

Interactive Computer Graphics: Lecture 1

3D graphical scenes:
Projections and Transformations

Two dimensional graphics

- The lowest level of graphics processing operates directly on the pixels in a window provided by the operating system.
- Typical Primitives are:

```
SetPixel(int x, int y, int colour);  
DrawLine(int xs, int ys, int xf, int yf);
```

- etc.

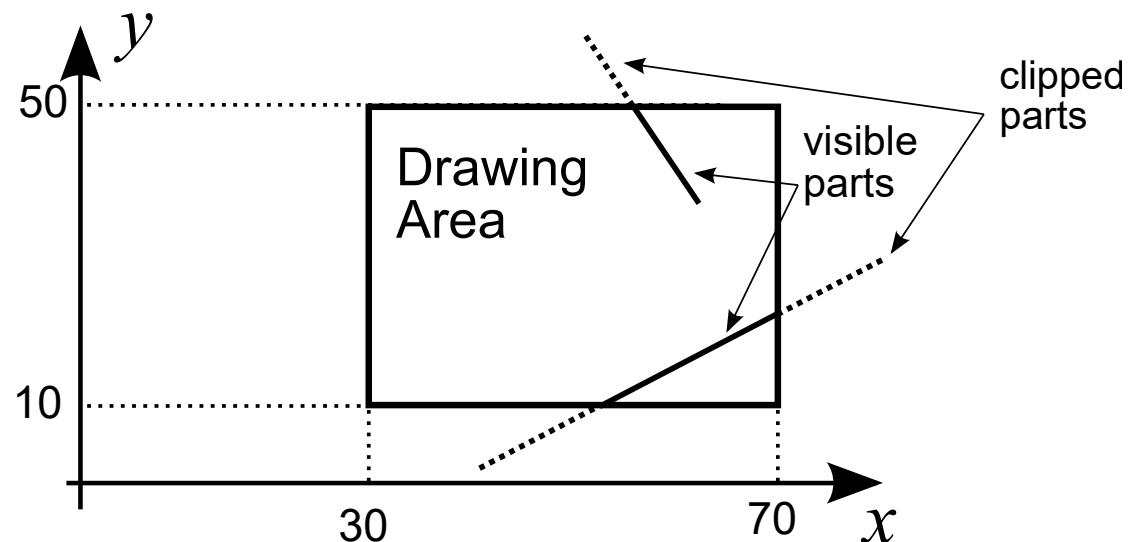
World coordinate systems

- To achieve *device independence* when drawing objects we can define a **world coordinate system**.
- This will define our drawing area in units that are suited to the application:
 - meters
 - light years
 - microns
 - etc

Example

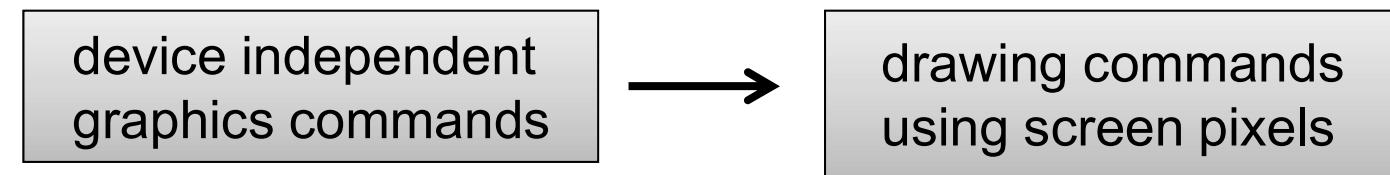
We can give our window ‘World Coordinates’ and draw objects using them.

```
SetWindow(30, 10, 70, 50)  
DrawLine(40, 3, 90, 30)  
DrawLine(50, 60, 60, 40)
```



Normalisation

To make the conversion



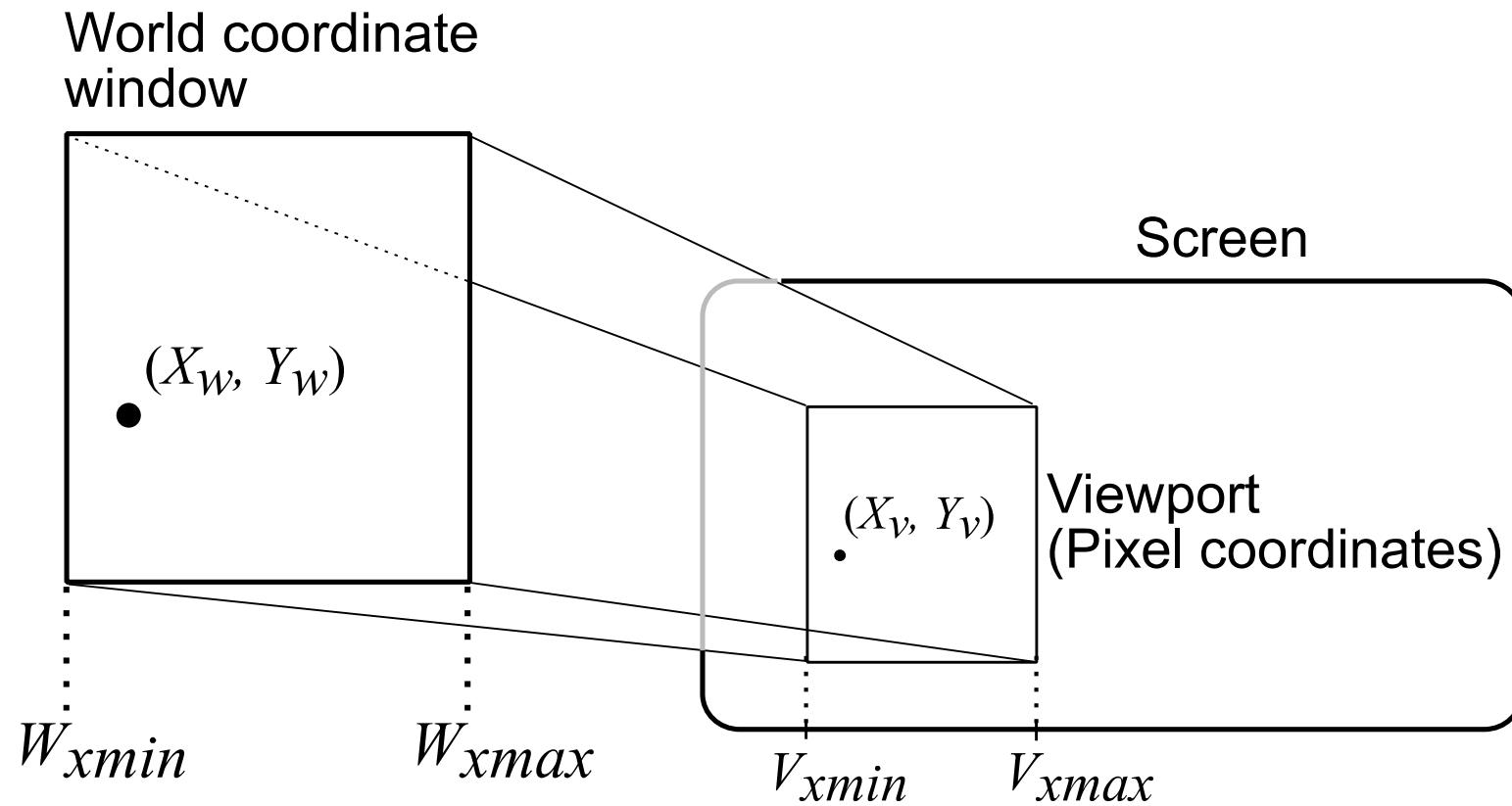
we need a process of normalisation

First we must ask the operating system* for the pixel addresses of the corners of the area we are using.

Then we can translate our world coordinates to pixel coordinates.

*making a ‘system call’ through the API

Normalisation



Normalisation

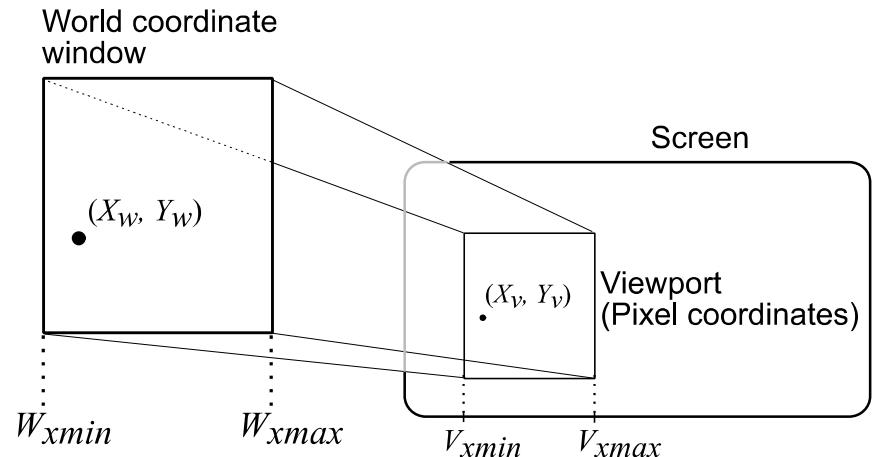
- Having defined our world coordinates, and obtained our device coordinates we relate the two by simple ratios:

$$\frac{(X_w - W_{xmin})}{(W_{xmax} - W_{xmin})} = \frac{(X_v - V_{xmin})}{(V_{xmax} - V_{xmin})}$$

- Rearranging, we get:

$$X_v = \frac{(X_w - W_{xmin})(V_{xmax} - V_{xmin})}{W_{xmax} - W_{xmin}} + V_{xmin}$$

- with a similar expression for Y_v



Normalisation

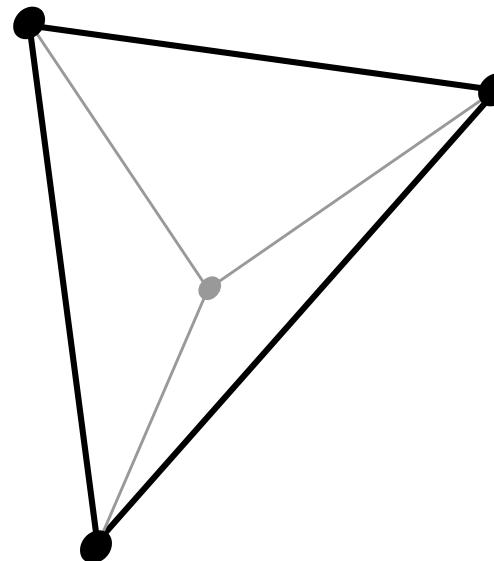
- So we have two equations for calculating pixel coordinates (X_v, Y_v) .
- We can simplify them to form a simple pair of linear equations:

$$\begin{aligned} X_v &= AX_w + B \\ Y_v &= CY_w + D \end{aligned}$$

- Here A, B, C and D are constants that define the normalisation. A, B, C, D are found from the known values of $W_{xmin}, V_{xmin}, \dots$

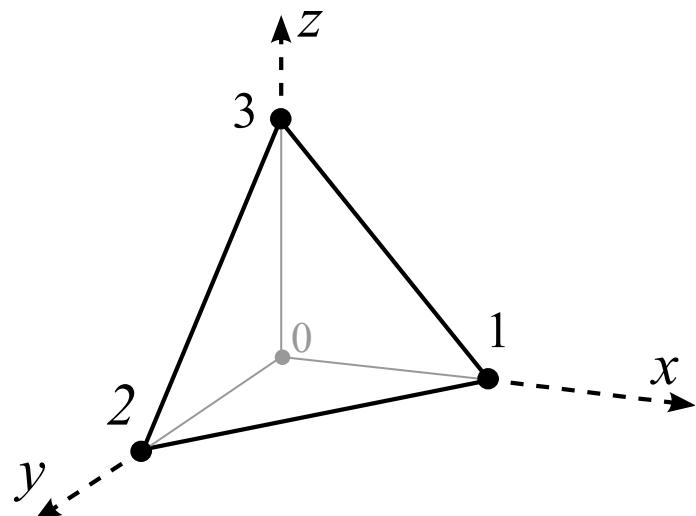
Polygon rendering

- Many graphics applications use scenes built out of planar polyhedra.
- These are three dimensional objects whose faces are all planar polygons (often called faces or facets).



Representing planar polygons

- In order to represent planar polygons in the computer we need a mixture of different data:
 - Numerical Data
 - Actual 3D coordinates of vertices, etc.
 - Topological Data
 - Details of what is connected to what.

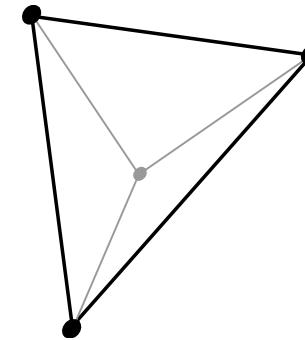


Vertex data	
Index	Location
0	(0, 0, 0)
1	(1, 0, 0)
2	(0, 1, 0)
3	(0, 0, 1)

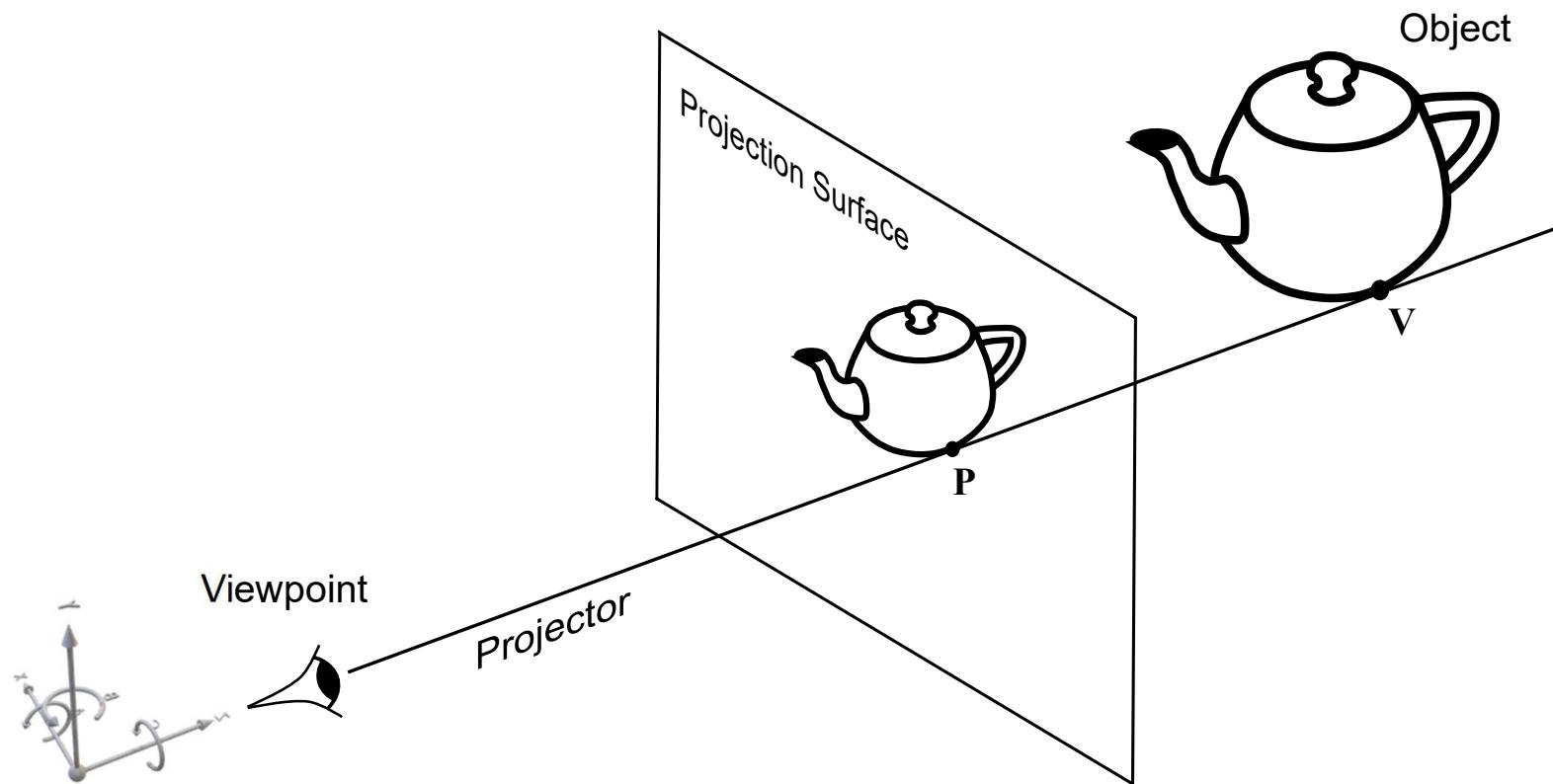
Face data	
Index	Vertices
0	0 1 3
1	0 2 1
2	0 3 2
3	1 2 3

Projections of wire frame models

- Wire frame models simply include points and lines.
- In order to draw a 3D wire frame model we must:
 - First convert the points to a 2D representation.
 - Then we can use simple drawing primitives to draw them.
- The conversion from 3D into 2D is a *projection*.



Projection



The projector takes a point on the object to a point on 2D projection surface.

Non-linear projections

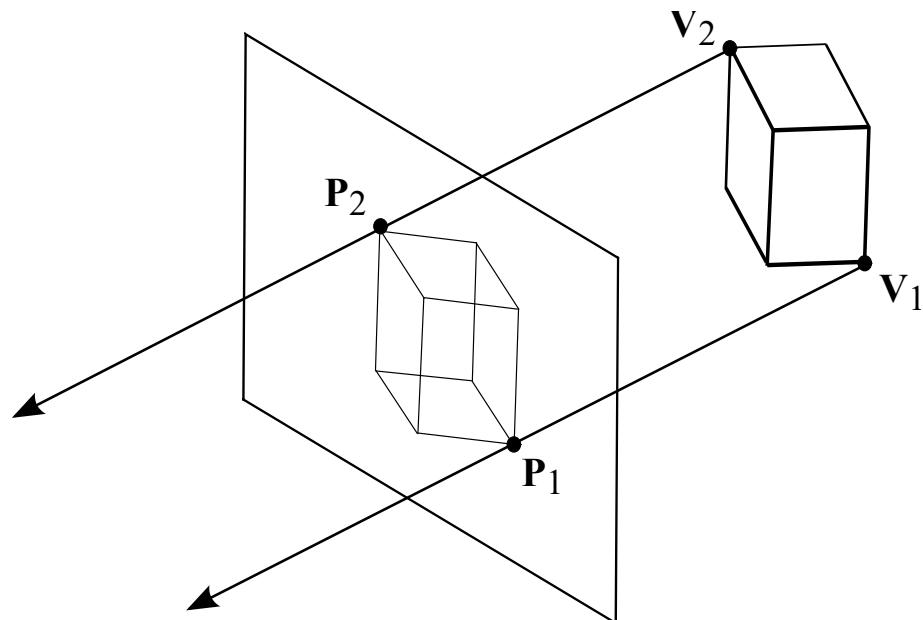
- In general it is possible to project onto any surface:
 - Sphere
 - Cone
 - Etc.
- or to use curved projectors, for example to produce lens effects.
- But we will only consider linear projections onto a flat (planar) surface.

Orthographic projection

- This is the simplest form of projection, and effective in many cases.
- Make simplifying assumptions:
 - The viewpoint is at $z = -\infty$
 - The plane of projection is $z = 0$
- So all projectors have the same direction:

$$\mathbf{d} = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$$

Orthographic projection onto $z = 0$



Each projection line has equation

$$\mathbf{P} = \mathbf{V} + \mu \mathbf{d}$$

where

$$\mathbf{d} = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$$

Calculating an orthographic projection

- Substitute $\mathbf{d} = (0, 0, -1)^T$ into the projector vector equation:

$$\mathbf{P} = \mathbf{V} + \mu\mathbf{d}$$

- Gives Cartesian equations for each component

$$P_x = V_x + 0 \quad P_y = V_y + 0 \quad P_z = V_z - \mu$$

- Projection plane is $z = 0 \Rightarrow P_z = 0$

Calculating an orthographic projection (cont.)

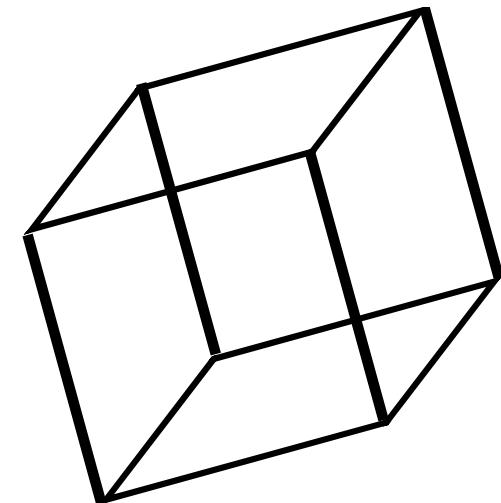
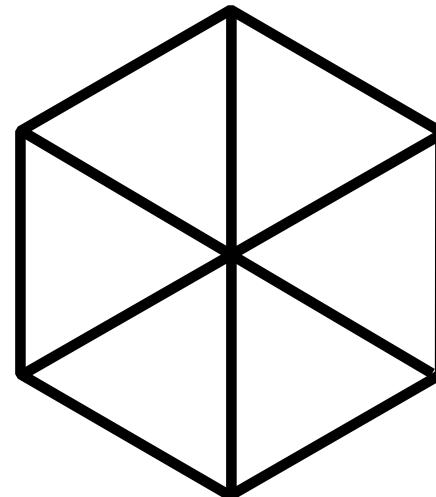
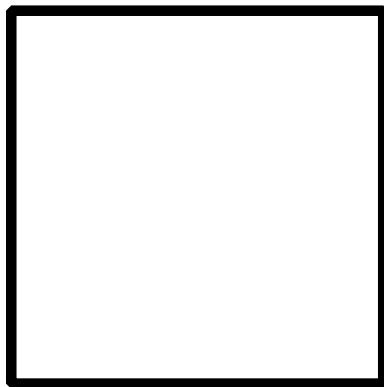
- So the projected location on the screen is

$$\mathbf{P} = \begin{pmatrix} Vx \\ Vy \\ 0 \end{pmatrix}$$

- i.e. we simply take the 3D x and y components of the vertex!

Orthographic projections of a cube

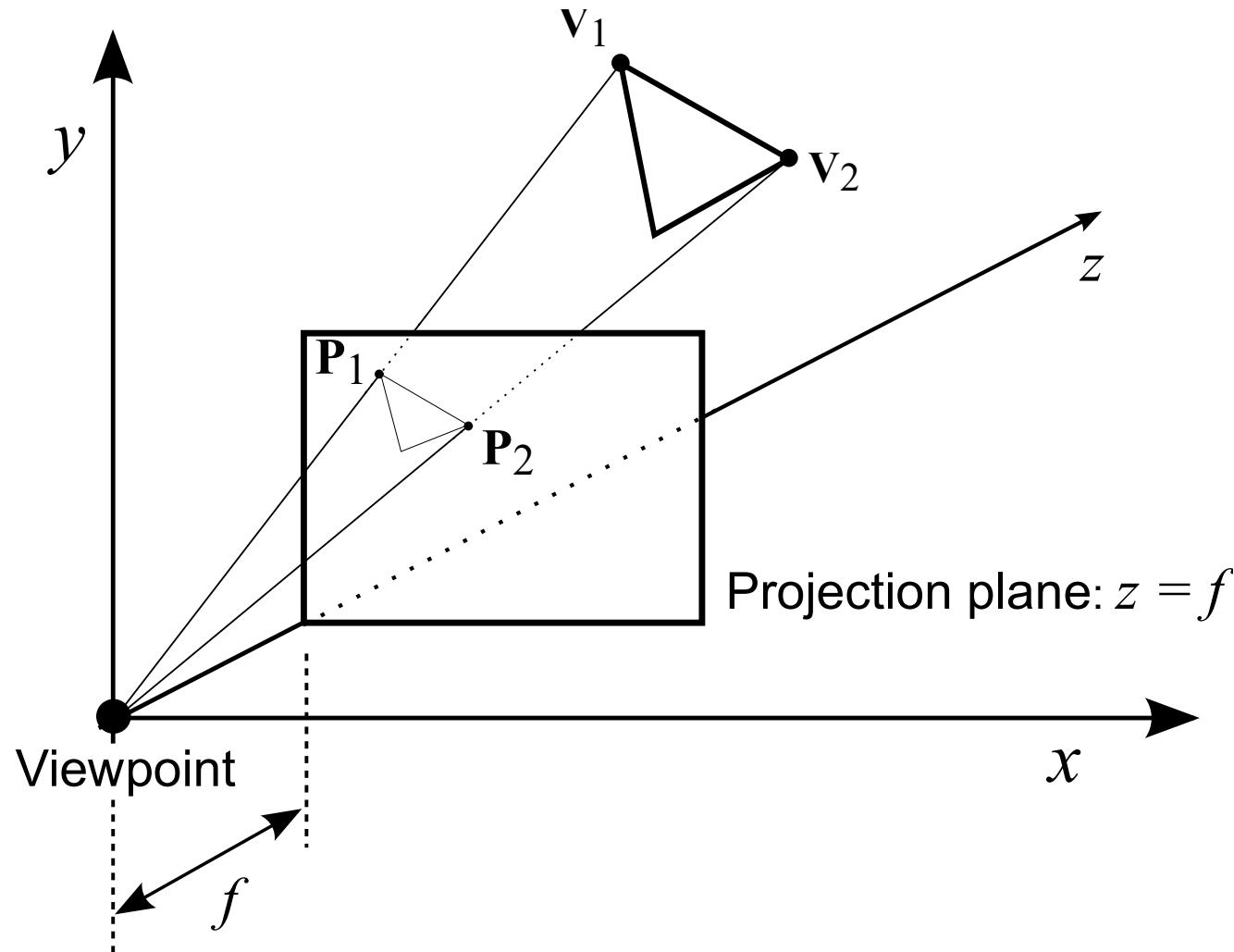
- Looking at a face, a vertex and a more general view...



Perspective projection

- Orthographic projection is fine in cases where we are not worried about depth
 - e.g. when most objects are at the same distance from the viewer
- However, for close work - particularly computer games - it will not do.
- Instead, we use *perspective projection*.

Canonical form for perspective projection



Calculating perspective projection

The perspective projector equation from vertex \mathbf{V} is

$$\mathbf{P} = \mu \mathbf{V}$$

because all projectors go through the origin. At the projected point we have $P_z = f$.

Let the value of μ at this point be μ_p

$$\mu_p = P_z/V_z = f/V_z$$

and

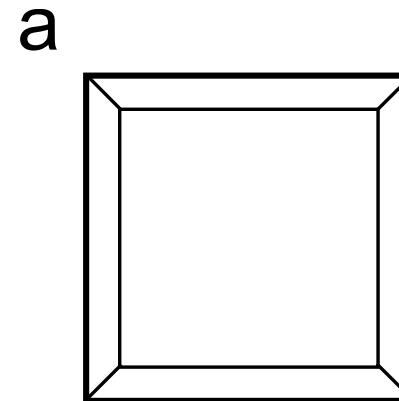
$$P_x = \mu_p V_x, \quad P_y = \mu_p V_y$$

Therefore

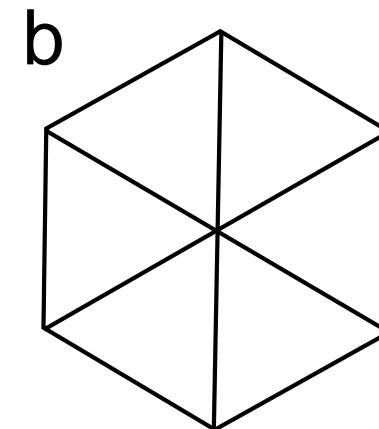
$$P_x = fV_x/V_z, \quad P_y = fV_y/V_z$$

Perspective projections of a cube

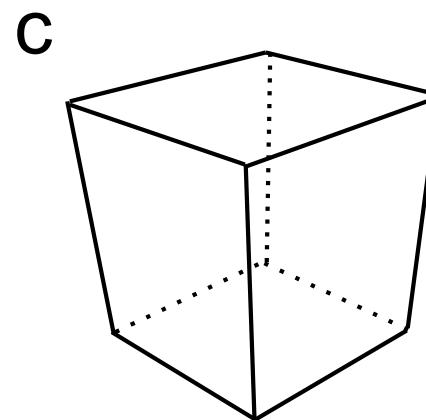
(a) Viewing a face



(b) Viewing a vertex



(c) A general view



Problem break

Given that the viewing plane is at $z = 5$, what point on the view plane corresponds to the 3D vertex

$$\mathbf{V} = \begin{pmatrix} 10 \\ 10 \\ 10 \end{pmatrix}$$

when we use the different projections:

1. Perspective
2. Orthographic

Problem break

Given that the viewing plane is at $z = 5$, what point on the view plane corresponds to the 3D vertex

$$\mathbf{V} = \begin{pmatrix} 10 \\ 10 \\ 10 \end{pmatrix}$$

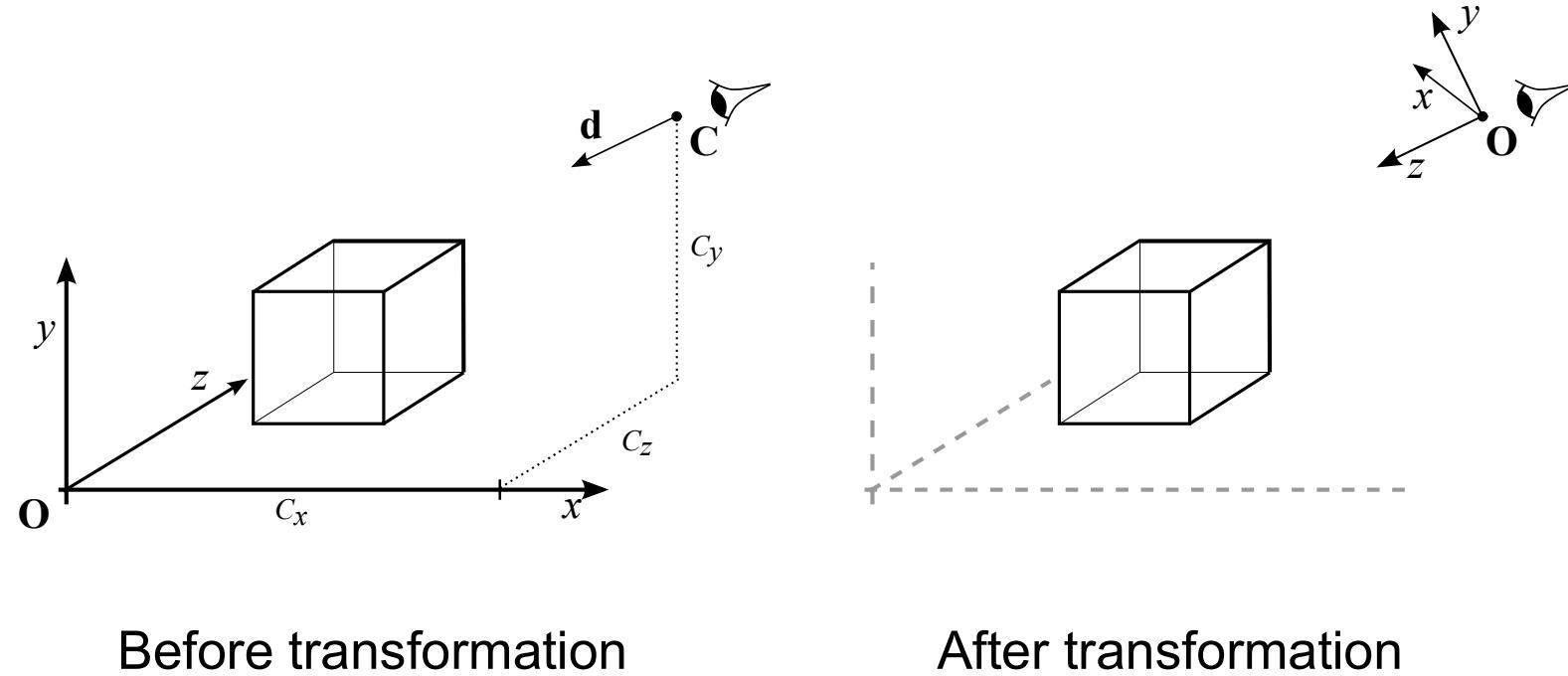
when we use the different projections:

1. Perspective $P_x = fV_x/V_z = 5$ and $P_y = fV_y/V_z = 5$
2. Orthographic $P_x = 10$ and $P_y = 10$

The need for transformations

- Graphics scenes are defined in a particular coordinate system.
- We want to draw a graphics scene from any angle
- **But** to draw a graphics scene, it is a lot easier to have:
 - The viewpoint at the origin
 - The z -zaxis as the direction of view
- Hence, we need to be able to transform the coordinates of a graphics scene.

Transformation of viewpoint



Other transformations

- We also need transformations for other purposes:
 - Animating Objects
 - e.g. flying titles, rotating, shrinking etc.
 - Multiple Instances
 - the same object may appear at different places or different sizes
 - Reflections and other special effects

Matrix transformations of points

To transform points we use matrix multiplications, e.g. to make an object at the origin twice as big we could use:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

which, when multiplied out, gives:

$$x' = 2x \quad y' = 2y \quad z' = 2z$$

Translation by matrix multiplication

- Many of our transformations will require translation of the points. For example if we want to move all the points two units along the x -axis we would require

$$x' = x + 2$$

$$y' = y$$

$$z' = z$$

- But how can we do this with a matrix? I.e.

$$\begin{pmatrix} ? \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x + 2 \\ y \\ z \end{pmatrix}$$

... can't be done

Homogenous coordinates

- The answer is to use 4D homogenous coordinates.
- They have a 4th ordinate allowing us to use the last column for translation

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

- which, when multiplied out, gives:

$$x' = x + 2 \quad y' = y \quad z' = z$$

General homogenous coordinates

- In most cases the last ordinate will be 1
- But in general, it is a scale factor.

Homogeneous Cartesian

$$(p_x, p_y, p_z, s) \iff \left(\frac{p_x}{s}, \frac{p_y}{s}, \frac{p_z}{s} \right)$$

Affine transformations

- Affine transformations are those that preserve parallel lines.
- Most transformations we require are affine, the most important being:
 - Scaling
 - Rotation
 - Translation
- Other more complex transforms can be built from these.
- An example of a non-affine transformation:
 - Perspective projection (parallels not preserved).

Translation with a matrix

- We can apply a general translation by (t_x, t_y, t_z) to the points of a scene by using the following matrix multiplication

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x + t_x \\ p_y + t_y \\ p_z + t_z \\ 1 \end{pmatrix}$$

Inverting a translation

- Since we know what a translation matrix physically does, we can write down its inversion directly, e.g.

Translation matrix inverse

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Can you show that the product of these matrices is the identity?

Scaling with a matrix

- Scaling simply multiplies each ordinate by a scaling factor.
- It can be done with the following homogenous matrix:

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} s_x p_x \\ s_y p_y \\ s_z p_z \\ 1 \end{pmatrix}$$

Inverting a scaling

- To invert a scaling we simply divide the individual ordinates by the scale factor.

Scaling matrix		inverse
$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$		$\begin{pmatrix} 1/s_x & 0 & 0 & 0 \\ 0 & 1/s_y & 0 & 0 \\ 0 & 0 & 1/s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Combining transformations

- Suppose we want to make an object centred at the origin twice as big and then move it so that the centre is at (5, 5, 20).
- The transformation is a scaling followed by a translation:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 5 \\ 0 & 0 & 1 & 20 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Combined transformations

- We can multiply out the transformation matrices
- This gives us a single matrix which we can use to apply both transformations to any point

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 & 5 \\ 0 & 2 & 0 & 5 \\ 0 & 0 & 2 & 20 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

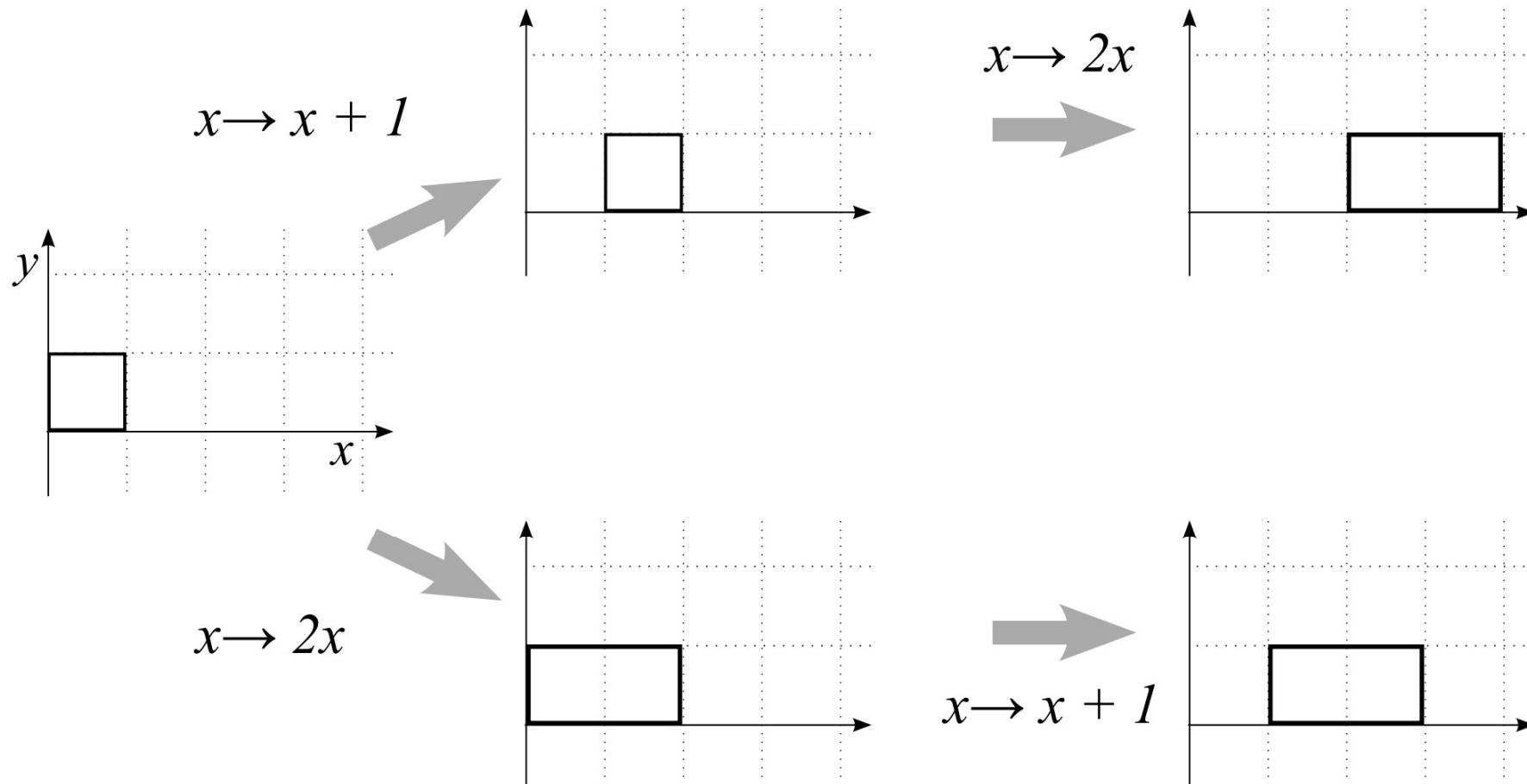
Careful: Transformations are not commutative

- The order of applying transformations matters:
- In general

$T \bullet S$ is not the same as $S \bullet T$

- *Check this for the transformation matrices on the last two slides*

The order of transformations is significant



The results at the end of each route are different.

Rotation

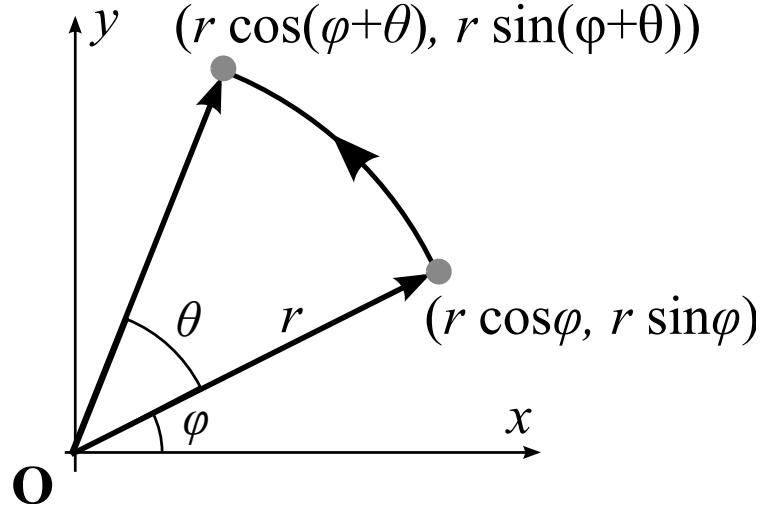
- To define a rotation, we need an axis and an angle.
- The simplest rotations are about the Cartesian axes.
- For example:
 - R_x Rotate about the x -axis
 - R_y Rotate about the y -axis
 - R_z Rotate about the z -axis

Rotation matrices

By θ about each of the axes

$$\mathcal{R}_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$$\mathcal{R}_y = \begin{pmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$$\mathcal{R}_z = \begin{pmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Example: Derivation of \mathcal{R}_z

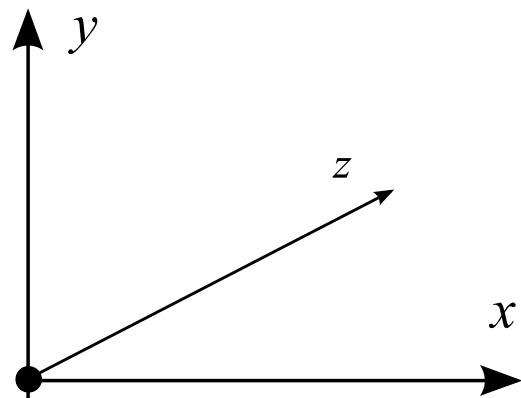


z -axis goes into page

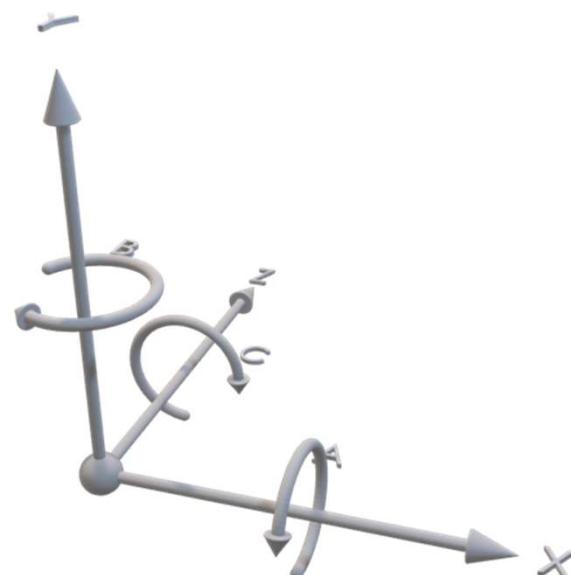
$$\begin{aligned}
 \begin{pmatrix} x \\ y \end{pmatrix} &= \begin{pmatrix} r \cos \varphi \\ r \sin \varphi \end{pmatrix} \\
 \rightarrow & \begin{pmatrix} r \cos(\varphi + \theta) \\ r \sin(\varphi + \theta) \end{pmatrix} \\
 = & \begin{pmatrix} r \cos \varphi \cos \theta - r \sin \varphi \sin \theta \\ r \cos \varphi \sin \theta + r \sin \varphi \cos \theta \end{pmatrix} \\
 = & \begin{pmatrix} x \cos \theta - y \sin \theta \\ x \sin \theta + y \cos \theta \end{pmatrix} \\
 = & \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \\
 & \quad \downarrow \quad \downarrow \\
 & \begin{pmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
 \end{aligned}$$

Rotations have a direction

- Note the following about the matrix formulations given in these notes:
 - We will stick to a left-handed coordinate system
 - Rotation is anti-clockwise when looking along the axis of rotation (in the previous slide, the z-axis goes into the page).
 - Rotation is clockwise when looking back towards the origin from the positive side of the axis



Graphics Lecture 1: Slide 64



Inverting rotation

Inverting a rotation
by angle θ



Rotating through
angle $-\theta$

- i.e. we can use the following relations to help us find the inverse of a rotation:

$$\cos(-\theta) = \cos(\theta) \quad \text{and} \quad \sin(-\theta) = -\sin(\theta)$$

Inverting rotation

- So for example:

Rotation

Inverse

$$\mathcal{R}_z(\theta)$$

$$\mathcal{R}_z(-\theta)$$

$$\begin{pmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} \cos \theta & \sin \theta & 0 & 0 \\ -\sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Interactive Computer Graphics: Lecture 2

Transformations for animation

The most useful operations: Previously defined transformation matrices

- Translation

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

- Scaling

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

Rotations about x, y and z axes.

$$\mathcal{R}_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathcal{R}_y = \begin{pmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathcal{R}_z = \begin{pmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotations about x , y and z axes.

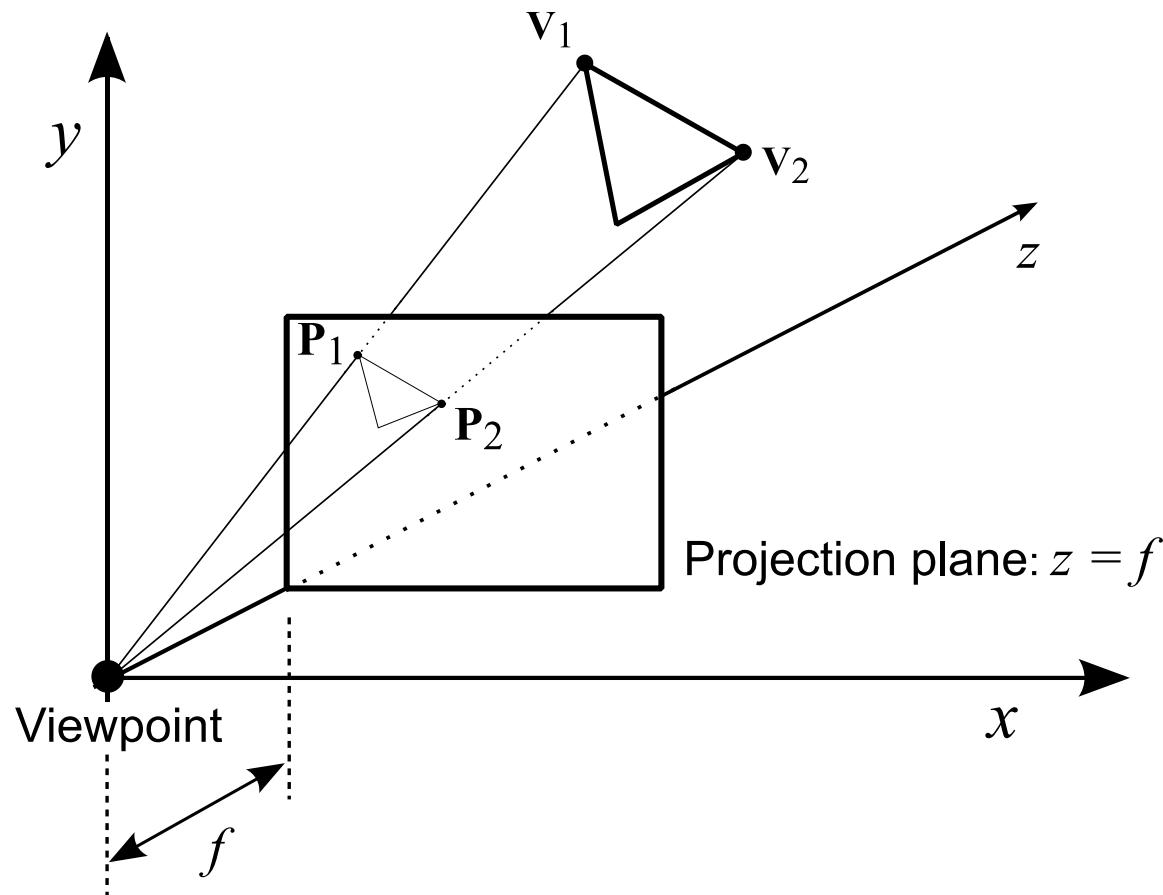
$$\mathcal{R}_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$$\mathcal{R}_y = \begin{pmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$$\mathcal{R}_z = \begin{pmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

We now consider more complex transformations which are combinations of translations, scalings and rotations

Flying sequences

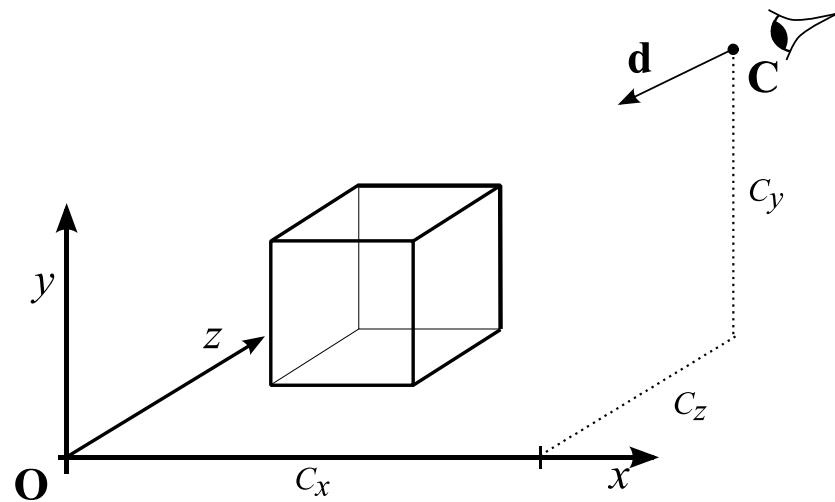
- In generating animated flying sequences, we require the viewpoint to move around the scene.
- This implies a change of origin
- Let
 - the required viewpoint be $\mathbf{C} = (C_x, C_y, C_z)$
 - the required view direction be $\mathbf{d} = \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix}$

Recall the canonical form for perspective projection

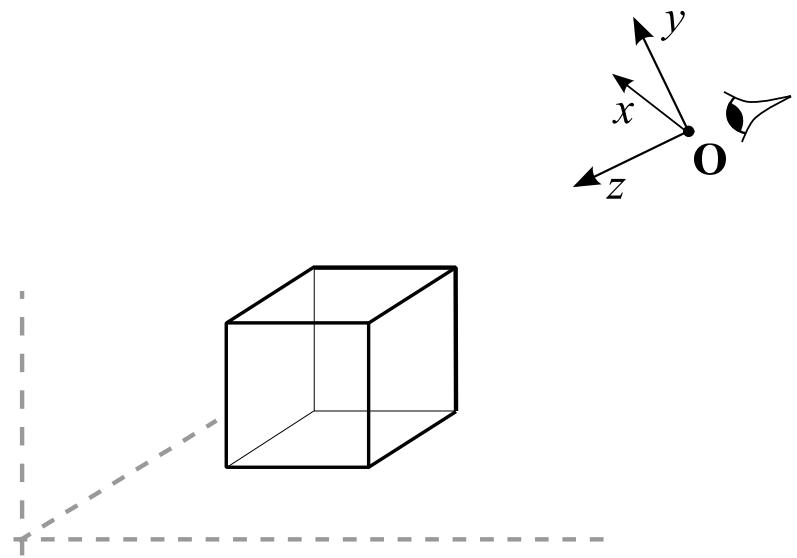


We look along the z -axis and the the y -axis is 'up'

Transformation of viewpoint



Coordinate system for definition

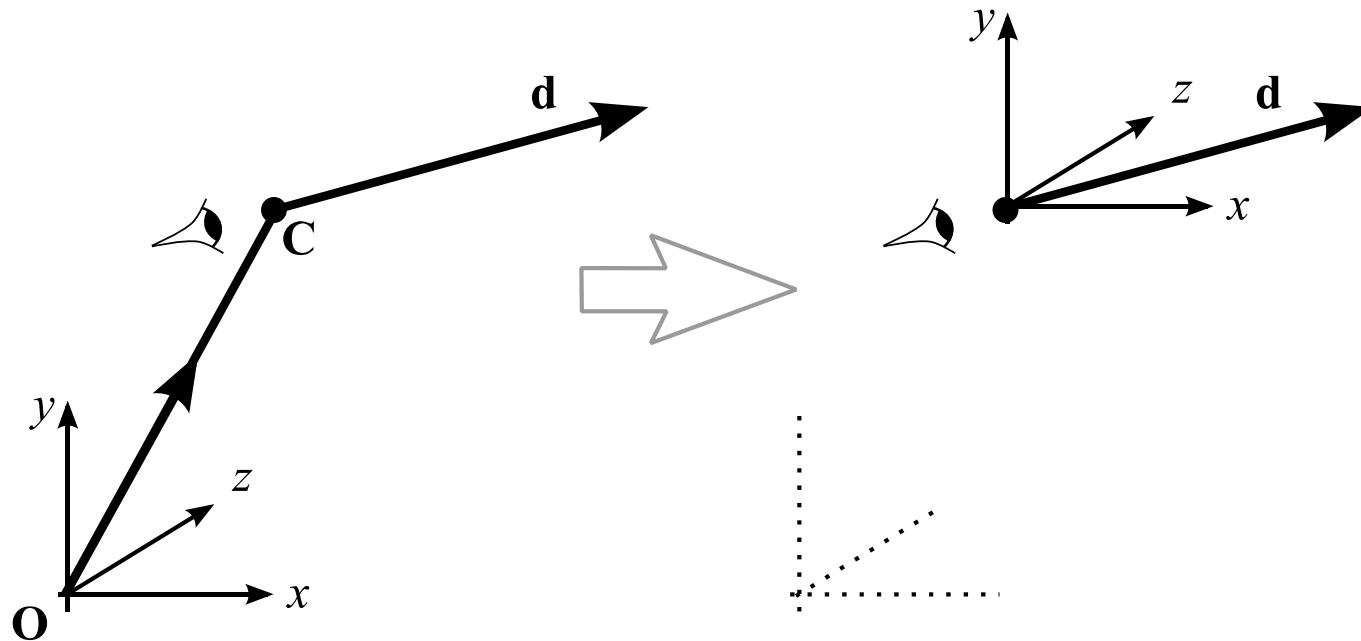


Coordinate system for viewing

Flying Sequences

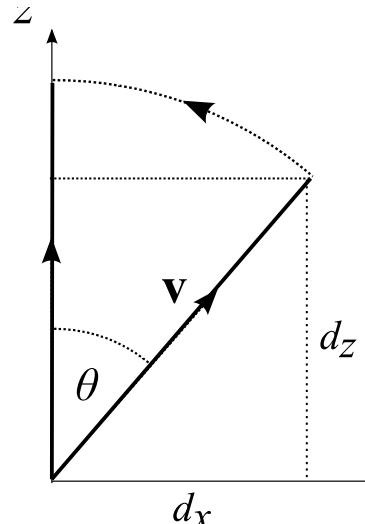
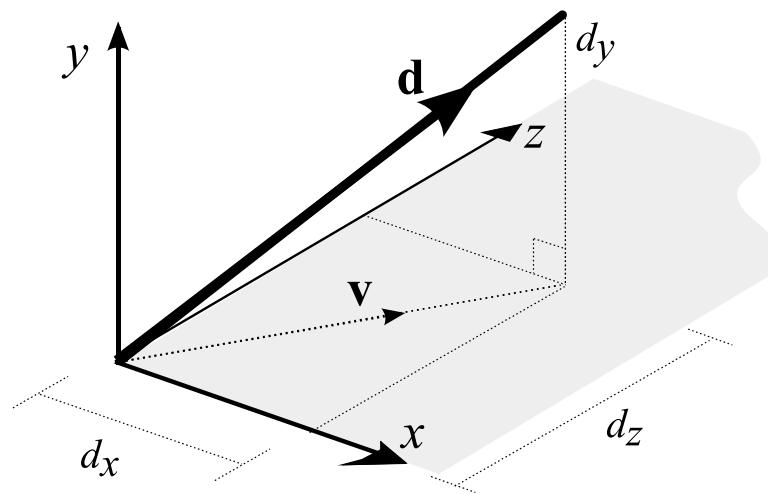
- The required transformation is in three parts:
 1. Translation of the origin
 2. Rotate about y-axis
 3. Rotate about x-axis
- The two rotations are to line up the z-axis with the view direction

1. Translation of the Origin



$$\mathcal{A} = \begin{pmatrix} 1 & 0 & 0 & -C_x \\ 0 & 1 & 0 & -C_y \\ 0 & 0 & 1 & -C_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

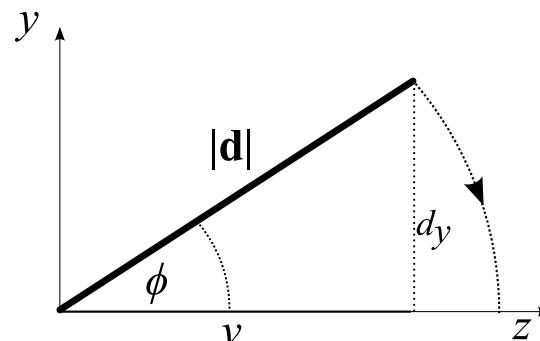
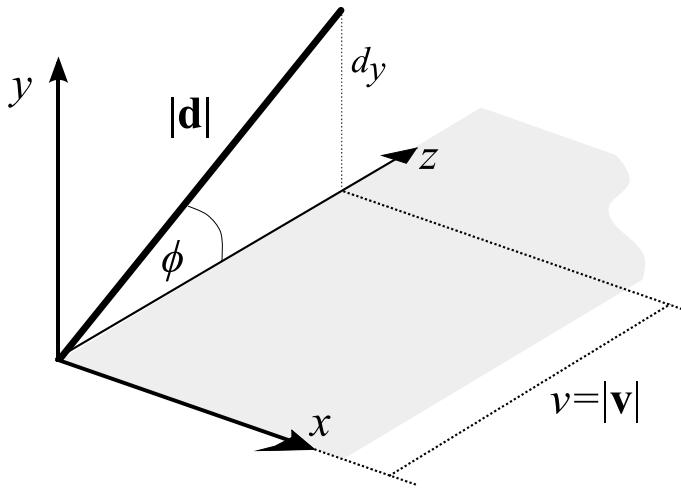
2. Rotate about y until \mathbf{d} is in the y - z plane



$$\begin{aligned}\|\mathbf{v}\| &= v & = & \sqrt{d_x^2 + d_z^2} \\ \cos \theta & = & d_z / v \\ \sin \theta & = & d_x / v\end{aligned}$$

$$\mathcal{B} = \begin{pmatrix} \cos \theta & 0 & -\sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} d_z/v & 0 & -d_x/v & 0 \\ 0 & 1 & 0 & 0 \\ d_x/v & 0 & d_z/v & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3. Rotate about x until \mathbf{d} points along the z -axis



$$\begin{aligned} v &= \sqrt{d_x^2 + d_z^2} \\ \cos \phi &= v/|\mathbf{d}| \\ \sin \phi &= d_y/|\mathbf{d}| \end{aligned}$$

$$C = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \phi & -\sin \phi & 0 \\ 0 & \sin \phi & \cos \phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & v/|\mathbf{d}| & -d_y/|\mathbf{d}| & 0 \\ 0 & d_y/|\mathbf{d}| & v/|\mathbf{d}| & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Combining the matrices

- A single matrix that transforms the scene can be obtained from the matrices \mathcal{A} , \mathcal{B} and \mathcal{C} by multiplication

$$\mathcal{T} = \mathcal{C}\mathcal{B}\mathcal{A}$$

- And for every point \mathbf{P} of the scene, we calculate

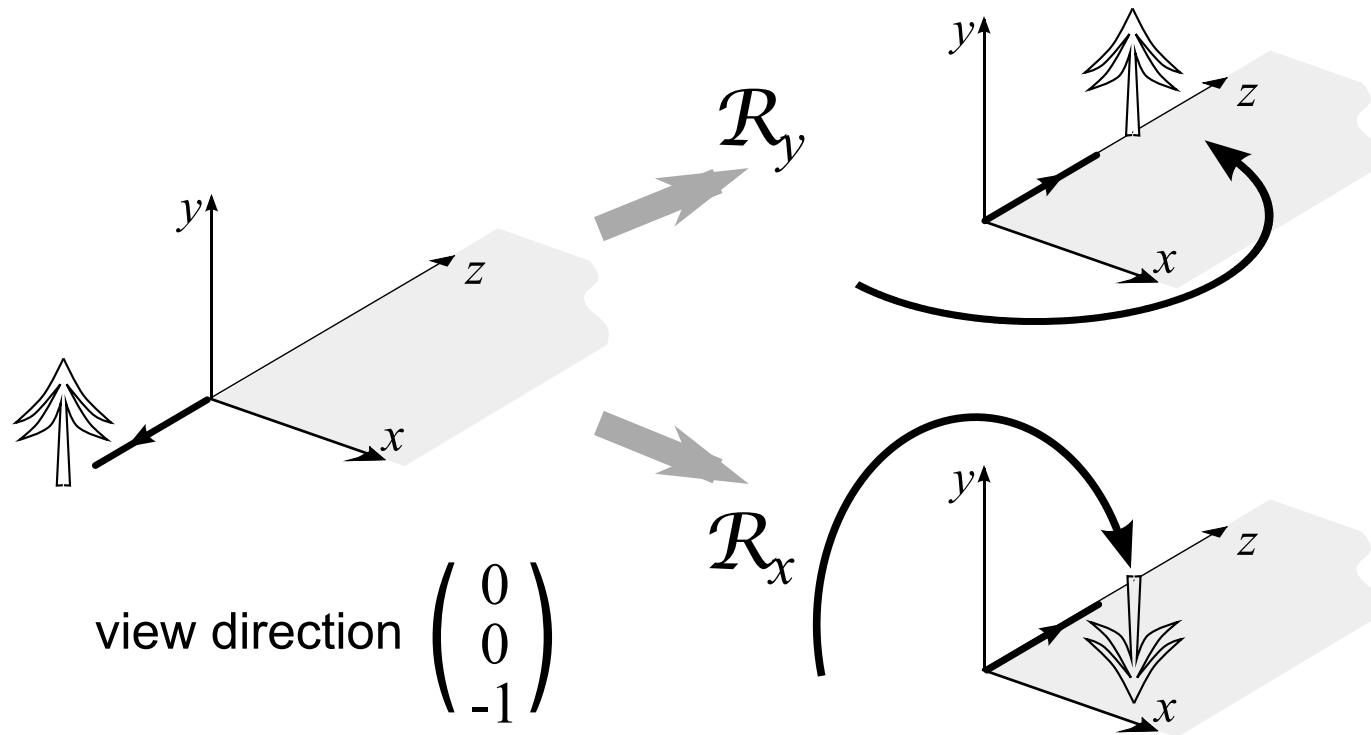
$$\mathbf{P}_t = \mathcal{T}\mathbf{P}$$

- The view is now in ‘canonical’ form and we can apply the standard perspective or orthographic projection.

Verticals

- So far we have not looked at verticals
- Usually, the y direction is treated as vertical, and by doing the R_y transformation first, things work out correctly
- However it is possible to invert the vertical

Transformations and verticals



Rotation about a general line

- Special effects, such as rotating a scene about a general line can be achieved by multiple transformations
- The transformation is formed by:
 - Making the line of rotation one of the Cartesian axes
 - Doing the rotation (about the chosen axis)
 - Restoring the line to its original place

Rotation about a general line

- The first part is achieved using the same matrices that we derived for the flying sequences

$$CBA$$

- This rotates the general line so it is aligned with the z -axis.
- We then carry out the rotation about the z -axis then follow this by the inversion of the initial matrices.
- So the full matrix T of the combined transformation is

$$T = A^{-1}B^{-1}C^{-1}R_zCBA$$

Other effects

- Similar effects can be created using this approach
- e.g. to make an object shrink (and stay in place)
 1. Move the object to the origin
 2. Apply a scaling matrix
 3. Move the object back to where it was

Projection by matrix multiplication

- Usually projection and drawing of a scene comes after the transformation(s)
- It is therefore convenient to combine the projection with the other parts of the transformation
- So it is useful to have matrices for the projection operation

Orthographic projection matrix

- For (canonical) orthographic projection, we simply drop the z -coordinate:

$$\mathcal{M}_o = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathcal{M}_o \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 0 \\ 1 \end{pmatrix}$$

Perspective projection matrix

- Perspective projection of homogenous coordinates can also be done by matrix multiplication:

$$\mathcal{M}_p = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{pmatrix}$$

$$\mathcal{M}_p \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ z/f \end{pmatrix}$$

Perspective projection matrix: Normalisation

- Remember we can normalise homogeneous coordinates, so

$$\mathcal{M}_p \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ z/f \end{pmatrix} \text{ which is the same as } \begin{pmatrix} xf/z \\ yf/z \\ f \\ 1 \end{pmatrix}$$

- as required.

Projection matrices are singular

- Notice that both projection matrices are singular (i.e. ‘non-invertible’, zero-determinant, ...)

$$\mathcal{M}_p = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{pmatrix} \quad \mathcal{M}_o = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- This is because a projection transformation cannot be inverted.
- Given a 2D image, we cannot in general reconstruct the original 3D scene.

Homogenous coordinates as vectors

- We now take a second look at homogeneous coordinates, and their relation to vectors.
- In the previous lecture we described the fourth ordinate as a scale factor.

Homogeneous Cartesian

$$\begin{pmatrix} x \\ y \\ z \\ s \end{pmatrix} \rightarrow \begin{pmatrix} x/s \\ y/s \\ z/s \end{pmatrix}$$

Homogenous coordinates and vectors

- Homogenous coordinates fall into two types:

1. Position vectors

- Those with non-zero final ordinate ($s > 0$).
- Can be normalised into Cartesian form.

$$\begin{pmatrix} x \\ y \\ z \\ s \end{pmatrix}$$

2. Direction vectors

- Those with zero in the final ordinate.
- Have direction and magnitude.

$$\begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix}$$

Adding direction vectors

- If we add two direction vectors we obtain a direction vector

$$\begin{pmatrix} x_i \\ y_i \\ z_i \\ 0 \end{pmatrix} + \begin{pmatrix} x_j \\ y_j \\ z_j \\ 0 \end{pmatrix} = \begin{pmatrix} x_i + x_j \\ y_i + y_j \\ z_i + z_j \\ 0 \end{pmatrix}$$

- This is the normal vector addition rule.

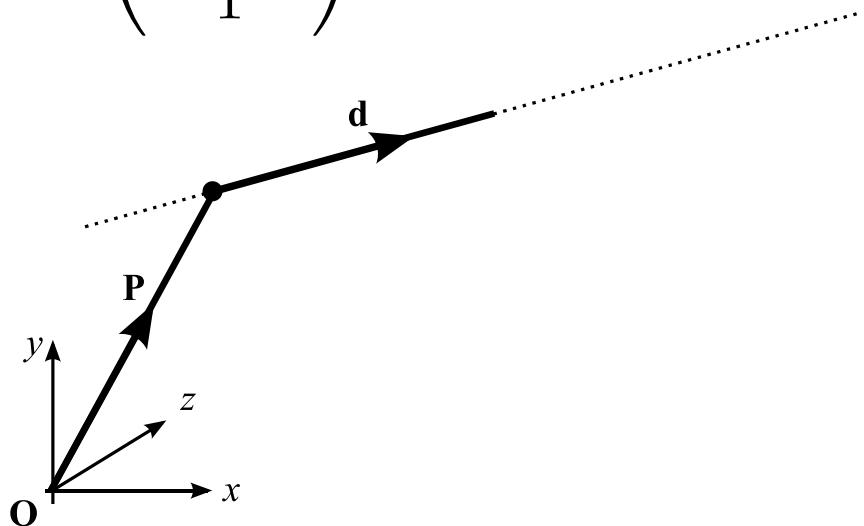
Adding position and direction vectors

- If we add a direction vector to a position vector, we obtain a position vector:

$$\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} + \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} = \begin{pmatrix} X + x \\ Y + y \\ Z + z \\ 1 \end{pmatrix}$$

Nice result.

Ties in with definition of straight line in Cartesian space which uses a point and a direction



Adding two position vectors

- If we add two position vectors, we obtain their mid-point

$$\begin{pmatrix} X_a \\ Y_a \\ Z_a \\ 1 \end{pmatrix} + \begin{pmatrix} X_b \\ Y_b \\ Z_b \\ 1 \end{pmatrix} = \begin{pmatrix} X_a + X_b \\ Y_a + Y_b \\ Z_a + Z_b \\ 2 \end{pmatrix} = \begin{pmatrix} (X_a + X_b) / 2 \\ (Y_a + Y_b) / 2 \\ (Z_a + Z_b) / 2 \\ 1 \end{pmatrix}$$

- This is reasonable since adding two position vectors has no real meaning in vector geometry

The structure of a transformation matrix

- The bottom row is always 0 0 0 1
- The columns of a transformation matrix comprise three direction vectors and one position vector

Matrix	Direction vectors	Position vectors
$\begin{pmatrix} q_x & r_x & s_x & C_x \\ q_y & r_y & s_y & C_y \\ q_z & r_z & s_z & C_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} q_x \\ q_y \\ q_z \\ 0 \end{pmatrix} \quad \begin{pmatrix} r_x \\ r_y \\ r_z \\ 0 \end{pmatrix} \quad \begin{pmatrix} s_x \\ s_y \\ s_z \\ 0 \end{pmatrix}$	$\begin{pmatrix} C_x \\ C_y \\ C_z \\ 1 \end{pmatrix}$

Characteristics of transformation matrices

- Direction vector: Zero, in the last ordinate \Rightarrow not affected by the translation.

$$\begin{pmatrix} q_x & r_x & s_x & C_x \\ q_y & r_y & s_y & C_y \\ q_z & r_z & s_z & C_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} * \\ * \\ * \\ 0 \end{pmatrix} = \begin{pmatrix} * \\ * \\ * \\ 0 \end{pmatrix}$$

- Position vector: 1 in the last ordinate \Rightarrow all vectors will have the same displacement.

$$\begin{pmatrix} q_x & r_x & s_x & C_x \\ q_y & r_y & s_y & C_y \\ q_z & r_z & s_z & C_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} * \\ * \\ * \\ 1 \end{pmatrix} = \begin{pmatrix} * + C_x \\ * + C_y \\ * + C_z \\ 1 \end{pmatrix}$$

- If we do not shear the object the three vectors \mathbf{q} , \mathbf{r} and \mathbf{s} will remain orthogonal, ie:

$$\mathbf{q} \cdot \mathbf{r} = \mathbf{r} \cdot \mathbf{s} = \mathbf{q} \cdot \mathbf{s} = \mathbf{0}$$

What do the individual columns mean?

- To see this, consider the effect of the transformation in simple cases.
- For example take the unit direction vectors along the Cartesian axes
 - e.g. along the x -axis, $\mathbf{i} = (1, 0, 0, 0)^T$

$$\begin{pmatrix} q_x & r_x & s_x & C_x \\ q_y & r_y & s_y & C_y \\ q_z & r_z & s_z & C_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} q_x \\ q_y \\ q_z \\ 0 \end{pmatrix}$$

What do the individual columns mean?

- The other axis transformations:

Similarly, we find the following transformations of unit vectors \mathbf{j} and \mathbf{k}

$$\mathbf{j} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} r_x \\ r_y \\ r_z \\ 0 \end{pmatrix} \quad \mathbf{k} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} s_x \\ s_y \\ s_z \\ 0 \end{pmatrix}$$

What do the individual columns mean?

- Transforming the origin:
 - If we transform the origin, $(0, 0, 0, 1)^T$, we end up with the last column of the transformation matrix

$$\begin{pmatrix} q_x & r_x & s_x & C_x \\ q_y & r_y & s_y & C_y \\ q_z & r_z & s_z & C_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} C_x \\ C_y \\ C_z \\ 1 \end{pmatrix}$$

The meaning of a transformation matrix

Putting everything together ...

The columns are the original axis system after transforming to the new coordinate system

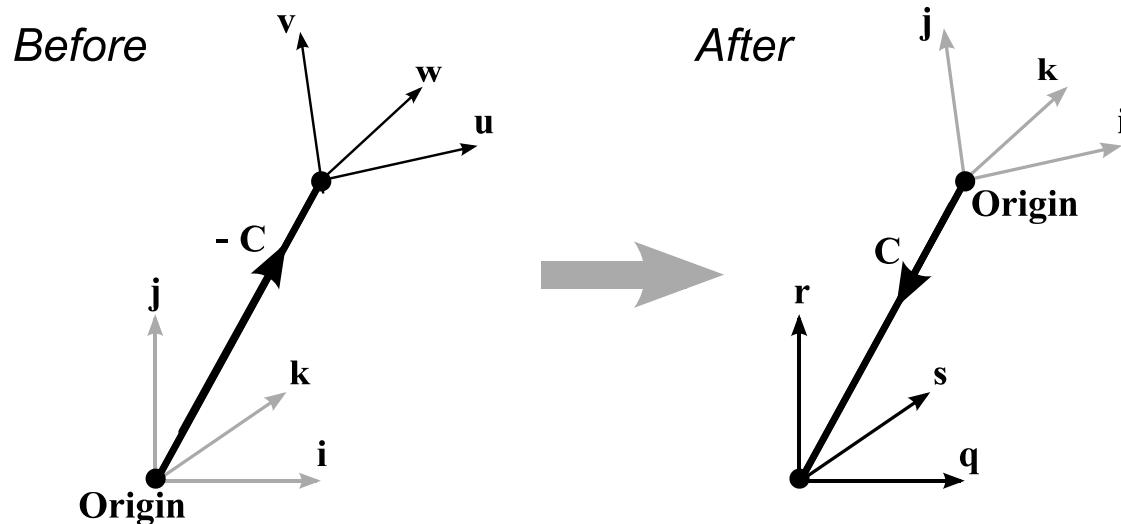
$$\begin{pmatrix} q_x & r_x & s_x & C_x \\ q_y & r_y & s_y & C_y \\ q_z & r_z & s_z & C_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

↓ ↓ ↓ ↓

q **r** **s** **C**

- q** transformed x -axis
- r** transformed y -axis
- s** transformed z -axis
- C** transformed origin

Effect of a transformation matrix

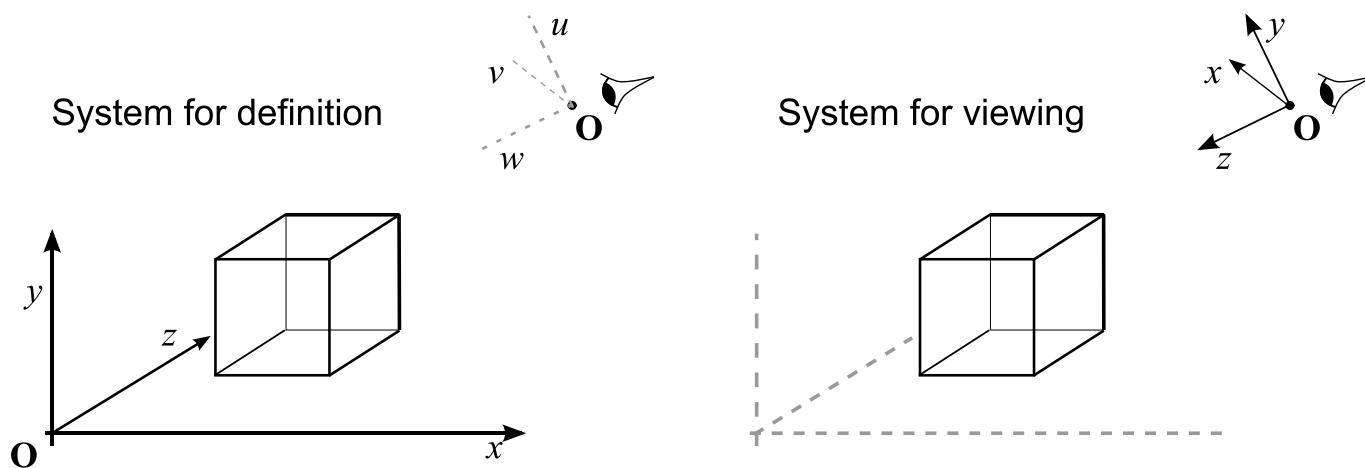


Tells us the old axes and origin in the new coordinate system.

$$\begin{pmatrix} q_x & r_x & s_x & C_x \\ q_y & r_y & s_y & C_y \\ q_z & r_z & s_z & C_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = [\mathbf{q} \quad \mathbf{r} \quad \mathbf{s} \quad \mathbf{C}]$$

What we want is the other way round

- Normally,
 - We are not given the transformation matrix that moves the scene to that coordinate system, we need to find it
 - We are given a view direction \mathbf{d} and location \mathbf{C}



To see how to get the matrix, we introduce the idea of the dot product as a *projection*

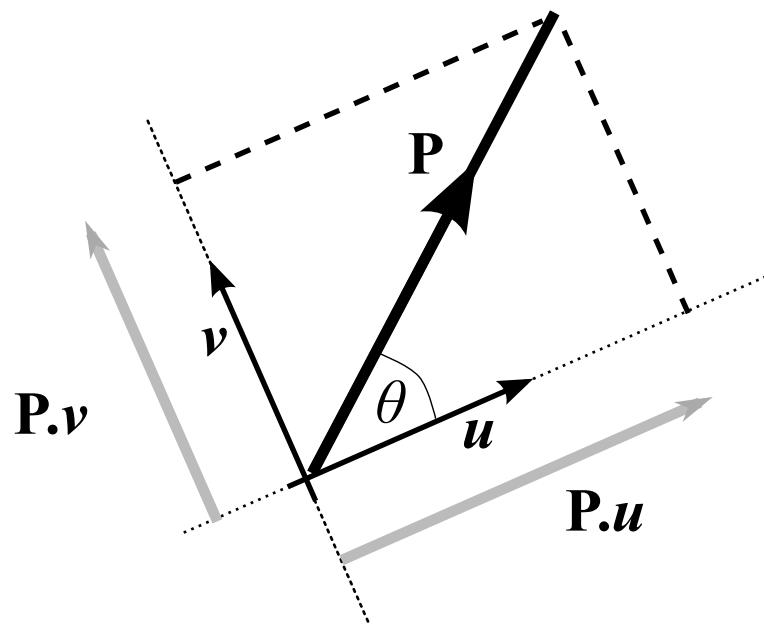
The dot product as a projection

- The dot product is defined as

$$\mathbf{P} \cdot \mathbf{u} = |\mathbf{P}| |\mathbf{u}| \cos \theta$$

- If \mathbf{u} is

- a unit vector then $\mathbf{P} \cdot \mathbf{u} = |\mathbf{P}| \cos \theta$
- along a co-ordinate axis then $\mathbf{P} \cdot \mathbf{u}$ is the ordinate of \mathbf{P} in the direction of \mathbf{u}

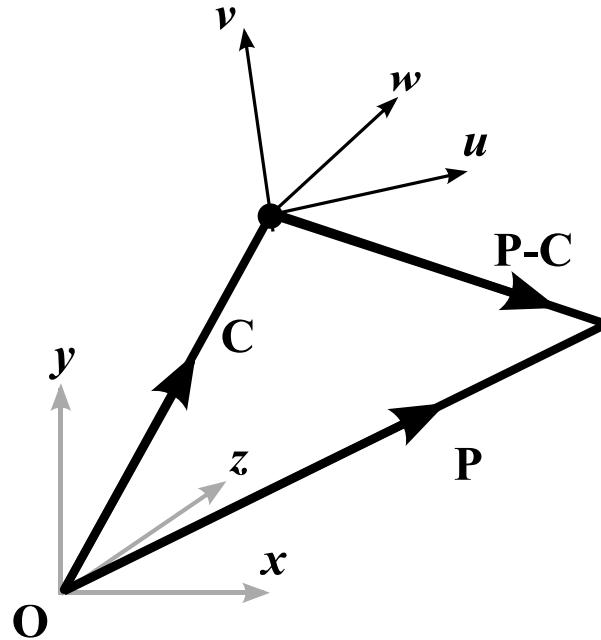


Changing axes by projection

- Extending the idea to three dimensions we can see that a change of axes can be expressed as projections using the dot product

For example, call the first coordinate of \mathbf{P} in the new system \mathbf{P}_x^t

$$\begin{aligned}\mathbf{P}_x^t &= (\mathbf{P} - \mathbf{C}) \cdot \mathbf{u} \\ &= \mathbf{P} \cdot \mathbf{u} - \mathbf{C} \cdot \mathbf{u}\end{aligned}$$



Transforming point \mathbf{P}

- Given point \mathbf{P} in the (x, y, z) axis system, we can calculate the corresponding point in the (u, v, w) system as:

$$P_x^t = (\mathbf{P} - \mathbf{C}) \cdot \mathbf{u} = \mathbf{P} \cdot \mathbf{u} - \mathbf{C} \cdot \mathbf{u}$$

$$P_y^t = (\mathbf{P} - \mathbf{C}) \cdot \mathbf{v} = \mathbf{P} \cdot \mathbf{v} - \mathbf{C} \cdot \mathbf{v}$$

$$P_z^t = (\mathbf{P} - \mathbf{C}) \cdot \mathbf{w} = \mathbf{P} \cdot \mathbf{w} - \mathbf{C} \cdot \mathbf{w}$$

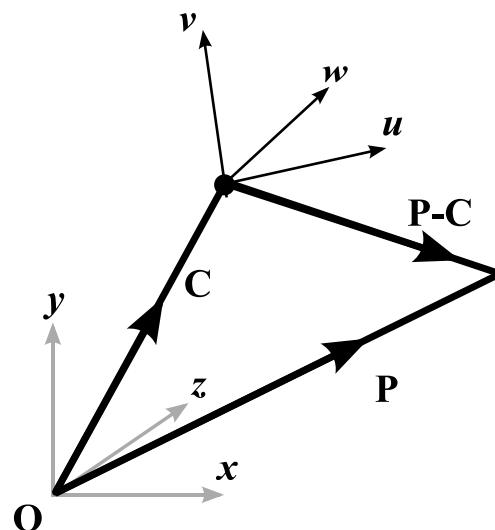
- Or, in matrix notation:

$$\begin{pmatrix} P_x^t \\ P_y^t \\ P_z^t \\ 1 \end{pmatrix} = \begin{pmatrix} u_x & u_y & u_z & -\mathbf{C} \cdot \mathbf{u} \\ v_x & v_y & v_z & -\mathbf{C} \cdot \mathbf{v} \\ w_x & w_y & w_z & -\mathbf{C} \cdot \mathbf{w} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Verticals revisited ...

Unlike the previous analysis we now can control the vertical

i.e. we can assume the v -direction is the vertical and constrain it in the software to be upwards

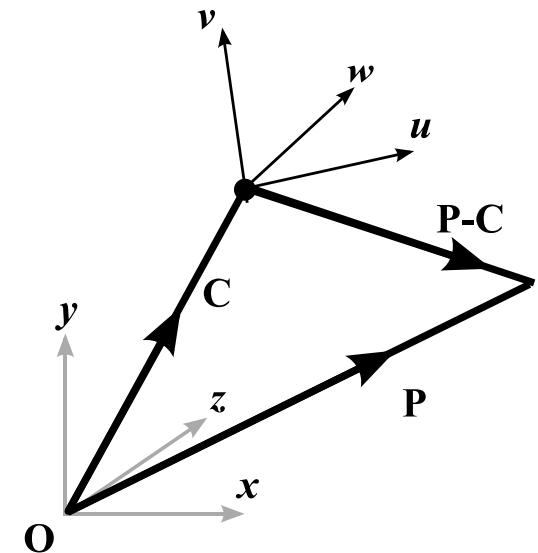


Back to flying sequences

- We now return to the original problem
 - Given a viewpoint point C and a view direction \mathbf{d} , we need to find the transformation matrix that gives us the canonical view.
 - We do this by first finding the vectors u , v and w .

We know that \mathbf{d} is the direction of the new axis, so we can write immediately

$$w = \frac{\mathbf{d}}{|\mathbf{d}|}$$



Now the horizontal direction

- We can write u in terms of some vector p in the horizontal direction

$$u = \frac{p}{|p|}$$

- To ensure that p is horizontal we set

$$p_y = 0$$

- so that p has no vertical component

And the vertical direction

- Let \mathbf{q} be some vector in the vertical direction, we can then write \mathbf{v} as

$$\Rightarrow \mathbf{v} = \frac{\mathbf{q}}{|\mathbf{q}|}$$

- \mathbf{q} must have a positive y component, so we can say that

$$q_y = 1$$

So we have four unknowns

$$\begin{aligned}\mathbf{p} &= [p_x, 0, p_z] \quad \text{new horizontal} \\ \mathbf{q} &= [q_x, 1, q_z] \quad \text{new vertical}\end{aligned}$$

To solve for these we use the cross product and dot product.

We can write the view direction \mathbf{d} , which is along the new z axis, as

$$\mathbf{d} = \mathbf{p} \times \mathbf{q}$$

(We can do this because the magnitude of \mathbf{p} is not yet set)

Evaluating the cross-product

$$\begin{aligned}\mathbf{d} &= \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix} = \mathbf{p} \times \mathbf{q} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ p_x & 0 & p_z \\ q_x & 1 & q_z \end{vmatrix} \\ &= -p_z \mathbf{i} + (p_z q_x - p_x q_z) \mathbf{j} + p_x \mathbf{k} = \begin{pmatrix} -p_z \\ p_z q_x - p_x q_z \\ p_x \end{pmatrix} \\ d_x &= -p_z \\ d_y &= p_z q_x - p_x q_z \\ d_z &= p_x\end{aligned}$$

So we can write vector \mathbf{p} completely in terms of \mathbf{d}

$$\mathbf{p} = \begin{pmatrix} d_z \\ 0 \\ -d_x \end{pmatrix}$$

Using the dot product

- Lastly we can use the fact that the vectors \mathbf{p} and \mathbf{q} are orthogonal

$$\begin{aligned}\mathbf{p} \cdot \mathbf{q} &= 0 \\ \Rightarrow p_x q_x + p_z q_z &= 0\end{aligned}$$

- And from the cross product (previous slide)

$$d_y = p_z q_x - p_x q_z$$

- So we have two simple linear equations to solve for \mathbf{q} and write it in terms of the components of \mathbf{d}

The final matrix

- Once we have expressions for \mathbf{p} and \mathbf{q} in terms of the given vector \mathbf{d} , we have

$$\mathbf{u} = \frac{\mathbf{p}}{|\mathbf{p}|} \quad \mathbf{v} = \frac{\mathbf{q}}{|\mathbf{q}|} \quad \mathbf{w} = \frac{\mathbf{d}}{|\mathbf{d}|}$$

- We already know \mathbf{C} as that is also given. So we can write down the matrix

$$\begin{pmatrix} u_x & u_y & u_z & -\mathbf{C} \cdot \mathbf{u} \\ v_x & v_y & v_z & -\mathbf{C} \cdot \mathbf{v} \\ w_x & w_y & w_z & -\mathbf{C} \cdot \mathbf{w} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

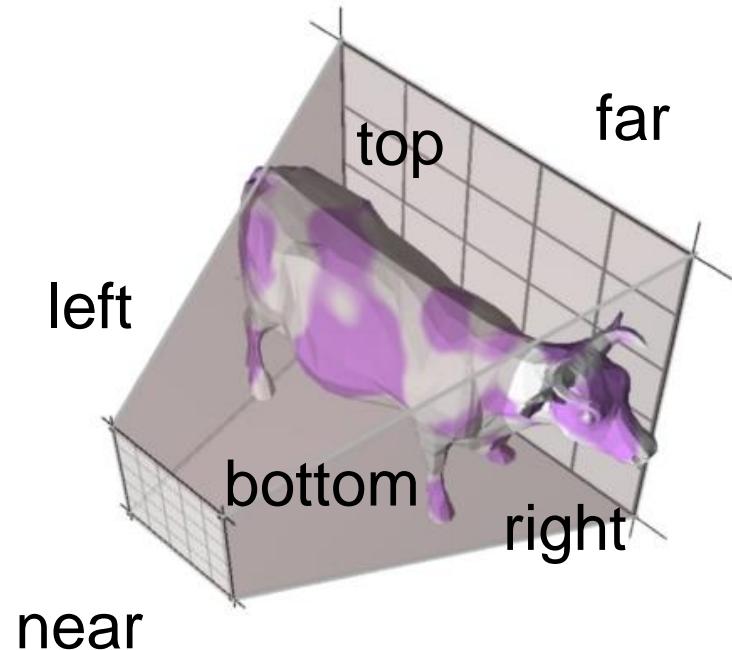
Interactive Computer Graphics: Lecture 3

Clipping

Some slides adopted from
F. Durand and B. Cutler, MIT

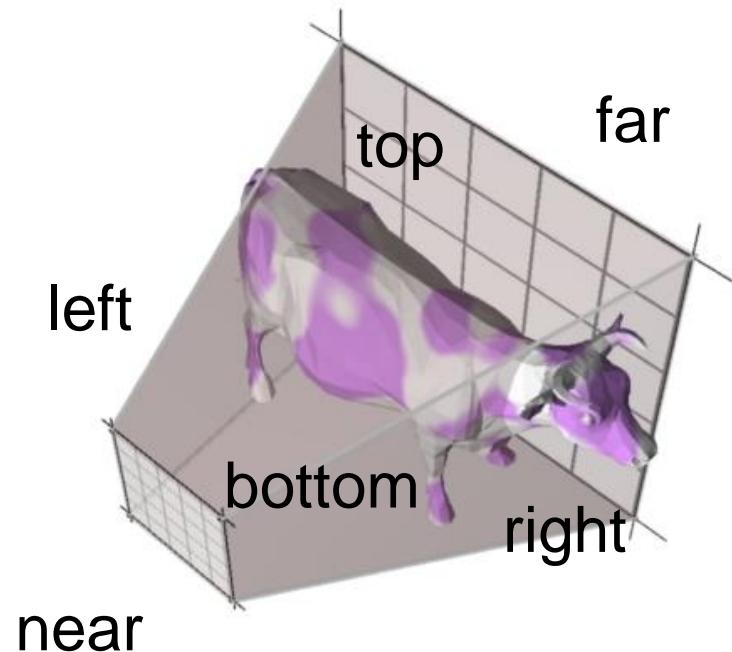
Clipping

- 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



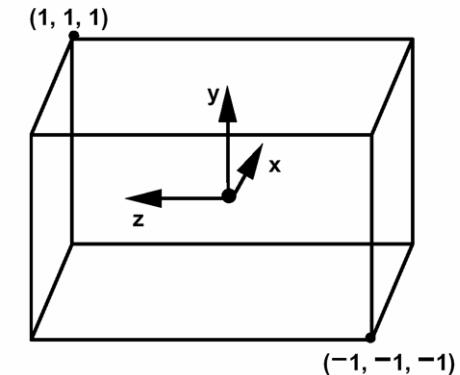
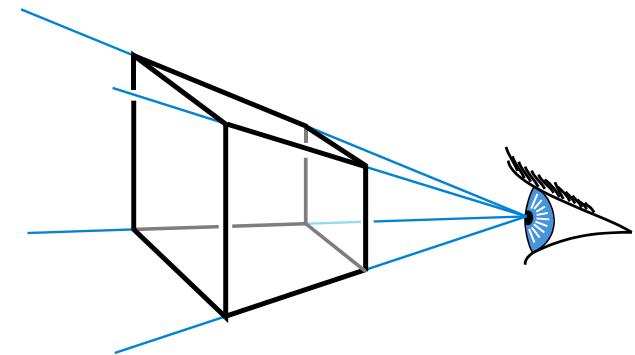
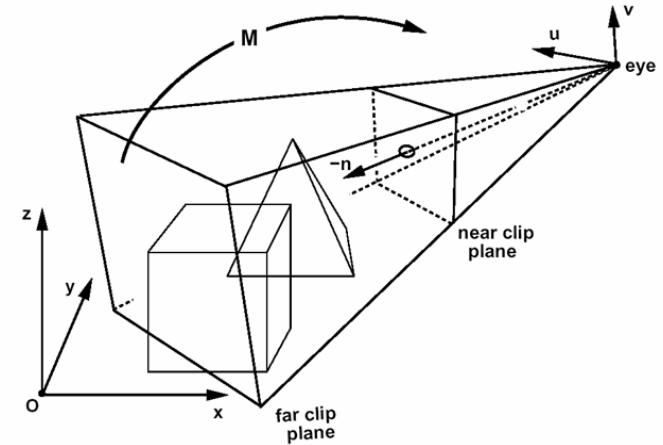
Why clipping ?

