Master IGMMV
Synthèse d'images et de sons

George Drettakis

Nicolas Tsingos

INRIA
SOPHIA ANTIPOLIS

Séances

1. **Première séance (14 oct. 13:30-17:30, 4h)**

   Introduction en Images de Synthèse (3h) et Programmation OpenGL I (1h)

2. Deuxième séance (27 oct : 13:30-17:00, 3h30)

   1ère partie : Programmation OpenGL II et introduction mini projet (1h)
   2ème partie : Rendu Temps Réel (1h30)
   3ème partie : Rendu Audio I (1h)

3. Troisième séance : (8 nov, 13:30-17:00, 3h30)

   1ère partie : Rendu Audio II (2h)
   2ème partie : API et programmation Audio (1h30)

4. Quatrième séance : (1er déc, 13:30-17:30, A CONFIRMER, 4h)

   1ère partie :Visibilité, ombres et éclairage global (2h30)
   2ème partie : Perception (1h30)

Séance 1 : Introduction en Images de Synthèse (GD)

- 1ère partie (30min)
  - – Vue générale de la synthèse d'image et de sons
- 2ème partie (2h30)
  - – Pipeline graphique « classique »
    - Transformations, paramètres de vue, lighting, clipping
- *PAUSE*
    - Parties cachées, scanline
- 3ème partie (1h)
  - – Intro à OpenGL

- *Diapositives basées sur le cours de Durand et Cutler au MIT*

Image de Synthèse et des Sons : Applications

- Jeux
- Simulation
- CAO et design
- Architecture/urbanisme
- Réalité Virtuelle
- Visualisation
- Imagerie Médicale

Jeux

Jeux

- Temps réel
- Approximation
- Matériel spécifique
  - Playstation, X-box
  - Très performants mais difficile pour le développement
- « Moteur » de l’industrie actuellement

Simulation

Simulation

- Visualisation d'un processus réel
- Précision de calculs
- Masses de données
- Historiquement, premières applications temps réel
  - Simulateurs de vols

CAO & design

CAO/Design

- Géométrie
- Précision
- « Beauté » d'affichage
- Problèmes géométriques
  - Toute l'histoire de la modélisation

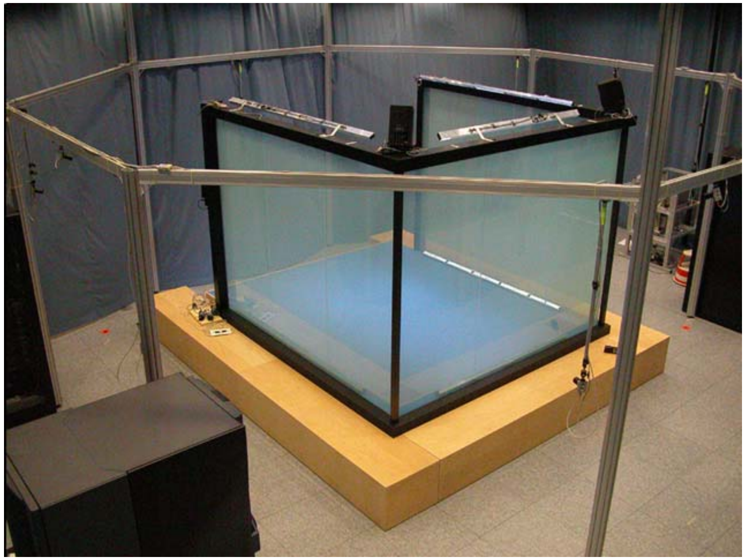
Architecture/urbanisme

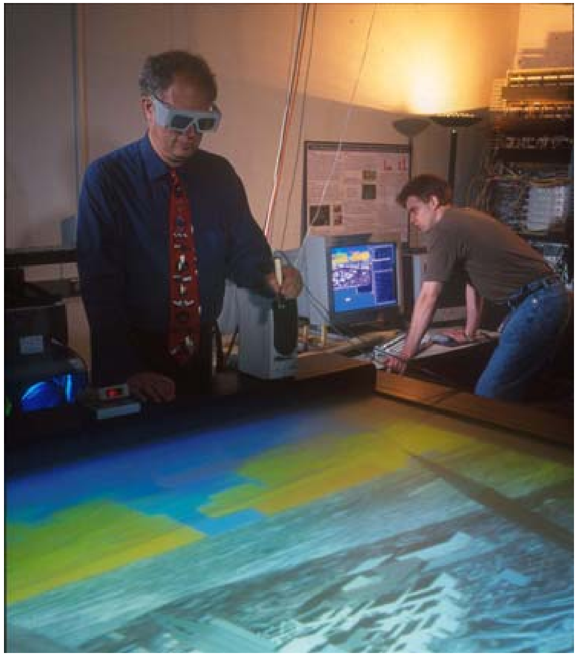
Architecture/Urbanisme

- Intérieur/extérieur
- Réalisme très important
  - Image/éclairage
  - Son
- Interactivité
  - « vivre » la nouvelle conception

- DEMO Garibaldi

Réalité Virtuelle

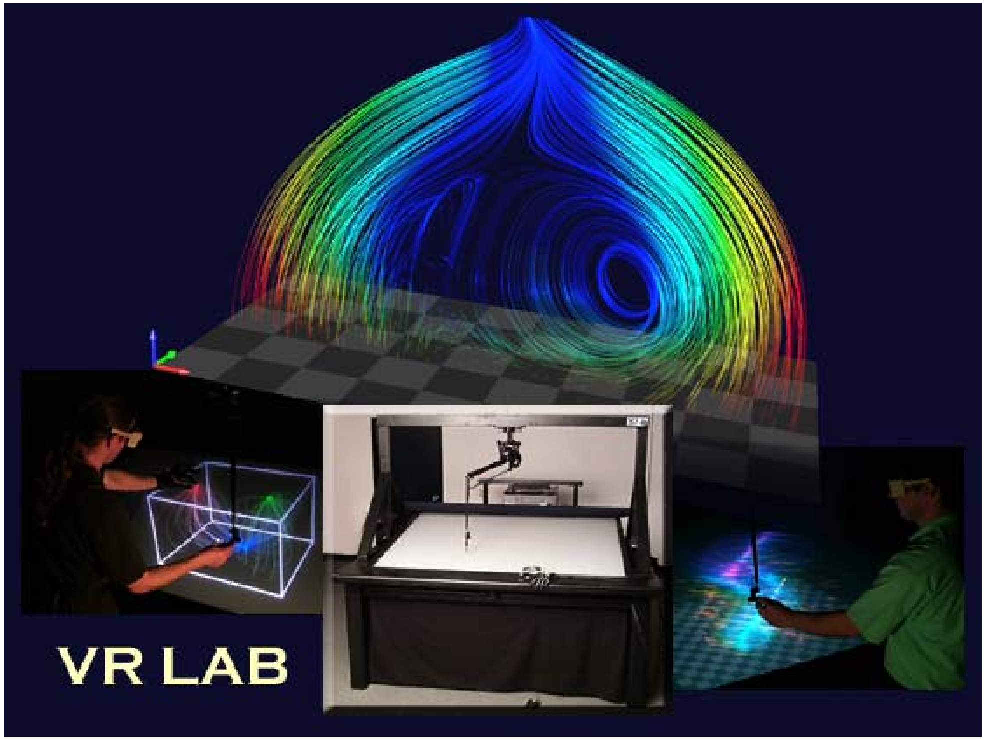


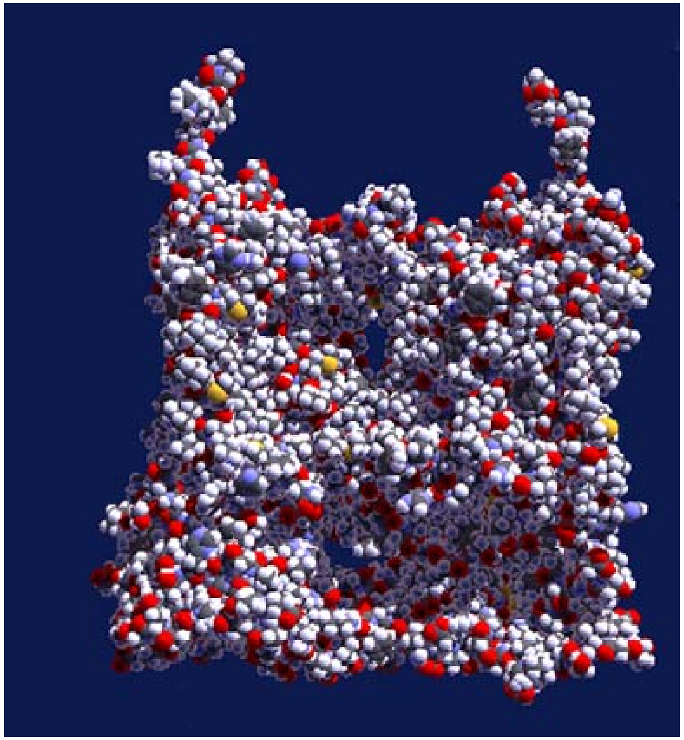


Réalité Virtuelle

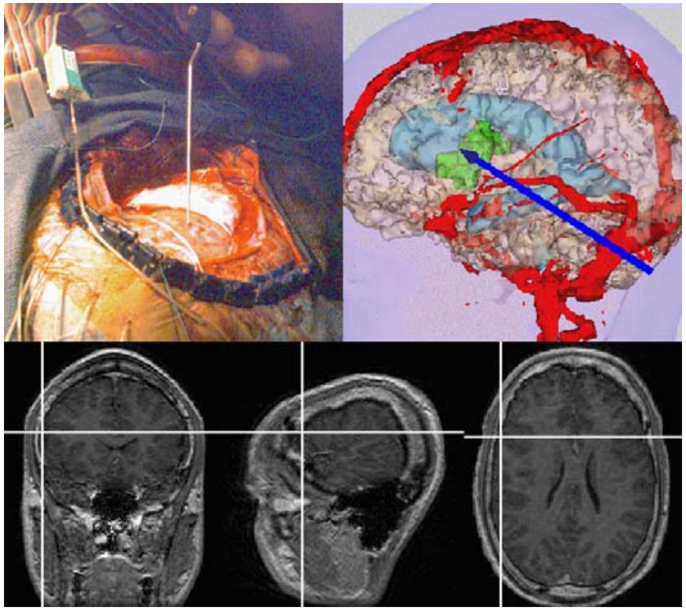
- Multidisciplinaire
- Systèmes d'affichage divers
  - Workbench (à Sophia aussi), visite prévue
  - CAVE
- Stéréovision
- Tracking

Visualisation





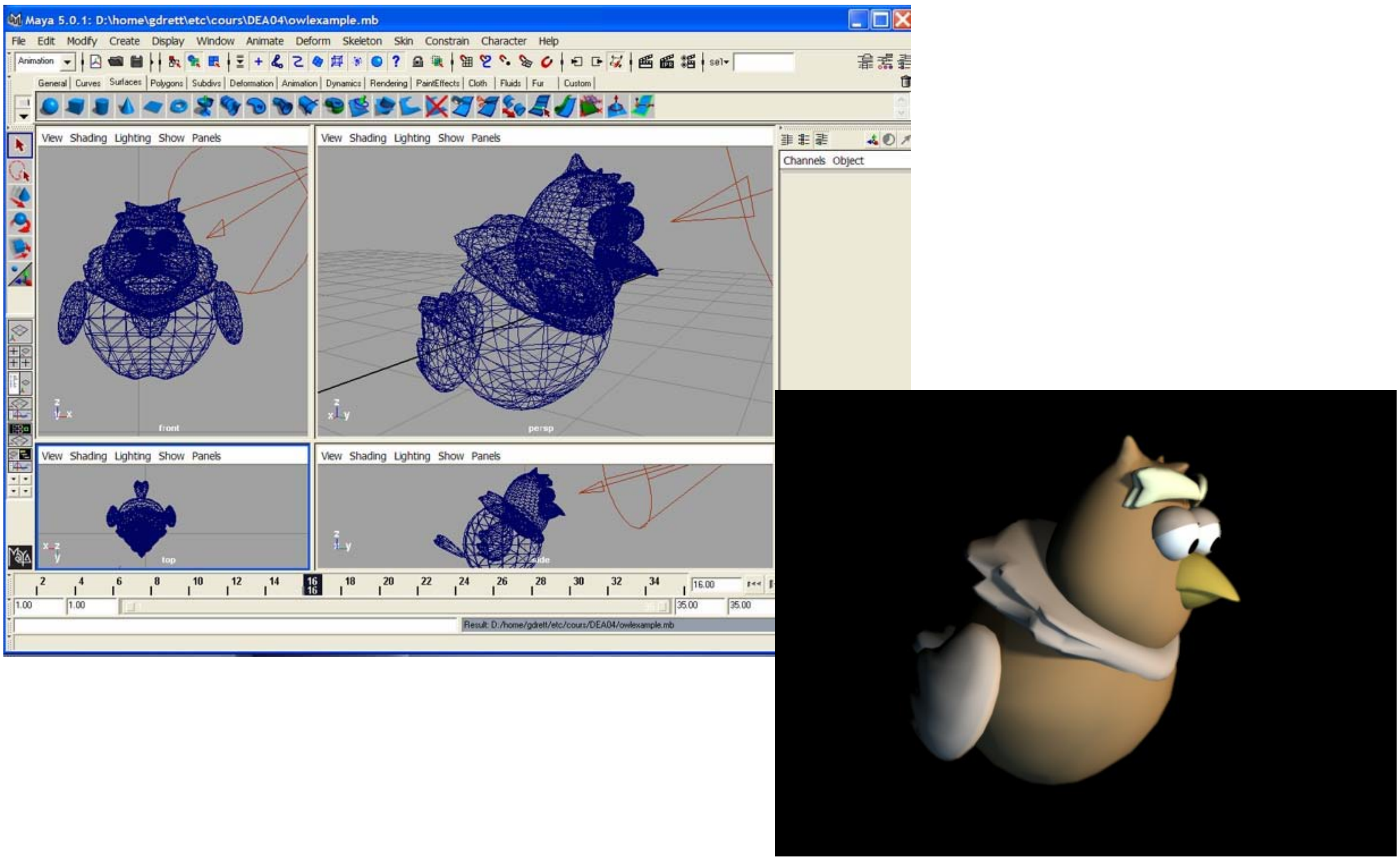
Imagerie Médicale



Vizu + Applis Médicales

- Temps réel
- Certification
- Visualisation des données
  - Représentation de quantités physiques
- Outil important pour comprendre
- Outil « d’intervention » pour le médical

Création d’images : le Procesus



Étapes du Processus

- Modélisation
- Animation
- **Rendu**

*Dans ce cours nous nous concentrons sur la partie **Rendu***

Modélisation

- Représentations 3D
  - Points, lignes, polygones
  - Surfaces « lisses »
- Transformations
  - Déplacement
  - Échelle
  - Rotation

Modélisation

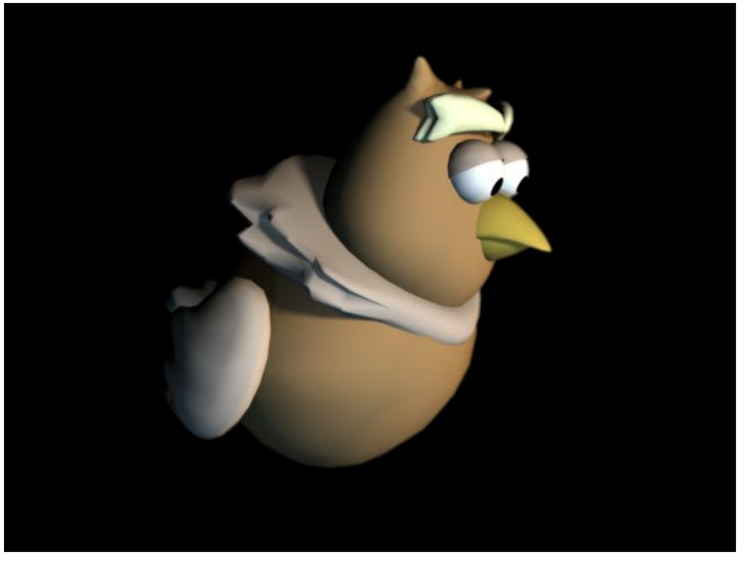
- Types de surfaces
  - Surfaces d’interpolation
    - Lineaires (polygones), quadratiques, cubiques
  - Splines
    - Bézier, B-splines
    - NURBS
  - Surfaces de subdivision
- Exemples Maya

Animation

- Keyframes
  - Déterminer des positions clefs
  - Interpolation entre ces positions

Animation

- Simulation physique
  - Cinématique inverse



- Exemple Maya

Rendu

- Pipeline graphique
  - Traditionnel
  - Logiciel
- Important de comprendre les principes
- Accélération Graphique matériel
  - Puissance des GPUs
  - Z-buffer
  - Scan conversion

Séance 1 : Le « pipeline » graphique

- Transformations de modélisation
- Éclairage (Shading)
- Transformation de vue (Perspective / Orthographique)
- Clipping
- Projection (à l’espace ecran)
- Scan Conversion (Rasterization)
- Visibilité / Affichage

Entrée :

modèle 3D,
source de lumière,
description de matériaux,
position de la caméra,
fenêtre sur l’écran

Sortie : une image (tableau de pixels)

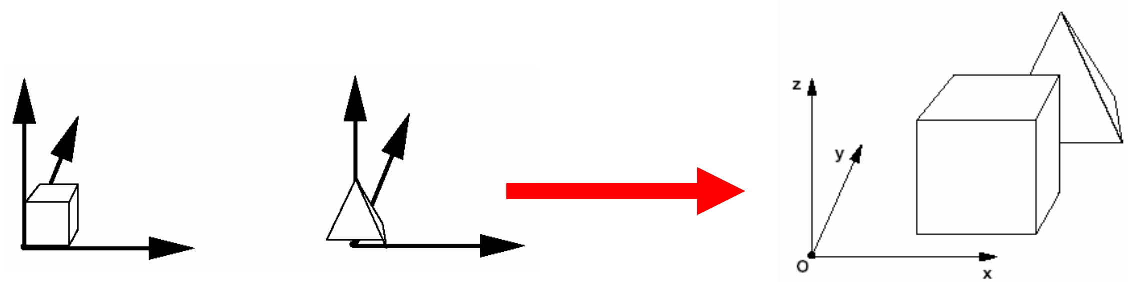
Séance 1 : Pipeline graphique

- Transformations
  - Matrices 4x4
- Éclairage
  - Modèle d’éclairage local
- Paramètres de vue
  - Projection perspective/orthographique
- Scan conversion
  - Algorithmes incrementaux
- Parties cachées et affichage
  - Algorithmes discrets (accélérés par le matériel)

Modeling Transformations

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

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



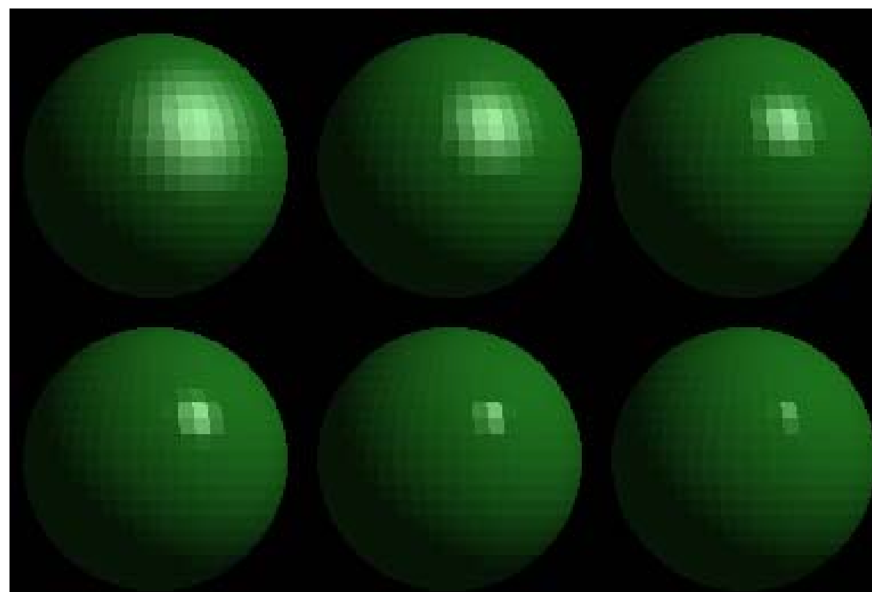
Object space World space

Durand/Cutler MIT

Illumination (Shading) (Lighting)

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

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

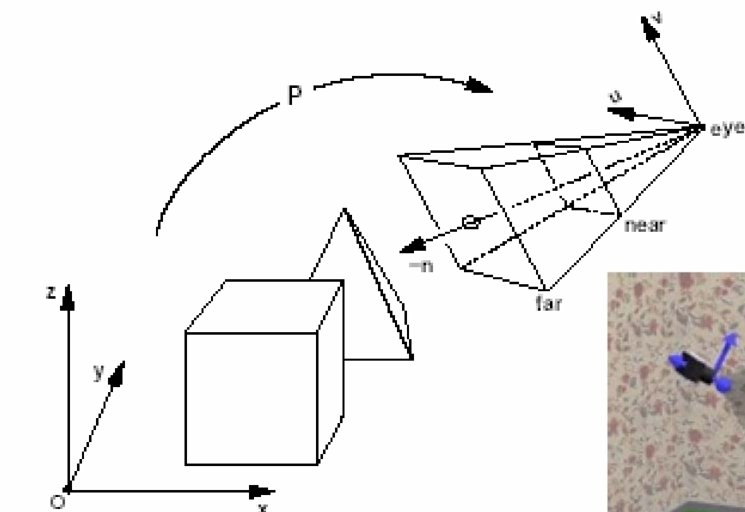


Durand/Cutler MIT

Viewing Transformation

- Modeling Transformations
- Illumination (Shading)
- **Viewing Transformation (Perspective / Orthographic)**
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

- Maps world space to eye space
- Viewing position is transformed to origin & direction is oriented along some axis (usually *z*)



Eye space

World space

Durand/Cutler MIT

Clipping

Modeling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

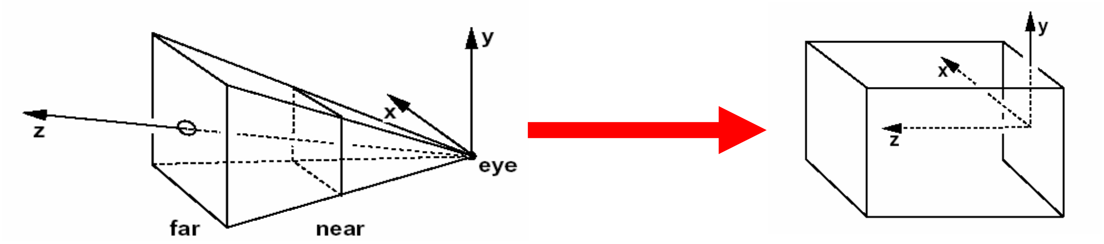
Clipping

Projection (to Screen Space)

Scan Conversion (Rasterization)

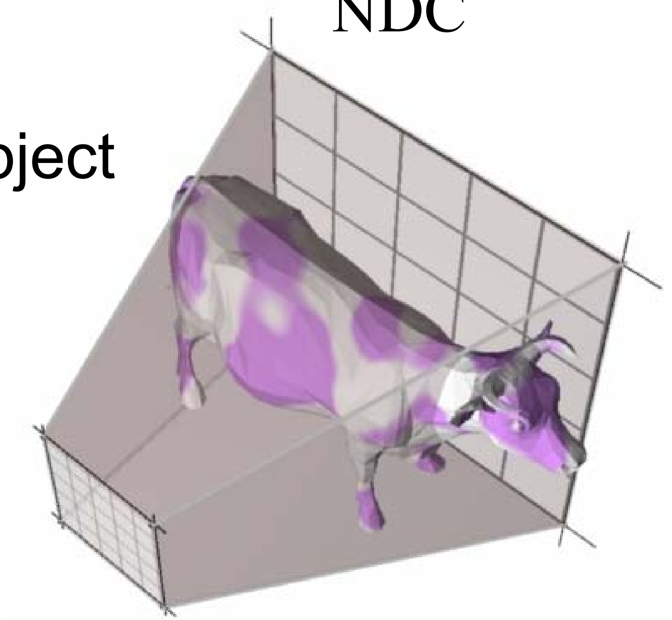
Visibility / Display

- Transform to Normalized Device Coordinates (NDC)



Eye space NDC

- Portions of the object outside the view volume (view frustum) are removed

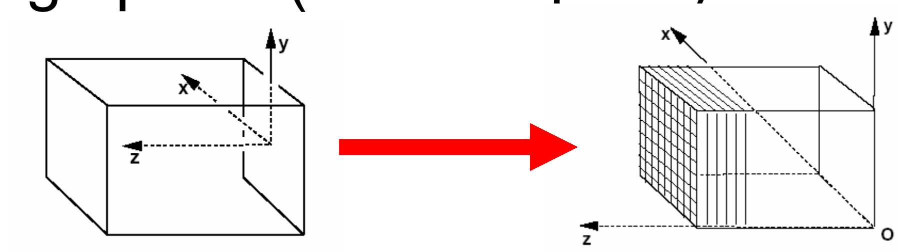


Durand/Cutler MIT

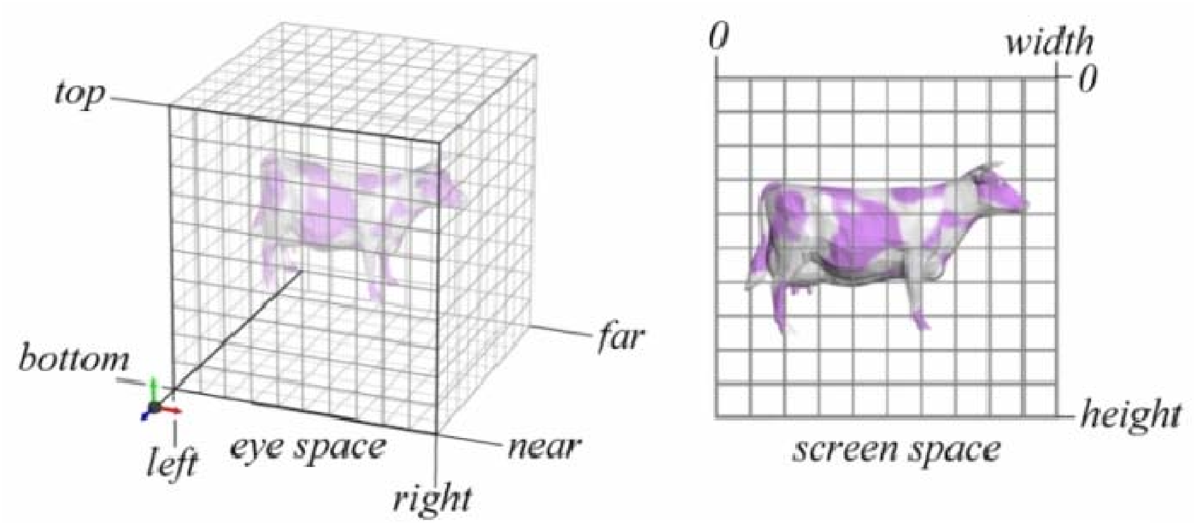
Projection

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- **Projection (to Screen Space)**
- Scan Conversion (Rasterization)
- Visibility / Display

- The objects are projected to the 2D image place (screen space)



NDC   Screen Space



Durand/Cutler MIT

Scan Conversion (Rasterization)

Modeling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

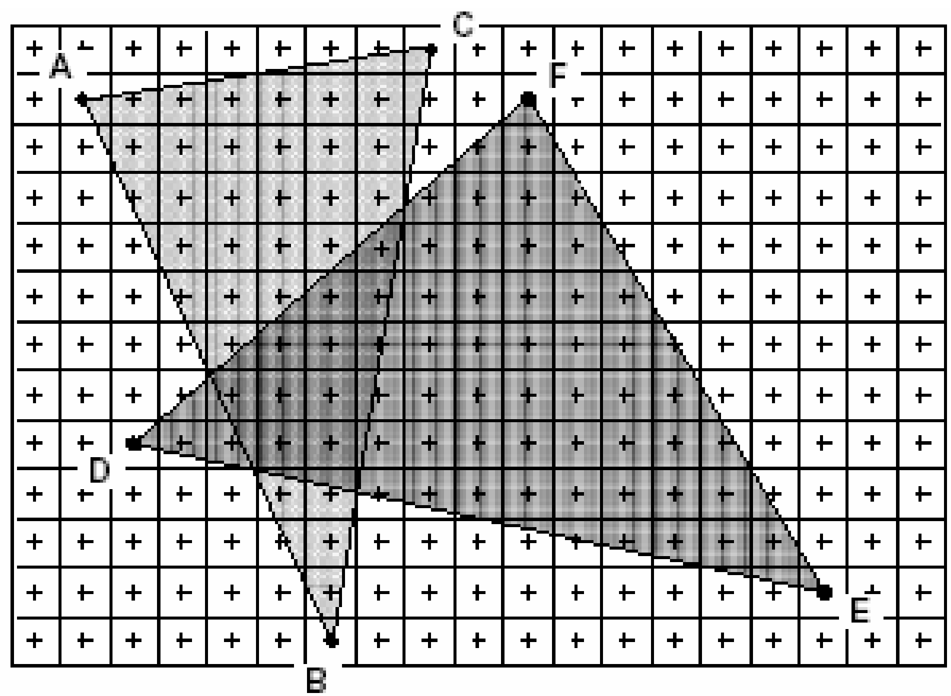
Clipping

Projection (to Screen Space)

Scan Conversion (Rasterization)

Visibility / Display

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



Durand/Cutler MIT

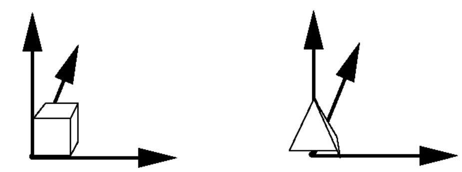
Visibility / Display

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

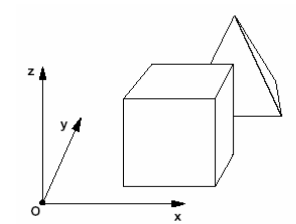
- Each pixel remembers the closest object (depth buffer)
- Almost every step in the graphics pipeline involves a change of coordinate system. Transformations are central to understanding 3D computer graphics.

Common Coordinate Systems

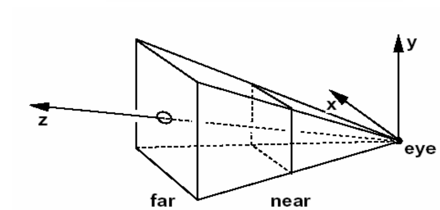
- Object space
  - local to each object



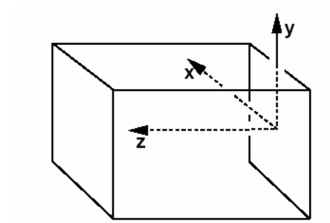
- World space
  - common to all objects



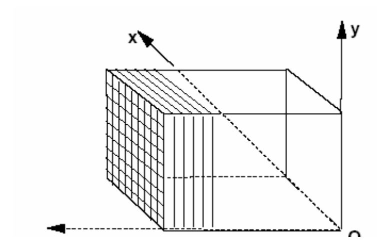
- Eye space / Camera space
  - derived from view frustum



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

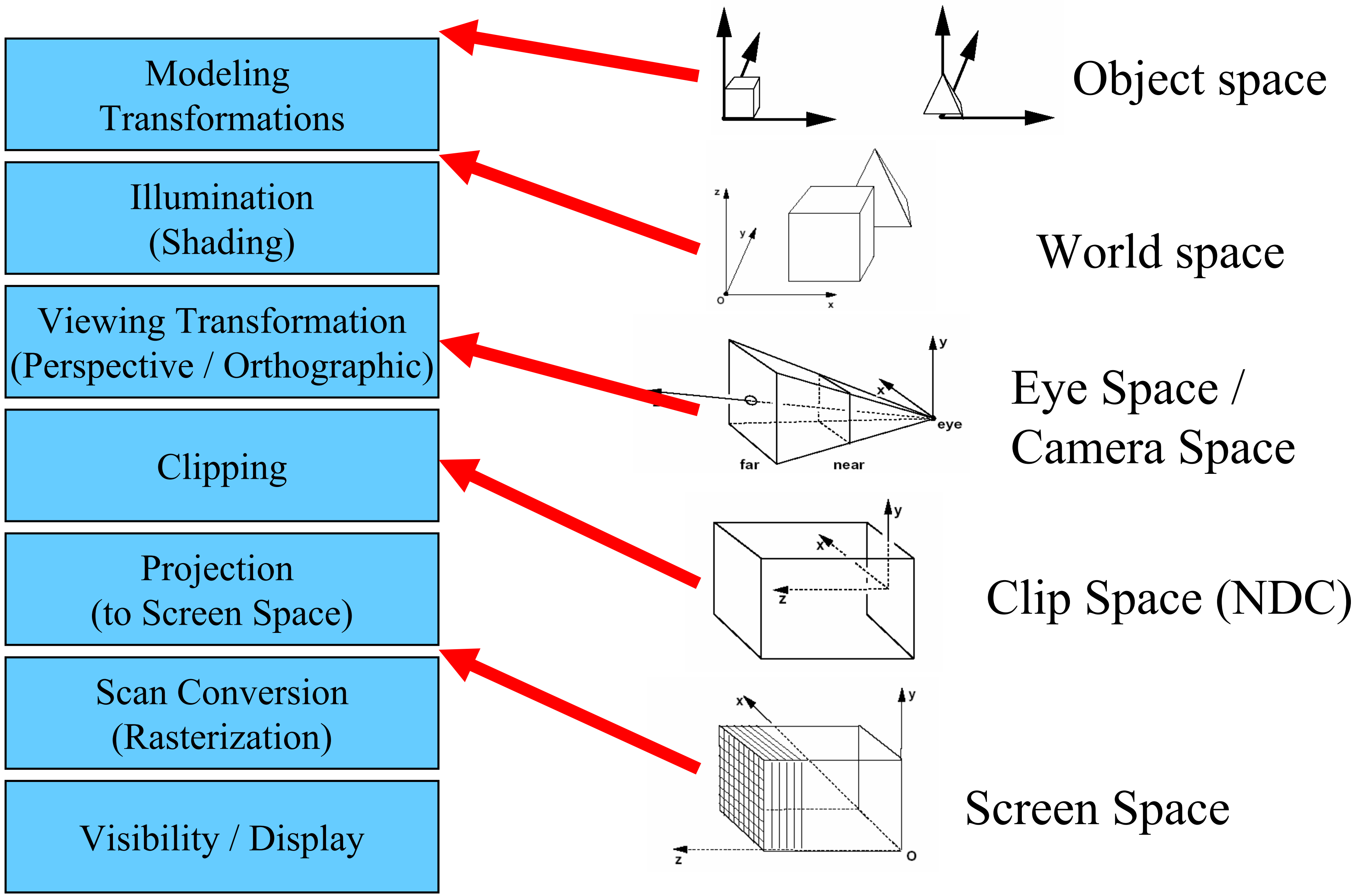


- Screen space
  - indexed according to hardware attributes



Durand/Cutler MIT

Coordinate Systems in the Pipeline



Durand/Cutler MIT

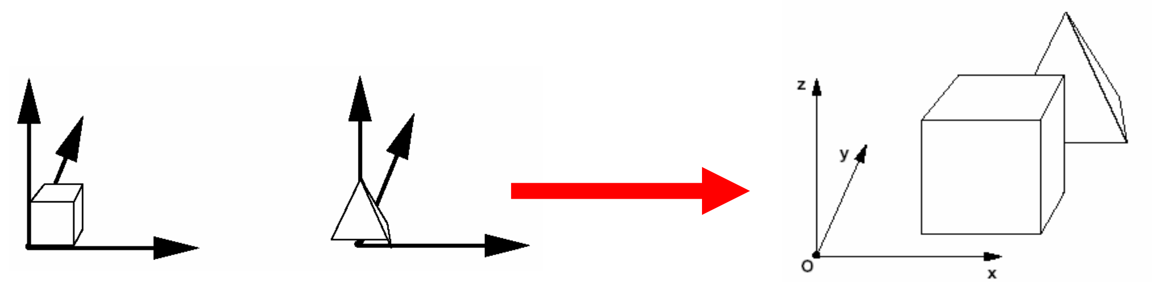
Transformations

- Deux buts principaux :
  - Déplacer, placer ou dimensionner les objets
  - Déterminer les paramètres de vue
- Systèmes de coordonnées
  - Homogènes
  - Transformations par matrices 4x4

Modeling Transformations

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

- 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

Durand/Cutler MIT

How are Transforms Represented?

$$x' = ax + by + c$$

$$y' = dx + ey + f$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{bmatrix} a & b \\ d & e \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{bmatrix} c \\ f \end{bmatrix}$$

$$p' = Mp + t$$

Homogeneous Coordinates

- Add an extra dimension
  - in 2D, we use 3 x 3 matrices
  - In 3D, we use 4 x 4 matrices
- Each point has an extra value, *w*

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

$$p' = \color{red}{M}\, p$$

Translation in homogenous coordinates

$$x' = ax + by + c$$

$$y' = dx + ey + f$$

Affine formulation

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{bmatrix} a & b \\ d & e \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{bmatrix} c \\ f \end{bmatrix}$$

$$p' = M p + t$$

Homogeneous formulation

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

$$p' = M p$$

Homogeneous Coordinates

- Most of the time w = 1, and we can ignore it

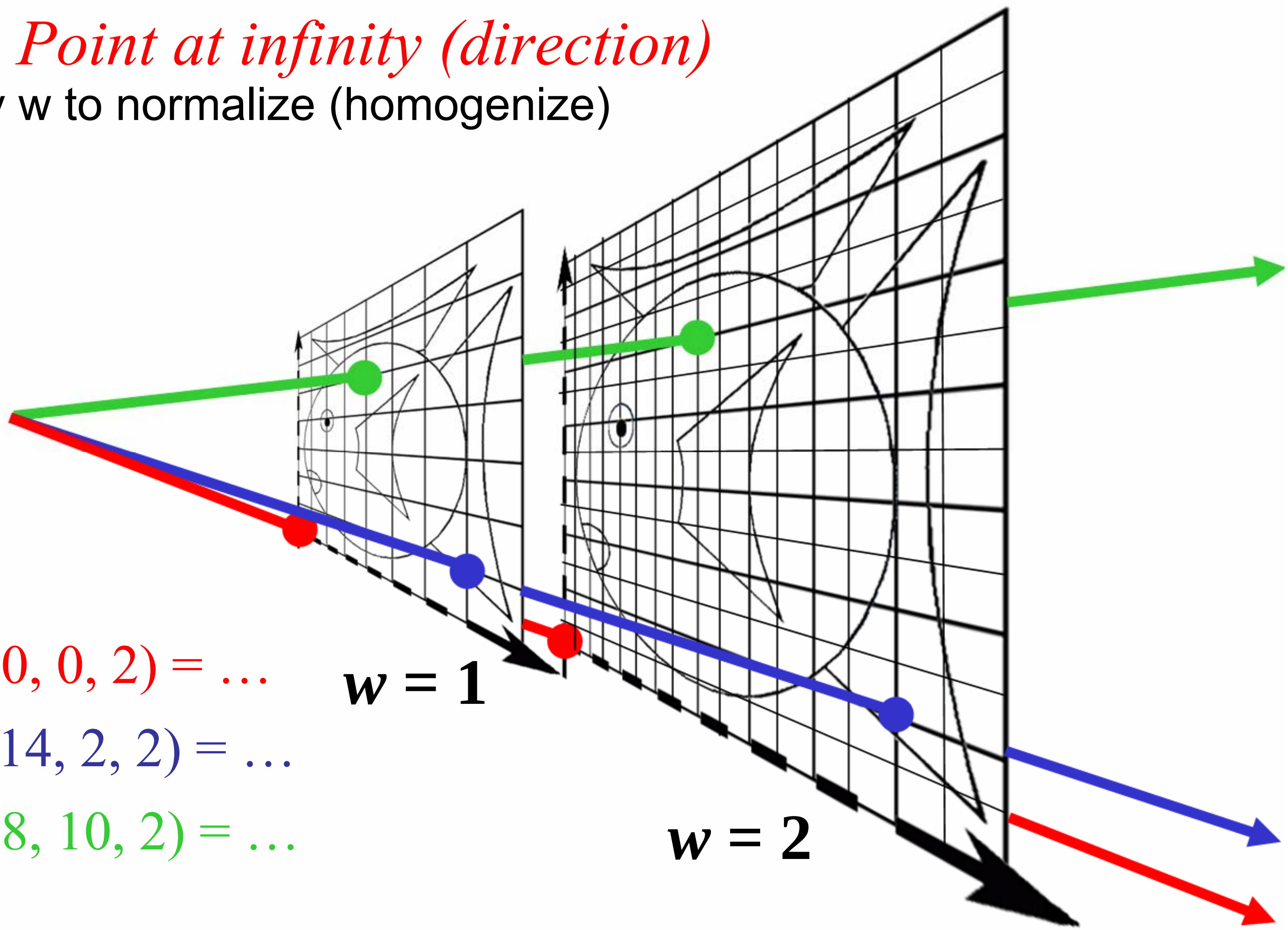
$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

- If we multiply a homogeneous coordinate by an *affine matrix*, w is unchanged

Homogeneous Visualization

Point at infinity (direction)

- Divide by w to normalize (homogenize)
- W = 0?



(0, 0, 1) = (0, 0, 2) = …

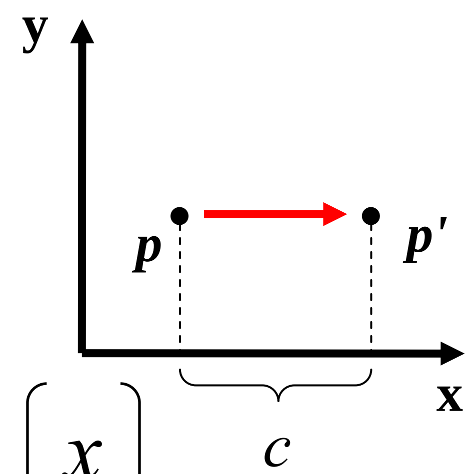
(7, 1, 1) = (14, 2, 2) = …

(4, 5, 1) = (8, 10, 2) = …

Translate (t_x, t_y, t_z)

- Why bother with the extra dimension?
  Because now translations can be encoded in the matrix!

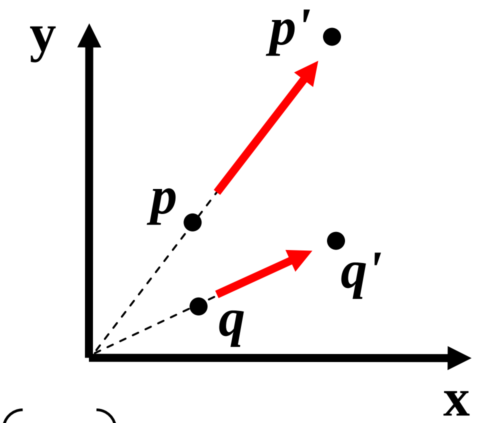
Translate(c,0,0)



$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Scale (s_x, s_y, s_z)

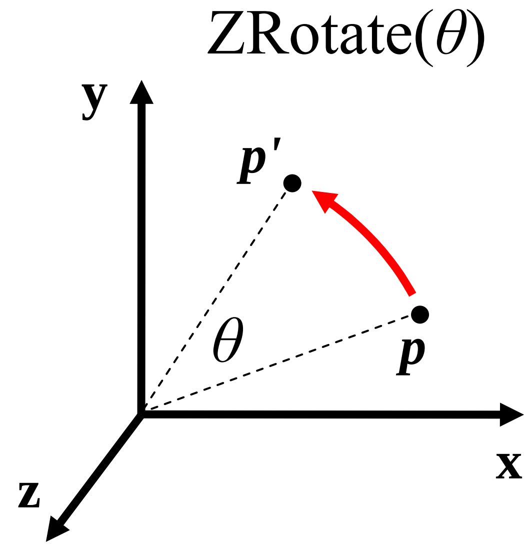
- Isotropic (uniform) scaling: $s_x = s_y = s_z$



$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Rotation

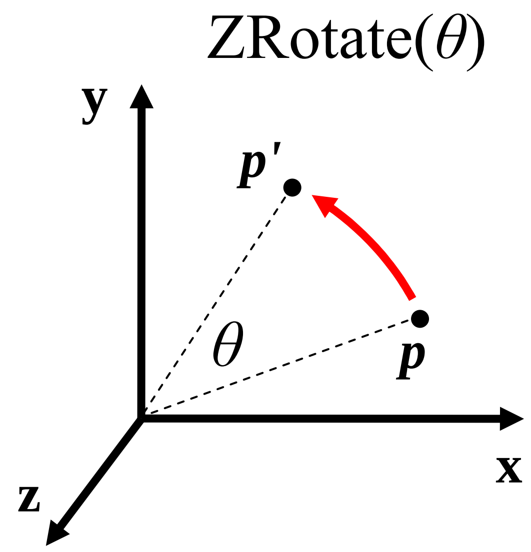
- About z axis



$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Rotation

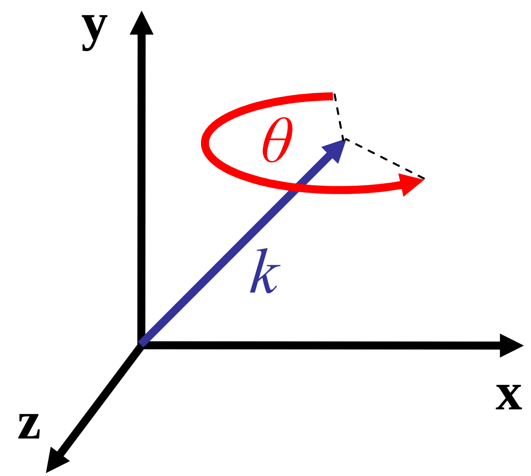
- About x axis



$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Rotate(k, θ)

Rotation



- About (k_x, k_y, k_z), a unit vector on an arbitrary axis (Rodrigues Formula)

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} k_x k_x(1-c)+c & k_z k_x(1-c)-k_z s & k_x k_z(1-c)+k_y s & 0 \\ k_y k_x(1-c)+k_z s & k_z k_x(1-c)+c & k_y k_z(1-c)-k_x s & 0 \\ k_z k_x(1-c)-k_y s & k_z k_x(1-c)-k_x s & k_z k_z(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

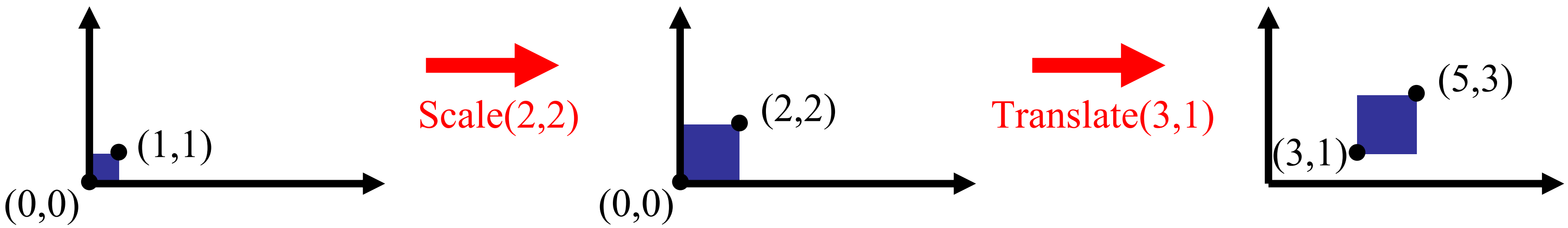
where $c = \cos\theta$ & $s = \sin\theta$

Storage

- Often, *w* is not stored (always 1)
- Needs careful handling of direction vs. point
  – Mathematically, the simplest is to encode directions with w=0
  – In terms of storage, using a 3-component array for both direction and points is more efficient
  – Which requires to have special operation routines for points vs. directions

How are transforms combined?

Scale then Translate



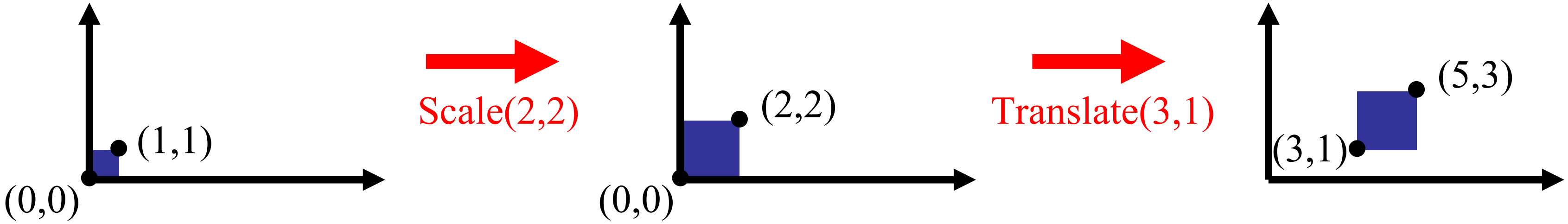
Use matrix multiplication: $p' = T(Sp) = TSp$

$$TS = \begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 3 \\ 0 & 2 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

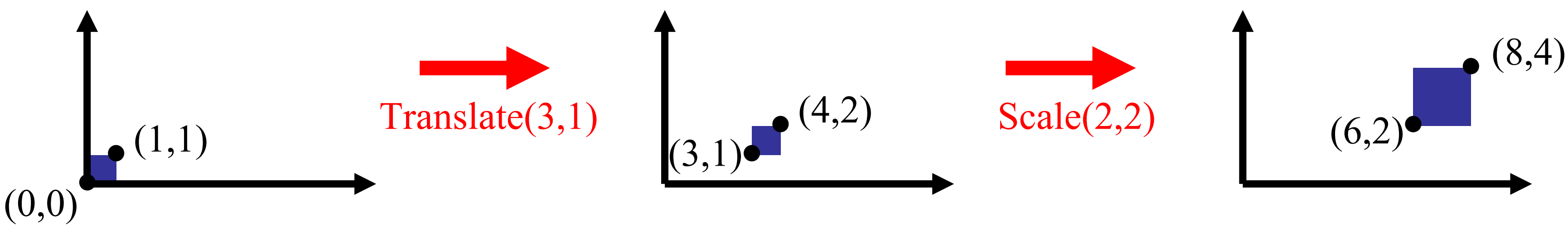
Caution: matrix multiplication is NOT commutative!

