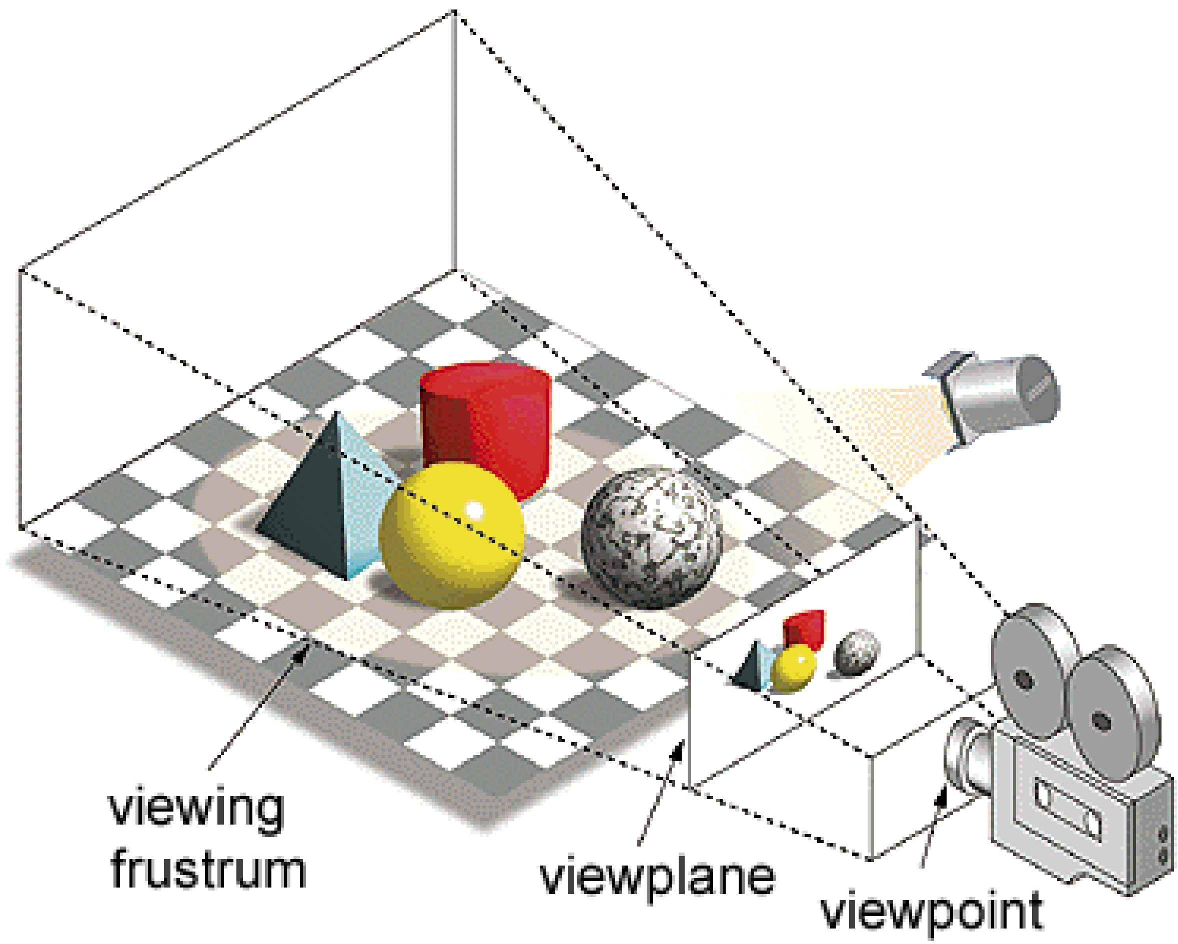
Rendering

From Computer Desktop Encyclopedia
Reprinted with permission.

Rendering
is the ”engine” that creates images from 3D scenes and a virtual camera



Rendering : PART I

- **Pipeline Based Rendering (forward rendering)**
  - Object in scene are rendered in a sequence of steps called Rendering Pipeline. The real-time graphics pipeline (GPU)

- **Ray-Tracing (backward rendering)**
  - Project rays through the view plane and assign color to pixels according to the first ray intersection.

    From the screen window process the geometric primitives which are projected onto it. (CPU/GPU)

Rendering Pipeline

A **3D scene** is:

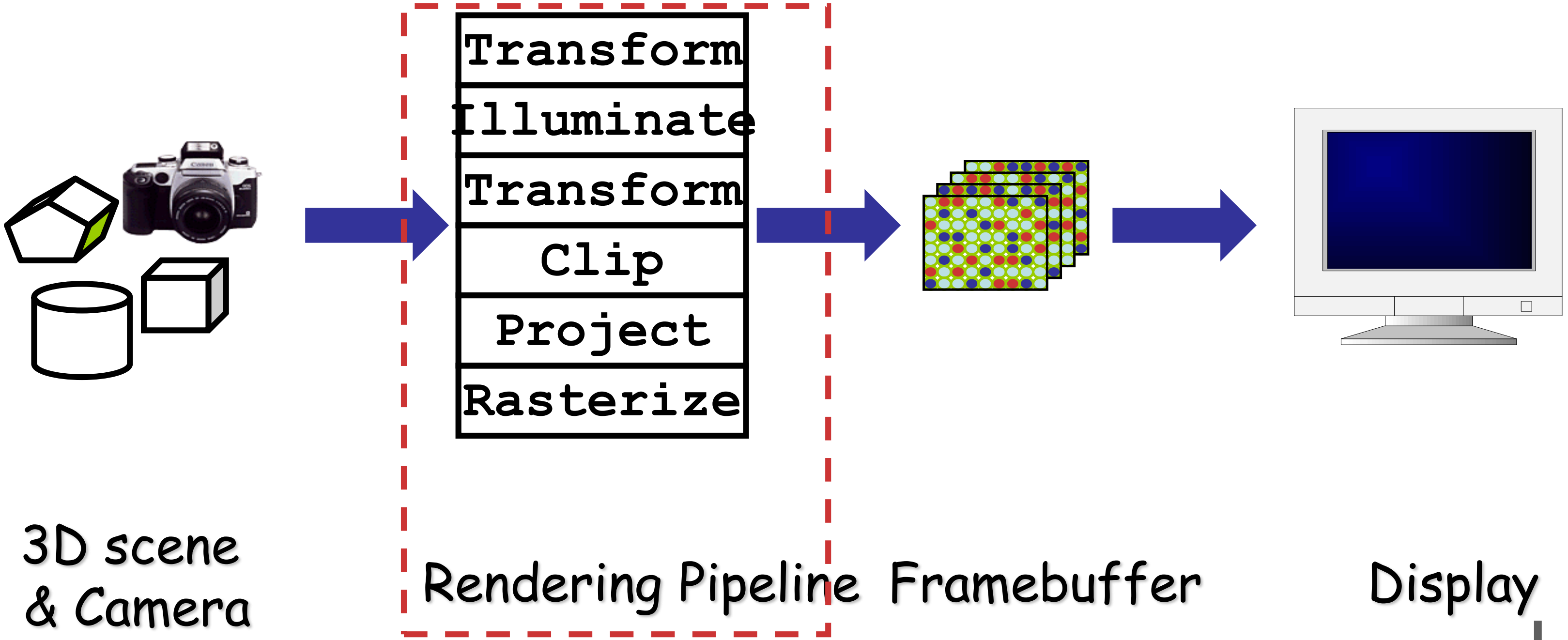
– Geometry (triangles, lines, points, and more)

– Light sources

– Material properties of geometry

– Textures (images to glue onto the geometry)

A **virtual camera** (Decides what should end up in the final image)



Pipeline Based Rendering: idea

- Take a collection of 2D Polygonal Objects and draw them onto a framebuffer

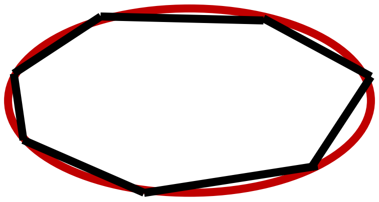
- Real-time
- An object is approximated by a number of simple primitives (points, lines, triangles).
- Use tessellation to convert complex models into simple geometric primitives

Tessellation

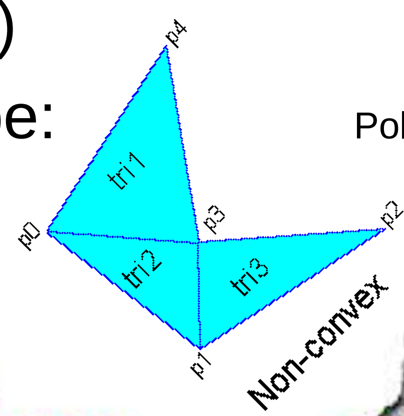
– All curves/surfaces can be broken up (approximate) into a sequence of segments/polygons (triangles)

– Triangles are guaranteed to be:

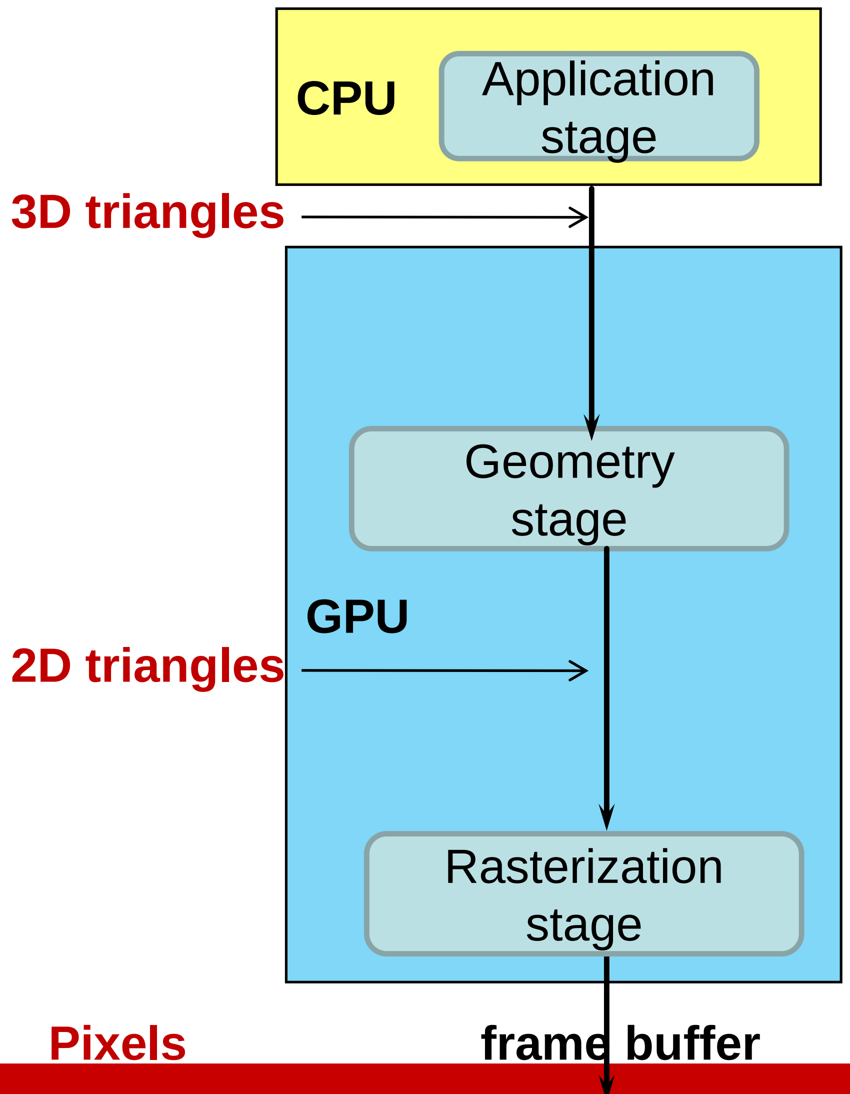
  • Planar and convex

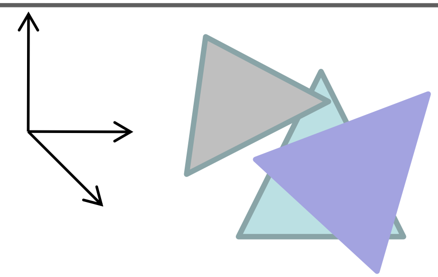


Polygonal approximation to a curve



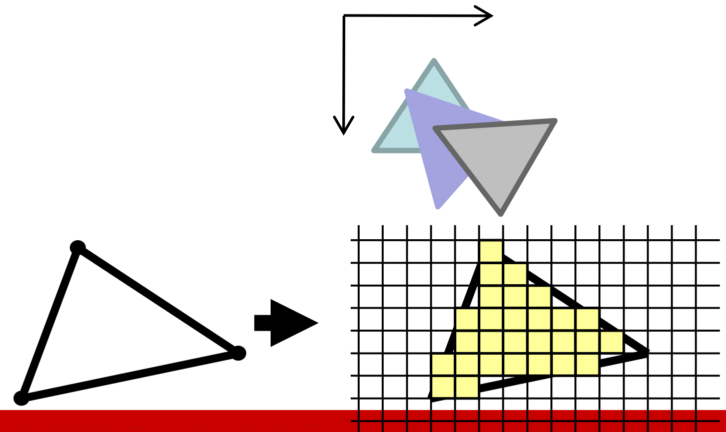
Rendering pipeline: a functional overview





Three conceptual stages of the pipeline:

- Application (executed on the **CPU**)
- Geometry (hardware/software on the **GPU**)
- Rasterizer (hardware/software on the **GPU**)

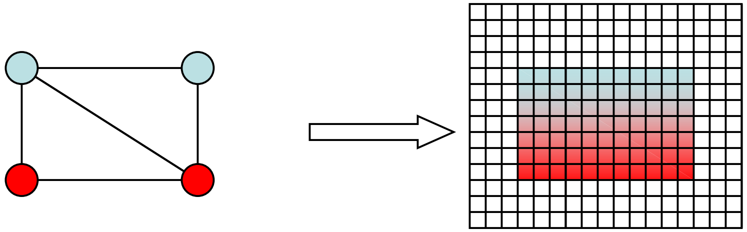


The Geometry Stage: per vertex operations

- **Task**: ”geometrical” operations on the input data (e.g. vertices of the triangles)
- **Allows:**
  – Move objects (matrix multiplication)
  – Move the camera (matrix multiplication)
  – Compute lighting at vertices of triangle
  – Project onto screen (3D to 2D)
  – Clipping (avoid triangles outside screen)
  – Map to window

The Rasterizer Stage: per pixel operations

- **Task**: take output from GEOMETRY (2D polys) and turn into visible pixels on frame buffer (screen)

