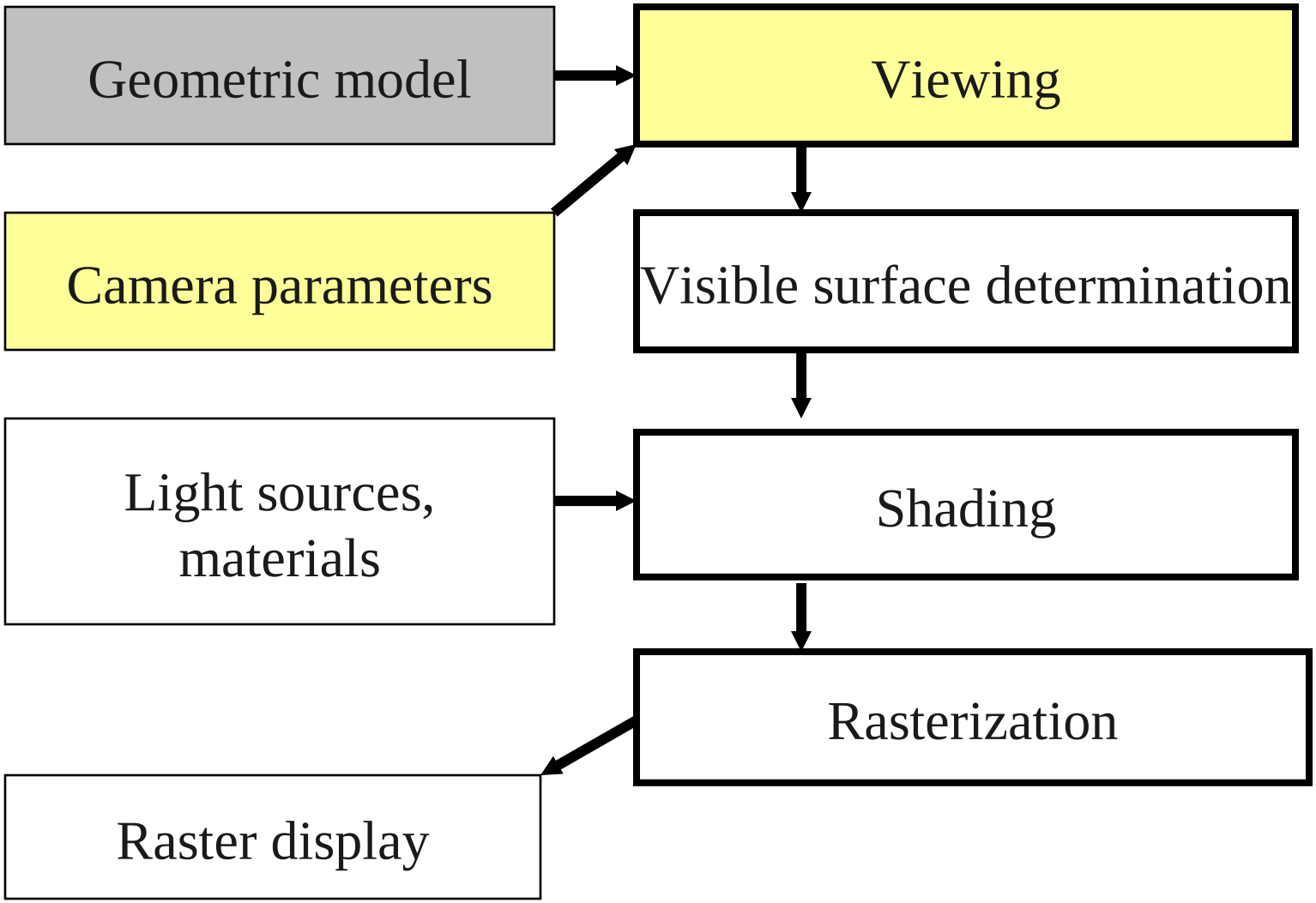
7M836
Animation & Rendering

Viewing, clipping, projection, visible surface determination

Arjan Kok, Kees Huizing, Huub van de Wetering

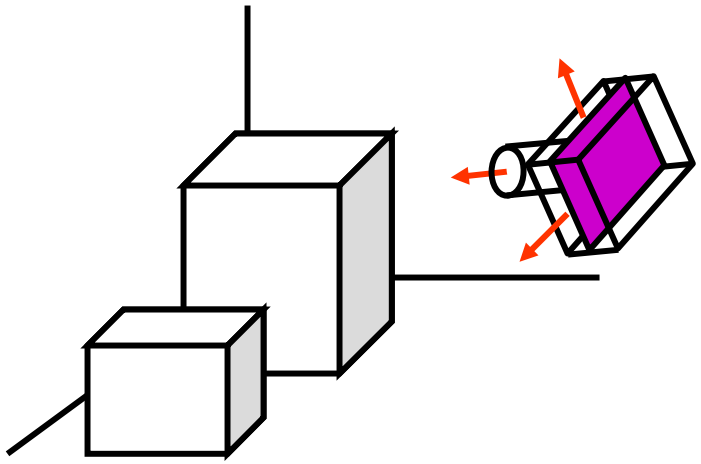
h.v.d.wetering@tue.nl

Graphics pipeline



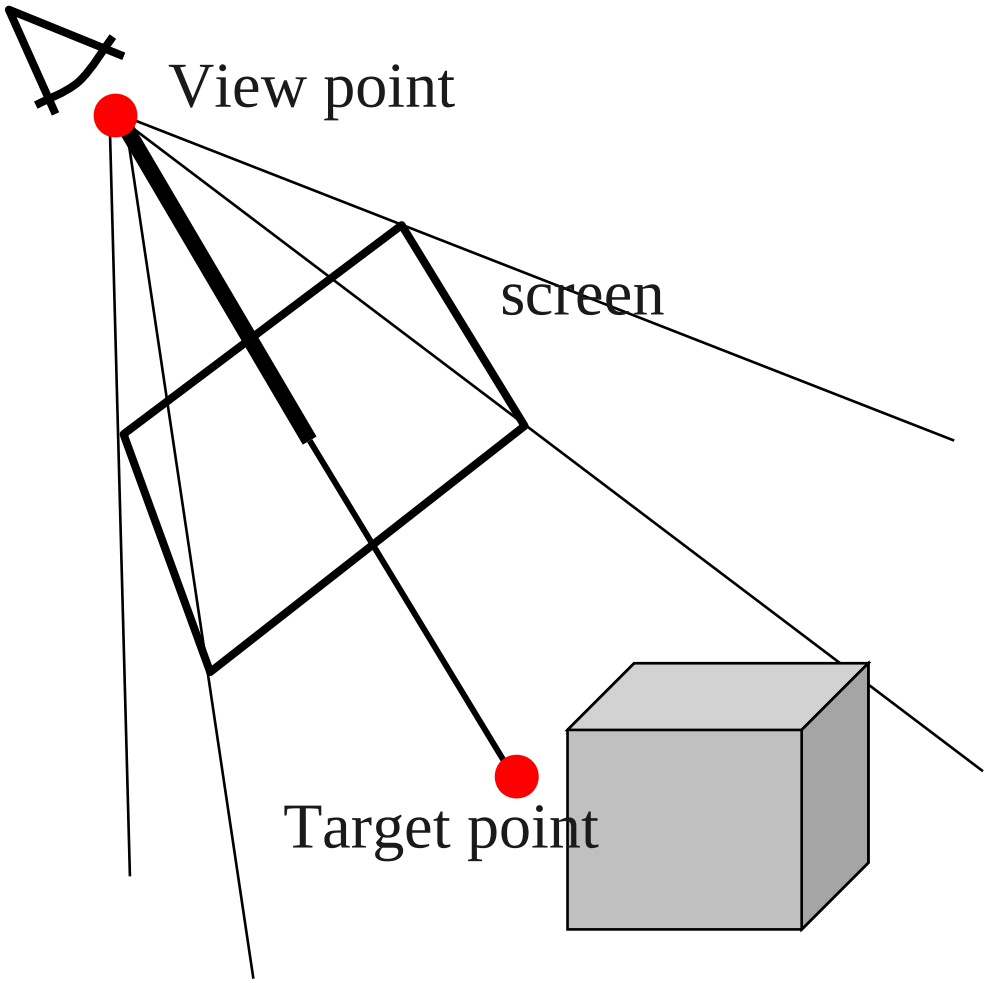
Camera

- Given a 3D model, how do we project this model on screen?

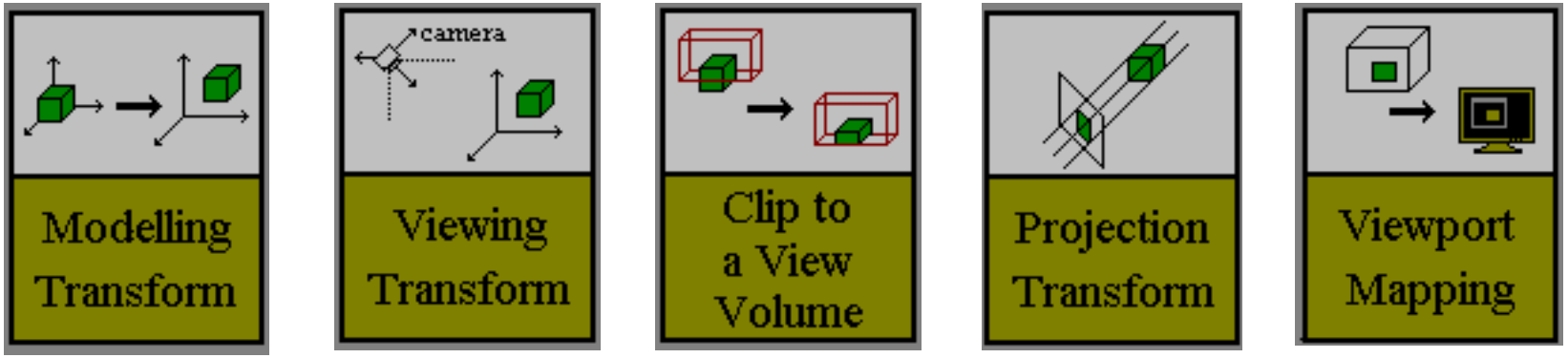


- Camera model
  - Eye/view point:
    - From which point do we look at the scene?
  - Target point or viewing direction:
    - Where do we look at?
  - Viewing angles:
    - What lens do we use?