Non-commutative Composition

Scale then Translate: $p' = T(Sp) = TSp$



Translate then Scale: $p' = S(Tp) = STp$



Non-commutative Composition

Scale then Translate: $p' = T(S\,p) = TS\,p$

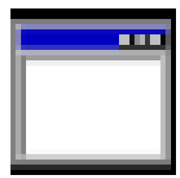
$$TS = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 3 \\ 0 & 2 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Translate then Scale: $p' = S(T\,p) = ST\,p$

$$ST = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 6 \\ 0 & 2 & 2 \\ 0 & 0 & 1 \end{bmatrix}$$

Demo

- Transformation with 2 matrices



transformation.exe

Illumination (Shading) (Lighting)

Modeling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

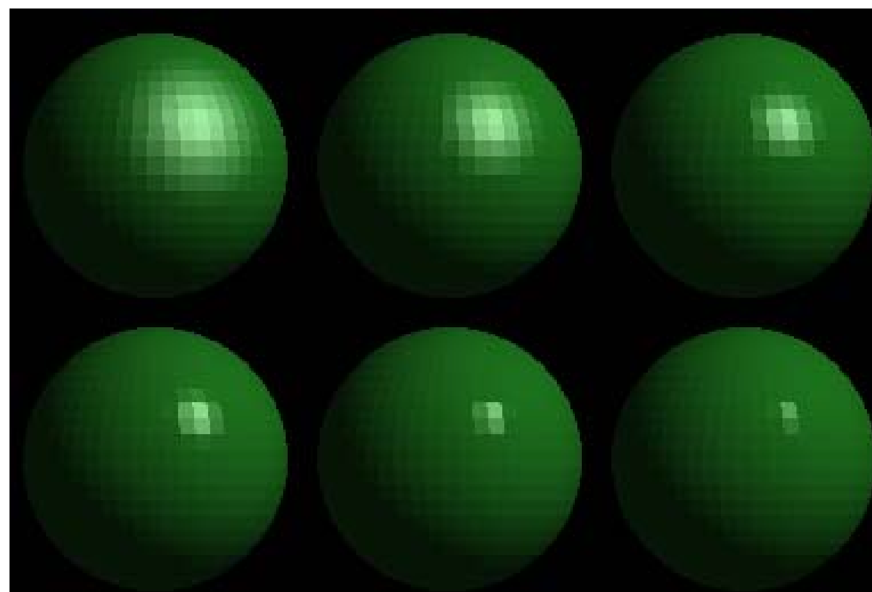
Clipping

Projection (to Screen Space)

Scan Conversion (Rasterization)

Visibility / Display

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



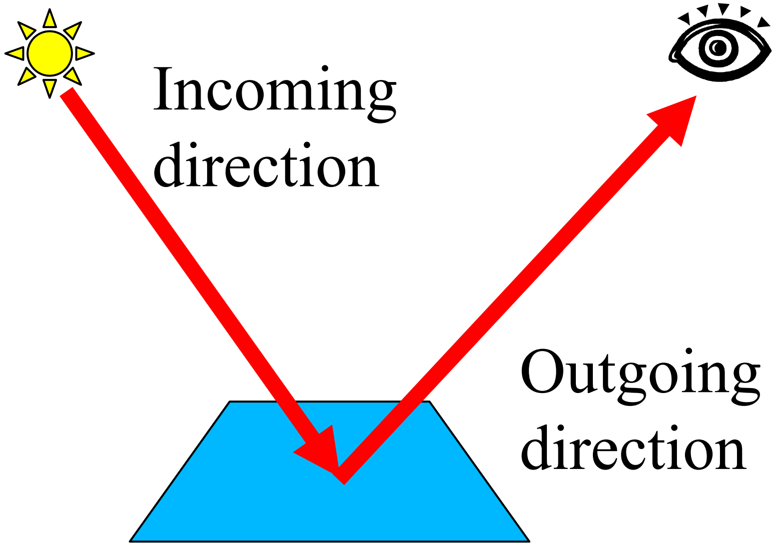
Durand/Cutler MIT

Local Illumination



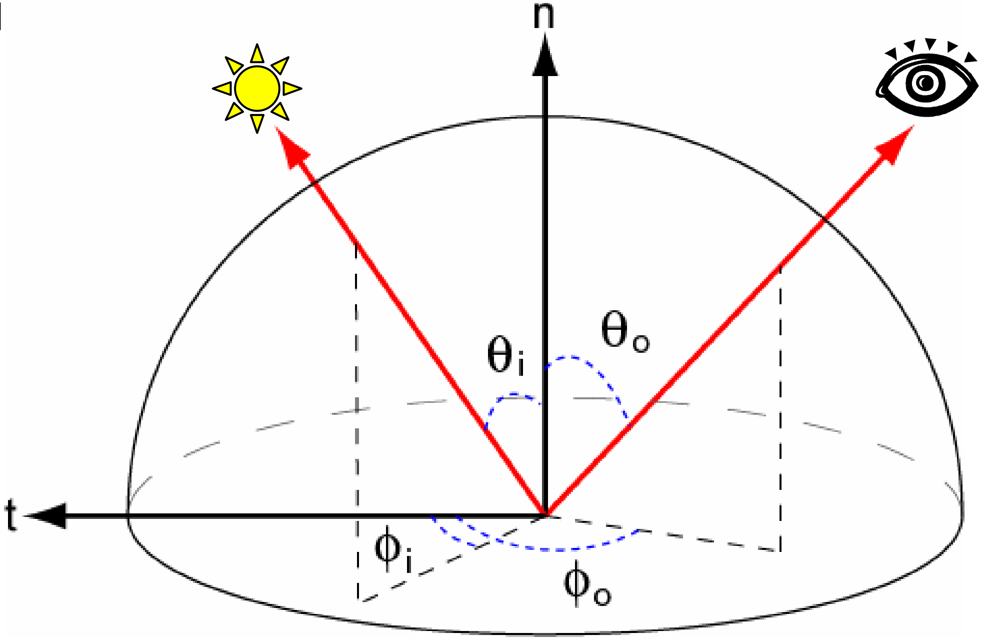

BRDF

- Ratio of light coming from one direction that gets reflected in another direction
- Bidirectional Reflectance Distribution Function



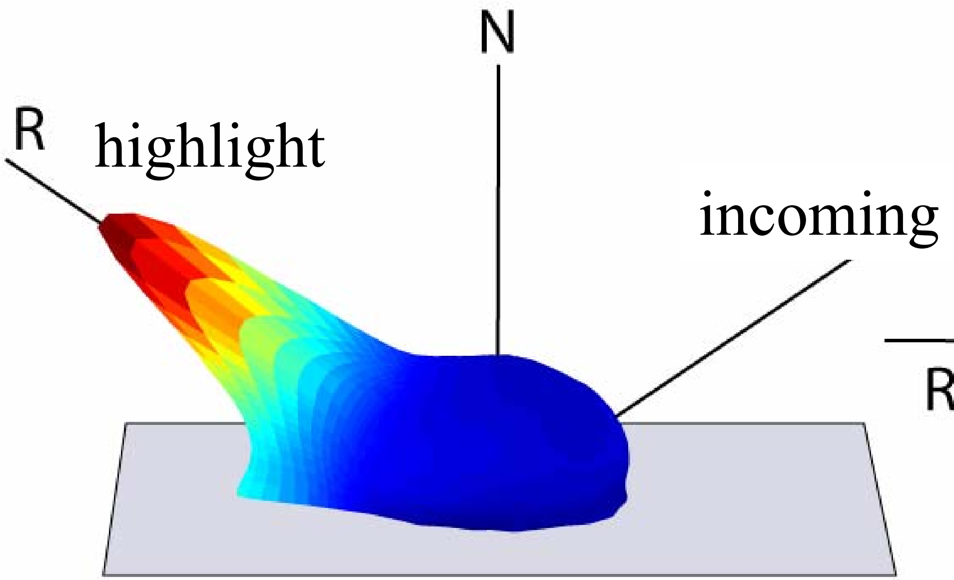
BRDF

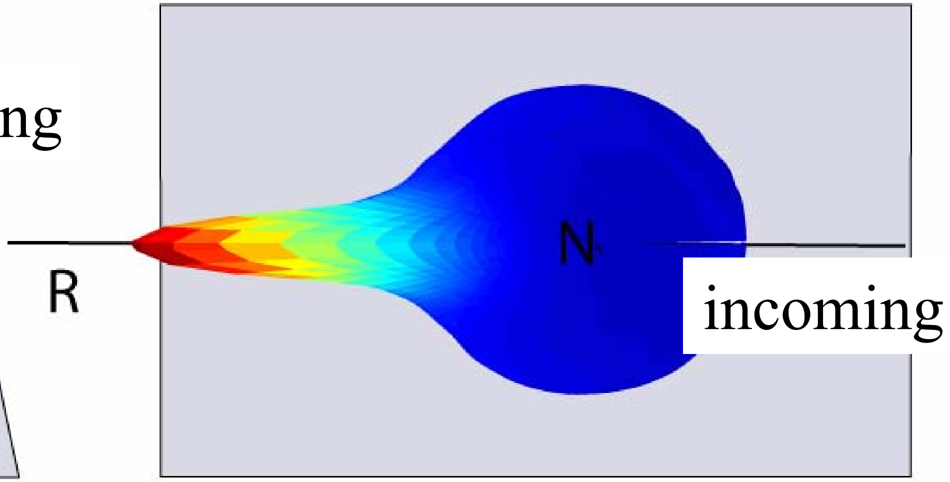
- Bidirectional Reflectance Distribution Function
  - 4D
    - 2 angles for each directi
  - $R(\theta_i, \phi_i; \theta_o, \phi_o)$



Slice at constant incidence

- 2D spherical function





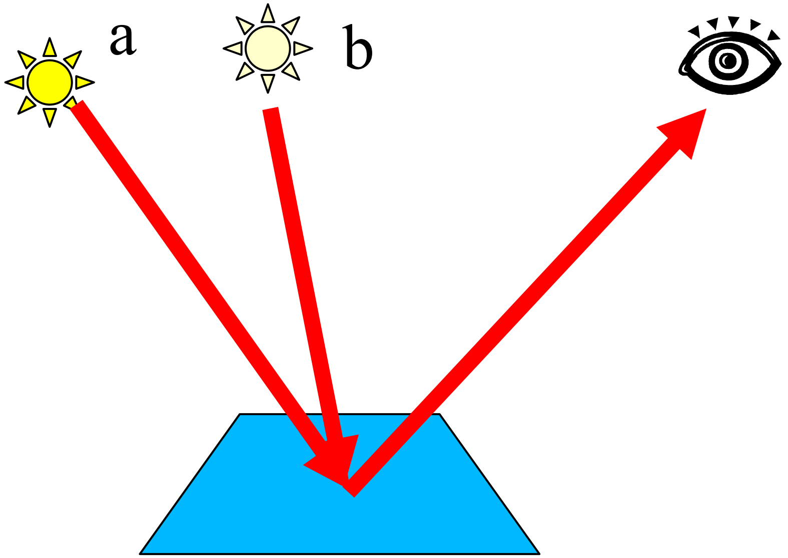
Example: Plot of "PVC" BRDF at 55° incidence

Unit issues - radiometry

- We will not be too formal in this lecture
- Typical issues:
  - Directional quantities vs. integrated over all directions
  - Differential terms: per solid angle, per area, per time
  - Power, intensity, flux

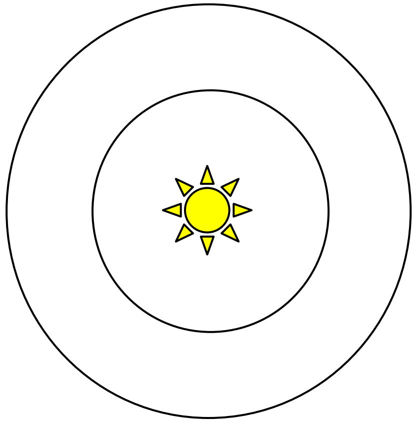
Light sources

- Today, we only consider point light sources
- For multiple light sources, use linearity
  - We can add the solutions for two light sources
    - I(a+b)=I(a)+I(b)
  - We simply multiply the solution when we scale the light intensity
    - I(s a) = s I(a)



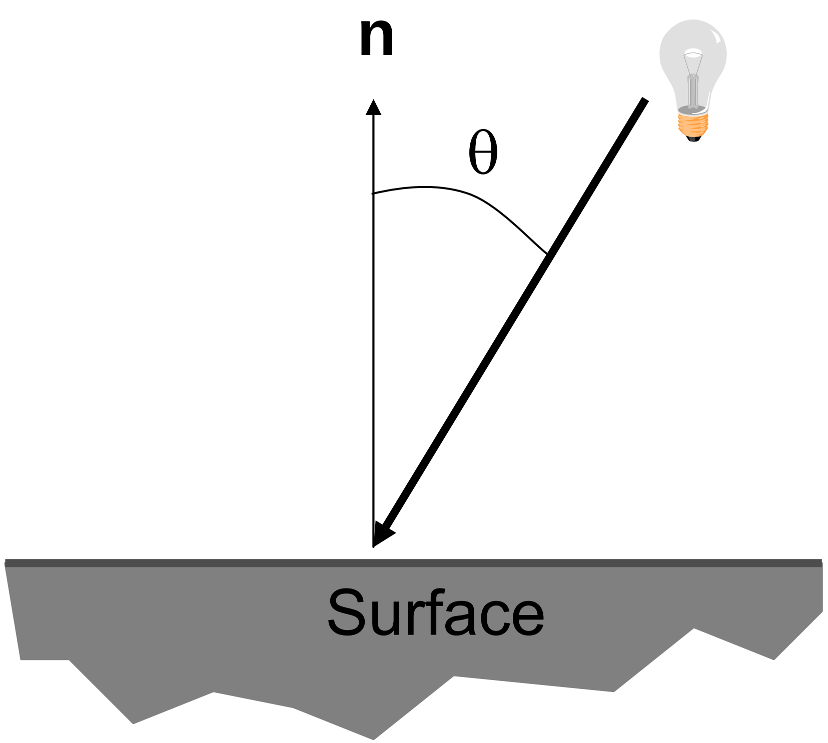
Light intensity

- $1/r^2$ falloff
  - Why?
  - Same power in all concentric circles



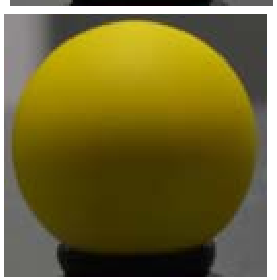
Incoming radiance

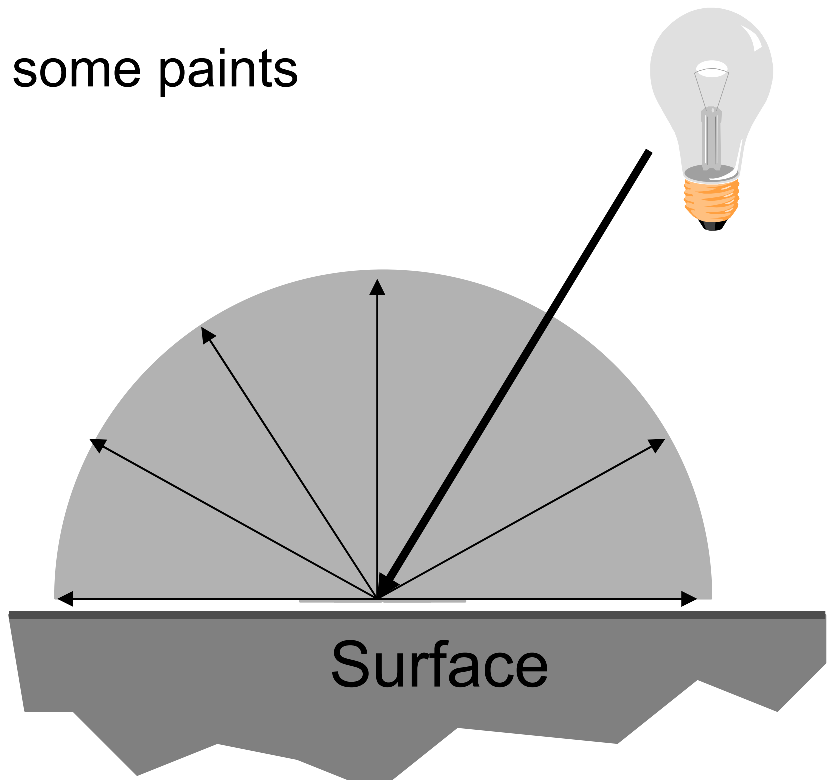
- The amount of light received by a surface depends on incoming angle
  - Bigger at normal incidence
    - Similar to Winter/Summer difference
- By how much?
  - Cos θ law
  - Dot product with normal
  - This term is sometimes included in the BRDF, sometimes not



Ideal Diffuse Reflectance

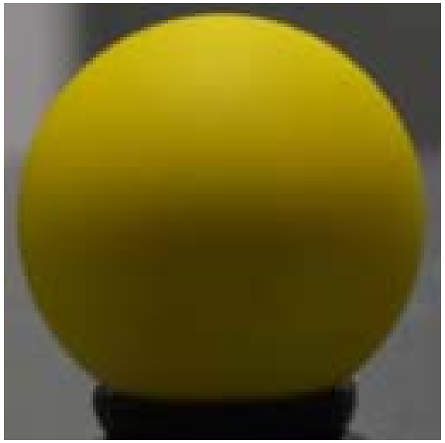
- Assume surface reflects equally in all directions.
- An ideal diffuse surface is, at the microscopic level, a very rough surface.
  - Example: chalk, clay, some paints

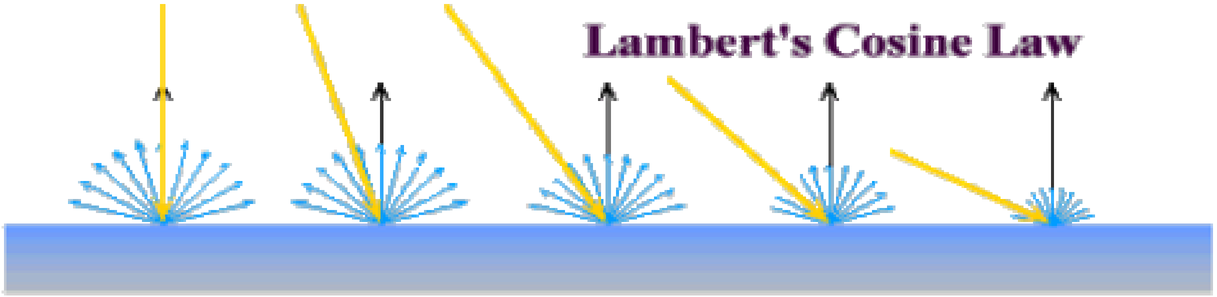




Ideal Diffuse Reflectance

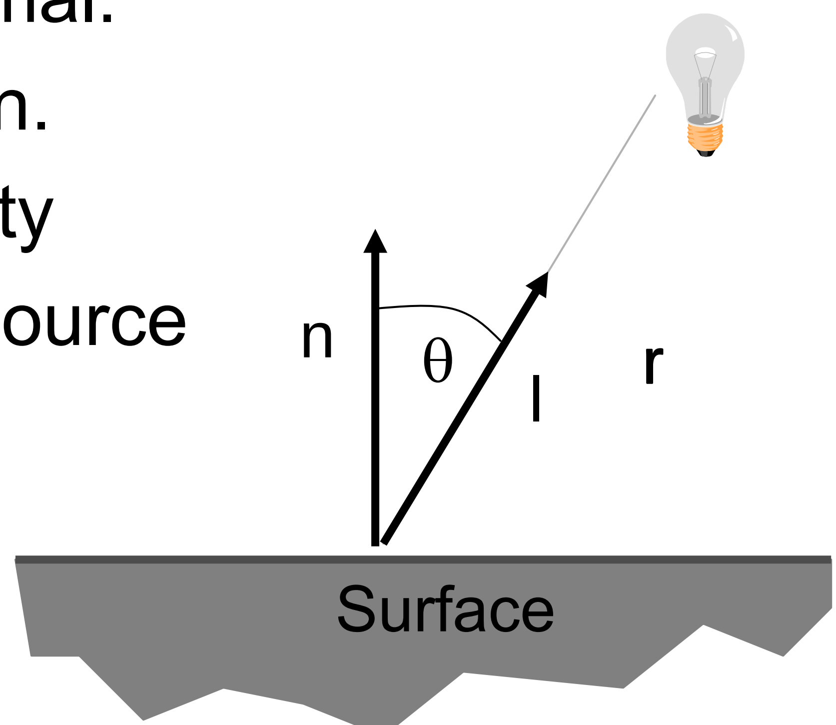
- Ideal diffuse reflectors reflect light according to Lambert's cosine law.





Ideal Diffuse Reflectance

- Single Point Light Source $$L_o = k_d(\mathbf{n} \cdot \mathbf{l})\frac{L_i}{r^2}$$
  - k_d: diffuse coefficient.
  - **n**: Surface normal.
  - **l**: Light direction.
  - L_i: Light intensity
  - r: Distance to source

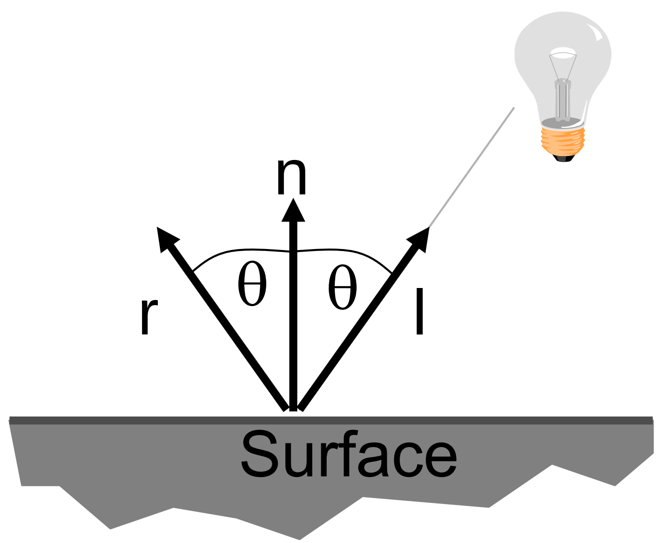


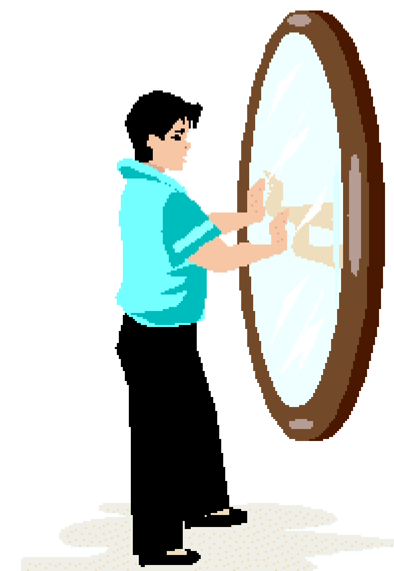
Ideal Diffuse Reflectance – More Details

- If n and l are facing away from each other, n • l becomes negative.
- Using max((n • l),0) makes sure that the result is zero.
  - From now on, we mean max() when we write •.

Ideal Specular Reflectance

- Reflection is only at mirror angle.
  - View dependent
  - Microscopic surface elements are usually oriented in the same direction as the surface itself.
  - Examples: mirrors, highly polished metals.



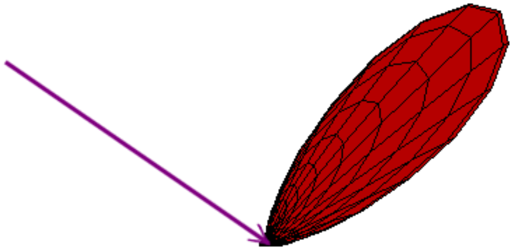


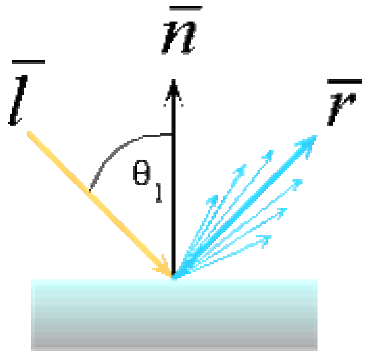
Non-ideal Reflectors

- Real materials tend to deviate significantly from ideal mirror reflectors.
- Highlight is blurry
- They are not ideal diffuse surfaces either …

Non-ideal Reflectors

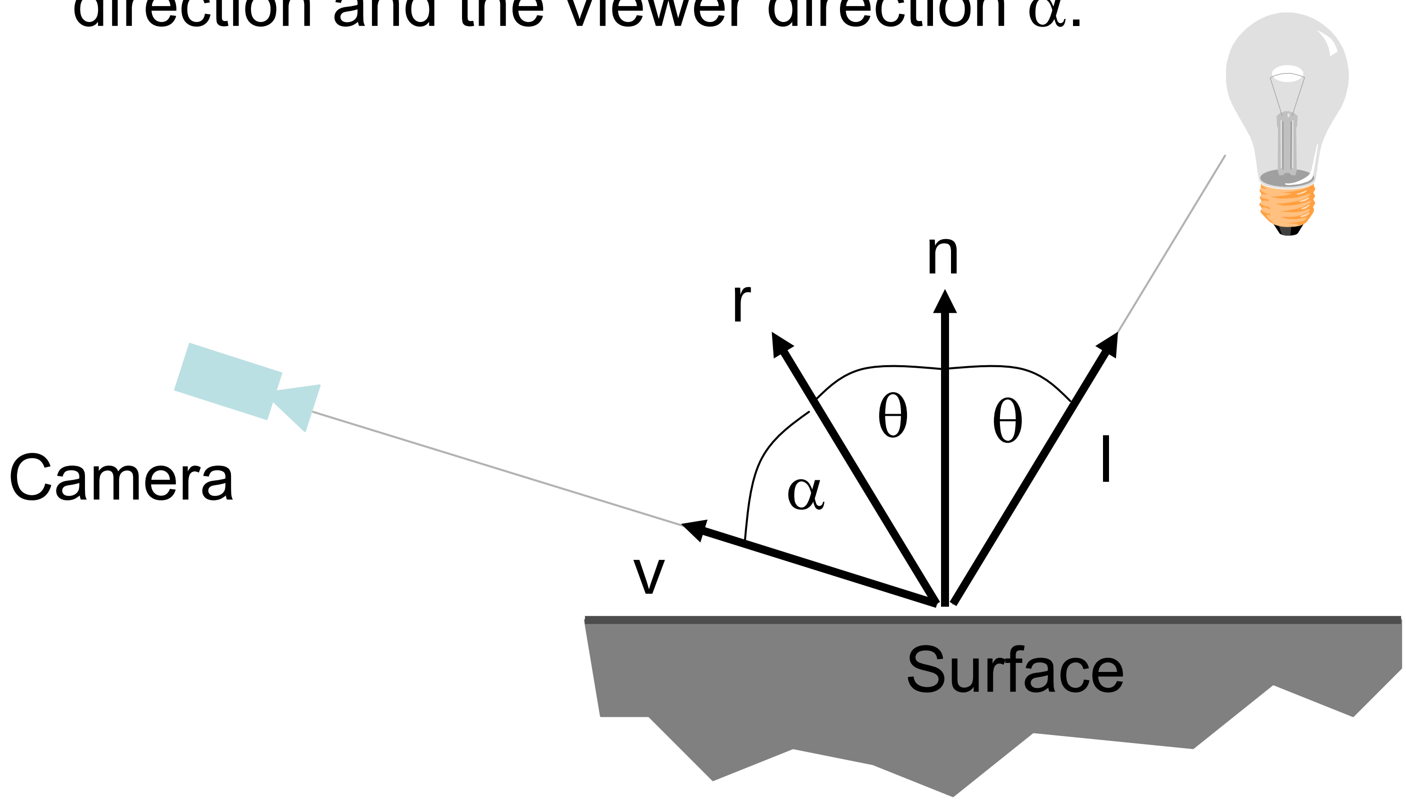
- Simple Empirical Model:
  - We expect most of the reflected light to travel in the direction of the ideal ray.
  - However, because of microscopic surface variations we might expect some of the light to be reflected just slightly offset from the ideal reflected ray.
  - As we move farther and farther, in the angular sense, from the reflected ray we expect to see less light reflected.





The Phong Model

- How much light is reflected?
  - Depends on the angle between the ideal reflection direction and the viewer direction α.

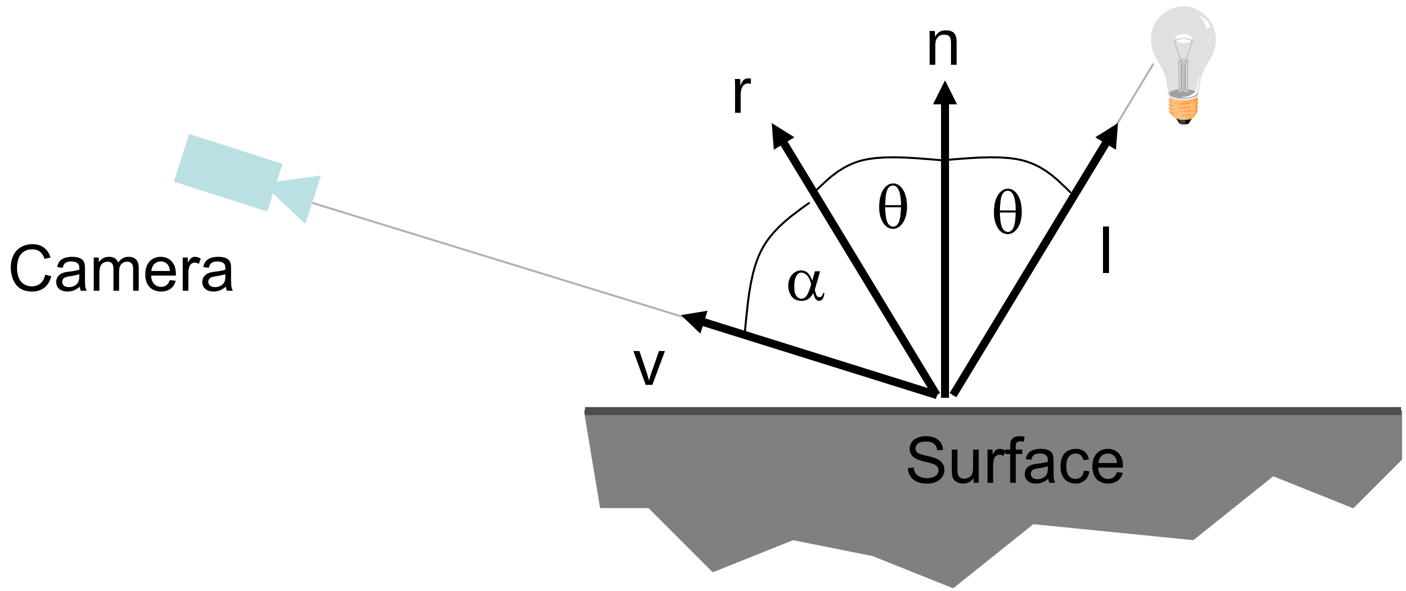


The Phong Model

- Parameters
  - k_s: specular reflection coefficient
  - q : specular reflection exponent

$$L_o = k_s (\cos\alpha)^q \frac{L_i}{r^2}$$

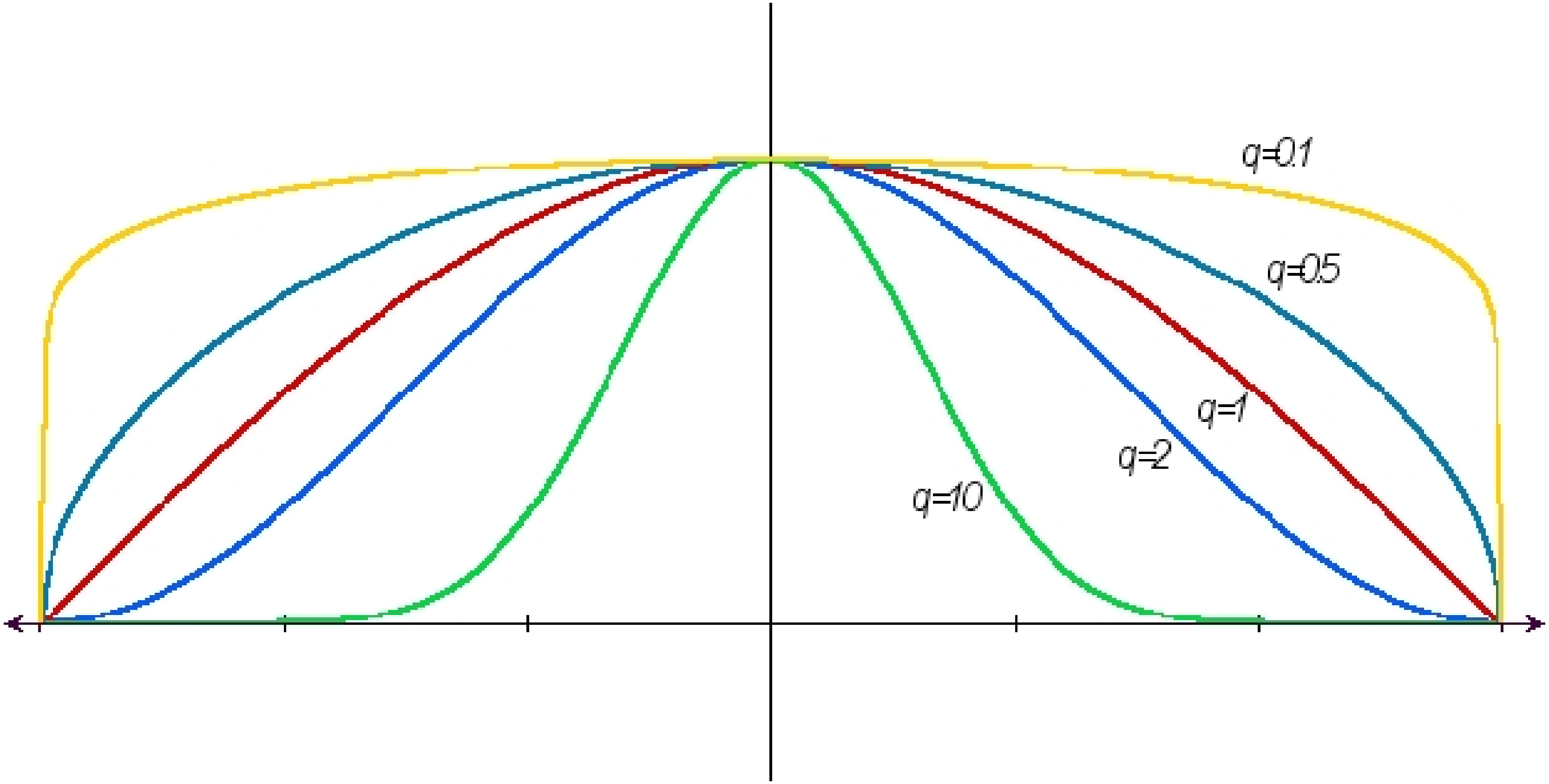
$$L_o = k_s (\mathbf{v}\cdot\mathbf{r})^q \frac{L_i}{r^2}$$



Demo

The Phong Model

- Effect of the q coefficient



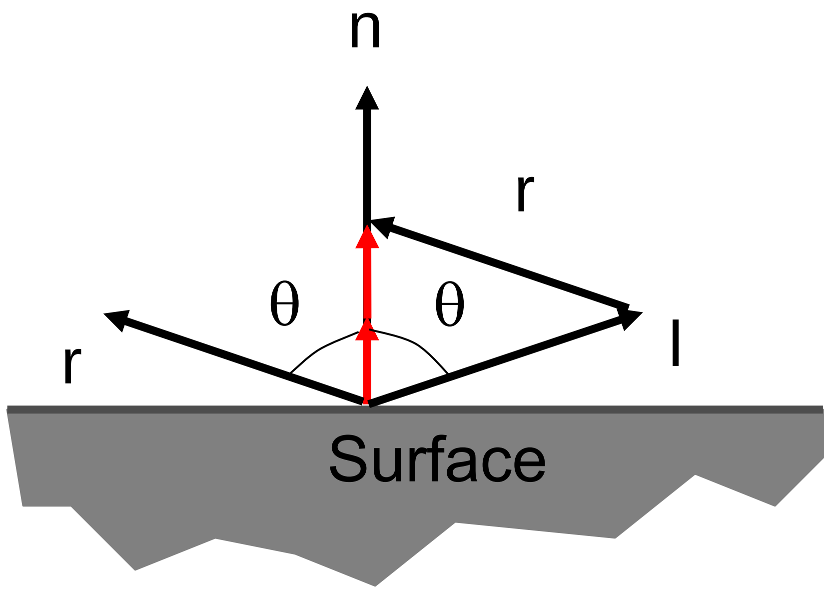
How to get the mirror direction?

$$\mathbf{r} + \mathbf{l} = 2\ \cos\theta\,\mathbf{n}$$

$$\mathbf{r} = 2(\mathbf{n}\cdot\mathbf{l})\mathbf{n} - \mathbf{l}$$

$$L_o = k_s(\mathbf{v}\cdot\mathbf{r})^q \frac{L_i}{r^2} =$$

$$= k_s(\mathbf{v}\cdot(2(\mathbf{n}\cdot\mathbf{l})\mathbf{n} - \mathbf{l}))^q \frac{L_i}{r^2}$$

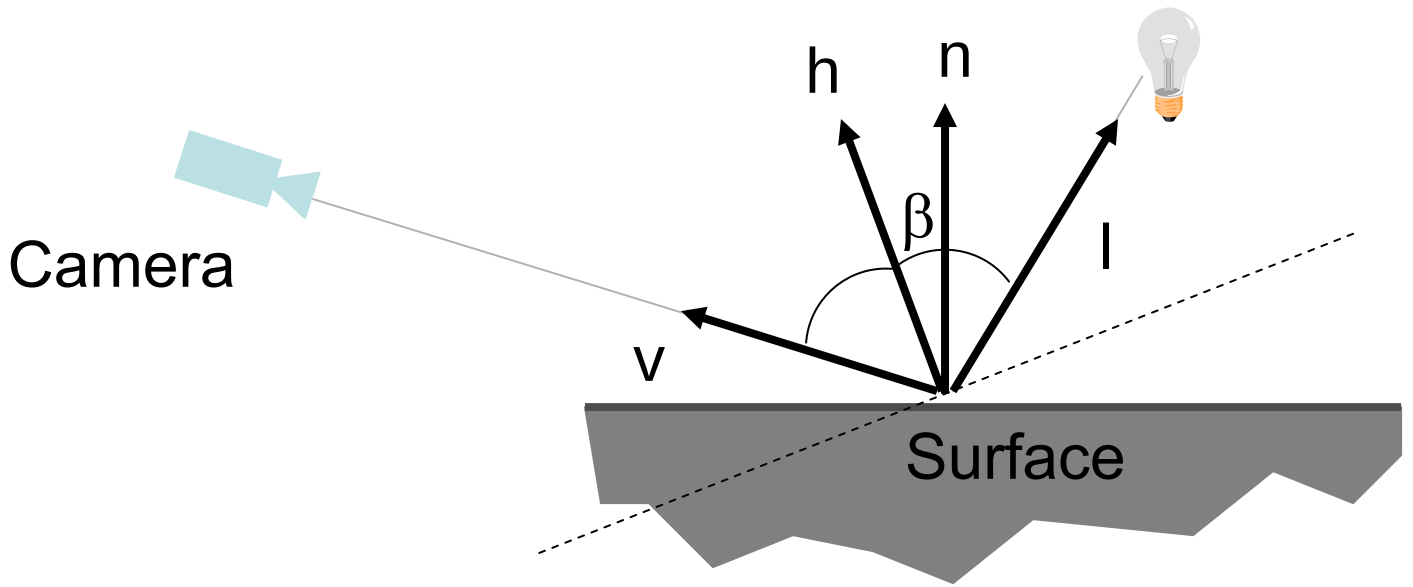


Blinn-Torrance Variation

- Uses the halfway vector **h** between **l** and **v**.

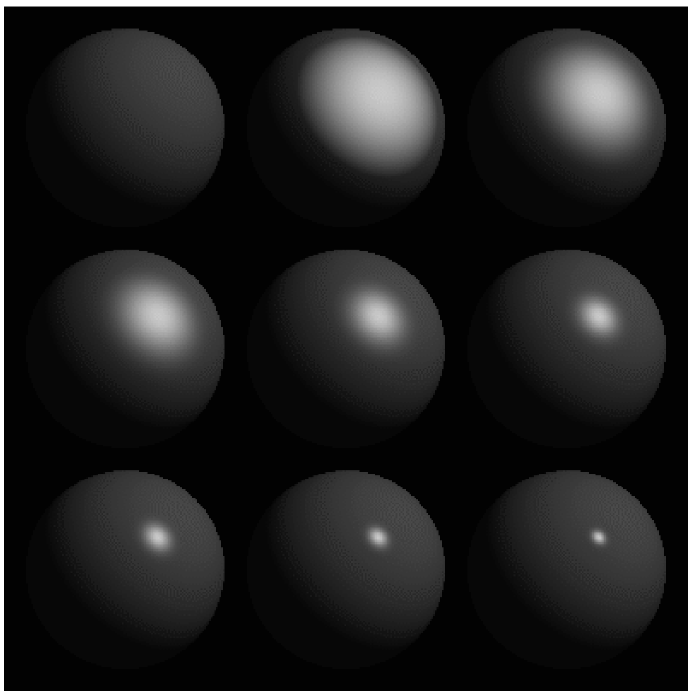
$$\mathbf{h} = \frac{\mathbf{l} + \mathbf{v}}{\|\mathbf{l} + \mathbf{v}\|}$$

$$L_o = k_s (\cos\beta)^q \frac{L_i}{r^2} = k_s (\mathbf{n} \cdot \mathbf{h})^q \frac{L_i}{r^2}$$

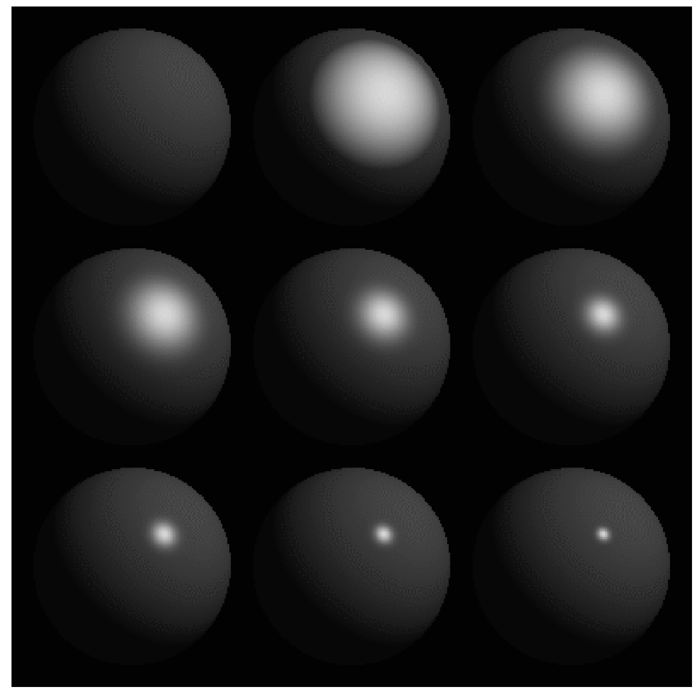


Phong Examples

- The following spheres illustrate specular reflections as the direction of the light source and the coefficient of shininess is varied.



Phong



Blinn-Torrance

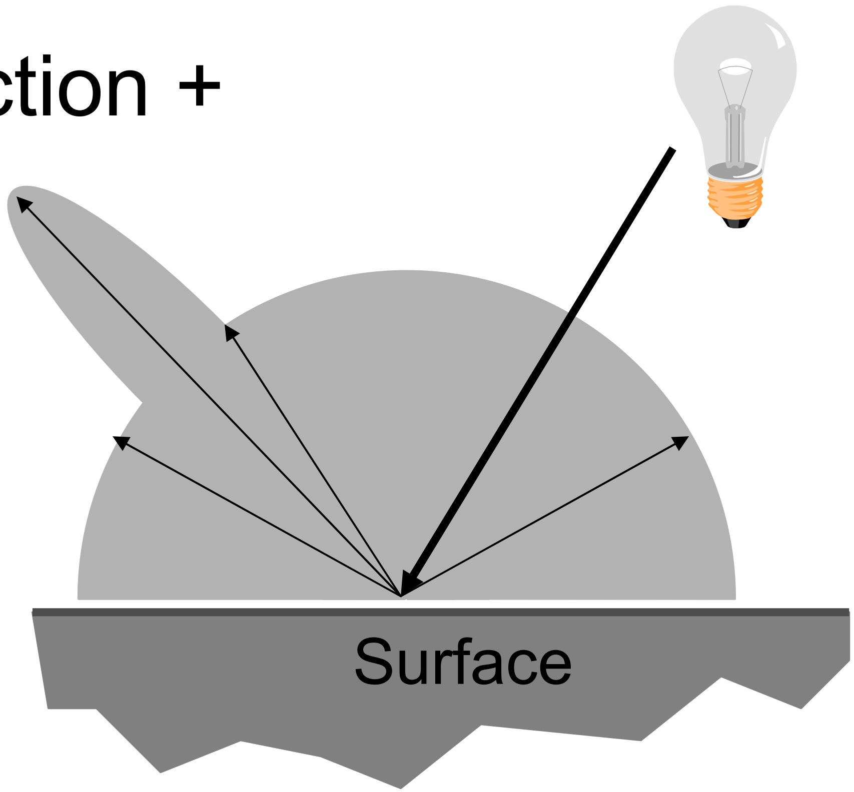
The Phong Model

- Sum of three components:

  diffuse reflection +

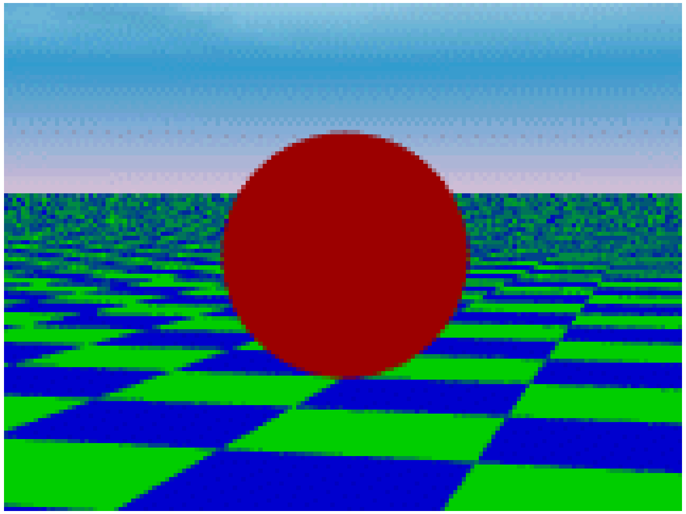
  specular reflection +

  “ambient”.



Ambient Illumination

- Represents the reflection of all indirect illumination.
- This is a total hack!
- Avoids the complexity of global illumination.

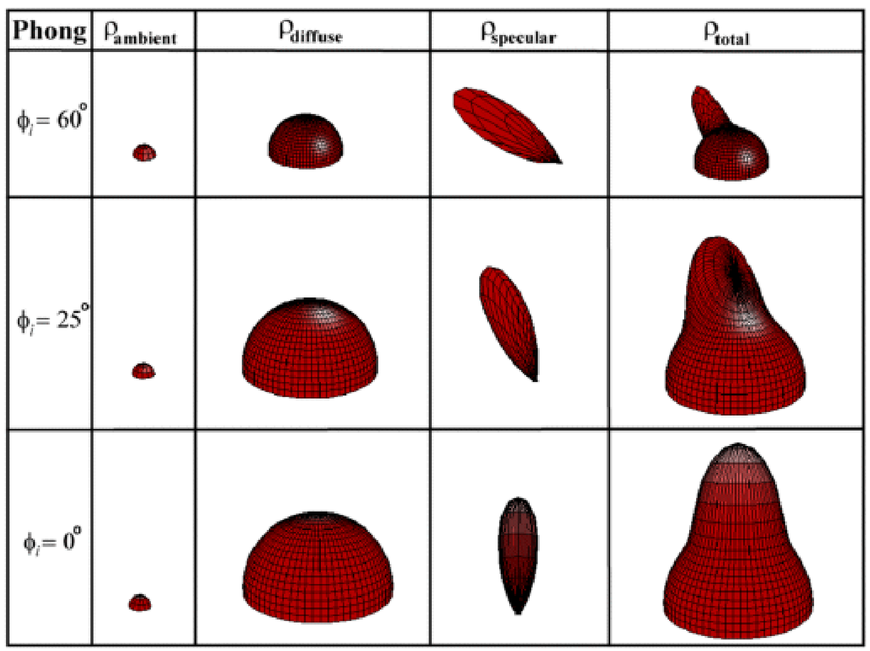


$$L(\omega_r) = k_a$$

Putting it all together

- Phong Illumination Model

$$L_o = k_a + \left(k_d (\mathbf{n} \cdot \mathbf{l}) + k_s (\mathbf{v} \cdot \mathbf{r})^q \right) \frac{L_i}{r^2}$$

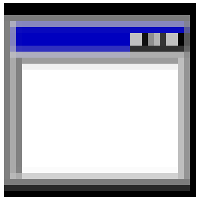


Adding color

- Diffuse coefficients:
  - $k_{d\text{-}red}, k_{d\text{-}green}, k_{d\text{-}blue}$
- Specular coefficients:
  - $k_{s\text{-}red}, k_{s\text{-}green}, k_{s\text{-}blue}$
- Specular exponent: q

Materials

- Demo



lightmaterial.exe

Shaders

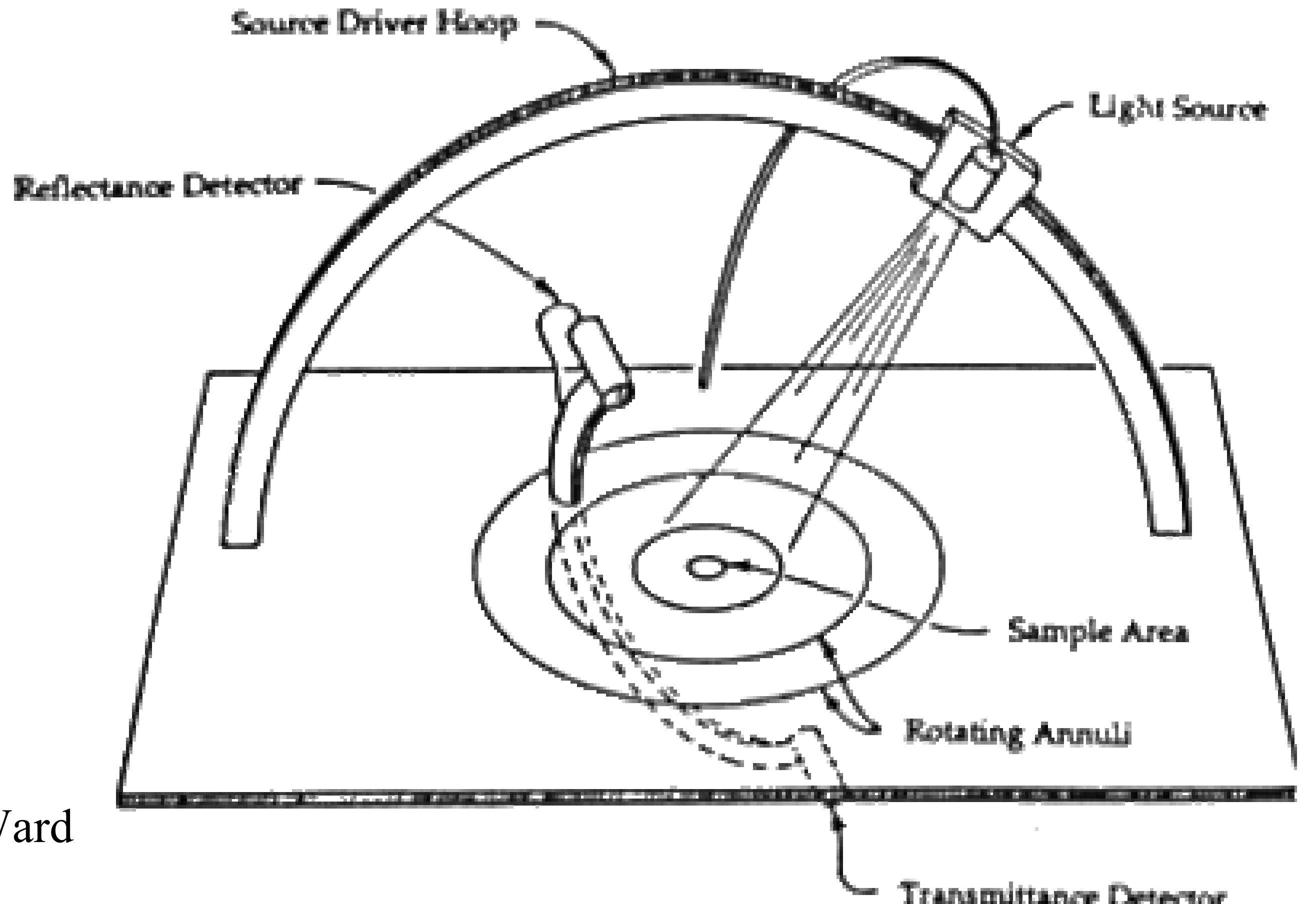
- Functions executed when light interacts with a surface
- Constructor:
  – set shader parameters
- Inputs:
  – Incident radiance
  – Incident & reflected light directions
  – surface tangent (anisotropic shaders only)
- Output:
  – Reflected radiance

BRDFs in the movie industry

- http://www.virtualcinematography.org/publications/acrobat/BRDF-s2003.pdf
- For the Matrix movies
- Clothes of the agent Smith are CG, with measured BRDF

How do we obtain BRDFs?