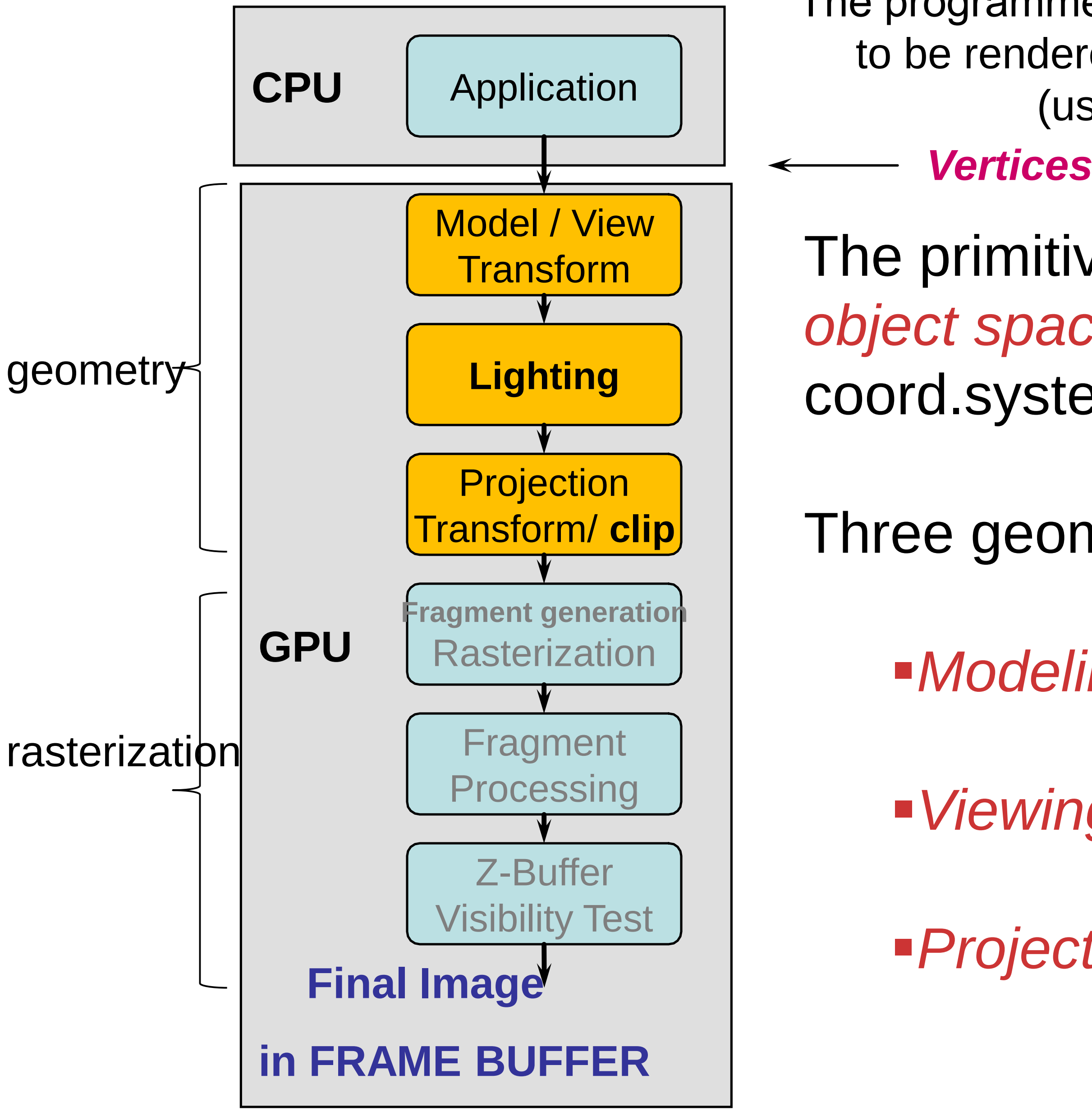


- **Allows:**
  - Scan conversion

    Converts a geometric primitive in a set of fragments

    Fragment: location (x,y); depth; color; texture coord.,...
  - Interpolation (lighting, texturing, z values, ..)
  - Color combining (light and texture colors) and other pixel operations (alpha, stencil test,..)
  - Visibility (depth test)

Geometry stage: transformations



The programmer ”sends” down primitives to be rendered through the pipeline (using API calls)

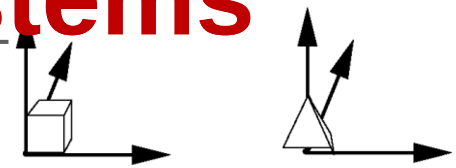
The primitives are modeled in *object space or* object coord.system OCS

Three geometric transformations:

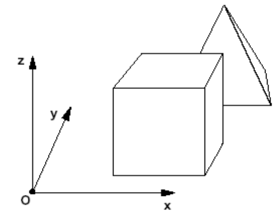
- *Modeling*
- *Viewing*
- *Projection*

Geometry Stage: transformation of coordinate systems

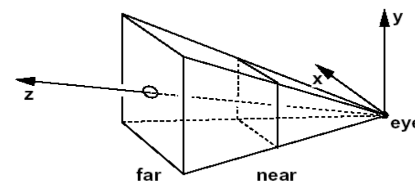
- OCS Object space
  – local to each object



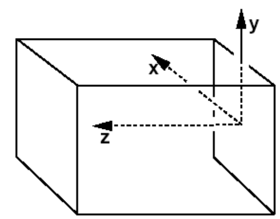
- WCS World space
  – common to all objects



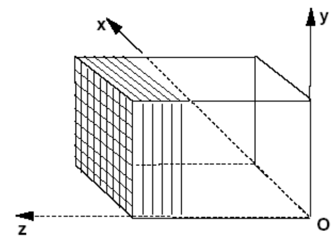
- VCS Eye space / Camera space
  – derived from view frustum



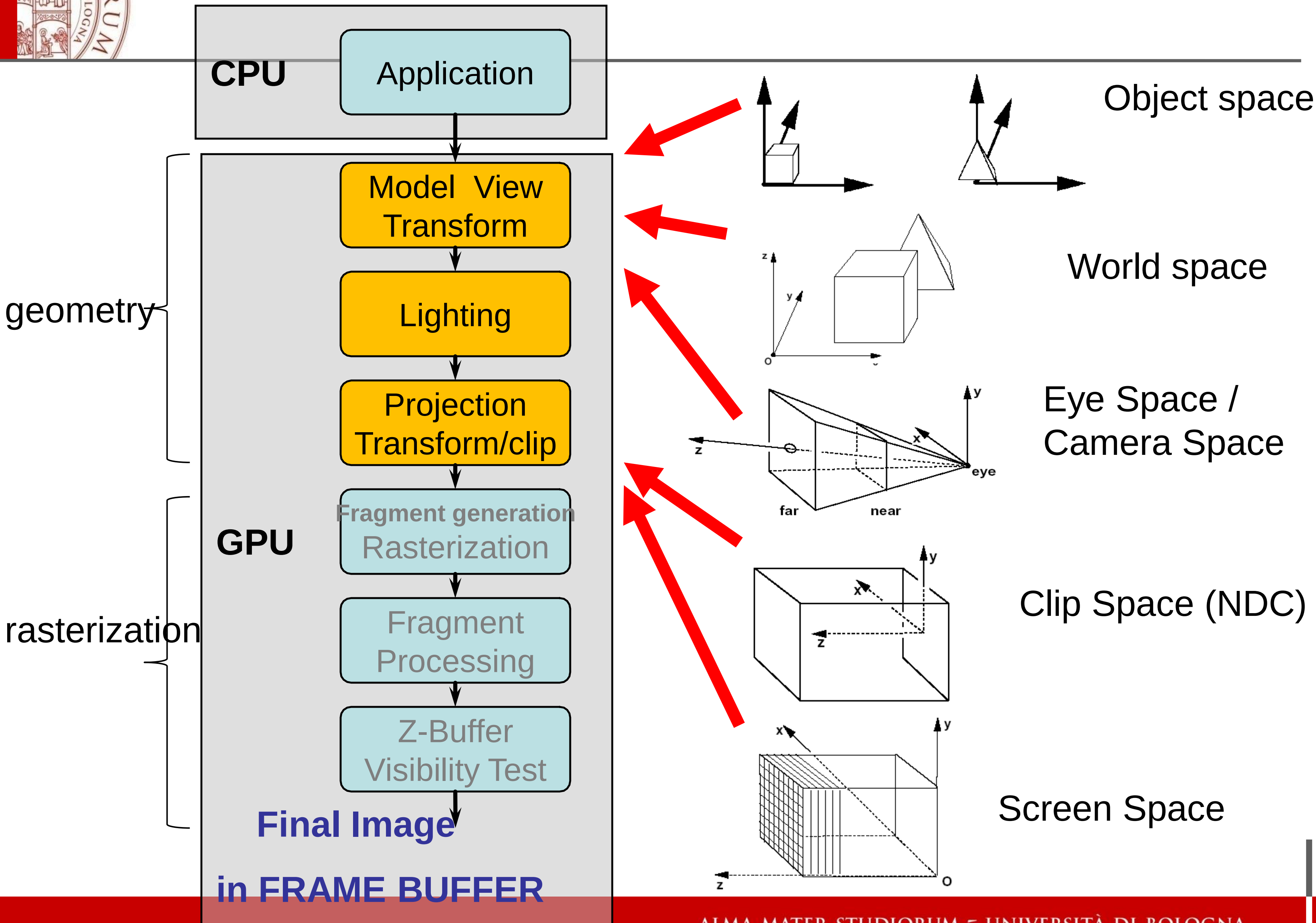
- NDC Clip space / Normalized Device Coordinates
  – [-1,-1,-1] → [1,1,1]



- SCS Screen space
  – indexed according
    to hardware attributes



Geometry stage: coords. systems



The rendering pipeline: modeling transforms

INPUT: vertices in object coord. system **OCS**

OUTPUT:

All vertices of scene in shared 3-D "world" coordinate system (WCS)

3D vertices

geometry

Model View **Transform**

Lighting

Projection Transform/clip

rasterization

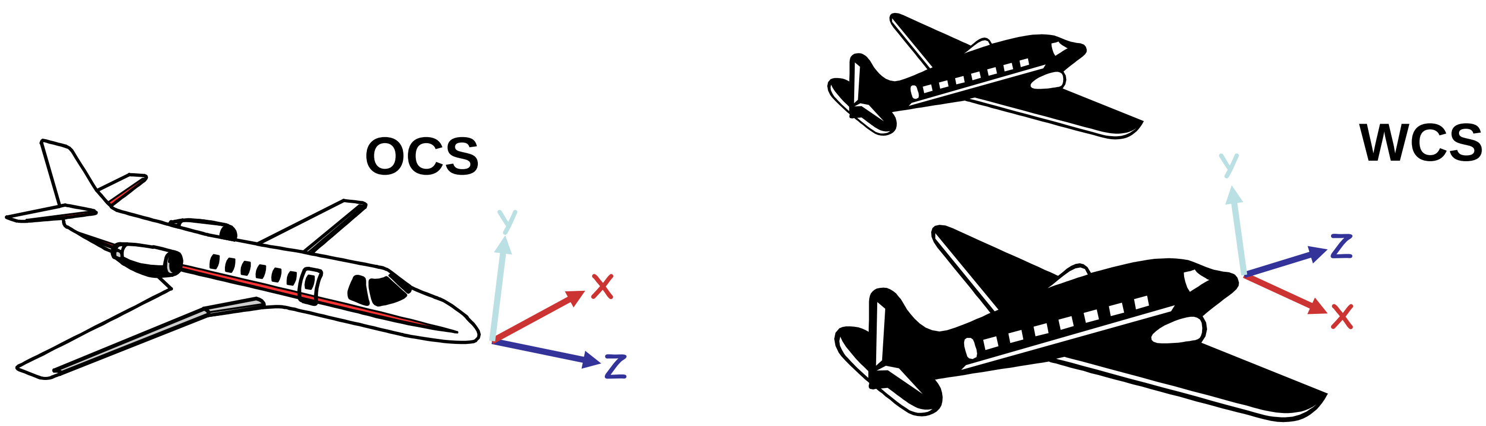
Fragment generation Rasterization

Fragment Processing

Z-Buffer Visibility Test

Modeling transforms

- Originally, an object is in ”model space”
- Move, orient, and transform geometrical objects and parts of the models w.r.t. each other into ”world space”
- Object coords (OCS)   world coords (WCS)
- Multiply each vertex by an affine transf. matrix 4x4 **Tm**



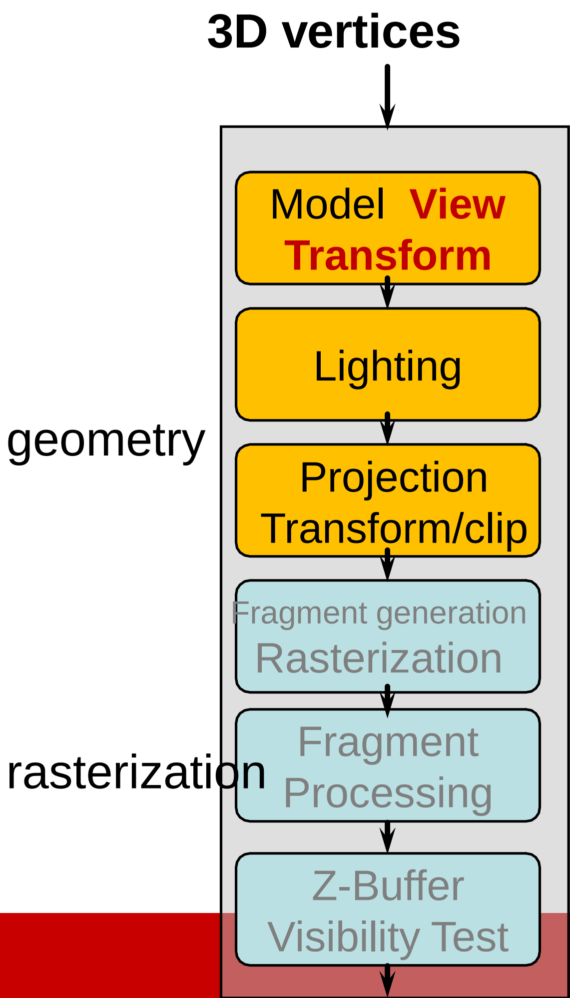
The user can apply different matrices over time to animate objects

