3D Viewing

Joaquim Madeira

March 2024

Topics

- Recap
- Projections
- Matricial Representation
- View Volume & Clipping
- Visible-Surface Determination
- Three.js – Cameras

RECAP

CG Main Tasks

- Modeling
  - Construct individual models / objects
  - Assemble them into a 2D or 3D scene
- Animation
  - Static vs. dynamic scenes
  - Movement and / or deformation
- Rendering
  - Generate final images
  - Where is the viewer / camera ?
  - How is he / she looking at the scene?

Modeling vs Rendering

- Modeling
  - Create models
  - Apply materials to models
  - Place models around scene
  - Place lights in the scene
  - Place the camera

YouTube Demo

- Rendering
  - Take picture with the camera

[van Dam]

Transformations

- Position, rotate and scale objects

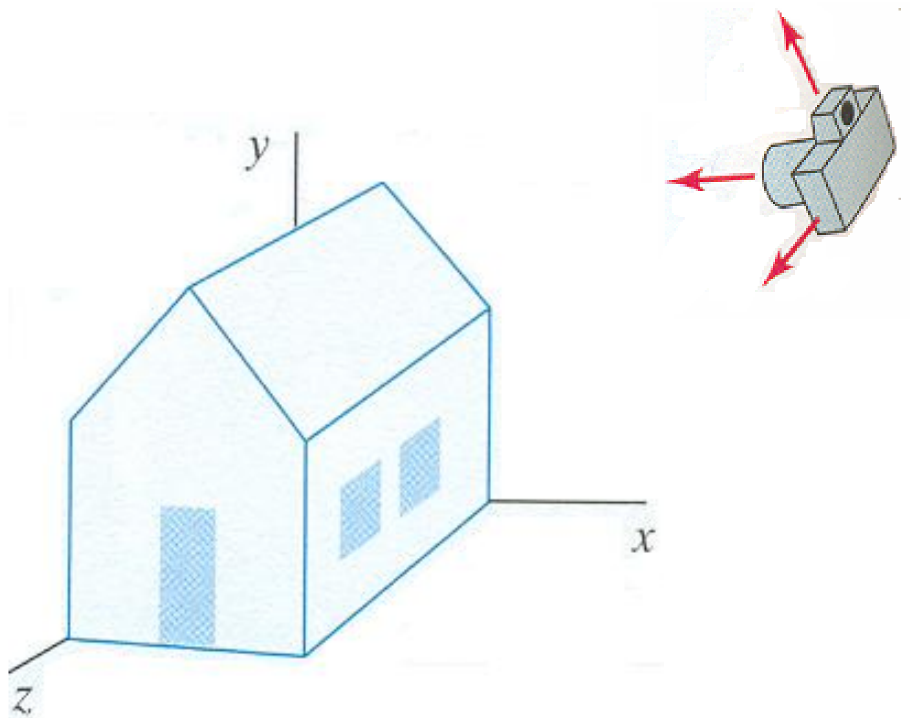
- Basic transformations
  - Translation
  - Rotation
  - Scaling

- Matricial representation
  - Homogeneous coordinates !!
  - Concatenation = Matrix products

- Complex transformations ?
  - Decompose into a sequence of basic transformations

3D VIEWING

3D Viewing



3D Viewing

- Where is the observer / the camera ?
  - Position ?
  - Close to the 3D scene ?
  - Far away ?

- How is the observer looking at the scene ?
  - Orientation ?

- How the represent as a 2D image ?
  - Projection ?

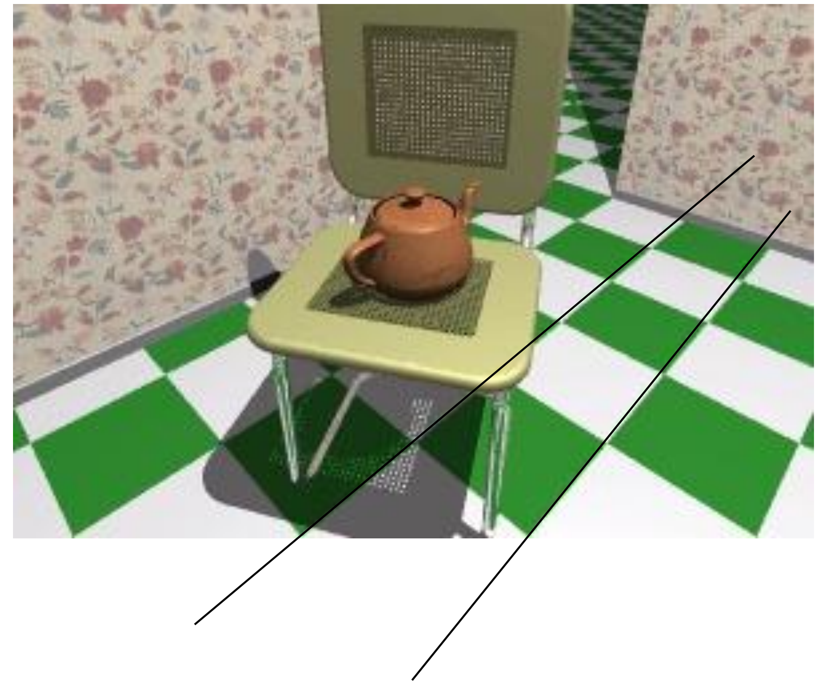
Three.js – The camera

```
// The CAMERA

// --- Where the viewer is and how he is looking at the scene

camera = new THREE.PerspectiveCamera( 70,

                    window.innerWidth / window.innerHeight, 1, 1000 );

camera.position.z = 400;
```

PROJECTIONS

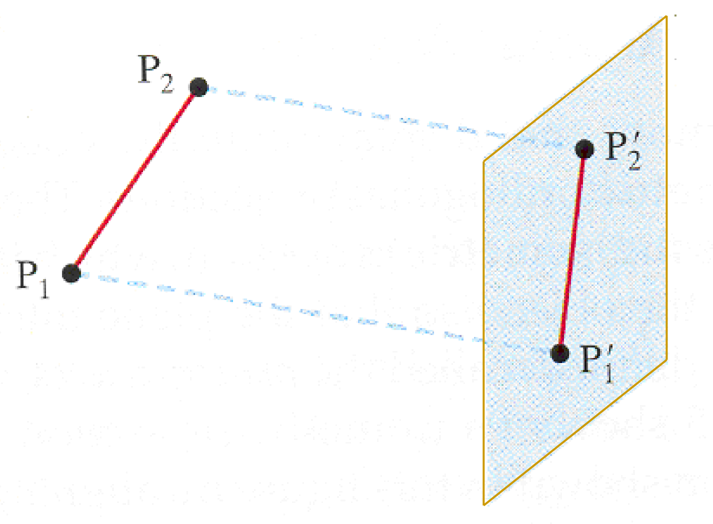
Projections

Parallel Projection

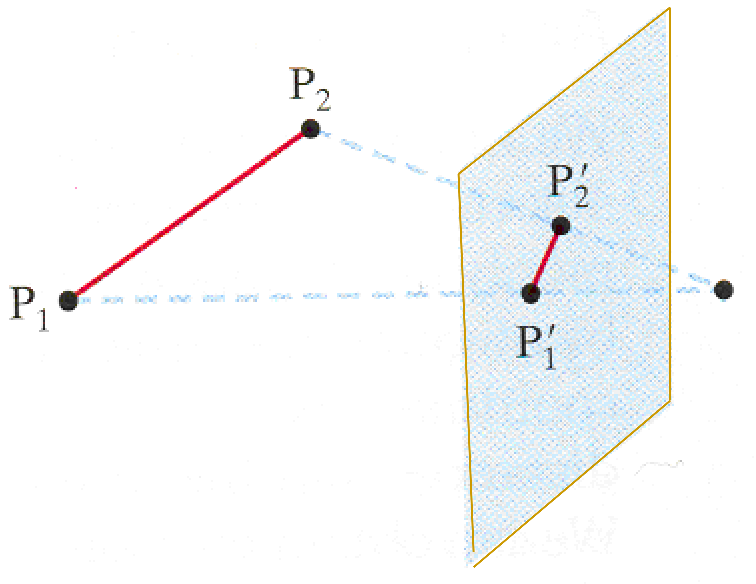


Perspective Projection

Projections

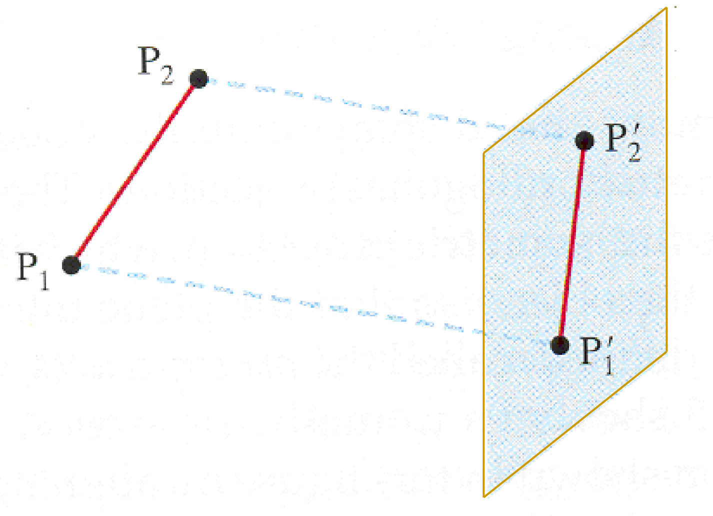


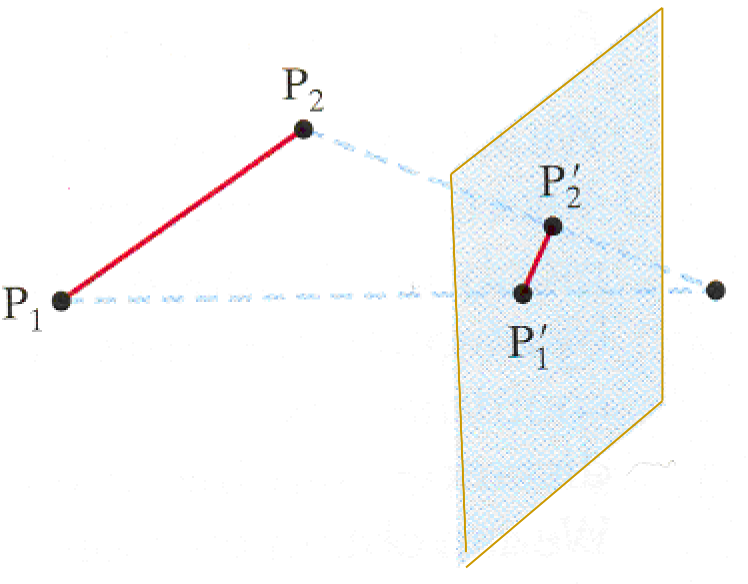
Parallel Projection



Perspective Projection

Projections

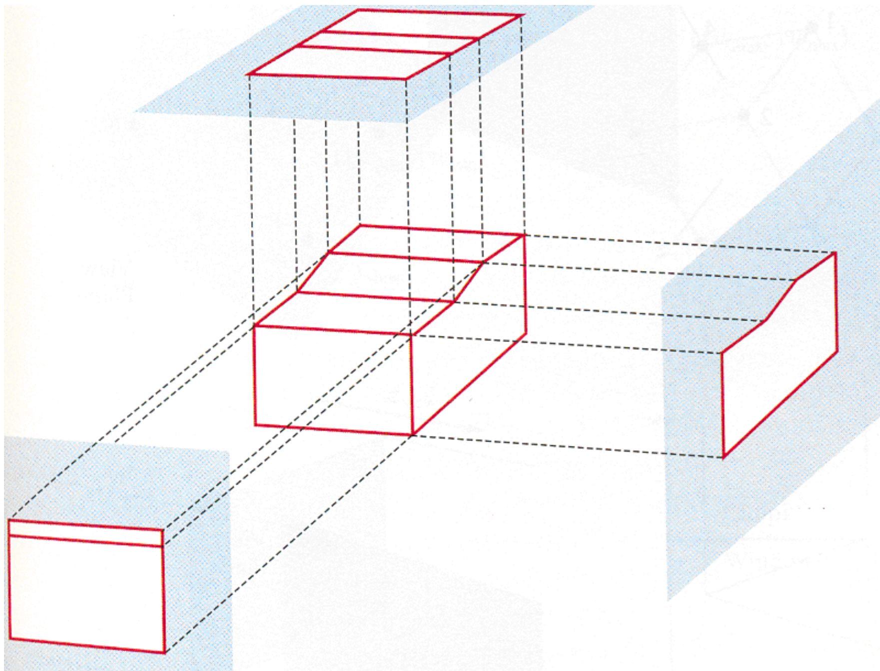


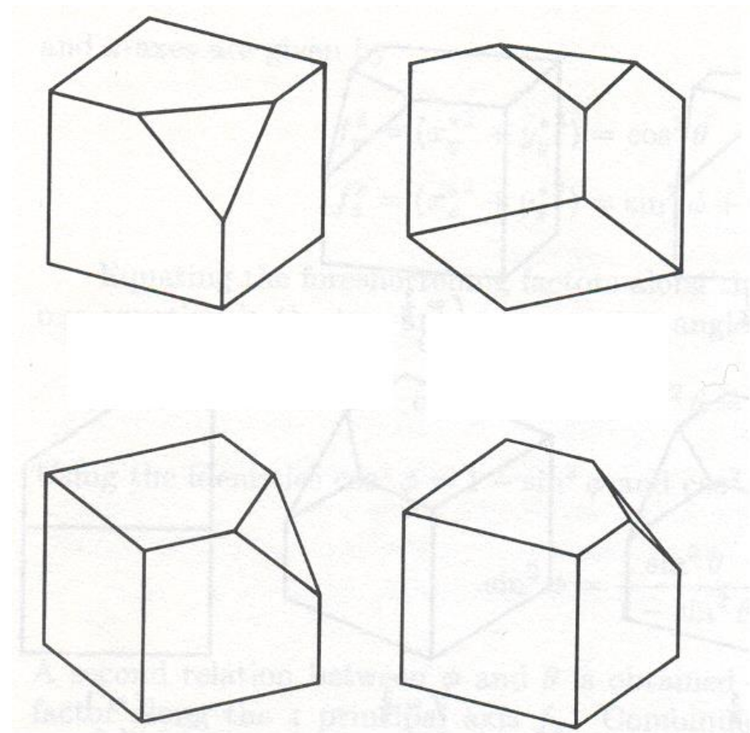


The projector straight-lines are parallel, i.e., converge at an indefinite distance

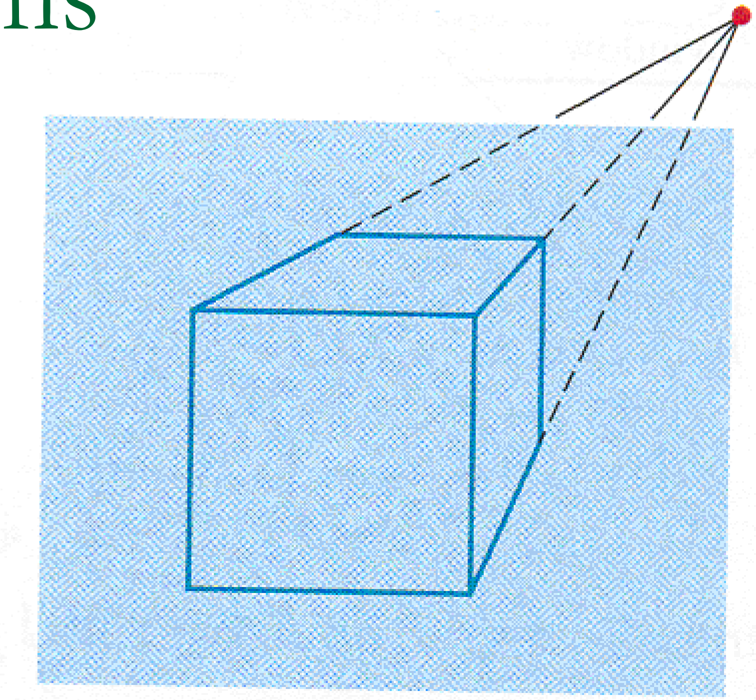
The projector straight-lines converge at the projection center

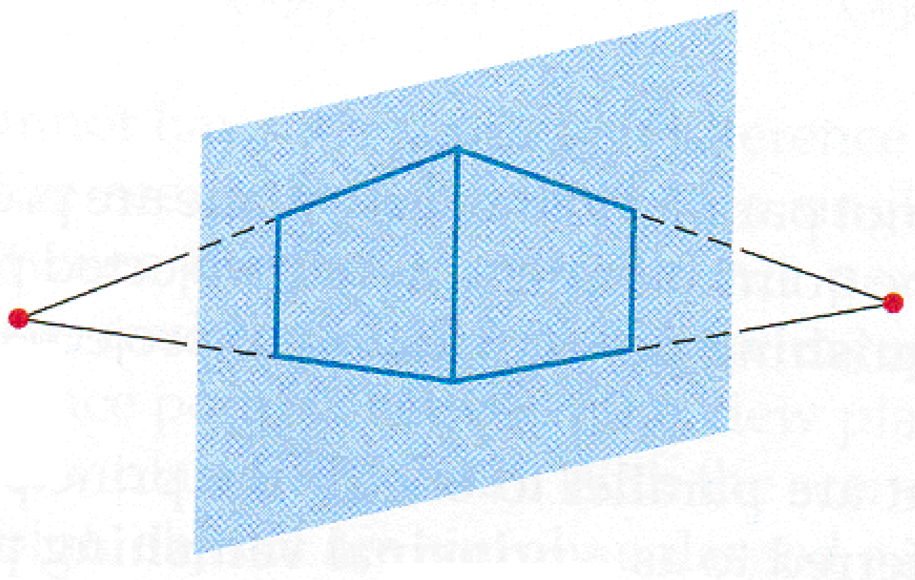
Parallel Projections





Perspective Projections





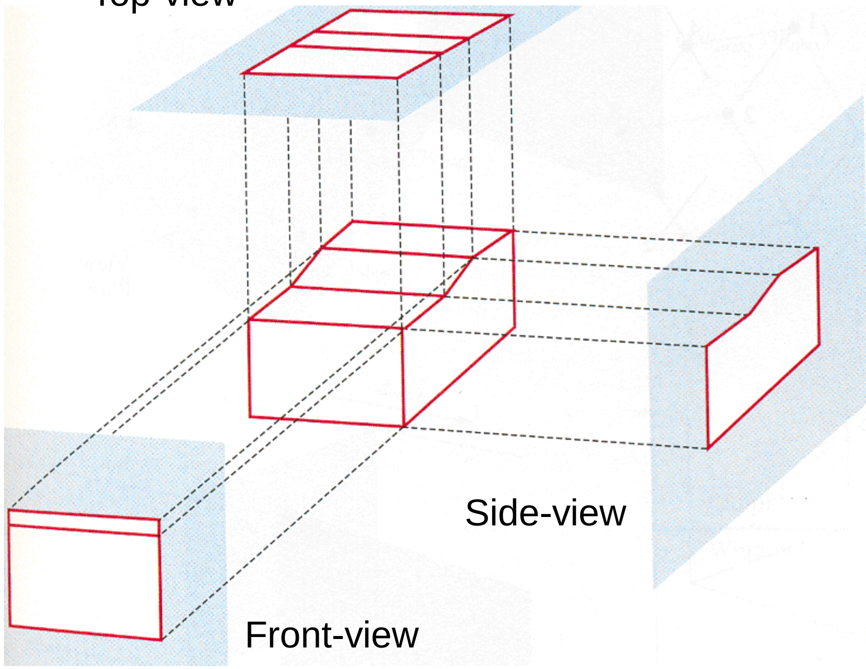
How to represent ?

