# 3D Rendering Pipeline

Fundamentals of Game Development

Instructor : Dr. Behrouz Minaei (b_minaei@iust.ac.ir)
Teaching Assistant : Morteza Rajabi (mtz.rajabi@gmail.com)

## Goals

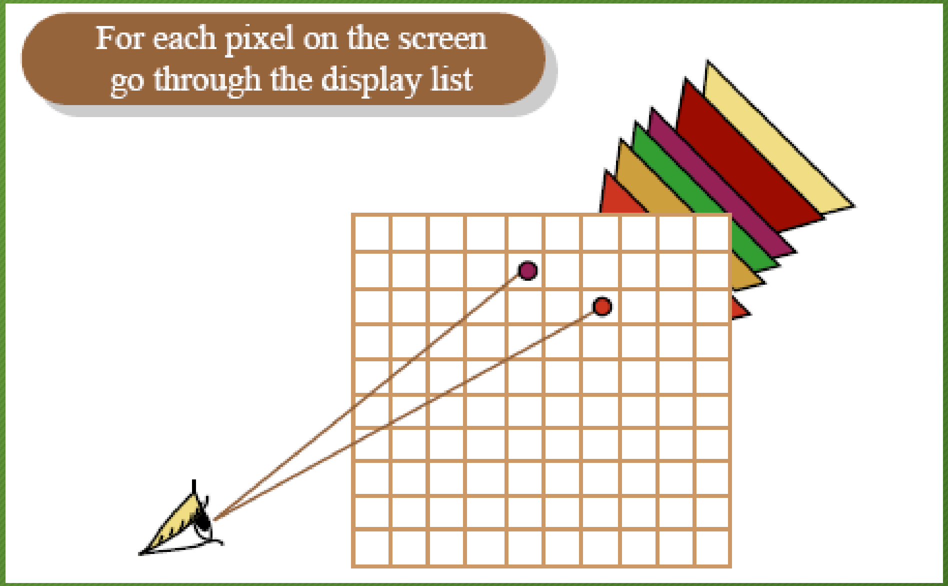
- Understand the difference between inverse-mapping and forward-mapping approaches to computer graphics rendering
- Be familiar with the graphics pipeline
  - From transformation perspective
  - From operation perspective

## Approaches to graphics rendering

- **Ray-tracing approach**
  - Inverse-mapping approach: starts from pixels
  - A ray is traced from the camera through each pixel
  - Takes into account reflection, refraction, and diffraction in a multi-resolution fashion
  - High quality graphics but computationally expensive
    - Not for real-time applications
- **Pipeline approach**
  - Forward-mapping approach
  - Used by OpenGL and DirectX
  - State-based approach:
    - Input is 2D or 3D data
    - Output is frame buffer
    - Modify state to modify functionality
  - For real-time and interactive applications, especially games

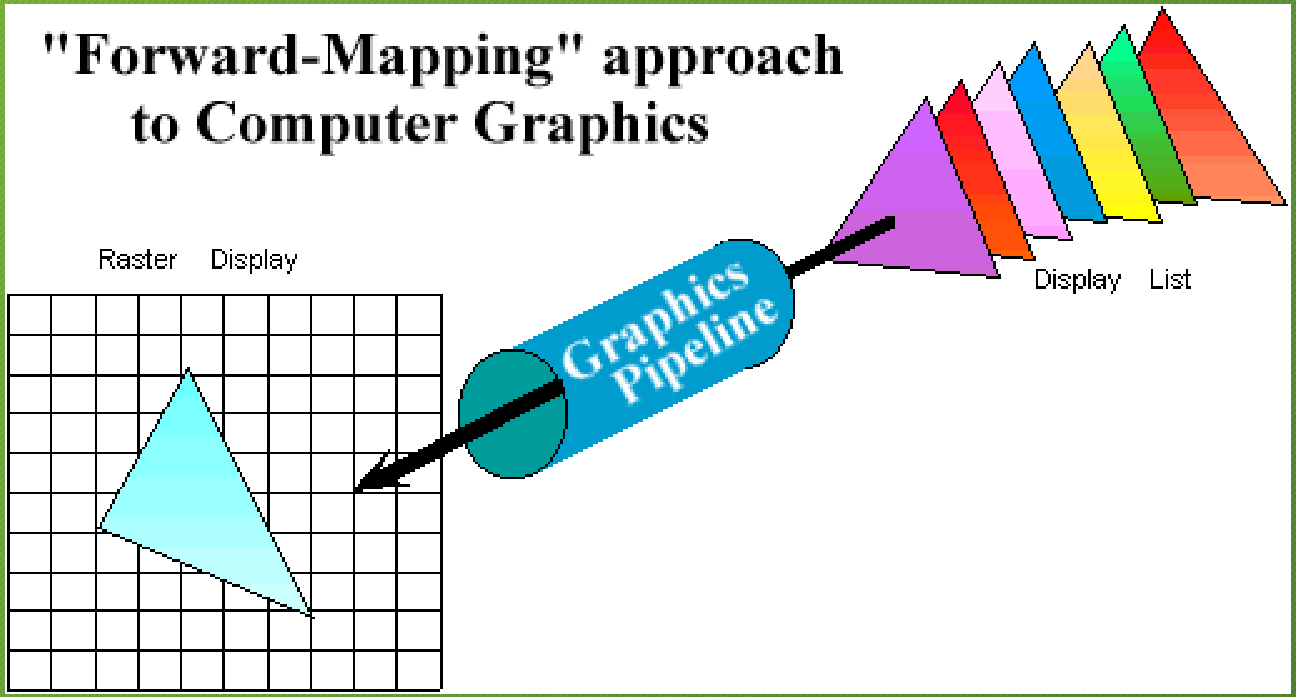
## Ray-tracing – Inverse mapping

1. For every pixel, construct a ray from the eye
2. for every object in the scene, intersect ray with object
3. find closest intersection with the ray
4. compute normal at point of intersection
5. compute color for pixel
6. shoot secondary rays

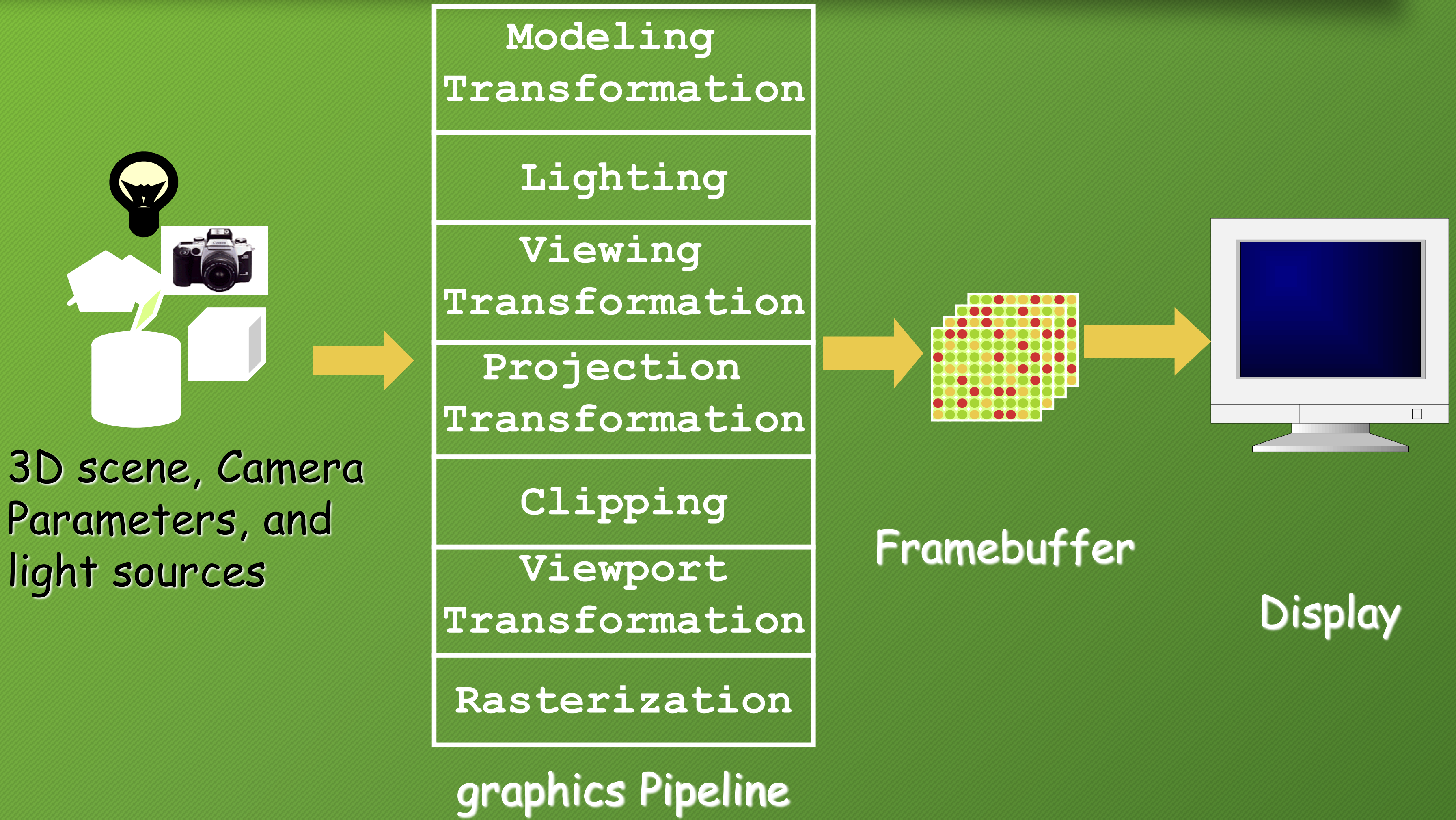


## Pipeline – Forward mapping

- Start from the geometric primitives to find the values of the pixels



## The general view (Transformations)



# Input and output of the graphics pipeline

- Input:
  - Geometric model
    - Objects
    - Light sources geometry and transformations
  - Lighting model
    - Description of light and object properties
  - Camera model
    - Eye position, viewing volume
  - Viewport model
    - Pixel grid onto which the view window is mapped
- Output:
  - Colors suitable for framebuffer display

## Graphics pipeline

- What is it?

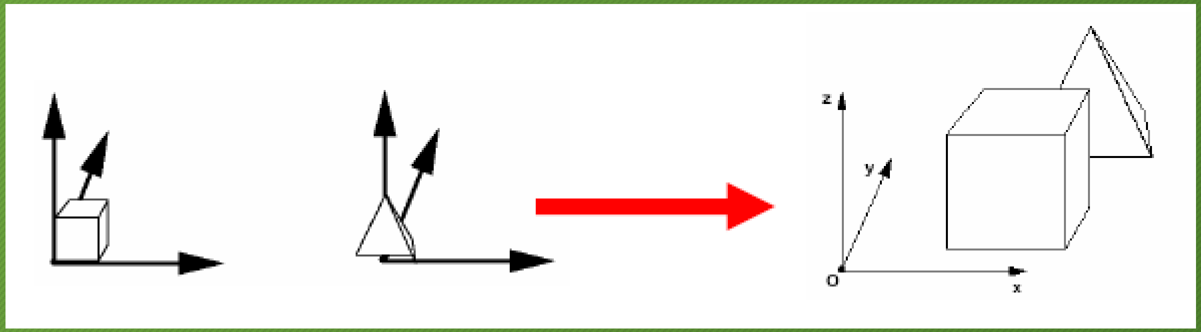
The nature of the processing steps to display a computer graphic and the order in which they must occur.