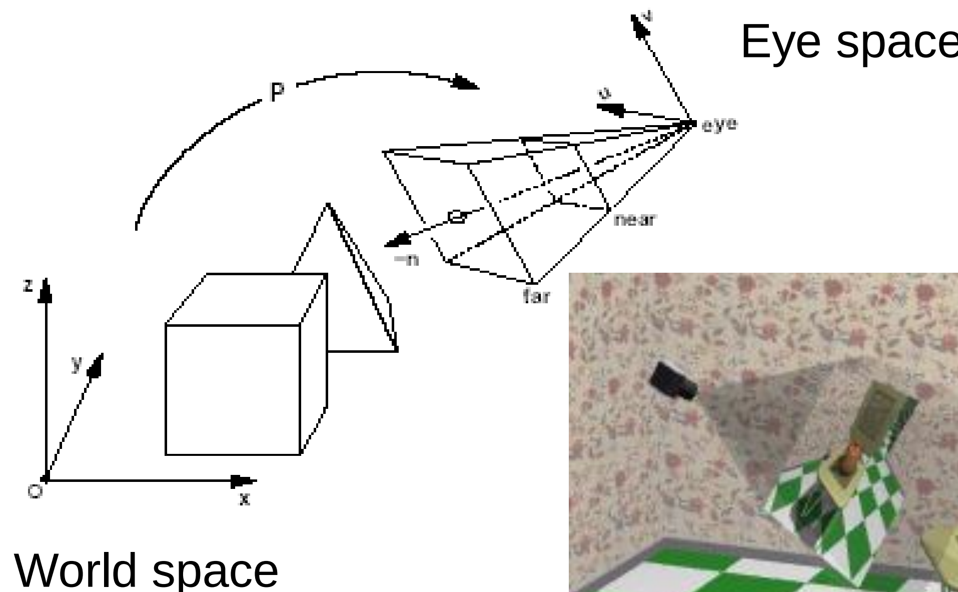
The rendering pipeline: the view transform

INPUT: vertices in **WCS**

OUTPUT:

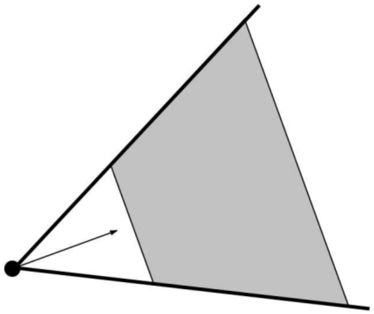
Scene vertices in 3-D “**View**” or “camera” **Coordinate System (VCS)**





Building Viewing Transformation

WCS → **VCS**



- Specifying 3D view—setting up synthetic camera
  - The camera is located and oriented in WCS

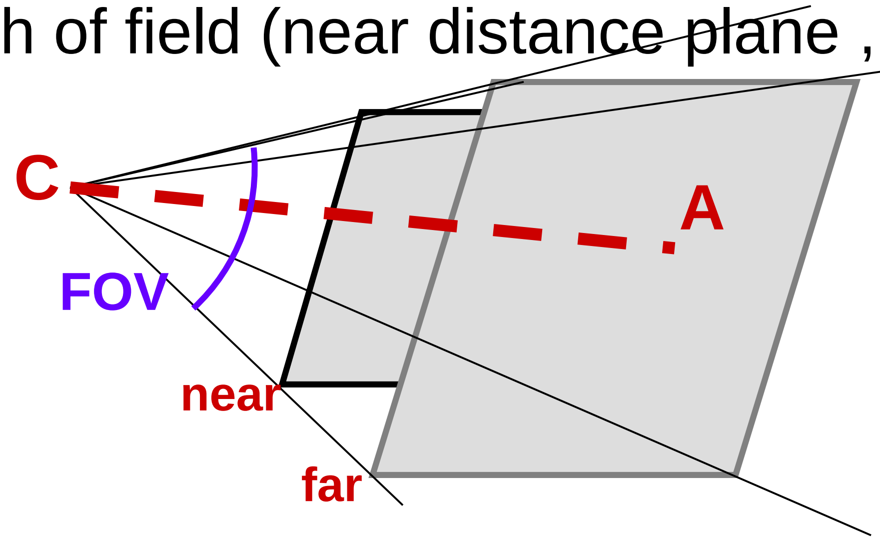
- Building Viewing Transformation from View Specification
  - Build a transformation matrix $\mathbf{T_v}$
    (nothing else but a change of reference system)

- Apply this transformation to every object vertex

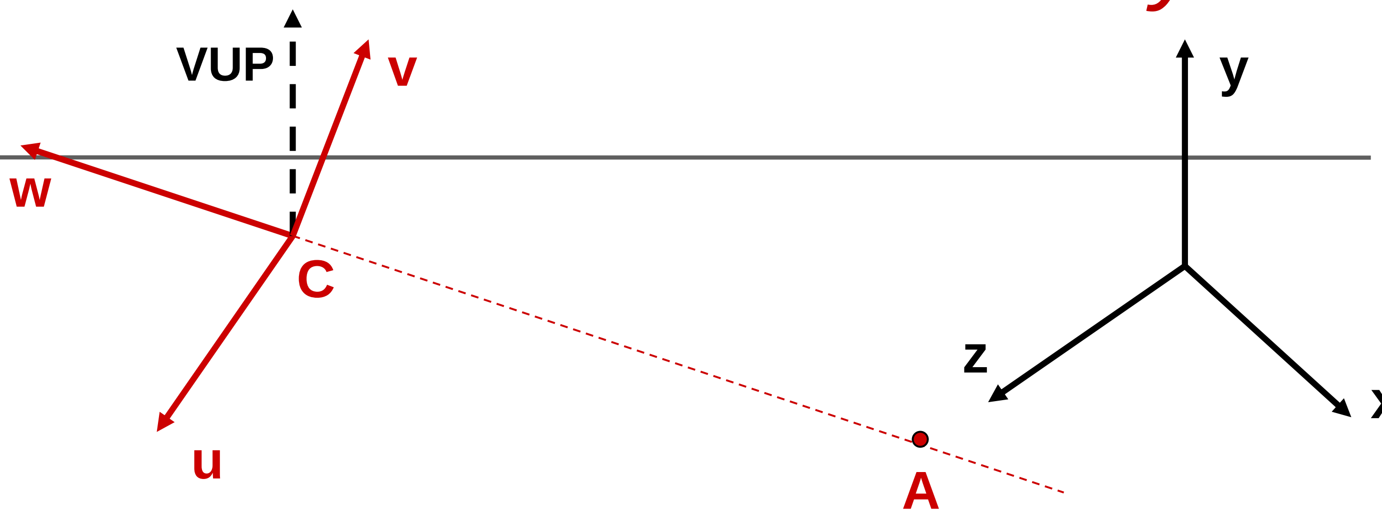
View Specification: **Camera "look at"**

We need to know four things about our synthetic camera model in order to make up the view transform

- *Point* **C**: *Position* of the camera in WCS (from where it's looking)
- *Point* ***A***: The *Look vector* specifies in what direction the camera is pointing (view direction C-A , A =center of scene)
- field of view **FOV** (wide angle, normal…)
- depth of field (near distance plane , far distance plane)



View reference coordinate system

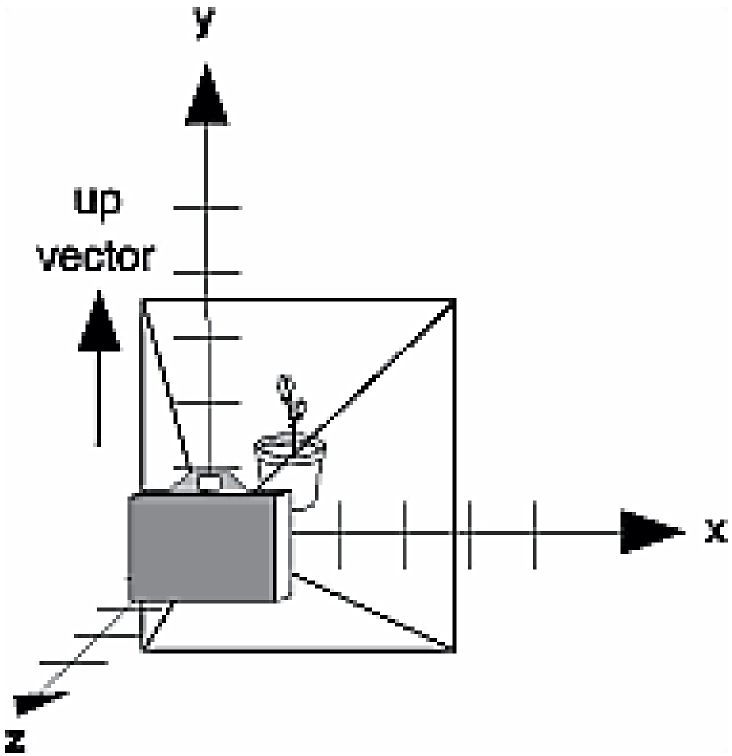


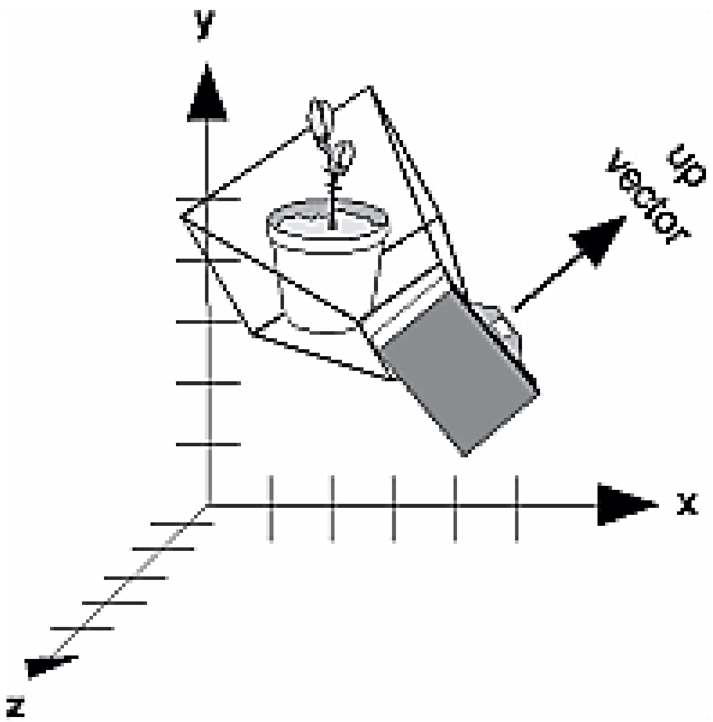
Define the camera frame in WCS: F(C,u,v,w)

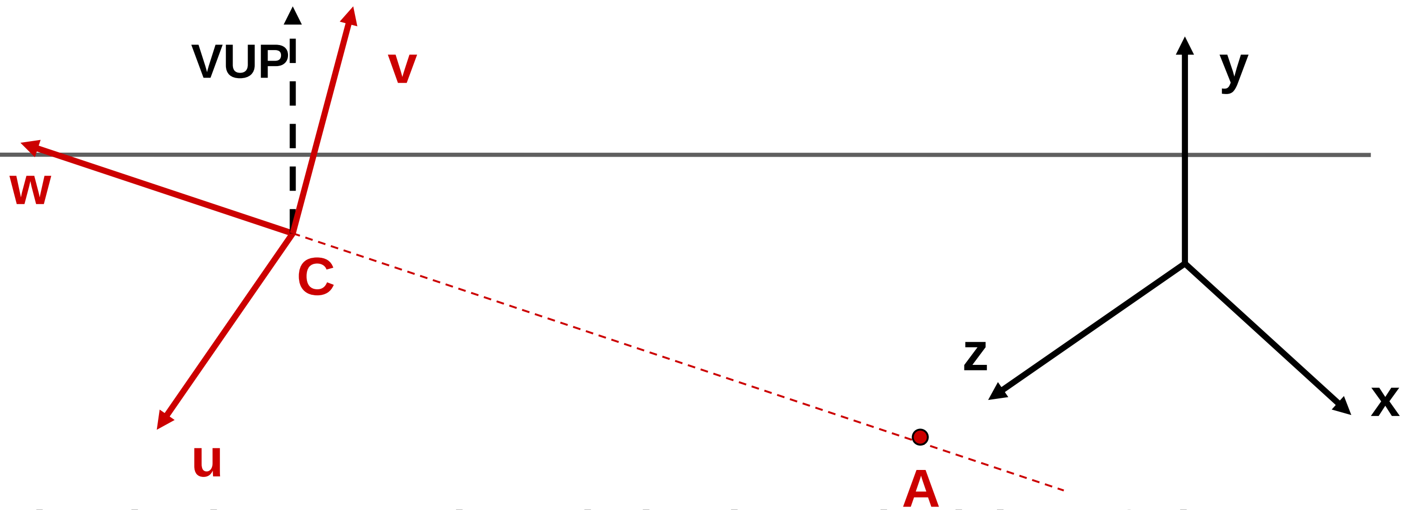
- **C** camera location (view point)
- **w**-*axis (Look Vector)*
  - the direction the camera is pointing
  - by convention the camera looks into a direction $-w$

$$w = \frac{C - A}{\left\| C - A \right\|}$$

- *View Up Vector (**VUP**)*
  - determines how the camera is rotated around the *Look vector (assume it 's parallel to y-axis)*


y
up
vector
x
z


y
up
vector
x
z



- **u-**axis (pointing on the right-hand side of the camera), is orthogonal to both **w-axis** and **VUP.**

$$u = \frac{VUP \times w}{\left\| VUP \times w \right\|}$$

- In case VUP || y-axis optimize the cross vector:

$$\begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \times w = \begin{bmatrix} w_z & 0 & -w_x \end{bmatrix}$$

- **v** axis orthogonal to both **w** and **u axes**

$$v = w \times u$$

- w and u normalized, v is already normalized (unit length)

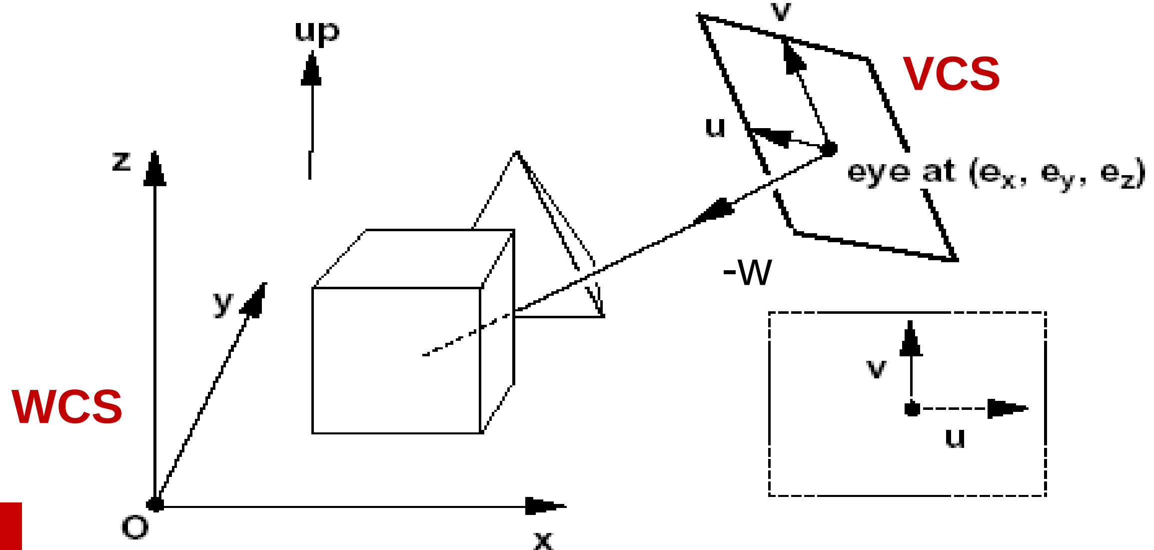
$$|v| = |u||w| \sin 90° = 1*1*1$$

Remark: there exist other methods for view specification, like flight simulation, rotating camera,...