- Projection matrices
- Homogeneous coordinates
- Concatenation through matrix multiplication
- Don’t worry !
- Graphics APIs implement usual projections !

ORTHOGONAL PARALLEL PROJECTIONS

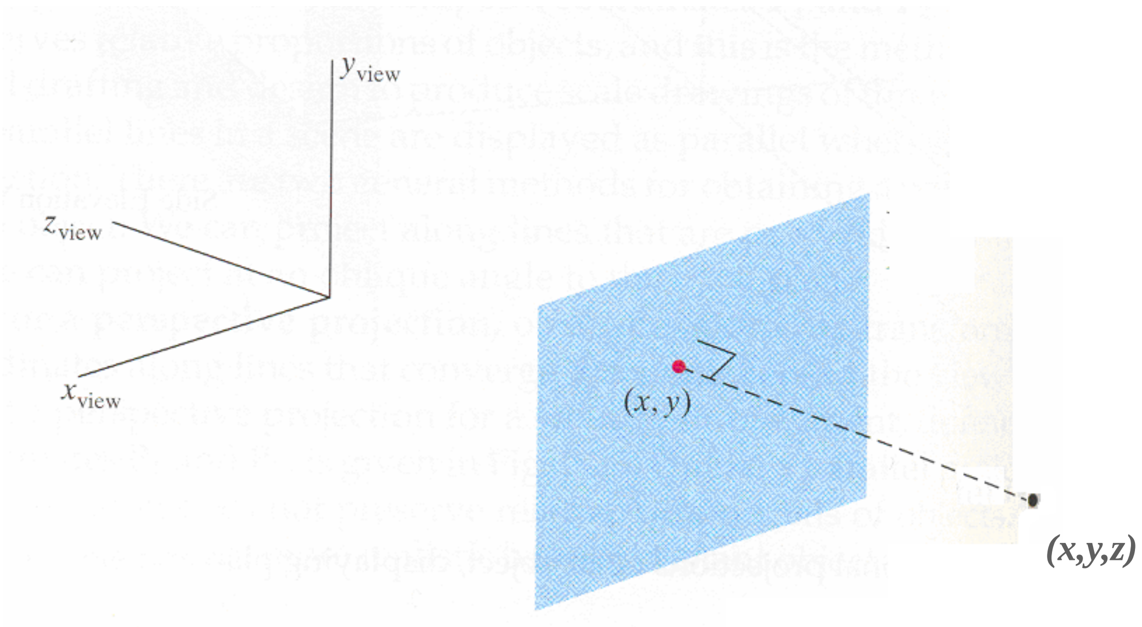
Orthogonal Parallel Projections

- The projectors are perpendicular to the projection plane
- The projection plane is parallel to a set of the object's faces
- Some angles, lengths and areas can be directly measured
- The views might not convey the 3D structure / shape of the objects
- Frequently used in Engineering and Architecture

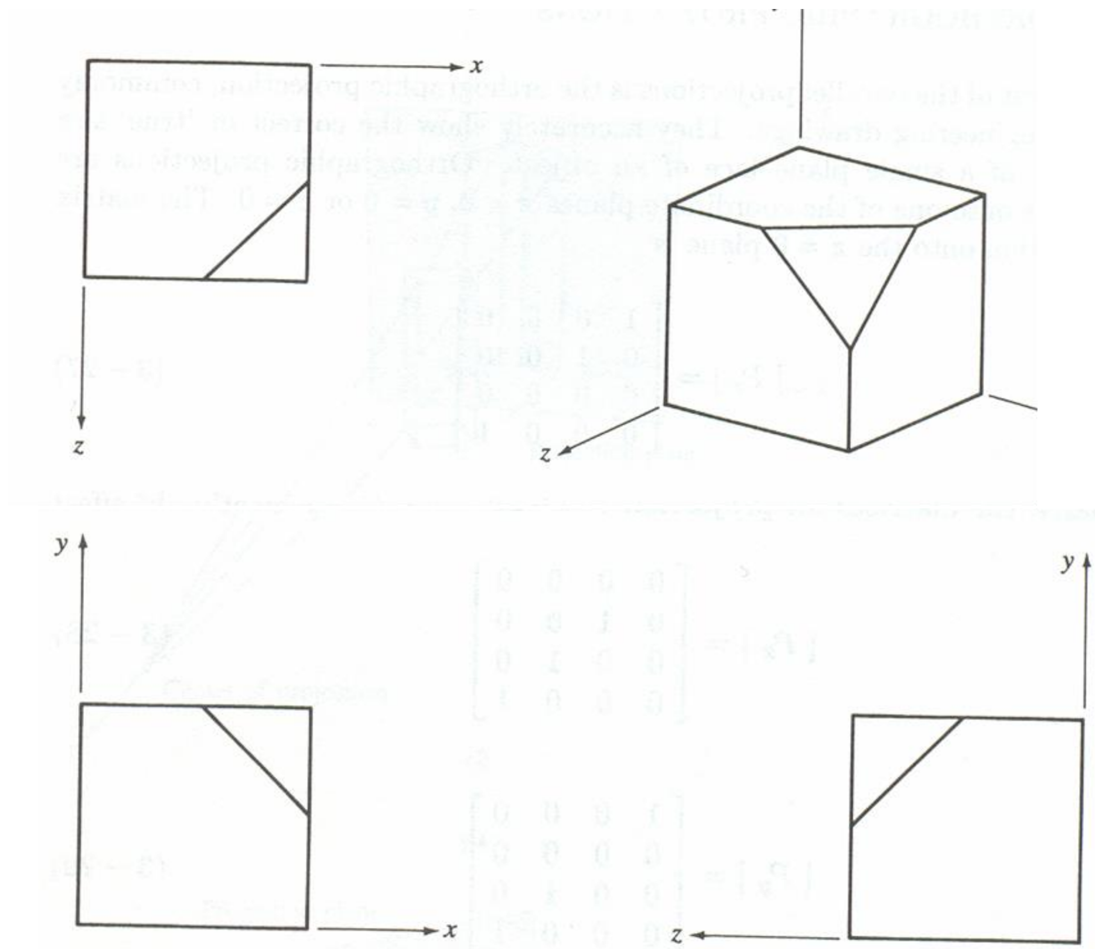


Parallel Projection Coordinates

- If the direction projection is parallel to the ZZ' axis, what are the coordinates of the projected point ?

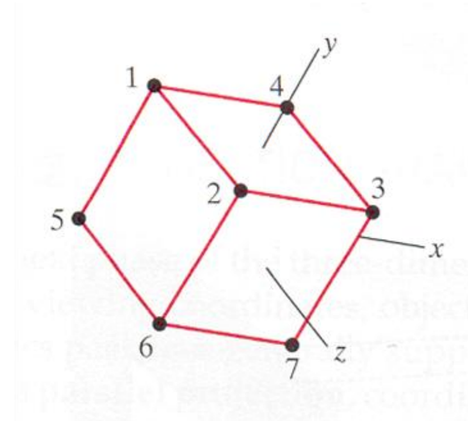


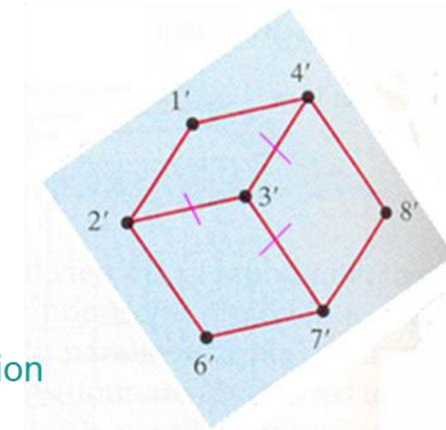
Orthogonal Parallel Projections



Axonometric Projections

- Orthogonal parallel projections, where the projection plane in not parallel to a set of the object's faces
- Give a better idea of the object's 3D structure / shape
- 3 classes
  - Isometric
  - Dimetric
  - Trimetric

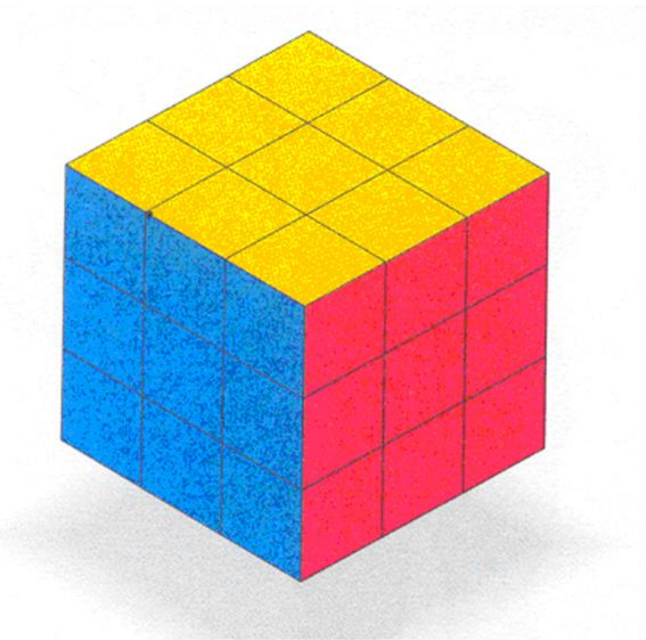


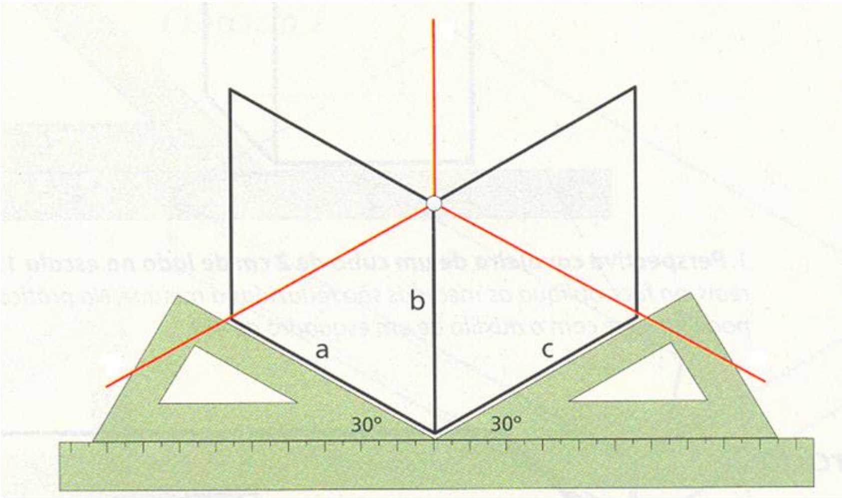


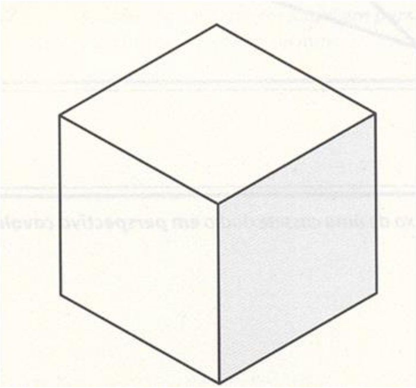
Projection plane

Isometric projection of a cube: 3 faces are shown and all edges have the same length

