Projective Rendering

Dr. Hamish Carr & Dr. Rafael Kuffner dos Anjos

Agenda

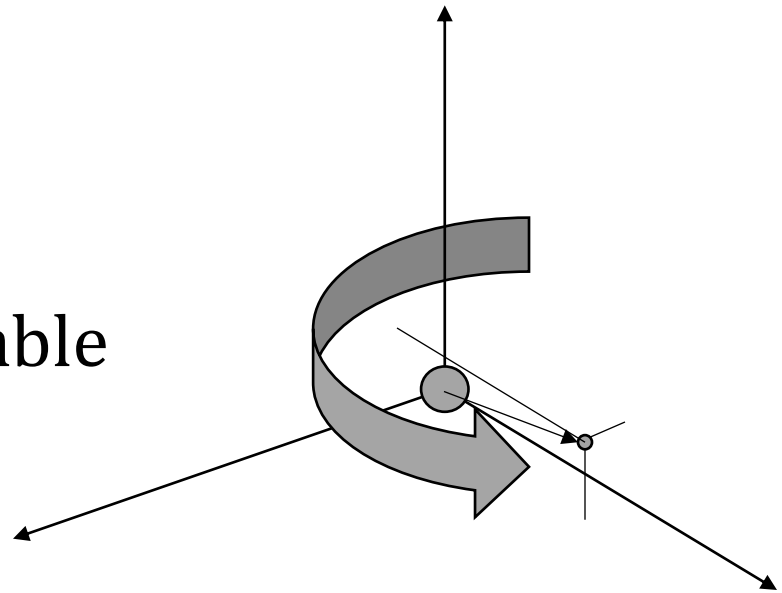
1. Vector maths for computer graphics
2. The synthetic camera.
3. Coordinate Systems
4. Projective pipeline

What tools do we need?

- Represent 3D things (points, lines, objects)
- Transformations (move, scale, rotate, etc…)
- Quick operations related to projection.
- Not obvious now: useful when get more hands on.

Coordinate Systems

- A *coordinate* system consists of:
  - an origin
  - (usually orthonormal) x,y,z vectors as axes
  - that usually obey the right-hand rule
  - which give rise to Cartesian coordinates
- Points & vectors are (mostly) interchangeable
- But their semantics differ



Vector Operations: basics recap

- Addition ($\vec{v} + \vec{u}$), subtraction ($\vec{v} - \vec{u}$), scalar multiplication (k$\vec{v}$)
- Vector length (magnitude) ($|\vec{v}|$)& normalization ($\hat{v}$)
- Dot product ($v.u$) & cross product ($v \times u$)
- Matrix multiplication ($M_1 M_2$) with matrices & vectors ($M_1 \vec{v}$)
- Matrix transpose (M^T)
- But *not* matrix inversion (we can avoid it)

Dot Product: useful for several things!

- Length of a vector: $\|\vec{v}\| = \sqrt{\vec{v} \cdot \vec{v}}$
- Angle between vectors: $\vec{v} \cdot \vec{w} = \|\vec{v}\| \, \|\vec{w}\| \cos\theta$

  $\cos\theta = \frac{\vec{v} \cdot \vec{w}}{\|\vec{v}\| \, \|\vec{w}\|}$
- Test for right angles: $\vec{n} \cdot \vec{v} = 0$
- Projection onto a line: $\Pi_{\vec{v}}(\vec{w}) = \frac{\vec{v} \cdot \vec{w}}{\vec{v} \cdot \vec{v}} \vec{v}$
- Distance to a line: $d = \frac{\vec{n} \cdot p - c}{\|\vec{n}\|}$

Normal Vectors

- A normal vector is perpendicular to v

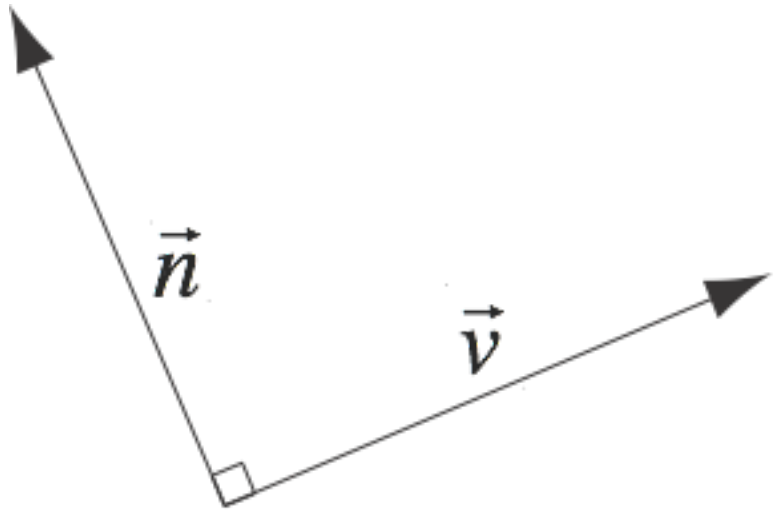
$$\frac{\vec{v}\cdot\vec{n}}{\|\vec{v}\|\|\vec{n}\|}=\cos 90°=0$$

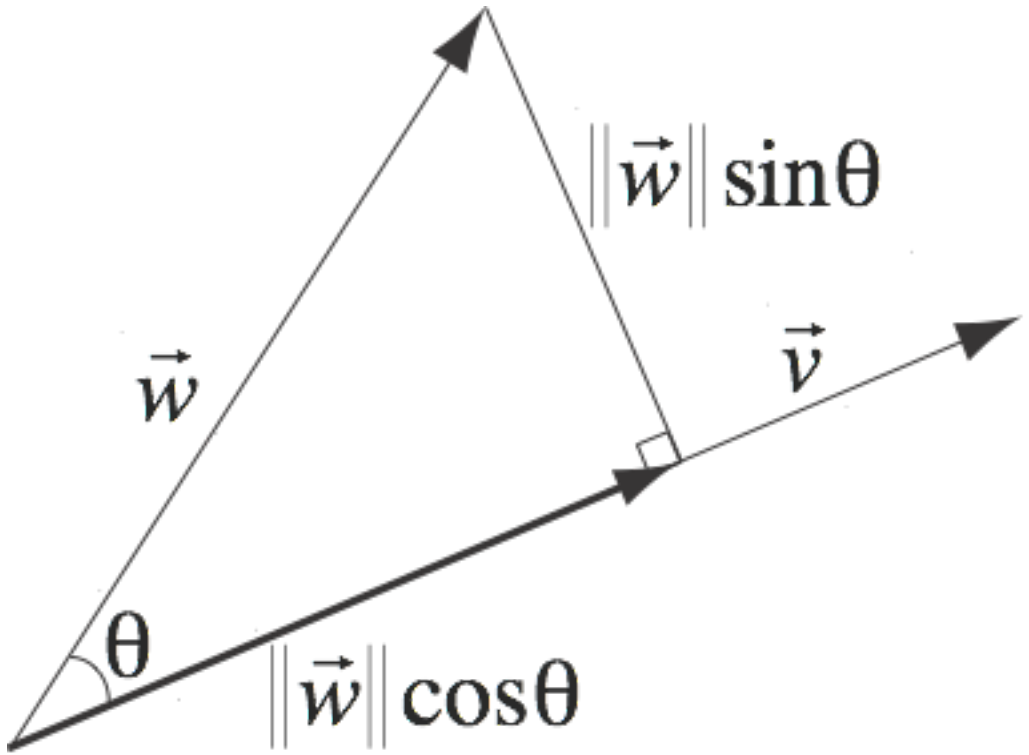
$$\vec{v}\cdot\vec{n}=0$$

$$v_x n_x + v_y n_y = 0$$

$$\text{Let } \vec{n}=\begin{bmatrix}-v_y\\ v_x\end{bmatrix}$$

$$v_x\left(-v_y\right)+v_y\left(v_x\right)=0$$





$$\prod_{\vec{v}}(\vec{w}) = (Length)(Unit\ Vector)$$

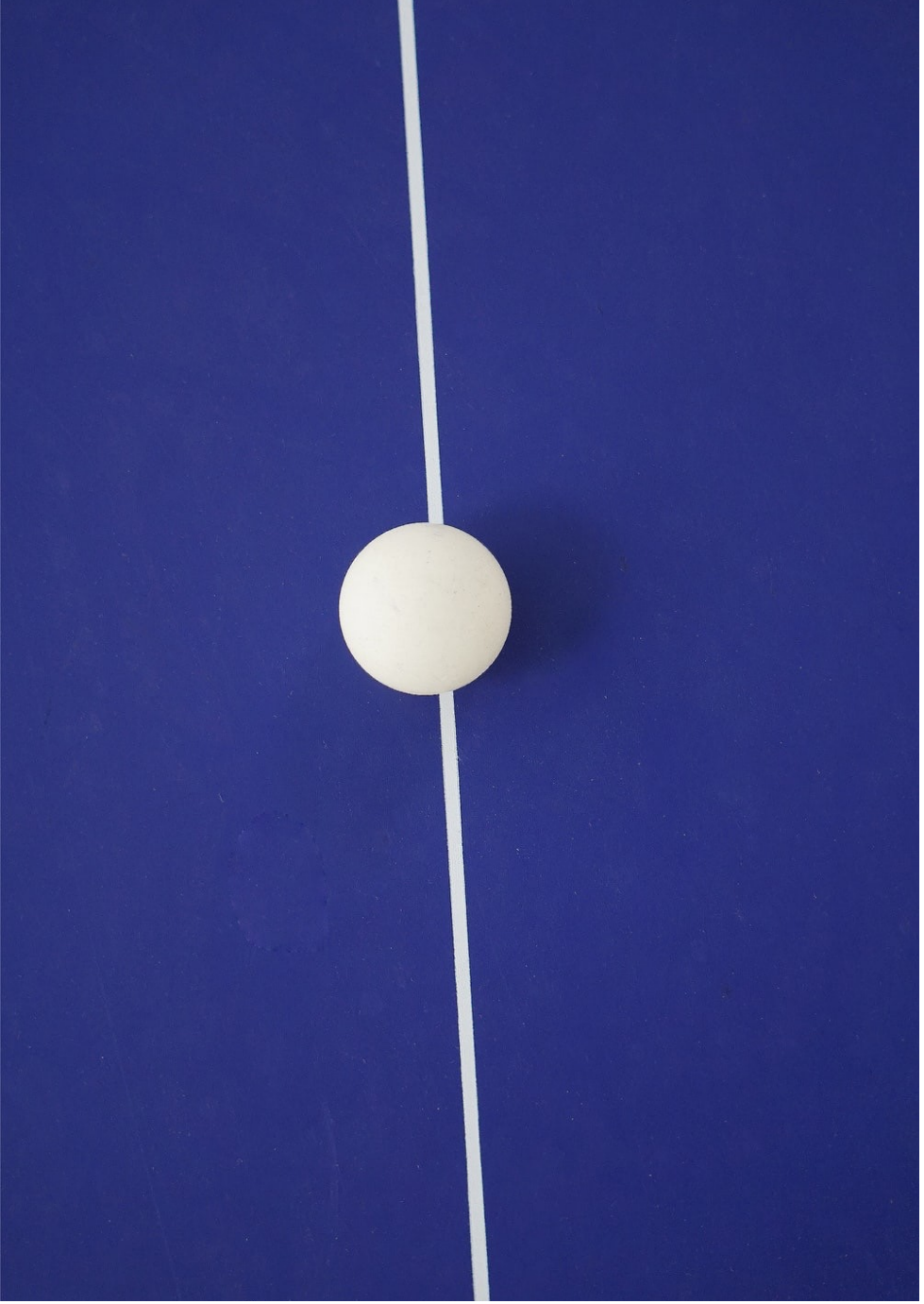
$$= (\|\vec{w}\| \cos\theta)\frac{\vec{v}}{\|\vec{v}\|}$$