Building Viewing Transformation from View Specification

- Given the frames WCS & VCS,
- **Compute the matrix transformation** Tv
- For each point P_w in WCS coord., convert its coordinates in VCS

$$\mathbf{P_v = T_v P_w}$$



Change of reference systems (frames) in 3D

Given the WCS (o,x,y,z) and VCS (C,u,v,w) represent P:

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & x_v & x_w & x_e \\ y_u & y_v & y_w & y_e \\ z_u & z_v & z_w & z_e \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ w_p \\ 1 \end{bmatrix}$$

$$P_{xyz} = \begin{bmatrix} u & v & w & e \\ 0 & 0 & 0 & 1 \end{bmatrix} P_{uvw}$$

$$P_{xyz} = M \; P_{uvw}$$

$$\begin{bmatrix} u_p \\ v_p \\ w_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & x_v & x_w & x_C \\ y_u & y_v & y_w & y_C \\ z_u & z_v & z_w & z_C \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix}$$

$$P_{uvw} = \begin{bmatrix} u & v & w & e \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} P_{xyz}$$

$$P_{uvw} = M^{-1} P_{xyz}$$

- The matrix **M** maps the WCS to the VCS frame.
- The change of coordinate of a vector/point from one system to another $\mathbf{v}_w$ in WCS and $\mathbf{v}_v$ in VCS is:

$$\mathbf{v}_w = \mathbf{M}\,\mathbf{v}_v$$

- The inverse matrix $(\mathbf{M})^{-1}$ from WCS to VCS

$$\mathbf{v}_v = (\mathbf{M})^{-1}\mathbf{v}_w = T_v\,\mathbf{v}_w$$

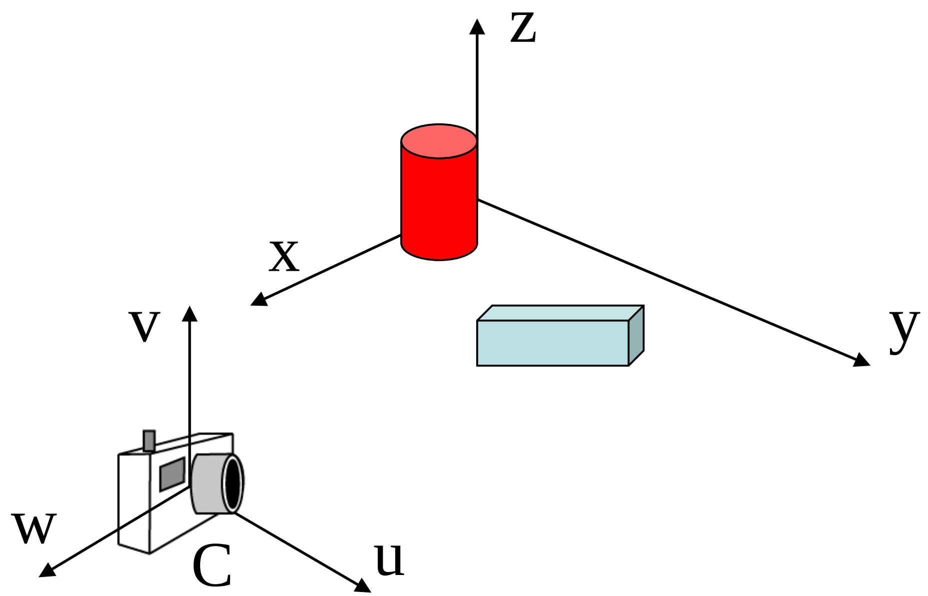
given a point $\mathbf{v}_w$ in homogeneous WCS coordinate provides its corresponding representation $\mathbf{v}_v$ in VCS coords.

Example: WCS->VCS

- WCS: (x,y,z,O)
- VCS: (x_v,y_v,z_v,C)
- Matrix M represents VCS w.r.t. WCS:

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

C location in WCS

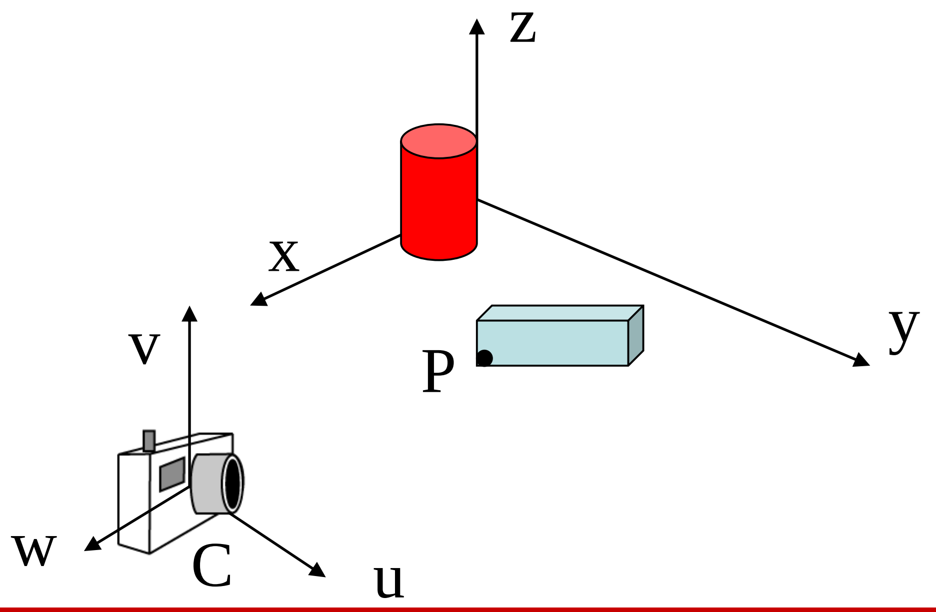


Example

- Given a point P_w $(x_p,y_p,z_p,1)$ in WCS

- The coords of P_v in VCS are given by

$$P_v = (M)^{-1}\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix}$$



Camera Space: VCS

- Let’s say we want to render an image of a chair from a certain camera’s point of view
- The chair is placed in world space with matrix $\mathbf{T_m}$
- The camera is placed in world space with matrix $\mathbf{T_v}$
- The following transformation takes vertices from the chair’s object space into world space, and then from world space into camera space:

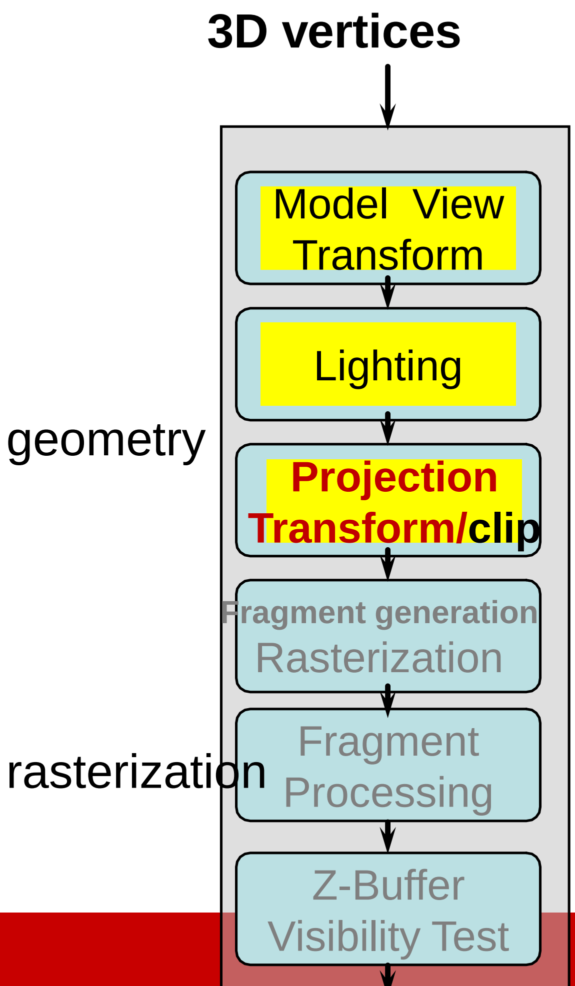
$$v' = T_v \cdot T_m \cdot v$$

- Now that we have the object transformed into a space relative to the camera, we can focus on the next step, which is to project this 3D space into a 2D image space

Projection transformation

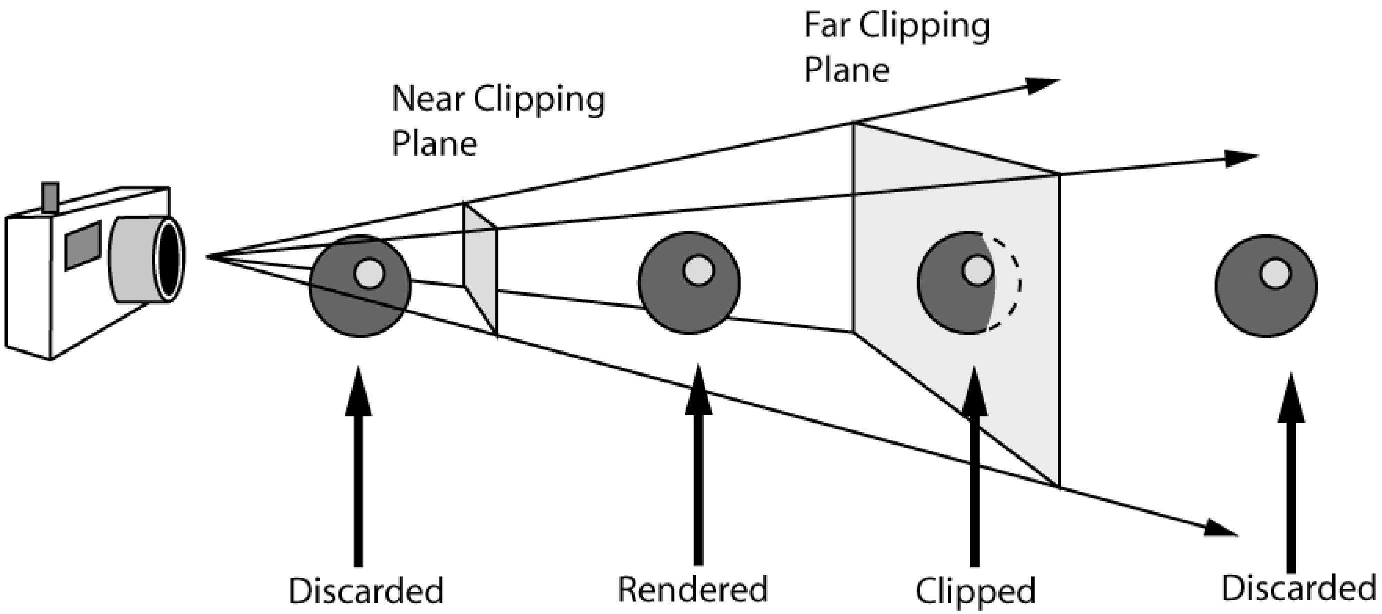
INPUT: 3D vertices in **VCS**

OUTPUT:
2D screen coordinates of visible vertices
Screen space

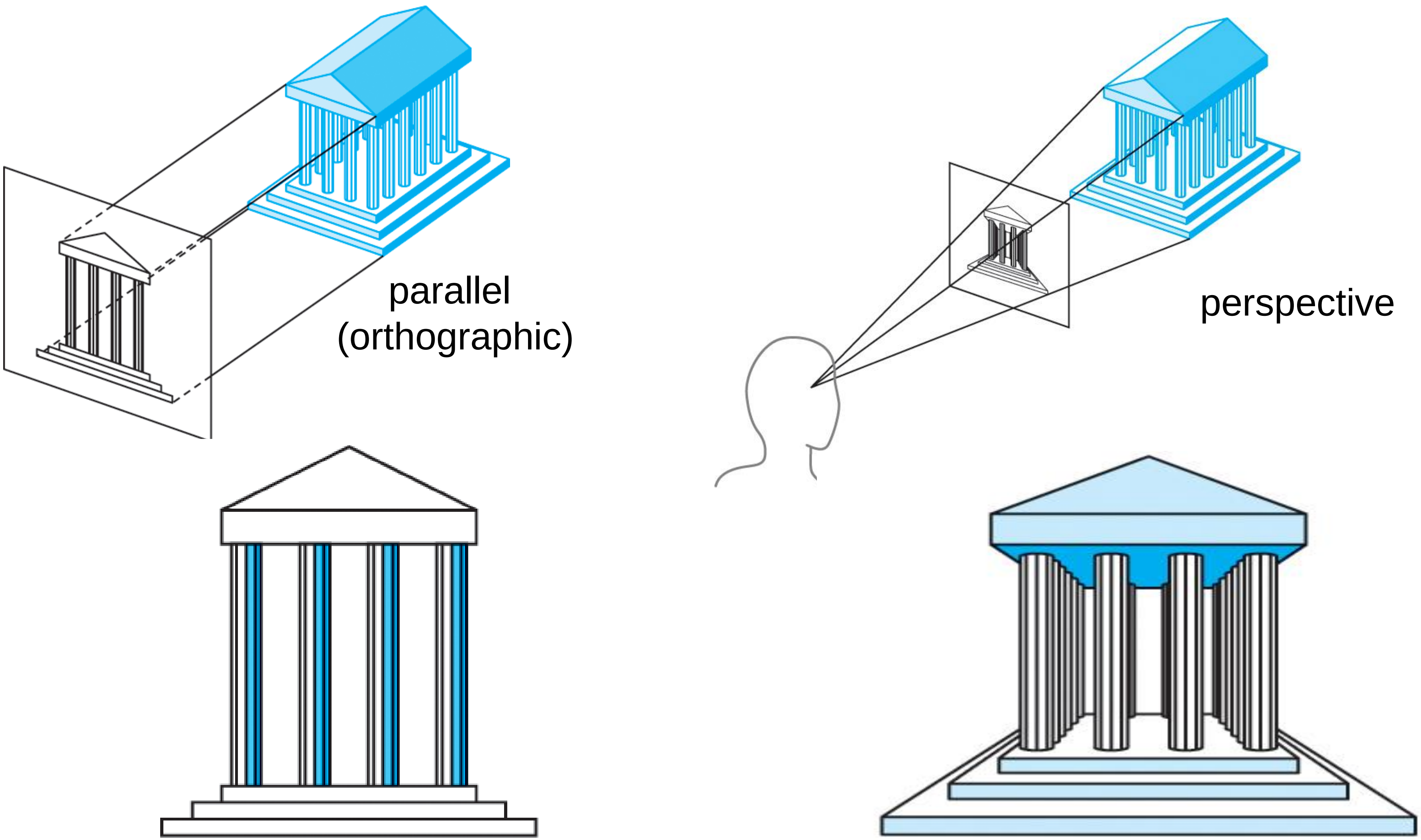


Clipping

- Volume of space between *Front* and *Back clipping planes* defines what camera can see
- Position of planes defined by distance along *Look vector*
- Objects appearing outside of view volume don’t get drawn
- Objects intersecting view volume get clipped