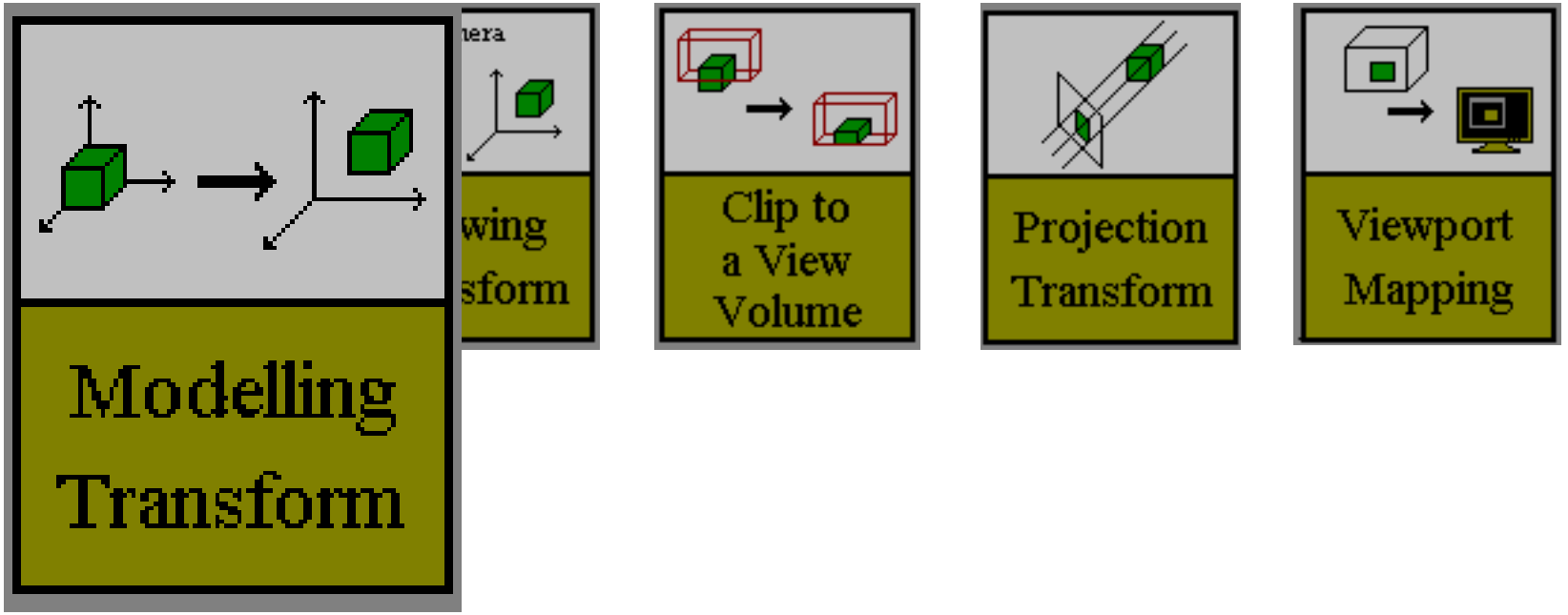
Camera



Viewing pipeline

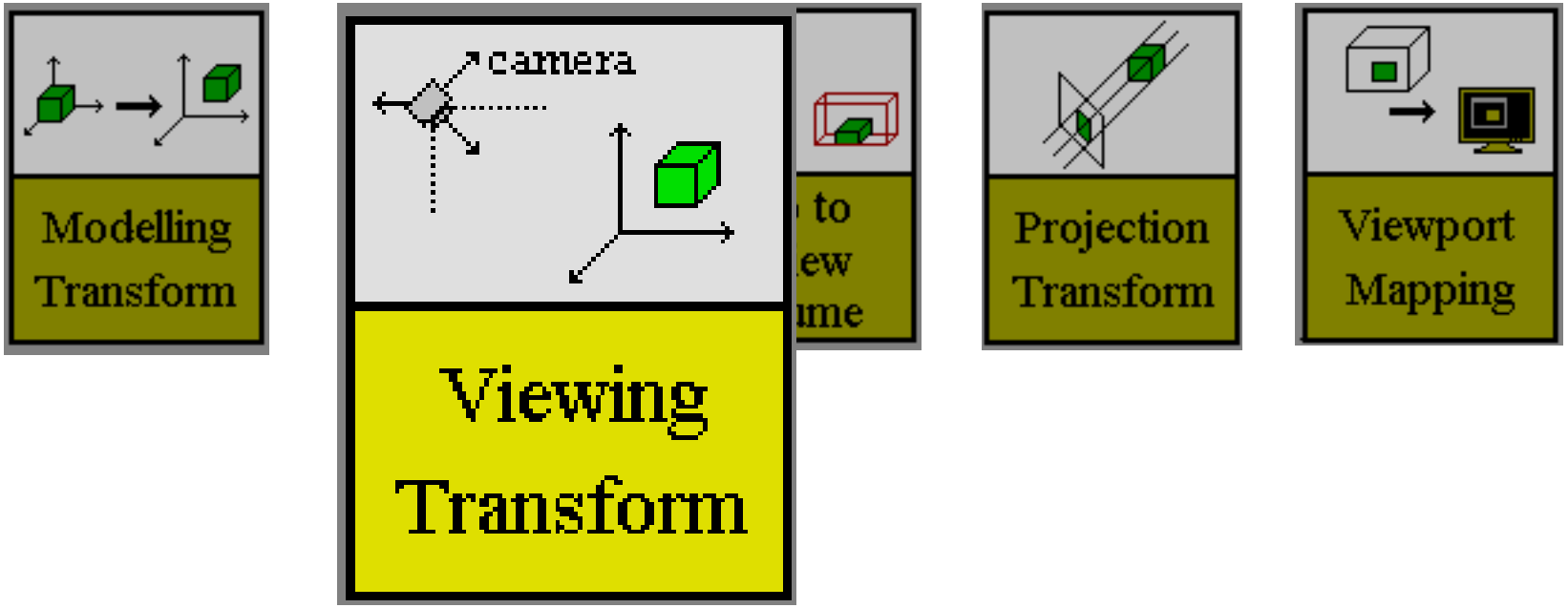


Viewing pipeline



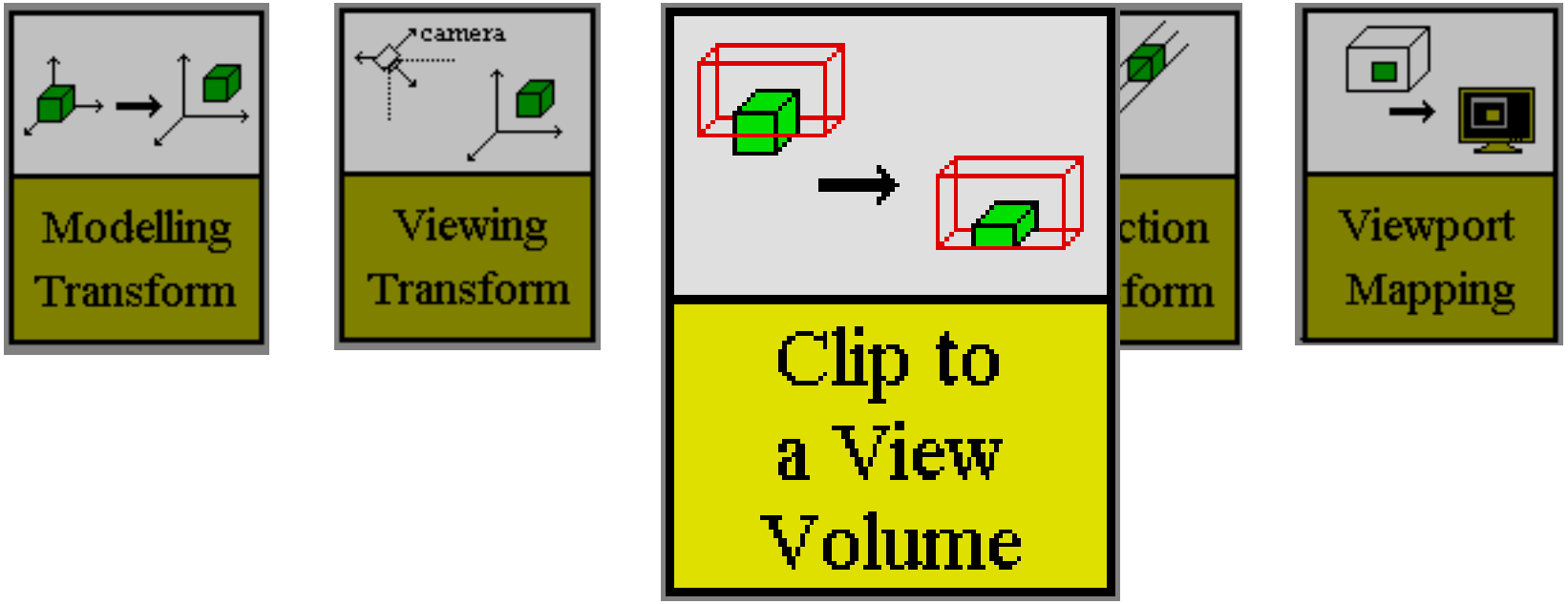
- Modelling transform
  - Transforms model coordinates into 3D world coordinates

Viewing pipeline



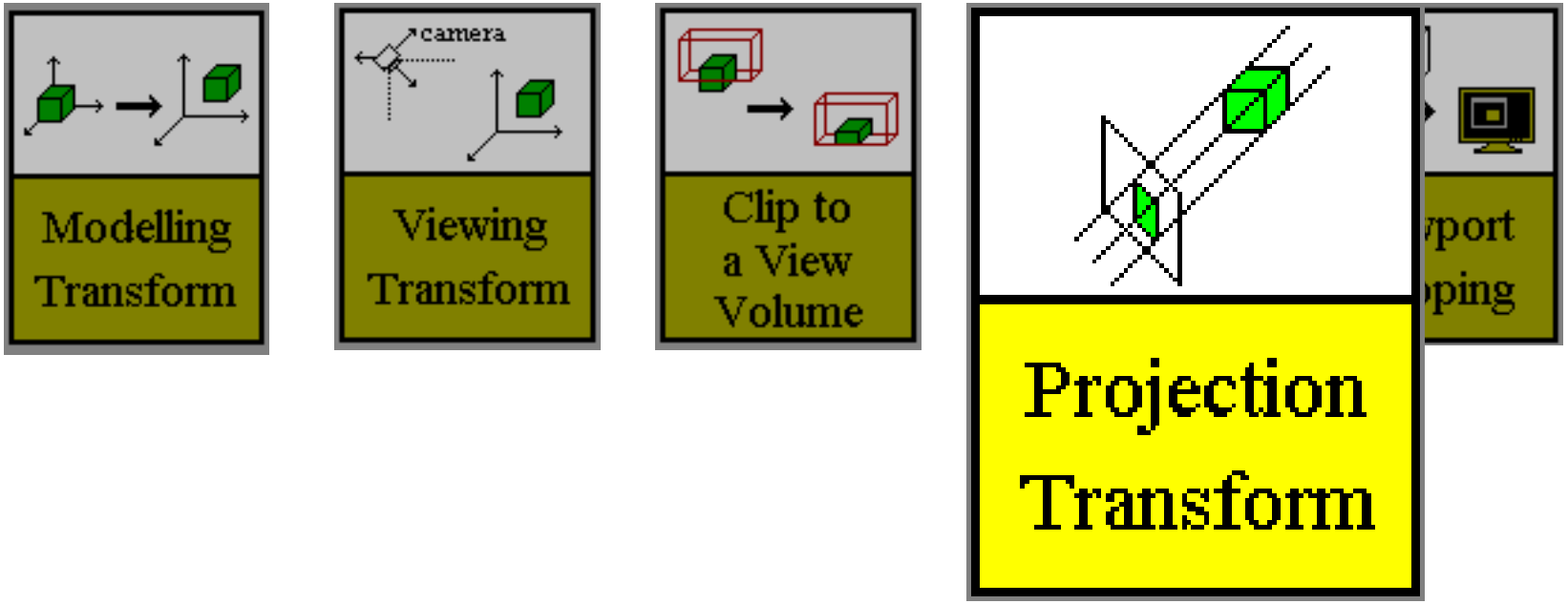
- Viewing (camera) transform
  - Transforms 3D world coordinates into 3D camera coordinates

Viewing pipeline



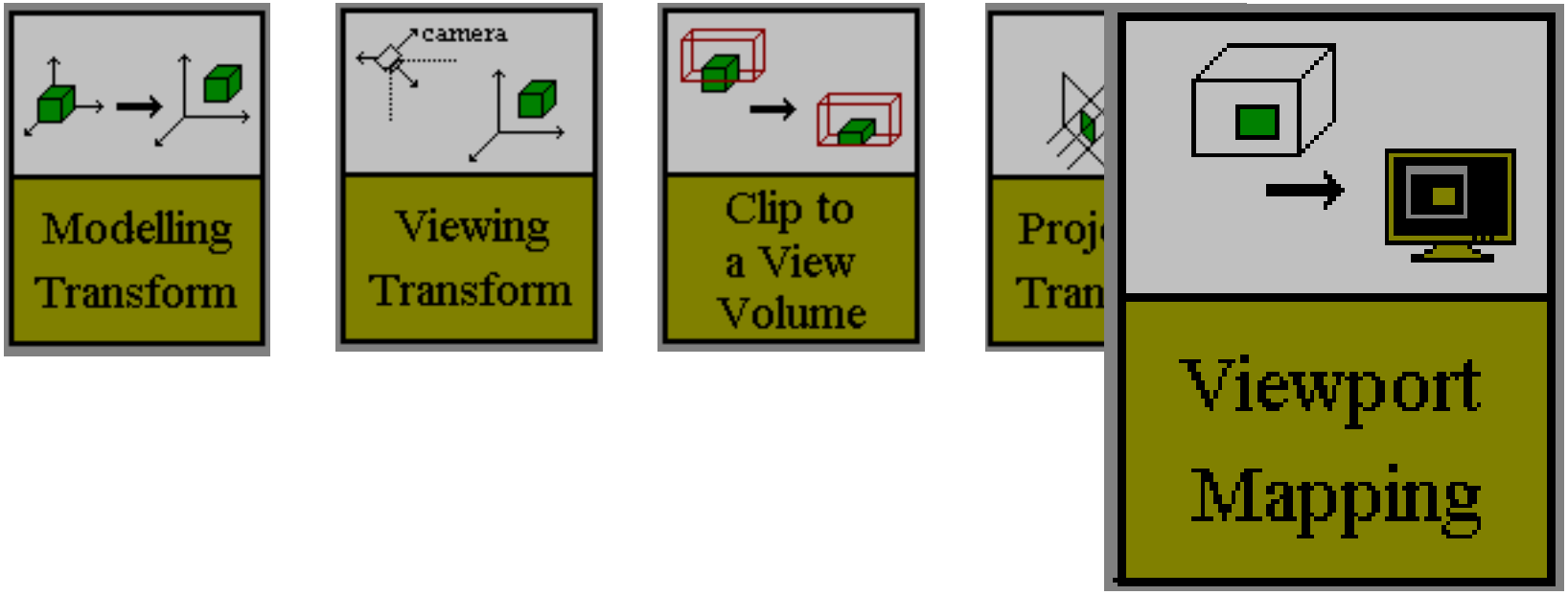
- Clipping to view volume
  - Determines which parts of objects might be visible , i.e. reside inside view volume