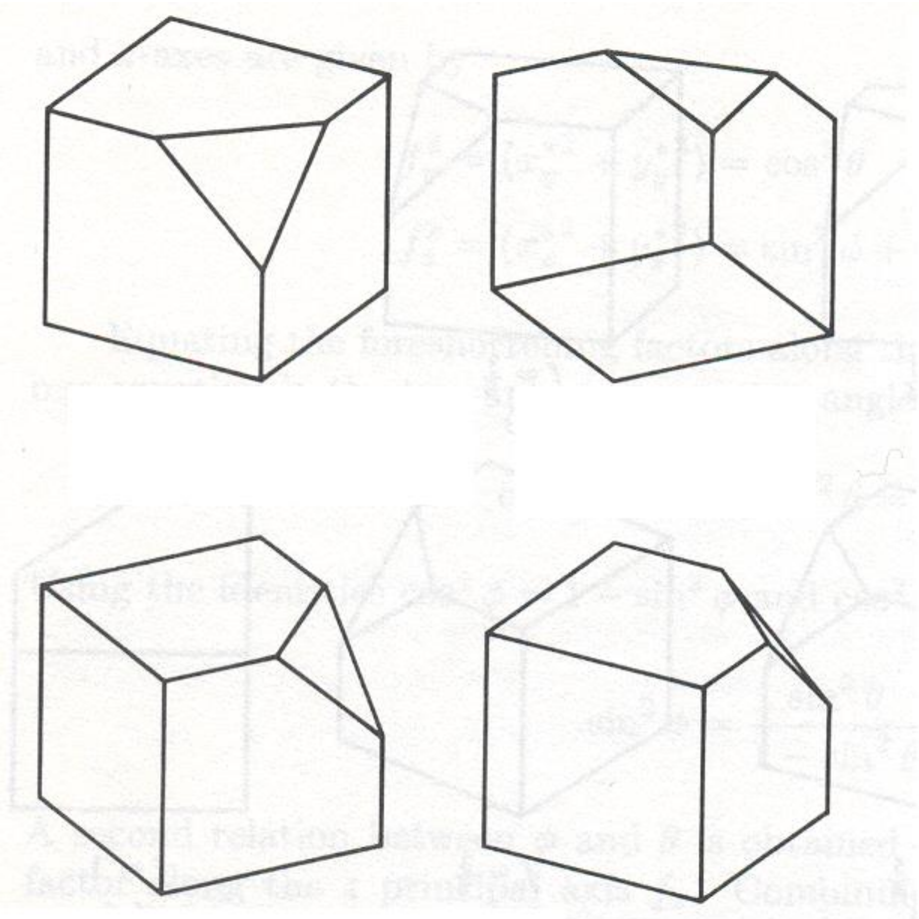
Drawing an isometric projection



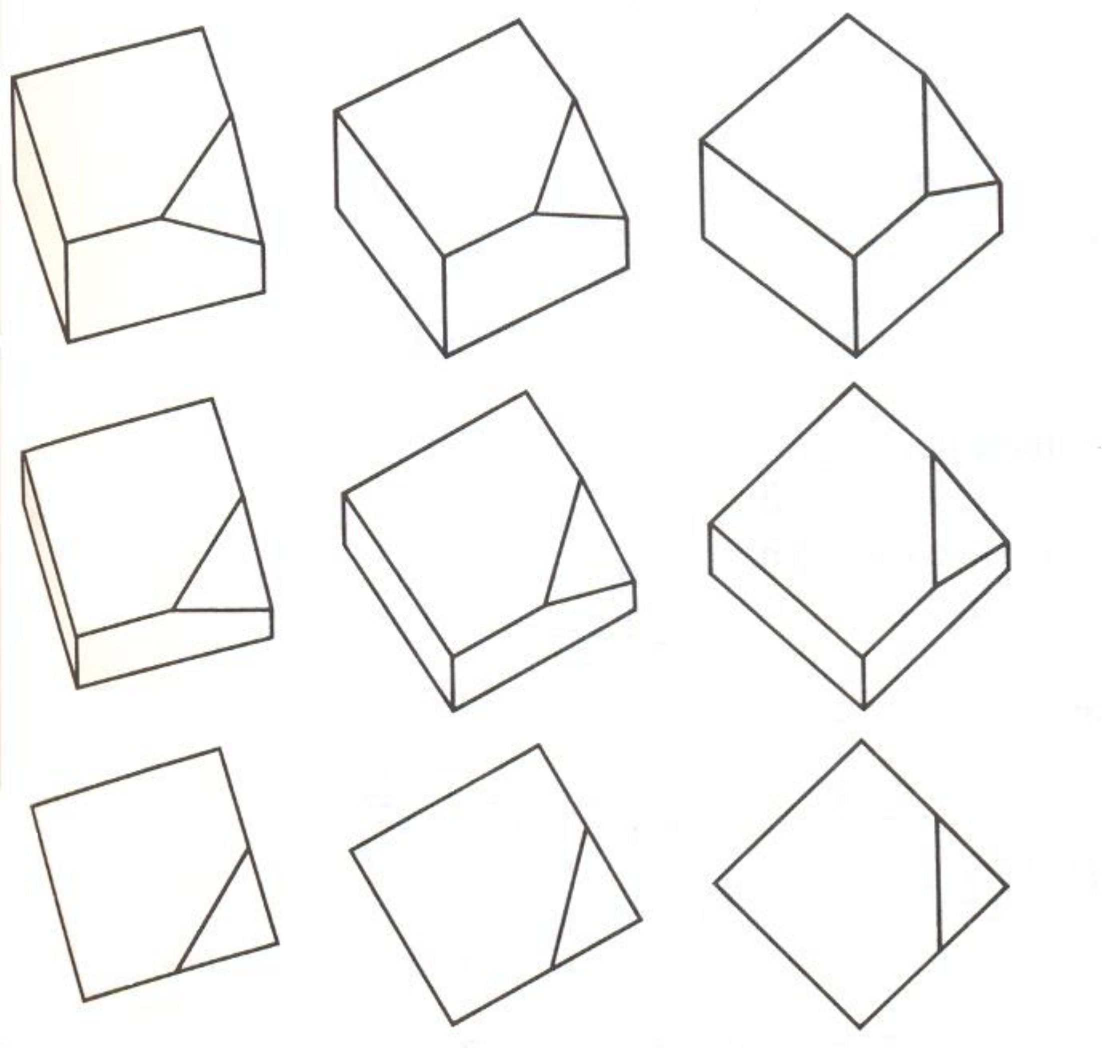




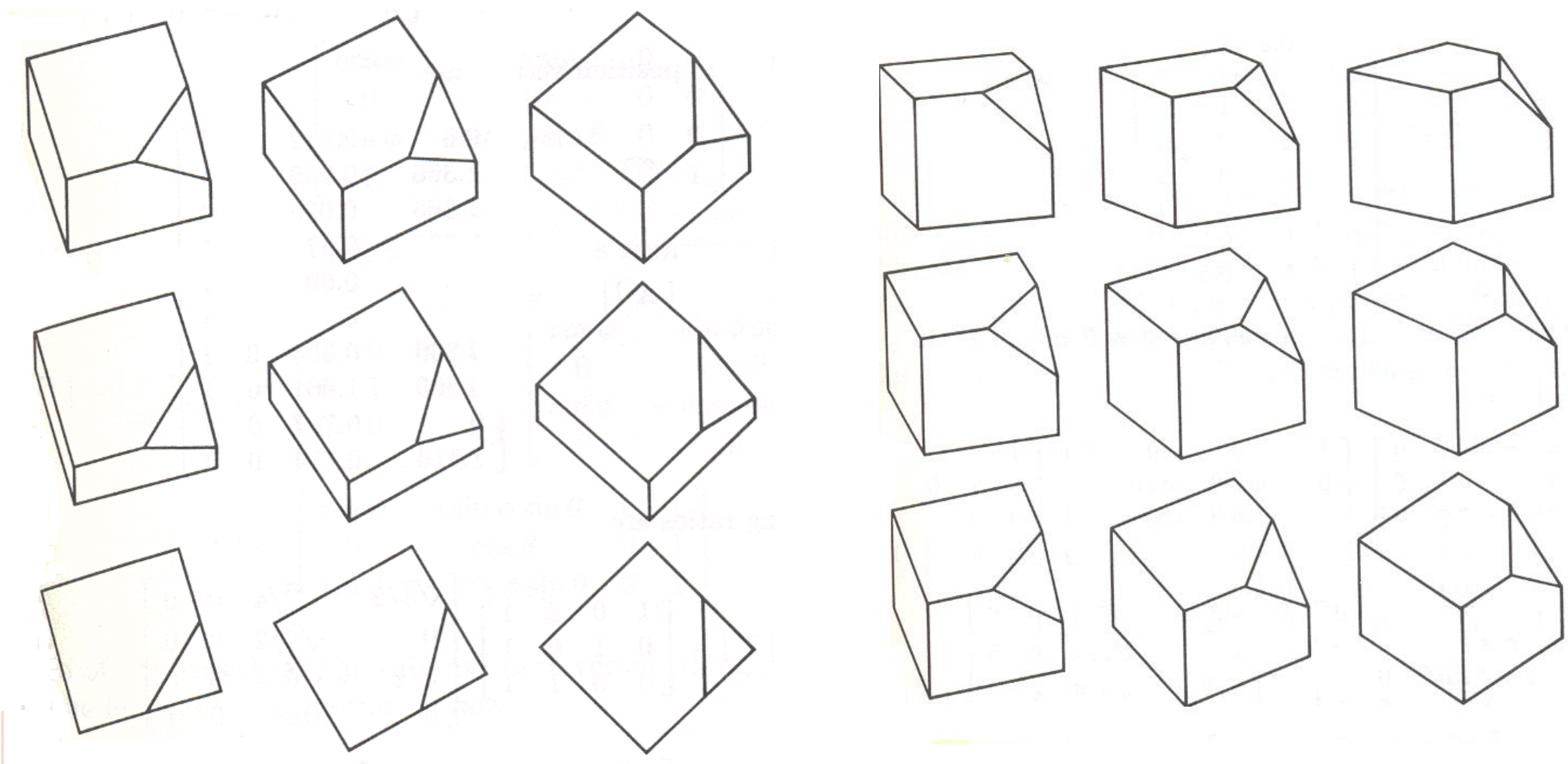
Isometric Projections



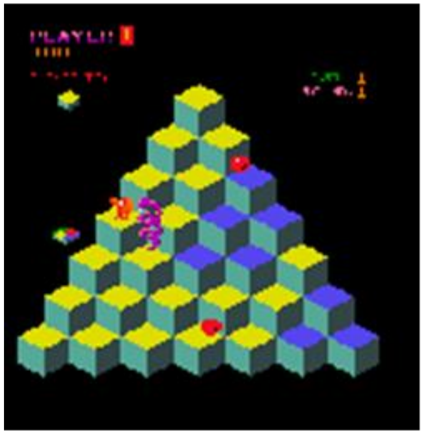
Dimetric Projections

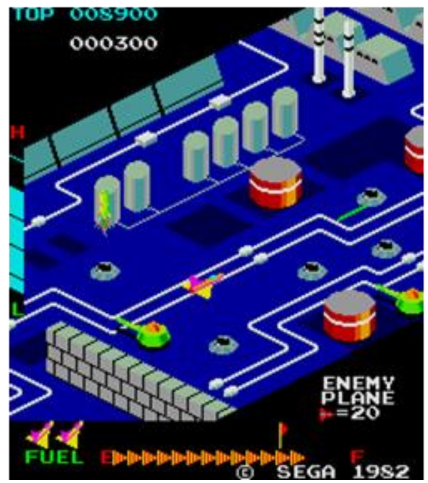


Trimetric Projections



Axonometric Projection in Games

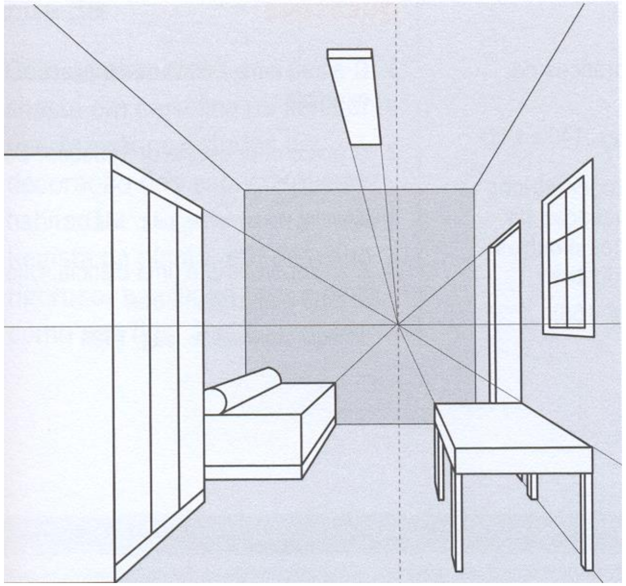


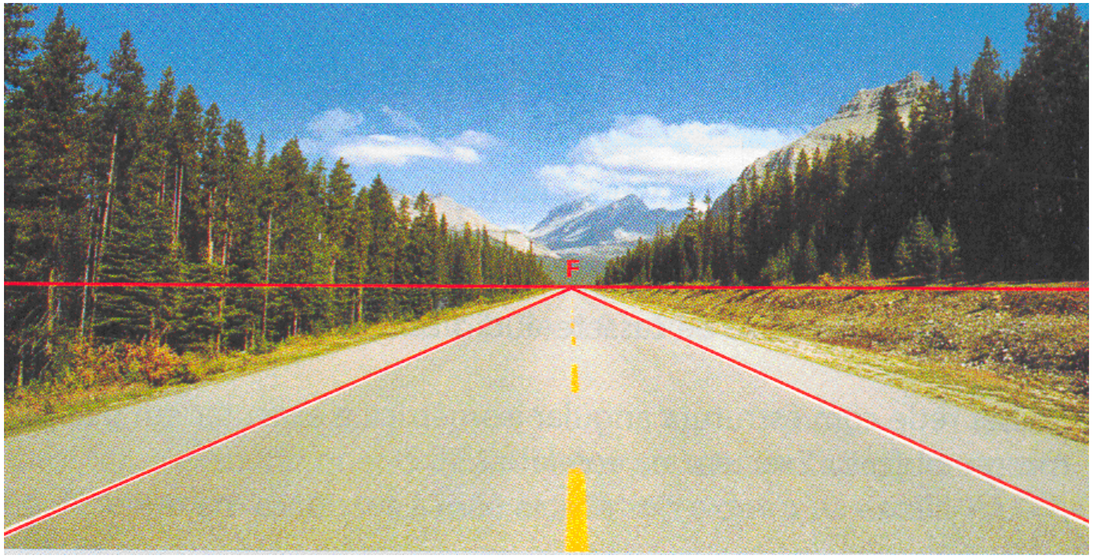


[van Dam]

PERSPECTIVE PROJECTIONS

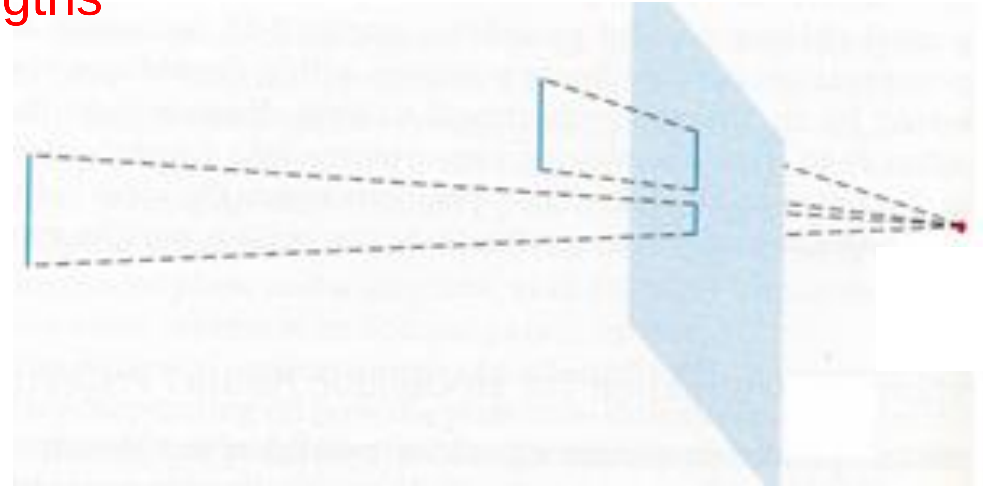
Perspective Projections





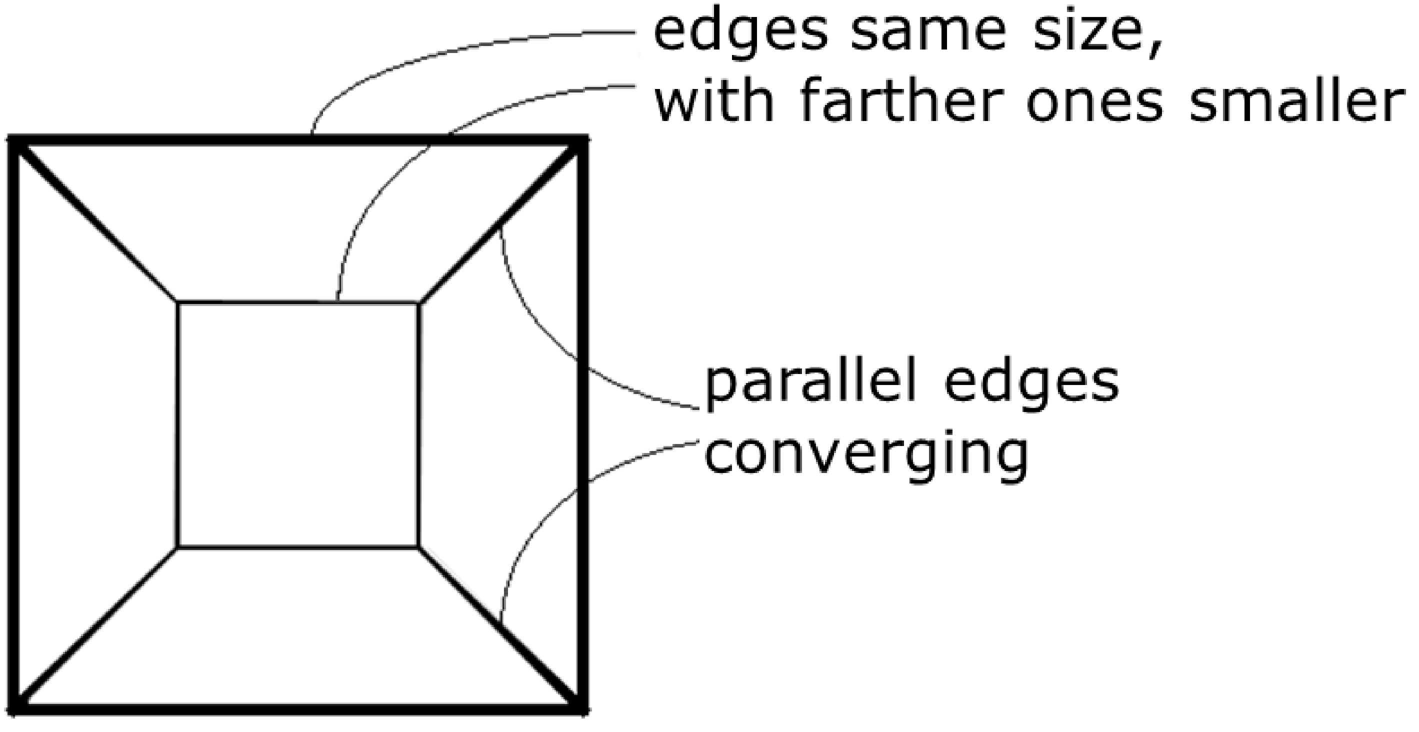
Perspective Projection

- The projections of straight-line segments with the same length, but located at different distances from the projection plane, are projected with different lengths



Regarding the parallel projections:
- It generates more realistic images
- But it does not preserve relative sizes of objects
- It requires more calculations