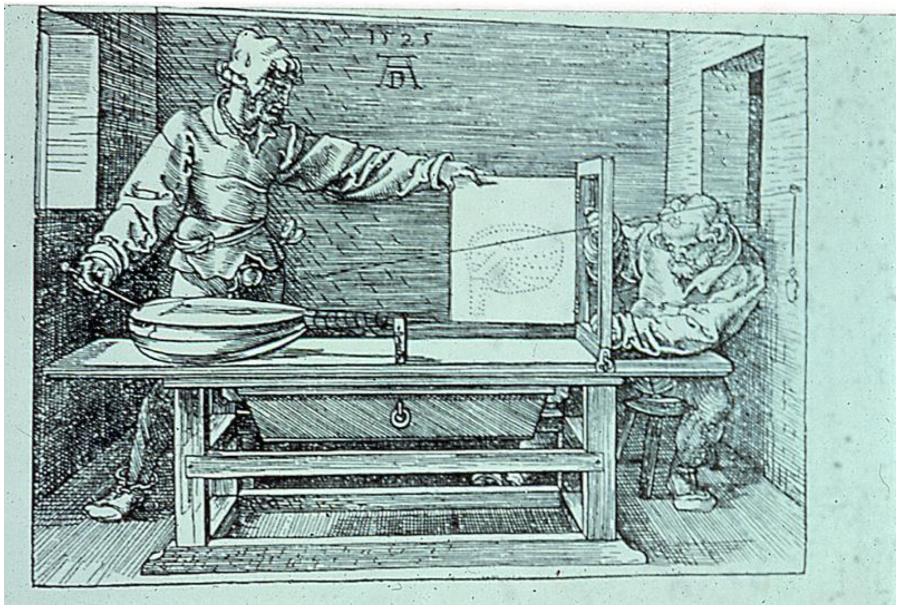
Projection



“Invention” of **Perspective** Geometry

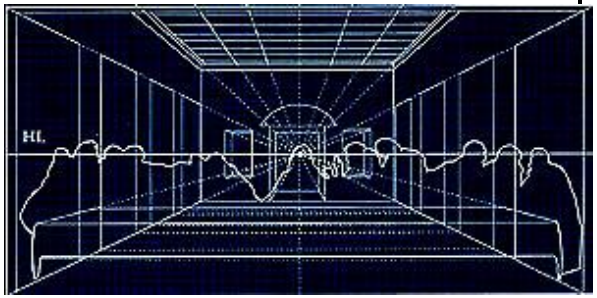
(geometria proiettiva)

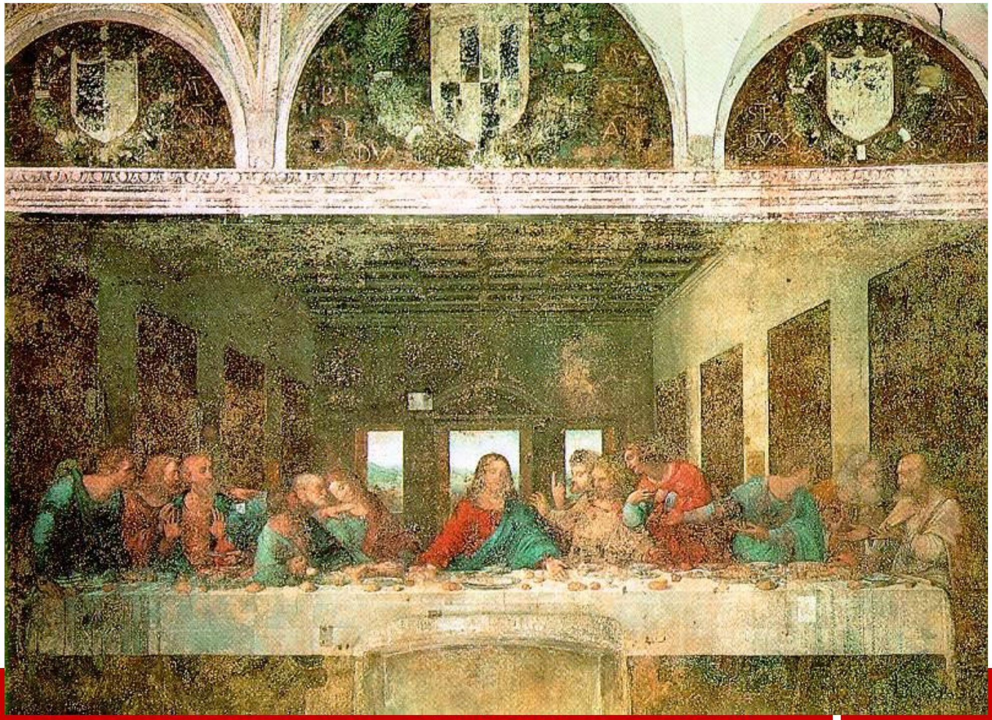
The Renaissance



Albrecht Dürer (1471-1528)
“Artista che disegna un liuto”
Artist Drawing a Lute

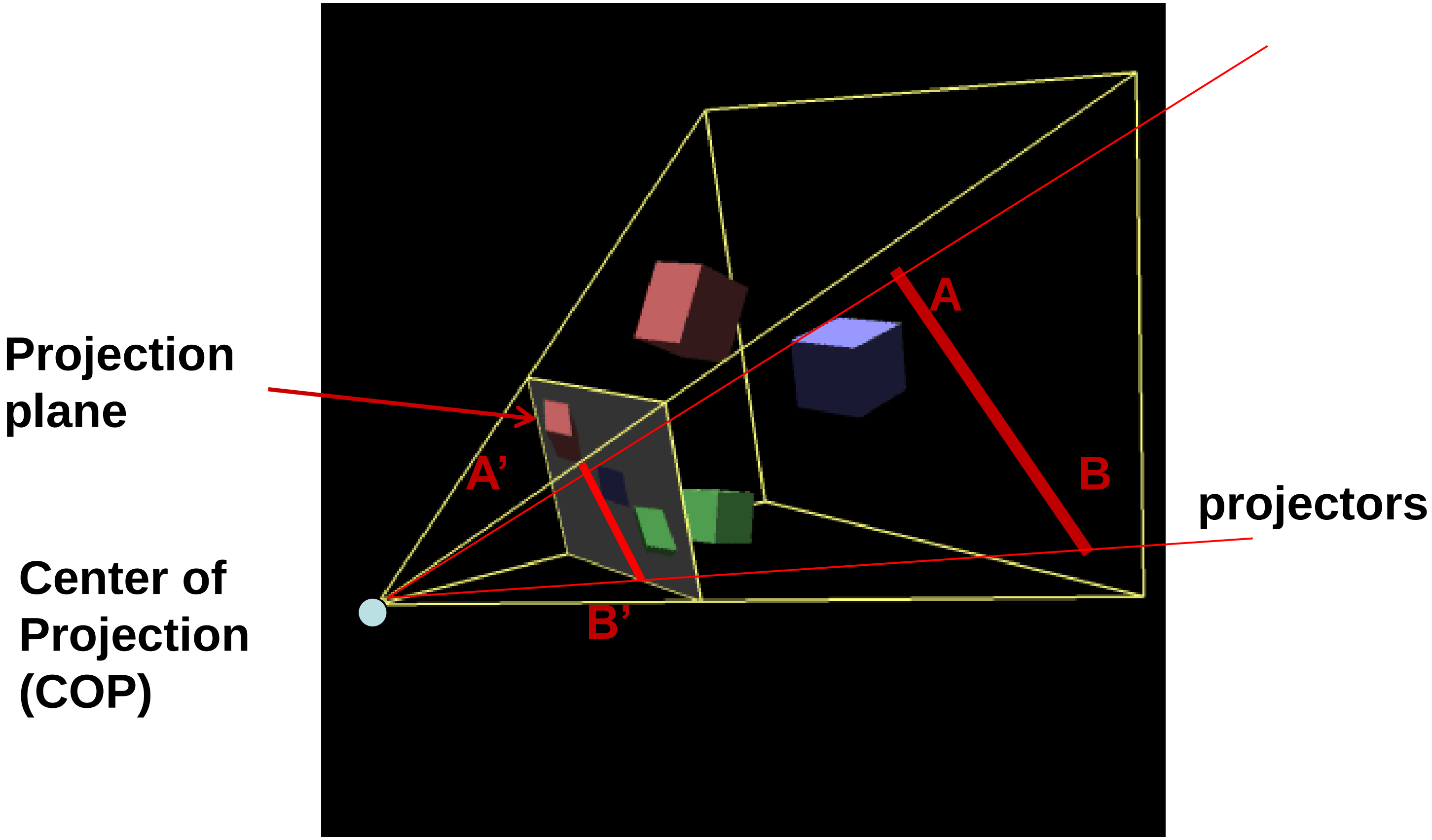
Leonardo da Vinci (1495)
“L’ultima cena” The Last Supper





Perspective Projection

Projectors are lines that converge at Center Of Projection (COP)



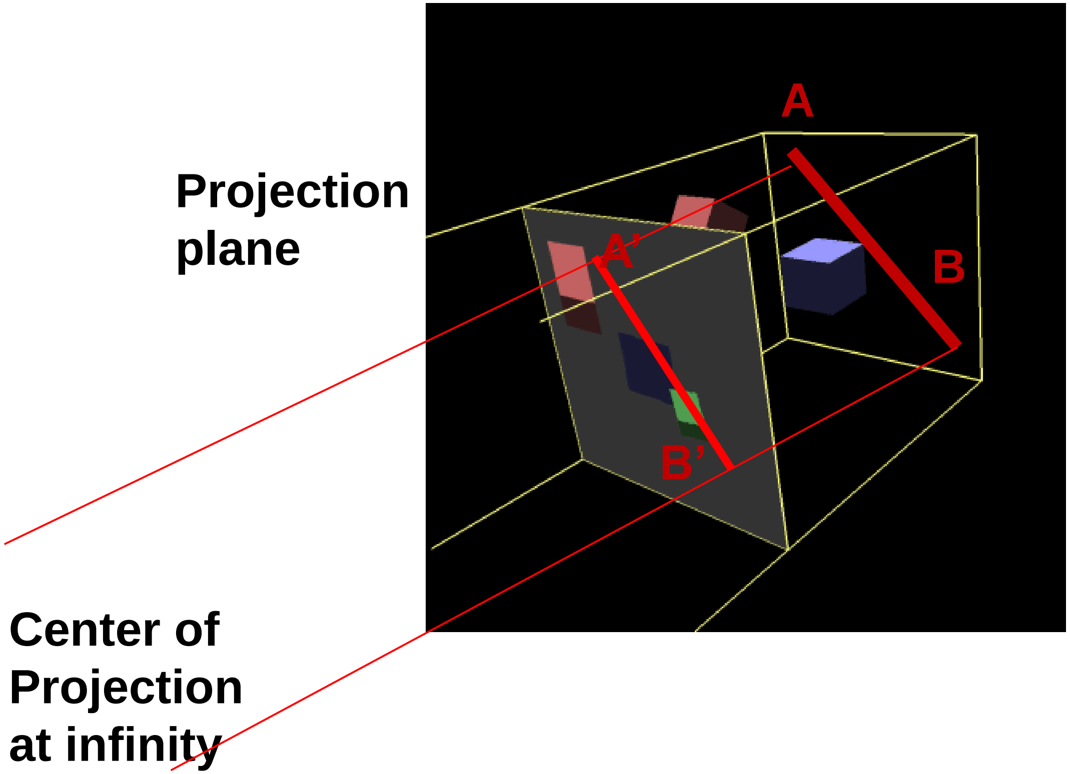
Properties of Perspective

- **Diminution** – Objects further from viewer are appear smaller than the same objects closer to the viewer
- **Foreshortening** – Equal distances along a line are not projected into equal distances on the image plane (it is not an affine transformation)
- **Angles** are preserved only in planes parallel to the projection plane
- It's **realistic**

Orthographic Projection

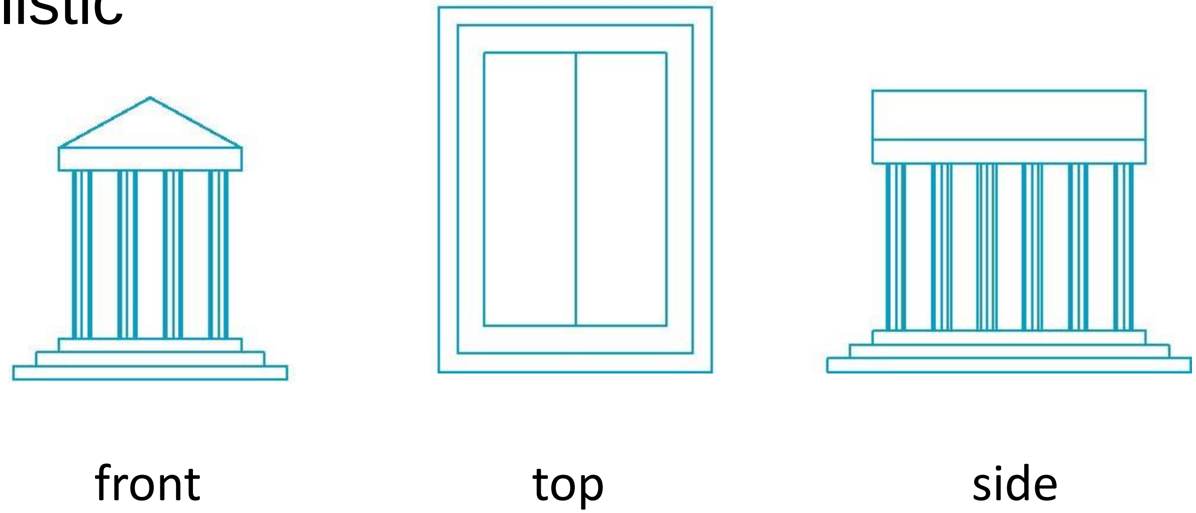
Viewer at infinity

Projectors are *parallel* lines *orthogonal* to projection plane



Multi-view Orthographic Projection (MOP)