- Primitives are processed in a series of stages
- Each stage forwards its result on to the next stage
- The pipeline can be drawn and implemented in different ways
- Some stages may be in hardware, others in software
- Optimizations and additional programmability are available at some stages
- Two ways of viewing the pipeline:
  - Transformation perspective
  - Operation perspective

# Modeling transformation

- Modeling Transformation
- Lighting
- Viewing Transformation
- Projection Transformation
- Clipping
- Viewport Transformation
- Rasterization

- 3D models defined in their own coordinate system (object space)
- Modeling transforms orient the models within a common coordinate frame (world space)



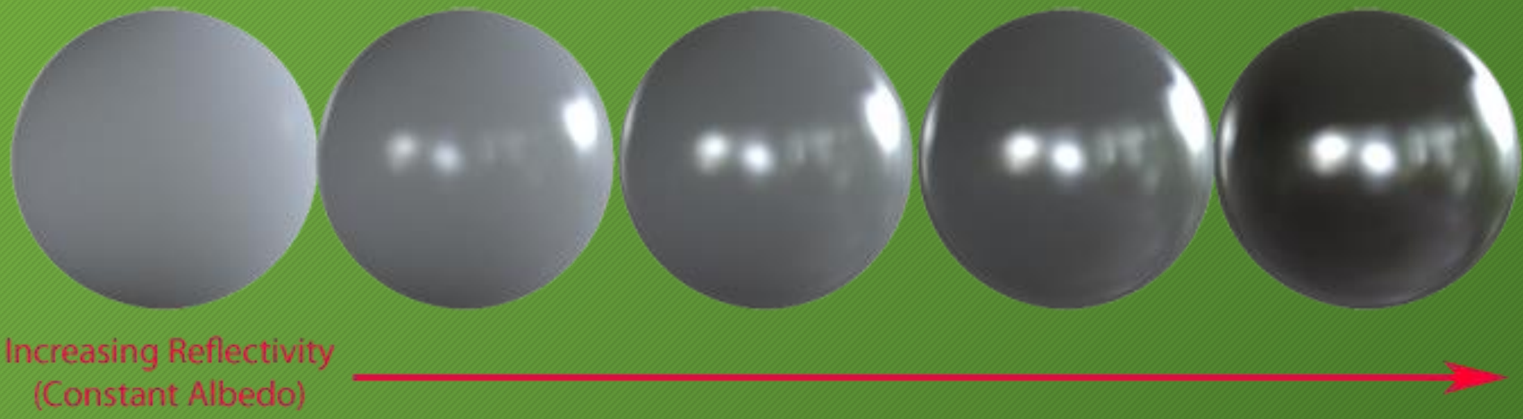
Object space

World space

# Lighting (shading)

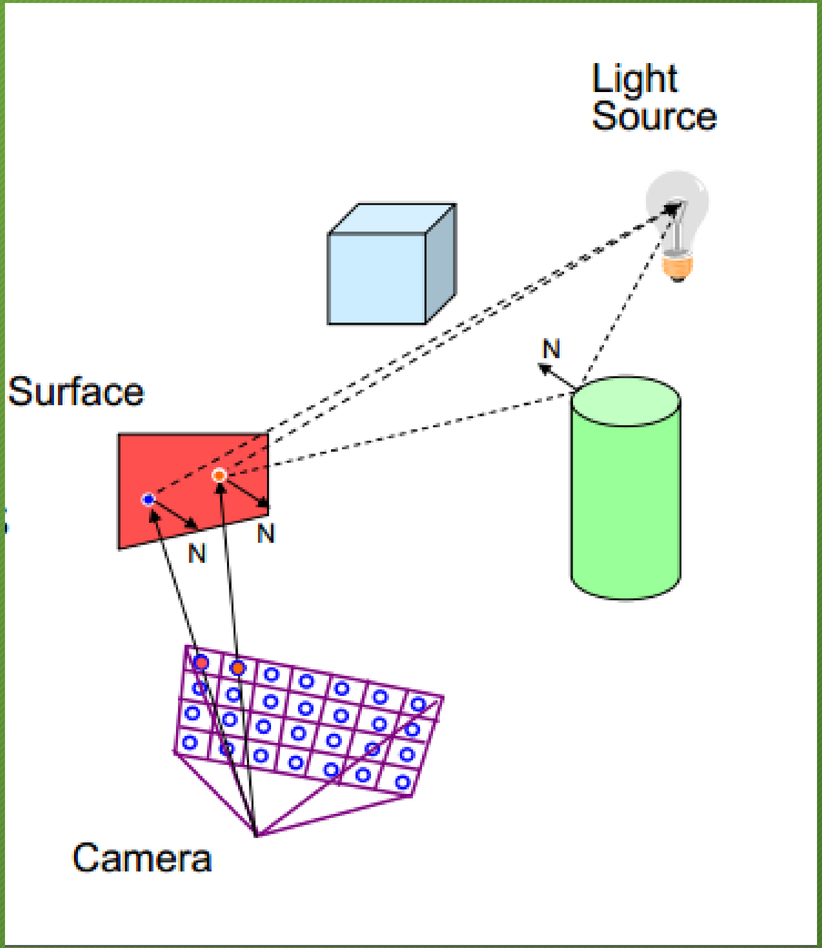
| |
|---|
| Modeling Transformation |
| Lighting |
| Viewing Transformation |
| Projection Transformation |
| Clipping |
| Viewport Transformation |
| Rasterization |

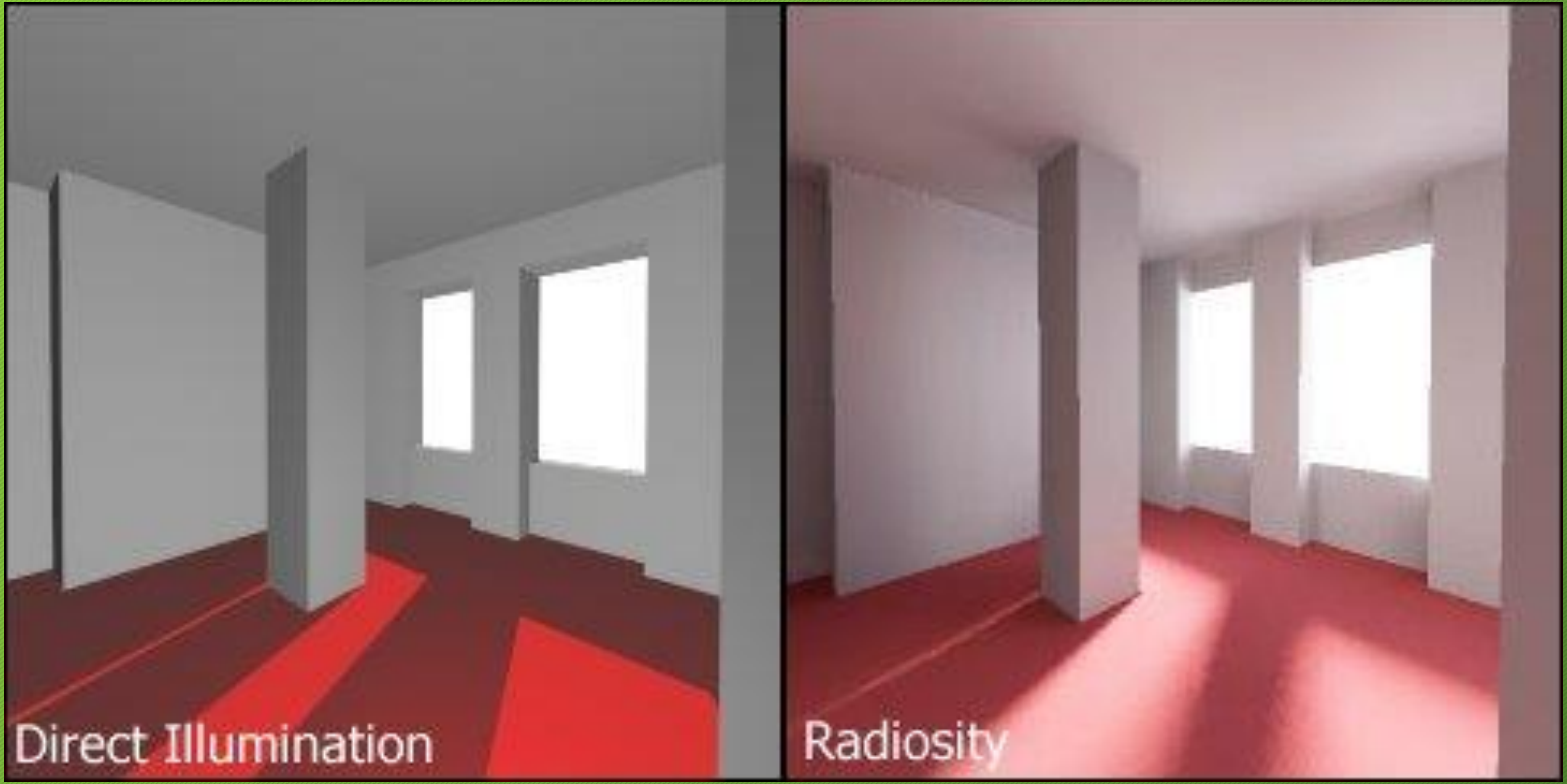
- Vertices lit (shaded) according to material properties, surface properties (normal) and light sources
- Local lighting model (Diffuse, Ambient, Phong, etc.)

