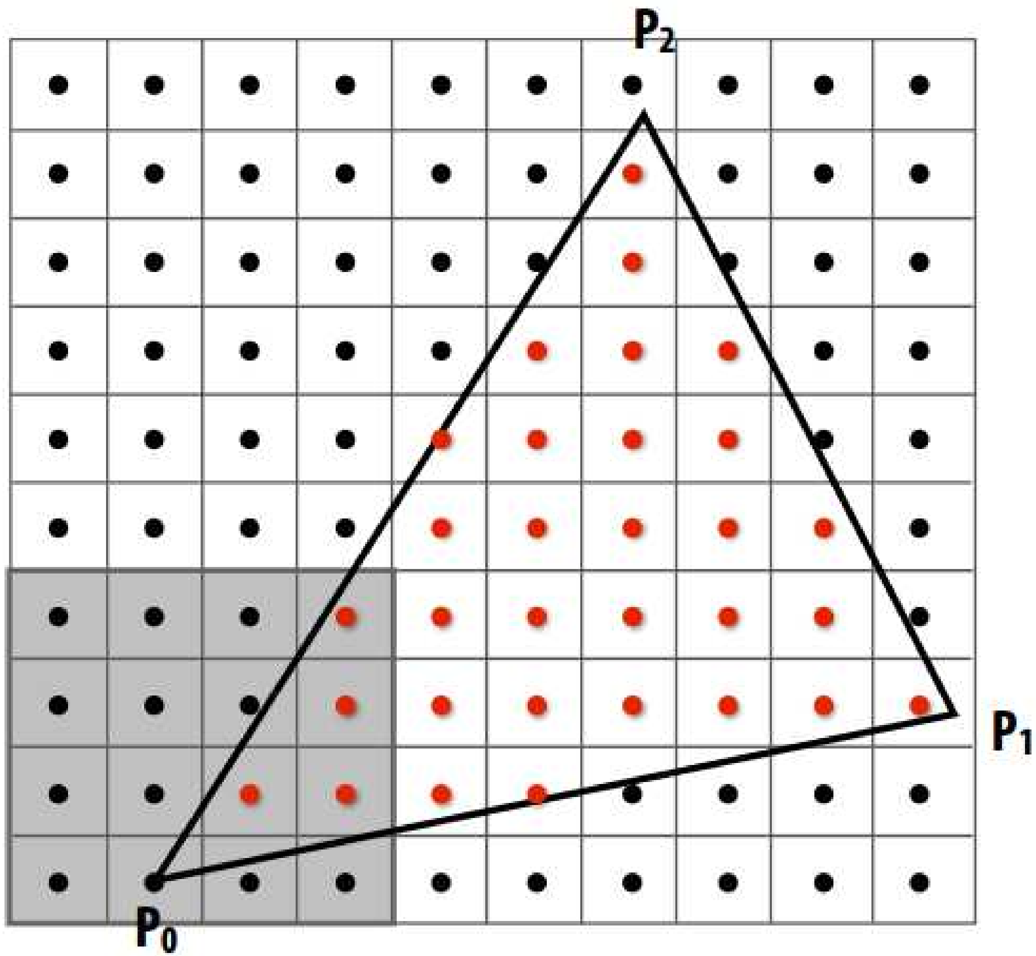
Test all samples in block against triangle in parallel

Advantages:

- **Simplicity of wide parallel execution overcomes cost of extra point-in-triangle tests (most triangles cover many samples, especially when super-sampling coverage)**

- **Can skip sample testing work: entire block not in triangle ("early out"), entire block entirely within triangle ("early in")**

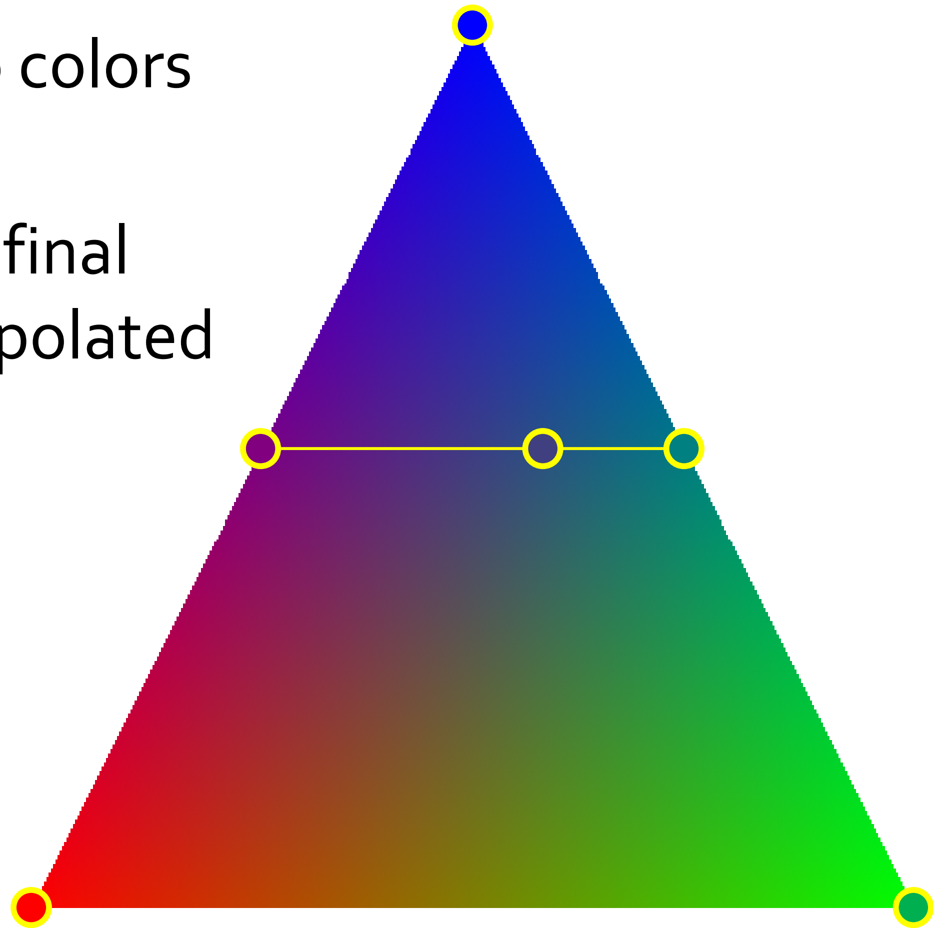
- **Additional advantaged related to accelerating occlusion computations (not discussed today)**



All modern GPUs have special-purpose hardware for efficiently performing point-in-triangle tests