- Projection plane is positioned parallel to axis-planes
- Usually form front, top, side views
- Often used for CAD and architecture
- Preserves both distances and angles (affine transf.)
- Not realistic



The projection transformation

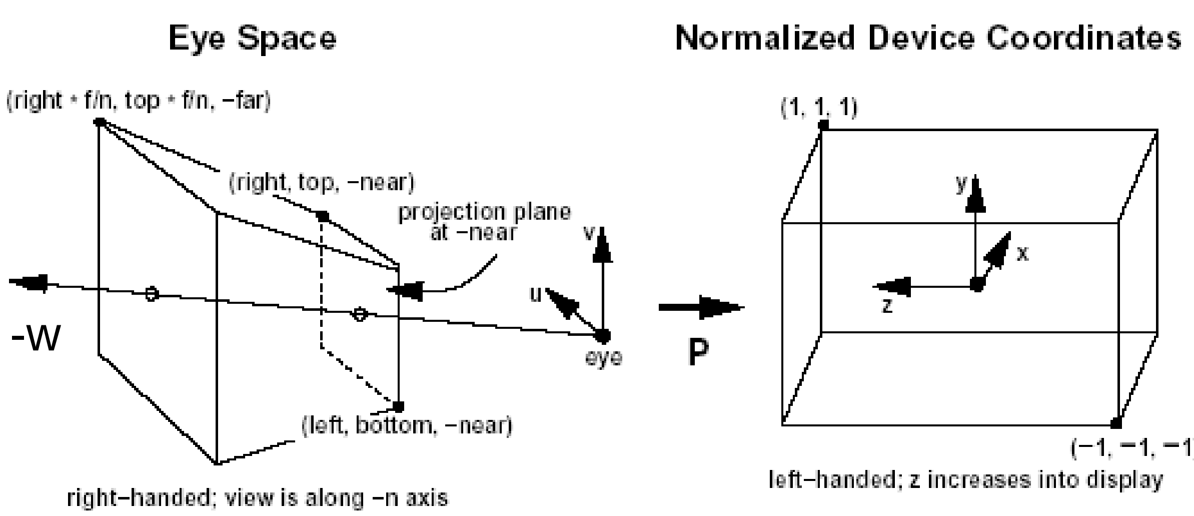
Before a real projection 3D -> 2D:

1. Determine the volume of space between *Front* and *Back clipping planes* which defines the bounded space that camera can “see” (**view volume**)

   The type of projection defines the shape of the **view volume**

2. Project the view volume into the **normalized coord. system**

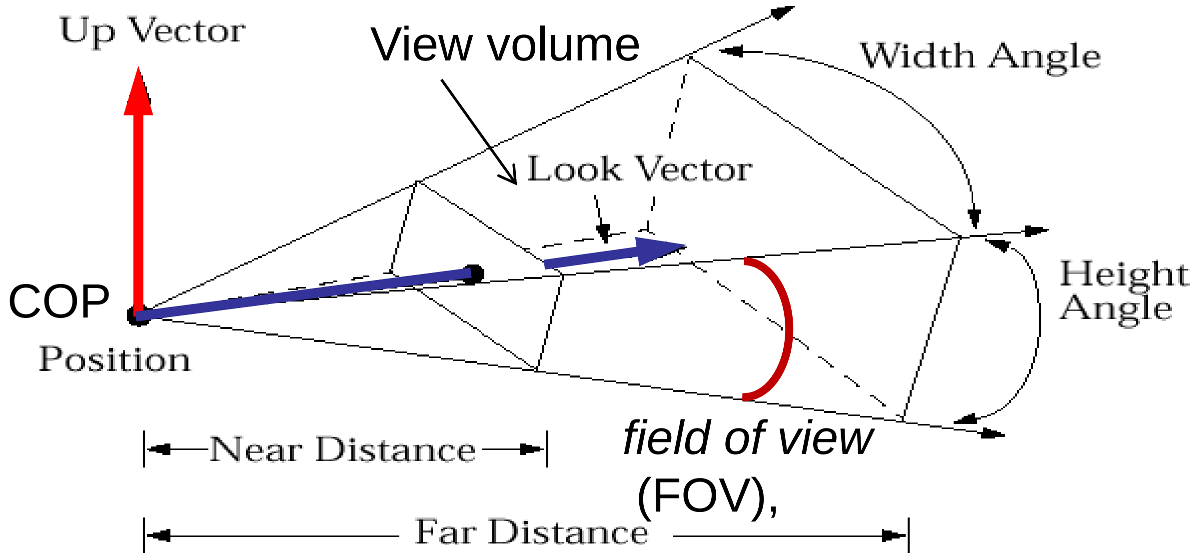
   (cuboid at the origin coords. lie in $[-1,1]^3$).



Clipping is more efficient in a cuboid axis-aligned: (-1,-1,-1) → (1,1,1)

View Volume

- **Perspective Projection**: Truncated Pyramid – **Frustum**

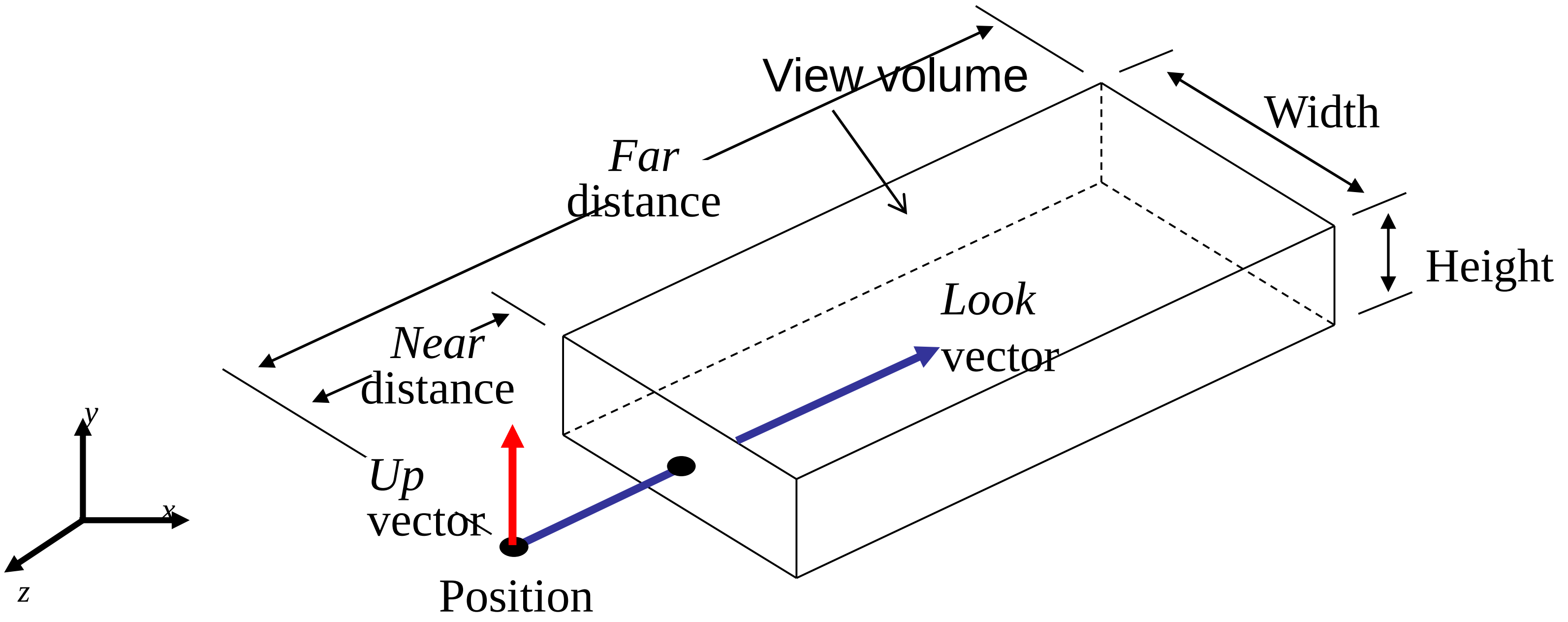


- *Look vector* is the center line of the pyramid,
- Aspect Ratio: determines proportion of width to height of image displayed on screen

aspect = **w/h**

View Volume

- **Orthographic Parallel Projection**: Truncated View Volume – Cuboid



- Orthographic parallel projection has no view angle parameter

Normalization: Project the view volume into the **normalized coord. system**

Transform the view volume into a parallel (cuboid) view volume (Image Space)

Why do we do it this way?

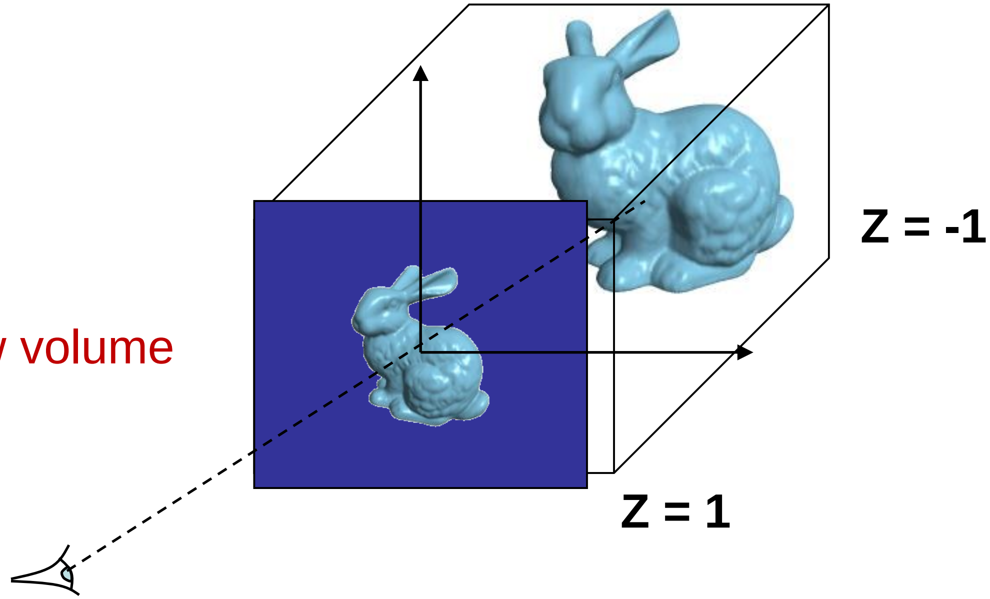
- Normalization allows for a single pipeline for both perspective and orthogonal viewing
- We stay in four dimensional homogeneous coordinates as long as possible to retain three-dimensional information needed for hidden-surface removal and shading
- We simplify both clipping and projection

Project the view volume into the **normalized coord. system (Image Space)**

Image Space

The scene is bounded by a cuboid at the origin with coord. x,y that lie in [-1, 1] and the z coordinate will also range from -1 to 1 and will represent the depth (1 being nearest and -1 being farthest)

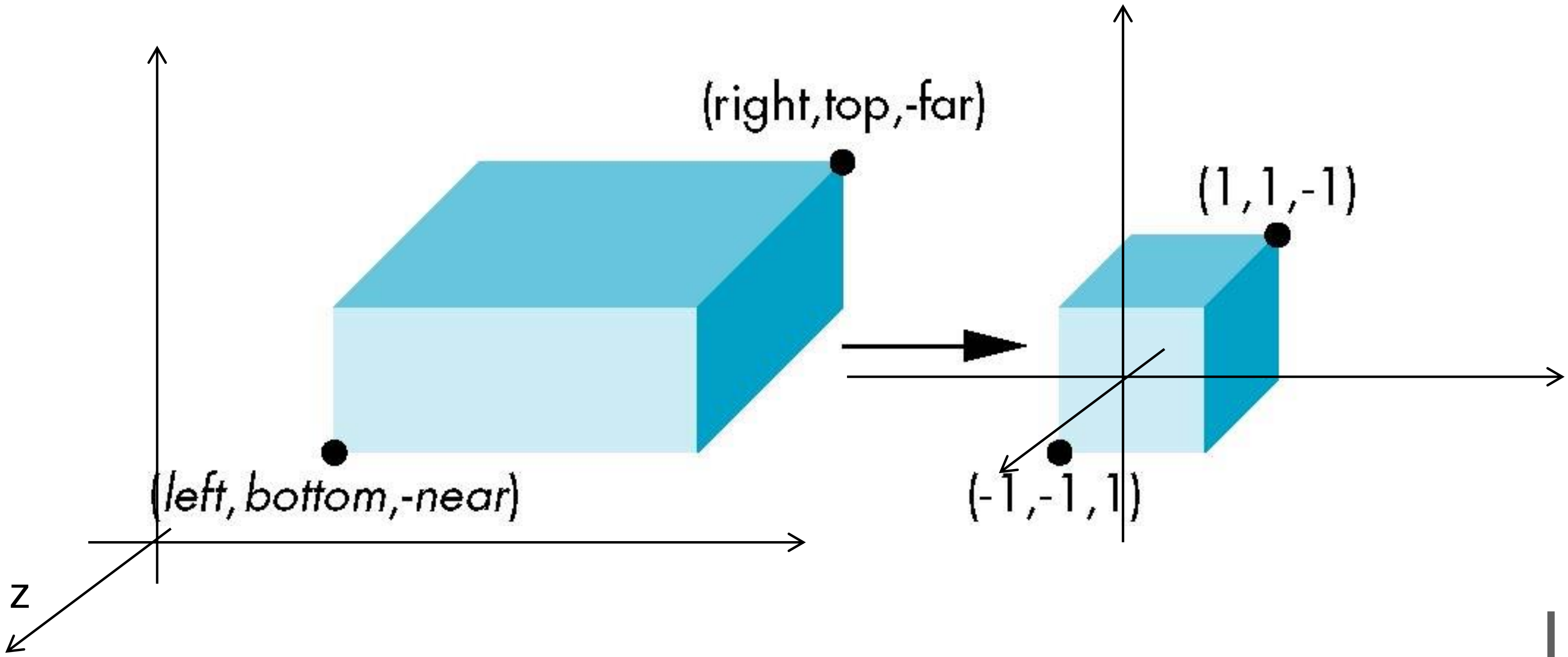
Canonical view volume



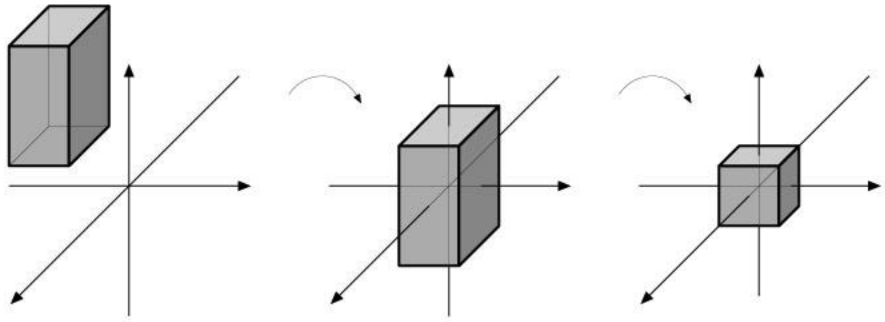
- The final 2D view of the 3D scene (the final image) will be finally computed by projecting the portion of scene contained in the Canonical view volume into a window in the image plane