Viewing pipeline



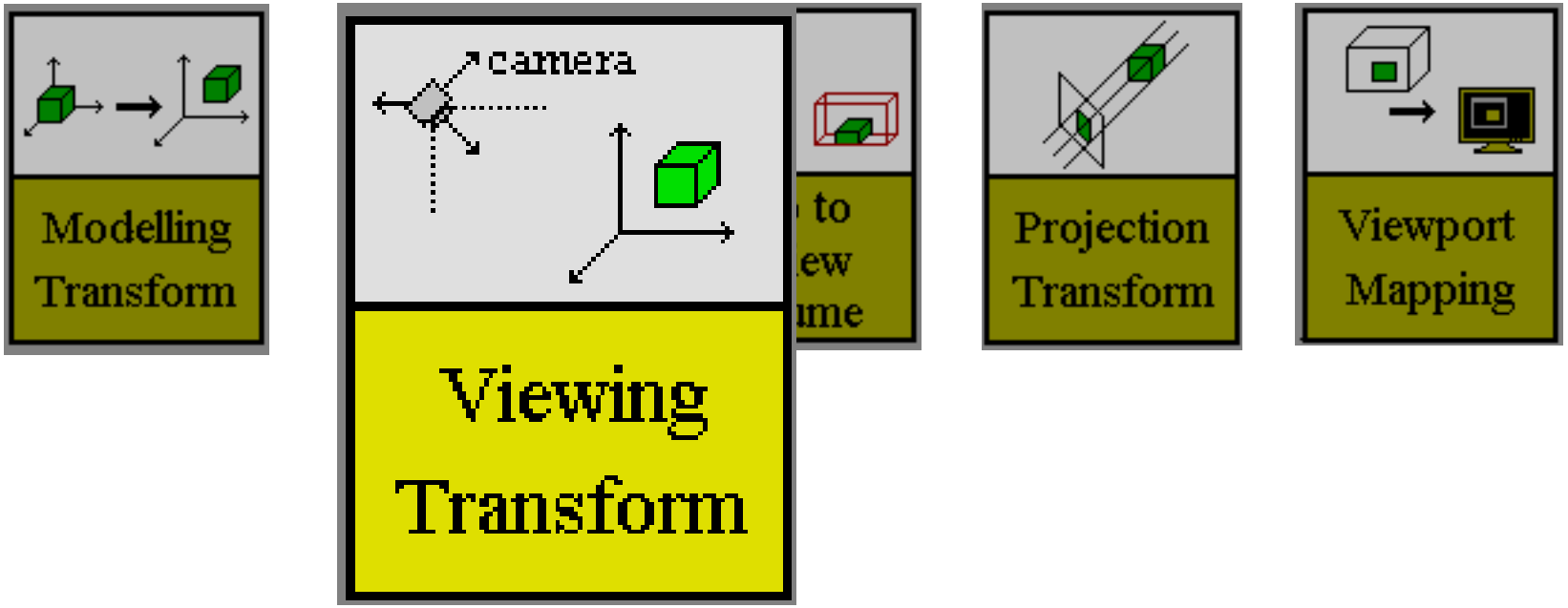
- Projection
  - Transforms 3D camera coordinates into 2D screen coordinates

Viewing pipeline



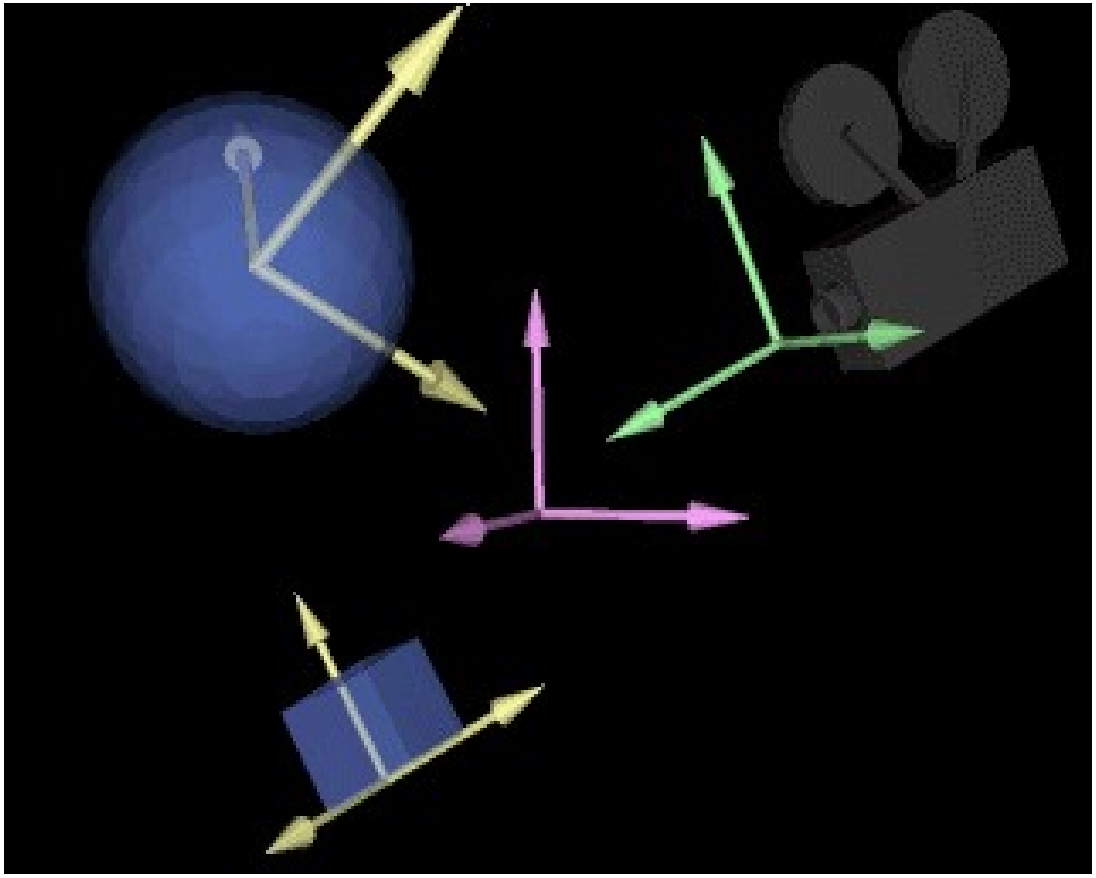
- Viewport mapping
  - Transforms 2D screen coordinates to pixels