- Avoid degeneracy
 - e.g. don't draw objects behind the camera
- Improve efficiency
 - e.g. do not process objects which are not visible

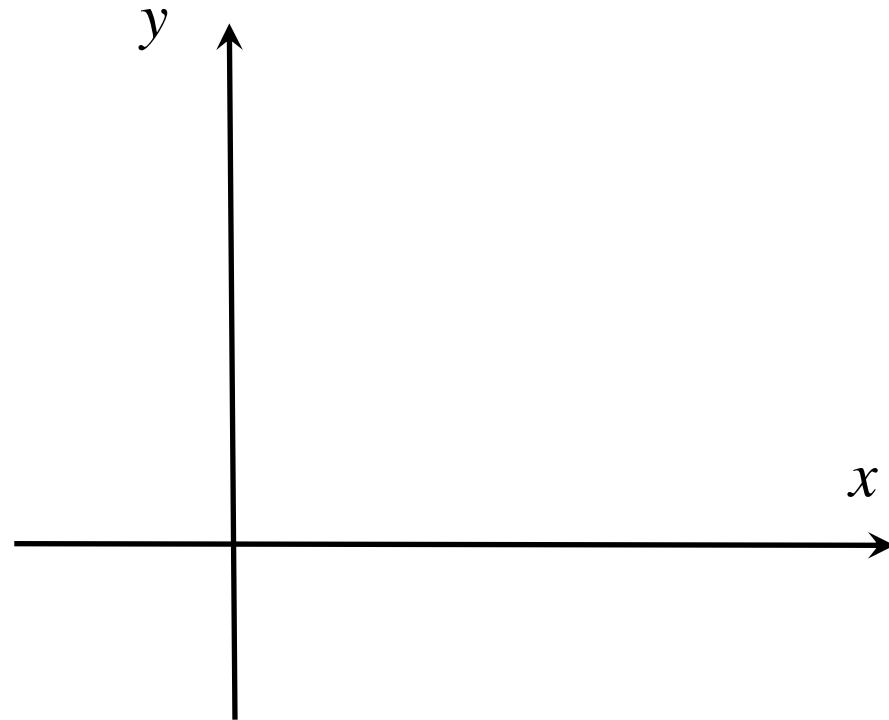


When to clip?

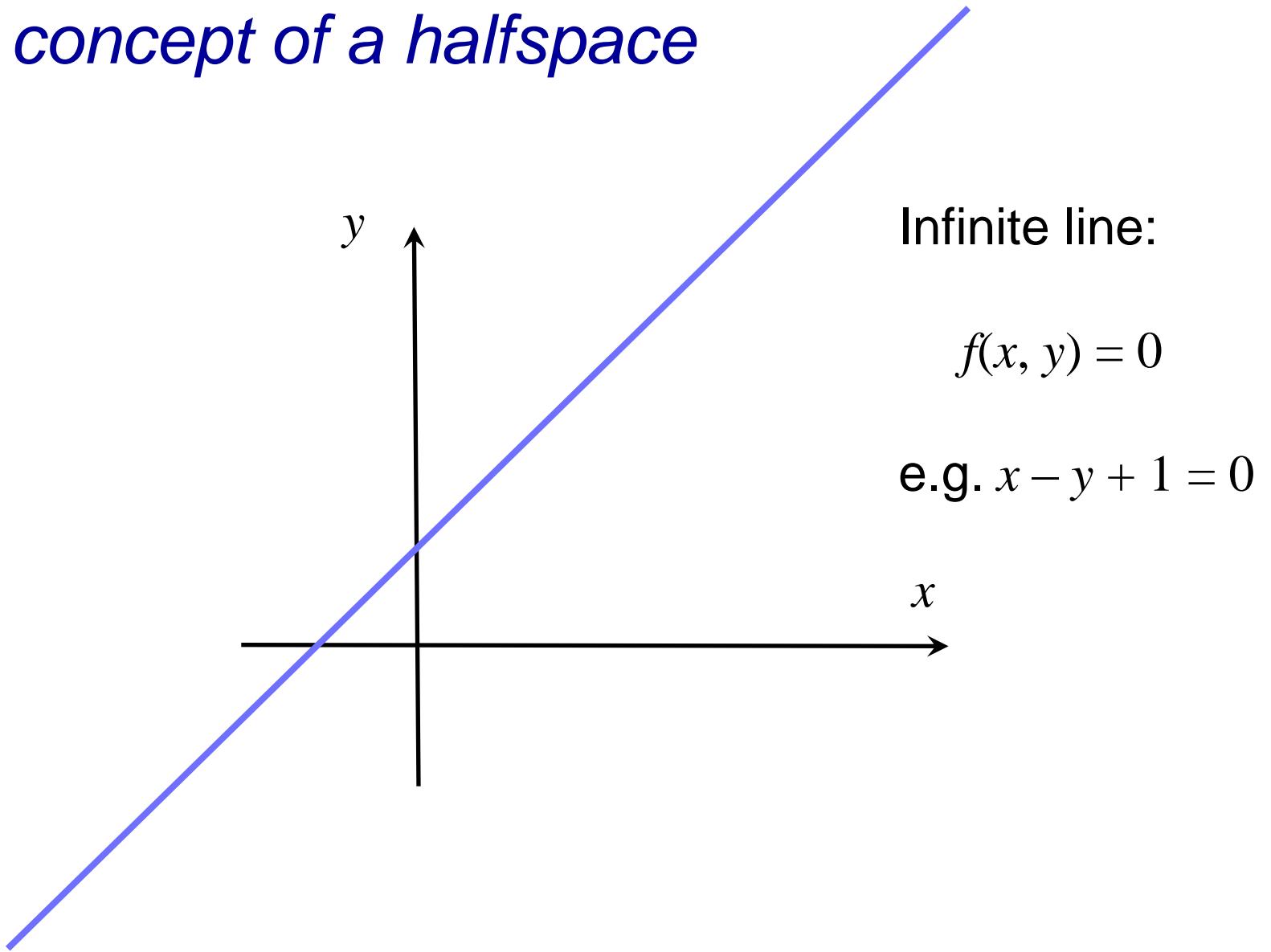
- Before perspective transform in 3D space
 - use the equation of 6 planes
 - natural, not too degenerate
- In homogeneous coordinates after perspective transform (clip space)
 - before perspective divide (4D space, weird w values)
 - canonical, independent of camera
 - simplest to implement
- In the transformed 3D screen space after perspective division
 - problem: objects in the plane of the camera



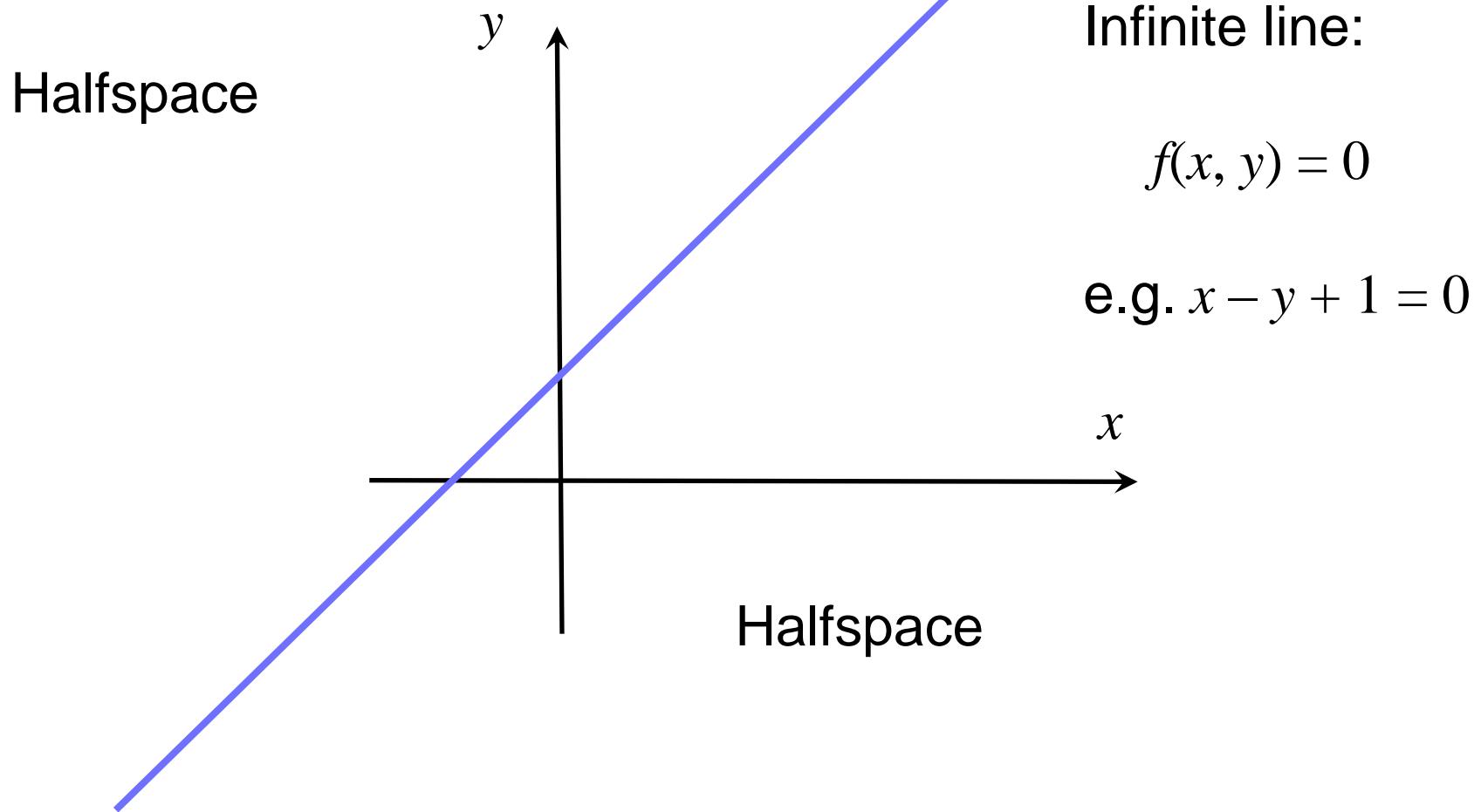
The concept of a halfspace



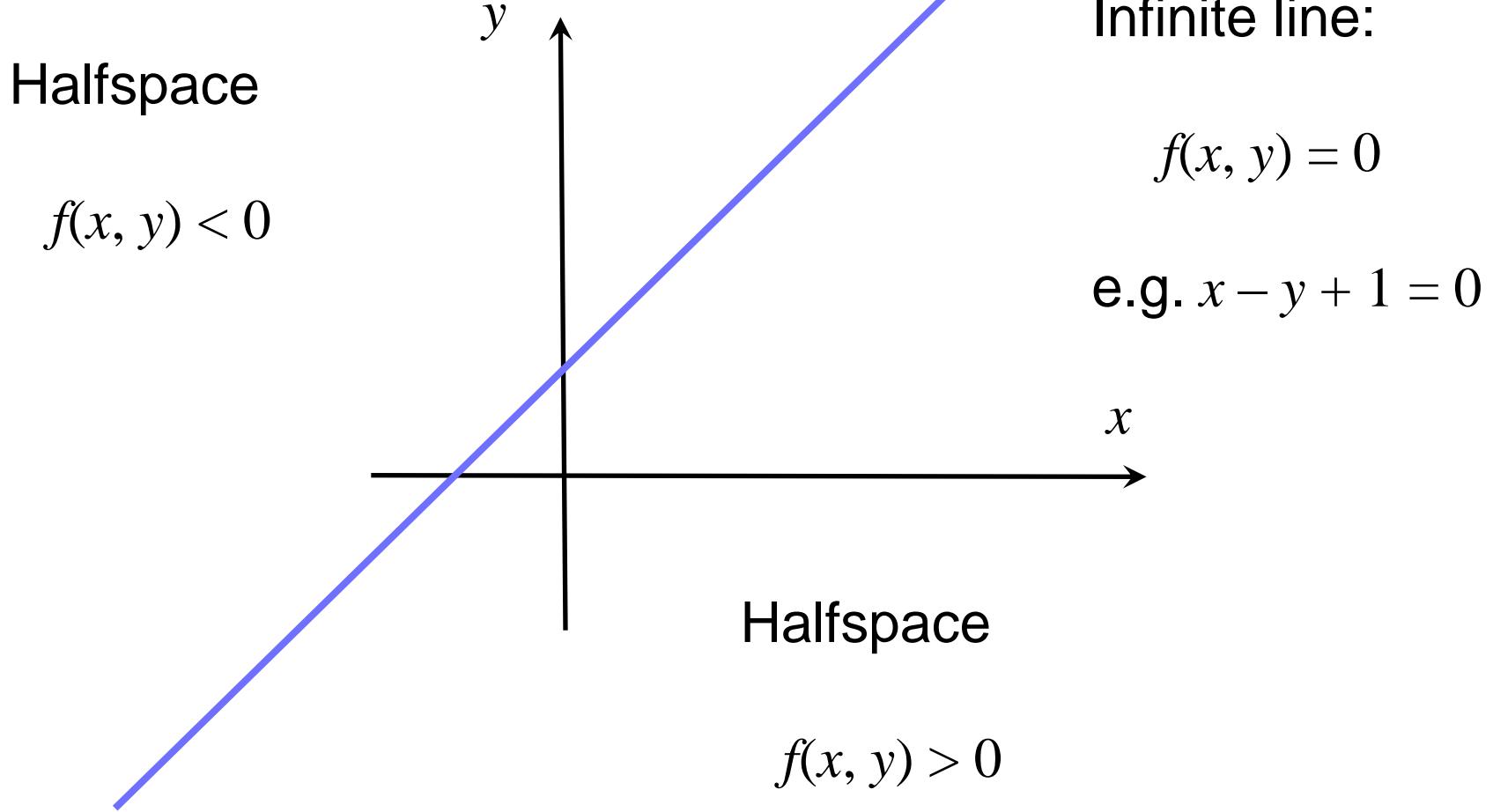
The concept of a halfspace



The concept of a halfspace



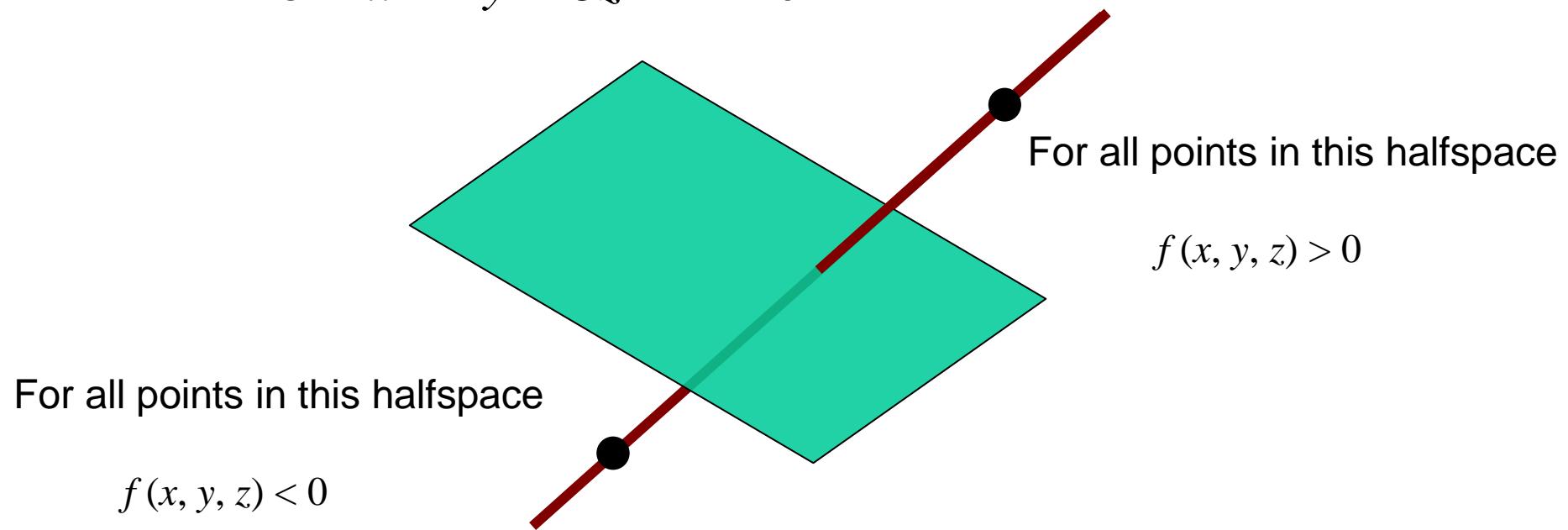
The concept of a halfspace



The concept of a halfspace in 3D

Plane equation $f(x, y, z) = 0$

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



Reminder: Homogeneous Coordinates

- Link plane equation $Ax + By + Cz + D = 0$ with vector $\mathbf{H} = (A, B, C, D)^T$ in homogeneous coordinates

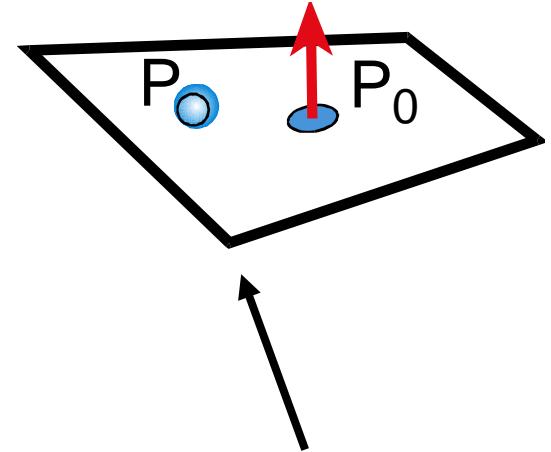
- Each point (x, y, z, w) has an infinite number of equivalent homogenous coordinates:

$$(sx, sy, sz, sw) , s \neq 0$$

$$\mathbf{H} = (A, B, C, D)^T$$

- Relates to infinite number of equivalent plane equations:

$$sAx + sBy + sCz + sD = 0 \rightarrow \mathbf{H} = \begin{pmatrix} sA \\ sB \\ sC \\ sD \end{pmatrix}$$



Point-to-Plane Distance

- Scale \mathbf{H} so that (A, B, C) becomes normalized, i.e. that

$$A^2 + B^2 + C^2 = 1$$

- Then distance is easily calculated

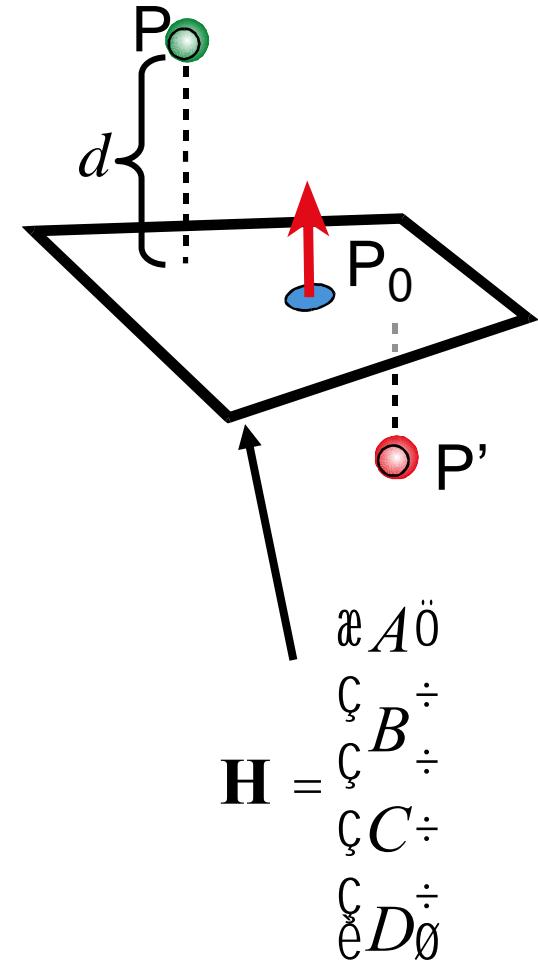
$$d = \mathbf{H} \cdot \mathbf{p} = \mathbf{H}^T \mathbf{p}$$

n.b. dot product is in *homogeneous* coordinates

- d is a *signed distance*:

positive = "inside"

negative = "outside"



Which side of the plane is a point on?

(Recall the planes in the frustum)

- If $d = \mathbf{H} \cdot \mathbf{p} \geq 0$

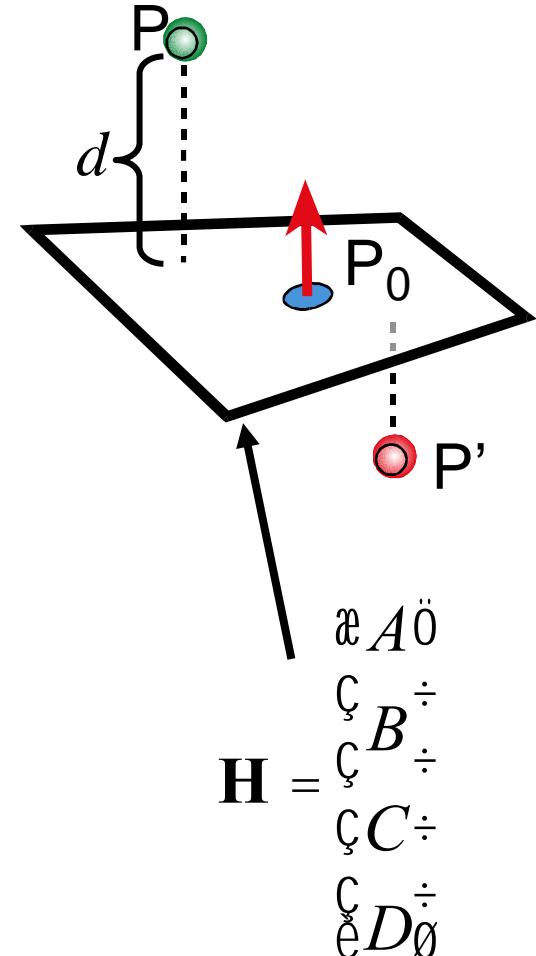
Pass through

- If $d = \mathbf{H} \cdot \mathbf{p} < 0$

Clip (or cull or reject)

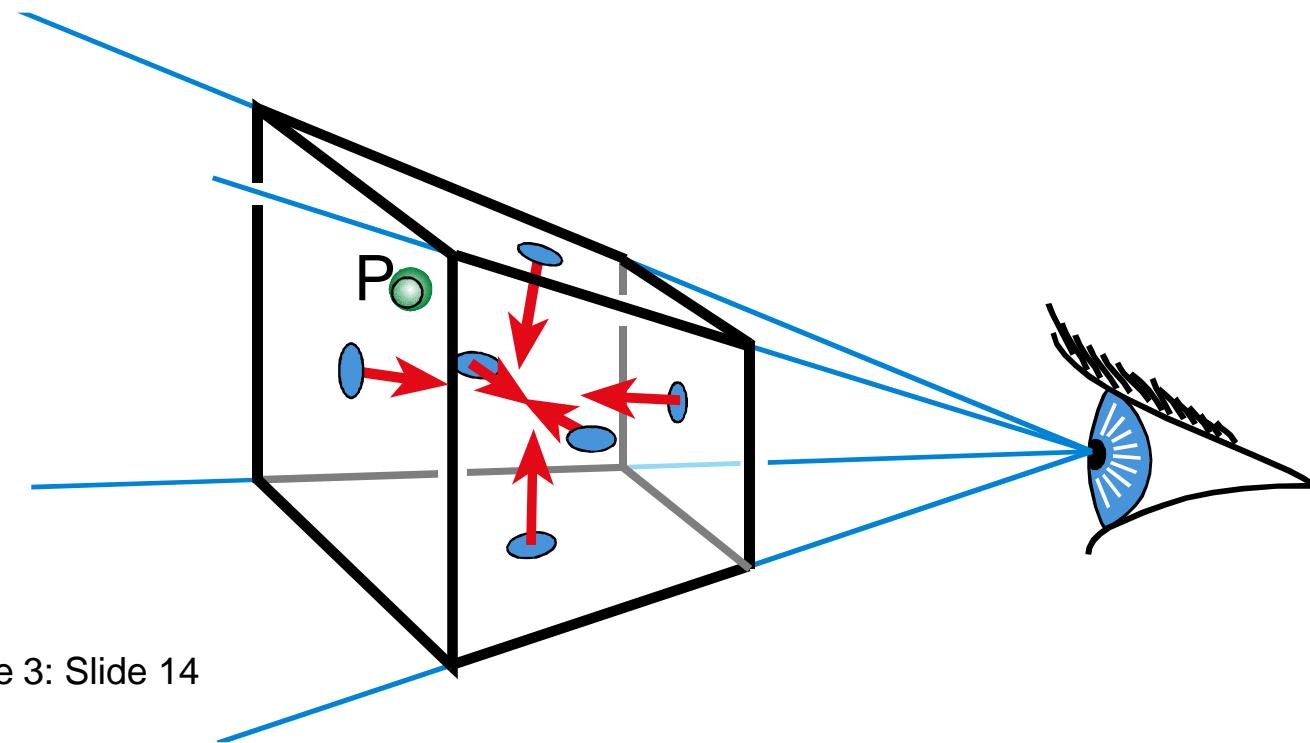
Don't really need to normalize A, B, C

We only test the *sign* of $\mathbf{H} \cdot \mathbf{p}$

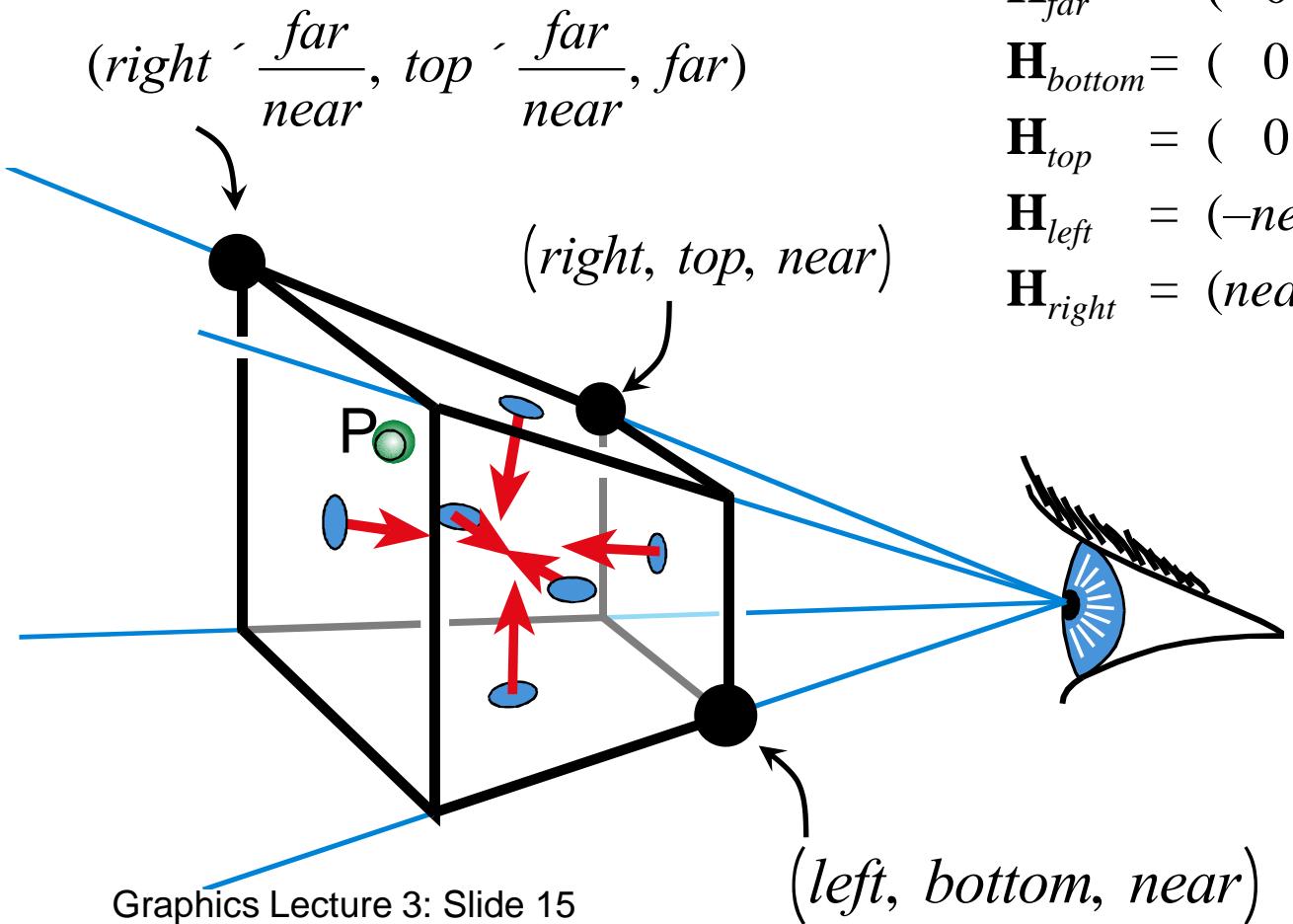


Clipping with respect to View Frustum

- Test point p against each of the 6 planes
 - Normals oriented towards the interior
 - Each has its own H
- If $H \cdot p < 0$ for any H then clip p (‘cull’ / ‘reject’)



What are the View Frustum Planes?



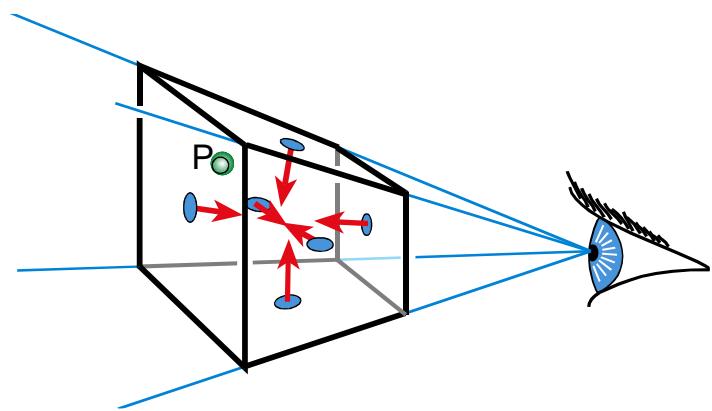
Eye at O
looking along $z+$

Example derivation

$$\begin{pmatrix} l \\ b \\ n \end{pmatrix} \times \begin{pmatrix} r \\ b \\ n \end{pmatrix} = \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ l & b & n \\ r & b & n \end{vmatrix}$$

$$\hat{i}(bn - bn) + \hat{j}(rn - ln) + \hat{k}(lb - rb) \equiv (0, n, -b)^T$$

$$\begin{aligned} (\mathbf{P} - \mathbf{P}_1) \cdot \mathbf{n} &= 0 \\ \Rightarrow ny - bz &= 0 \\ \Rightarrow \mathbf{H}_{bottom} &= (0, n, -b, 0)^T \end{aligned}$$



Line-Plane Intersection

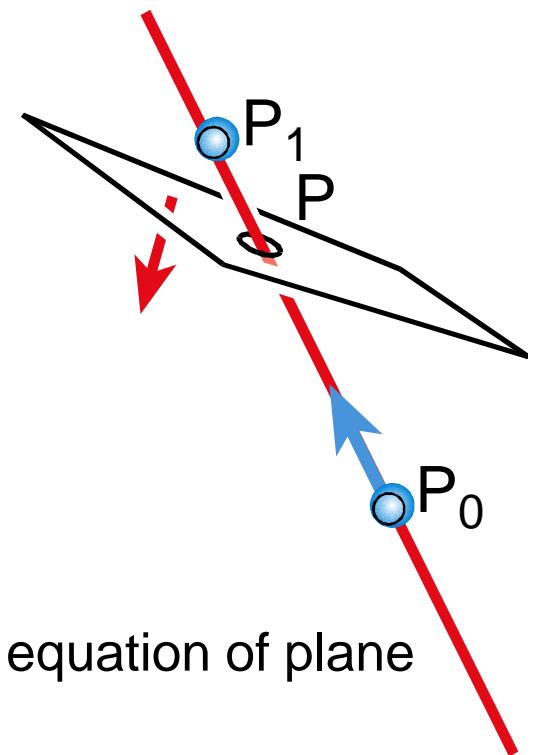
- Sometimes we need to clip lines and line segments!
- Explicit (Parametric) Line Equation

$$\mathbf{L}(\mu) = \mathbf{p}_0 + \mu (\mathbf{p}_1 - \mathbf{p}_0)$$

or

$$\mathbf{L}(\mu) = \mu \mathbf{p}_1 + (1 - \mu) \mathbf{p}_0$$

- How do we intersect?
 - Insert explicit equation of line into implicit equation of plane
 - use the normal vector



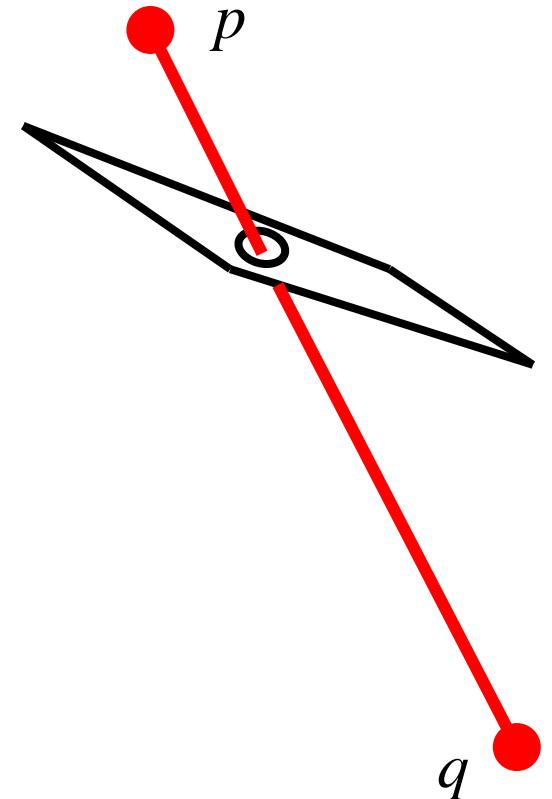
Line-Plane Intersection: Example method

To compute intersection line joining p_0, p_1 and plane:

1. For any vector \mathbf{v} lying in the plane $\mathbf{n} \cdot \mathbf{v} = 0$
2. Let the intersection point be $\mu \mathbf{p}_1 + (1-\mu) \mathbf{p}_0$
3. Choose \mathbf{v} to be *any* point on the plane.
4. A vector in the plane is given by $\mu \mathbf{p}_1 + (1-\mu) \mathbf{p}_0 - \mathbf{v}$
5. So $\mathbf{n} \cdot (\mu \mathbf{p}_1 + (1-\mu) \mathbf{p}_0 - \mathbf{v}) = 0$
6. We can solve this for μ and hence find the point of intersection

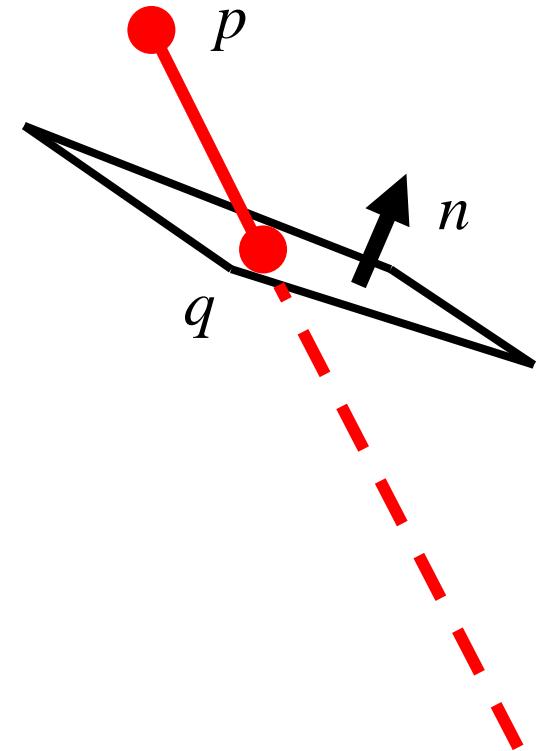
Segment Clipping

- If $\mathbf{H} \cdot \mathbf{p} > 0$ and $\mathbf{H} \cdot \mathbf{q} < 0$
- If $\mathbf{H} \cdot \mathbf{p} < 0$ and $\mathbf{H} \cdot \mathbf{q} > 0$
- If $\mathbf{H} \cdot \mathbf{p} > 0$ and $\mathbf{H} \cdot \mathbf{q} > 0$
- If $\mathbf{H} \cdot \mathbf{p} < 0$ and $\mathbf{H} \cdot \mathbf{q} < 0$



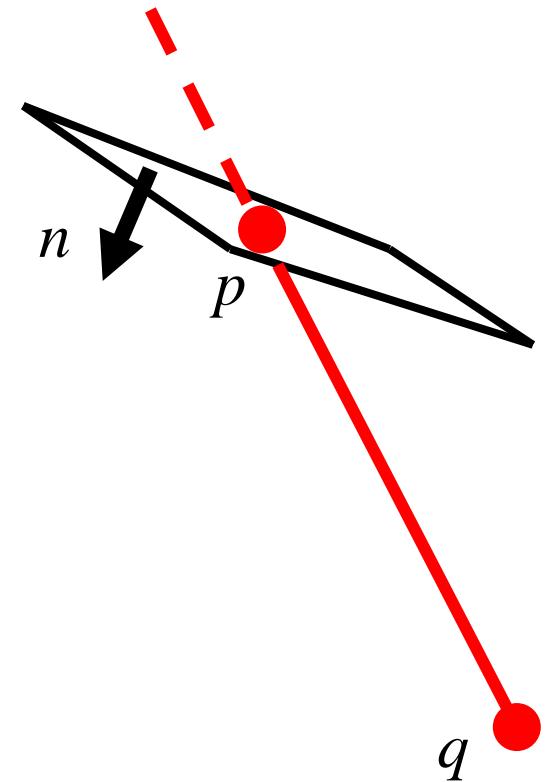
Segment Clipping

- If $\mathbf{H} \cdot \mathbf{p} > 0$ and $\mathbf{H} \cdot \mathbf{q} < 0$
 - clip q to plane
- If $\mathbf{H} \cdot \mathbf{p} < 0$ and $\mathbf{H} \cdot \mathbf{q} > 0$
- If $\mathbf{H} \cdot \mathbf{p} > 0$ and $\mathbf{H} \cdot \mathbf{q} > 0$
- If $\mathbf{H} \cdot \mathbf{p} < 0$ and $\mathbf{H} \cdot \mathbf{q} < 0$



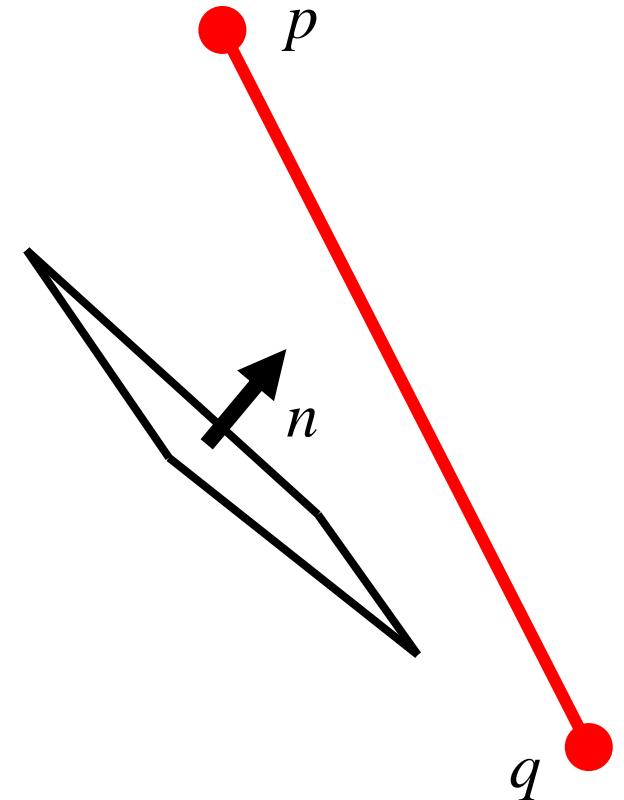
Segment Clipping

- If $\mathbf{H} \cdot \mathbf{p} > 0$ and $\mathbf{H} \cdot \mathbf{q} < 0$
 - clip q to plane
- If $\mathbf{H} \cdot \mathbf{p} < 0$ and $\mathbf{H} \cdot \mathbf{q} > 0$
 - clip p to plane
- If $\mathbf{H} \cdot \mathbf{p} > 0$ and $\mathbf{H} \cdot \mathbf{q} > 0$
- If $\mathbf{H} \cdot \mathbf{p} < 0$ and $\mathbf{H} \cdot \mathbf{q} < 0$



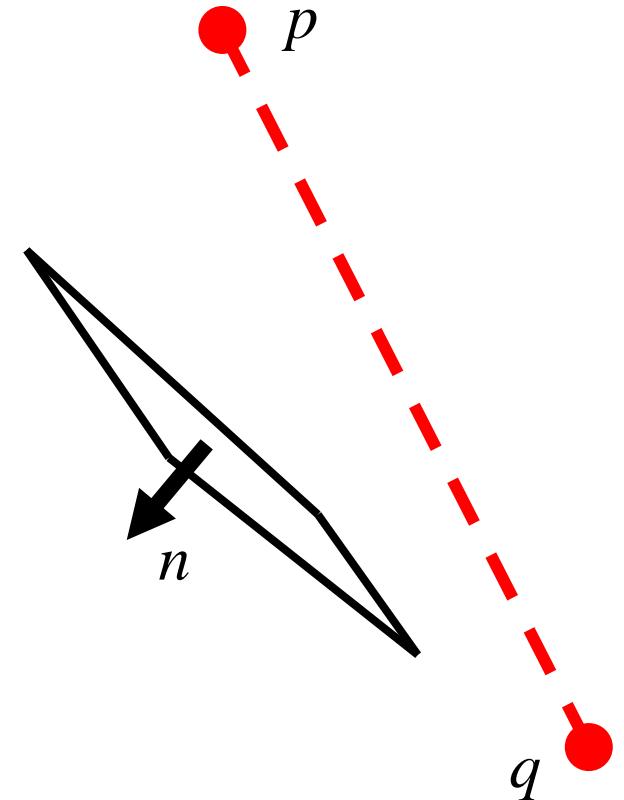
Segment Clipping

- If $\mathbf{H} \cdot \mathbf{p} > 0$ and $\mathbf{H} \cdot \mathbf{q} < 0$
 - clip q to plane
- If $\mathbf{H} \cdot \mathbf{p} < 0$ and $\mathbf{H} \cdot \mathbf{q} > 0$
 - clip p to plane
- If $\mathbf{H} \cdot \mathbf{p} > 0$ and $\mathbf{H} \cdot \mathbf{q} > 0$
 - pass through
- If $\mathbf{H} \cdot \mathbf{p} < 0$ and $\mathbf{H} \cdot \mathbf{q} < 0$



Segment Clipping

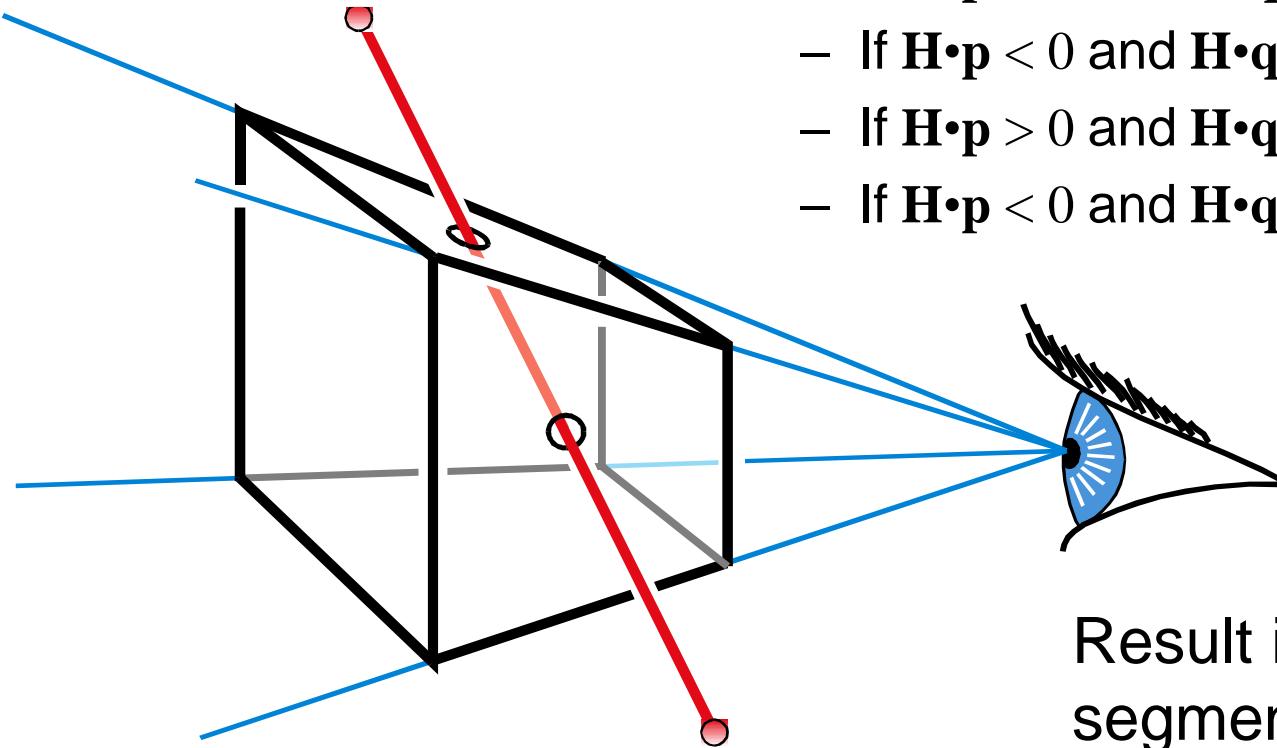
- If $\mathbf{H} \cdot \mathbf{p} > 0$ and $\mathbf{H} \cdot \mathbf{q} < 0$
 - clip q to plane
- If $\mathbf{H} \cdot \mathbf{p} < 0$ and $\mathbf{H} \cdot \mathbf{q} > 0$
 - clip p to plane
- If $\mathbf{H} \cdot \mathbf{p} > 0$ and $\mathbf{H} \cdot \mathbf{q} > 0$
 - pass through
- If $\mathbf{H} \cdot \mathbf{p} < 0$ and $\mathbf{H} \cdot \mathbf{q} < 0$
 - clipped out



Clipping against the frustum

For each frustum plane H

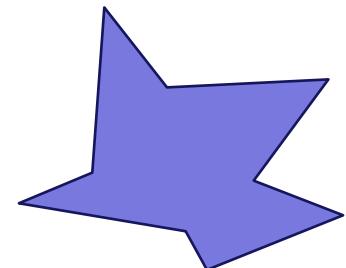
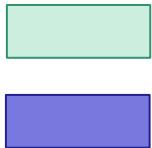
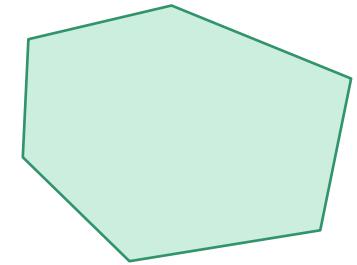
- If $H \cdot p > 0$ and $H \cdot q < 0$, clip q
- If $H \cdot p < 0$ and $H \cdot q > 0$, clip p
- If $H \cdot p > 0$ and $H \cdot q > 0$, pass through
- If $H \cdot p < 0$ and $H \cdot q < 0$, clipped out



Result is a single segment.

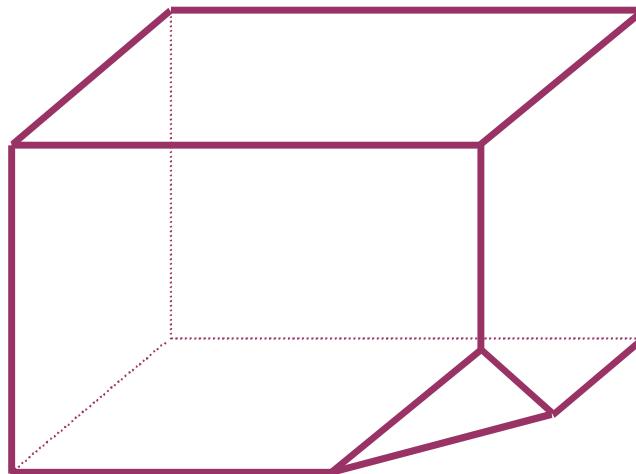
Clipping and containment

- Clipping can be carried out against any object
 - Not just a viewing frustum
- Clipping against an arbitrary object
- Need a test for containment
 - i.e. is a point inside or outside the object
- Can develop containment test for
 - Convex objects: Common problem, e.g. convex polyhedra
 - Concave objects: Harder than convex case



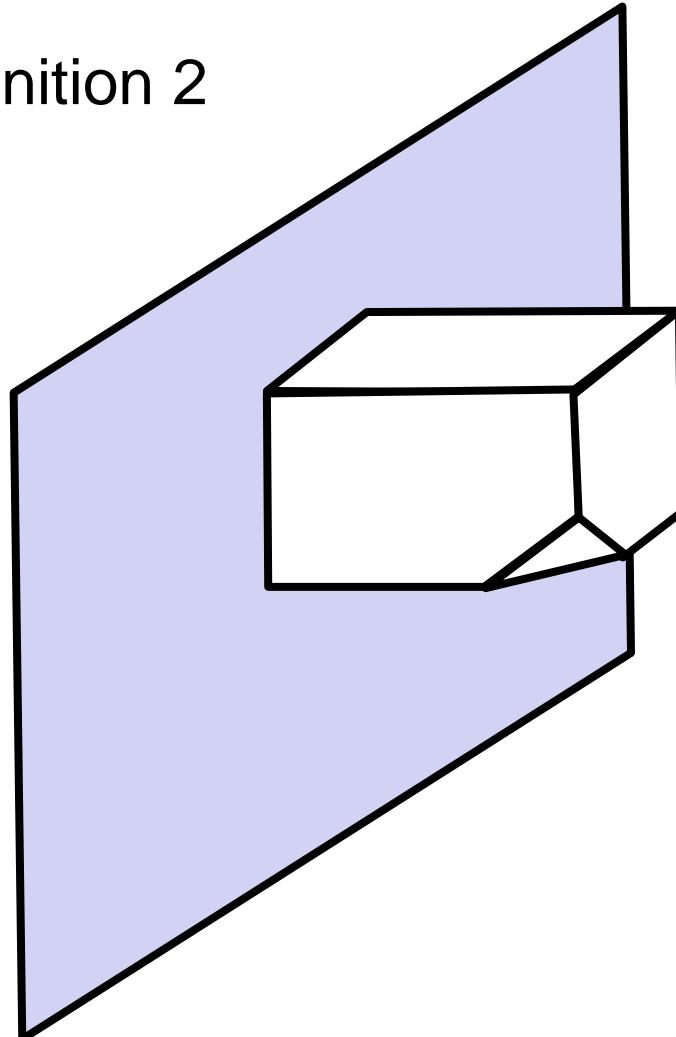
Convex objects: Two Definitions

1. A line joining any two points on the boundary lies inside the object.
2. The object is the intersection of planar halfspaces.



Testing if an object is convex

Illustration of definition 2



Testing if an object is convex: Algorithm

```
convex = true
for each face of the object {
    find plane equation of face: F(x,y,z) = 0
    choose object point (xi,yi,zi) not on the face

    for all other points of the object {
        if (sign(F(xj,yj,zj)) != sign(F(xi,yi,zi)))
            then convex = false
    }
}
```

Works due to definition 2, all points of the object must lie entirely to one side of each face

Test containment within a convex object: Algorithm

```
let the test point be ( $x_t, y_t, z_t$ )
contained = true
for each face of the convex object {
    find plane equation of face:  $F(x, y, z) = 0$ 
    choose an object point ( $x_i, y_i, z_i$ ) not on the face

    if (sign(  $F(x_t, y_t, z_t)$  ) != sign(  $F(x_i, y_i, z_i)$  ))
        then contained = false
}
```

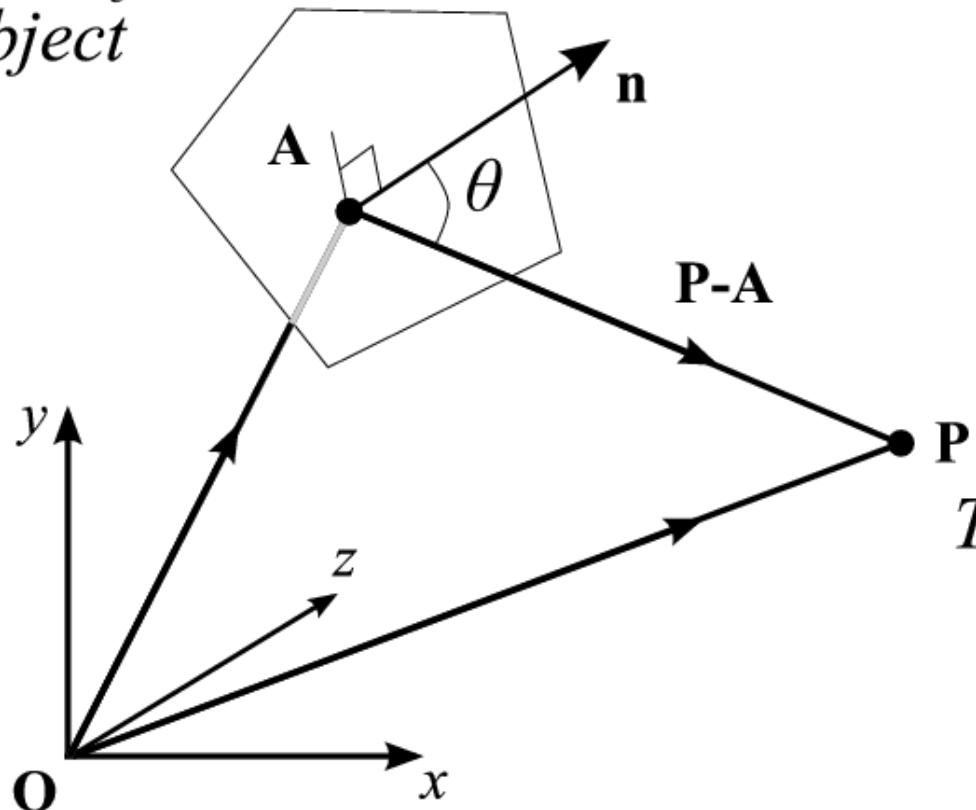
Vector formulation

- The same test can be expressed in vector form.
- This avoids the need to calculate the Cartesian equation of the plane, if, in our model we store the normal vector \mathbf{n} for each face of our object.

Vector test for containment

P is on the ‘inside’ of the face if:

face of convex object



θ is acute

$$\cos \theta > 0$$

$$\mathbf{n} \cdot (\mathbf{P-A}) > 0$$

Because

$$\mathbf{n} \cdot (\mathbf{P-A}) = |\mathbf{n}| |\mathbf{P-A}| \cos \theta$$

Test point

Normal vector to a face

- The vector formulation does not require us to find the plane equation of a face, but it does require us to find a normal vector to the plane;
- Same thing really since for plane

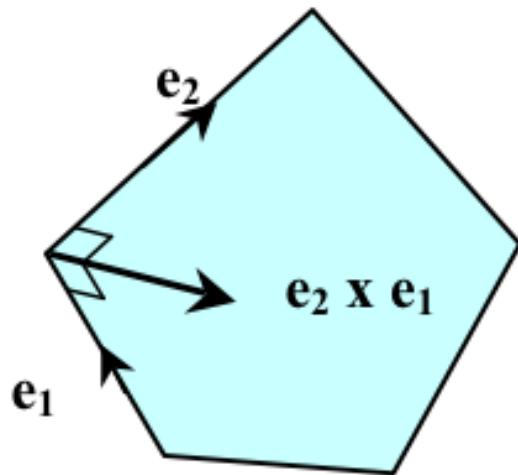
$$Ax + By + Cz + D = 0$$

- A normal vector is

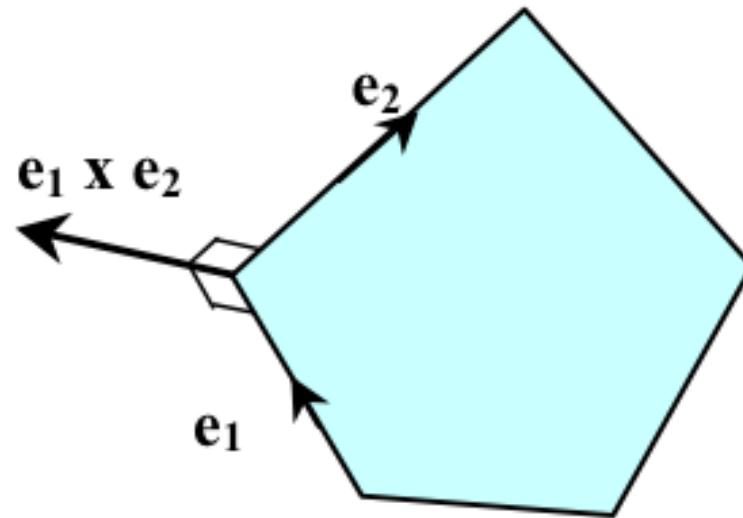
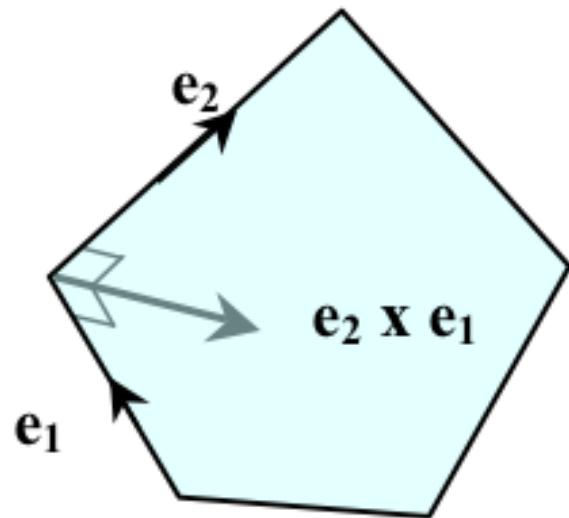
$$\mathbf{n} = \frac{A}{\sqrt{A^2 + B^2 + C^2}} \begin{pmatrix} A \\ B \\ C \end{pmatrix}$$

Finding a normal vector

- The normal vector can be found from the cross product of two vectors on the plane, say two edge vectors

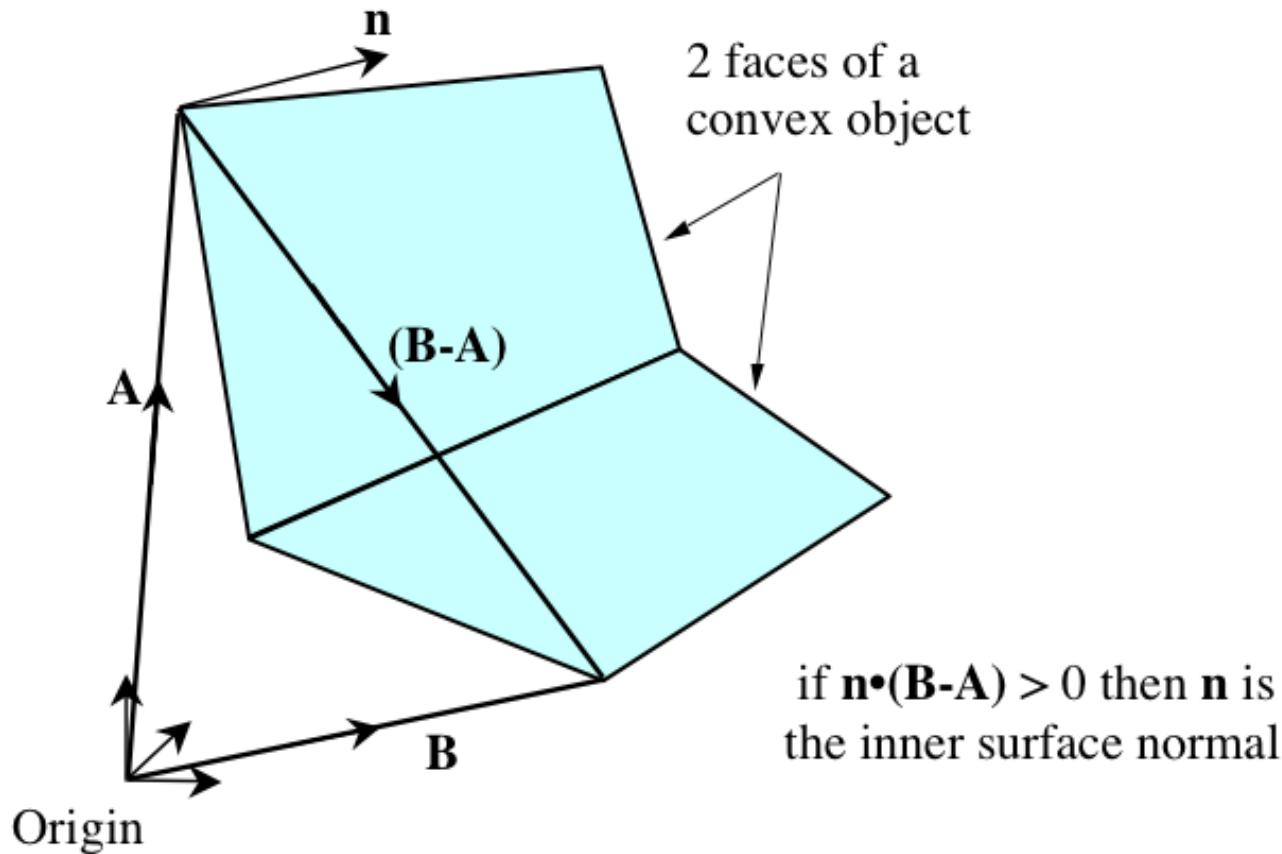


But which normal vector points inwards?



Checking normal direction (convex object)

Is \mathbf{n} an inner normal?



Problem Break

- A face of a convex object lies in the plane

$$3x + 5y + 7z + 1 = 0$$

and a vertex \mathbf{v} is $(-1, -1, 1)$. A normal vector is therefore

$$\mathbf{n} = (3, 5, 7)^T$$

- Problems:
 1. If another vertex of the object is $\mathbf{w} = (1, 1, 1)$ determine whether \mathbf{n} is an inner or outer surface normal.
 2. Determine whether the point $\mathbf{p} = (1, 0, -1)$ is on the inside or the outside of the face.

Solution to Q2

Method 2:

The inner surface normal is $\mathbf{n} = (3, 5, 7)$

for the test point
and face vertex

$$\mathbf{p} = (1, 0, -1)$$

$$\mathbf{v} = (-1, -1, 1)$$

$$\mathbf{p} - \mathbf{v} = (2, 1, -2)$$

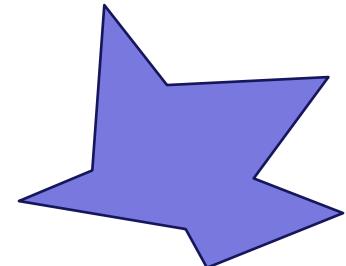
$$\mathbf{n} \cdot (\mathbf{p} - \mathbf{v}) = -3$$

Thus the angle to the normal is > 90

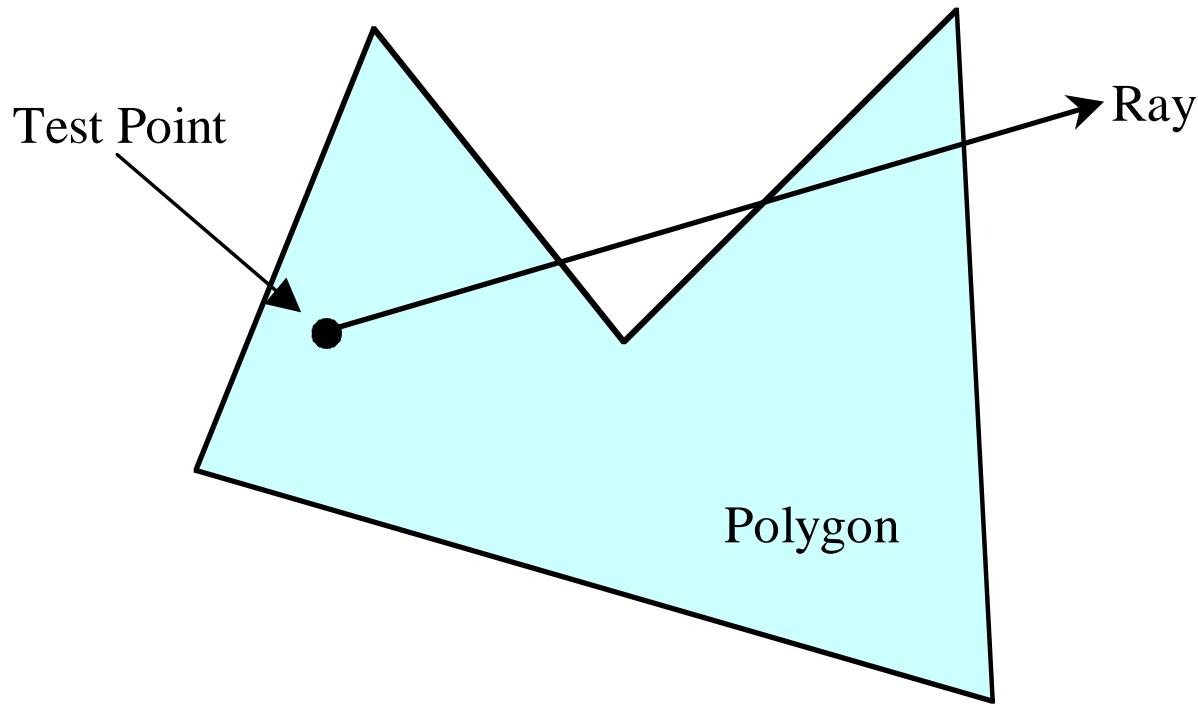
So the point \mathbf{p} is on the outside

Concave Objects

- Containment and clipping can also be carried out with concave objects.
- Most algorithms are based on the ray containment test.



The Ray test in two dimensions



Find all intersections between the ray and the polygon edges.
If the number of intersections is odd the point is contained

Calculating intersections with rays

- Rays have equivalent equations to lines, but go in only one direction. For test point \mathbf{T} a ray is defined as

$$\mathbf{R} = \mathbf{T} + \mu \mathbf{d} , \quad \mu > 0$$

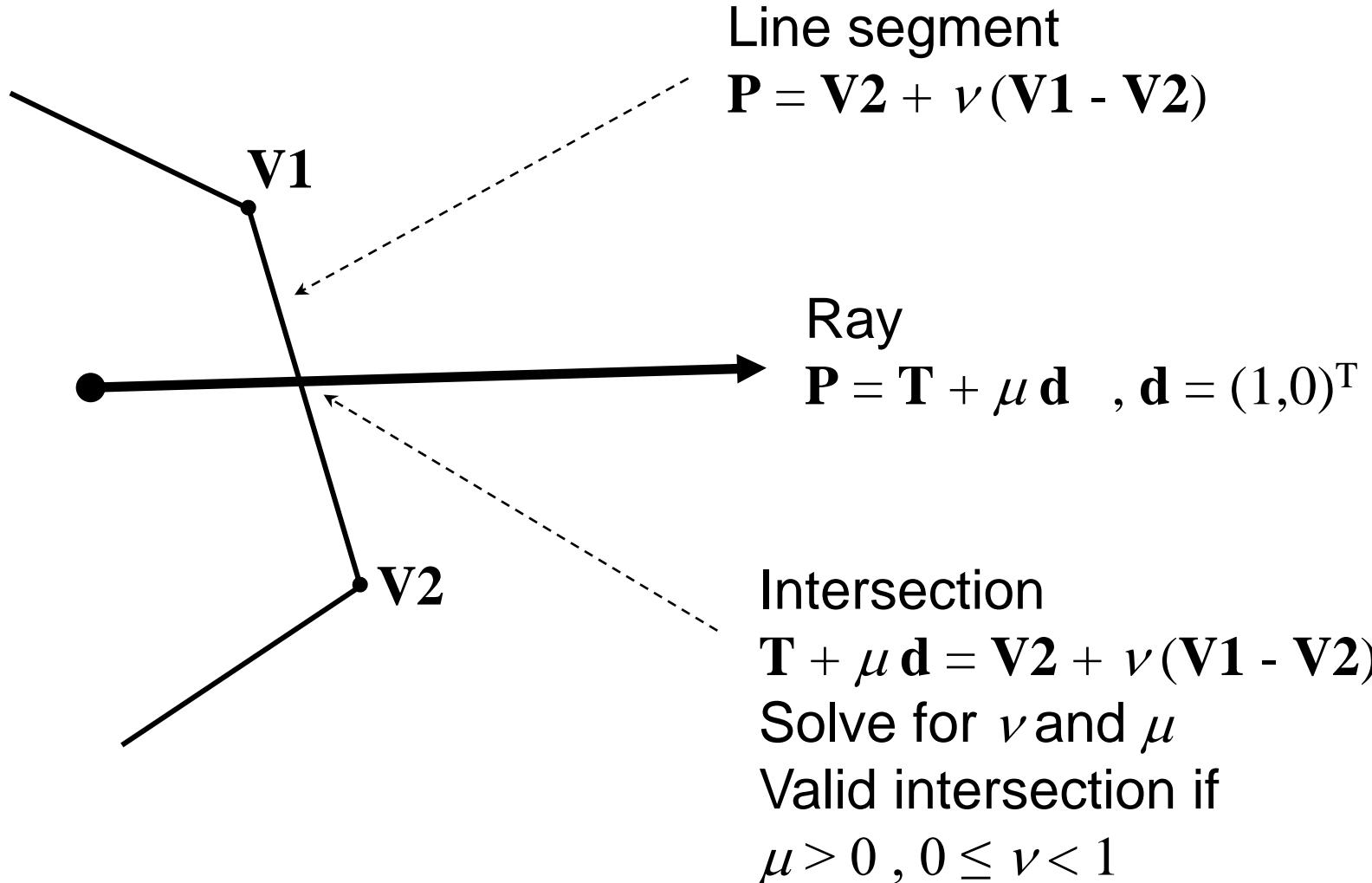
- We choose a simple to compute direction e.g.

$$\mathbf{d} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

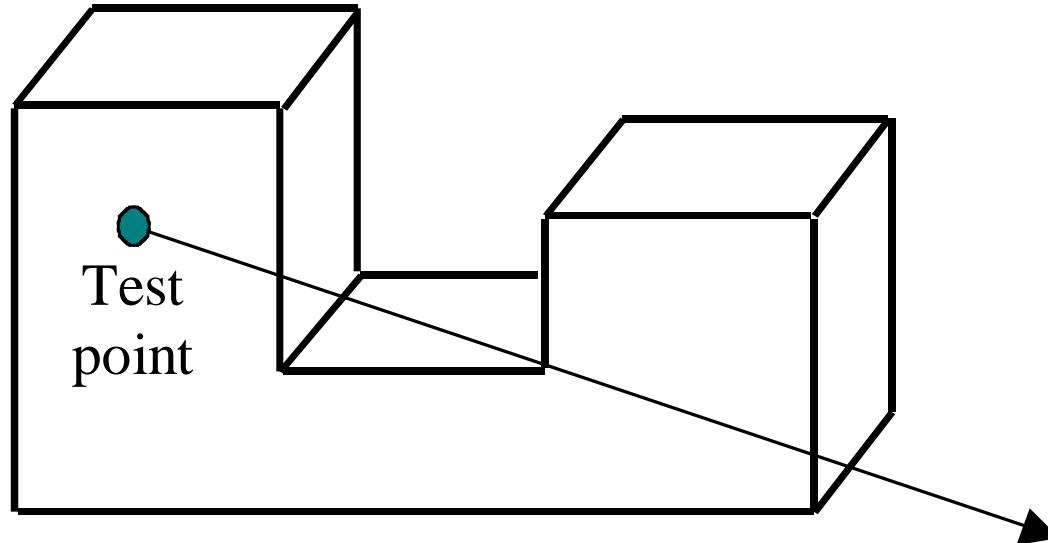
or

$$\mathbf{d} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Valid Intersections



Extending the ray test to 3D

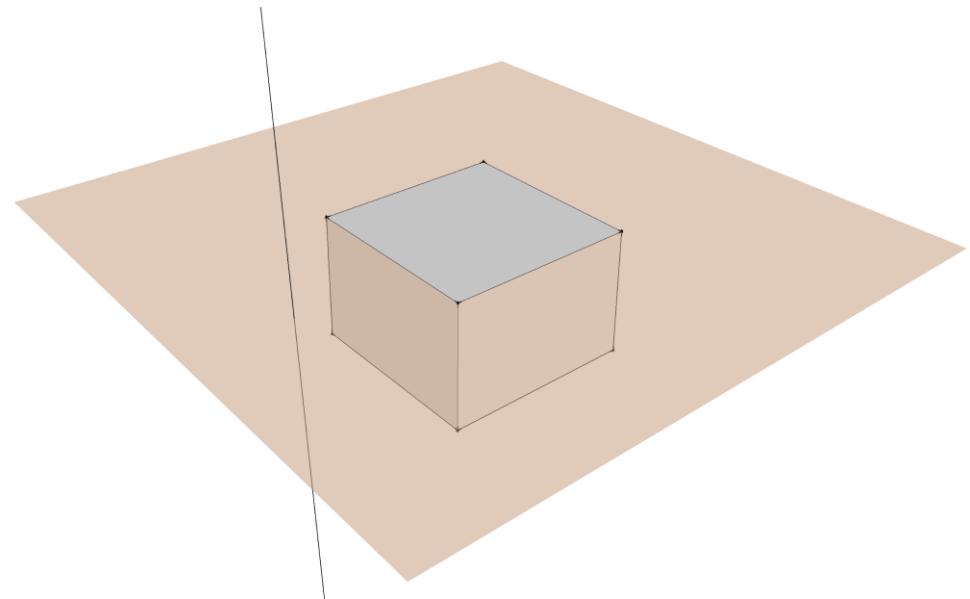
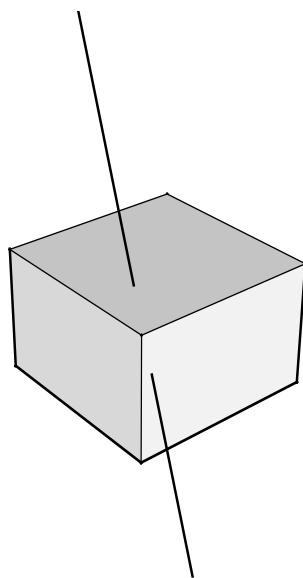


A ray is projected in any direction.

If the number of intersections with the object is odd, then the test point is inside

3D Ray test

- There are two stages:
 1. Compute the intersection of the ray with the plane of each face.
 2. If the intersection is in the positive part of the ray ($\mu > 0$) check whether the intersection point is contained in the face (i.e. not just in the planar *extension* of the face).

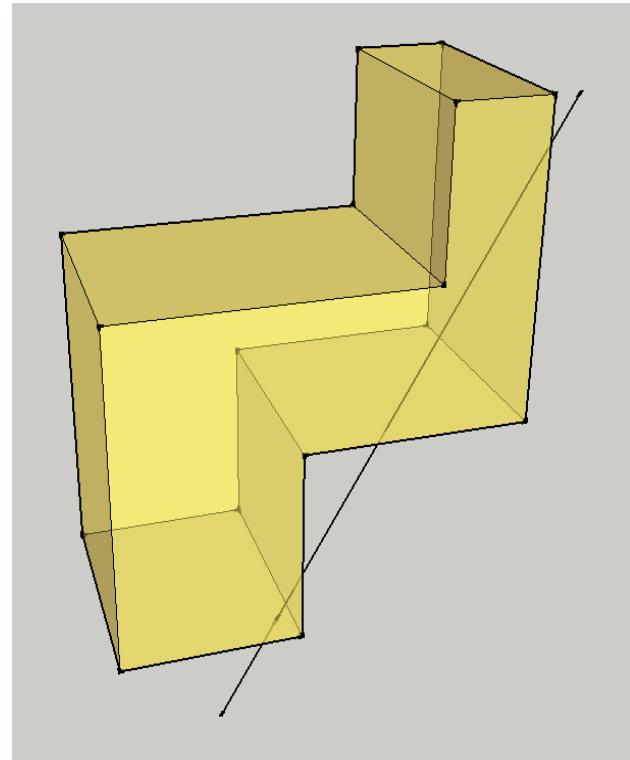


The plane of a face

- Unfortunately the plane of a face does not in general line up with the Cartesian axes, so the second part is not a two dimensional problem.
- However, containment is invariant under orthographic projection, so it can be simply reduced to two dimensions.

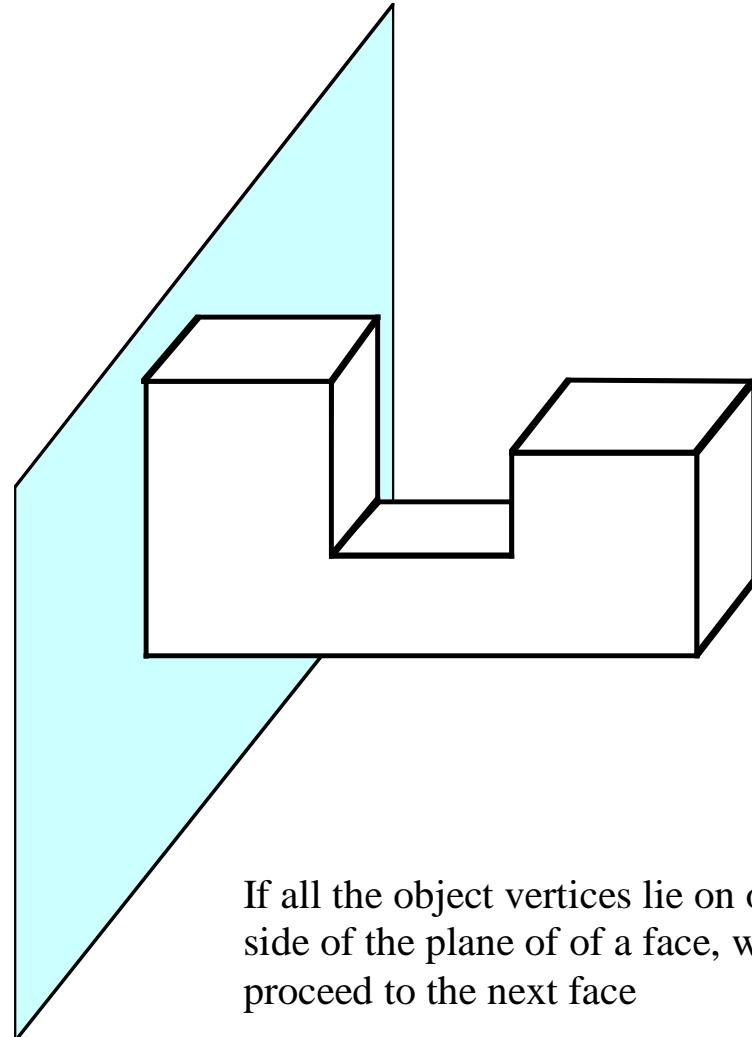
Clipping to concave volumes

- Find every intersection of the line to be clipped with the volume
- This divides the line into one or more segments.
- Test a point on the first segment for containment
- Adjacent segments will be alternately inside and out.