$$= \left(\|\vec{w}\| \frac{\vec{v} \cdot \vec{w}}{\|\vec{v}\|\|\vec{w}\|}\right)\frac{\vec{v}}{\|\vec{v}\|}$$

$$= \frac{\vec{v} \cdot \vec{w}}{\|\vec{v}\|^2}\vec{v}$$

$$= \frac{\vec{v} \cdot \vec{w}}{\vec{v} \cdot \vec{v}}\vec{v}$$

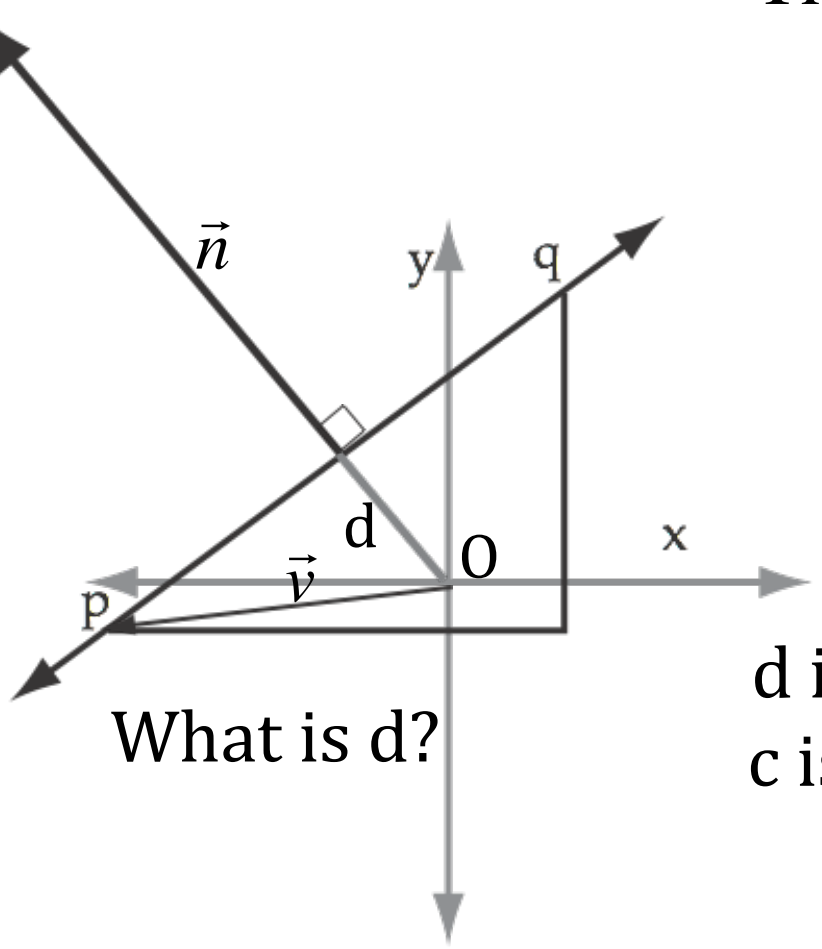
Projection



How can we represent a line?

- Explicit: $y = mx + b$
- Implicit: $Ax + By = c$
- Normal form: $\vec{n} \cdot p - c = 0$
- Parametric form: $\overleftrightarrow{l} = p + \vec{v}t$
- We will use normal & parametric forms

Normal Form



What is d?

The length of $\Pi_{\vec{n}}(\vec{v})$ (the projection of $\vec{v}$ onto $\vec{n}$)

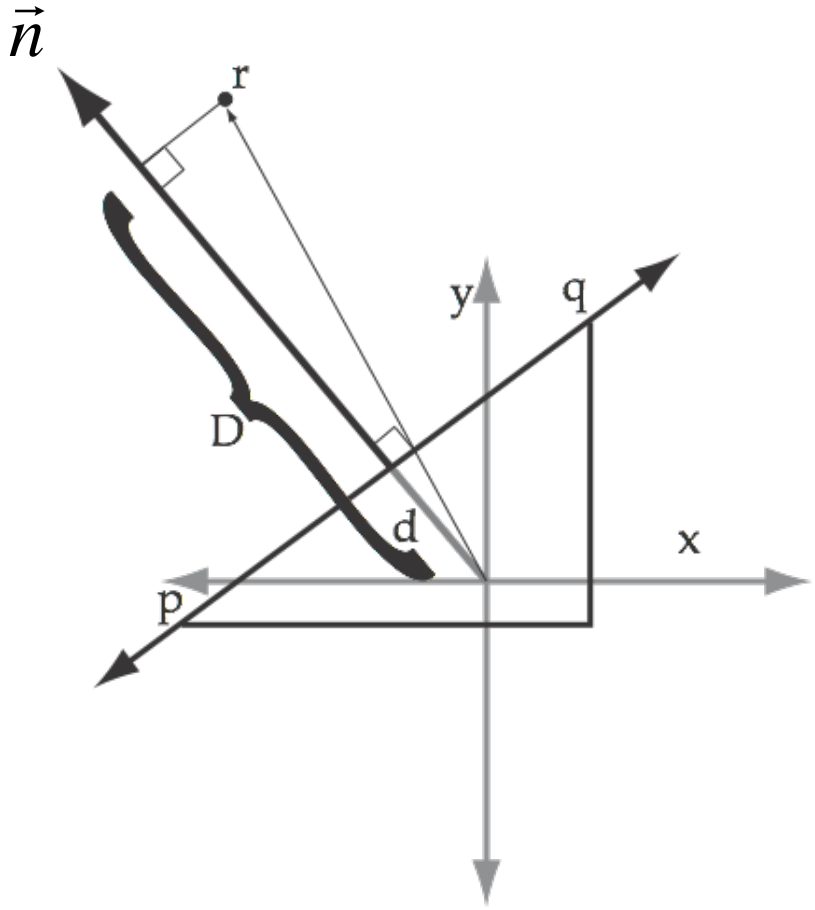
$$d = \frac{\vec{n} \bullet \vec{v}}{\|\vec{n}\|}$$

$$\begin{aligned} \|\vec{n}\| d &= \vec{n} \bullet \vec{v} \\ &= \vec{n} \bullet (p - \mathrm{O}) \\ &= \vec{n} \bullet p \\ &= c \end{aligned}$$

d is the *distance* from the origin to the line
c is the distance scaled by the length of the normal

Distance From A Line

$$dist\left(\vec{l},r\right) = D - d$$
$$= \frac{\vec{n}\bullet r}{\|\vec{n}\|} - \frac{\vec{n}\bullet p}{\|\vec{n}\|}$$
$$= \frac{\vec{n}\bullet r - c}{\|\vec{n}\|}$$
$$= \frac{\vec{n}\bullet r}{\|\vec{n}\|} - d$$



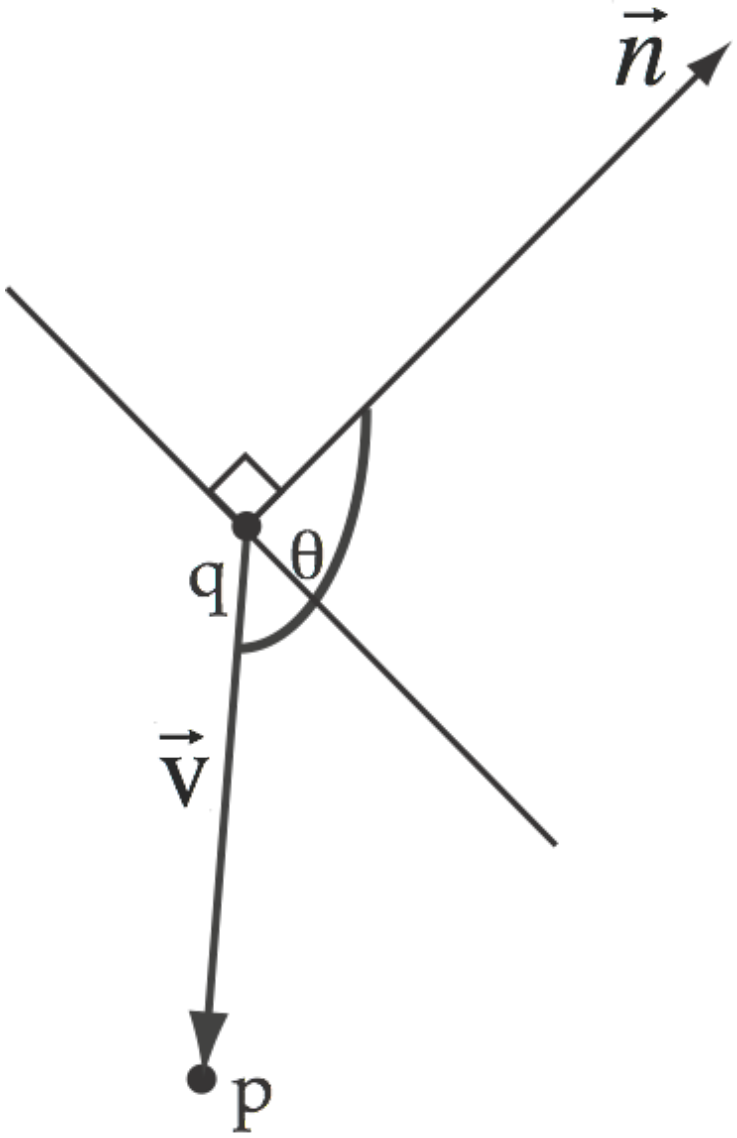
Which side of a line

- Testing which side a point is on is easy with the normal form

$$\frac{\vec{n} \cdot p - c}{\|\vec{n}\|} = \vec{n} \cdot \vec{v}$$

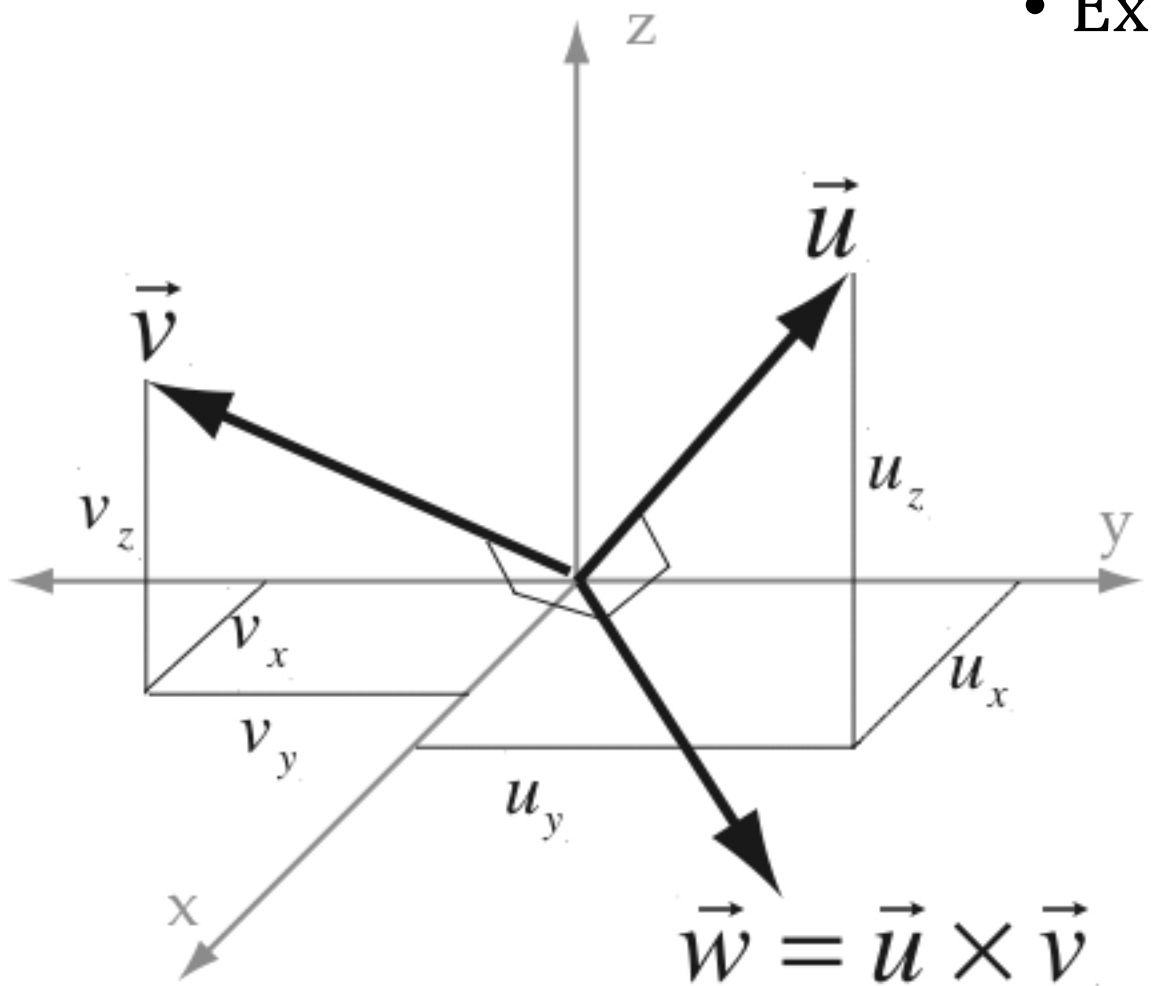
$$= \|\vec{n}\| \|\vec{v}\| \cos\theta$$