Viewing pipeline

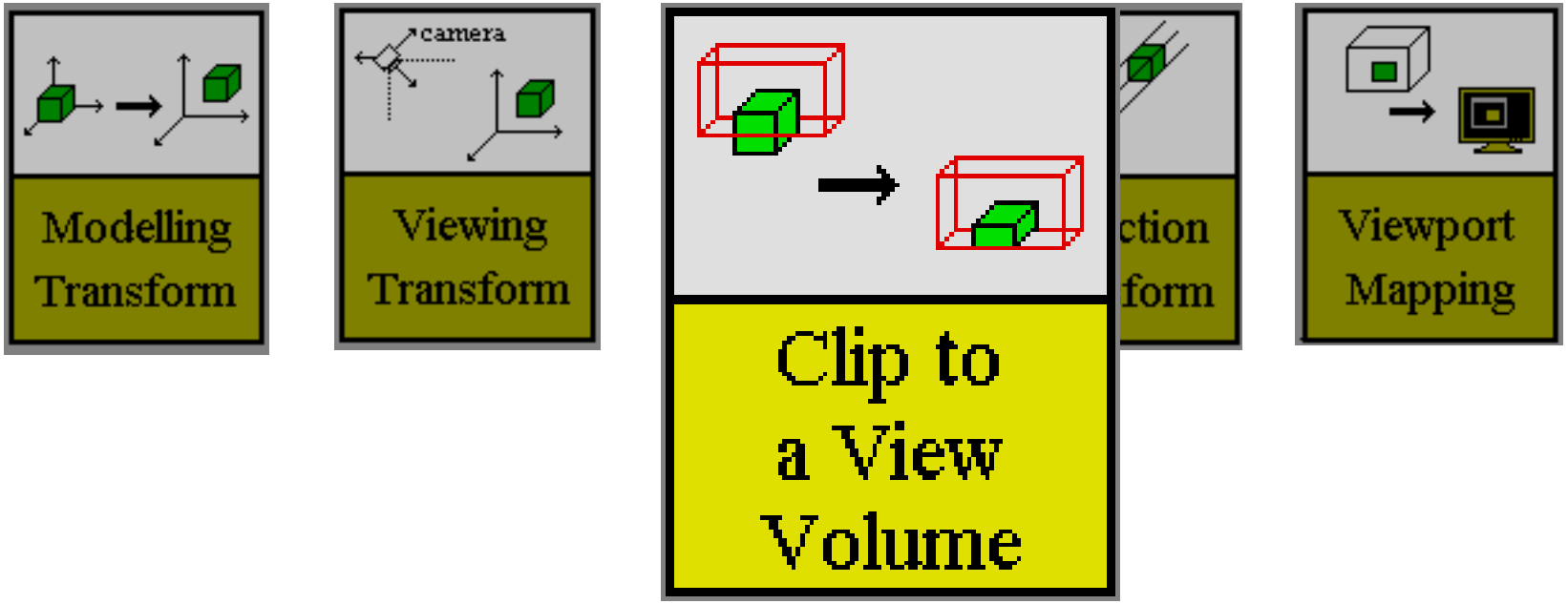


Viewing transform

- Transforms world coordinates to viewing (camera) coordinates



Viewing pipeline

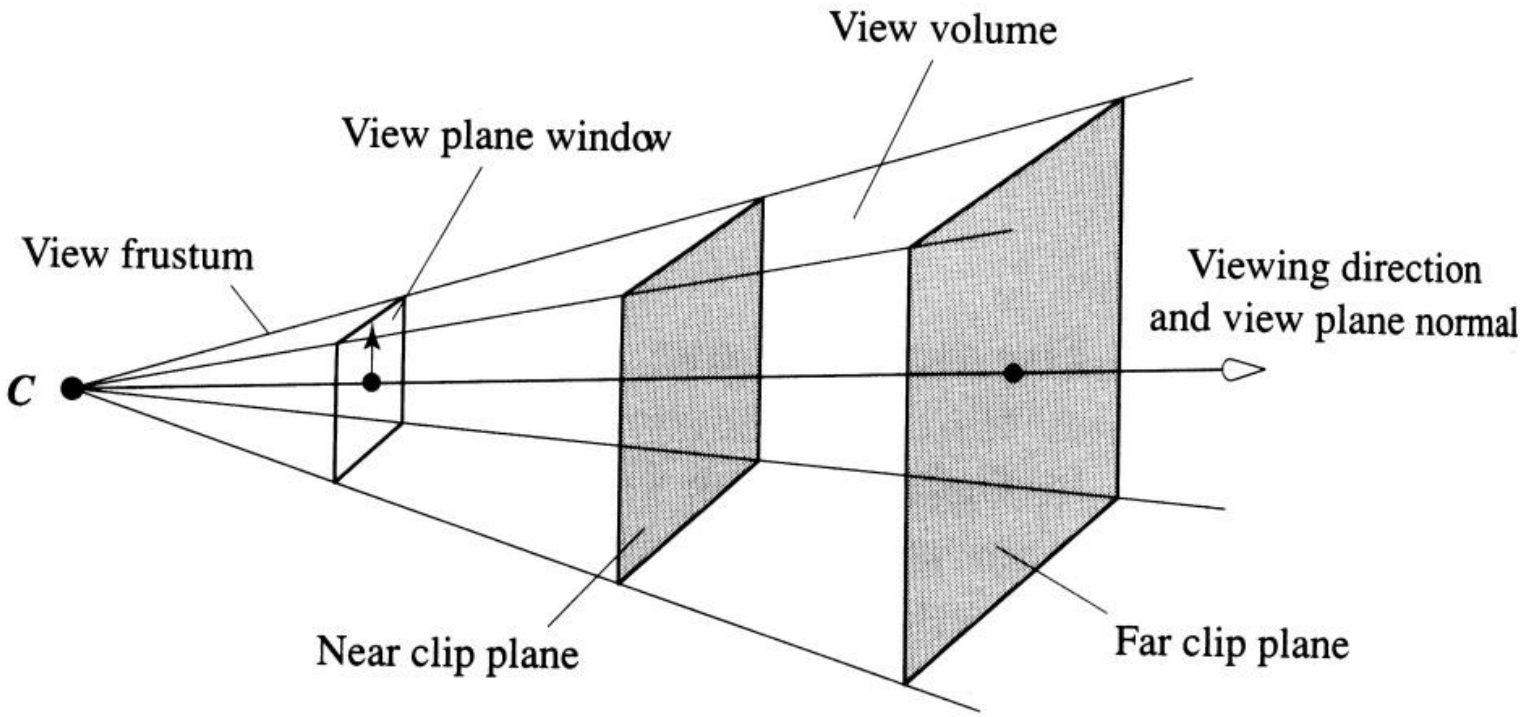


Clipping

- Part of geometry can reside outside window
- *Clipping* removes geometry outside window
- Draw only primitives (partly) inside window

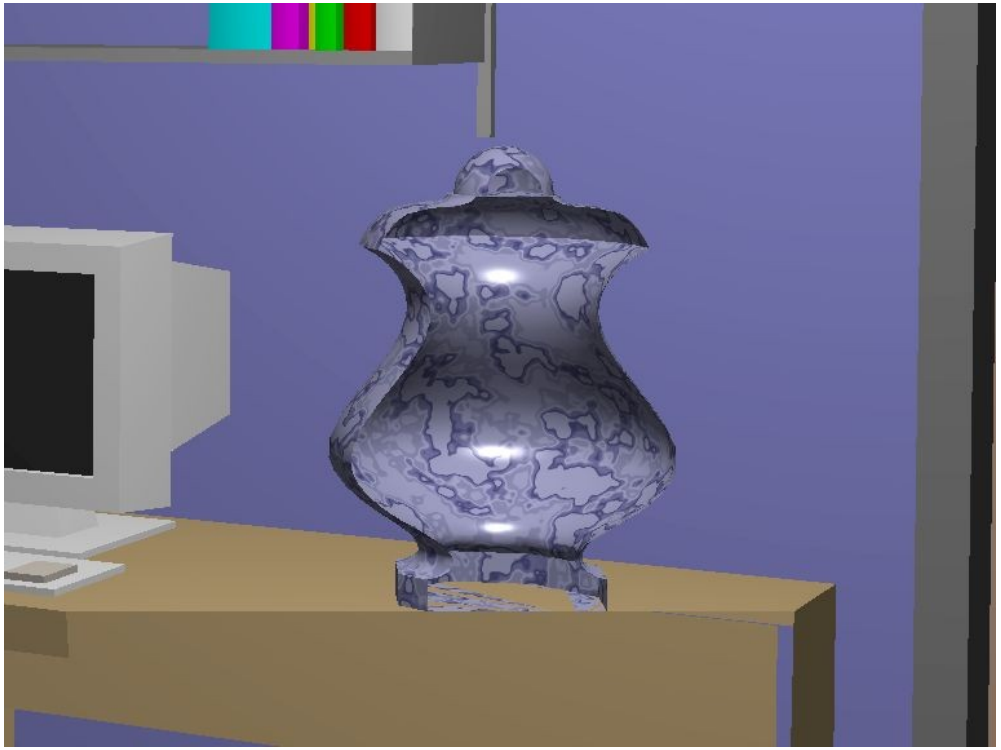
Clipping

Clipping


View volume
View plane window
View frustum
Viewing direction
and view plane normal
C
Near clip plane
Far clip plane

Near-far clipping

- Remove/clip objects closer than near plane or farther than far plane



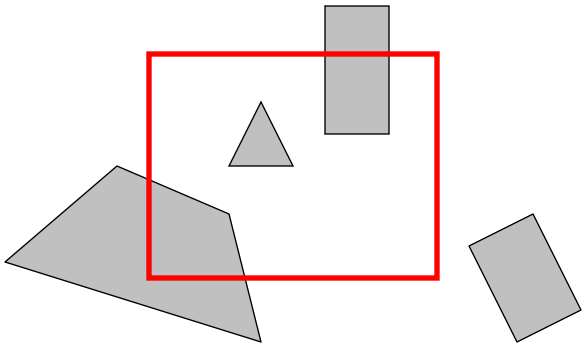
Clipping

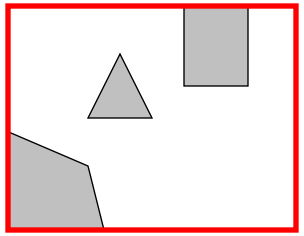
- Point clipping
  - Test on which side of the viewing-volume planes a point is situated
- Line clipping
  - Determine part of line within viewing volume
  - Cohen-Sutherland, Liang-Barskey
- Polygon clipping
  - Determine part of polygon within viewing volume
  - Sutherland-Hodgman

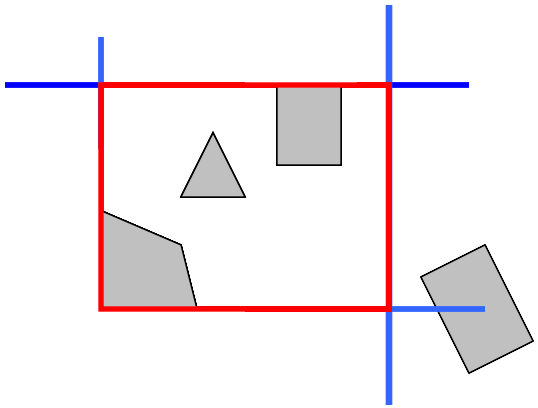
Sutherland-Hodgman clipping

- Clip each polygon against each plane (4 or 6) of viewing volume
- For each polygon and each plane determine which part of polygon is on inside of viewing-volume plane

Sutherland-Hodgman clipping



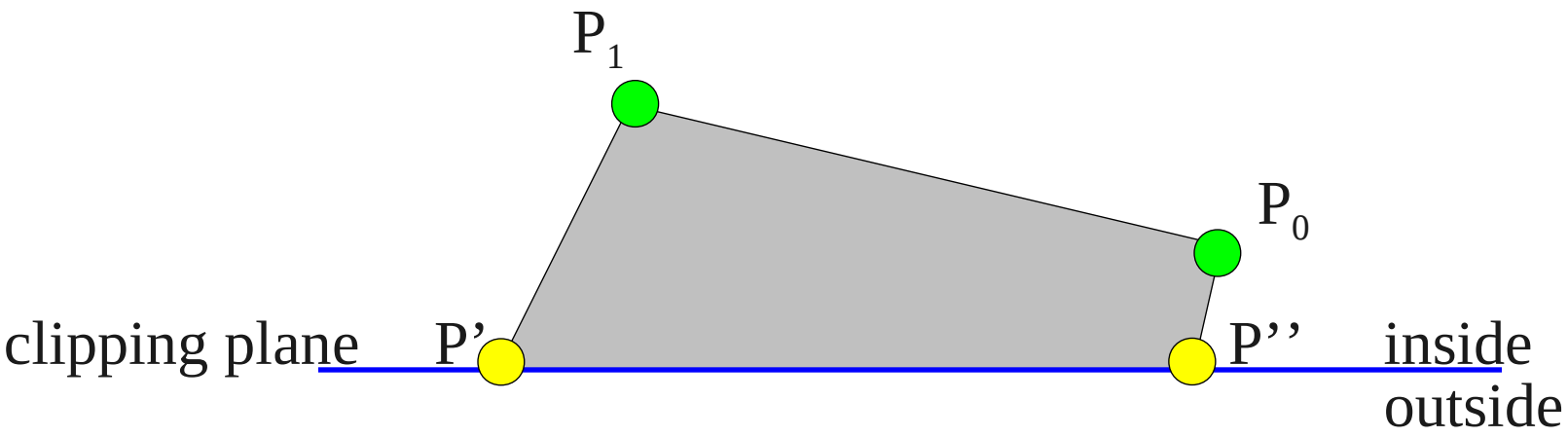




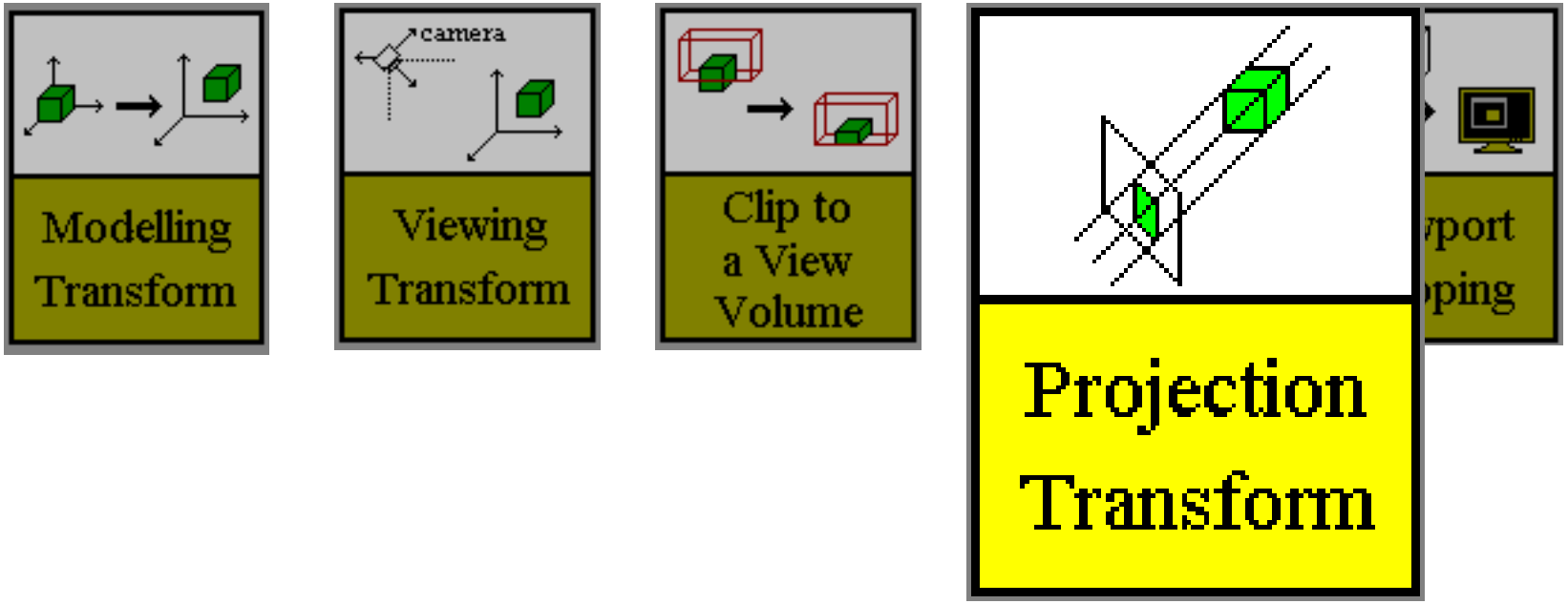
Sutherland-Hodgman clipping

- Clip polygon against plane:
  - Determine if first vertex inside/outside volume
  - If inside volume, store vertex
  - For all consecutive vertices of polygon
    - Determine is vertex inside/outside volume
    - If from inside to outside, store intersection point
    - If from outside to inside, store intersection point and vertex
    - If from inside to inside, store vertex
    - If from outside to outside, store nothing