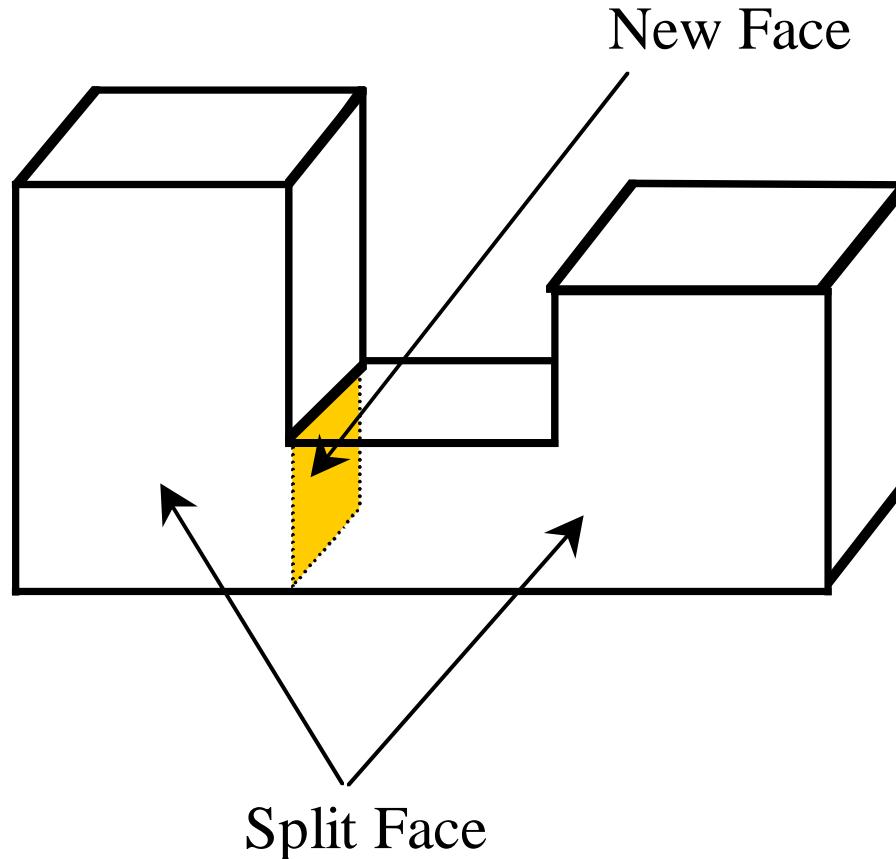


Splitting a volume into convex parts

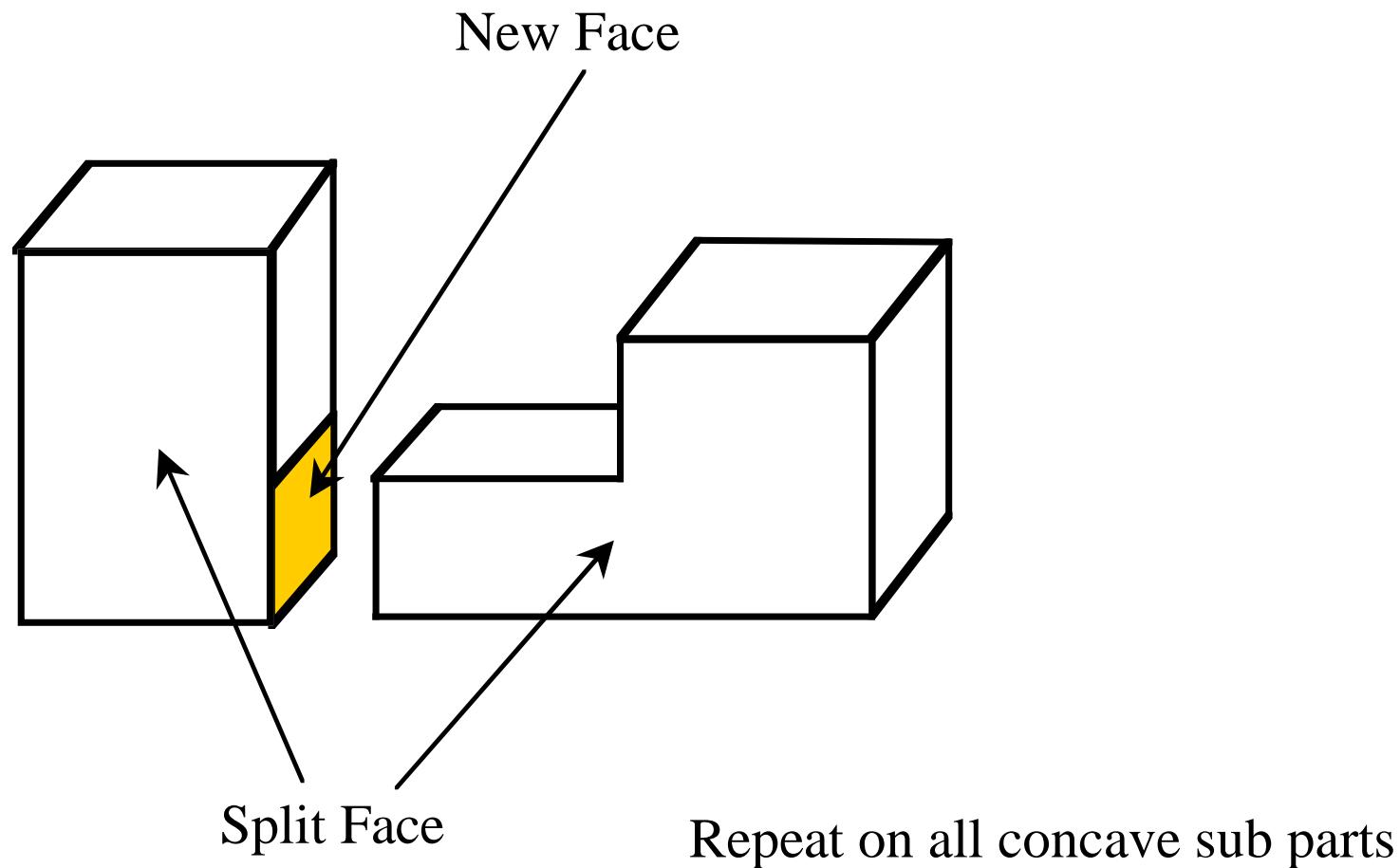
- Split concave volume into convex parts
- Can apply tests for convex objects to each of the parts
- Consider each face



If the plane of a face cuts the object:



Split the Object



Interactive Computer Graphics: Lecture 4

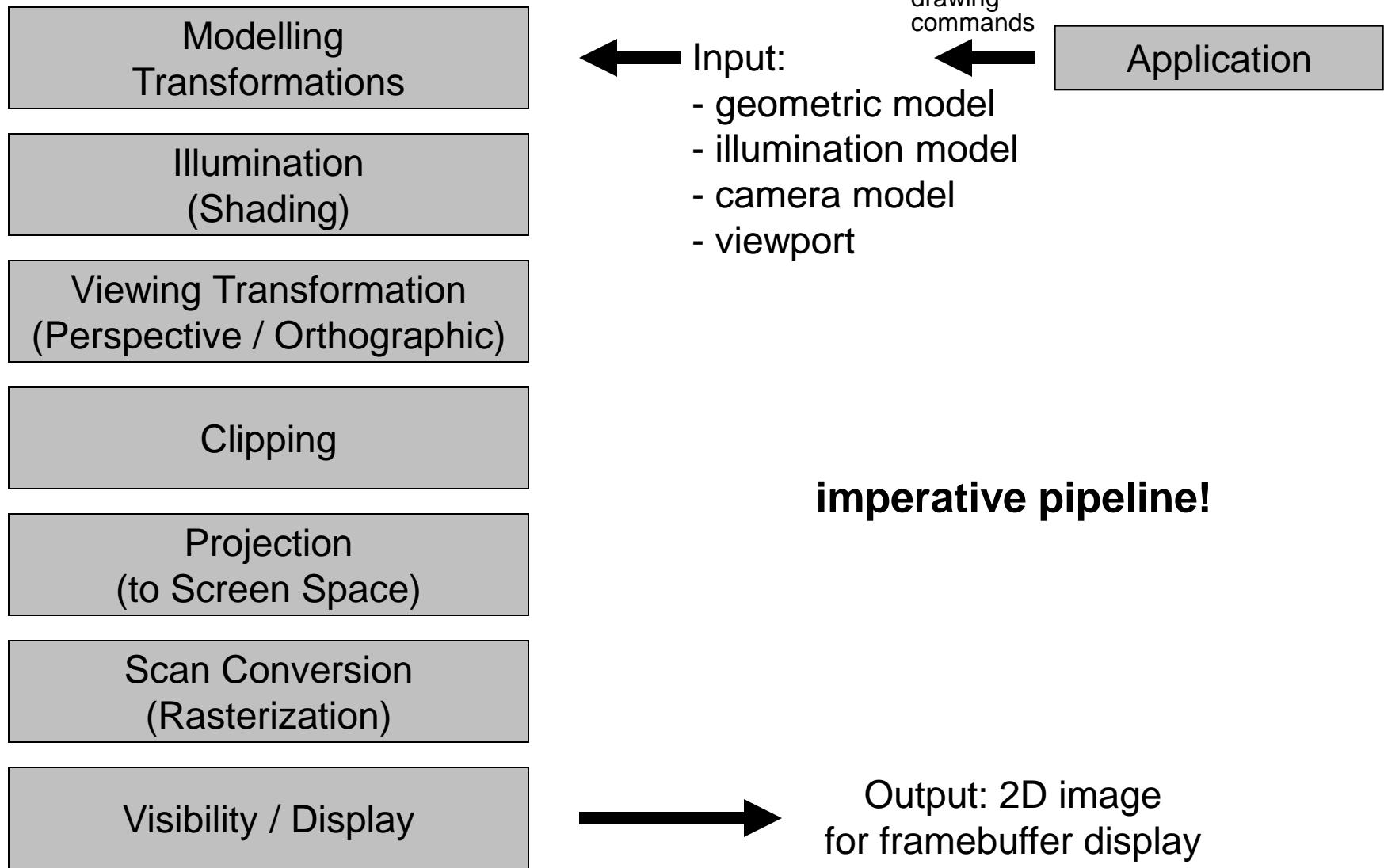
Graphics Pipeline and APIs

Some slides adopted from Markus Steinberger and Dieter Schmalstieg

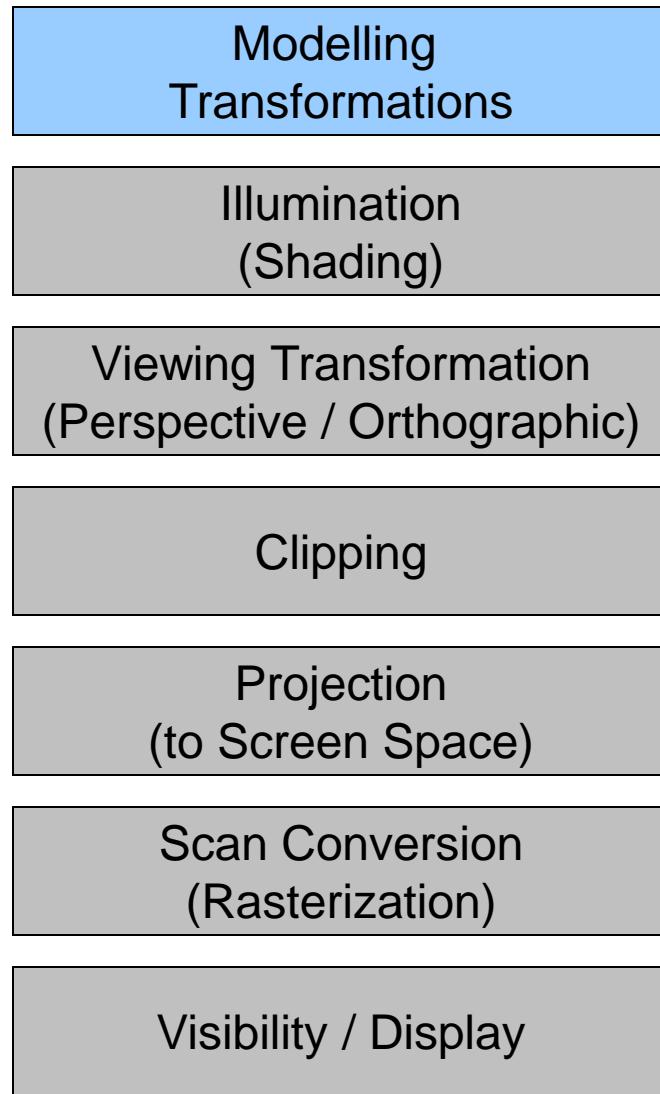
The Graphics Pipeline: High-level view

- Declarative (What, not How)
 - For example virtual camera with scene description, e.g. scene graphs
 - Every object may know about every other object
 - Renderman, Inventor, OpenSceneGraph,...
- Imperative (How, not What)
 - Emit a sequence of drawing commands
 - For example: draw a point (vertex) at position (x,y,z)
 - Objects can be drawn independant from each other
 - OpenGL, PostScript, etc.
- You can always build a declarative pipeline on top of imperative model

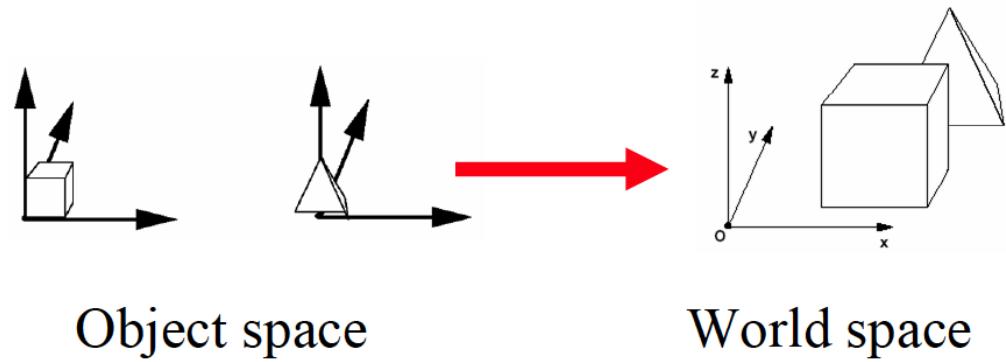
The Graphics Pipeline



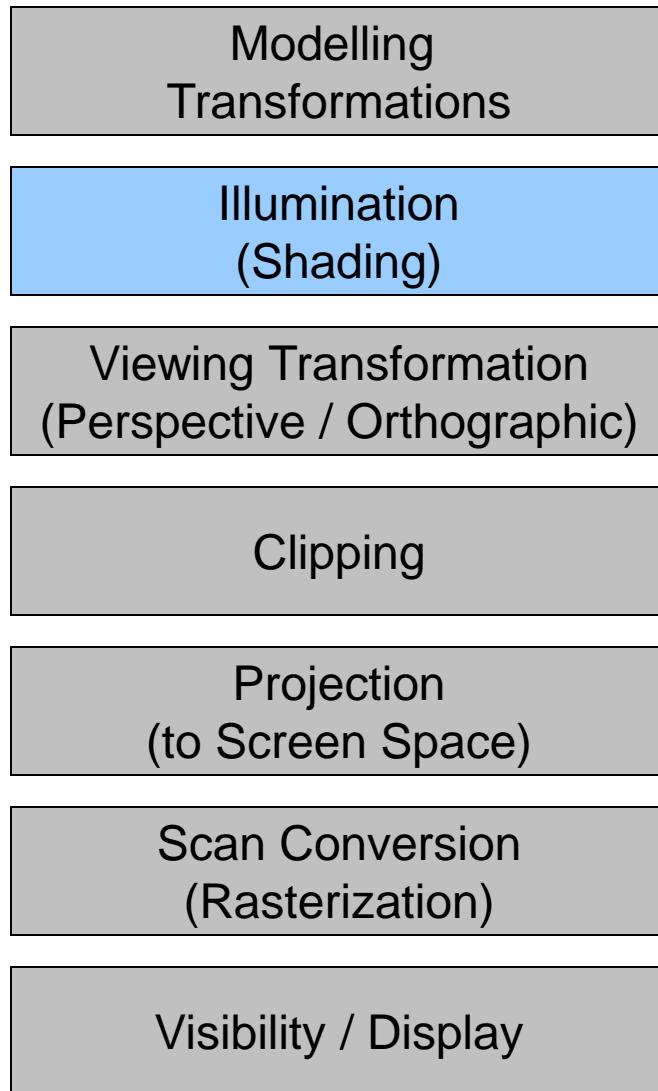
The Graphics Pipeline



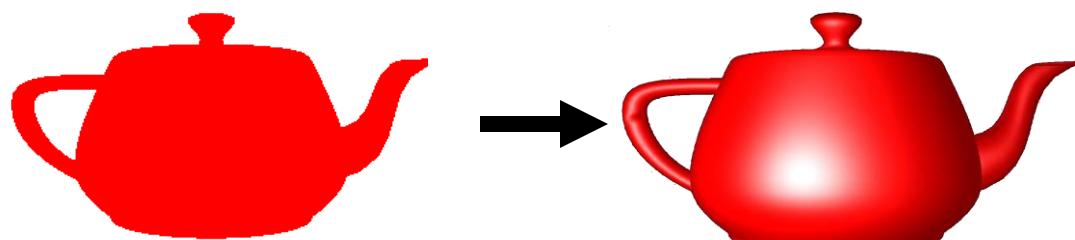
- 3D models are defined in their own coordinate system
- Modeling transformations orient the models within a common coordinate frame (world coordinates)



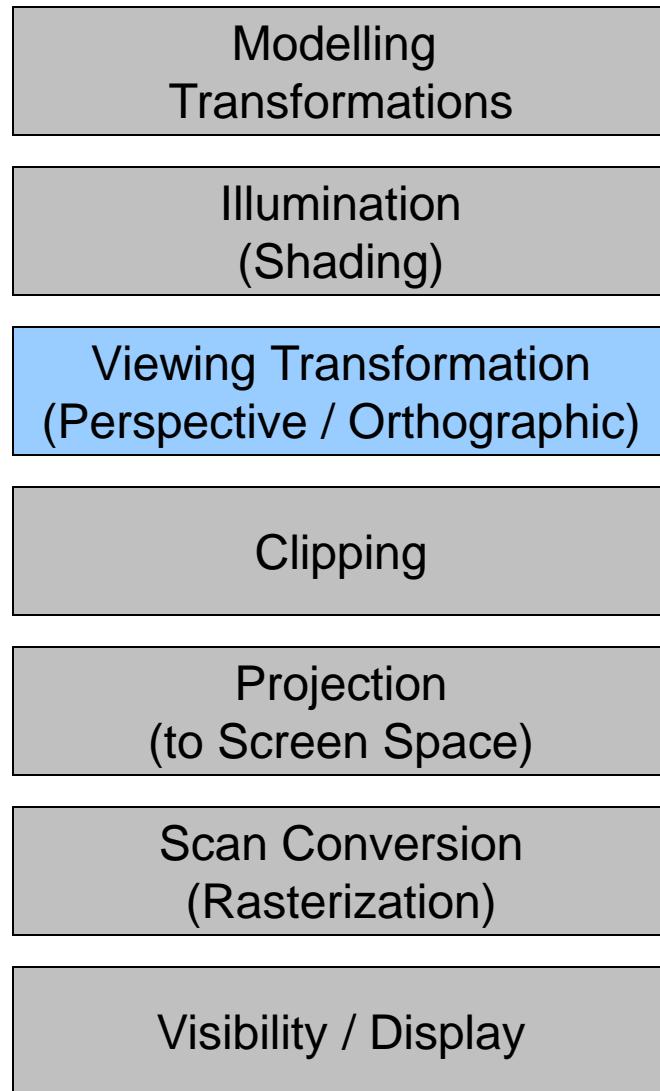
The Graphics Pipeline



- Vertices are lit (shaded) according to material properties, surface properties and light sources
- Uses a local lighting model



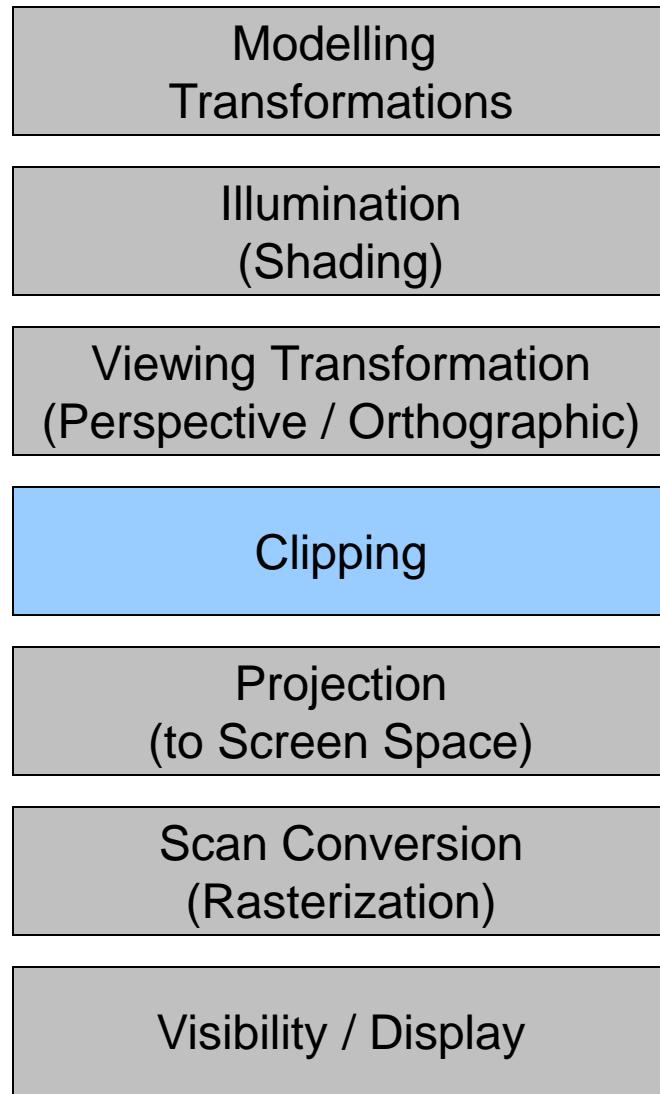
The Graphics Pipeline



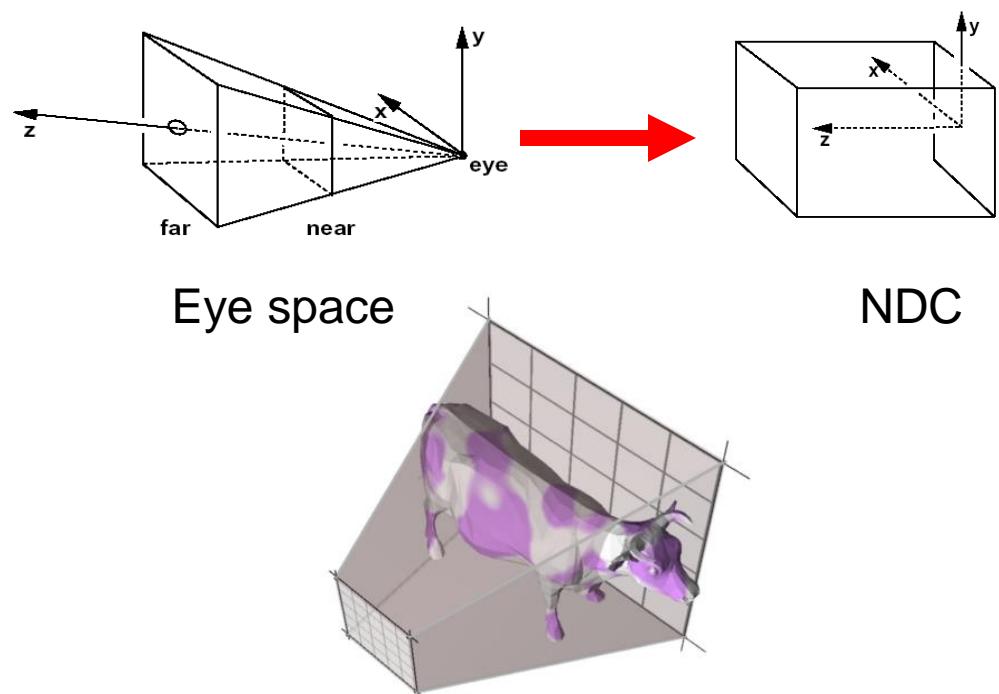
- Maps world space to eye (camera) space (matrix evaluation)
- Viewing position is transformed to origin and viewing direction is oriented along some axis (typically z)



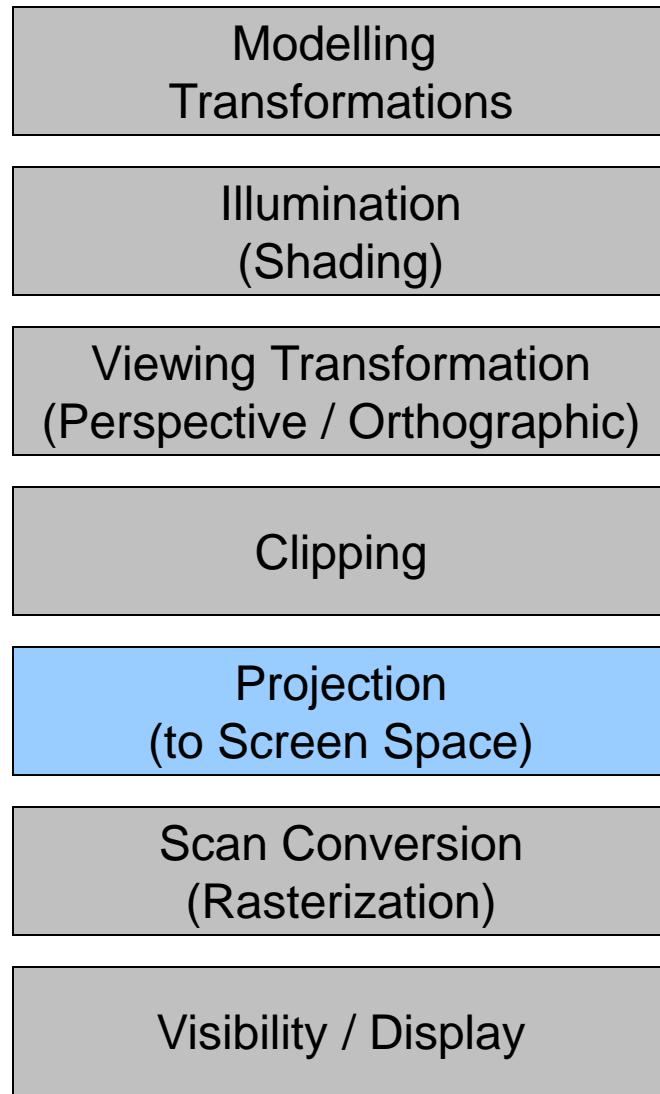
The Graphics Pipeline



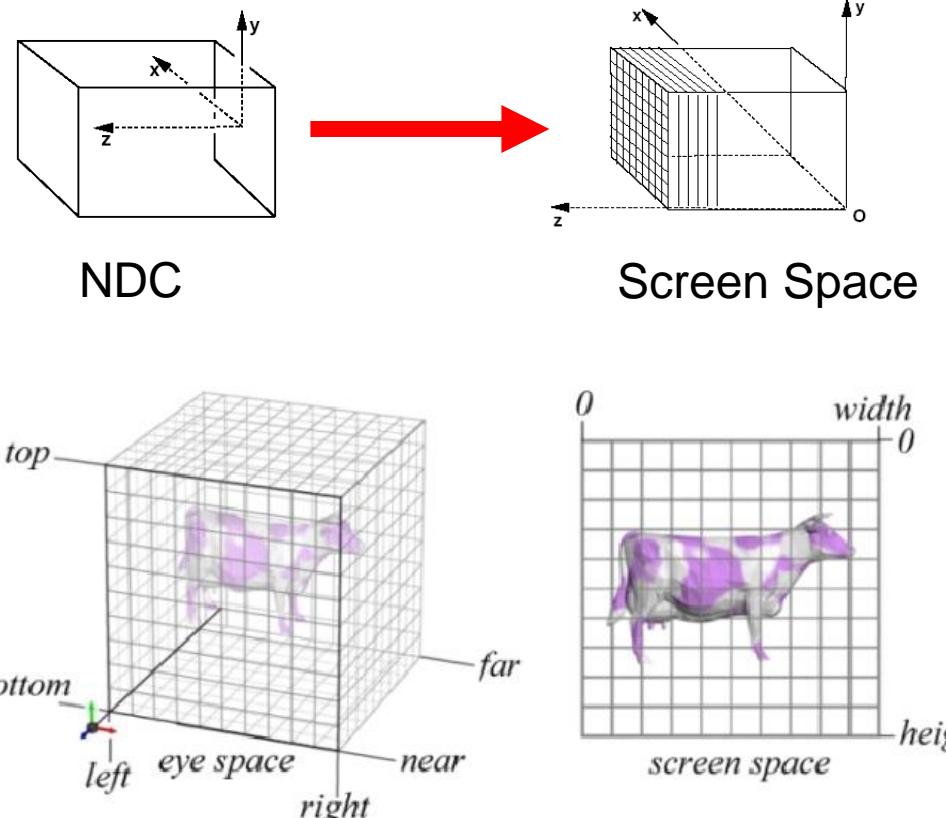
- Portions of the scene outside the viewing volume (view frustum) are removed (clipped)
- Transform to Normalized Device Coordinates



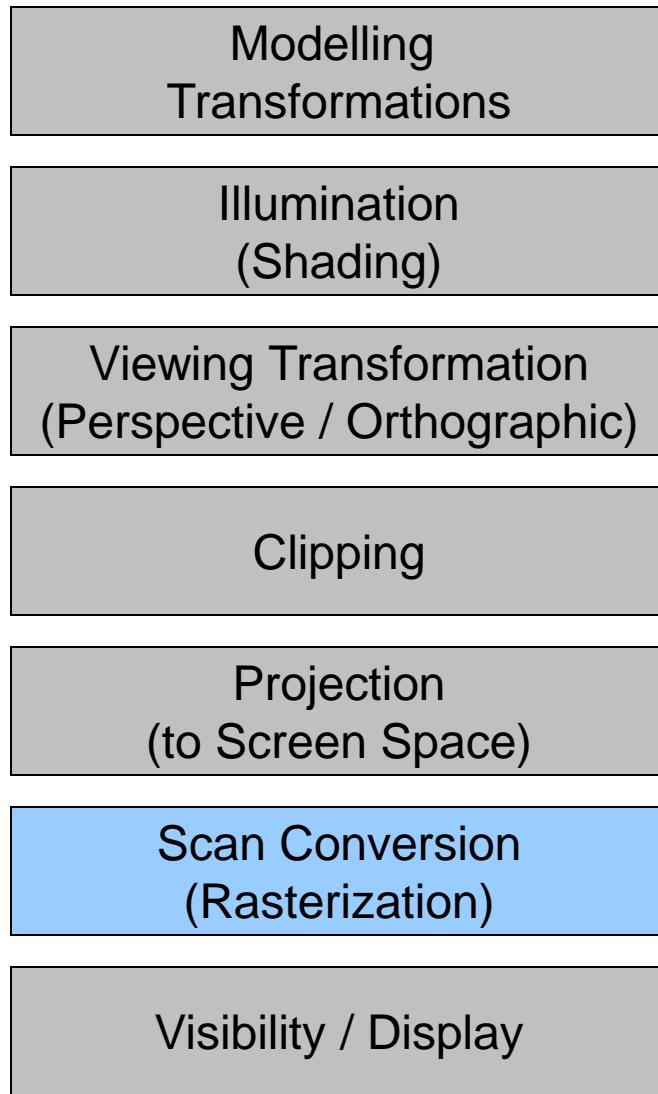
The Graphics Pipeline



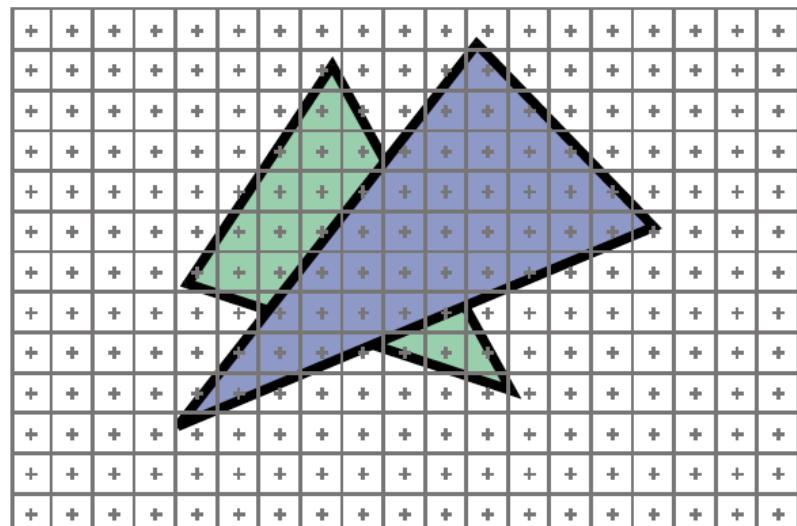
- The objects are projected to the 2D imaging plane (screen space)



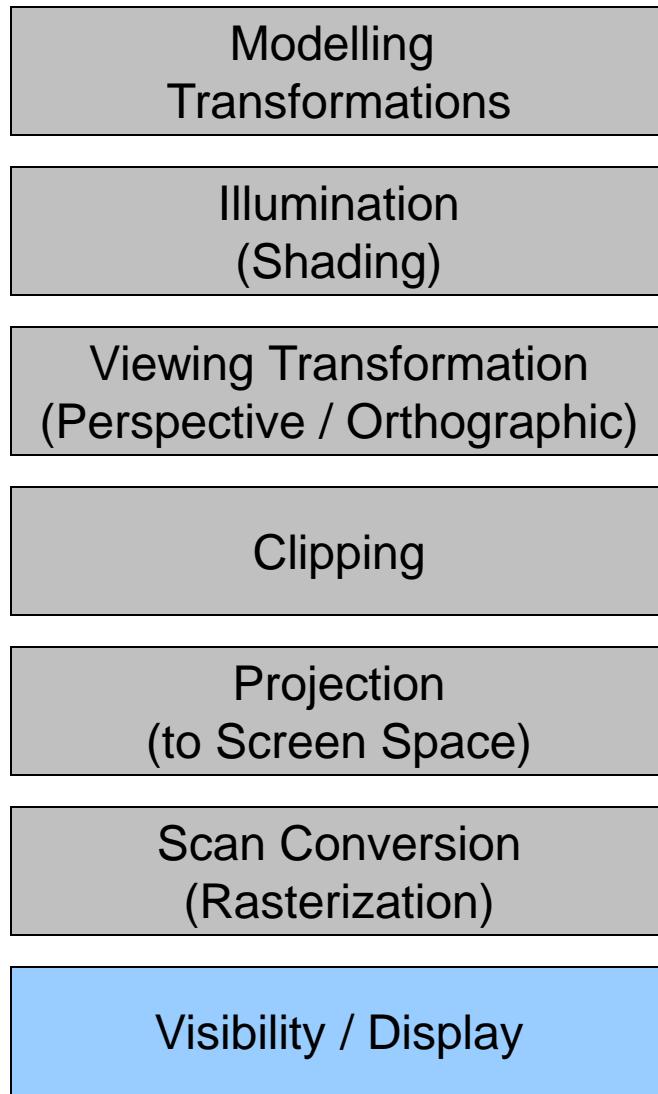
The Graphics Pipeline



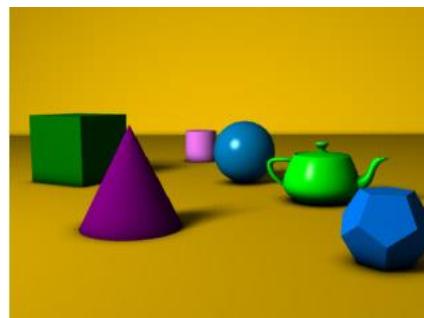
- Rasterizes objects into pixels
- Interpolate values inside objects (color, depth, etc.)



The Graphics Pipeline



- Handles occlusions and transparency blending
- Determines which objects are closest and therefore visible
- Depth buffer

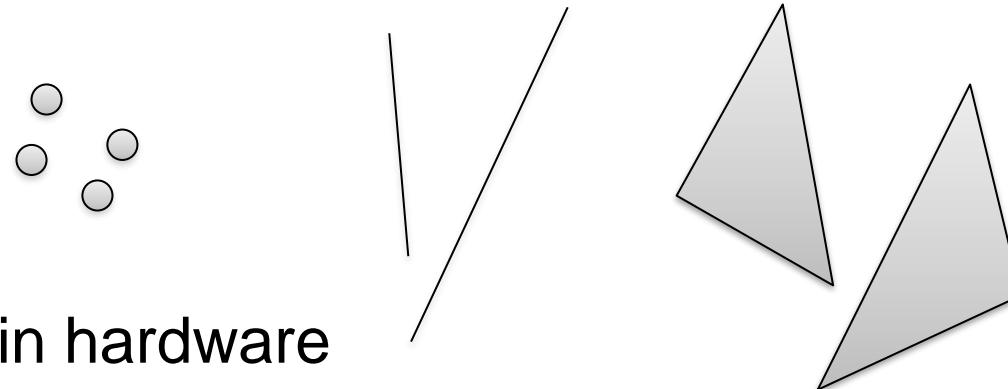


What do we want to do?

- Computer-generated imagery (CGI) in real-time
 - Very computationally demanding:
 - full HD at 60hz:
 $1920 \times 1080 \times 60\text{hz} = 124 \text{ Mpx/s}$
 - and that's just the output data
- use specialized hardware for immediate mode (real-time) graphics

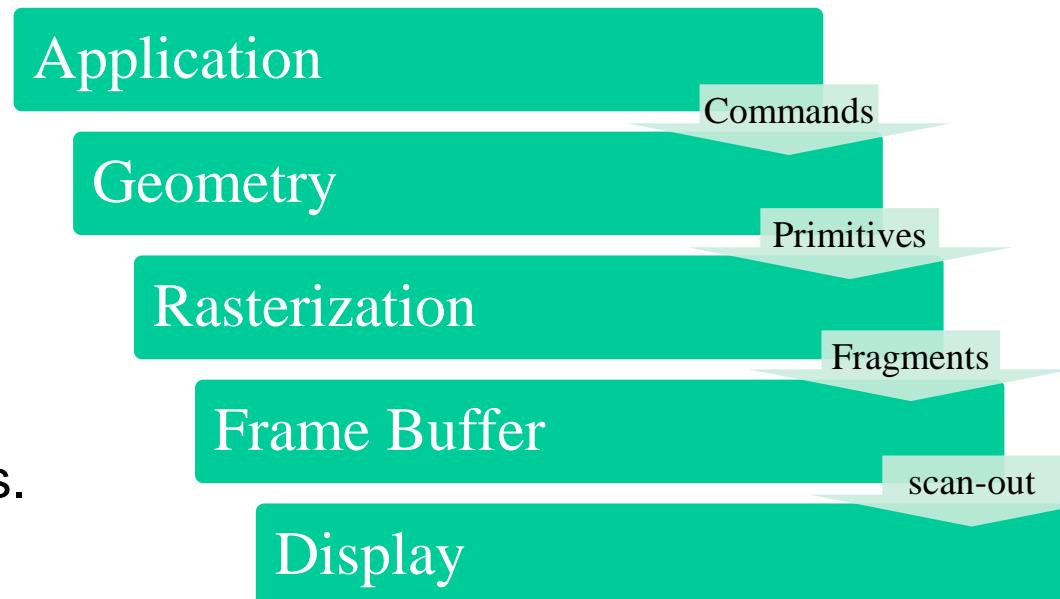
Solution

Most of real-time graphics is based on

- rasterization of graphic *primitives*
 - points
 - lines
 - triangles
 - ...
 - Implemented in hardware
 - *graphics processing unit* (GPU)
 - controlled through an API such as OpenGL
 - certain parts of graphics pipeline are programmable, e.g. using GLSL
- shaders
- 

The Graphics Pipeline different view

- High-level view:
- “Vertex”
 - a point in space defining geometry
- “Fragment”:
 - Sample produced during rasterization
 - Multiple fragments are *merged* into pixels.



Application Stage

- Generate database
 - Usually only once
 - Load from disk
 - Build acceleration structures (hierarchy, ...)
- Simulation
- Input event handlers
- Modify data structures
- Database traversal
- Utility functions

Application Stage

- Generate render area in OS
- Generate database
 - Usually only once
 - Load from disk
 - Build acceleration structures (hierarchy, ...)
- Simulation
- Input event handlers
- Modify data structures
- Database traversal
- Utility functions

Application Stage

```
solid TEATEST
facet normal  0.986544E+00  0.100166E+00  0.129220E+00
  outer loop
    vertex  0.167500E+02  0.505000E+02  0.000000E+00
    vertex  0.164599E+02  0.505000E+02  0.221480E+01
    vertex  0.166819E+02  0.483135E+02  0.221480E+01
  endloop
endfacet
facet normal  0.986495E+00  0.100374E+00  0.129434E+00
  outer loop
    vertex  0.166819E+02  0.483134E+02  0.221470E+01
    vertex  0.169653E+02  0.483840E+02  0.000000E+00
    vertex  0.167500E+02  0.505000E+02  0.000000E+00
  endloop
Endfacet
....
```

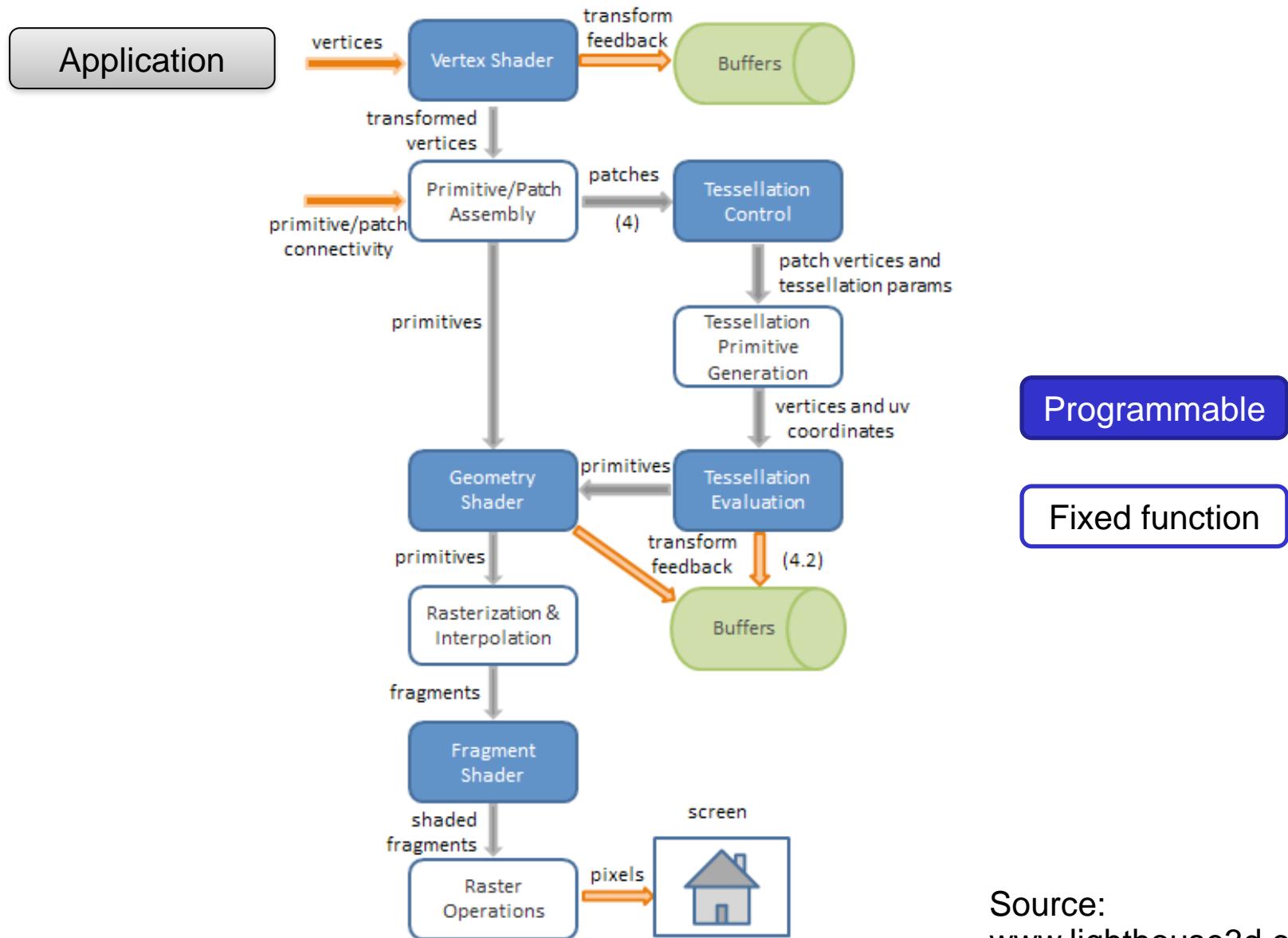


Application Stage

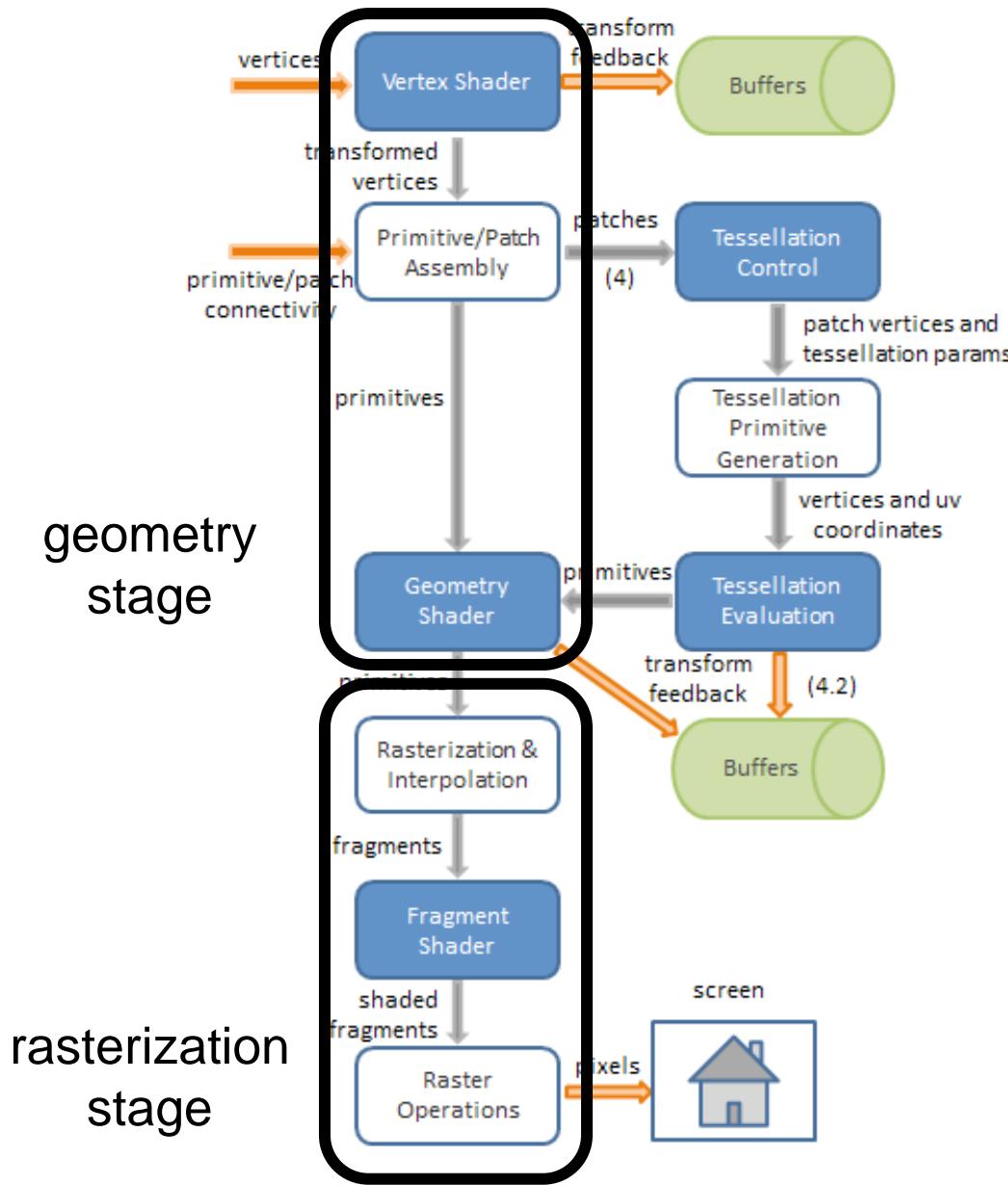
```
solid TEATEST
facet normal  0.986544E+00  0.100166E+00  0.129220E+00
outer loop
vertex  0.167500E+02  0.505000E+02  0.000000E+00
vertex  0.164599E+02  0.505000E+02  0.221480E+01
vertex  0.166819E+02  0.483135E+02  0.221480E+01
endloop
endfacet
facet normal  0.986495E+00  0.100374E+00  0.129434E+00
outer loop
vertex  0.166819E+02  0.483134E+02  0.221470E+01
vertex  0.169653E+02  0.483840E+02  0.000000E+00
vertex  0.167500E+02  0.505000E+02  0.000000E+00
endloop
Endfacet
....
```



The Graphics Pipeline: OpenGL 3.2 and later

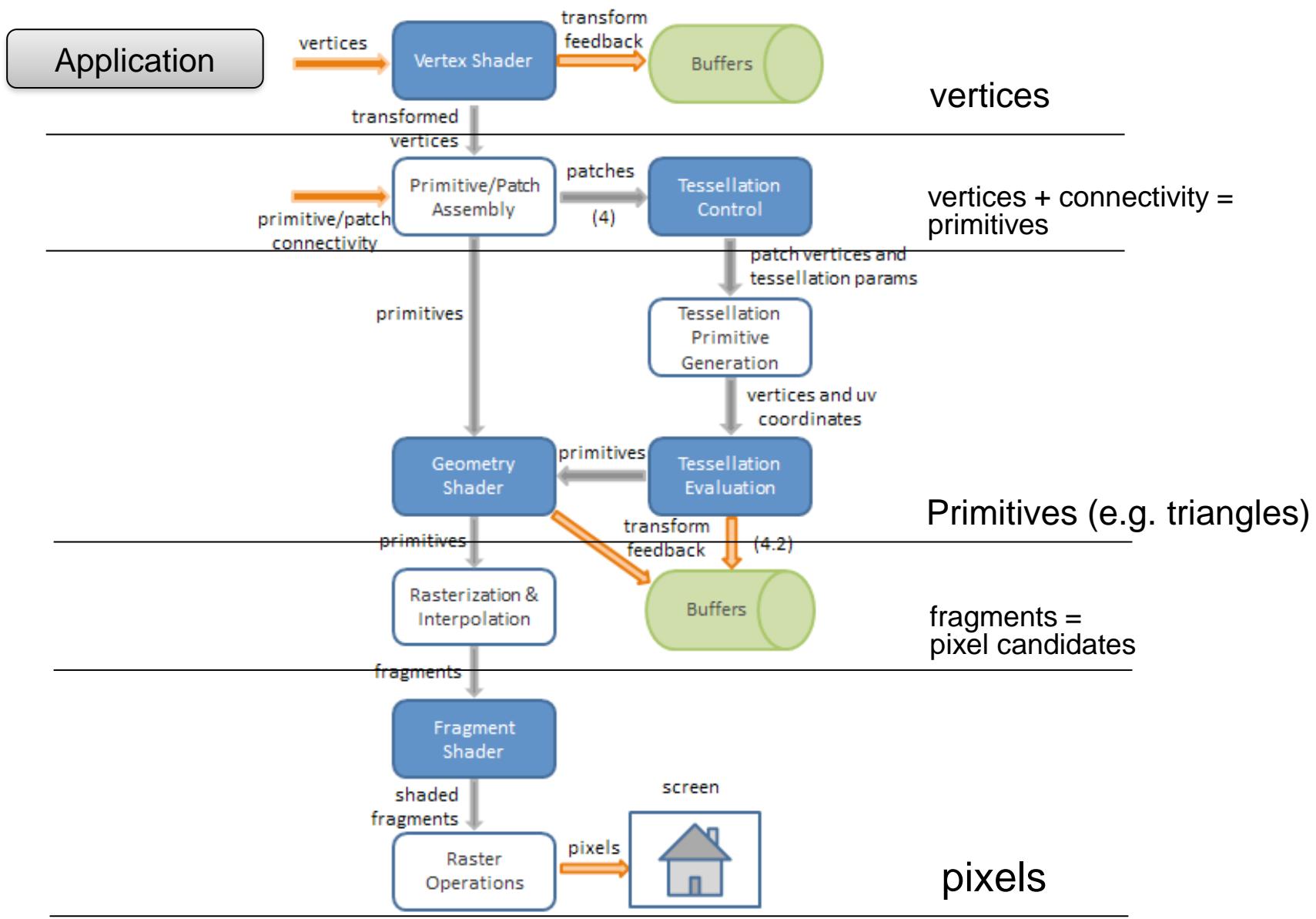


The Graphics Pipeline: OpenGL 3.2 and later

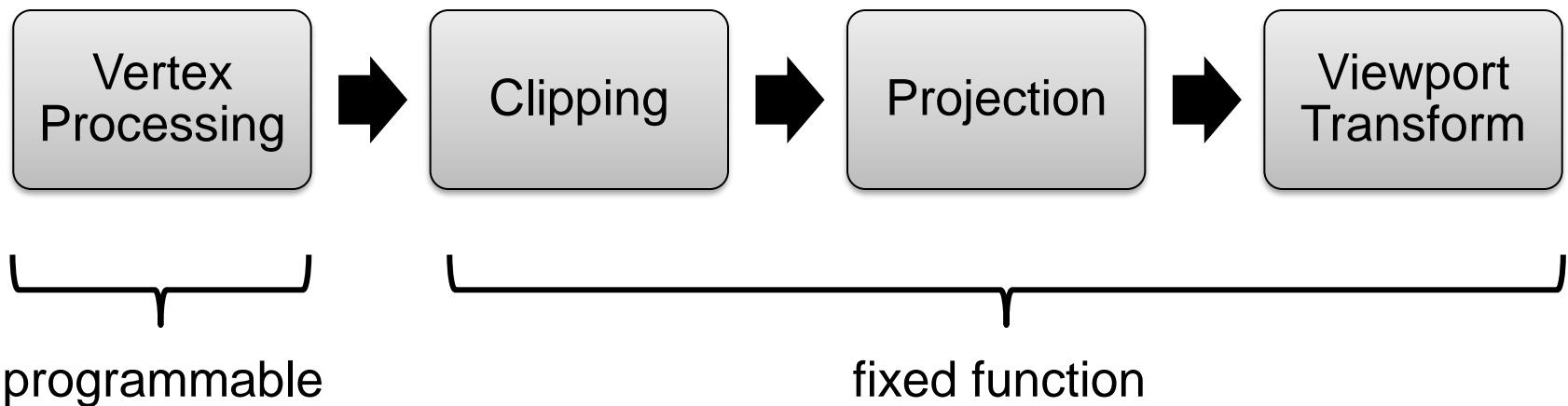


Source:
www.lighthouse3d.com

The Graphics Pipeline: OpenGL 3.2 and later

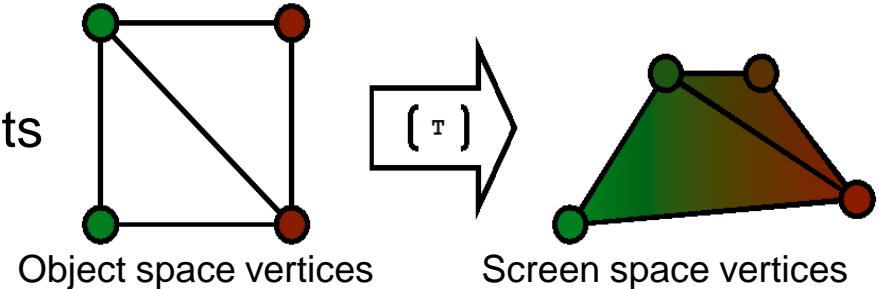


Geometry Stage



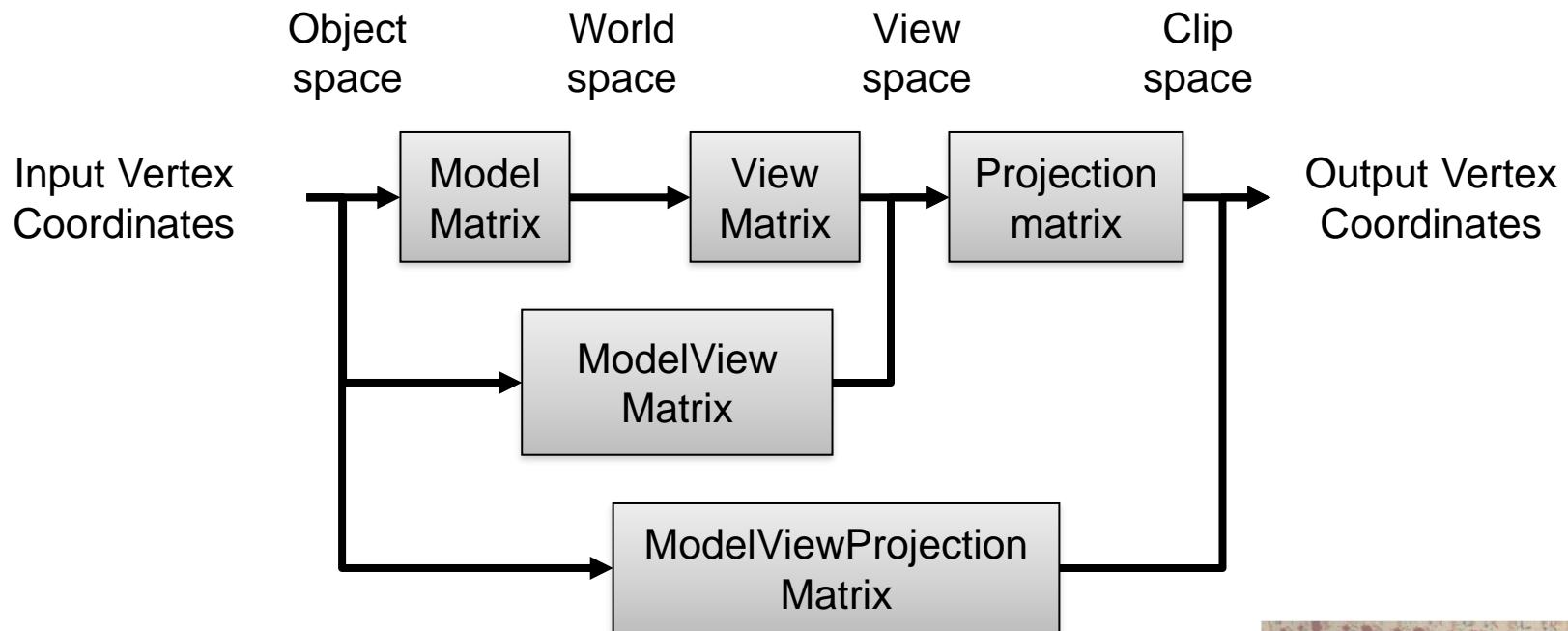
Geometry Stage: Vertex Processing

- The input vertex stream
 - composed of arbitrary vertex attributes (position, color, ...).
- is transformed into stream of vertices mapped onto the screen
 - composed of their clip space coordinates and additional user-defined attributes (color, texture coordinates, ...).
 - clip space: homogeneous coordinates
- by the **vertex shader**
 - GPU program that implements this mapping.
- Historically, “Shaders” were small programs performing lighting calculations, hence the name.



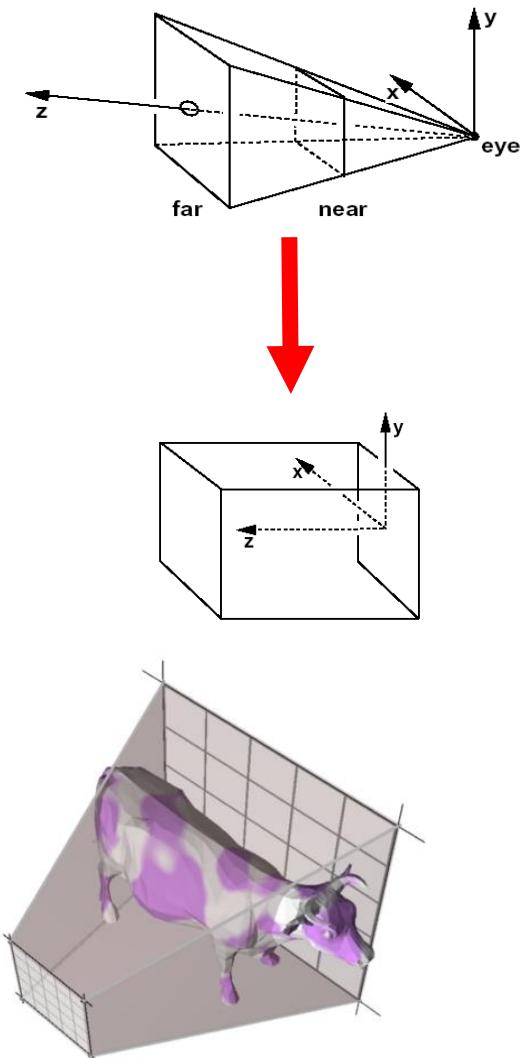
Geometry Stage: Vertex Post-Processing

- Uses a common transformation model in rasterization-based 3D graphics:



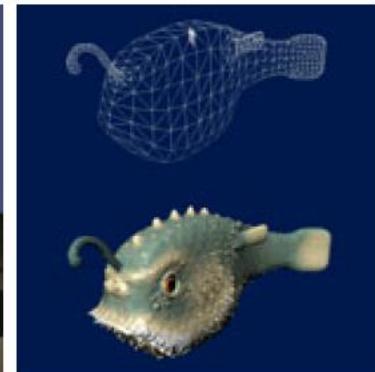
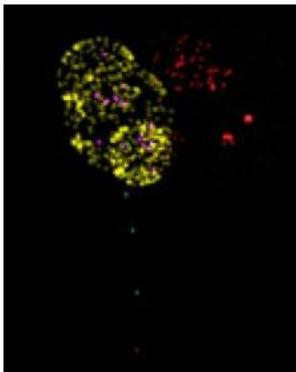
Geometry Stage: Vertex Post-Processing

- Clipping
 - Primitives not entirely in view are clipped to avoid projection errors
- Projection
 - Projects clip space coordinates to the image plane
 - Primitives in normalized device coordinates
- Viewport Transform:
 - Maps resolution-independent normalized device coordinates to a rectangular window in the frame buffer, the viewport.
 - Primitives in window (pixel) coordinates

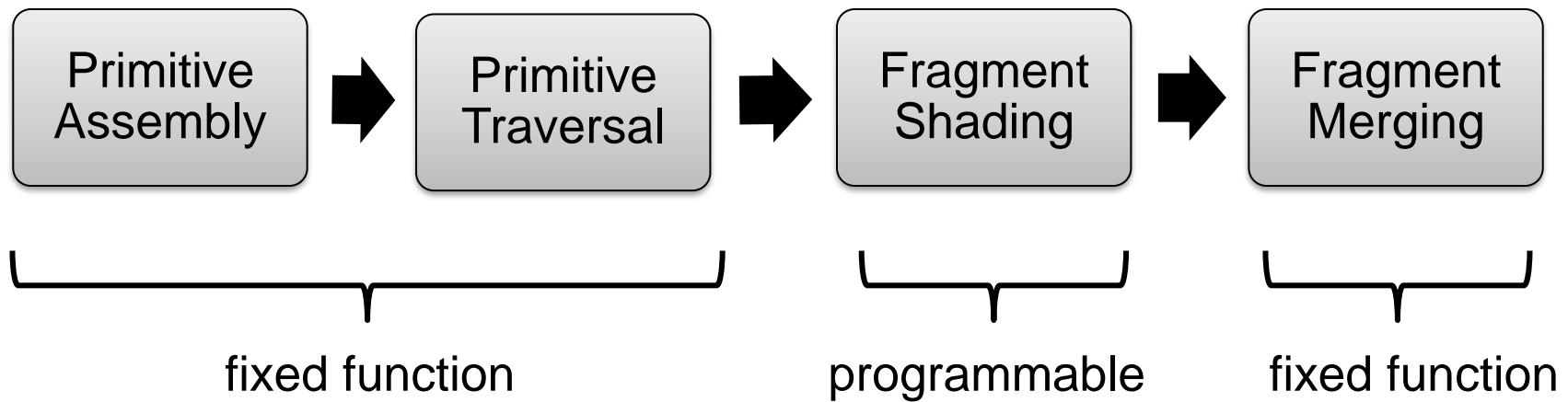


Geometry Shader

- Optional stage between vertex and fragment shader
- In contrast to the vertex shader, the geometry shader has full knowledge of the primitive it is working on
 - For each input primitive, the geometry shader has access to all the vertices that make up the primitive, including adjacency information.
- Can generate primitives dynamically
 - Procedural geometry, e.g. growing plants

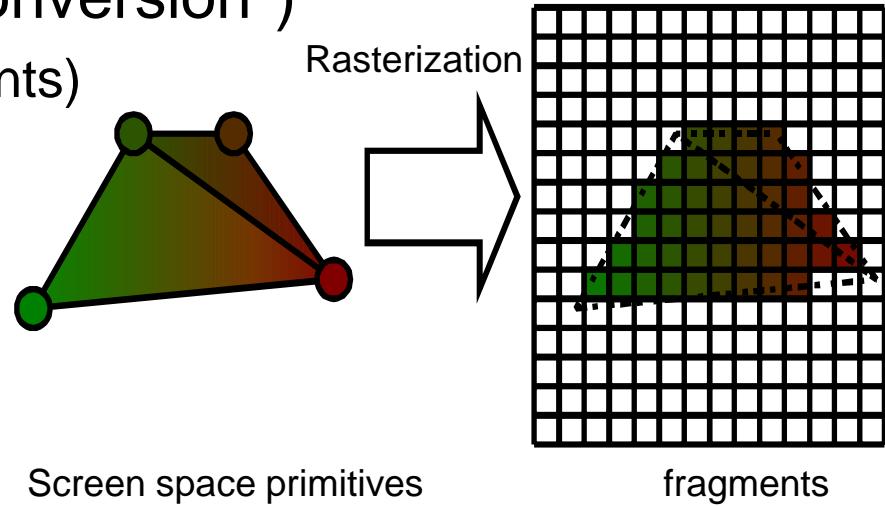


Rasterization Stage

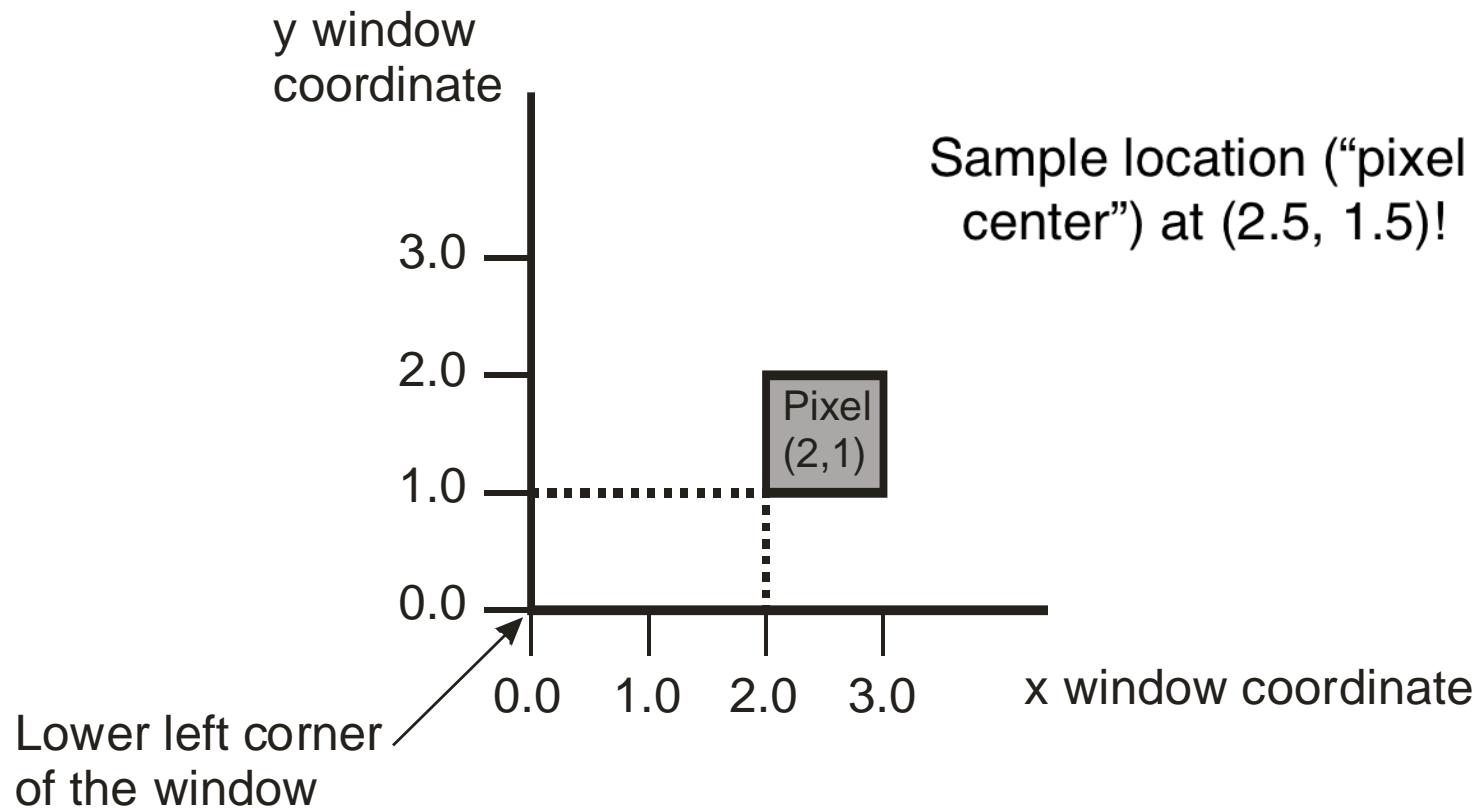


Rasterization Stage

- Primitive assembly
 - Backface culling
 - Setup primitive for traversal
- Primitive traversal (“scan conversion”)
 - Sampling (triangle → fragments)
 - Interpolation of vertex attributes (depth, color, ...)
- Fragment shading
 - Compute fragment colors
- Fragment merging
 - Compute pixel colors from fragments
 - Depth test, blending, ...

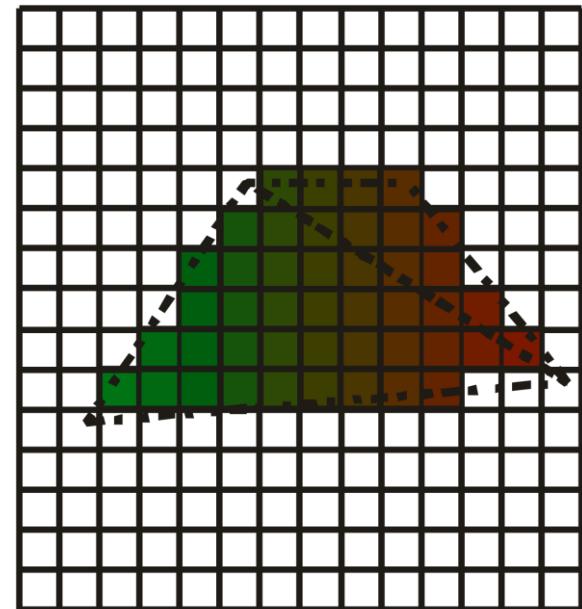


Rasterization – Coordinates



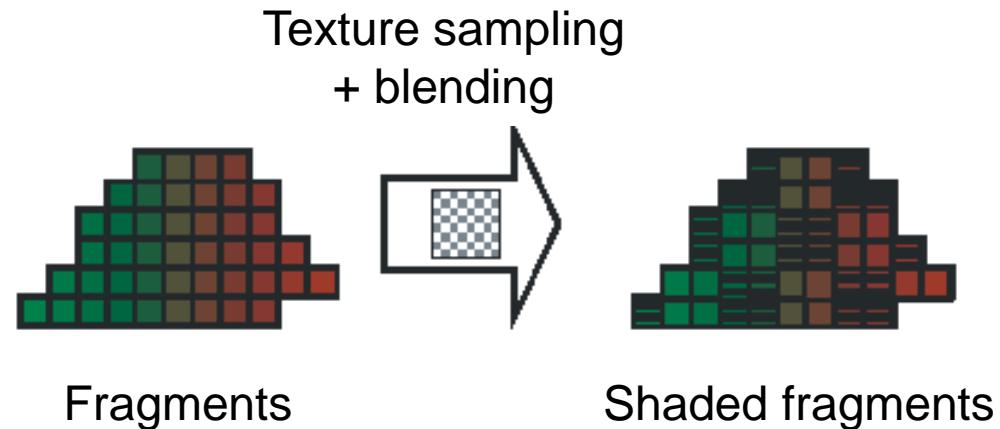
Rasterization – Rules

- Different rules for each primitive type
 - “fill convention”
- Non-ambiguous!
 - artifacts...
- Polygons:
 - Pixel center contained in polygon
 - Pixels on edge: only one is rasterized



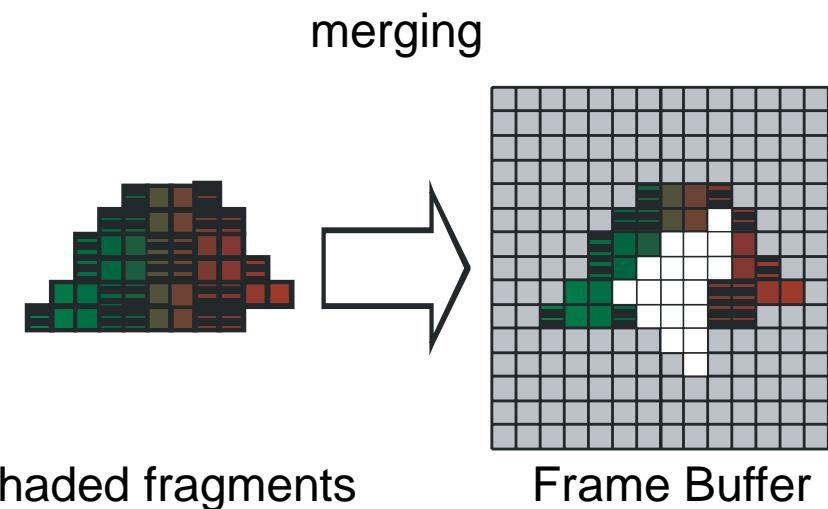
Fragment Shading

- “Fragment”:
 - Sample produced during rasterization
 - Multiple fragments are *merged* into pixels.
- Given the interpolated vertex attributes,
 - output by the Vertex Shader
- the *Fragment Shader* computes color values for each fragment.
 - Apply textures
 - Lighting calculations
 - ...

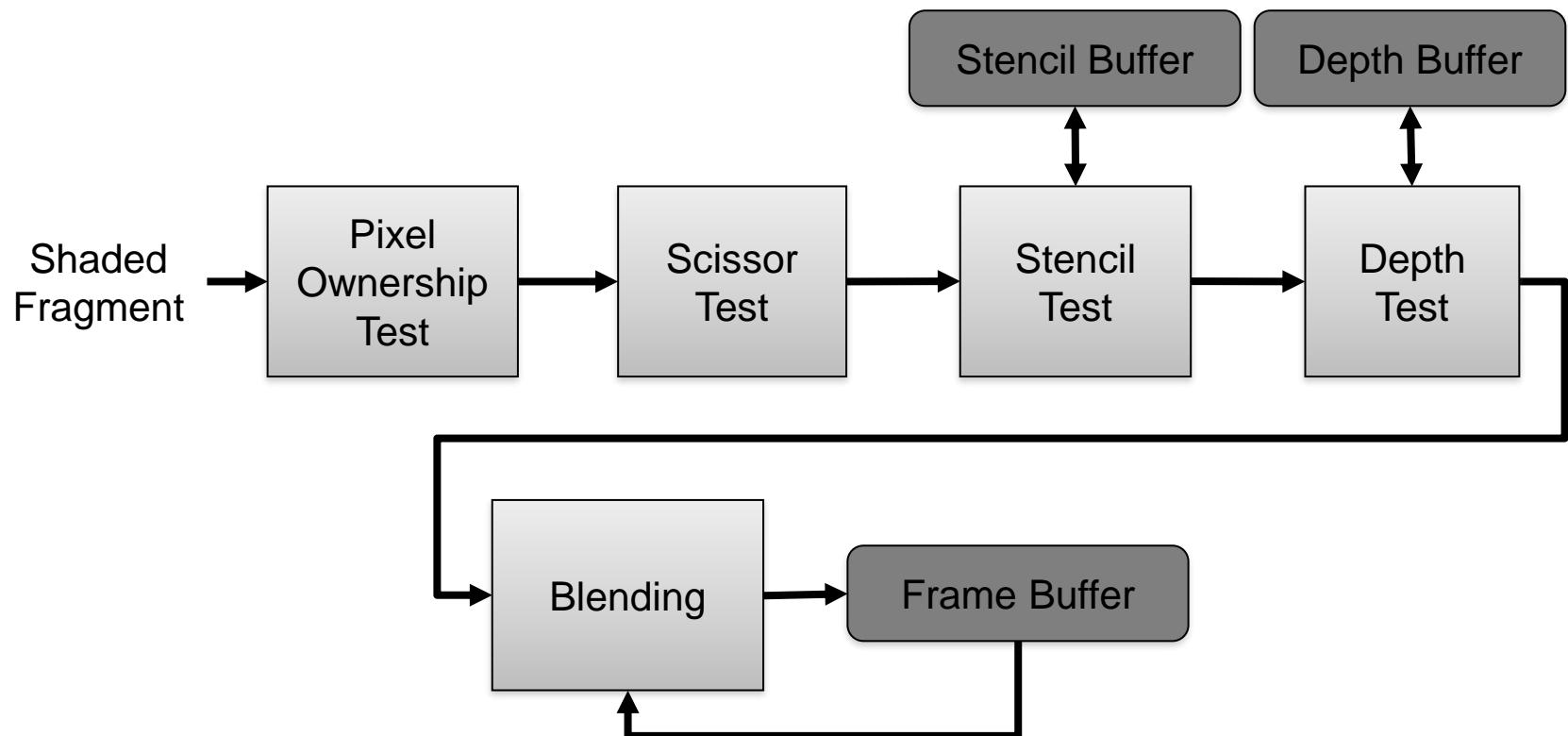


Fragment Merging

- Multiple primitives can cover the same pixel.
- Their Fragments need to be composed to form the final pixel values.
 - Blending
 - Resolve Visibility
 - Depth buffering

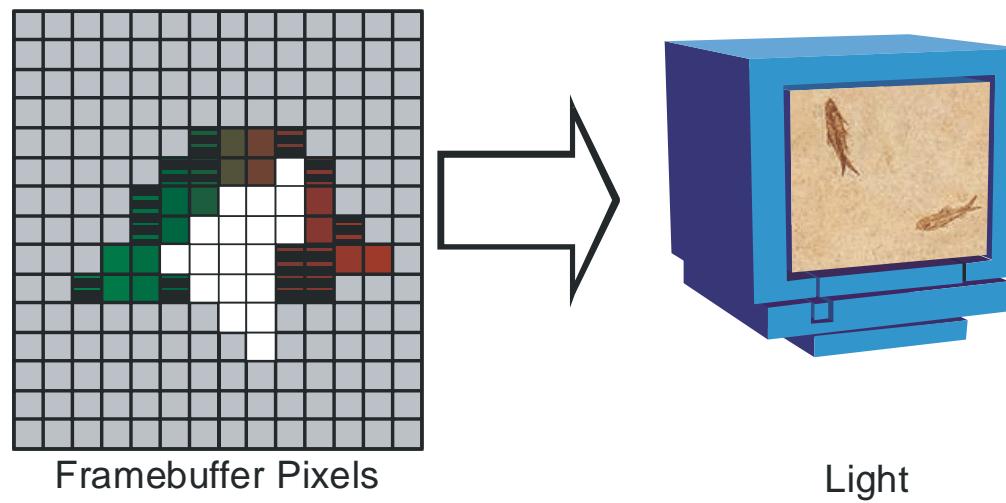


Fragment Merging



Display Stage

- Gamma correction
- Historically: Digital to Analog conversion
- Today: Digital scan-out, HDMI encryption, etc.



Display Format

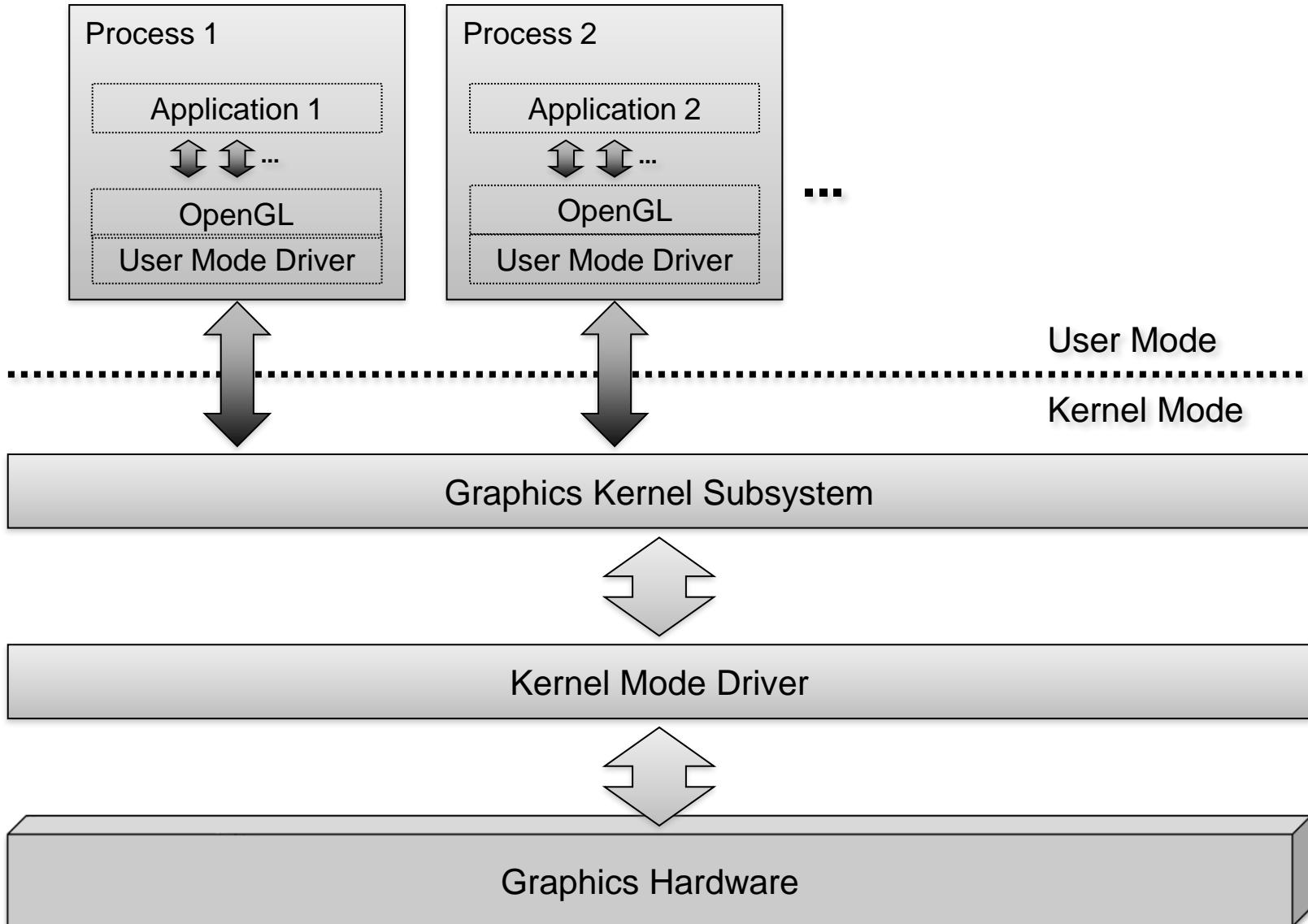
- Frame buffer pixel format:
 RGBA vs. index (obsolete)
- Bits: 16, 32, 64, 128 bit floating point, ...
- Double buffer vs. single buffer
- Quad-buffered stereo
- Overlays (extra bitplanes)
- Auxilliary buffers: alpha, stencil

Functionality vs. Frequency

- Geometry processing = per-vertex
 - Transformation and Lighting (T&L)
 - Historically floating point, complex operations
 - **Millions** of vertices per second
 - Today: Vertex Shader
- Fragment processing = per-fragment
 - Blending, texture combination
 - Historically fixed point and limited operations
 - **Billions** of fragments (“Gigapixel” per second)
 - Today: Fragment Shader

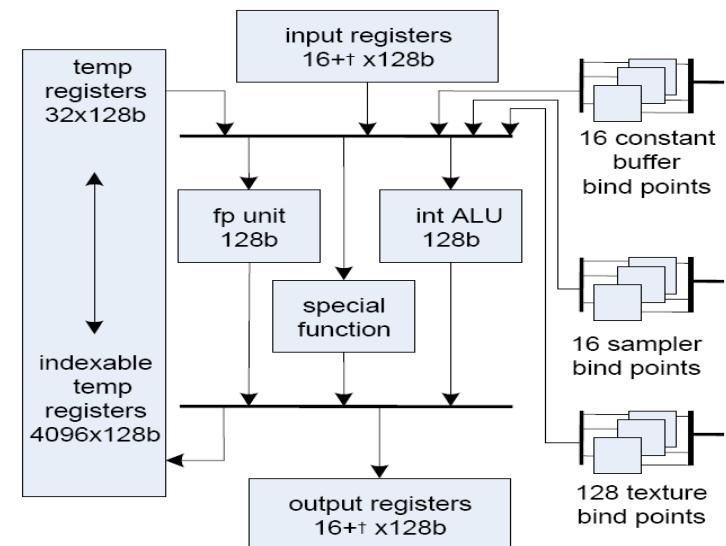
Architectural Overview

- Graphics Hardware is a shared resource
- User Mode Driver (UMD)
 - Prepares Command Buffers for the hardware
- Graphics Kernel Subsystem
 - Schedules access to the hardware
- Kernel Mode Driver (KMD)
 - Submits Command Buffers to the hardware



Unified Shader Model

- Since shader Model 4.0
- Unified Arithmetic and Logic Unit (ALU)
- Same instruction set and capabilities for all Shader types
- Dynamic load balancing geometry/fragment
- Floating point or integer everywhere
- IEEE-754 compliant
- Geometry Shader can write to memory
 - „Stream Output“
 - Enables multi-pass for geometry



Graphics APIs

Low-level 3D API

- OpenGL
 - Open Graphics Library (OpenGL) is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics.
- OpenGL ES
 - OpenGL for Embedded Systems is a subset of OpenGL
- DirectX, Direct3D
 - a graphics API for Microsoft Windows

Graphics APIs cont.

- **Vulcan**
 - OpenGL successor
 - targets high-performance realtime 3D graphics applications across all platforms
 - offers higher performance and lower CPU usage than older APIs.
- **Mantle**
 - low level graphics API by AMD. AMD will move to Vulcan
- **Metal**
 - low-level, low-overhead hardware-accelerated graphics and compute API by Apple (since IOS 8)

Graphics APIs cont.

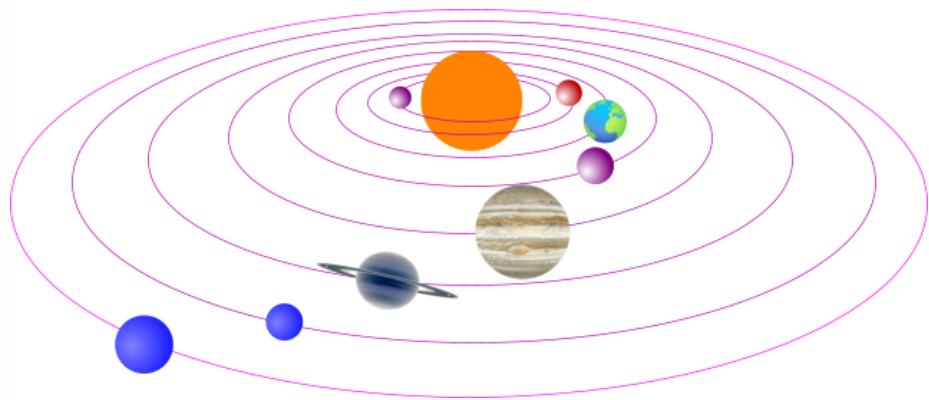
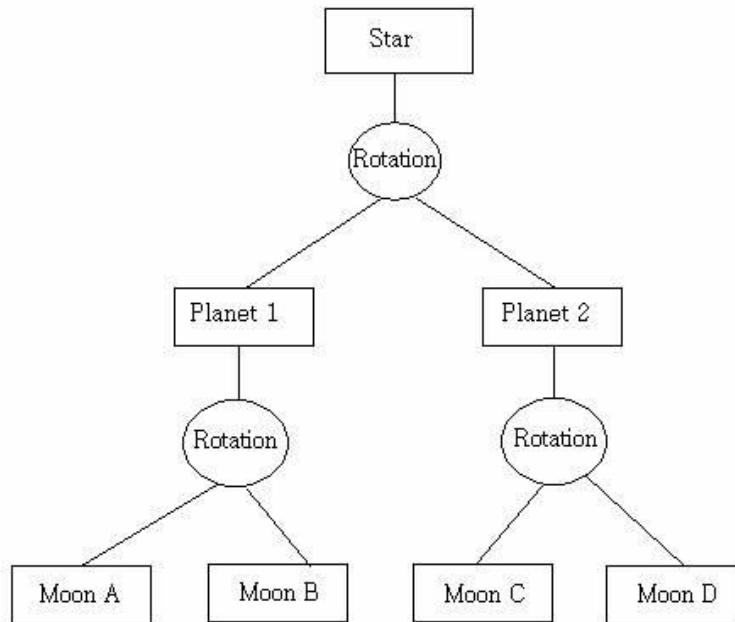
- RenderMan
 - Interface Specification by Pixar Animation Studios
 - open API
 - describe three-dimensional scenes and turn them into digital photorealistic images.
 - It includes the RenderMan Shading Language.
- WebGL
 - JavaScript API for rendering interactive 3D computer graphics and 2D graphics within any compatible web browser without the use of plug-ins.

Graphics APIs cont.

High-level 3D API – declarative models

a lot! Java, SceneGraphs, performer, Irrlicht, mobile SDKs

e.g. SceneGraph APIs (openSG, openInventor, etc.)



Interactive Computer Graphics: Lecture 5

Graphics APIs and Shading languages

Thanks to Markus Steinberger and
Dieter Schmalstieg, Dave Shreiner, Ed
Angel, Vicki Shreiner

Graphics APIs

Low-level 3D API

- OpenGL
- OpenGL ES
- DirectX, Direct3D
- Vulkan
- Mantle
- WebGL
- ...

Graphics APIs

Low-level 3D API

- **OpenGL**
- OpenGL ES
- DirectX, Direct3D
- Vulcan
- Mantle
- WebGL
- ...

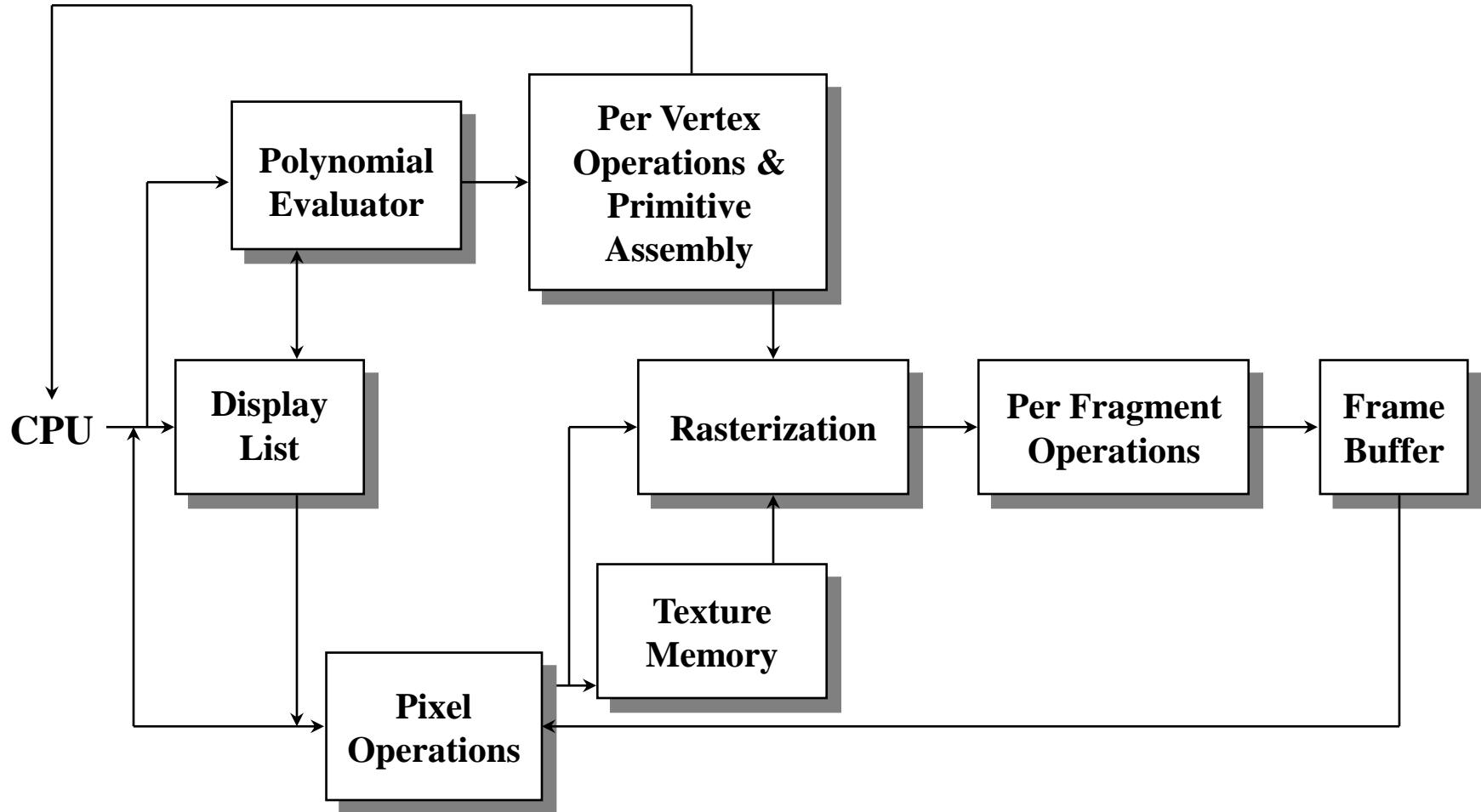
What is OpenGL?

- a low-level graphics API specification
 - not a library!
 - The interface is platform independent,
 - but the implementation is platform dependent.
 - Defines
 - an abstract rendering device.
 - a set of functions to operate the device.
 - “immediate mode” API
 - drawing commands
 - no concept of permanent objects

What is OpenGL?

- Platform provides OpenGL *implementation*.
 - Part of the graphics driver, or
 - runtime library built on top of the driver
- Initialization through platform specific API
 - WGL (Windows)
 - GLX (Unix/Linux)
 - EGL (mobile devices)
 - ...
- State machine for high efficiency!

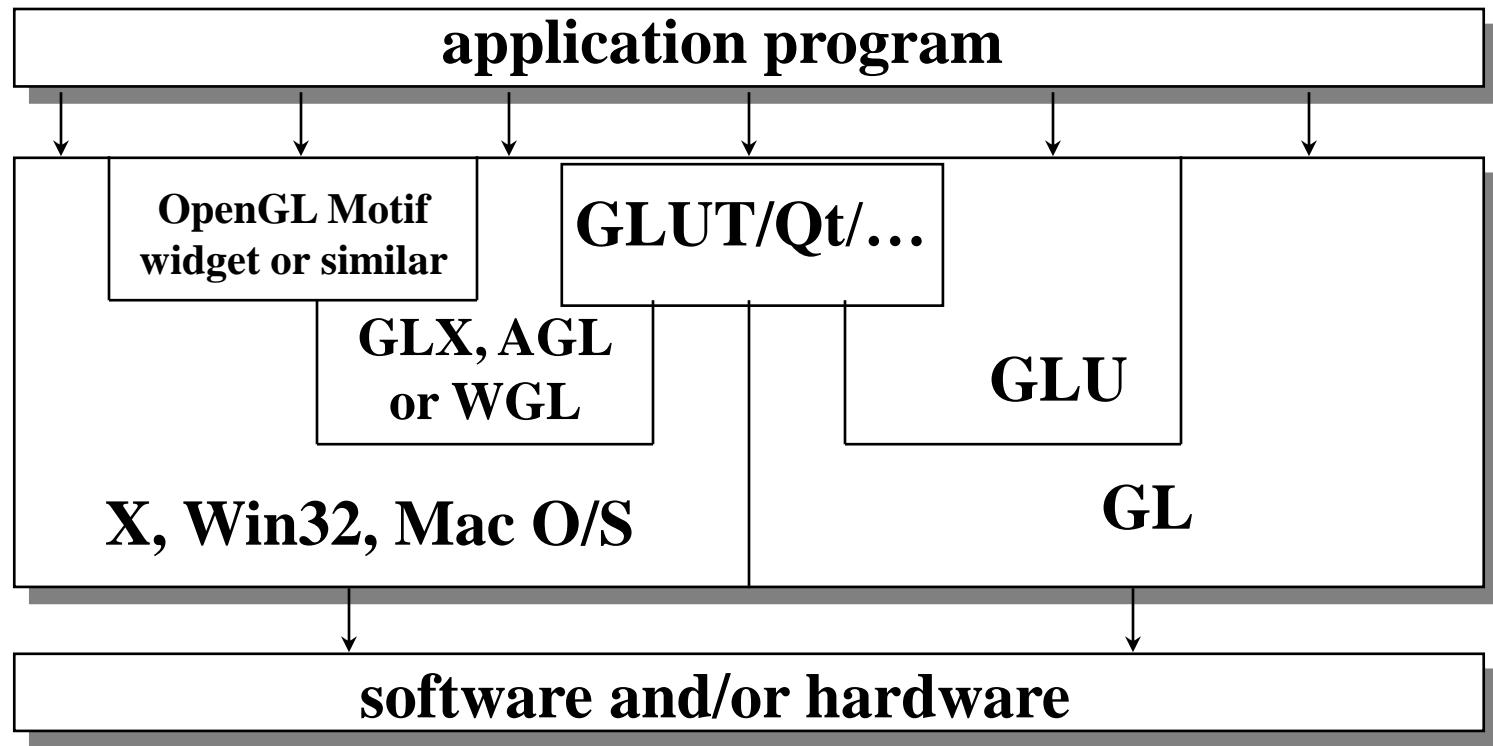
OpenGL Architecture



writing OpenGL programs

- Render window, i.e., context providing libraries (glut, Qt, browser SDKs etc.)
- setup and initialization functions
 - viewport
 - model transformation
 - file I/O (shader, textures, etc.)
- frame generation (update/rendering) functions
 - define what happens in every frame

OpenGL and Related APIs



Preliminaries

- Headers Files
 - `#include <GL/gl.h>`
 - `#include <GL/glu.h>`
 - `#include <GL/glut.h>`
- Or in case of a Qt application
 - `#include <QtOpenGL>`
- <https://www.opengl.org/resources/libraries/glut/spec3/spec3.html>
- Enumerated Types
 - OpenGL defines numerous types for compatibility
 - `GLfloat`, `GLint`, `GLenum`, etc.