  - +: in direction of normal
  - 0: on line
  - -: away from normal



Cross-Product

- Computes $\vec{w}$ perpendicular to $\vec{u}, \vec{v}$
  - in a right-hand system
- Extremely useful!!

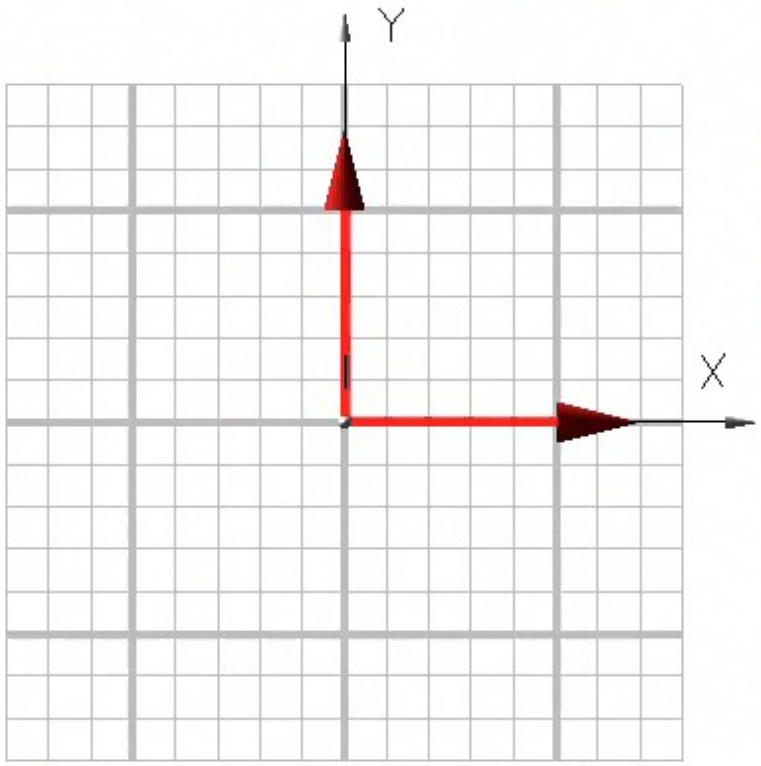


$$\vec{w} = \vec{u} \times \vec{v}$$

$$= \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix}$$

Parametric Plane Equation

- *p is any point in* $\prod$
- *u, v are any two independent vectors in* $\prod$



$$\prod = p + s\vec{u} + t\vec{v}$$

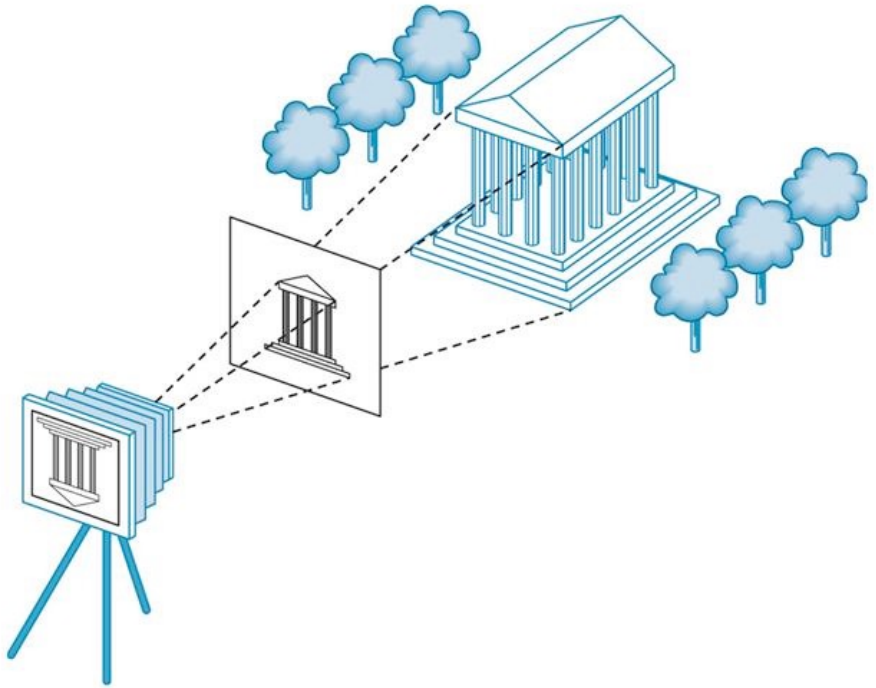
Normal Form of Plane

- Find $n = u \times v$ (the *cross−product*)
- Perpendicular to the plane

$$\vec{n} \cdot q = \vec{n} \cdot \left(p + s\vec{u} + t\vec{v} \right)$$
$$= \vec{n} \cdot p + \vec{n} \cdot s\vec{u} + \vec{n} \cdot t\vec{v}$$
$$= \vec{n} \cdot p + s\left(\vec{n} \cdot \vec{u}\right) + t\left(\vec{n} \cdot \vec{v}\right)$$
$$= \vec{n} \cdot p + s(0) + t(0)$$
$$= c(\text{a constant})$$
$$\vec{n} \cdot q - c = 0$$

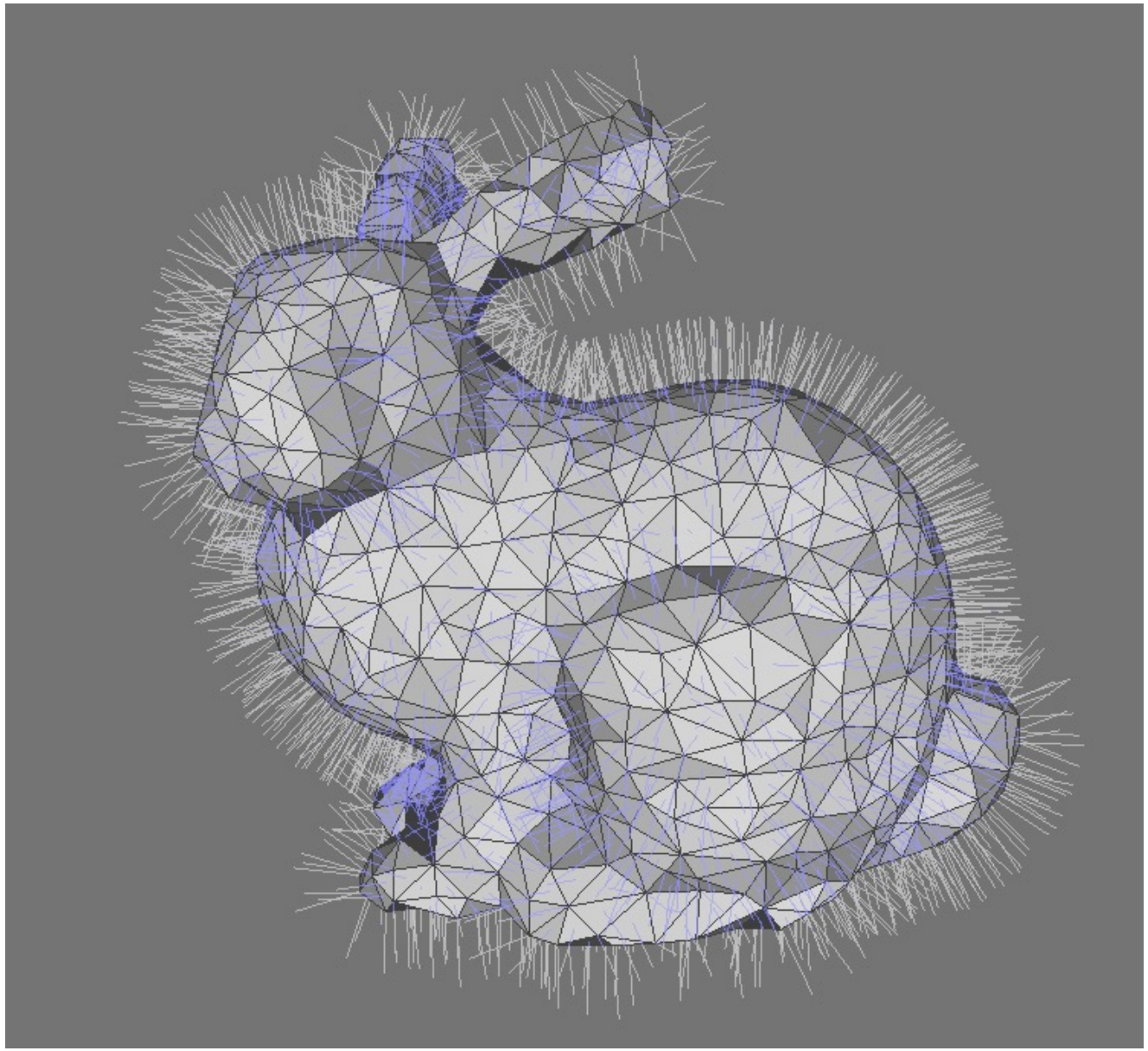
The Synthetic Camera

FIGURE 1.18
Imaging with the synthetic camera.