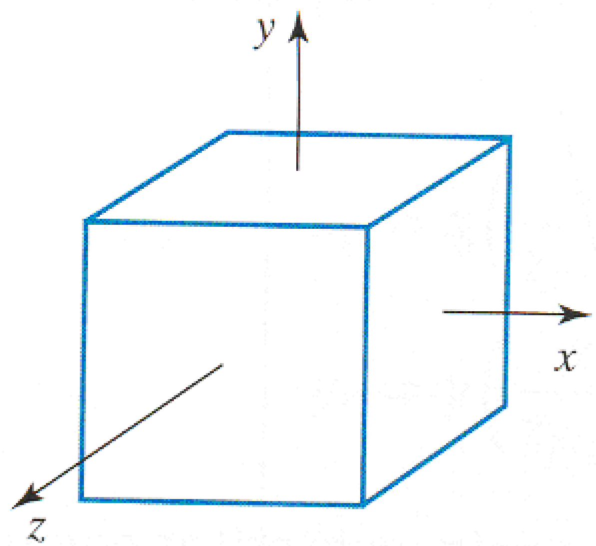
Perspective Projection

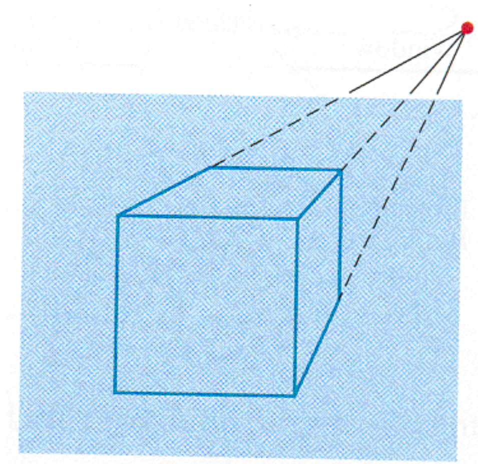


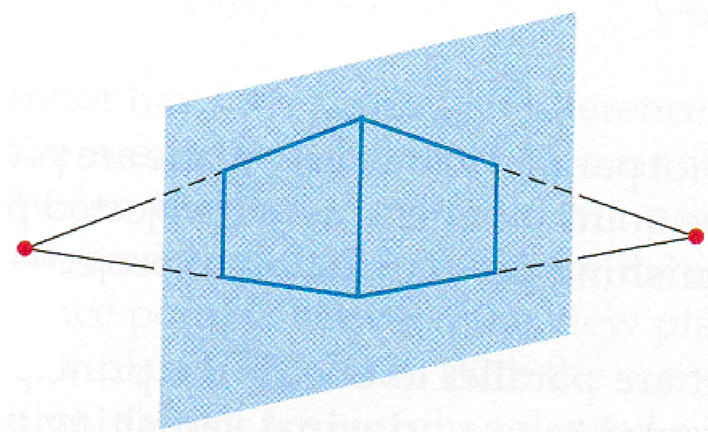
[van Dam]

Vanishing Points

- Straight-lines, parallel to a coordinate axis that intersects the projection plane, converge to that axis' vanishing point



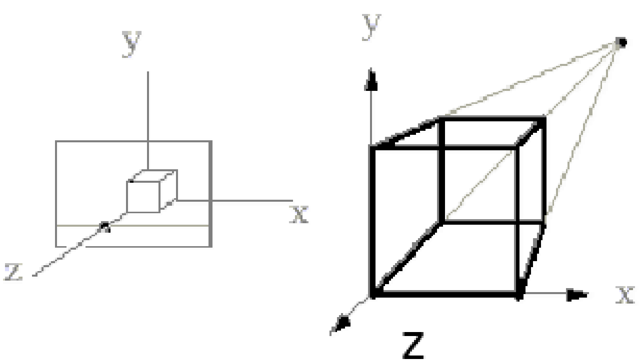




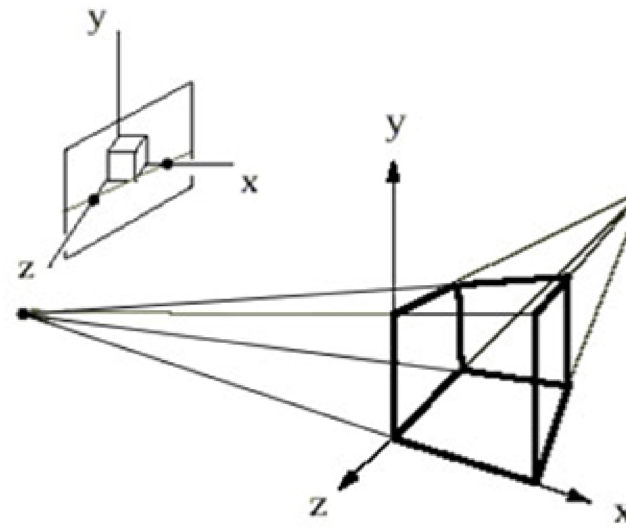
Number of vanishing points:

number of coordinate axes intersecting the projection plane

Vanishing Points



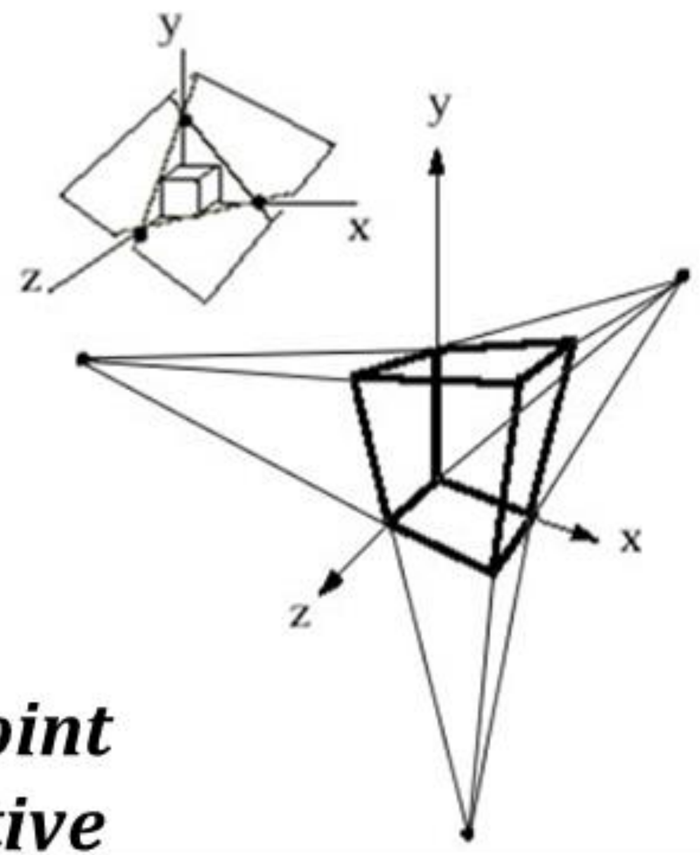
One Point Perspective
(z-axis vanishing point)



Two Point Perspective
(z and x-axis vanishing points)

[van Dam]

Vanishing Points



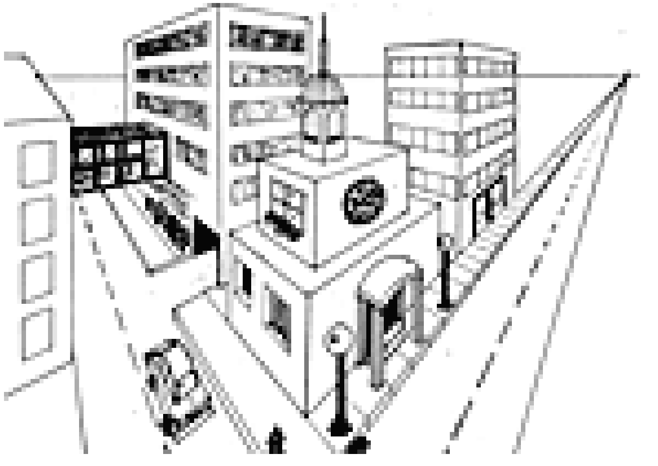
Three Point Perspective

(z, x, and y-axis vanishing points)

[van Dam]

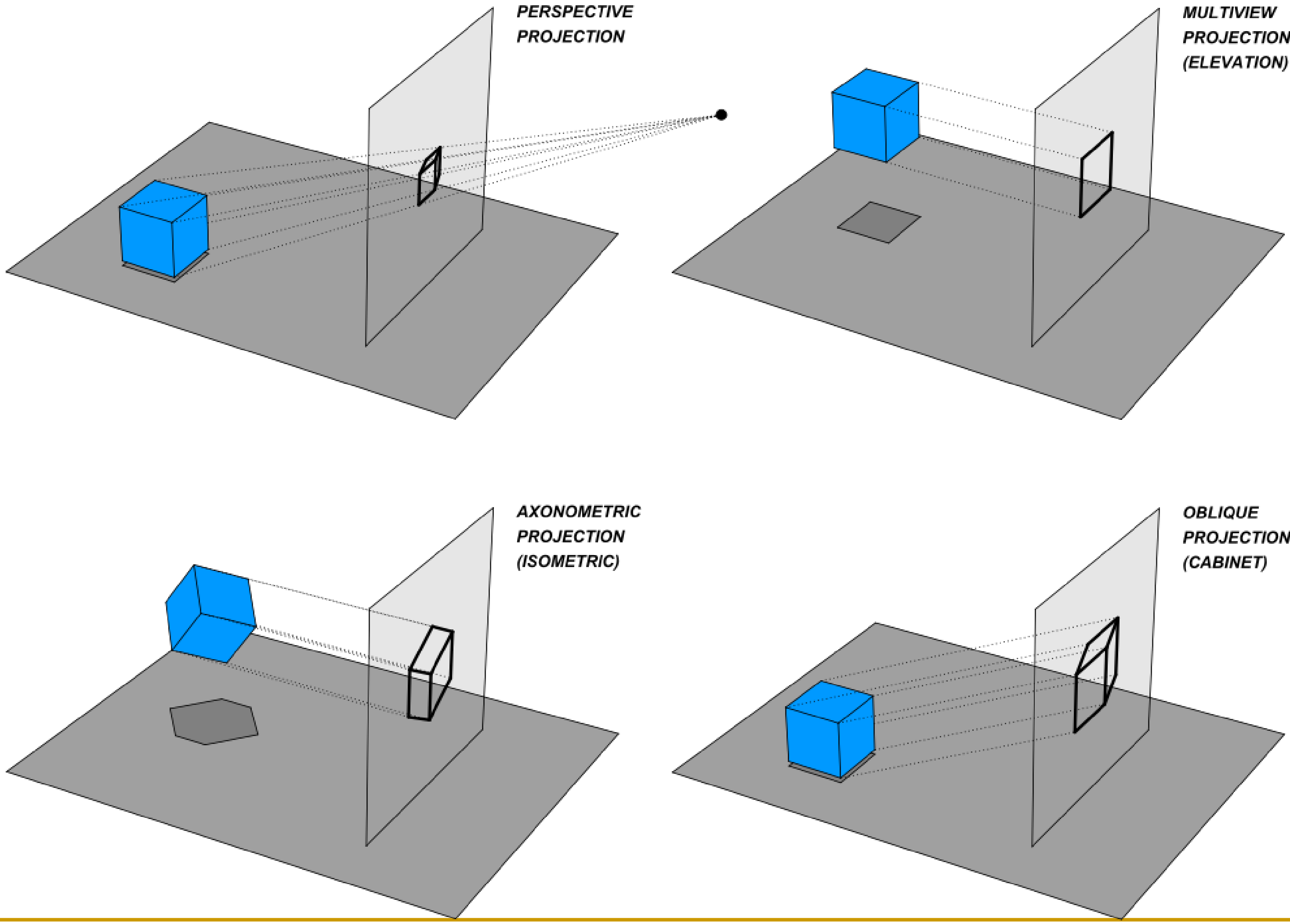
1 and 2 vanishing points

Frontal perspective



Angular perspective

Cube – Various projections



MATRICIAL REPRESENTATION

The Mathematics

- Projection is achieved through matrix multiplication, using a (4 x 4) projection matrix in homogeneous coordinates

- The projection matrix can be concatenated with the model-view matrix to carry out any modeling transformations before the actual projection
  - Animations
  - More complex projections are decomposed into a sequence of simpler transformations

- Let's consider the simplest cases, when the projection plane is XOY or a plane parallel to XOY

Projection plane at z = d

$P\ (x, y, z)$ – original point
$P_p\ (x_p, y_p, z_p)$ – projected point

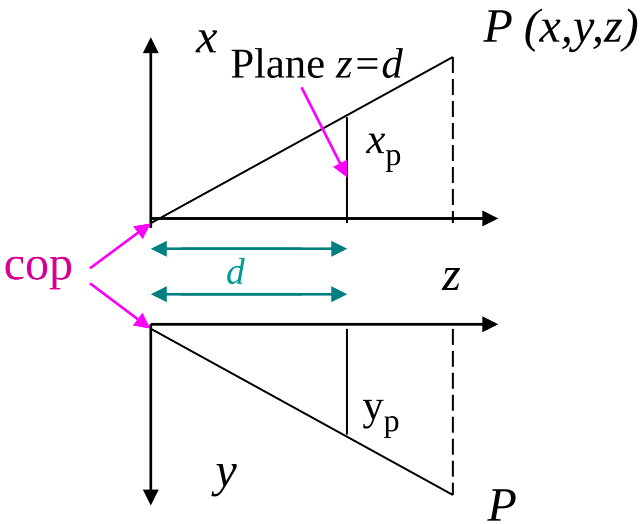
Distance ratios:

$x_p / d = x / z \qquad\qquad y_p / d = y / z$

Multiplying by d:

$$x_p = \frac{d \cdot x}{z} = \frac{x}{z / d}$$

$$y_p = \frac{d \cdot y}{z} = \frac{y}{z / d}$$



Dividing by z implies that objects further away appear smaller

Projection plane at z = d

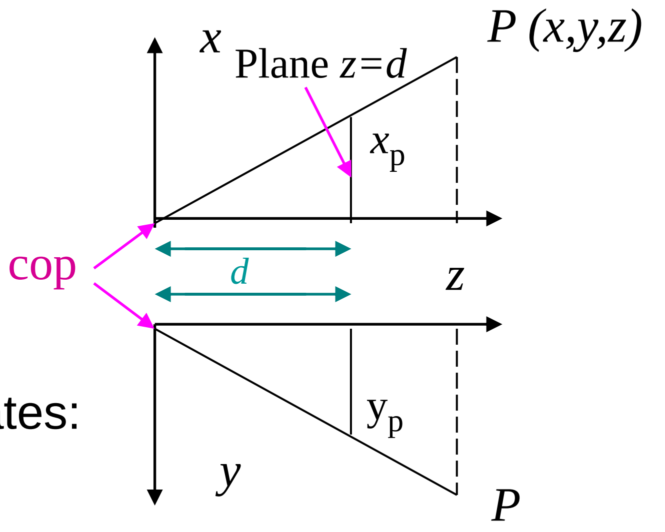
$P\ (x, y, z)$ – original point

$P_p\ (x_p, y_p, z_p)$ – projected point

All z values are possible except $z=0$

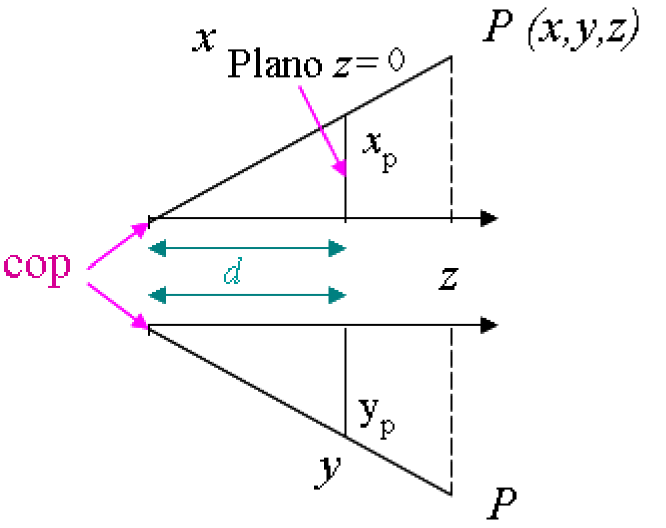
The projection matrix in homogeneous coordinates:

$$M_{pers} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \longrightarrow P_p = M_{pers} \cdot P$$



Center of projection at z = -d

$P\ (x, y, z)$ – original point
$P_p\ (x_p, y_p, z_p)$ – projected point



Distance rations:

$x_p / d = x / (z + d) \qquad y_p / d = y / (z + d)$

Multiplying by d:

$$x_p = \frac{d \cdot x}{z + d} = \frac{x}{z / d + 1}$$

$$y_p = \frac{d \cdot y}{z + d} = \frac{y}{z / d + 1}$$

$$M'_{pers} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix}$$

Center of projection at $+\infty$

This matricial representation allows to replace d with ∞, and we obtain the matrix for the orthogonal, parallel projection on the projection plane $z = 0$:

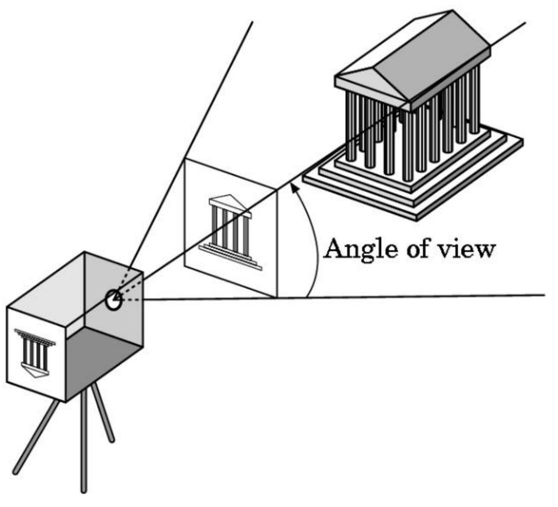
$$M_{orto} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

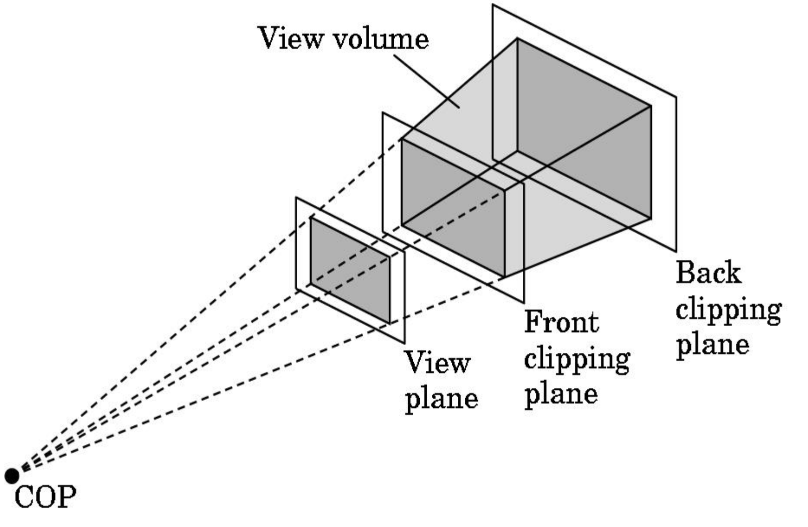
What are the coordinates of a projected point ?
Is that the expected result ?