- Gonioreflectometer
  - 4 degrees of freedom



Source: Greg Ward

BRDFs in the movie industry

- http://www.virtualcinematography.org/publications/acrobat/BRDF-s2003.pdf
- For the Matrix movies

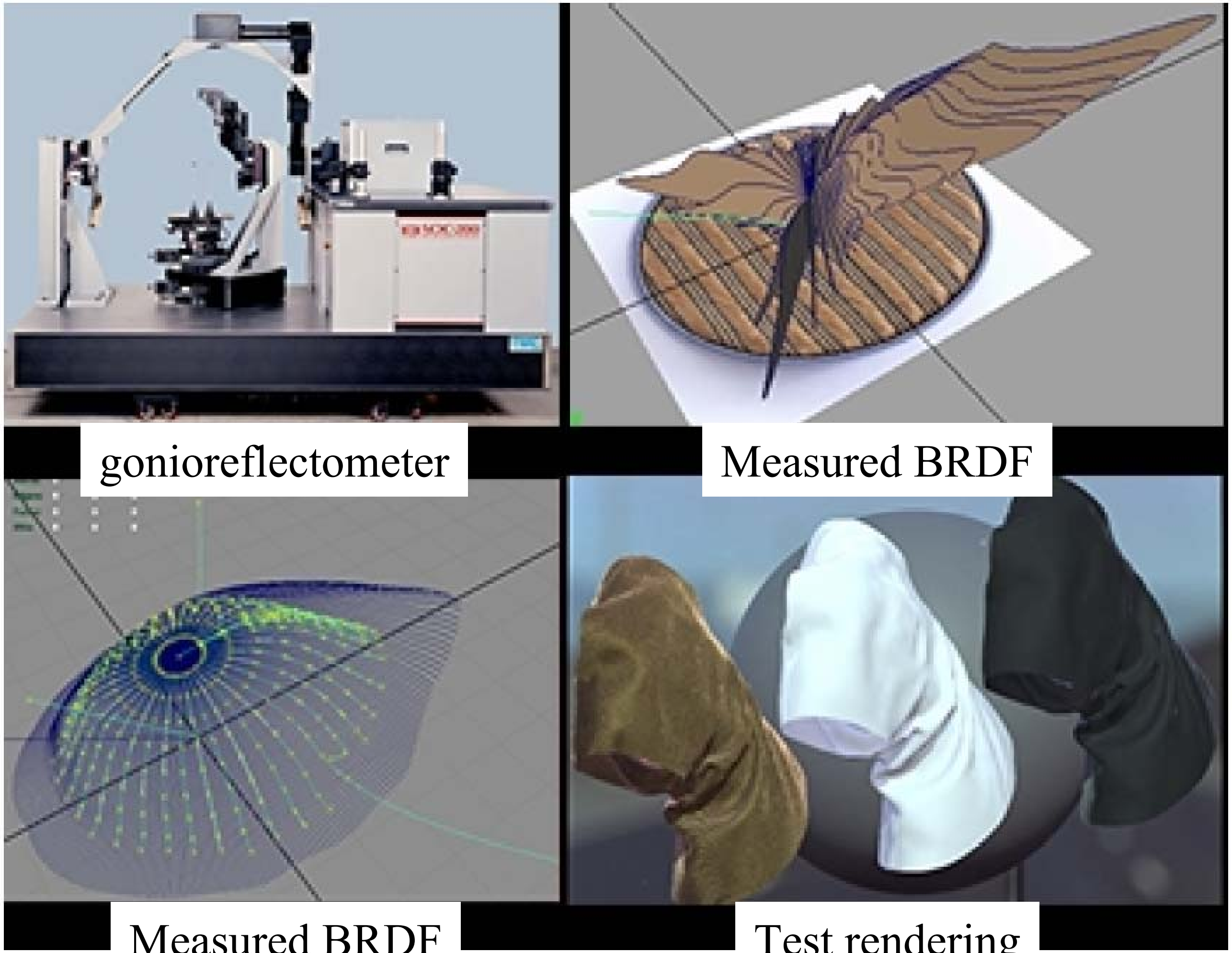
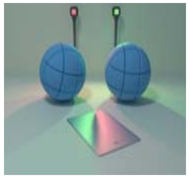


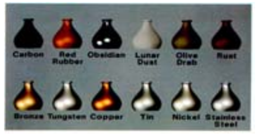
Photo CG Photo CG

BRDF Models

- Phenomenological
  - Phong [75]
    - Blinn-Phong [77]
  - Ward [92]
  - Lafortune et al. [97]
  - Ashikhmin et al. [00]
- Physical
  - Cook-Torrance [81]
  - He et al. [91]

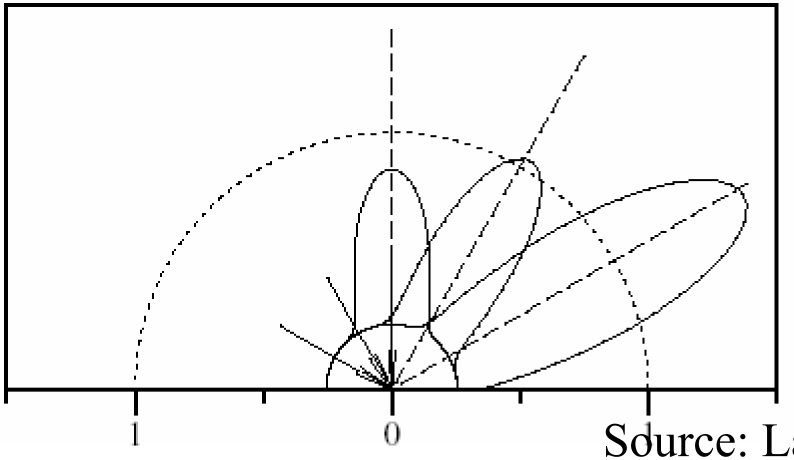


Roughly increasing computation time



Fresnel Reflection

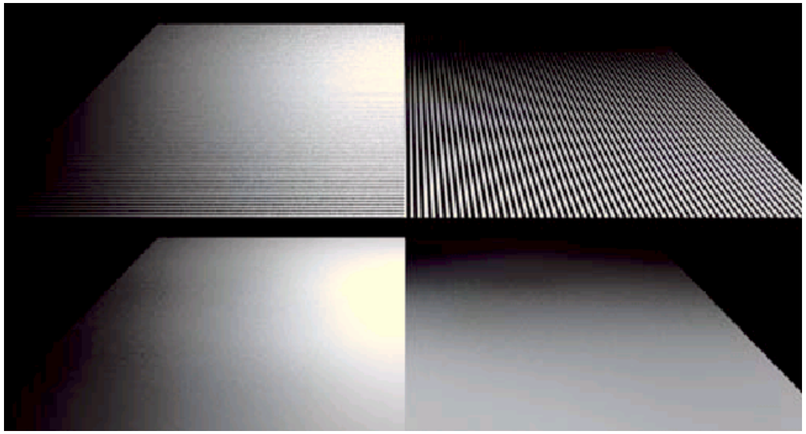
- Increasing specularity near grazing angles



Source: Lafortune et al. 97

Anisotropic BRDFs

- Surfaces with strongly oriented microgeometry elements
- Examples:
  - brushed metals,
  - hair, fur, cloth, velvet



Source: Westin et.al 92

Viewing Transformation

Modeling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

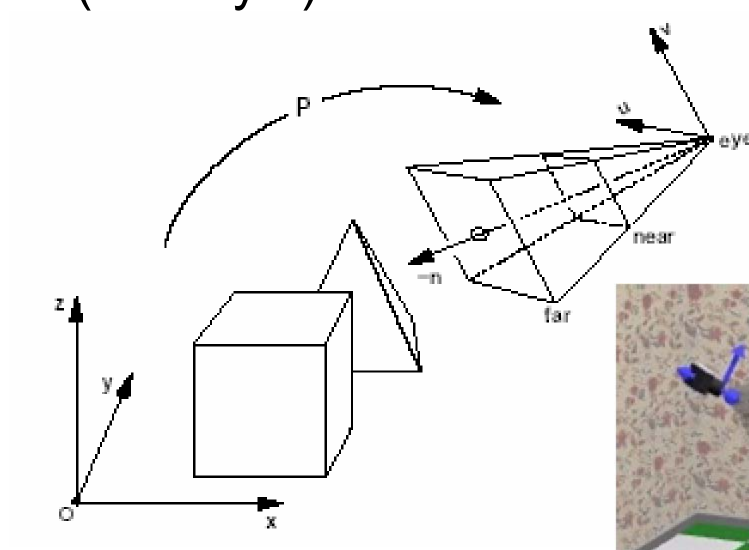
Clipping

Projection (to Screen Space)

Scan Conversion (Rasterization)

Visibility / Display

- Maps world space to eye space
- Viewing position is transformed to origin & direction is oriented along some axis (usually *z*)



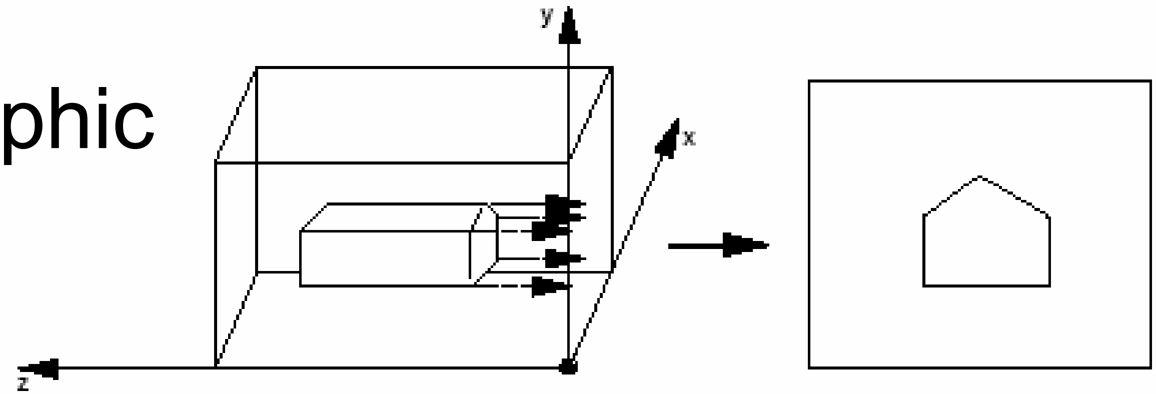
Eye space

World space

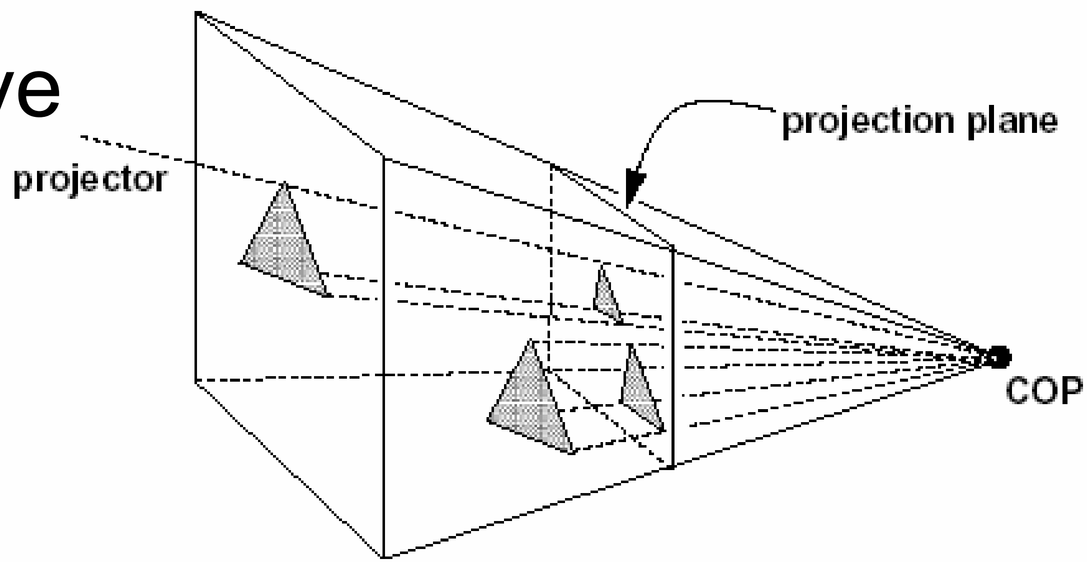
Durand/Cutler MIT

Orthographic vs. Perspective

- Orthographic

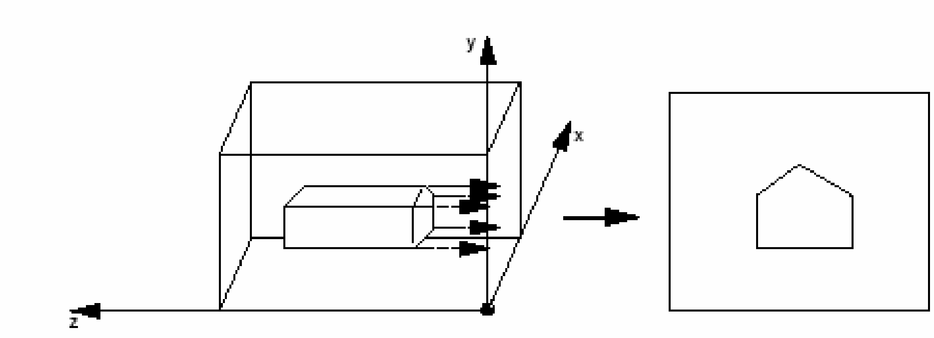


- Perspective



Simple Orthographic Projection

- Project all points along the *z* axis to the *z* = 0 plane

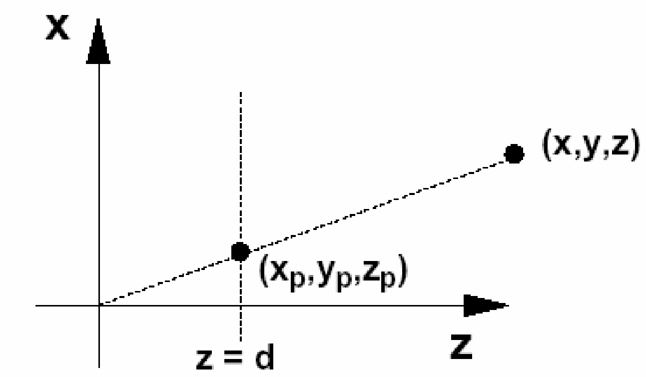


$$\begin{pmatrix} x \\ y \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Simple Perspective Projection

- Project all points to the *z* = *d* plane, eyepoint at the origin:

$$
\begin{aligned}
x_p &= \frac{d \cdot x}{z} = \frac{x}{z/d} \\
y_p &= \frac{d \cdot y}{z} = \frac{y}{z/d} \\
z_p &= d
\end{aligned}
$$



homogenize

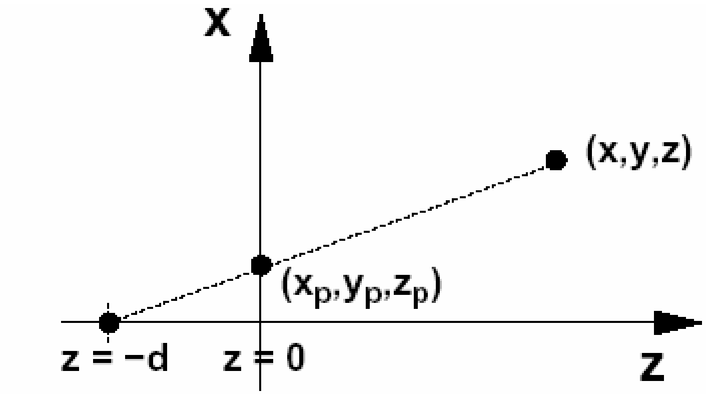
$$
\begin{pmatrix} x * d / z \\ y * d / z \\ d \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ z / d \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}
$$

Alternate Perspective Projection

- Project all points to the *z* = 0 plane, eyepoint at the (0,0,-*d*):

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

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



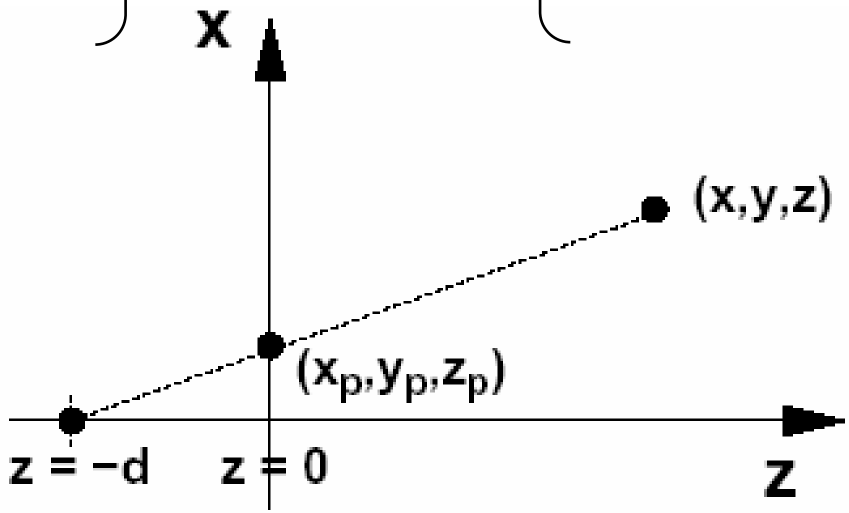
homogenize

$$\begin{pmatrix} x * d / (z+d) \\ y * d / (z+d) \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 0 \\ (z+d)/d \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

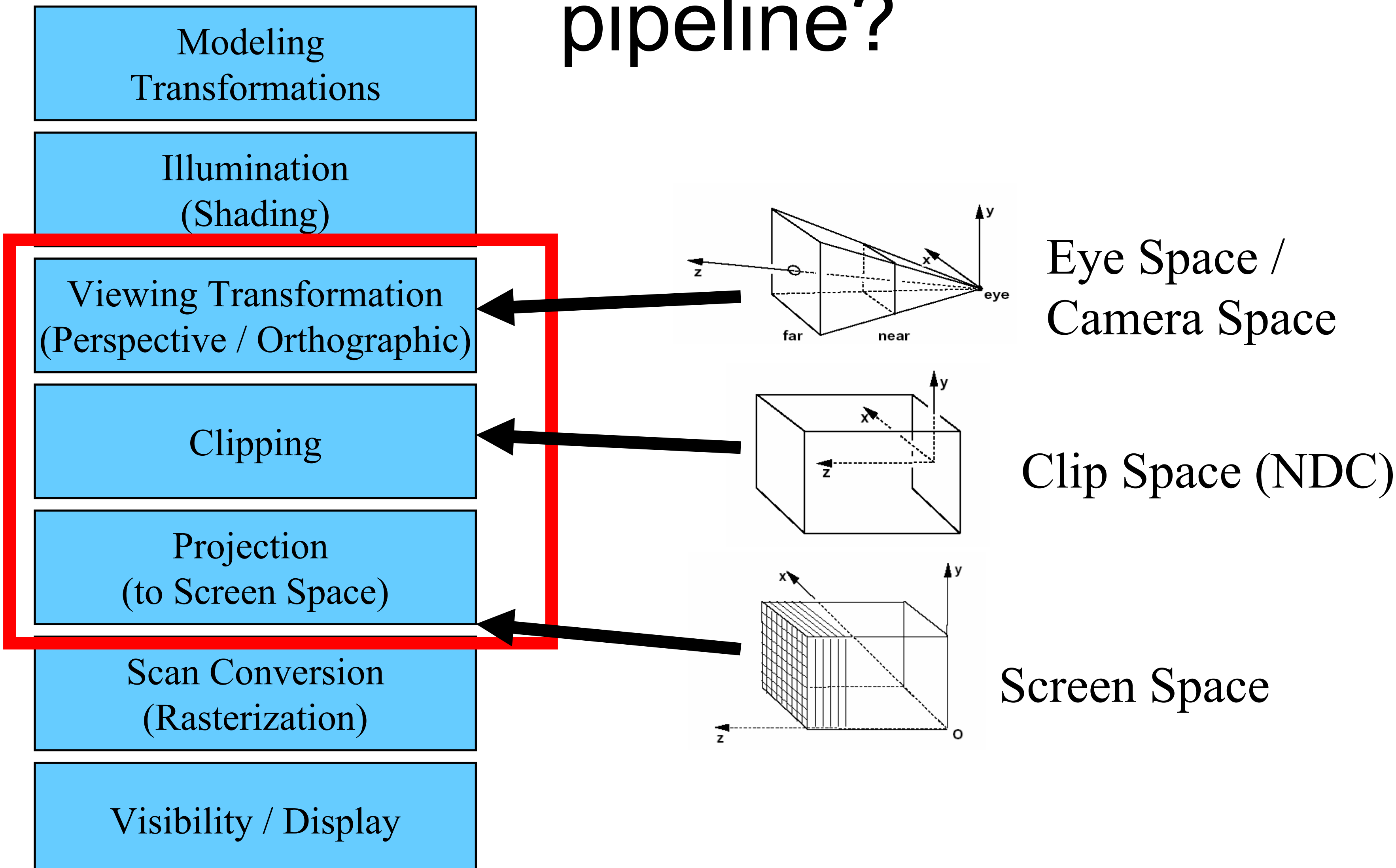
In the limit, as $d \to \infty$

this perspective projection matrix... ...is simply an orthographic projection

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 1 \end{pmatrix} \to \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

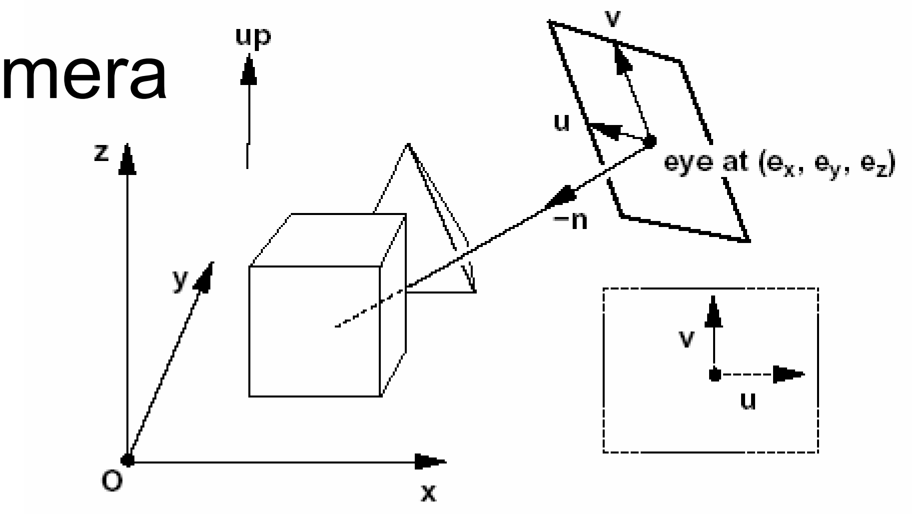


Where are projections in the pipeline?



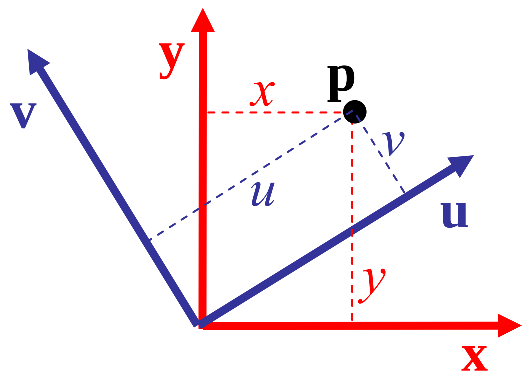
World Space → Eye Space

Positioning the camera



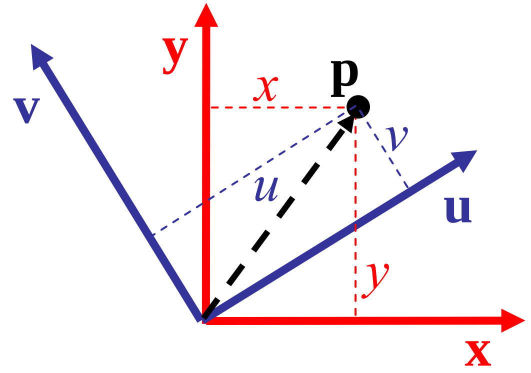
Translation + Change of orthonormal basis

- Given: coordinate frames **xyz** & **uvn,** and point **p** = (*x*,*y*,*z*)
- Find: **p** = (*u*,*v*,*n*)



Change of Orthonormal Basis

$$\begin{bmatrix} u \\ v \\ n \end{bmatrix} = \begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ n_x & n_y & n_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$



where:

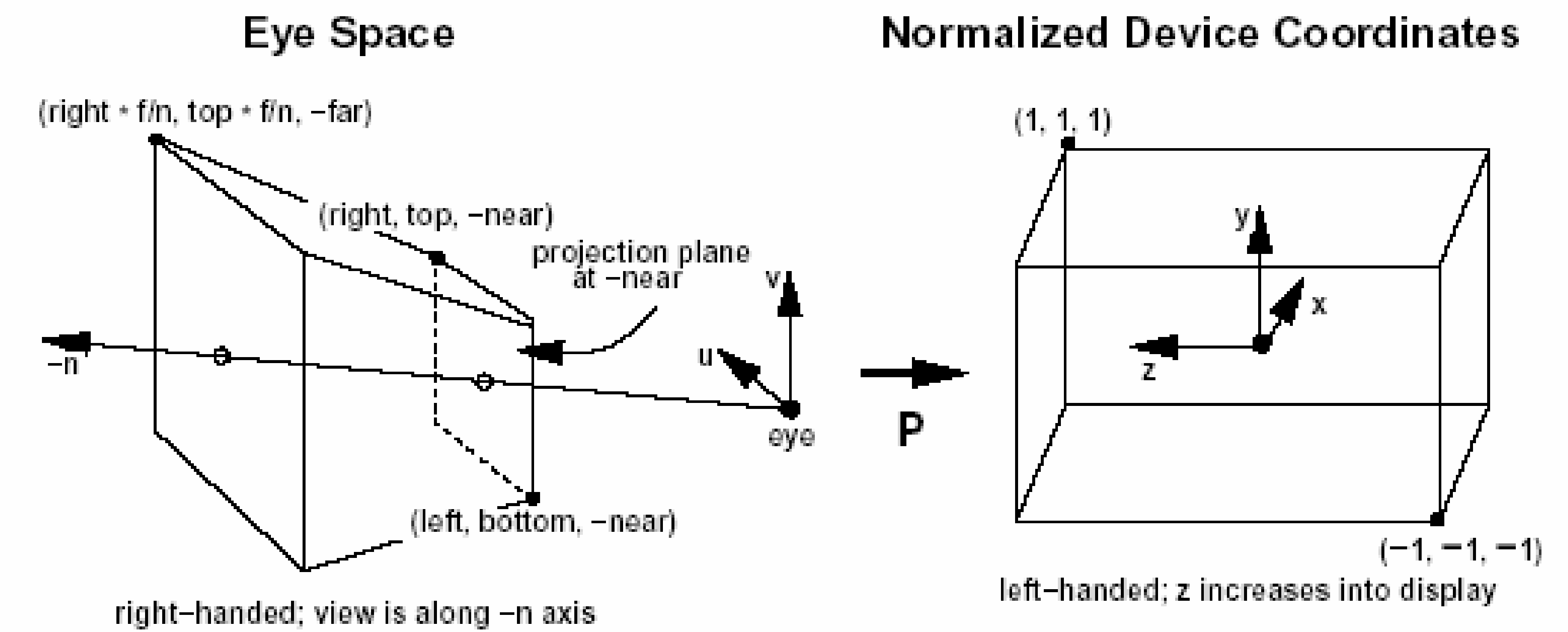
$u_x = \mathbf{x} \,.\, \mathbf{u}$

$u_y = \mathbf{y} \,.\, \mathbf{u}$

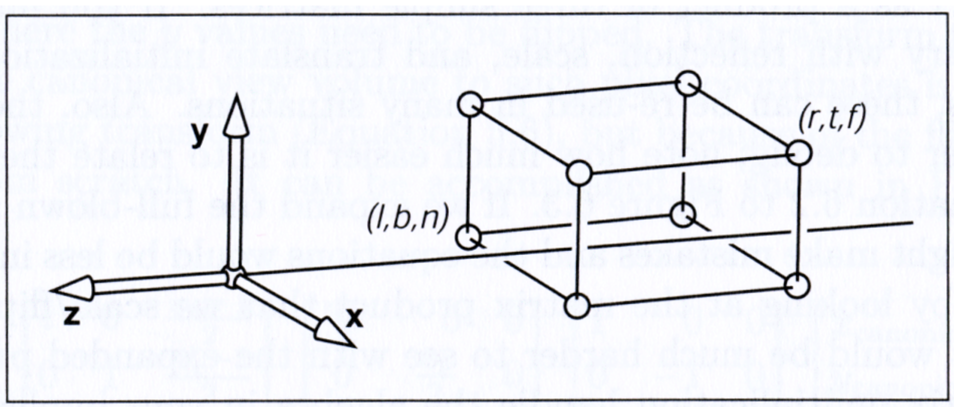
etc.

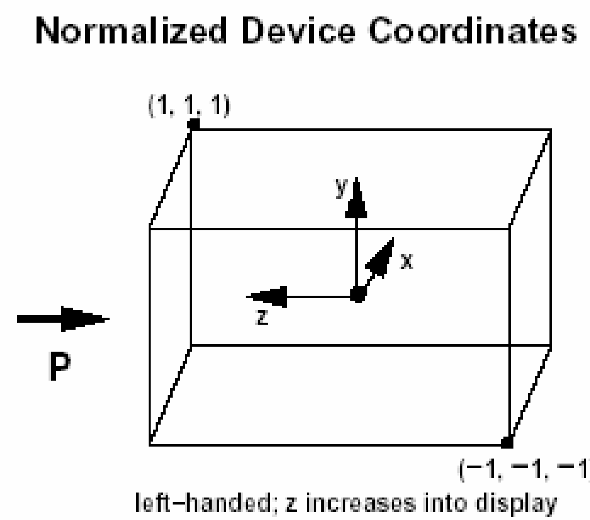
Normalized Device Coordinates

- Clipping is more efficient in a rectangular, axis-aligned volume: (-1,-1,-1) → (1,1,1) *OR* (0,0,0) → (1,1,1)



Canonical Orthographic Projection

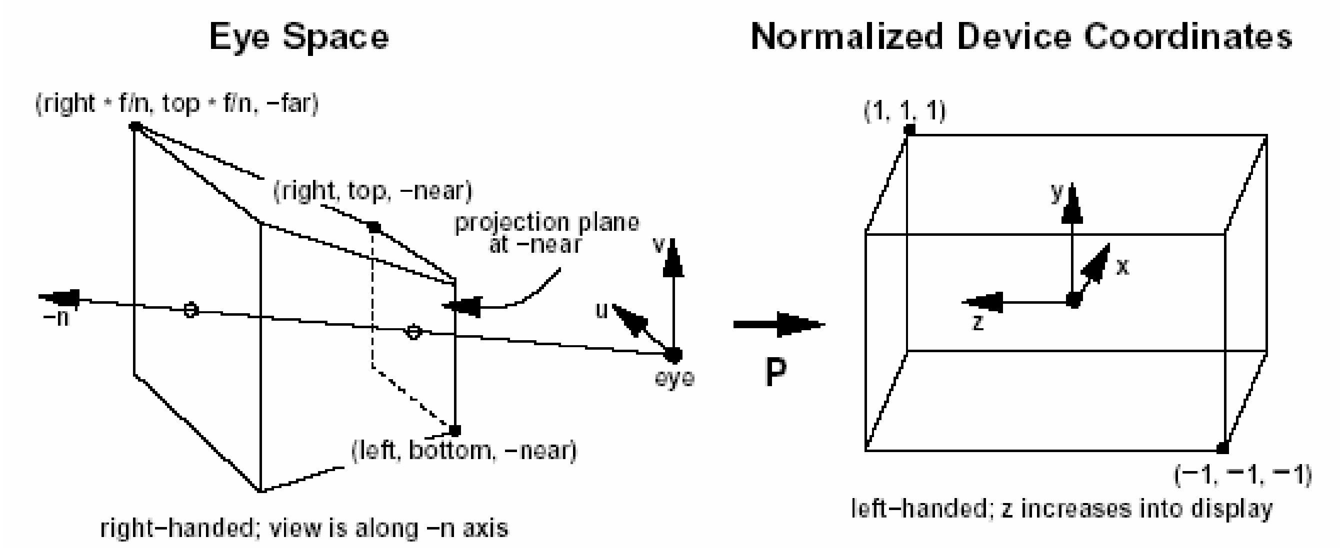




Orthographic

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{\text{right}-\text{left}} & 0 & 0 & \frac{-(\text{right}+\text{left})}{\text{right}-\text{left}} \\ 0 & \frac{2}{\text{bottom}-\text{top}} & 0 & \frac{-(\text{bottom}+\text{top})}{\text{bottom}-\text{top}} \\ 0 & 0 & \frac{2}{\text{far}-\text{near}} & \frac{-(\text{far}+\text{near})}{\text{far}-\text{near}} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Canonical Perspective Projection

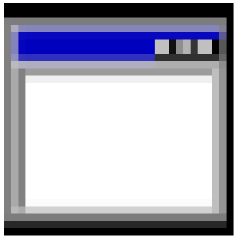


Perspective

$$
\begin{bmatrix} x'w \\ y'w \\ z'w \\ w \end{bmatrix} =
\begin{bmatrix}
\dfrac{2 \cdot \mathrm{near}}{\mathrm{right} - \mathrm{left}} & 0 & \dfrac{-(\mathrm{right} + \mathrm{left})}{\mathrm{right} - \mathrm{left}} & 0 \\
0 & \dfrac{2 \cdot \mathrm{near}}{\mathrm{bottom} - \mathrm{top}} & \dfrac{-(\mathrm{bottom} + \mathrm{top})}{\mathrm{bottom} - \mathrm{top}} & 0 \\
0 & 0 & \dfrac{\mathrm{far} + \mathrm{near}}{\mathrm{far} - \mathrm{near}} & \dfrac{-2 \cdot \mathrm{near} \cdot \mathrm{far}}{\mathrm{far} - \mathrm{near}} \\
0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}
$$

Viewing parameters

- Demo

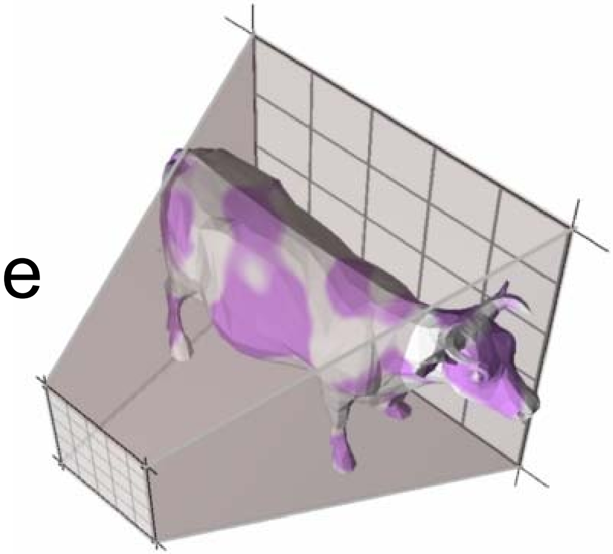


projection.exe

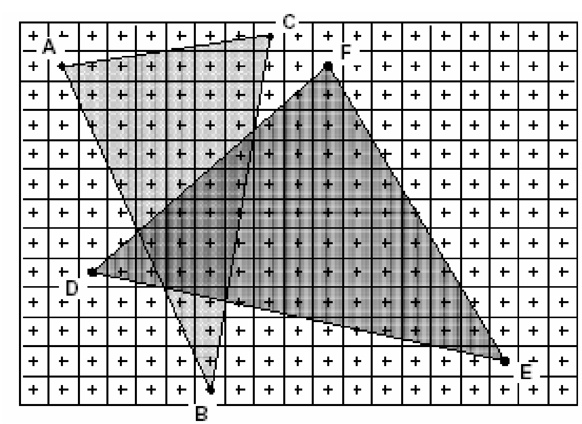
Clipping & Line Rasterization

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- **Clipping**
- Projection (to Screen Space)
- **Scan Conversion (Rasterization)**
- Visibility / Display

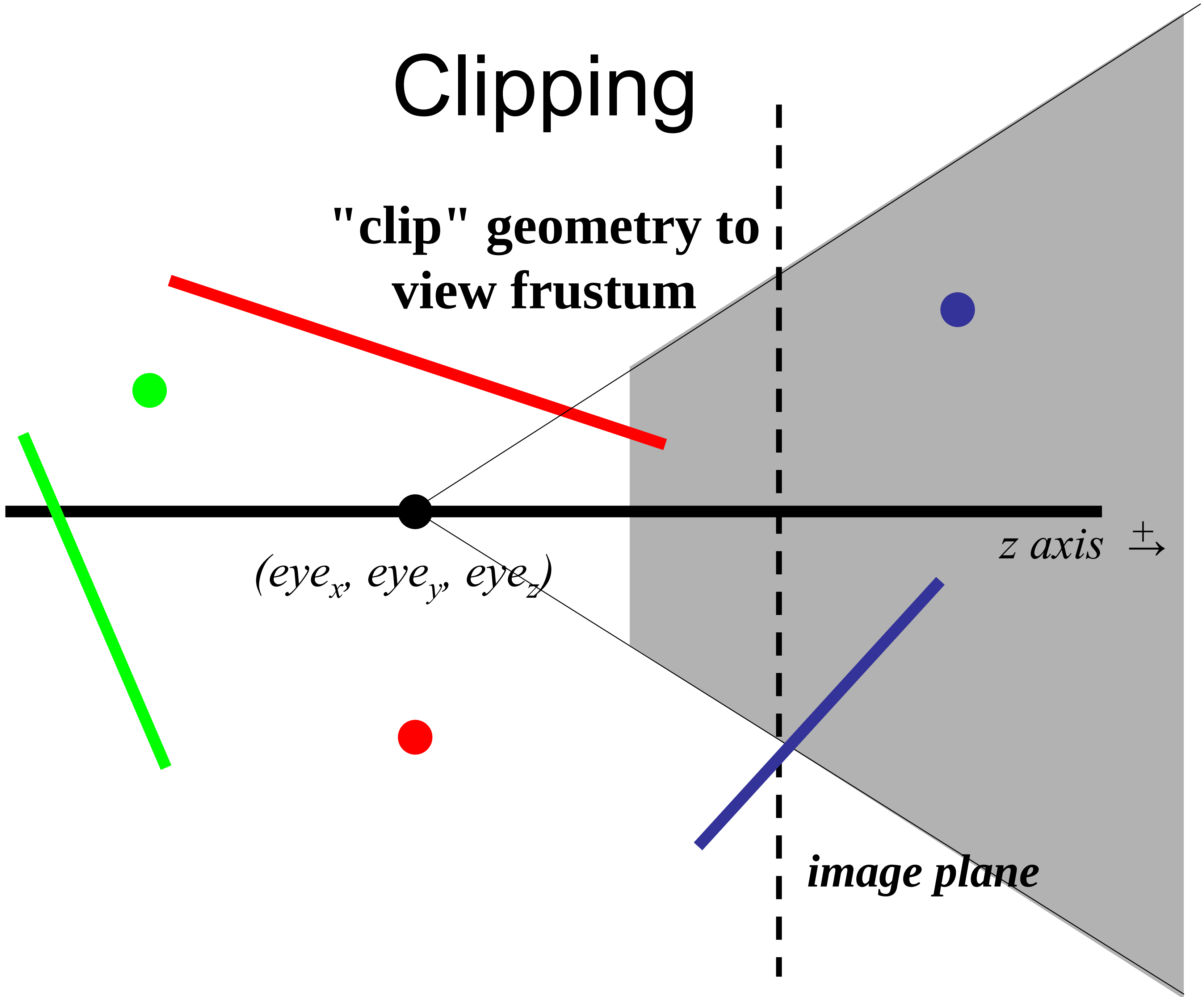
- Portions of the object outside the view frustum are removed



- Rasterize objects into pixels



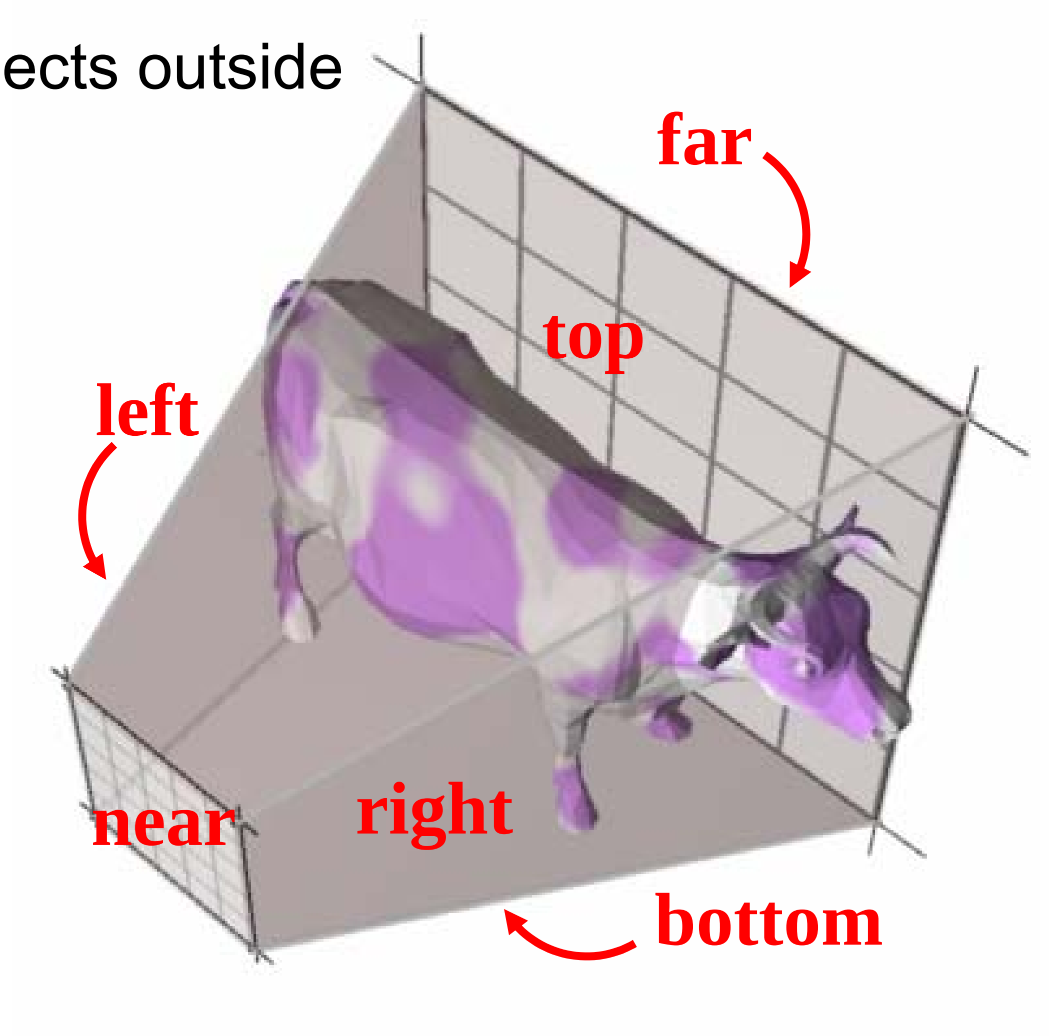
Clipping



"clip" geometry to
view frustum
(eye_x, eye_y, eye_z)
z axis +
image plane

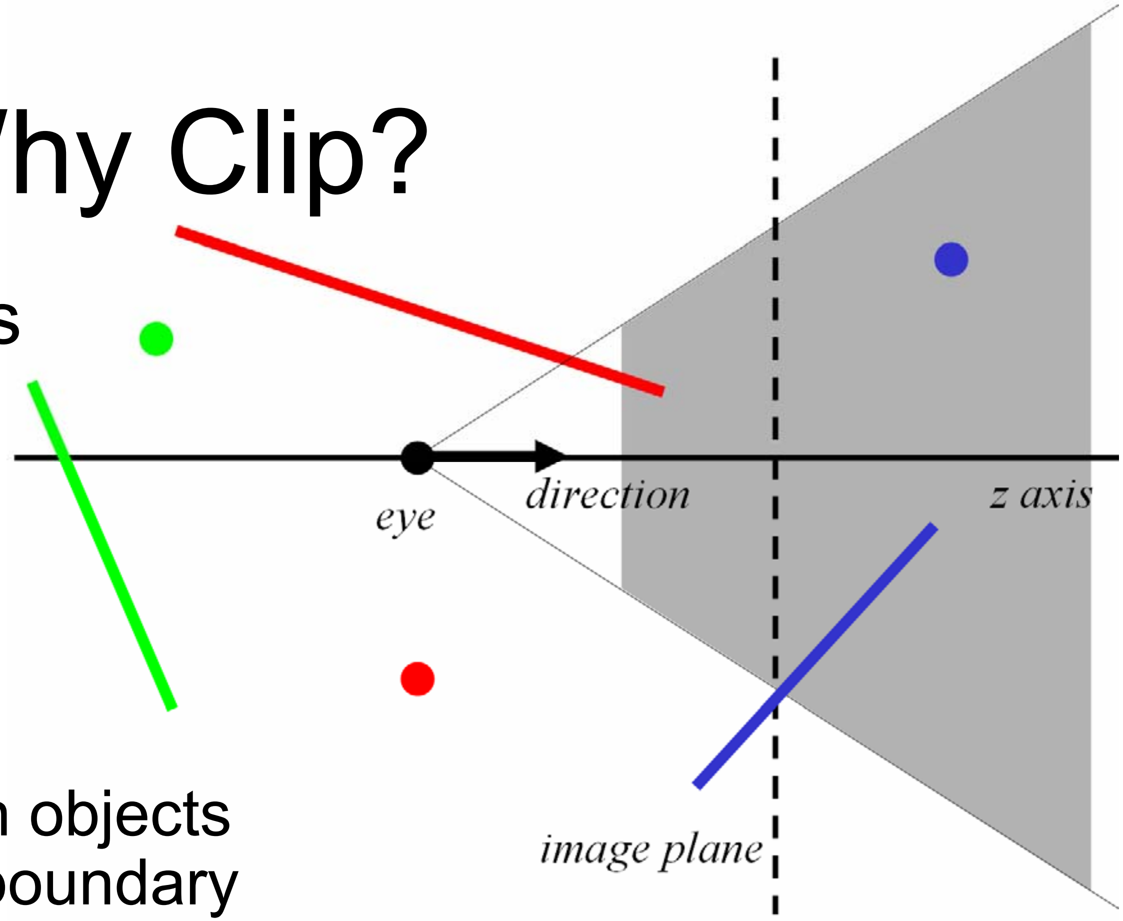
Clipping

- Eliminate portions of objects outside the viewing frustum
- View Frustum
  - boundaries of the image plane projected in 3D
  - a near & far clipping plane
- User may define additional clipping planes



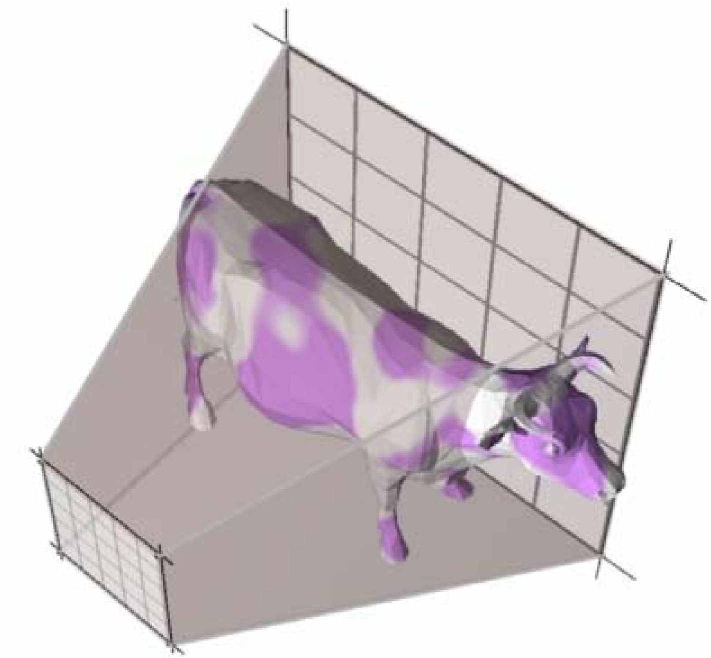
Why Clip?

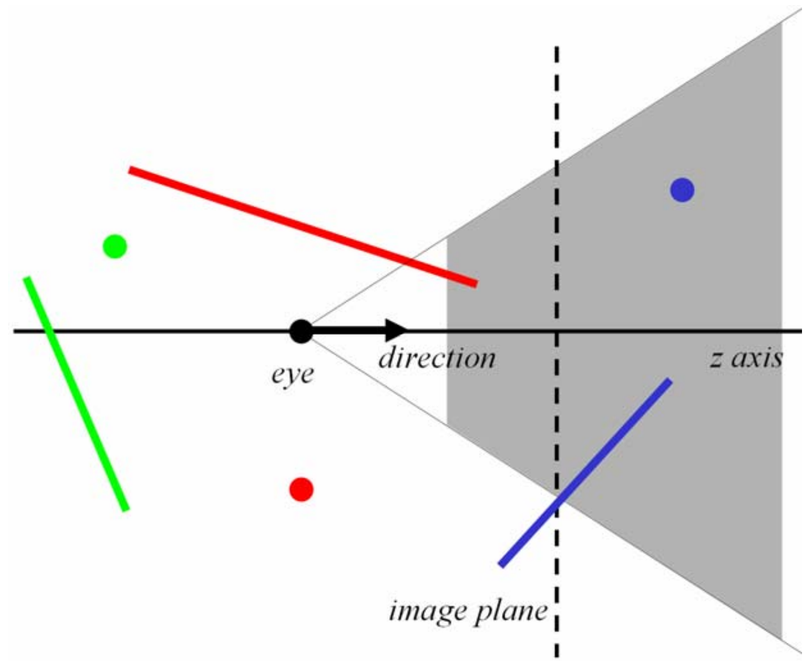
- Avoid degeneracies
  - Don't draw stuff behind the eye
  - Avoid division by 0 and overflow
- Efficiency
  - Don't waste time on objects outside the image boundary
- Other graphics applications (often non-convex)
  - Hidden-surface removal, Shadows, Picking, Binning, CSG (Boolean) operations (2D & 3D)



Clipping Strategies

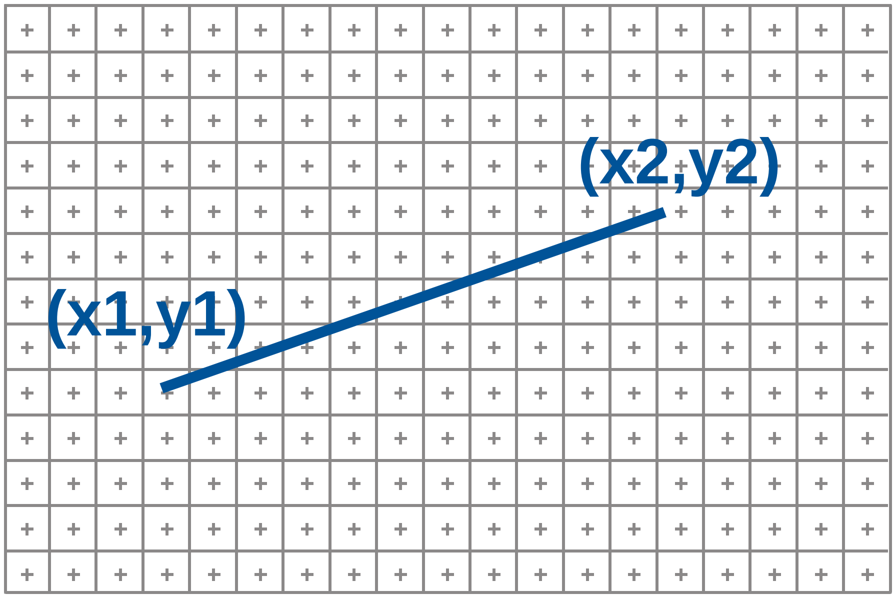
- Don’t clip (and hope for the best)
- Clip on-the-fly during rasterization
- Analytical clipping: alter input geometry





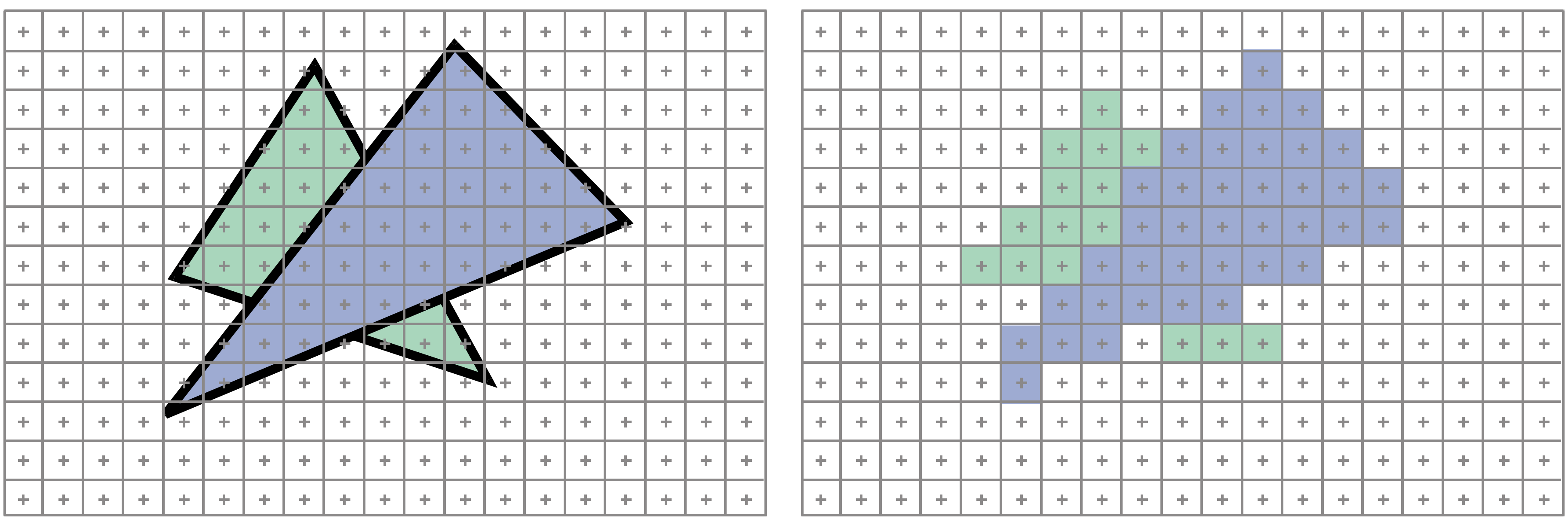
Framebuffer Model

- Raster Display: 2D array of picture elements (pixels)
- Pixels individually set/cleared (greyscale, color)
- Window coordinates: pixels centered at integers



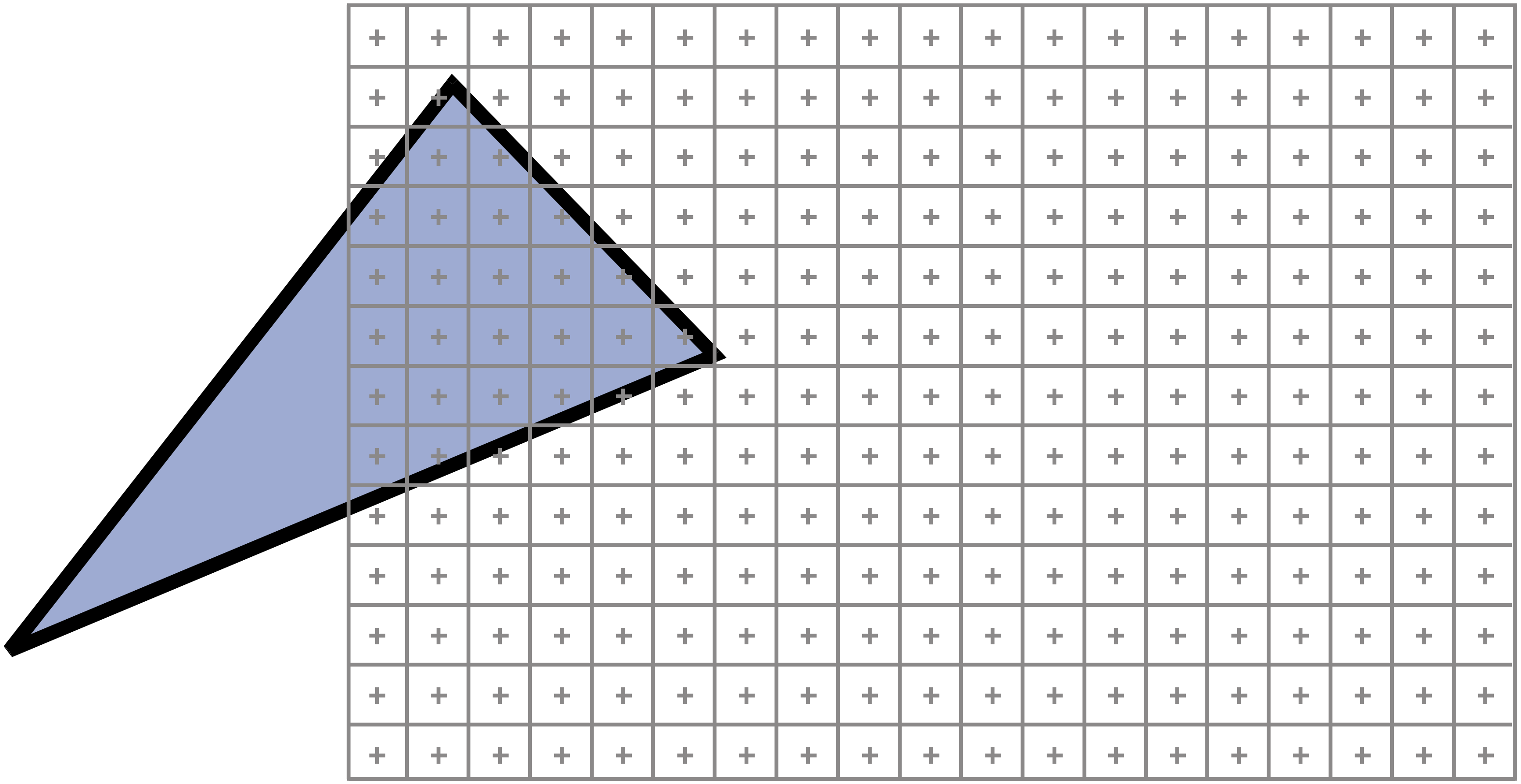
2D Scan Conversion

- Geometric primitives
  (point, line, polygon, circle, polyhedron, sphere...)
- Primitives are continuous; screen is discrete
- Scan Conversion: algorithms for *efficient* generation of the samples comprising this approximation



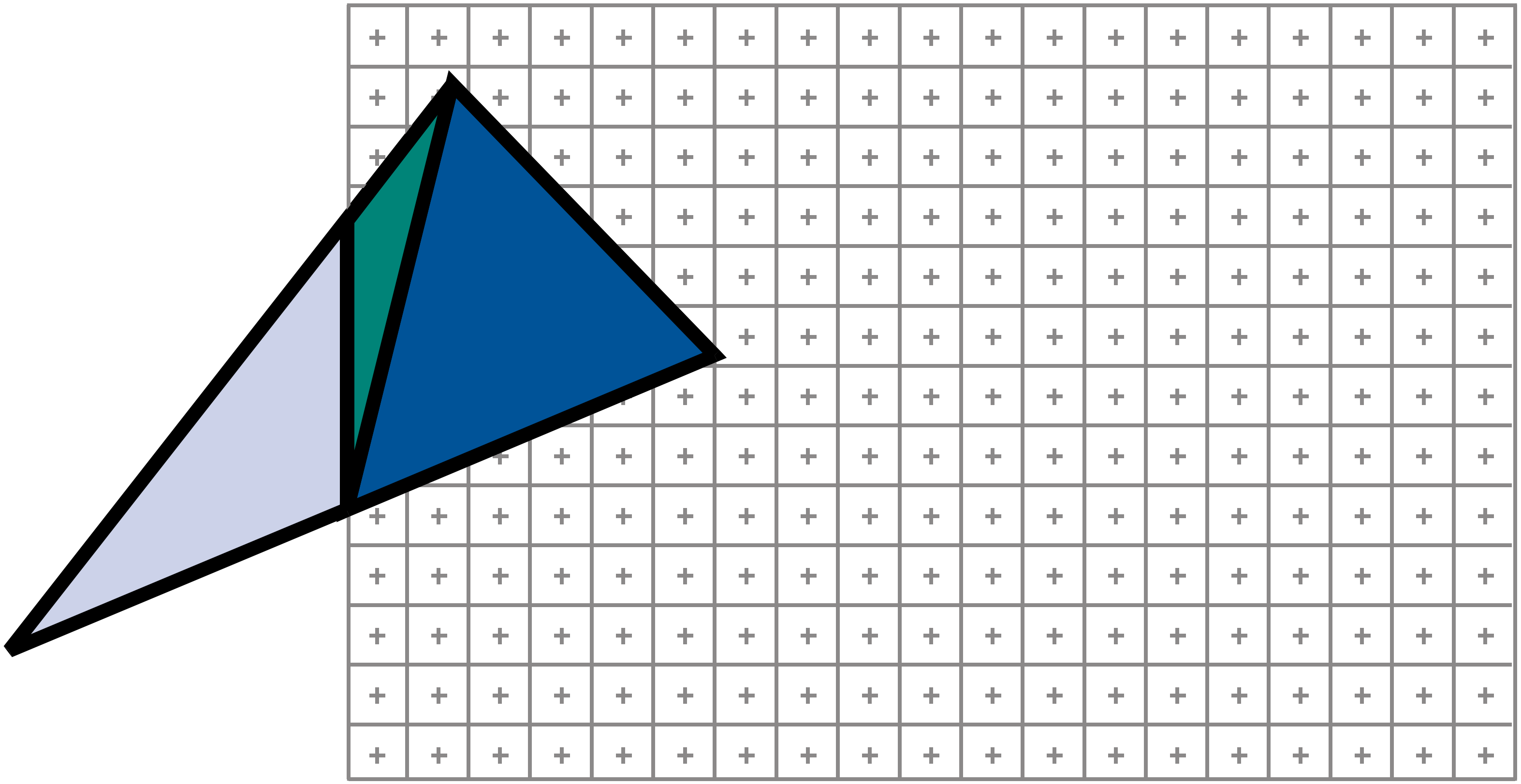
Clipping problem

- How do we clip parts outside window?