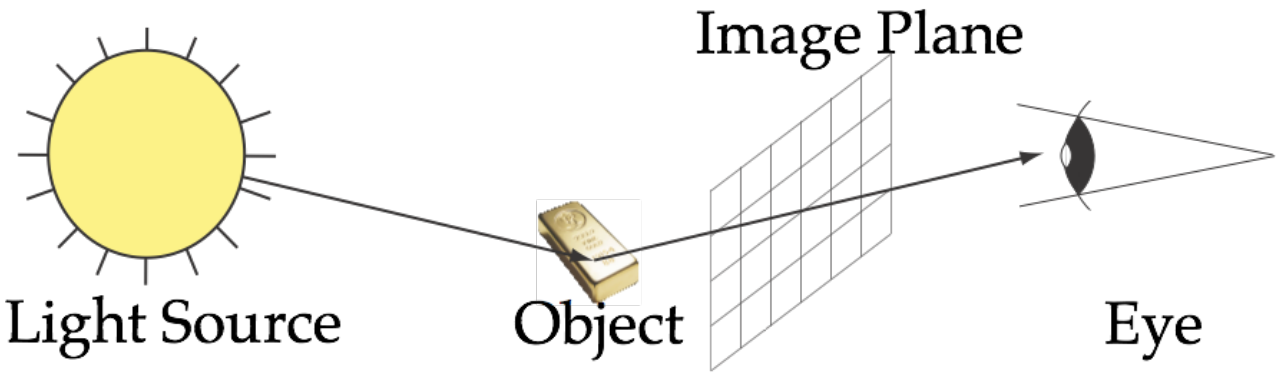
Assume we have:

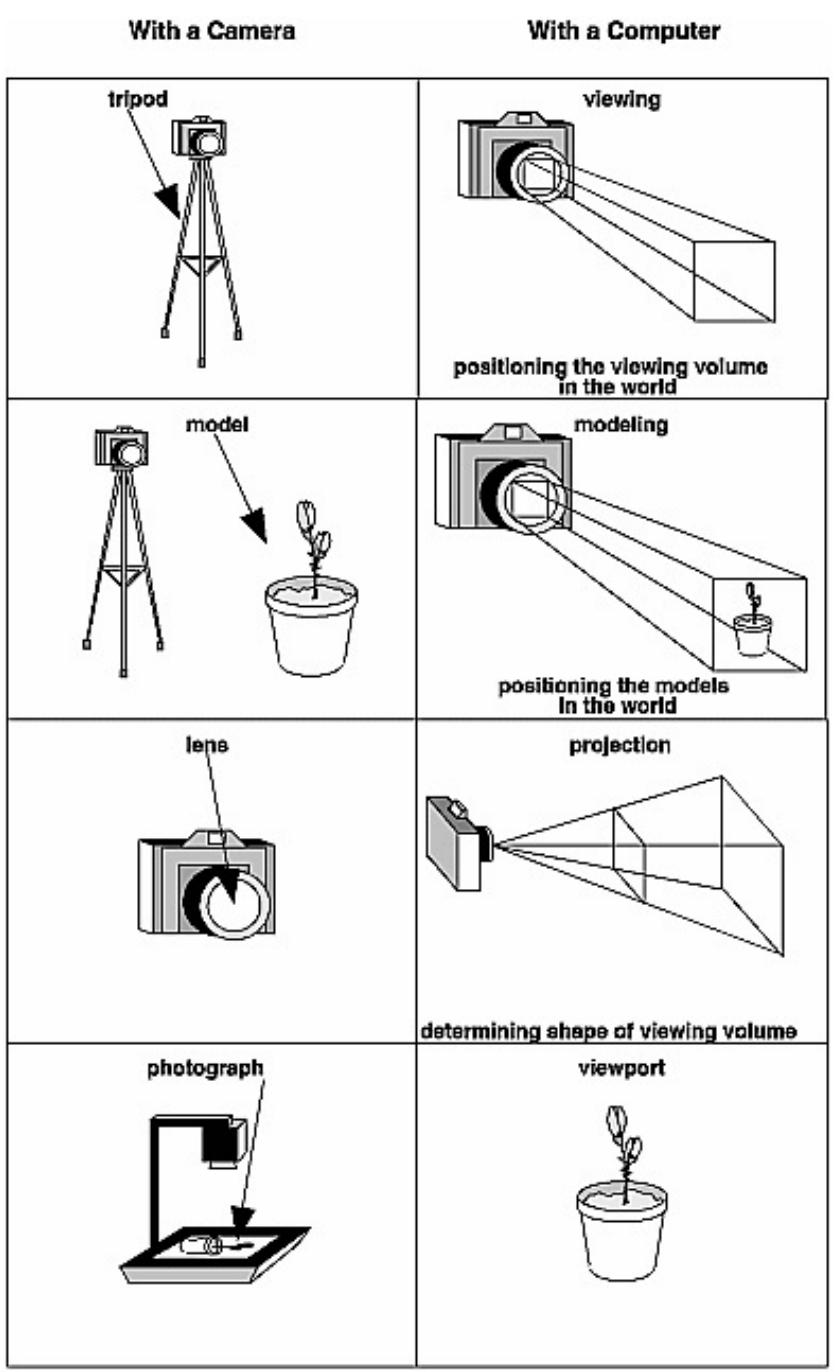
- A triangle mesh M, with:
  - Vertex positions
  - Vertex normals
  - Texture coordinates (& a texture)
- An eye E
- An image plane Π made up of pixels



Alberti's Window

- Place glass sheet in front of the eye
- Draw what you see on the glass sheet
- Then take the sheet with you





Use Geometry to Place Camera

(View Transformation)

Use Geometry to Place Models

(Model Transformation)

Use Projection to Describe Camera

(Projection Transformation)

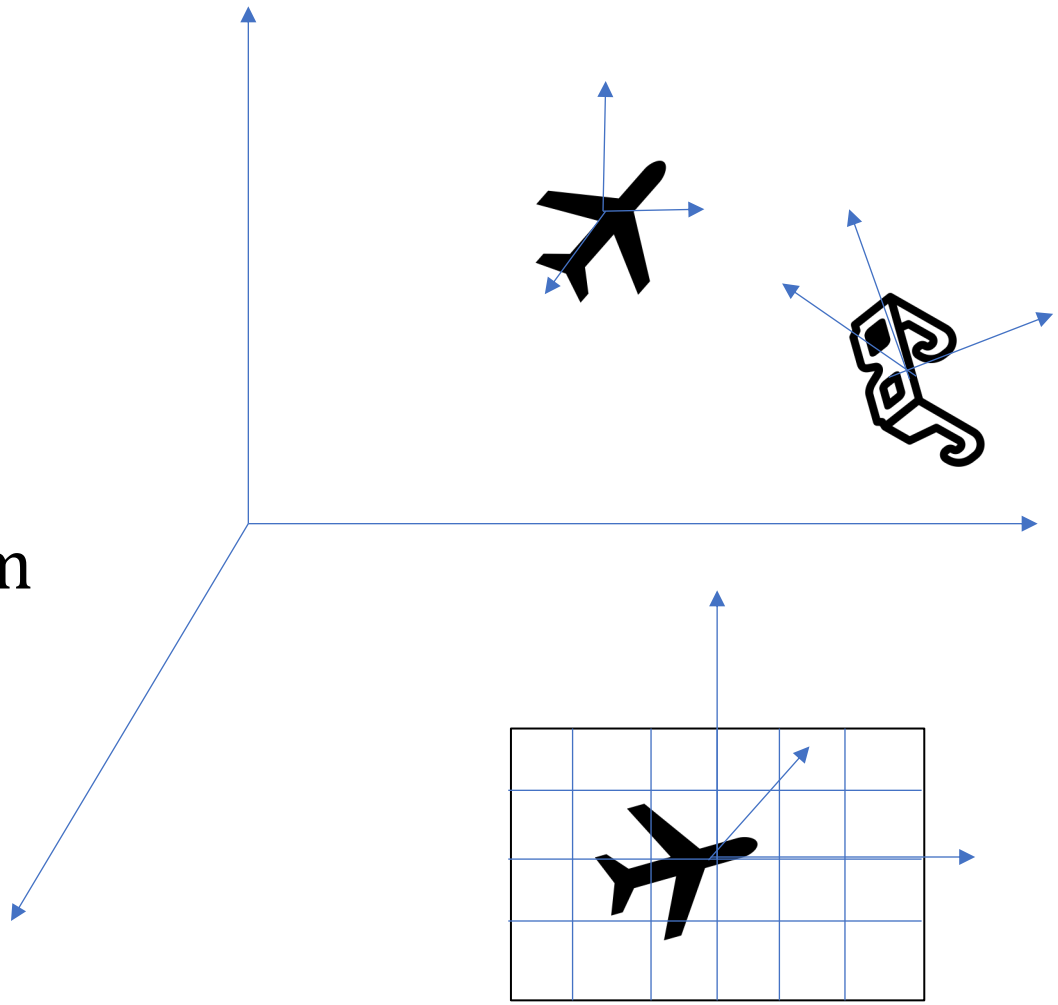
Use Projection to Describe Screen

(Viewport Transformation)

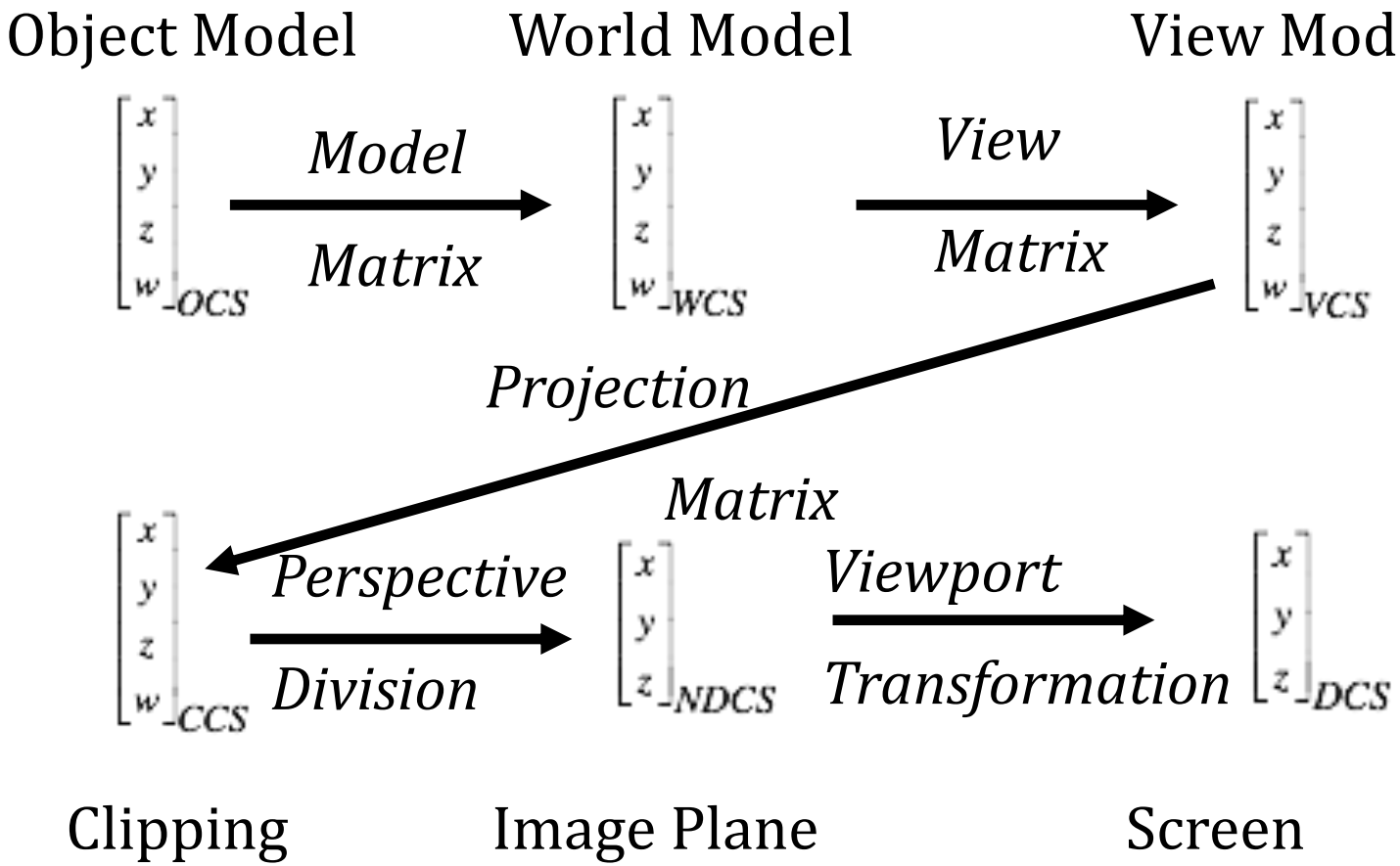
Synthetic Camera

Coordinate Systems

- OCS is the Object Coordinate System
- WCS is the World Coordinate System
- VCS is the View Coordinate System
- CCS is the Clipping Coordinate System
- NDCS is the Normalized DCS
- DCS is the Device Coordinate System