Sutherland-Hodgman clipping



Viewing pipeline

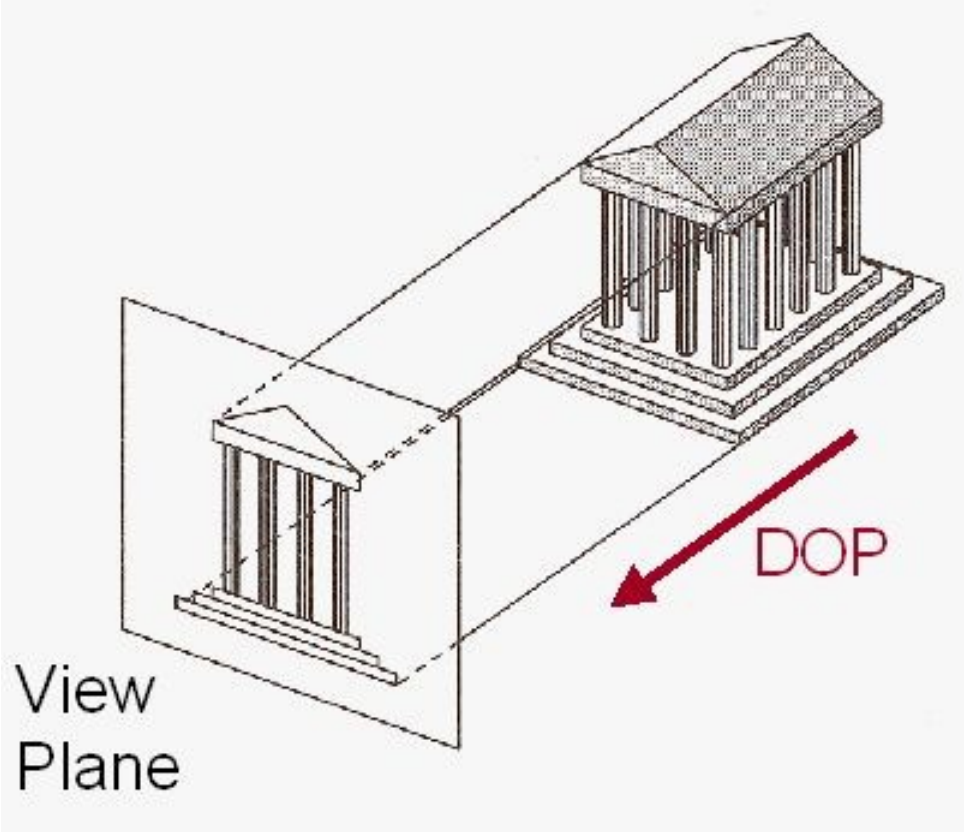


Projection

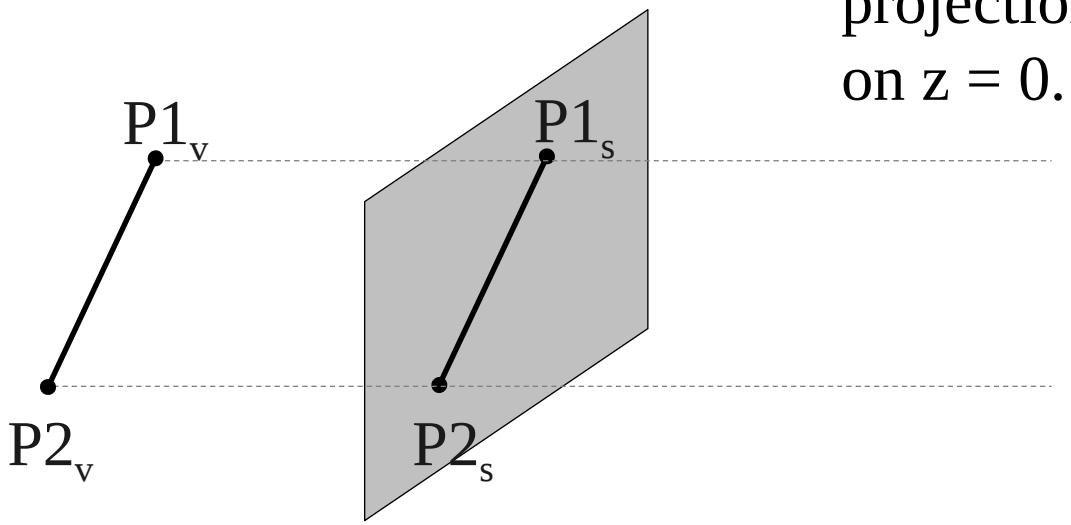
- Transform 3D camera coordinates to 2D screen coordinates

- Parallel (orthographic) projection
- Perspective projection

Parallel projection



Parallel projection



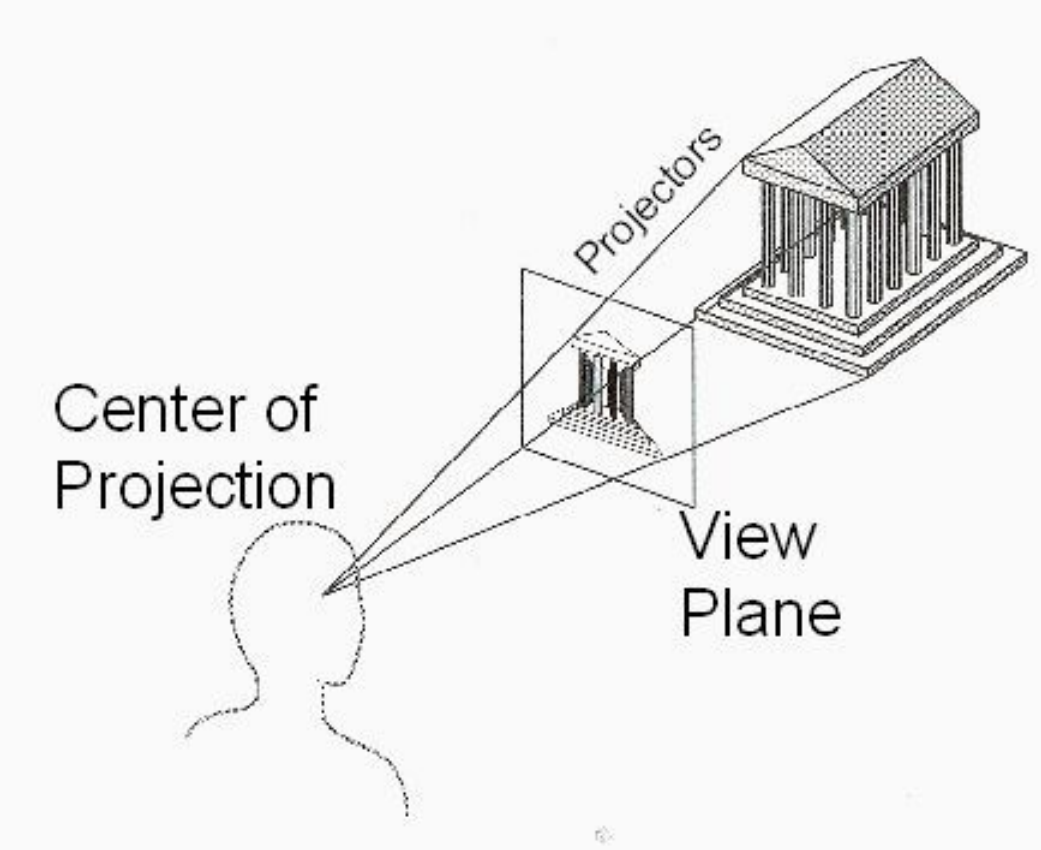
projection in z-directon
on z = 0.