Color interpolation

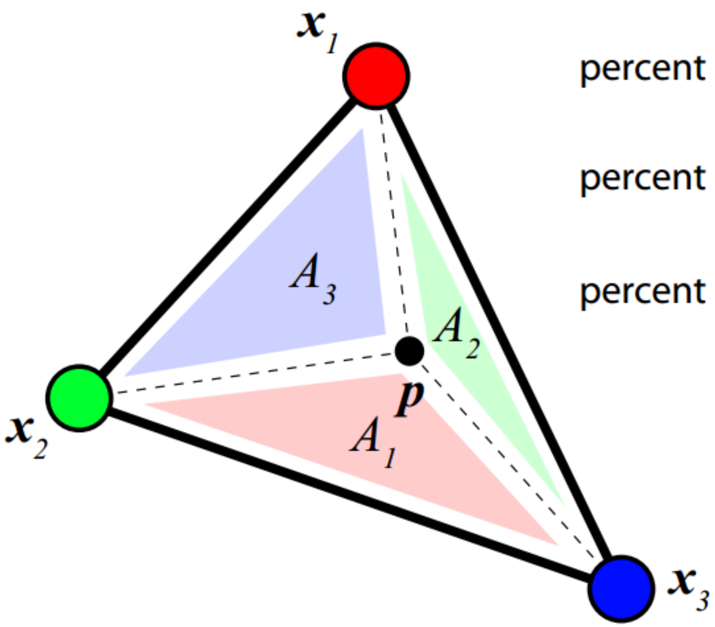
- **How to fill the color of the pixels inside the triangle region?**
  - Linearly interpolate two colors along two edges
  - Linearly interpolate the final color based on the interpolated two colors



Color interpolation

- **How to fill the color of the pixels inside the triangle region?**
  - Use barycentric interpolation (another approach)

Barycentric Interpolation



percent red $= \dfrac{A_1}{A} = \lambda_1$

percent green $= \dfrac{A_2}{A} = \lambda_2$

percent blue $= \dfrac{A_3}{A} = \lambda_3$

"barycentric coordinates"

$$\Sigma_i \lambda_i = 1$$

Value at $\boldsymbol{p}$:

$$(A_1\boldsymbol{x}_1 + A_2\boldsymbol{x}_2 + A_3\boldsymbol{x}_3)/A$$

Fragment/pixel shader

- **Set customized color for each rasterized fragment/pixel**
  - The process is done after the automatic rasterization
  - Can transfer interpolated properties from vertex shader

```
varying vec4 vColor;

void main(void)
{
   vColor = gl_Color;
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

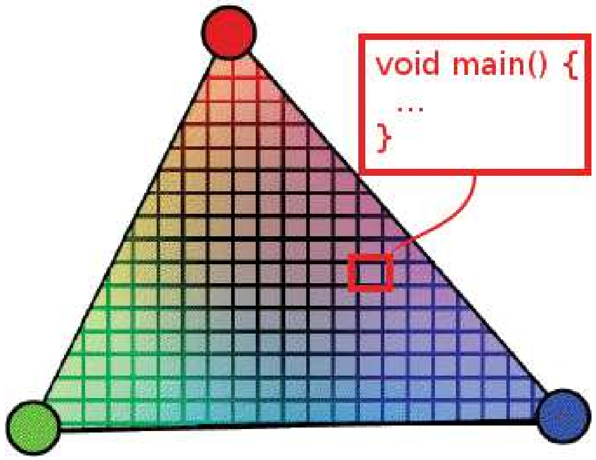
vertex shader

```
varying vec4 vColor;