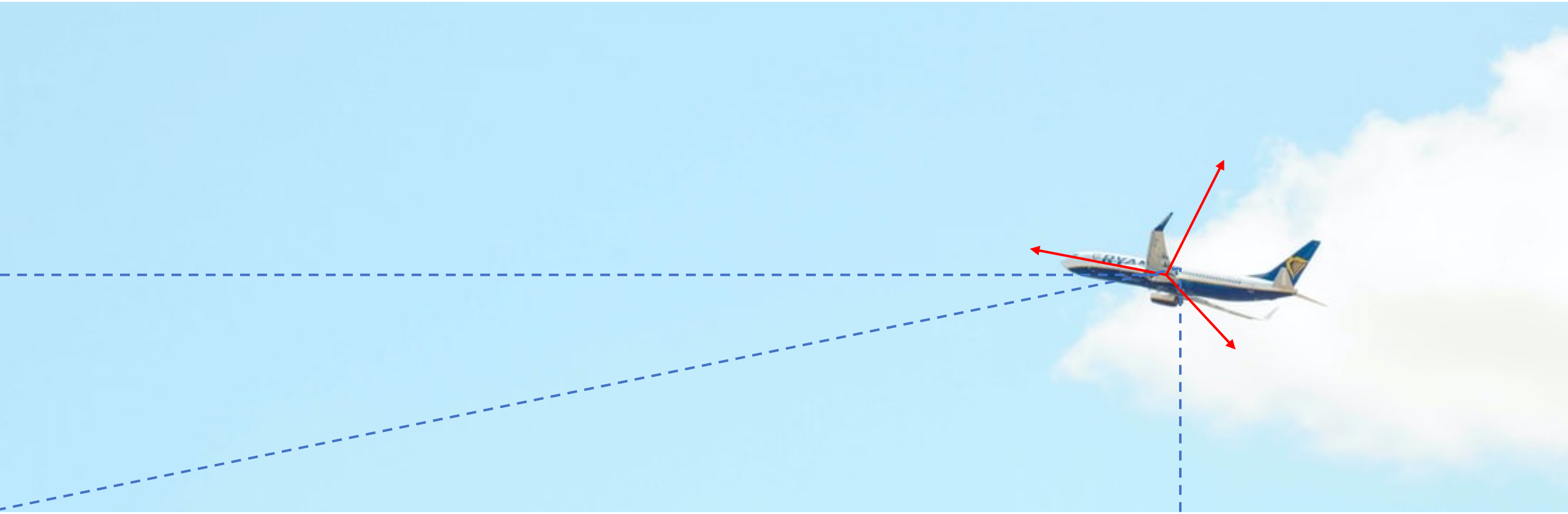


Why so many?



Actually makes our lives easier when representing a scene

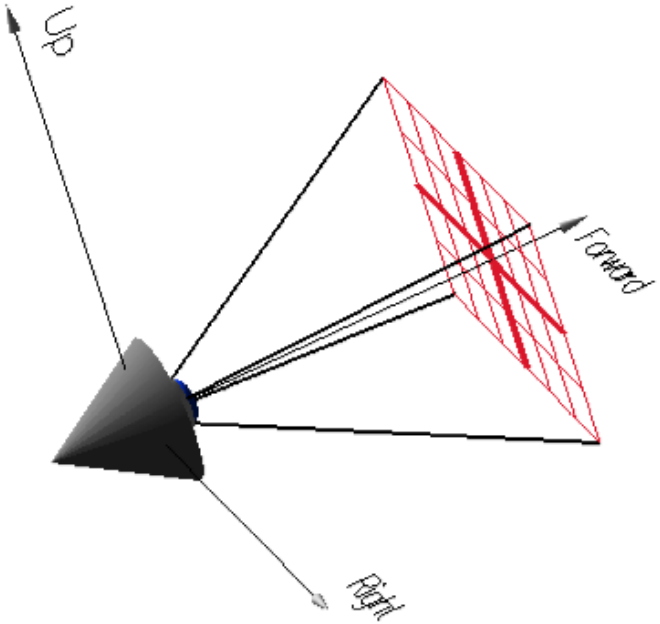
Object Coordinate System



World Coordinate System

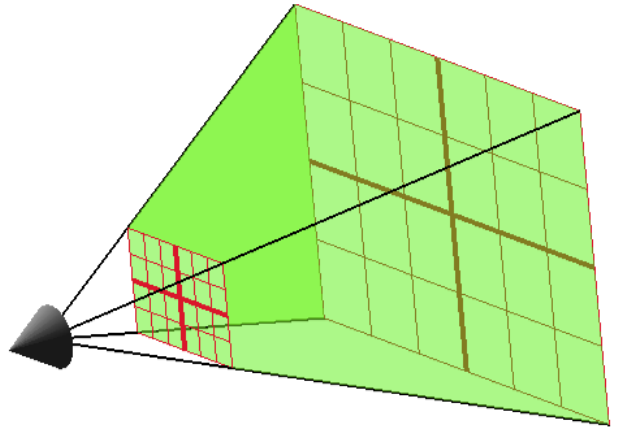
View Coordinates

- Coordinates from *camera's* viewpoint
- Allows viewpoint to move in the world
- Effectively, an extra rotation & translation



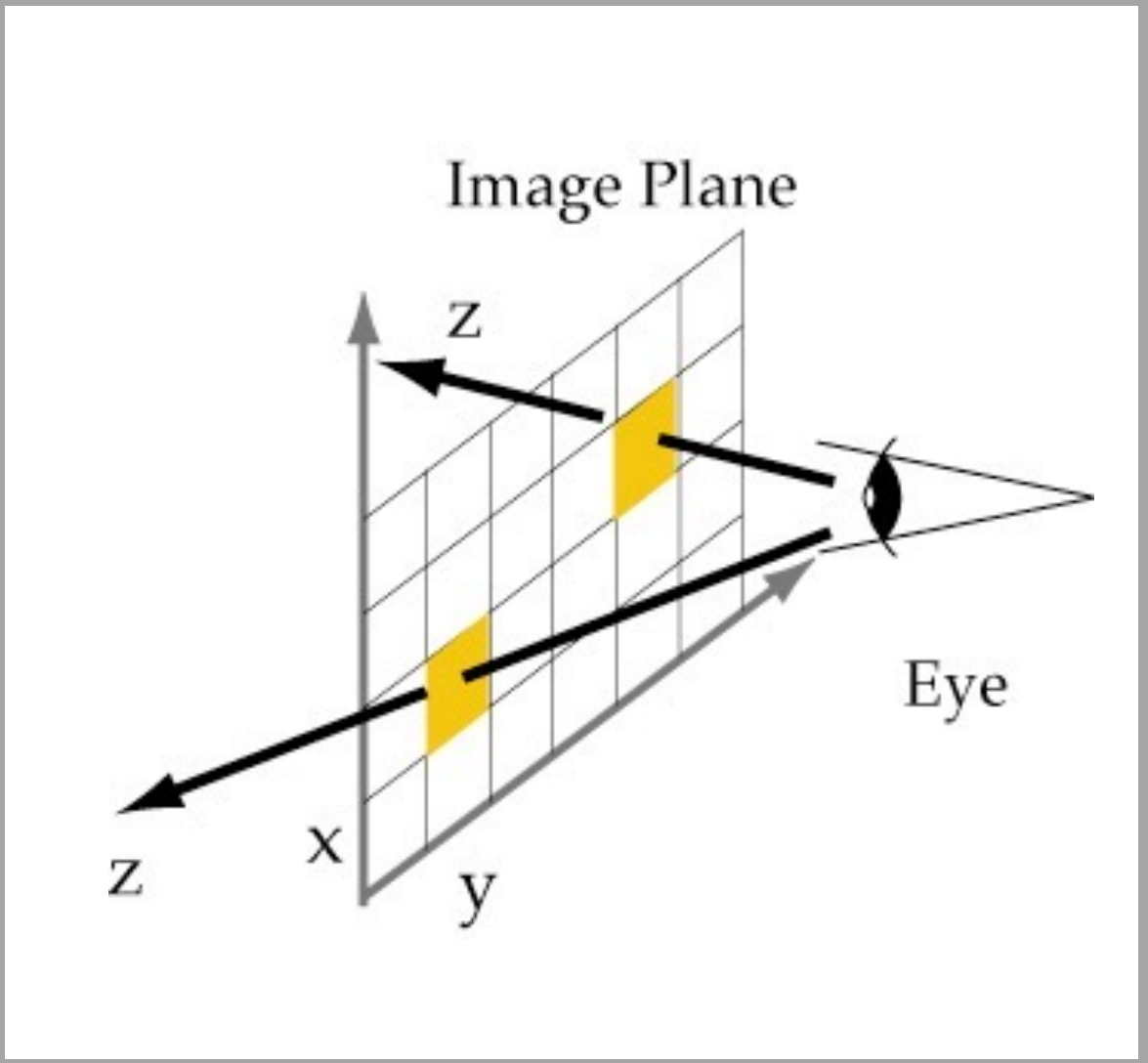
View Frustrum

- For perspective, view volume is
  - a view frustum (a truncated pyramid)
- For orthogonal, view volume is a box



Clipping Coordinates

- Point coordinates after perspective projection.
- Projection: remember Alberti's window.
- They are "2D", with z being the distance from the eye



Normalized DCS

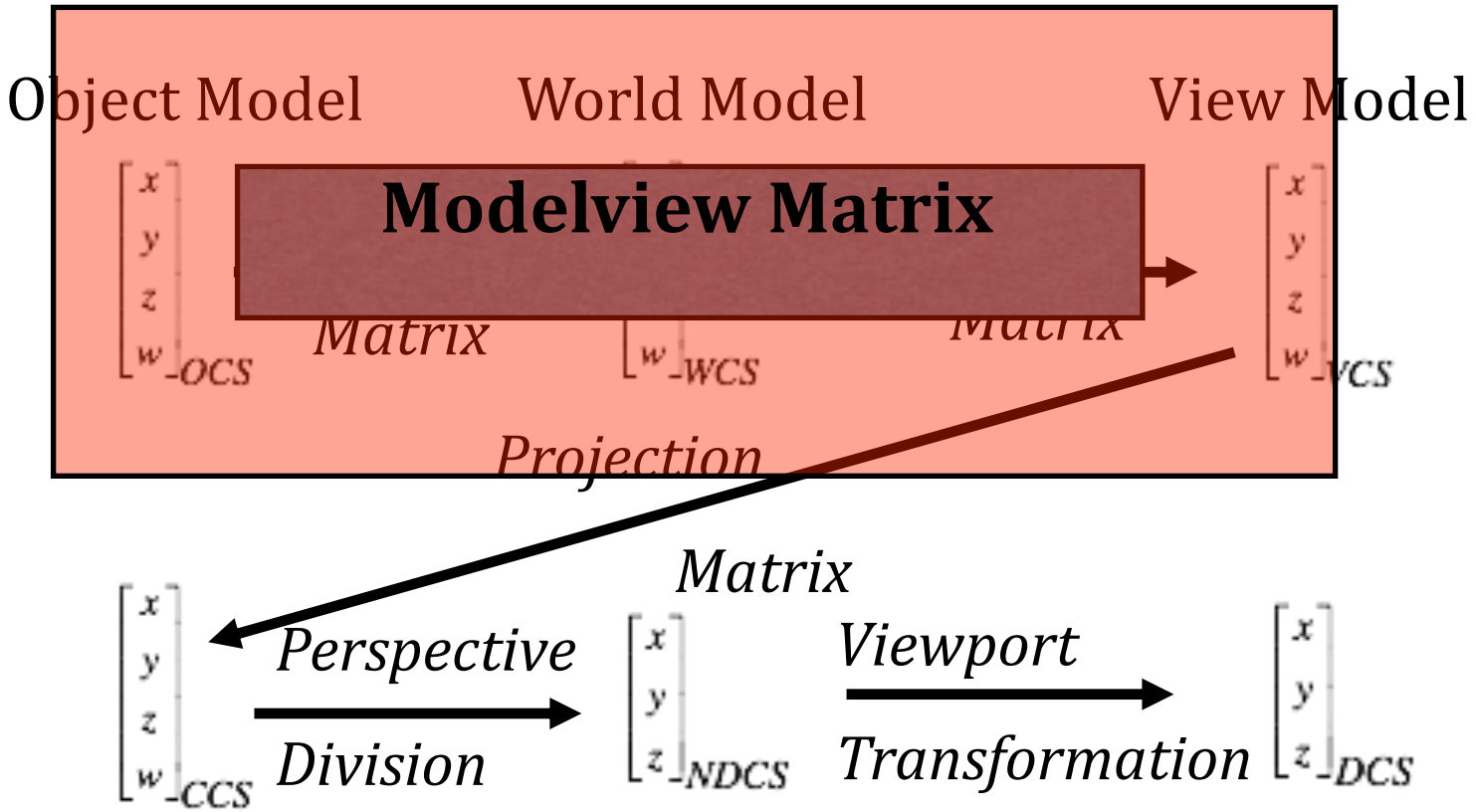
- Normalized Device Coordinate System
- Divide CCS through by w
  - converts homogeneous coords to Cartesian
  - z will be used for z buffer
- Independent of screen coordinates
- I.e. range is [0,1]x[0,1]x[0,1]