Preliminaries

- Easier with Qt but more overhead
- Headers Files
 - `#include <QOpenGLWidget>`
 - `#include <QOpenGLFunctions>`
 - ...
- <http://doc.qt.io/qt-5/qtopengl-index.html>

OpenGL Basic Concepts

- Context
- Resources
- Object Model
 - Objects
 - Object Names
 - Bind Points (Targets)

Context

- Represents an instance of OpenGL
- A process can have multiple contexts
 - These can share resources
- A context can be *current* for a given thread
 - one to one mapping
 - only one current context per thread
 - context only current in one thread at the same time
 - OpenGL operations work on the current context

Resources

- Act as
 - sources of input
 - sinks for output
- Examples:
 - buffers
 - images
 - state objects
 - ...

Resources

- Buffer objects
 - linear chunks of memory
- Texture images
 - 1D, 2D, or 3D arrays of *texels*
 - Can be used as input for *texture sampling*

Object Model

- OpenGL is object oriented
 - but in its own, strange way
- Object instances are identified by a *name*
 - basically just an unsigned integer handle
- Commands work on *targets*
 - Each target has an object currently *bound* to the target
 - That's the one commands will work with
- Object oriented, you said?
 - target \Leftrightarrow type
 - commands \Leftrightarrow methods

Object Model

- By binding a name to a target
 - the object it identifies becomes current for that target
 - “latched state”
 - change in OpenGL 4.5 (`EXT_direct_state_access`)
 - An object is created when a name is first bound.
- Notable exceptions: Shader Objects, Program Objects
 - Some commands work directly on object names.

Buffer Objects

- store an array of unformatted memory allocated by the OpenGL context (aka: the GPU)
- regular OpenGL objects
- can be used to store vertex data, pixel data retrieved from images or the framebuffer, and a variety of other things
- to set up its internal state, you must bind it to the context.

```
void glBindBuffer(enum target, uint bufferName)
```

- Immutable

```
void glBufferStorage(...);
```

- or mutable depending on initialisation

```
void glBufferData(...)
```

Example: Buffer Object

```
GLuint my_buffer;

// request an unused buffer object name
glGenBuffers(1, &my_buffer);

// bind name as GL_ARRAY_BUFFER
// bound for the first time ⇒ creates
glBindBuffer(GL_ARRAY_BUFFER, my_buffer);

// put some data into my_buffer
glBufferStorage(GL_ARRAY_BUFFER, ...);

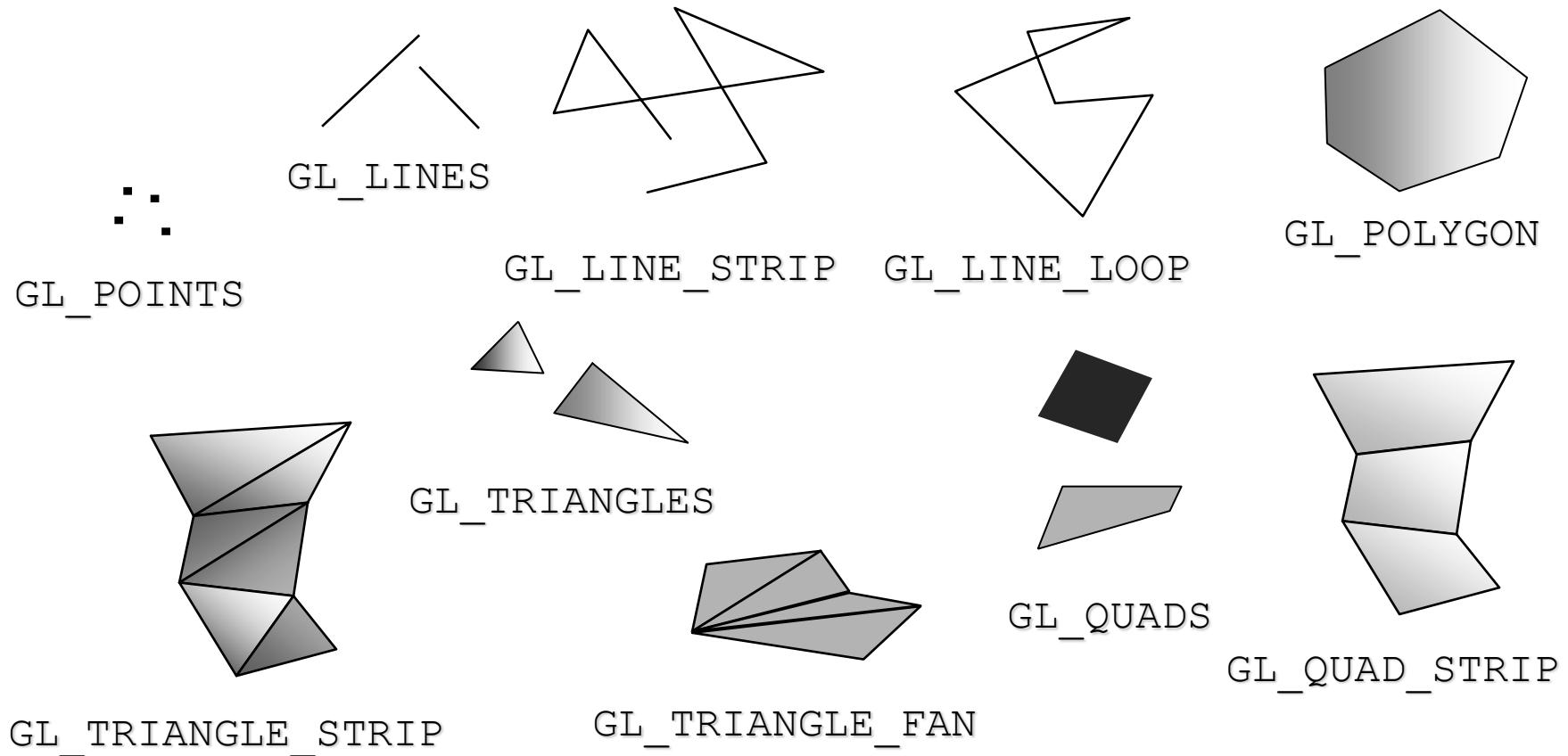
// “unbind” buffer
glBindBuffer(GL_ARRAY_BUFFER, 0);

// probably do something else...
glBindBuffer(GL_ARRAY_BUFFER, my_buffer);
// use my_buffer...

glDrawArrays(GL_TRIANGLES, 0, 33);
// draw content example (type, startIdx, numer of elements)

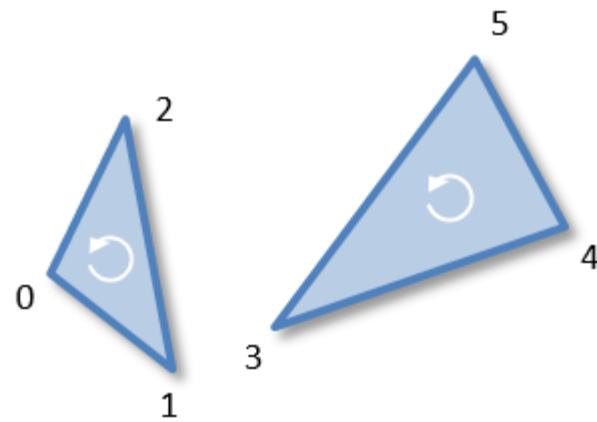
// delete buffer object, free resources, release buffer object name
glDeleteBuffers(1, &my_buffer);
```

Primitive types

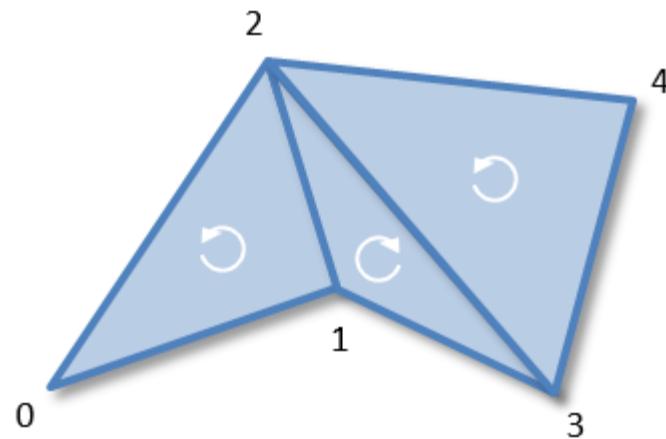


Primitive types

- triangle vertex orientations in OpenGL



GL_TRIANGLES



GL_TRIANGLE_STRIP

Draw Call

- After pipeline is configured:
 - issue *draw call* to actually draw something

e. g.:

```
glBegin(GL_TRIANGLE_STRIP);
glColor3f(0.0, 1.0, 0.0); ← Color “state”
glVertex3f(1.0, 0.0, 0.0);
...
glEnd();
```

primitive type

vertex index

Buffer Objects -- drawing

- For continuous groups of vertices

```
glDrawArrays(GL_TRIANGLES, 0, num_vertices);
```

- usually invoked in display callback
- initiates vertex shader

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)`

writing (old) OpenGL programs

- pseudo example

```
#include <whateverYouNeed.h>

main() {
    InitializeAWindowPlease();

    glClearColor (0.0, 0.0, 0.0, 0.0);
    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);

    registerDisplayCallback(
        UpdateTheWindowAndCheckForEvents())
}

UpdateTheWindowAndCheckForEvents() {
    glBegin(GL_POLYGON);
        glVertex3f (0.25, 0.25, 0.0);
        glVertex3f (0.75, 0.25, 0.0);
        glVertex3f (0.75, 0.75, 0.0);
        glVertex3f (0.25, 0.75, 0.0);
    glEnd();
}
```

Matrix stack (old OpenGL)

- There used to be a stack of matrices for each of the matrix modes.
- The current transformation matrix in any mode is the matrix on the top of the stack for that mode.
- **glPushMatrix** pushes the current matrix stack down by one, duplicating the current matrix.
- **glPopMatrix** pops the current matrix stack, replacing the current matrix with the one below it on the stack.
- Initially, each of the stacks contains one matrix, an identity matrix.
- used to ‘save’ transformation state

Example Textures

```
glEnable(GL_TEXTURE_2D);
glActiveTexture(GL_TEXTURE0);
textureImage = readPPM("pebbles_texture.ppm");
 glGenTextures(1, &tex);
 glBindTexture(GL_TEXTURE_2D, tex);
 glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, textureImage->x,
 textureImage->y, 0, GL_RGB, GL_UNSIGNED_BYTE, textureImage-
>data);

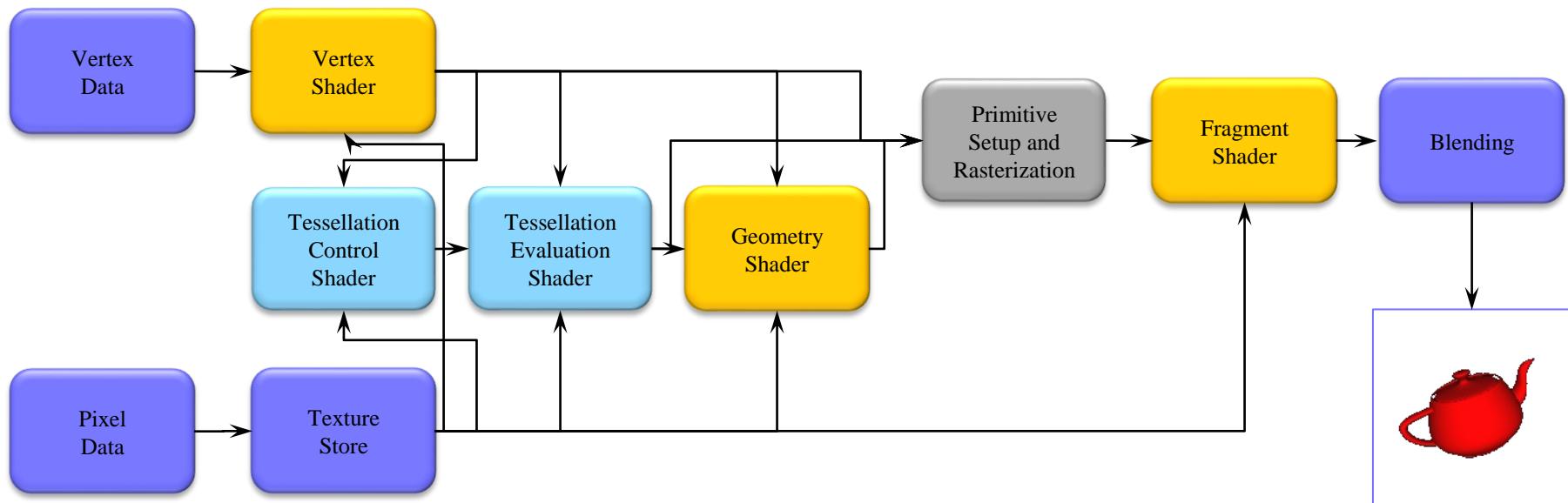
 glBindTexture(GL_TEXTURE_2D, 0);
 ...
 glBindTexture(GL_TEXTURE_2D, tex);
 glutSolidTeapot(0.5);
 glBindTexture(GL_TEXTURE_2D, 0);
```

OpenGL 4

- Enforces a new way to program with OpenGL
 - Allows more efficient use of GPU resources
- In contrast to “classic” graphics pipelines, modern OpenGL doesn’t support
 - Fixed-function graphics operations
 - Lighting, transformations, etc.
- All applications must use shaders and buffers for their graphics processing

OpenGL 4

- OpenGL 4.1 (released July 25th, 2010) included additional shading stages – *tessellation-control and tessellation-evaluation shaders*
- Latest version is 4.3



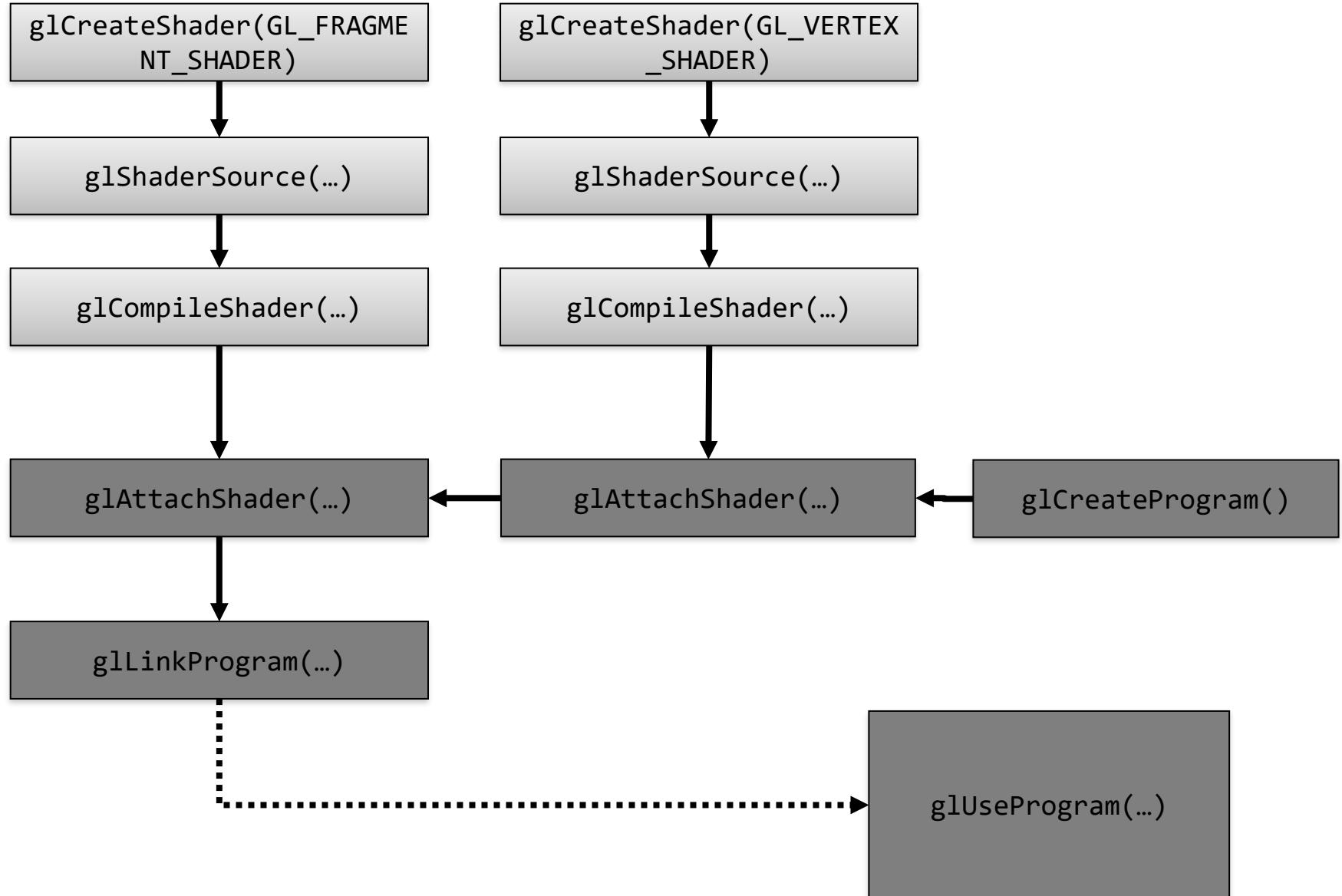
OpenGL 4

- Modern OpenGL programs essentially do the following steps:
 1. Create shader programs
 2. Create buffer objects and load data into them
 3. “Connect” data locations with shader variables
 4. Render

Shaders

- Shader Objects
 - parts of a pipeline (Vertex Shader, Fragment Shader, etc.)
 - compiled during runtime from GLSL code
 - OpenGL Shading Language
 - C-like syntax
- Program Object
 - a whole pipeline
 - Shader objects linked together during runtime
- OpenGL shader language: GLSL

Shaders



GLSL Data Types

Scalar types: `float`, `int`, `bool`

Vector types: `vec2`, `vec3`, `vec4`
`ivec2`, `ivec3`, `ivec4`
`bvec2`, `bvec3`, `bvec4`

Matrix types: `mat2`, `mat3`, `mat4`

Texture sampling: `sampler1D`, `sampler2D`, `sampler3D`,
`samplerCube`

C++ style constructors: `vec3 a = vec3(1.0, 2.0, 3.0);`

Operators

- Standard C/C++ arithmetic and logic operators
- Operators overloaded for matrix and vector operations

```
mat4 m;  
vec4 a, b, c;  
  
b = a*m;  
c = m*a;
```

Components and Swizzling

For vectors can use [], xyzw, rgba or stpq

Example:

```
vec3 v;
```

v[1], v.y, v.g, v.t all refer to the same element

Swizzling:

```
vec3 a, b;
```

```
a.xy = b.yx;
```

Qualifiers

- **in, out**
 - Copy vertex attributes and other variables to/from shaders
 - `in vec2 tex_coord;`
 - `out vec4 color;`
- Uniform: variable from application
 - `uniform float time;`
 - `uniform vec4 rotation;`

Flow Control

- **if**
- **if else**
- expression ? true-expression : false-expression
- **while, do while**
- **for**

Functions

- Built in
 - Arithmetic: `sqrt`, `power`, `abs`
 - Trigonometric: `sin`, `asin`
 - Graphical: `length`, `reflect`
- User defined

Built-in Variables

- **gl_Position**: output position from vertex shader
- **gl_FragColor**: output color from fragment shader
 - Only for ES, WebGL and older versions of GLSL
 - Present version use an out variable

Anatomy of a GLSL Shader

```
1 #version 400
2
3 uniform mat4 some_uniform; ← Set by application
4
5 layout(location = 0) in vec3 some_input; ← Optional flexible
6 layout(location = 1) in vec4 another_input; register configuration
7
8 out vec4 some_output; ← between shaders
9 void main()
10 {
11
12 }
```

Set by application
(configuration values, e.g.
ModelViewProjection Matrix)

Optional flexible
register
configuration
between shaders

Output definition for
next shader stage

Vertex Shader

- Processes each vertex
- Input: vertex attributes
- Output: vertex attributes
 - gl_Position

Rasterizer

- Fixed-function
- Rasterizes primitives
- Input: primitives
 - vertex attributes
- Output: fragments
 - interpolated vertex attributes

Fragment Shader

- Processes each fragment
- Input: interpolated vertex attributes
- Output: fragment color

Fragment Shader

- Interface to fixed-function parts of the pipeline (shader model > 4 – OpenGL4 requires to define these).
 - e.g. Vertex Shader:
 - `in int gl_VertexID;`
 - `out vec4 gl_Position;`
 - e.g. Fragment Shader:
 - `in vec4 gl_FragCoord;`
 - `out float gl_FragDepth;`

Example: Vertex Shader

```
#version 400

uniform mat4 mvMatrix;
uniform mat4 pMatrix;
uniform mat3 normalMatrix; //mv matrix without translation

uniform vec4 lightPosition_camSpace; //light Position in camera space

in vec4 vertex_worldSpace;
in vec3 normal_worldSpace;
in vec2 textureCoordinate_input;

out data
{
    vec4 position_camSpace;
    vec3 normal_camSpace;
    vec2 textureCoordinate;
    vec4 color;
}vertexIn;

//Vertex shader compute the vectors per vertex
void main(void)
{
    //Put the vertex in the correct coordinate system by applying the model view matrix
    vec4 vertex_camSpace = mvMatrix*vertex_worldSpace;
    vertexIn.position_camSpace = vertex_camSpace;
    //Apply the model-view transformation to the normal (only rotation, no translation)
    //Normals put in the camera space
    vertexIn.normal_camSpace = normalize(normalMatrix*normal_worldSpace);
    //Color chosen as red
    vertexIn.color = vec4(1.0, 0.0, 0.0, 1.0);
    //Texture coordinate
    vertexIn.textureCoordinate = textureCoordinate_input;
    gl_Position = pMatrix * vertex_camSpace;
}
```

Example: Fragment Shader

```
#version 400

uniform vec4 ambient;
uniform vec4 diffuse;
uniform vec4 specular;
uniform float shininess;

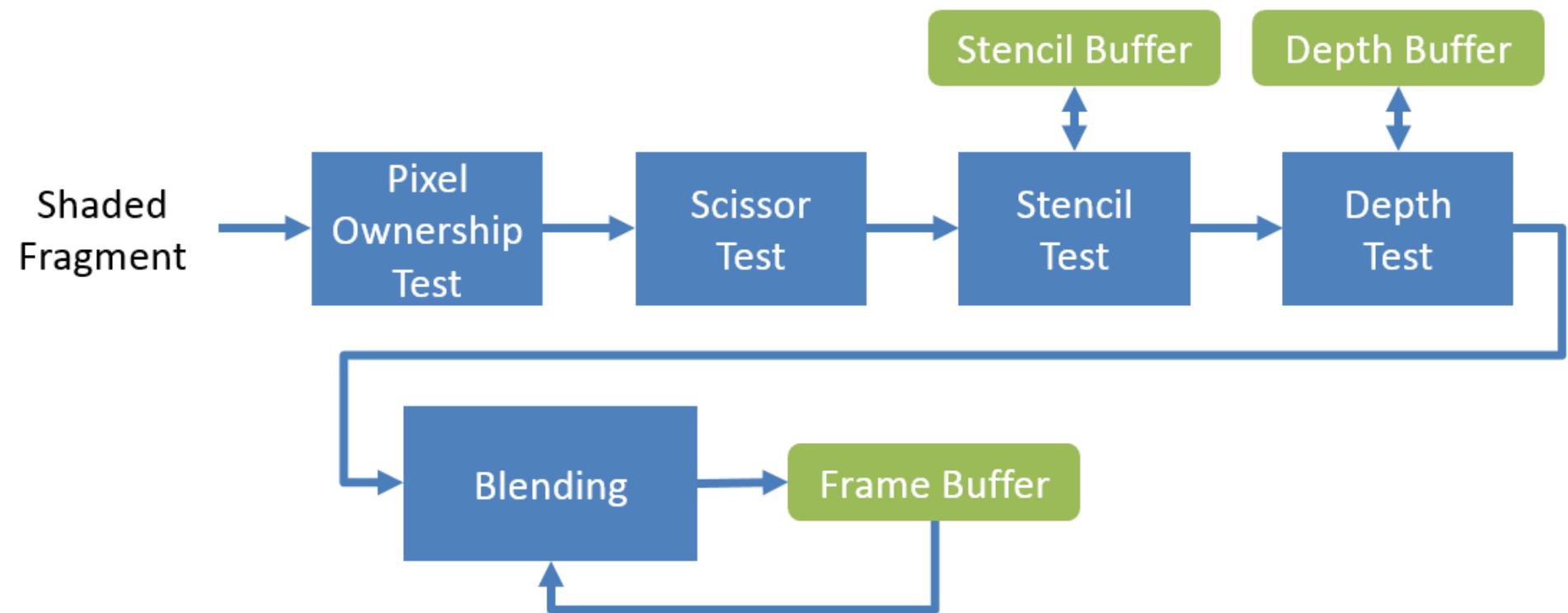
uniform vec4 lightPosition_camSpace; //light Position in camera space

in fragmentData
{
    vec4 position_camSpace;
    vec3 normal_camSpace;
    vec2 textureCoordinate;
    vec4 color;
} frag;

out vec4 fragColor;

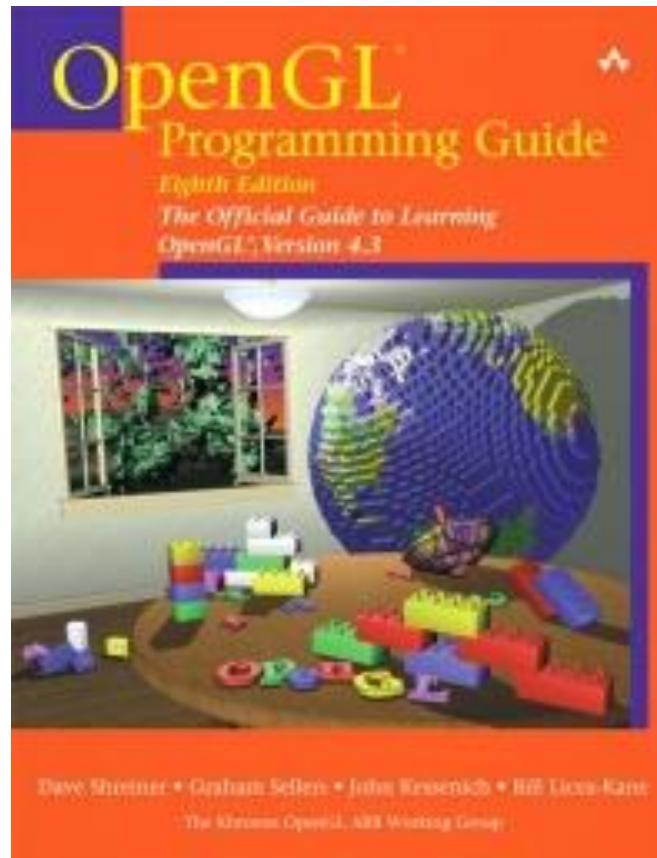
//Fragment shader computes the final color
void main(void)
{
    fragColor = frag.color;
}
```

Fragment Merging



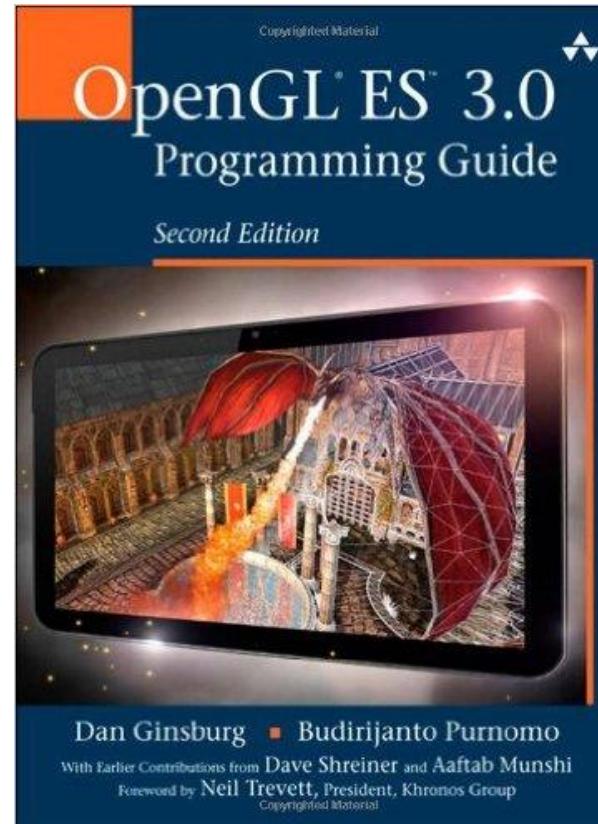
Please read the OpenGL Programming Guide

- free full online version:
<http://www.glprogramming.com/red>



OpenGL ES (Embedded Systems)

- OpenGL is just too big for embedded systems like mobile devices
- compact API, purely shader-based

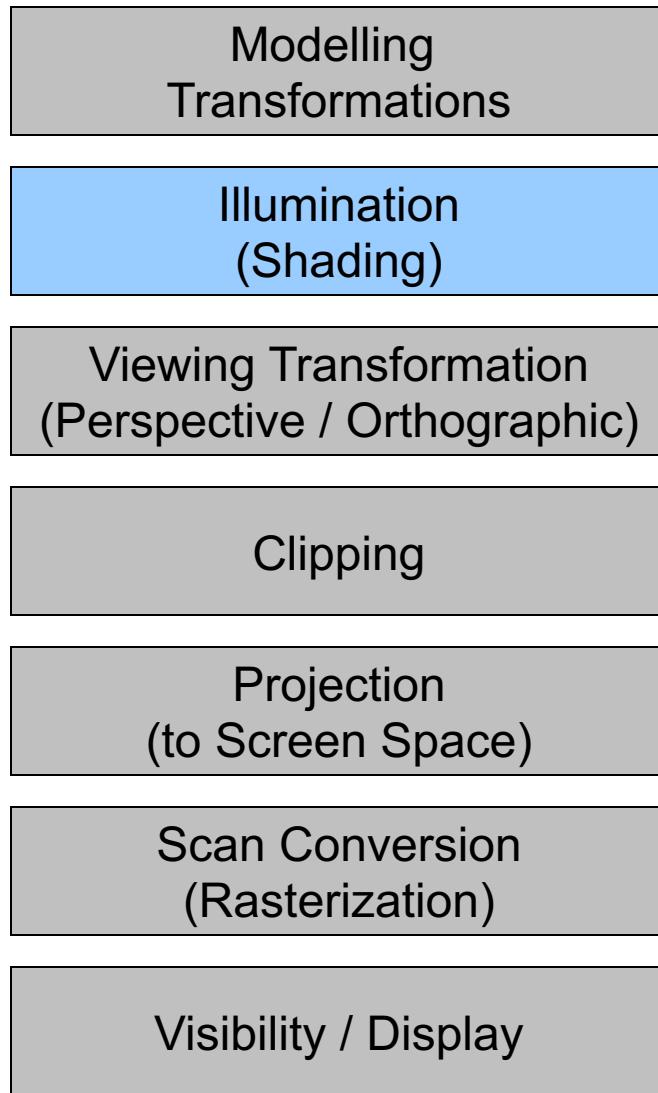


Interactive Computer Graphics: Lecture 6

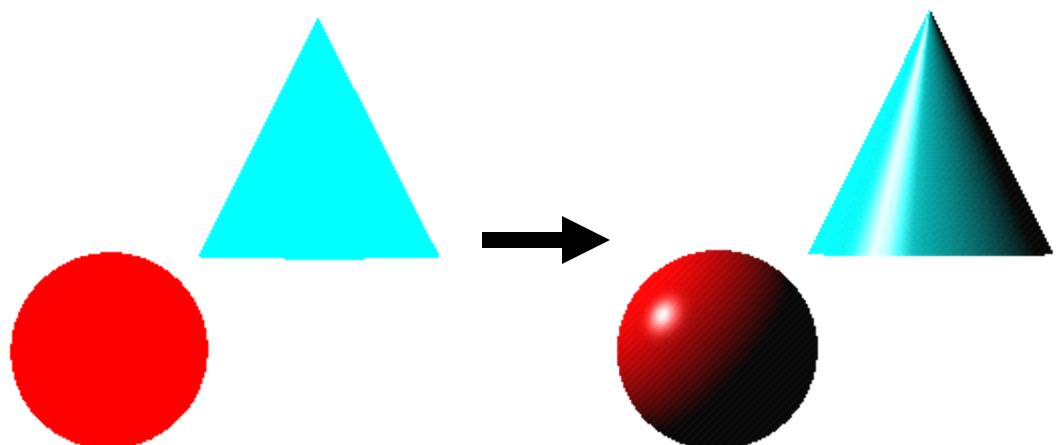
Illumination and Shading

Some slides adopted from
F. Durand and B. Cutler, MIT

The Graphics Pipeline



- Vertices are lit (shaded) according to material properties, surface properties and light sources
- Uses a local lighting model



The Physics of shading

- If we look at a point on an object we perceive a colour and a shading intensity that depends on the various characteristics of the object and the light sources that illuminate it.
- For the time being we will consider only the brightness at each point. We will extend the treatment to colour in the next lecture.

The Physics of shading

- Object properties:
 - Looking at a point on an object we see the reflection of the light that falls on it. This reflection is governed by:
 1. The position of the object relative to the light sources
 2. The surface normal vector
 3. The albedo of the surface (ability to adsorb light energy) or reflectivity of the surface
- Light source properties:
 - The important properties of the light source are:
 1. Intensity of the emitted light
 2. Distance to the point on the surface

Radiometry

- Energy of a photon

$$e_\lambda = \frac{hc}{\lambda} \quad h \approx 6.63 \cdot 10^{-34} \text{ J} \cdot \text{s} \quad c \approx 3 \cdot 10^8 \text{ m/s}$$

- Radiant Energy of n photons

$$Q = \sum_{i=1}^n \frac{hc}{\lambda_i}$$

- Radiation flux (electromagnetic flux, radiant flux)
 - Units: Watts

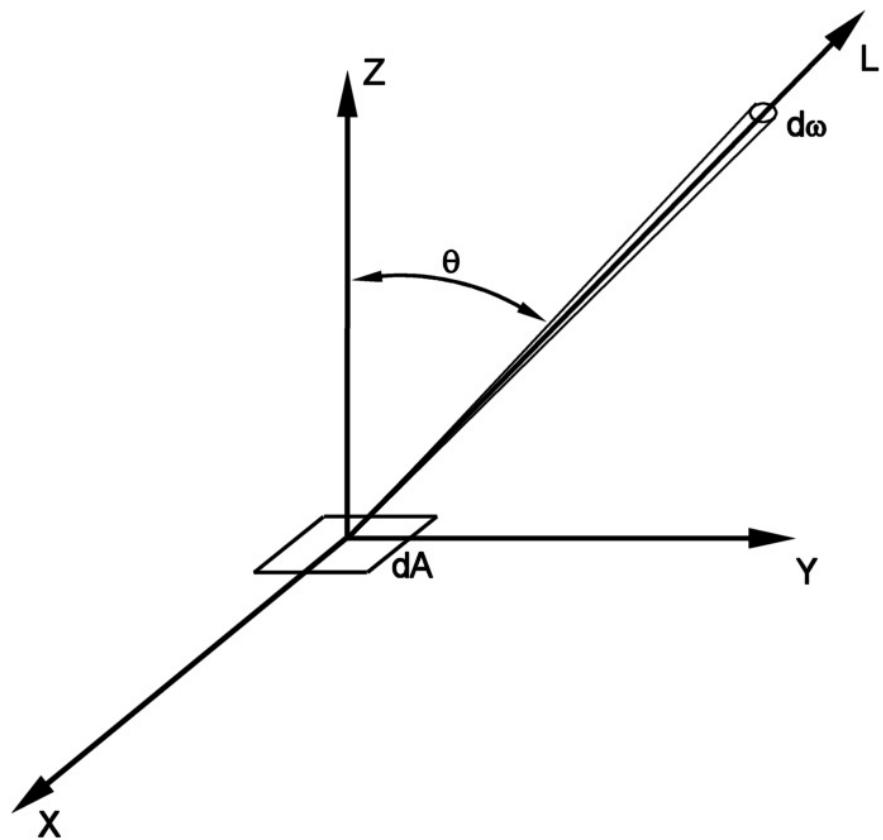
$$\Phi = \frac{dQ}{dt}$$

Radiometry

- **Radiance** – radiant flux per unit solid angle per unit projected area
 - Number of photons
 - 1. per time at a small area
 - 2. in a particular direction

$$L(\omega) = \frac{d^2\Phi}{\cos\theta \ dA \ d\omega}$$

Units : $\frac{\text{Watt}}{\text{meter}^2 \text{ steradian}}$



Radiometry

- **Irradiance** – differential flux falling onto differential area

$$E = \frac{d\Phi}{dA} \qquad \text{Units: } \frac{\text{Watt}}{\text{meter}^2}$$

- Irradiance can be seen as a density of the incident flux falling onto a surface.
- It can be also obtained by integrating the radiance over the solid angle.

Reflection & Reflectance

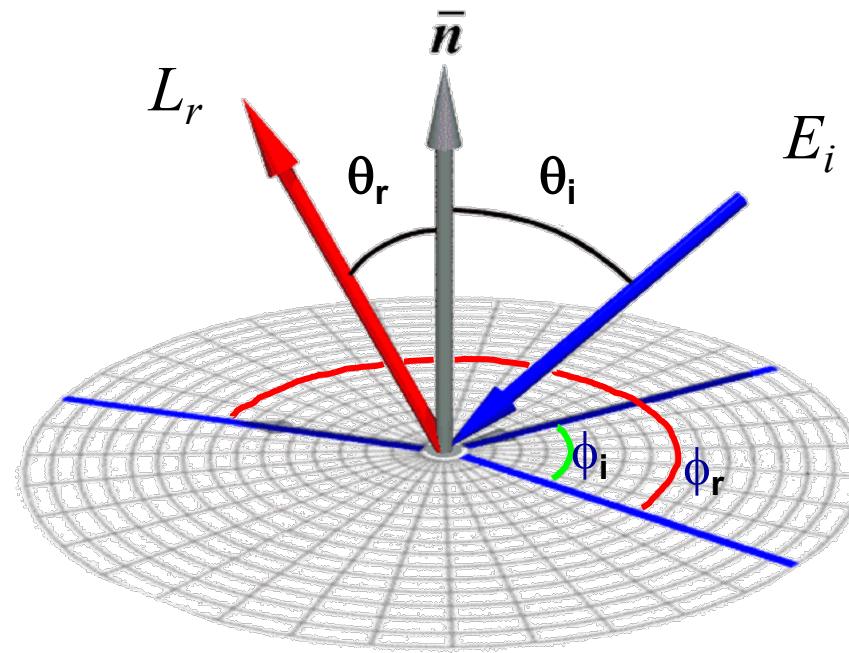
- Reflection - the process by which electromagnetic flux incident on a surface leaves the surface without a change in frequency.
- Reflectance – a fraction of the incident flux that is reflected
- We do not consider:
 - absorption, transmission, fluorescence
 - diffraction

Reflectance

- Bidirectional Reflectance Distribution Function (BRDF)

$$f_r(\theta_i, \phi_i, \theta_r, \phi_r) = \frac{dL_r(\theta_r, \phi_r)}{dE_i(\theta_i, \phi_i)}$$

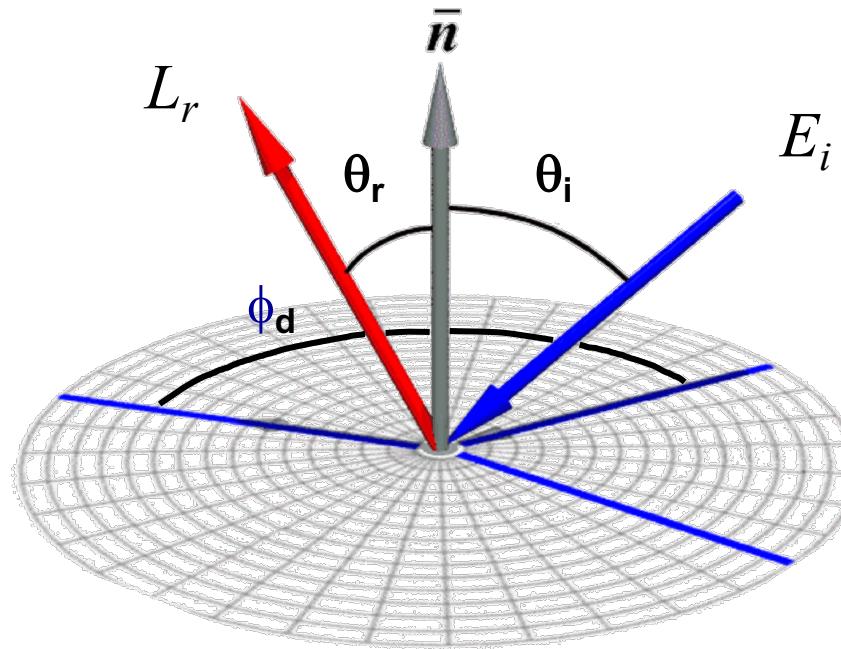
Units : $\frac{1}{\text{steradian}}$



Isotropic BRDFs

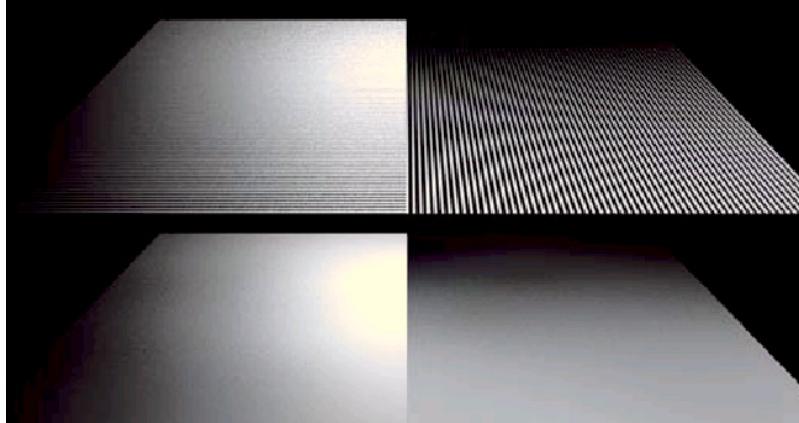
- Rotation along surface normal does not change reflectance

$$f_r(\theta_i, \theta_r, \phi_r - \phi_i) = f_r(\theta_i, \theta_r, \phi_d) = \frac{dL_r(\theta_r, \phi_d)}{dE_i(\theta_i, \phi_d)}$$



Anisotropic BRDFs

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



Source: Westin et al. 92



Properties of BRDFs

- Non-negativity

$$f_r(\theta_i, \phi_i, \theta_r, \phi_r) \geq 0$$

- Energy Conservation

$$\int_{\Omega} f_r(\theta_i, \phi_i, \theta_r, \phi_r) d\mu(\theta_r, \phi_r) \leq 1 \quad \text{for all } (\theta_i, \phi_i)$$

- Reciprocity

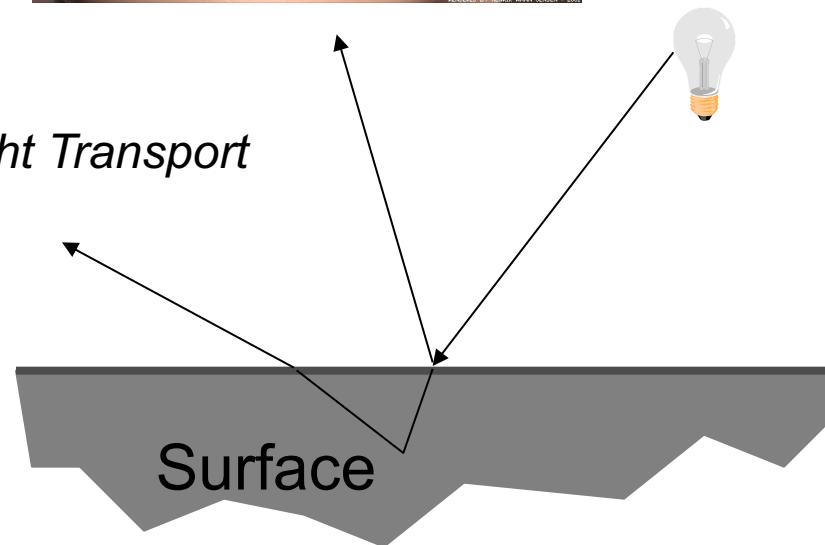
$$f_r(\theta_i, \phi_i, \theta_r, \phi_r) = f_r(\theta_r, \phi_r, \theta_i, \phi_i)$$

Reflectance

- Bidirectional scattering-surface distribution Function (BSSRDF)



Jensen et al. SIGGRAPH 2001
A Practical Model for Subsurface Light Transport



How to compute reflected radiance?

- Continuous version

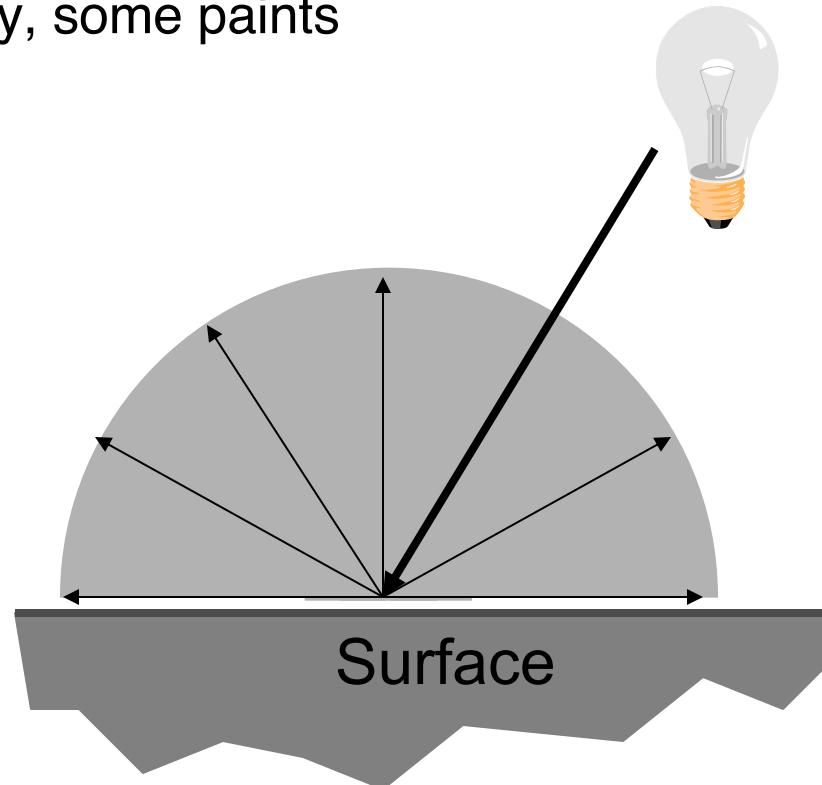
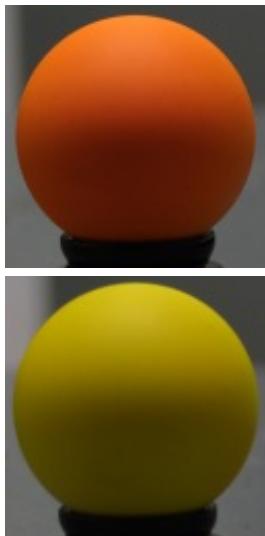
$$\begin{aligned} L_r(\omega_r) &= \int_{\Omega} f_r(\omega_i, \omega_r) dE_i(\omega_i) = \\ &= \int_{\Omega} f_r(\omega_i, \omega_r) dL_i(\omega_i) \cos(\omega_i \cdot n) d\omega_i \quad \omega = (\theta, \phi) \end{aligned}$$

- Discrete version – n point light sources

$$\begin{aligned} L_r(\omega_r) &= \sum_{j=1}^n f_r(\omega_{ij}, \omega_r) E_j = \\ &= \sum_{j=1}^n f_r(\omega_{ij}, \omega_r) \cos \theta_j \frac{\Phi_{sj}}{4\pi d_j^2} \end{aligned}$$

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

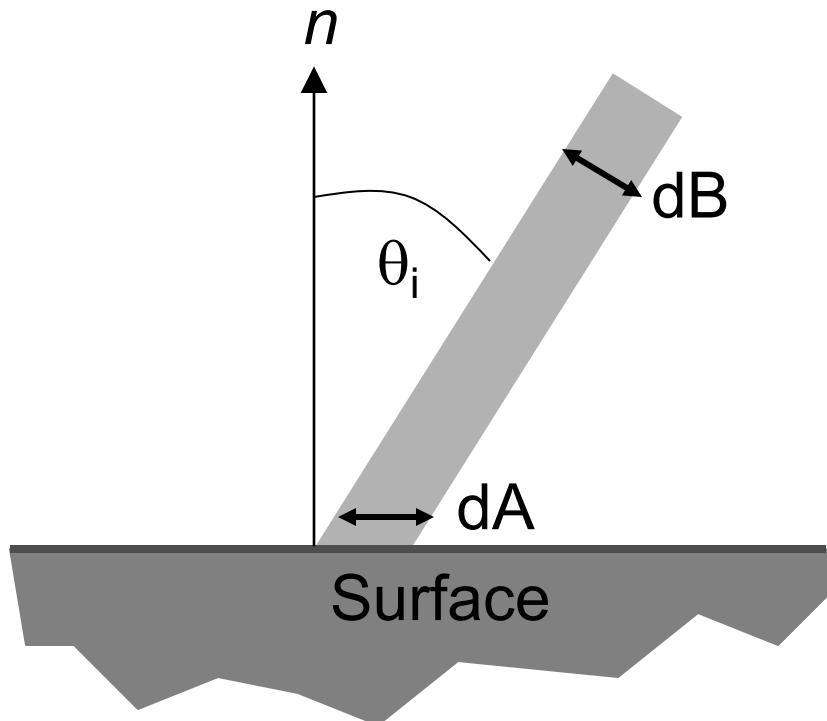


Ideal Diffuse Reflectance

- BRDF value is constant

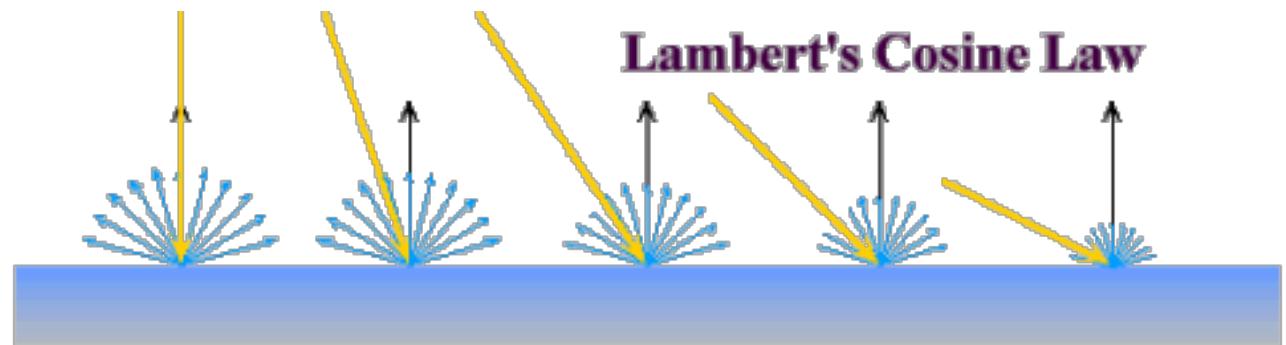
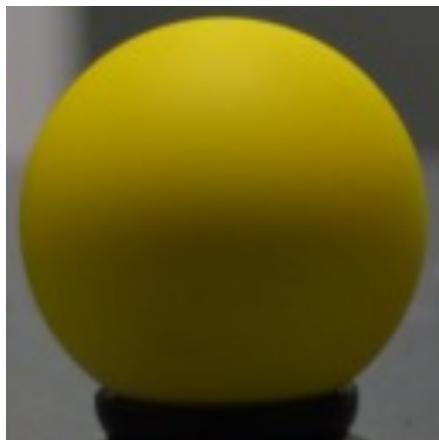
$$\begin{aligned}L_r(\omega_r) &= \int_{\Omega} f_r(\omega_i, \omega_r) dE_i(\omega_i) = \\&= f_r \int_{\Omega} dE_i(\omega_i) = \\&= f_r E_i\end{aligned}$$

$$dB = dA \cos \theta_i$$



Ideal Diffuse Reflectance

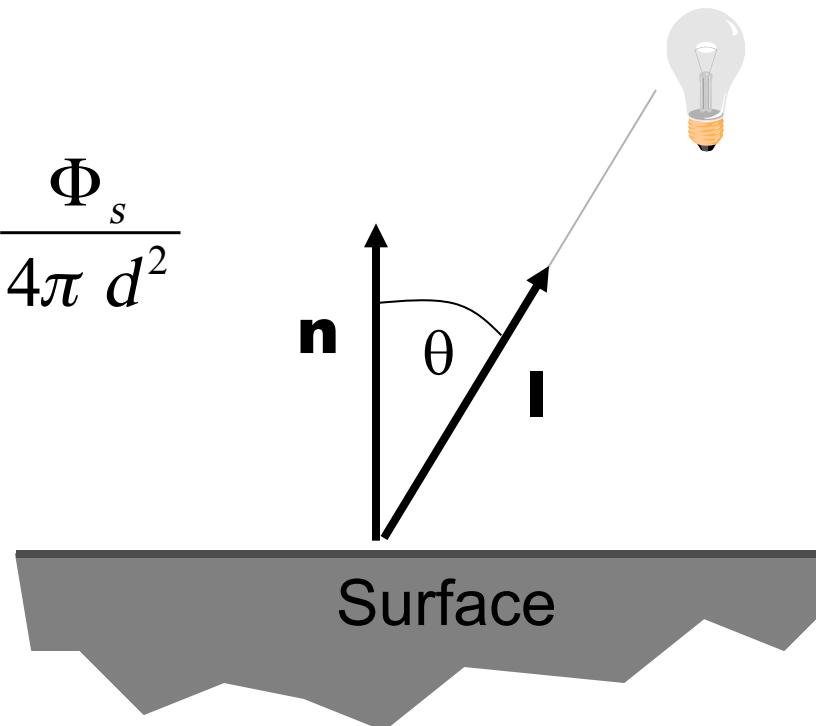
- Ideal diffuse reflectors reflect light according to Lambert's cosine law.



Ideal Diffuse Reflectance

- Single Point Light Source
 - k_d : The diffuse reflection coefficient.
 - n : Surface normal.
 - l : Light direction.

$$L(\omega_r) = k_d (\mathbf{n} \cdot \mathbf{l}) \frac{\Phi_s}{4\pi d^2}$$



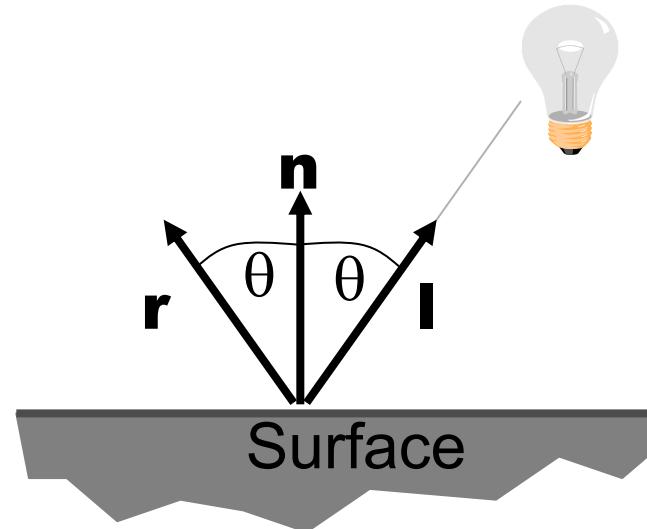
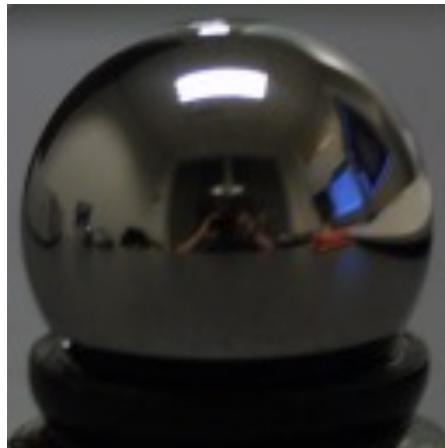
Ideal Diffuse Reflectance – More Details

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

Do not forget to normalize your vectors
for the dot product!

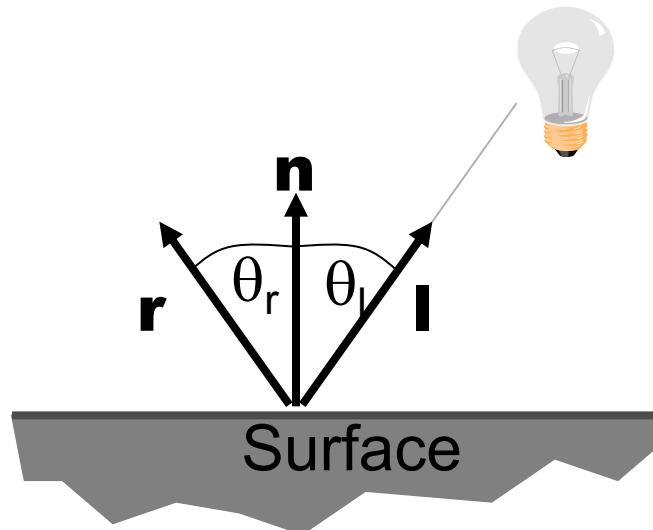
Ideal Specular Reflectance

- 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.



Ideal Specular Reflectance

- Special case of Snell's Law
 - The incoming ray, the surface normal, and the reflected ray all lie in a common plane.



$$n_l \sin \theta_l = n_r \sin \theta_r$$

$$n_l = n_r$$

$$\theta_l = \theta_r$$

Non-ideal Reflectors

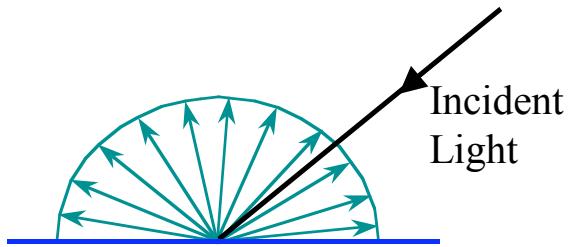
- Snell's law applies only to ideal mirror reflectors.
- Real materials tend to deviate significantly from ideal mirror reflectors.
- They are not ideal diffuse surfaces either ...



Non-ideal Reflectors

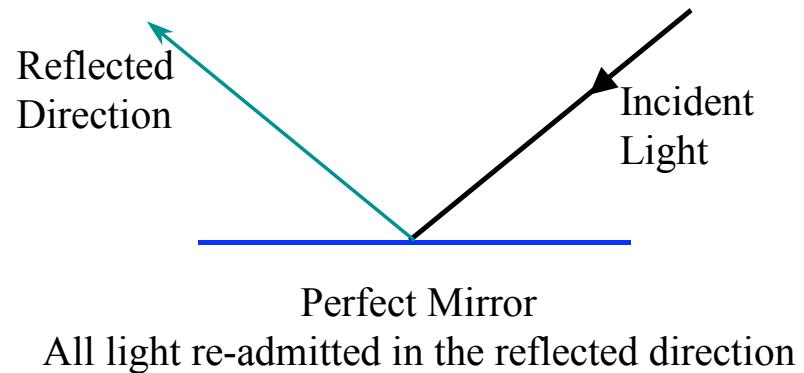
- Simple Empirical Model:
 - We expect most of the reflected light to travel in the direction of the ideal ray.
 - However, because of microscopic surface variations we might expect some of the light to be reflected just slightly offset from the ideal reflected ray.
 - As we move farther and farther,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

Non-ideal Reflectors: Surface Characteristics



Perfectly Matt surface

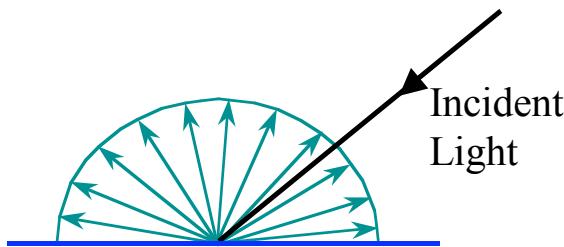
The reflected intensity is the same in all directions



Perfect Mirror

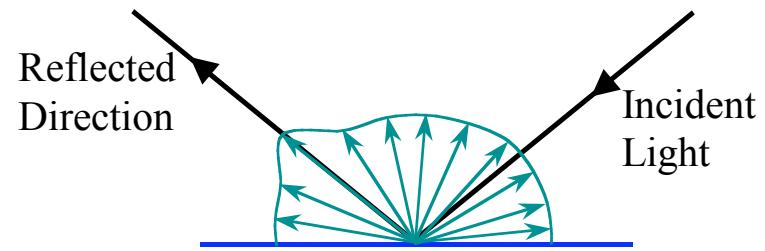
All light re-admitted in the reflected direction

Non-ideal Reflectors: Surface Characteristics



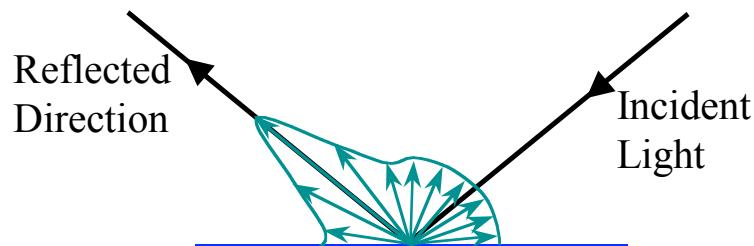
Perfectly Matt surface

The reflected intensity is the same in all directions

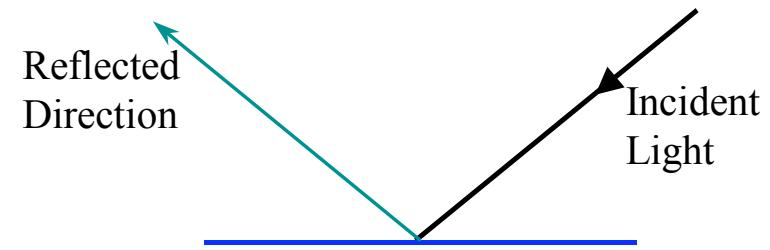


Slightly specular (shiny) surface

Slightly higher intensity in the reflected direction



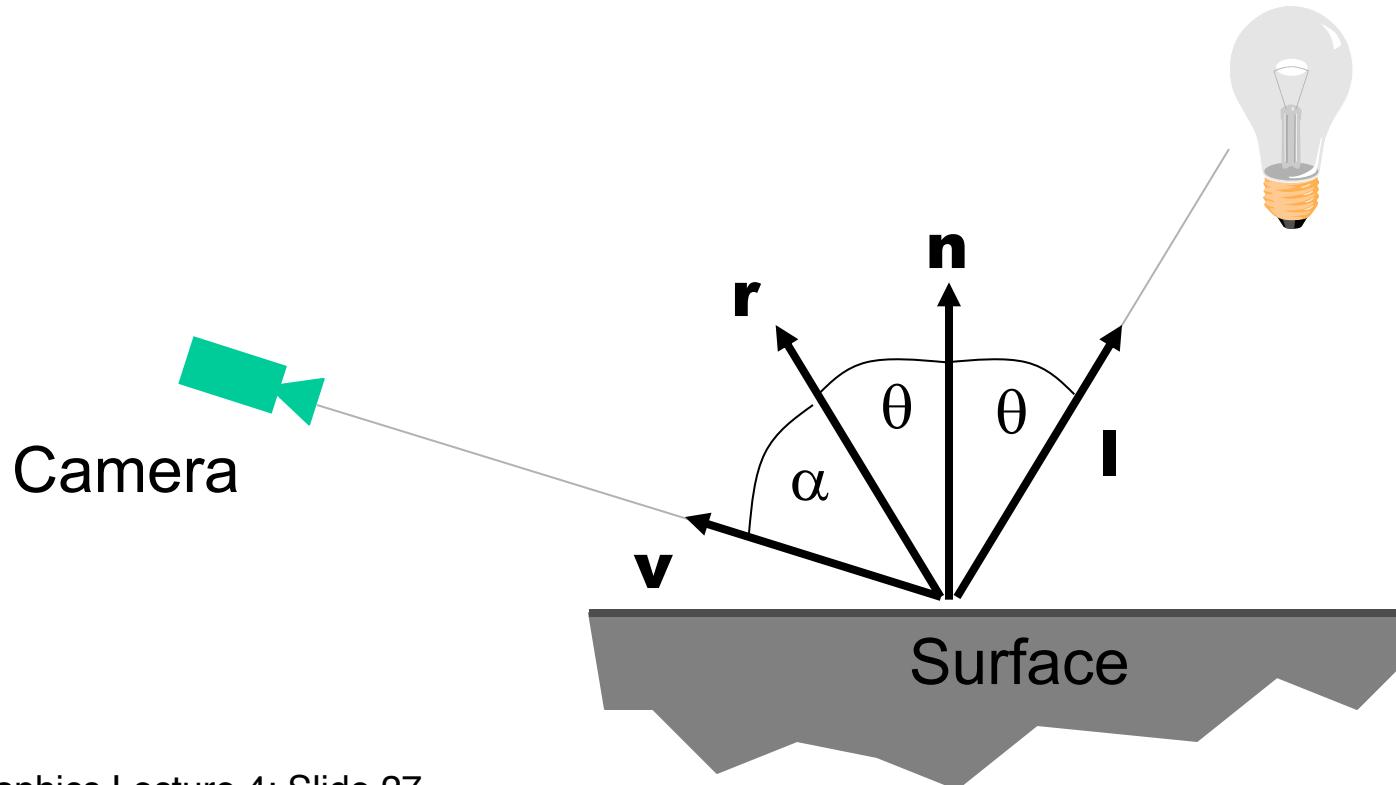
Highly specular (shiny) surface
High intensity in the reflected direction



Perfect Mirror
All light re-admitted in the reflected direction

The Phong Model

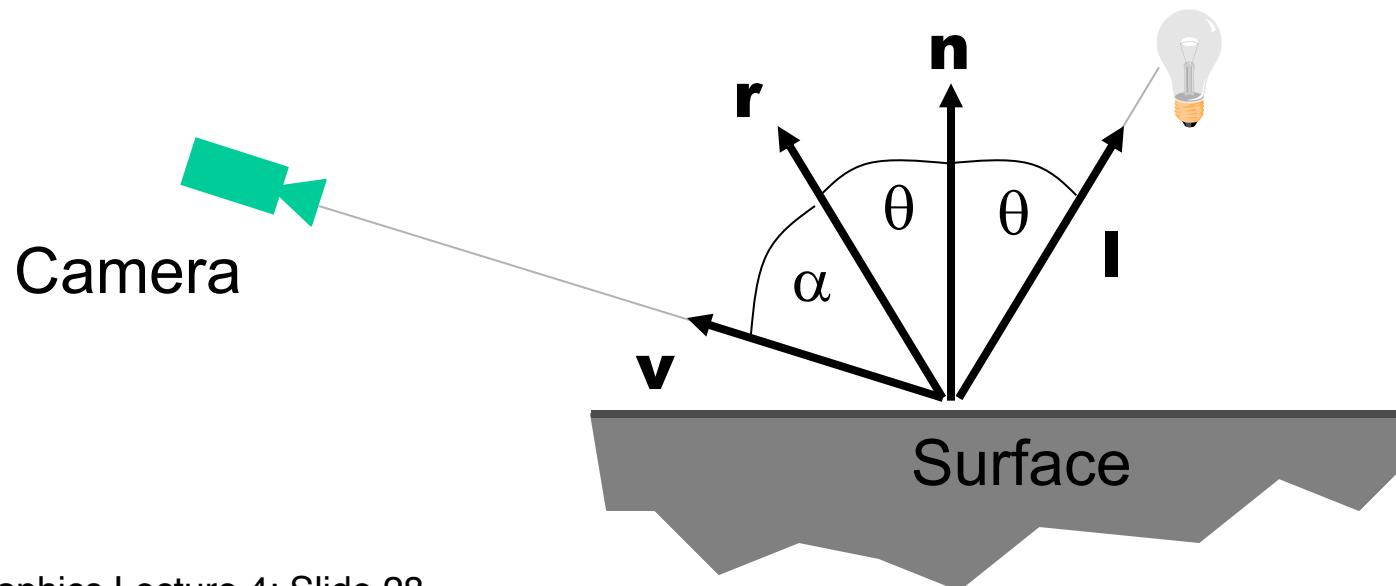
- How much light is reflected?
 - Depends on the angle between the ideal reflection direction and the viewer direction α .



The Phong Model

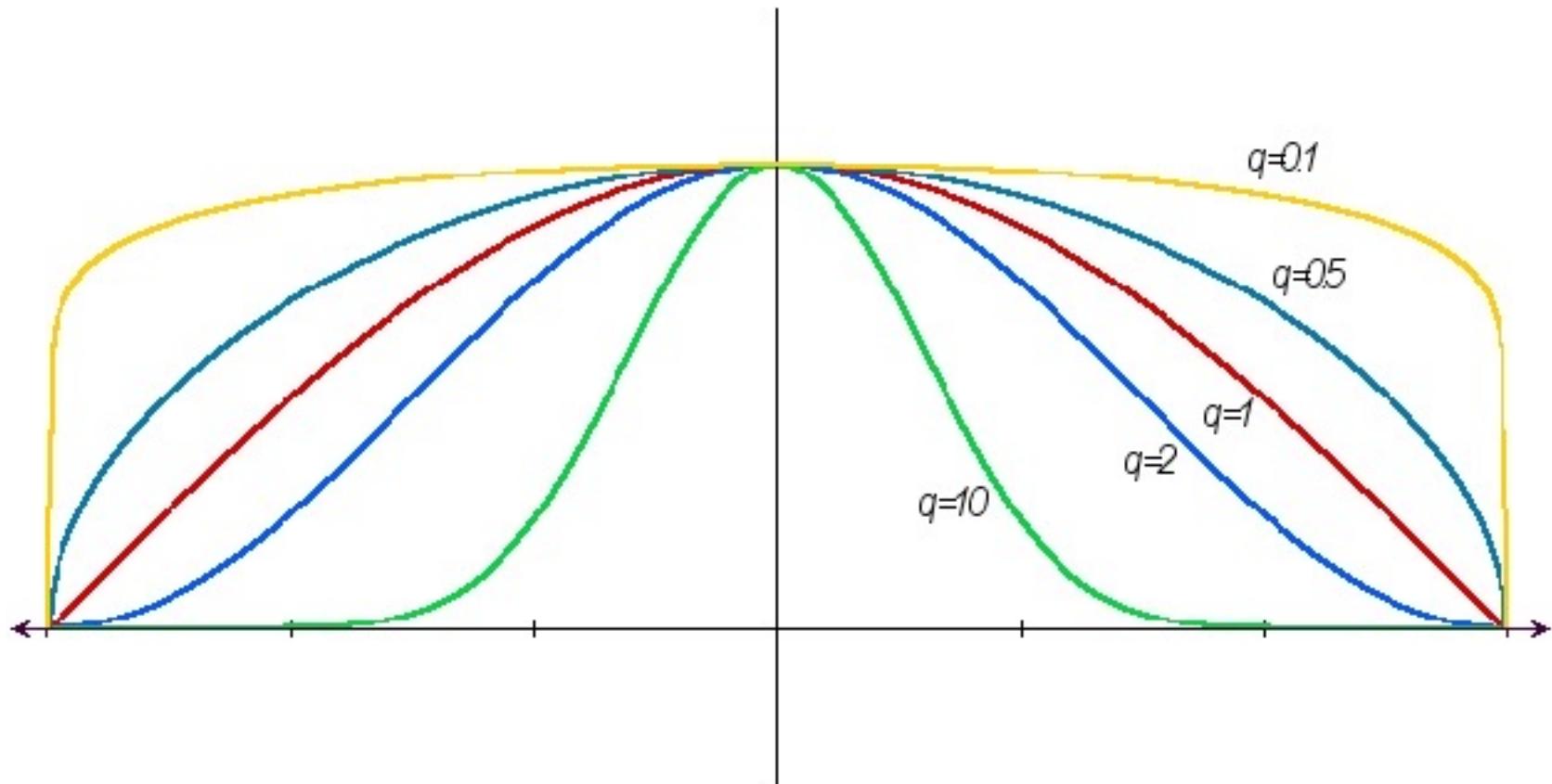
- Parameters
 - k_s : specular reflection coefficient
 - q : specular reflection exponent

$$L(\omega_r) = k_s (\cos \alpha)^q \frac{\Phi_s}{4\pi d^2} = k_s (\mathbf{v} \cdot \mathbf{r})^q \frac{\Phi_s}{4\pi d^2}$$



The Phong Model

- Effect of the q coefficient

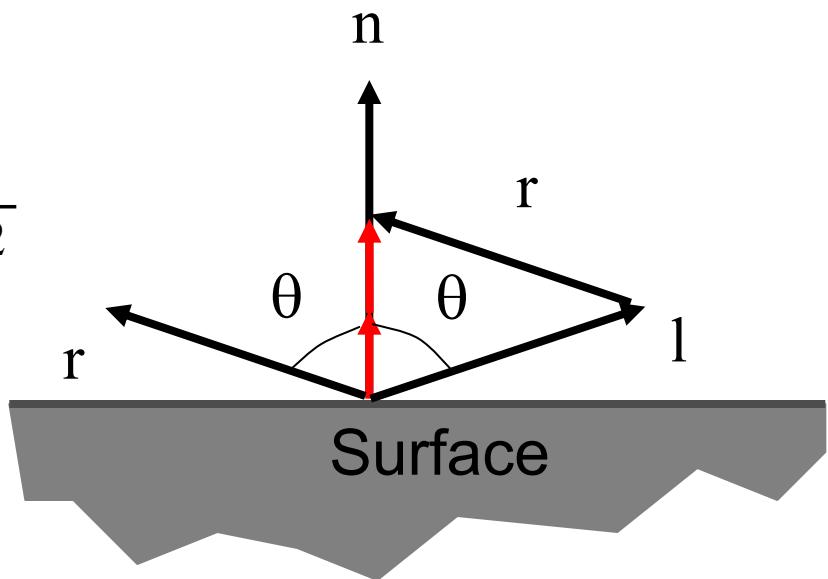


The Phong Model

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

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

$$L(\omega_r) = k_s (\mathbf{v} \cdot \mathbf{r})^q \frac{\Phi_s}{4\pi d^2} = \\ = k_s (\mathbf{v} \cdot (2(\mathbf{n} \cdot \mathbf{l})\mathbf{n} - \mathbf{l}))^q \frac{\Phi_s}{4\pi d^2}$$

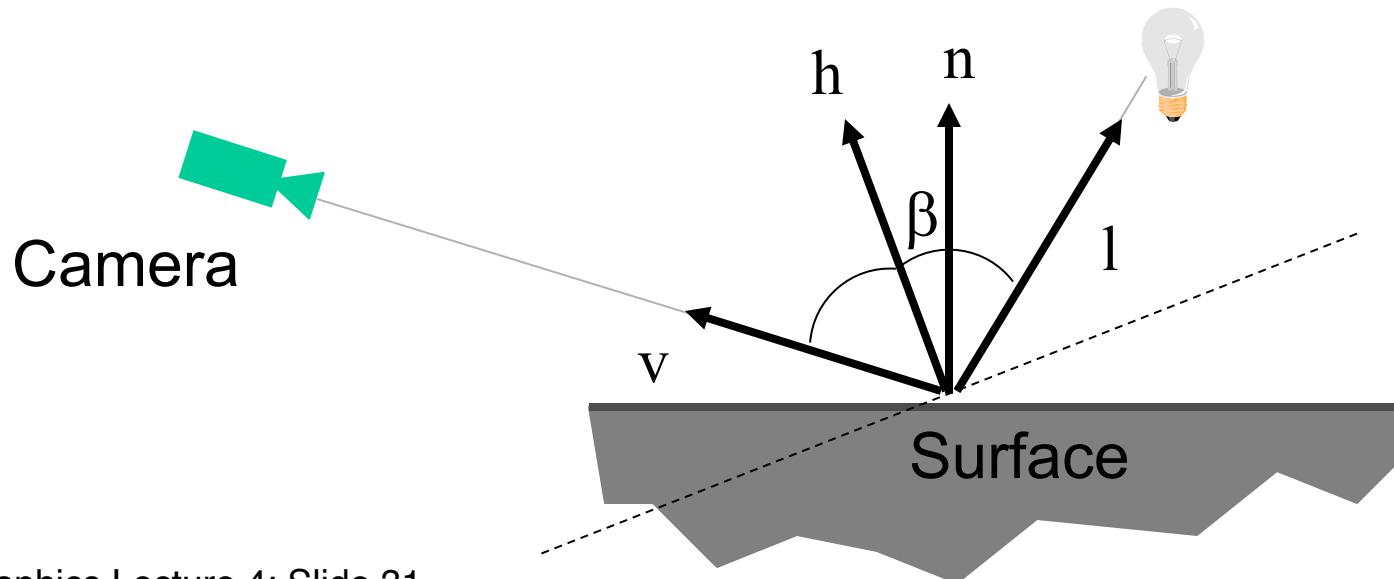


Blinn-Phong Variation

- Uses the halfway vector \mathbf{h} between \mathbf{l} and \mathbf{v} .

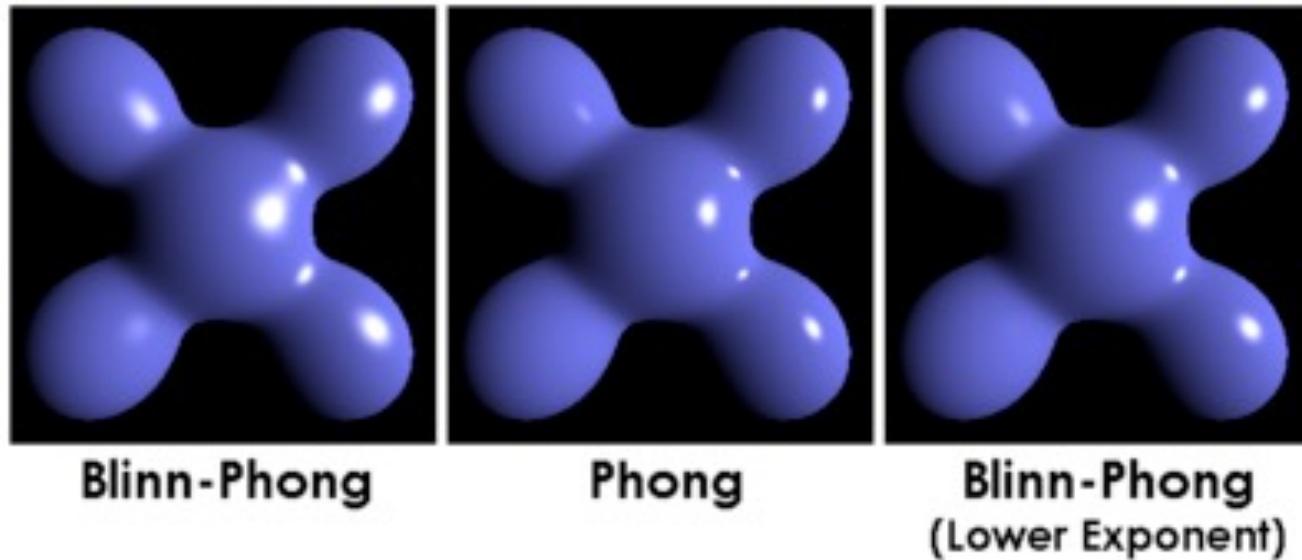
$$\mathbf{h} = \frac{\mathbf{l} + \mathbf{v}}{\|\mathbf{l} + \mathbf{v}\|}$$

$$L(\omega_r) = k_s (\cos \beta)^q \frac{\Phi_s}{4\pi d^2} = k_s (\mathbf{n} \cdot \mathbf{h})^q \frac{\Phi_s}{4\pi d^2}$$



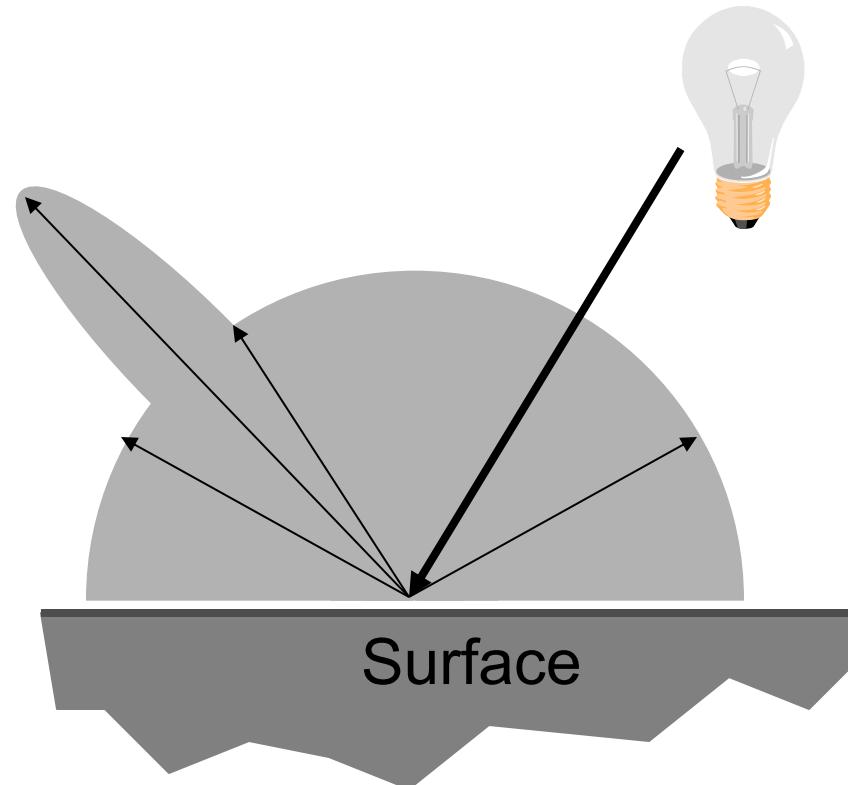
Phong vs Blinn-Phong

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



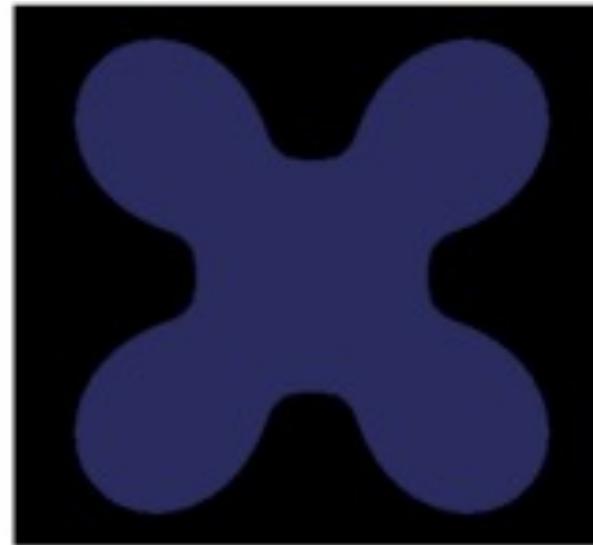
The Phong Model

- Sum of three components:
 - diffuse reflection + specular reflection + **ambient**.



Ambient Illumination

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

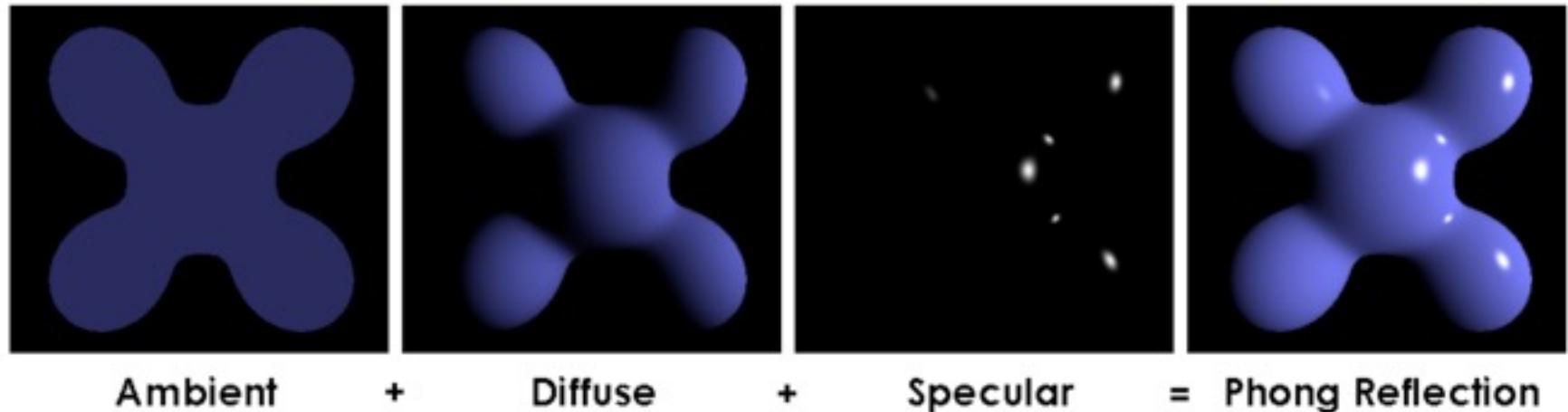


$$L(\omega_r) = k_a$$

Putting it all together

- Phong Illumination Model

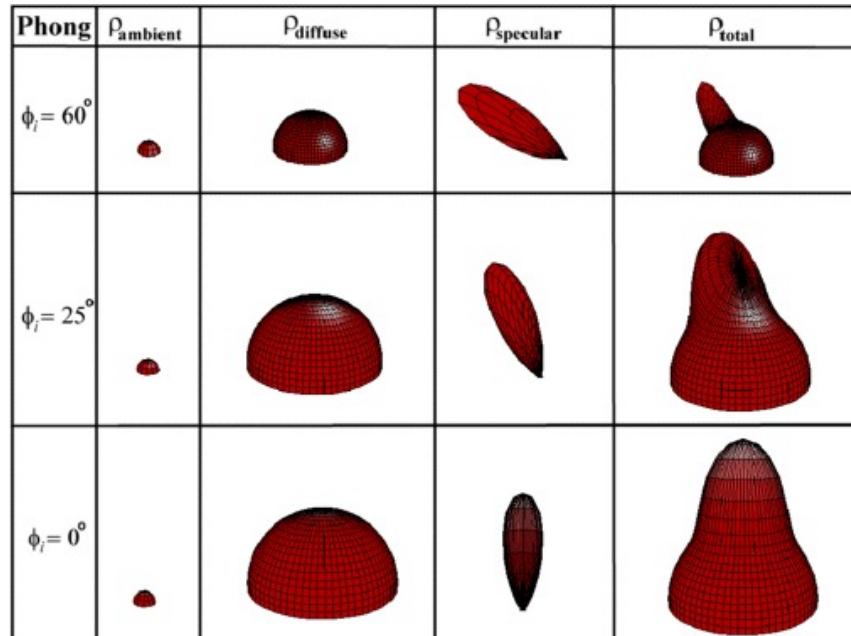
$$L(\omega_r) = k_a + \left(k_d (\mathbf{n} \cdot \mathbf{l}) + k_s (\mathbf{v} \cdot \mathbf{r})^q \right) \frac{\Phi_s}{4\pi d^2}$$



Putting it all together

- Phong Illumination Model

$$L(\omega_r) = k_a + \left(k_d (\mathbf{n} \cdot \mathbf{l}) + k_s (\mathbf{v} \cdot \mathbf{r})^q \right) \frac{\Phi_s}{4\pi d^2}$$



Inverse Square Law

- It is well known that light falls off according to an inverse square law. Thus, if we have light sources close to our polygons we should model this effect.

$$L(\omega_r) = k_a + \left(k_d (\mathbf{n} \cdot \mathbf{l}) + k_s (\mathbf{v} \cdot \mathbf{r})^q \right) \frac{\Phi_s}{4\pi d^2}$$

where d is the distance from the light source to the object

Heuristic Law

- Although physically correct the inverse square law does not produce the best results.
- Instead the following is often used:

$$L(\omega_r) = k_a + \left(k_d (\mathbf{n} \cdot \mathbf{l}) + k_s (\mathbf{v} \cdot \mathbf{r})^q \right) \frac{\Phi_s}{4\pi(d+s)}$$

where s is an heuristic constant.

- One might be tempted to think that light intensity falls off with the distance to the viewpoint, but it doesn't!
- Why not?

Φ_s

= 5000

(for the coursework)

Questions?