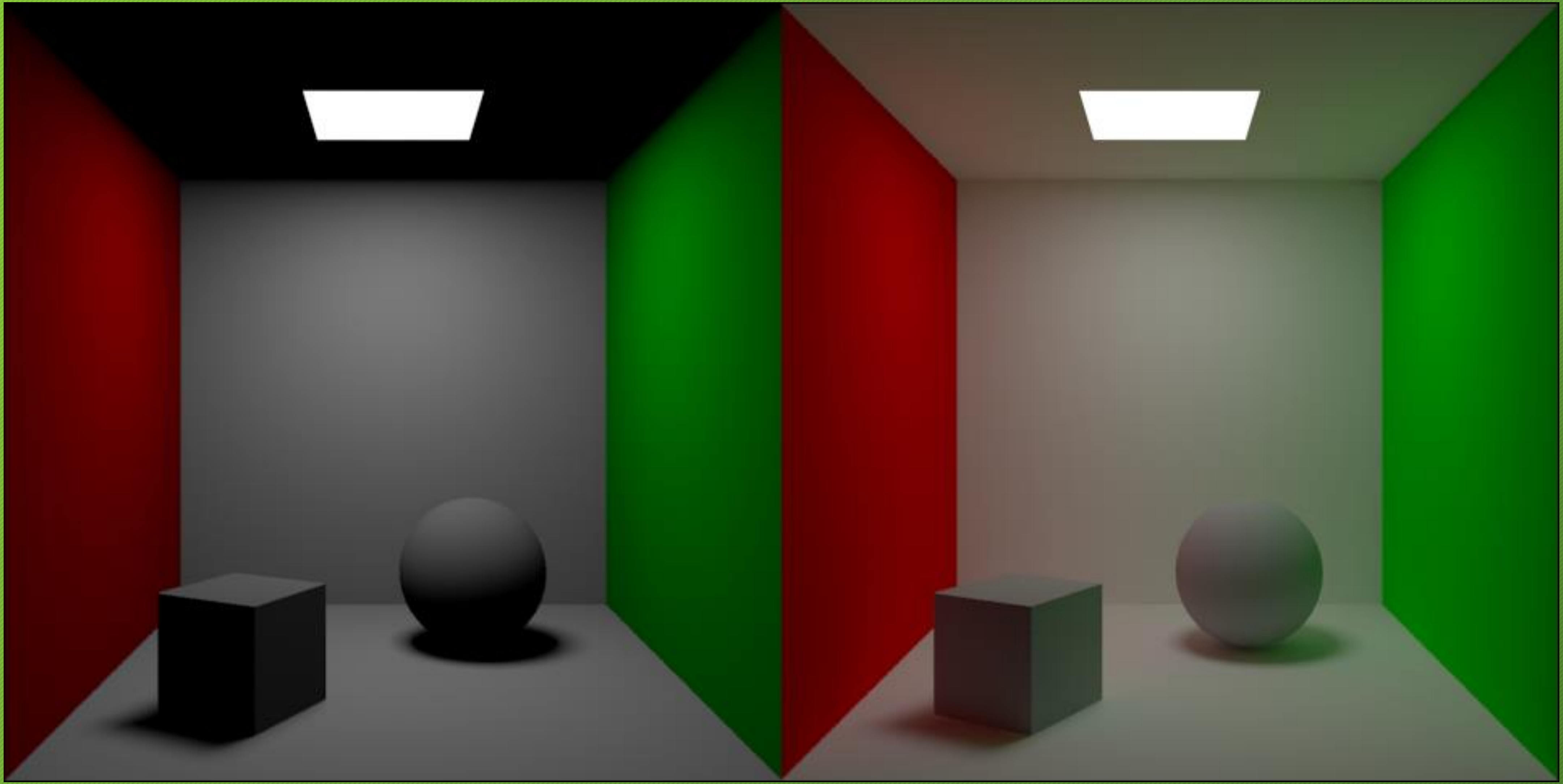


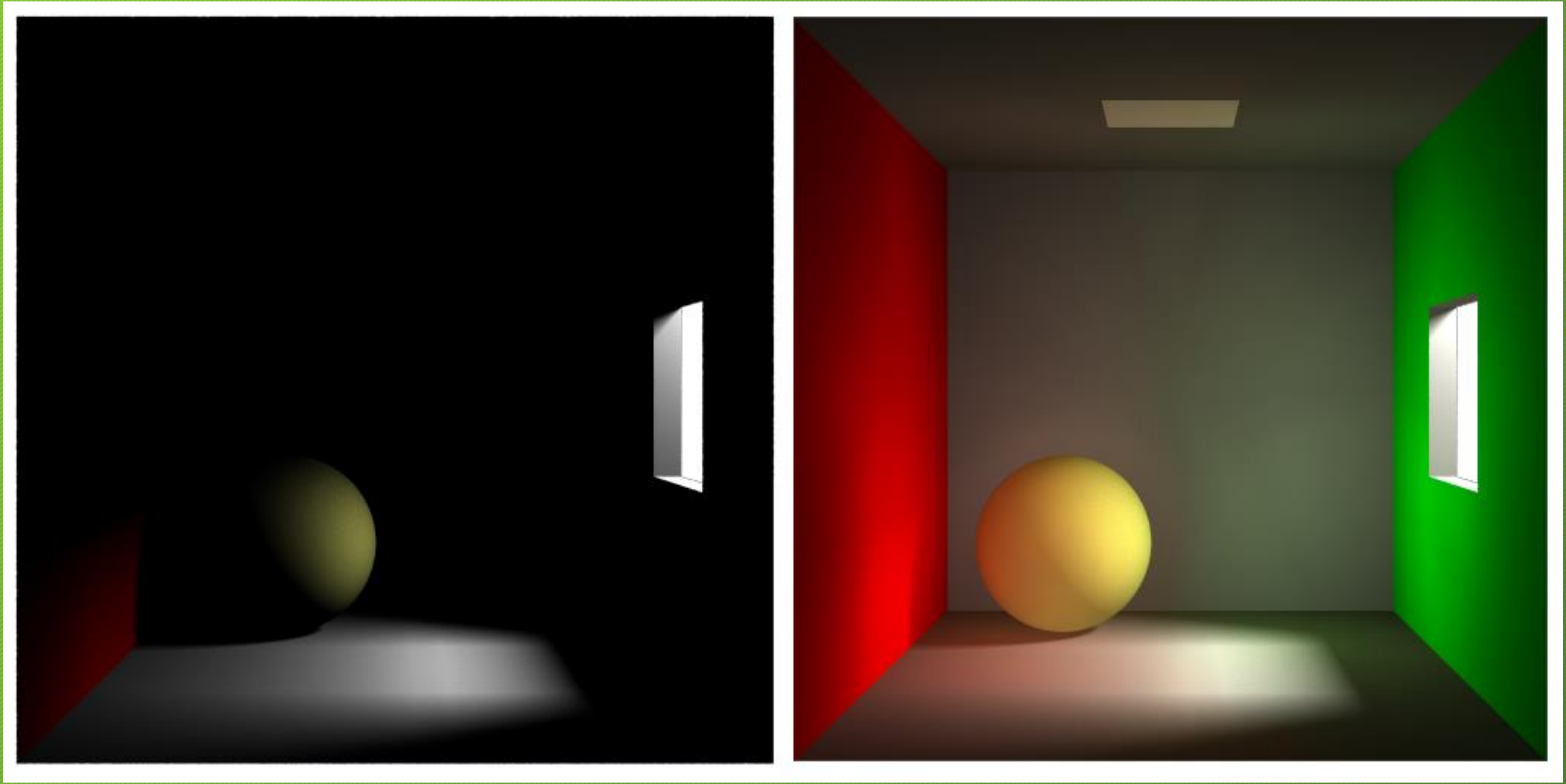
## Lighting Simulation

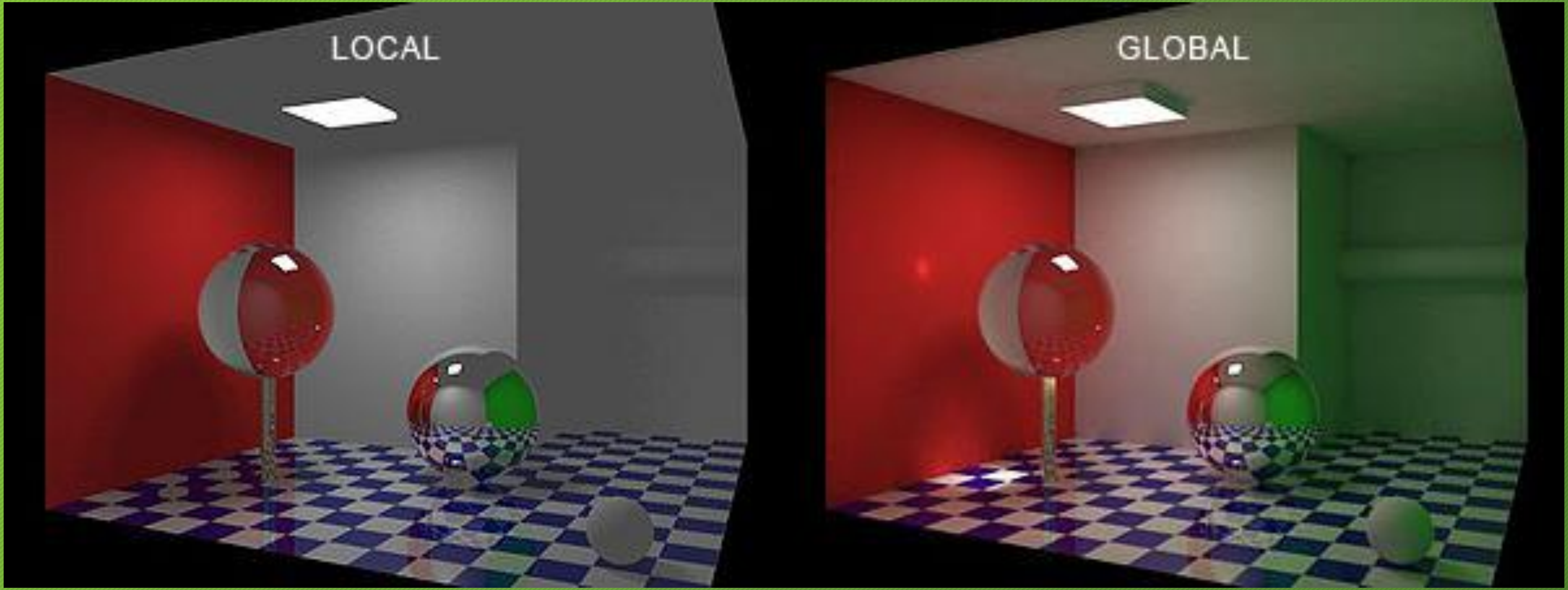
- Direct illumination
  - Ray casting
  - Polygon shading
- Global illumination
  - Ray tracing
  - Monte Carlo methods
  - Radiosity methods




Direct Illumination
Radiosity



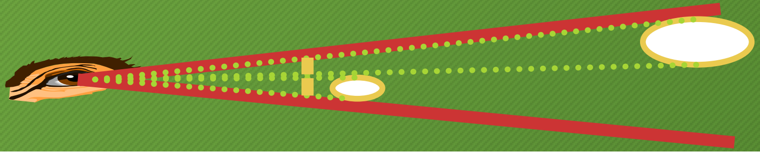


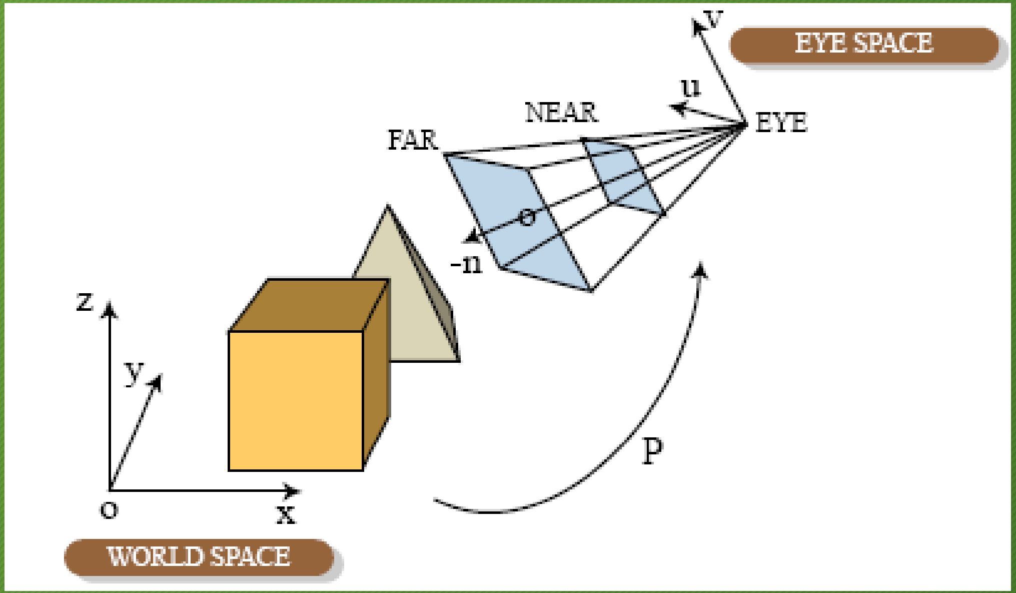

LOCAL
GLOBAL

## Viewing transformation

- Modeling Transformation
- Lighting
- **Viewing Transformation**
- Projection Transformation
- Clipping
- Viewport Transformation
- Rasterization

- It maps world space to eye (camera) space
- Viewing position is transformed to origin and viewing direction is oriented along some axis (usually z)