void main (void)
{
   gl_FragColor = vColor;
}
```

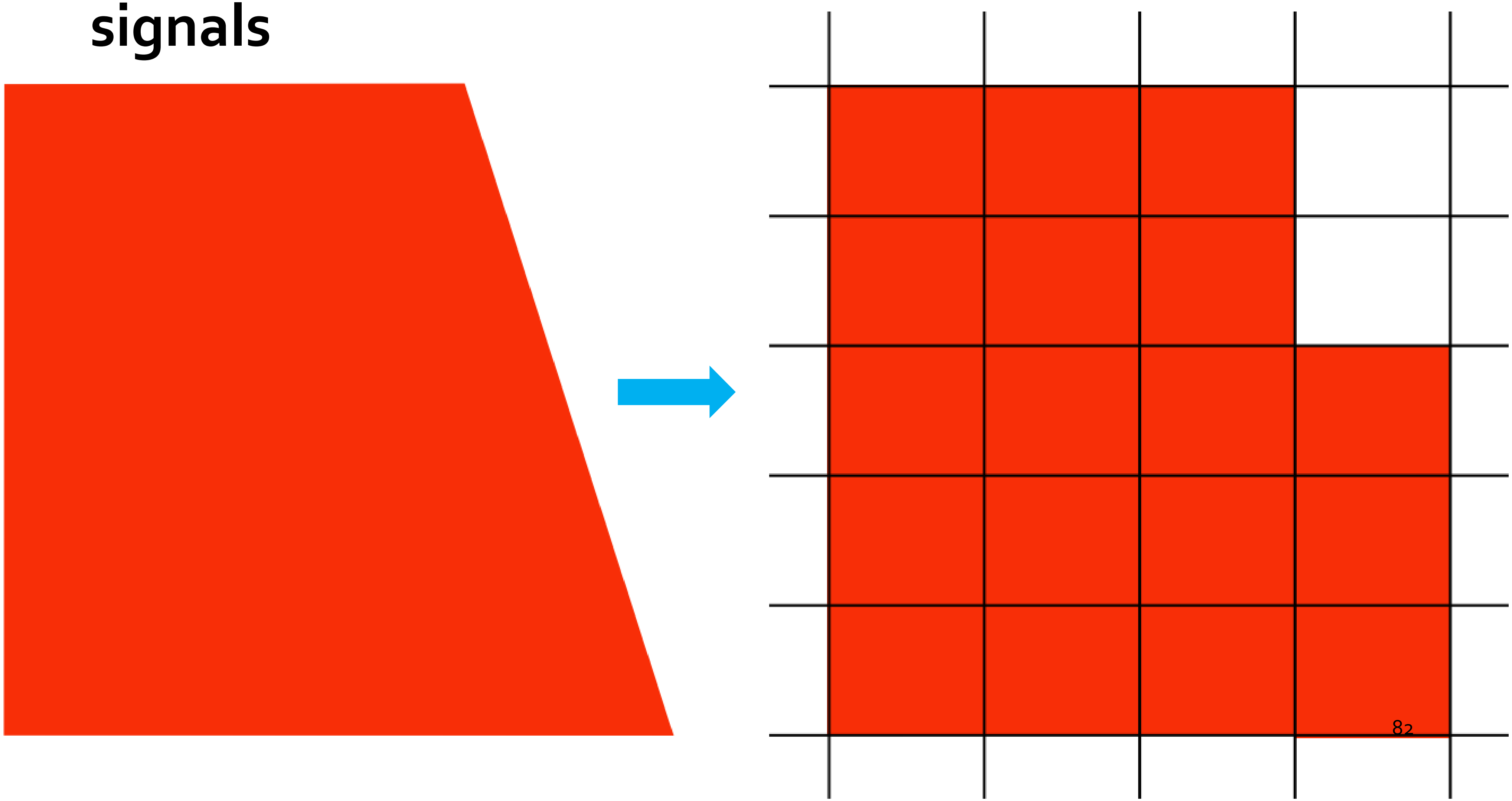
color interpolated from the vertex automatically after the rasterization

fragment shader



Aliasing in rasterization

- **Comparison between continuous and rasterized signals**



Reason for aliasing

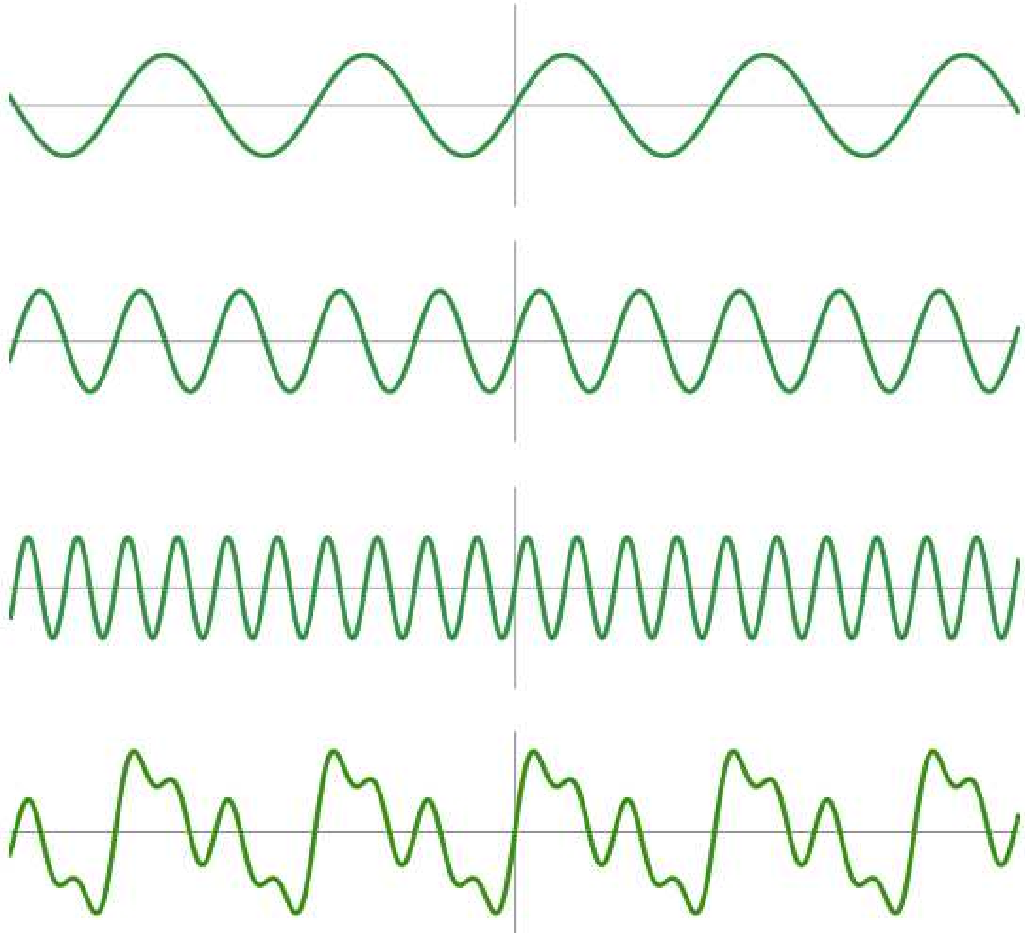
- **Represent a signal as a superposition of frequencies**