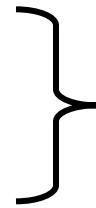
Using Shading

- There are three levels at which shading can be applied in polygon based systems:

Flat Shading

Gouraud Shading

Phong Shading



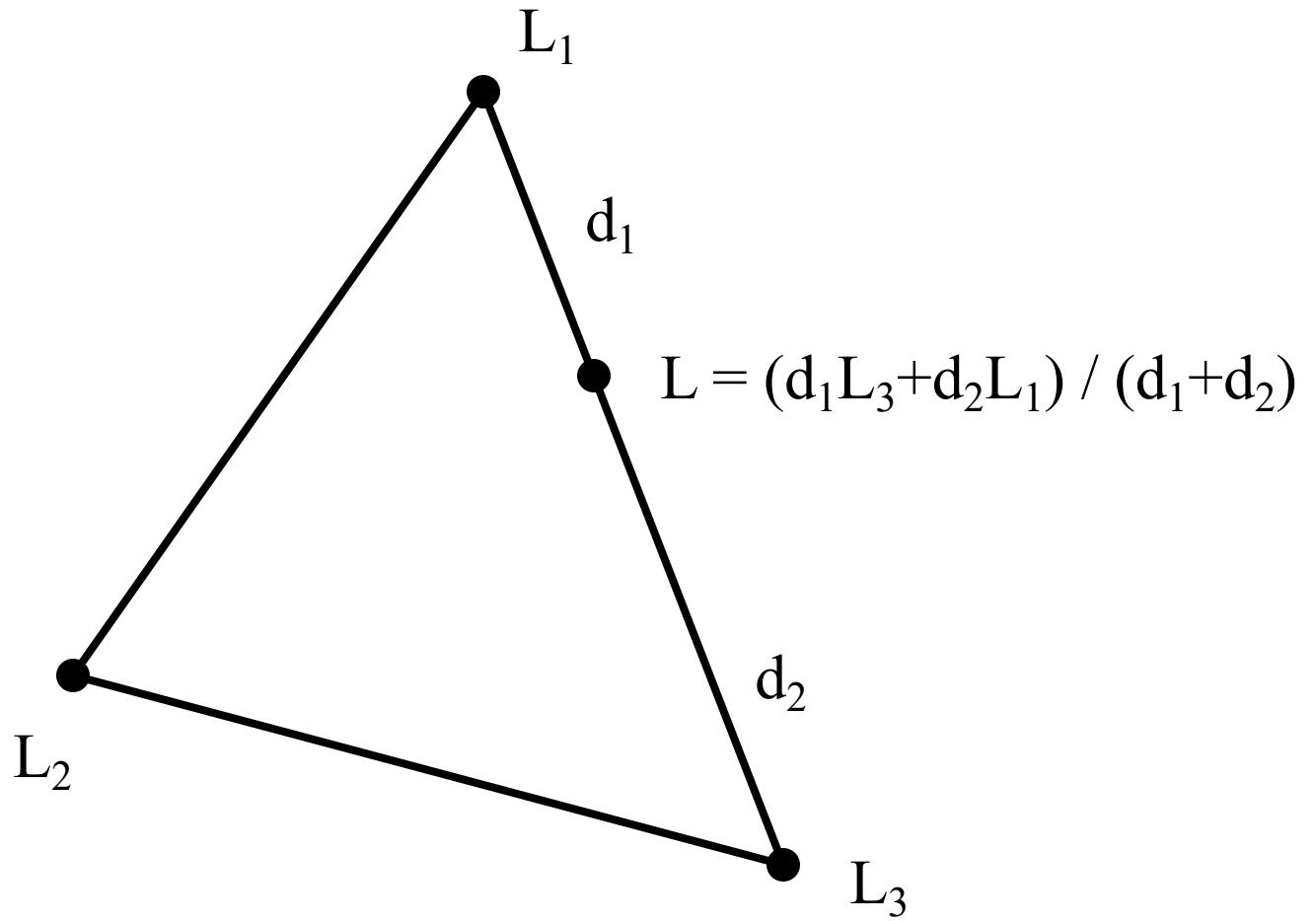
Interpolation Shading

- They provide increasing realism at higher computational cost

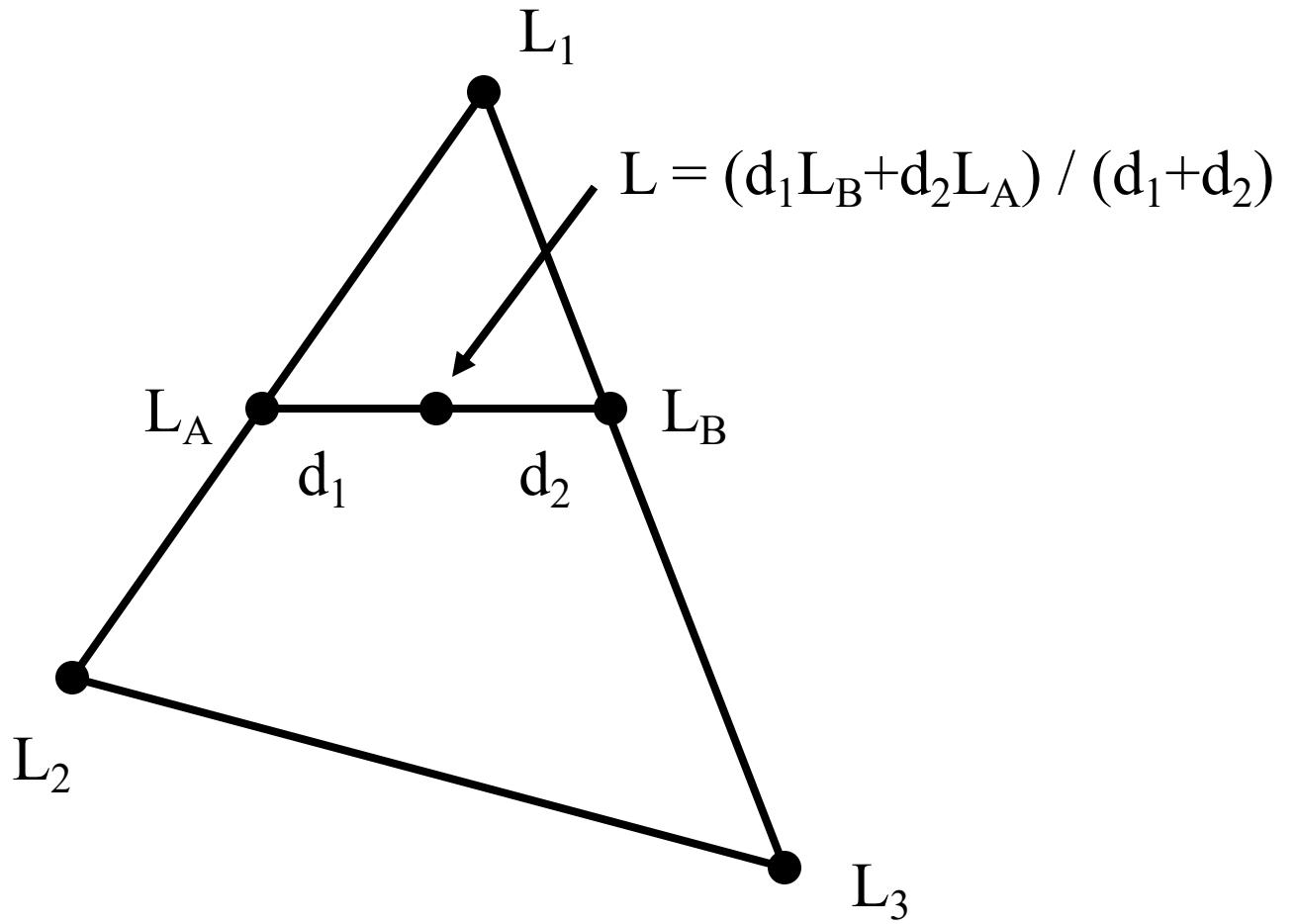
Using Shading

- Flat Shading:
 - Each polygon is shaded uniformly over its surface.
 - The shade is computed by taking a point in the centre and the surface normal vector. (Equivalent to a light source at infinity)
 - Usually only diffuse and ambient components are used.
- Interpolation Shading:
 - A more accurate way to render a shaded polygon is to compute an independent shade value at each point.
 - This is done quickly by interpolation:
 1. Compute a shade value at each vertex
 2. Interpolate to find the shade value at the boundary
 3. Interpolate to find the shade values in the middle

Calculating the shades at the edges

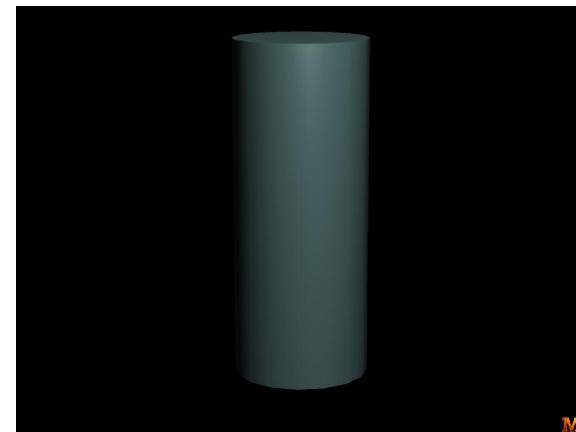
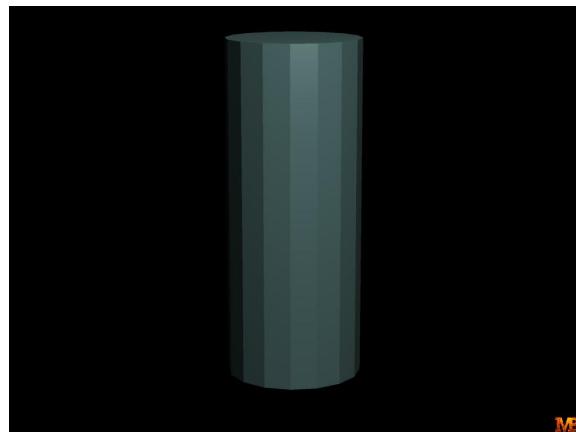


Calculating the internal shades

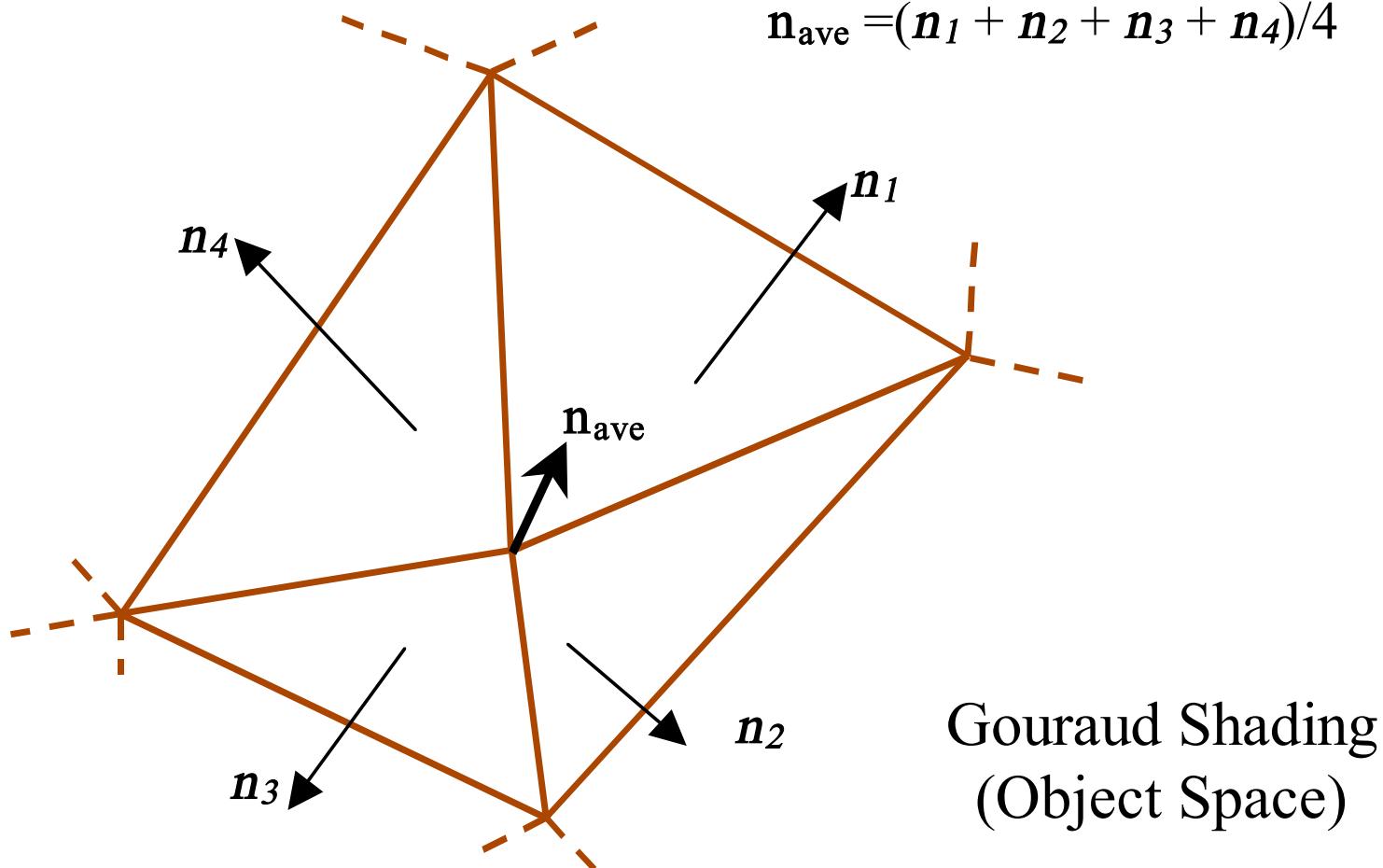


Interpolating over polygons

- In addition to interpolating shades over polygons, we can interpolate them over groups of polygons to create the impression of a smooth surface.
- The idea is to create at each vertex an averaged intensity from all the polygons that meet at that vertex.



Computing an average normal vector at a vertex



Smooth Shading

- Need to have per-vertex normals
- Gouraud Shading
 - Interpolate color across triangles
 - Fast, supported by most of the graphics accelerator cards
 - Can't model specular components accurately, since we do not have the normal vector at each point on a polygon.
- Phong Shading
 - Interpolate normals across triangles
 - More accurate modelling of specular components, but slower.

Interpolation of the 3D normals

- We may express any point for this facet in parametric form:

$$\mathbf{P} = \mathbf{V}_1 + \mu_1(\mathbf{V}_2 - \mathbf{V}_1) + \mu_2(\mathbf{V}_3 - \mathbf{V}_1)$$

- The average normal vector at the same point may be calculated as the vector \mathbf{a} :

$$\mathbf{a} = \mathbf{n}_1 + \mu_1(\mathbf{n}_2 - \mathbf{n}_1) + \mu_2(\mathbf{n}_3 - \mathbf{n}_1)$$

- and then

$$\mathbf{n}_{\text{average}} = \mathbf{a} / | \mathbf{a} |$$

2D or 3D

- The interpolation calculations may be done in either 2D or 3D
- For specular reflections the calculation of the reflected vector and viewpoint vector must be done in 3D.

Interactive Computer Graphics: Lecture 7

Colour

Ways of looking at colour

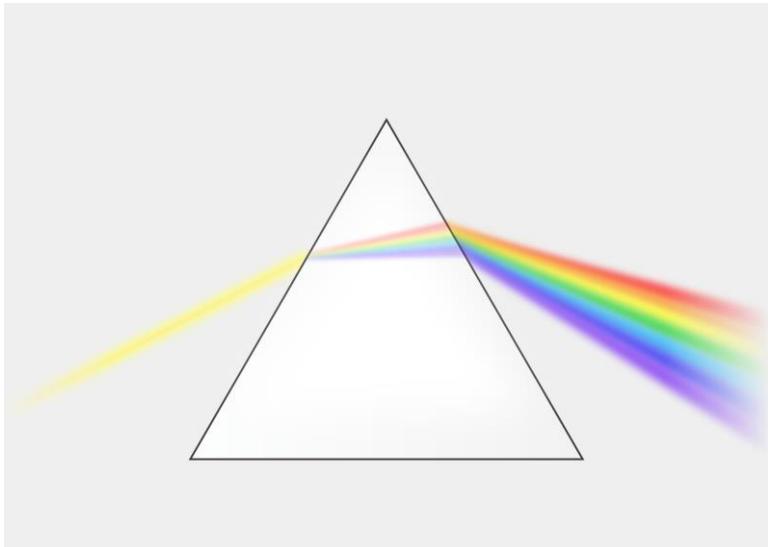
1. Physics
2. Biology (how do human visual receptors work?)
3. Psychology (how do humans subjectively assess colour?)

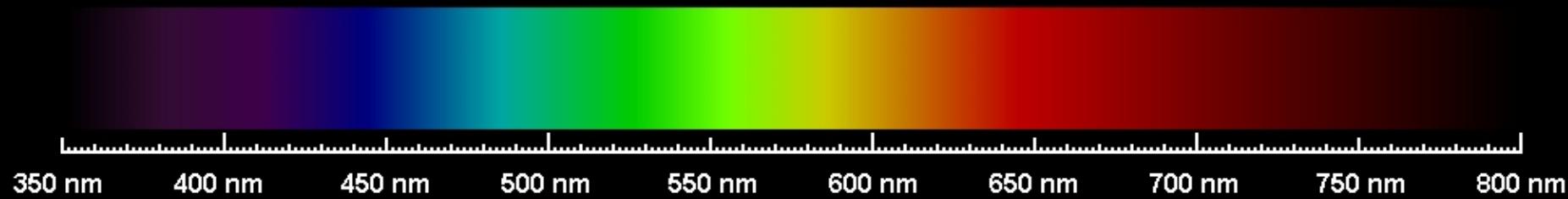
The physics of colour

- A pure colour is a wave with:

Wavelength (λ)

Amplitude (intensity or energy) (I)





Visible Continuous Spectrum 2

(Perceived Brightness Partially to Scale)

Colours are energy distributions

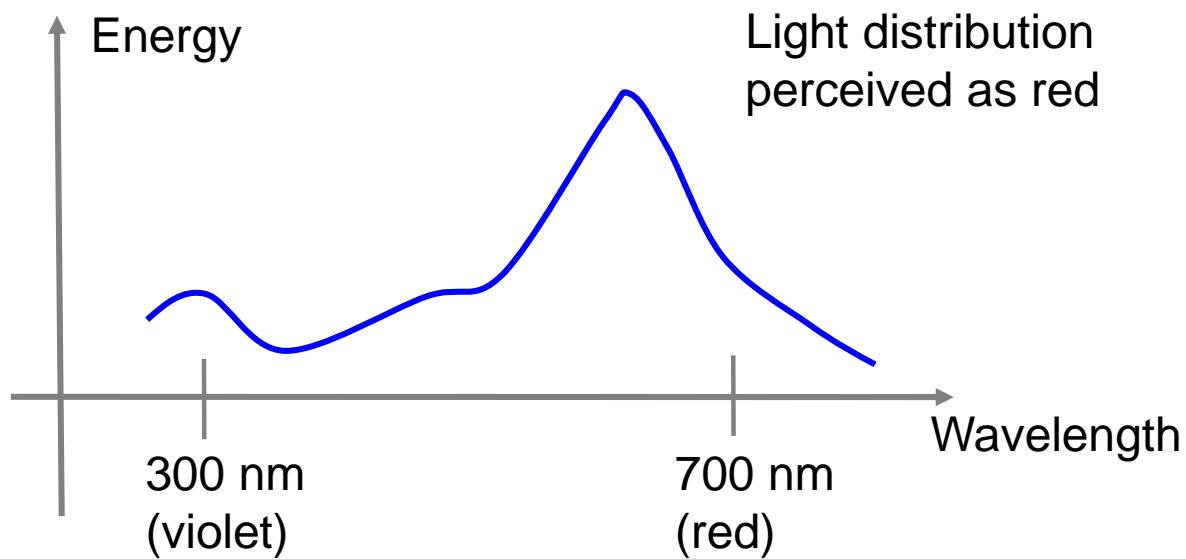
Color	Frequency	Wavelength
violet	668–789 THz	380–450 nm
blue	631–668 THz	450–475 nm
cyan	606–630 THz	476–495 nm
green	526–606 THz	495–570 nm
yellow	508–526 THz	570–590 nm
orange	484–508 THz	590–620 nm
red	400–484 THz	620–750 nm



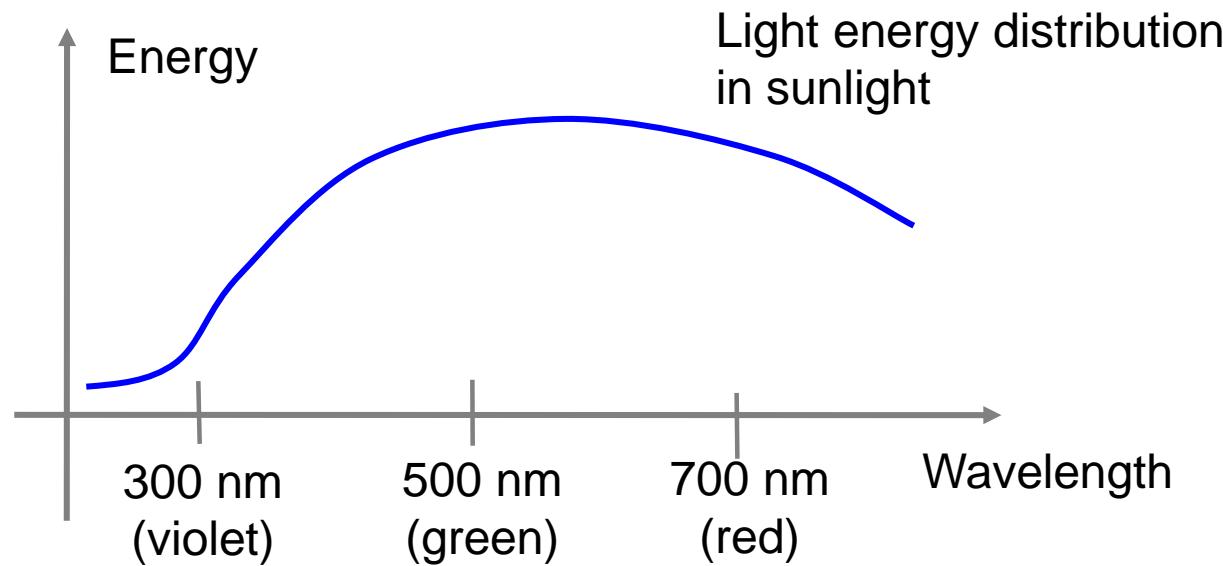
Colours are energy distributions

- Lasers are light sources that contain a single wavelength (or a very narrow band of wavelengths)
- In practice light is made up of a mixture of many wavelengths with an energy distribution.

Light distribution for red



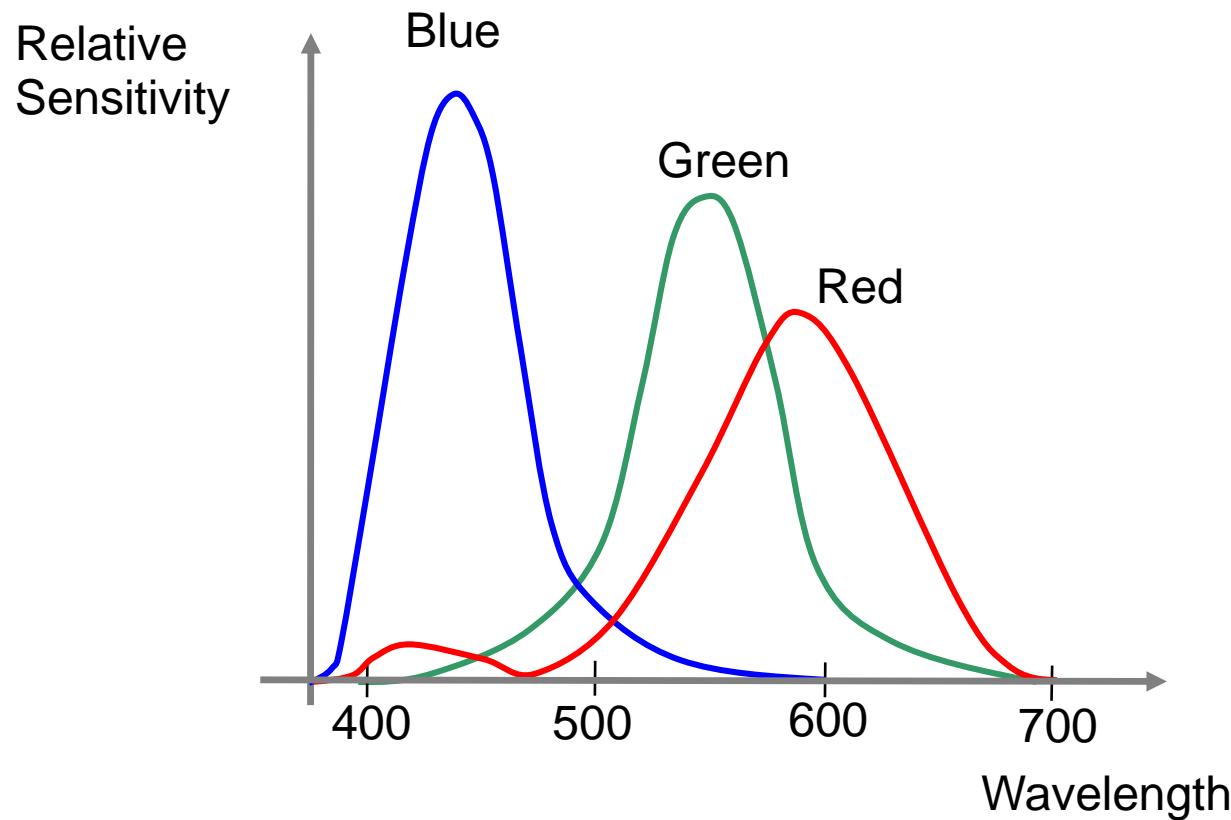
Sunlight



Human Colour Vision

- Human colour vision is based on three ‘cone’ cell types which respond to light energy in different bands of wavelength.
- The bands overlap in a curious manner.

Human receptor response



Tri-Stimulus Colour theory

The receptor performance implies that colours do not have a unique energy distribution.

And more importantly:

Colours which are a distribution over all wavelengths can be matched by mixing three.

R G B

Colour Matching

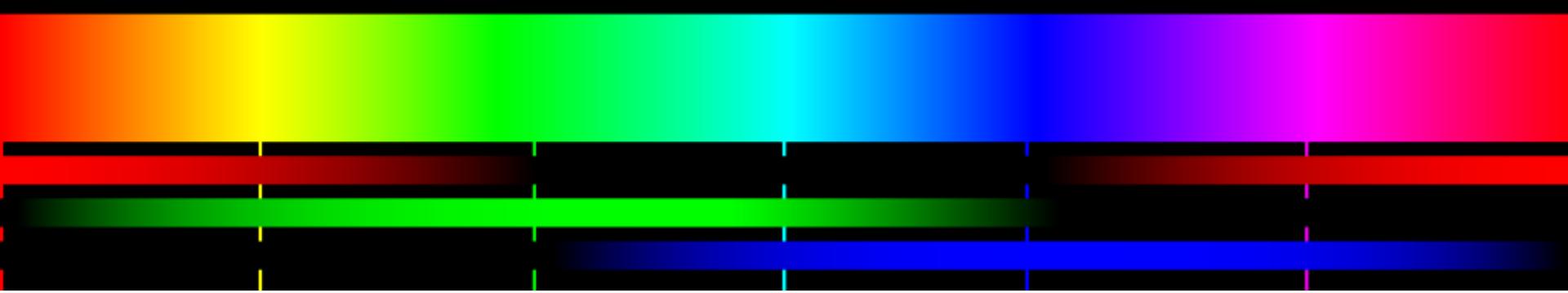
Given any colour light source, regardless of the distribution of wavelengths that it contains, we can try to match it with a mixture of three light sources

$$X = r R + g G + b B$$

where R, G and B are pure light sources and r, g and b their intensities

For simplicity we can drop the R G B.

Colour Matching



Subtractive matching

Not all colours can be matched with a given set of light sources (we shall see why later)

However, we can add light to the colour we are trying to match:

$$X + r = g + b$$

With this technique all colours can be matched.

The CIE diagram

The CIE diagram was devised as a standard normalised representation of colour.

As we noted, given three light sources we can mix them to match any given colour, providing we allow ourselves subtractive matching.

Suppose we normalise the ranges found to [0..1] to avoid the negative signs.

Normalised colours

Having normalised the range over which the matching is done we can now normalise the colours such that the three components sum to 1.

Thus

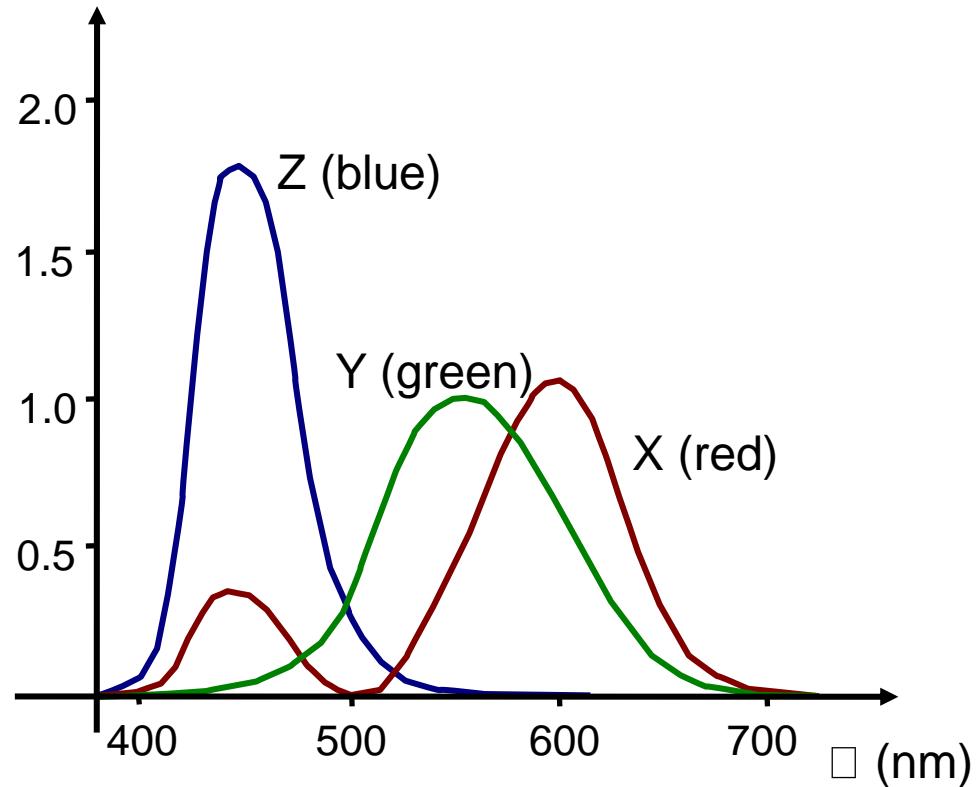
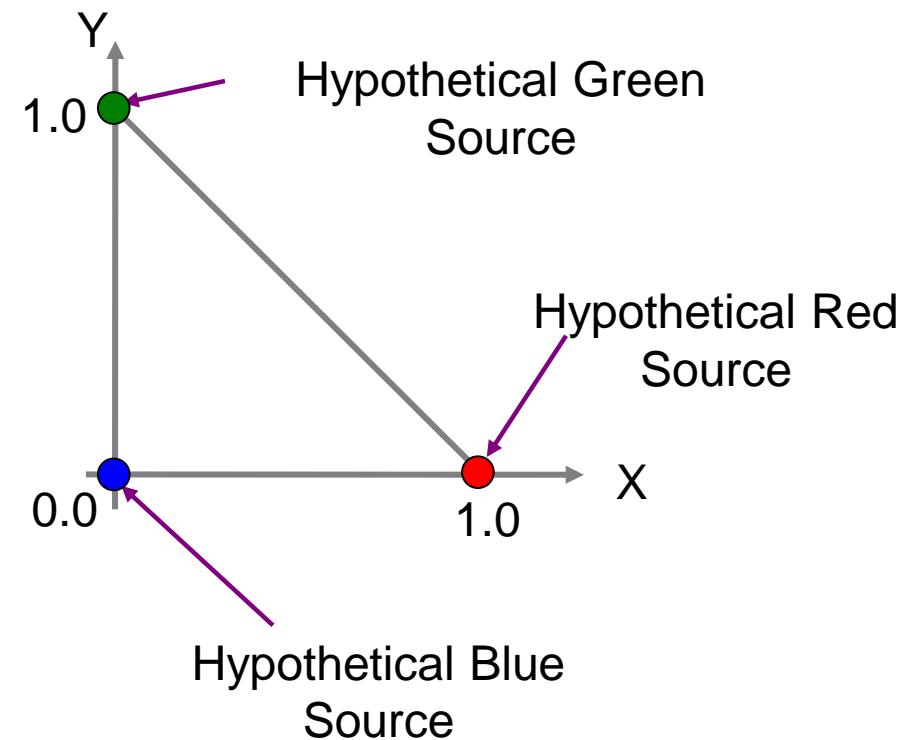
$$x = r/(r+g+b)$$

$$y = g/(r+g+b)$$

$$z = b/(r+g+b) = 1 - x - y$$

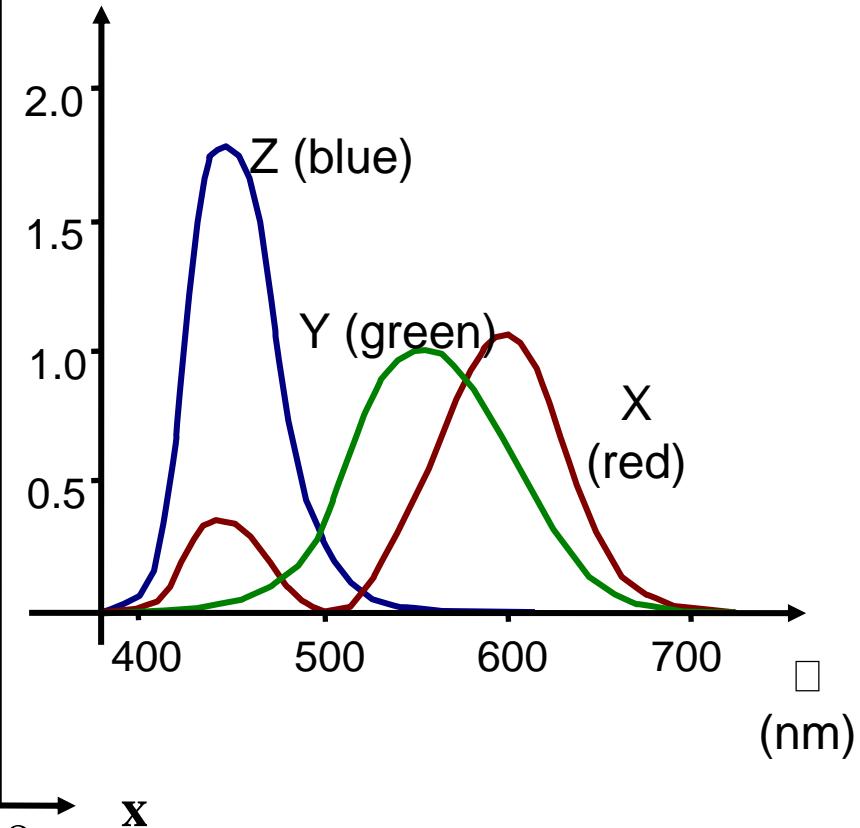
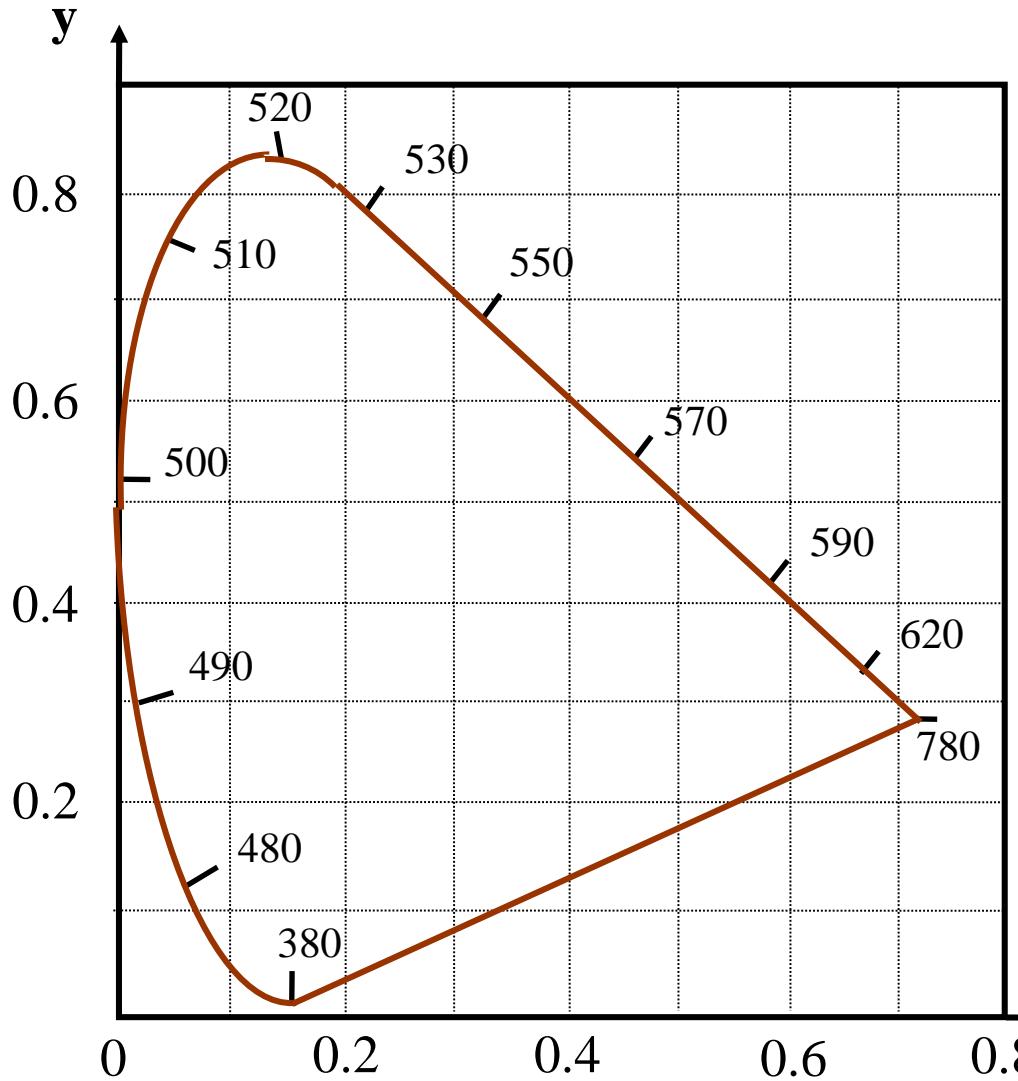
We can now represent all our colours in a 2D space.

Defining the normalised CIE diagram

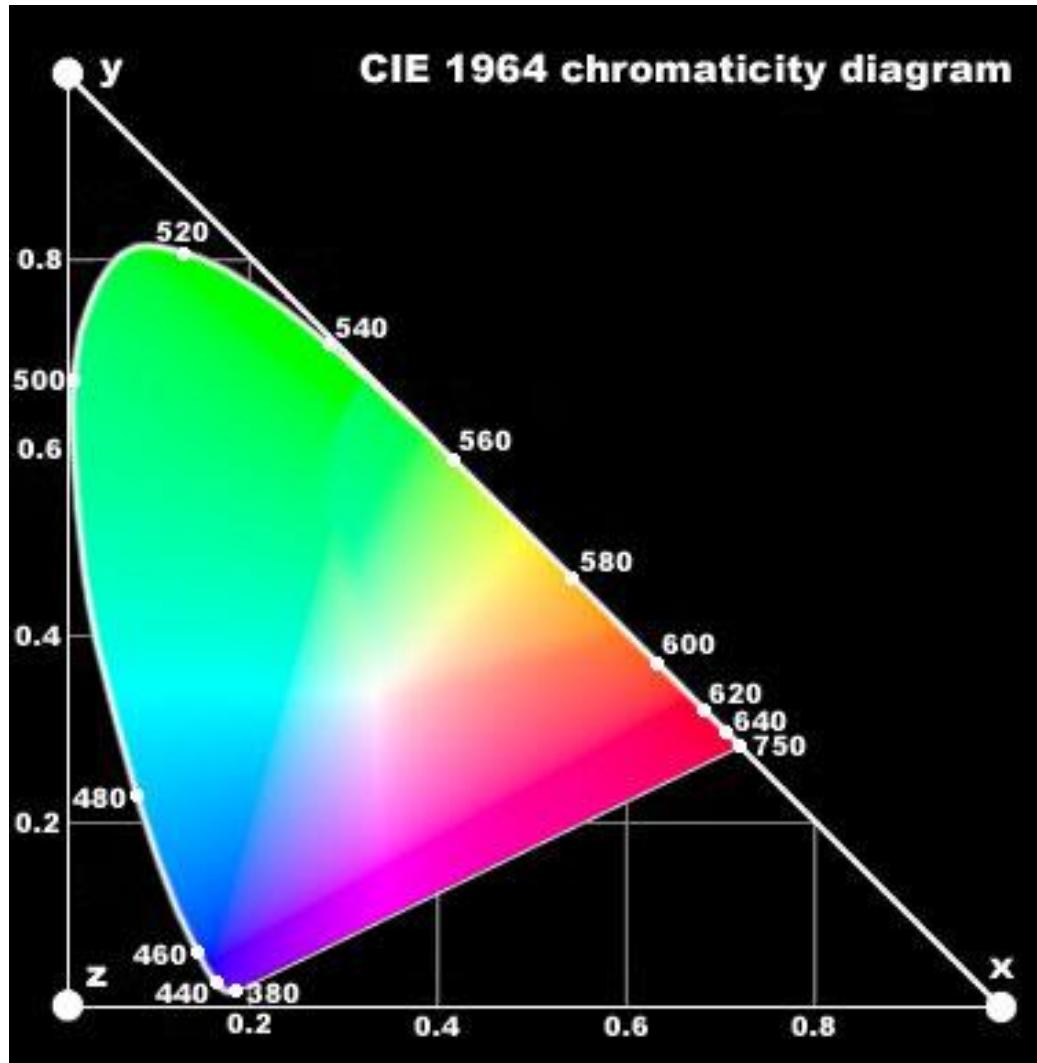


Standard observer response
accounting for the cone cell
densities in a solid angle

Actual Visible Colours



The CIE Diagram 1964 standard



Convex Shape

Notice that the pure colours (coherent λ) are round the edge of the CIE diagram.

The shape must be convex, since any blend (interpolation) of pure colours should create a colour in the visible region.

The line joining purple and red has no pure equivalent.
The colours can only be created by blending.

Intensities

Since the colours are all normalised there is no representation of intensity.

By changing the intensity perceptually different colours can be seen.

White Point

When the three colour components are equal, the colour is white:

$$x = 0.33$$

$$y = 0.33$$

This point is clearly visible on the CIE diagram

Saturation

Pure colours are called fully saturated.

These correspond to the colours around the edge of the horseshoe.

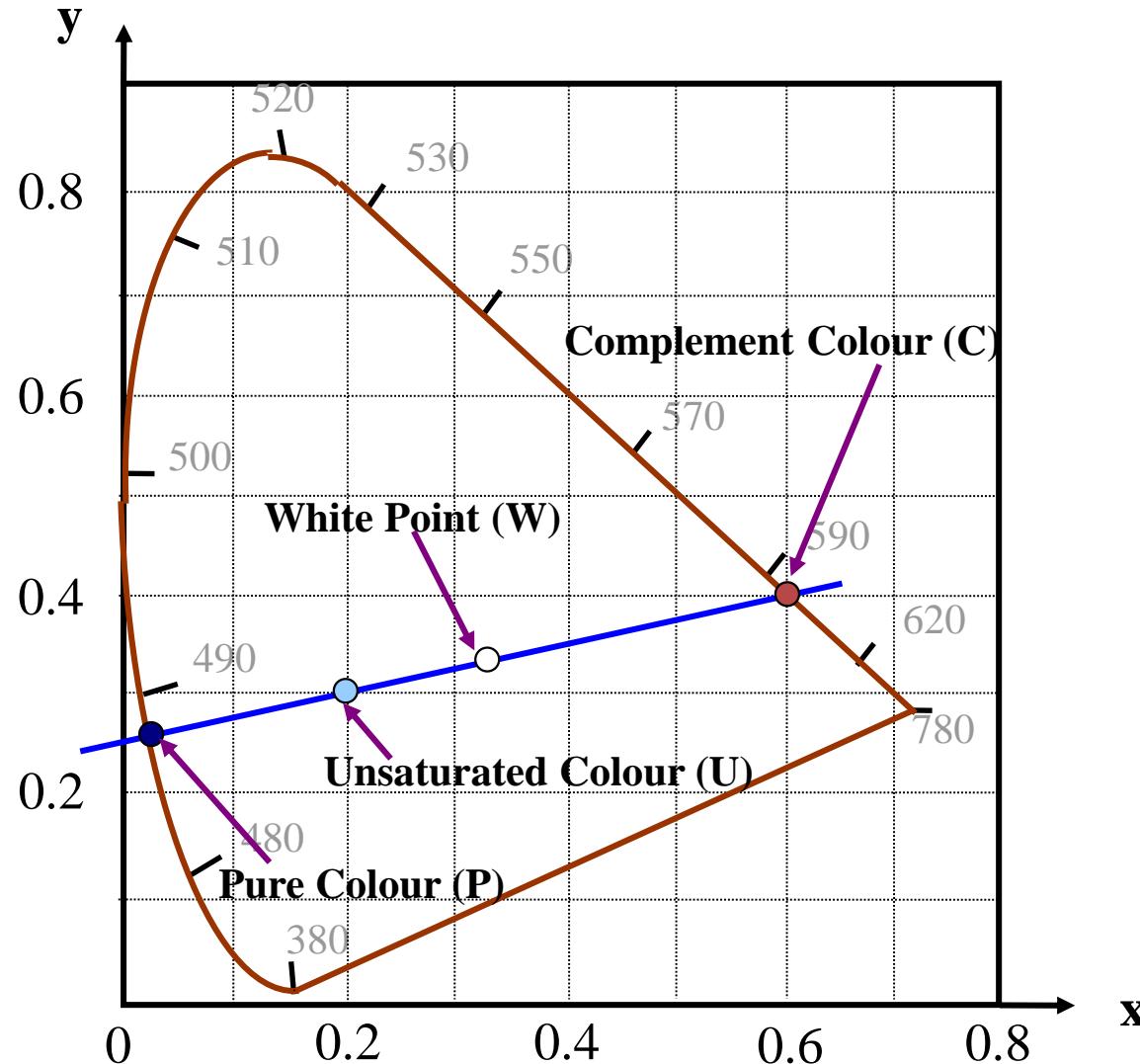
Saturation of a arbitrary point is the ratio of its distance to the white point over the distance of the white point to the edge.

Complement Colour

The complement of a fully saturated colour is the point diametrically opposite through the white point.

A colour added to its complement gives us white.

Actual Visible Colours



Subtractive Primaries

When printing colour we use a subtractive representation.

Inks absorb wavelengths from the incident light, hence they subtract components to create the colour.

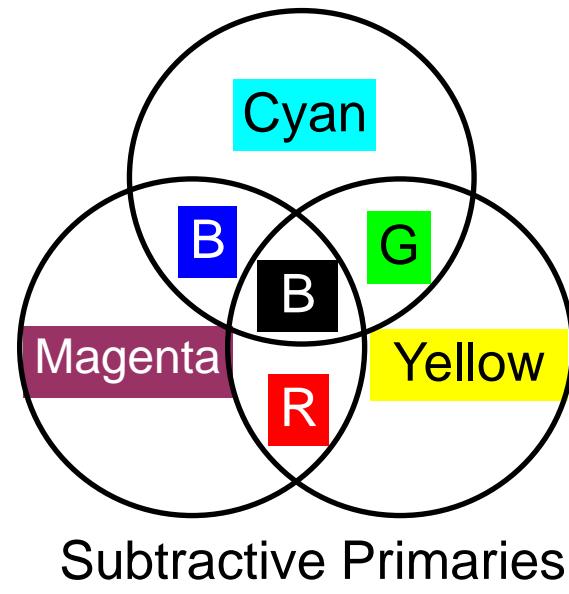
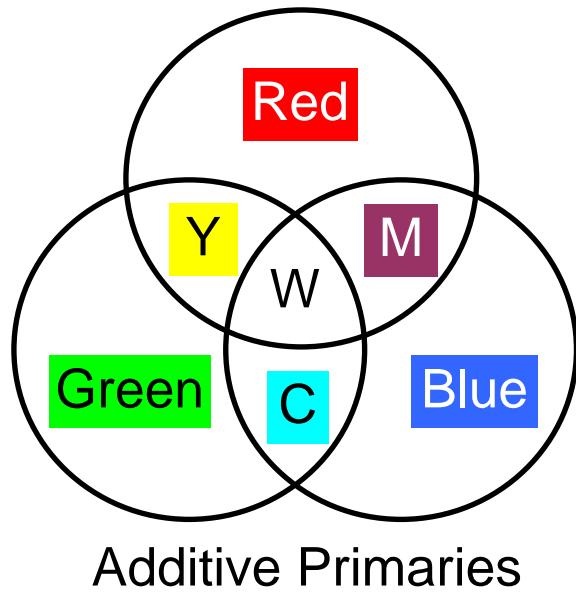
The subtractive primaries are

Magenta (purple)

Cyan (light Blue)

Yellow

Additive and Subtractive Primaries



Colour Perception

Perceptual tests suggest that humans can distinguish:

128 different hues

For each hue around 30 different saturation.

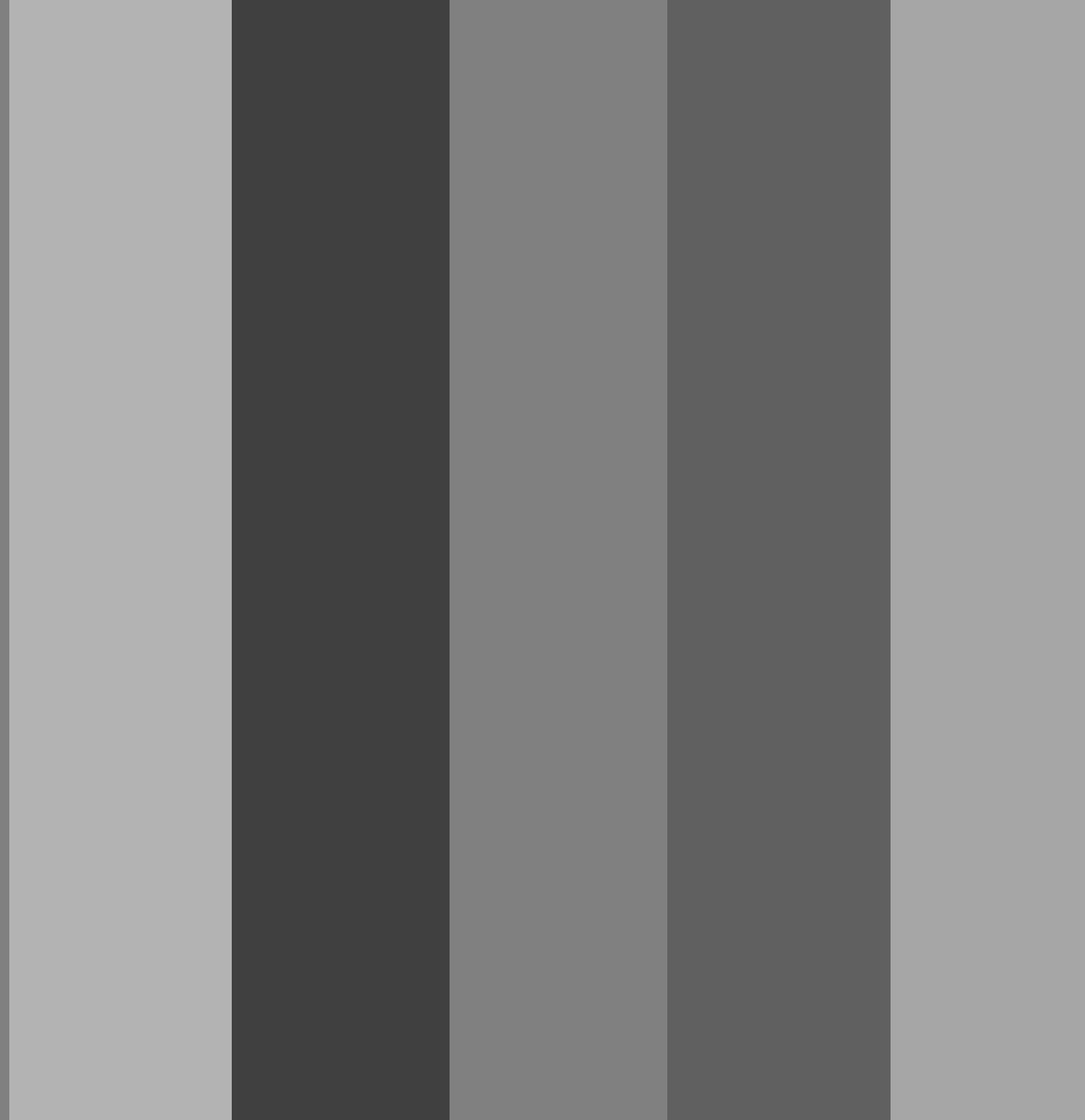
60 and 100 different brightness levels.

If we multiply these three numbers, we get approximately 350,000 different colours.

Colour Perception

These figures must be treated with caution since there seems to be a much greater sensitivity to differentials in colour.

Never the less, a representation with 24 bits (8 bits for red, 8 bits for green and 8 bits for blue does provide satisfactory results.



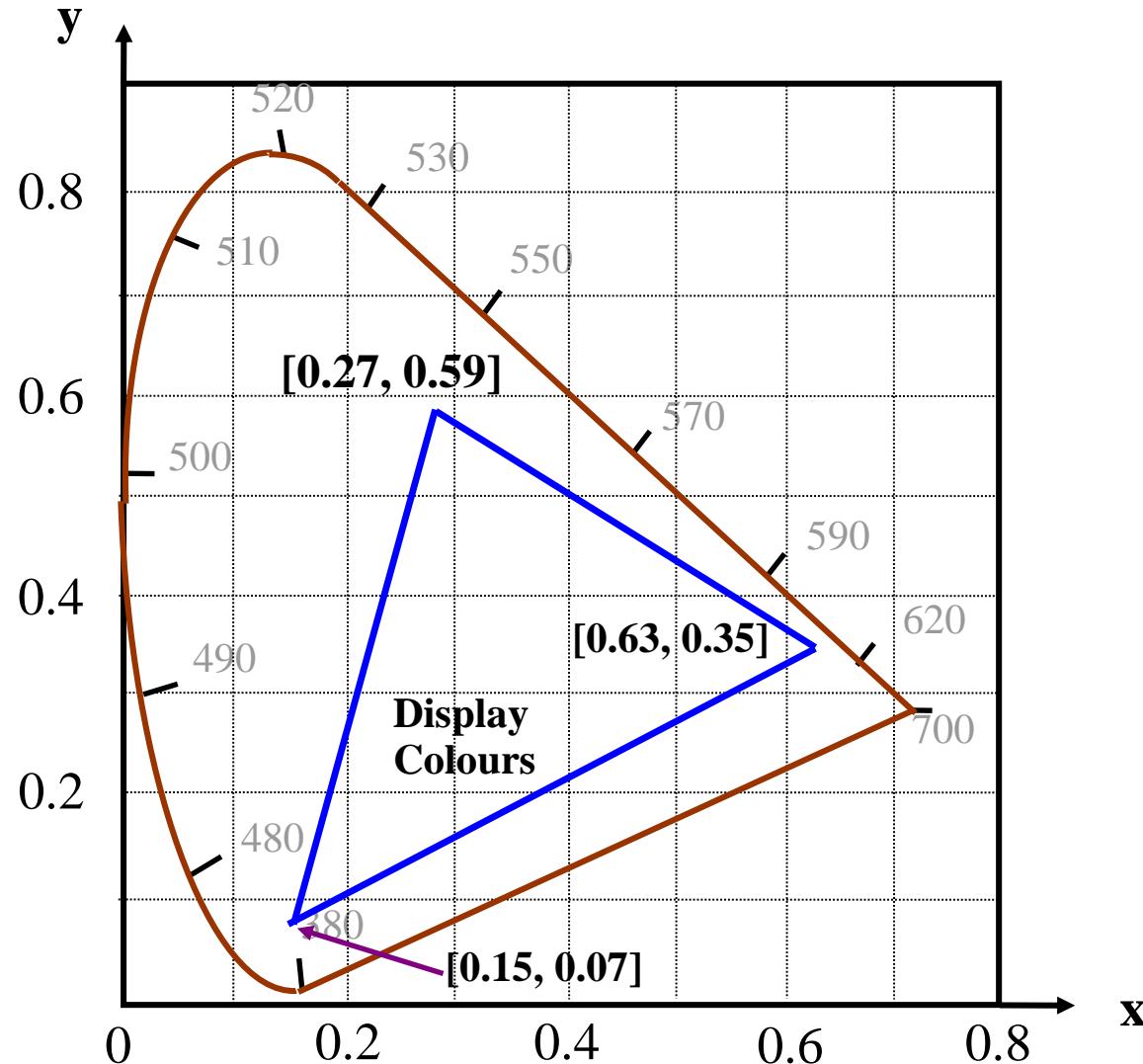
Reproducible colours

Colour monitors are based on adding three the output of three different light emitting phosphors or diodes.

The nominal position of these on the CIE diagram is given by:

	x	y	z
Red	0.628	0.346	0.026
Green	0.268	0.588	0.144
Blue	0.150	0.07	0.780

Actual Visible Colours



RGB to CIE

The monitor RGB representation is related to the CIE colours by the equation:

$$(x, y, z) = \begin{pmatrix} 0.628 & 0.268 & 0.15 \\ 0.346 & 0.588 & 0.07 \\ 0.026 & 0.144 & 0.78 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

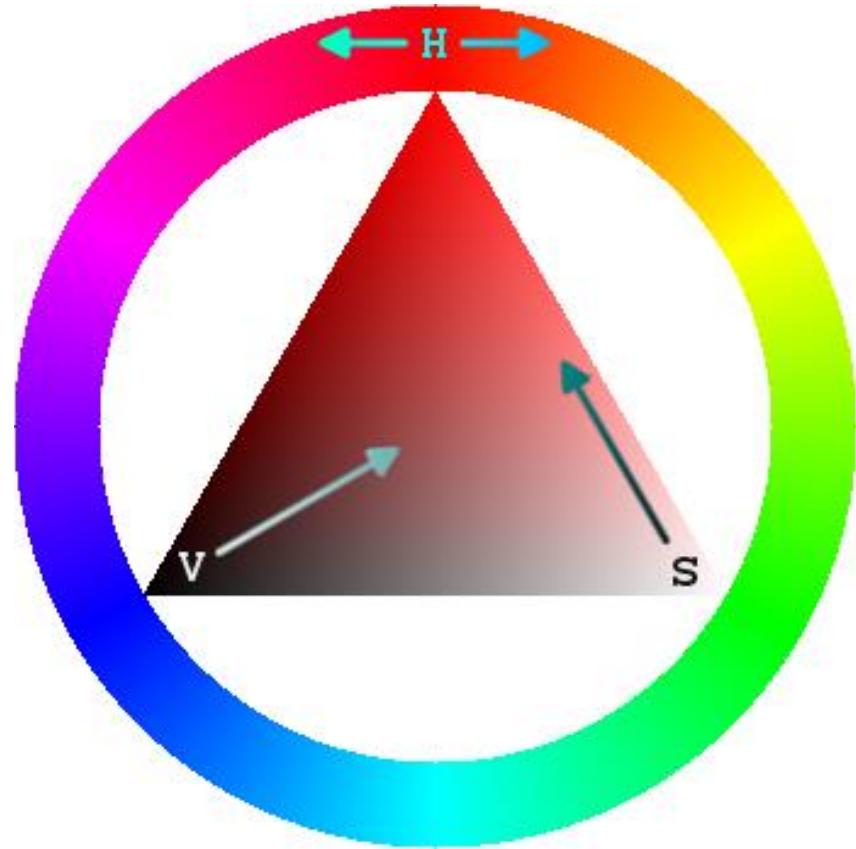
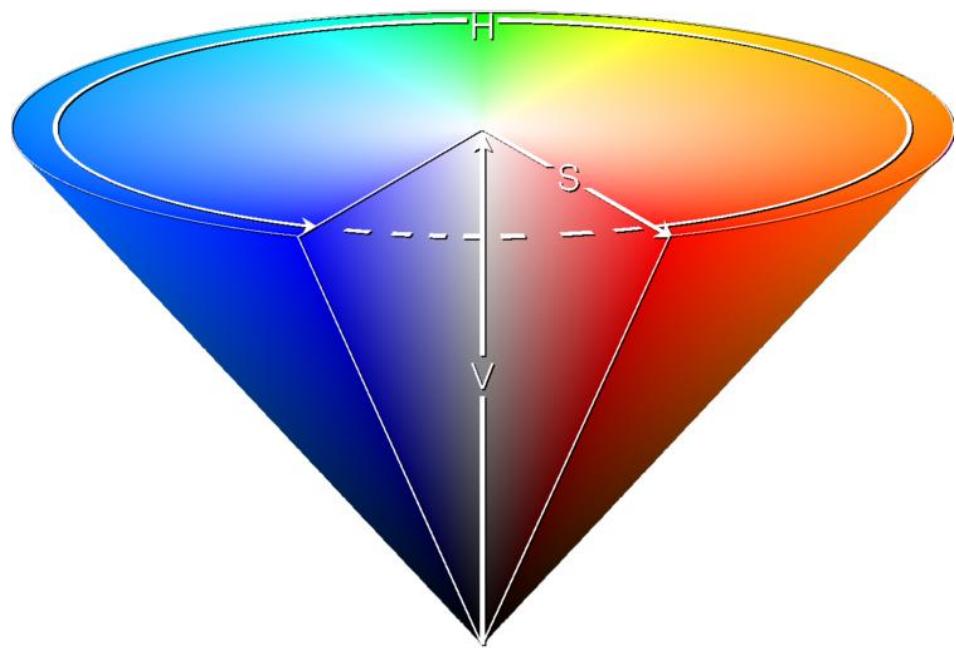
HSV Colour representation

The RGB and CIE systems are practical representations, but do not relate to the way we perceive colours.

For interactive image manipulation it is preferable to use the HSV (or HSI) representation. HSV has three values per colour:

- Hue - corresponds notionally to pure colour.
- Saturation - The proportion of pure colour
- Value - the brightness (Sometimes called Intensity (I))

Visualising the Perceptual Colour Space



Conversion between RGB and HSV

$$V = \max(r, g, b)$$

$$S = (\max(r, g, b) - \min(r, g, b)) / \max(r, g, b)$$

Hue (which is an angle between 0 and 360°) is best described procedurally

Calculating hue

if ($r=g=b$) Hue is undefined, the colour is black, white or grey.

if ($r>b$) and ($g>b$) $Hue = 120 * (g-b) / ((r-b)+(g-b))$

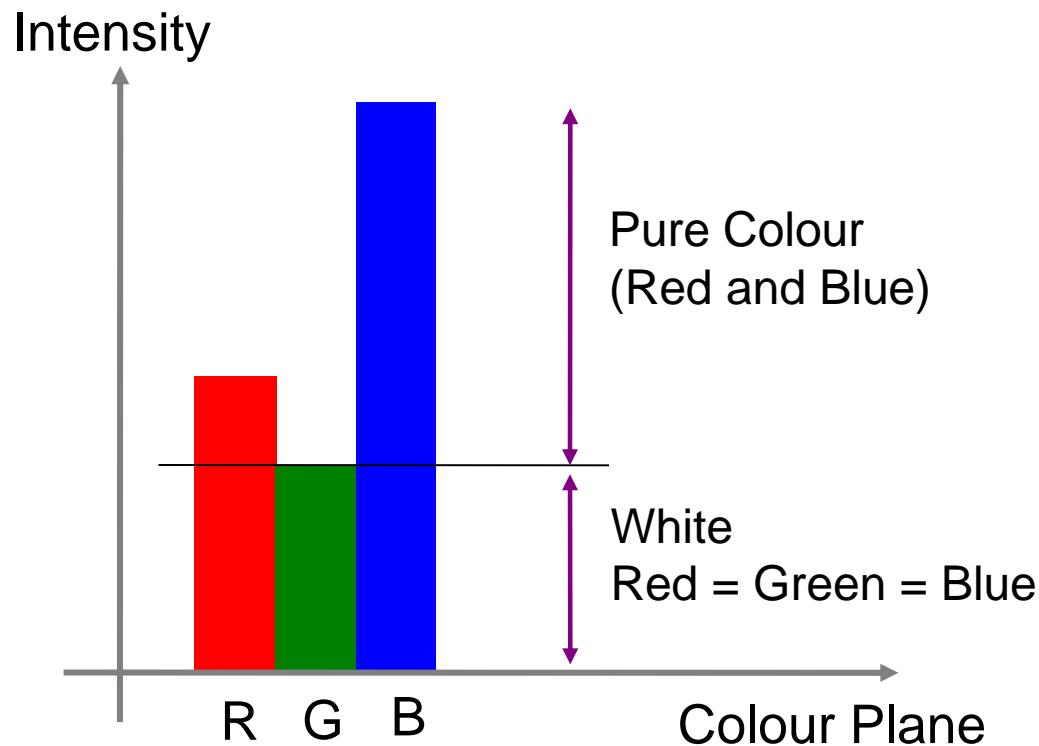
if ($g>r$) and ($b>r$) $Hue = 120 + 120 * (b-r) / ((g-r)+(b-r))$

if ($r>g$) and ($b>g$) $Hue = 240 + 120 * (r-g) / ((r-g)+(b-g))$

Saturation in the RGB system

- In the RGB system we can treat each point as a mixture of pure colour and white.
- Note however that the so called pure colours are not coherent wavelengths as in the CIE diagram

The composition of a tri-stimulus colour



Alpha Channels

- Colour representations in computer systems sometimes use four components - r g b α .
- The fourth is simply an attenuation of the intensity which:
 - allows greater flexibility in representing colours.
 - avoids truncation errors at low intensity
 - allows convenient masking certain parts of an image.

Alpha Channels

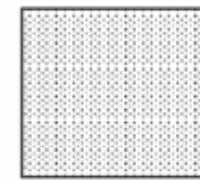
- Images are represented by quadruples:
 - R, G, B indicating color
 - Alpha channel encodes pixel coverage information
 - $\alpha = 0$ transparent
 - $0 < \alpha < 1$ semi-transparent
 - $\alpha = 1$ opaque

- Example: $\alpha = 0.3$



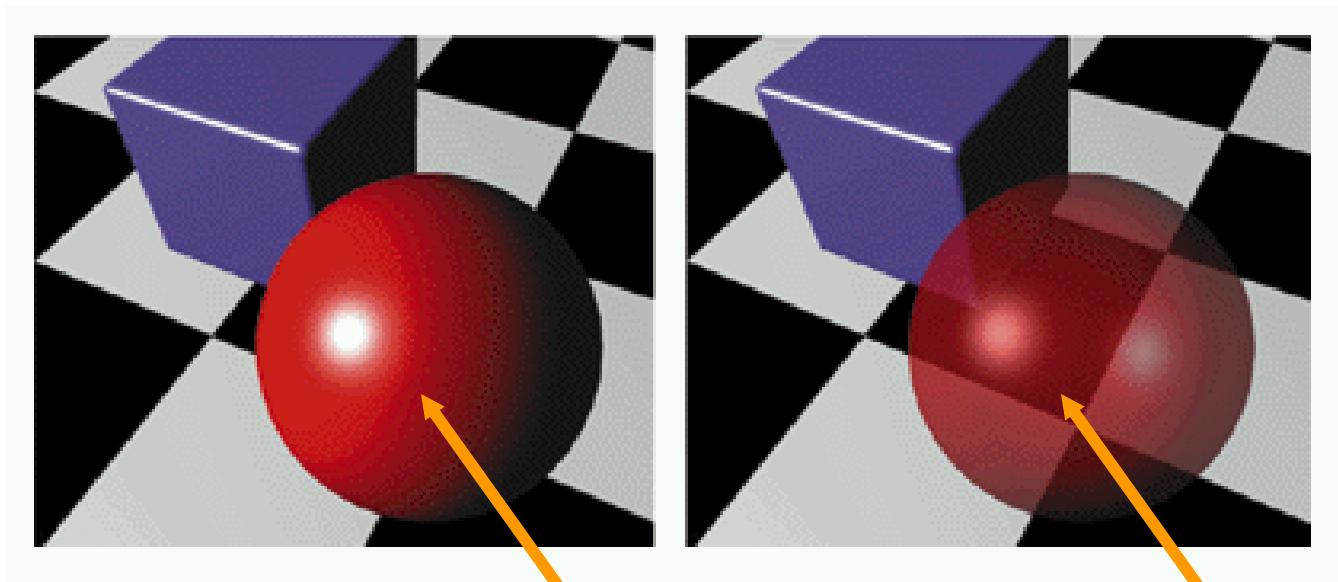
Partial
Coverage

or



Semi-
Transparent

Image Combination: Alpha channel blending



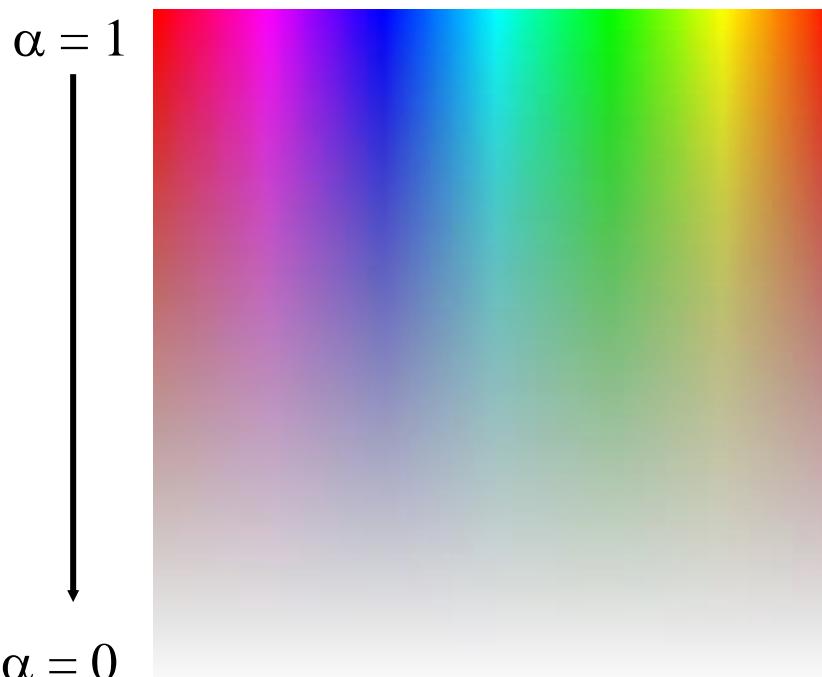
$\alpha = 1$

$\alpha = 0.5$

Image Combination: Alpha channel blending

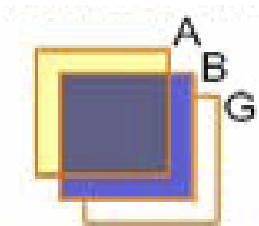
- Convention:
 - RGBA represents a pixel with color $C = (R, G, B)$ as

$$C = (ar, ag, ab, a)$$



Alpha Channels

- Suppose we put A over B over background G



- How much of B is blocked by A?

$$\alpha A$$

- How much of B shows through A?

$$(1 - \alpha A)$$

- How much of G shows through both A and B?

$$(1 - \alpha A) (1 - \alpha B)$$

Interactive Computer Graphics: Lecture 8

Texture mapping

Some slides adopted from
H. Pfister, Harvard

The Problem:



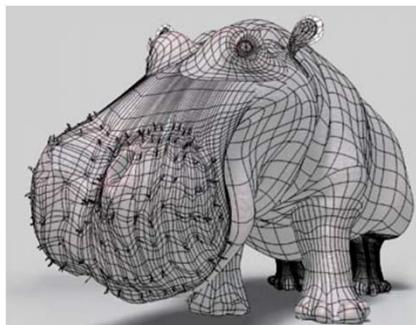
- We don't want to represent all this detail with geometry

Graphics Lecture 8: Slide 2

The Solution: Textures

- The visual appearance of a graphics scene can be greatly enhanced by the use of texture.
- Consider a brick building, using a polygon for every brick require a huge effort in scene design.
- So why not use one polygon and draw a repeating brick pattern (texture) onto it?

The Quest for Visual Realism



Graphics Lecture 8: Slide 4

Texture Definition

- Textures may be defined as:
 - One-dimensional functions
 - parameter can have arbitrary domain
(e.g., incident angle)
 - Two-dimensional functions
 - information is calculated for every (u, v) , many possibilities
 - Raster images (“texels”)
 - Most often used method
 - Images from scanner, photos or calculation
 - Three-dimensional functions
 - Volume $T(u, v, w)$
- Procedural texture vs. raster data

Procedural textures

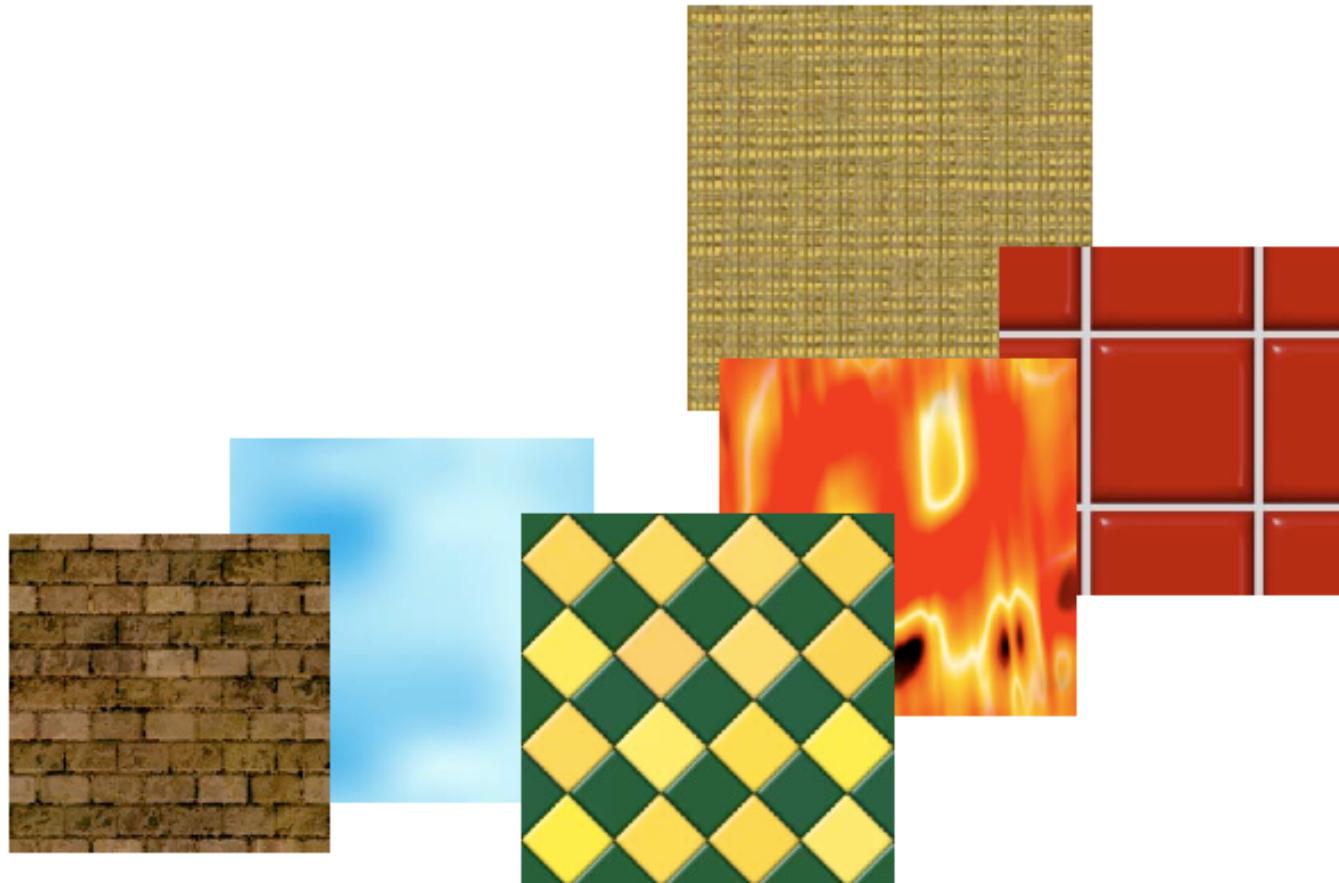
- Write a function: $F(\mathbf{p}) \rightarrow \text{color}$



- non-intuitive
- difficult to match a texture that already exists in the ‘real’ world

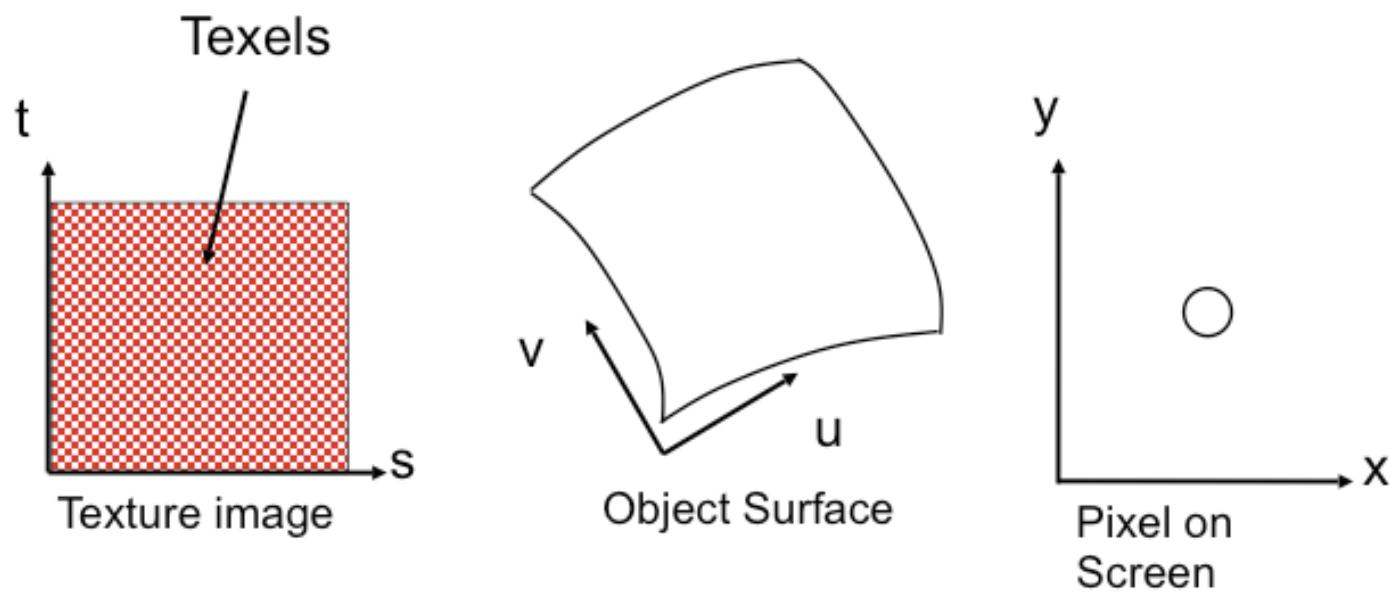
Graphics Lecture 8: Slide 6

Photo textures



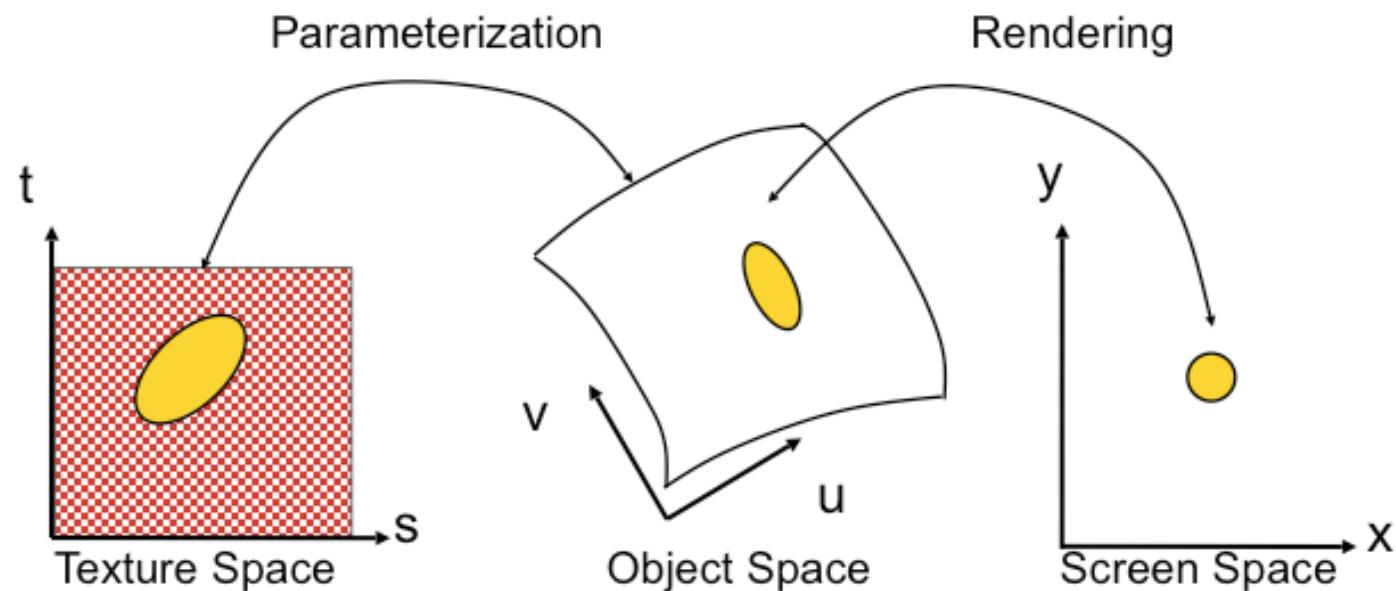
Graphics Lecture 8: Slide 7

The concept of texture mapping



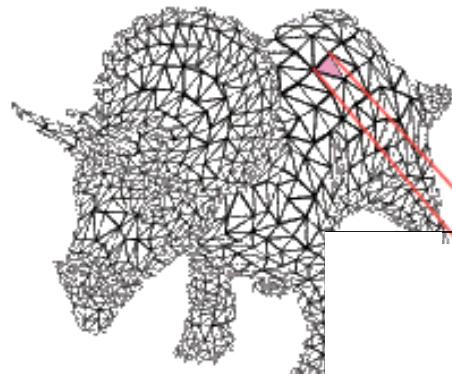
Graphics Lecture 8: Slide 8

Texture mapping: Terminology

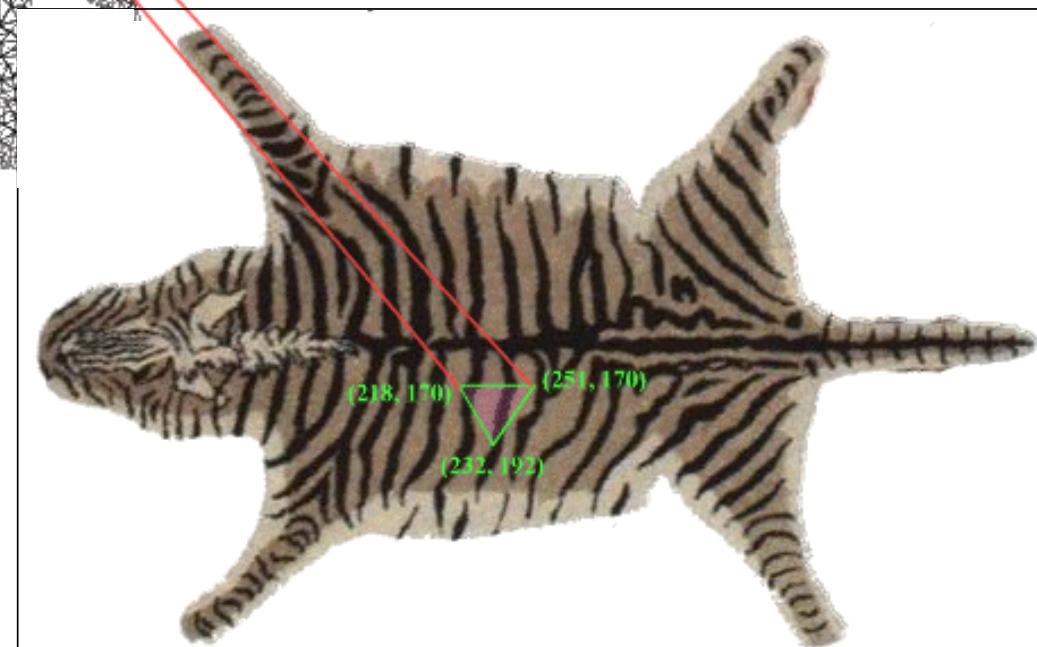


Graphics Lecture 8: Slide 9

The concept of texture mapping



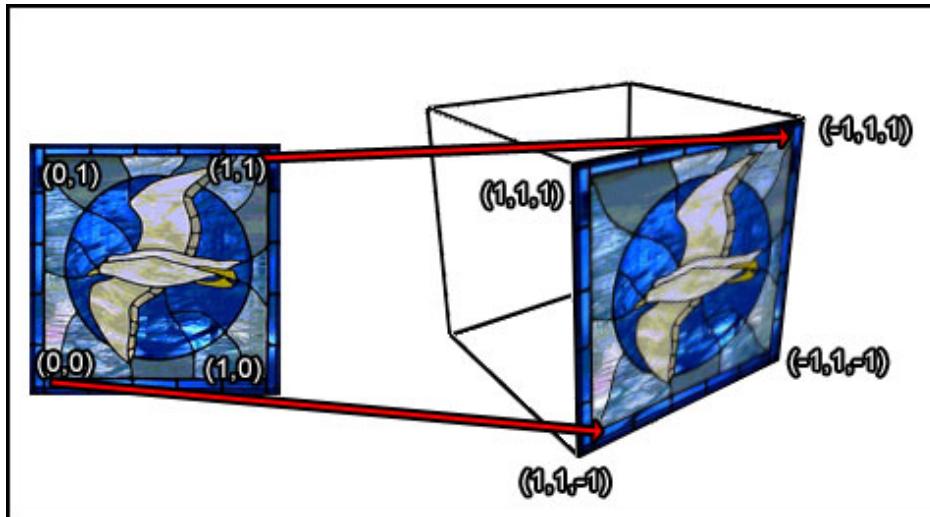
For each triangle/polygon in the model establish a corresponding region in the texture



During rasterization interpolate the coordinate indices into the texture map

Parametrization

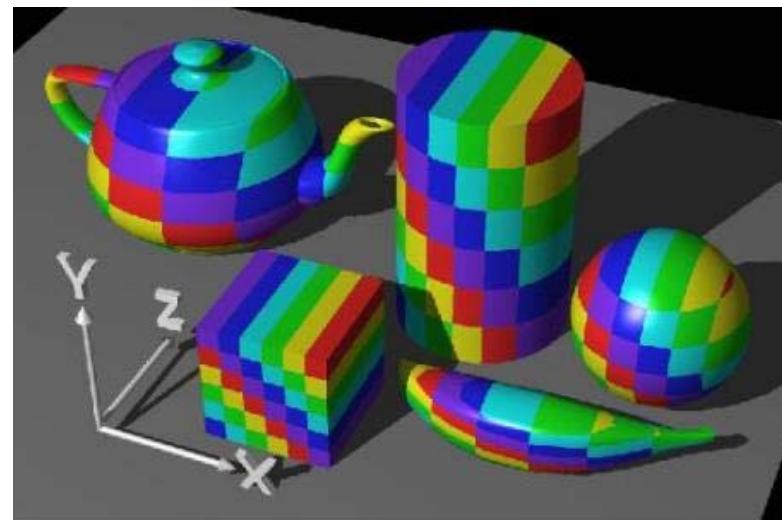
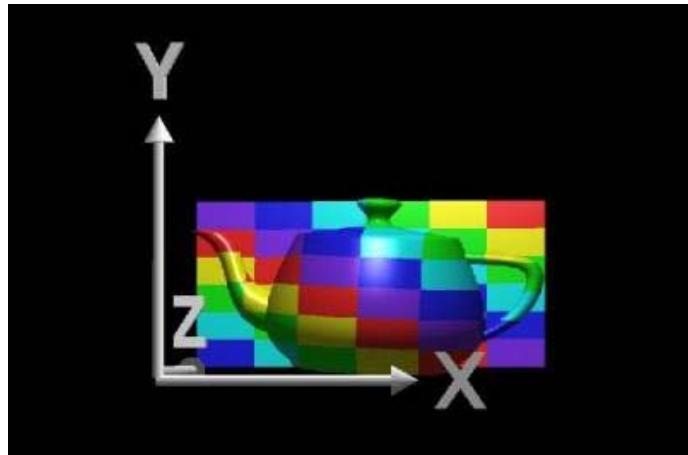
- How to do the mapping?



- Usually objects are not that simple

Parametrization: Planar

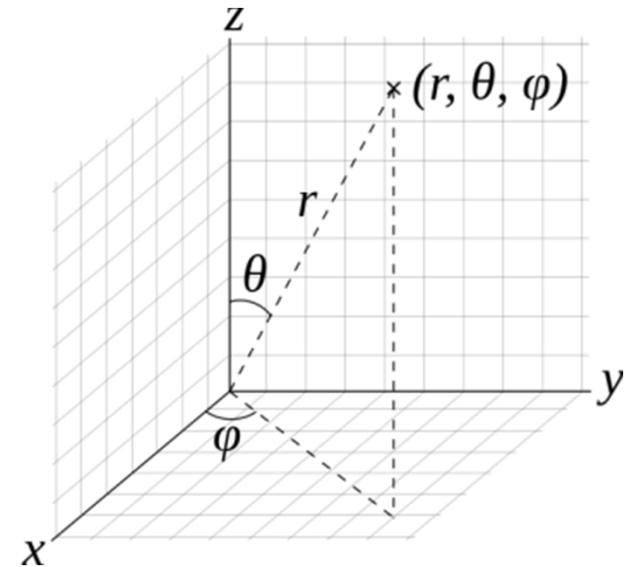
- Planar mapping: dump one of the coordinates



Only looks good from the front!