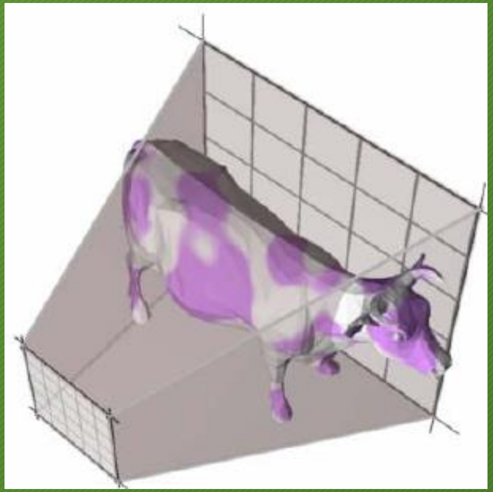


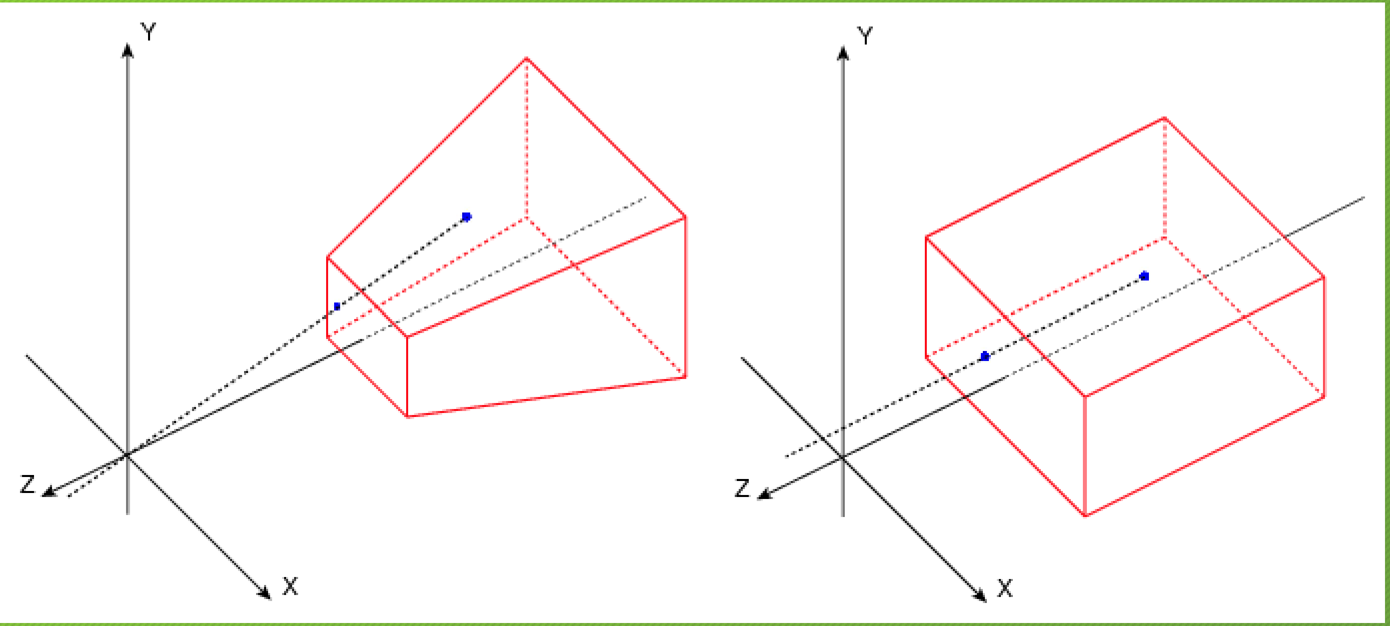


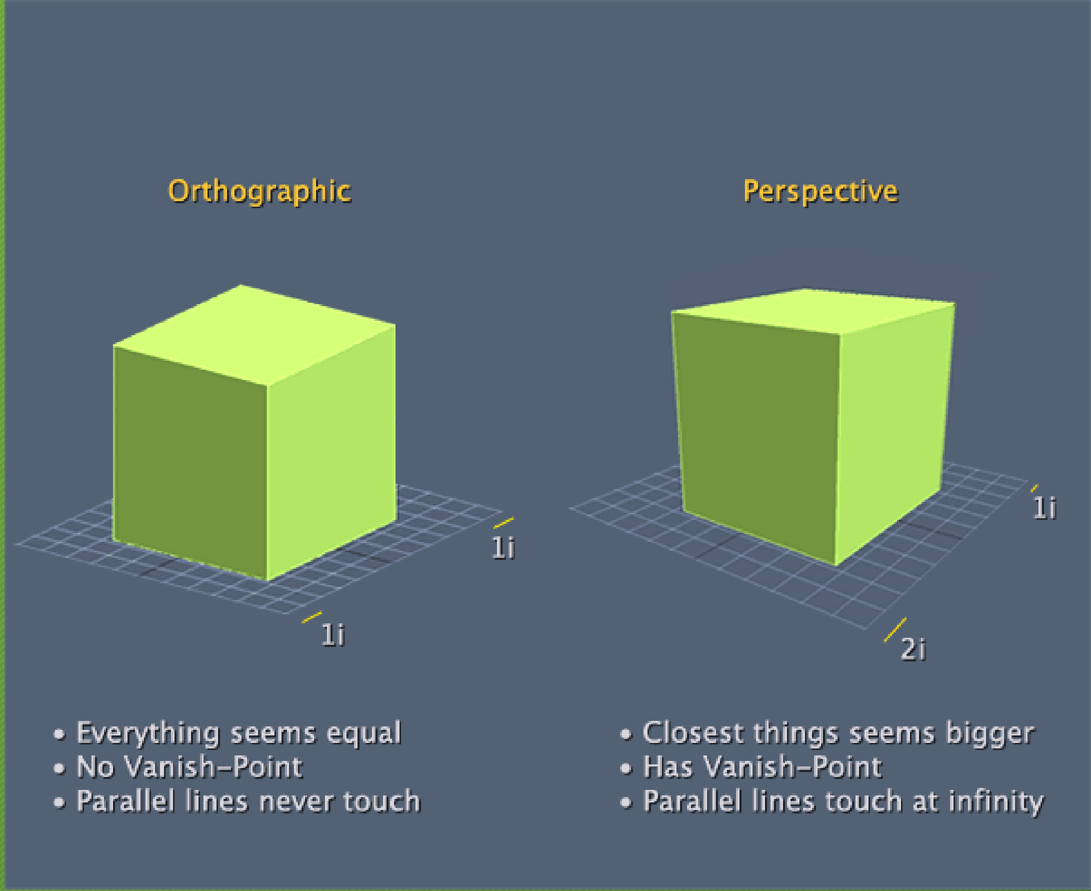
# Projection transformation (Perspective/Orthogonal)

- Modeling Transformation
- Lighting
- Viewing Transformation
- Projection Transformation
- Clipping
- Viewport Transformation
- Rasterization

- Specify the *view volume* that will ultimately be visible to the camera
- Two clipping planes are used: near plane and far plane
- Usually *perspective* or *orthogonal*

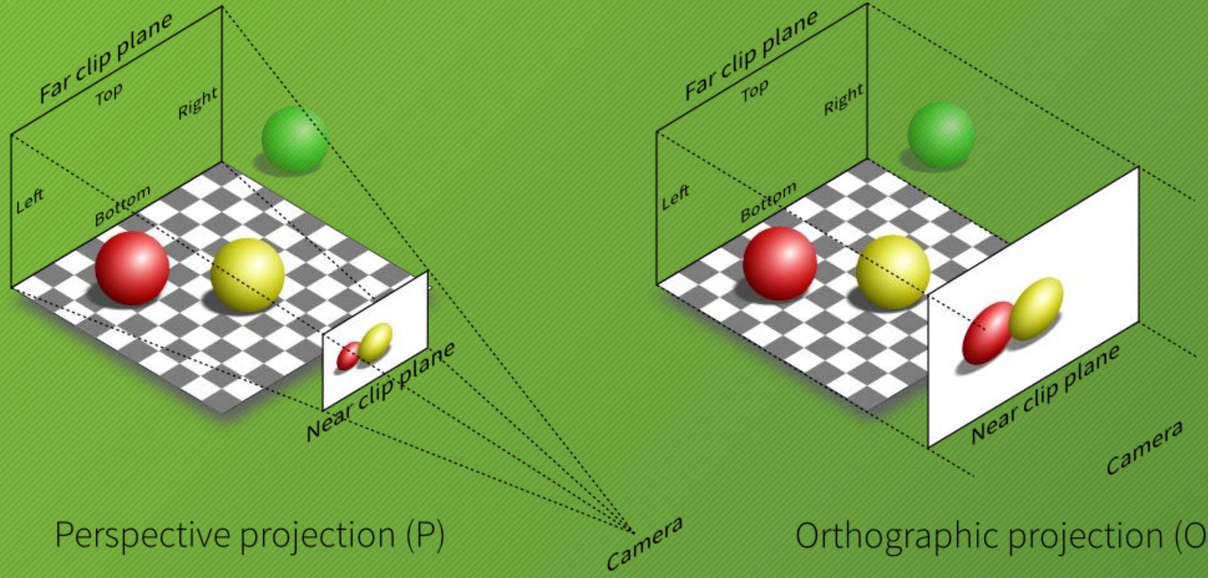






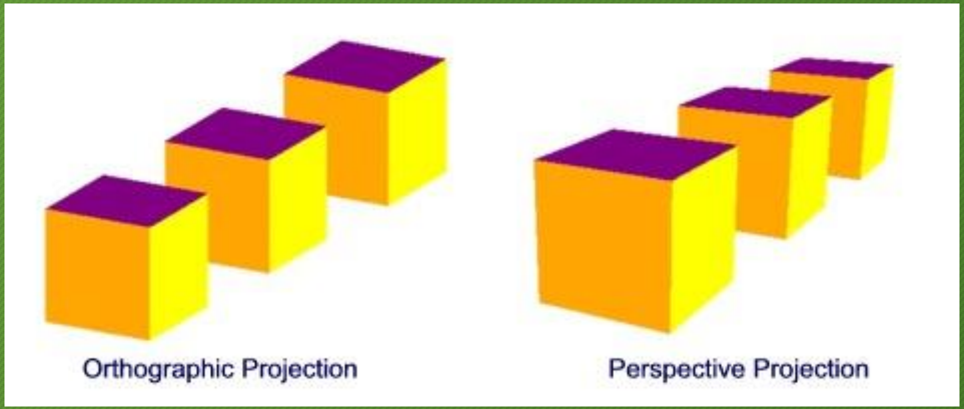
- Everything seems equal
- No Vanish-Point
- Parallel lines never touch

- Closest things seems bigger
- Has Vanish-Point
- Parallel lines touch at infinity



Perspective projection (P)

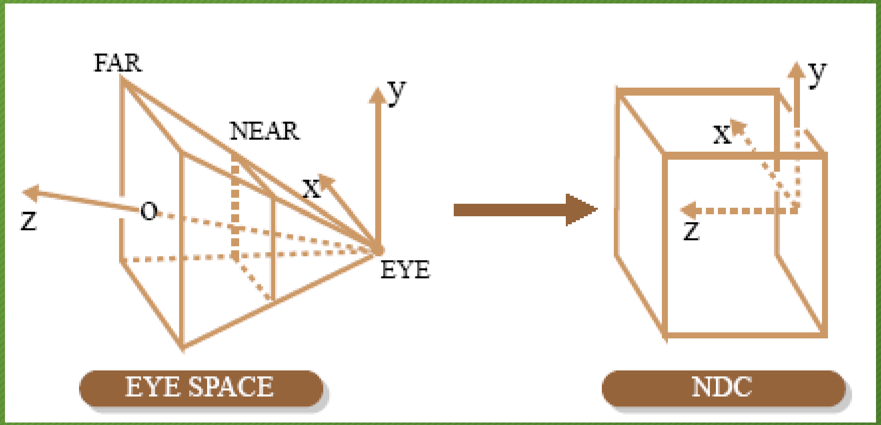
Orthographic projection (O)



# Clipping

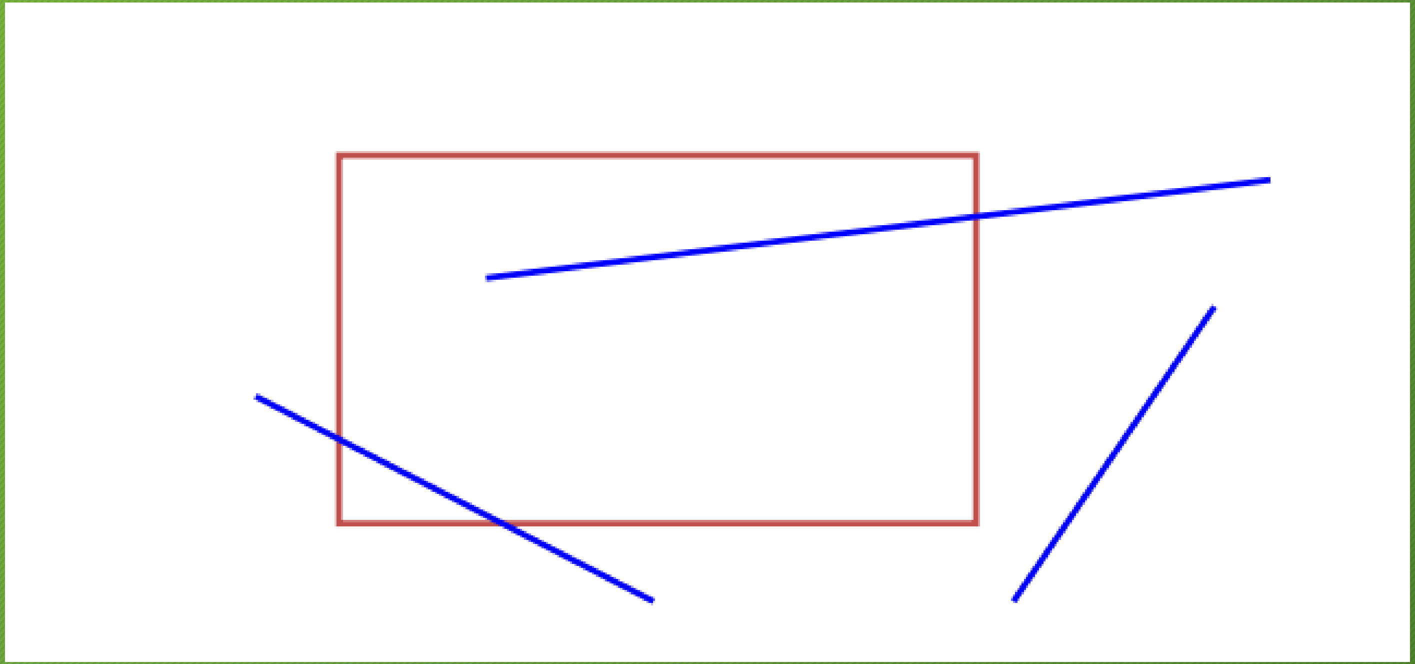
- Modeling Transformation
- Lighting
- Viewing Transformation
- Projection Transformation
- Clipping
- Viewport Transformation
- Rasterization

- The view volume is transformed into standard cube that extends from -1 to 1 to produce Normalized Device Coordinates.
- Portions of the object outside the NDC cube are removed (clipped)