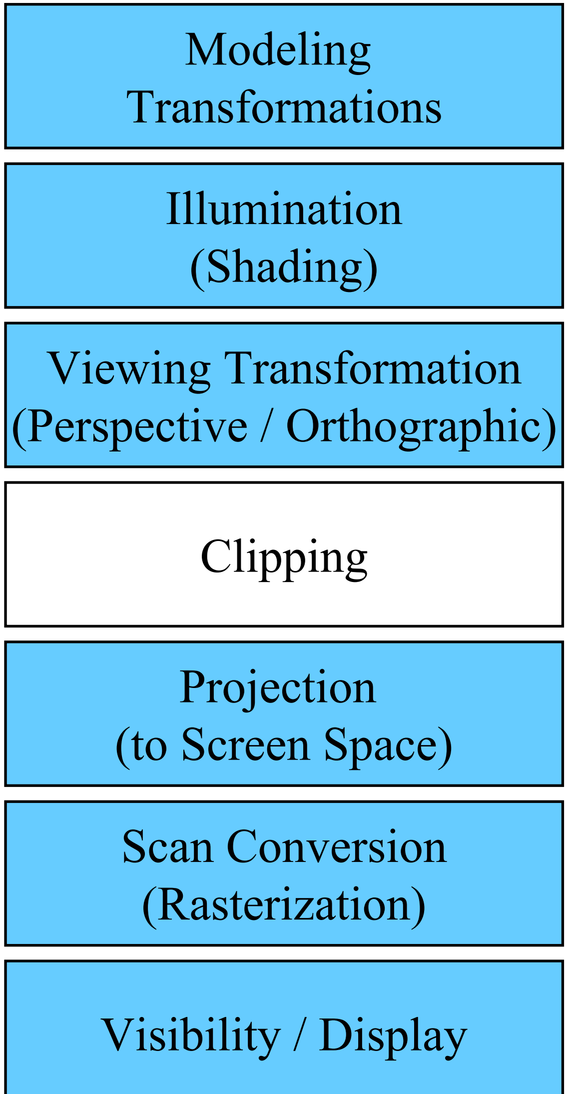


Clipping problem

- How do we clip parts outside window?
- Create two triangles or more. Quite annoying.

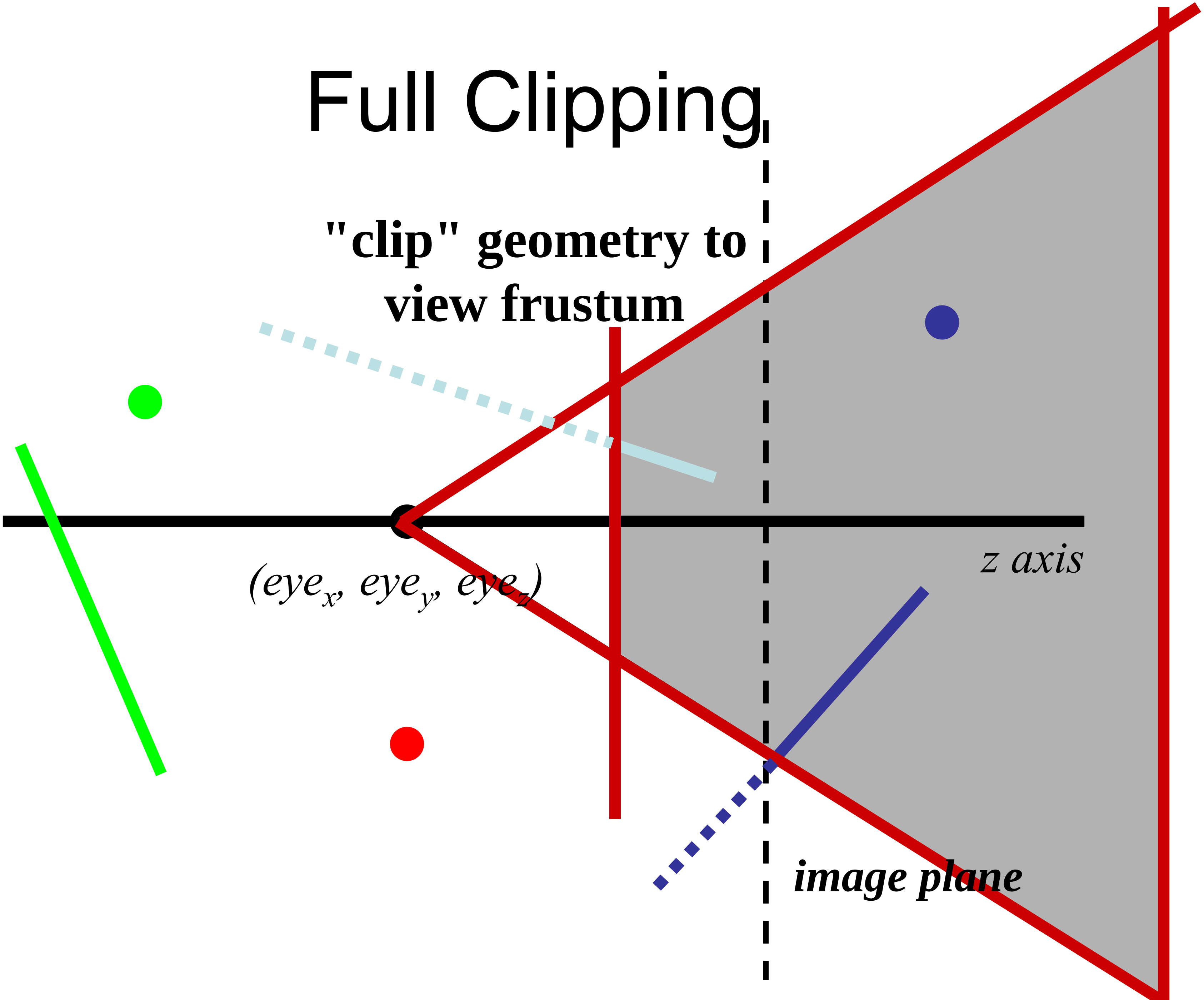


The Graphics Pipeline

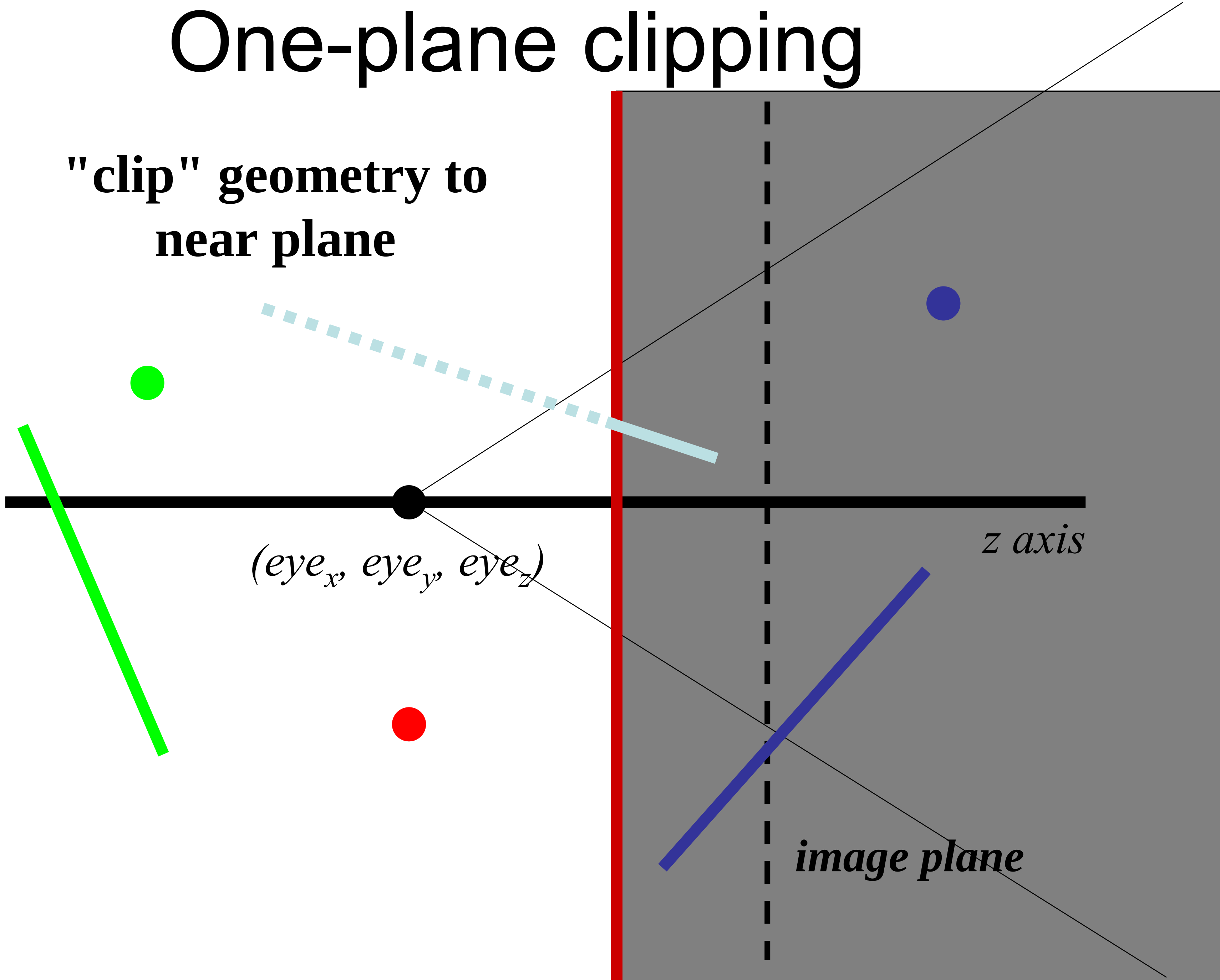


- Former hardware relied on full clipping
- Modern hardware mostly avoids clipping
  - Only with respect to plane z=0
- In general, it is useful to learn clipping because it is similar to many geometric algorithms

Full Clipping

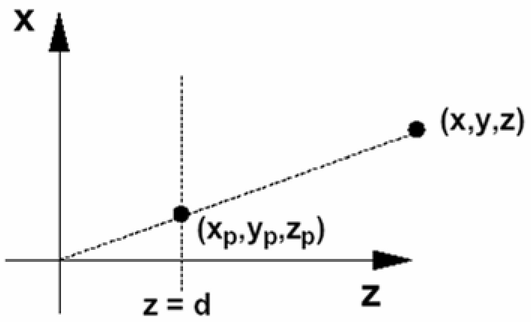


One-plane clipping



When to clip?

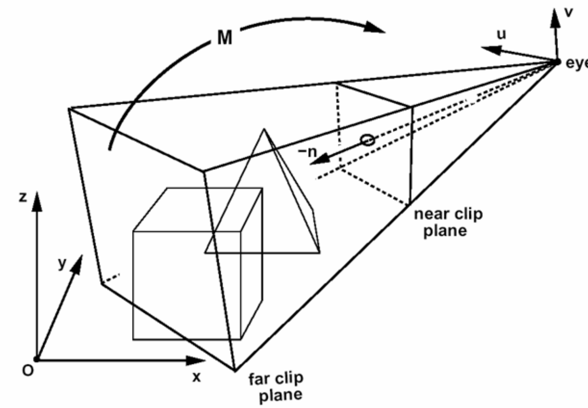
- Perspective Projection: 2 conceptual steps:
  - 4x4 matrix
  - Homogenize
    - In fact not always needed
    - Modern graphics hardware performs most operations in 2D homogeneous coordinates



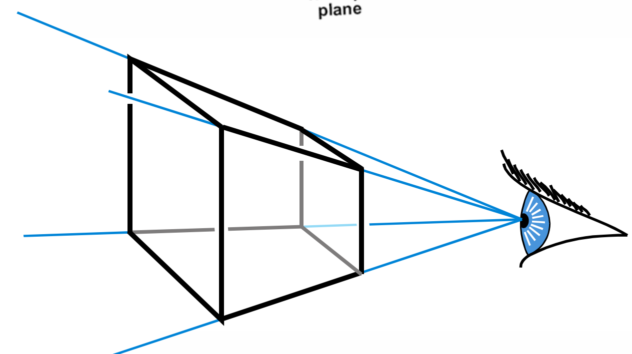
$$\begin{pmatrix} x * d / z \\ y * d / z \\ d/z \\ 1 \end{pmatrix} \xleftarrow{\text{homogenize}} \begin{pmatrix} x \\ y \\ 1 \\ z / d \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

When to clip?

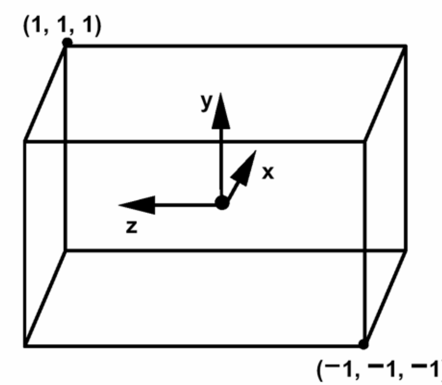
- Before perspective transform in 3D space
  - Use the equation of 6 planes
  - Natural, not too degenerate



- In homogeneous coordinates after perspective transform (Clip space)
  - Before perspective divide (4D space, weird *w* values)
  - Canonical,independent of camera
  - The simplest to implement in fact



- In the transformed 3D screen space after perspective division
  - Problem: objects in the plane of the camera



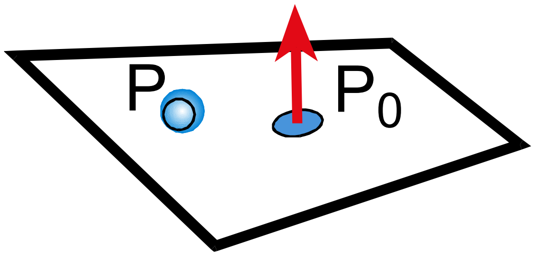
Working in homogeneous coordinates

- In general, many algorithms are simpler in homogeneous coordinates before division
  - Clipping
  - Rasterization

Implicit 3D Plane Equation

- Plane defined by:

  point *p* & normal *n* OR
  normal *n* & offset *d* OR
  3 points



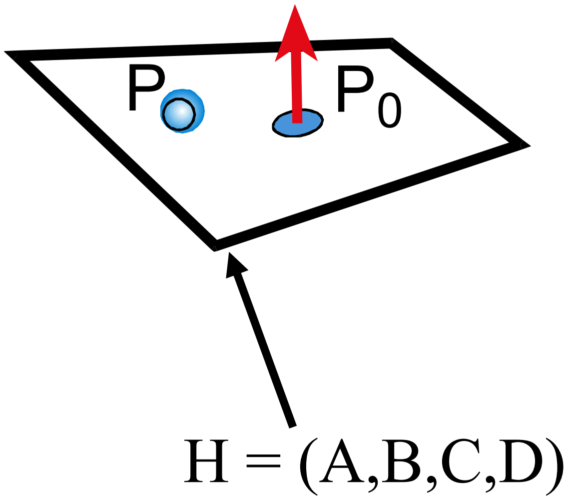
- Implicit plane equation

  $$Ax+By+Cz+D = 0$$

Homogeneous Coordinates

- Homogenous point: (x,y,z,w)

  infinite number of equivalent homogenous coordinates: (sx, sy, sz, sw)



- Homogenous Plane Equation: $Ax+By+Cz+D = 0 \rightarrow H = (A,B,C,D)$

  Infinite number of equivalent plane expressions: $sAx+sBy+sCz+sD = 0 \rightarrow H = (sA,sB,sC,sD)$

Point-to-Plane Distance

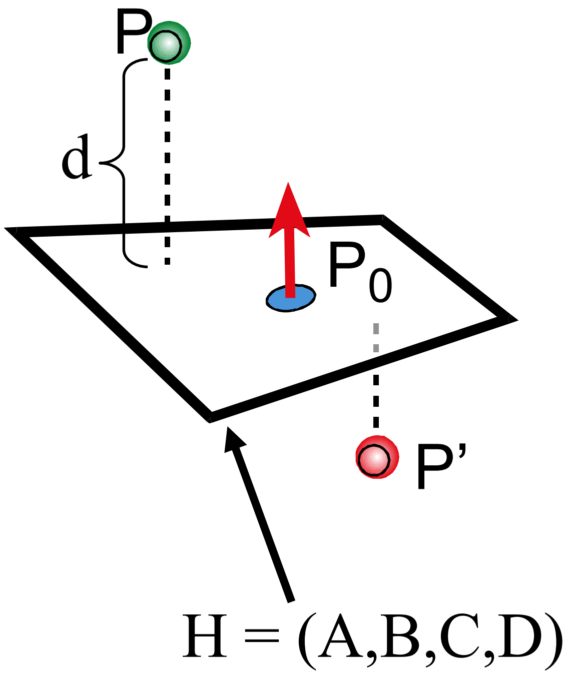
- If (A,B,C) is normalized:

  $d = H \cdot p = H^T p$
  (the dot product in homogeneous coordinates)

- d is a *signed distance*

  positive = "inside"

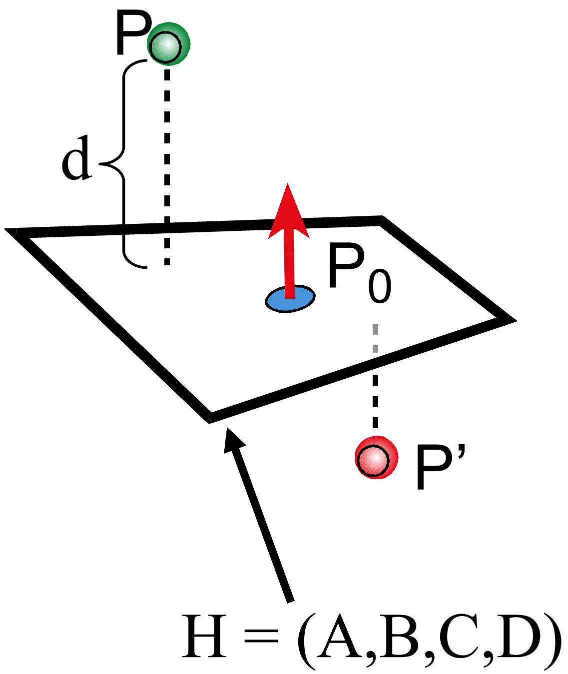
  negative = "outside"



Clipping a Point with respect to a Plane

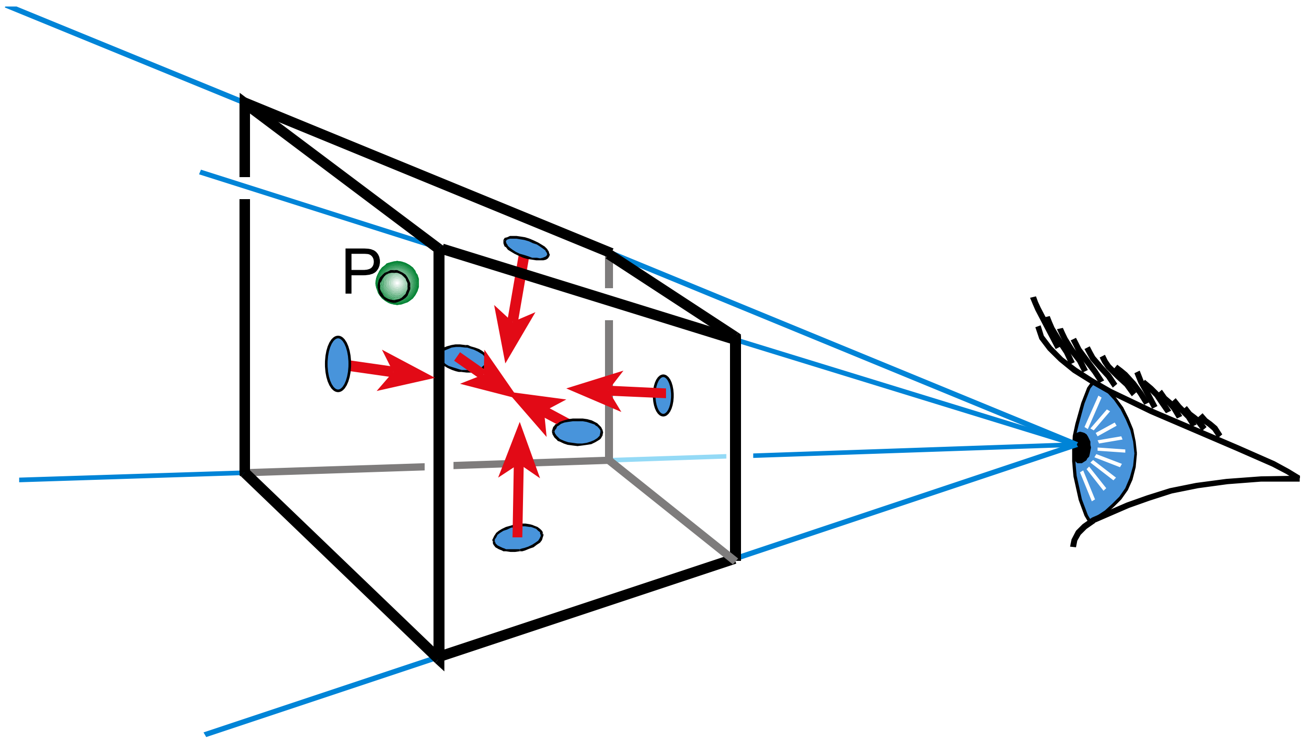
- If $d = H \bullet p \geq 0$
  Pass through

- If $d = H \bullet p < 0$:
  Clip (or cull or reject)

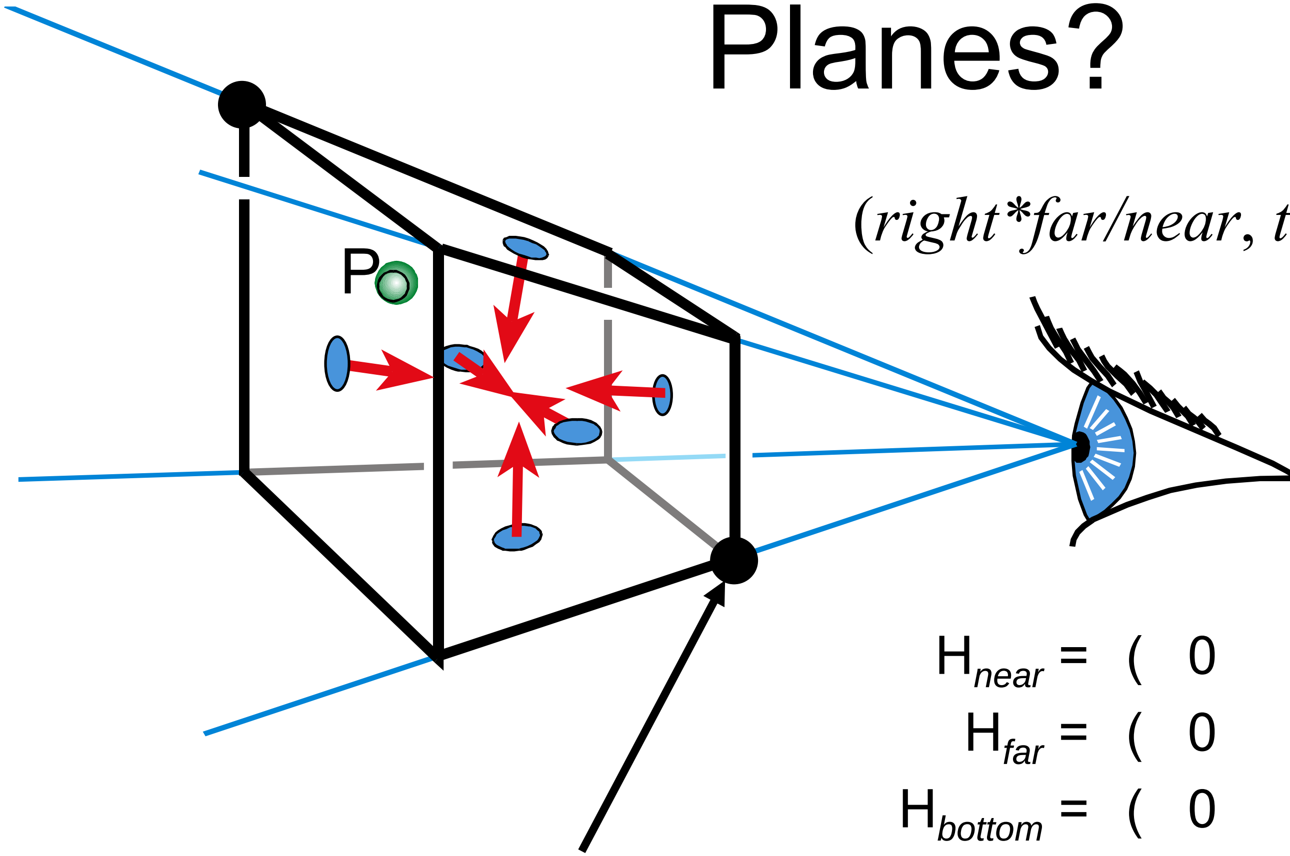


Clipping with respect to View Frustum

- Test against each of the 6 planes
  - Normals oriented towards the interior
- Clip (or cull or reject) point p if any $H \cdot p < 0$



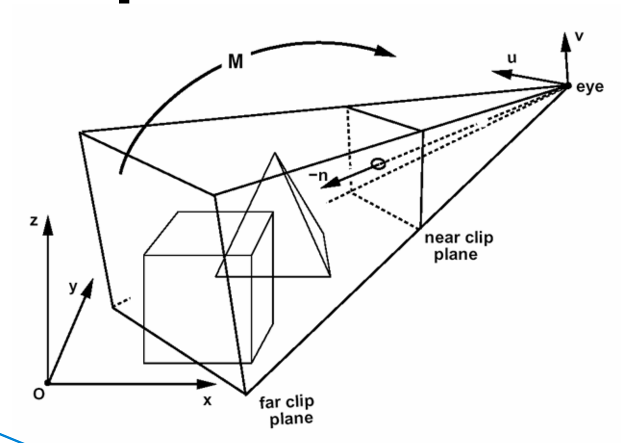
What are the View Frustum Planes?

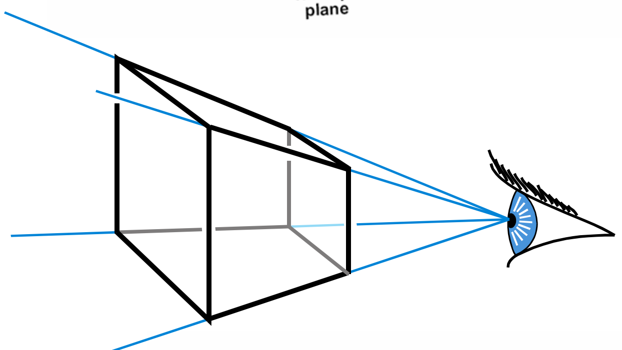


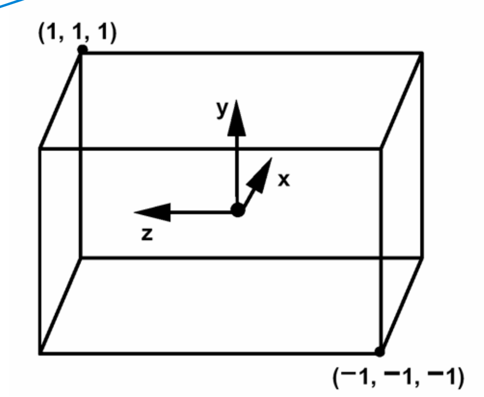
$$
\begin{array}{rcllll}
H_{near} & = & (\; 0 & 0 & -1 & -near) \\
H_{far} & = & (\; 0 & 0 & 1 & far \;) \\
H_{bottom} & = & (\; 0 & near & bottom & 0 \;) \\
H_{top} & = & (\; 0 & -near & -top & 0 \;) \\
H_{left} & = & (\; left & near & 0 & 0 \;) \\
H_{right} & = & (-right & -near & 0 & 0 \;)
\end{array}
$$

Recall: When to clip?

- Before perspective transform in 3D space
  - Use the equation of 6 planes
  - Natural, not too degenerate
- In homogeneous coordinates after perspective transform (Clip space)
  - Before perspective divide (4D space, weird *w* values)
  - Canonical,independent of camera
  - The simplest to implement in fact
- In the transformed 3D screen space after perspective division
  - Problem: objects in the plane of the camera







Line – Plane Intersection

- Explicit (Parametric) Line Equation

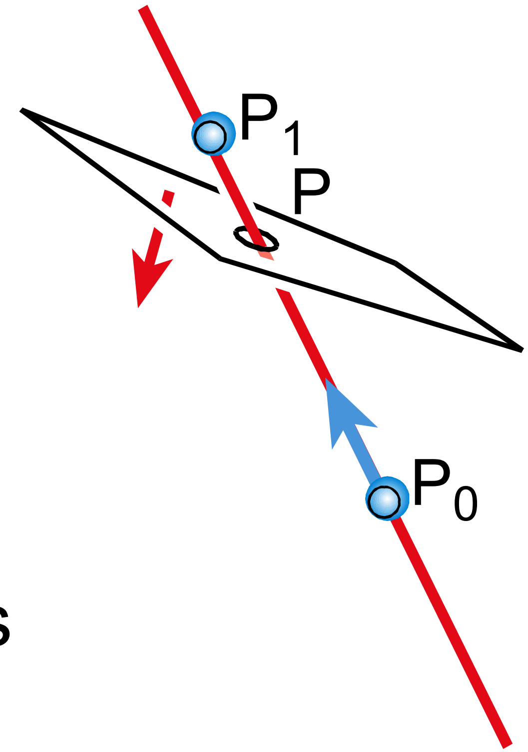
  $L(t) = P_0 + t * (P_1 - P_0)$

  $L(t) = (1-t) * P_0 + t * P_1$

- How do we intersect?

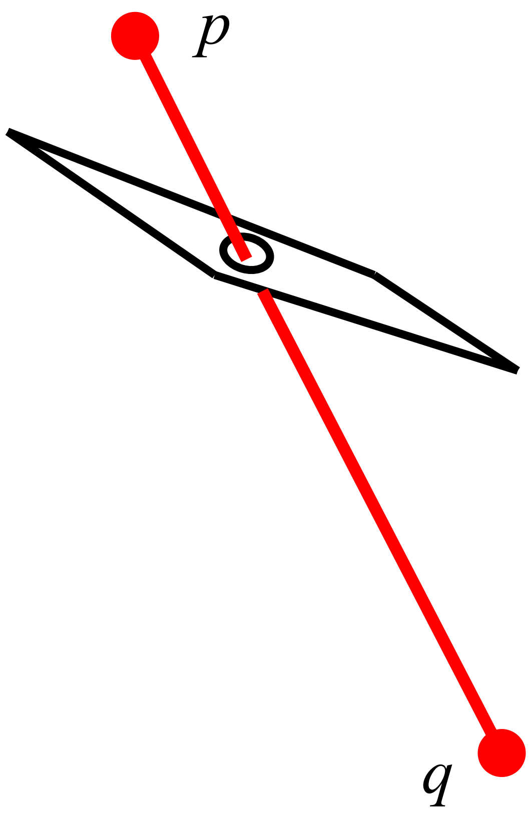
  Insert explicit equation of line into implicit equation of plane

- Parameter *t* is used to interpolate associated attributes (color, normal, texture, etc.)



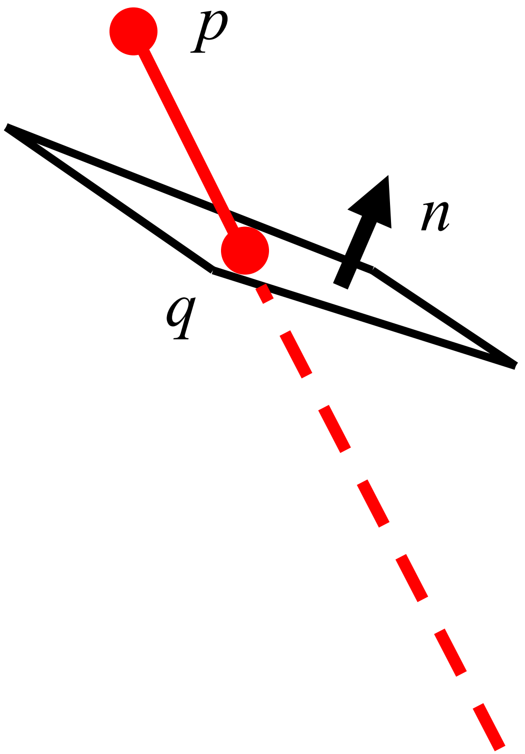
Segment Clipping

- If H•p > 0 and H•q < 0
- If H•p < 0 and H•q > 0
- If H•p > 0 and H•q > 0
- If H•p < 0 and H•q < 0



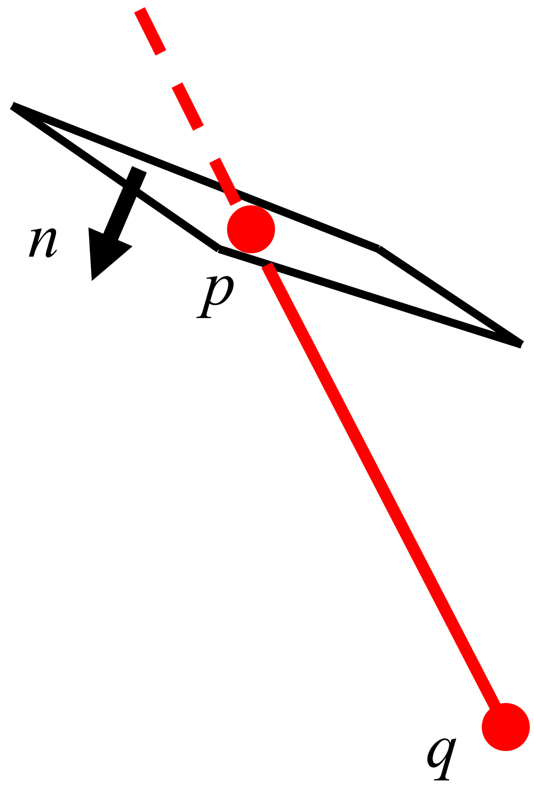
Segment Clipping

- If H•p > 0 and H•q < 0
  - clip q to plane
- If H•p < 0 and H•q > 0
- If H•p > 0 and H•q > 0
- If H•p < 0 and H•q < 0



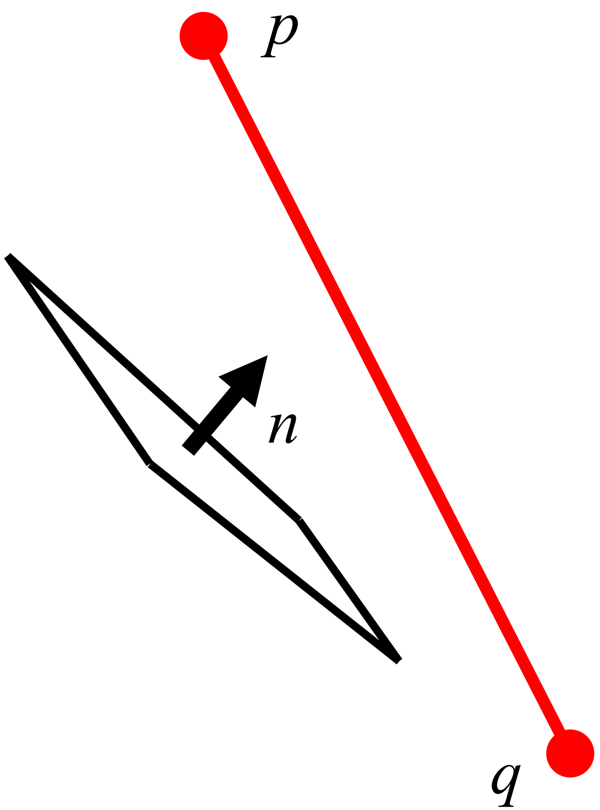
Segment Clipping

- If H•p > 0 and H•q < 0
  - clip q to plane
- If H•p < 0 and H•q > 0
  - clip p to plane
- If H•p > 0 and H•q > 0
- If H•p < 0 and H•q < 0



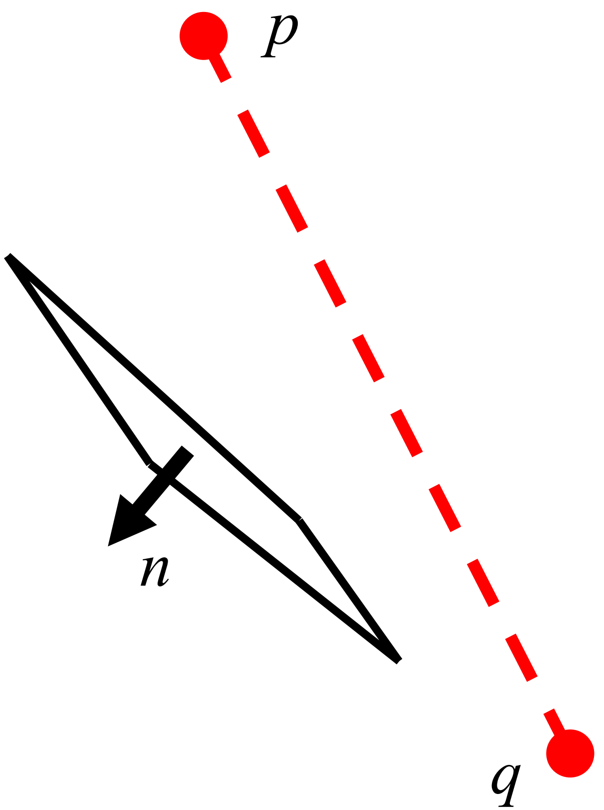
Segment Clipping

- If H•p > 0 and H•q < 0
  – clip q to plane
- If H•p < 0 and H•q > 0
  – clip p to plane
- If H•p > 0 and H•q > 0
  – pass through
- If H•p < 0 and H•q < 0



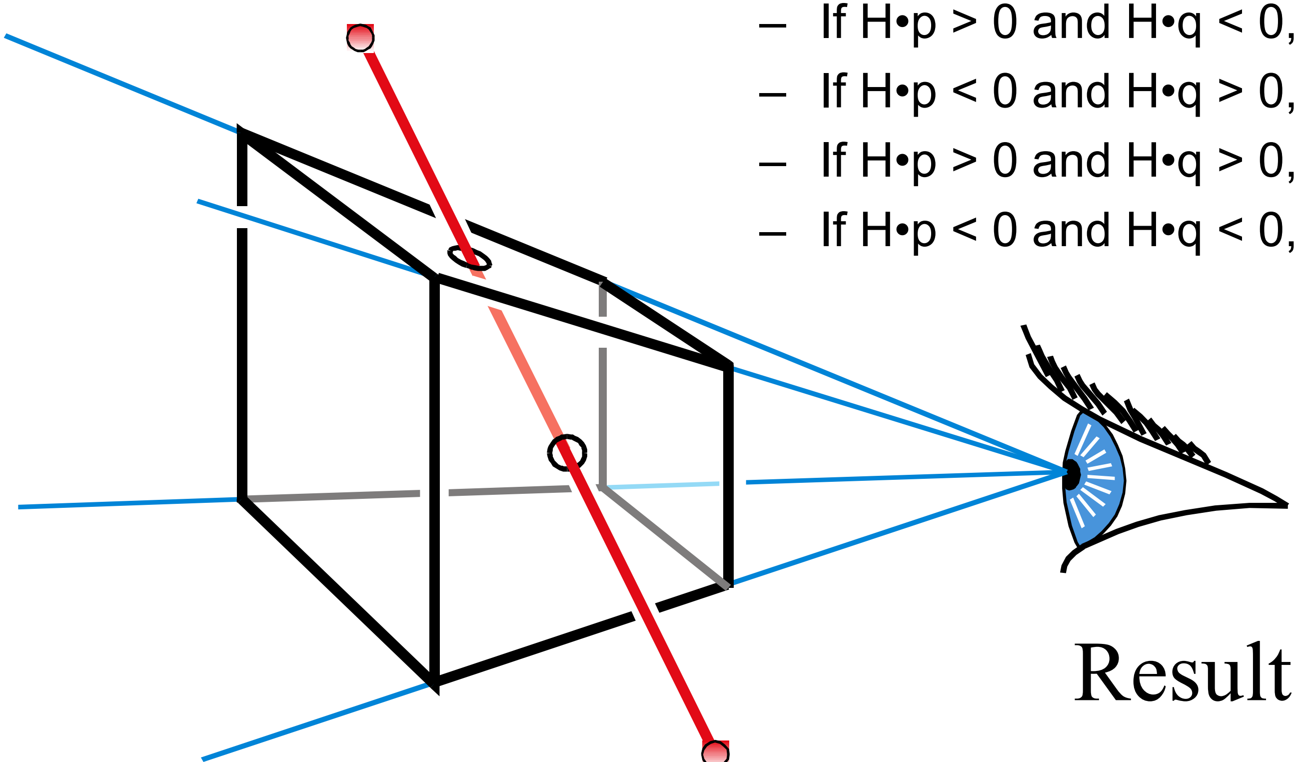
Segment Clipping

- If H•p > 0 and H•q < 0
  – clip q to plane
- If H•p < 0 and H•q > 0
  – clip p to plane
- If H•p > 0 and H•q > 0
  – pass through
- If H•p < 0 and H•q < 0
  – clipped out



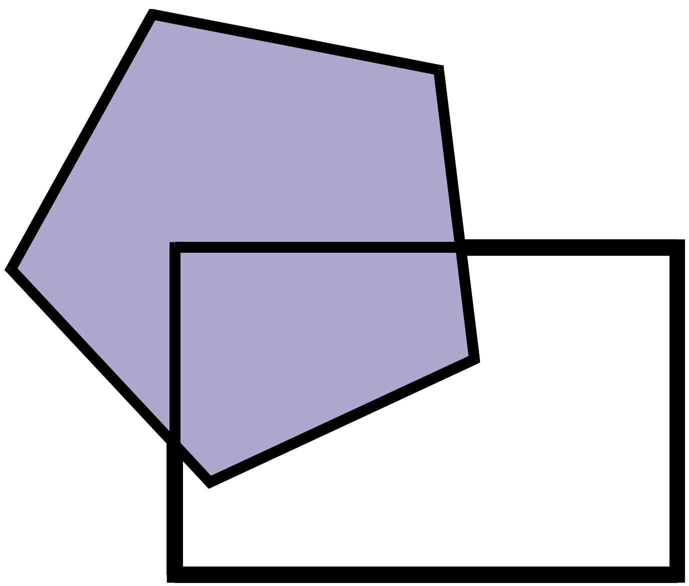
Clipping against the frustum

- For each frustum plane H
  - If H•p > 0 and H•q < 0, clip q to H
  - If H•p < 0 and H•q > 0, clip p to H
  - If H•p > 0 and H•q > 0, pass through
  - If H•p < 0 and H•q < 0, clipped out

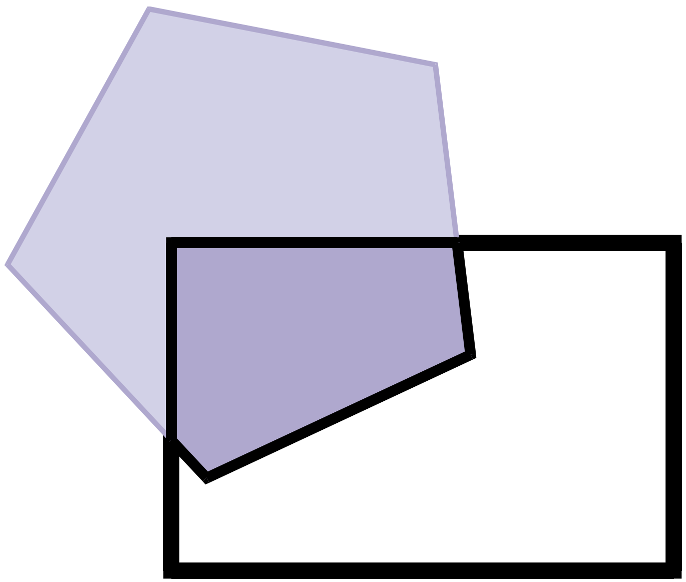


Result is a single segment.

Polygon clipping

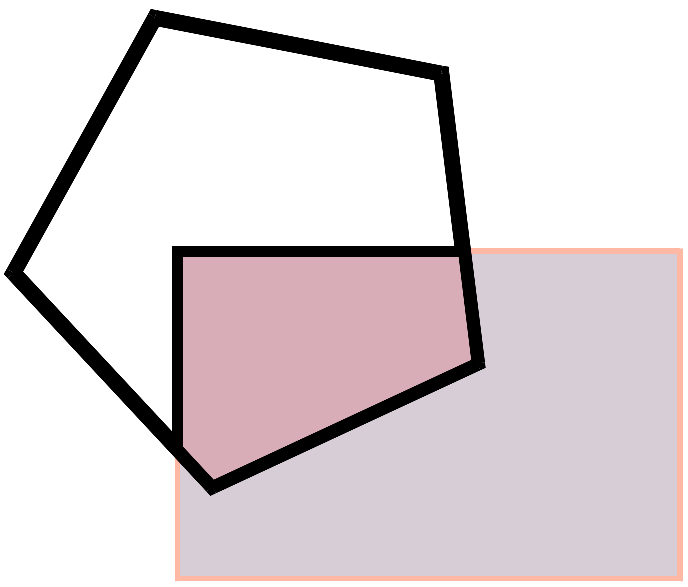


Polygon clipping



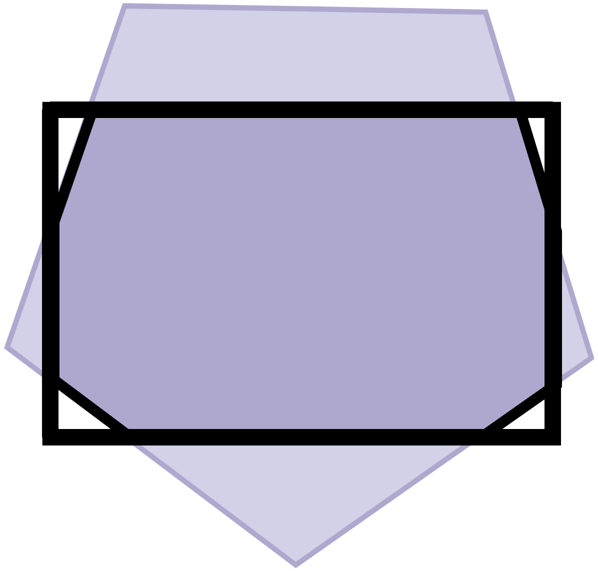
Polygon clipping

- Clipping is symmetric



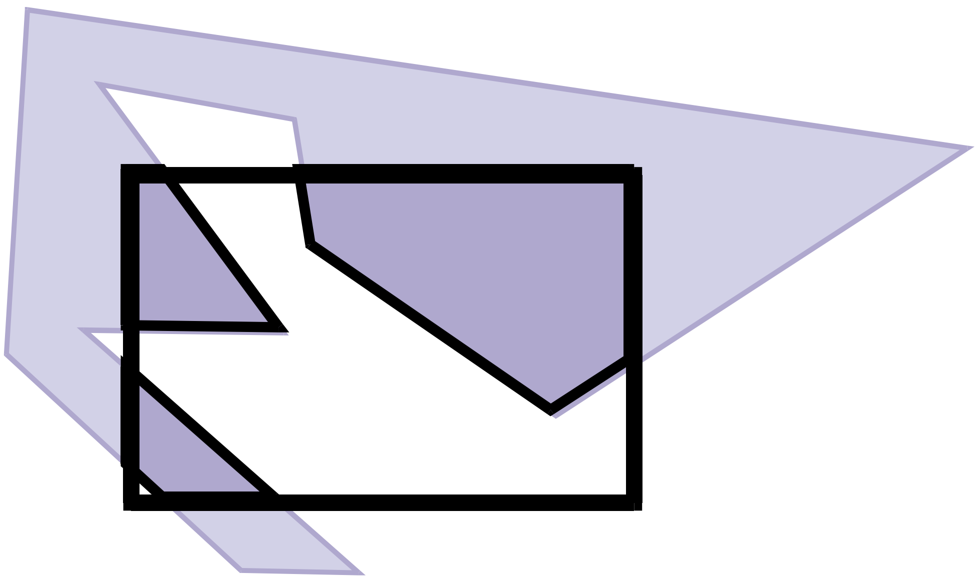
Polygon clipping is complex

- Even when the polygons are convex



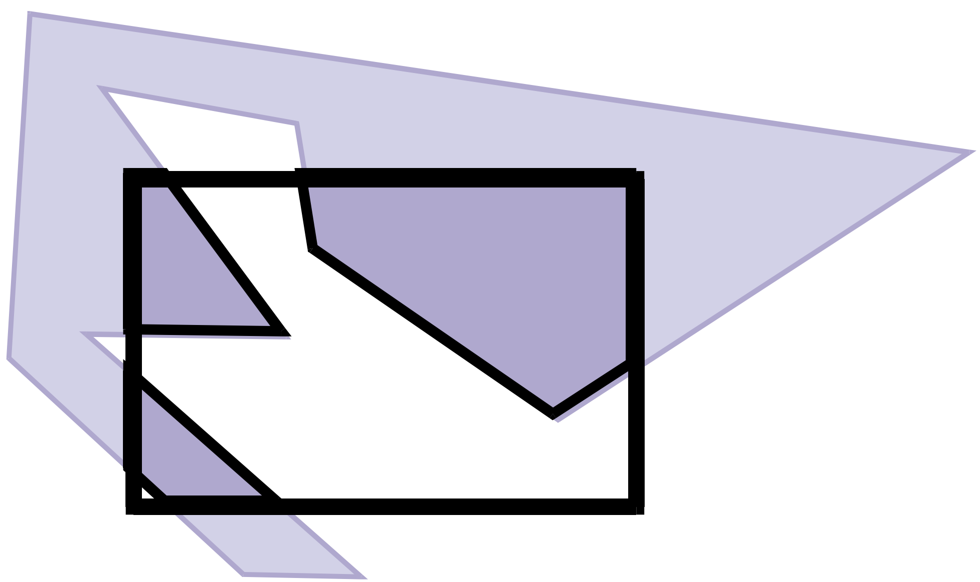
Polygon clipping is nasty

- When the polygons are concave



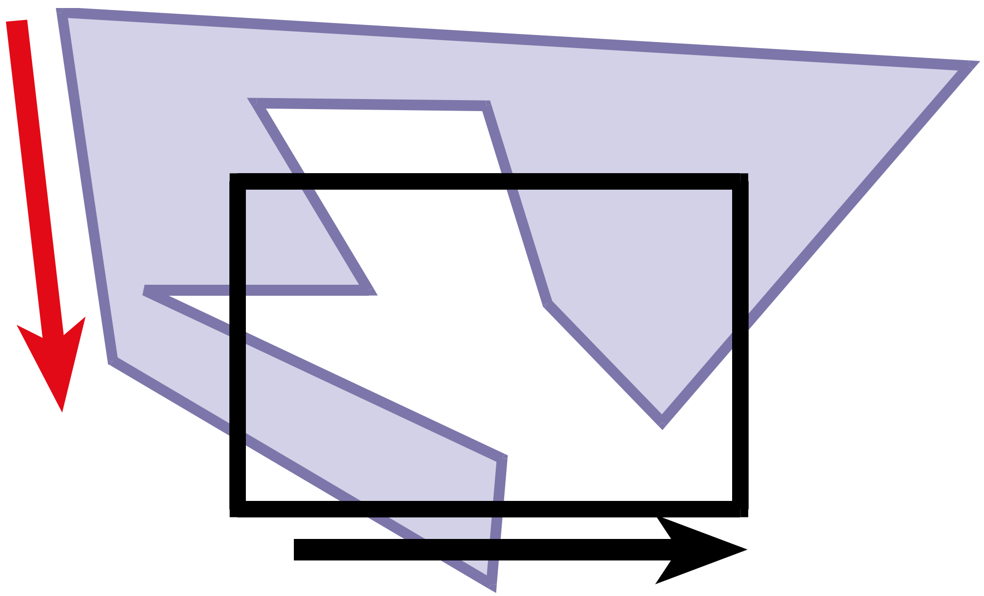
Naïve polygon clipping?