Parametrization

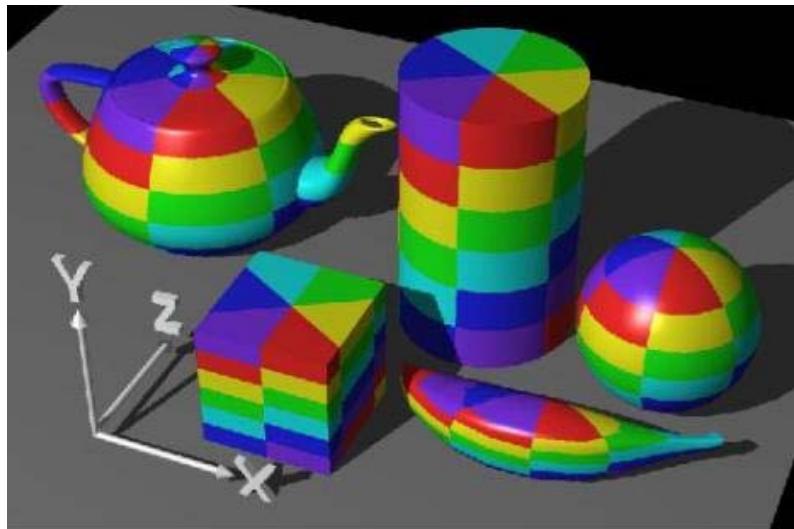
- Cylindrical and spherical mapping: compute angles between vertex and object center
- Compare to polar/spherical coordinate systems



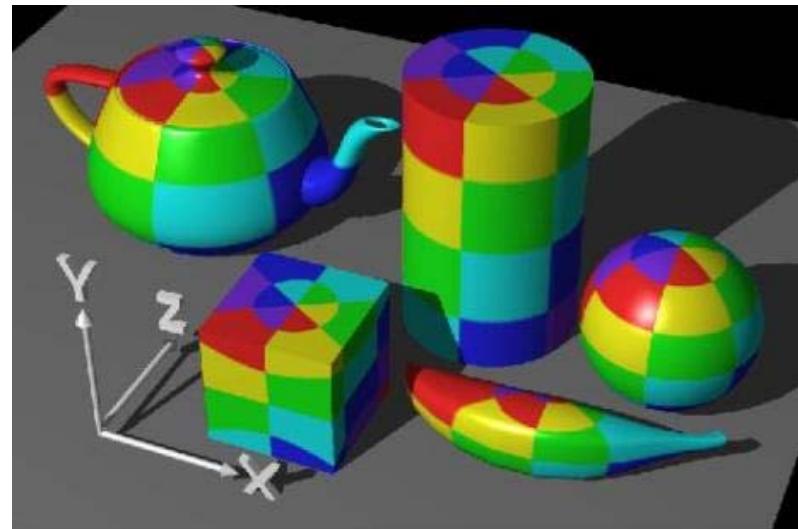
Graphics Lecture 8: Slide 13

Parametrization

- Cylindrical and spherical mapping



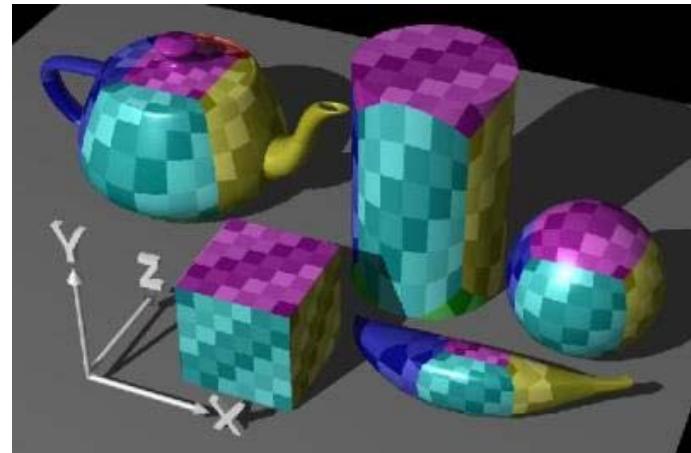
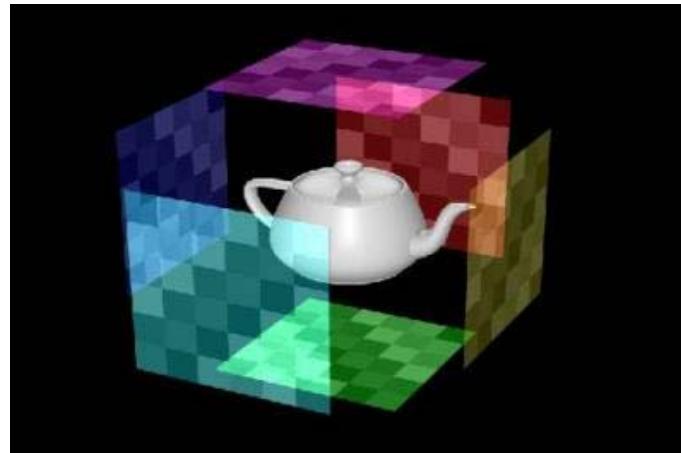
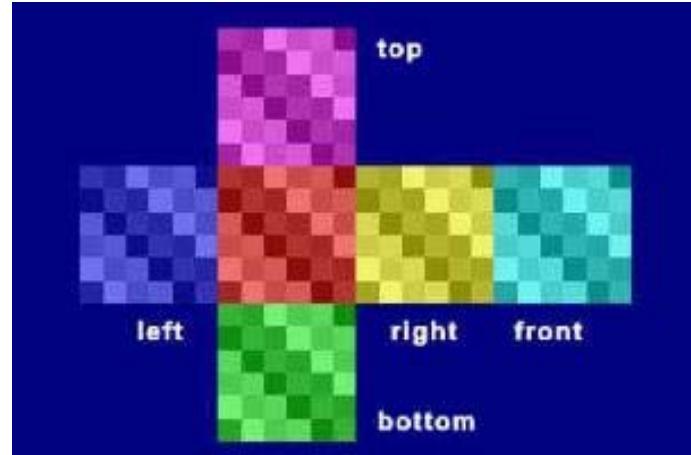
Cylindrical



Spherical

Parametrization: Box

- Box mapping: used mainly for environment mapping (see later)



Graphics Lecture 8: Slide 15

Parametrization

- Manual mapping using CAD Software
 - “Unwrapping”

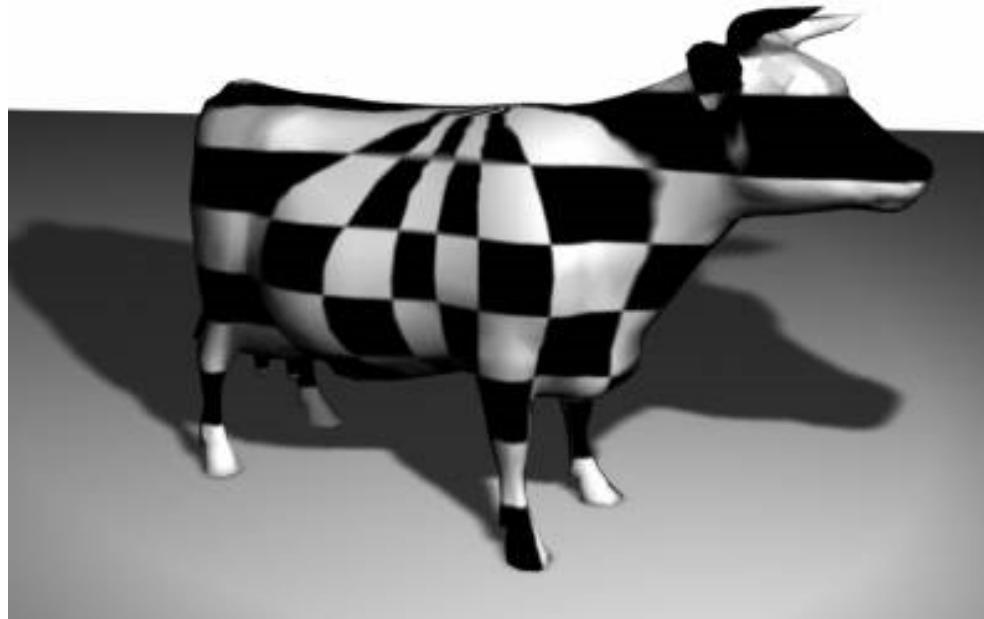


Images by Martin Kenzel

Graphics Lecture 8: Slide 16

Parameterization problems

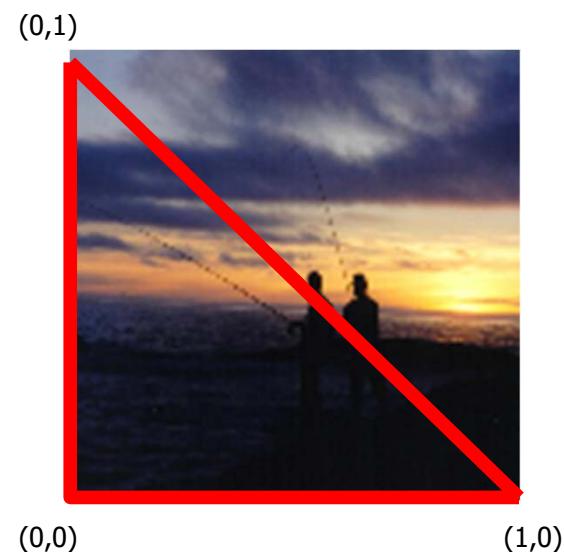
- All mappings have distortions and singularities
- Often they need to be fixed manually (CAD software)



Graphics Lecture 8: Slide 17

Texture Coordinates

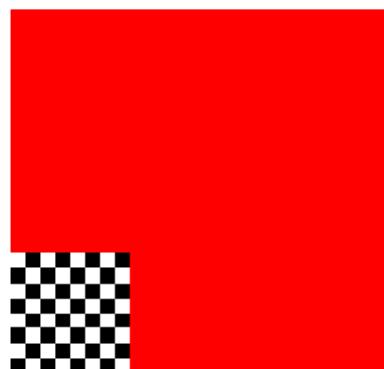
- Specify a texture coordinate at each vertex
- Canonical texture coordinates $(0,0) \rightarrow (1,1)$
- Often the texture size is a power of 2
(but it doesn't have to be)
- How can we tile this texture?



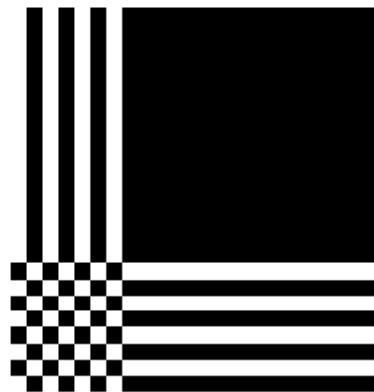
Graphics Lecture 8: Slide 18

Texture Addressing

- What happens outside $[0,1]$?
- Border, repeat, clamp, mirror
- Also called texture addressing modes

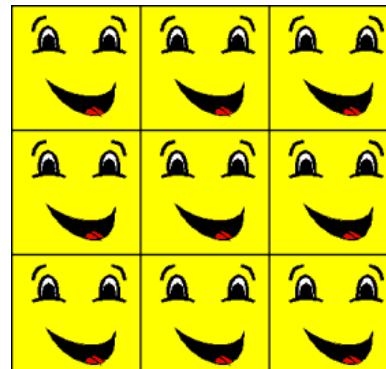


Texture with red border applied to primitive

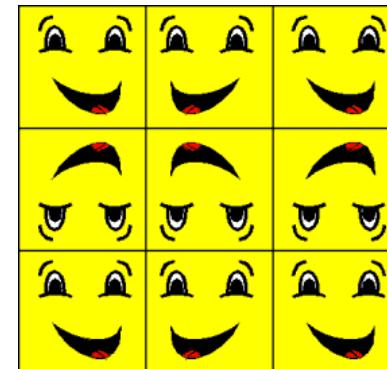


Clamped texture applied to primitive

Static color

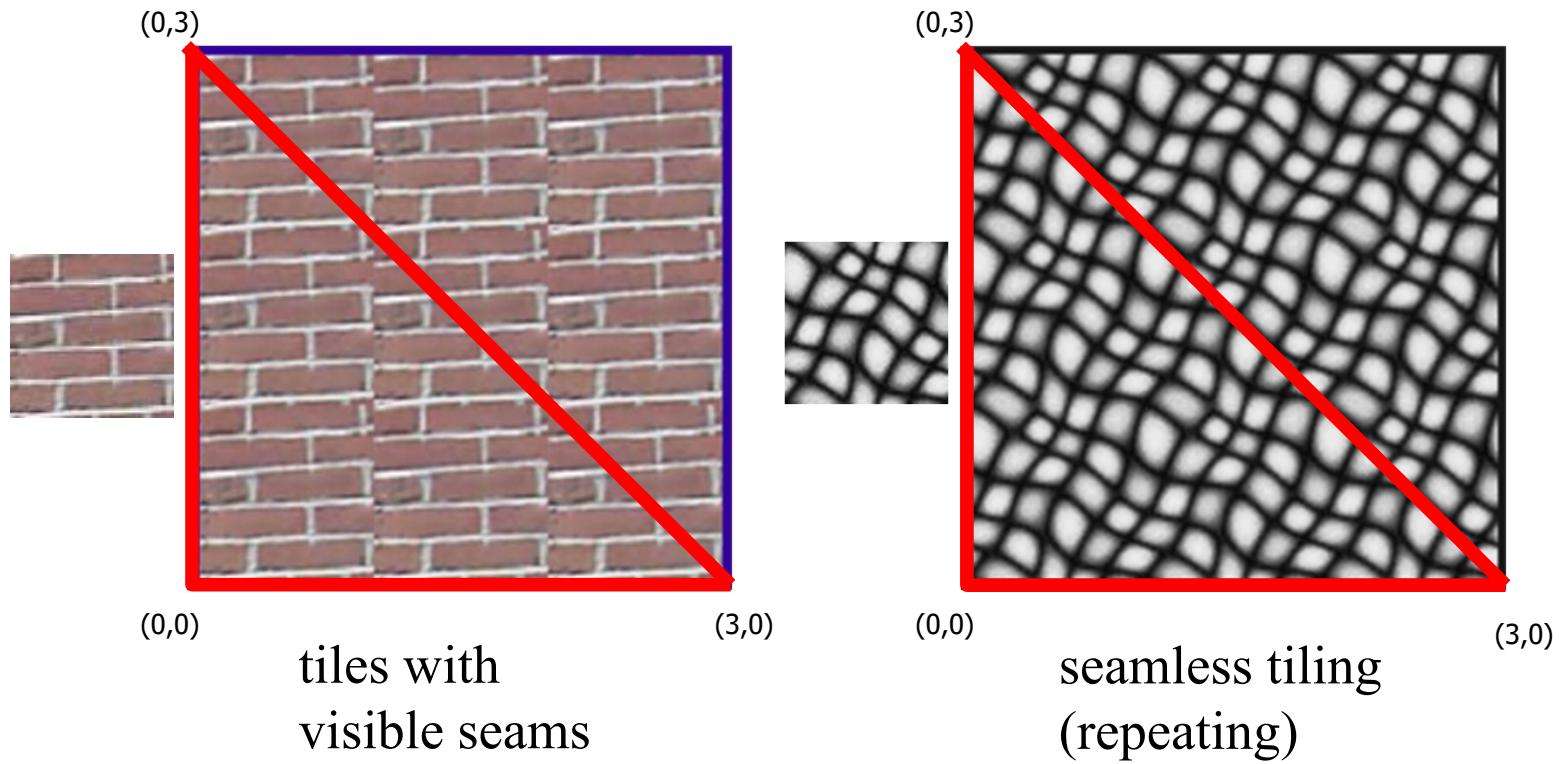


Repeated



Mirrored

Tiling Texture



Graphics Lecture 8: Slide 20

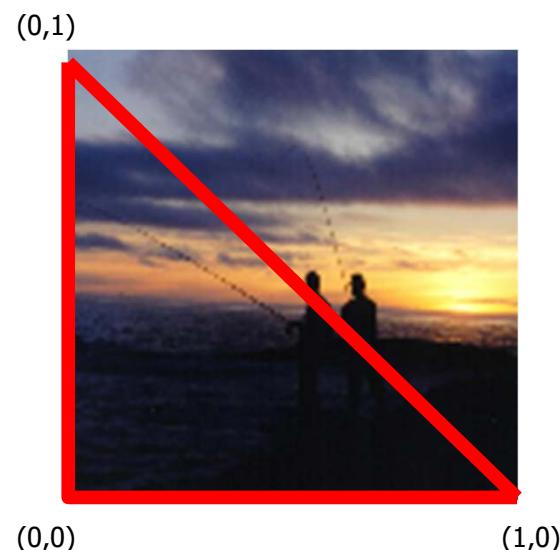
Texture synthesis



Graphics Lecture 8: Slide 21

Texture Coordinates

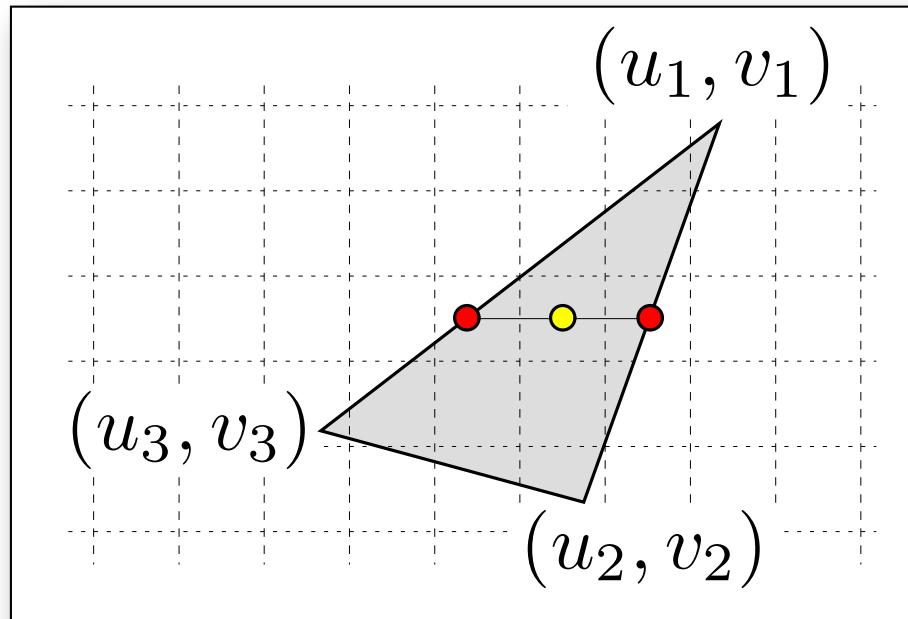
- Specify a texture coordinate at each vertex
- Canonical texture coordinates $(0,0) \rightarrow (1,1)$
- Linearly interpolate the values in screen space



Graphics Lecture 8: Slide 22

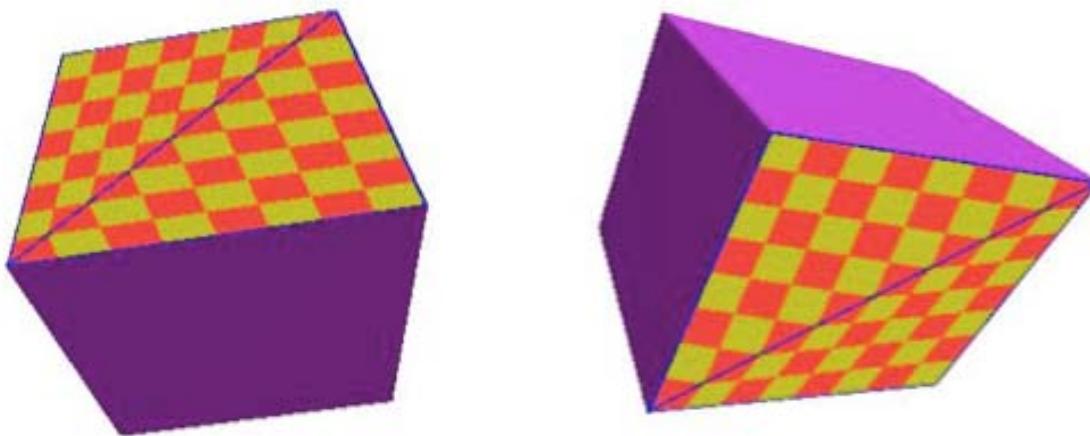
Mapping texture to individual pixels

- Interpolate texture coordinates across scanlines
- Same as Gouraud shading but now for texture coordinates not shading values



Graphics Lecture 8: Slide :

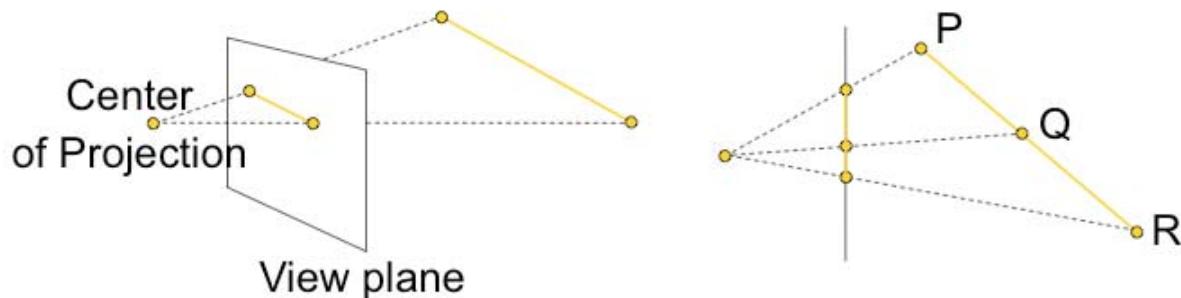
What goes wrong when we linearly interpolate texture coordinates?



- Notice the distortion along the diagonal triangle edge of the cube face after perspective projection

Perspective projection

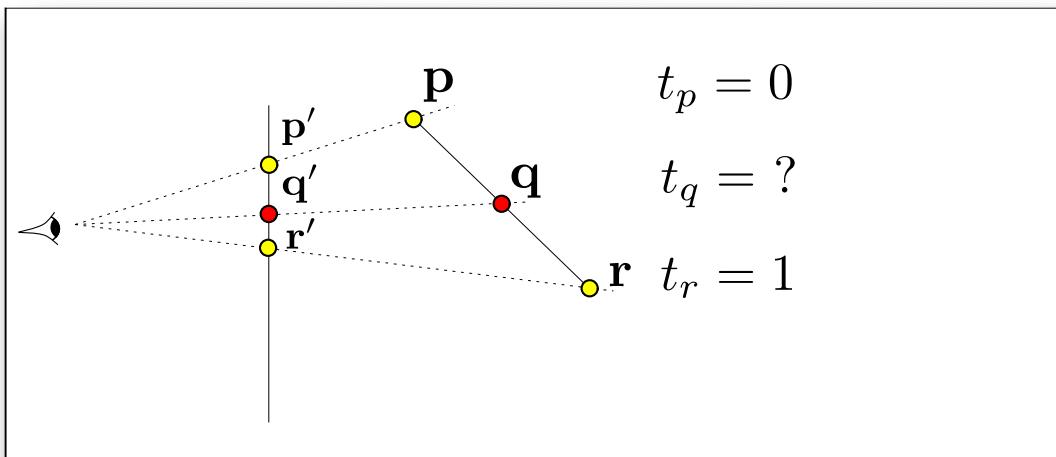
- The problem is that perspective projection does not preserve linear combinations of points!
- In particular; equal distances in 3D space **do not** map to equal distances in screen space



- Linear interpolation in screen space is not the same as linear interpolation in 3D space

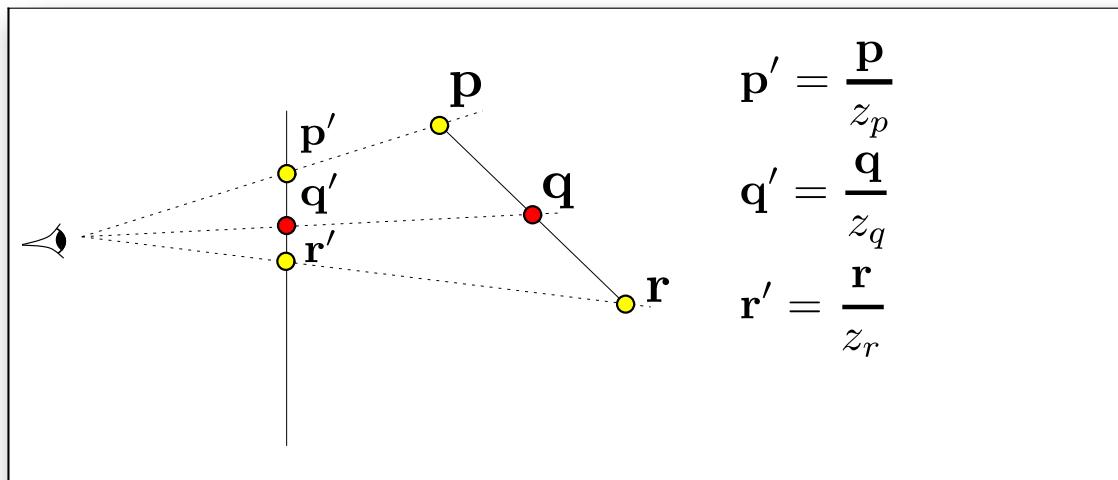
How to fix?

- Assign parameter t to 3-D vertices \mathbf{p} and \mathbf{r}
- t controls linear blend of texture coordinates of \mathbf{p} and \mathbf{r}
- Let $t = 0$ at \mathbf{p} , and $t = 1$ at \mathbf{r}
- Assume for simplicity that the image plane is at $z = 1$ *



How to fix?

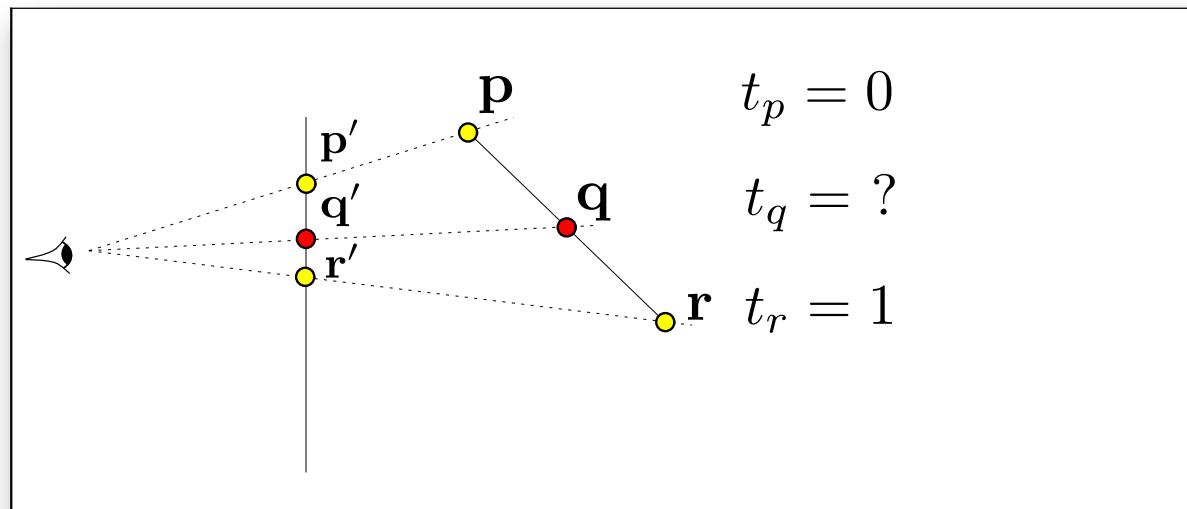
- p projects to p' and r projects to r' (simply divide by z coordinate)
- What value should t have at location q' ?



Graphics Lecture 8: Slide 27

How to fix?

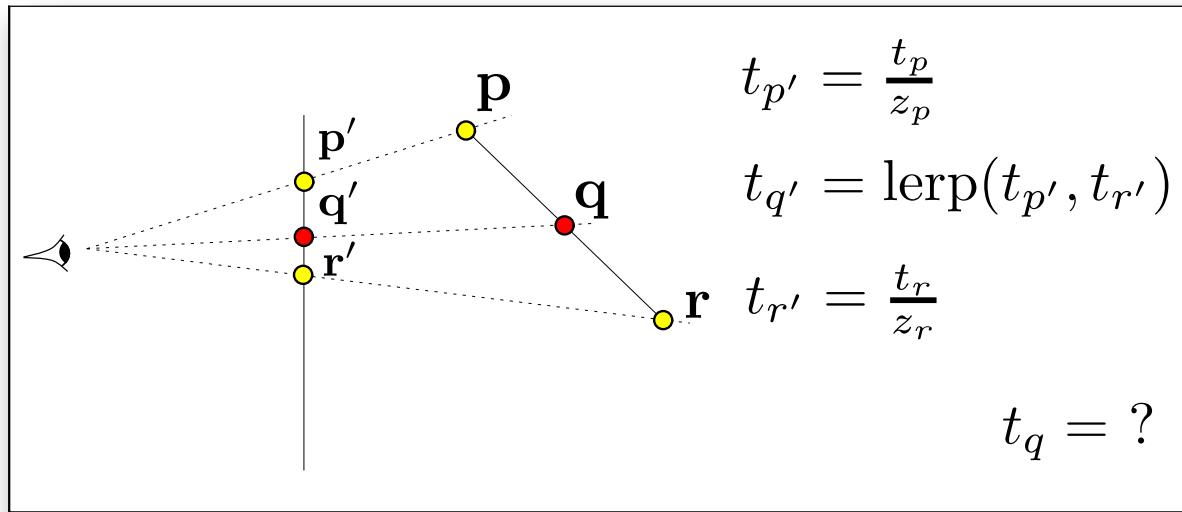
- We cannot linearly interpolate t between p' and r'
- Only projected values can be linearly interpolated in screen space
- Solution: perspective-correct interpolation



Graphics Lecture 8: Slide 28

Perspective Correct Interpolation

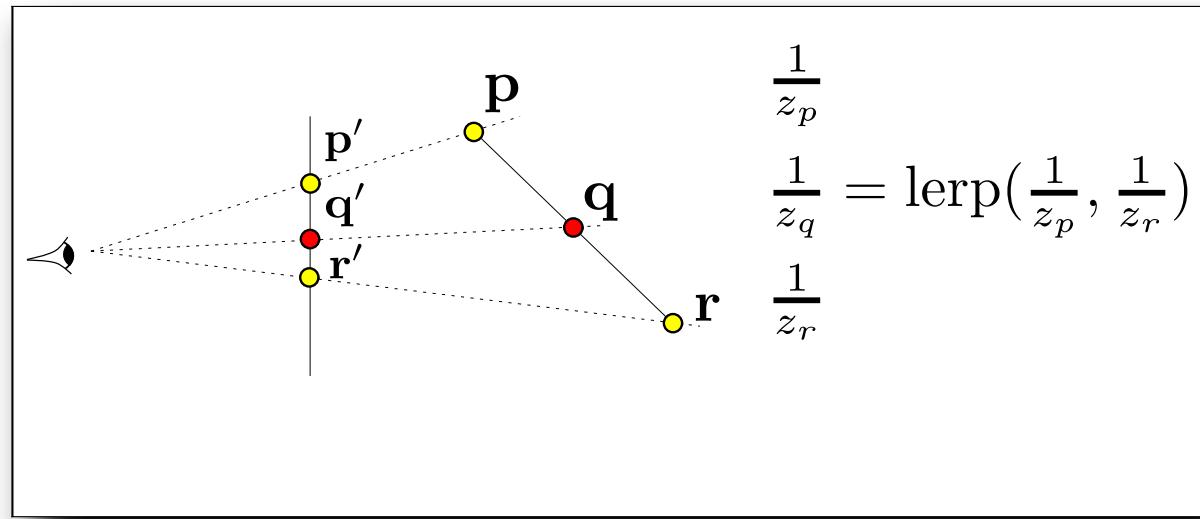
- Linearly interpolate t/z (not t) between \mathbf{p}' and \mathbf{r}' .
 - Compute $t_{\mathbf{p}'} = t_{\mathbf{p}}/z_{\mathbf{p}}$ $t_{\mathbf{r}'} = t_{\mathbf{r}}/z_{\mathbf{r}}$
 - Linearly interpolate (lerp) $t_{\mathbf{p}'}$ and $t_{\mathbf{r}'}$ to get $t_{\mathbf{q}'}$, at location \mathbf{q}'
- But, we want the un-projected parameter $t_{\mathbf{q}}$ (not $t_{\mathbf{q}'}$)



Graphics Lecture 8: Slide 29

Perspective Correct Interpolation

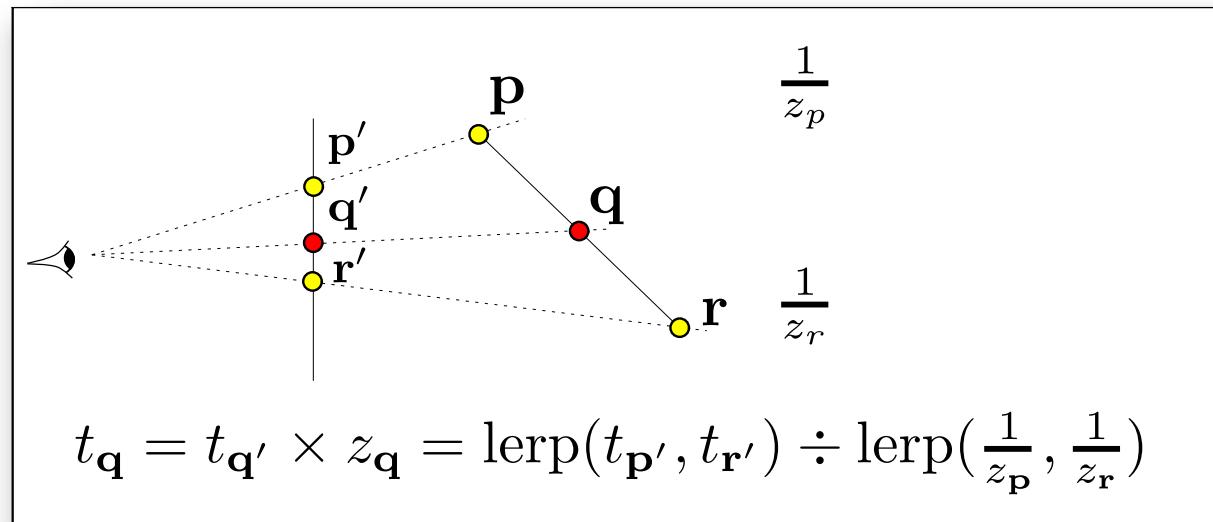
- The parameters t_p , & t_q , are related to t_p & t_q by perspective factors of $1/z_p$ and $1/z_q$
 - lerp $1/z_p$ and $1/z_r$ to obtain $1/z_q$ at point q'



Graphics Lecture 8: Slide 30

Perspective Correct Interpolation

- The parameters t_p , & t_q , are related to t_p & t_q by perspective factors of $1/z_p$ and $1/z_q$
 - lerp $1/z_p$ and $1/z_r$ to obtain $1/z_q$ at point q'
 - Divide t_q , by $1/z_q$ to get t_q



Graphics Lecture 8: Slide 31

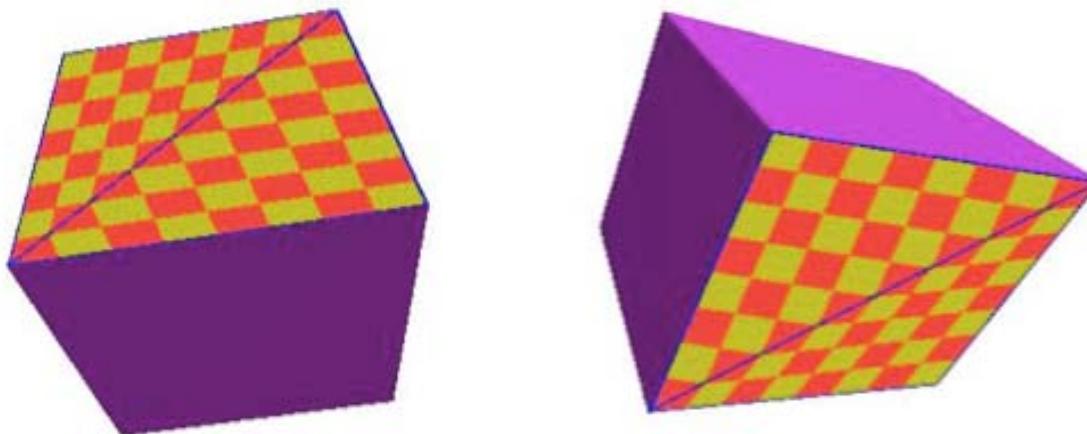
Perspective Correct Interpolation

- Summary:
 - Given texture parameter t at vertices:
 - Compute $1/z$ for each vertex
 - Linearly interpolate $1/z$ across the triangle
 - Linearly interpolate t/z across the triangle
 - Do perspective division:
Divide t/z by $1/z$ to obtain interpolated parameter t

$$t_{\mathbf{q}} = \frac{\text{lerp} \left(\frac{t_{\mathbf{p}}}{z_{\mathbf{p}}}, \frac{t_{\mathbf{r}}}{z_{\mathbf{r}}} \right)}{\text{lerp} \left(\frac{1}{z_{\mathbf{p}}}, \frac{1}{z_{\mathbf{r}}} \right)}$$

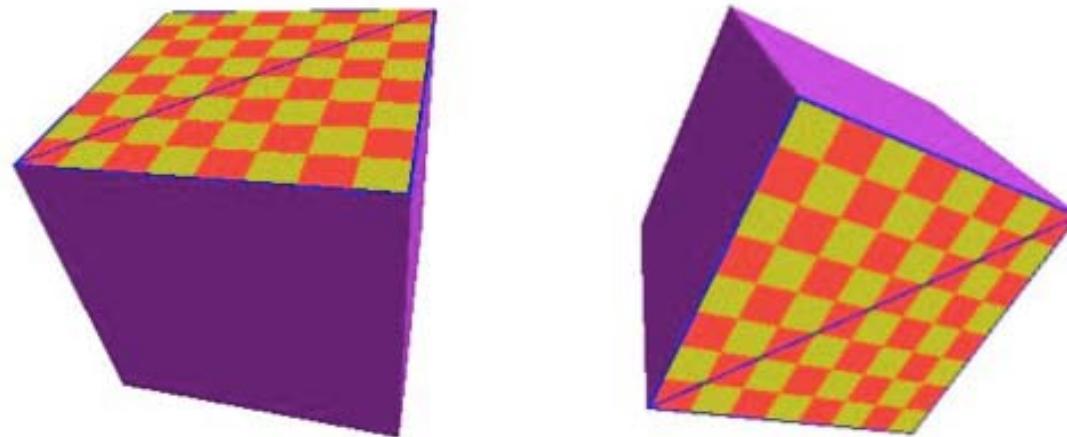
Graphics Lecture 8: Slide 32

What Goes Wrong?



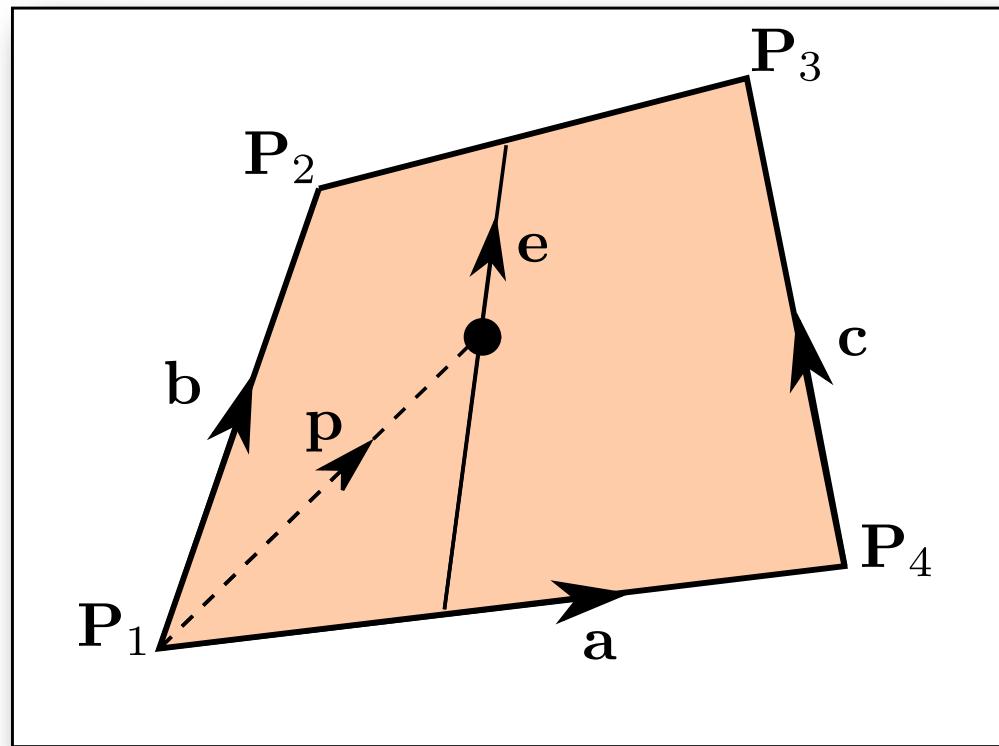
Graphics Lecture 8: Slide 33

Perspective Correct Interpolation



Mapping texture to individual pixels

Alternative



$P_{1..4}$: Polygon vertices

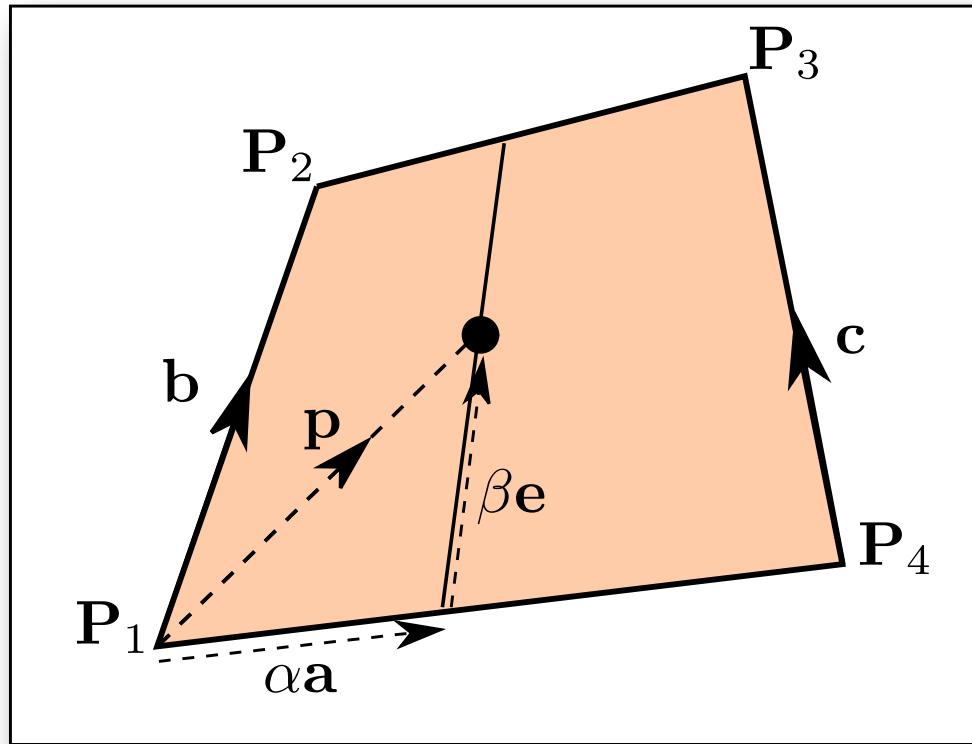
p : Pixel to be textured

Bilinear Texture mapping

Bi-linear Map - Solving for a and b

$$\begin{aligned} \mathbf{p} &= \alpha \mathbf{a} + \beta \mathbf{e} \\ \mathbf{e} &= \mathbf{b} + \alpha (\mathbf{c} - \mathbf{b}) \end{aligned}$$

so



$$\mathbf{p} = \alpha \mathbf{a} + \beta \mathbf{b} + \alpha\beta(\mathbf{c} - \mathbf{b}) \quad \text{Quadratic in the unknowns !}$$

Non Linearities in texture mapping

- The second order term means that straight lines in the texture may become curved when the texture is mapped.
- However, if the mapping is to a parallelogram:

$$\mathbf{p} = \alpha \mathbf{a} + \beta \mathbf{b} + \alpha\beta(\mathbf{c} - \mathbf{b})$$

and

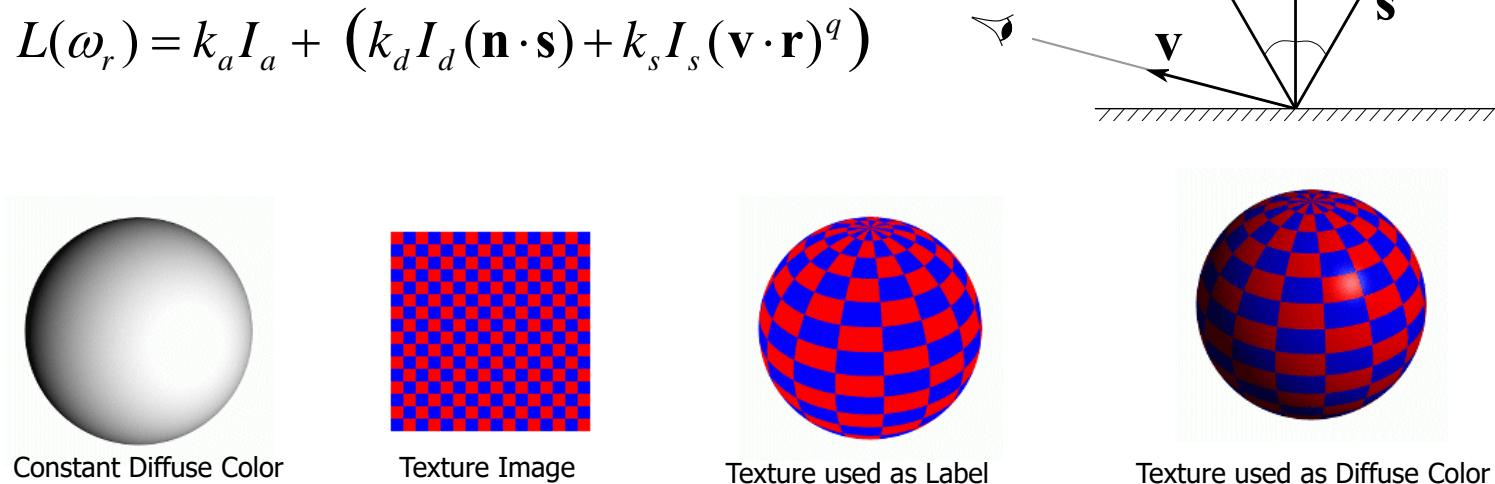
$$\mathbf{b} = \mathbf{c}$$

so

$$\mathbf{p} = \alpha \mathbf{a} + \beta \mathbf{b}$$

Texture Mapping & Illumination

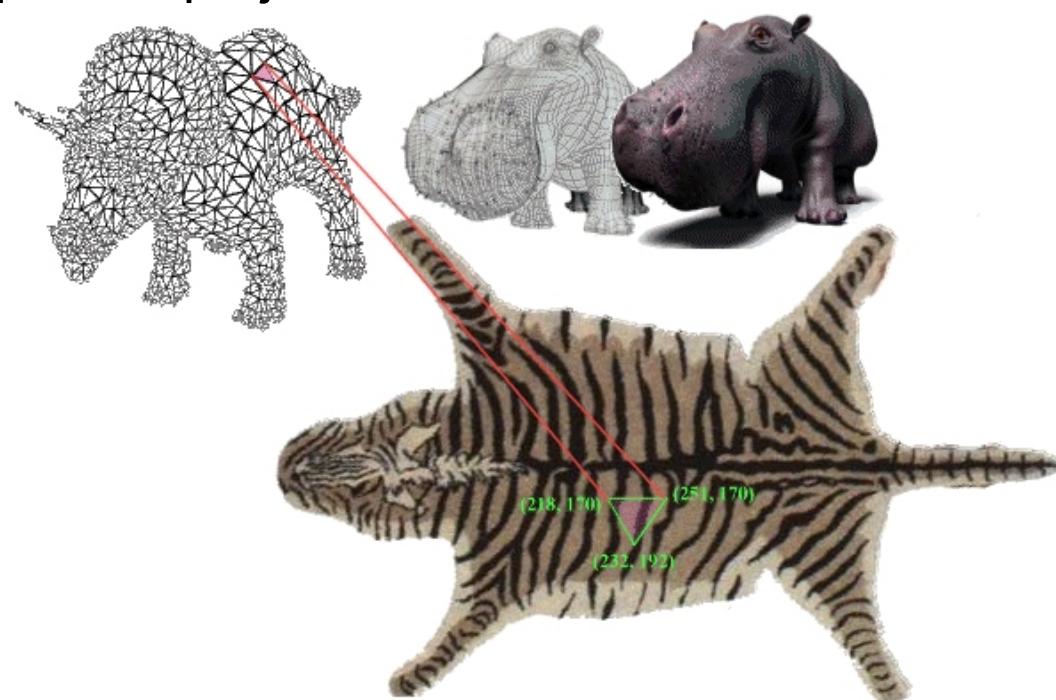
- Texture mapping can be used to alter parts of the illumination equation



Graphics Lecture 8: Slide 38

2D Texture Mapping

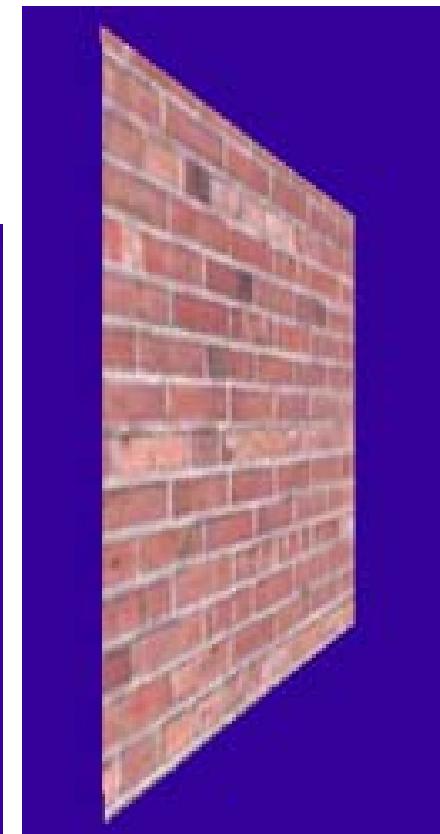
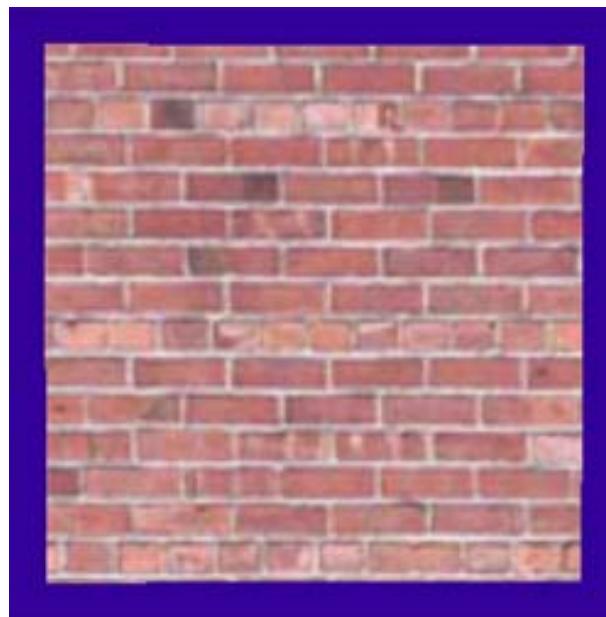
- Increases the apparent complexity of simple geometry
- Requires perspective projection correction
- Can specify variations in shading within a primitive:
 - Illumination
 - Surface Reflectance



Graphics Lecture 8: Slide 39

What's Missing?

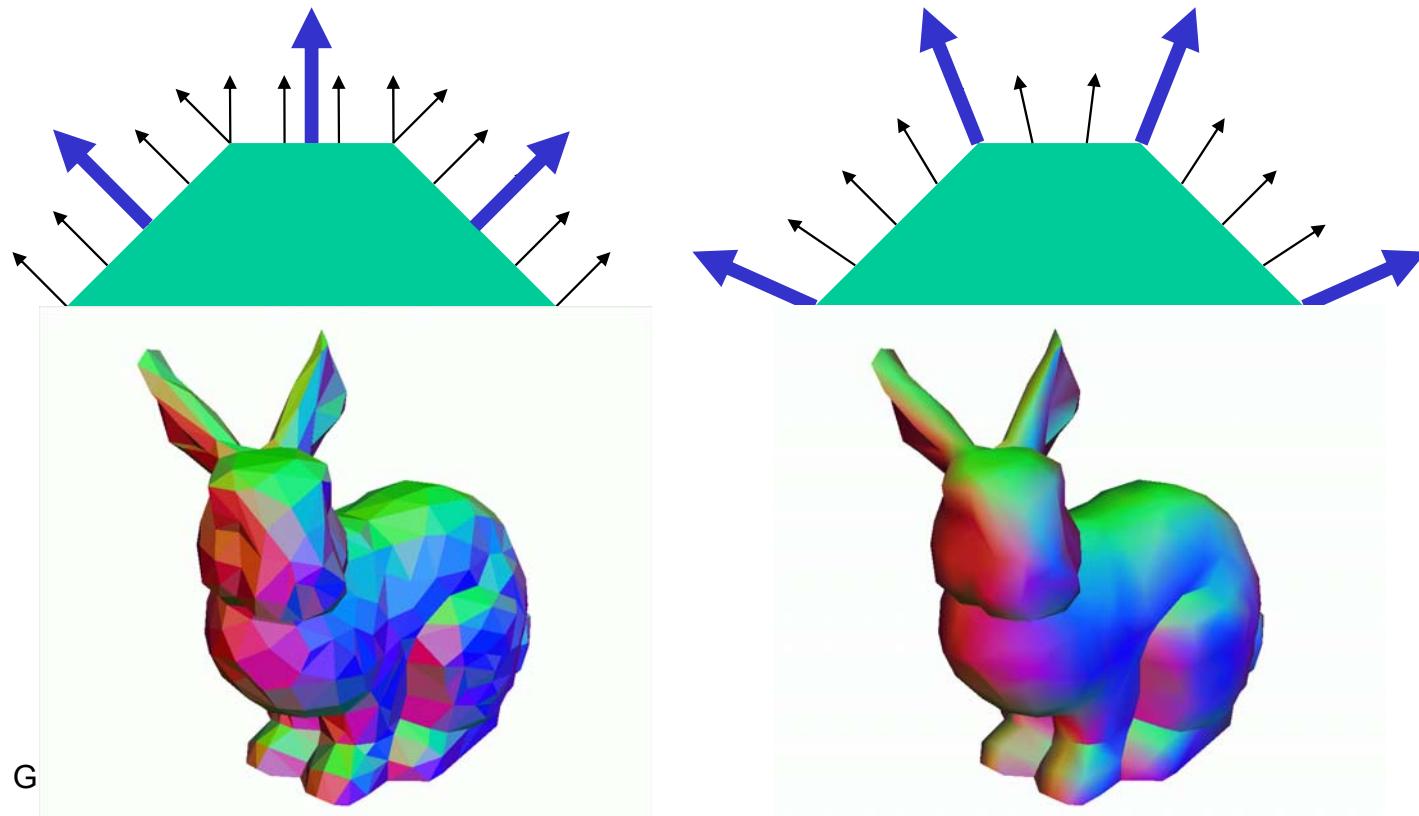
- What's the difference between a real brick wall and a photograph of the wall texture-mapped onto a plane?
- What happens if we change the lighting or the camera position?



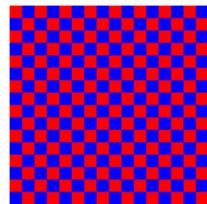
Graphics Lecture 8: Slide 40

Remember Normal Averaging for Shading?

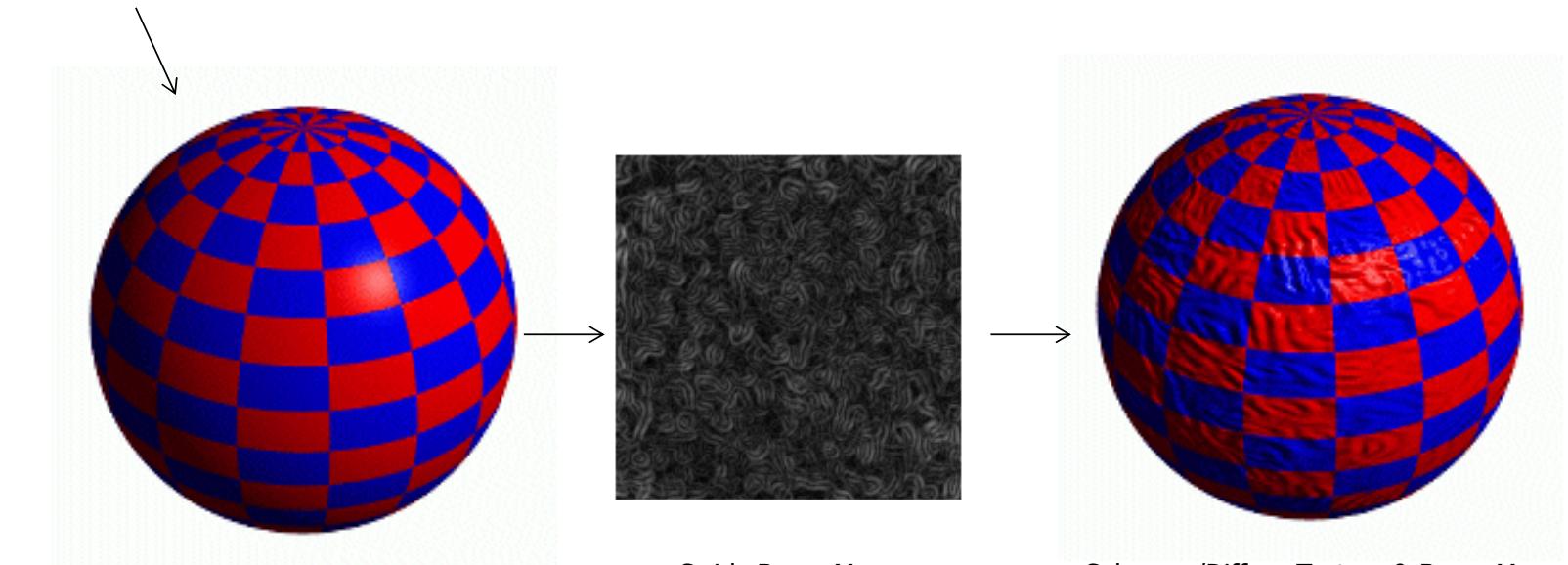
- Instead of using the normal of the triangle, interpolate an averaged normal at each vertex across the face



Bump Mapping



- Textures can be used to alter the surface normal of an object.
- Does not change actual shape of the surface – we only shade it as if it were a different shape!



Sphere w/Diffuse Texture

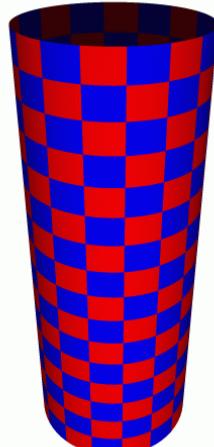
Swirly Bump Map

Sphere w/Diffuse Texture & Bump Map

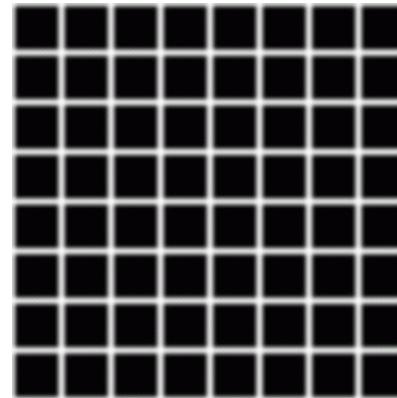
Graphics Lecture 8: Slide 42

Bump Mapping

- The texture map is treated as a single-valued height function.
- The partial derivatives of the texture tell us how to alter the true surface normal at each point to make the object appear as if it were deformed by the height function.



Cylinder w/Diffuse Texture Map



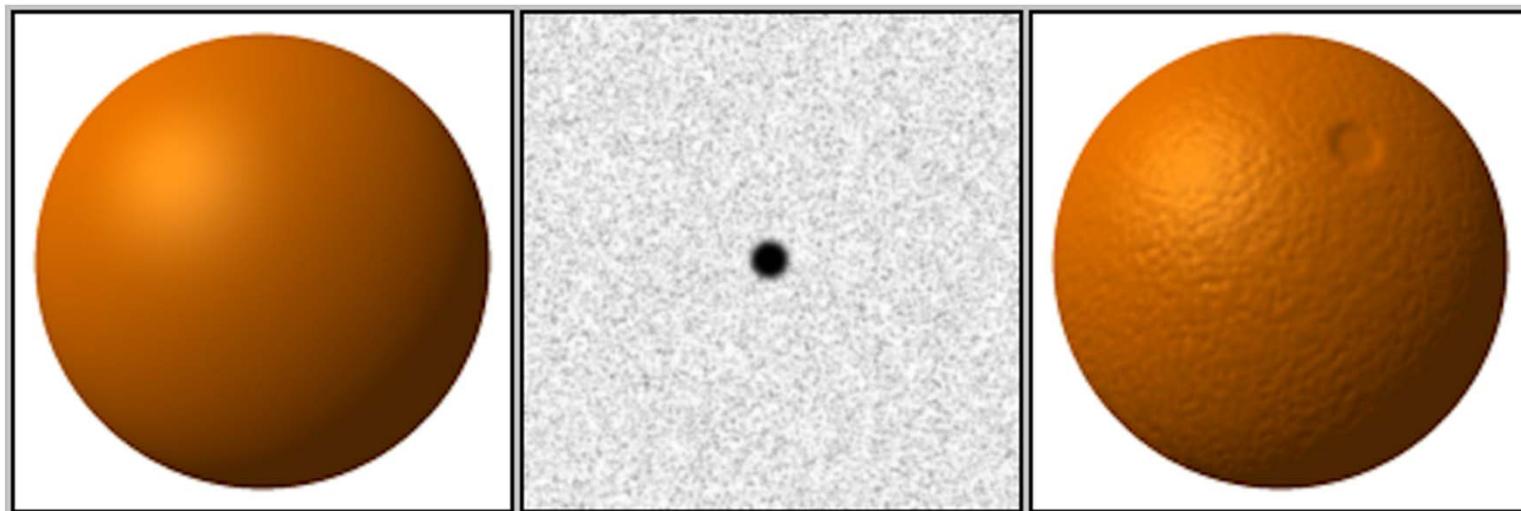
Bump Map



Cylinder w/Texture Map & Bump Map

Graphics Lecture 8: Slide 43

Another Bump Map Example

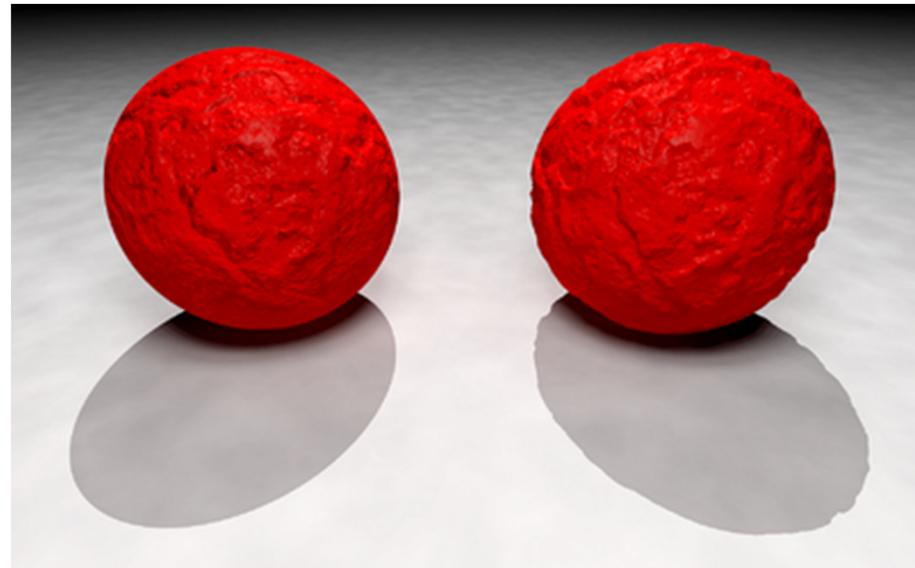


Graphics Lecture 8: Slide 44

Image source: Wikipedia, 2016

What's Missing?

- What does a texture- & bump-mapped object look like as you move the viewpoint?
- What does the silhouette of a bump-mapped object look like?



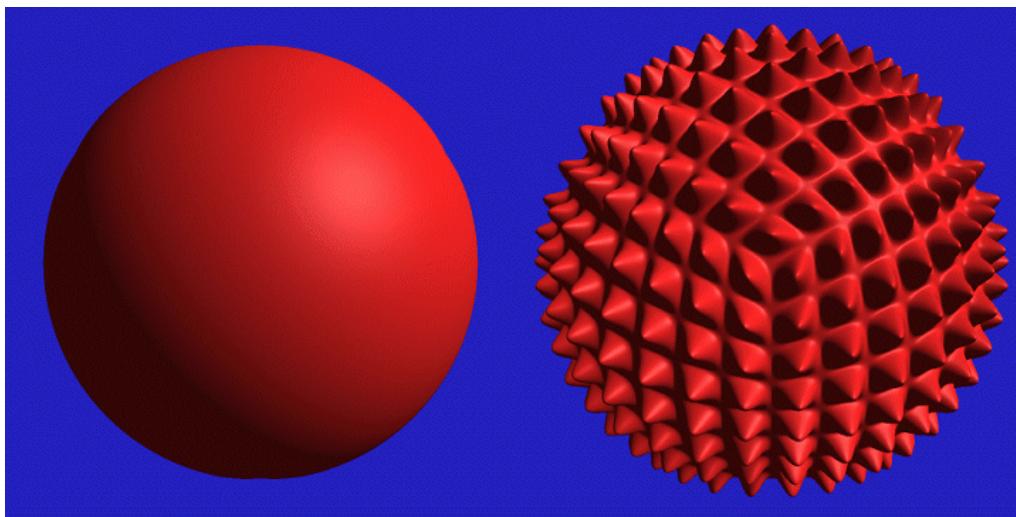
https://threejs.org/examples/webgl_materials_bumpmap.html

Graphics Lecture 8: Slide 45

Image source: Wikipedia, 2016

Displacement Mapping

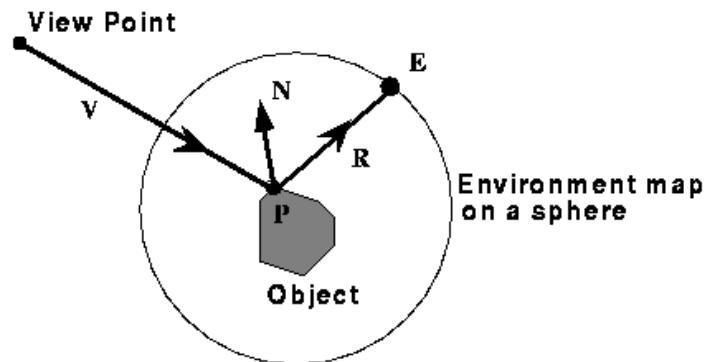
- Use the texture map to actually move the surface point.
 - How is this different than bump mapping?
- The geometry must be displaced before visibility is determined.



Graphics Lecture 8: Slide 46

Environment Maps

- We can simulate reflections by using the direction of the reflected ray to index a spherical texture map at "infinity".
- Assumes that all reflected rays begin from the same point.



Graphics Lecture 8: Slide 47

Environment Mapping Example



https://threejs.org/examples/webgl_materials_cubemap.html

Graphics Lecture 8: Slide 48

Environment Mapping Example



Graphics Lecture 8: Slide 49

Interactive examples

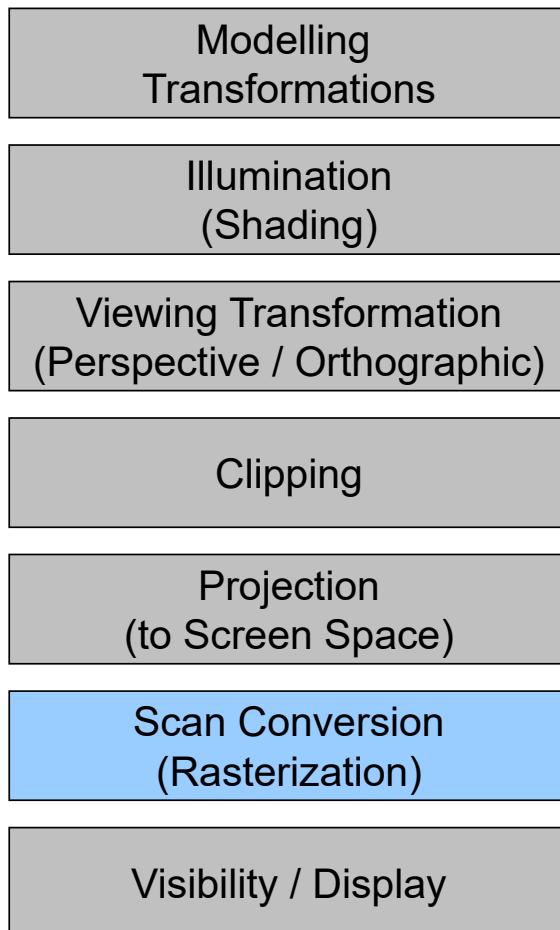
- https://threejs.org/examples/#webgl_materials_bumpmap
- https://threejs.org/examples/#webgl_materials_displacementmap
- https://threejs.org/examples/webgl_materials_cubemap_dynamic2.html
- <https://www.youtube.com/watch?v=K5n3p97-tuQ>

Interactive Computer Graphics: Lecture 9

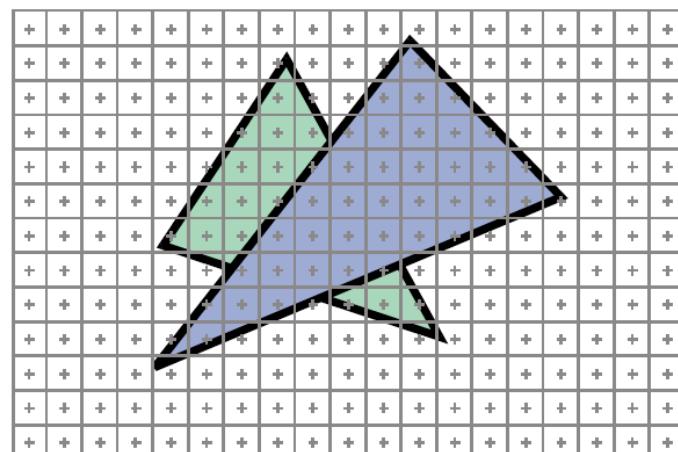
Rasterization, Visibility & Anti-aliasing

Some slides adopted from
F. Durand and B. Cutler, MIT

The Graphics Pipeline

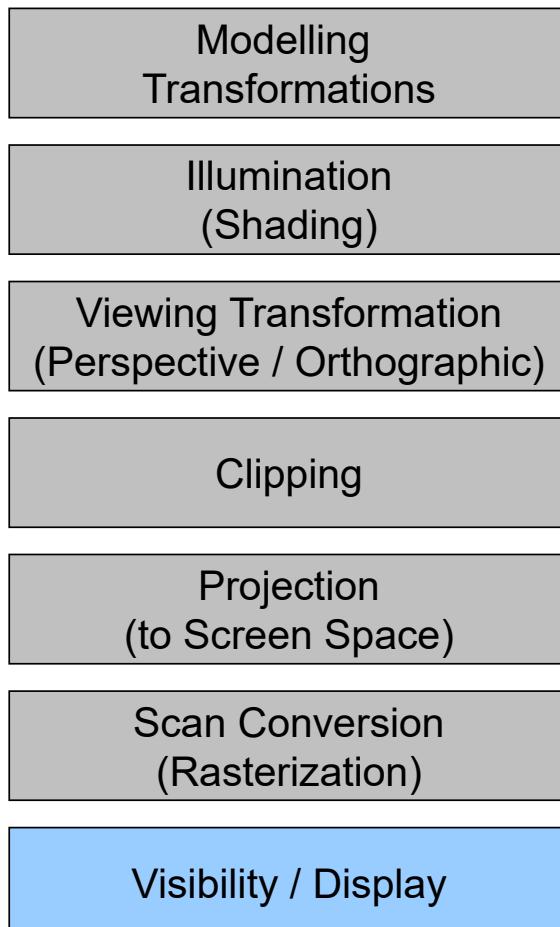


- Rasterizes objects into pixels
- Interpolate values inside objects (color, depth, etc.)

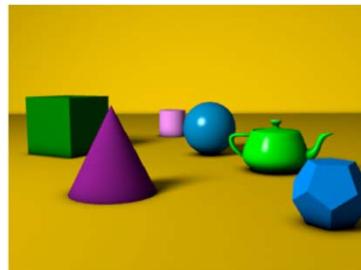


Graphics Lecture 9: Slide 2

The Graphics Pipeline



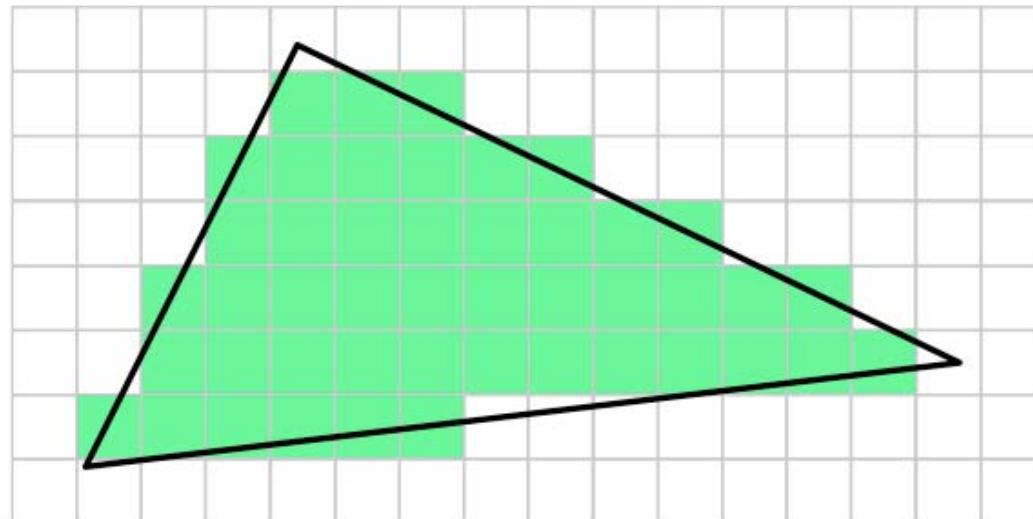
- Handles occlusions
- Determines which objects are closest and therefore visible



Graphics Lecture 9: Slide 3

Rasterization

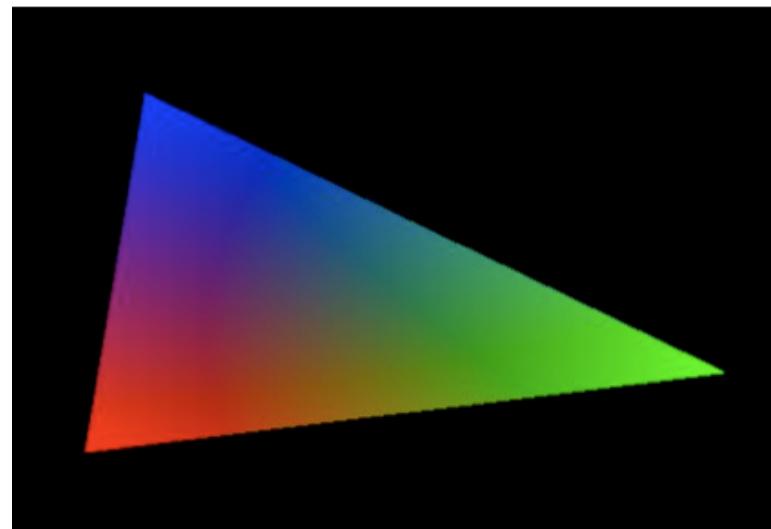
- Determine which pixels are drawn into the framebuffer
- Interpolate parameters (colors, texture coordinates, etc.)



Graphics Lecture 9: Slide 4

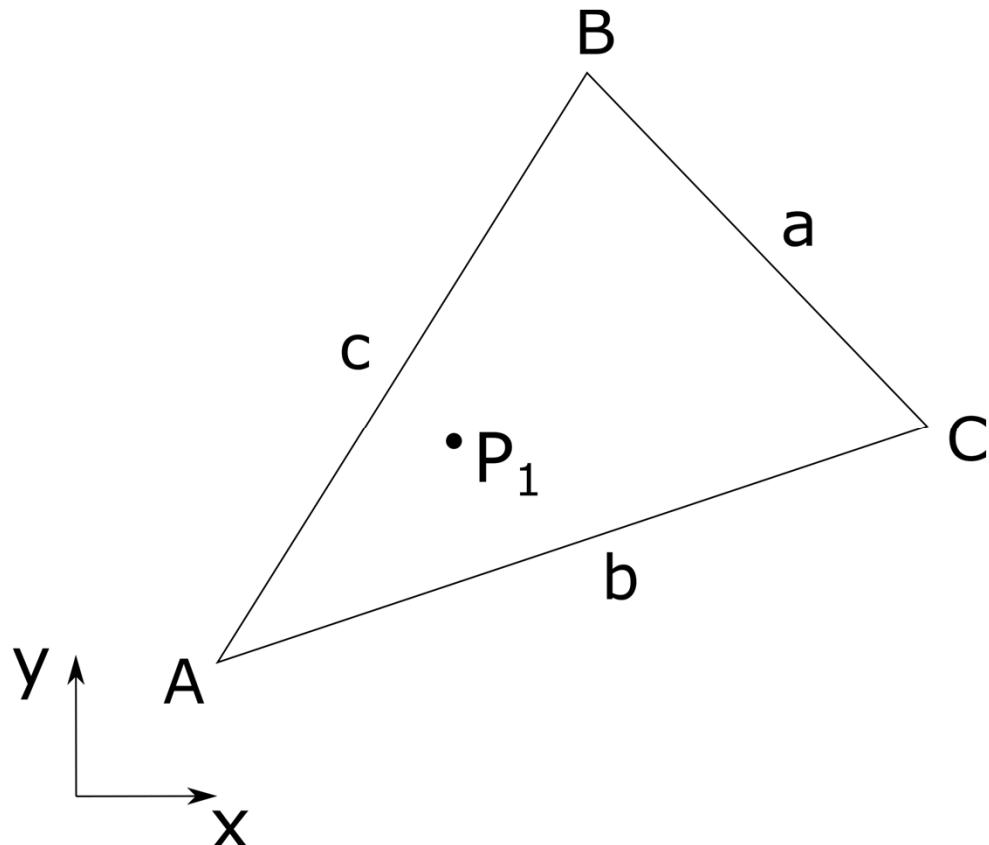
Rasterization

- What does interpolation mean?
- Examples: Colors, normals, shading, texture coordinates



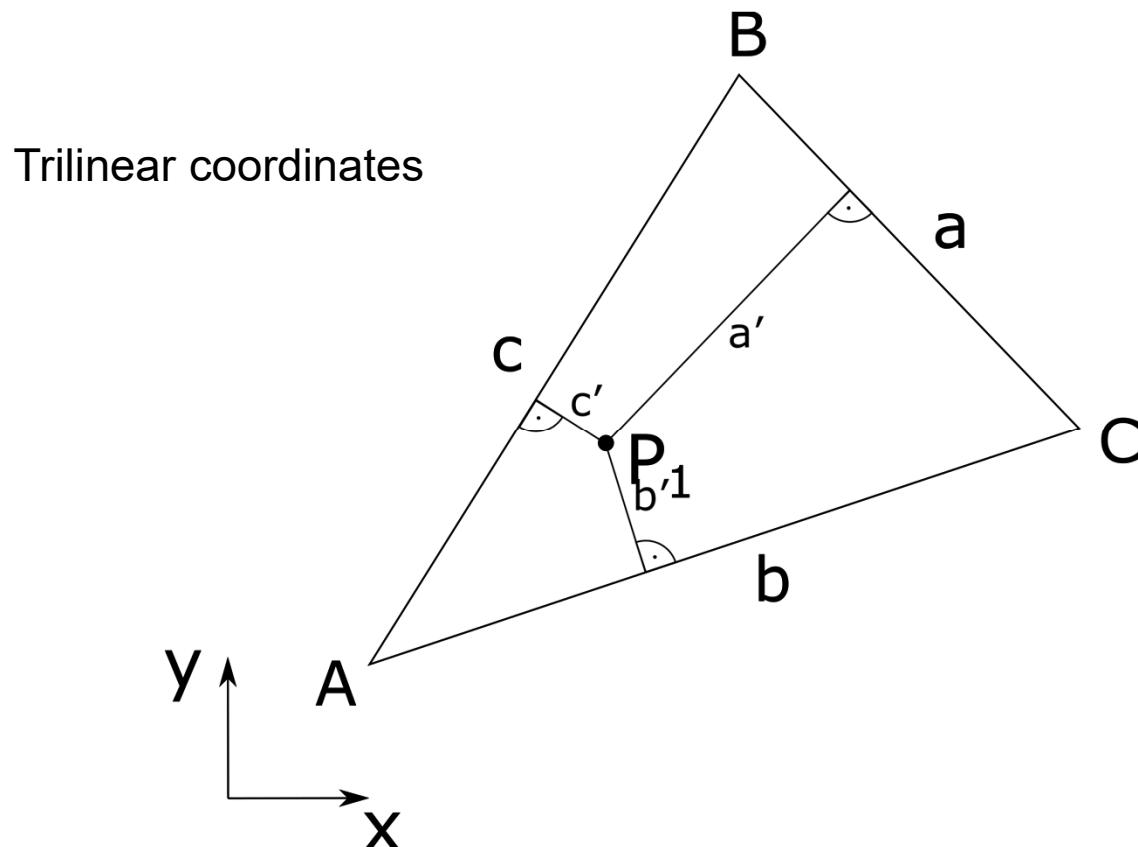
Graphics Lecture 9: Slide 5

Coordinate intuition



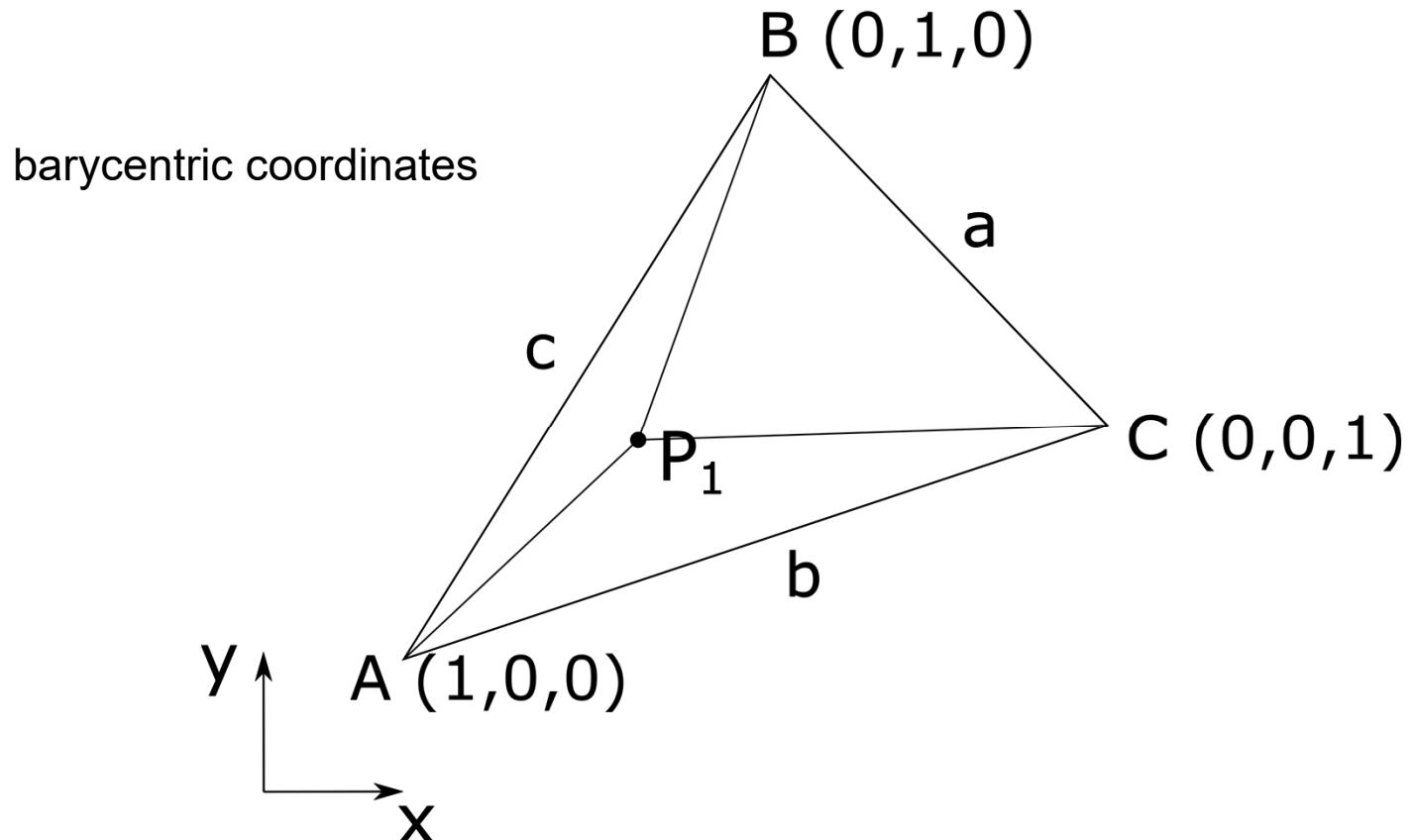
Graphics Lecture 9: Slide 6

Coordinate intuition



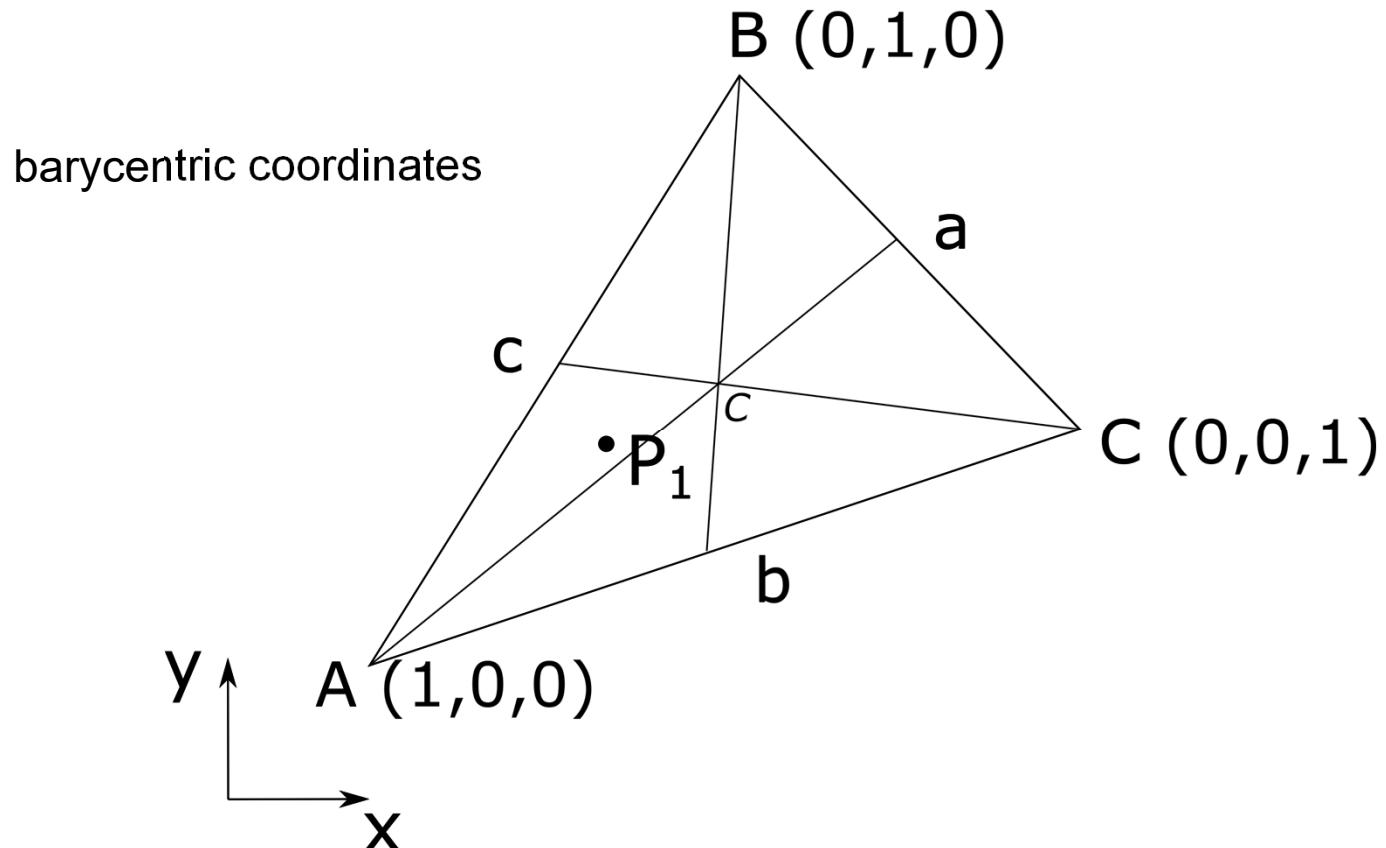
Graphics Lecture 9: Slide 7

Coordinate intuition



Graphics Lecture 9: Slide 8

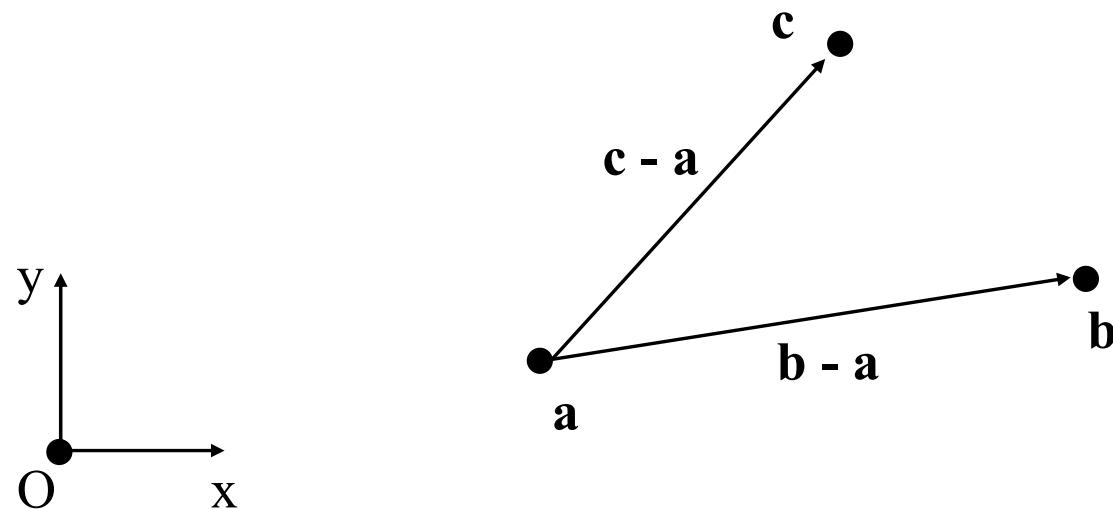
Coordinate intuition



Graphics Lecture 9: Slide 9

A triangle in terms of vectors

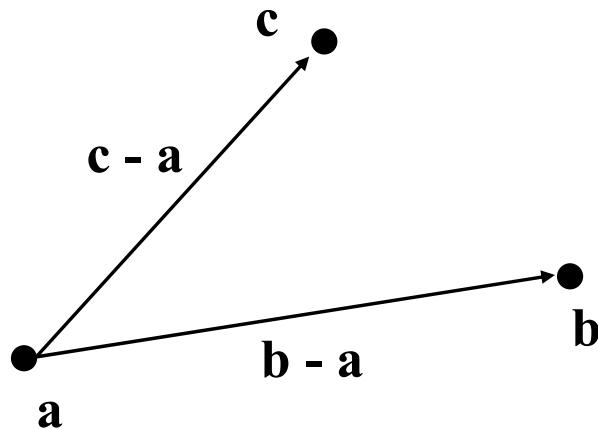
- We can use vertices \mathbf{a} , \mathbf{b} and \mathbf{c} to specify the three points of a triangle
- We can also compute the edge vectors