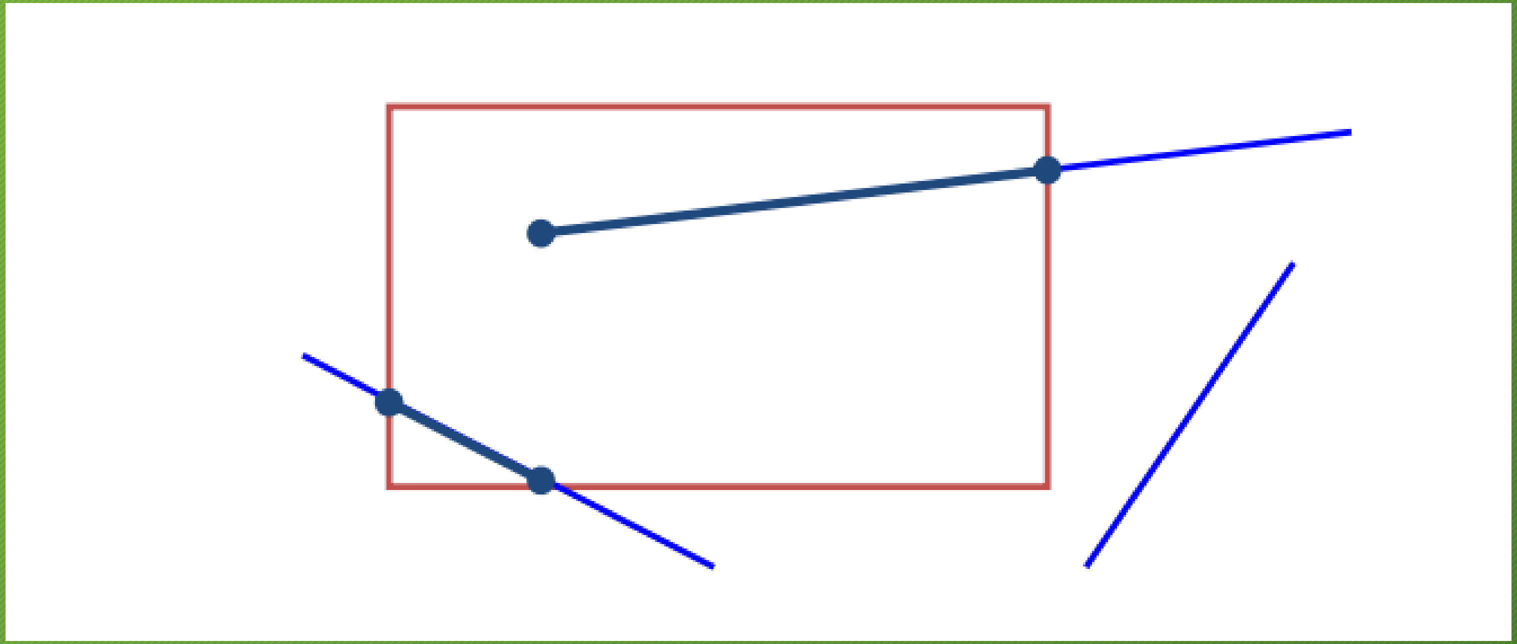
# Why clip?

- We don’t want to waste time rendering objects that are outside the viewing window (or clipping window)
- Bad idea to rasterize outside of framebuffer bounds



## What is clipping?

- Analytically calculating the portions of primitives within the view window

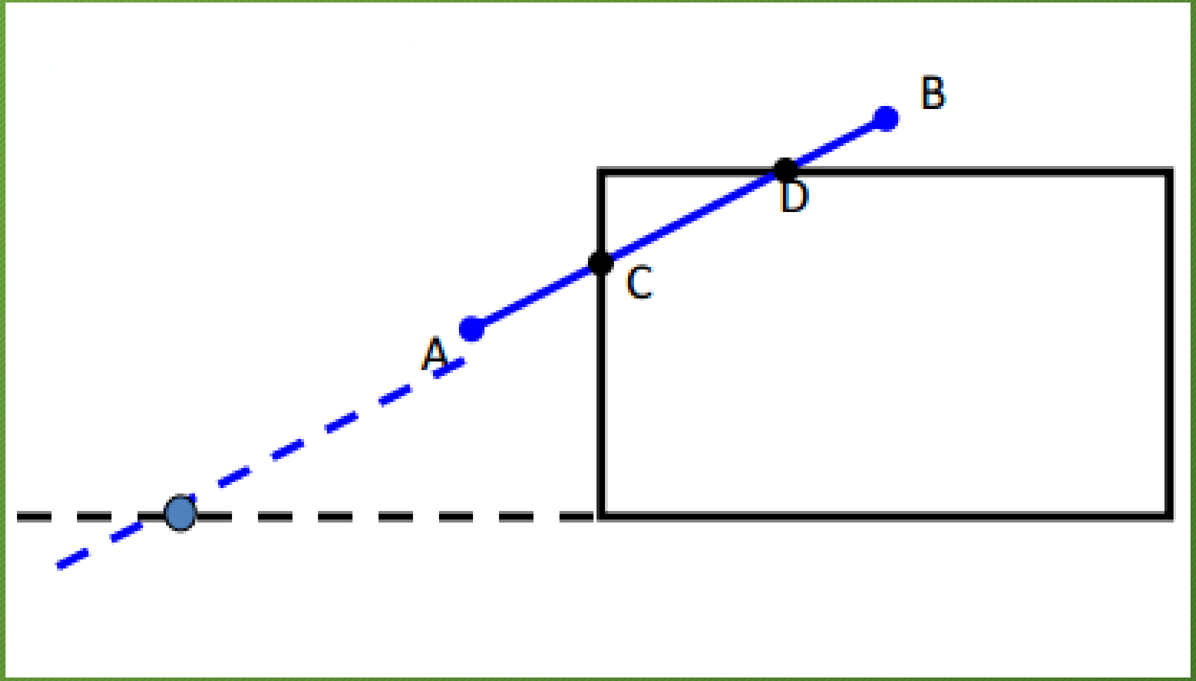


# Clipping

- The native approach to clipping lines:

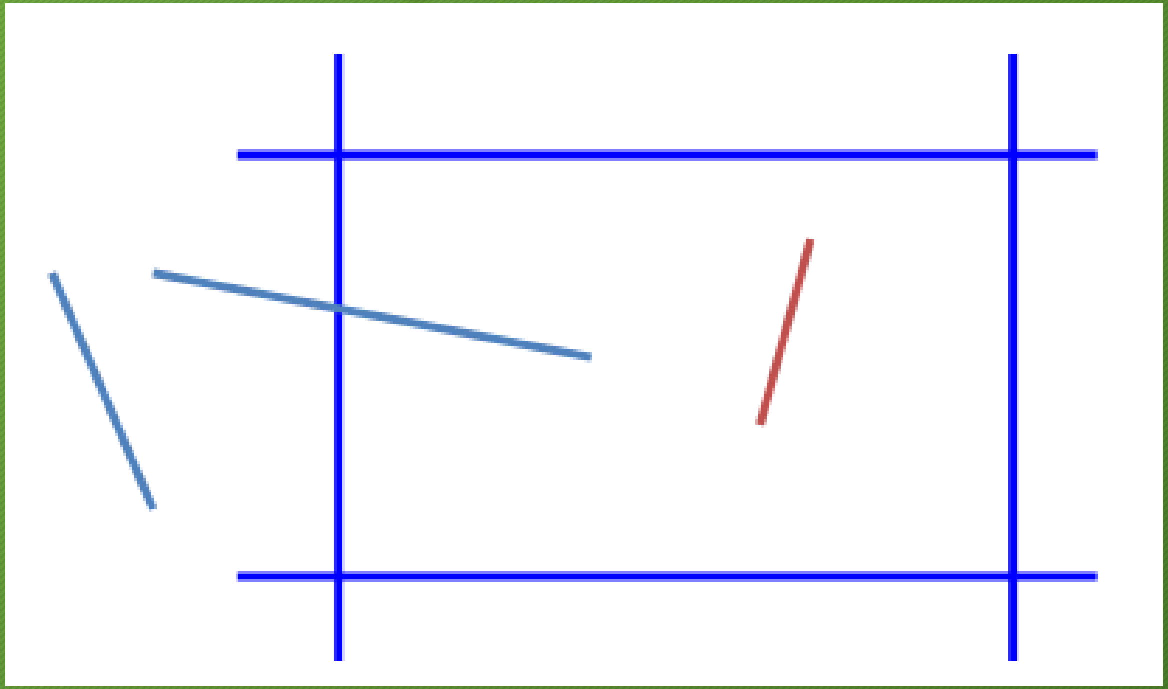
```
for each line segment
      for each edge of view_window
      find intersection point
      pick “nearest” point
if anything is left, draw it
```

- What do we mean by “nearest”?
- How can we optimize this?



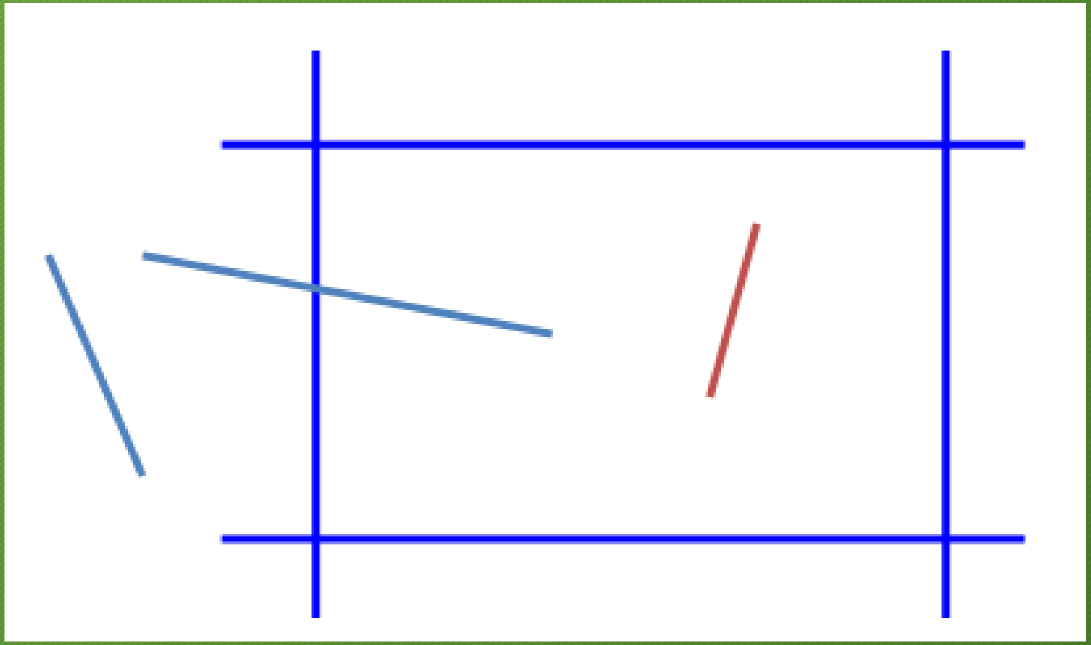
## Trivial Accepts

- Big optimization: trivial accept/rejects
- How can we quickly determine whether a line segment is entirely inside the view window?
- A: test both endpoints.



# Trivial Rejects

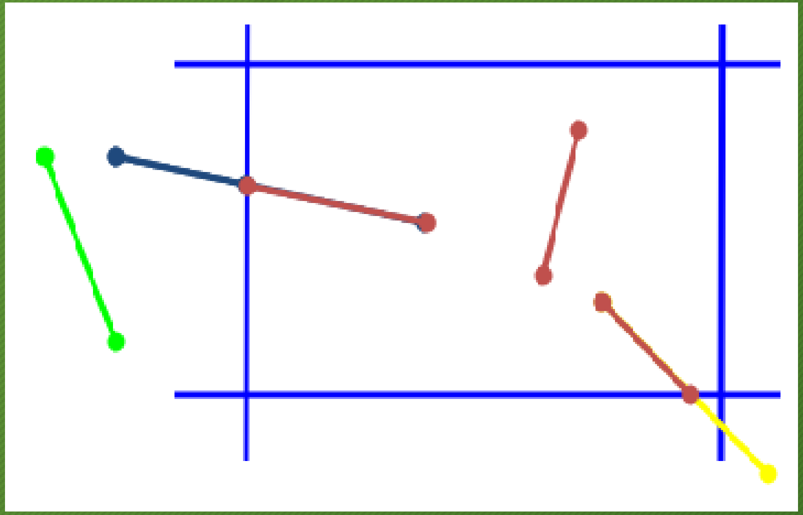
- How can we know a line is outside view window?
- A: if both endpoints on wrong side of same edge, can trivially reject line



## Clipping Lines To Viewport

- Combining trivial accepts/rejects
- Trivially accept lines with both endpoints inside all edges of the view
  window
- Trivially reject lines with both endpoints outside the same edge of the
  view window
- Otherwise,

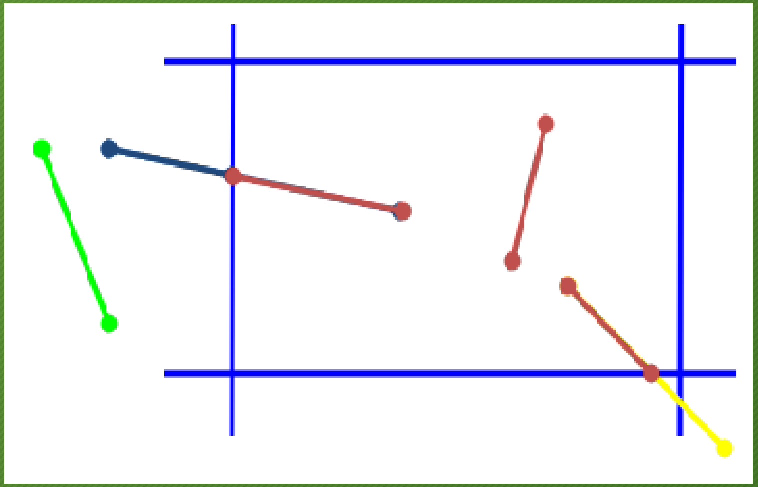
reduce to trivial cases by splitting into two segments