$f_1(x) = sin(\pi x)$

$f_2(x) = sin(2\pi x)$

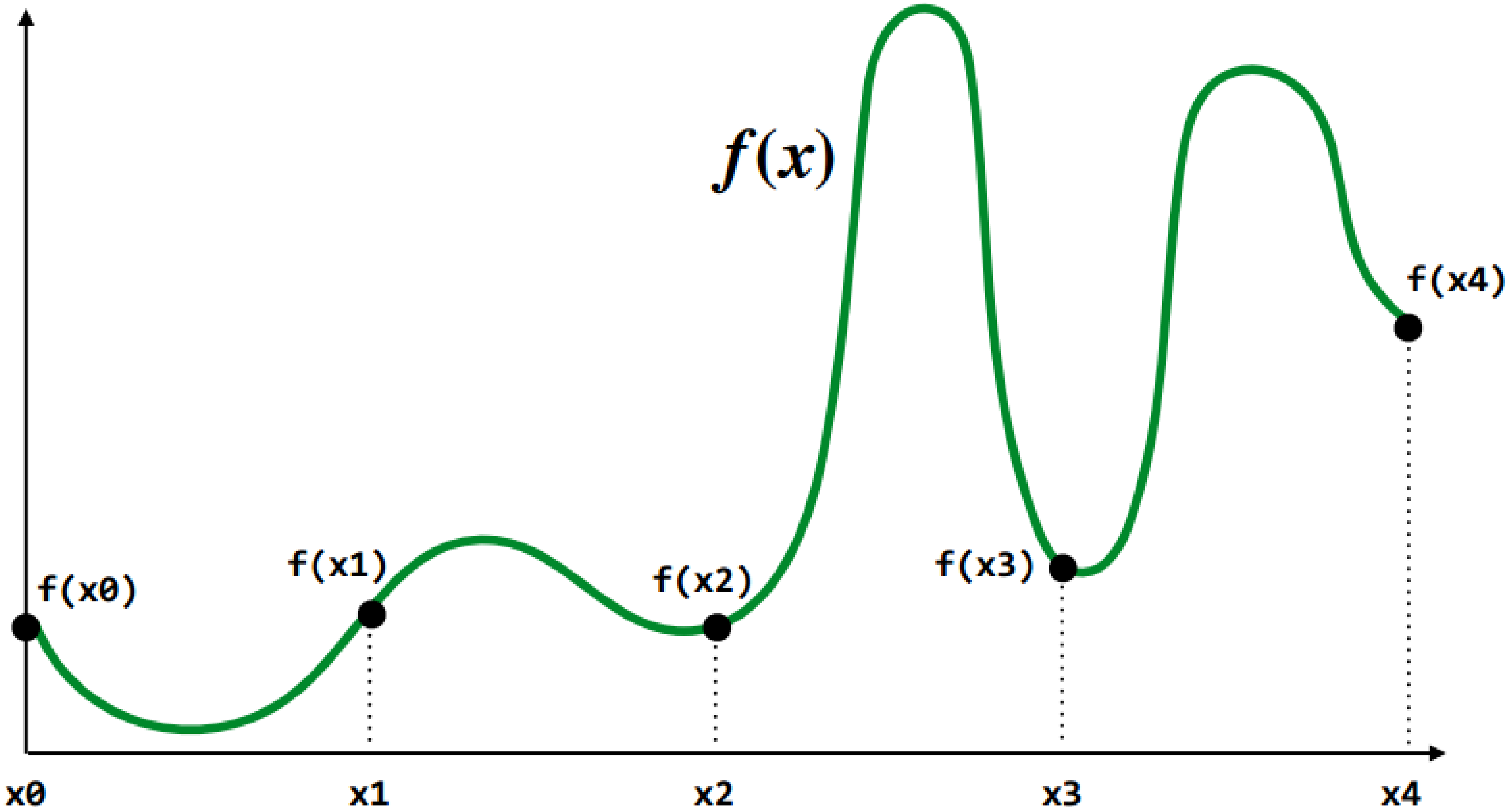
$f_4(x) = sin(4\pi x)$

$f(x) = f_1(x) + 0.75\, f_2(x) + 0.5\, f_4(x)$



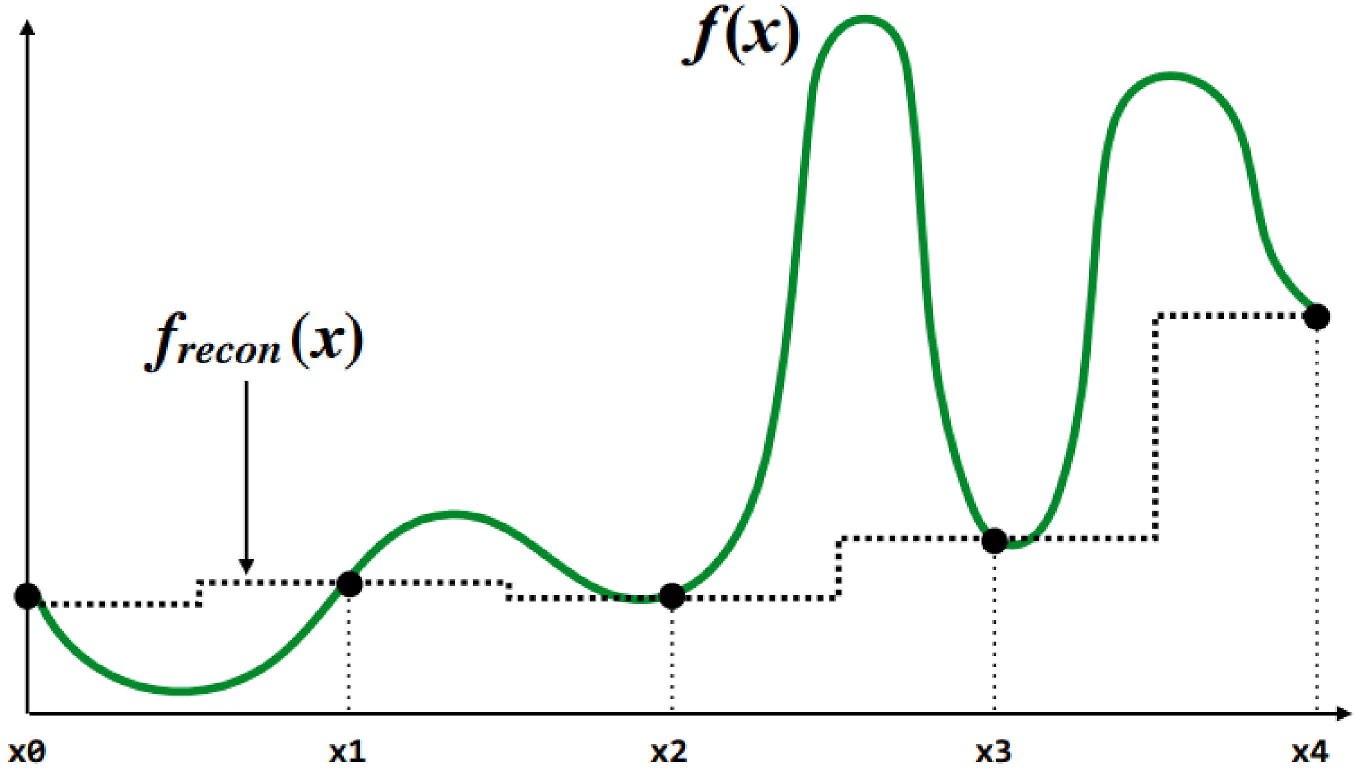
Reason for aliasing

- **Sampling: taking measurements of a signal**



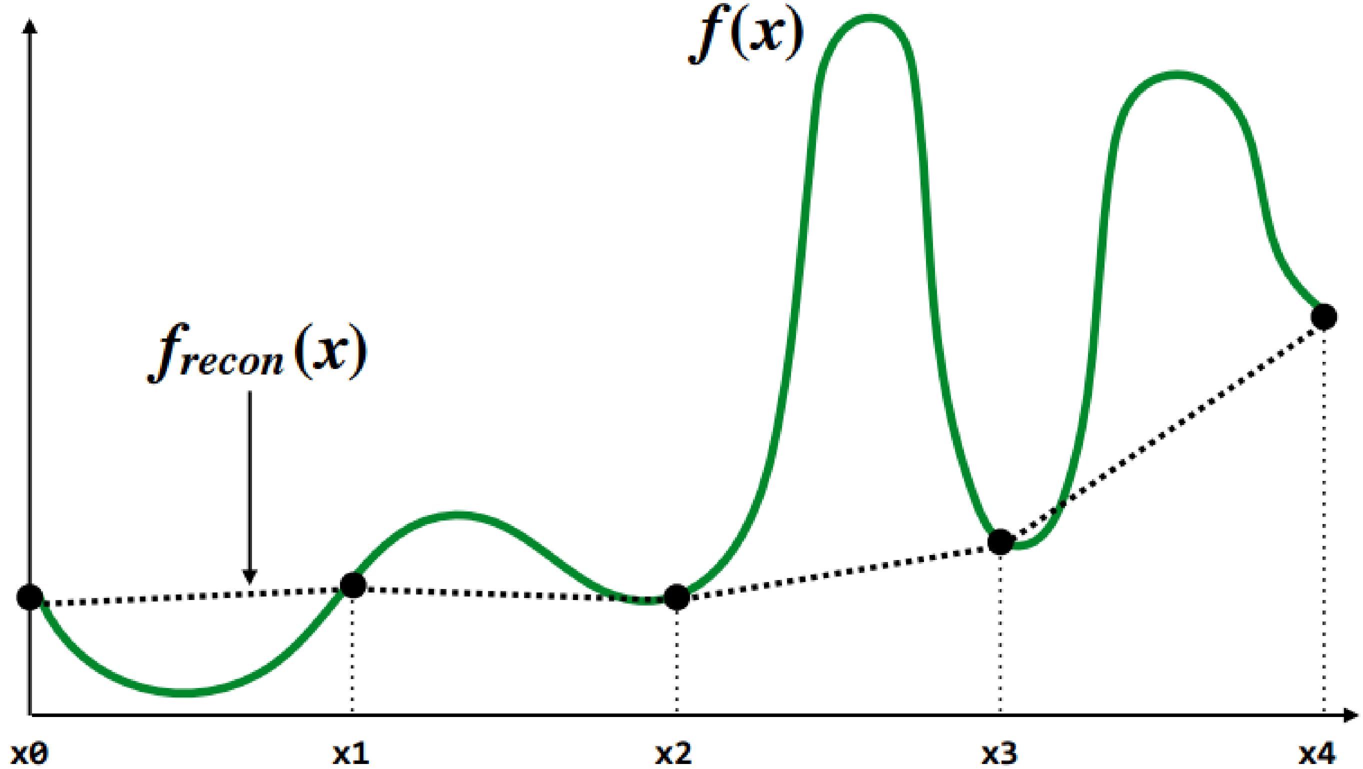
Reason for aliasing

- **Reconstruction:**
  - Given a set of samples, how can we attempt to reconstruct the original signal f(x)?