- N*m intersections
- Then must link all segment
- Not efficient and not even easy



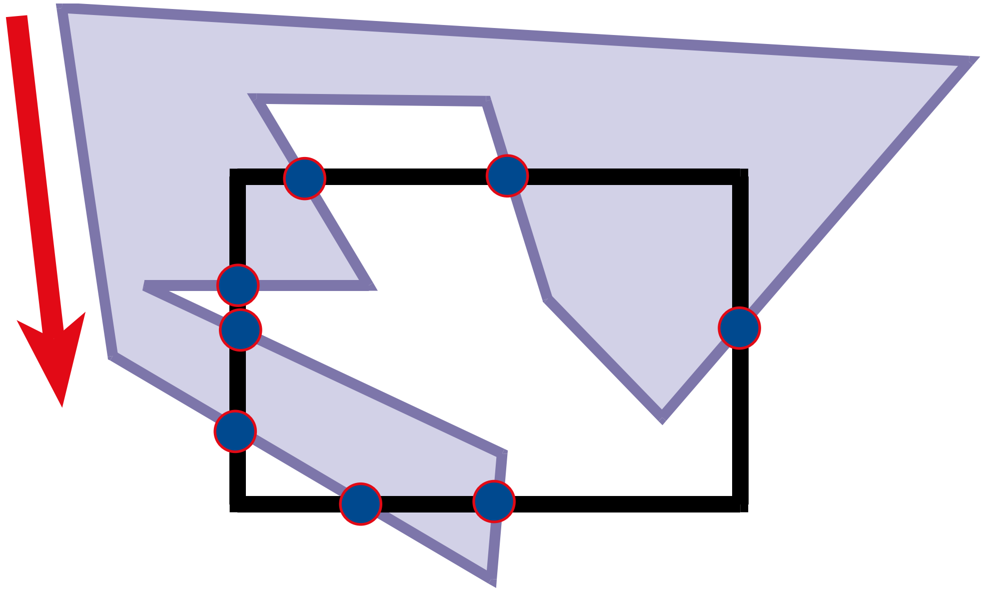
Weiler-Atherton Clipping

- Strategy: “Walk" polygon/window boundary
- Polygons are oriented (CCW)



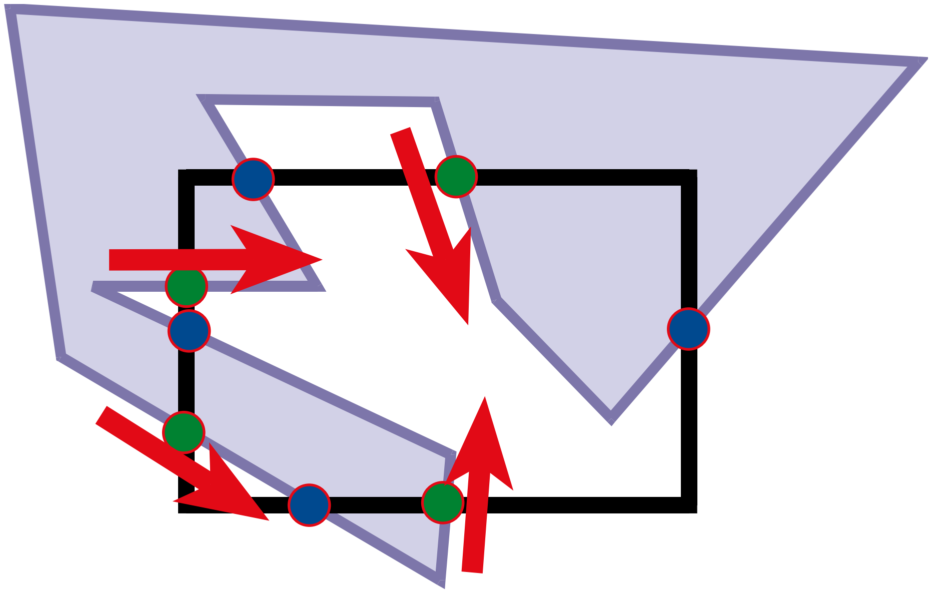
Weiler-Atherton Clipping

- Compute intersection points



Weiler-Atherton Clipping

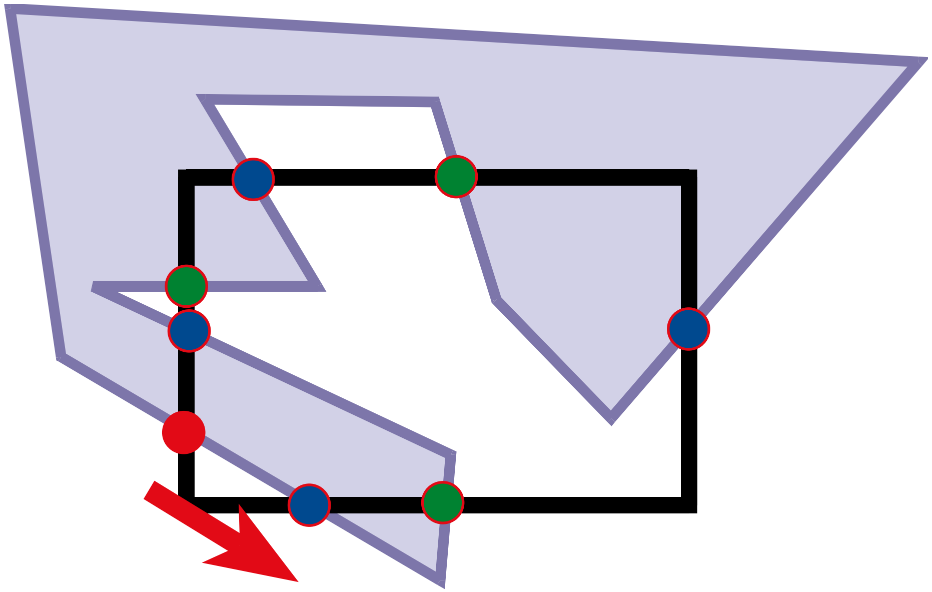
- Compute intersection points
- Mark points where polygons enters clipping window (green here)



Clipping

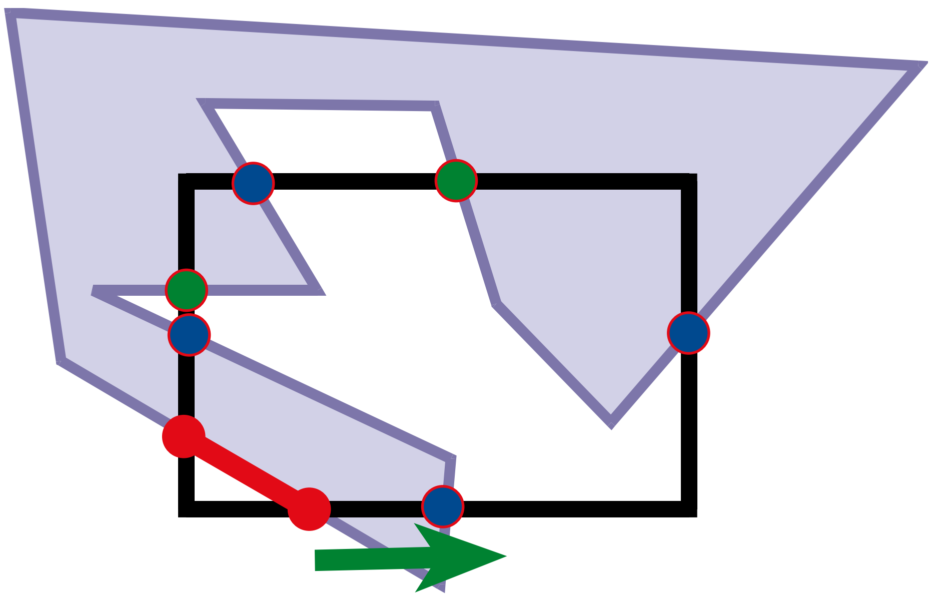
While there is still an unprocessed entering intersection

Walk" polygon/window boundary



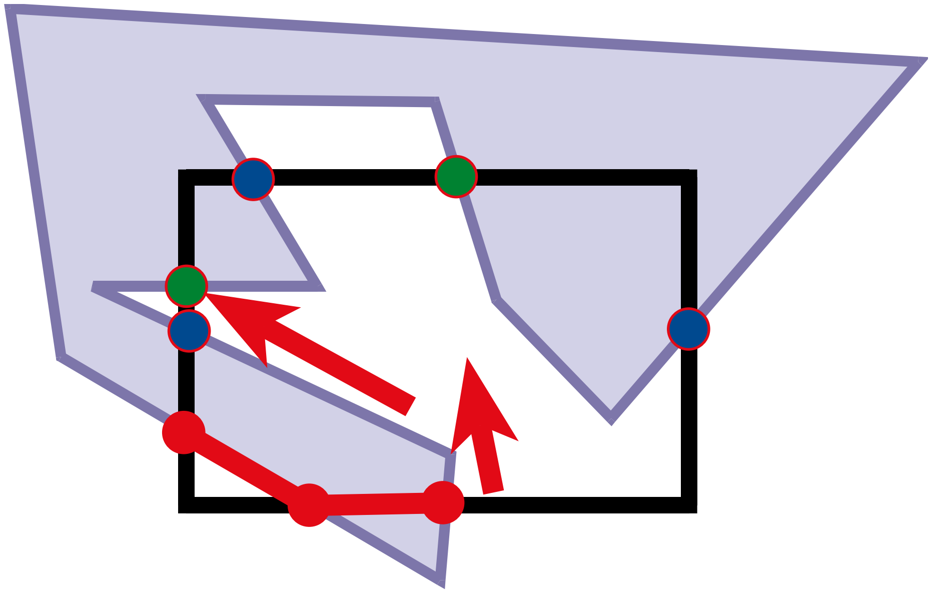
Walking rules

- Out-to-in pair:
  - Record clipped point
  - Follow polygon boundary (ccw)
- In-to-out pair:
  - Record clipped point
  - Follow window boundary (ccw)



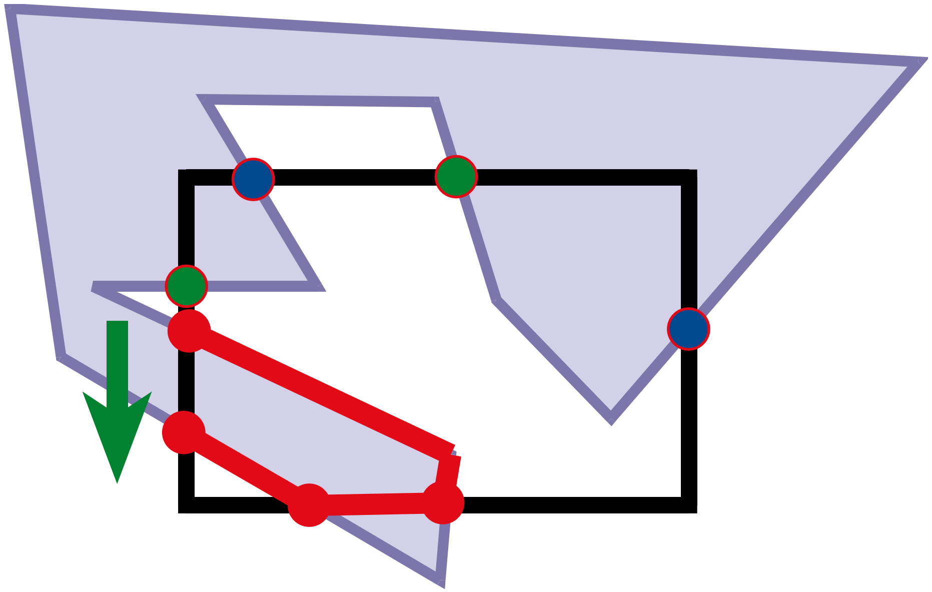
Walking rules

- Out-to-in pair:
  - Record clipped point
  - Follow polygon boundary (ccw)
- In-to-out pair:
  - Record clipped point
  - Follow window boundary (ccw)



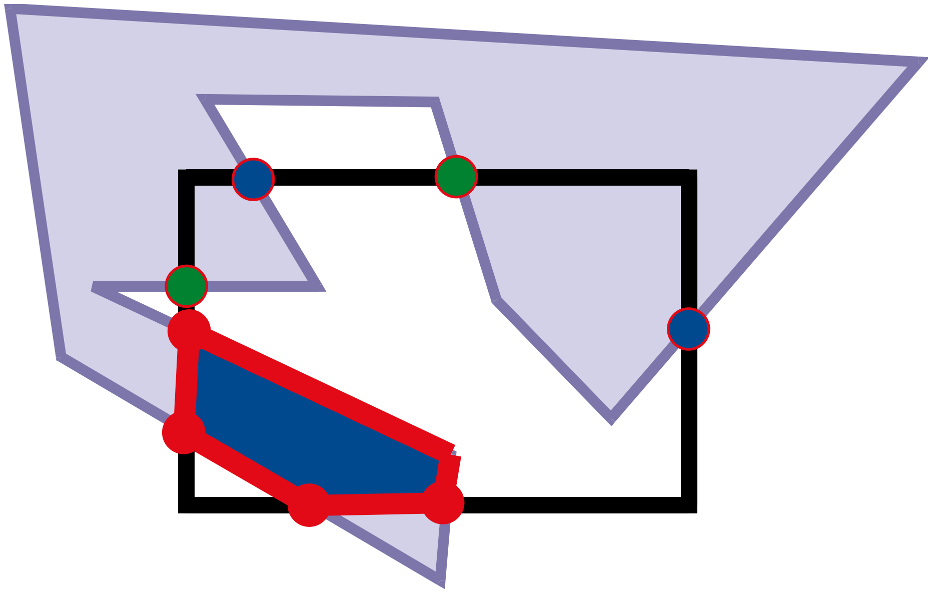
Walking rules

- Out-to-in pair:
  - Record clipped point
  - Follow polygon boundary (ccw)
- In-to-out pair:
  - Record clipped point
  - Follow window boundary (ccw)



Walking rules

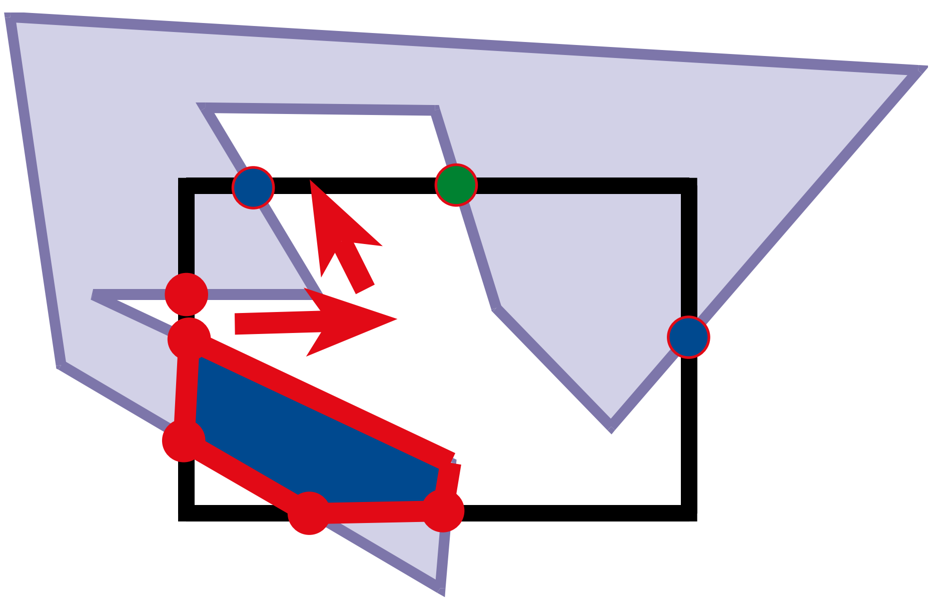
- Out-to-in pair:
  - Record clipped point
  - Follow polygon boundary (ccw)
- In-to-out pair:
  - Record clipped point
  - Follow window boundary (ccw)



Walking rules

While there is still an unprocessed entering intersection

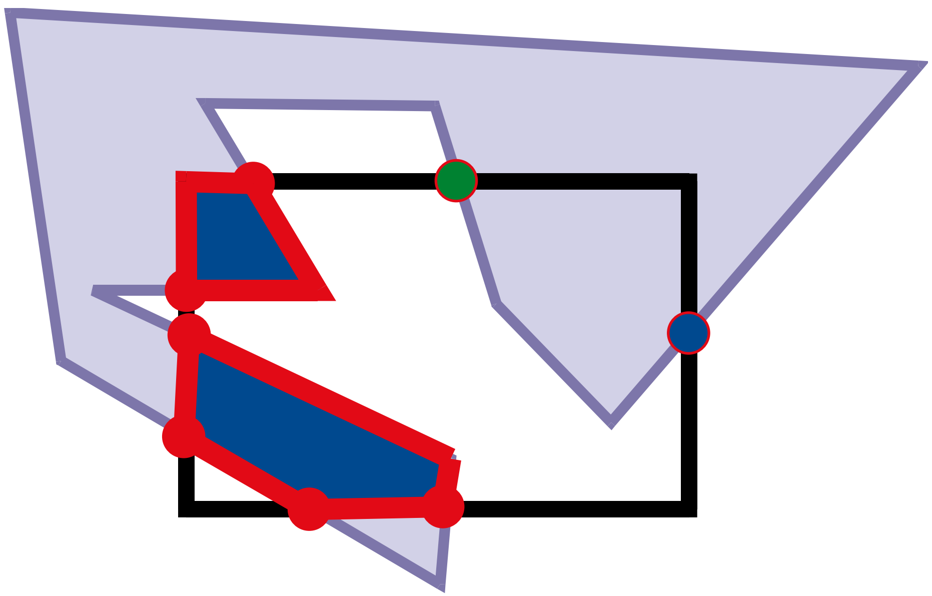
Walk" polygon/window boundary



Walking rules

While there is still an unprocessed entering intersection

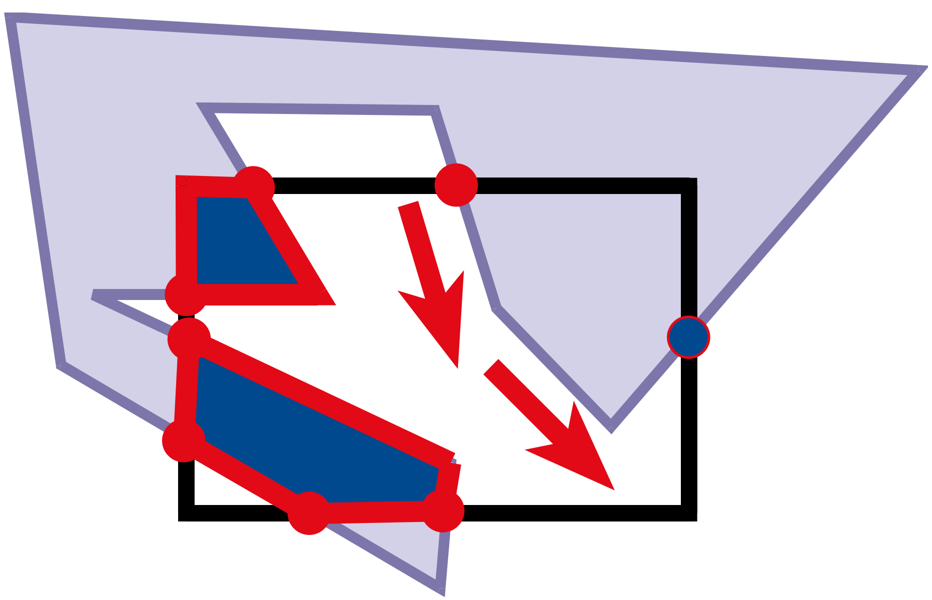
Walk" polygon/window boundary



Walking rules

While there is still an unprocessed entering intersection

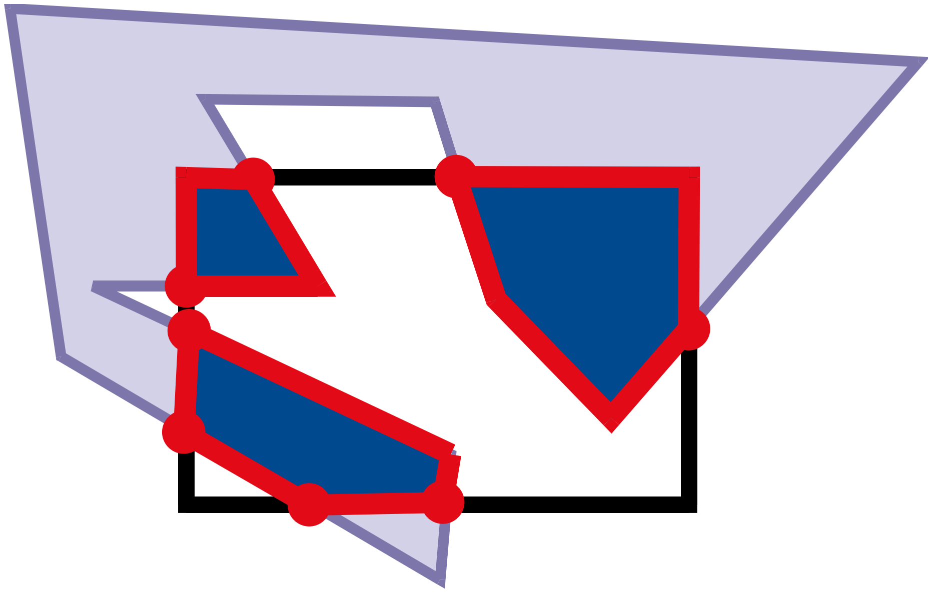
Walk" polygon/window boundary



Walking rules

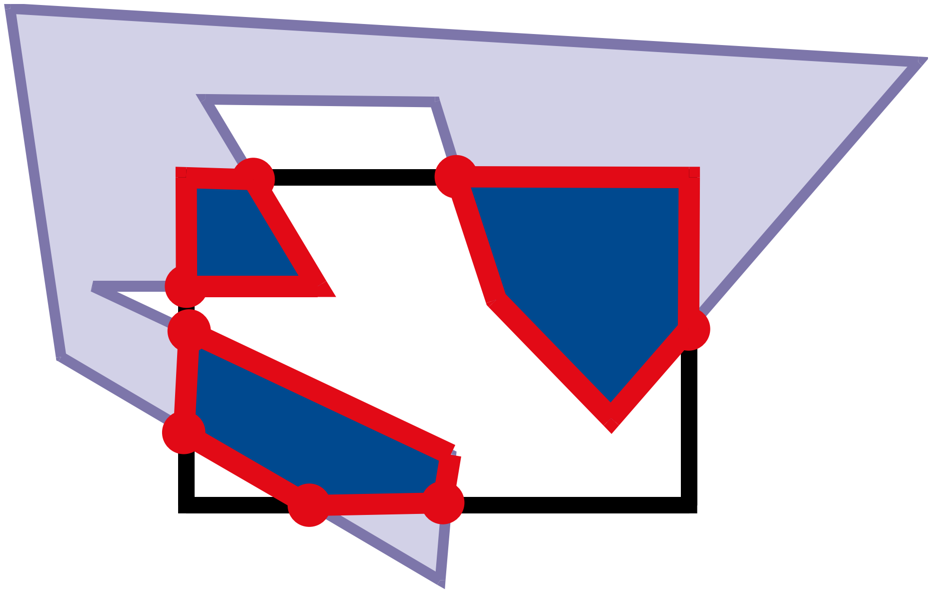
While there is still an unprocessed entering intersection

Walk" polygon/window boundary



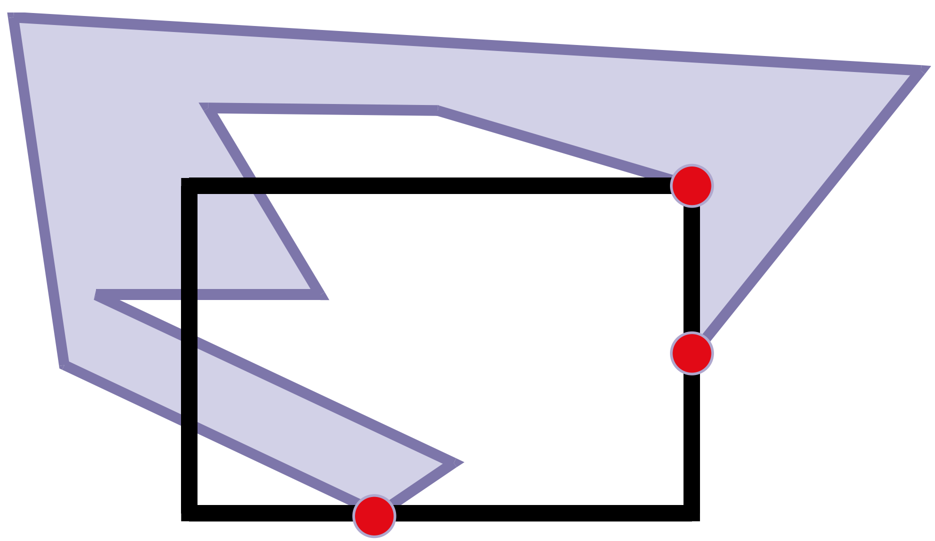
Weiler-Atherton Clipping

- Importance of good adjacency data structure
  (here simply list of oriented edges)



Robustness, precision, degeneracies

- What if a vertex is on the boundary?
- What happens if it is “almost” on the boundary?
  - Problem with floating point precision
- Welcome to the real world of geometry!



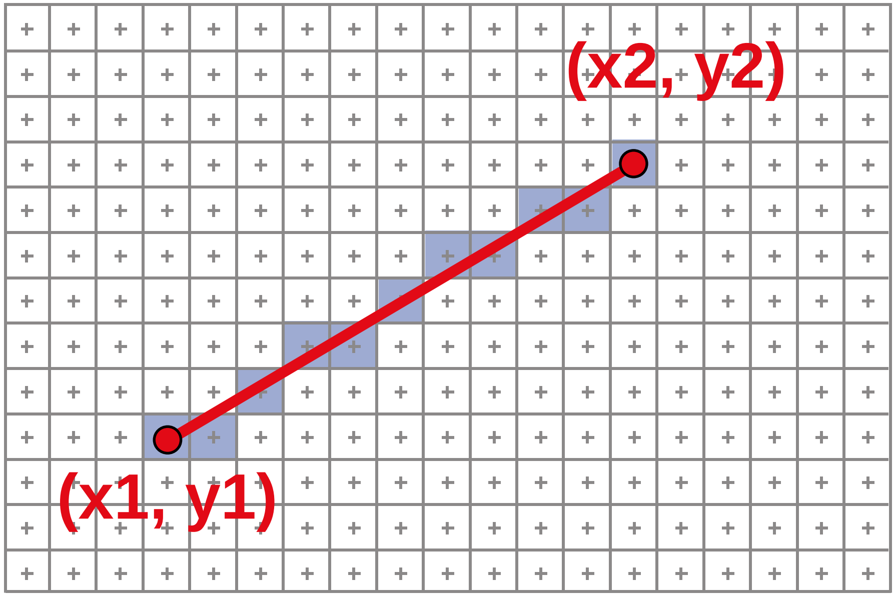
Clipping

- Many other clipping algorithms:
- Parametric, general windows, region-region, One-Plane-at-a-Time Clipping, etc.

PAUSE

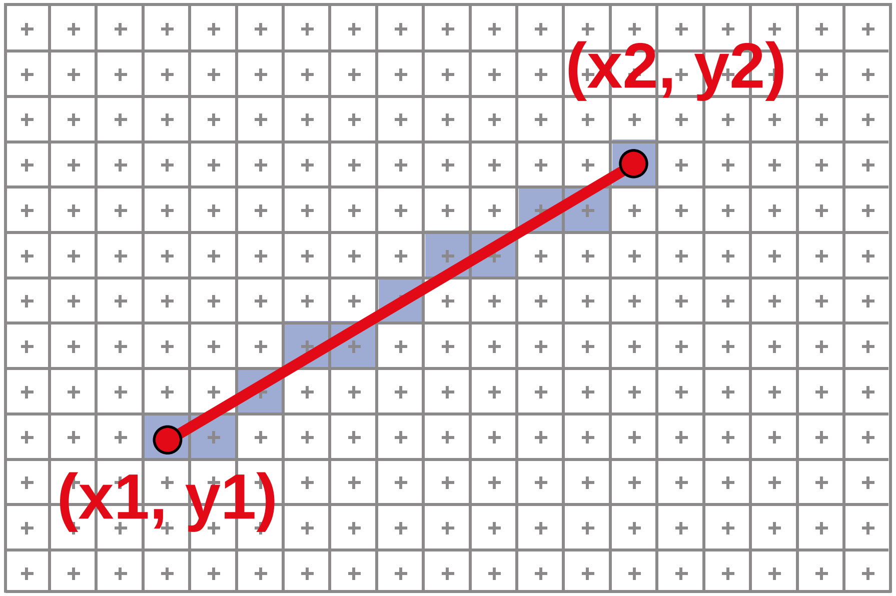
Scan Converting 2D Line Segments

- Given:
  - Segment endpoints (integers x1, y1; x2, y2)
- Identify:
  - Set of pixels (x, y) to display for segment



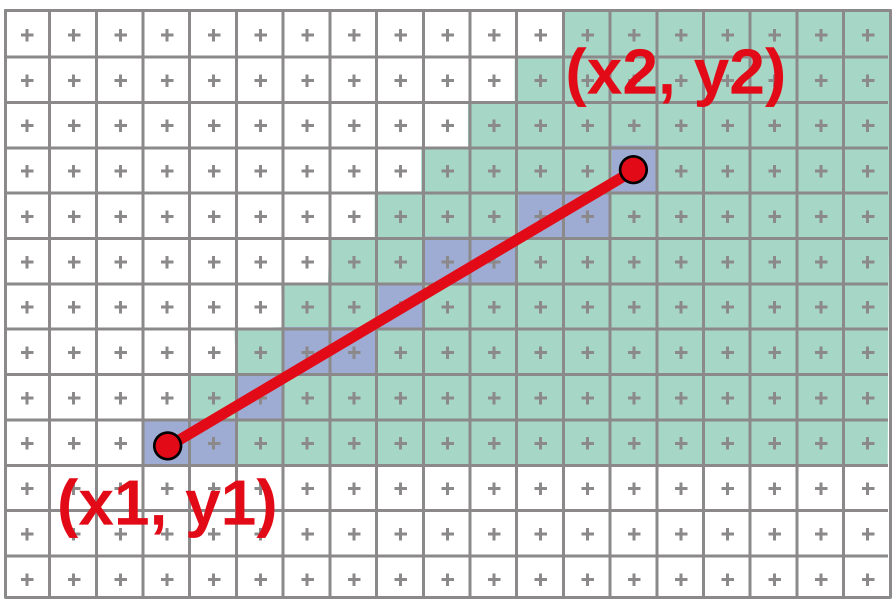
Line Rasterization Requirements

- Transform **continuous** primitive into **discrete** samples
- Uniform thickness & brightness
- Continuous appearance
- No gaps
- Accuracy
- Speed



Algorithm Design Choices

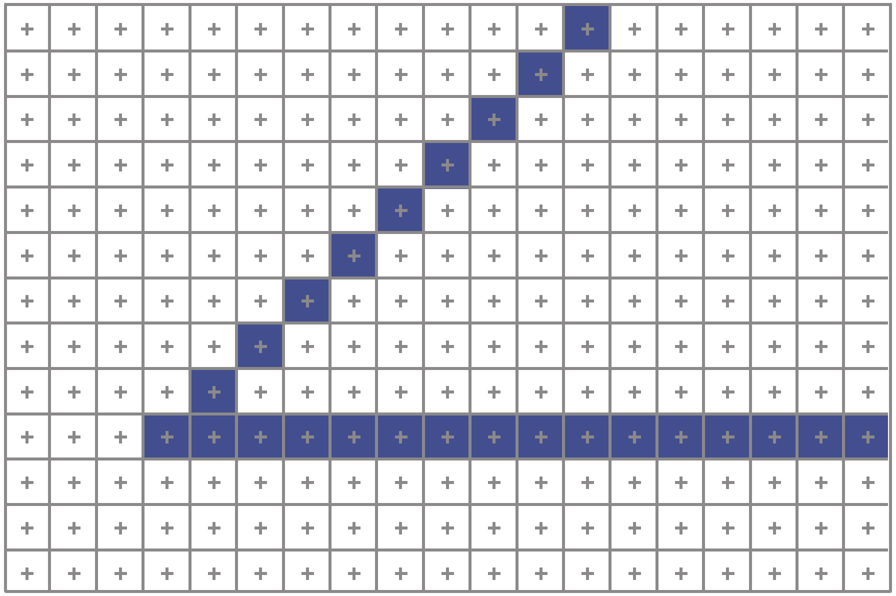
- Assume:
  - $m = dy/dx, \;\; 0 < m < 1$
- Exactly one pixel per column
  - fewer → disconnected, more → too thick

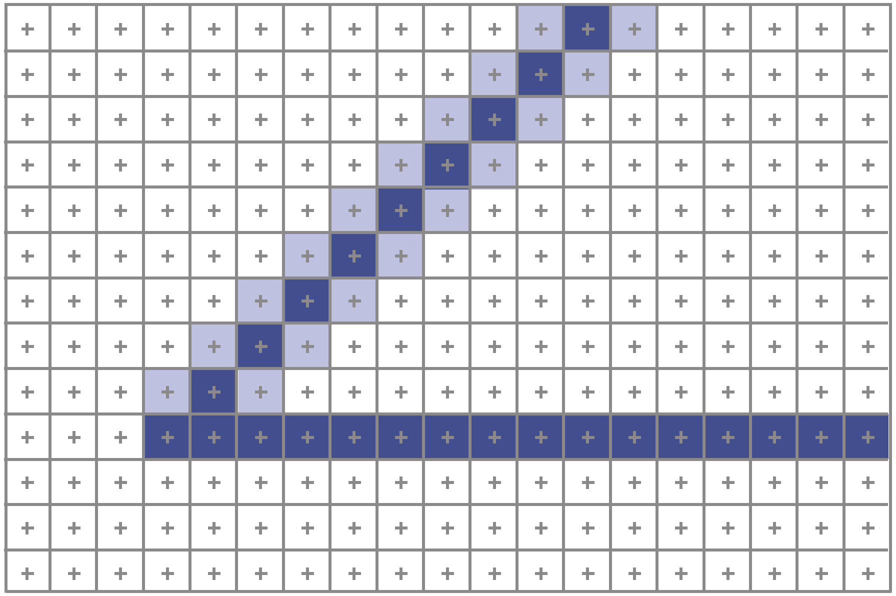


Algorithm Design Choices

- Note: brightness can vary with slope
  - What is the maximum variation?
- How could we compensate for this?
  - Answer: antialiasing

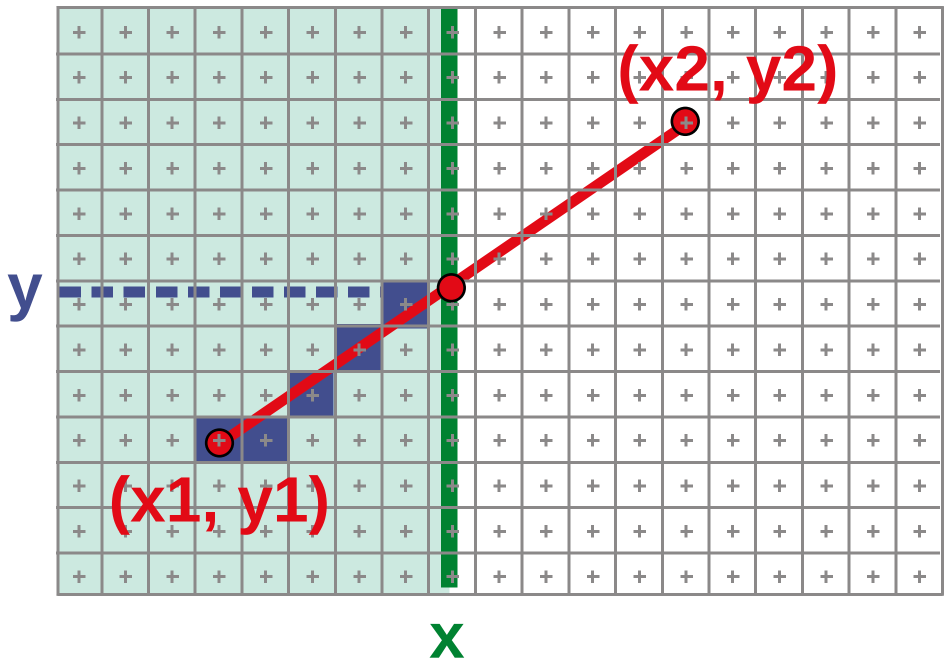
$$\sqrt{2}$$





Naive Line Rasterization Algorithm

- Simply compute y as a function of x
  - Conceptually: move vertical scan line from x1 to x2
  - What is the expression of y as function of x?
  - Set pixel (x, round (y(x)))



$$y = y1 + \frac{x - x1}{x2 - x1}(y2 - y1)$$

$$= y1 + m(x - x1)$$

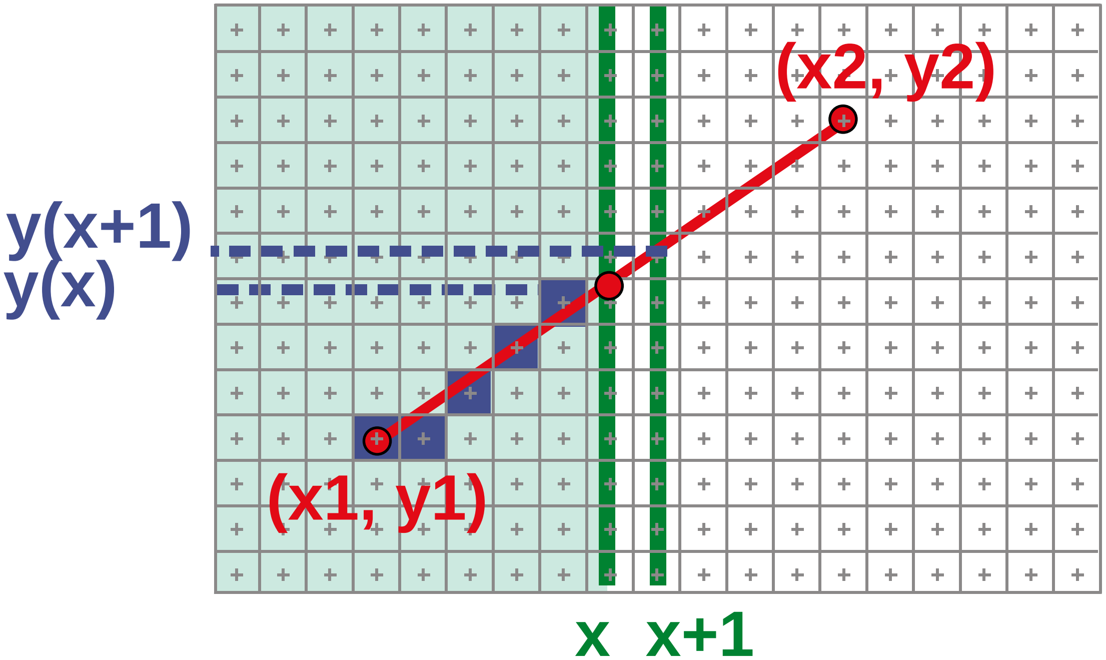
$$m = \frac{dy}{dx}$$

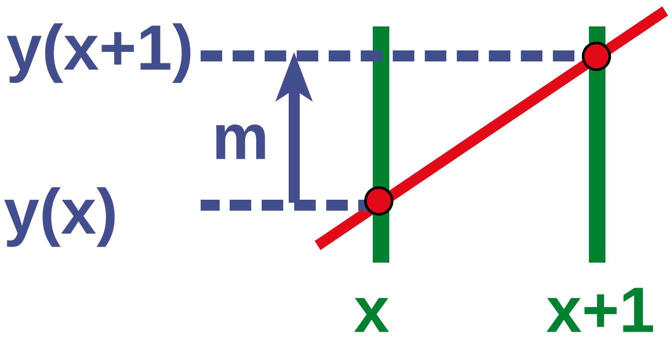
Efficiency

- Computing y value is expensive

$$y = y1 + m(x - x1)$$

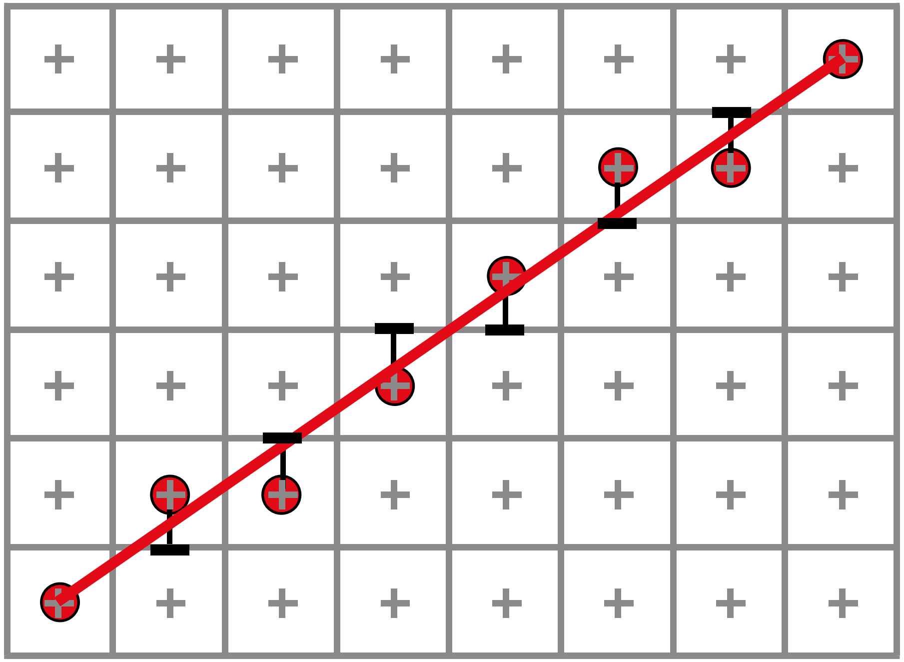
- Observe: *y* += *m* at each *x* step (*m* = *dy/dx*)





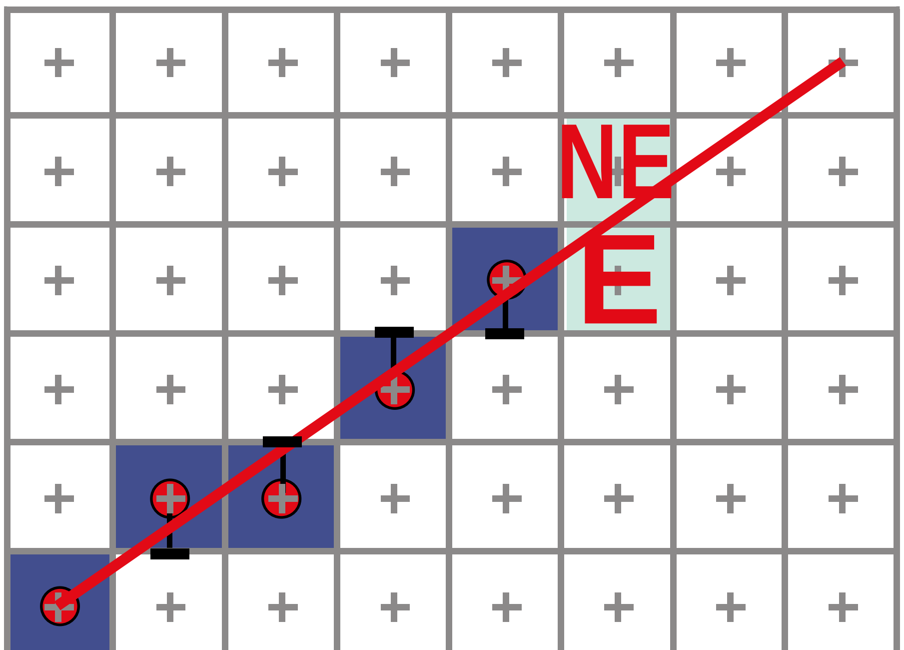
Bresenham's Algorithm (DDA)

- Select pixel vertically closest to line segment
  - intuitive, efficient,
    pixel center always within 0.5 vertically
- Same answer as naive approach



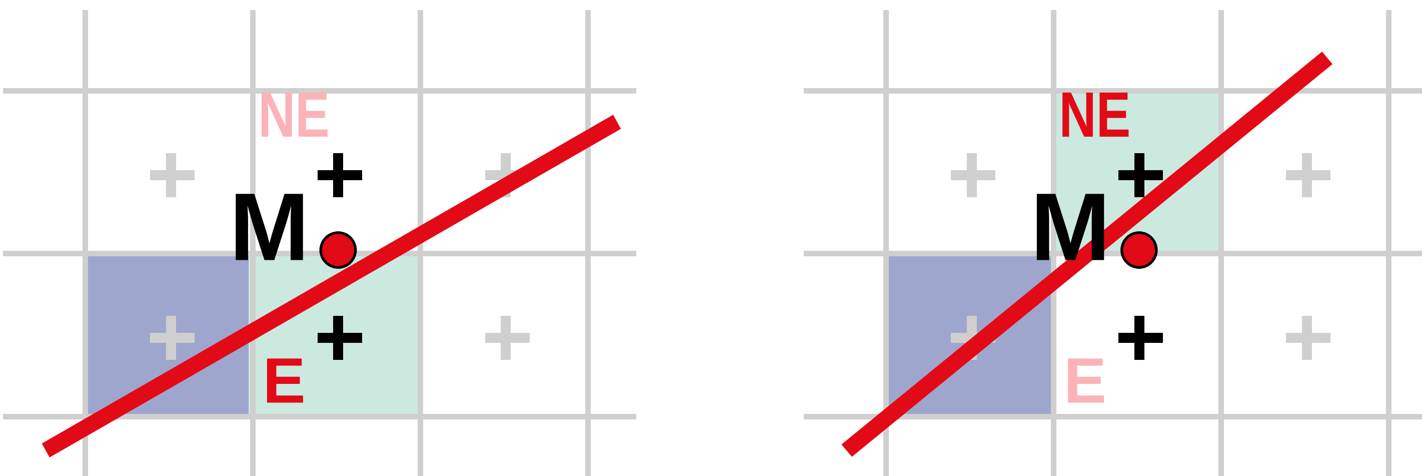
Bresenham's Algorithm (DDA)

- Observation:
  - If we're at pixel P (x_p, y_p), the next pixel must be either E (x_p+1, y_p) or NE (x_p, y_p+1)



Bresenham Step

- Which pixel to choose: E or NE?
  - Choose E if segment passes below or through middle point M
  - Choose NE if segment passes above M



Bresenham Step

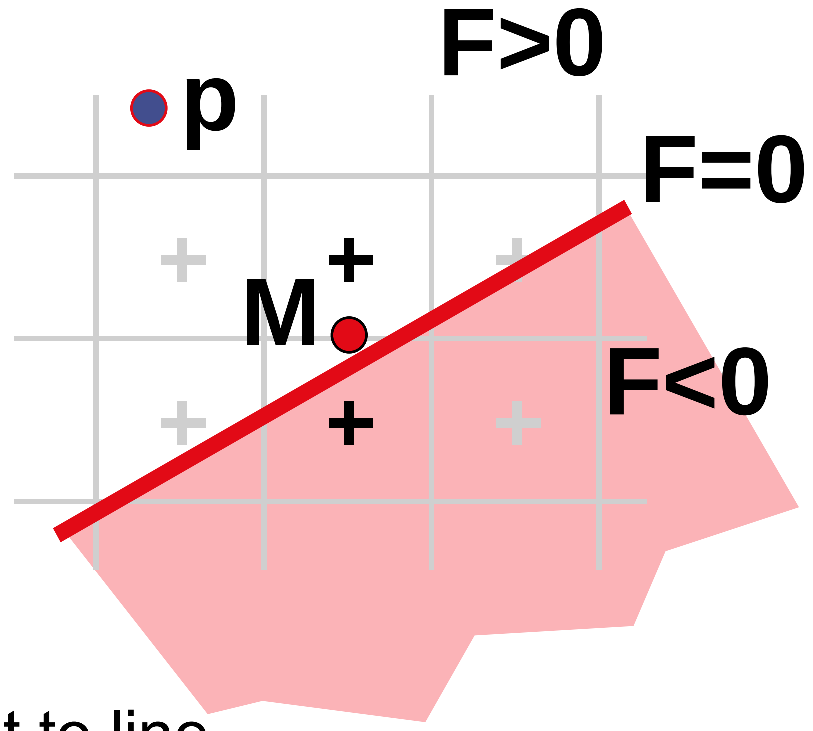
- Use *decision function* D to identify points underlying line L:

  $D(x, y) = y - mx - b$

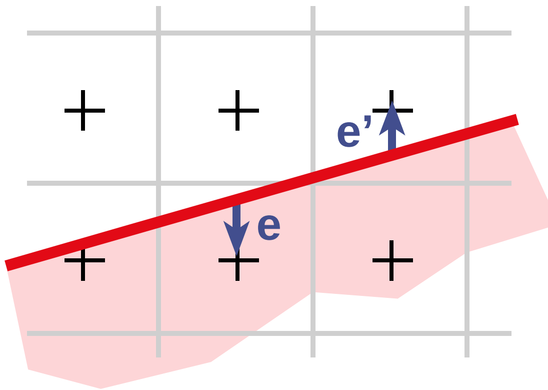
  - positive above L
  - zero on L
  - negative below L

  $D(p_x, p_y) =$

  vertical distance from point to line



Bresenham's Algorithm (DDA)



- Decision Function:

  $D(x, y) = y - mx - b$

- Initialize:

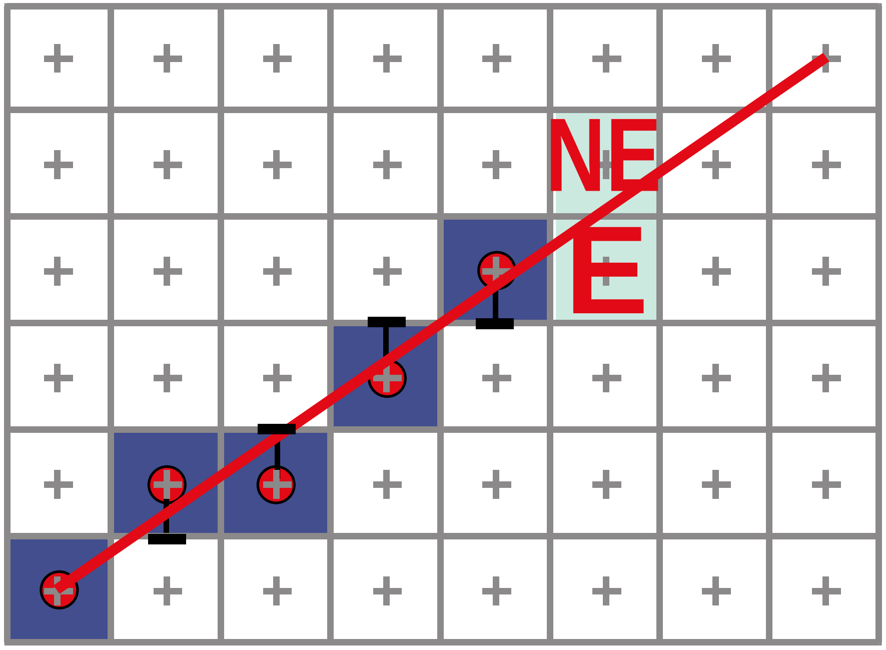
  error term $e = -D(x,y)$

- On each iteration:

  update x: $x' = x+1$
  update e: $e' = e + m$
  if ($e \leq 0.5$): $y' = y$ (choose pixel E)
  if (e > 0.5): y' = y + (choose pixel NE) e' = e - 1

Summary of Bresenham

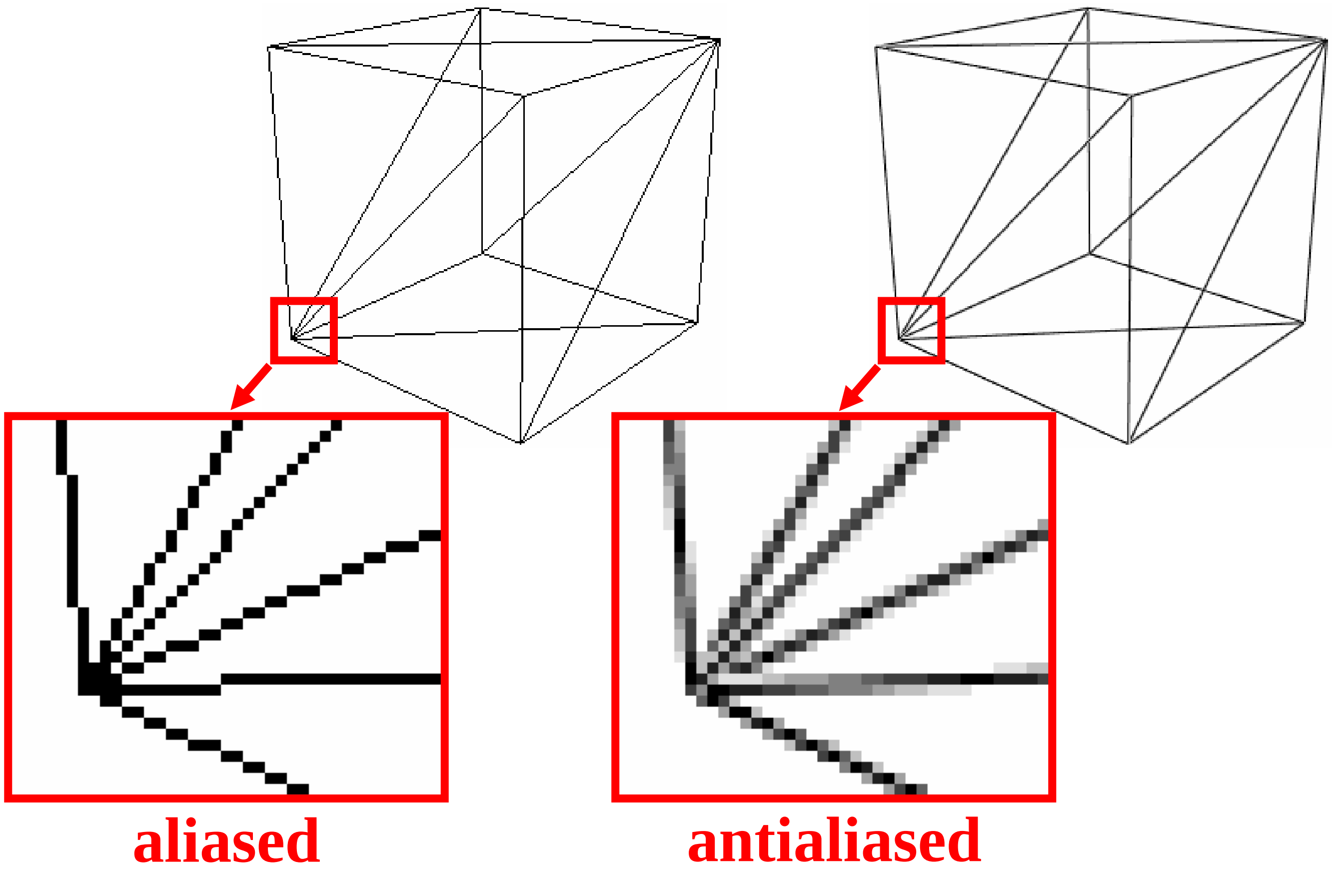
- initialize *x, y, e*
- for (*x* = x1; *x* ≤ x2; *x*++)
  - plot (*x*,*y*)
  - update *x, y, e*



- Generalize to handle all eight octants using symmetry
- Can be modified to use only integer arithmetic

Antialiased Line Rasterization

- Use gray scales to avoid jaggies



Scan Conversion (Rasterization)

Modeling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

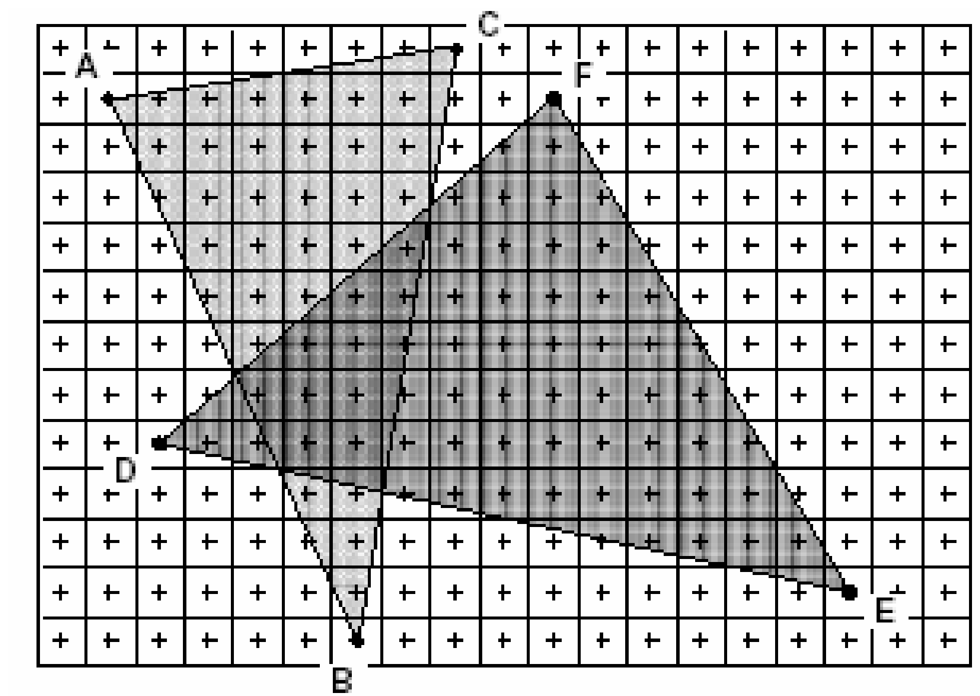
Clipping

Projection (to Screen Space)

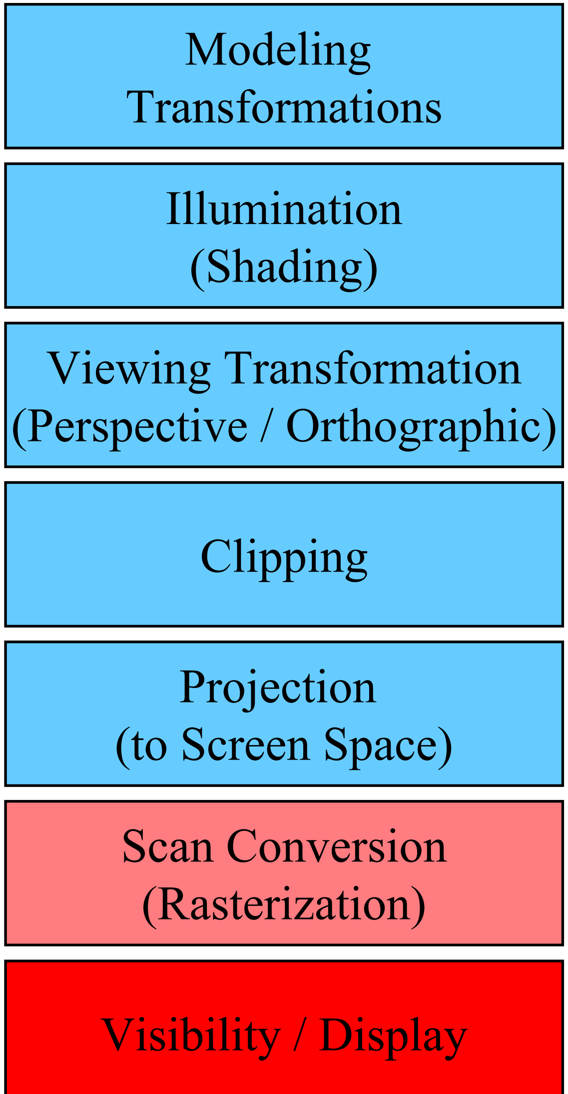
Scan Conversion (Rasterization)

Visibility / Display

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

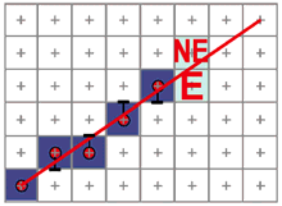


Visibility / Display

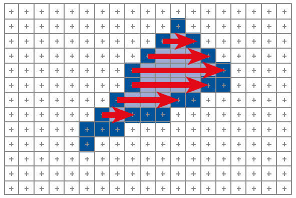


- Each pixel remembers the closest object (depth buffer)
- Almost every step in the graphics pipeline involves a change of coordinate system. Transformations are central to understanding 3D computer graphics.

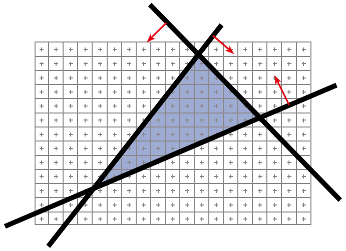
- Line scan-conversion



- Polygon scan conversion

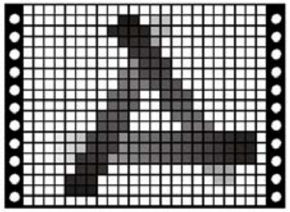


  – smart



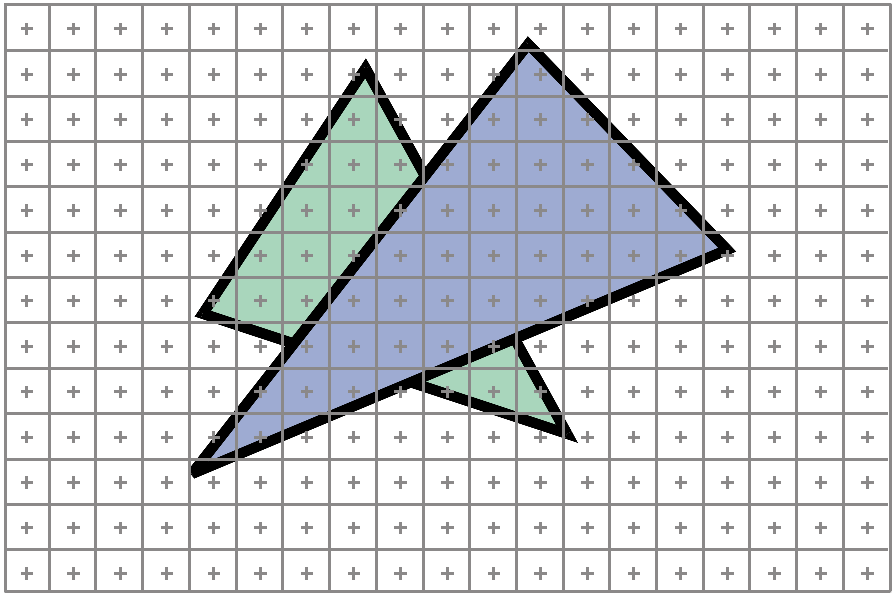
  – back to brute force

- Visibility



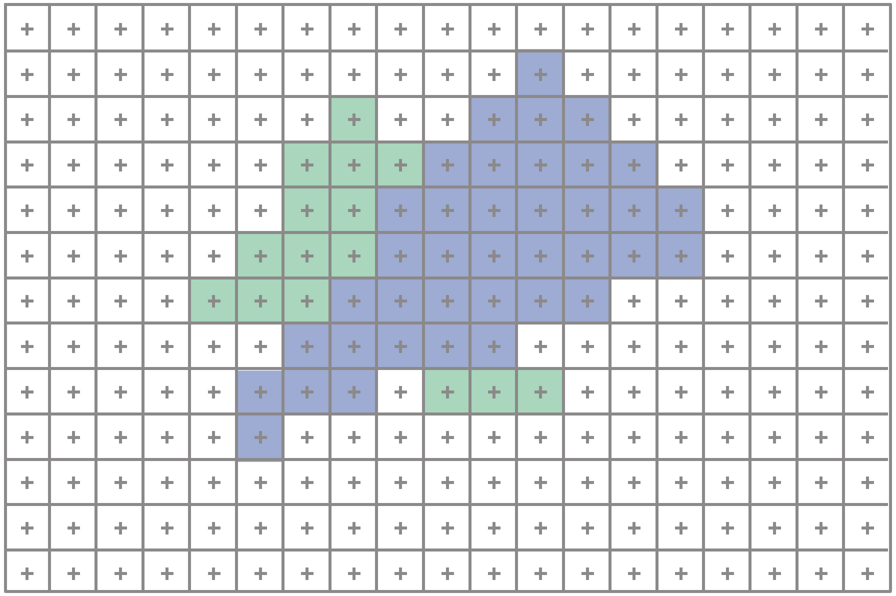
2D Scan Conversion

- Geometric primitive
  - 2D: point, line, polygon, circle...
  - 3D: point, line, polyhedron, sphere...
- Primitives are continuous; screen is discrete



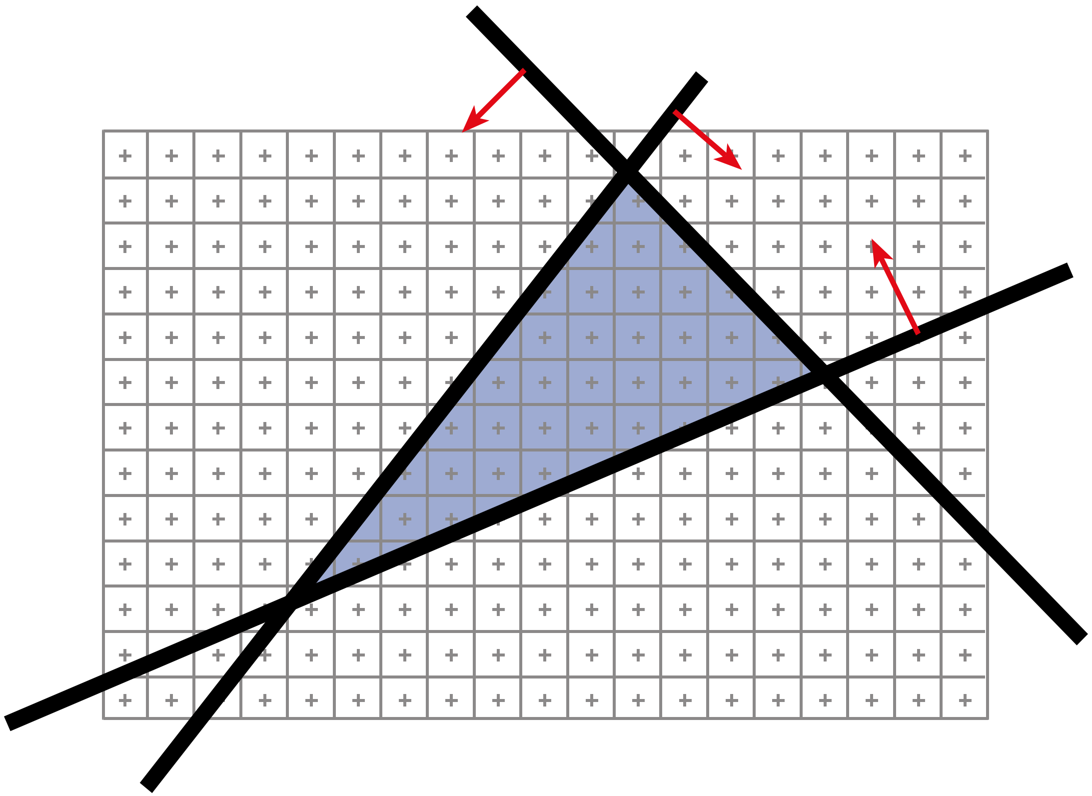
2D Scan Conversion

- Solution: compute discrete approximation
- Scan Conversion:
  algorithms for efficient generation of the samples comprising this approximation



Brute force solution for triangles

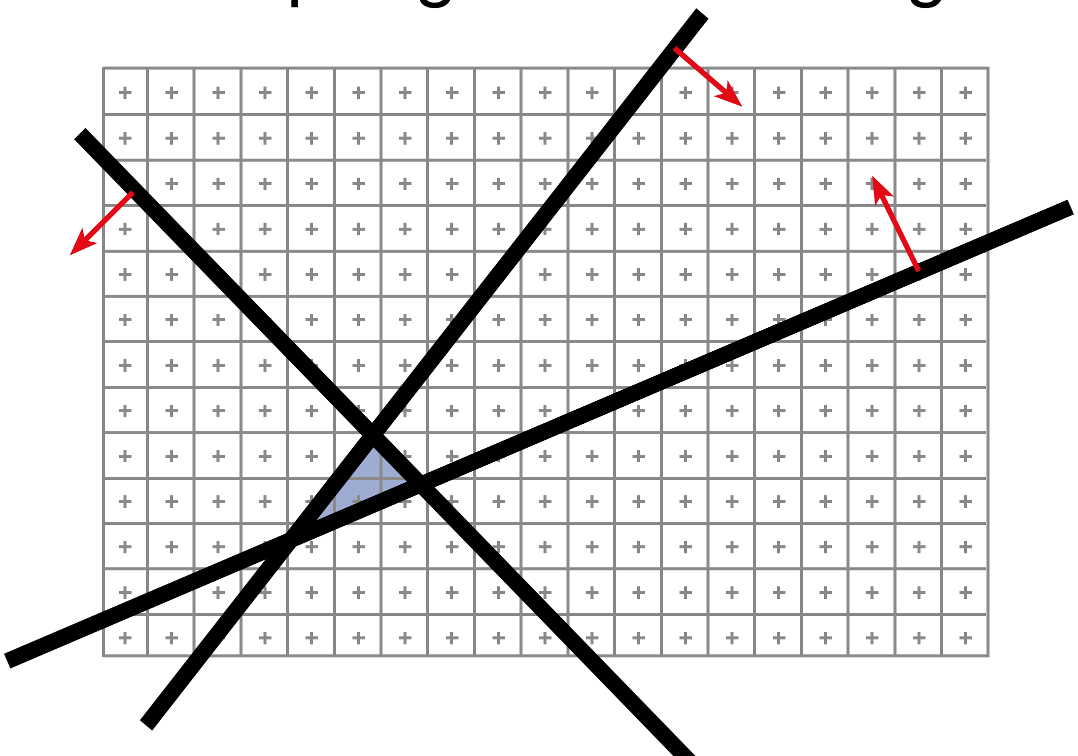
- For each pixel
  - Compute line equations at pixel center
  - “clip” against the triangle



Problem?

Brute force solution for triangles

- For each pixel
  - Compute line equations at pixel center
  - “clip” against the triangle

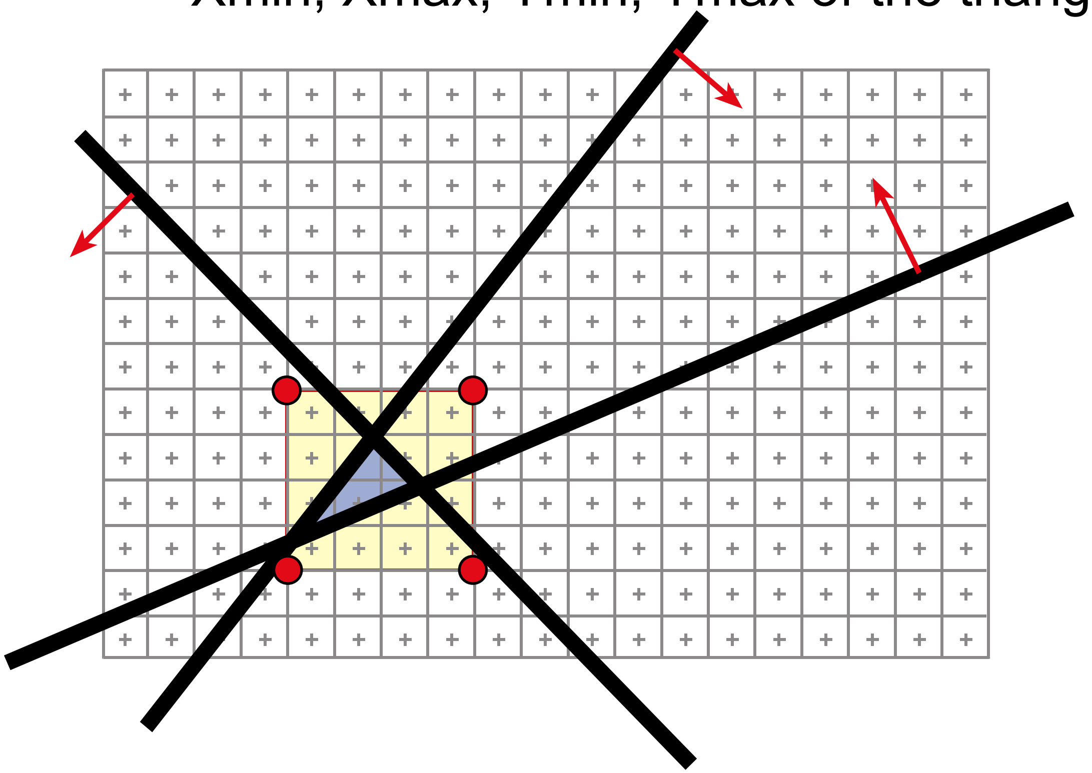


Problem?

If the triangle is small, a lot of useless computation

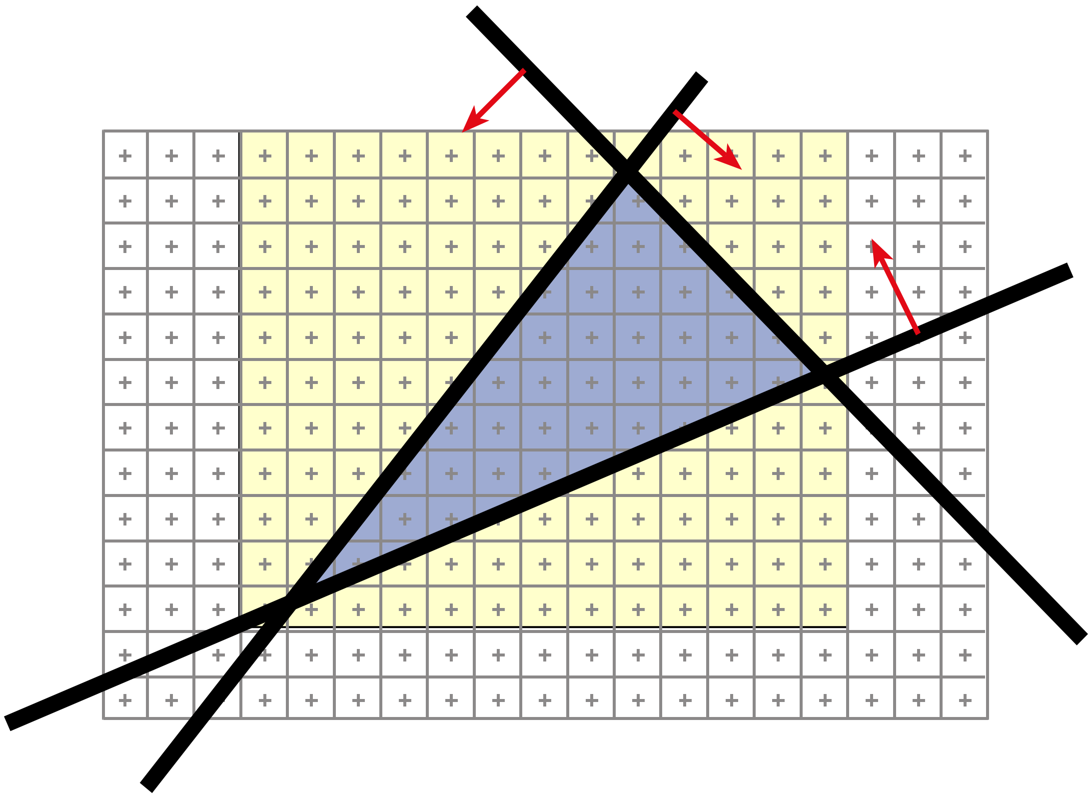
Brute force solution for triangles

- Improvement: Compute only for the *screen bounding box* of the triangle
- How do we get such a bounding box?
  - Xmin, Xmax, Ymin, Ymax of the triangle vertices



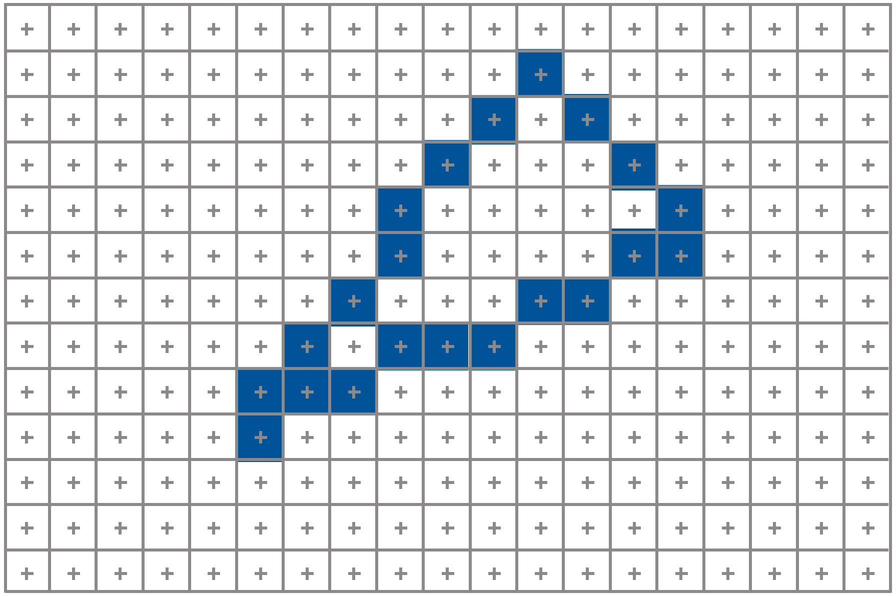
Can we do better? Kind of!

- We compute the line equation for many useless pixels
- What could we do?



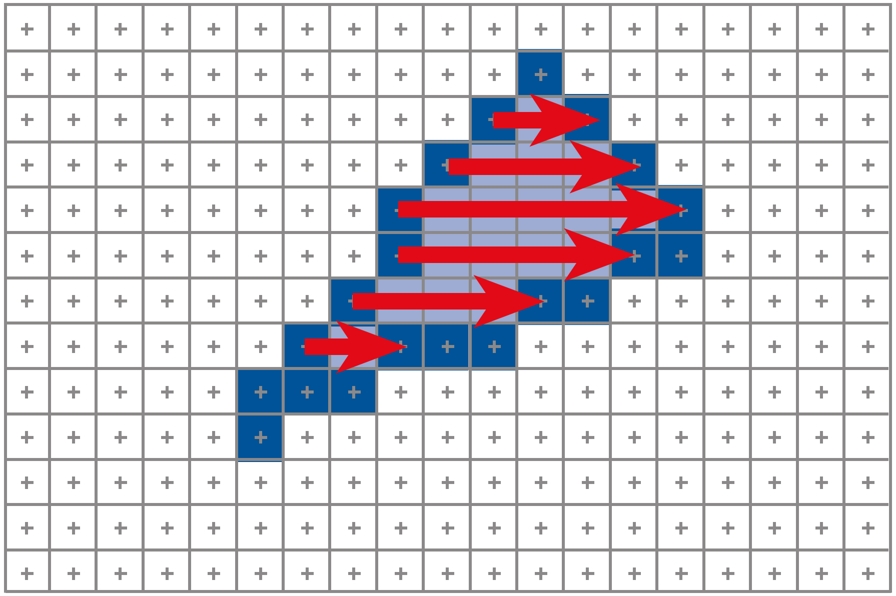
Use line rasterization

- Compute the boundary pixels



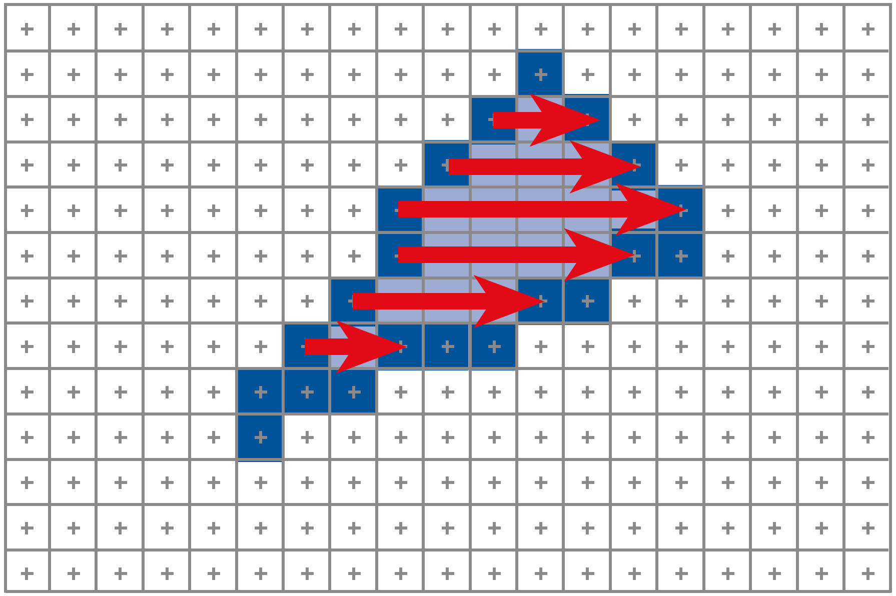
Scan-line Rasterization

- Compute the boundary pixels
- Fill the spans



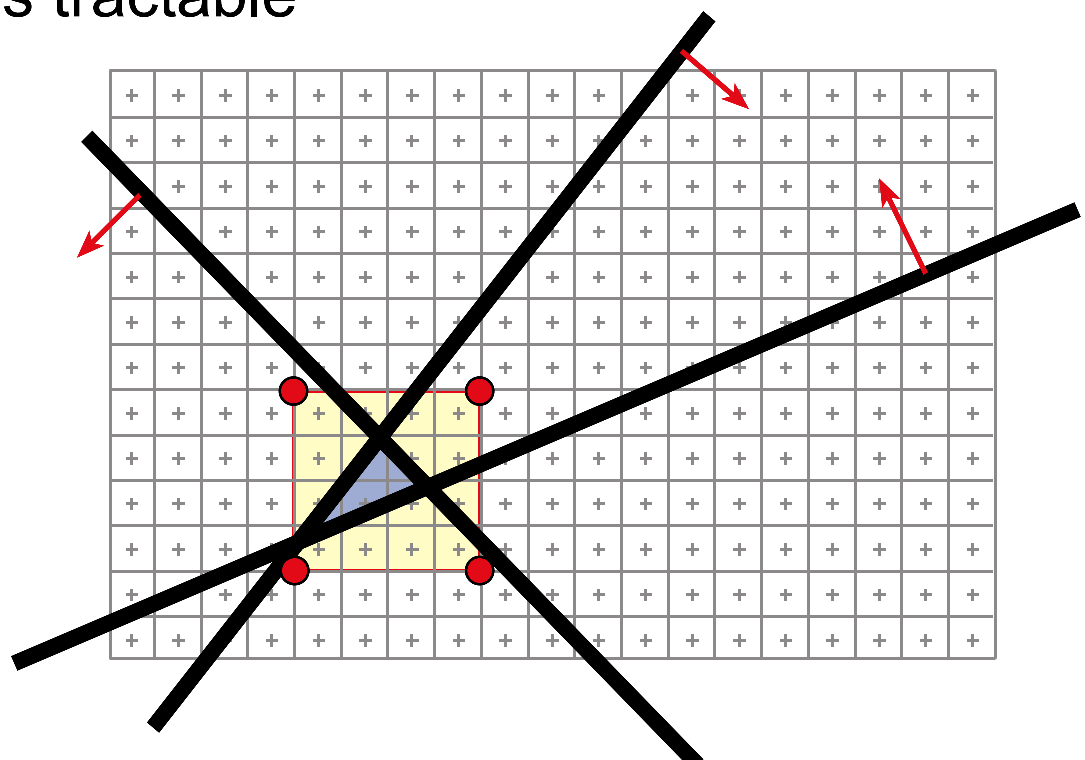
Scan-line Rasterization

- Requires some initial setup to prepare



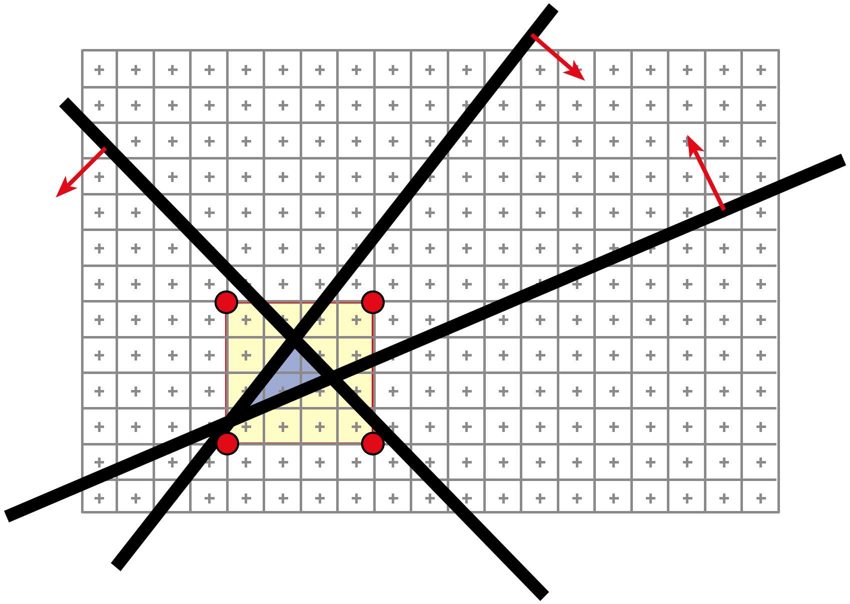
For modern graphics cards

- Triangles are usually very small
- Setup cost are becoming more troublesome
- Clipping is annoying
- Brute force is tractable



Modern rasterization

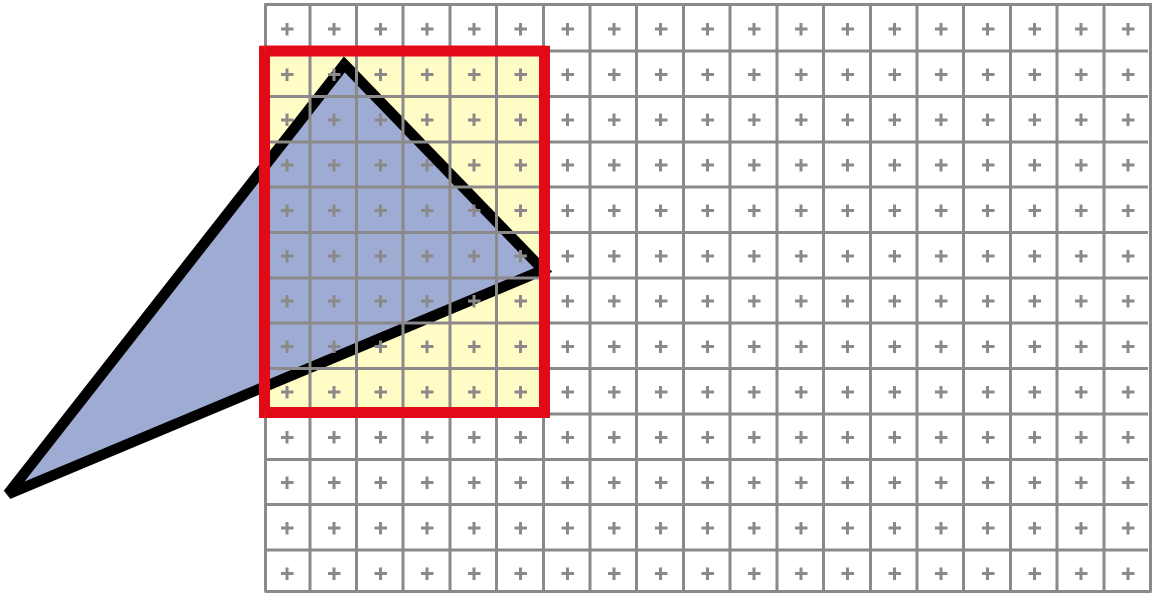
```
For every triangle
   ComputeProjection
   Compute bbox, clip bbox to screen limits
   For all pixels in bbox
      Compute line equations
      If all line equations>0 //pixel [x,y] in triangle
            Framebuffer[x,y]=triangleColor
```



Modern rasterization

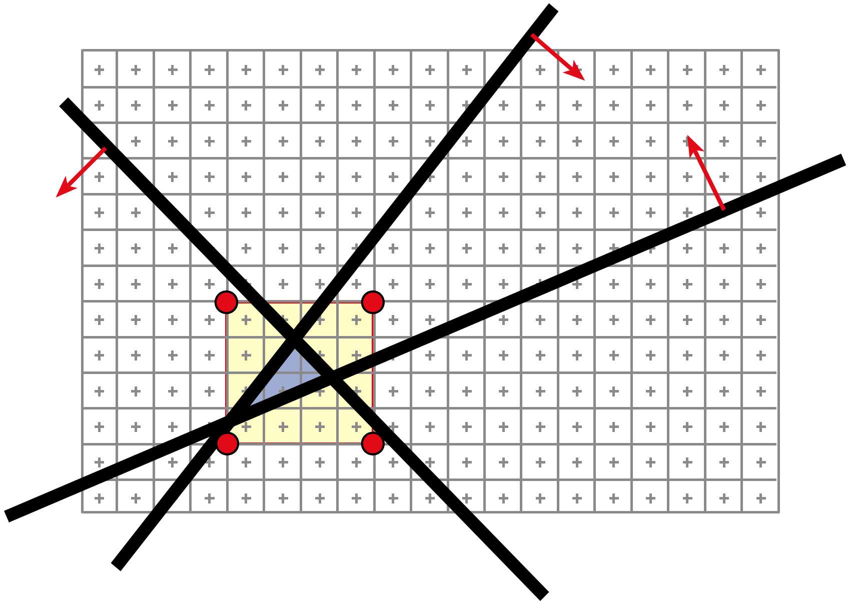
```
For every triangle
    ComputeProjection
    Compute bbox, clip bbox to screen limits
    For all pixels in bbox
        Compute line equations
        If all line equations>0  //pixel [x,y] in triangle
                Framebuffer[x,y]=triangleColor
```

- Note that Bbox clipping is trivial



Can we do better?

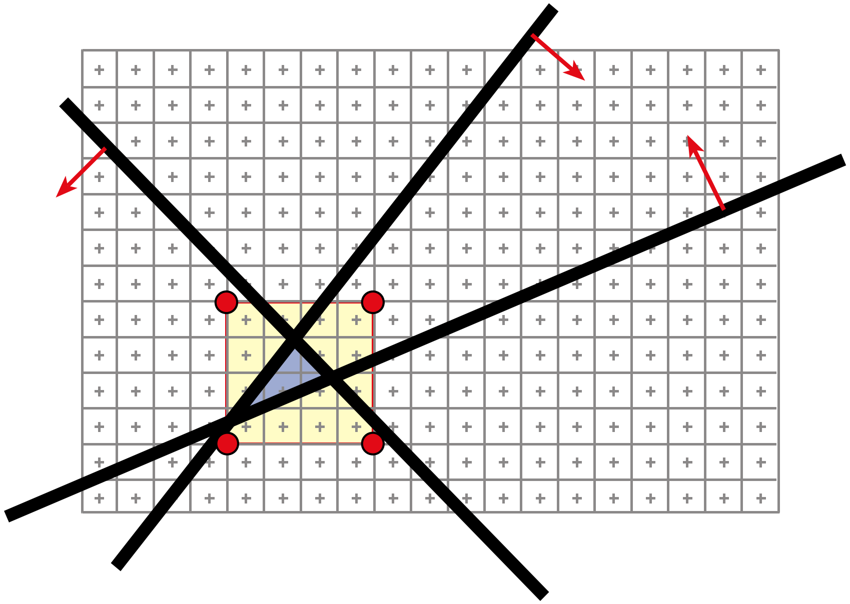
```
For every triangle
    ComputeProjection
    Compute bbox, clip bbox to screen limits
    For all pixels in bbox
        Compute line equations
        If all line equations>0  //pixel [x,y] in triangle
            Framebuffer[x,y]=triangleColor
```



Can we do better?

```
For every triangle
     ComputeProjection
     Compute bbox, clip bbox to screen limits
     For all pixels in bbox
          Compute line equations ax+by+c
          If all line equations>0 //pixel [x,y] in triangle
                    Framebuffer[x,y]=triangleColor
```

- We don’t need to recompute line equation from scratch



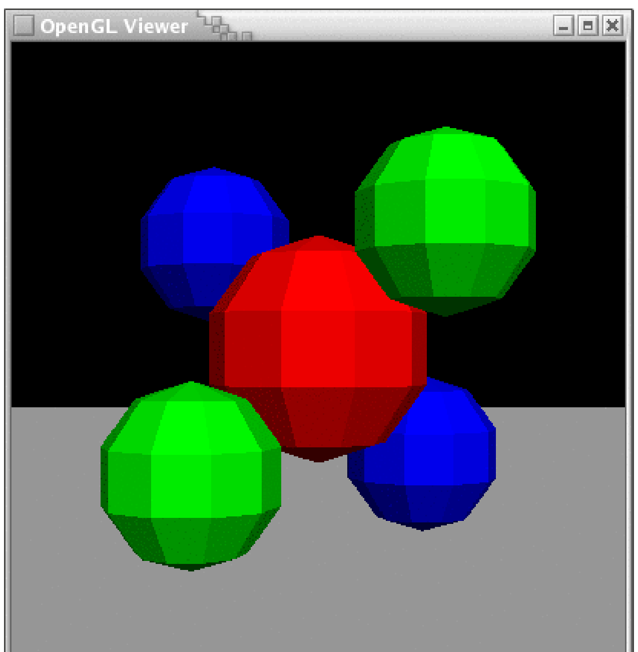
Can we do better?

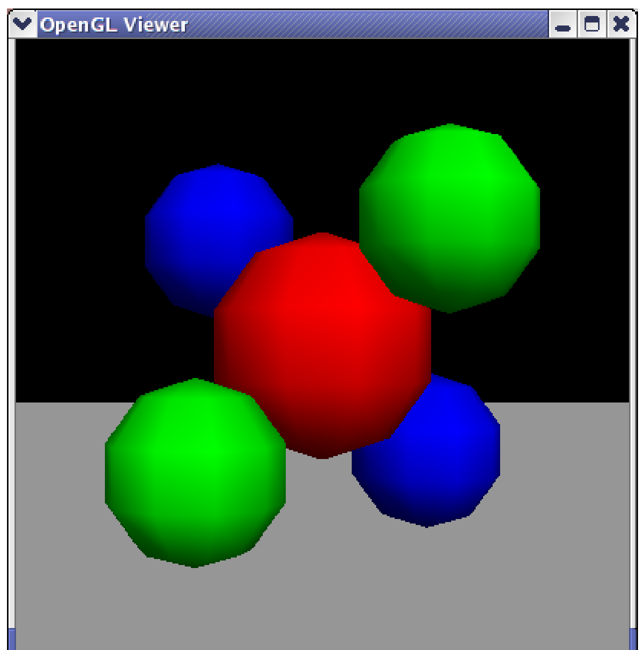
```
For every triangle
    ComputeProjection
    Compute bbox, clip bbox to screen limits
    Setup line eq
        compute a_i dx, b_i dy for the 3 lines
        Initialize line eq, values for bbox corner
            L_i=a_i x0+b_i y+c_i
    For all scanline y in bbox
        For 3 lines, update Li
        For all x in bbox
            Increment line equations: Li+=adx
        If all Li>0 //pixel [x,y] in triangle
                Framebuffer[x,y]=triangleColor
```

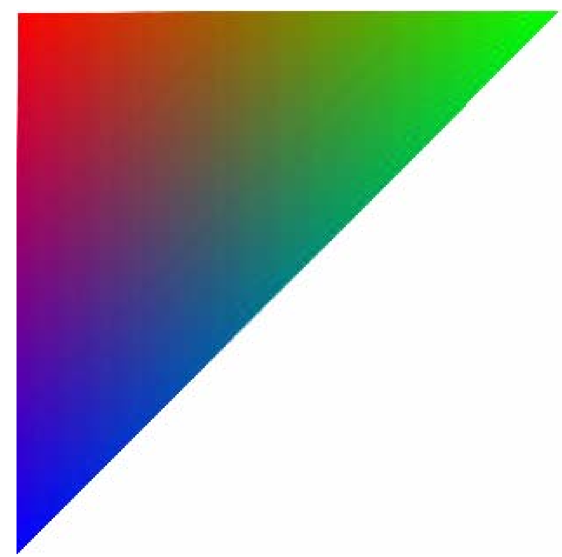
- We save one multiplication per pixel

Adding Gouraud shading

- Interpolate colors of the 3 vertices
- Linear interpolation

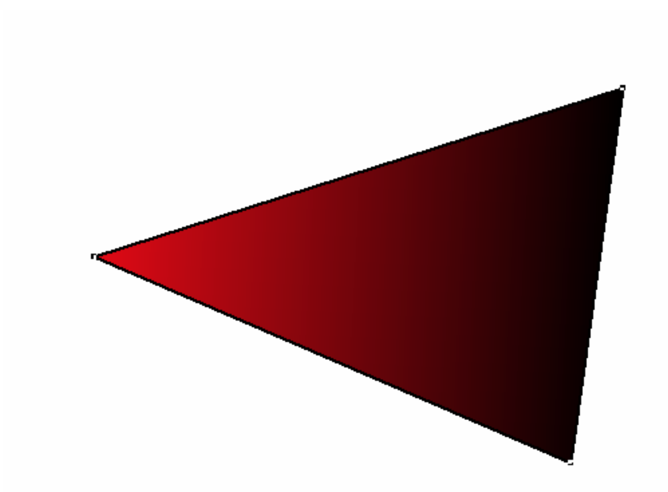


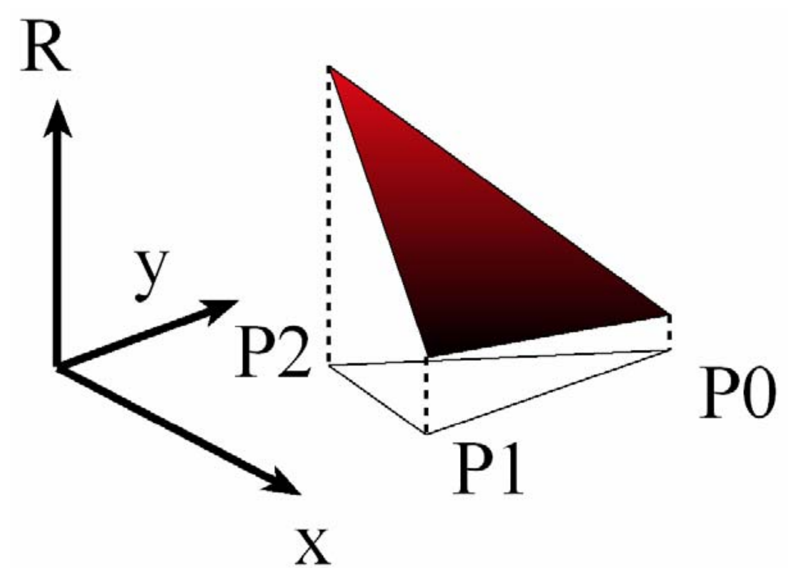




Adding Gouraud shading

- Interpolate colors of the 3 vertices
- Linear interpolation, e.g. for R channel:
  - $R=a_R x+b_R y+c_R$
  - Such that R[x0,y0]=R0; R[x1, y1]=R1; R[x2,y2]=R2
  - Same as a plane equation in (x,y,R)



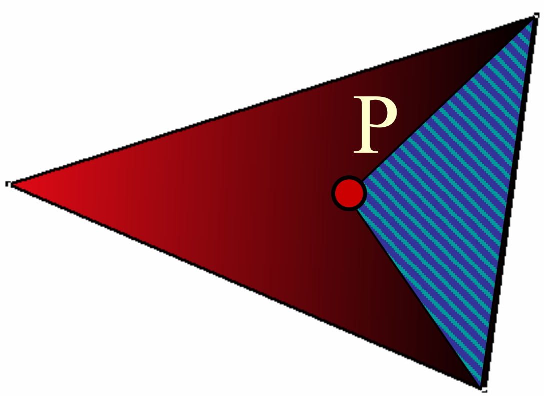


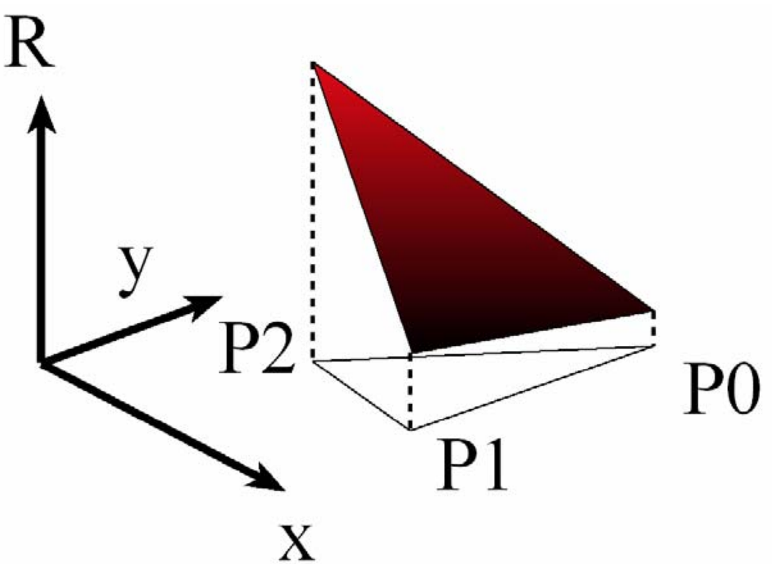
Adding Gouraud shading

```
Interpolate colors
For every triangle
   ComputeProjection
   Compute bbox, clip bbox to screen limits
   Setup line eq
   Setup color equation
   For all pixels in bbox
      Increment line equations
      Increment color equation
      If all Li>0  //pixel [x,y] in triangle
            Framebuffer[x,y]=interpolatedColor
```

Adding Gouraud shading

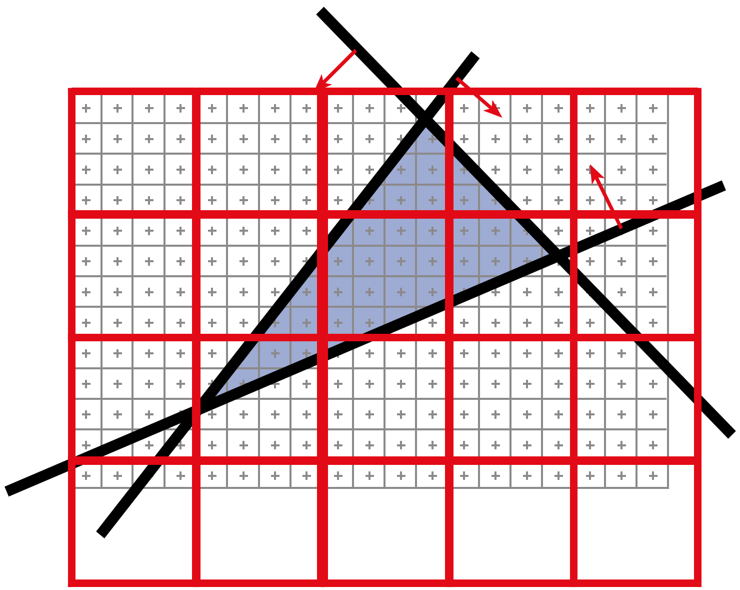
- Interpolate colors of the 3 vertices
- Other solution: use barycentric coordinates
- $R = \alpha R_0 + \beta R_1 + \gamma R_2$
- Such that $P = \alpha P_0 + \beta P_1 + \gamma P_2$





In the modern hardware

- Edge eq. in homogeneous coordinates [x, y, w]
- Tiles to add a mid-level granularity
  - Early rejection of tiles
  - Memory access coherence