Graphics Lecture 9: Slide 10

Points and planes

- The three non-collinear points determine a plane

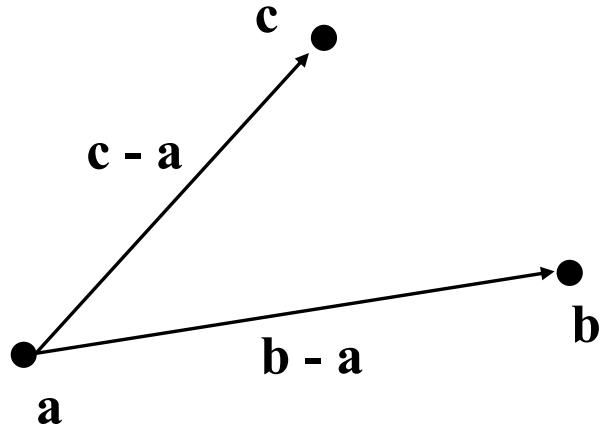


- Example: The vertices \mathbf{a} , \mathbf{b} and \mathbf{c} determine a plane
- The vectors $\mathbf{b} - \mathbf{a}$ and $\mathbf{c} - \mathbf{a}$ form a basis for this plane

Basis vectors

- This (non-orthogonal) basis can be used to specify the location of any point \mathbf{p} in the plane

$$\mathbf{p} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$$



Graphics Lecture 9: Slide 12

Barycentric coordinates

- We can reorder the terms of the equation:

$$\begin{aligned}\mathbf{p} &= \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}) \\ &= (1 - \beta - \gamma)\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c} \\ &= \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}\end{aligned}$$

- In other words:

$$\mathbf{p}(\alpha, \beta, \gamma) = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$$

- α, β, γ and called barycentric coordinates

Barycentric coordinates

- **Homogenous barycentric coordinates:**
 - normalised so that $\alpha + \beta + \gamma = \text{area of triangle}$
- **Areal coordinates or absolute barycentric coordinates** : barycentric coordinates *normalized by the area of the original triangle* $\alpha + \beta + \gamma = 1$

Barycentric coordinates

- Barycentric coordinates describe a point \mathbf{p} as an affine combination of the triangle vertices

$$\mathbf{p}(\alpha, \beta, \gamma) = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c} \quad \alpha + \beta + \gamma = 1$$

- For any point \mathbf{p} inside the triangle ($\mathbf{a}, \mathbf{b}, \mathbf{c}$):

$$0 < \alpha < 1$$

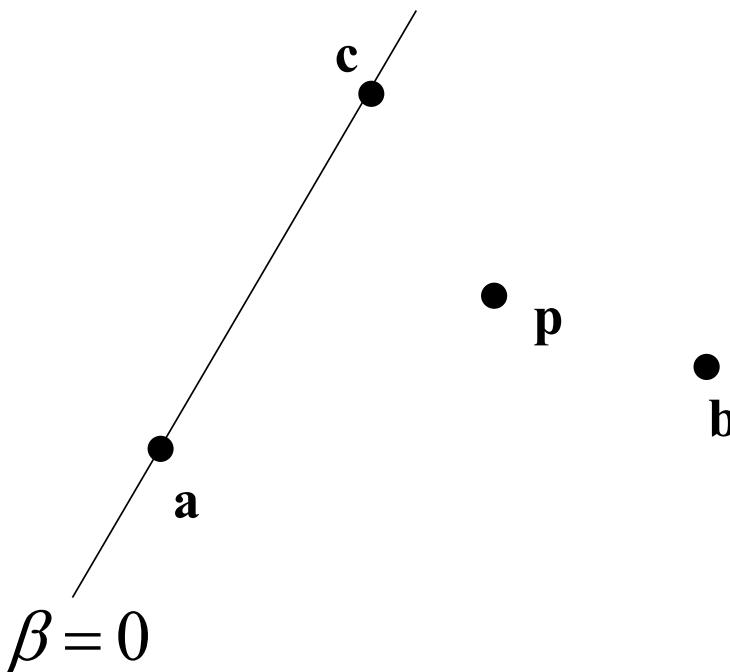
$$0 < \beta < 1$$

$$0 < \gamma < 1$$

- Point on an edge: one coefficient is 0
- Vertex: two coefficients are 0, remaining one is 1

Barycentric coordinates and signed distances

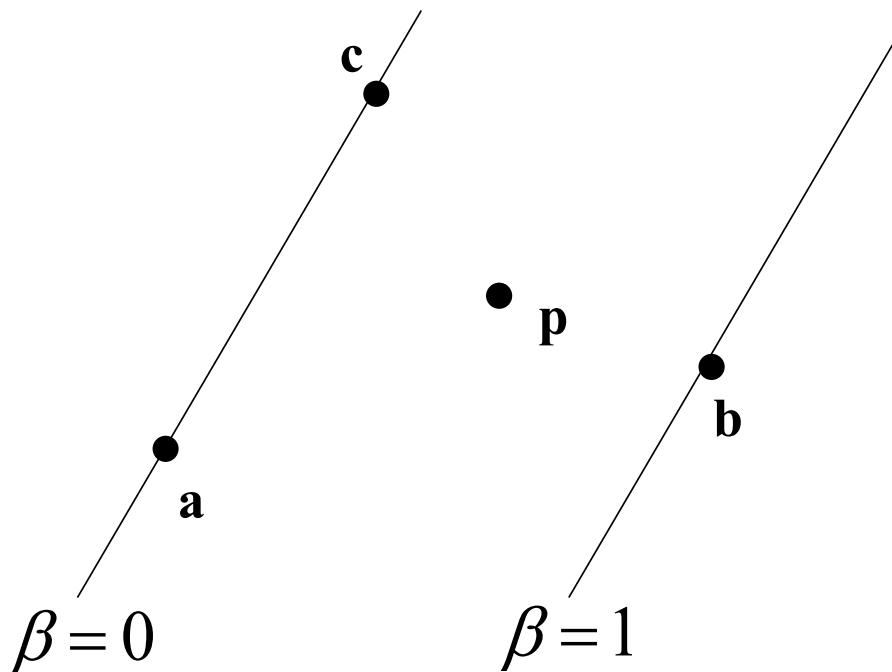
- Let $\mathbf{p} = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$. Each coordinate (e.g. β) is the signed distance from \mathbf{p} to the line through a triangle edge (e.g. \mathbf{ac})



Graphics Lecture 9: Slide 16

Barycentric coordinates and signed distances

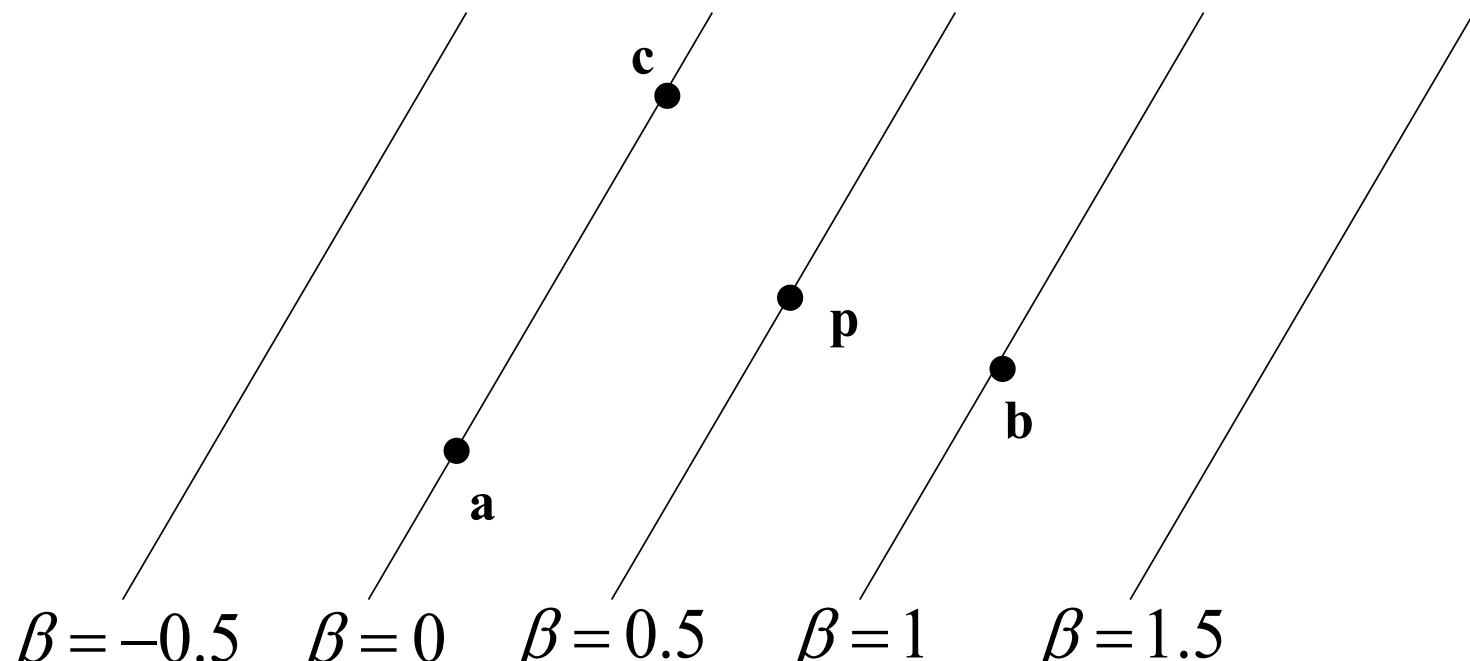
- Let $\mathbf{p} = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$. Each coordinate (e.g. β) is the signed distance from \mathbf{p} to the line through a triangle edge (e.g. \mathbf{ac})



Graphics Lecture 9: Slide 17

Barycentric coordinates and signed distances

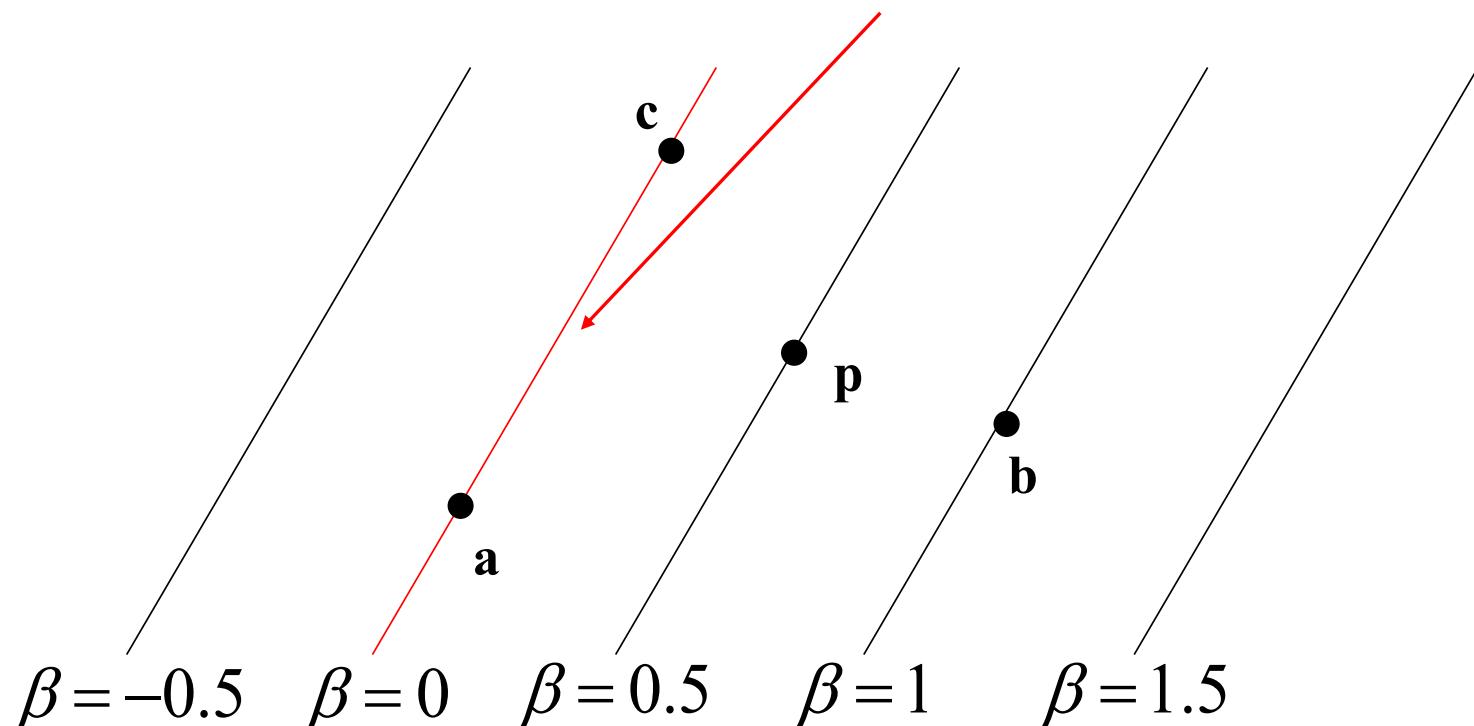
- Let $\mathbf{p} = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$. Each coordinate (e.g. β) is the signed distance from \mathbf{p} to the line through a triangle edge (e.g. \mathbf{ac})



Graphics Lecture 9: Slide 18

Barycentric coordinates and signed distances

- The signed distance can be computed by evaluating implicit line equations, e.g., $f_{ac}(x,y)$ of edge ac



Graphics Lecture 9: Slide 19

Recall: Implicit equation for lines

- Implicit equation in 2D:

$$f(x, y) = 0$$

- Points with $f(x, y) = 0$ are on the line
- Points with $f(x, y) \neq 0$ are not on the line

- General implicit form

$$Ax + By + C = 0$$

- Implicit line through two points (x_a, y_a) and (x_b, y_b)

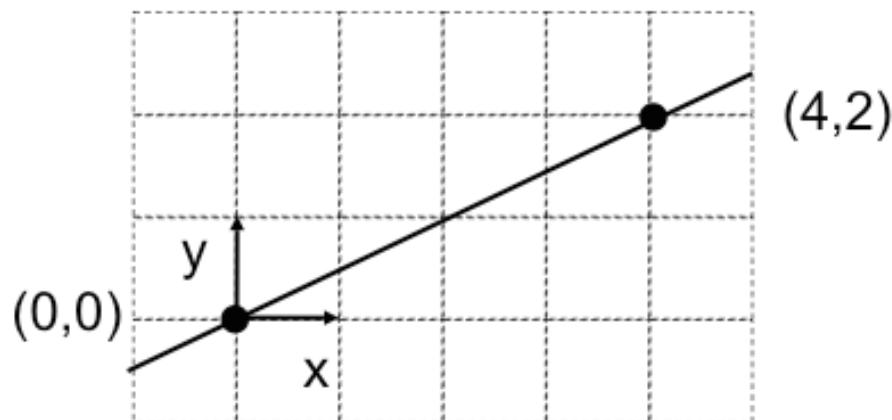
$$(y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a = 0$$

Implicit equation for lines: Example

A =

B =

C =

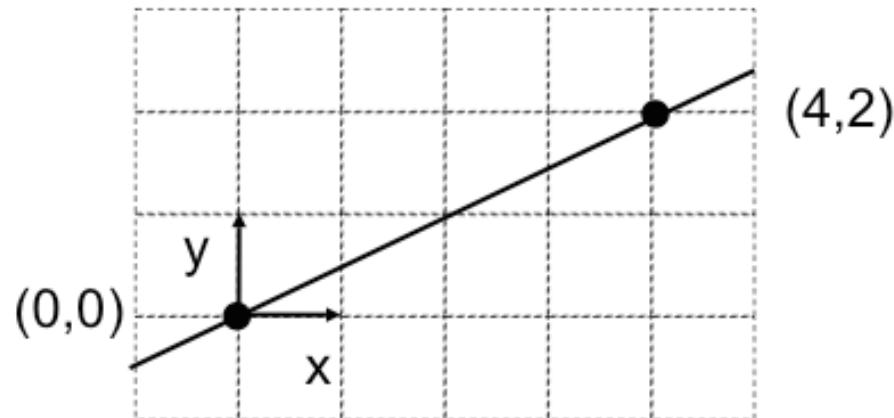


Implicit equation for lines: Example

Solution 1: $-2x + 4y = 0$

Solution 2: $2x - 4y = 0$

$$kf(x, y) = 0 \text{ for any } k$$



Graphics Lecture 9: Slide 22

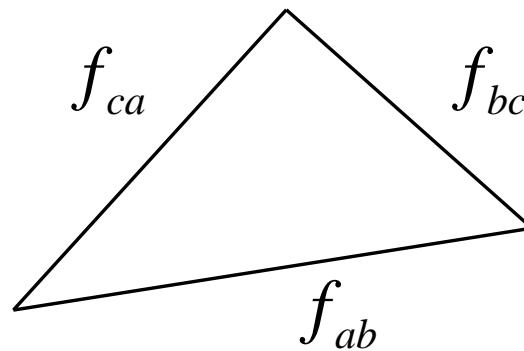
Edge equations

- Given a triangle with vertices (x_a, y_a) , (x_b, y_b) , and (x_c, y_c) .
- The line equations of the edges of the triangle are:

$$f_{ab}(x, y) = (y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a$$

$$f_{bc}(x, y) = (y_b - y_c)x + (x_c - x_b)y + x_b y_c - x_c y_b$$

$$f_{ca}(x, y) = (y_c - y_a)x + (x_a - x_c)y + x_c y_a - x_a y_c$$



Graphics Lecture 9: Slide 23

Barycentric Coordinates

- Remember that: $f(x, y) = 0 \Leftrightarrow kf(x, y) = 0$
- A barycentric coordinate (e.g. β) is a signed distance from a line (e.g. the line that goes through ac)
- For a given point p , we would like to compute its barycentric coordinate β using an implicit edge equation.
- We need to choose k such that

$$kf_{ac}(x, y) = \beta$$

Barycentric Coordinates

- We would like to choose k such that: $k f_{ac}(x, y) = \beta$
- We know that $\beta = 1$ at point **b**:

$$k f_{ac}(x, y) = 1 \Leftrightarrow k = \frac{1}{f_{ac}(x_b, y_b)}$$

- The barycentric coordinate β for point **p** is:

$$\beta = \frac{f_{ac}(x, y)}{f_{ac}(x_b, y_b)}$$

Barycentric Coordinates

- In general, the barycentric area coordinates for point p are:

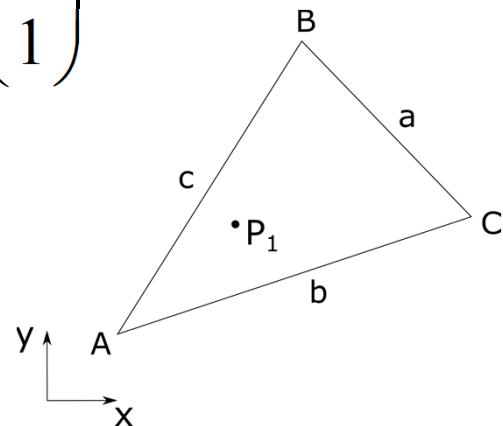
$$\alpha = \frac{f_{bc}(x, y)}{f_{bc}(x_a, y_a)} \quad \beta = \frac{f_{ac}(x, y)}{f_{ac}(x_b, y_b)} \quad \gamma = 1 - \alpha - \beta$$

- Given a point p with Cartesian coordinates (x, y) , we can compute its barycentric coordinates (α, β, γ) as above.

Barycentric Coordinates

- In general, the barycentric area coordinates for point p are the solution of the linear system of equations:

$$\begin{pmatrix} x_a & x_b & x_c \\ y_a & y_b & y_c \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$



Graphics Lecture 9: Slide 27

Barycentric Coordinates

- Can be easily converted to trilinear coordinates

$P_t(t_1, t_2, t_3)$ in trilinear coordinates has barycentric coordinates of $(t_1 \mathbf{a}, t_2 \mathbf{b}, t_3 \mathbf{c})$

where \mathbf{a} , \mathbf{b} , \mathbf{c} , are the side lengths of the triangle.

$P_b(\alpha, \beta, \gamma)$ in barycentric coordinates has trilinear coordinates $(\alpha/a, \beta/b, \gamma/c)$

Triangle Rasterization

- Many different ways to generate fragments for a triangle
- Checking (α, β, γ) is one method, e.g.
$$(0 < \alpha < 1 \ \&\& \ 0 < \beta < 1 \ \&\& \ 0 < \gamma < 1)$$
- In practice, the graphics hardware uses optimized methods:
 - fixed point precision (not floating-point)
 - incremental (use results from previous pixel)

Triangle Rasterization

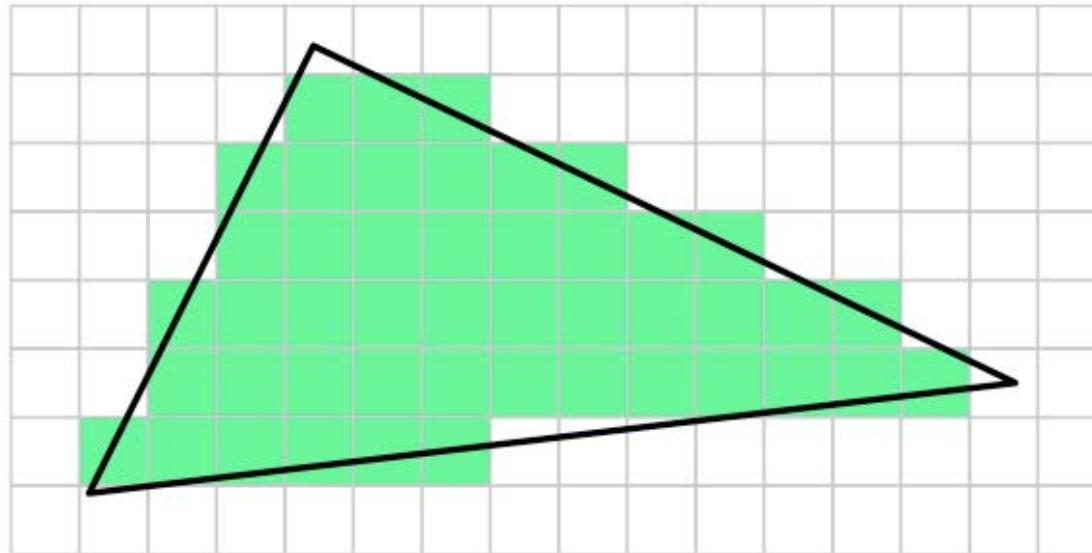
- We can use barycentric coordinates to rasterize and color triangles

```
for all x do
    for all y do
        compute (alpha, beta, gamma) for (x,y)
        if (0 < alpha < 1 and
            0 < beta < 1 and
            0 < gamma < 1 ) then
            c = alpha c0 + beta c1 + gamma c2
            drawpixel(x,y) with color c
```

- The color c varies smoothly within the triangle

Visibility: One triangle

- With one triangle, things are simple
- Pixels never overlap!



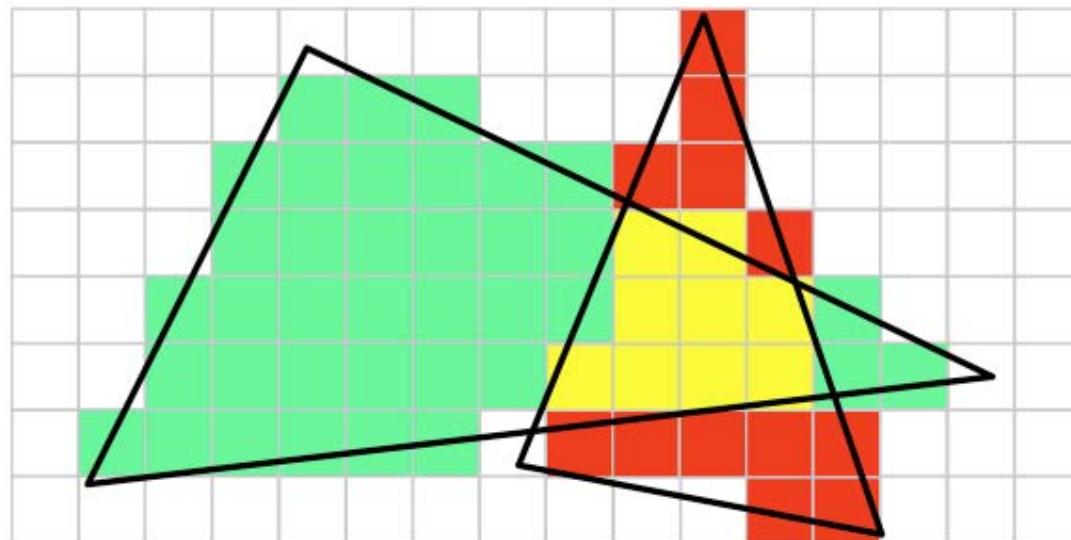
Graphics Lecture 9: Slide 31

Hidden Surface Removal

- Idea: keep track of visible surfaces
- Typically, we see only the front-most surface
- Exception: transparency

Visibility: Two triangles

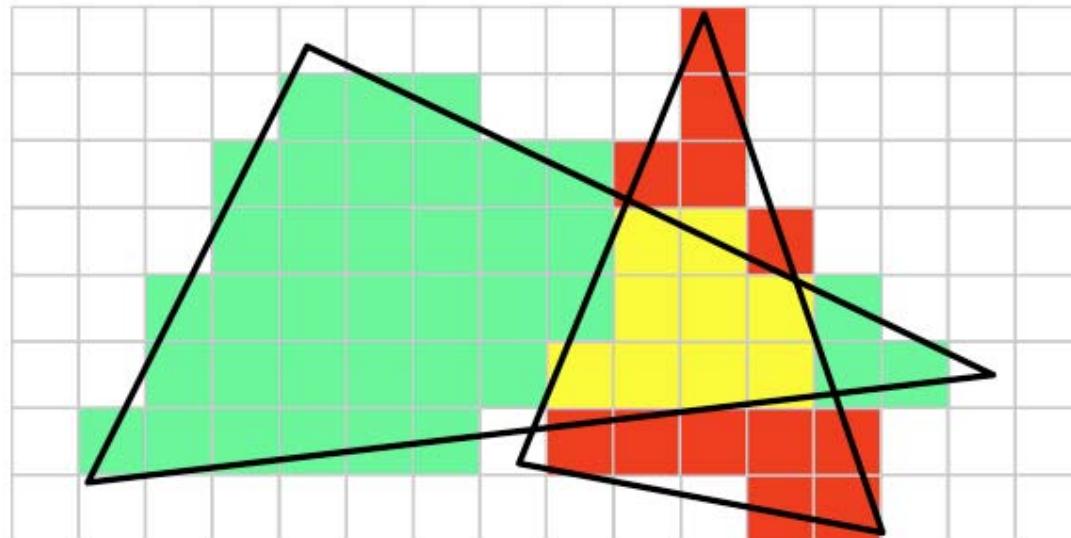
- Things get more complicated with multiple triangles
- Fragments might overlap in screen space!



Graphics Lecture 9: Slide 33

Visibility: Pixels vs Fragments

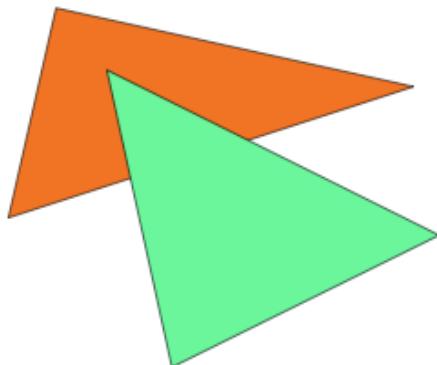
- Each pixel has a unique framebuffer (image) location
- But multiple fragments may end up at same address



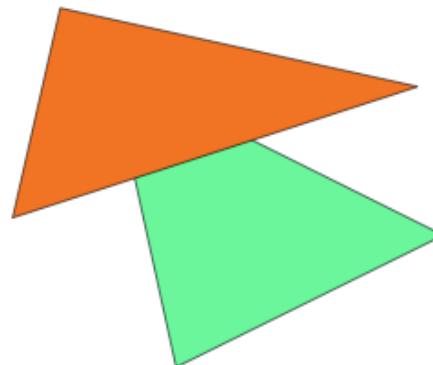
Graphics Lecture 9: Slide 34

Visibility: Which triangle should be drawn first?

- Two possible cases:



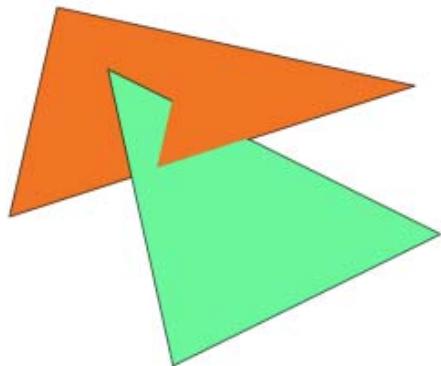
green triangle on top



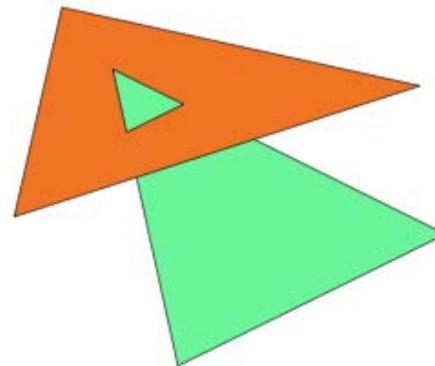
orange triangle on top

Visibility: Which triangle should be drawn first?

- Many other cases possible!



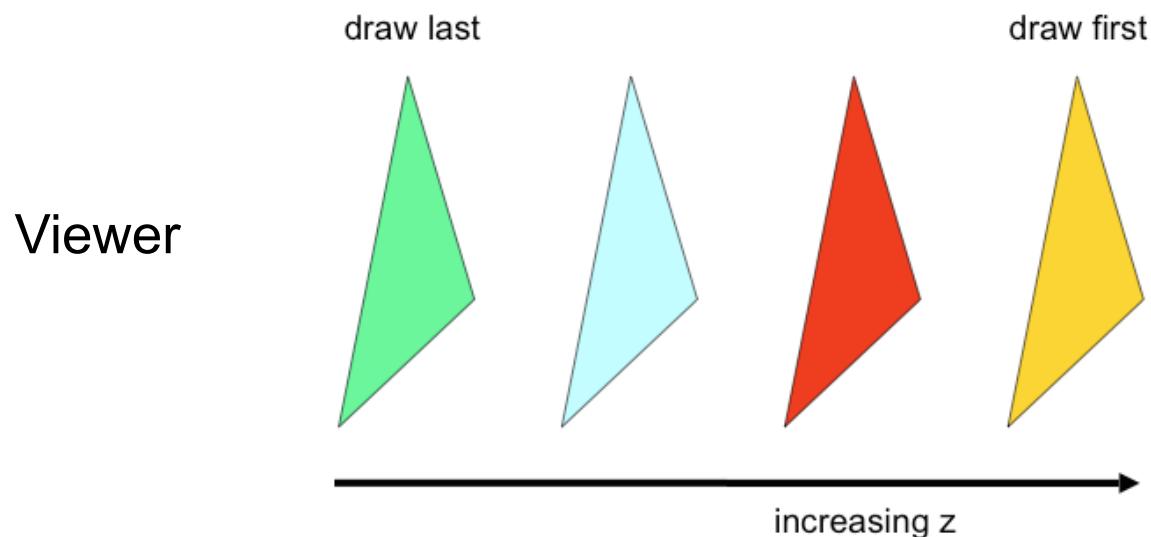
intersection #1



intersection #2

Visibility: Painter's Algorithm

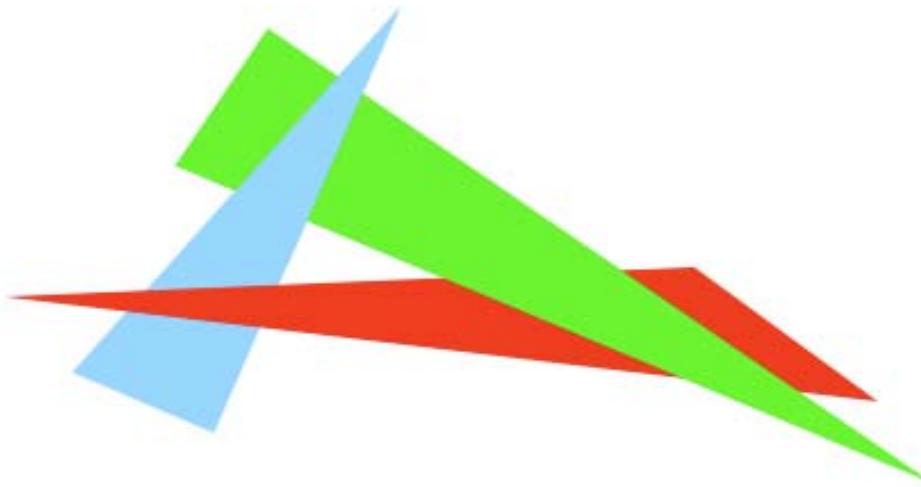
- Sort triangles (using z values in eye space)
- Draw triangles from back to front



Graphics Lecture 9: Slide 37

Visibility: Painter's Algorithm - Problems

- Correctness issues:
 - Intersections
 - Cycles
 - Solve by splitting triangles, but ugly and expensive
- Efficiency (sorting)



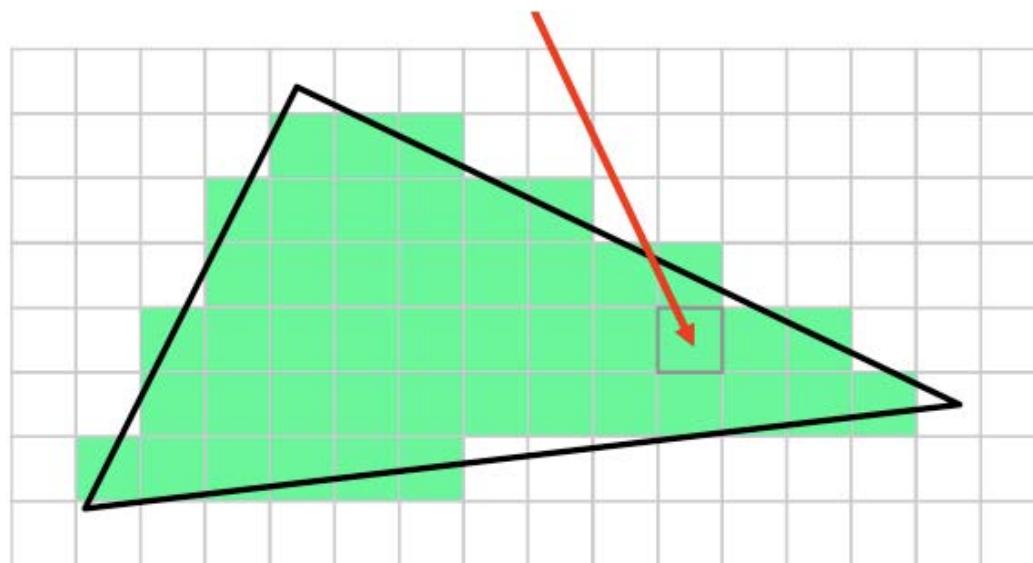
Graphics Lecture 9: Slide 38

The Depth Buffer (Z-Buffer)

- Perform hidden surface removal per-fragment
- Idea:
 - Each fragment gets a z value in screen space
 - Keep only the fragment with the smallest z value

The Depth Buffer (Z-Buffer)

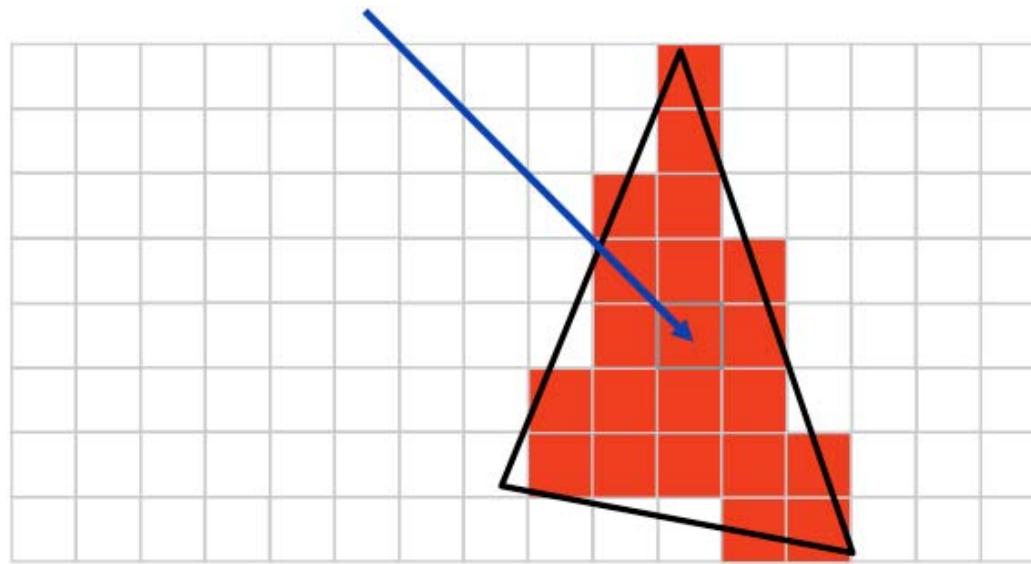
- Example:
 - fragment from green triangle has z value of 0.7



Graphics Lecture 9: Slide 40

The Depth Buffer (Z-Buffer)

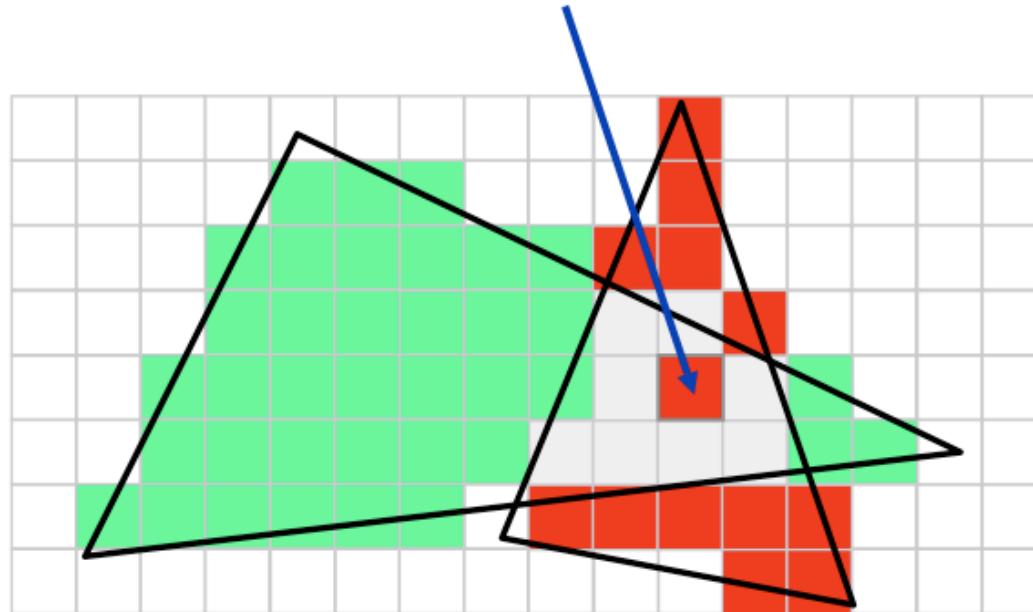
- Example:
 - fragment from red triangle has z value of 0.3



Graphics Lecture 9: Slide 41

The Depth Buffer (Z-Buffer)

- Since $0.3 < 0.7$, the red fragment wins



Graphics Lecture 9: Slide 42

The Depth Buffer (Z-Buffer)

- Many fragments might map to the same pixel location
- How to track their z-values?
- Solution: z-buffer (2D buffer, same size as image)

1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.1	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.1	0.1	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.2	0.2	0.3	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.3	0.3	0.4	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.3	0.4	0.4	0.5	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.4	0.4	0.5	0.5	0.5	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.4	0.5	1.0	1.0	1.0

Graphics Lecture 9: Slide 43

The Z-Buffer Algorithm

```
Let CB be color (frame) buffer, ZB be z-buffer  
Initialize z-buffer contents to 1.0 (far away)  
For each triangle T  
    Rasterize T to generate fragments  
    For each fragment F with screen position (x,y,z) and color value C  
        If (z < ZB[x,y]) then  
            Update color: CB[x,y] = C  
            Update depth: ZB[x,y] = z
```

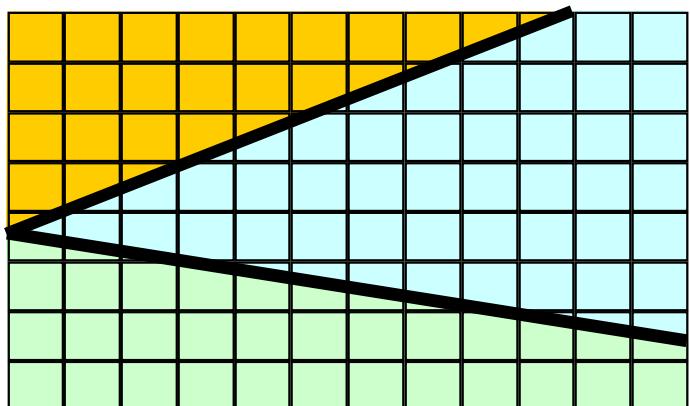
Z-buffer Algorithm Properties

- What makes this method nice?
 - simple (facilitates hardware implementation)
 - handles intersections
 - handles cycles
 - draw opaque polygons in any order

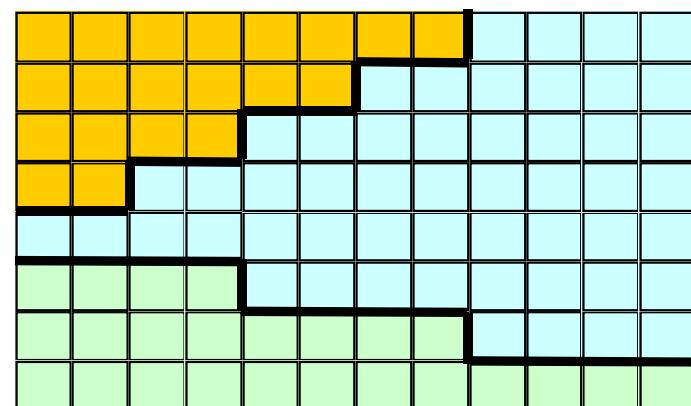
Alias Effects

- One major problem with rasterization is called alias effects, e.g straight lines or triangle boundaries look jagged
- These are caused by undersampling, and can cause unreal visual artefacts.
- It also occurs in texture mapping

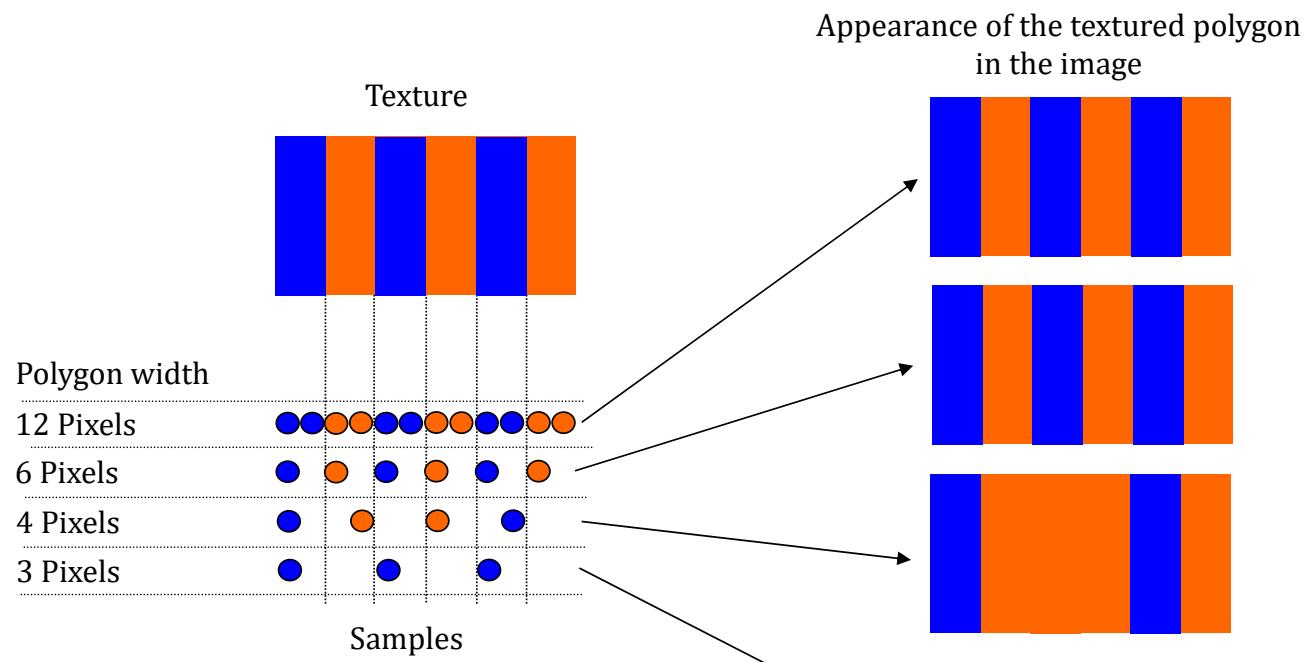
Alias Effects at straight boundaries in raster images.



Desired Boundaries



Pixels Set

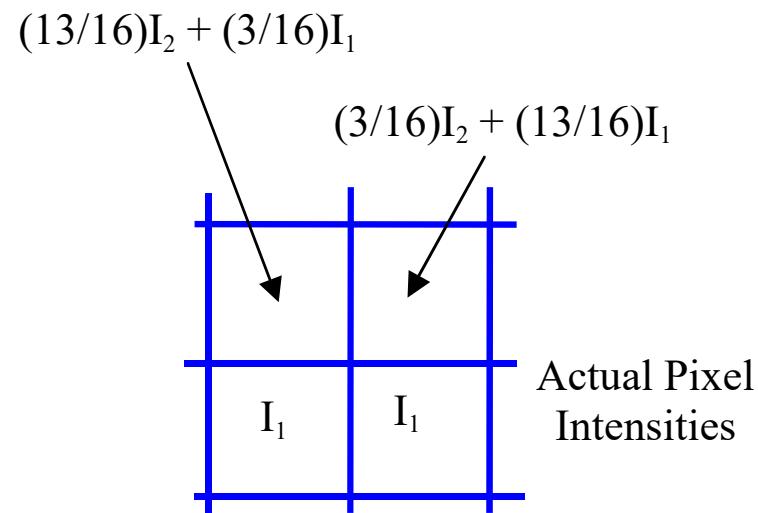
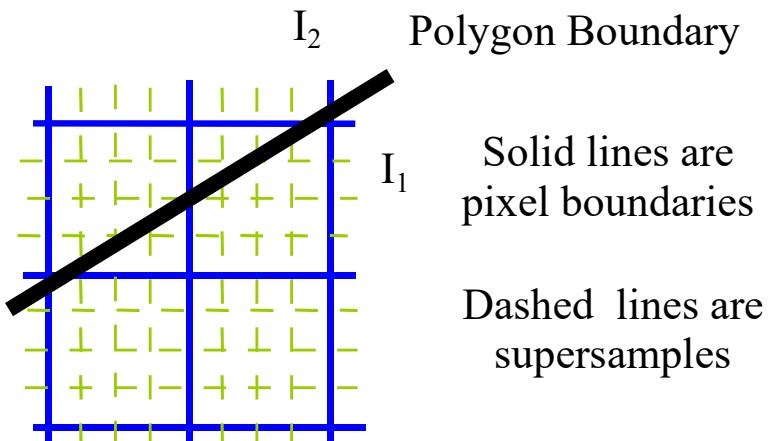


Anti-Aliasing

- The solution to aliasing problems is to apply a degree of blurring to the boundary such that the effect is reduced.
- The most successful technique is called **Supersampling**

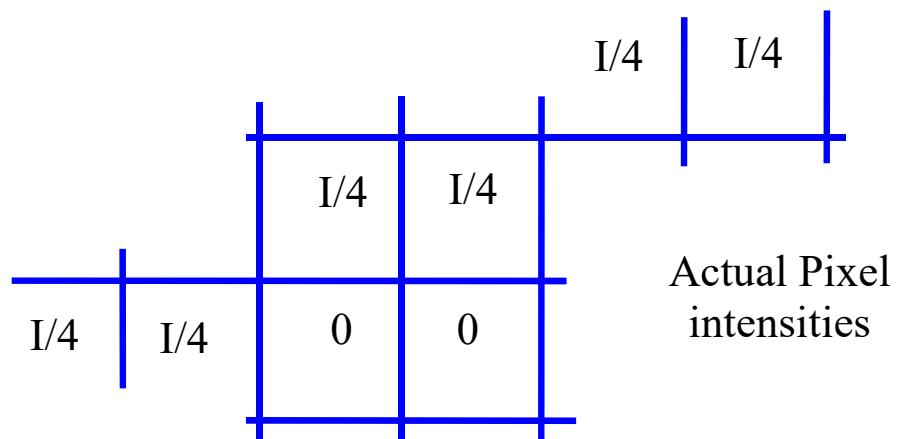
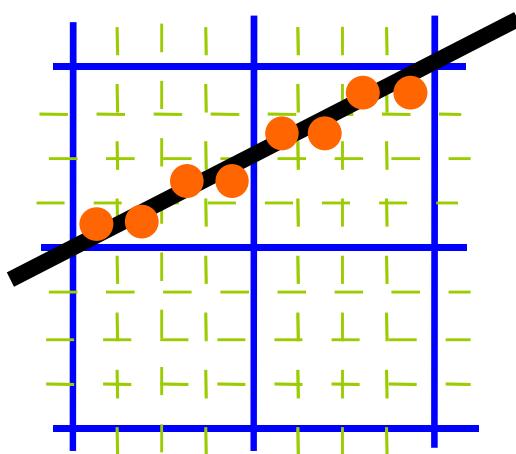
Supersampling

- The basic idea is to compute the picture at a higher resolution to that of the display area.
- Supersamples are averaged to find the pixel value.
- This has the effect of blurring boundaries, but leaving coherent areas of colour unchanged



Limitations of Supersampling

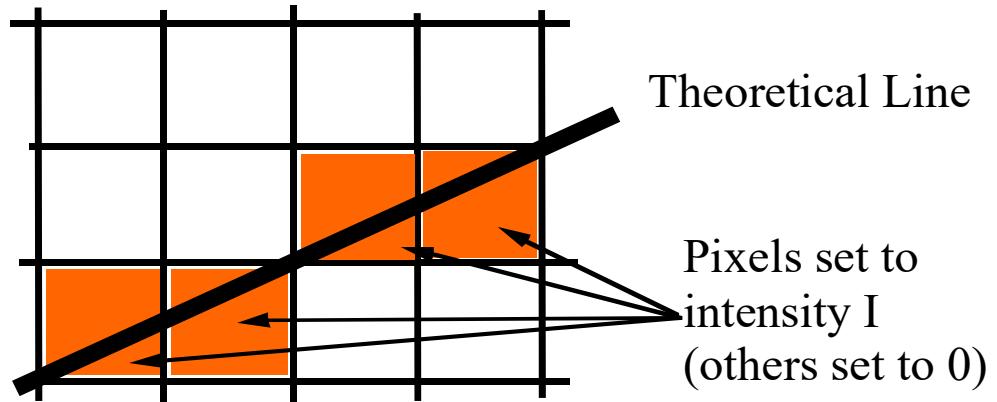
- Supersampling works well for scenes made up of filled polygons.
- However, it does require a lot of extra computation.
- It does not work for line drawings.



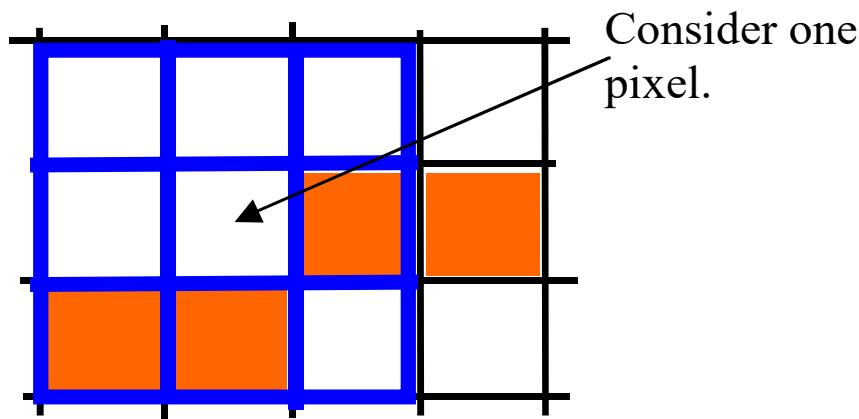
Convolution filtering

- The more common (and much faster) way of dealing with alias effects is to use a ‘filter’ to blur the image.
- This essentially takes an average over a small region around each pixel

For example consider the image of a line



Treat each pixel of the image



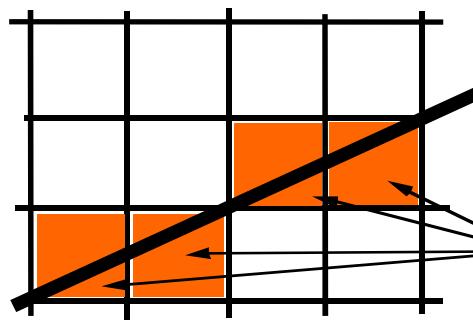
We replace the pixel by a local average,
one possibility would be $3*I/9$

Weighted averages

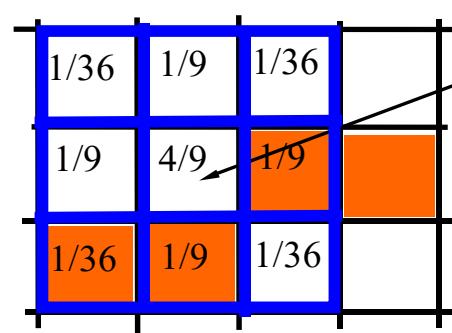
- Taking a straight local average has undesirable effects.
- Thus we normally use a weighted average.

$1/36 *$

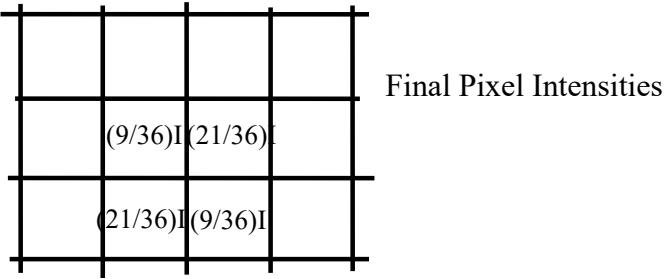
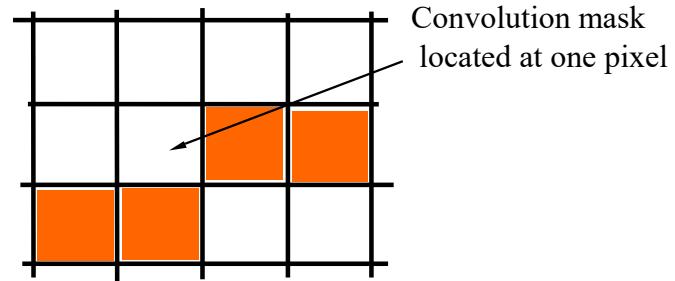
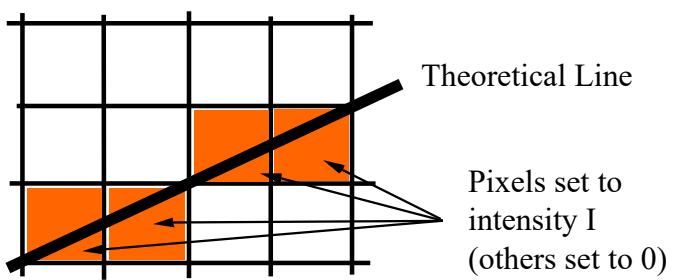
1	4	1
4	16	4
1	4	1



Theoretical Line
Pixels set to
intensity I
(others set to 0)



Convolution
mask located
at one pixel

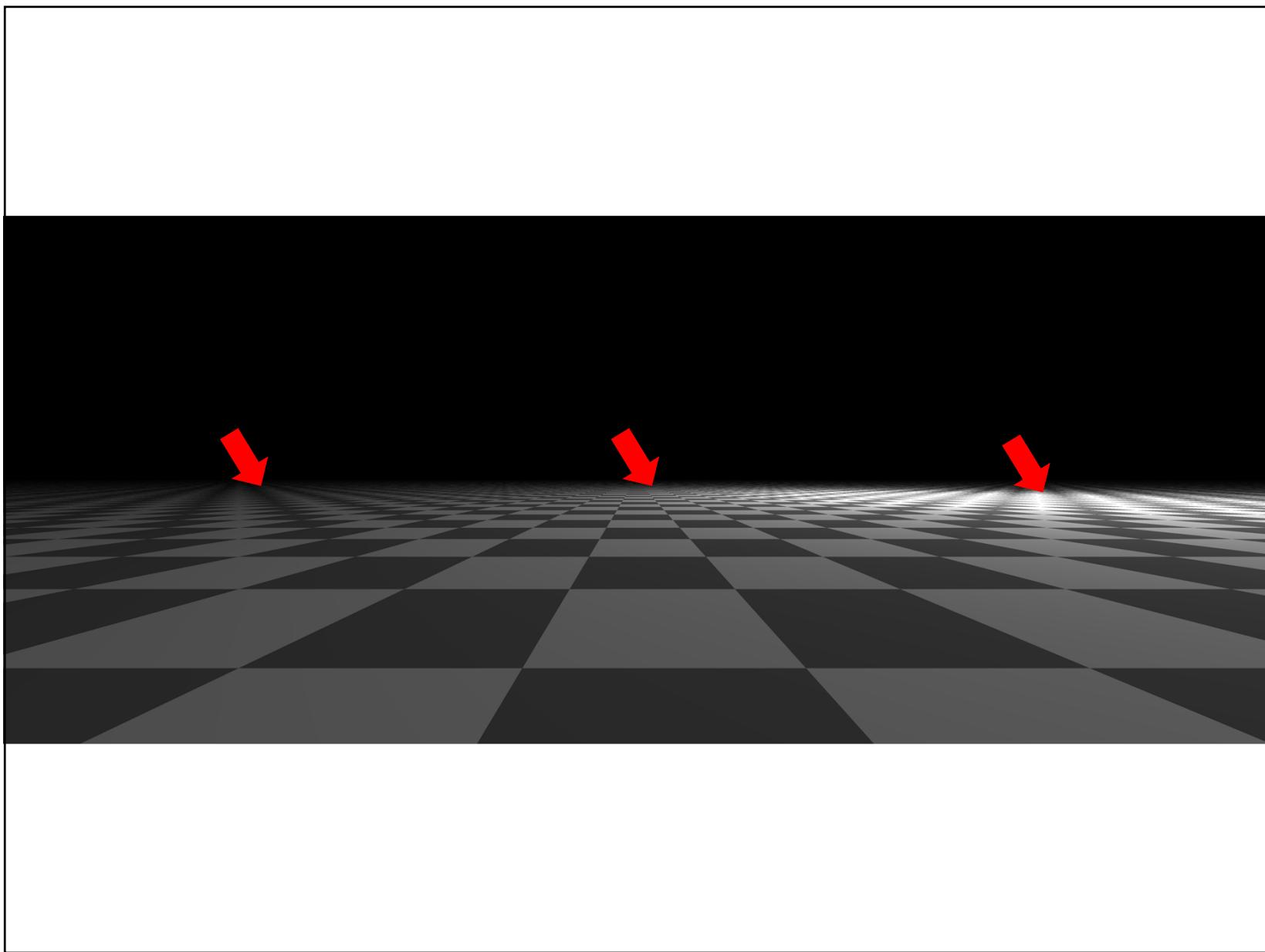


Pros and Cons of Convolution filtering

- Advantages:
 - It is very fast and can be done in hardware
 - Generally applicable
- Disadvantages:
 - It does degrade the image while enhancing its visual appearance.

Anti-Aliasing textures

- Similar
- When we identify a point in the texture map we return an average of texture map around the point.
- Scaling needs to be applied so that the less the samples taken the bigger the local area where averaging is done.



Interactive Computer Graphics: Lecture 10

Ray tracing



2

2

© Steve Anger 1993

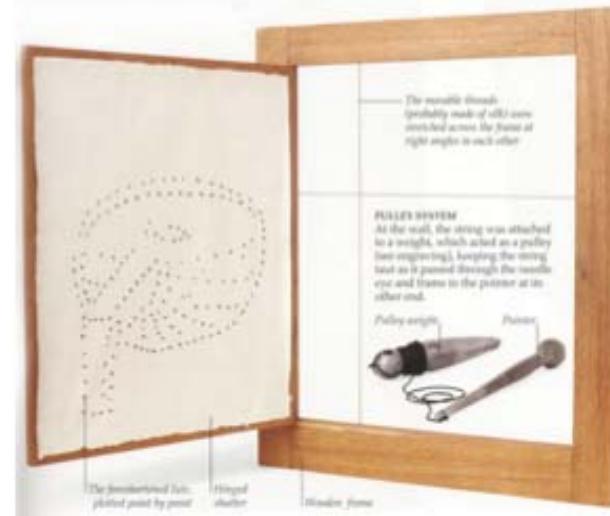
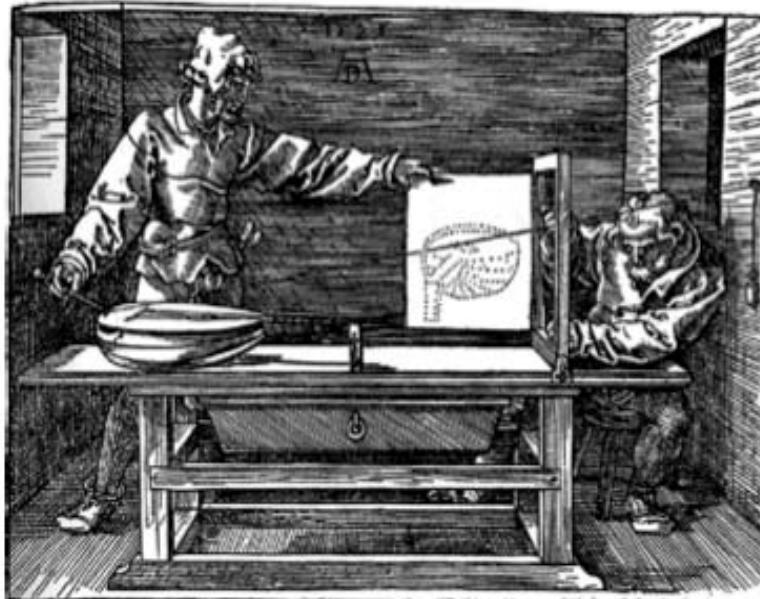


Direct and Global Illumination

- **Direct illumination**: A surface point receives light directly from all light sources in the scene.
 - Computed by the direct illumination model.
- **Global illumination**: A surface point receives light after the light rays interact with other objects in the scene.
 - Points may be in shadow.
 - Rays may refract through transparent material.
 - Computed by reflection and transmission rays.

Albrecht Dürer's Ray Casting Machine

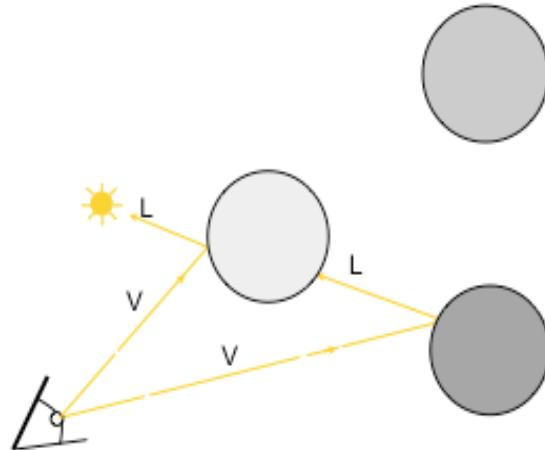
- Albrecht Dürer, 16th century



Graphics

Arthur Appel, 1968

- On calculating the illusion of reality, 1968
- Cast one ray per pixel (ray casting).
 - For each intersection, trace one ray to the light to check for shadows
 - Only a local illumination model
- Developed for pen-plotters



Ray casting

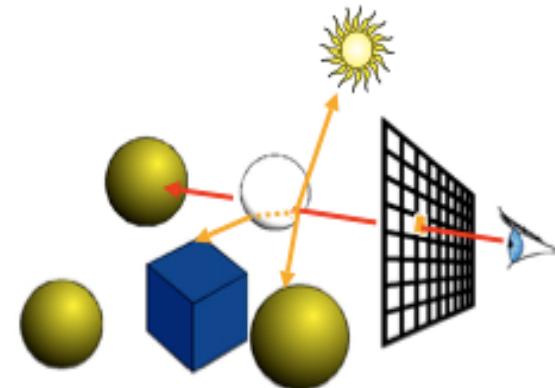
cast ray

 Intersect all objects

 color = ambient term

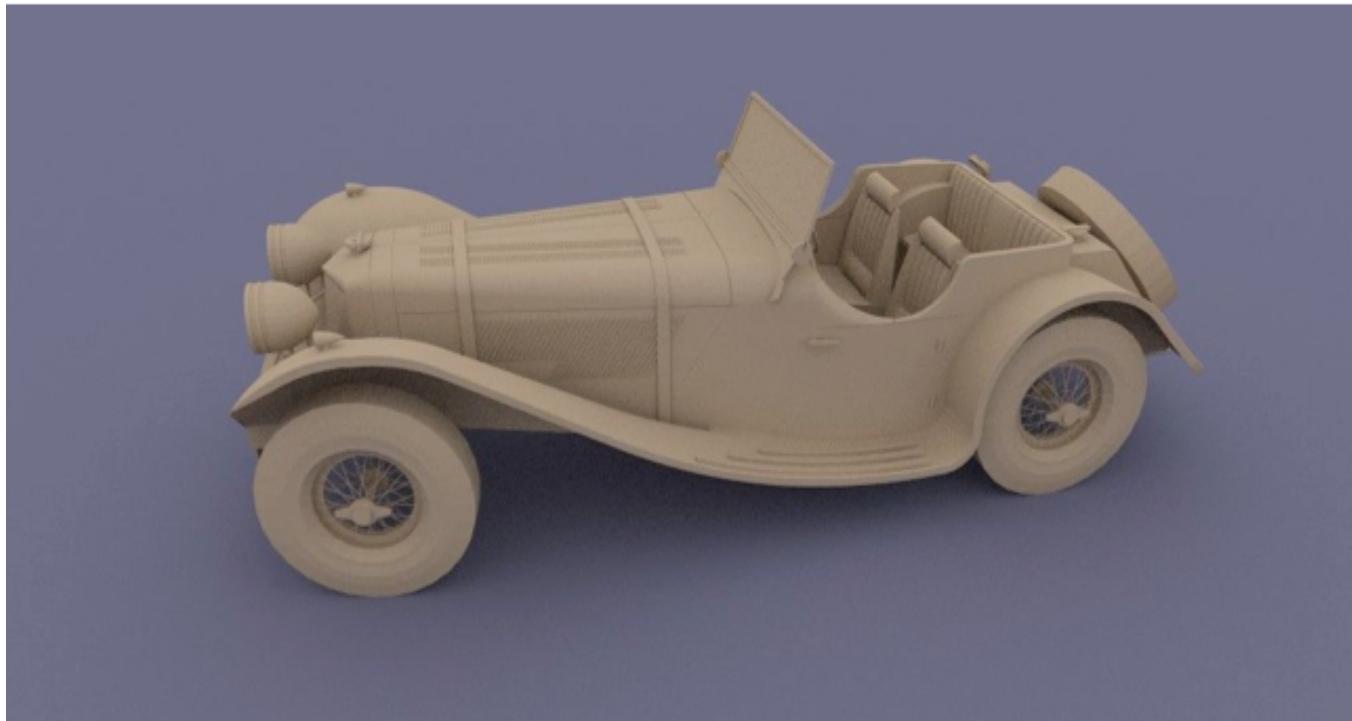
 For every light cast shadow ray

 col += local shading term



Graphics Lecture 10: Slide 7

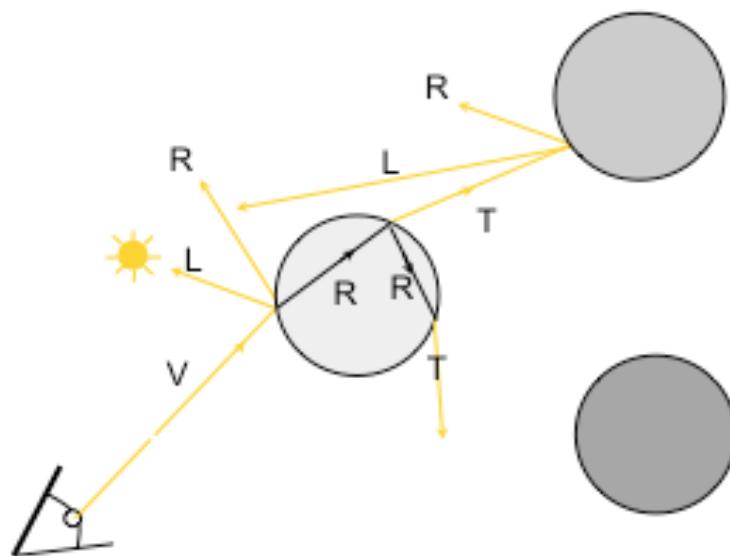
Ray casting



Graphics Lecture 10: Slide 8

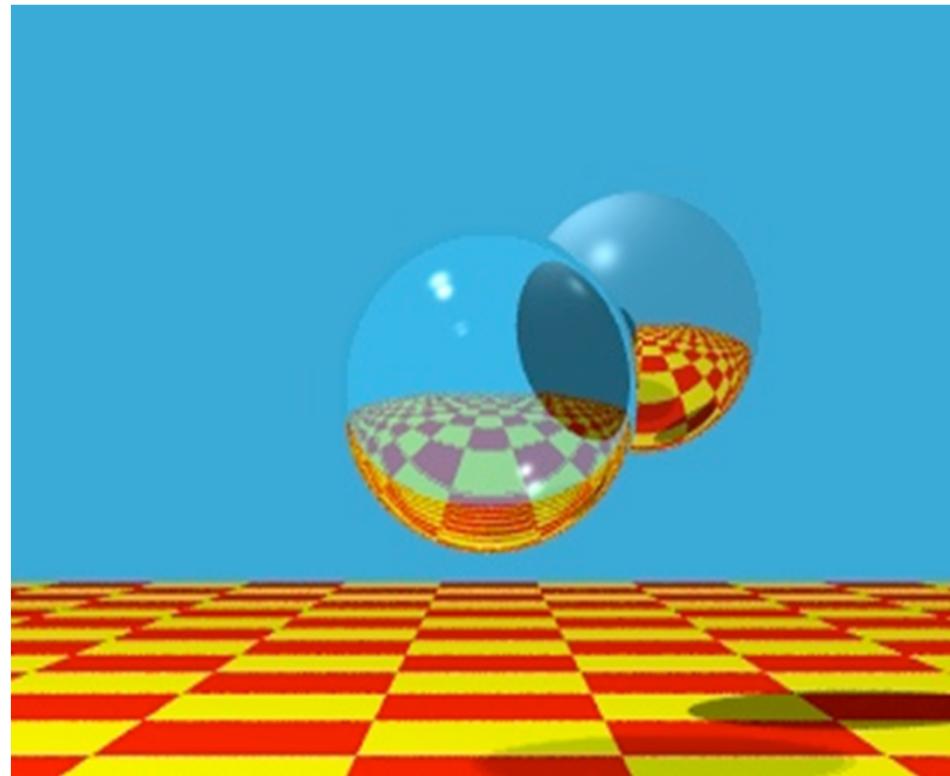
Turner Whitted, 1980

- An Improved Illumination Model for Shaded Display, 1980
- First global illumination model:
 - An object's color is influenced by lights and other objects in the scene
 - Simulates specular reflection and refractive transmission



Graphics Lecture 10: Slide 9

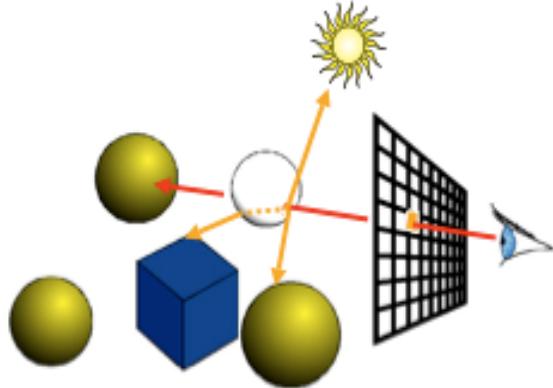
Turner Whitted, 1980



Graphics Lecture 10: Slide 10

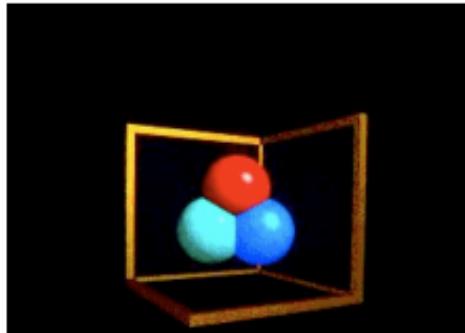
Recursive ray casting

```
trace ray
    Intersect all objects
    color = ambient term
    For every light
        cast shadow ray
        col += local shading term
    If mirror
        col += k_refl * trace reflected ray
    If transparent
        col += k_trans * trace transmitted ray
```



Does it ever end?

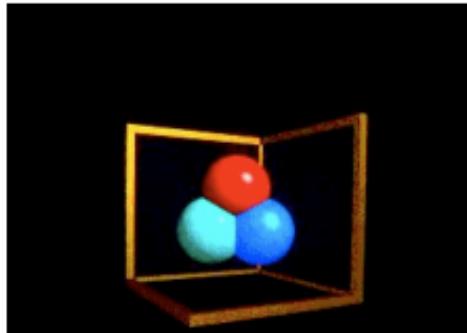
- Stopping criteria:
 - Recursion depth: Stop after a number of bounces
 - Ray contribution: Stop if reflected / transmitted contribution becomes too small



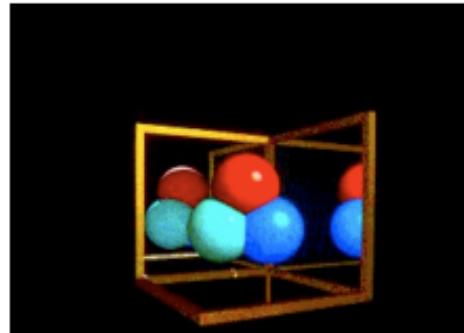
0 recursion

Does it ever end?

- Stopping criteria:
 - Recursion depth: Stop after a number of bounces
 - Ray contribution: Stop if reflected / transmitted contribution becomes too small



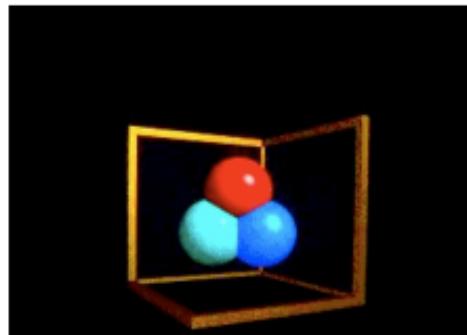
0 recursion



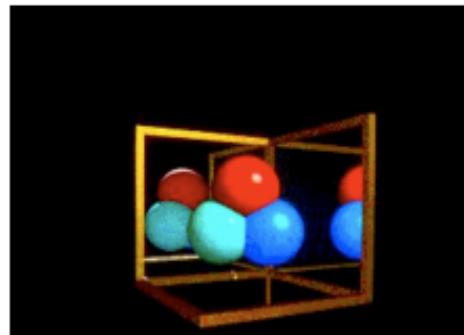
1 recursion

Does it ever end?

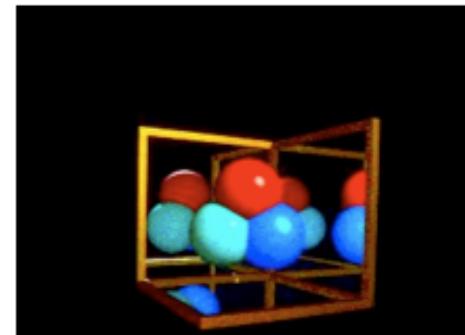
- Stopping criteria:
 - Recursion depth: Stop after a number of bounces
 - Ray contribution: Stop if reflected / transmitted contribution becomes too small



0 recursion



1 recursion



2 recursions

Ray tracing: Primary rays

- For each ray we need to test which objects are intersecting the ray:
 - If the object has an intersection with the ray we calculate the distance between viewpoint and intersection
 - If the ray has more than one intersection, the smallest distance identifies the visible surface.
- Primary rays are rays from the view point to the nearest intersection point
- Local illumination is computed as before:

$$L = k_a + (k_d(\mathbf{n} \cdot \mathbf{l}) + k_s(\mathbf{v} \cdot \mathbf{r})^q)I_s$$

Graphics Lecture 10: Slide 15

Ray tracing: Secondary rays

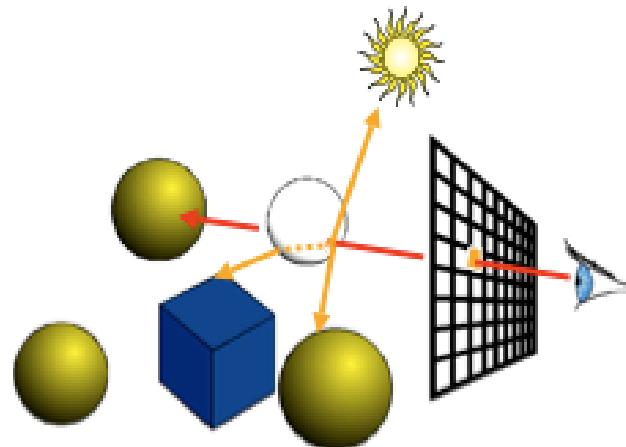
- Secondary rays are rays originating at the intersection points
- Secondary rays are caused by
 - rays reflected off the intersection point in the direction of reflection
 - rays transmitted through transparent materials in the direction of refraction
 - shadow rays

Graphics Lecture 10: Slide 16

Recursive ray tracing: Putting it all together

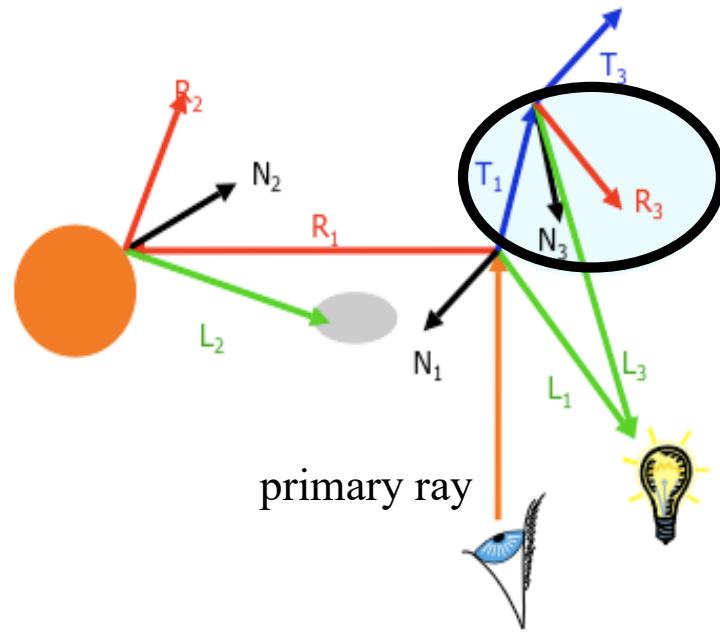
- Illumination can be expressed as

$$L = k_a + (k_d(\mathbf{n} \cdot \mathbf{l}) + k_s(\mathbf{v} \cdot \mathbf{r})^q)I_s + k_{reflected}L_{reflected} + k_{refracted}L_{refracted}$$

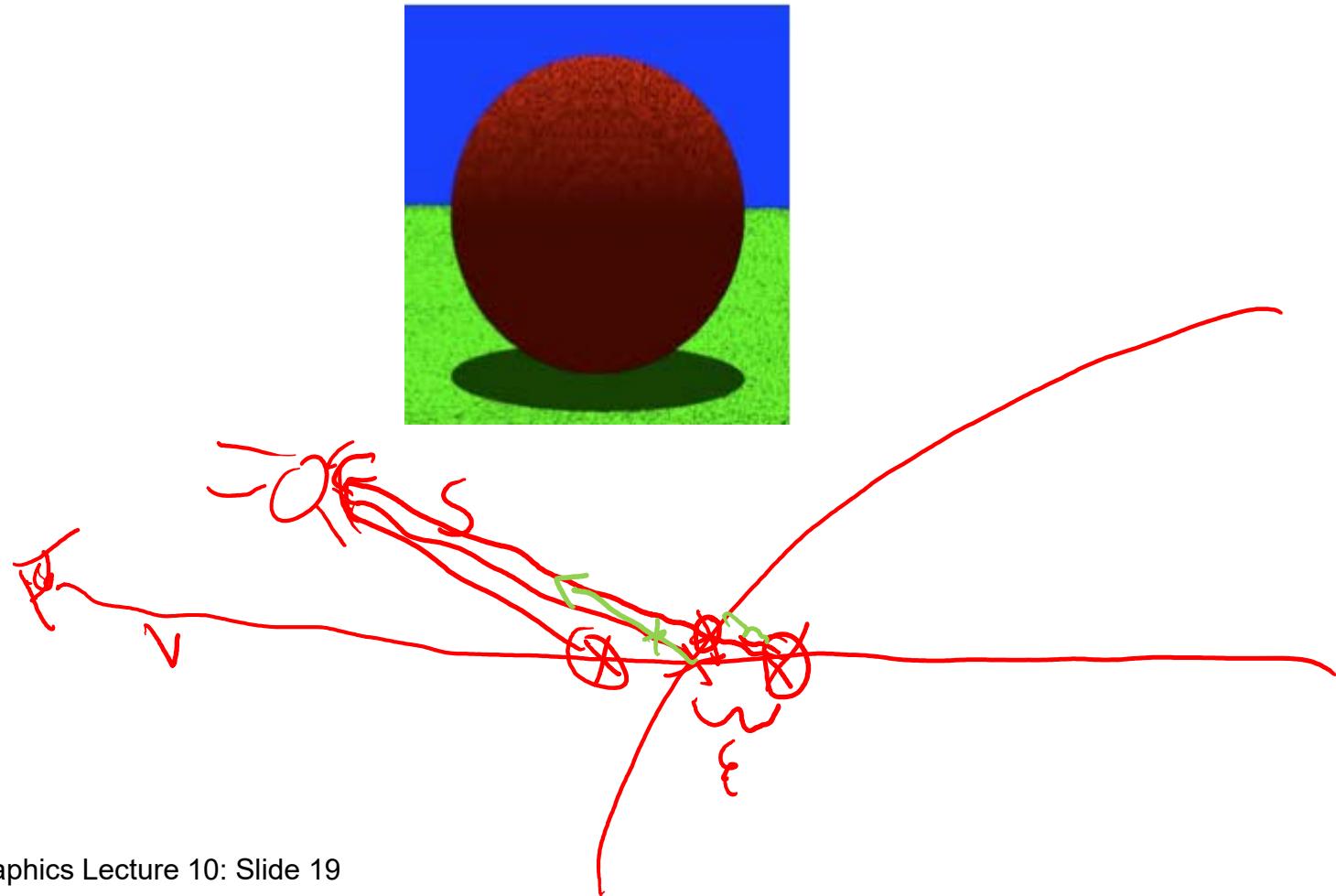


Graphics Lecture 10: Slide 17

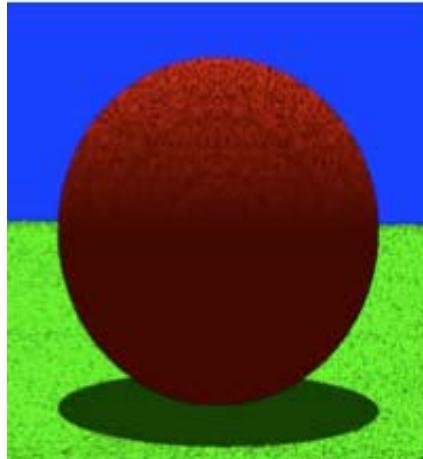
Recursive Ray Tracing: Ray Tree



Precision Problems



Precision Problems



- In ray tracing, the origin of (secondary) rays is often below the surface of objects
 - Theoretically, the intersection point should be on the surface
 - Practically, calculation imprecision creeps in, and the origin of the new ray is slightly beneath the surface
- Result: the surface area is shadowing itself or the ray continues on the wrong side

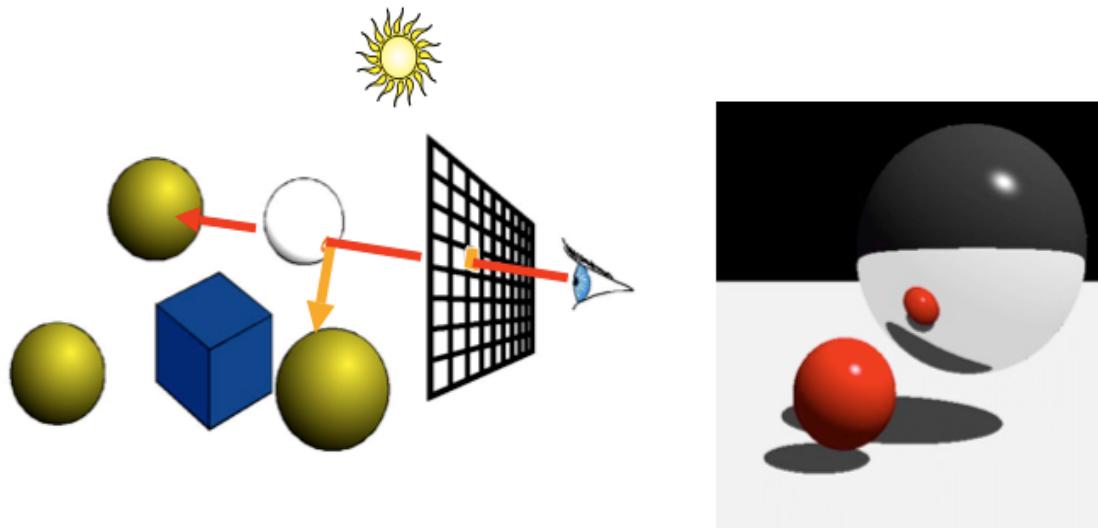
Graphics Lecture 10: Slide 20

ε to the rescue ...