    - Piecewise constant approximation



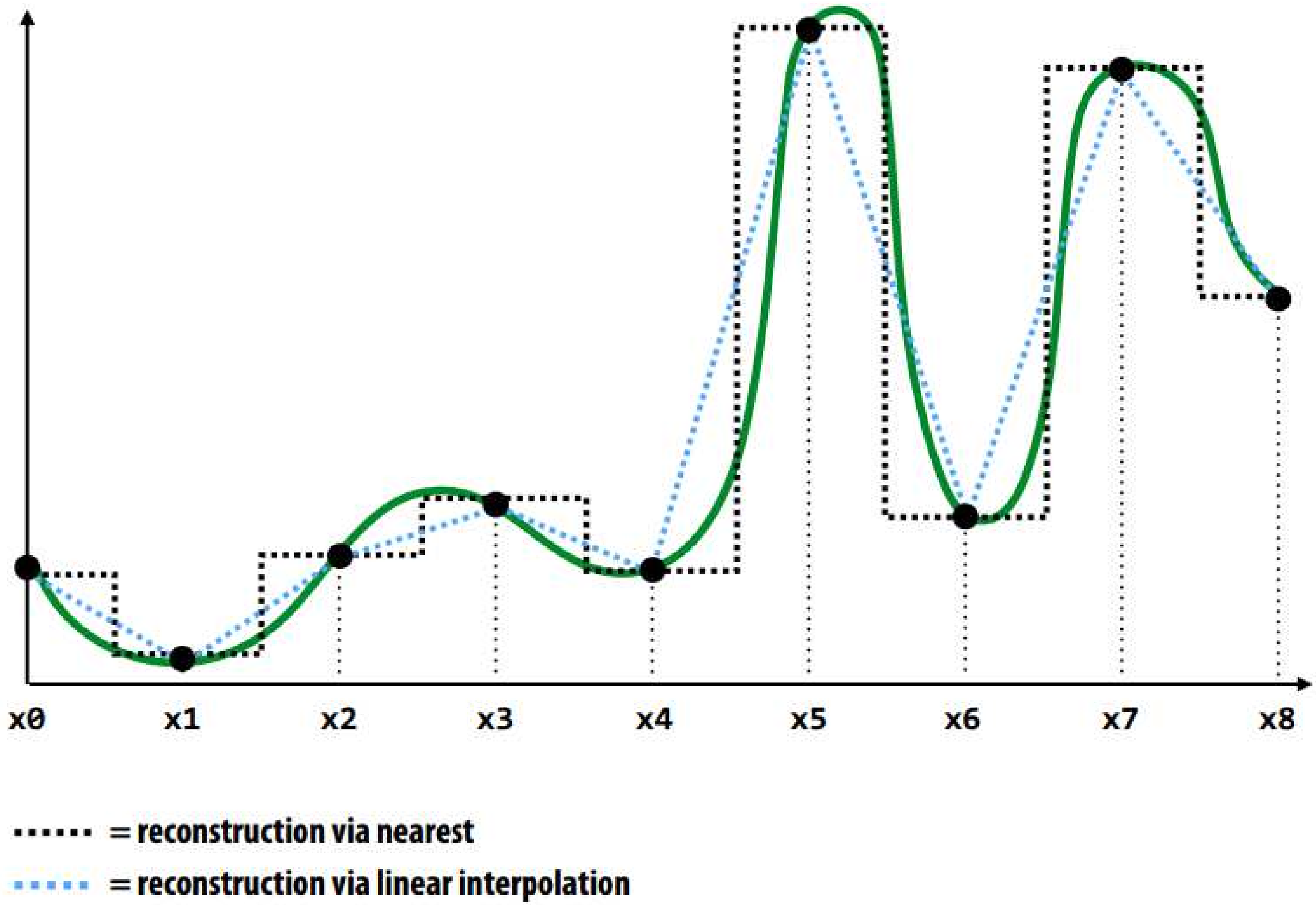
Reason for aliasing

- **Reconstruction:**
  - Given a set of samples, how can we attempt to reconstruct the original signal f(x)?
    - Piecewise linear approximation



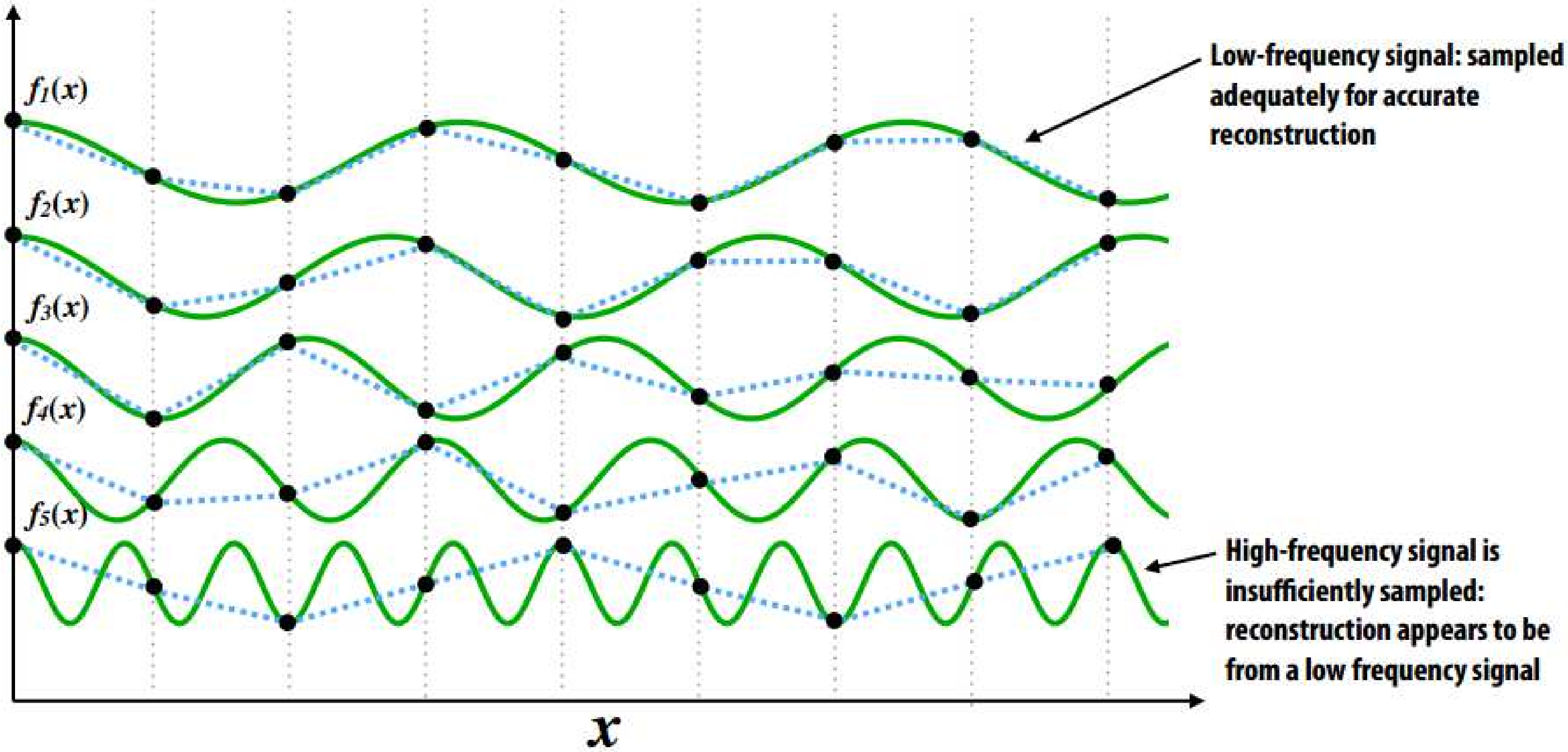
Reason for aliasing

- **How can we represent the signal more accurately?**