## Clipping Lines To Viewport

- Combining trivial accepts/rejects
- Trivially accept lines with both endpoints inside all edges of the view
  window
- Trivially reject lines with both endpoints outside the same edge of the
  view window
- Otherwise,

reduce to trivial cases by splitting into two segments



## Others Line clipping algorithms

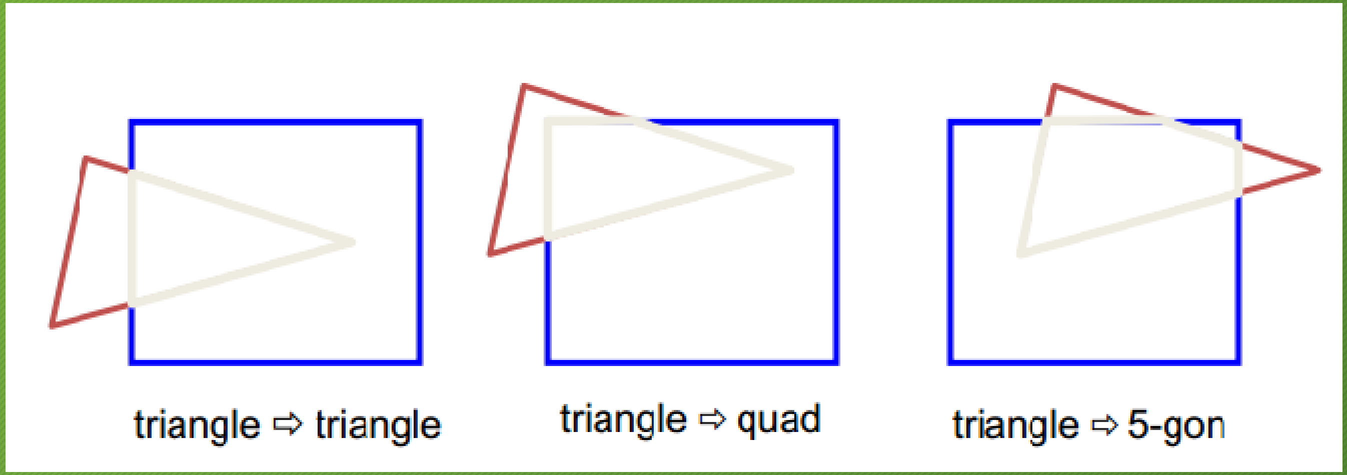
- Cohen-Sutherland
- Liang-Barsky
- Cyrus-Beck
- Nicholl-Lee-Nicholl
- Fast clipping
- $O(\lg N)$ algorithm
- Skala
- What is the best? What is the differences?

## Clipping Polygons

- Clipping polygons is more complex than clipping the individual lines
- Input: polygon
- Output: original polygon, new polygon, or nothing
- The biggest optimizer we had was trivial accept or reject…
- When can we trivially accept/reject a polygon as opposed to the line segments that make up the polygon?

## Why Is Clipping Hard?

- What happens to a triangle during clipping?
- Possible outcomes:

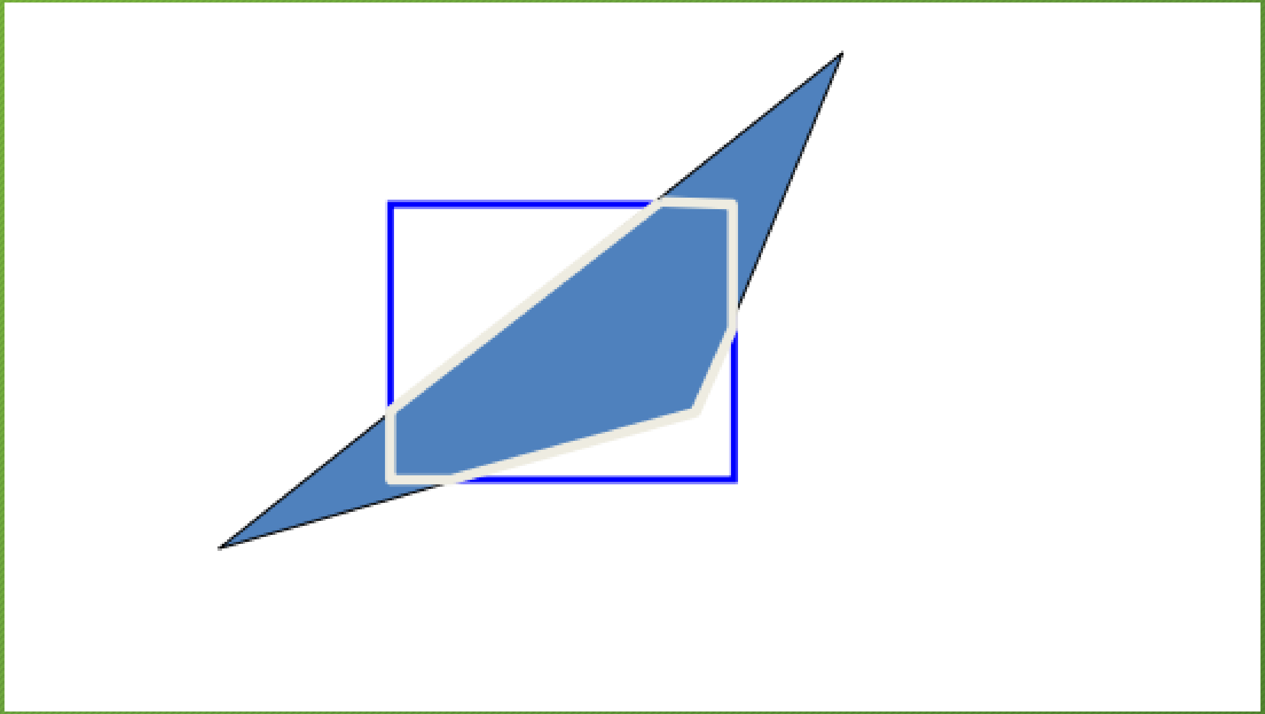


- ***How many sides can a clipped triangle have?***

## How many sides?

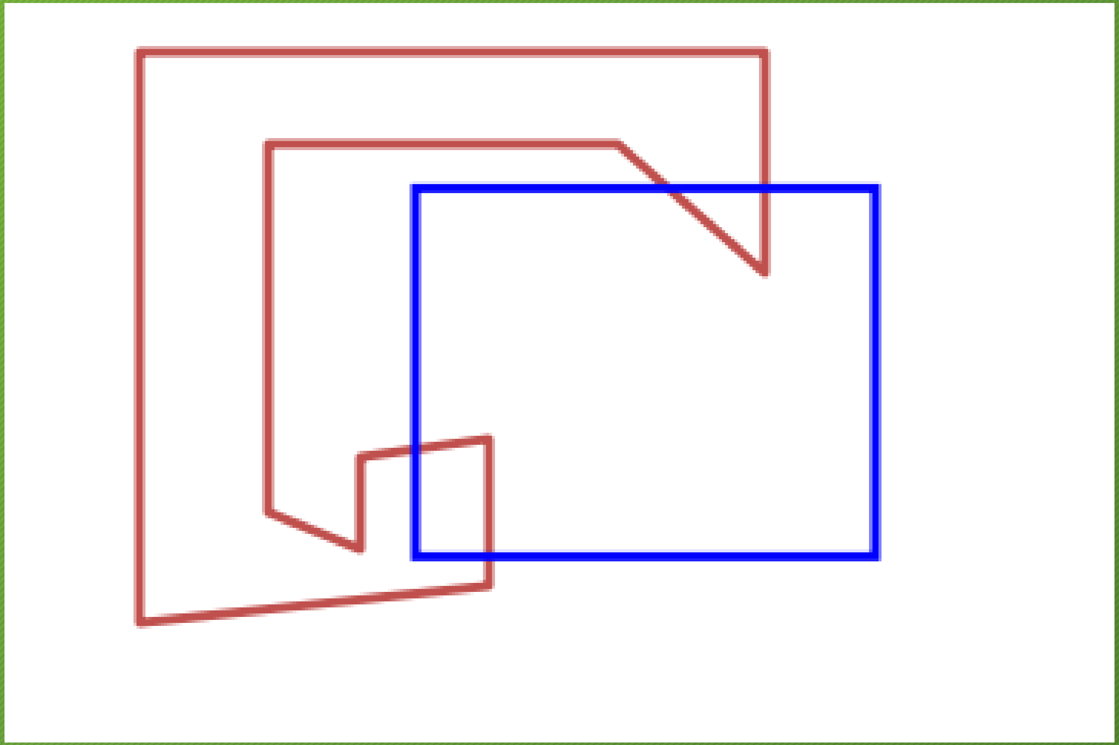
31

- Seven……..



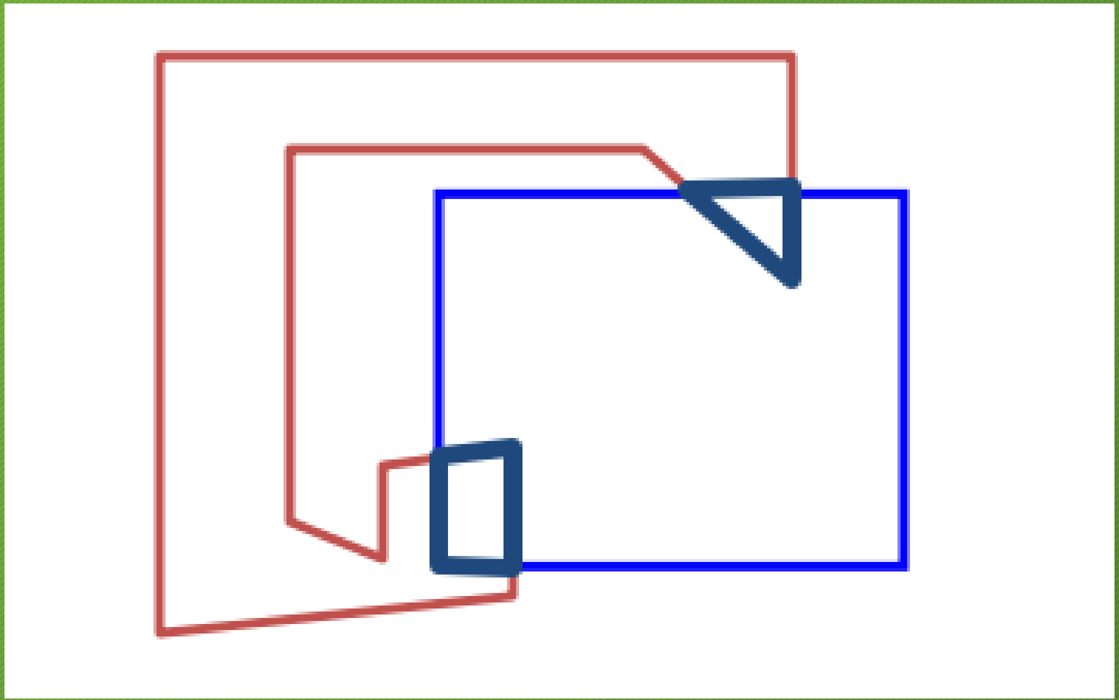
## Why Is Clipping Hard?