$x_s = x_v$

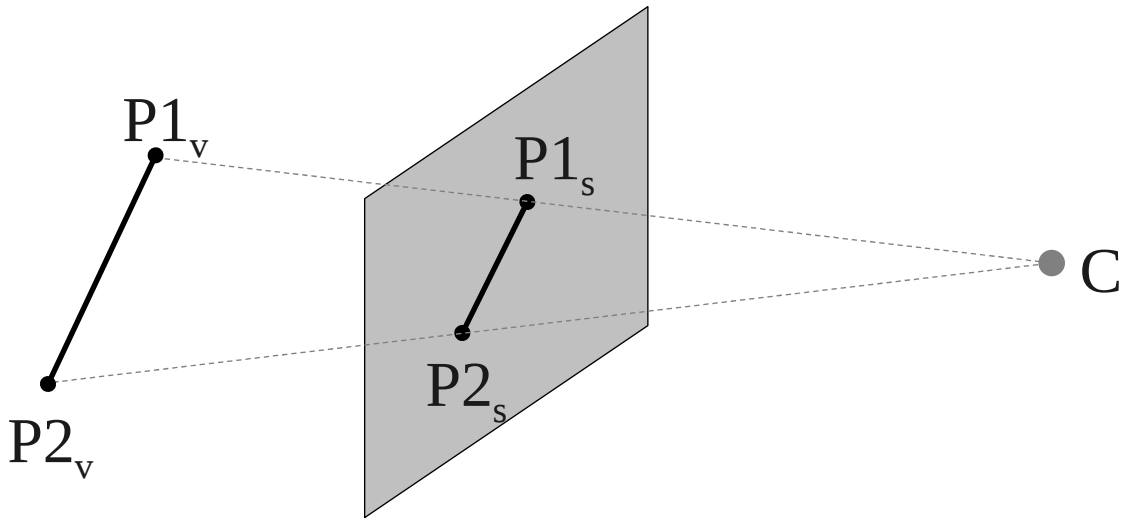
$y_s = y_v$

$z_s = 0$

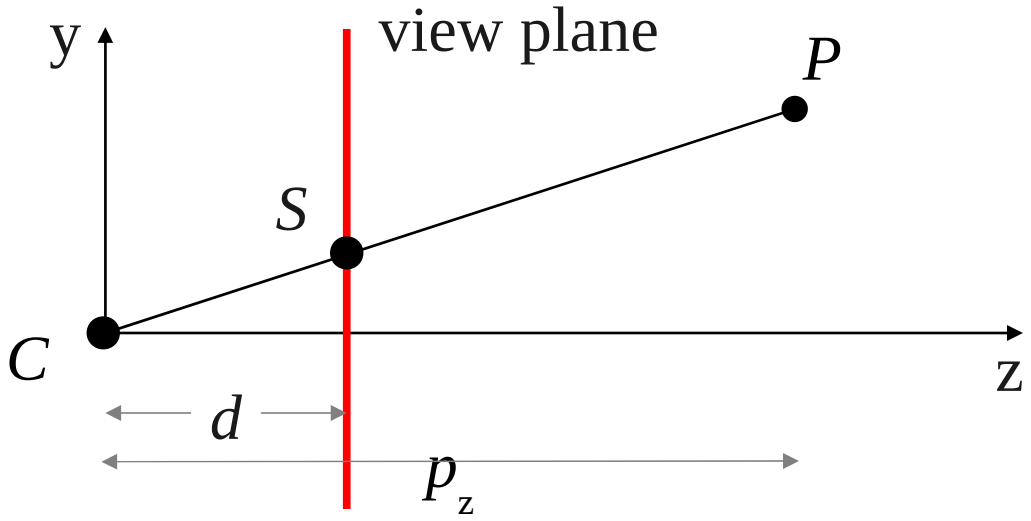
Perspective projection



Perspective projection



Simple perspective projection

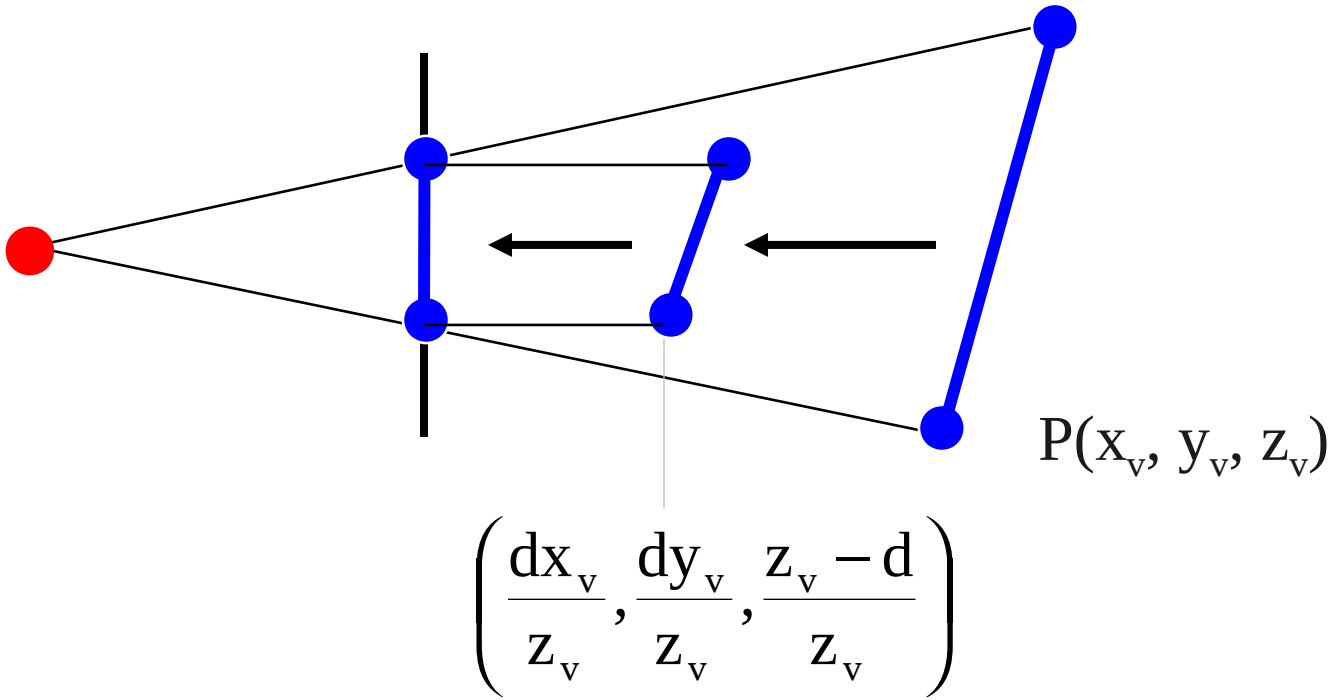


$$C=(0,0,0)$$
$$P=(p_x,p_y,p_z)$$
$$S=(s_x,s_y,s_z)$$
$$S=\frac{d}{p_z}P$$

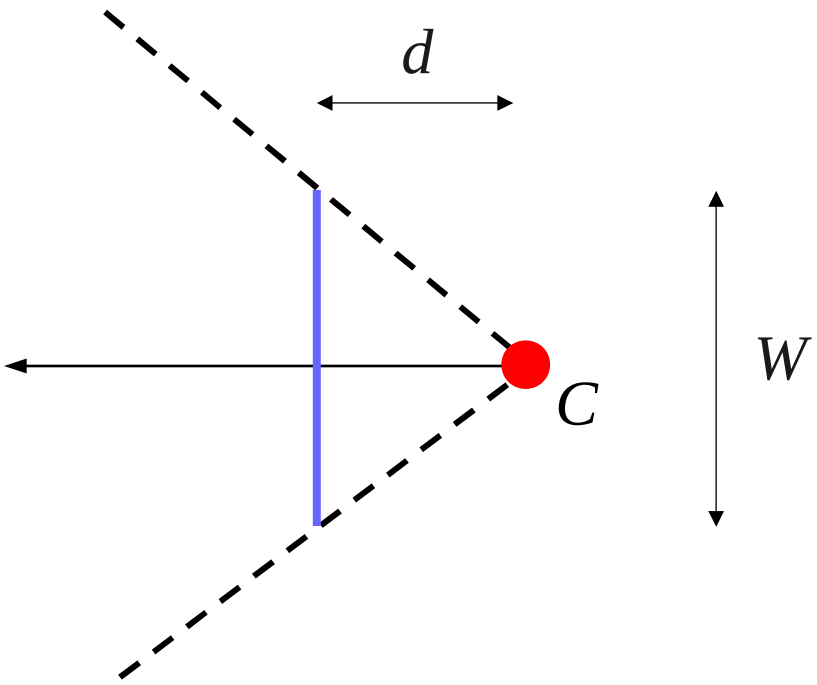
$$s_x=\frac{d}{p_z}p_x$$
$$s_y=\frac{d}{p_z}p_y$$
$$s_z=d$$

Perspective projection

- Perspective transform followed by isometric projection.

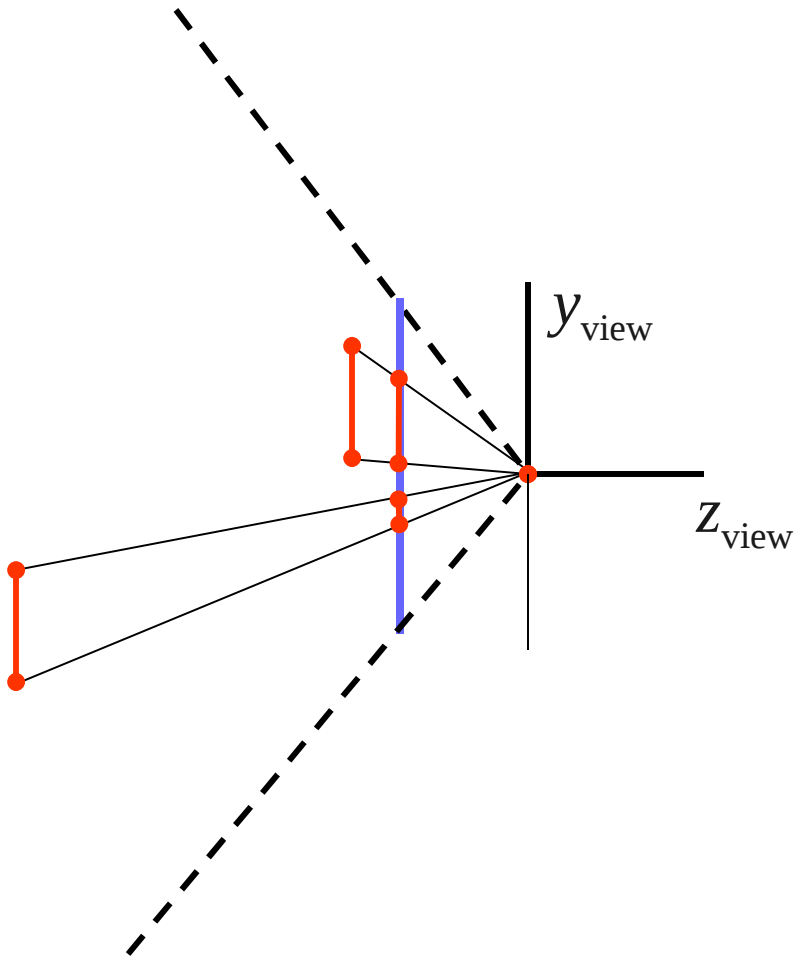


Perspective projection



- Ratio of *W* and *d* determines strength of perspective

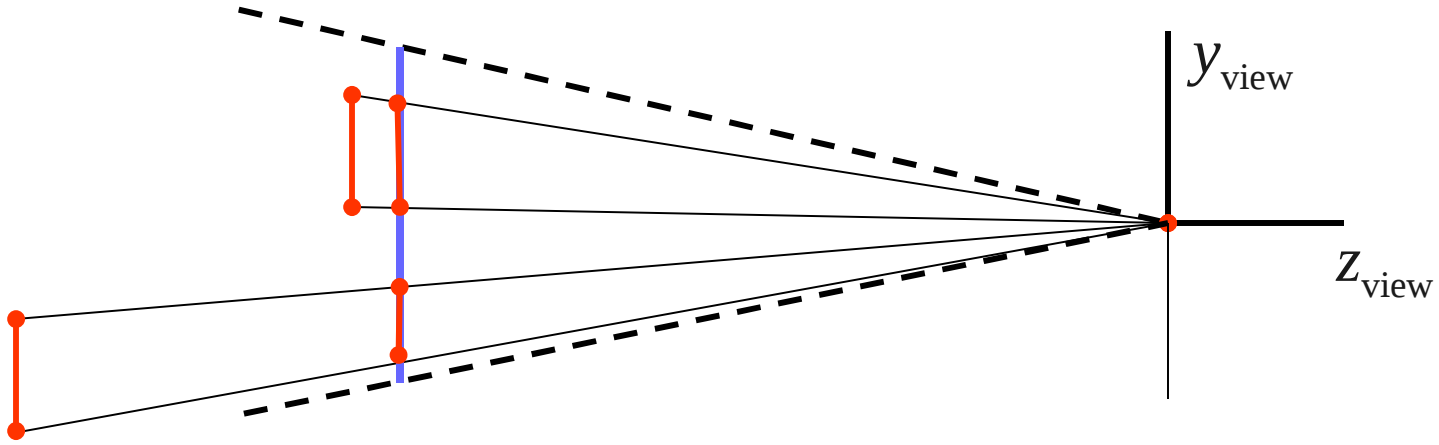
Perspective projection



Wide angle lens

Perspective projection

Tele lens

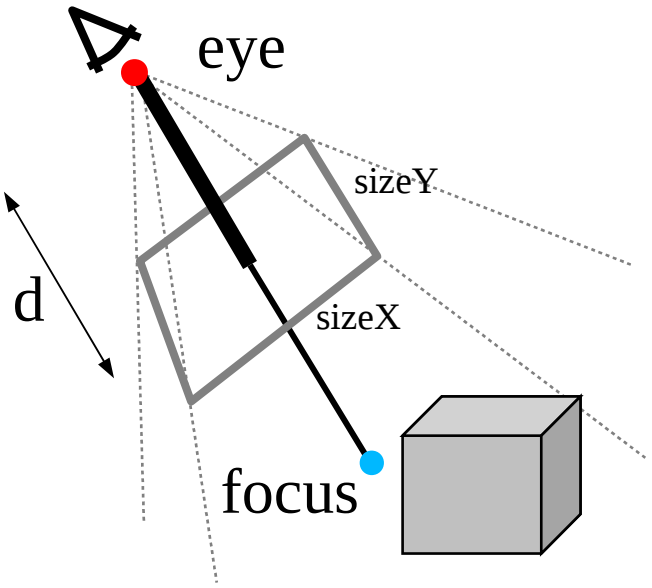


Povray – projection

```
#include "colors.inc"
#declare d     = 10;
#declare sizeX = 4;
#declare sizeY = 3;
#declare focus = <0,1,0>;
#declare eye   = <0, 2,-15>;

camera {
   perspective
   location   eye
   direction <0, 0, d>
   up         sizeY * y
   right      sizeX * x
   look_at    focus
}

light_source { <3,5,-2> color White  }
sphere { <0,1,0>,1 texture { pigment { White }} }   // radius 1
plane { <0,1,0>, 0                                  // xz plane
   texture { pigment {checker color White color Red } }
 }
```