  - Sample the signal more densely



Reason for aliasing

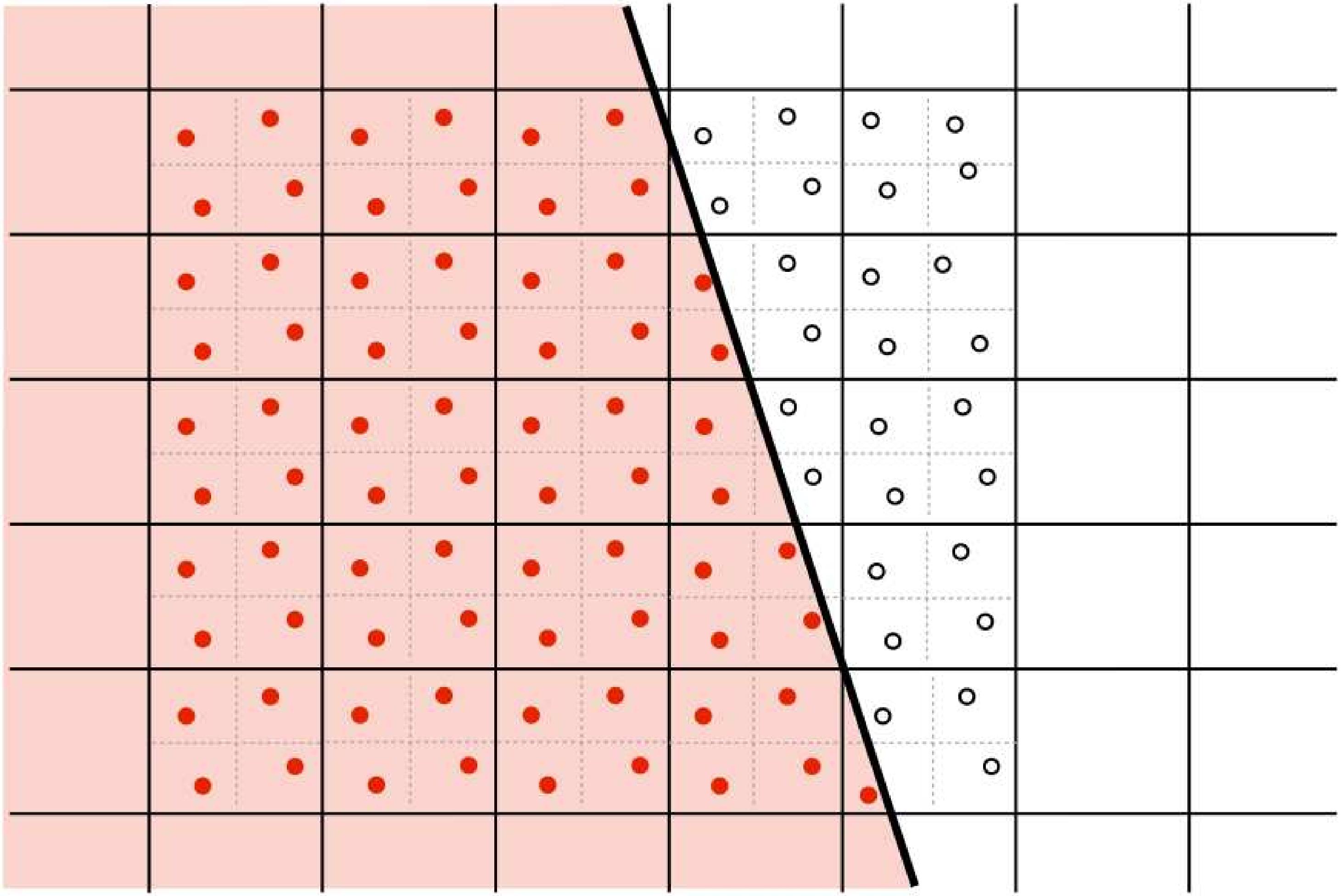
- **Under-sampling high-frequency signals results in aliasing**



"Aliasing": high frequencies in the original signal masquerade as low frequencies after reconstruction (due to undersampling)

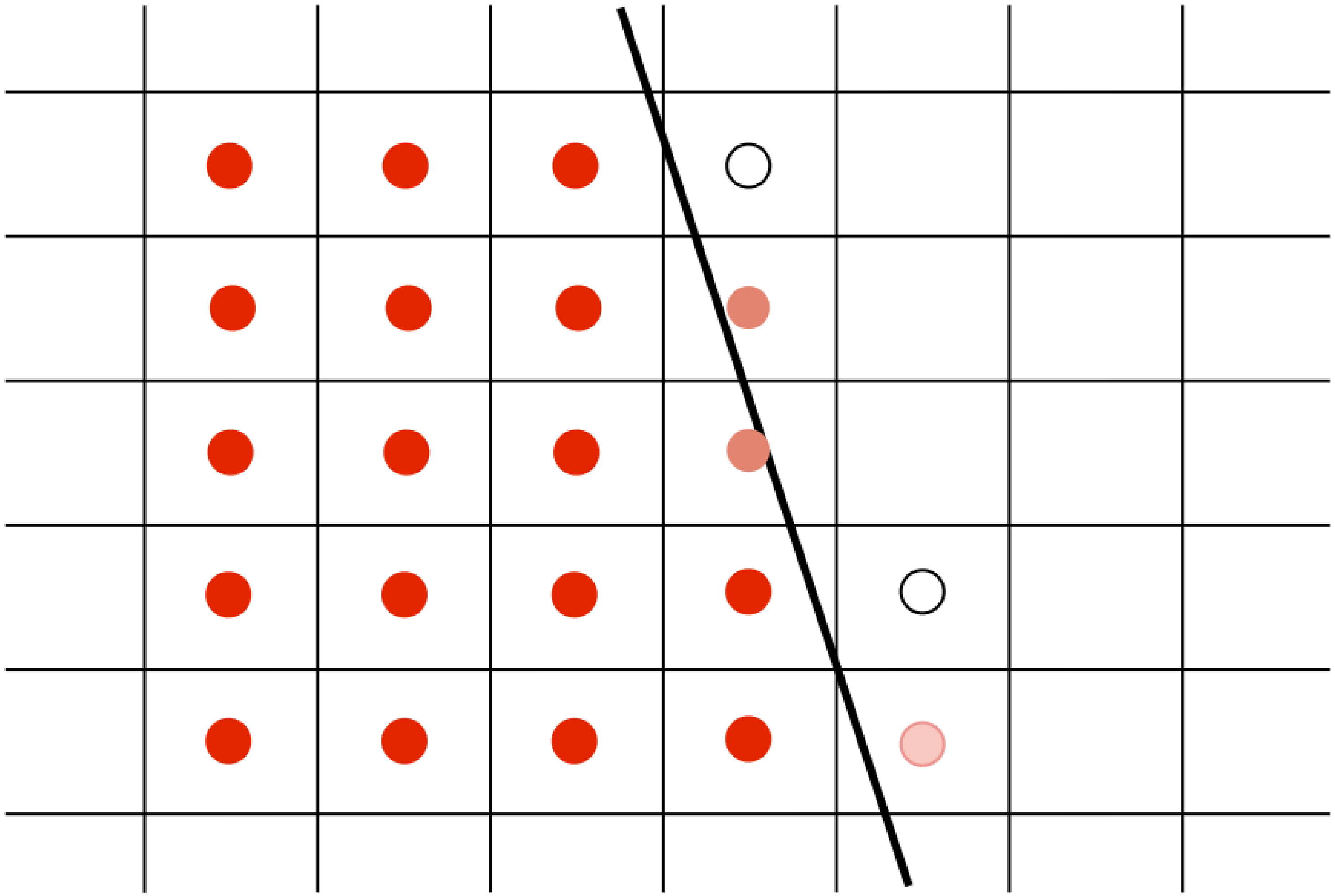
Antialiasing techniques

- **Super-sampling**