Example:
(1.5, 1, -1, 1) --> (1.5, 1, -1)
(1.5, 1, 0.2, 2) --> (0.75, 0.5, 0.1)
(3, 2, 0.2, 2) --> (1.5, 1, 0.1)

Device Coordinates (DCS)

- Device coordinates are:
  - used for final render to screen
  - expressed in *pixel* coordinates
  - x, y: position on image plane, z: “depth” in front of image plane
  - normalised from [0..1] to [0..255] or [0..65536]

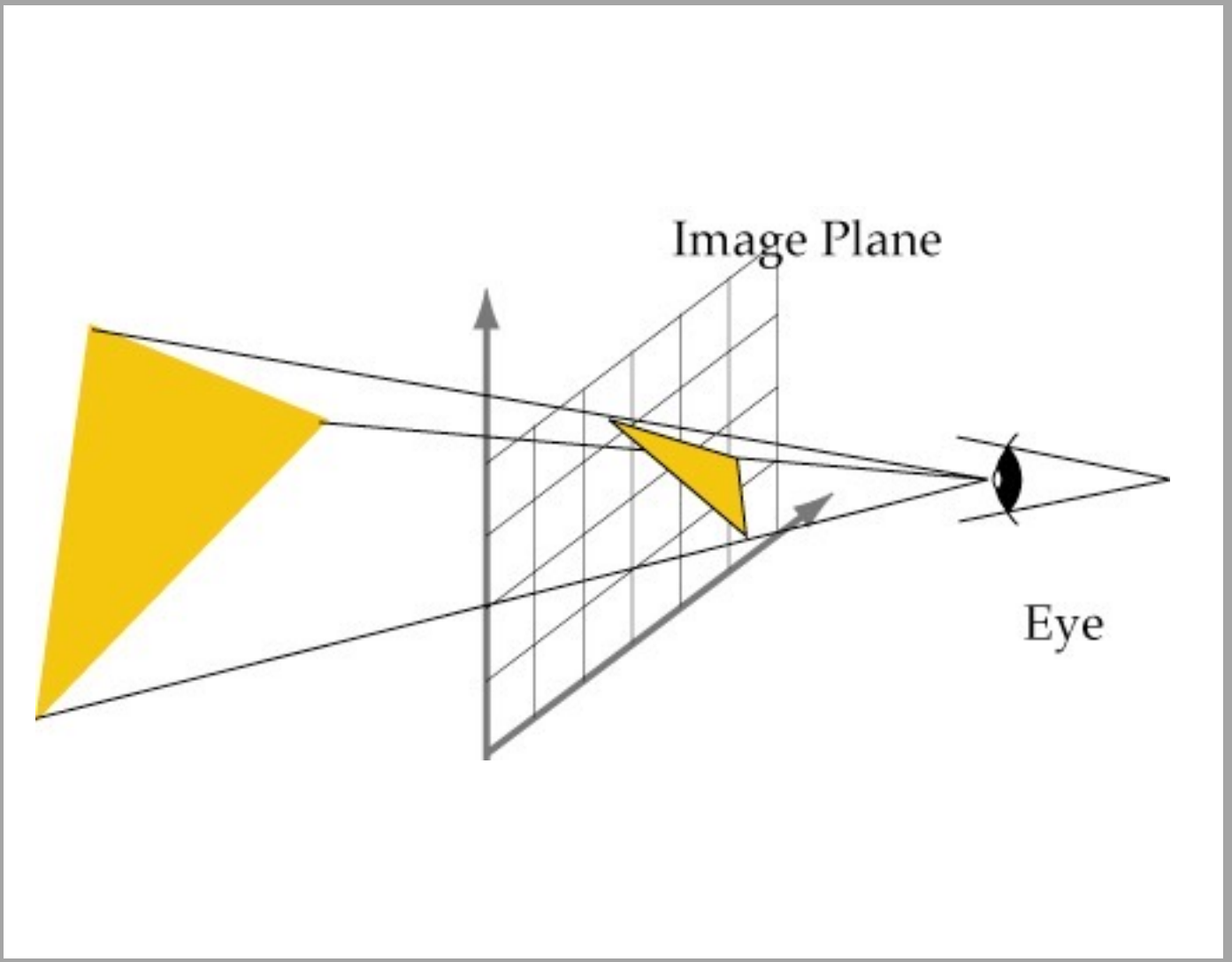
FF's Biggest Mistake



An optimization, but makes understanding the transformations more confusing.

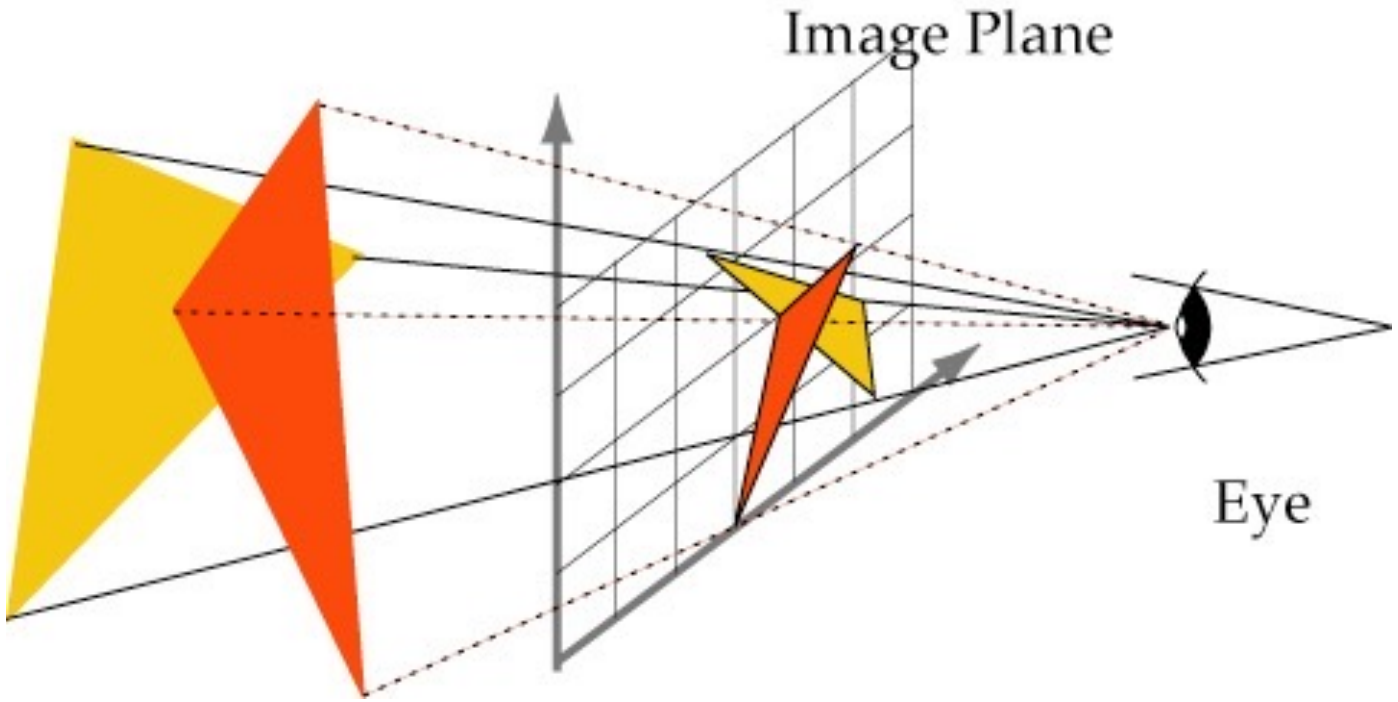
Projective Rendering

- Fundamental trick behind video card
  - Project each triangle to image plane
  - Triangles *always* project as triangles
  - We need to rasterise (draw) lines & triangles



Painter's Algorithm

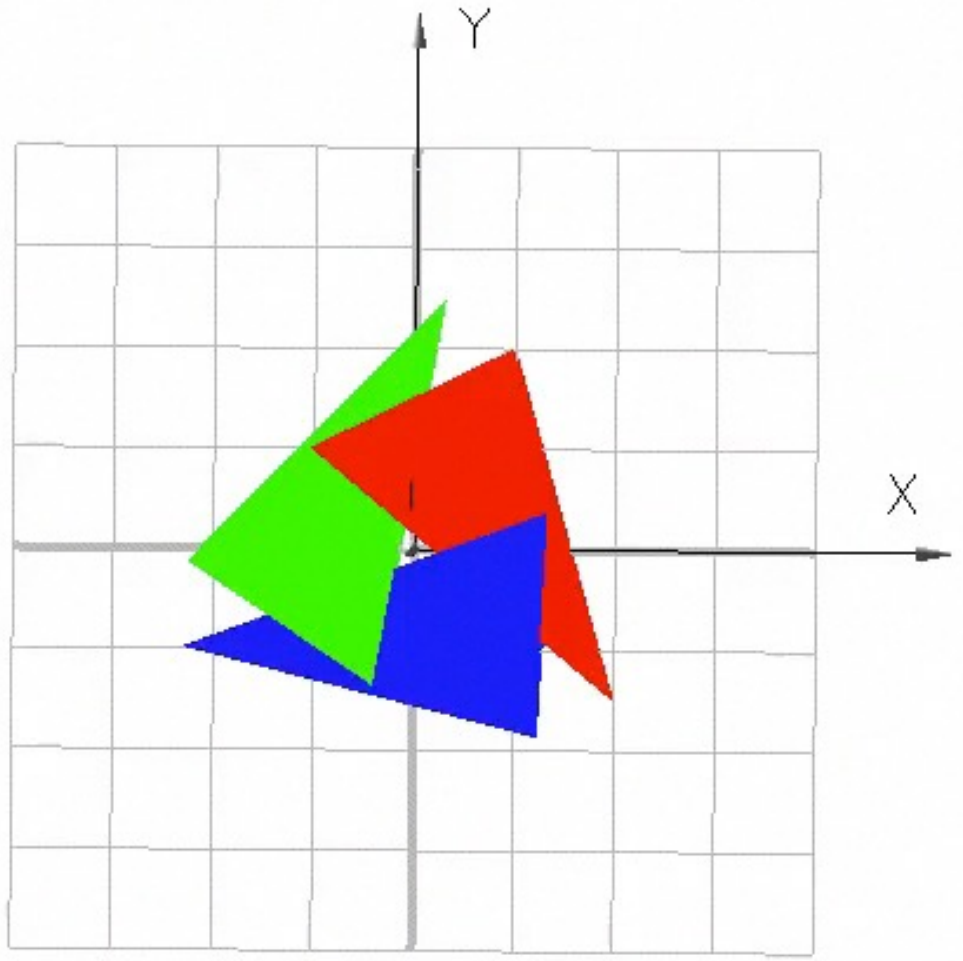
- If we draw objects from back to front
  - the back objects will be occluded
    - i.e. we will see only the front object
- We have to sort all objects for each image