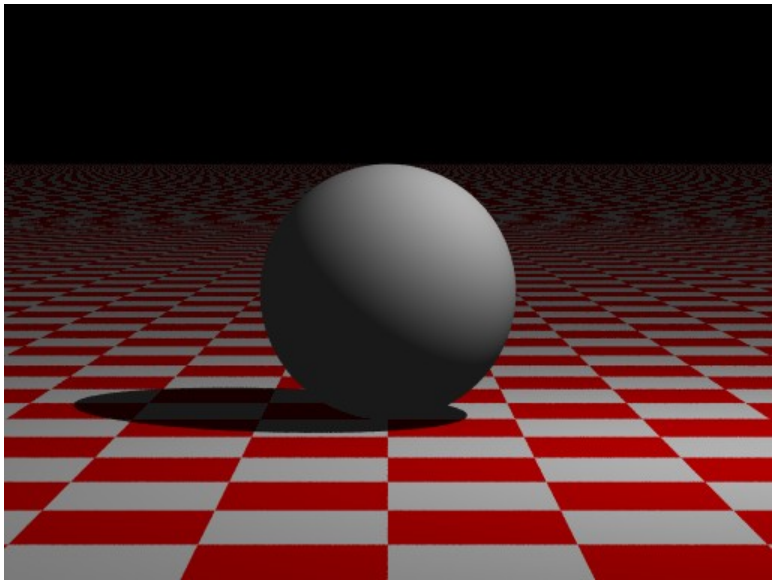


Povray – projection

```
#include "colors.inc"
#declare d     = 10;
#declare sizeX = 4;
#declare sizeY = 3;
#declare focus = <0,1,0>;
#declare eye   = <0, 2,-15>;

camera {
   perspective
   location   eye
   direction <0, 0, d>
   up         sizeY * y
   right      sizeX * x
   look_at    focus
}

light_source { <3,5,-2> color White  }
sphere { <0,1,0>,1 texture { pigment { White }} }   // radius 1
plane { <0,1,0>, 0                                  // xz plane
   texture { pigment {checker color White color Red } }
 }
```

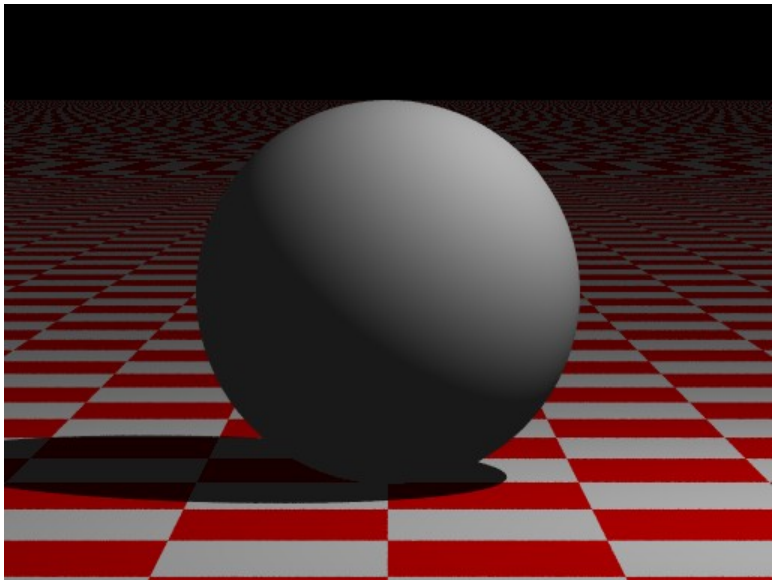


Povray – projection

```
#include "colors.inc"
#declare d     = 10 + 5 ;
#declare sizeX = 4;
#declare sizeY = 3;
#declare focus = <0,1,0>;
#declare eye   = <0, 2,-15>;

camera {
   perspective
   location   eye
   direction <0, 0, d>
   up         sizeY * y
   right      sizeX * x
   look_at    focus
}

light_source { <3,5,-2> color White  }
sphere { <0,1,0>,1 texture { pigment { White }} }   // radius 1
plane { <0,1,0>, 0                                  // xz plane
   texture { pigment {checker color White color Red } }
 }
```

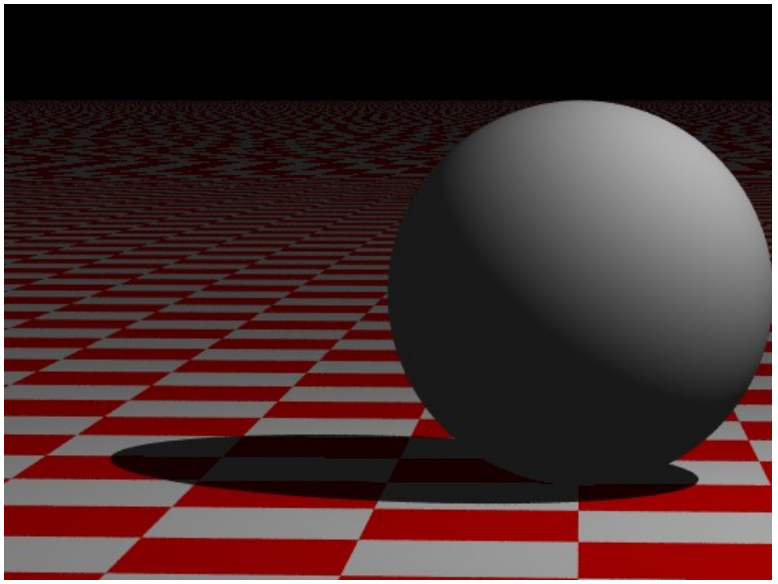


Povray – projection

```
#include "colors.inc"
#declare d     = 10 + 5 ;
#declare sizeX = 4;
#declare sizeY = 3;
#declare focus = <-1,1,0>;
#declare eye   = <0, 2,-15>;

camera {
   perspective
   location   eye
   direction <0, 0, d>
   up         sizeY * y
   right      sizeX * x
   look_at    focus
}

light_source { <3,5,-2> color White  }
sphere { <0,1,0>,1 texture { pigment { White }} }   // radius 1
plane { <0,1,0>, 0                                  // xz plane
   texture { pigment {checker color White color Red } }
 }
```

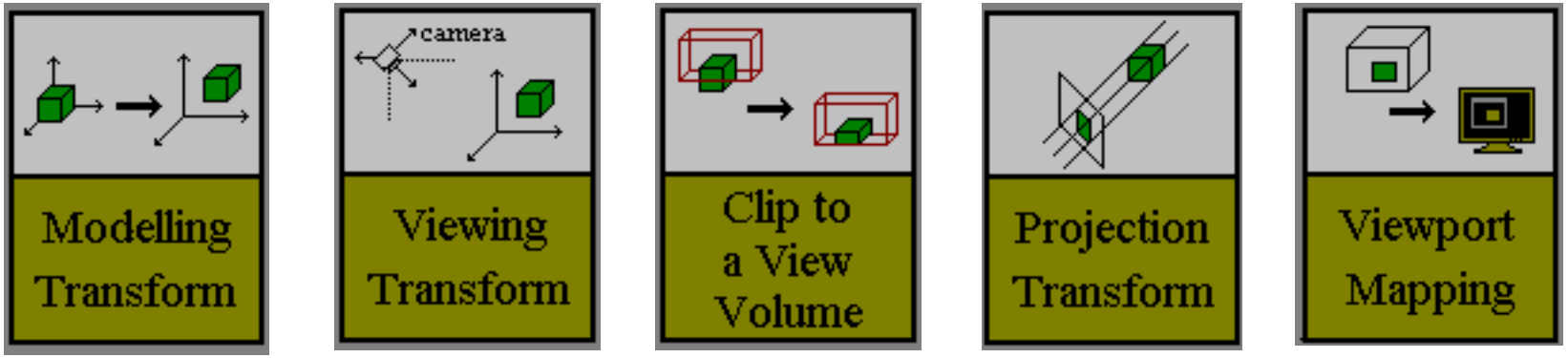


Projections

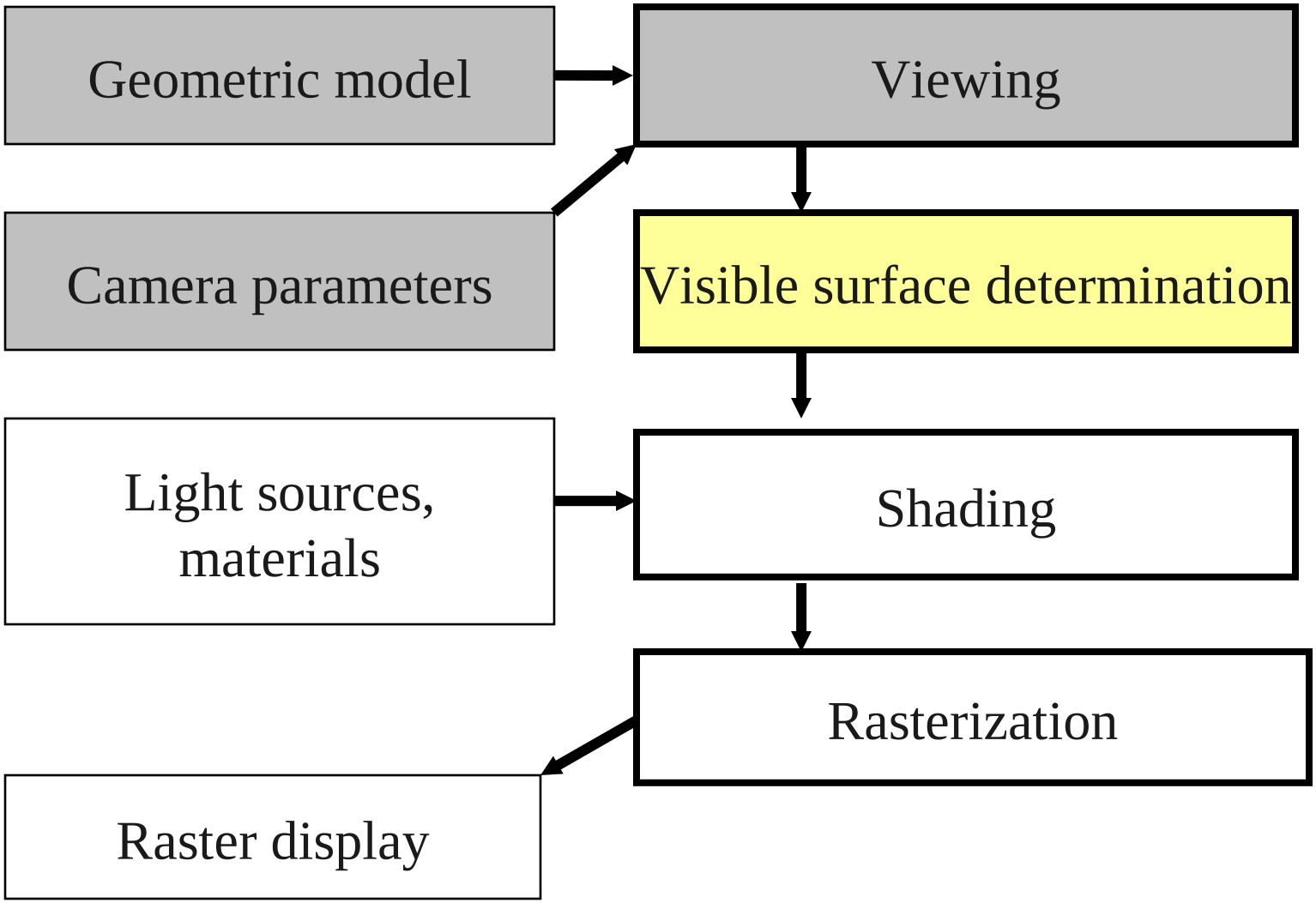
Projections

- Parallel projection
  - + Parallel lines remain parallel in image
  - + Used to measure in image
  - - Less realism
- Perspective projection
  - + Dimensions decrease with larger depth:
    - More realism
  - - Parallel lines do not remain parallel