VIEW VOLUME & CLIPPING

Clipping

- The virtual camera only “sees” part of the world or object space
  - View volume

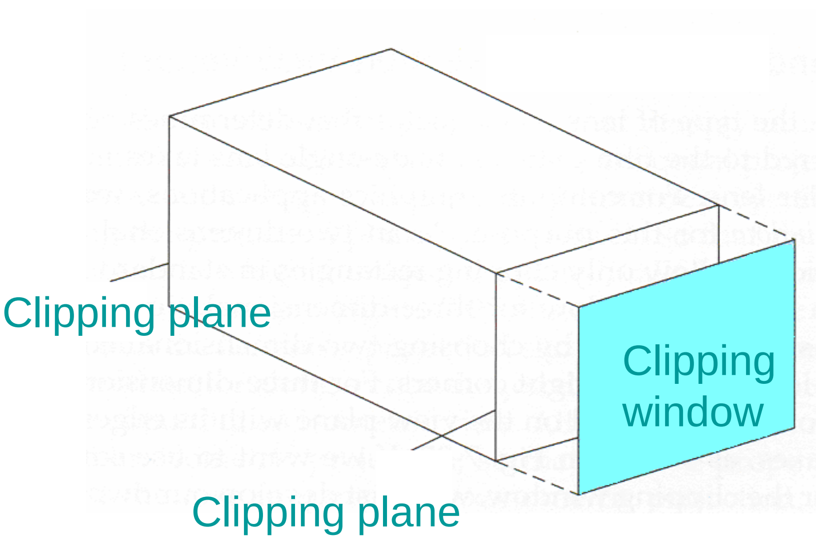


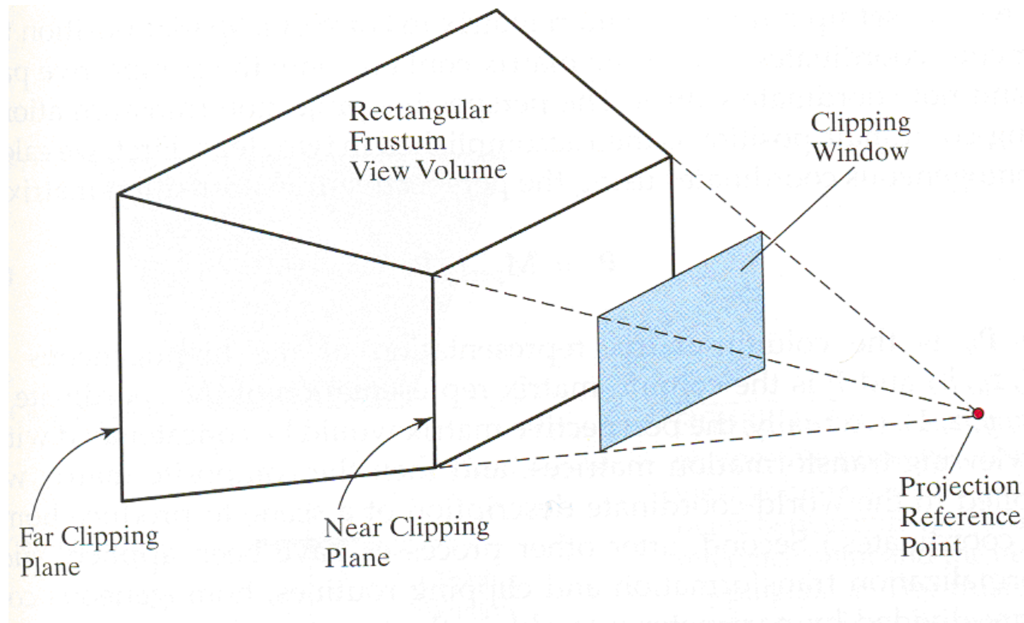


[Angel]

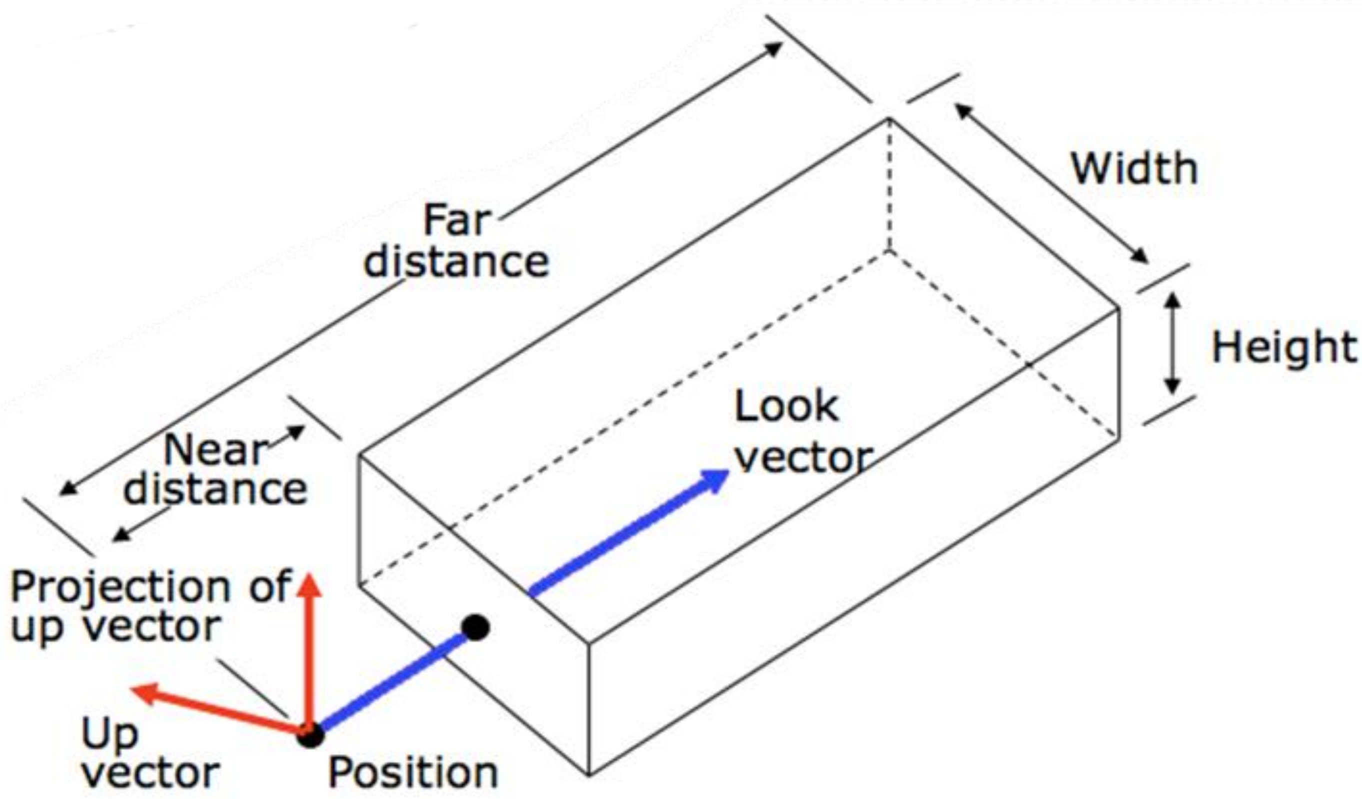
How to limit what is observed ?

- Clipping window on the projection plane
- View volume in 3D



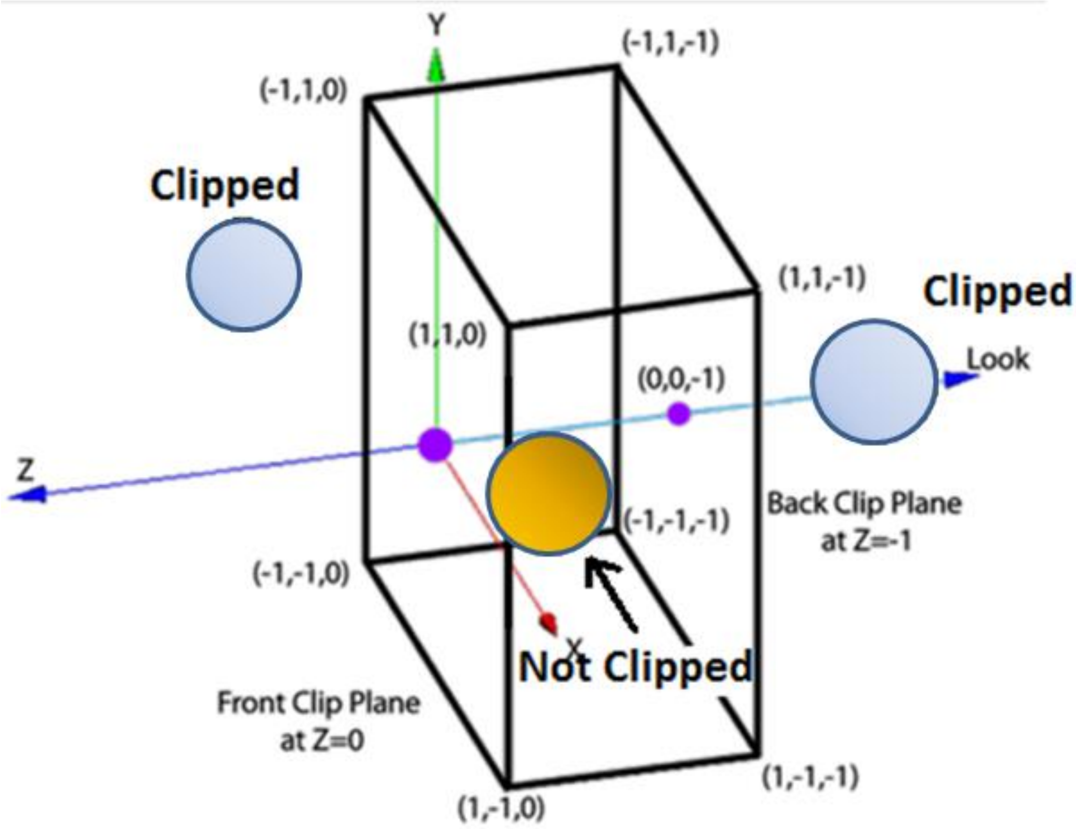


Parallel Projection – View Volume



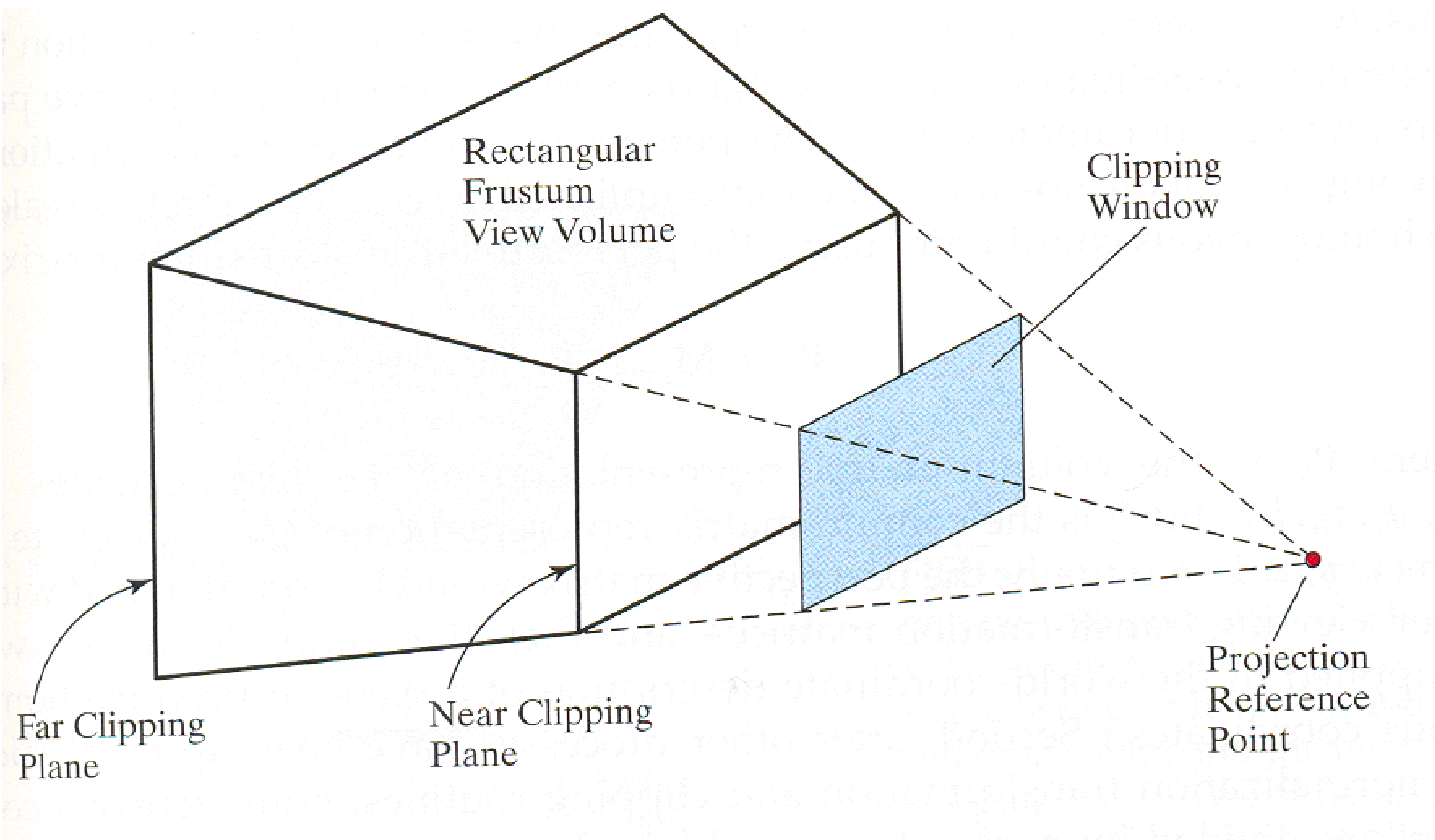
[van Dam]