  - Example: stratified sampling using four samples per pixel



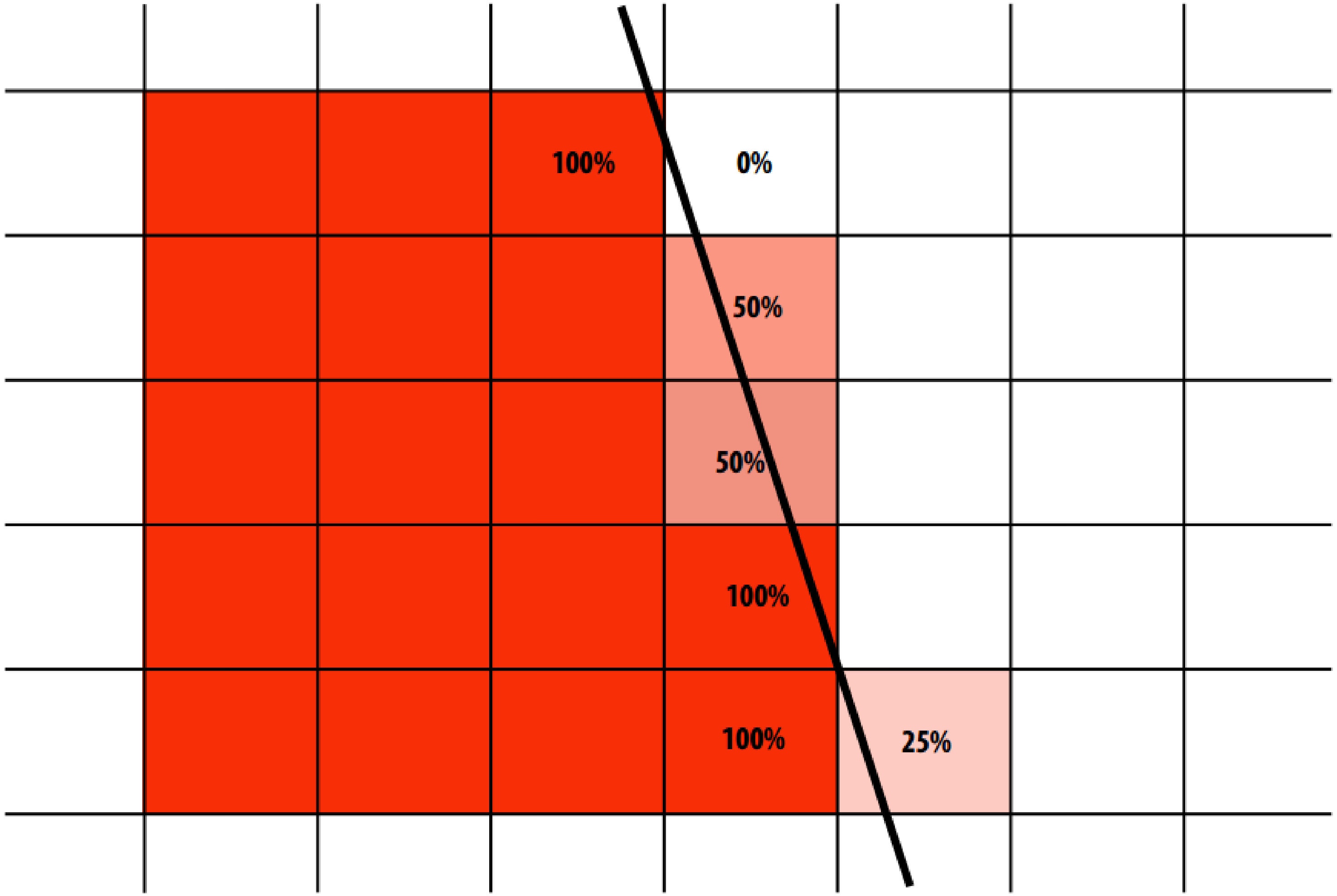
Antialiasing techniques

- **Super-sampling**
  - Resample to display's resolution (box filter)



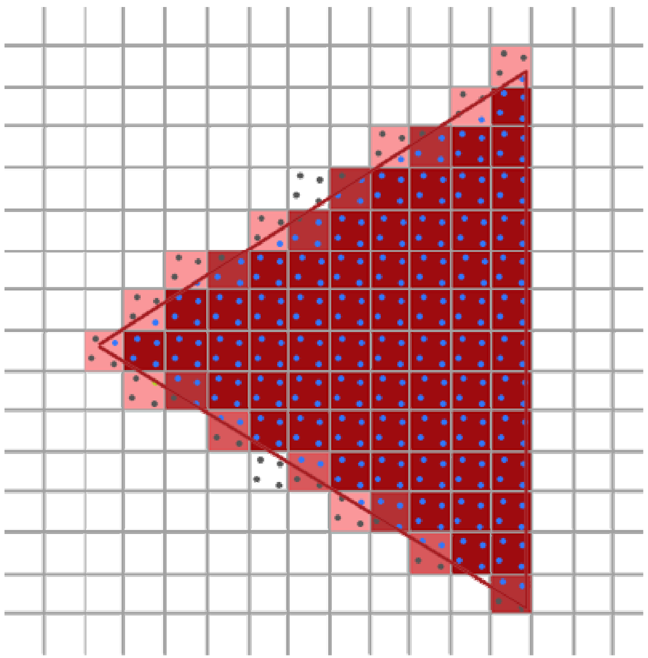
Antialiasing techniques

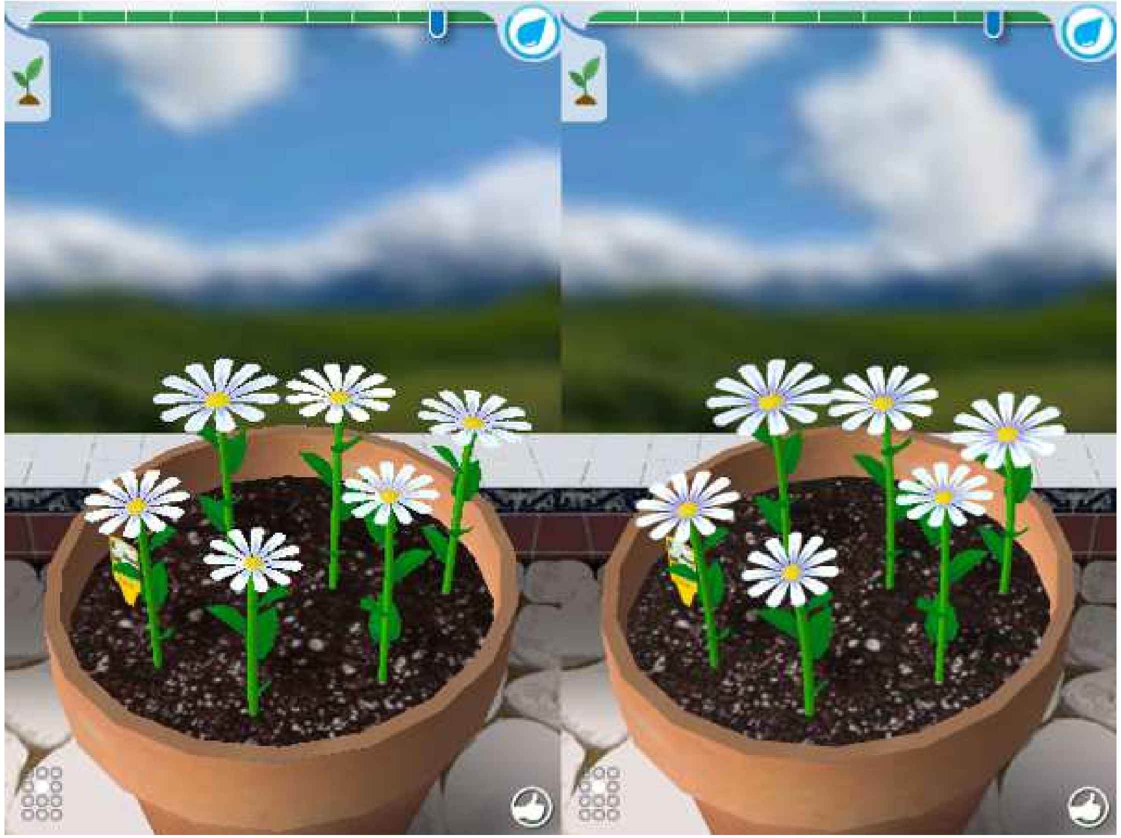
- **Supersampling**
  - Displayed result (note anti-aliased edges)



Antialiasing in OpenGL

- **Multi-sampling**
  - Render in higher resolution and down sample by averaging





Antialiasing in OpenGL

- **Enable multi-sample antialiasing in GLFW**
  - Create a window with multi-sample support
  - Call glfwWindowHint before creating the window

```
glfwWindowHint(GLFW_SAMPLES, 4);
```

  - Enable multi-sampling in OpenGL

```
glEnable(GL_MULTISAMPLE);
```

Next Lecture :

Geometric representations & triangulations