Orthogonal normalization

Find transformation to convert specified clipping volume to **the Canonical View Volume**



Orthogonal Matrix



- Two steps
  - Move center to origin

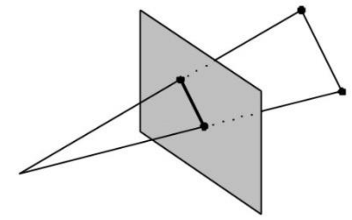
    T(-(left+right)/2, -(bottom+top)/2,(near+far)/2))
  - Scale to have sides of length 2

    S(2/(right-left),2/(top-bottom),2/(near-far))

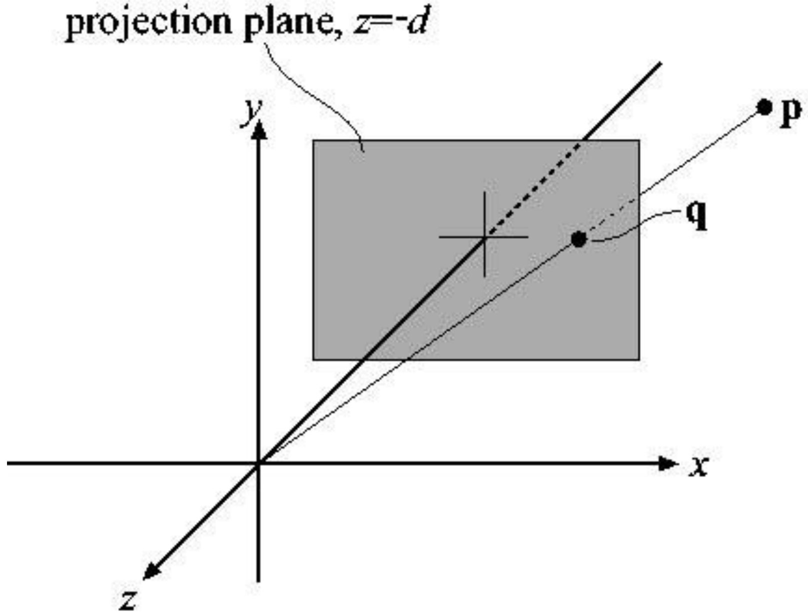
$$\mathbf{v}' = \mathbf{P} \cdot T_v \cdot T_m \cdot \mathbf{v}$$

$$\mathbf{P}\left(left, right, top, bottom, near, far\right) = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{2}{near-far} & \frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

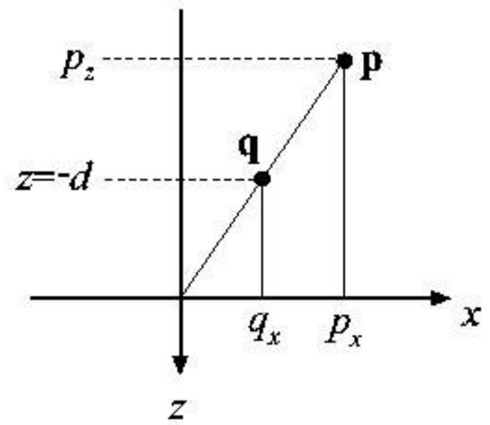
Perspective projection



d>0



Consider top view:

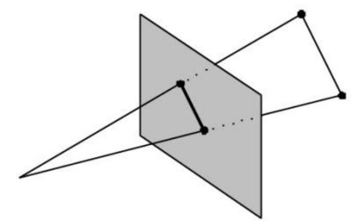


Based on
similar triangles

$$\frac{q_x}{p_x} = \frac{-d}{p_z} \Rightarrow q_x = -d\,\frac{p_x}{p_z}$$

Analogously for y: $q_y = -d\,\dfrac{p_y}{p_z}$

Perspective Projection



In homogeneous coords we express projection as 4x4 transform matrix

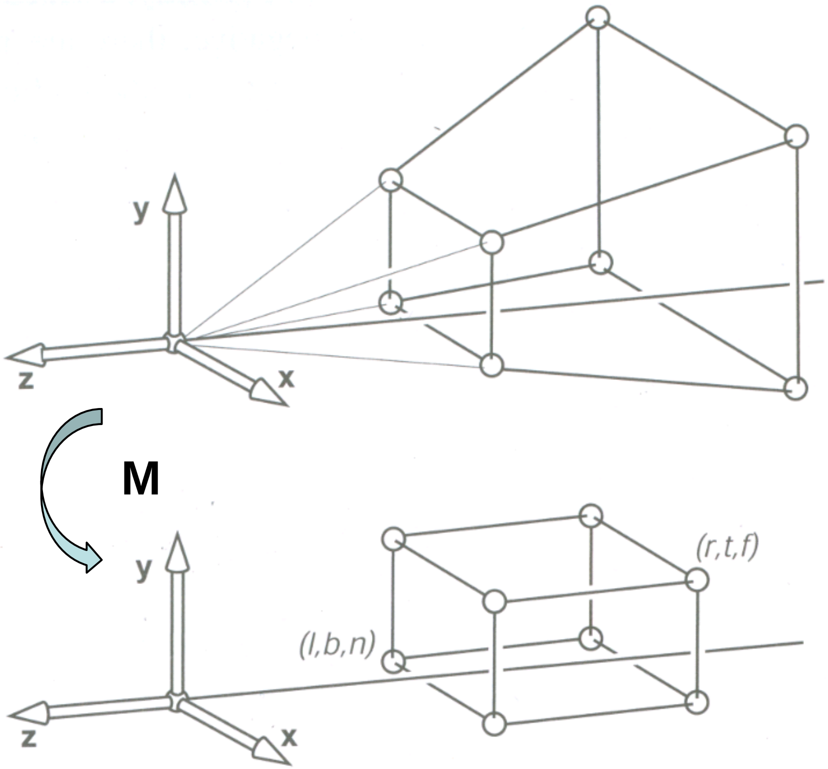
$$\mathbf{P}_p = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix} \qquad \mathrm{q} = \mathbf{P}_p\mathbf{p}$$

$$\mathbf{P}_p\mathbf{p} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix}\begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ p_z \\ -p_z/d \end{pmatrix} \Rightarrow \quad \mathbf{q} = \begin{pmatrix} -dp_x/p_z \\ -dp_y/p_z \\ -dp_z/p_z \\ 1 \end{pmatrix} = \begin{pmatrix} -d\frac{p_x}{p_z} \\ -d\frac{p_y}{p_z} \\ -d \\ 1 \end{pmatrix}$$

$$q_x = -d\frac{p_x}{p_z} \qquad q_y = -d\frac{p_y}{p_z}$$

- The ”arrow” is the homogenization process

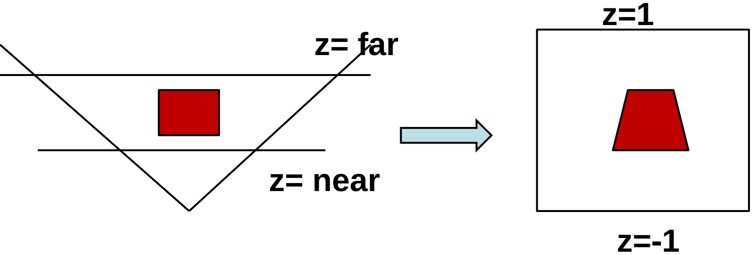
Perspective projection

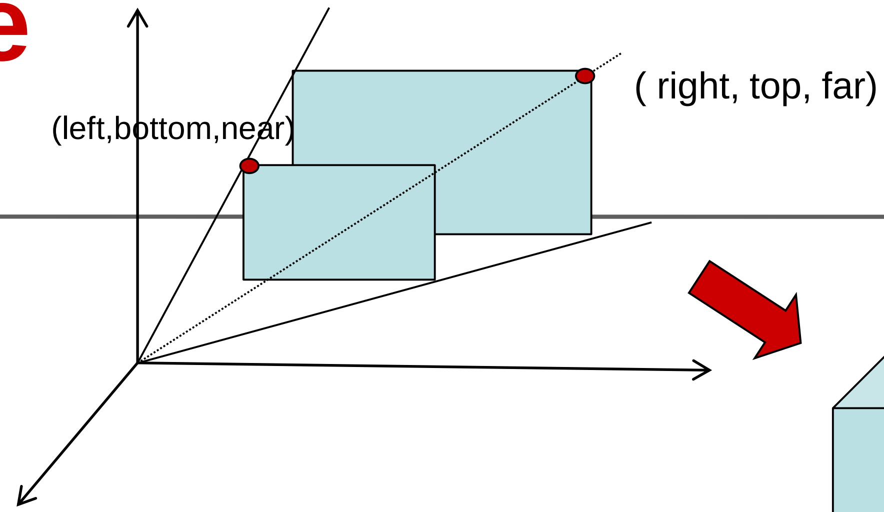


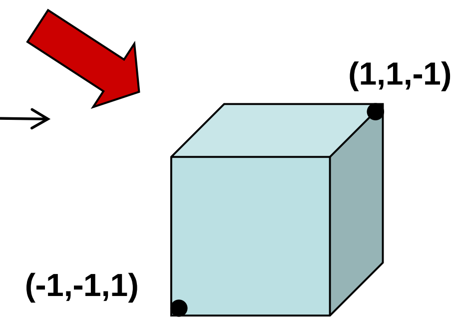
$P_{pers} = P_{ortho} M$

- In a sense, we can think of this view frustrum as a distorted cube, since it has six faces, each with 4 sides
- The perspective projection leaves points on the z=n plane unchanged and maps the large z=f rectangle at the back of the perspective volume to the small rectangle at the back of the orthographic volume
- We need a way to represent this transformation mathematically

Perspective Projection







$$v'_{(4D)} = \mathrm{P} \cdot T_v \cdot T_m \cdot \mathrm{v}$$

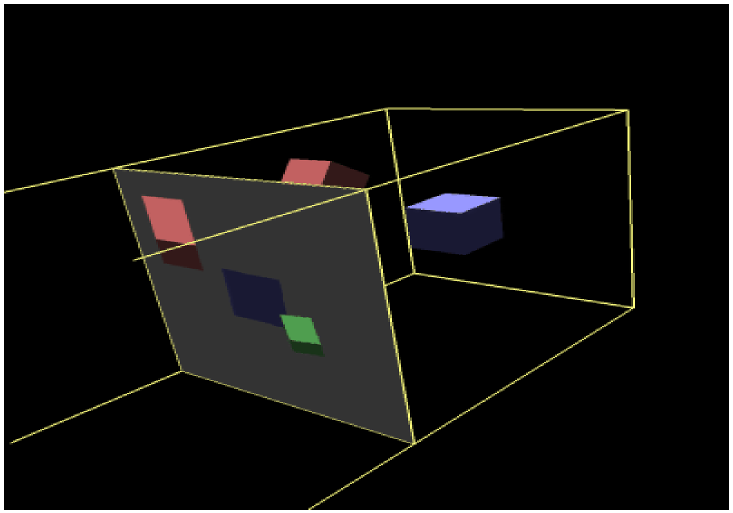
$$\mathrm{P}(left, right, top, bottom, near, far) =$$

$$\begin{bmatrix} \frac{2near}{right-left} & 0 & \frac{left+right}{left-right} & 0 \\ 0 & \frac{2near}{top-bottom} & \frac{bottom+top}{bottom-top} & 0 \\ 0 & 0 & \frac{far+near}{near-far} & \frac{2\,far\,near}{far-near} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

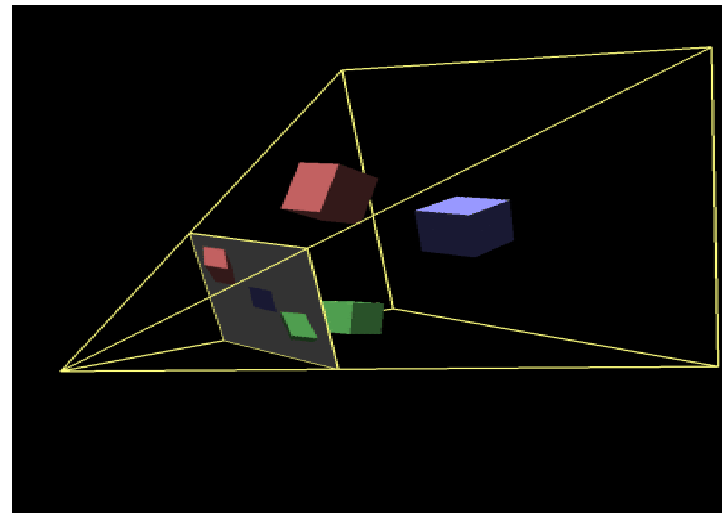
From frustum to canonical view volume (cuboid)

-1 <=x,y,z<=1

Orthographic View



Perspective View



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

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

Effects of the Perspective Projection on points in homogeneous coord.

- If we look at the perspective matrix, we see that it doesn't have [0 0 0 1] on the bottom row

- This means that when we transform a 3D position vector [v_x v_y v_z 1], we will not necessarily end up with a 1 in the 4th component of the result vector

- Instead, we end up with a true 4D vector [v_x' v_y' v_z' v_w']
- The final step of perspective projection is to map this 4D vector back into the 3D w=1 subspace:

$$\begin{bmatrix} v_x & v_y & v_z & v_w \end{bmatrix} \Rightarrow \begin{bmatrix} \frac{v_x}{v_w} & \frac{v_y}{v_w} & \frac{v_z}{v_w} \end{bmatrix} \Rightarrow \begin{bmatrix} \frac{v_x}{v_w} & \frac{v_y}{v_w} & \frac{v_z}{v_w} & 1 \end{bmatrix}$$

$$\mathbf{v}'_{(4D)} = \mathbf{P} \cdot T_v \cdot T_m \cdot \mathbf{v}$$

$$\mathbf{v}'' = \begin{bmatrix} \frac{v'_x}{v'_w} & \frac{v'_y}{v'_w} & \frac{v'_z}{v'_w} & 1 \end{bmatrix}$$

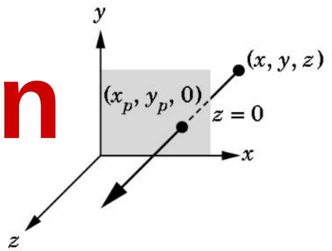
$$\mathbf{v}''' = \mathbf{D} \cdot \mathbf{v}''$$

D ??