Worst Case

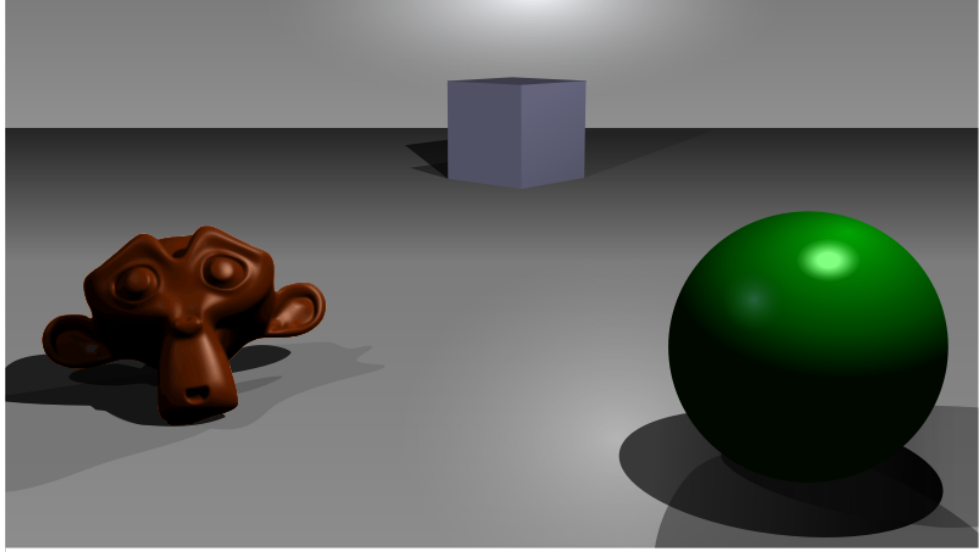
- Three triangles
  - Red overlaps Green
  - Green overlaps Blue
  - Blue overlaps Red

- Painter's Algorithm fails!

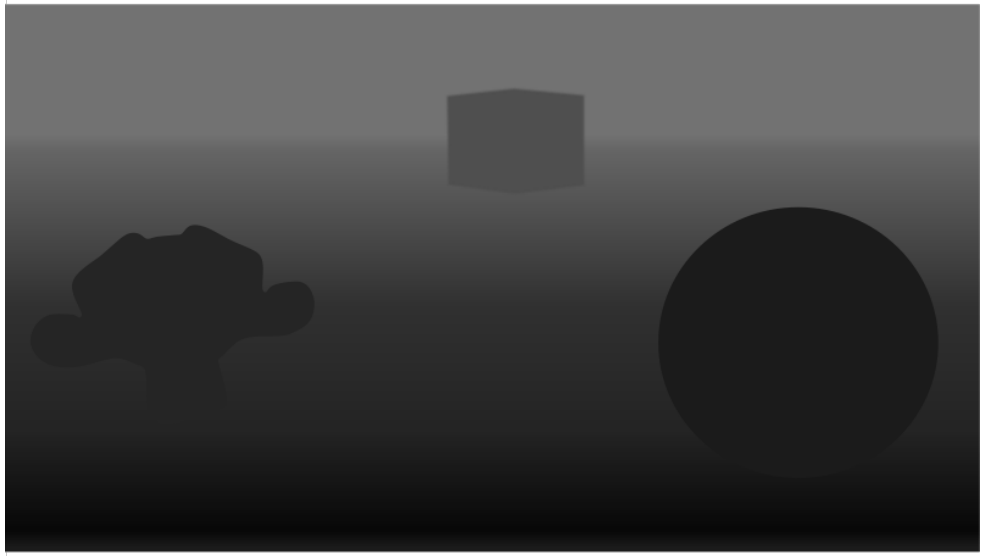


Depth (Z-) Buffering

- For each pixel, set Z = -∞
- For each triangle
  - Transform to image coordinates
  - Rasterise to *fragments* - possible pixels
  - For each fragment (X, Y, Z, R, G, B)
    - If depth(X,Y) < Z, discard
    - Else depth(X,Y) = Z, colour(X,Y) = (R,G,B)

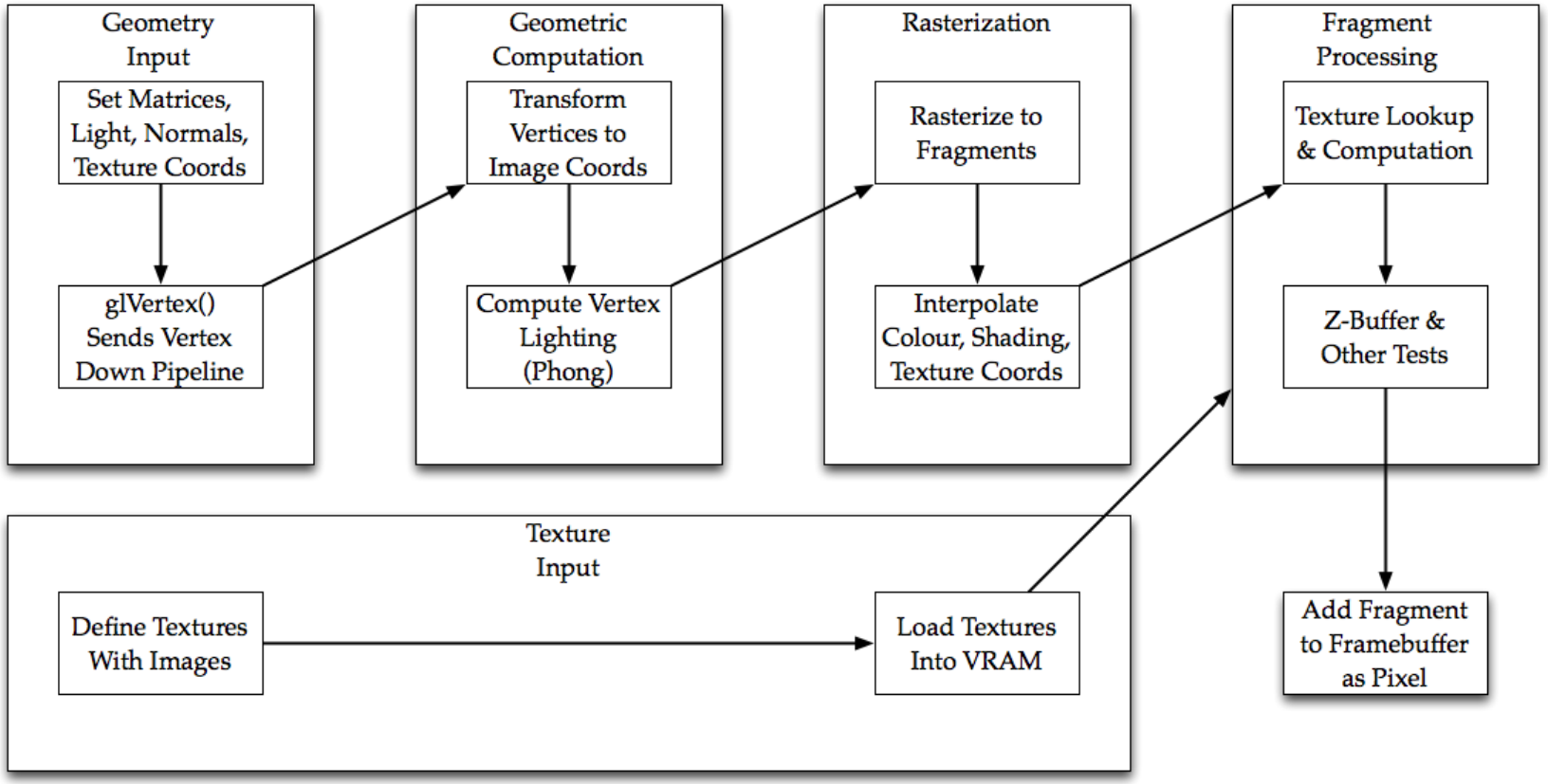


A simple three-dimensional scene



Z-buffer representation

Projective Pipeline

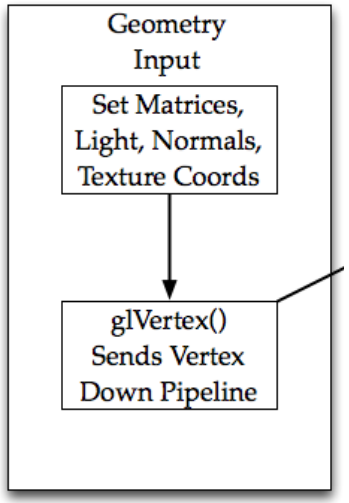


State & Streams

- Textures, matrices, render options, Vertex attributes are states!
- State is applied to passing vertices
  - And must be changed explicitly

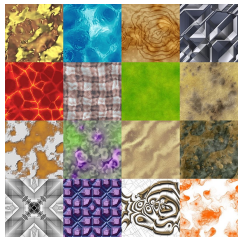
Input Stage

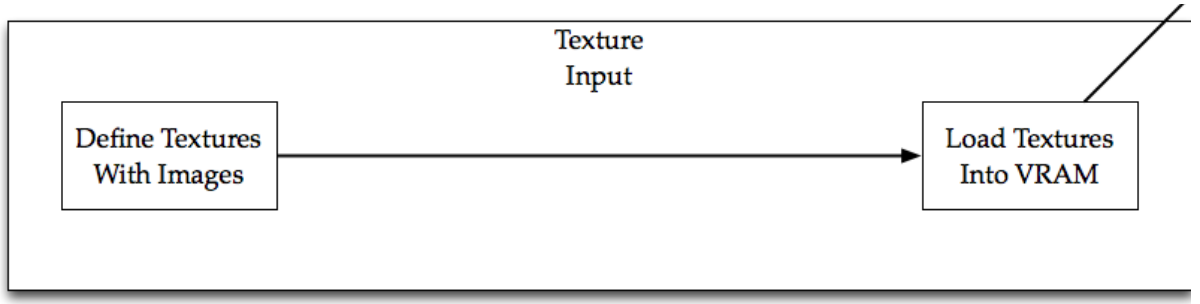
- Each set of vertices defines a primitive
- Attributes are applied first
- Optimisations come later on:
  - Vertex Arrays
  - Vertex Buffer Objects (VBOs)
  - Tessellation (in CPU library)



0.3	0.1	0.4	1.3	...
0.4	1.1	1.3	1.2	...