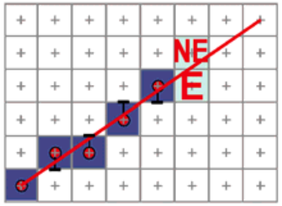
Ref

- Henry Fuchs, Jack Goldfeather, Jeff Hultquist, Susan Spach, John Austin, Frederick Brooks, Jr., John Eyles and John Poulton, “Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes”, Proceedings of SIGGRAPH ‘85 (San Francisco, CA, July 22–26, 1985). In *Computer Graphics*, v19n3 (July 1985), ACM SIGGRAPH, New York, NY, 1985.
- Juan Pineda, “A Parallel Algorithm for Polygon Rasterization”, Proceedings of SIGGRAPH ‘88 (Atlanta, GA, August 1–5, 1988). In *Computer Graphics*, v22n4 (August 1988), ACM SIGGRAPH, New York, NY, 1988. Figure 7: Image from the spinning teapot performance test.
- Triangle Scan Conversion using 2D Homogeneous Coordinates, Marc Olano Trey Greer http://www.cs.unc.edu/~olano/papers/2dh-tri/2dh-tri.pdf

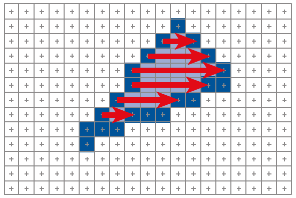
Take-home message

- The appropriate algorithm depends on
  - Balance between various resources (CPU, memory, bandwidth)
  - The input (size of triangles, etc.)
- Smart algorithms often have initial preprocess
  - Assess whether it is worth it
- To save time, identify redundant computation
  - Put outside the loop and interpolate if needed

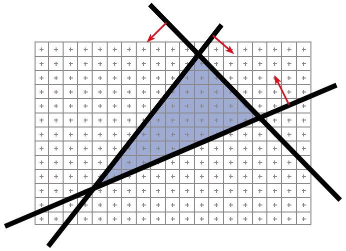
- Line scan-conversion



- Polygon scan conversion

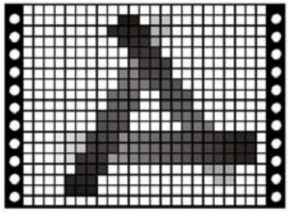


  – smart



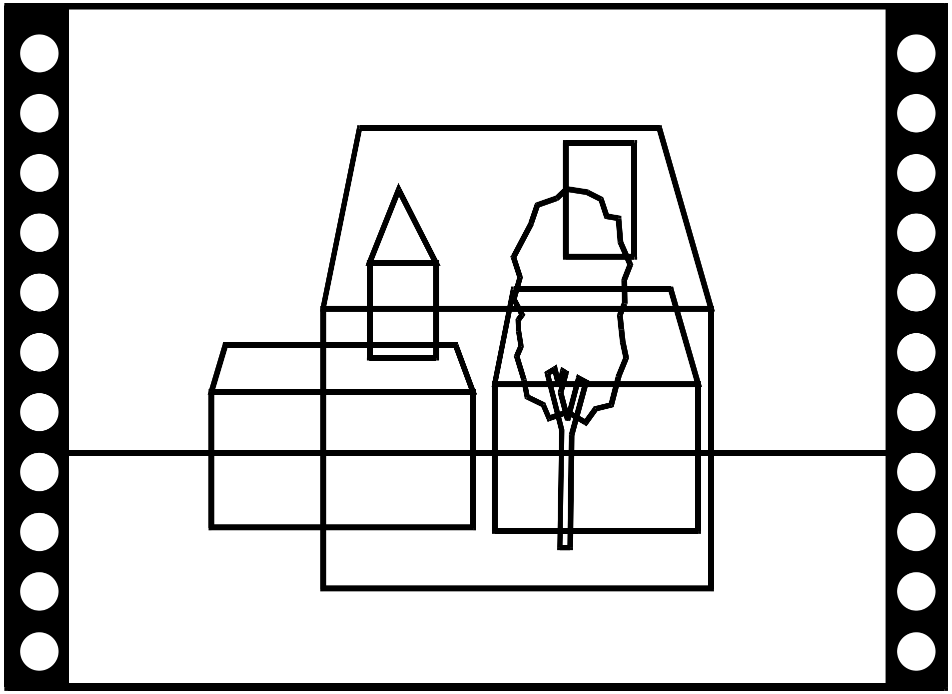
  – back to brute force

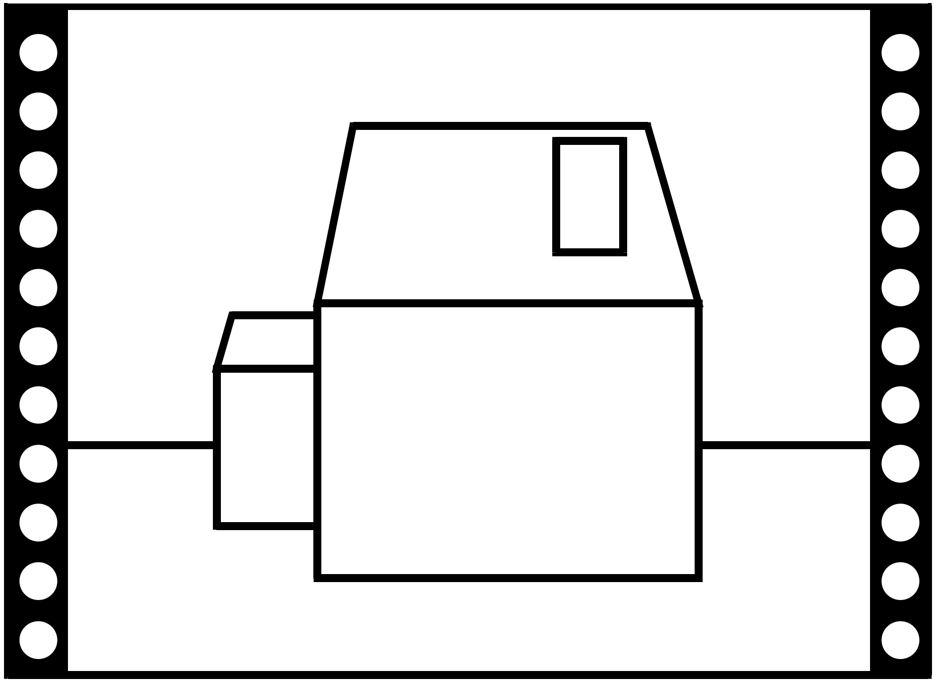
- Visibility



Visibility

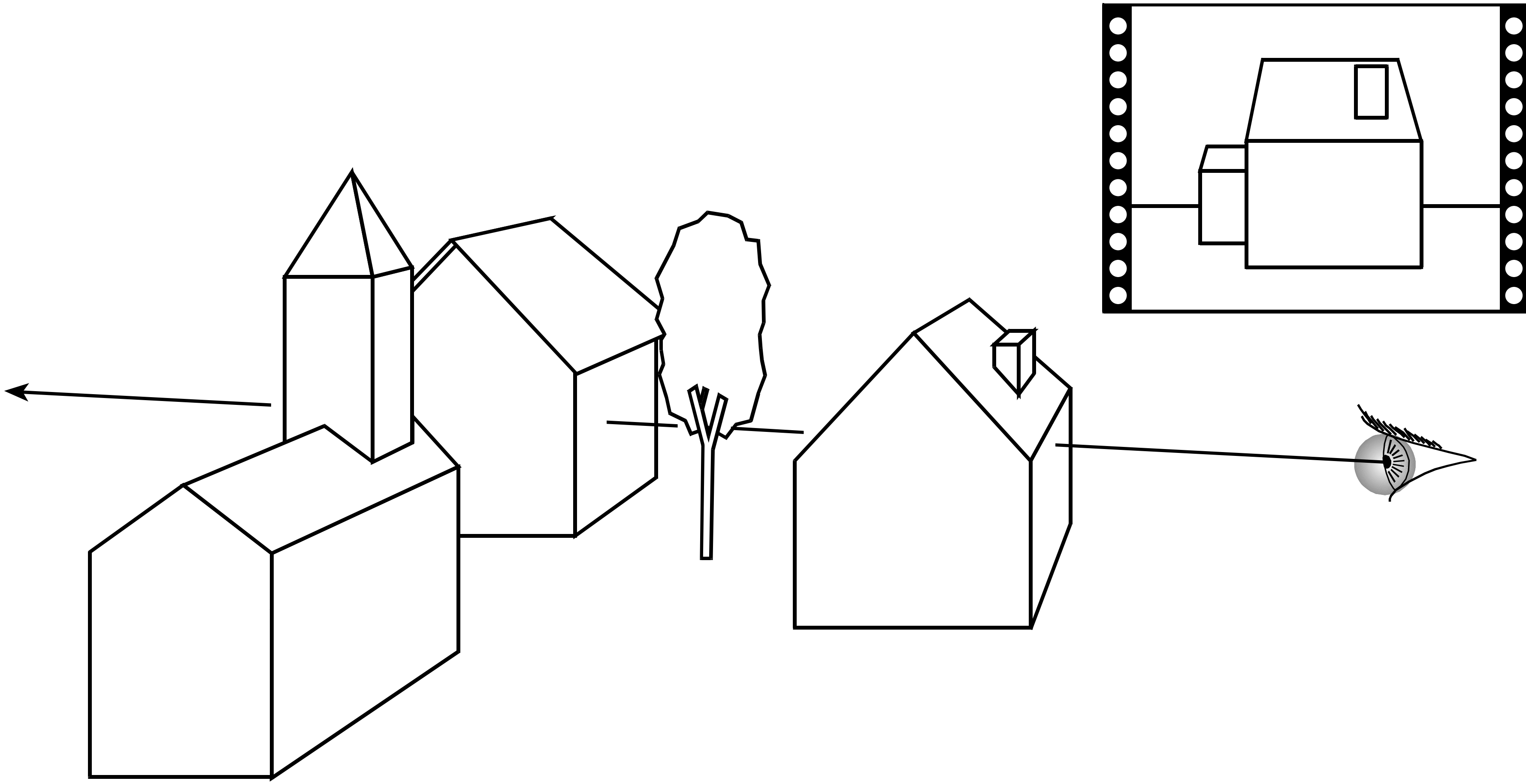
- How do we know which parts are visible/in front?





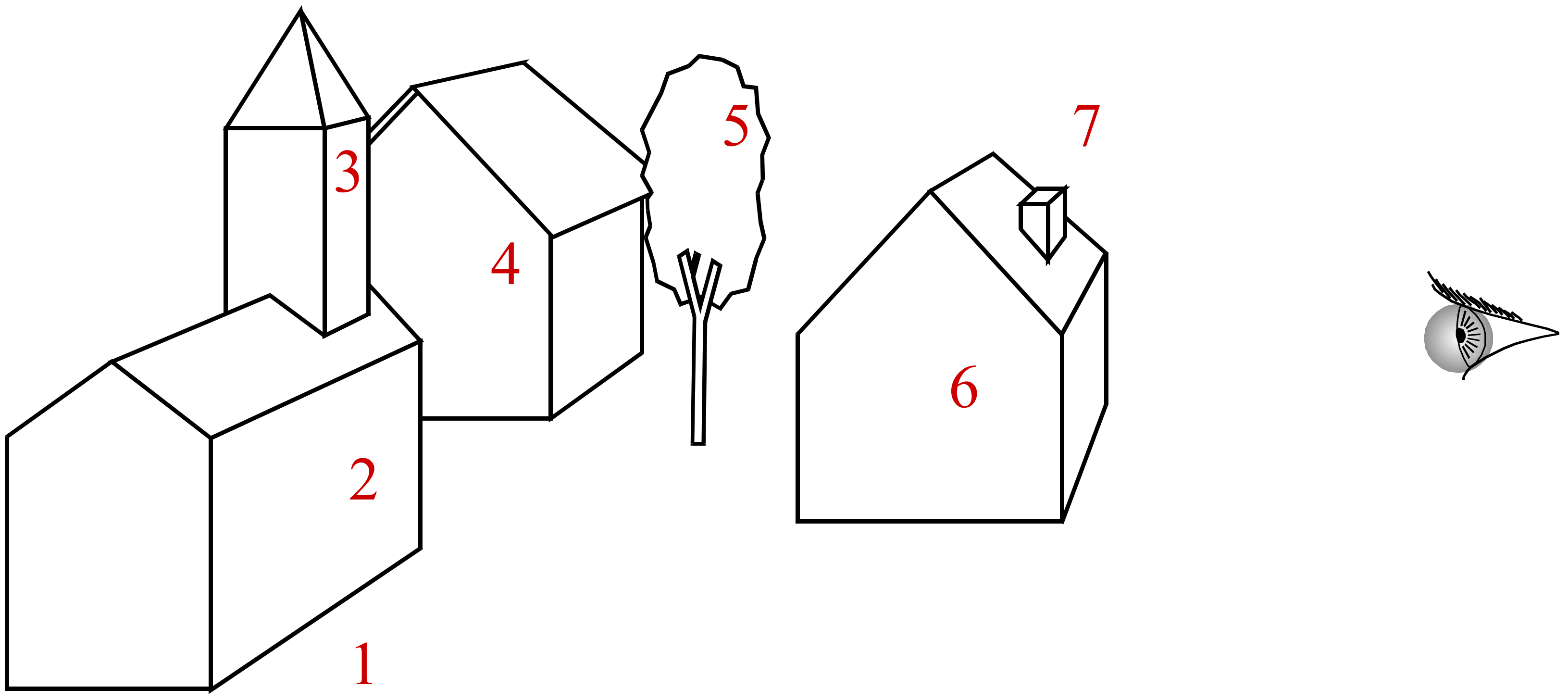
Ray Casting

- Maintain intersection with closest object

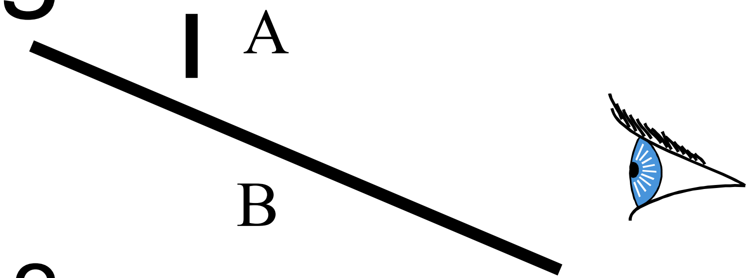


Painter's algorithm

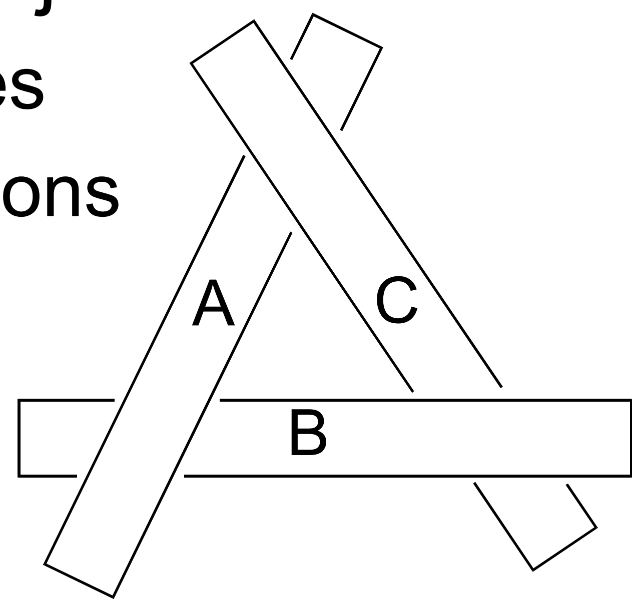
- Draw back-to-front
- How do we sort objects?
- Can we always sort objects?



Painter's algorithm



- Draw back-to-front
- How do we sort objects?
- Can we always sort objects?
  - No, there can be cycles
  - Requires to split polygons

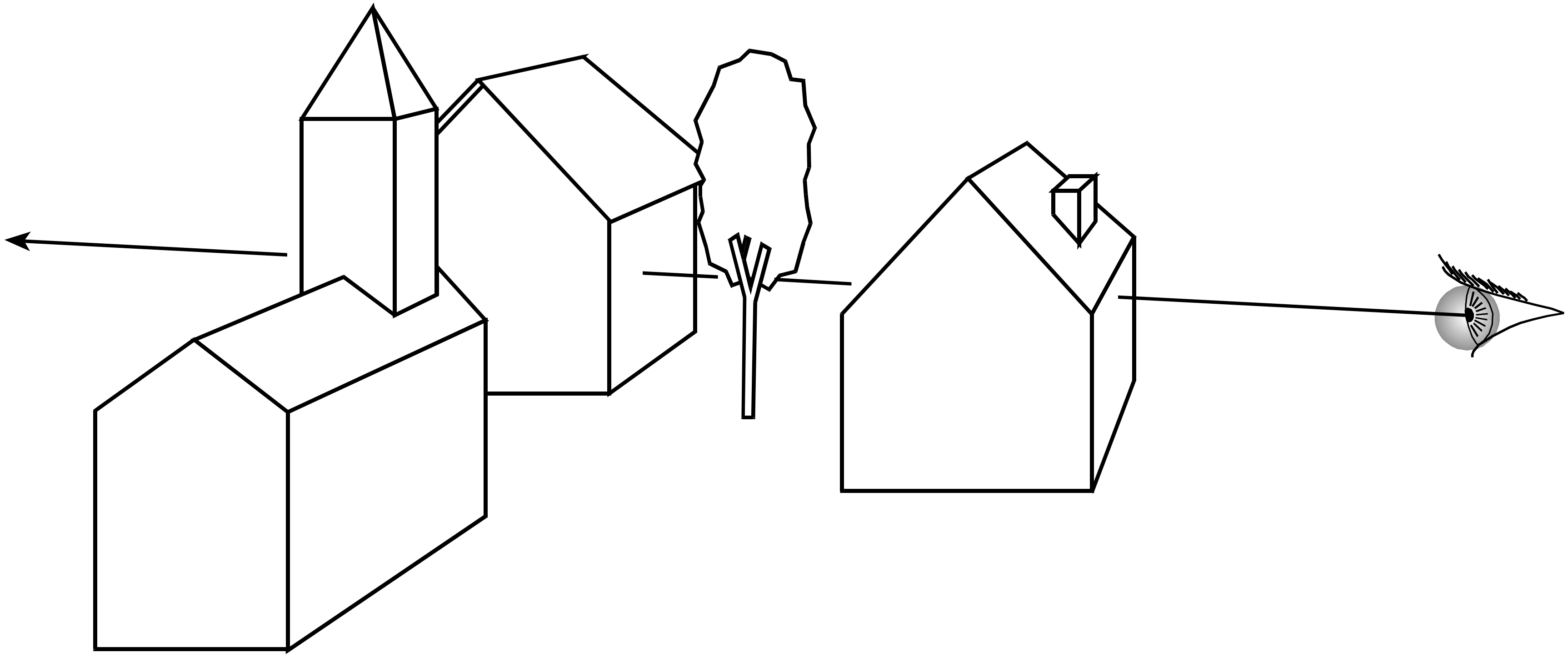


Painter’s algorithm

- Old solution for hidden-surface removal
  - Good because ordering is useful for other operations (transparency, antialiasing)
- But
  - Ordering is tough
  - Cycles
  - Must be done by CPU
- Hardly used now
- But some sort of partial ordering is sometimes useful
  - Usuall front-to-back
  - To make sure foreground is rendered first
  - For transparency

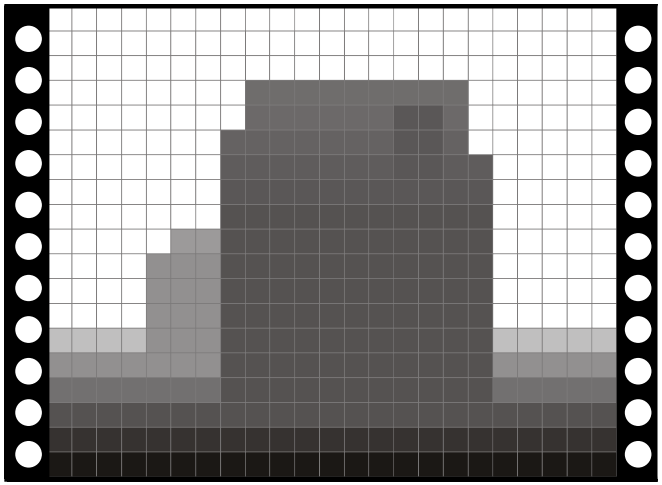
Visibility

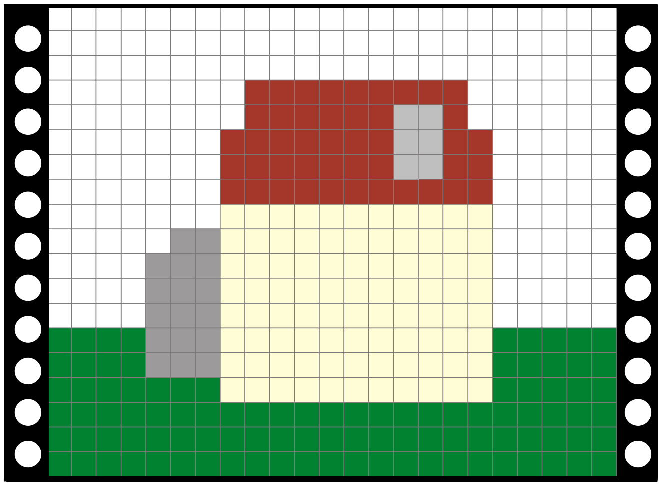
- In ray casting, use intersection with closest t
- Now we have swapped the loops (pixel, object)
- How do we do?



Z buffer

- In addition to frame buffer (R, G, B)
- Store distance to camera (z-buffer)
- Pixel is updated only if new z is closer than z-buffer value

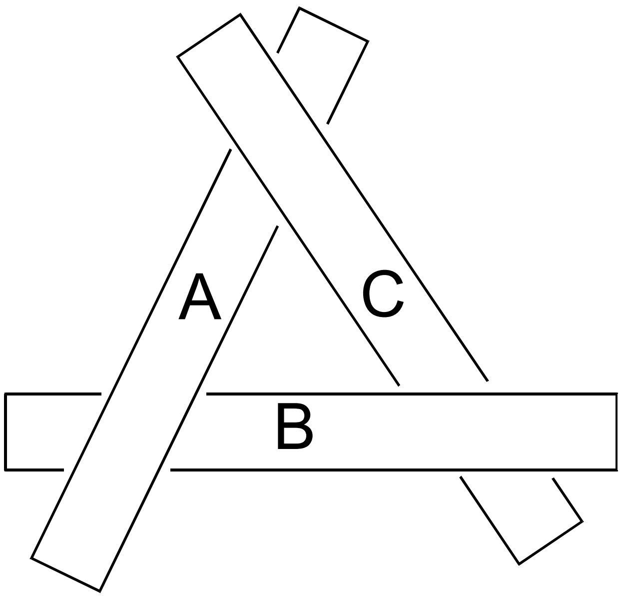


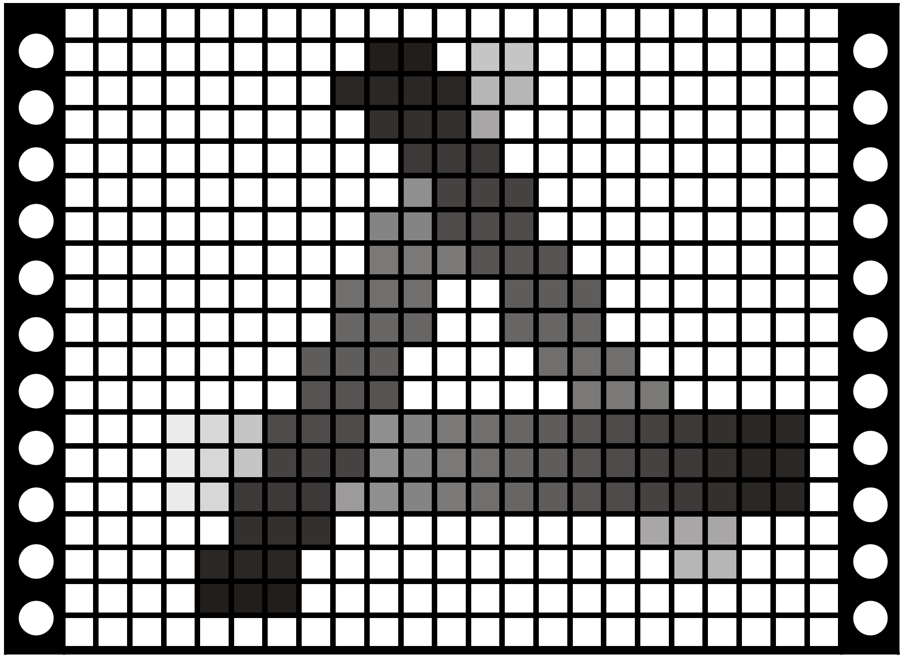


Z-buffer pseudo code

```
For every triangle
   Compute Projection, color at vertices
   Setup line equations
   Compute bbox, clip bbox to screen limits
   For all pixels in bbox
      Increment line equations
      Compute currentZ
      Increment currentColor
      If all line equations>0  //pixel [x,y] in triangle
         If currentZ<zBuffer[x,y]  //pixel is visible
            Framebuffer[x,y]=currentColor
            zBuffer[x,y]=currentZ
```

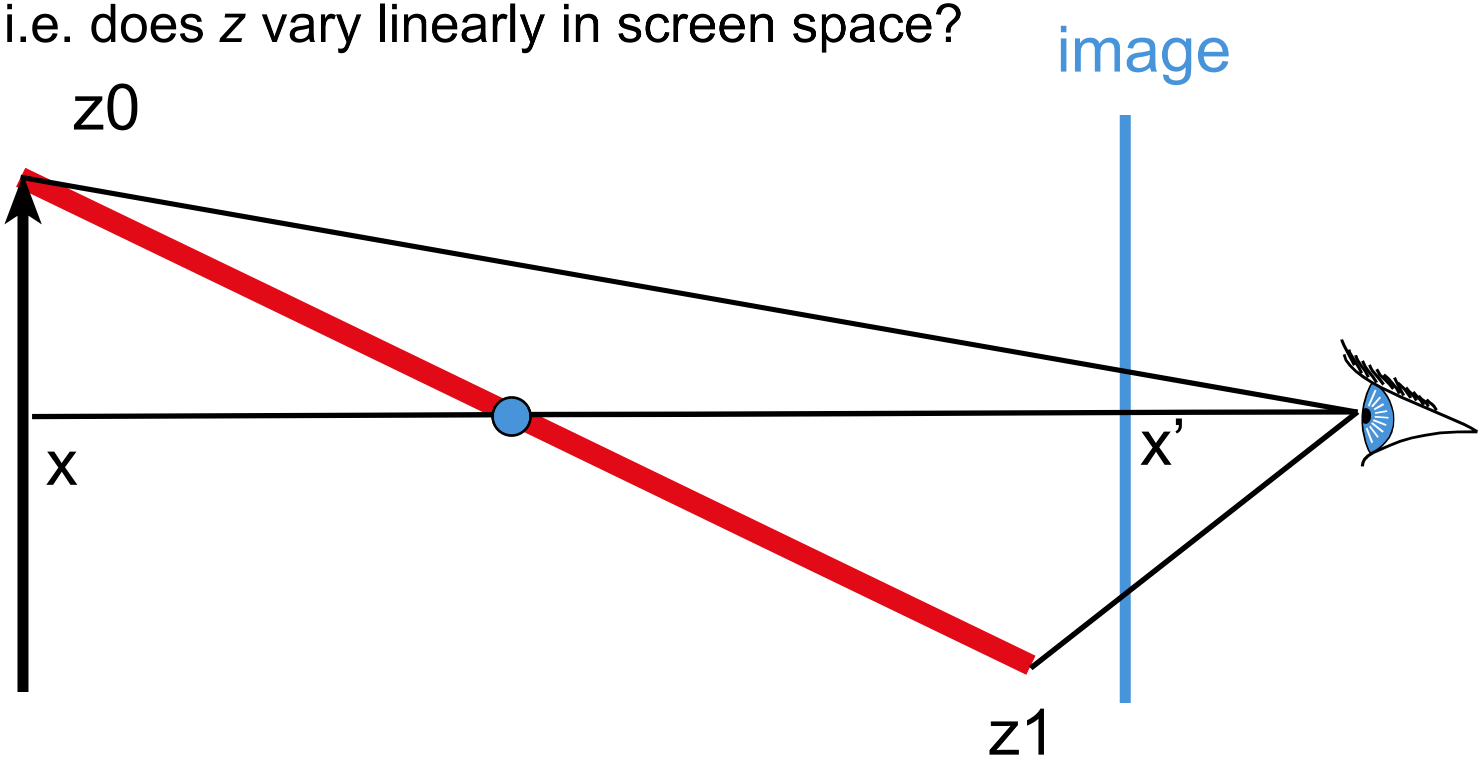
Works for hard cases!





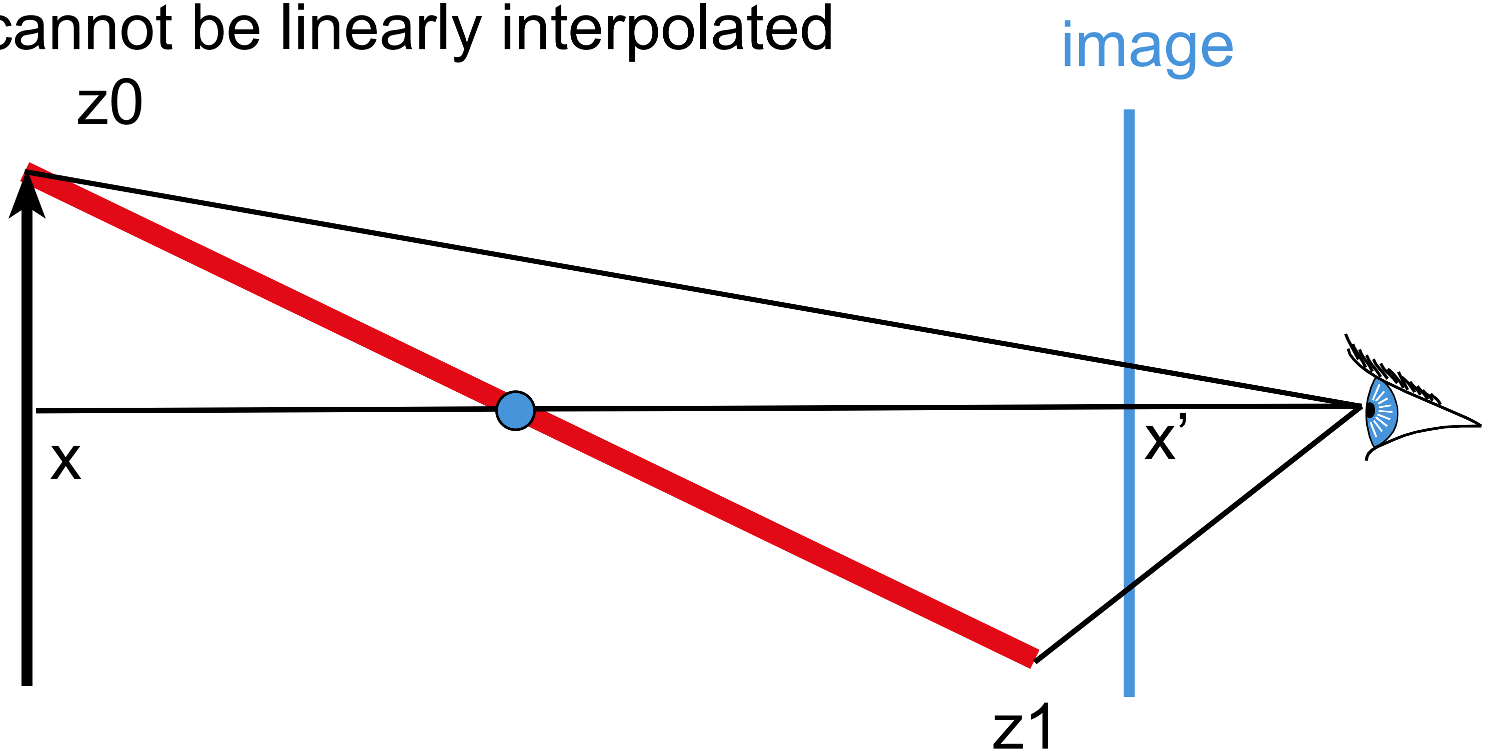
What exactly do we store

- Floating point distance
- Can we interpolate *z* in screen space?
  - i.e. does *z* vary linearly in screen space?



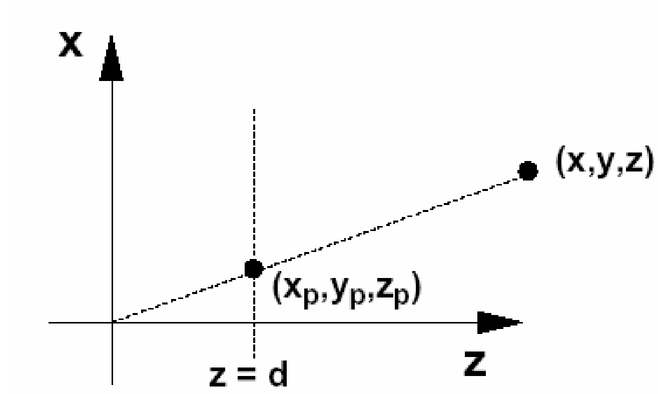
Z interpolation

- X'=x/z
- Hyperbolic variation
- Z cannot be linearly interpolated



Simple Perspective Projection

- Project all points along the *z* axis to the *z* = *d* plane, eyepoint at the origin

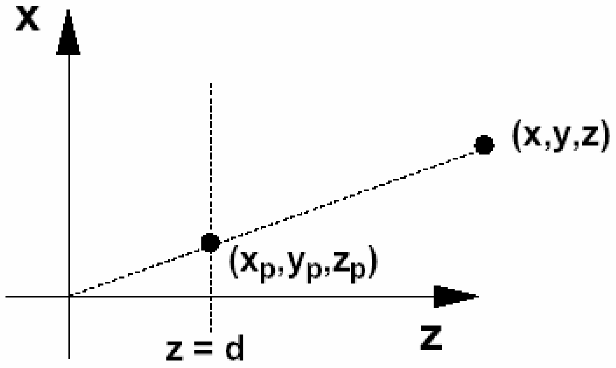


homogenize

$$\begin{pmatrix} x * d / z \\ y * d / z \\ d \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ z / d \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Yet another Perspective Projection

- Change the z component
- Compute d/z
- Can be linearly interpolated



homogenize

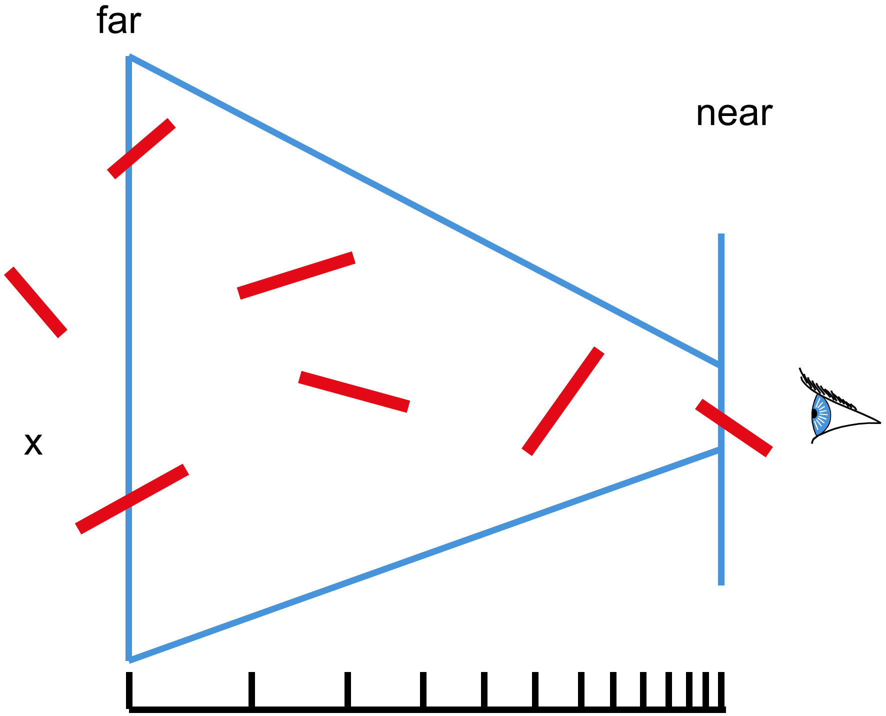
$$\begin{pmatrix} x * d / z \\ y * d / z \\ d/z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \\ z / d \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Advantages of 1/z

- Can be interpolated linearly in screen space
- Puts more precision for close objects
- Useful when using integers
  - more precision where perceptible

Integer z-buffer

- Use 1/z to have more precision in the foreground
- Set a near and far plane
  - 1/z values linearly encoded between 1/near and 1/far
- Careful, test direction is reversed

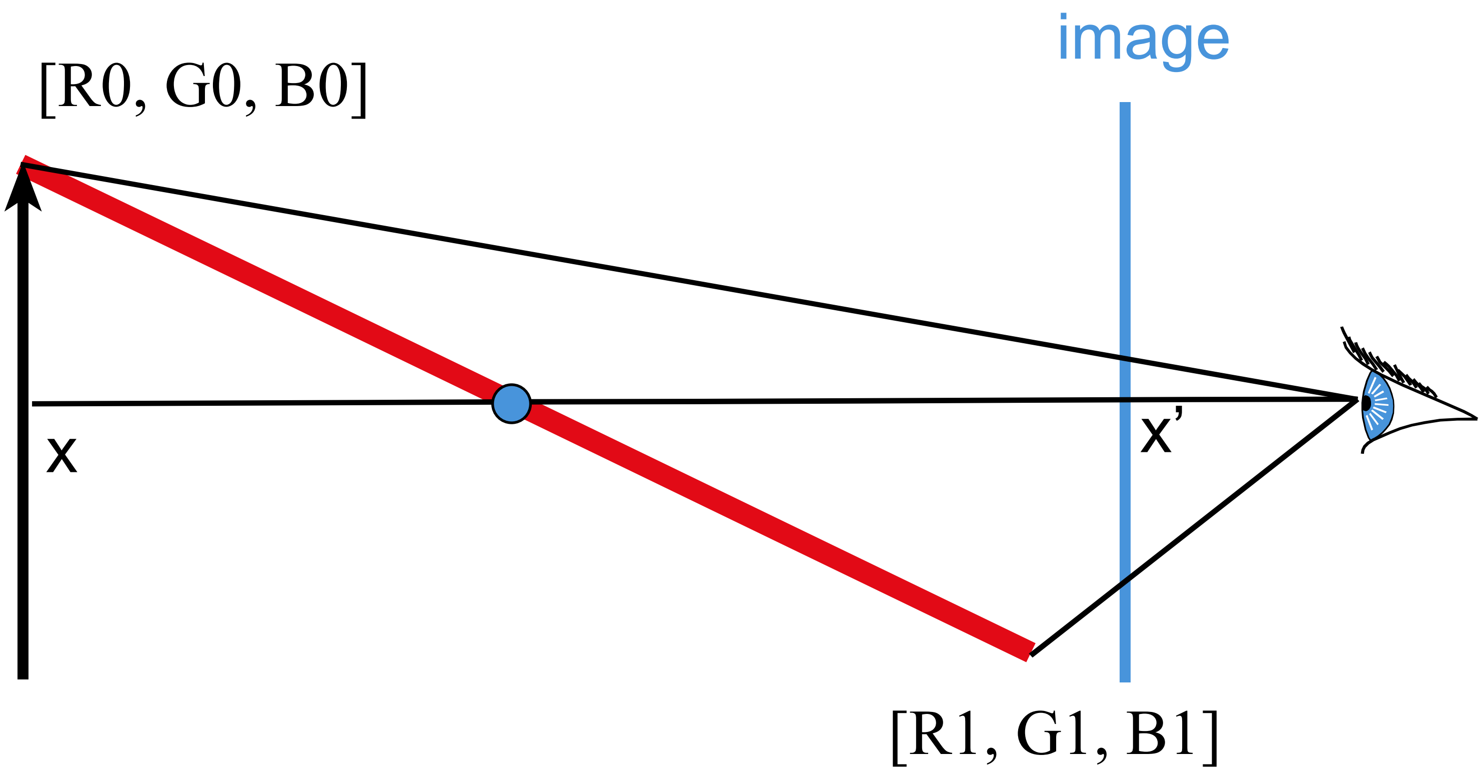


Integer Z-buffer pseudo code

```
For every triangle
   Compute Projection, color at vertices
   Setup line equations, depth equation
   Compute bbox, clip bbox to screen limits
   For all pixels in bbox
      Increment line equations
      Increment curent_1ovZ
      Increment currentColor
      If all line equations>0  //pixel [x,y] in triangle
         If current_1ovZ>1ovzBuffer[x,y]//pixel is visible
            Framebuffer[x,y]=currentColor
            1ovzBuffer[x,y]=current1ovZ
```

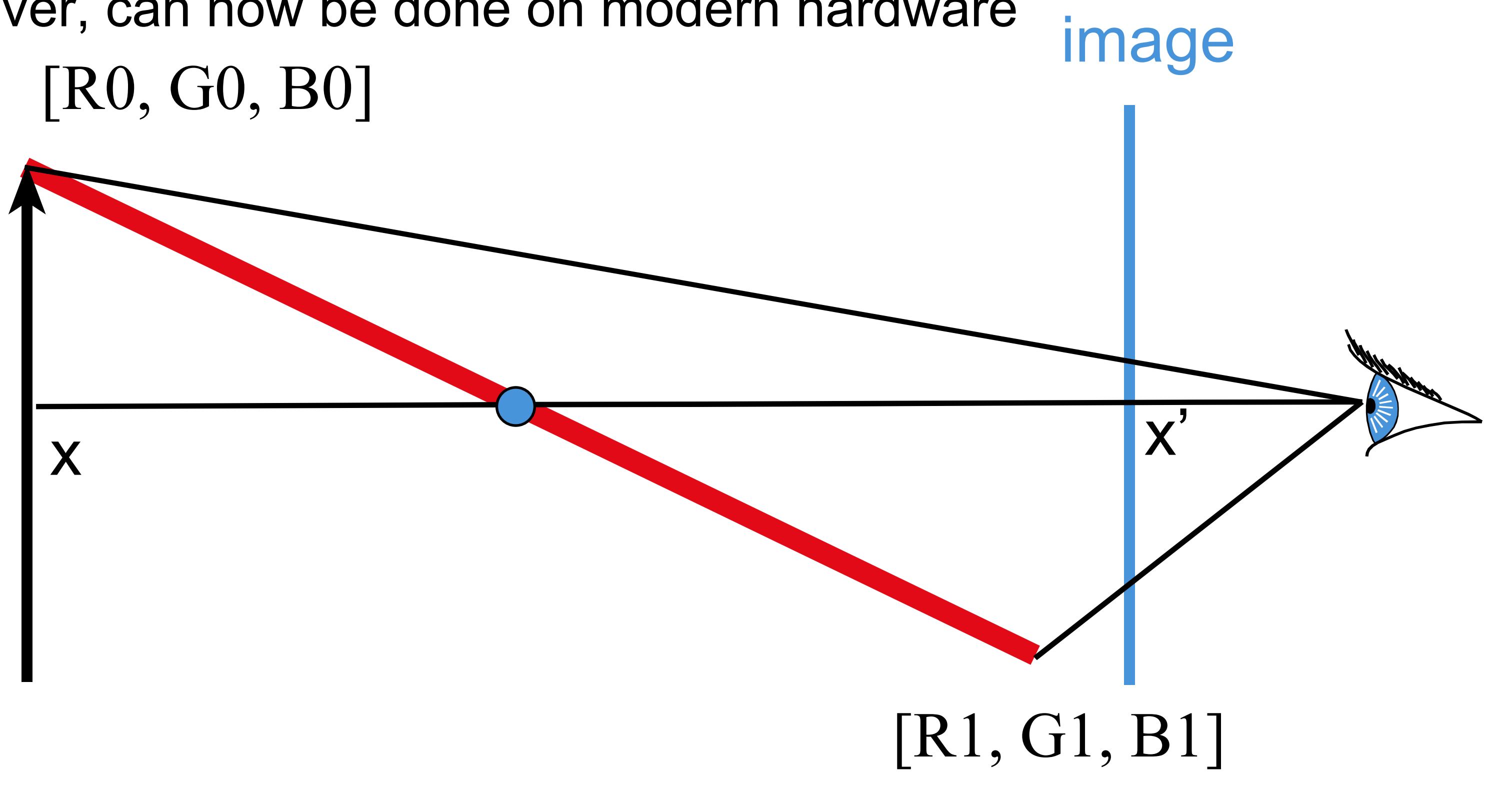
Gouraud interpolation

- Gouraud: interpolate color linearly in screen space
- Is it correct?



Gouraud interpolation

- Gouraud: interpolate color linearly in screen space
- Not correct. We should use hyperbolic interpolation
- But quite costly (division)
- However, can now be done on modern hardware



The Graphics Pipeline

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

Input:

Geometric model:
Description of all object, surface, and light source geometry and transformations

Lighting model:
Computational description of object and light properties, interaction (reflection)

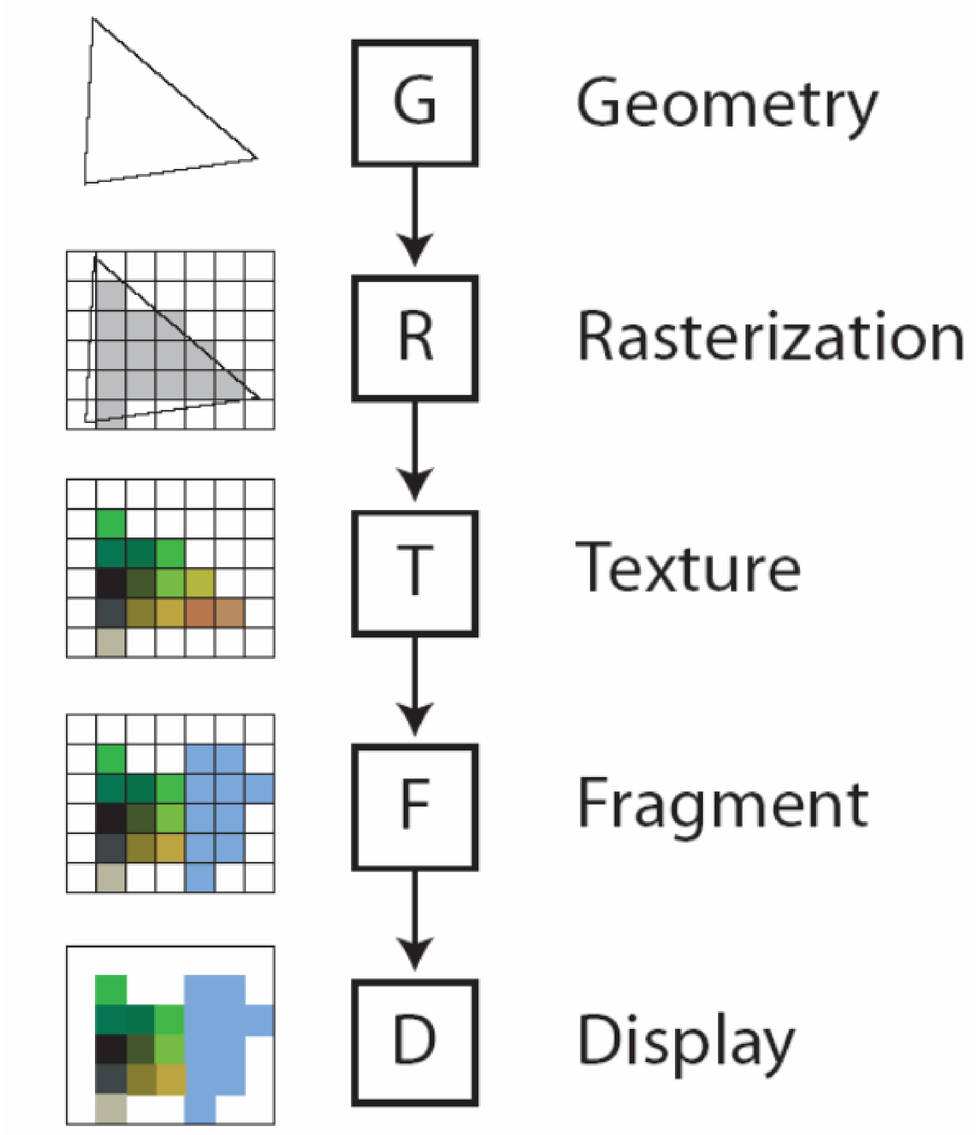
Synthetic Viewpoint (or *Camera*):
Eye position and viewing frustum

Raster Viewport:
Pixel grid onto which image plane is mapped

Output:

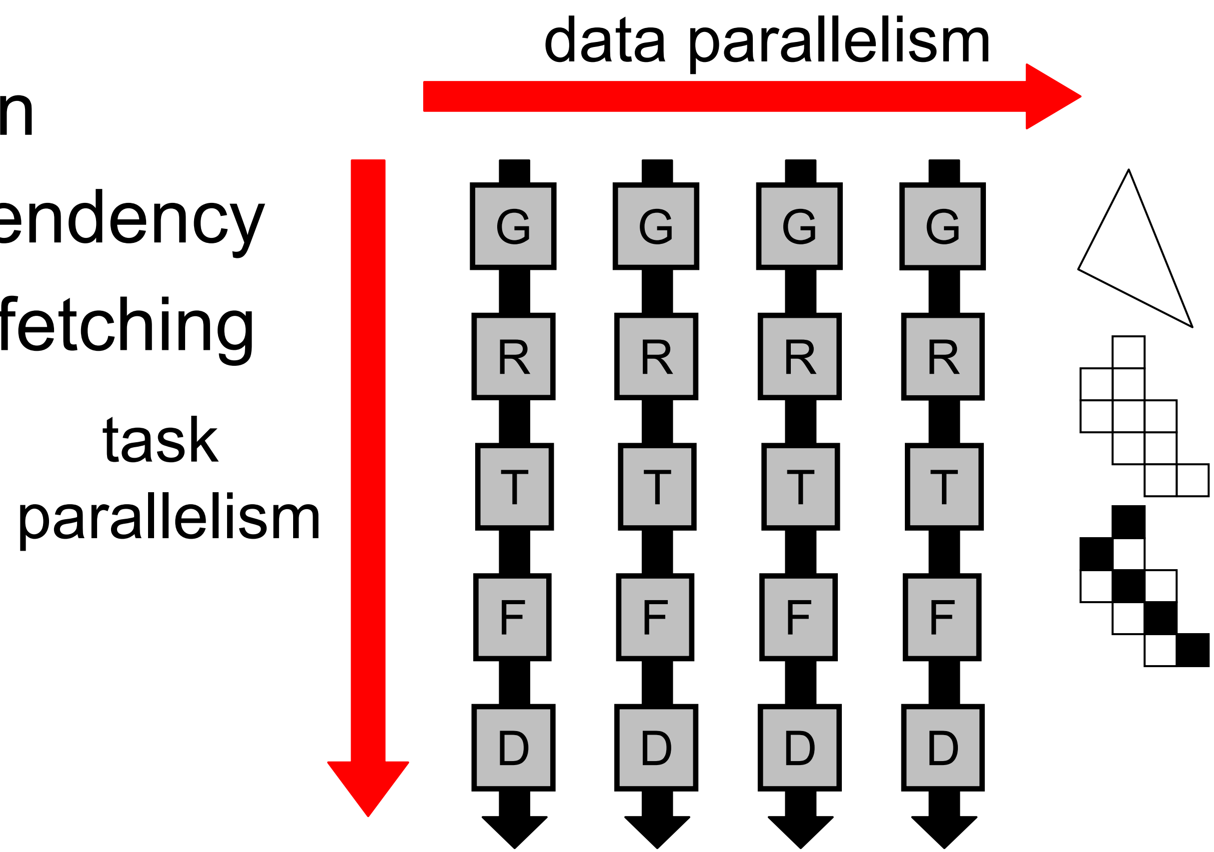
Colors/Intensities suitable for framebuffer display
(For example, 24-bit RGB value at each pixel)

Modern Graphics Hardware

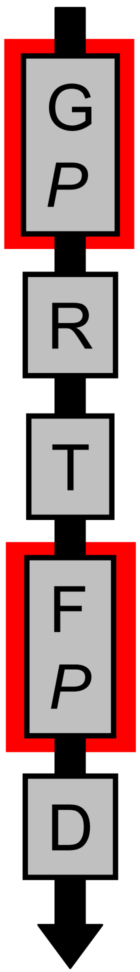


Graphics Hardware

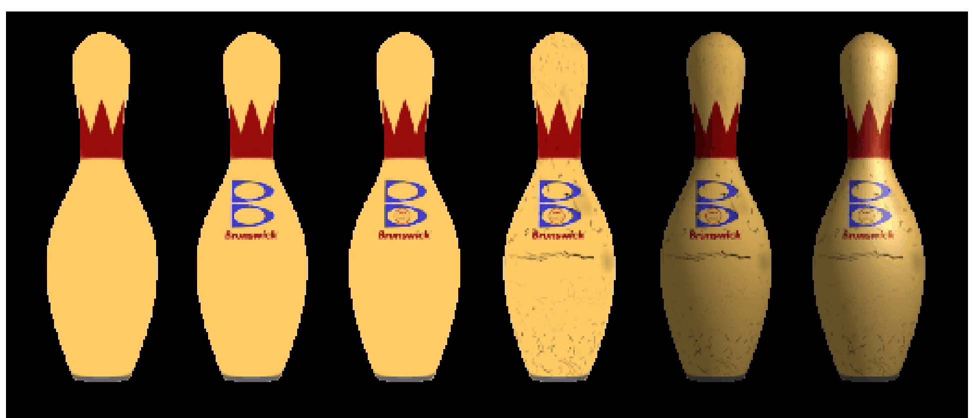
- High performance through
  - Parallelism
  - Specialization
  - No data dependency
  - Efficient pre-fetching



Programmable Graphics Hardware



- Geometry and pixel (fragment) stage become programmable
  - Elaborate appearance
  - More and more general-purpose computation (GPU hacking)



Cg

- Demo

Modern Graphics Hardware

- About 4-6 geometry units
- About 16 fragment units
- Deep pipeline (~800 stages)
- Tiling (about 4x4)
  - Early z-rejection if entire tile is occluded
- Pixels rasterized by quads (2x2 pixels)
  - Allows for derivatives
- Very efficient texture pre-fetching
  - And smart memory layout

Current GPUs

- Programmable geometry and fragment stages
- 600 million vertices/second, 6 billion texels/second
- In the range of tera operations/second
- Floating point operations only
- Very little cache

An Interactive Introduction to OpenGL Programming

Basé sur les diapos de

Dave Shreiner

Ed Angel

Vicki Shreiner

OpenGL course a SIGGRAPH 2000

What You'll See Today

- General OpenGL Introduction
- Rendering Primitives
- Rendering Modes
- Lighting
- Texture Mapping
- Additional Rendering Attributes
- Imaging

Goals

- **Demonstrate enough OpenGL to write an interactive graphics program with**
  - custom modeled 3D objects or imagery
  - lighting
  - texture mapping
- **Introduce advanced topics for future investigation**

OpenGL and GLUT Overview

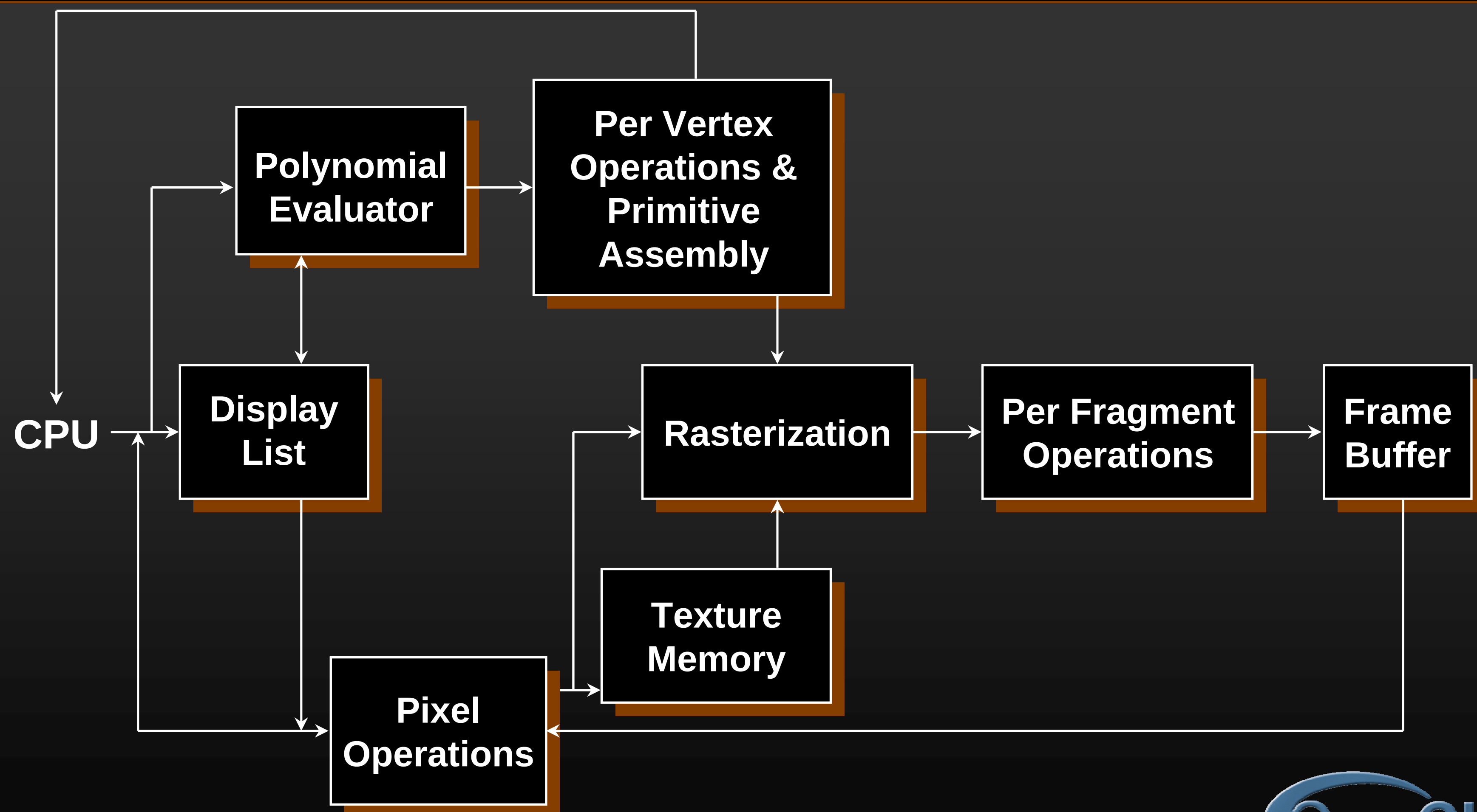
OpenGL and GLUT Overview

- What is OpenGL & what can it do for me?
- OpenGL in windowing systems
- Why GLUT
- A GLUT program template

What Is OpenGL?