Window Transformation and

Window-viewport Transformation

Windowing Transformation



- **The final 2D image is obtained by merely dropping the z coordinate after orthogonal projection into plane z=0**
- Window transformation is an orthogonal projection that maps points in NCS (x,y,z) in $[-1,1]^3$ (image space 3D) into a rectangular region (x,y) in $[-1,1]^2$(window 2D)

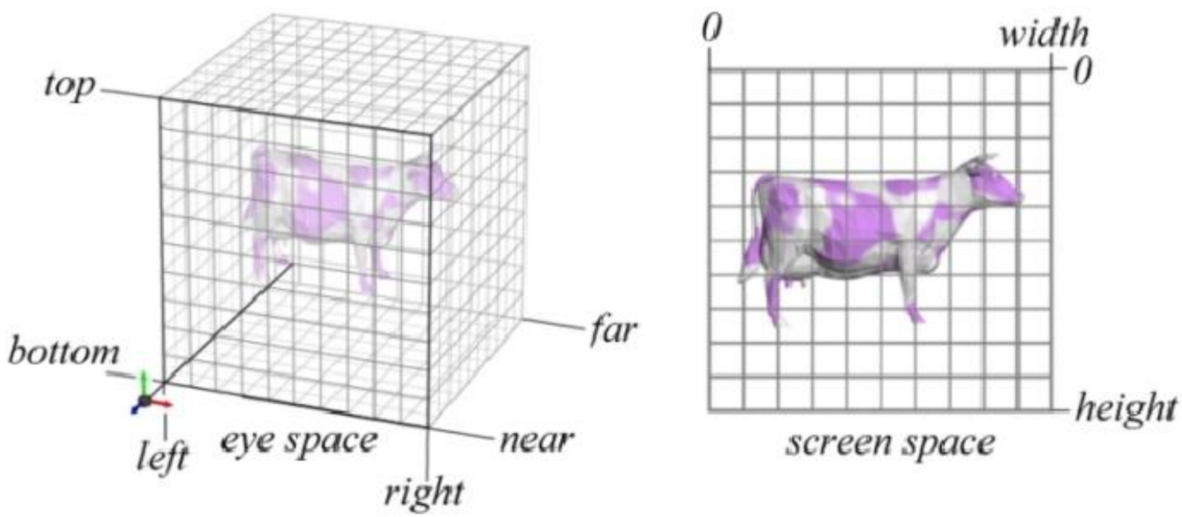
$$P_{ortho} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & \boxed{0} \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \Rightarrow \qquad P_{ortho} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ 0 \\ 1 \end{pmatrix}$$

- Projected points into the view plane (z=0) maintain the x,y, coord. but z=0

Windowing Transformation

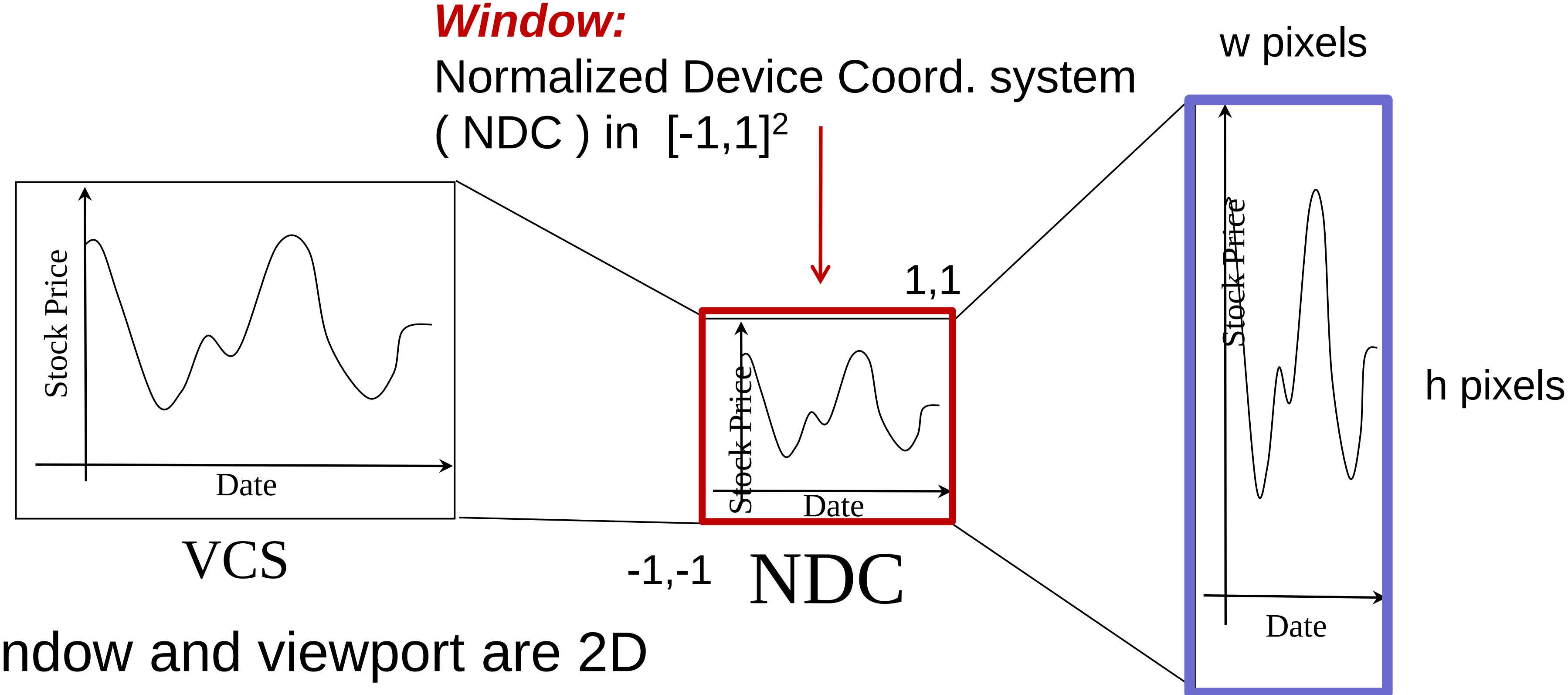
$$\mathbf{v}'_{(2D)} = P_{ortho} \cdot \mathbf{P} \cdot T_v \cdot T_m \cdot \mathbf{v}$$

- The depth value (z) is usually mapped to a 32 bit fixed point value ranging from 0 (near) to 0xffffffff (far)
- Finally, transformation window-viewport..



Window-Viewport Transformation

Window:
Normalized Device Coord. system
(NDC) in [-1,1]^2

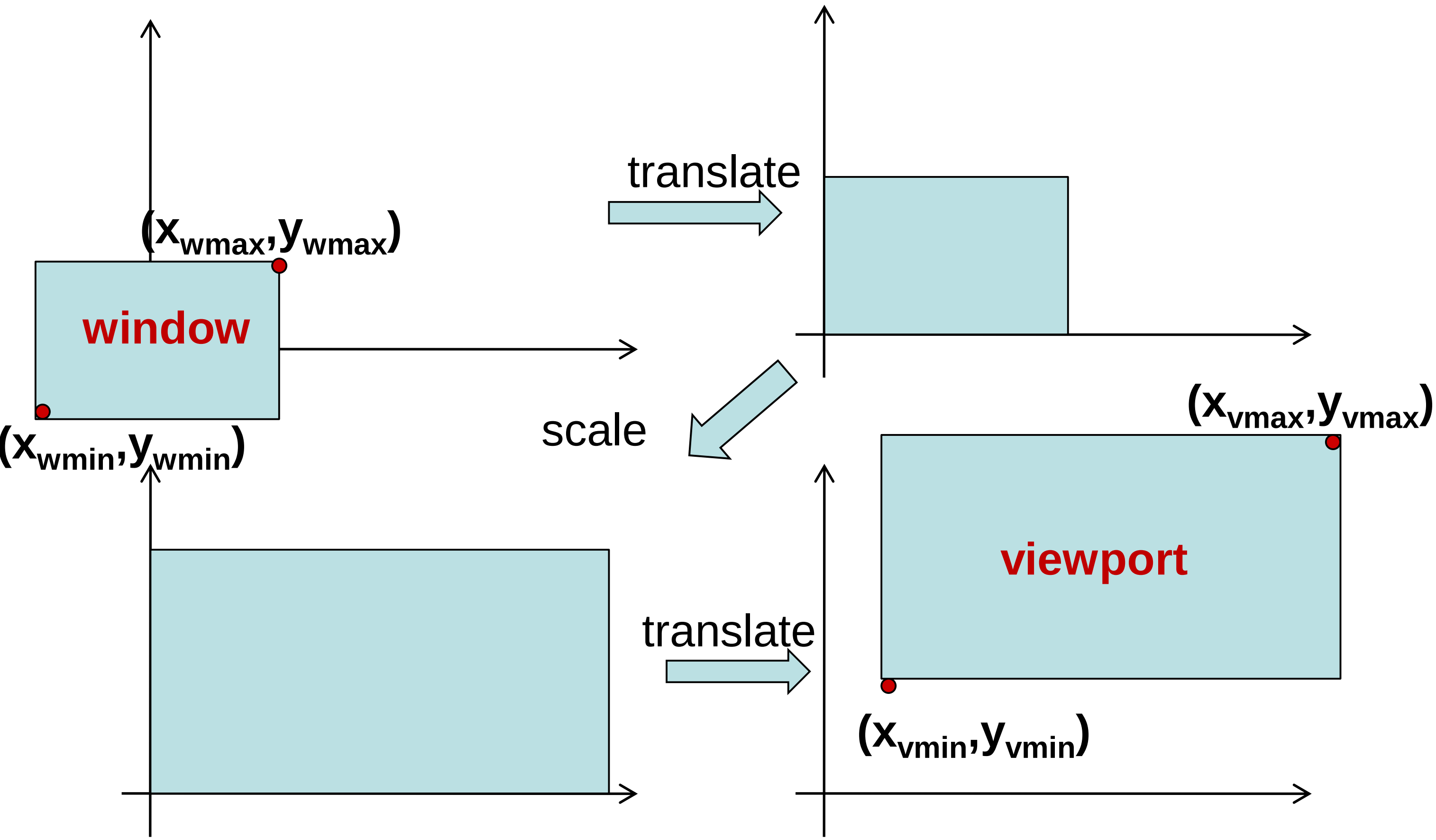


Viewport
Coordinates

Window and viewport are 2D rectangular regions.
Viewport is a 2D integer coordinate region of screen space to which the window contents are mapped.

Viewport:
(Screen Coord. System in pixel)

Window-Viewport Transformation



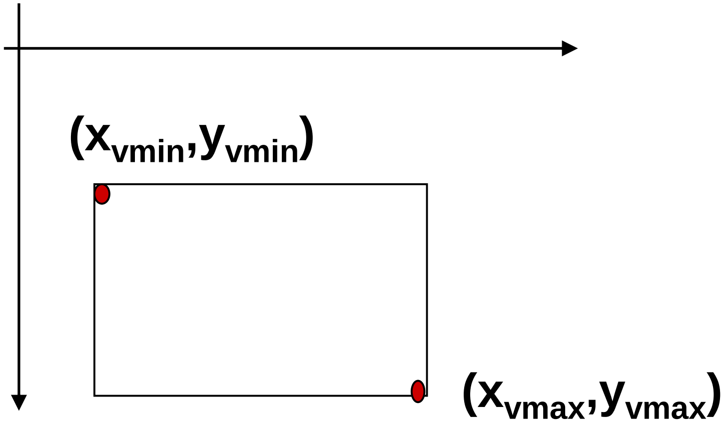
Window-Viewport Transformation

$$x_v = ceil(S_x(x_w - x_{w\min}) + x_{v\min})$$

$$y_v = ceil(S_y(y_{w\min} - y_w) + y_{v\max})$$

$$S_x = \frac{x_{v\max} - x_{v\min}}{x_{w\max} - x_{w\min}} \qquad S_y = \frac{y_{v\max} - y_{v\min}}{y_{w\max} - y_{w\min}}$$

- Every y-coord in window is up side down in viewport



Window-Viewport Transformation

In matrix form:

$$T_{wv} = \begin{bmatrix} 1 & 0 & x_{v\min} \\ 0 & 1 & y_{v\min} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_{w\min} \\ 0 & 1 & -y_{w\min} \\ 0 & 0 & 1 \end{bmatrix} =$$

$$= \begin{bmatrix} S_x & 0 & x_{v\min} - S_x x_{w\min} \\ 0 & S_y & y_{v\min} - S_y y_{w\min} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & \dfrac{x_{v\min} x_{w\max} - x_{v\max} x_{w\min}}{x_{w\max} - x_{w\min}} \\ 0 & S_y & \dfrac{y_{v\min} y_{w\max} - y_{v\max} y_{w\min}}{y_{w\max} - y_{w\min}} \\ 0 & 0 & 1 \end{bmatrix}$$

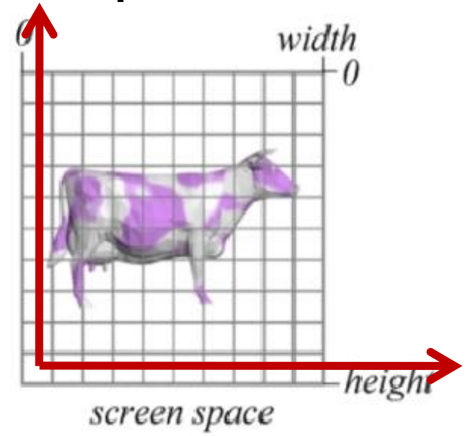
Window-Viewport Transformation: the infamous half pixel

$$viewport:(n_x,n_y)\quad \text{pixels}$$

$$\underbrace{[-1,1]\times[-1,1]}_{window}\Rightarrow\underbrace{[-0.5,\mathrm{n_x}-0.5]\times[-0.5,\mathrm{n_y}-0.5]}_{viewport}$$

$$\mathrm{n_x}=x_{v\max}-x_{v\min},\ \mathrm{n_y}=y_{v\max}-y_{v\min}$$

$$\begin{bmatrix}x_v\\y_v\\0\\1\end{bmatrix}=\begin{bmatrix}\frac{n_x}{2}&0&0&\frac{n_x-1}{2}\\0&\frac{n_y}{2}&0&\frac{n_y-1}{2}\\0&0&0&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}x_w\\y_w\\0\\1\end{bmatrix}$$



$$(x_w,y_w)\in window$$

$$(x_v,y_v)\in viewport$$

Each pixel owns a unit square centered at integer coordinates

$$v_{(screen)}=T_{wv}\cdot P_{ortho}\cdot P\cdot T_v\cdot T_m\cdot v$$

- Inverse Transform viewport-window??

- Select points on the screen
- Simulate zooming in a viewport region

- Panning

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Serena Morigi

Dipartimento di Matematica

serena.morigi@unibo.it

http://www.dm.unibo.it/~morigi