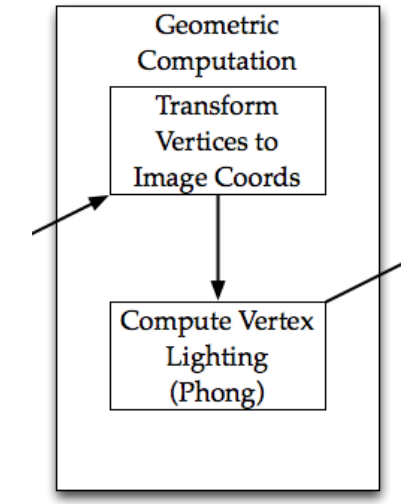
$$\begin{bmatrix} 1 & \cdots & 0.4 \\ \vdots & \ddots & \vdots \\ 3 & \cdots & 0.5 \end{bmatrix}$$





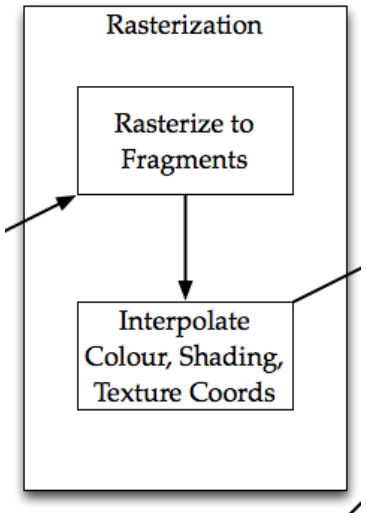
Geometry Stage

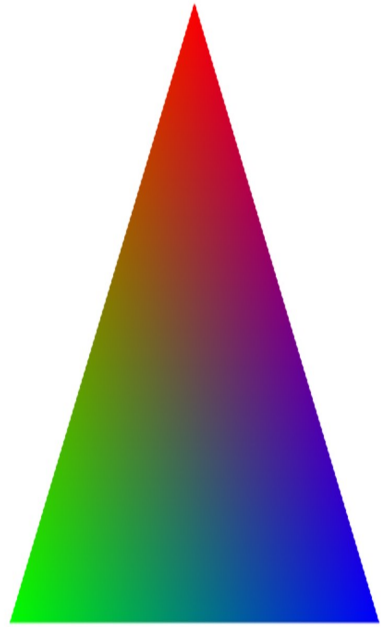
- Geometric transformations are applied
  - Using homogeneous matrices
- Lighting calculated per vertex
- Gives one colour per vertex
- Colours interpolated during rasterization
- Programmable with vertex shaders, &c.
- Vertices cached for later reuse



Rasterisation Stage

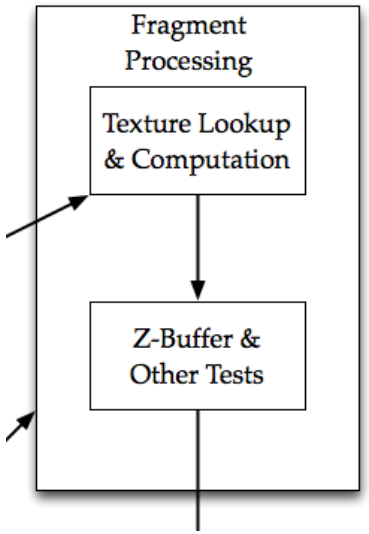
- Primitives are rasterised into fragments
- Barycentric coordinates are computed
- Attributes are interpolated:
  - colour after lighting
  - texture coordinates (later)
  - depth
- Not yet programmable with shaders

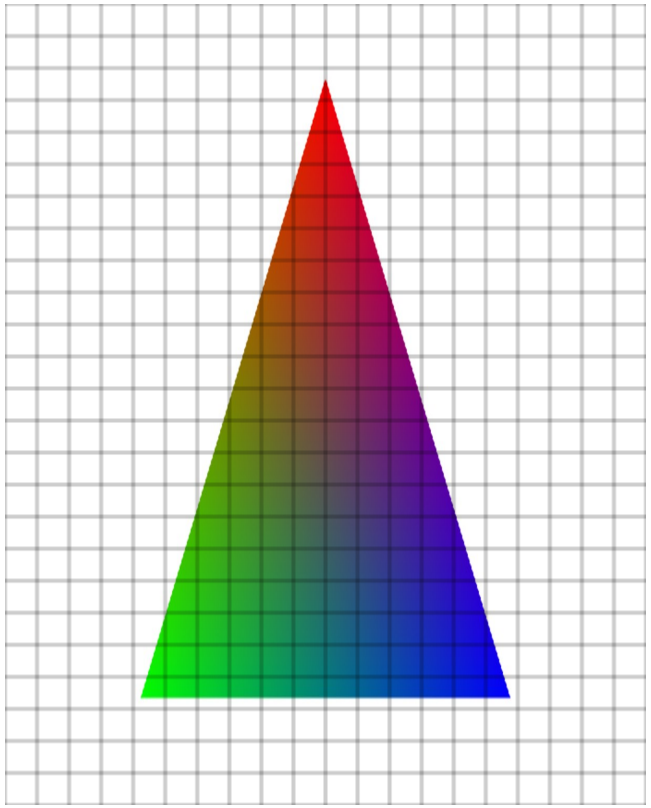




Fragment Processing

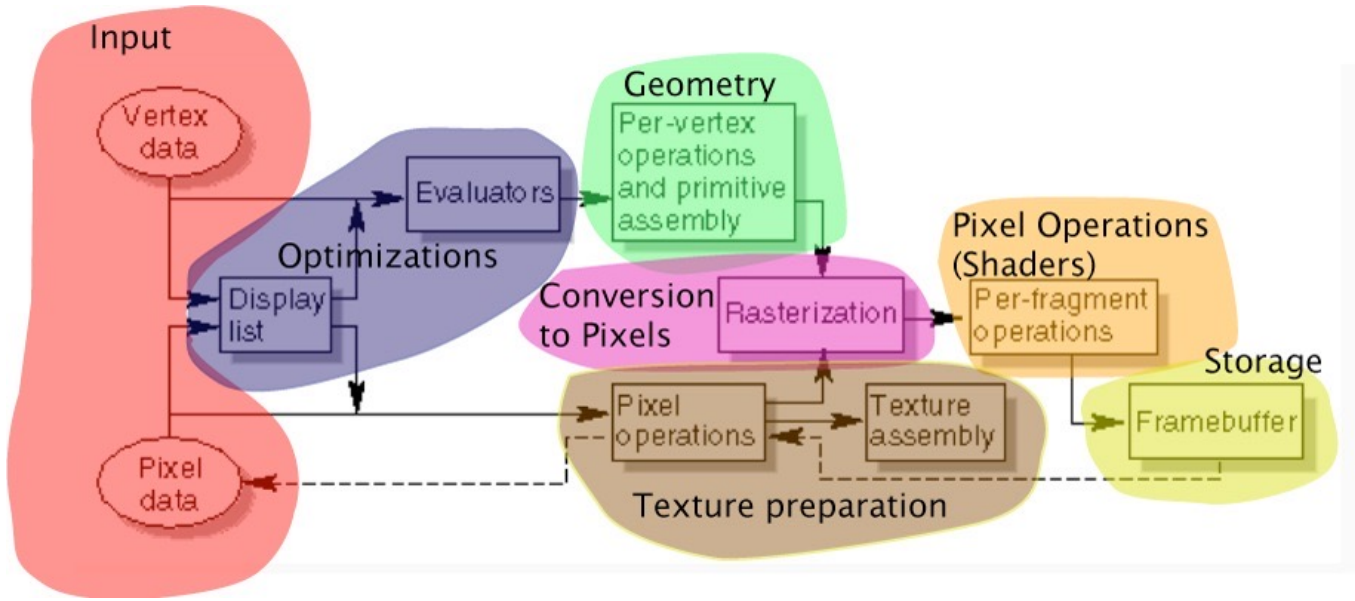
- Each fragment is now processed separately
  - in parallel
  - textures are computed
  - depth and other tests performed
    - to see if we keep the fragment
  - fragment is written to output image
- Programmable with fragment shaders





1st Gen Hardware - Fixed Function Pipeline

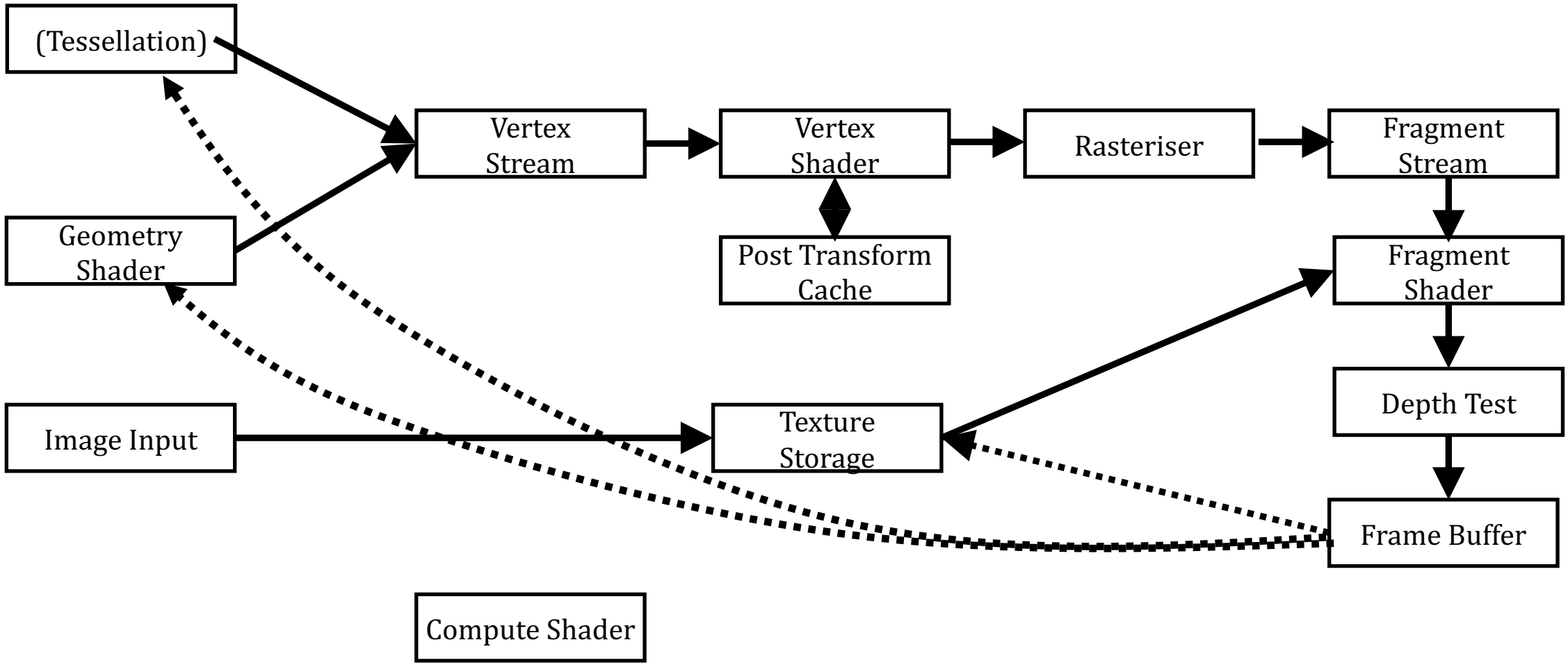
- This is the *original* hardware design
- Each stage uses different processors
- Now simulated by GP processors
- Everything can be done in parallel (depth test must be atomic)
- Deeply pipelined, massively parallel



Programmable Pipeline

- Drop in replacements for particular stages
  - Vertex Shaders
  - Tessellation Control Shaders
  - Tessellation Evaluation Shaders
  - Geometry Shaders
  - Fragment Shaders
  - Compute Shaders

Evolved Pipeline



Problems

- What if we want to do things out of order?
  - E.g. do depth test before fragment shader
- What if we want to preprocess *all* vertices?
  - Instead of using the post-transform cache
- The pipeline model is breaking down
  - Progressively replaced by arbitrary compute
  - But still the basis of most libraries

Images by

- S2 - Beazy@Unsplash
- S9 - Kiran CK@Unsplash
- S18 - Slrncl@Unsplash
- S22 - Kaotaru@Unsplash
- S31 - By -Zeus- - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=7355760
- S33- By Drummyfish - Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=76755935