- **Graphics rendering API**
  - high-quality color images composed of geometric and image primitives
  - window system independent
  - operating system independent

OpenGL Architecture



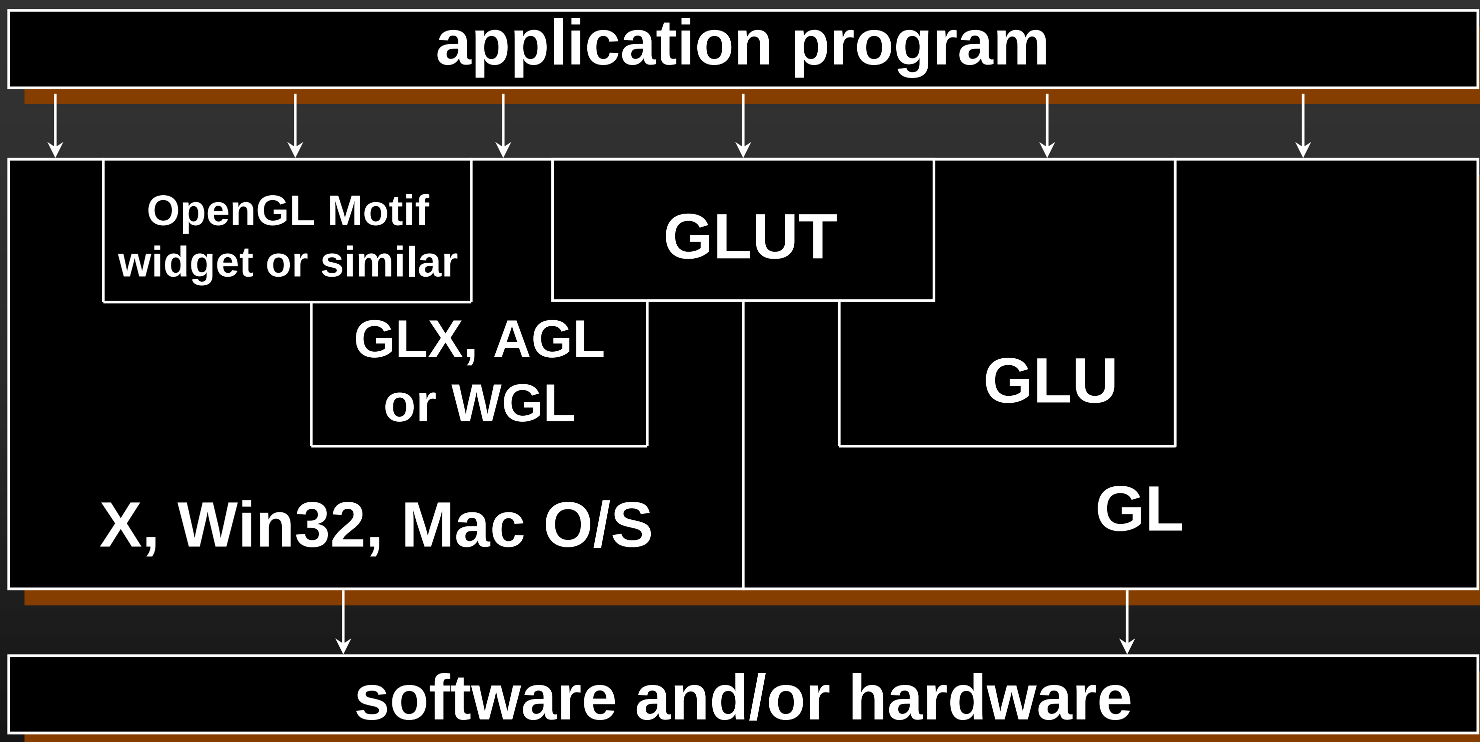
OpenGL as a Renderer

- **Geometric primitives**
  - points, lines and polygons
- **Image Primitives**
  - images and bitmaps
  - separate pipeline for images and geometry
    - linked through texture mapping
- **Rendering depends on state**
  - colors, materials, light sources, etc.

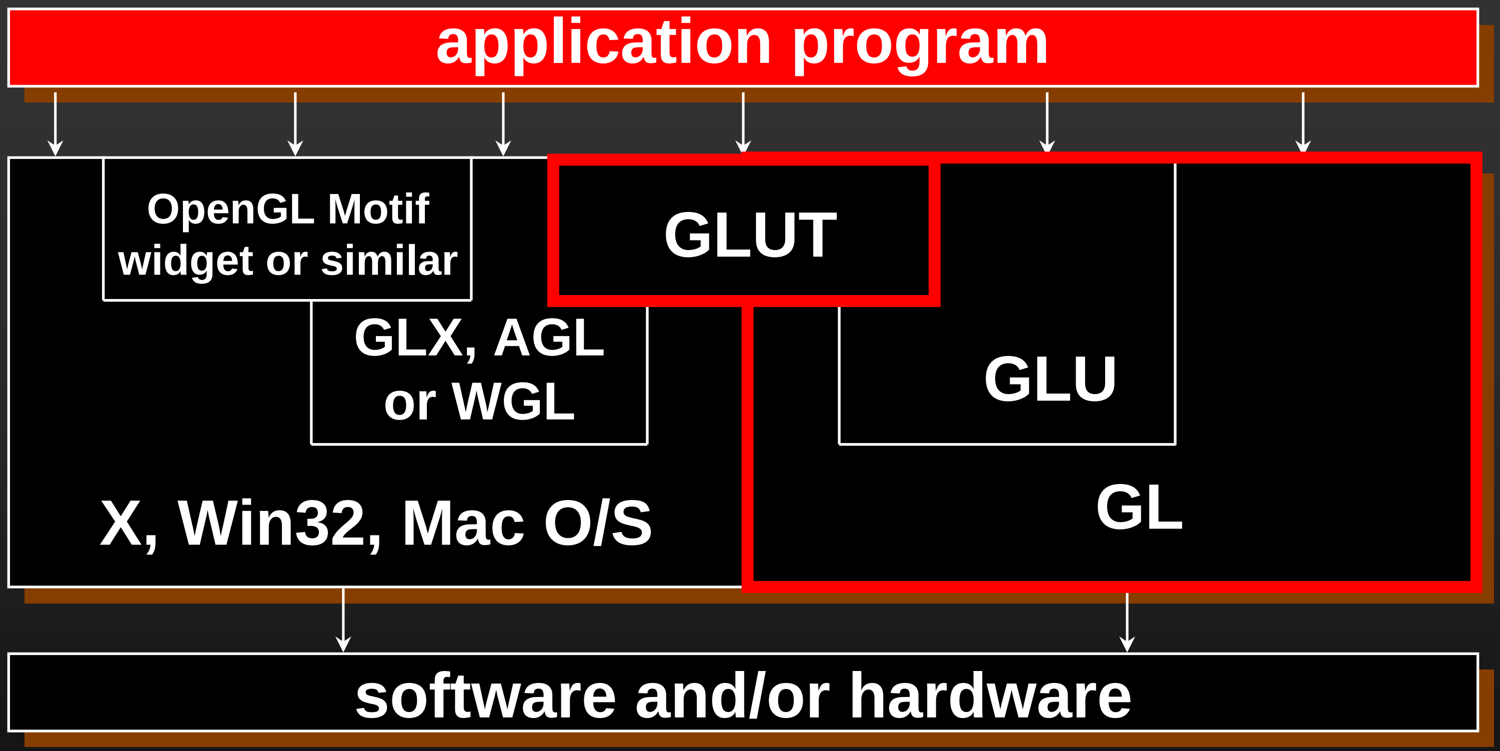
Related APIs

- **AGL, GLX, WGL**
  - glue between OpenGL and windowing systems
- **GLU (OpenGL Utility Library)**
  - part of OpenGL
  - NURBS, tessellators, quadric shapes, etc.
- **GLUT (OpenGL Utility Toolkit)**
  - portable windowing API
  - not officially part of OpenGL

OpenGL and Related APIs



OpenGL and Related APIs



Preliminaries

- **Headers Files**
  - #include <GL/gl.h>
  - #include <GL/glu.h>
  - #include <GL/glut.h>
- **Libraries**
- **Enumerated Types**
  - OpenGL defines numerous types for compatibility
    - GLfloat, GLint, GLenum, etc.

GLUT Basics

- **Application Structure**
  - Configure and open window
  - Initialize OpenGL state
  - Register input callback functions
    - render
    - resize
    - input: keyboard, mouse, etc.
  - Enter event processing loop

Sample Program

```
void main( int argc, char** argv )
{
  int mode = GLUT_RGB|GLUT_DOUBLE;
  glutInitDisplayMode( mode );
  glutCreateWindow( argv[0] );
  init();
  glutDisplayFunc( display );
  glutReshapeFunc( resize );
  glutKeyboardFunc( key );
  glutIdleFunc( idle );
  glutMainLoop();
}
```

OpenGL Initialization

- Set up whatever state you're going to use

```
void init( void )
{
  glClearColor( 0.0, 0.0, 0.0, 1.0 );
  glClearDepth( 1.0 );

  glEnable( GL_LIGHT0 );
  glEnable( GL_LIGHTING );
  glEnable( GL_DEPTH_TEST );
}
```

GLUT Callback Functions

- **Routine to call when something happens**
  - window resize or redraw
  - user input
  - animation
- **"Register" callbacks with GLUT**

```
glutDisplayFunc( display );
glutIdleFunc( idle );
glutKeyboardFunc( keyboard );
```

Rendering Callback

- Do all of your drawing here

```
glutDisplayFunc( display );
```

```
void display( void )
{
  glClear( GL_COLOR_BUFFER_BIT );
  glBegin( GL_TRIANGLE_STRIP );
    glVertex3fv( v[0] );
    glVertex3fv( v[1] );
    glVertex3fv( v[2] );
    glVertex3fv( v[3] );
  glEnd();
  glutSwapBuffers();
}
```

Idle Callbacks

- Use for animation and continuous update

```
glutIdleFunc( idle );
```

```
void idle( void )
{
  t += dt;
  glutPostRedisplay();
}
```

User Input Callbacks

- Process user input

```
glutKeyboardFunc( keyboard );
```

```
void keyboard( char key, int x, int y )
{
  switch( key ) {
    case 'q' : case 'Q' :
      exit( EXIT_SUCCESS );
      break;

    case 'r' : case 'R' :
      rotate = GL_TRUE;
      break;
  }
}
```

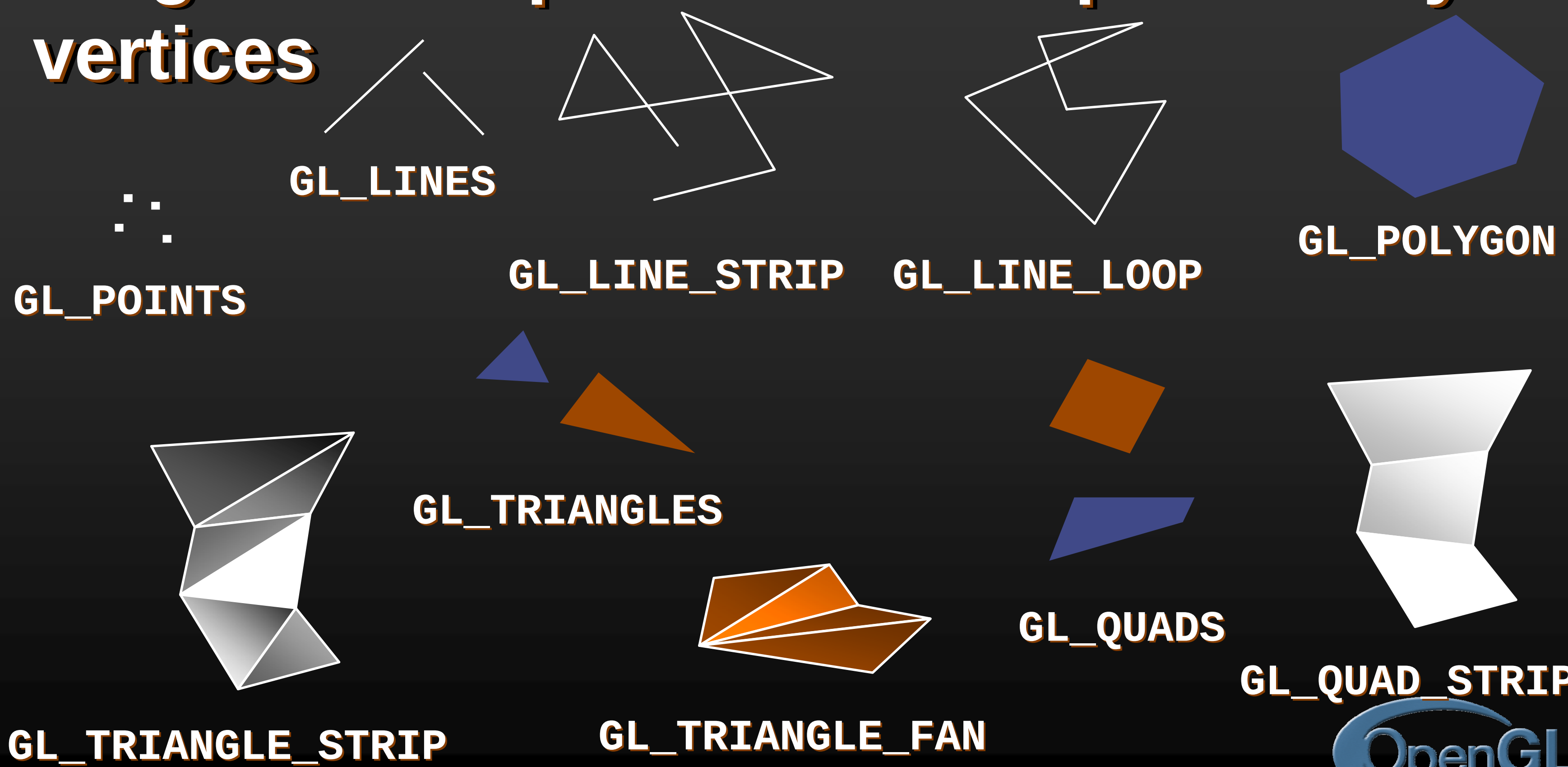
Elementary Rendering

Elementary Rendering

- Geometric Primitives
- Managing OpenGL State
- OpenGL Buffers

OpenGL Geometric Primitives

- All geometric primitives are specified by vertices



Simple Example

```
void drawRhombus( GLfloat color[] )
{
    glBegin( GL_QUADS );
    glColor3fv( color );
    glVertex2f( 0.0, 0.0 );
    glVertex2f( 1.0, 0.0 );
    glVertex2f( 1.5, 1.118 );
    glVertex2f( 0.5, 1.118 );
    glEnd();
}
```

OpenGL Command Formats

```
glVertex3fv( v )
```

Number of components

```
2 - (x,y)
3 - (x,y,z)
4 - (x,y,z,w)
```

Data Type

```
b  - byte
ub - unsigned byte
s  - short
us - unsigned short
i  - int
ui - unsigned int
f  - float
d  - double
```

Vector

omit "v" for scalar form

```
glVertex2f( x, y )
```

Specifying Geometric Primitives

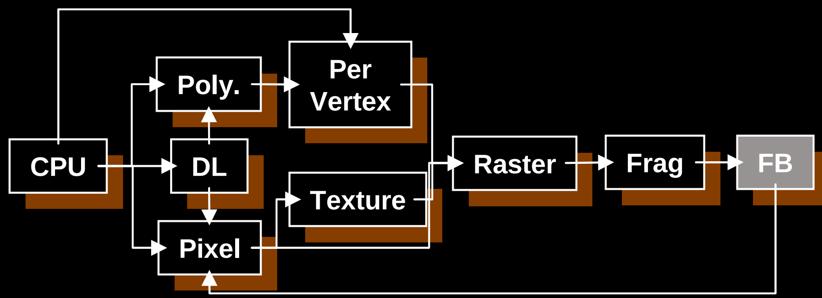
- **Primitives are specified using**

  ```
  glBegin( primType );
  glEnd();
  ```

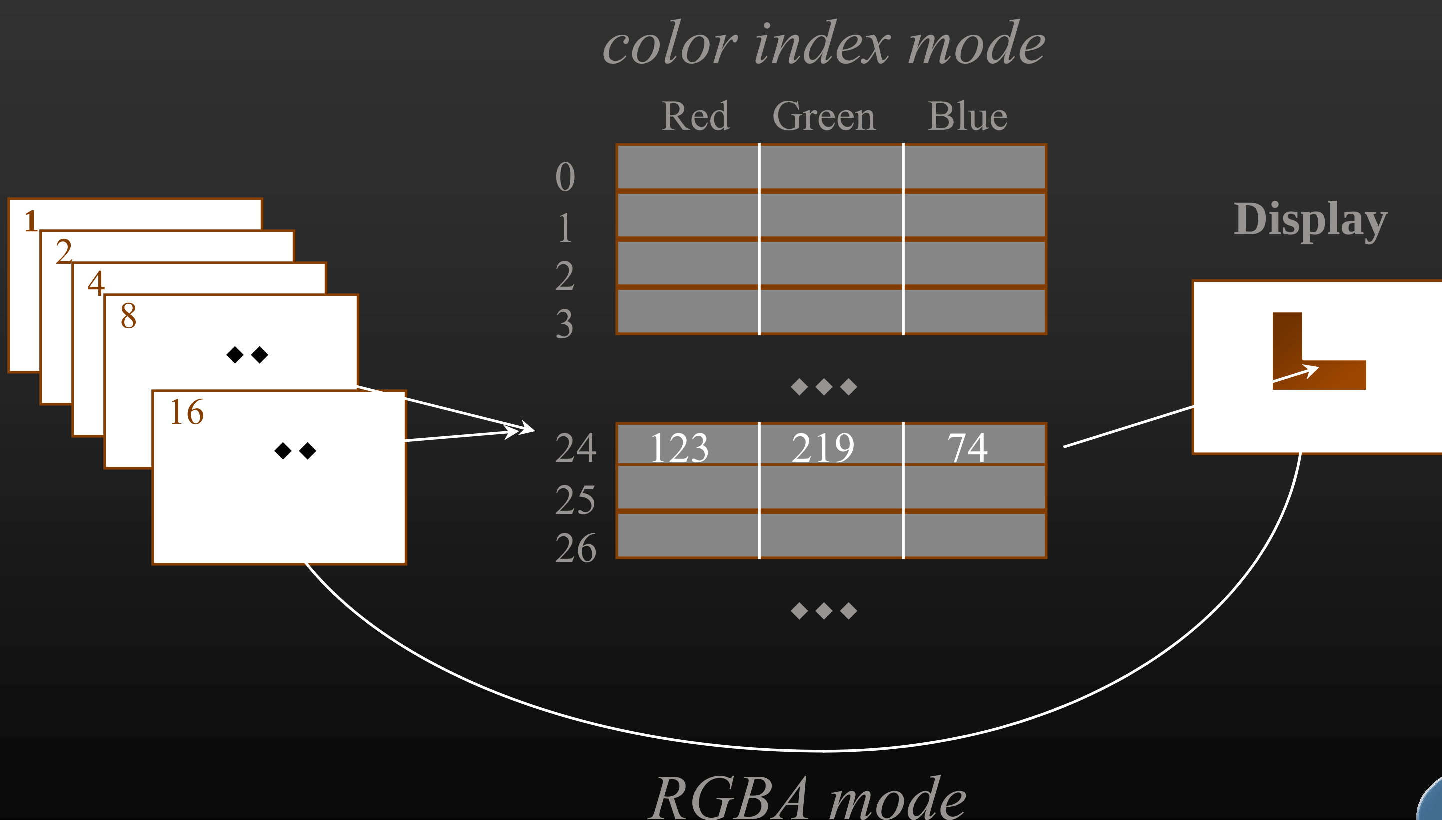
  - *primType* determines how vertices are combined

```
GLfloat red, greed, blue;
Glfloat coords[3];
glBegin( primType );
for ( i = 0; i < nVerts; ++i ) {
  glColor3f( red, green, blue );
  glVertex3fv( coords );
}
glEnd();
```

OpenGL Color Models



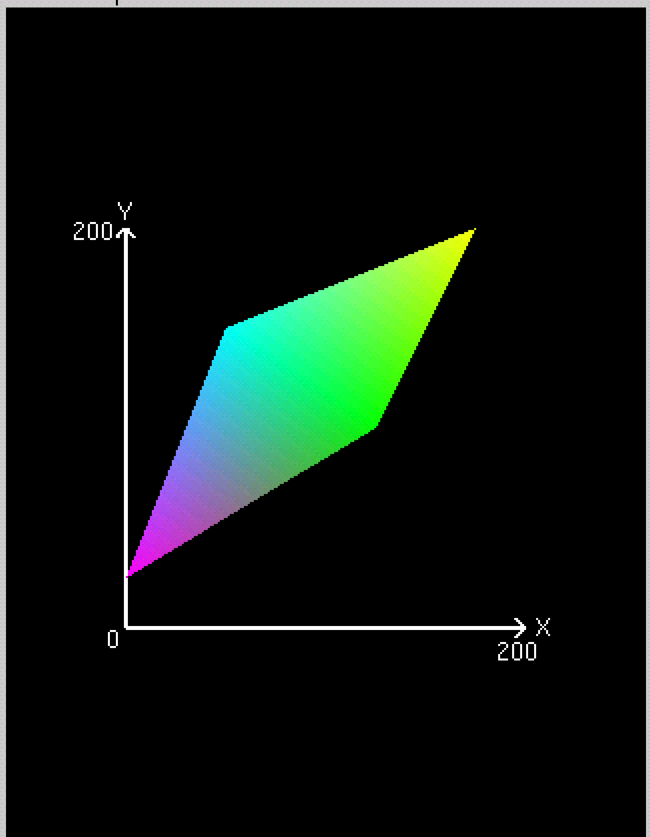
- **RGBA or Color Index**



Shapes Tutorial

Shapes

Screen-space view

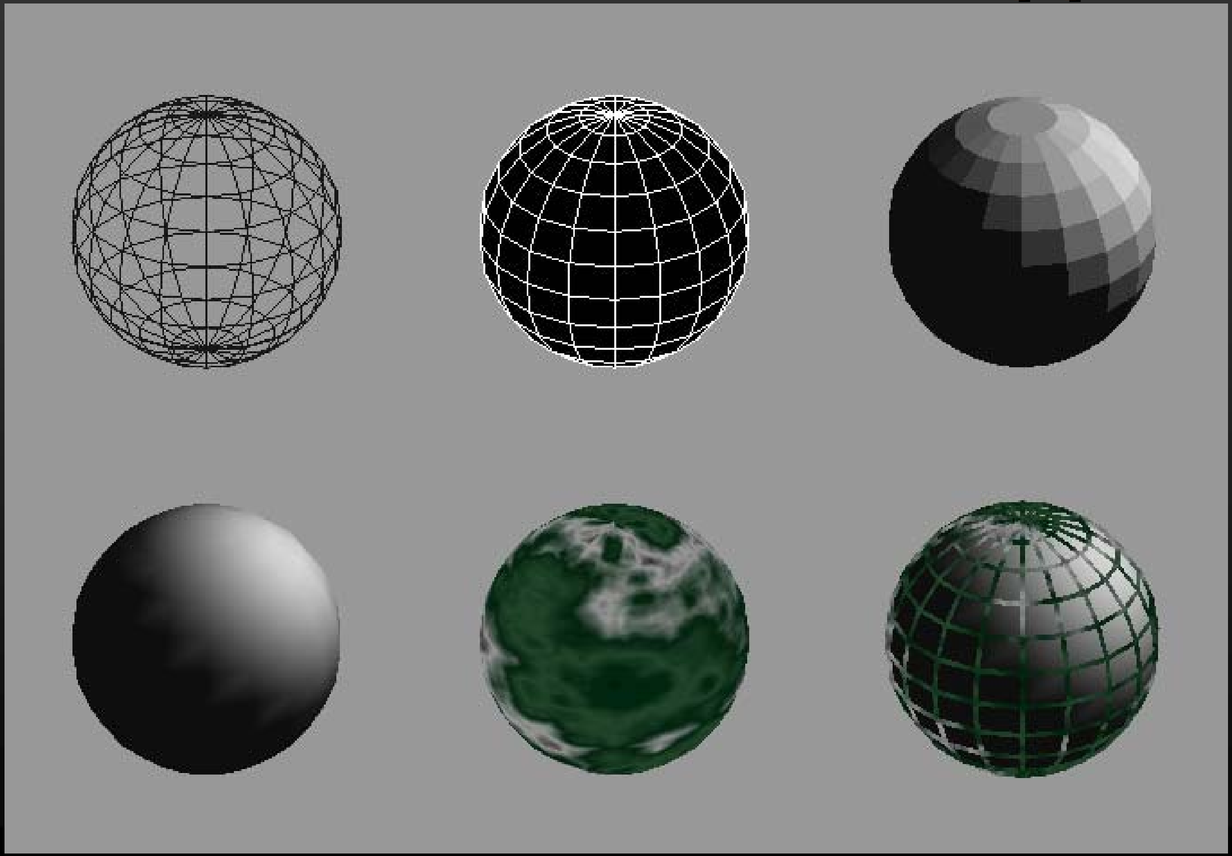


Command manipulation window

```
glBegin (GL_TRIANGLE_STRIP);
glColor3f  (1.00  , 0.00  , 1.00  );
glVertex2f (0.0   , 25.0  );
glColor3f  (0.00  , 1.00  , 1.00  );
glVertex2f (50.0  , 150.0 );
glColor3f  (0.00  , 1.00  , 0.00  );
glVertex2f (125.0 , 100.0 );
glColor3f  (1.00  , 1.00  , 0.00  );
glVertex2f (175.0 , 200.0 );
glEnd();
```

Controlling Rendering Appearance

- From Wireframe to Texture Mapped



OpenGL's State Machine

- **All rendering attributes are encapsulated in the OpenGL State**
  - rendering styles
  - shading
  - lighting
  - texture mapping

Manipulating OpenGL State

- **Appearance is controlled by current state**

```
for each ( primitive to render ) {
    update OpenGL state
    render primitive
}
```

- **Manipulating vertex attributes is most common way to manipulate state**

  `glColor*() / glIndex*()`
  `glNormal*()`
  `glTexCoord*()`

Controlling current state

- Setting State

```
glPointSize( size );
glLineStipple( repeat, pattern );
glShadeModel( GL_SMOOTH );
```

- Enabling Features

```
glEnable( GL_LIGHTING );
glDisable( GL_TEXTURE_2D );
```

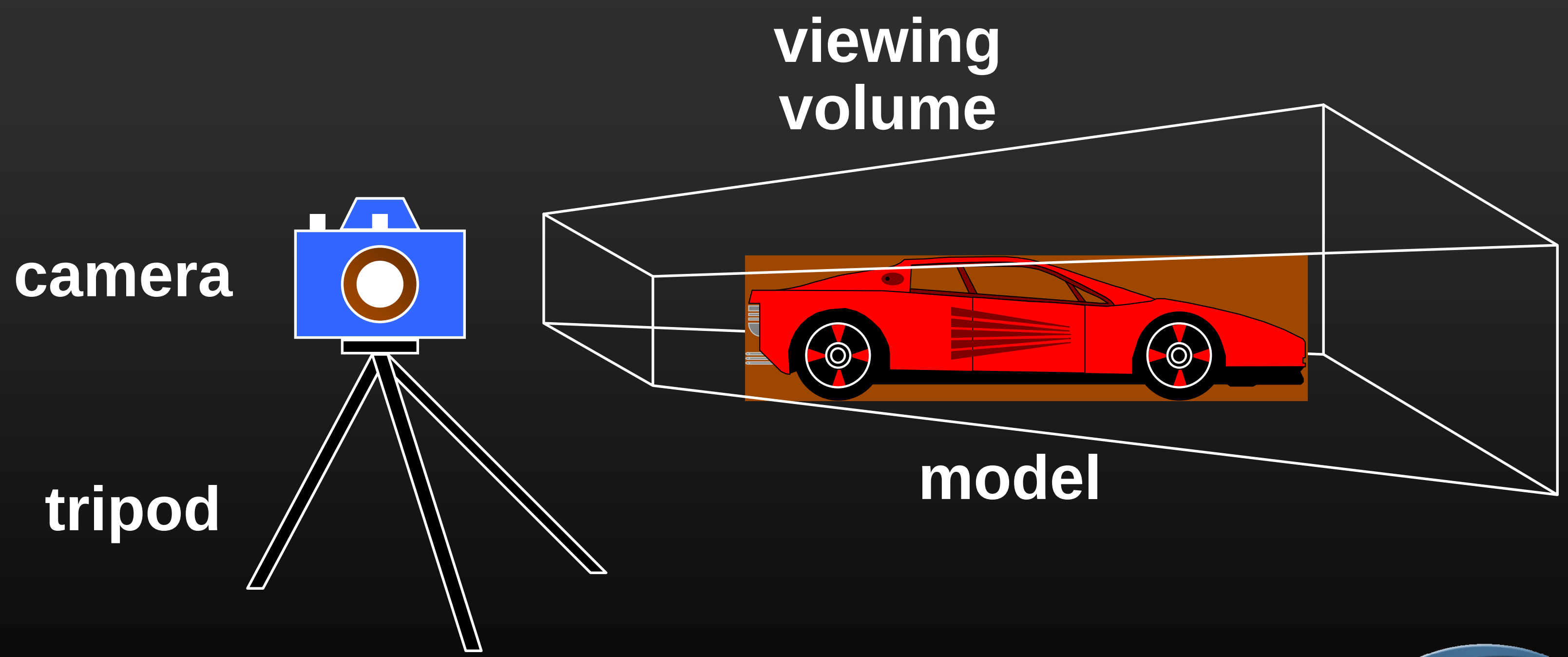
Transformations

Transformations in OpenGL

- **Modeling**
- **Viewing**
  - orient camera
  - projection
- **Animation**
- **Map to screen**

Camera Analogy

- 3D is just like taking a photograph (lots of photographs!)



Camera Analogy and Transformations

- **Projection transformations**
  - adjust the lens of the camera
- **Viewing transformations**
  - tripod–define position and orientation of the viewing volume in the world
- **Modeling transformations**
  - moving the model
- **Viewport transformations**
  - enlarge or reduce the physical photograph

Coordinate Systems and Transformations

- **Steps in Forming an Image**
  - specify geometry (world coordinates)
  - specify camera (camera coordinates)
  - project (window coordinates)
  - map to viewport (screen coordinates)
- **Each step uses transformations**
- **Every transformation is equivalent to a change in coordinate systems (frames)**

Affine Transformations

- **Want transformations which preserve geometry**
  - lines, polygons, quadrics
- **Affine = line preserving**
  - Rotation, translation, scaling
  - Projection
  - Concatenation (composition)

Homogeneous Coordinates

- each vertex is a column vector

$$\vec{v} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

- w is usually 1.0
- all operations are matrix multiplications
- directions (directed line segments) can be represented with $w = 0.0$

3D Transformations

- **A vertex is transformed by 4 x 4 matrices**
  - all affine operations are matrix multiplications
  - all matrices are stored column-major in OpenGL
  - matrices are always post-multiplied
  - product of matrix and vector is $\mathbf{M}\vec{v}$

$$\mathbf{M} = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

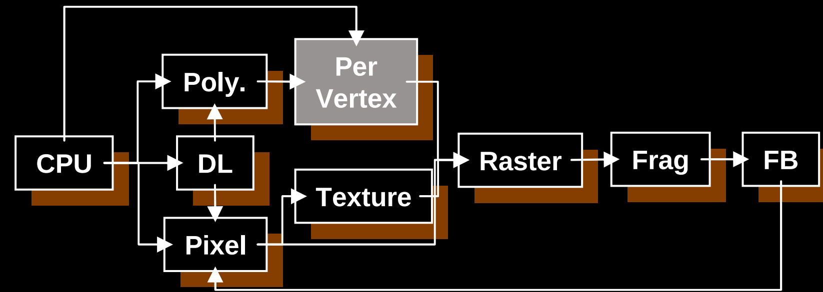
Specifying Transformations

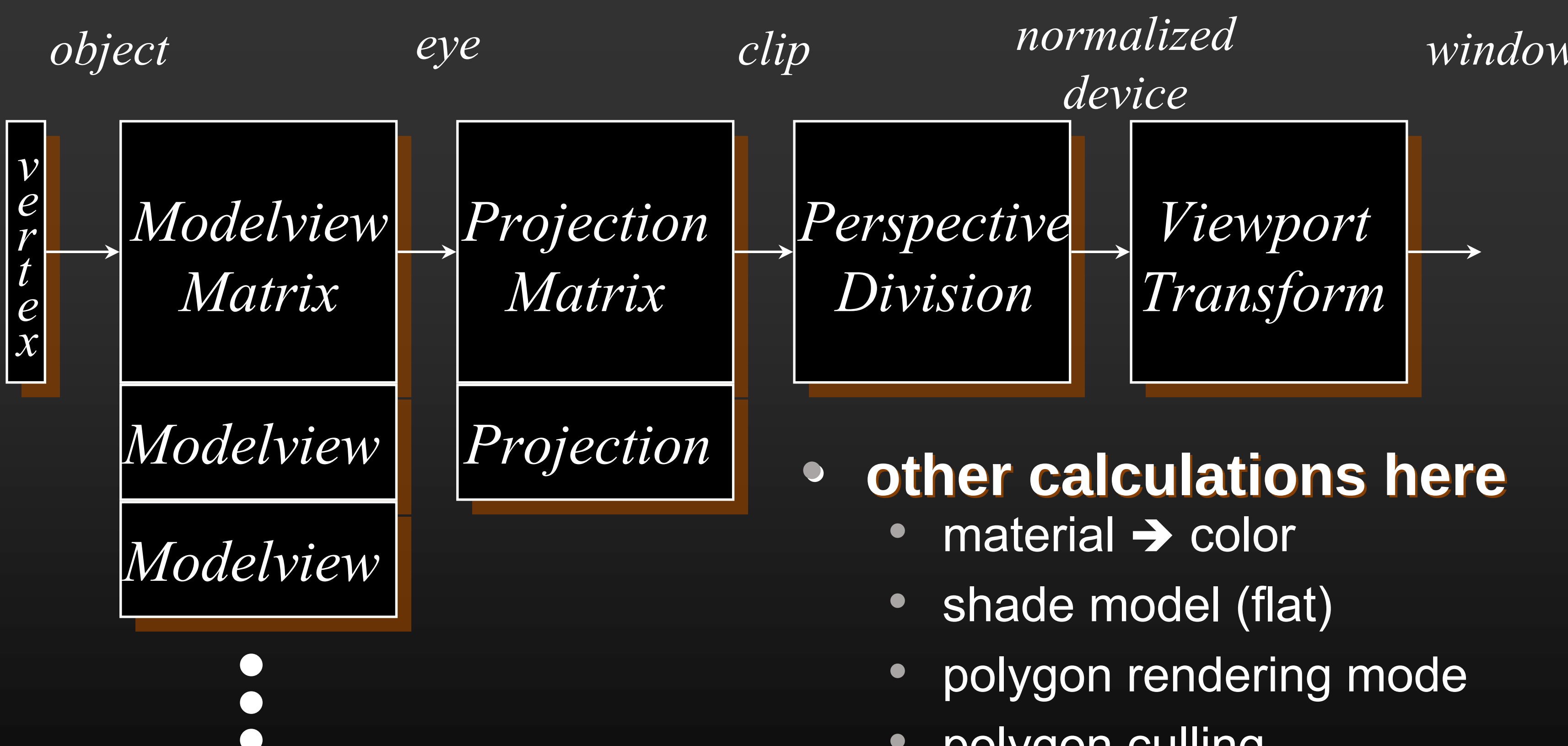
- **Programmer has two styles of specifying transformations**
  - specify matrices (`glLoadMatrix, glMultMatrix`)
  - specify operation (`glRotate, glOrtho`)
- **Programmer does not have to remember the exact matrices**
  - check appendix of Red Book (Programming Guide)

Programming Transformations

- **Prior to rendering, view, locate, and orient:**
  - eye/camera position
  - 3D geometry
- **Manage the matrices**
  - including matrix stack
- **Combine (composite) transformations**

Transformation Pipeline





- **other calculations here**
  - material ➔ color
  - shade model (flat)
  - polygon rendering mode
  - polygon culling
  - clipping

Matrix Operations

- Specify Current Matrix Stack

  ```
  glMatrixMode( GL_MODELVIEW or GL_PROJECTION )
  ```

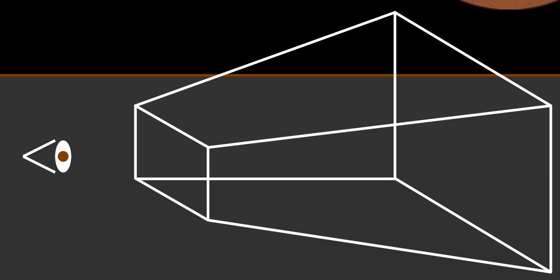
- Other Matrix or Stack Operations

  ```
  glLoadIdentity()        glPushMatrix()
              glPopMatrix()
  ```

- Viewport
  - usually same as window size
  - viewport aspect ratio should be same as projection transformation or resulting image may be distorted

```
glViewport( x, y, width, height )
```

Projection Transformation



- **Shape of viewing frustum**
- **Perspective projection**

  `gluPerspective( fovy, aspect, zNear, zFar )`

  `glFrustum( left, right, bottom, top, zNear, zFar )`

- **Orthographic parallel projection**

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

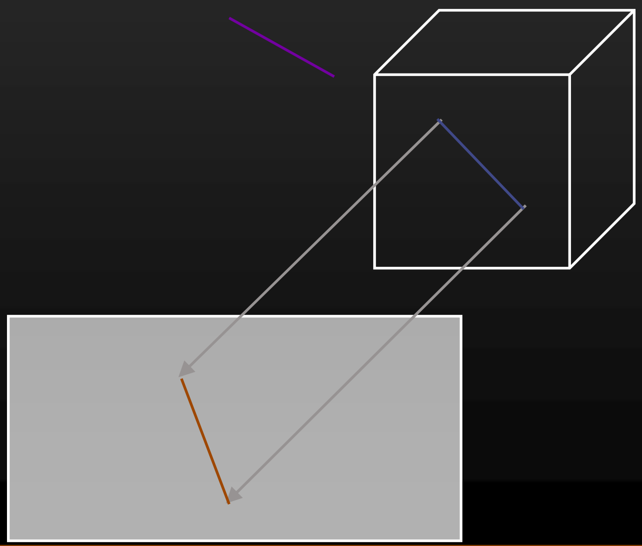
  `gluOrtho2D( left, right, bottom, top )`

  - calls `glOrtho` with z values near zero

Applying Projection Transformations

- Typical use (orthographic projection)

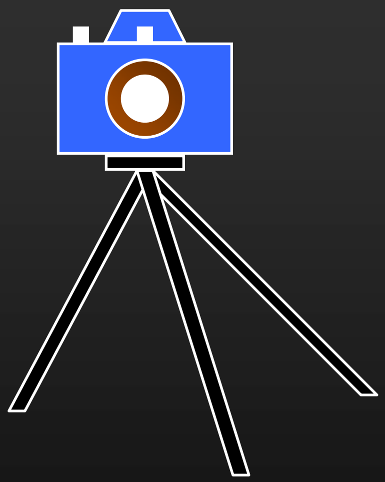
```
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
glOrtho( left, right, bottom, top, zNear, zFar );
```



Viewing Transformations

- **Position the camera/eye in the scene**
  - place the tripod down; aim camera
- **To “fly through” a scene**
  - change viewing transformation and redraw scene

tripod

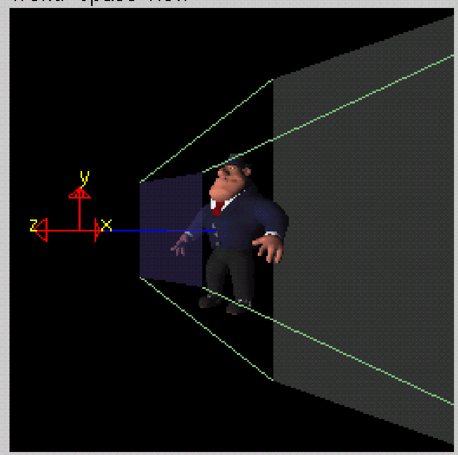


- $\texttt{gluLookAt(}\ eye_x,\ eye_y,\ eye_z,$
  $aim_x,\ aim_y,\ aim_z,$
  $up_x,\ up_y,\ up_z\ \texttt{)}$
  - up vector determines unique orientation
  - careful of degenerate positions

Projection Tutorial

Projection

World-space view



Screen-space view

Command manipulation window

	fovy	aspect	zNear	zFar	
gluPerspective(	60.0	, 1.00	, 1.0	, 10.0	);
gluLookAt(	0.00	, 0.00	, 2.00	,	<- eye
	0.00	, 0.00	, 0.00	,	<- center
	0.00	, 1.00	, 0.00	);	<- up

Click on the arguments and move the mouse to modify values.

OpenGL™

Modeling Transformations

- **Move object**

  `glTranslate{fd}( x, y, z )`

- **Rotate object around arbitrary axis** $\begin{pmatrix} x & y & z \end{pmatrix}$

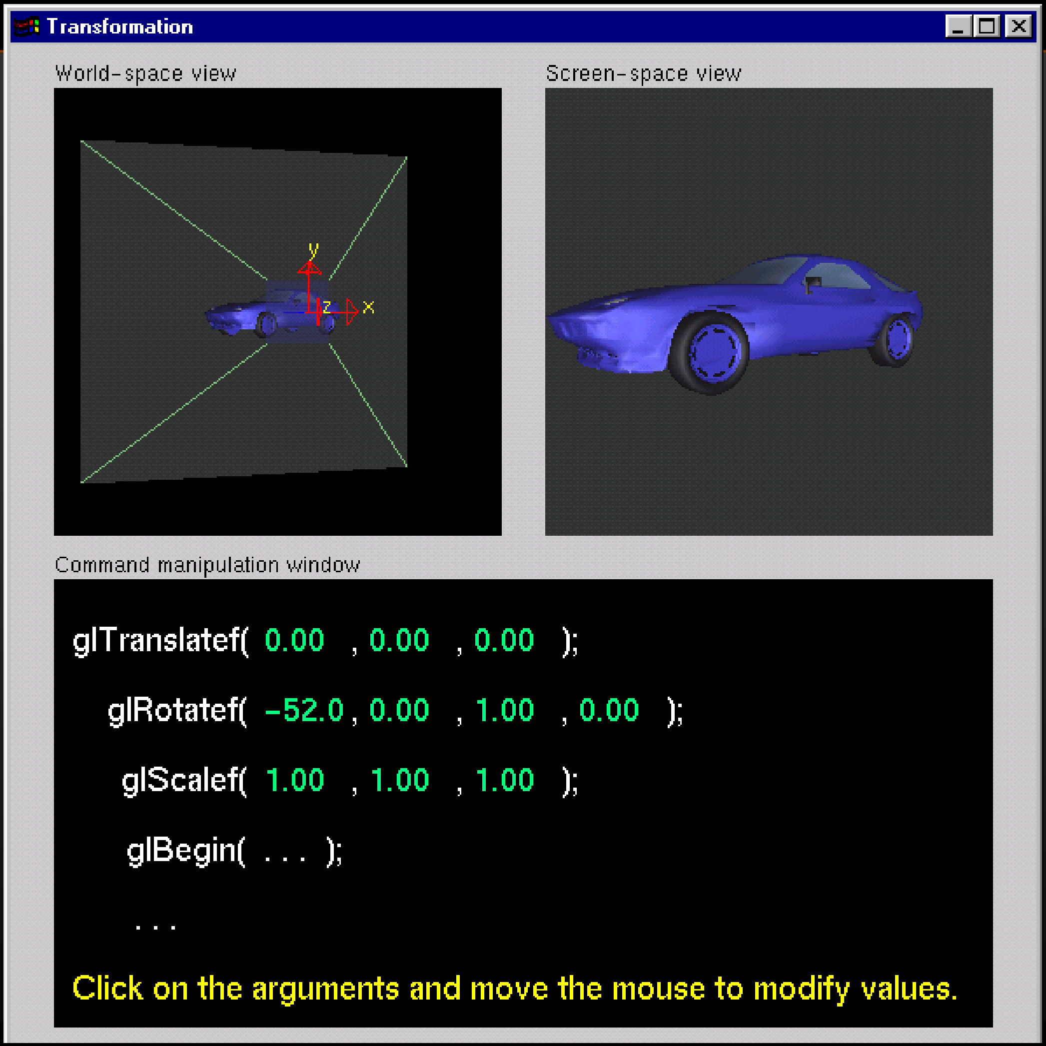
  `glRotate{fd}( angle, x, y, z )`

  - angle is in degrees

- **Dilate (stretch or shrink) or mirror object**

  `glScale{fd}( x, y, z )`

Transformation Tutorial



Connection: Viewing and Modeling

- **Moving the camera is equivalent to moving every object in the world towards a stationary camera**
- **Viewing transformations are equivalent to several modeling transformations**

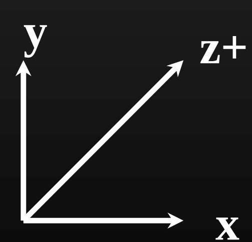
  `gluLookAt()` has its own command

  can make your own *polar view* or *pilot view*

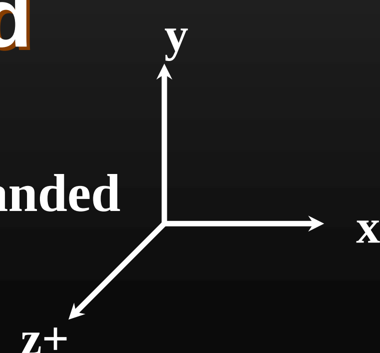
Projection is left handed

- **Projection transformations (`gluPerspective`, `glOrtho`) are left handed**
  - think of ***`zNear`*** and ***`zFar`*** as distance from view point
- **Everything else is right handed, including the vertices to be rendered**

left handed



right handed



Common Transformation Usage

- **3 examples of `resize()` routine**
  - restate projection & viewing transformations
- **Usually called when window resized**
- **Registered as callback for `glutReshapeFunc()`**

`resize()`: Perspective & LookAt

```
void resize( int w, int h )
{
   glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
   glMatrixMode( GL_PROJECTION );
   glLoadIdentity();
   gluPerspective( 65.0, (GLfloat) w / h,
                   1.0, 100.0 );
   glMatrixMode( GL_MODELVIEW );
   glLoadIdentity();
   gluLookAt( 0.0, 0.0, 5.0,
              0.0, 0.0, 0.0,
              0.0, 1.0, 0.0 );
}
```

`resize()`: Perspective & Translate

- Same effect as previous LookAt

```
void resize( int w, int h )
{
   glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
   glMatrixMode( GL_PROJECTION );
   glLoadIdentity();
   gluPerspective( 65.0, (GLfloat) w/h,
                   1.0, 100.0 );
   glMatrixMode( GL_MODELVIEW );
   glLoadIdentity();
   glTranslatef( 0.0, 0.0, -5.0 );
}
```

resize(): Ortho (part 1)

```
void resize( int width, int height )
{
    GLdouble aspect = (GLdouble) width / height;
    GLdouble left = -2.5, right = 2.5;
    GLdouble bottom = -2.5, top = 2.5;
    glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
```

… continued …

resize(): Ortho (part 2)

```
    if ( aspect < 1.0 ) {
        left /= aspect;
        right /= aspect;
    } else {
        bottom *= aspect;
        top *= aspect;
    }
    glOrtho( left, right, bottom, top, near, far );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
}
```

Compositing Modeling Transformations

- **Problem 1: hierarchical objects**
  - one position depends upon a previous position
  - robot arm or hand; sub-assemblies
- **Solution 1: moving local coordinate system**
  - modeling transformations move coordinate system
  - post-multiply column-major matrices
  - OpenGL post-multiplies matrices

Compositing Modeling Transformations

- **Problem 2: objects move relative to absolute world origin**
  - my object rotates around the wrong origin
    - make it spin around its center or something else
- **Solution 2: fixed coordinate system**
  - modeling transformations move objects around fixed coordinate system
  - pre-multiply column-major matrices
  - OpenGL post-multiplies matrices
  - must <u>reverse order of operations</u> to achieve desired effect

Reversing Coordinate Projection

- Screen space back to world space

```
glGetIntegerv( GL_VIEWPORT, GLint viewport[4] )
glGetDoublev( GL_MODELVIEW_MATRIX, GLdouble mvmatrix[16] )
glGetDoublev( GL_PROJECTION_MATRIX,
                 GLdouble projmatrix[16] )
gluUnProject( GLdouble winx, winy, winz,
                 mvmatrix[16], projmatrix[16],
                 GLint viewport[4],
                 GLdouble *objx, *objy, *objz )
```

- `gluProject` goes from world to screen space