Clipping against the View Volume

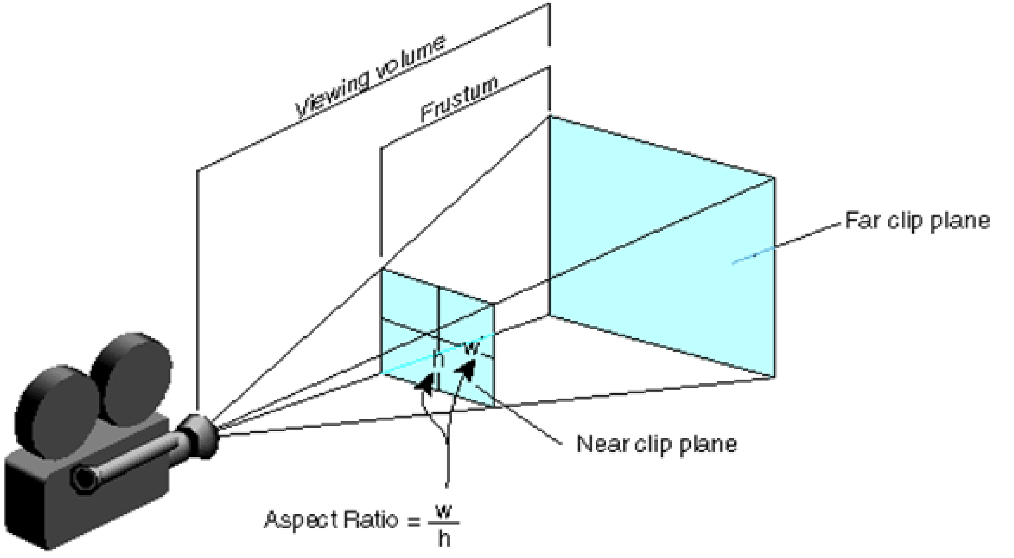


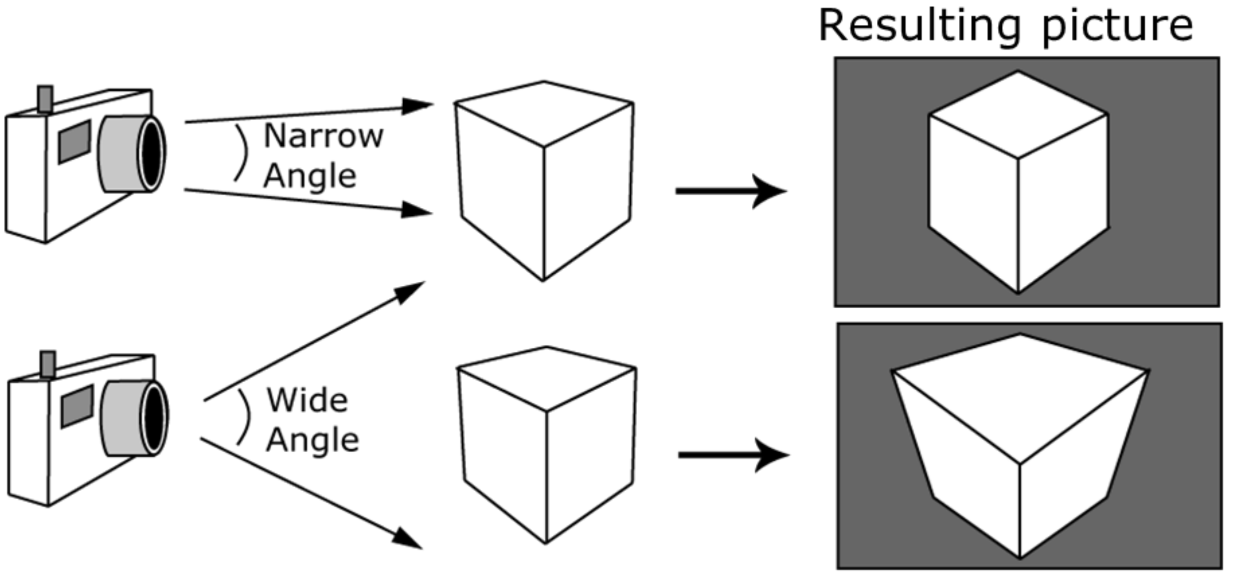
[van Dam]

Perspective Projection – View Volume



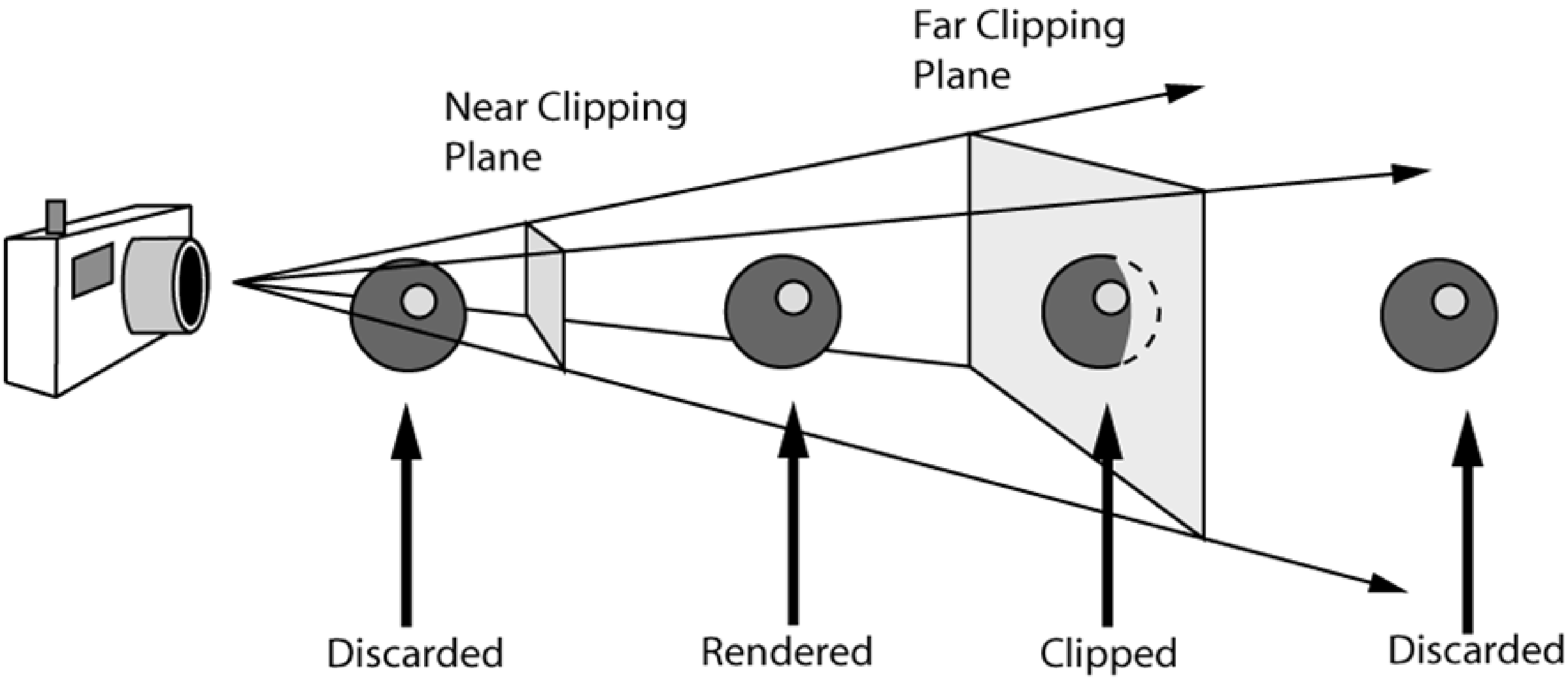
View Angle





[van Dam]

Clipping Planes

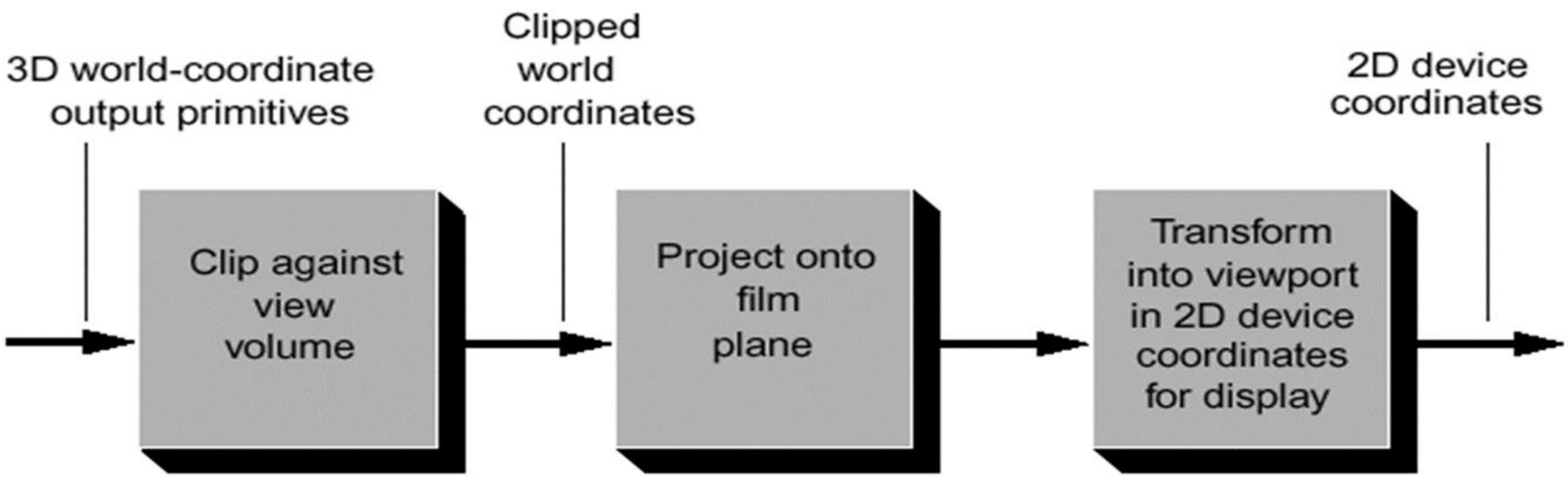


[van Dam]

3D Viewing

- How to view primitives that are outside the view volume ?
  - Translate !
- How to view a side face of a model ?
  - Rotate !
- …

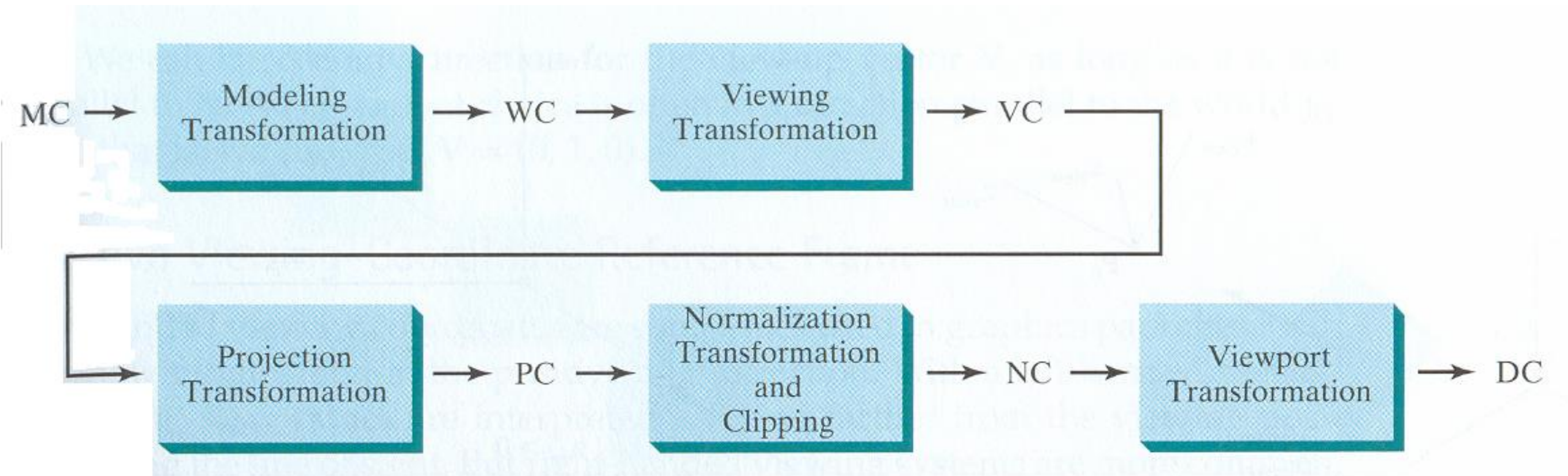
Pipeline



[van Dam]

THE 3D VIEWING PIPELINE

3D visualization pipeline



[Hearn & Baker]

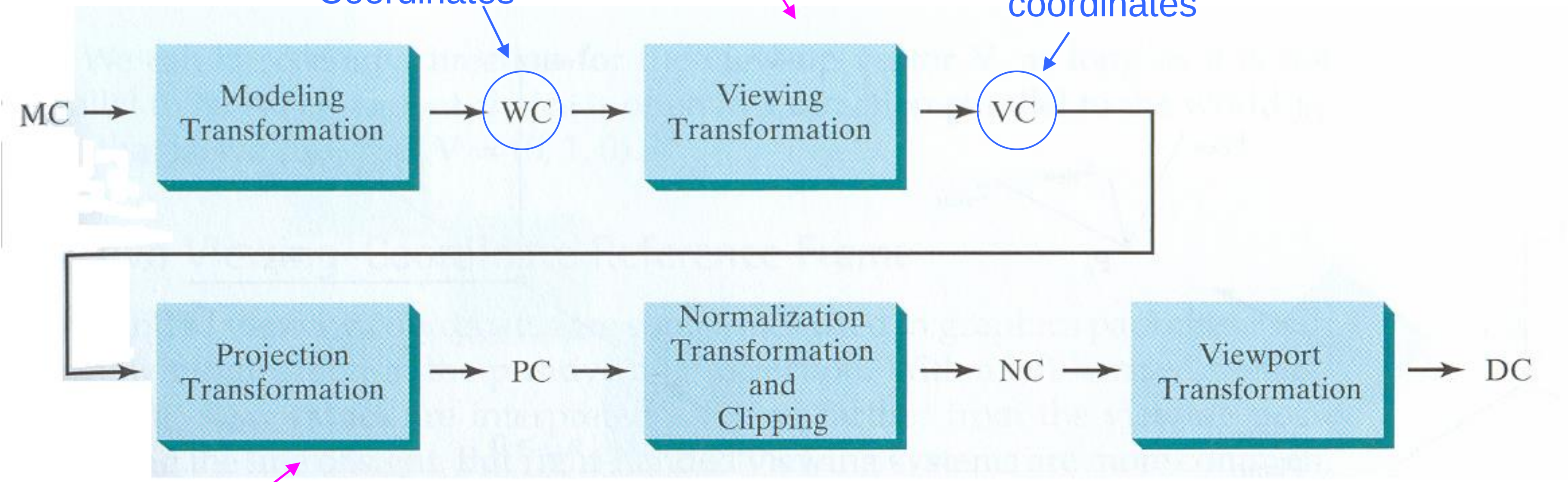
3D visualization pipeline

From scene coordinates to device coordinates:

1st – define the viewing transformation

world Coordinates

viewing coordinates



2nd – define the projection transformation

Need also to identify the visible surfaces and carry out the *surface rendering*

3D visualization pipeline

- Instantiate models
  - Position, orientation, size
- Establish viewing parameters
  - Camera position and orientation
- Compute illumination and shade polygons
- Perform clipping
- Project into 2D
- Rasterize

3D visualization pipeline

- Main operations represented as point transformations
  - Homogeneous coordinates
  - Transformation matrices
  - Projection matrix
  - Matrix multiplication

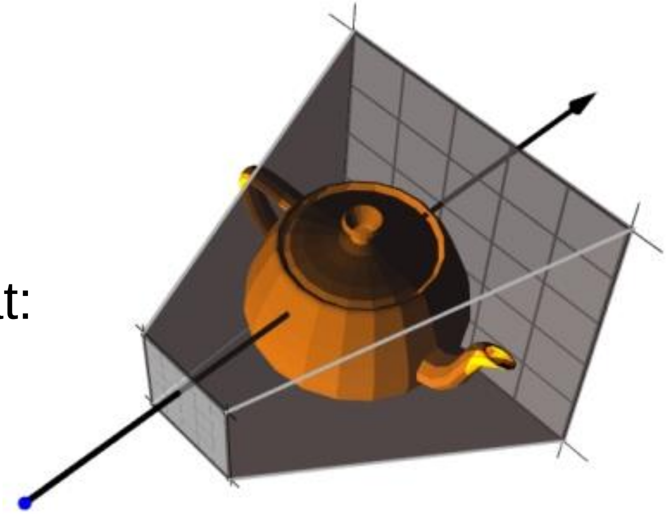
3D visualization pipeline

- Each object is processed separately
  - 3D triangles
- Object / triangle inside the view volume ?
  - No : go to next object / triangle
- Rasterization
  - Compute the location on the screen of each triangle
  - Compute the color of each pixel

VISIBLE SURFACE DETERMINATION

Visible Surface Determination

- The visibility problem
  - Which primitives – after modeling transformations, projection and lighting calculations – contribute for each image pixel



- In general, we solve the dual problem !!

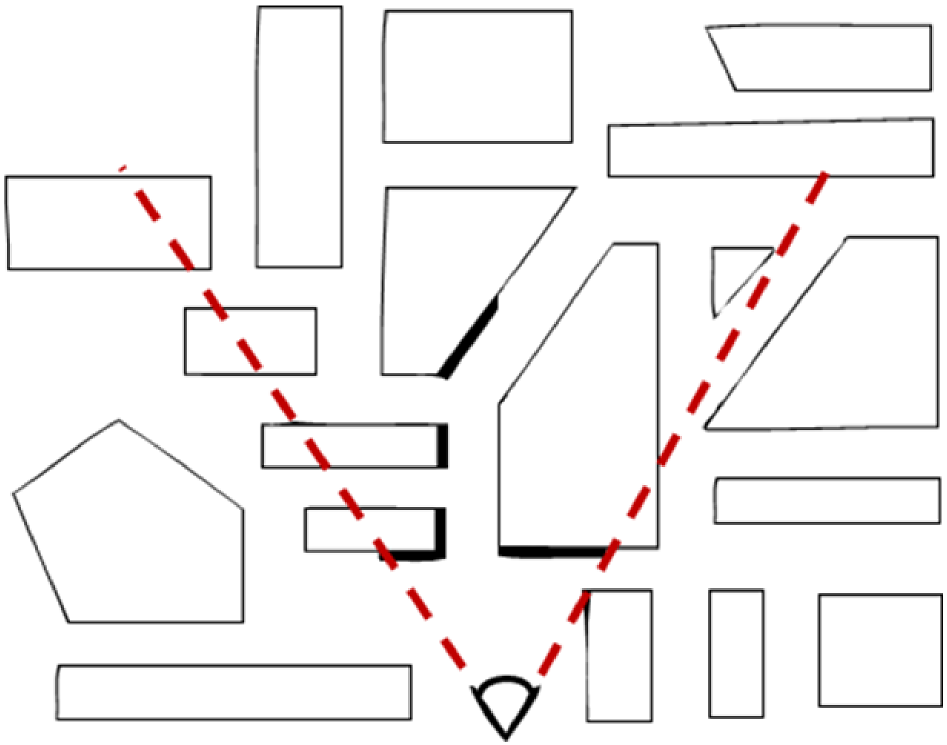
- Which are the hidden surfaces / faces that:

  - are outside of the view volume ?