Viewing pipeline



Graphics pipeline



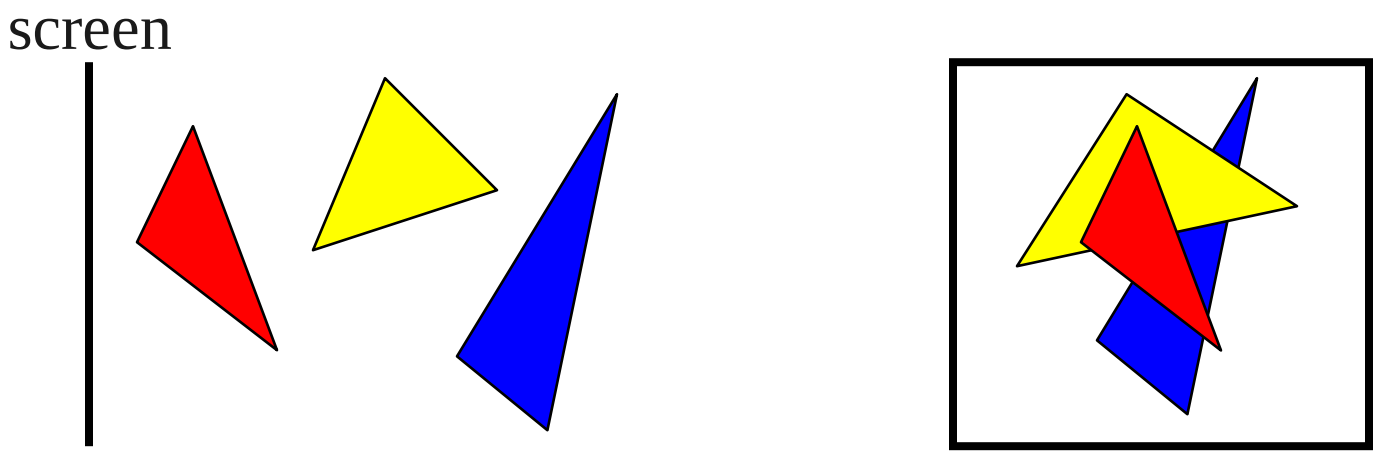
Visible surface determination

- Also called: *hidden surface removal*

- Determine which objects, or parts of objects, are visible on screen, given the position and direction of the camera
- Display only the visible (parts of) objects

- Several algorithms
  - Complexity of scene
  - Type of objects
  - Hardware support

Visible surface determination

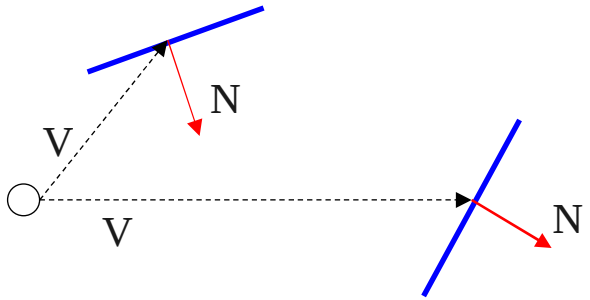


Visible surface determination

- Two approaches
  - Object space
    - Determine geometrical relations between objects and determine which parts of objects are not obscured by others
    - E.g. backface culling, depth-sort
  - Image space
    - Consider each pixel in image
    - Determine nearest object visible on pixel
    - E.g. z-buffer, ray casting

Backface culling

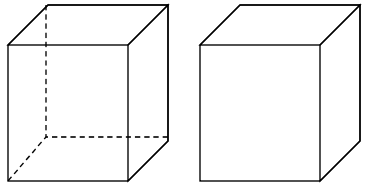
- Remove all polygons oriented away from eye, i.e. from which we only see backface (*backfacing*)



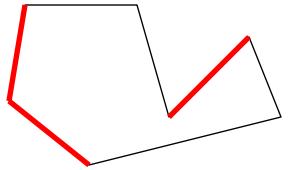
- A polygon is *backfacing* if: V • N > 0
  - N is polygon normal
  - V is vector from eye to (point on) polygon

Backface culling

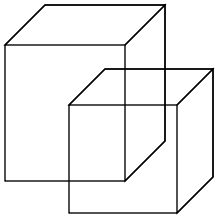
- If 1 convex object, problem solved



- Not a complete solution for concave objects



- Not a complete solution if more than 1 object



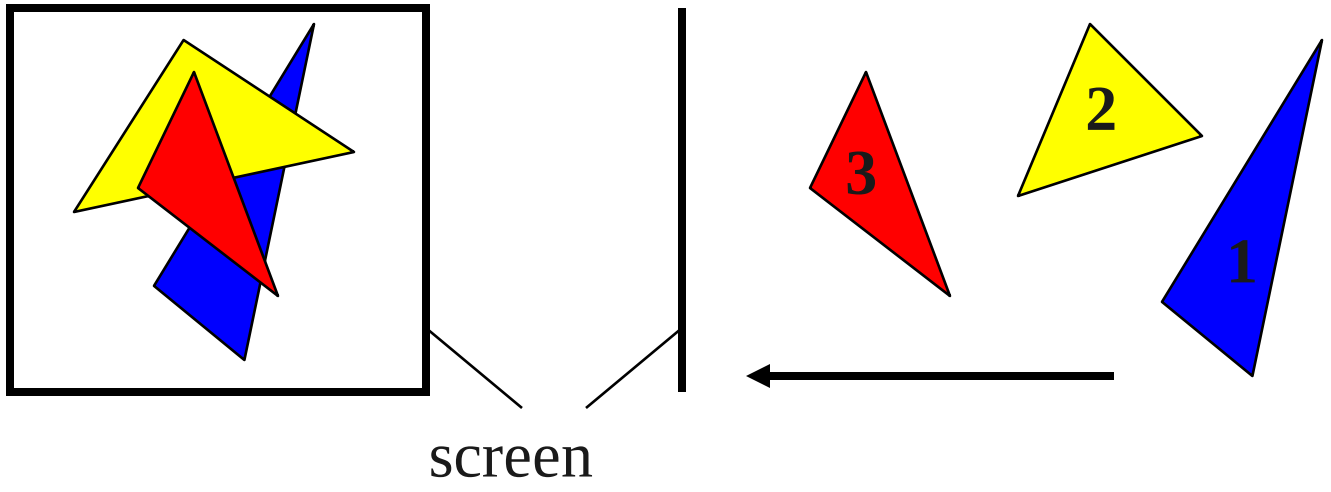
- On average 50% polygons removed
- Usually performed in conjunction with other (complete) methods
- Easy to integrate in hardware

Depth-sort algorithm

- Polygons close to eye hide polygons further away from camera
- So, draw polygons far away first, and then polygons close to eye
- Just like a painter: first draw horizon, then landscape, and finally scene in foreground
  - *painter's algorithm*

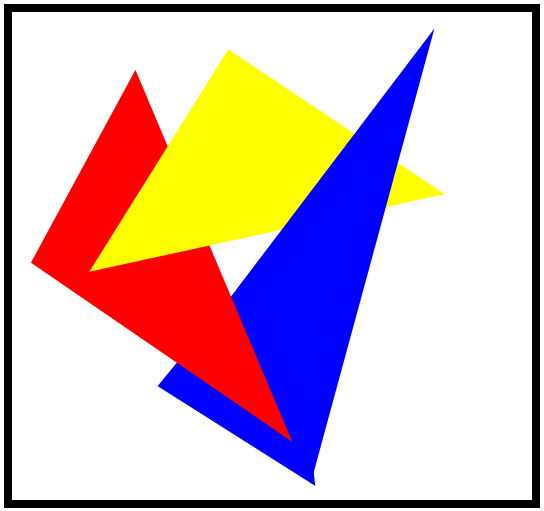
Depth-sort algoritme

- Sort polygons in order of decreasing maximum depth (do from back to front)
- Display them in this order



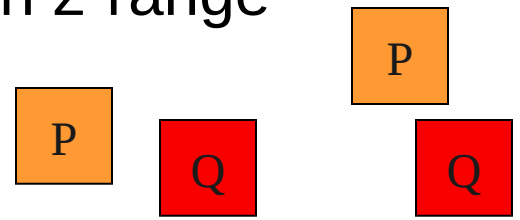
Depth-sort algorithm

- How to solve this one?

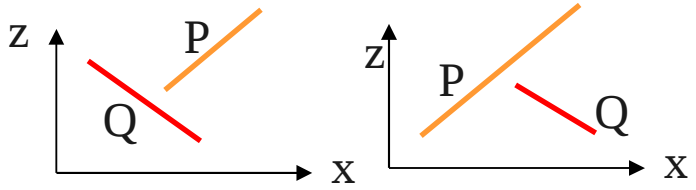


Depth-sort algorithm

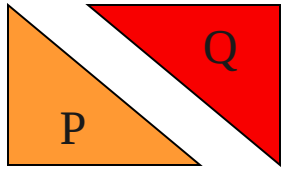
- Sort all polygons based on maximum z-value
- If two polygons have (partly) common z-range
  - Test if x- or y-range are different



  - Test if polygon P completely behind plane of polygon Q (or vice versa)



  - Test if projections of polygons have no overlap



  - If none of these tests pass, then one of the polygons needs to be subdivided

Depth-sort algorithm

- All polygons must be available at the same time
- Sorting and subdivision of polygons is difficult and expensive
- Slow (#polygons^2)
  - Not feasible for large scenes
  - No hardware support

Z-buffer algorithm

- Z-buffer (depth-buffer) is an array with the same size as the framebuffer
- For each pixel the z-buffer contains depth value (z-value) of the polygon closest to the eye

Z-buffer algorithm

Initialization

For all pixels (*x*,*y*) **do**

framebuffer(*x y*) := “background color”

zbuffer(*x*,*y*) := “maximum depth”

Algorithm

For each polygon *P* do

For each pixel (*x*,*y*) in projection of *P* do

Compute depth of *P* for this pixel

If depth < zbuffer(*x*,*y*) **then**

framebuffer(*x, y*) := color of *P* at *(x,y)*

zbuffer(*x, y*) := depth

Z-buffer example

1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

empty z-buffer

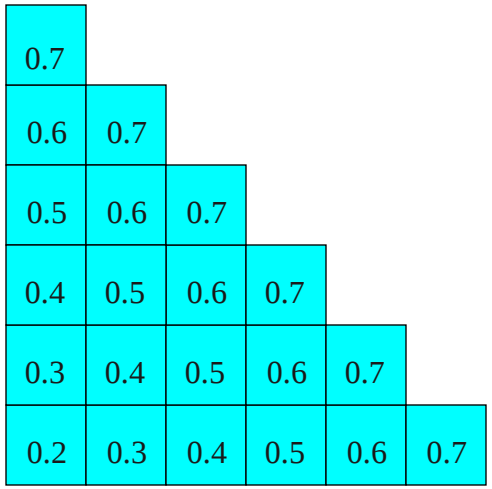
0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	
0.5	0.5	0.5	0.5	0.5		
0.5	0.5	0.5	0.5			
0.5	0.5	0.5				
0.5	0.5					
0.5						

polygon

Z-buffer example

0.5	0.5	0.5	0.5	0.5	0.5	0.5	1.0
0.5	0.5	0.5	0.5	0.5	0.5	1.0	1.0
0.5	0.5	0.5	0.5	0.5	1.0	1.0	1.0
0.5	0.5	0.5	0.5	1.0	1.0	1.0	1.0
0.5	0.5	0.5	1.0	1.0	1.0	1.0	1.0
0.5	0.5	1.0	1.0	1.0	1.0	1.0	1.0
0.5	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

z-buffer



polygon

Z-buffer example

0.5	0.5	0.5	0.5	0.5	0.5	0.5	1.0
0.5	0.5	0.5	0.5	0.5	0.5	1.0	1.0
0.5	0.5	0.5	0.5	0.5	1.0	1.0	1.0
0.5	0.5	0.5	0.5	1.0	1.0	1.0	1.0
0.5	0.4	0.5	0.6	0.7	1.0	1.0	1.0
0.5	0.3	0.4	0.5	0.6	0.7	1.0	1.0
0.5	0.2	0.3	0.4	0.5	0.6	0.7	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Z-buffer algorithm

- Fast and simple
  - No sorting, polygons can be drawn in any order
  - No object – object comparisons
  - Hardware support
- Requires lots of memory : 24-32 bits per pixel
- Commonly implemented in 3D graphics cards

Which algorithm?

- Depends on (complexity of) scene, required visual effects, and availability of hardware

Which algorithm?

- Backface culling
  - Always useful to reduce number of polygons
  - In combination with other algorithms
- Depth sort
  - Software renderer
  - Slow: O(#polygons * #polygons)
  - For simple scenes, with not many objects
- Z-buffer
  - Hardware commonly available
  - Fast O(#polygons)
  - For complex scenes with many polygons

Graphics pipeline