- A really tough case:



## Why Is Clipping Hard?

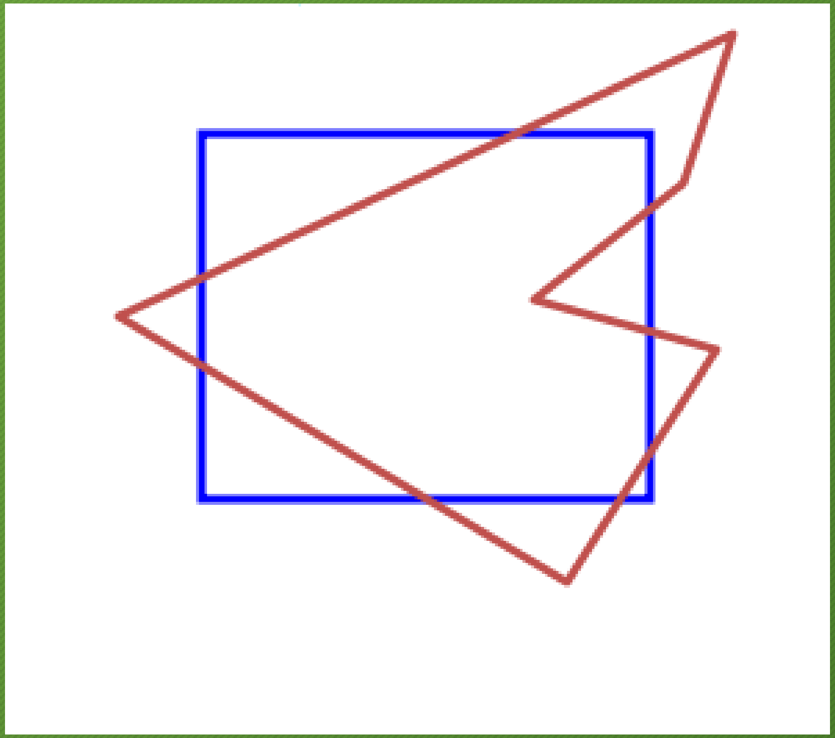
- A really tough case:

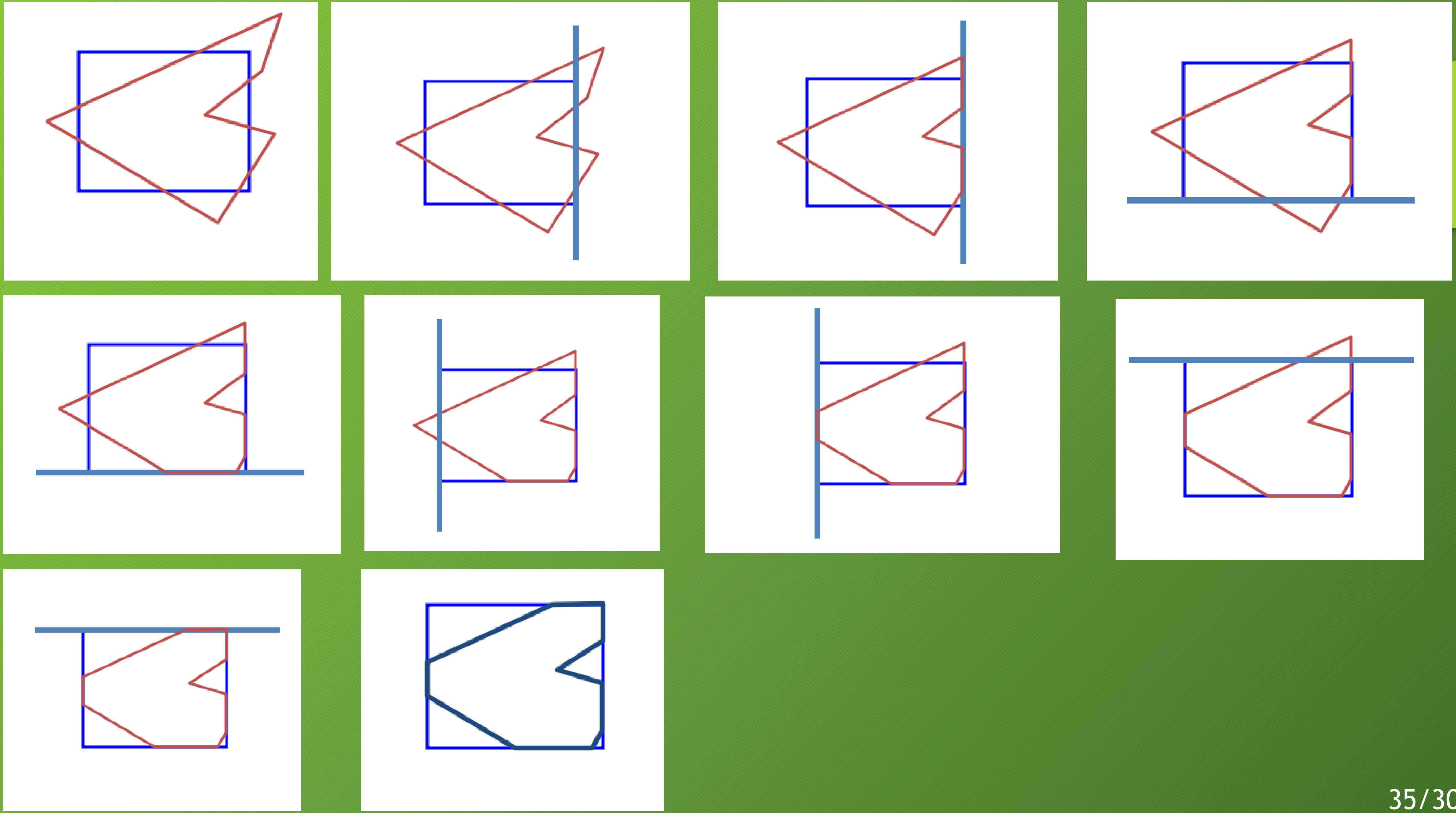


**concave polygon => multiple Polygon**

## Sutherland-Hodgman Clipping

- Basic idea:
- Consider each edge of the view window individually
-  Clip the polygon against the view window edge's
  equation





## Sutherland-Hodgman Clipping

- Input/output for algorithm:
  - Input: list of polygon vertices in order
  - Output: list of clipped polygon vertices consisting of old vertices (maybe) and new vertices (maybe)

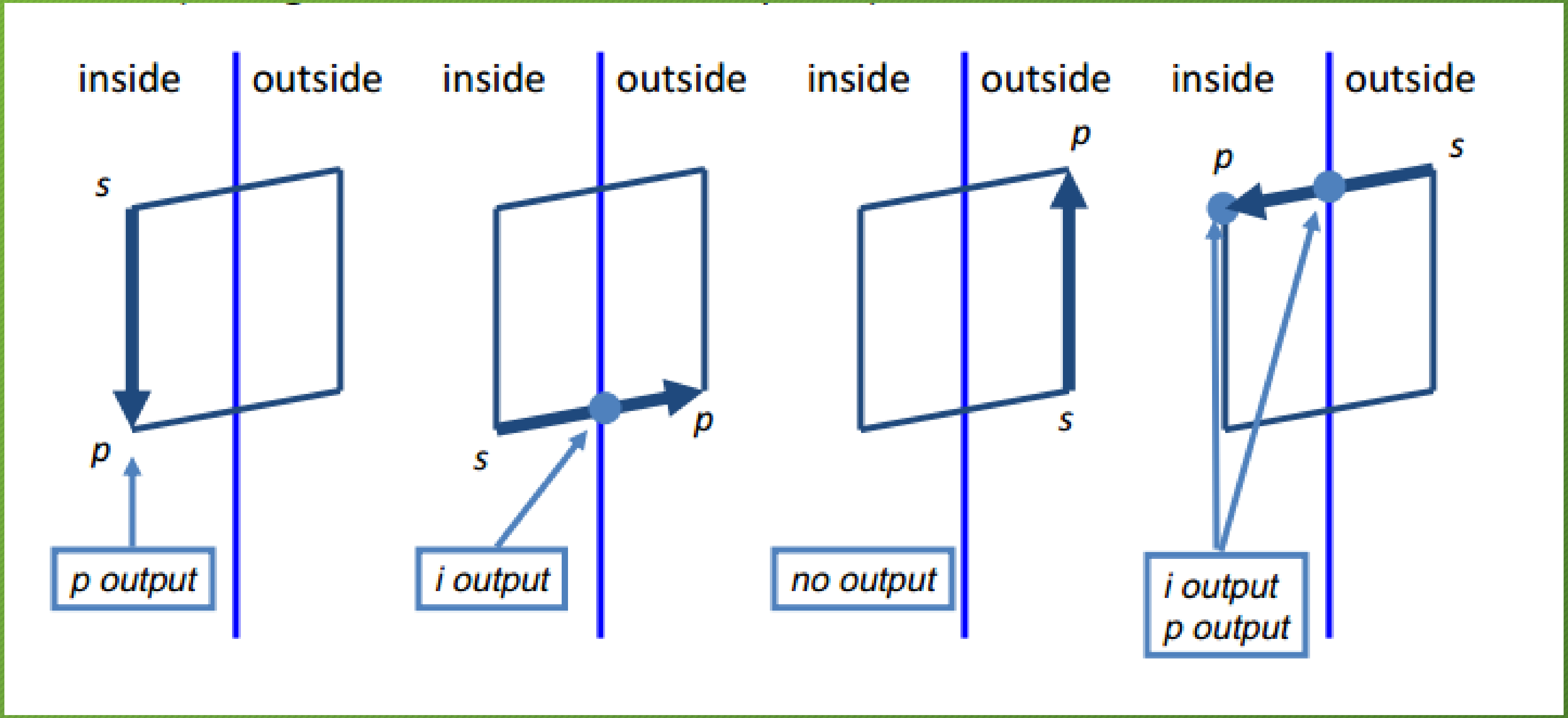
- Note: this is exactly what we expect from the clipping operation against each edge

## Sutherland-Hodgman Clipping

- Sutherland-Hodgman basic routine:
  - Go around polygon one vertex at a time
  - Current vertex has position *p*
  - Previous vertex had position *s*, and it has been added to the output if appropriate

## Sutherland-Hodgman Clipping

- Edge from s to p takes one of four cases:



## Sutherland-Hodgman Clipping

- **Four cases:**
  - ***s* inside plane and *p* inside plane**
    - Add *p* to output
    - Note: *s* has already been added
  - ***s* inside plane and *p* outside plane**
    - Find intersection point *i*
    - Add *i* to output
  - ***s* outside plane and *p* outside plane**
    - Add nothing
  - ***s* outside plane and *p* inside plane**
    - Find intersection point *i*
    - Add *i* to output, followed by *p*

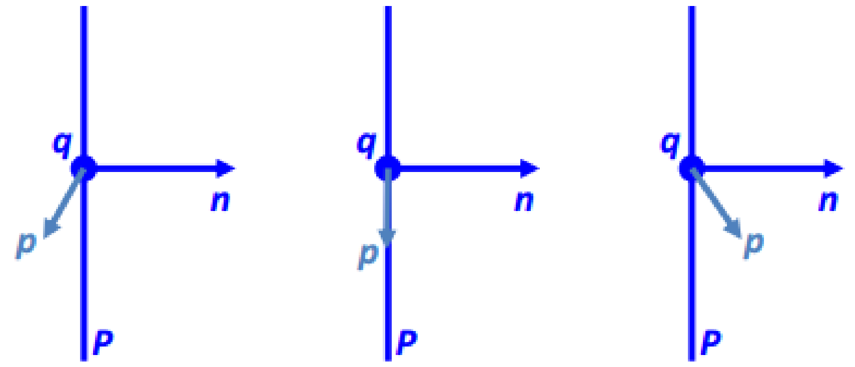
## Point-to-Plane test

- A very general test to determine if a point p is "inside" a plane P, defined by q and n:

| | |
|---|---|
| $(p - q) \bullet n < 0$: | $p$ inside $\boldsymbol{P}$ |
| $(p - q) \bullet n = 0$: | $p$ on $\boldsymbol{P}$ |
| $(p - q) \bullet n > 0$: | $p$ outside $\boldsymbol{P}$ |

***Remember:*** $\boldsymbol{p} \bullet n = |p|\ |n| \cos(\theta)$

$\theta$ = angle between p and n



## Finding Line-Plane Intersections

- Edge intersects plane P where E(t) is on P
  - q is a point on P
  - n is normal to P

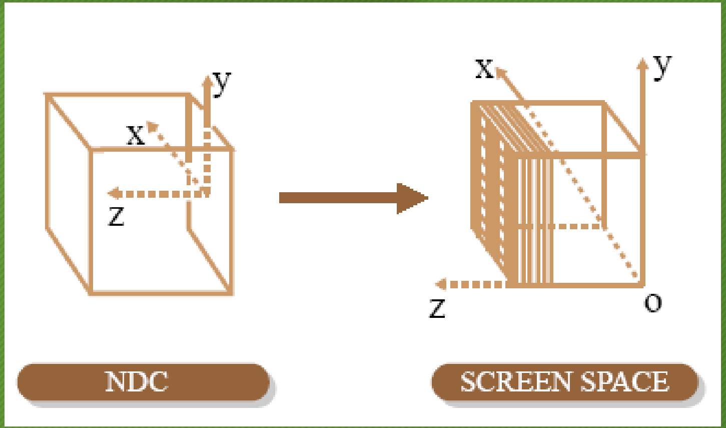
$$(L\,(t) - q) \bullet n = 0$$
$$(L0 + (L1 - L0)\,t - q) \bullet n = 0$$
$$t = [(q - L0) \bullet n] \,/\, [(L1 - L0) \bullet n]$$