  - are back-faces in a closed and convex polyhedron ?

  - are hidden by other faces closer to the viewpoint / camera ?

Visible Surface Determination



For each object compute:

- The visible edges and surfaces

Why might objects be hidden ?

- Clipping ?
- Occlusion ?

To render or not to render, that is the question…

[Andy Van Dam]

Clipping vs Occlusion

- Clipping against the view volume
  - It is done at object-level !
- Occlusion / Hidden-Surface Removal
  - It is done at scene-level !
  - Compare depth of object / edges / pixels against other objects / edges / pixels

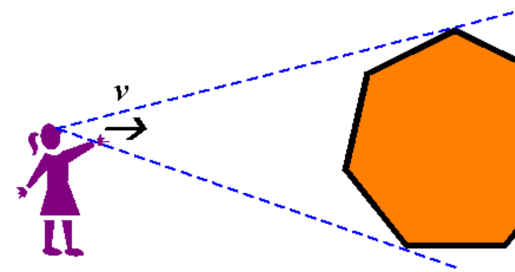
Possible approaches

- Object-precision algorithms
  - Analyze / compare objects or parts of objects to determine which surfaces / faces / edges are fully or partially visible
  - Back-Face Culling
- Image-precision algorithms
  - Determine visibility for every pixel in the viewing plane
  - Work in 3D to get / compare depth values (i.e., z values)
  - Z-buffer

BACK-FACE CULLING

Back-Face Culling

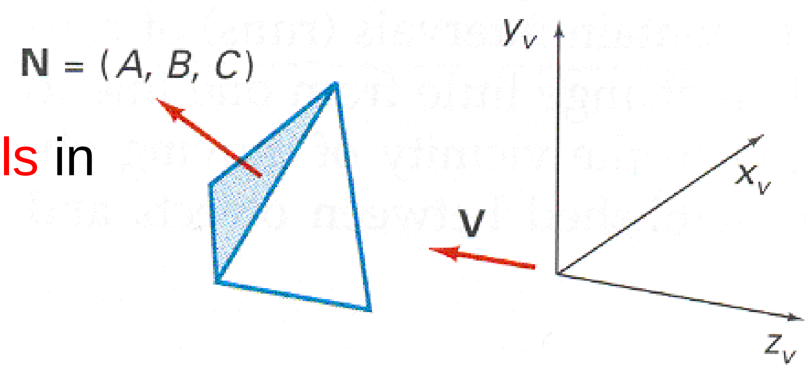
- Sufficient for a single convex polyhedron which is not sectioned by clipping



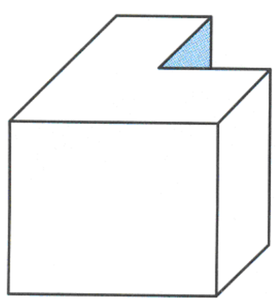
- Not sufficient for

  - concave polyhedra

  - when there are two or more models in front of each other



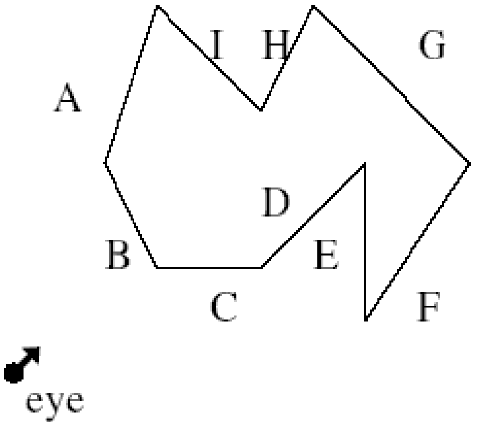
When looking at the negative ZZ' semi-axis, a polygon is a back-face if $C \leq 0$

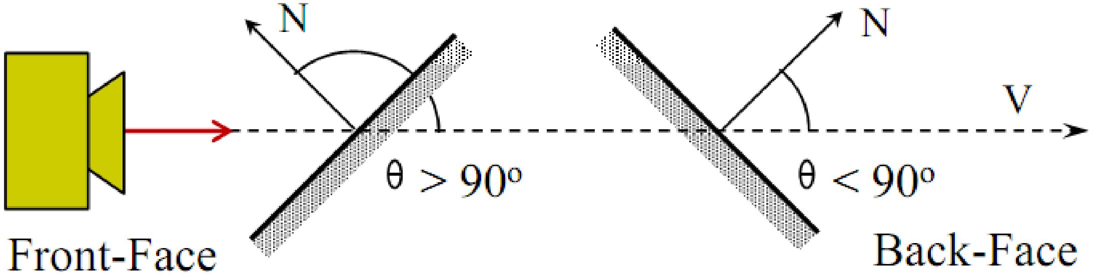


Concave model with a partially visible face

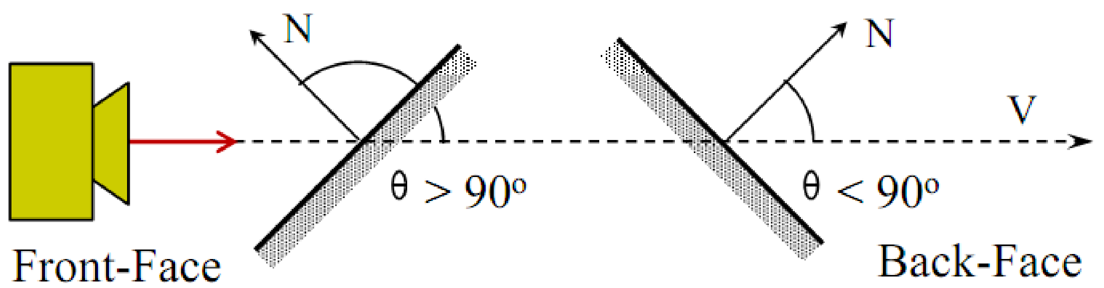
Back-Face Culling

- What to do in a general case ?
- For each face, compute the angle between
  - The normal vector to the face
  - The viewing direction, defined by the the viewpoint





Back-Face Culling

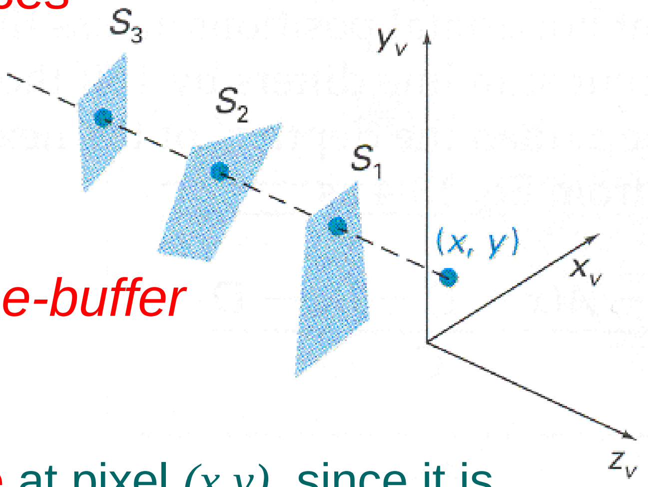


- Efficiency: just compute the scalar product !!
  - Reject a face if N · V > 0
  - What happens if N · V = 0 ?
- Simplification when V = (0, 0, -1) !!
- On average, approx. half of the faces are removed!

Z-BUFFER

Depth-Buffer (z-buffer)

- Works in *image-space*
- Compares the depth of each surface relative to each pixel in the viewplane
- Fast and easy to implement for planar surfaces
- Can be adapted for curved surfaces
- Needs a *depth-buffer* in addition to the *frame-buffer*



S1 is visible at pixel (x,y), since it is closer to the viewplane

Z-Buffer Algorithm

- Draw every polygon that can't be rejected trivially
  - I.e., it is totally outside the view volume
- If a piece (one or more pixels) of a polygon that is closer to the front is found
- Paint over whatever was behind it
- Use plane equation for polygon, z = f(x, y)
- Note: use positive z here [0, 1]

Z-Buffer Algorithm

```
void zBuffer() {
   int x, y;
   for (y = 0; y < YMAX; y++)
   for (x = 0; x < XMAX; x++) {
       WritePixel (x, y, BACKGROUND_VALUE);
       WriteZ (x, y, 1);
   }
   for each polygon {
     for each pixel in polygon's projection {
        //plane equation
        double pz = Z-value at pixel (x, y);
        if (pz <= ReadZ (x, y)) {
          // New point is closer to front of view
          WritePixel (x, y, color at pixel (x, y))
          WriteZ (x, y, pz);
        }
     }
   }
}
```

[Andy Van Dam]

Example

Initial *z-buffer* values

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Depth-values for polygon

5	5	5	5	5	5	5
5	5	5	5	5	5	
5	5	5	5	5		
5	5	5	5			
5	5	5				
5	5					
5						

Overwrite *z-buffer*

5	5	5	5	5	5	5	0
5	5	5	5	5	5	0	0
5	5	5	5	5	0	0	0
5	5	5	5	0	0	0	0
5	5	5	0	0	0	0	0
5	5	0	0	0	0	0	0
5	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Another polygon

3					
4	3				
5	4	3			
6	5	4	3		
7	6	5	4	3	
8	7	6	5	4	3

Final *z-buffer*

5	5	5	5	5	5	5	0
5	5	5	5	5	5	0	0
5	5	5	5	5	0	0	0
5	5	5	5	0	0	0	0
6	5	5	3	0	0	0	0
7	6	5	4	3	0	0	0
8	7	6	5	4	3	0	0
0	0	0	0	0	0	0	0

Note:
0 – background depth
Max – viewplane

Another example

255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255

+

64	64	64	64	64	64	64
64	64	64	64	64	64	
64	64	64	64	64		
64	64	64	64			
64	64	64				
64	64					
64						

=

64	64	64	64	64	64	64	255
64	64	64	64	64	64	255	255
64	64	64	64	64	255	255	255
64	64	64	64	255	255	255	255
64	64	64	255	255	255	255	255
64	64	255	255	255	255	255	255
64	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255

64	64	64	64	64	64	64	255
64	64	64	64	64	64	255	255
64	64	64	64	64	255	255	255
64	64	64	64	255	255	255	255
64	64	64	255	255	255	255	255
64	64	255	255	255	255	255	255
64	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255

+

127				
127	127			
127	127	127		
127	127	127	127	
127	127	127	127	127

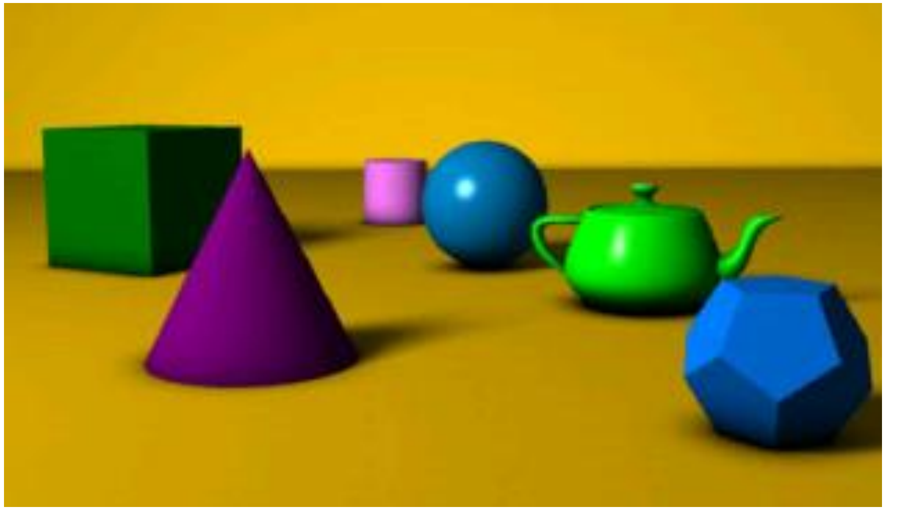
=

64	64	64	64	64	64	64	255
64	64	64	64	64	64	255	255
64	64	64	64	64	255	255	255
64	64	64	64	255	255	255	255
64	64	64	255	255	255	255	255
64	64	127	255	255	255	255	255
64	127	127	127	255	255	255	255
127	127	127	127	127	255	255	255

integer Z-buffer with
near = 0, far = 255

[Andy Van Dam]

3D scene



Z-buffer

Z-Buffer Algorithm

- Advantages:

  - Easy, no need to previously sort the various surfaces

  - Fast

- Disadvantages:

  - Need for additional memory

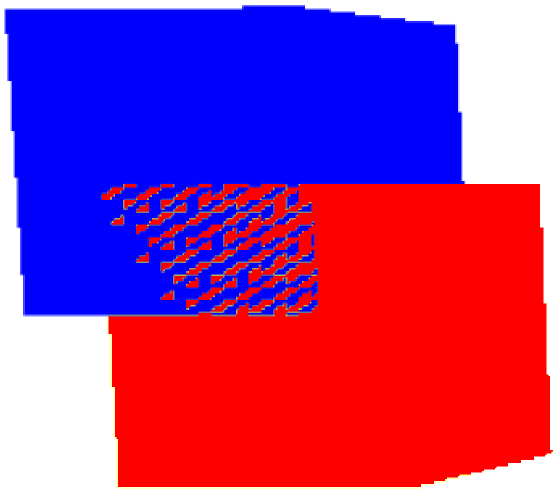
  - Depth precision problems / limitations

  - It finds one visible surface for each pixel

  - I.e., it can only handle opaque surfaces

Z-Fighting

- Z-fighting occurs when two primitives have similar values in the z-buffer
  - Coplanar polygons (two polygons that occupy the same space)
  - One is arbitrarily chosen over the other, but z varies across the polygons and binning will cause artifacts
  - Behavior is deterministic: the same camera position gives the same z-fighting pattern



Two intersecting cubes

[Andy Van Dam]

Possible References

- Projections and the Viewing Pipeline are presented in any Computer Graphics book

- E. Angel and D. Shreiner. Interactive Computer Graphics, 7th Ed., Addison-Wesley, 2015

- J M Pereira, et al. Introdução à Computação Gráfica. FCA, 2018

THREE.JS – THE CAMERA

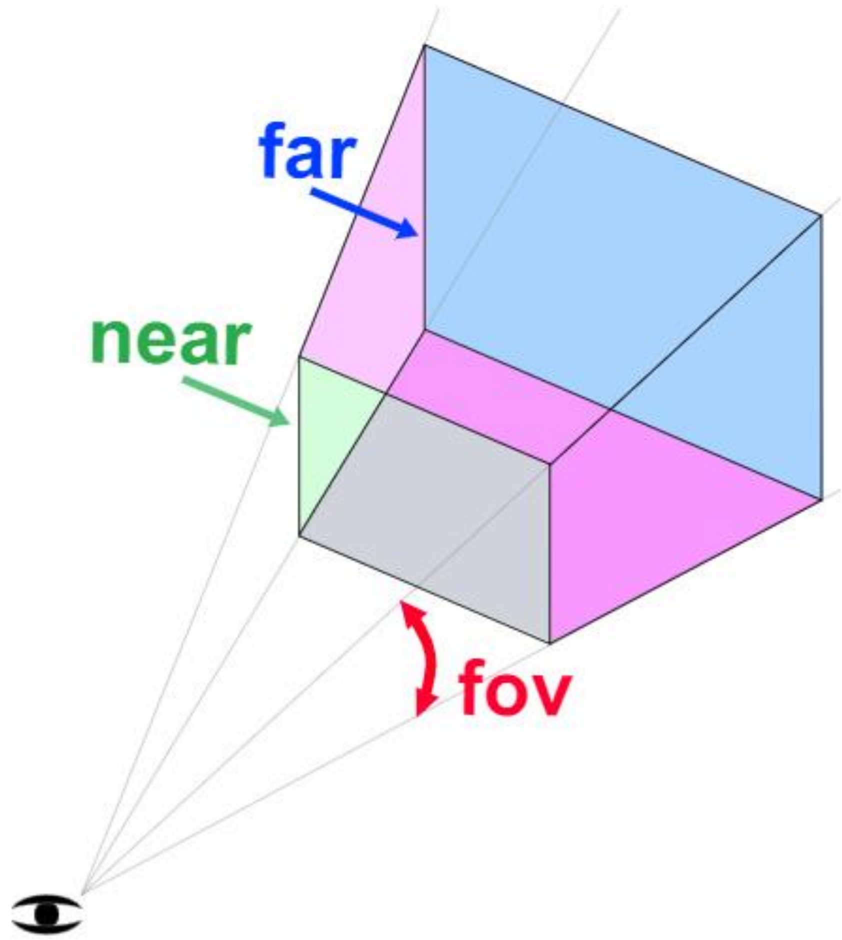
Three.js – PerspectiveCamera

```
// The CAMERA

// --- Where the viewer is and how he is looking at the scene

camera = new THREE.PerspectiveCamera( 70,

                    window.innerWidth / window.innerHeight, 1, 1000 );

camera.position.z = 400;
```

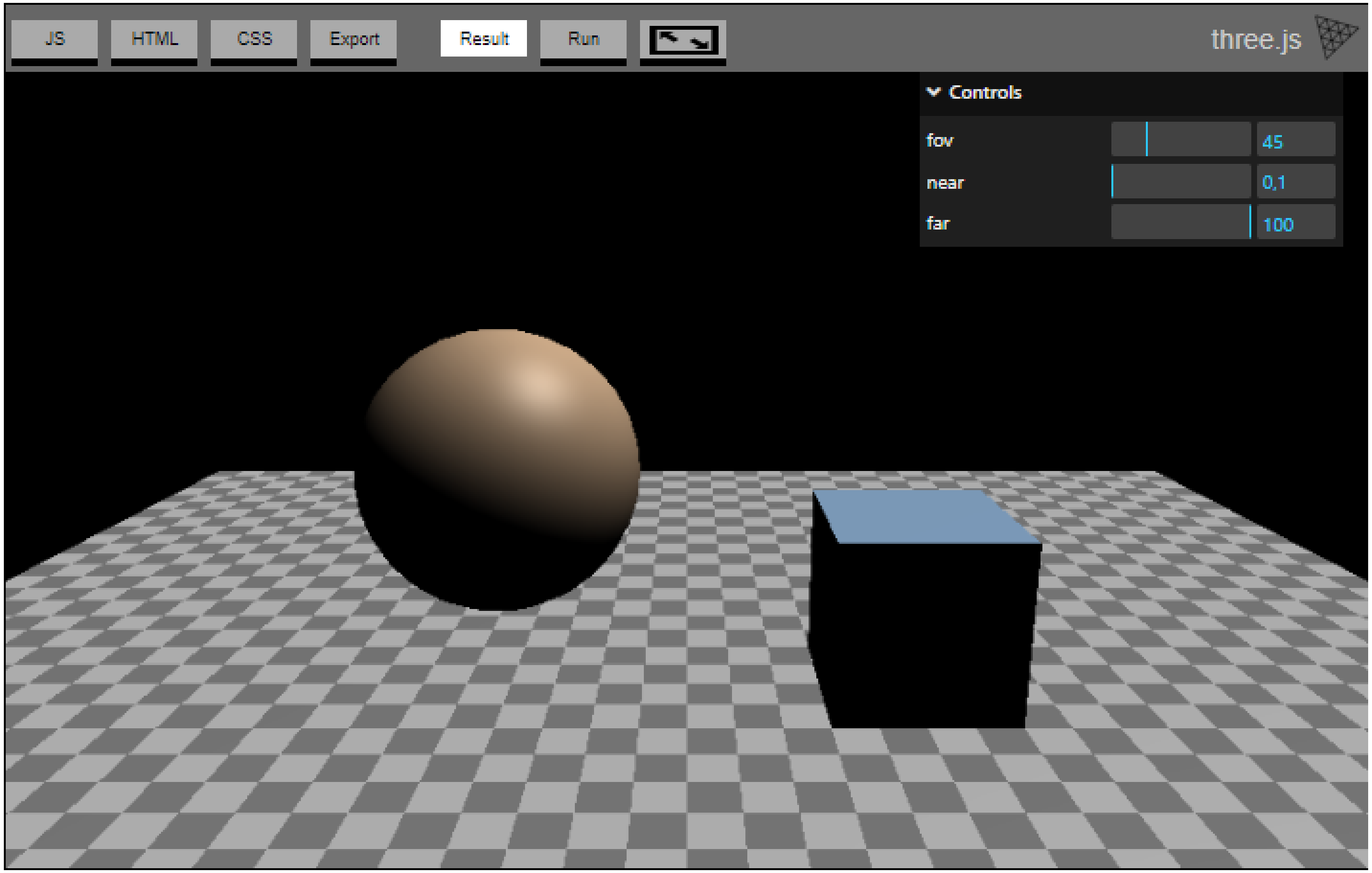
Three.js – PerspectiveCamera

- The most common camera in three.js
- Originates a 3D view where things in the distance appear smaller than things up close
- Defines a frustum as perspective view-volume

Three.js – Perspective frustum

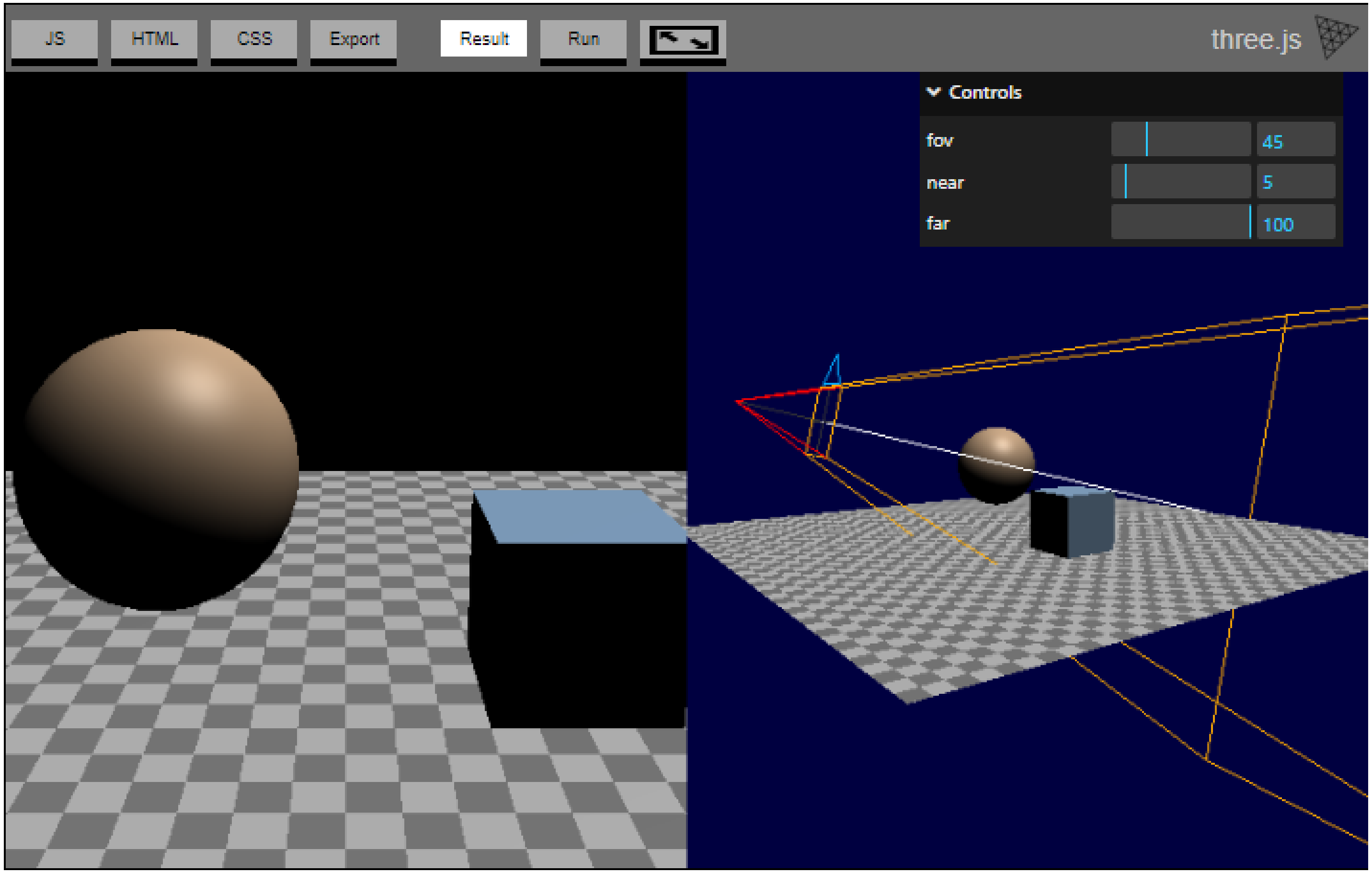


Three.js – PerspectiveCamera



https://threejs.org/manual/examples/cameras-perspective.html

Three.js – PerspectiveCamera



https://threejs.org/manual/examples/cameras-perspective-2-scenes.html

Three.js – OrthographicCamera

Cameras

ArrayCamera
Camera
CubeCamera
OrthographicCamera
PerspectiveCamera
StereoCamera

Constants

Constructor

```
OrthographicCamera( left : Number, right : Number, top : Number, bottom : Number, near : Number, far : Number )
```

left — Camera frustum left plane.
right — Camera frustum right plane.
top — Camera frustum top plane.
bottom — Camera frustum bottom plane.
near — Camera frustum near plane.
far — Camera frustum far plane.

[https://threejs.org/docs/index.html#api/en/cameras/OrthographicCamera]

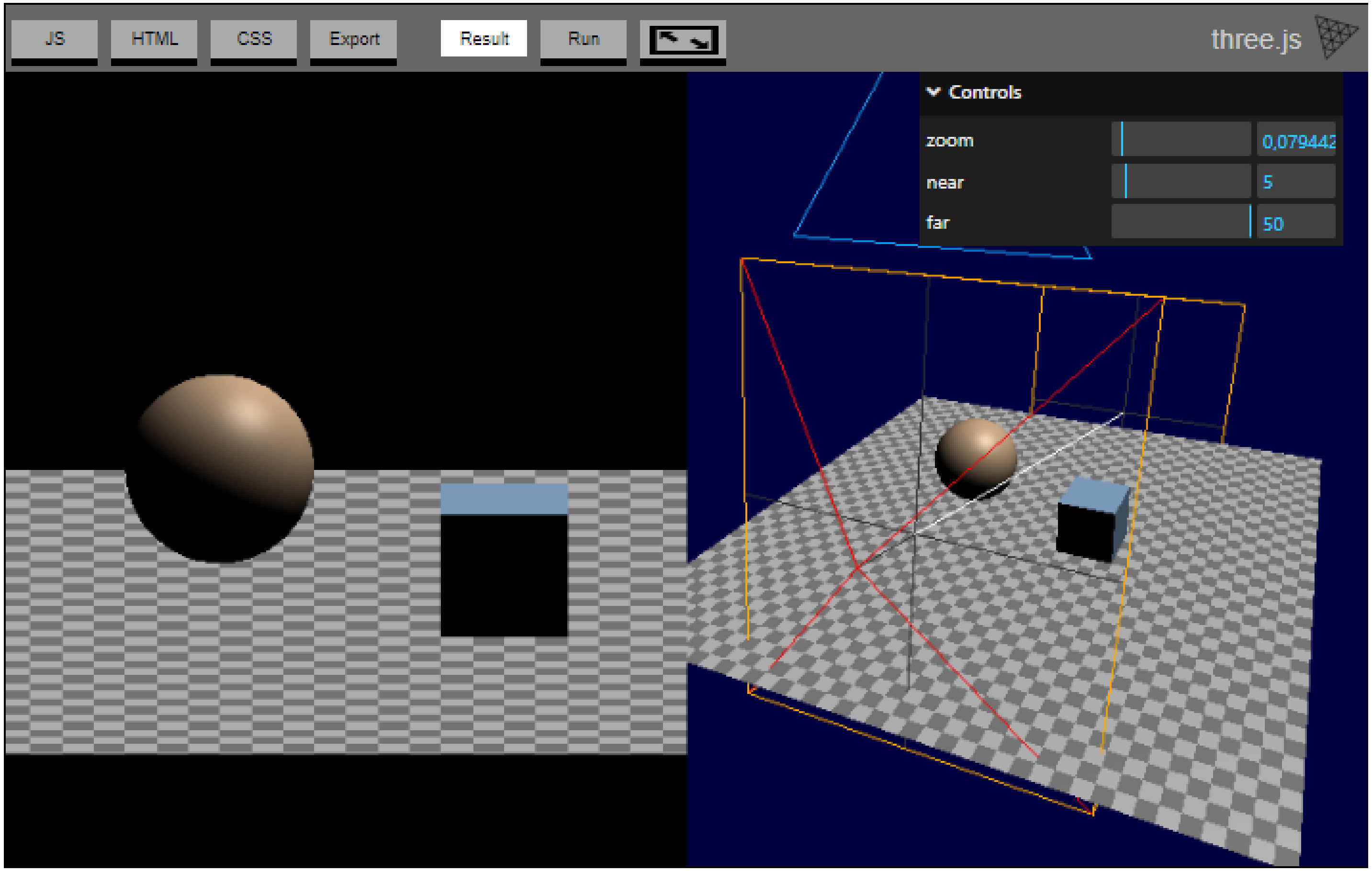
Three.js – OrthographicCamera

- Originates a 3D view using the orthogonal parallel projection

- Defines a box as orthographic view-volume

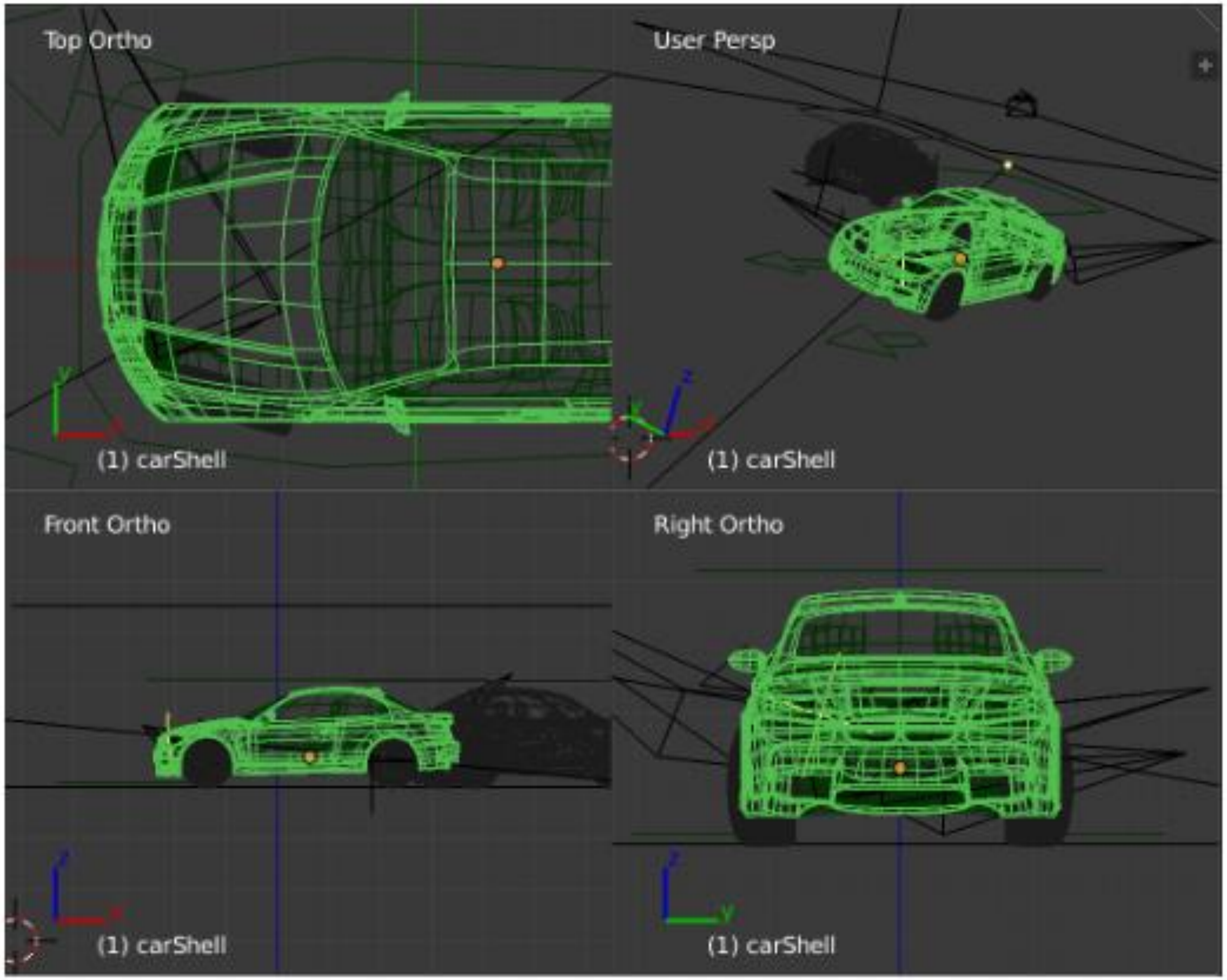
- Useful for 2D applications
- And for traditional modeling / design / engineering scenarios

Three.js – OrthographicCamera



https://threejs.org/manual/examples/cameras-orthographic-2-scenes.html

Three.js – OrthographicCamera



ACKNOWLEDGMENTS

Acknowledgments

- Some ideas and figures have been taken from slides of other CG courses.
- In particular, from the slides made available by Beatriz Sousa Santos, Ed Angel and Andy van Dam.
- Thanks!

Acknowledgment

- Example code and figures taken from

  https://threejs.org/manual/#en/cameras