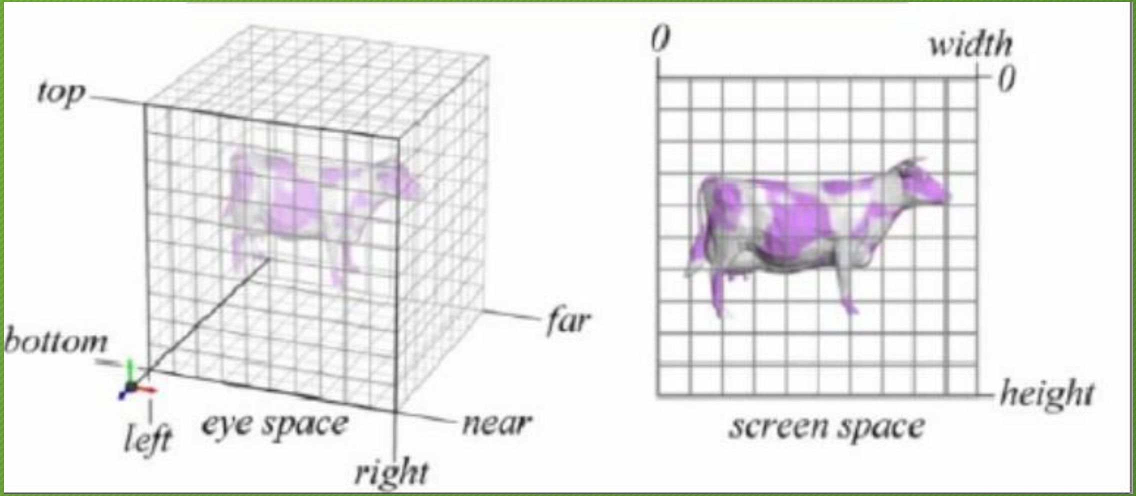
- **The intersection point** $i = L(t)$ **for this value of t**

# Viewport Transformation

- Modeling Transformation
- Lighting
- Viewing Transformation
- Projection Transformation
- Clipping
- **Viewport Transformation**
- Rasterization

- Maps NDC to 3D viewport:
  - xy gives the screen window
  - z gives the depth of each point

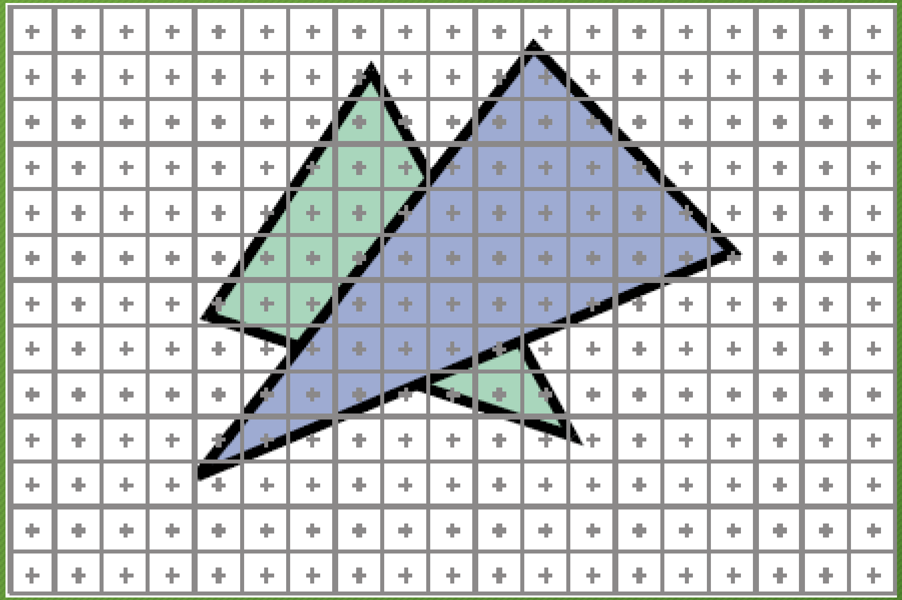




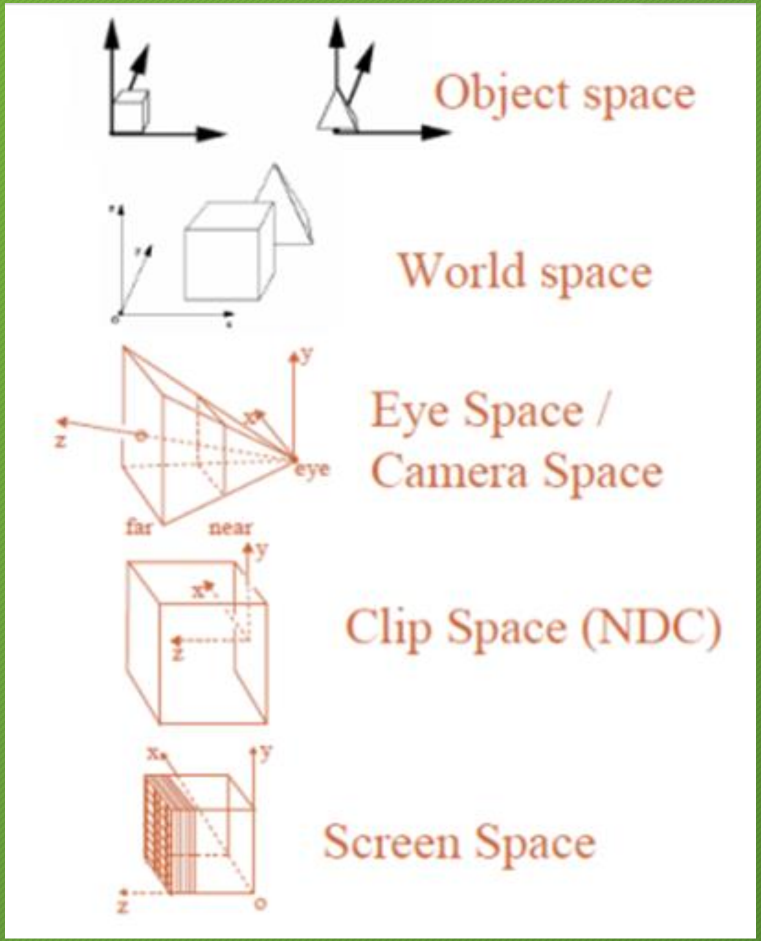
## Rasterization

- Modeling Transformation
- Lighting
- Viewing Transformation
- Projection Transformation
- Clipping
- Viewport Transformation
- **Rasterization**

- Rasterizes objects into pixels
- Interpolate values as we go (color, depth, etc.)



## Summary of transformations



## Recap: Rendering Pipeline

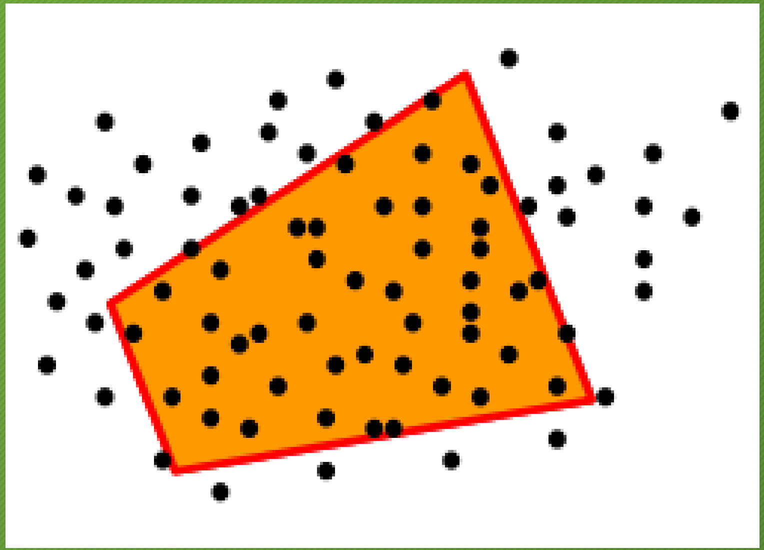
- Modeling transformations
- Viewing transformations
- Projection transformations
- Clipping
- Scan conversion
- We now know everything about how to draw a polygon on the screen, except visible surface determination

# Invisible Primitives

- Why might a polygon be invisible?
  - Polygon outside the *field of view*
  - Polygon is *backfacing*
  - Polygon is *occluded* by object(s) nearer the viewpoint
- For efficiency reasons, we want to avoid spending work on polygons outside field of view or backfacing
- For efficiency and correctness reasons, we need to know when polygons are occluded

## View Frustum Clipping

- Remove polygons entirely outside frustum
  - Note that this includes polygons “behind” eye (actually behind near plane)
- Pass through polygons entirely inside frustum
- Modify remaining polygons to include only portions intersecting view frustum

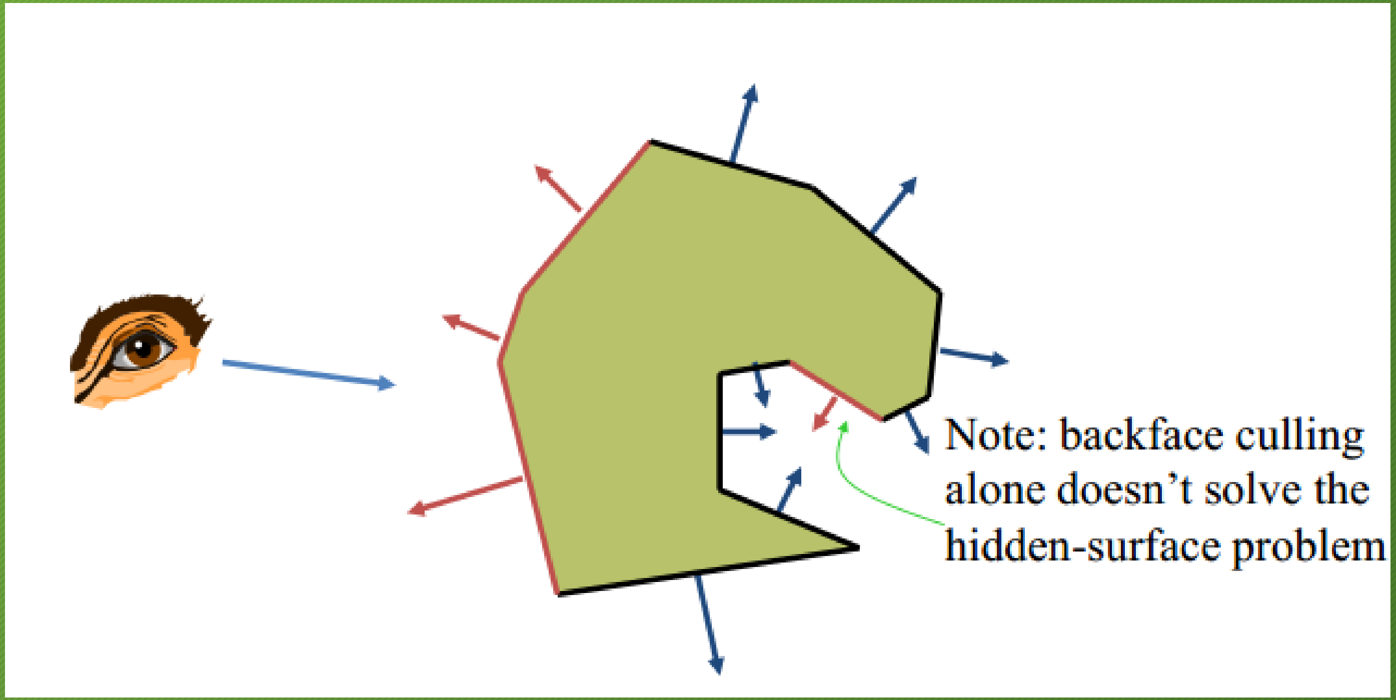


## Back-Face Culling

- Most objects in scene are typically “solid”
- More rigorously: closed, orientable manifolds
  - Must not cut through itself
  - Must have two distinct sides
    - A sphere is orientable since it has two sides, 'inside' and 'outside'.
    - A Mobius strip or a Klein bottle is not orientable
  - Cannot “walk” from one side to the other
    - A sphere is a closed manifold whereas a plane is not

## Back-Face Culling

- On the surface of a closed manifold, polygons whose normals point away from the camera are always occluded:


Note: backface culling alone doesn't solve the hidden-surface problem

## Back-Face Culling

- Not rendering backfacing polygons improves performance
- *By how much?*
  - *Reduces by about half the number of polygons to be considered for each pixel*
  - *Every front-facing polygon must have a corresponding rear-facing one*

# Occlusion

- For most interesting scenes, some polygons will overlap
- To render the correct image, we need to determine which polygons occlude which
- We don't focus on this here.