- Check if t is within some epsilon tolerance:
 - if $\text{abs}(\mu) < \varepsilon$
 - point is on the surface
 - else
 - point is inside/outside
 - Choose the ε tolerance empirically
- Move the intersection point by epsilon along the surface normal so it is outside of the object
- Check if point is inside/outside surface by checking the sign of the implicit (sphere etc.) equation

Mirror reflection

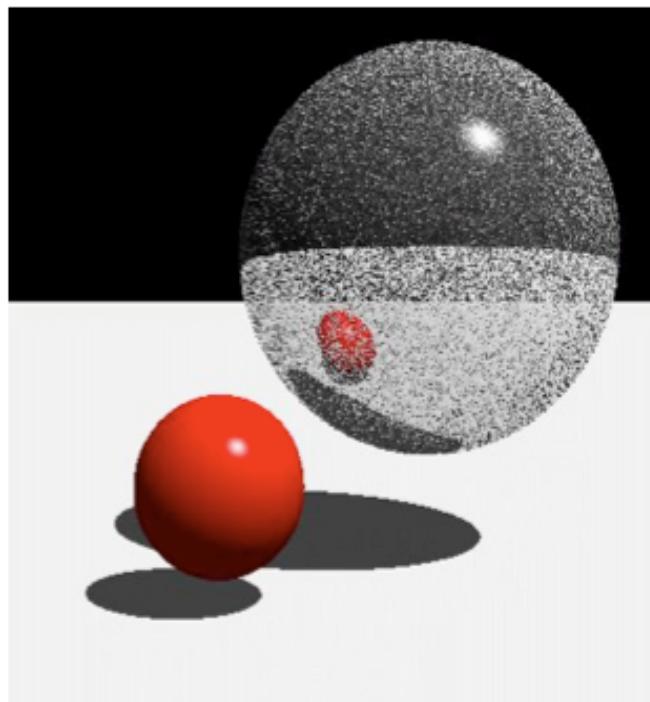
- Compute mirror contribution
- Cast ray in direction symmetric wrt. normal
- Multiply by reflection coefficient (color)



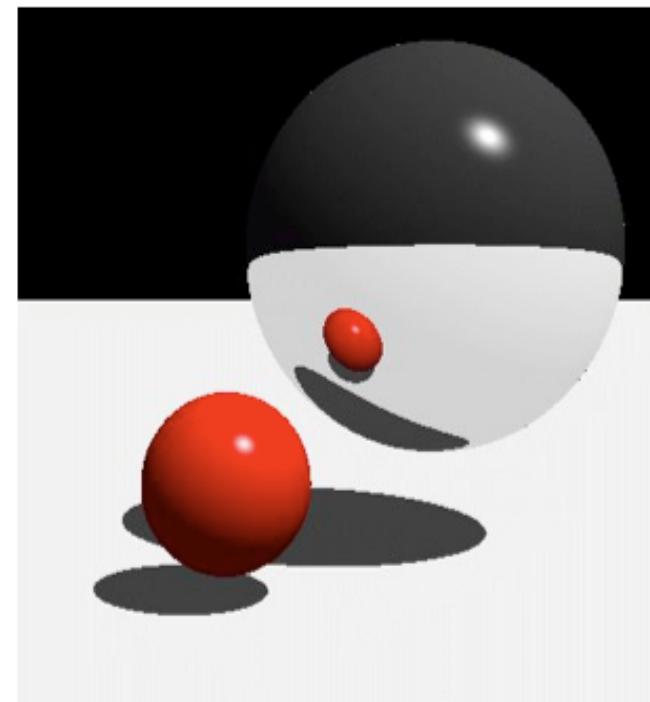
Graphics Lecture 10: Slide 22

Mirror reflection

- Don't forget to add epsilon to the ray



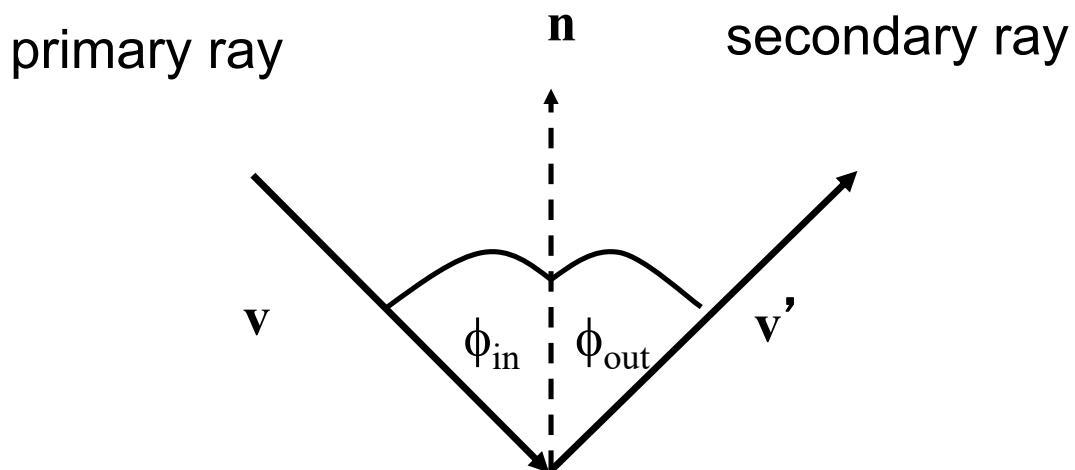
Without epsilon



With epsilon

Graphics Lecture 10: Slide 23

Mirror reflection

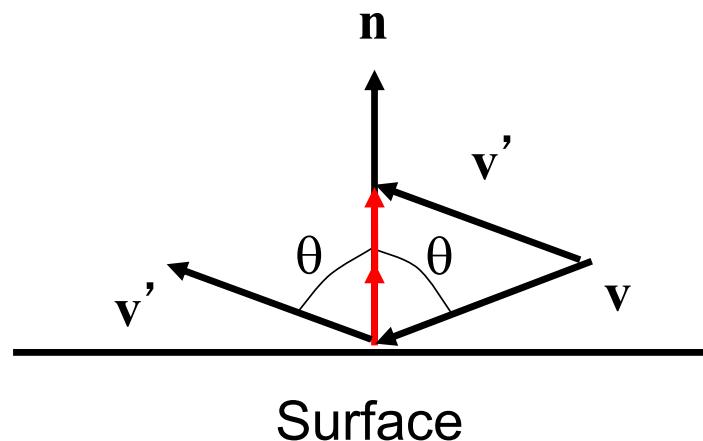


Mirror reflection

- To calculate illumination as a result of reflections
 - calculate the direction of the secondary ray at the intersection of the primary ray with the object.
- given that
- \mathbf{n} is the unit surface normal
 - \mathbf{v} is the direction of the primary ray
 - \mathbf{v}' is the direction of the secondary ray as a result of reflections

$$\mathbf{v}' = \mathbf{v} - (2\mathbf{v} \cdot \mathbf{n})\mathbf{n}$$

Mirror reflection

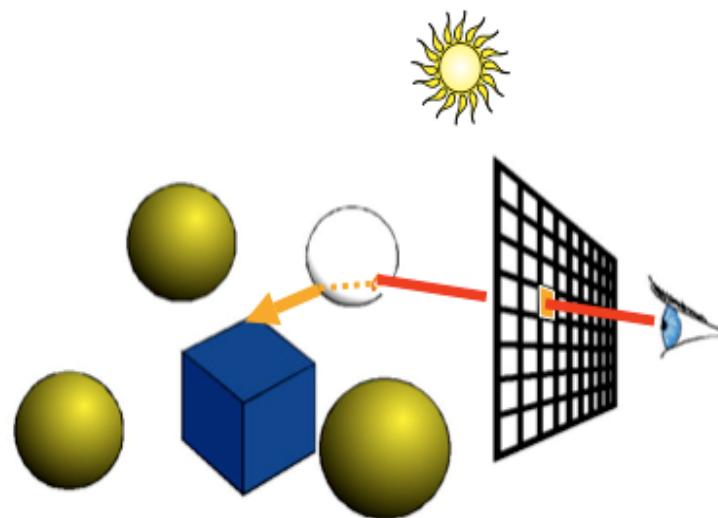


$$\mathbf{v}' = \mathbf{v} - (2\mathbf{v} \cdot \mathbf{n})\mathbf{n}$$

Graphics Lecture 10: Slide 26

Transparency

- Compute transmitted contribution
- Cast ray in refracted direction
- Multiply by transparency coefficient



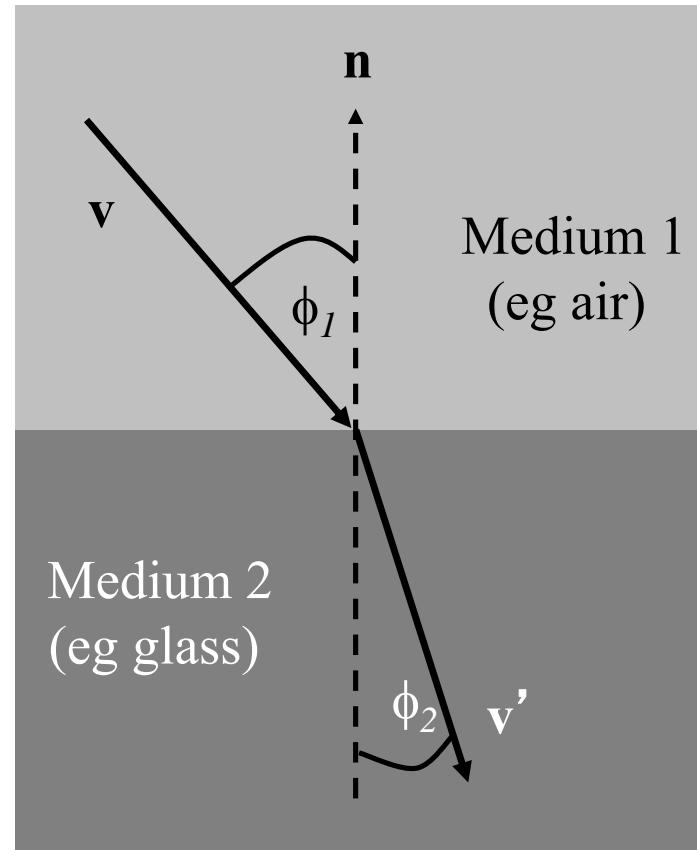
Graphics Lecture 10: Slide 28

Refraction

- The angle of the refracted ray can be determined by Snell's law:

$$\eta_1 \sin(\phi_1) = \eta_2 \sin(\phi_2)$$

- η_1 is a constant for medium 1
- η_2 is a constant for medium 2
- ϕ_1 is the angle between the incident ray and the surface normal
- ϕ_2 is the angle between the refracted ray and the surface normal



Refraction

- In vector notation Snell's law can be written:

$$k_1(v \cdot n) = k_2(v' \cdot n)$$

- The direction of the refracted ray is

$$\mathbf{v}' = \frac{\eta_1}{\eta_2} \left(\sqrt{\left(\mathbf{n} \cdot \mathbf{v}\right)^2 + \left(\frac{\eta_2}{\eta_1}\right)^2} - 1 - \mathbf{n} \cdot \mathbf{v} \right) \cdot \mathbf{n} + \mathbf{v}$$

Refraction

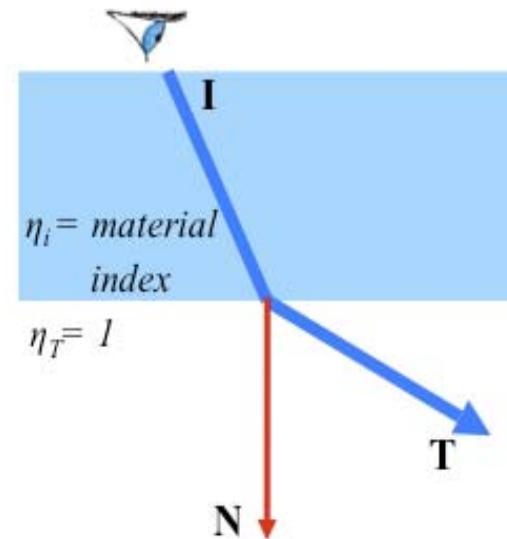
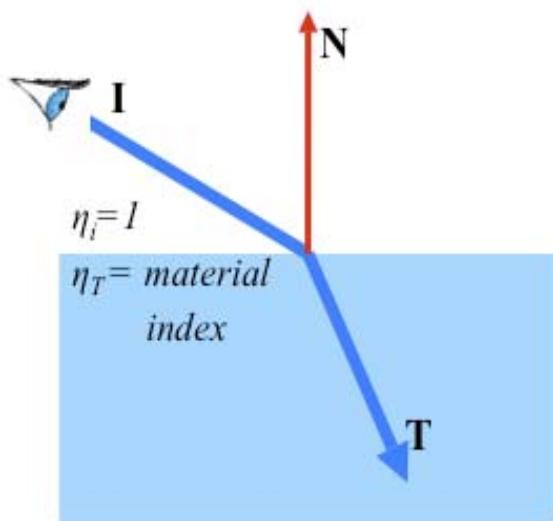
- This equation only has a solution if

$$(\mathbf{n} \cdot \mathbf{v})^2 > 1 - \left(\frac{\eta_2}{\eta_1} \right)^2$$

- This illustrates the physical phenomenon of the limiting angle:
 - if light passes from one medium to another medium whose index of refraction is low, the angle of the refracted ray is greater than the angle of the incident ray
 - if the angle of the incident ray is large, the angle of the refracted ray is larger than 90°
 - ➔ the ray is reflected rather than refracted

Refraction

- Make sure you know whether you are entering or leaving the transmissive material

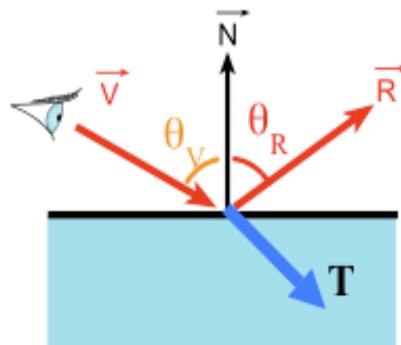


Graphics Lecture 10: Slide 32

Amount of reflection and refraction

- Traditional (hacky) ray tracing
 - Constant coefficient reflection
 - Component per component multiplication
- Better: Mix reflected and refracted light according to the Fresnel factor.

$$L = k_{fresnel} L_{reflected} + (1 - k_{fresnel}) L_{refracted}$$



Graphics Lecture 10: Slide 33

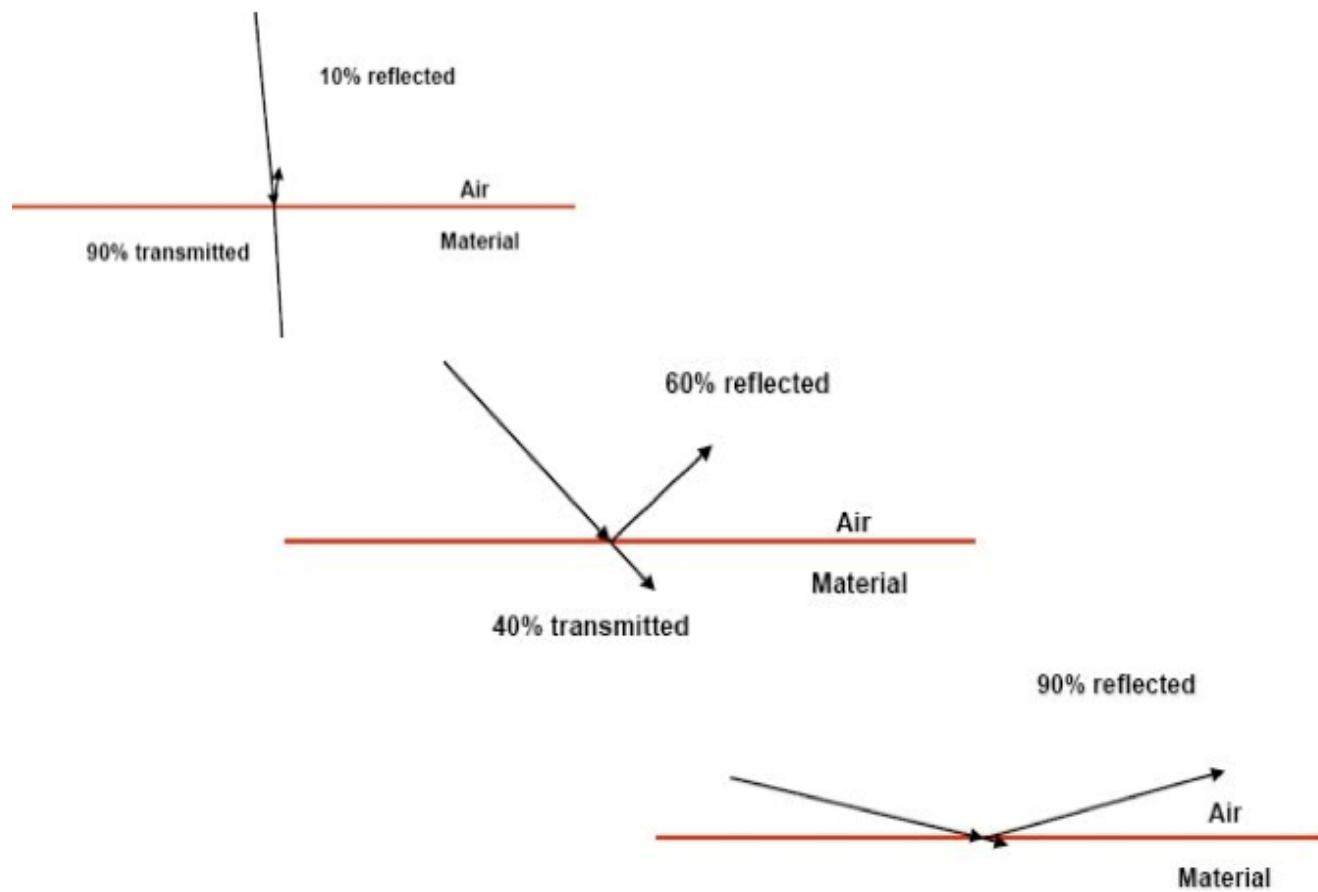
Fresnel factor

- More reflection at grazing angle



Graphics Lecture 10: Slide 34

Fresnel factor



Graphics Lecture 10: Slide 35

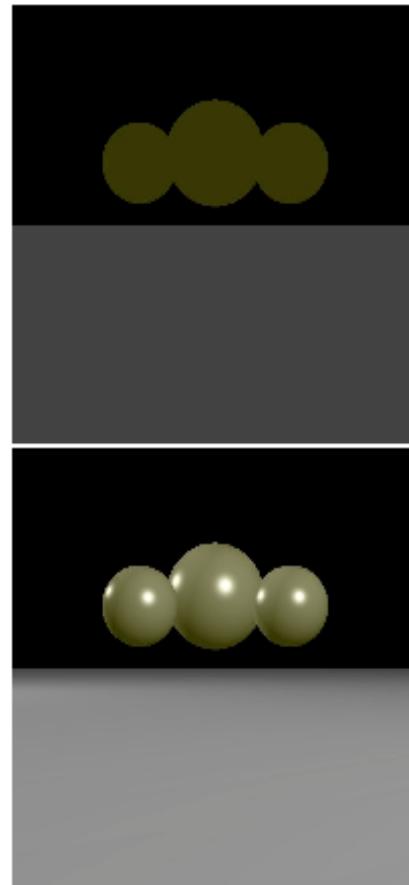
Schlick's Approximation

- Schlick' s approximation

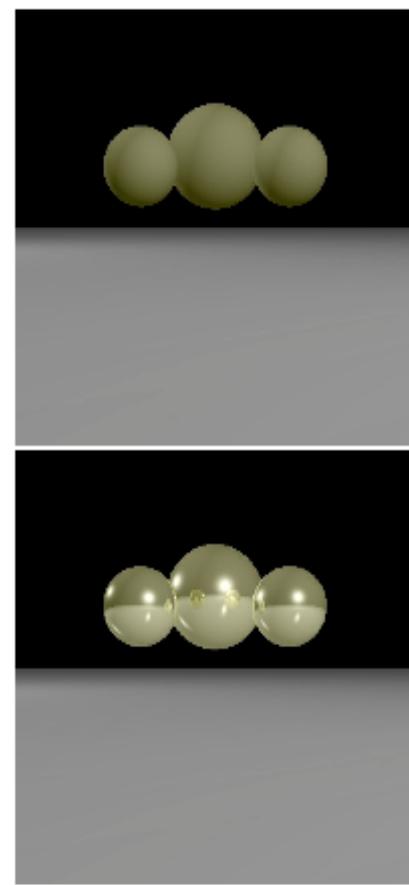
$$k_{fresnel}(\theta) = k_{fresnel}(0) + (1 - k_{fresnel}(0))(1 - (\mathbf{n} \cdot \mathbf{l}))^5$$

- $k_{fresnel}(0)$ = Fresnel factor at zero degrees
- Choose $k_{fresnel}(0) = 0.8$, this will look like stainless steel
- Fresnel factor at zero degrees has low value ~ 0.05 for dielectric materials like plastic.

Example

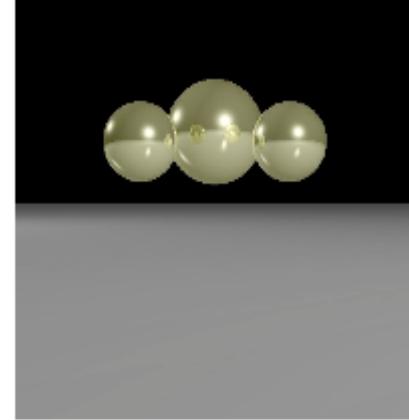


Ambient



+ Diffuse

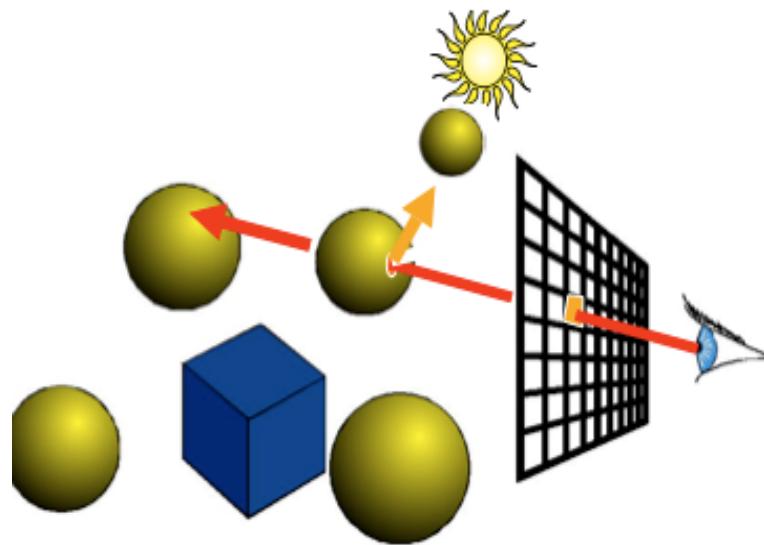
+ Specular



+ $I_{\text{reflected}}$

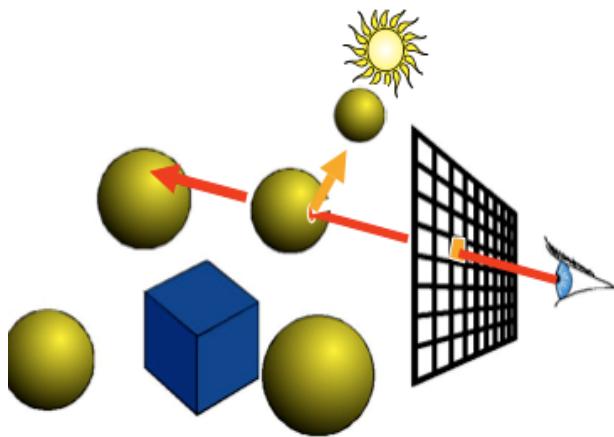
Graphics Lecture 10: Slide 37

How do we add shadows?



Graphics Lecture 10: Slide 38

How do we add shadows?

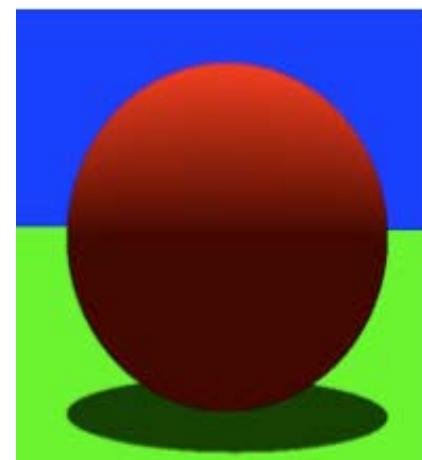
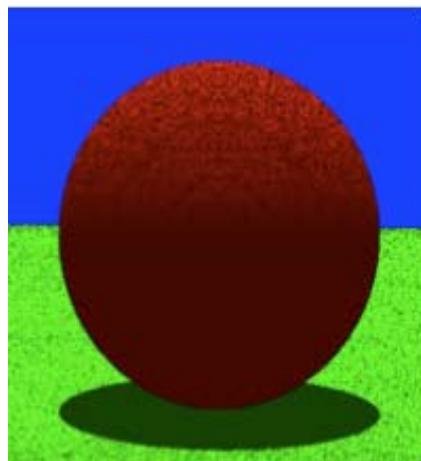


$$L = k_a + s(k_d(\mathbf{n} \cdot \mathbf{l}) + k_s(\mathbf{v} \cdot \mathbf{r})^q)I_s + k_{reflected}L_{reflected} + k_{refracted}L_{refracted}$$

$$s = \begin{cases} 0 & \text{if light source is obscured} \\ 1 & \text{if light source is not obscured} \end{cases}$$

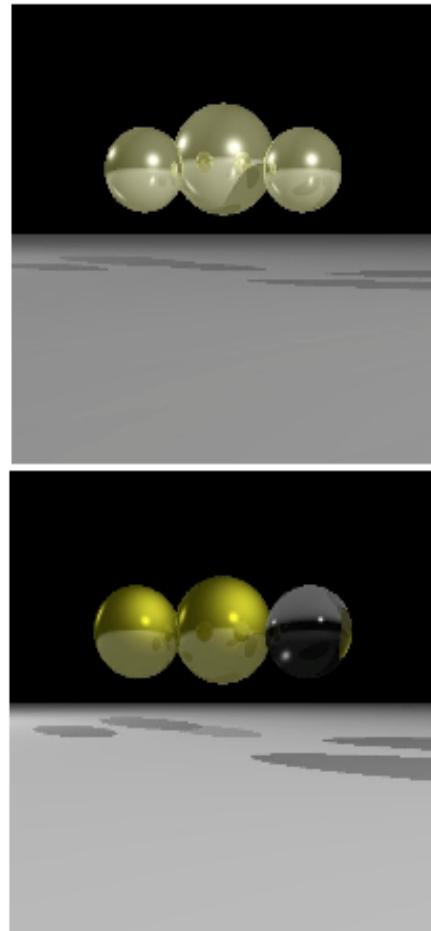
Shadows: Problems?

- Make sure to avoid self-shadowing

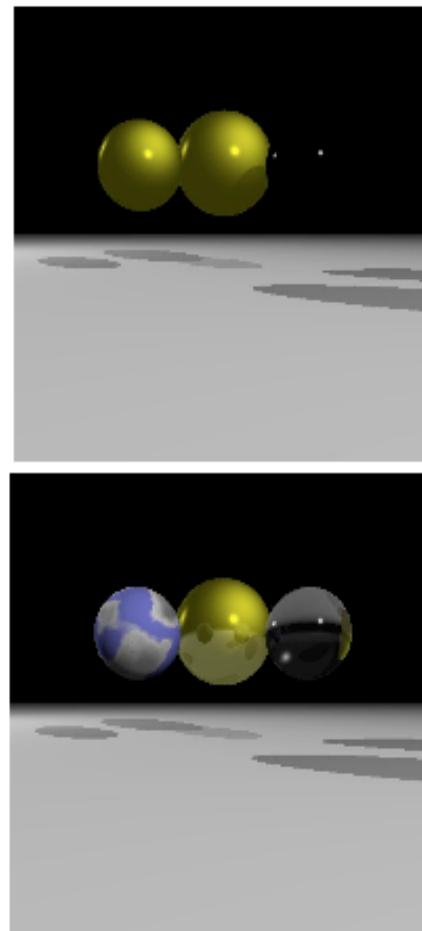


Graphics Lecture 10: Slide 40

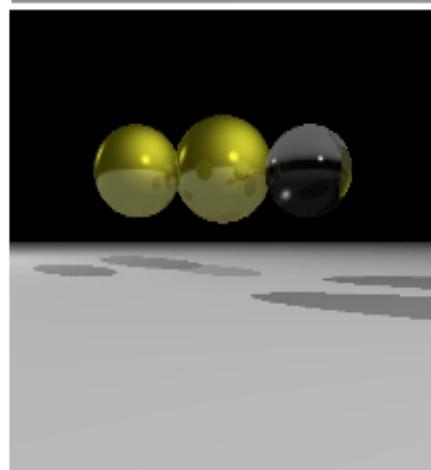
Example



+ Shadows



- $I_{\text{reflected}}$
+ transmitted



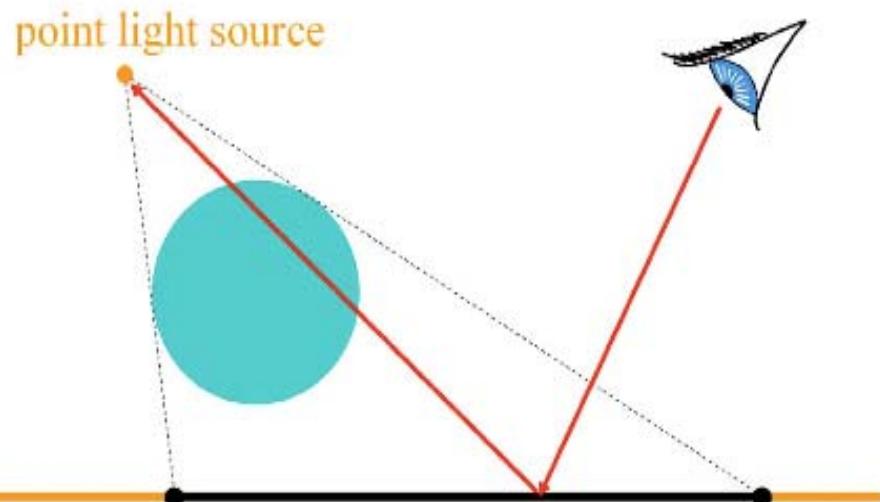
+ $I_{\text{reflected}}$

+ textures

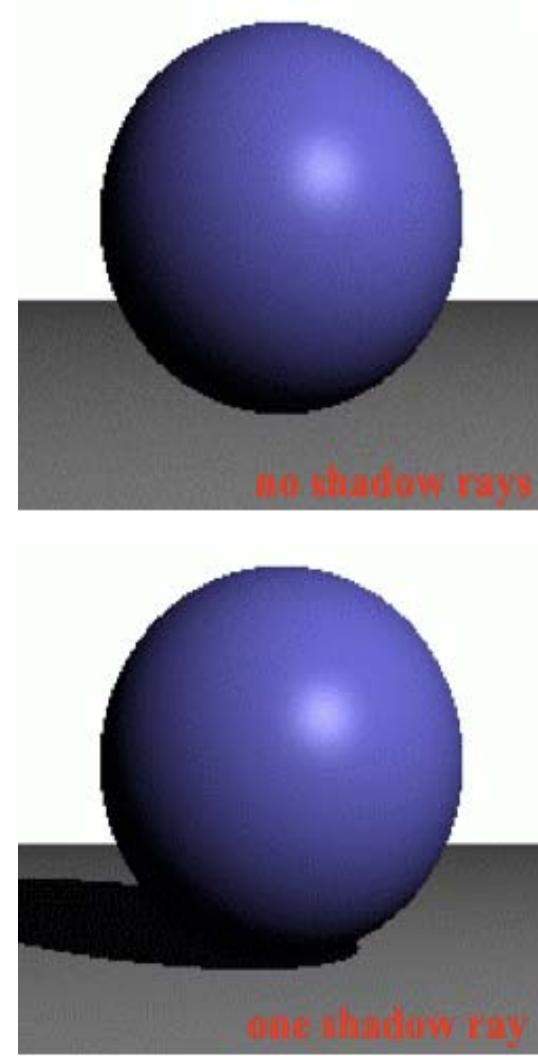
Graphics Lecture 10: Slide 41

Shadows

- One shadow ray per intersection per point light source

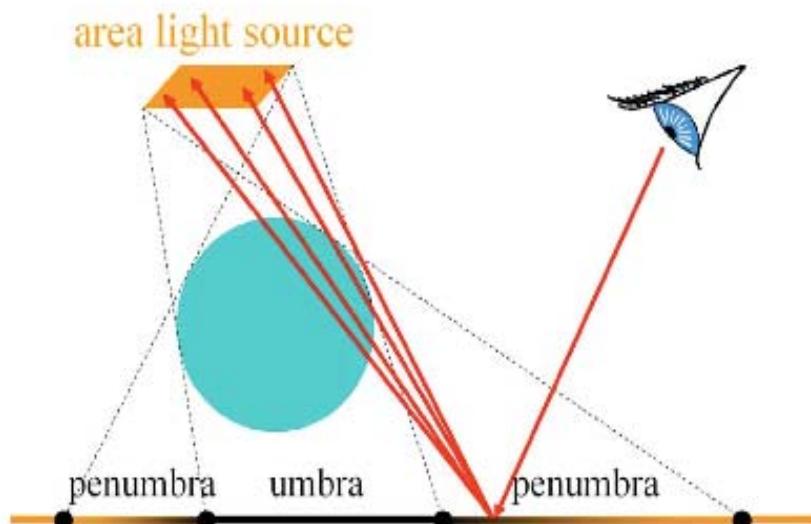


Graphics Lecture 10: Slide 42

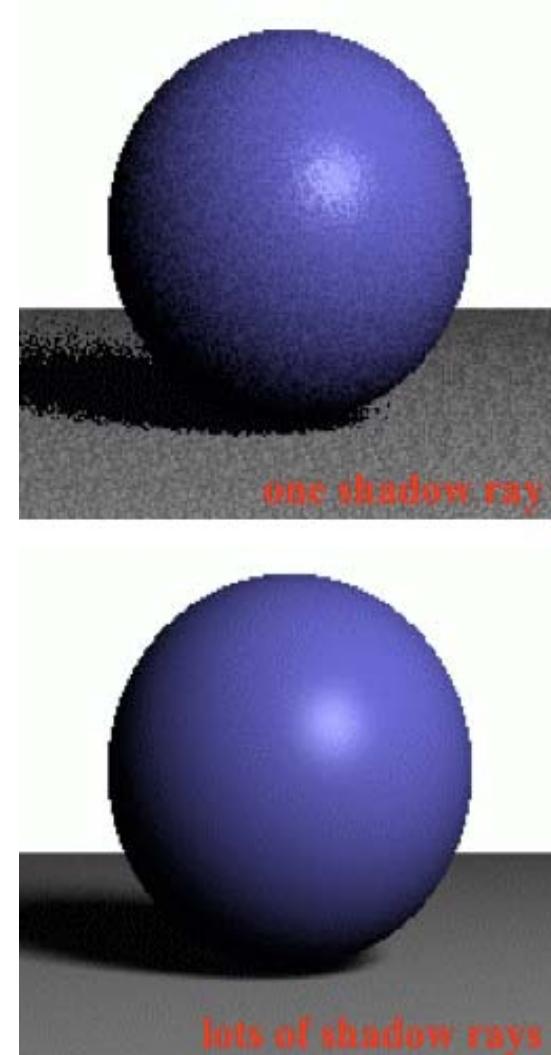


Soft shadows

- Multiple shadow rays to sample area light source

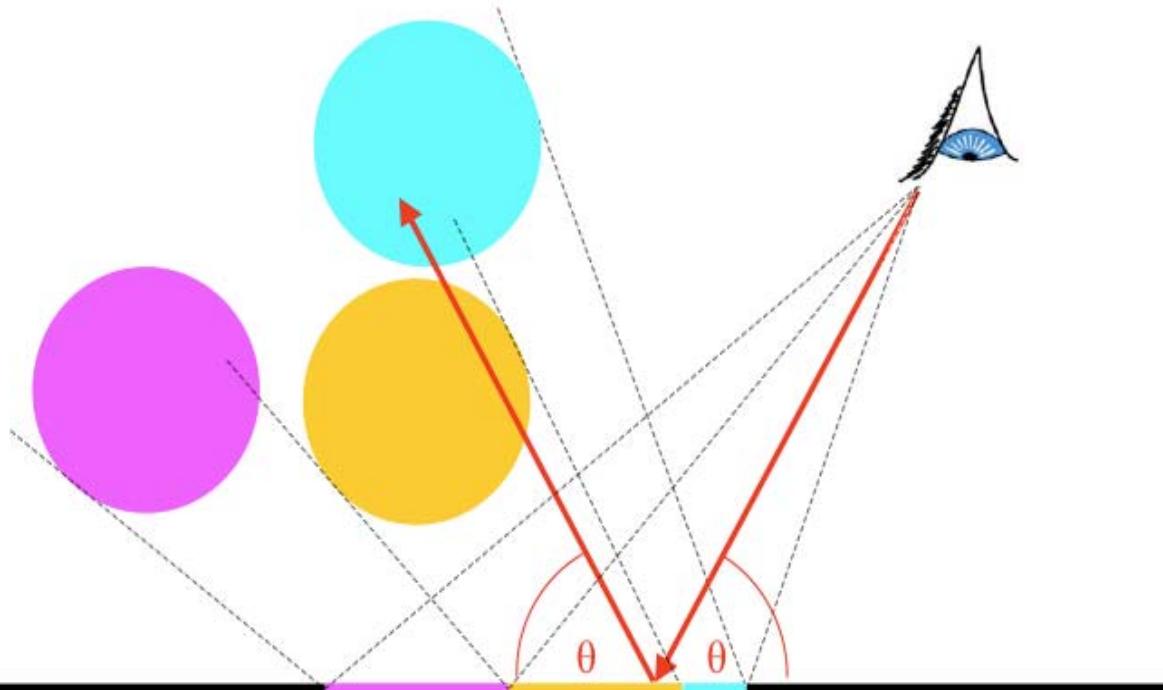


Graphics Lecture 10: Slide 43



Reflection: Conventional ray tracing

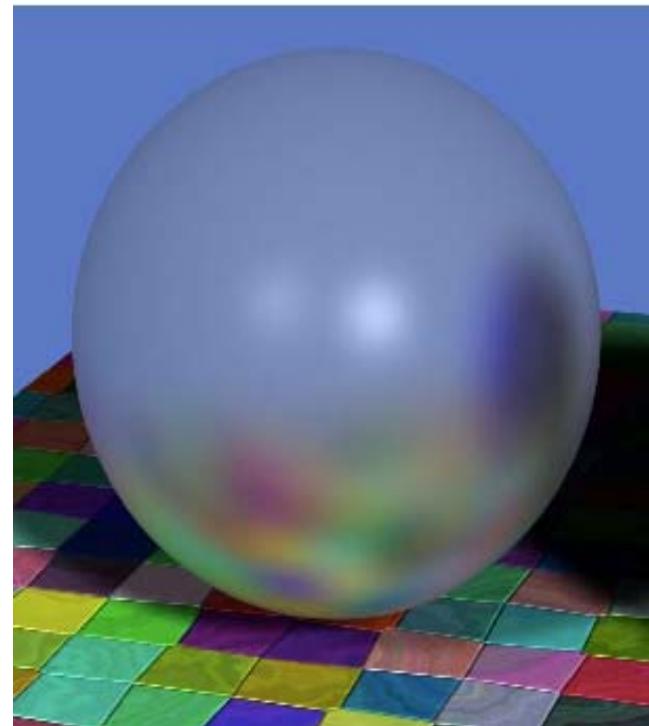
- One reflection per intersection



Graphics Lecture 10: Slide 44

Reflection: Conventional ray tracing

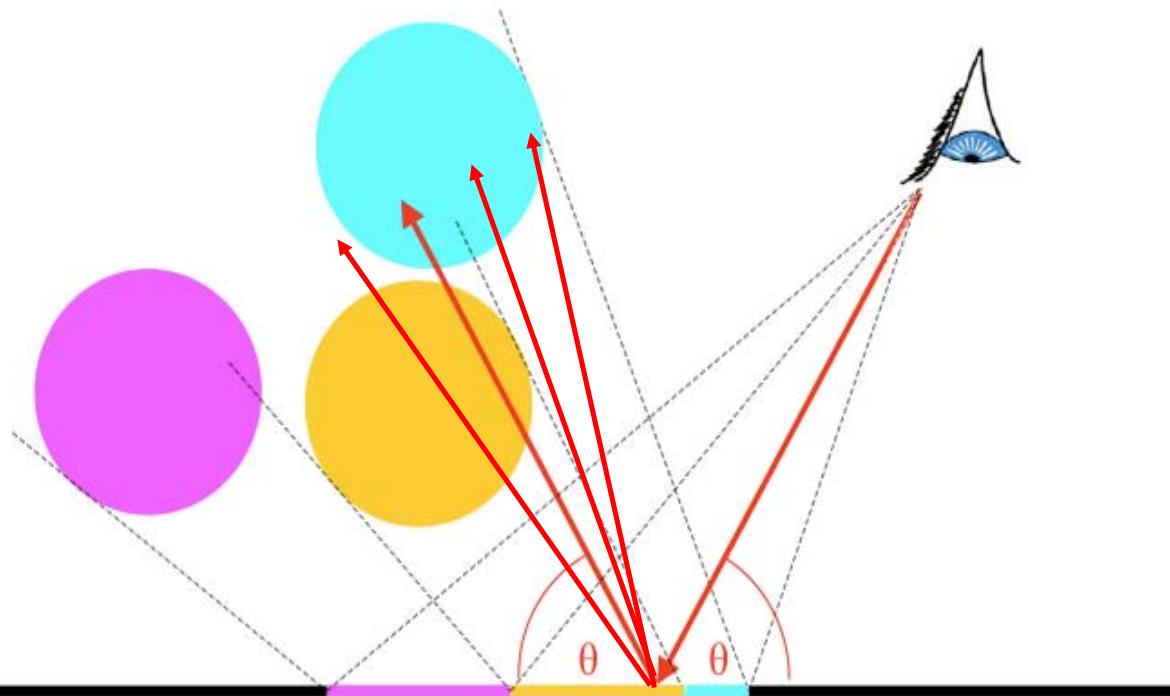
- How can we create effects like this?



Graphics Lecture 10: Slide 45

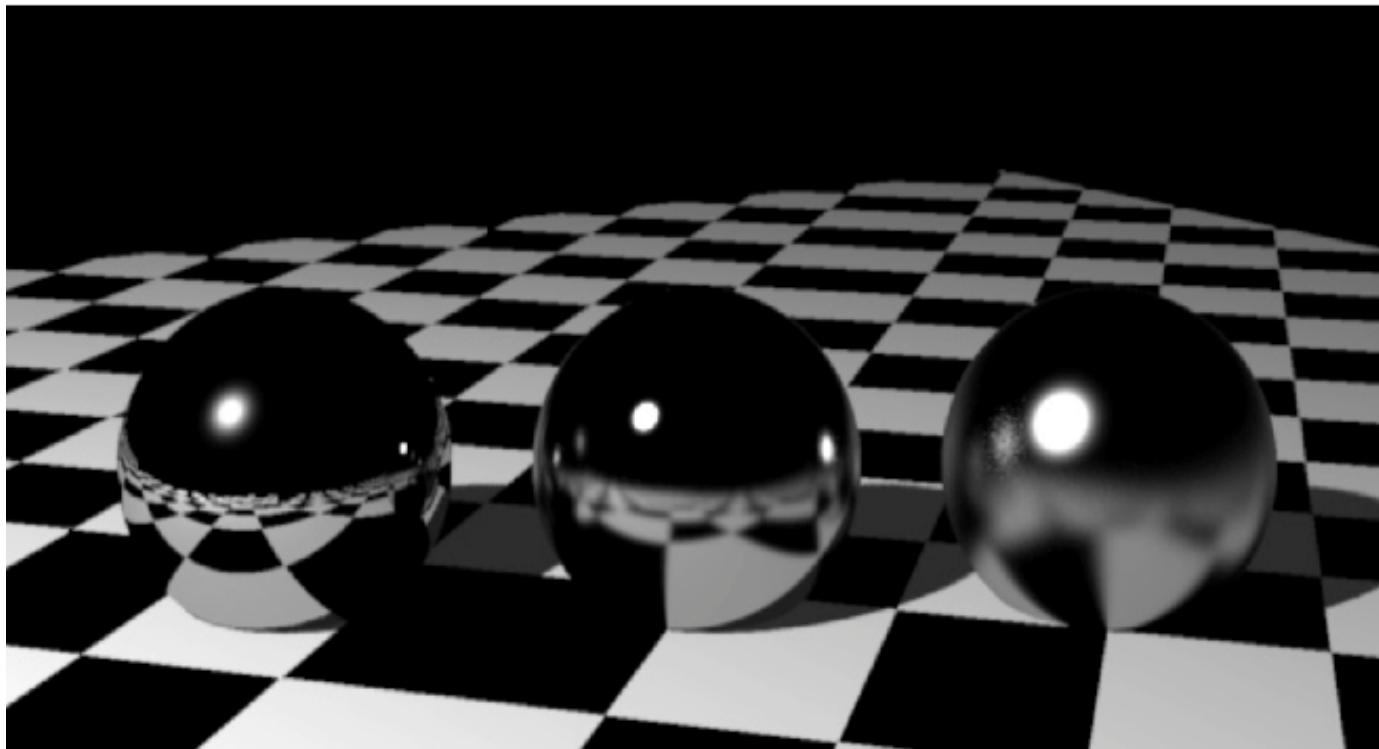
Reflection: Monte Carlo ray tracing

- Random reflection rays around mirror direction



Graphics Lecture 10: Slide 46

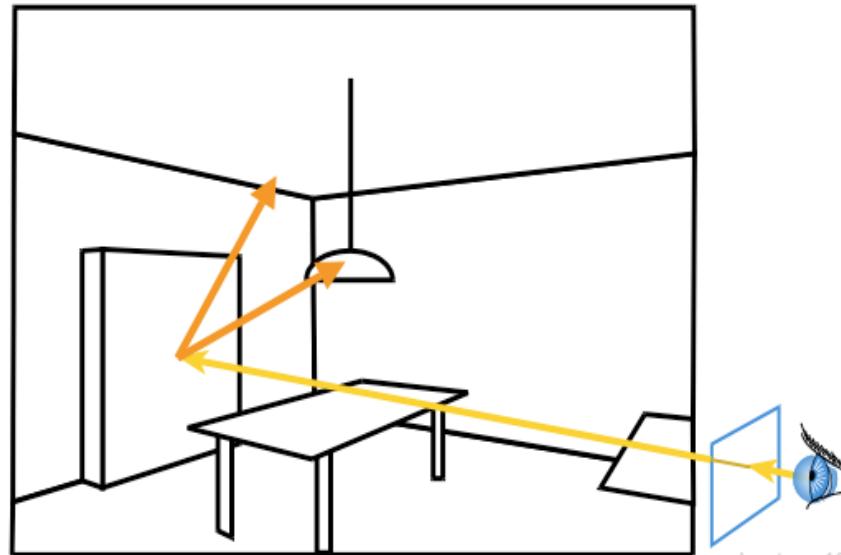
Glossy surfaces



Graphics Lecture 10: Slide 47

Ray tracing

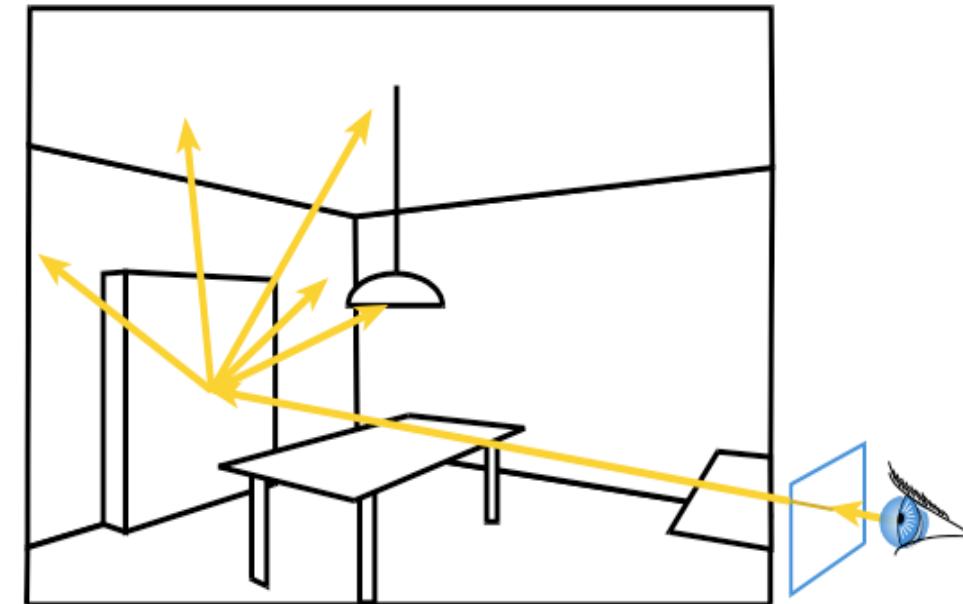
- Cast a ray from the eye through each pixel
- Trace secondary rays (light, reflection, refraction)



Graphics Lecture 10: Slide 48

Monte-Carlo Ray Tracing

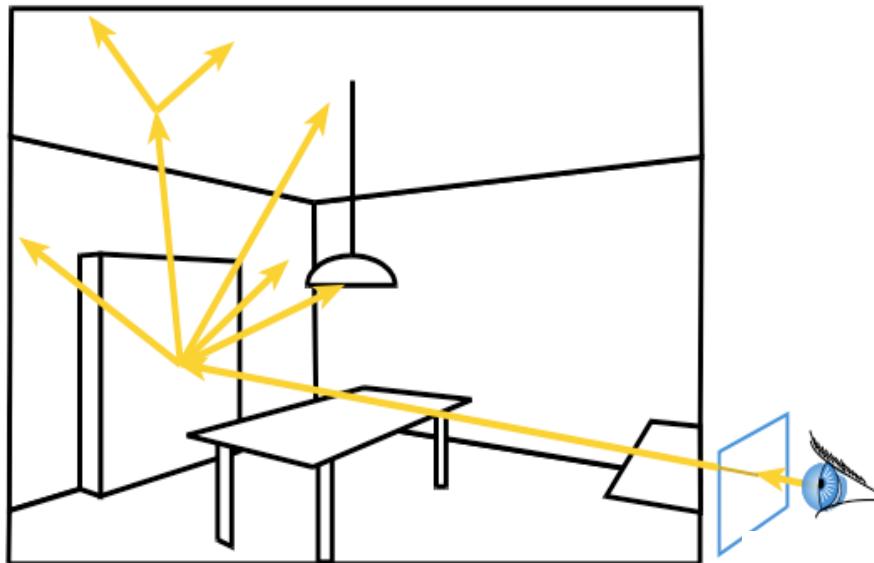
- Cast a ray from the eye through each pixel
- Cast random rays from the visible point
 - Accumulate radiance contribution



Graphics Lecture 10: Slide 49

Monte-Carlo Ray Tracing

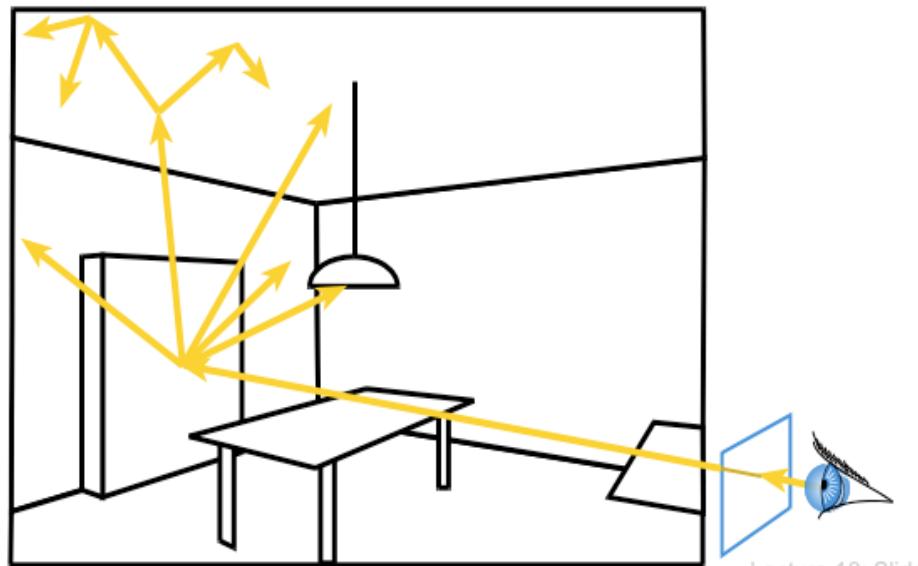
- Cast a ray from the eye through each pixel
- Cast random rays from the visible point
- Recurse



Graphics Lecture 10: Slide 50

Monte-Carlo Ray Tracing

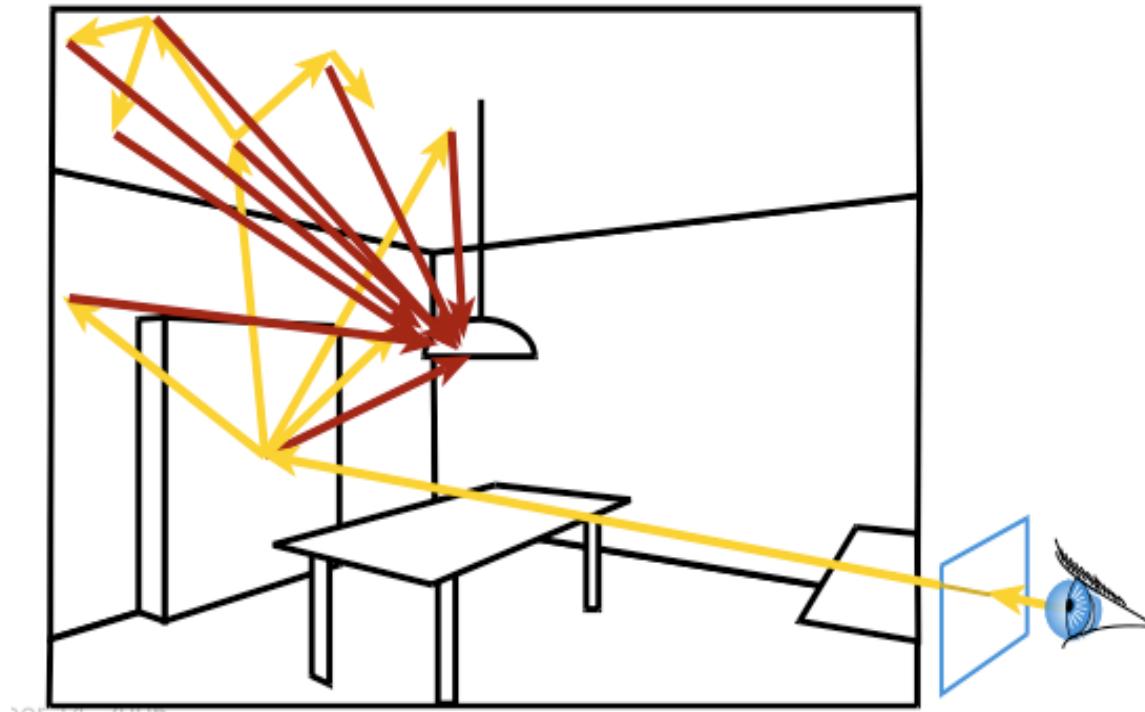
- Cast a ray from the eye through each pixel
- Cast random rays from the visible point
- Recurse



Graphics Lecture 10: Slide 51

Monte-Carlo Ray Tracing

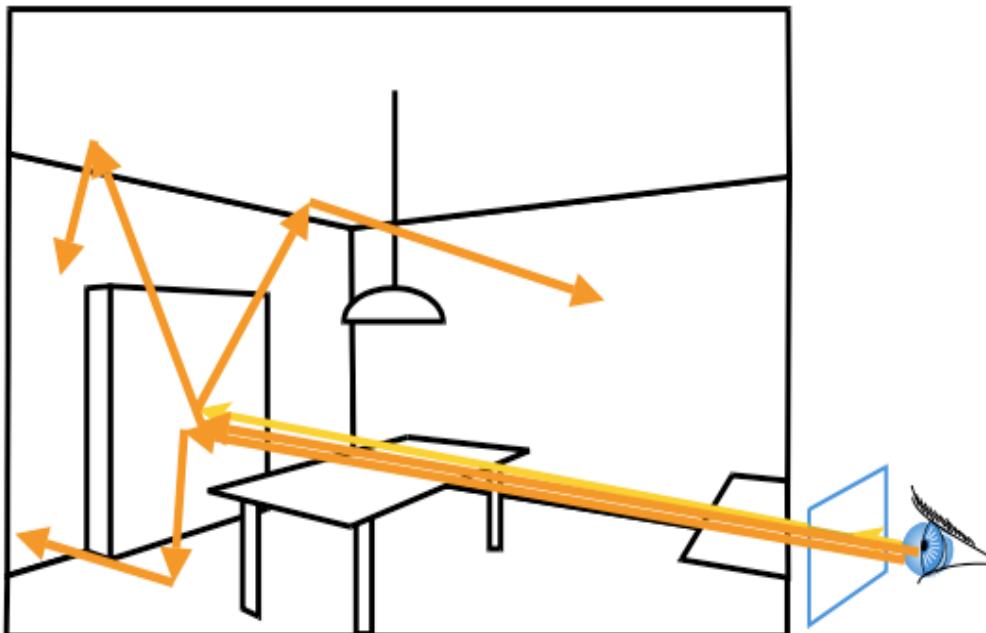
- Send rays to light



Graphics Lecture 10: Slide 52

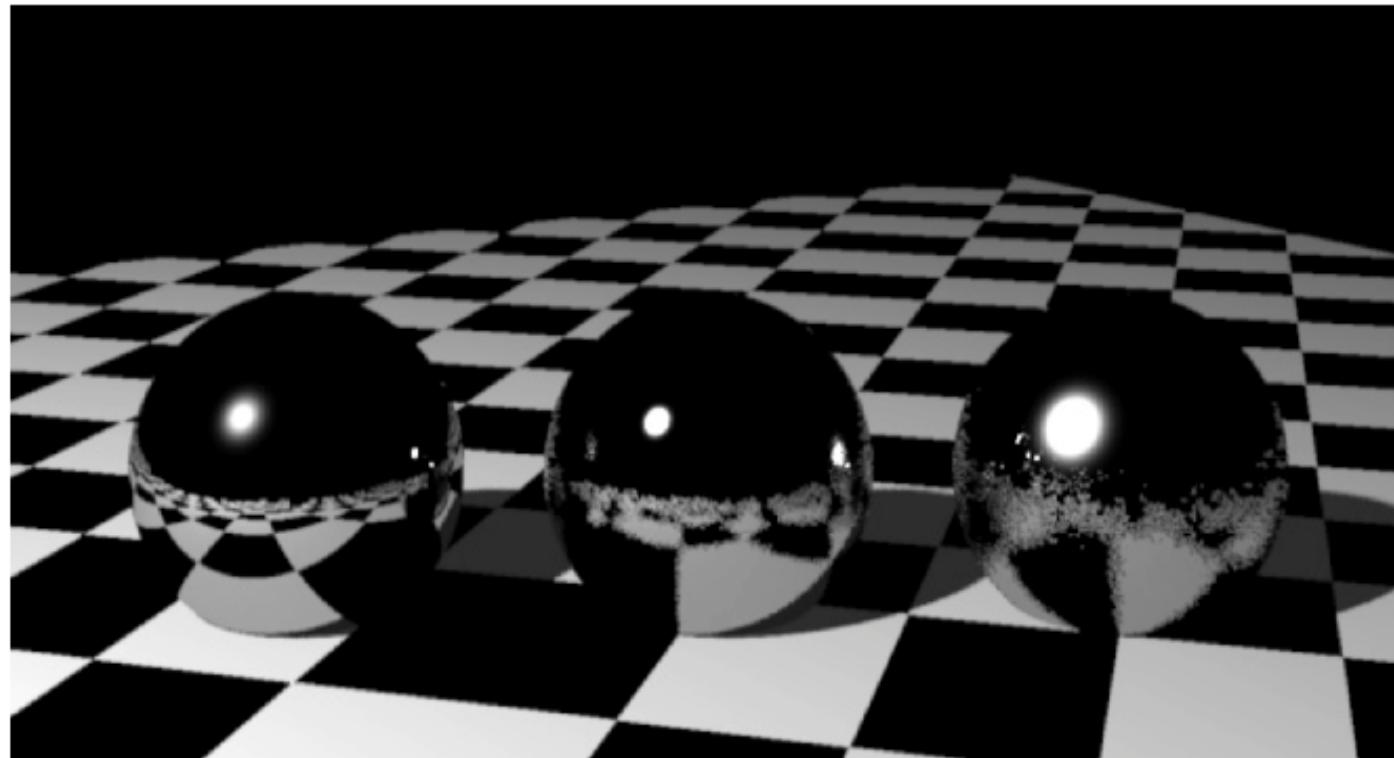
Monte-Carlo Path Tracing

- Trace only one secondary ray per recursion
- But send many primary rays per pixel



Graphics Lecture 10: Slide 53

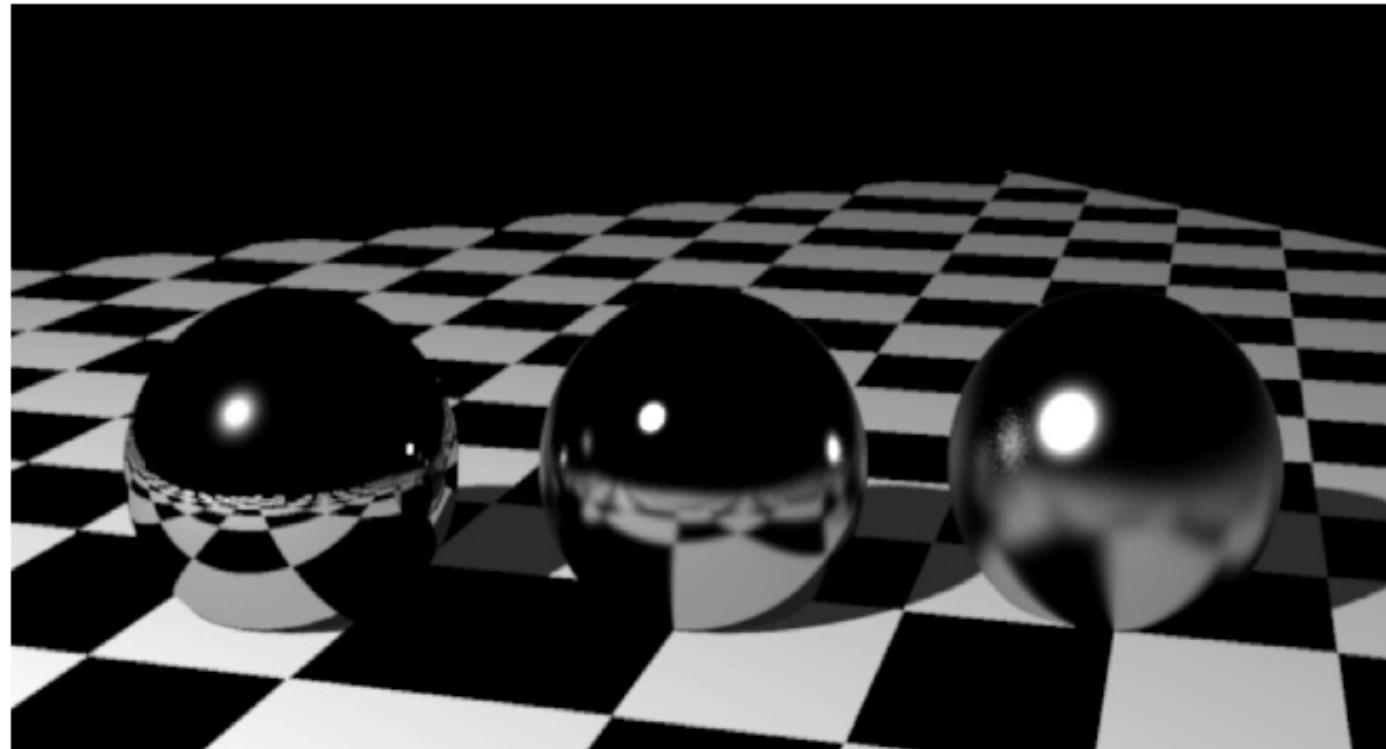
Example



1 sample per ray

Graphics Lecture 10: Slide 54

Example



256 samples per ray

Graphics Lecture 10: Slide 55

Some cool pictures



Graphics Lecture 10: Slide 56

Some cool pictures

"Fukaya House" Waro Kishi



Graphics Lecture 10: Slide 57

Some cool pictures



Graphics Lecture 10: Slide 58

Some cool pictures



Graphics Lecture 10: Slide 59

Some cool pictures



Copyright 2000 Gilles Tran

Graphics Lecture 10: Slide 6

Some cool pictures

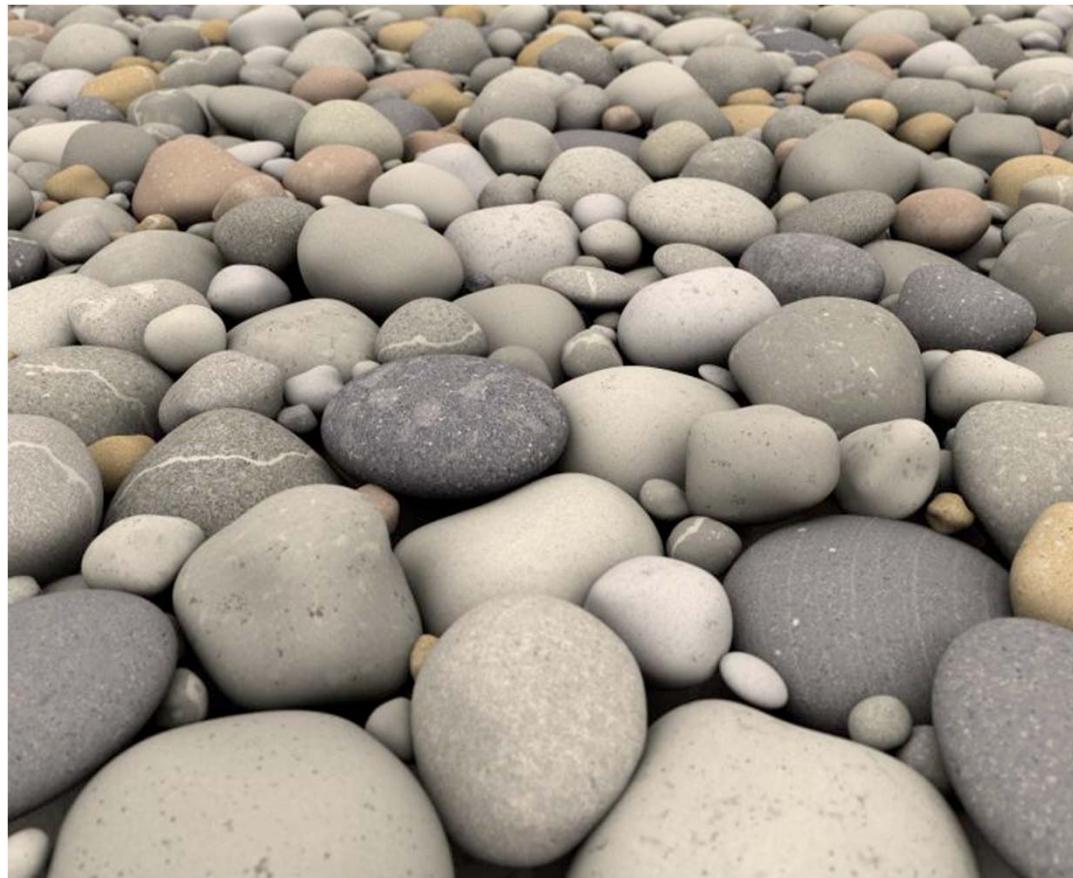


Some cool pictures



Graphics Lecture 10: Slide 62

Some cool pictures



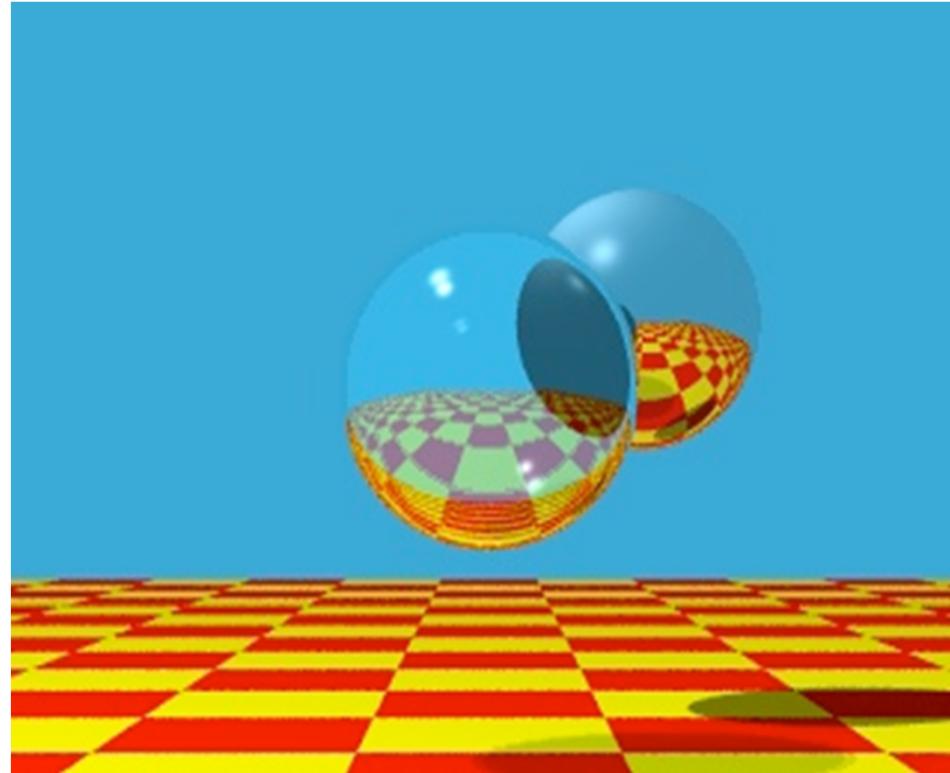
Graphics Lecture 10: Slide 63

took 4.5 days to render! (10 years ago)

Interactive Computer Graphics: Lecture 11

Ray tracing (cont.)

Ray tracing - Summary



Graphics Lecture 11: Slide 2

Ray tracing - Summary

trace ray

 Intersect all objects

 color = ambient term

 For every light

 cast shadow ray

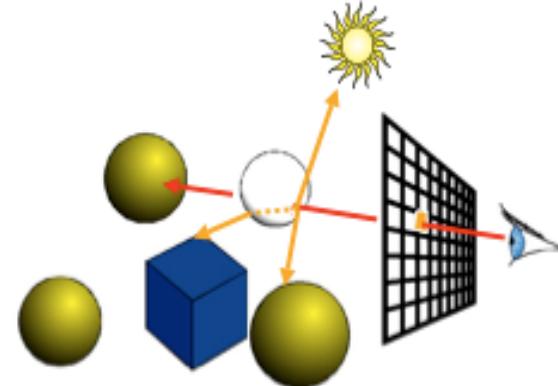
$\text{col} += \text{local shading term}$

 If mirror

$\text{col} += \text{k_refl} * \text{trace reflected ray}$

 If transparent

$\text{col} += \text{k_trans} * \text{trace transmitted ray}$



Ray tracing - Summary

trace ray

Intersect all objects

color = ambient term

For every light

 cast shadow ray

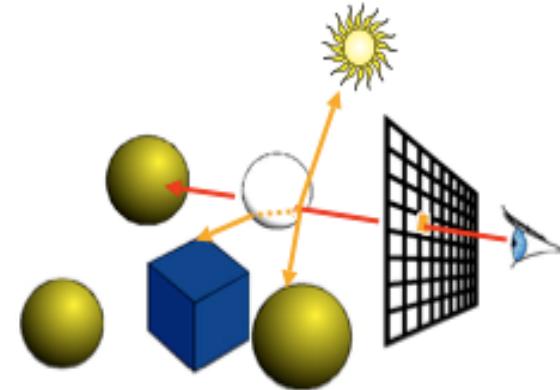
 col += local shading term

If mirror

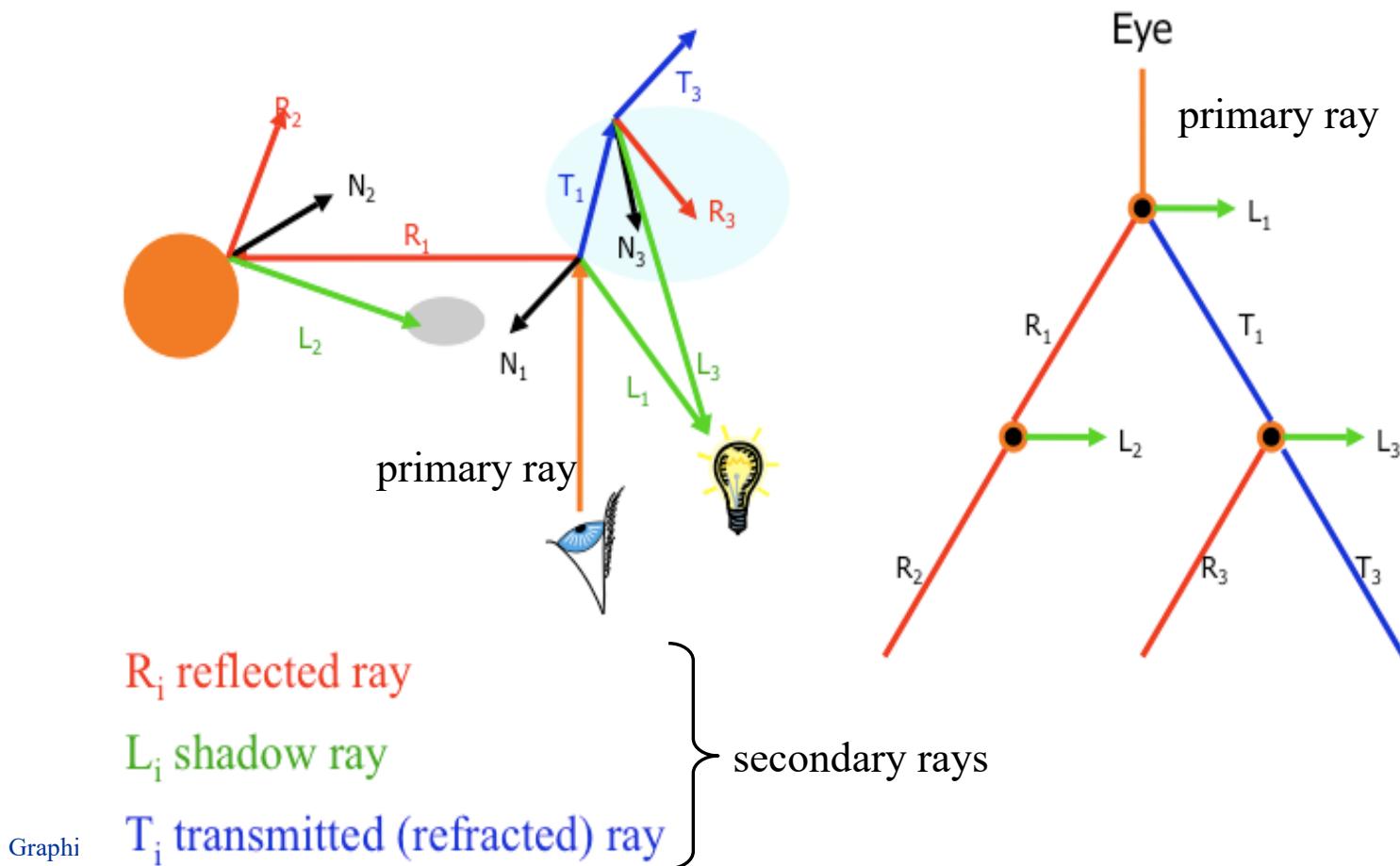
 col += k_refl * trace reflected ray

If transparent

 col += k_trans * trace transmitted ray



Ray tracing - Summary



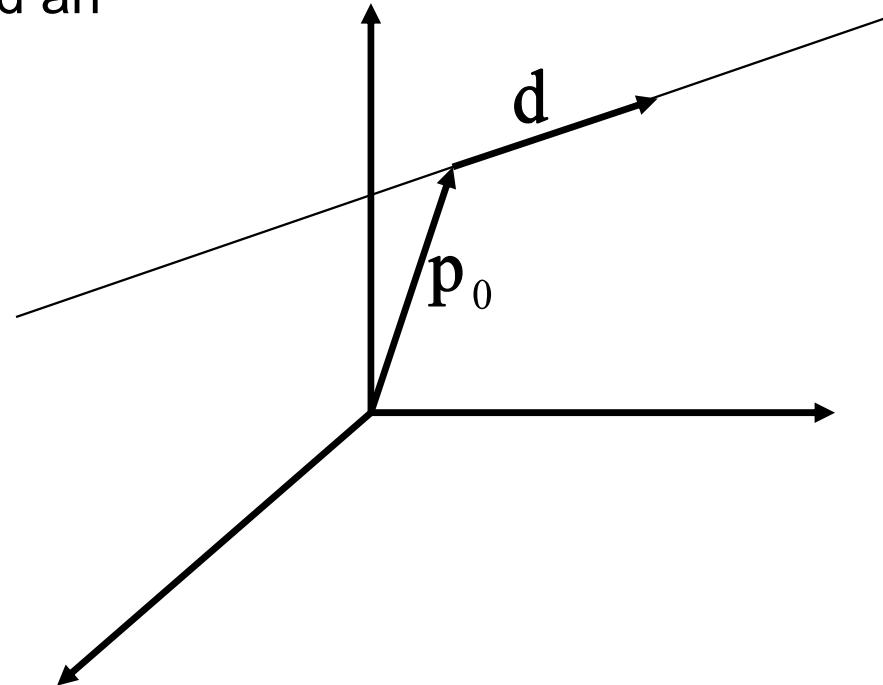
Intersection calculations

- For each ray we must calculate all possible intersections with each object inside the viewing volume
- For each ray we must find the nearest intersection point
- We can define our scene using
 - Solid models
 - sphere
 - cylinder
 - Surface models
 - plane
 - triangle
 - polygon

Rays

- Rays are parametric lines
- Rays can be defined as
 - origin p_0
 - direction d
- Equation of ray:

$$p(\mu) = p_0 + \mu d$$



Graphics Lecture 11: Slide 7

Ray tracing: Intersection calculations

- The coordinates of any point along each primary ray are given by:

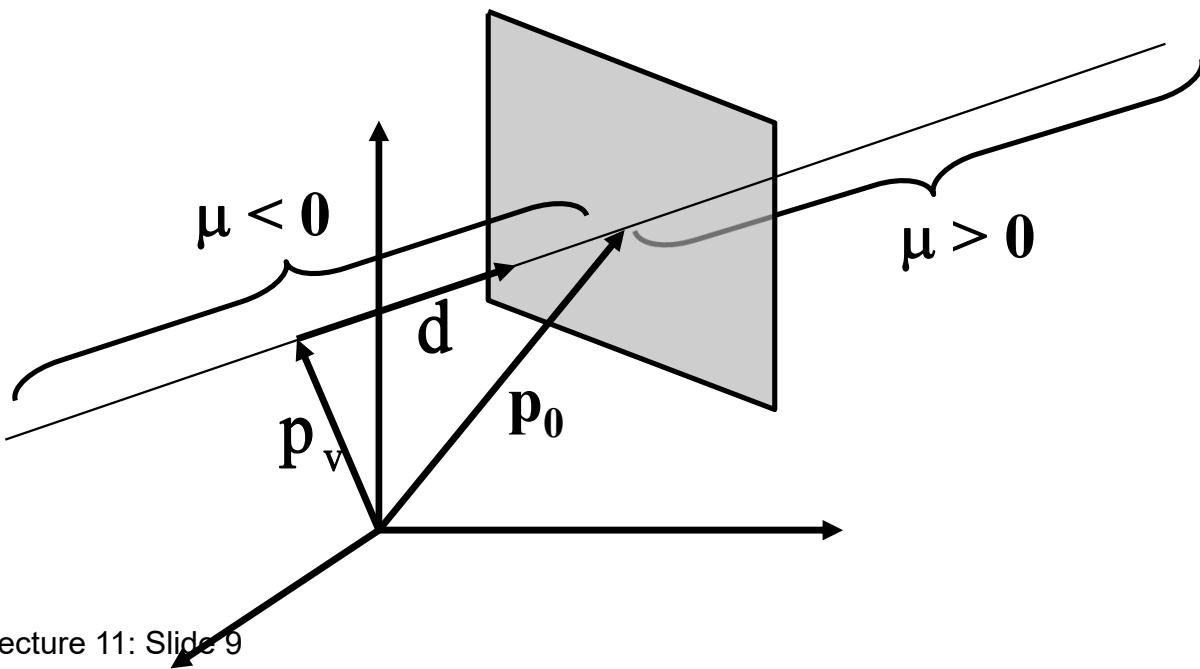
$$\mathbf{p} = \mathbf{p}_0 + \mu \mathbf{d}$$

- \mathbf{p}_0 is the current pixel on the viewing plane.
- \mathbf{d} is the direction vector and can be obtained from the position of the pixel on the viewing plane \mathbf{p}_0 and the viewpoint \mathbf{p}_v :

$$\mathbf{d} = \frac{\mathbf{p}_0 - \mathbf{p}_v}{|\mathbf{p}_0 - \mathbf{p}_v|}$$

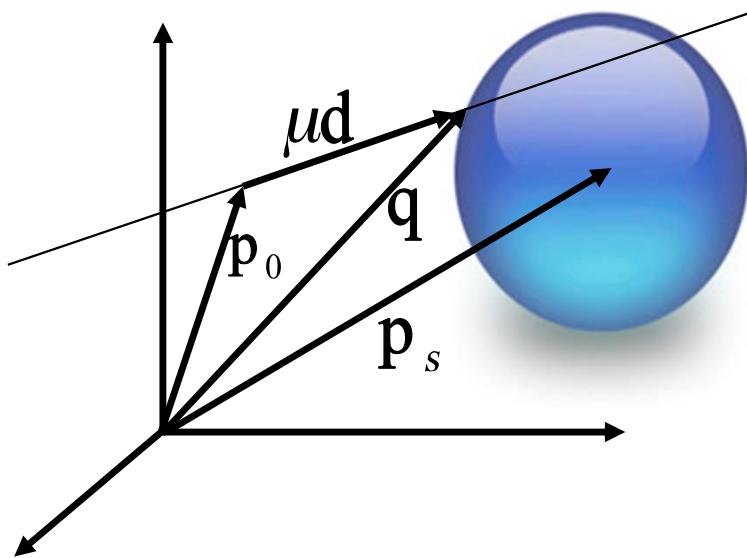
Ray tracing: Intersection calculations

- The viewing ray can be parameterized by μ :
 - $\mu > 0$ denotes the part of the ray behind the viewing plane
 - $\mu < 0$ denotes the part of the ray in front of the viewing plane
 - For any visible intersection point $\mu > 0$



Graphics Lecture 11: Slide 9

Intersection calculations: Spheres



- For any point on the surface of the sphere

$$|q - p_s|^2 - r^2 = 0$$

- where r is the radius of the sphere

Graphics Lecture 11: Slide 10

Intersection calculations: Spheres

- To test whether a ray intersects a surface we can substitute for q using the ray equation:

$$|\mathbf{p}_0 + \mu \mathbf{d} - \mathbf{p}_s|^2 - r^2 = 0$$

- Setting $\Delta\mathbf{p} = \mathbf{p}_0 - \mathbf{p}_s$ and expanding the dot product produces the following quadratic equation:

$$\mu^2 + 2\mu(\mathbf{d} \cdot \Delta\mathbf{p}) + |\Delta\mathbf{p}|^2 - r^2 = 0$$

Intersection calculations: Spheres

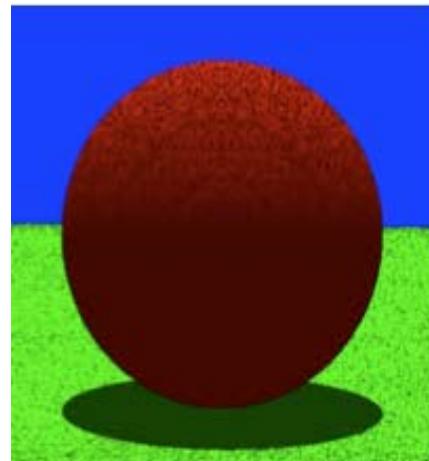
- The quadratic equation has the following solution:

$$\mu = -\mathbf{d} \cdot \Delta\mathbf{p} \pm \sqrt{(\mathbf{d} \cdot \Delta\mathbf{p})^2 - |\Delta\mathbf{p}|^2 + r^2}$$

- Solutions:
 - if the quadratic equation has no solution, the ray does not intersect the sphere
 - if the quadratic equation has two solutions ($\mu_1 < \mu_2$):
 - μ_1 corresponds to the point at which the rays enters the sphere
 - μ_2 corresponds to the point at which the rays leaves the sphere

Precision Problems

- In ray tracing, the origin of (secondary) rays is often on the surface of objects
 - Theoretically, $\mu = 0$ for these rays
 - Practically, calculation imprecision creeps in, and the origin of the new ray is slightly beneath the surface
- Result: the surface area is shadowing itself



Graphics Lecture 11: Slide 13

ε to the rescue ...

- Check if t is within some epsilon tolerance:
 - if $\text{abs}(\mu) < \varepsilon$
 - point is on the sphere
 - else
 - point is inside/outside
 - Choose the ε tolerance empirically
- Move the intersection point by epsilon along the surface normal so it is outside of the object
- Check if point is inside/outside surface by checking the sign of the implicit (sphere etc.) equation

Intersection calculations: Cylinders

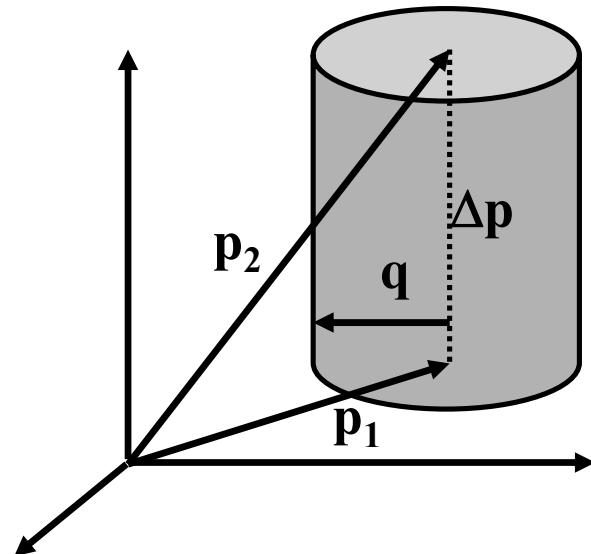
- A cylinder can be described by
 - a position vector \mathbf{p}_1 describing the first end point of the long axis of the cylinder
 - a position vector \mathbf{p}_2 describing the second end point of the long axis of the cylinder
 - a radius r
- The axis of the cylinder can be written as $\Delta\mathbf{p} = \mathbf{p}_1 - \mathbf{p}_2$ and can be parameterized by $0 \leq \alpha \leq 1$

Intersection calculations: Cylinders

- To calculate the intersection of the cylinder with the ray:

$$p_1 + \alpha \Delta p + q = p_0 + \mu d$$

- Since $q \cdot \Delta p = 0$ we can write



$$\alpha(\Delta p \cdot \Delta p) = p_0 \cdot \Delta p + \mu d \cdot \Delta p - p_1 \cdot \Delta p$$

Intersection calculations: Cylinders

- Solving for α yields:

$$\alpha = \frac{\mathbf{p}_0 \cdot \Delta\mathbf{p} + \mu\mathbf{d} \cdot \Delta\mathbf{p} - \mathbf{p}_1 \cdot \Delta\mathbf{p}}{\Delta\mathbf{p} \cdot \Delta\mathbf{p}}$$

- Substituting we obtain:

$$\mathbf{q} = \mathbf{p}_0 + \mu\mathbf{d} - \mathbf{p}_1 - \left(\frac{\mathbf{p}_0 \cdot \Delta\mathbf{p} + \mu\mathbf{d} \cdot \Delta\mathbf{p} - \mathbf{p}_1 \cdot \Delta\mathbf{p}}{\Delta\mathbf{p} \cdot \Delta\mathbf{p}} \right) \Delta\mathbf{p}$$

Intersection calculations: Cylinders

- Using the fact that $\mathbf{q} \cdot \mathbf{q} = r^2$ we can use the same approach as before to the quadratic equation for μ :

$$r^2 = \left(\mathbf{p}_0 + \mu \mathbf{d} - \mathbf{p}_1 - \left(\frac{\mathbf{p}_0 \cdot \Delta \mathbf{p} + \mu \mathbf{d} \cdot \Delta \mathbf{p} - \mathbf{p}_1 \cdot \Delta \mathbf{p}}{\Delta \mathbf{p} \cdot \Delta \mathbf{p}} \right) \Delta \mathbf{p} \right)^2$$

- If the quadratic equation has no solution:
no intersection
- If the quadratic equation has two solutions:
intersection

Intersection calculations: Cylinders

- Assuming that $\mu_1 \leq \mu_2$ we can determine two solutions:

$$\alpha_1 = \frac{p_0 \cdot \Delta p + \mu_1 d \cdot \Delta p - p_1 \cdot \Delta p}{\Delta p \cdot \Delta p}$$

$$\alpha_2 = \frac{p_0 \cdot \Delta p + \mu_2 d \cdot \Delta p - p_1 \cdot \Delta p}{\Delta p \cdot \Delta p}$$

- If the value of α_1 is between 0 and 1 the intersection is on the outside surface of the cylinder
- If the value of α_2 is between 0 and 1 the intersection is on the inside surface of the cylinder

Intersection calculations: Plane

- Objects are often described by geometric primitives such as
 - triangles
 - planar quads
 - planar polygons
- To test intersections of the ray with these primitives we must test whether the ray will intersect the plane defined by the primitive

Intersection calculations: Plane

- The intersection of a ray with a plane is given by

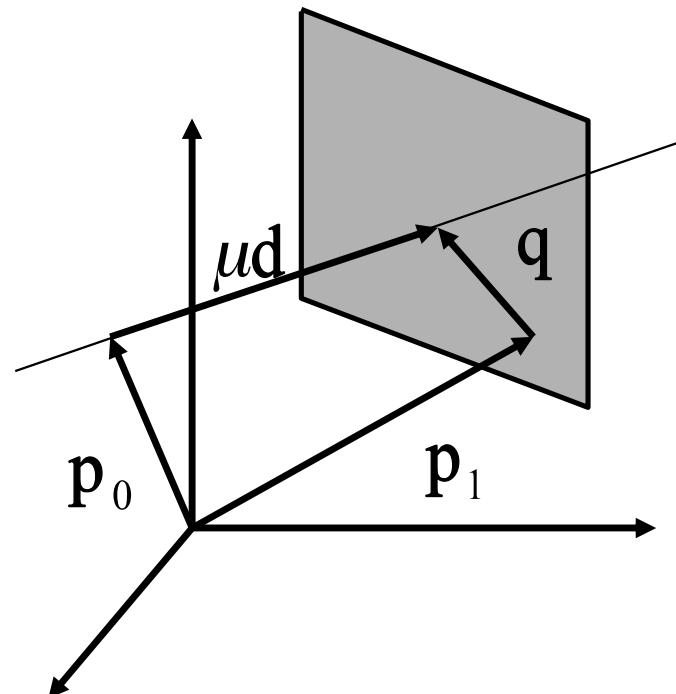
$$\mathbf{p}_1 + \mathbf{q} = \mathbf{p}_0 + \mu \mathbf{d}$$

- where \mathbf{p}_1 is a point in the plane.
Subtracting \mathbf{p}_1 and multiplying with the normal of the plane \mathbf{n} yields:

$$\mathbf{q} \cdot \mathbf{n} = 0 = (\mathbf{p}_0 - \mathbf{p}_1) \cdot \mathbf{n} + \mu \mathbf{d} \cdot \mathbf{n}$$

- Solving for μ yields:

$$\mu = -\frac{(\mathbf{p}_0 - \mathbf{p}_1) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$

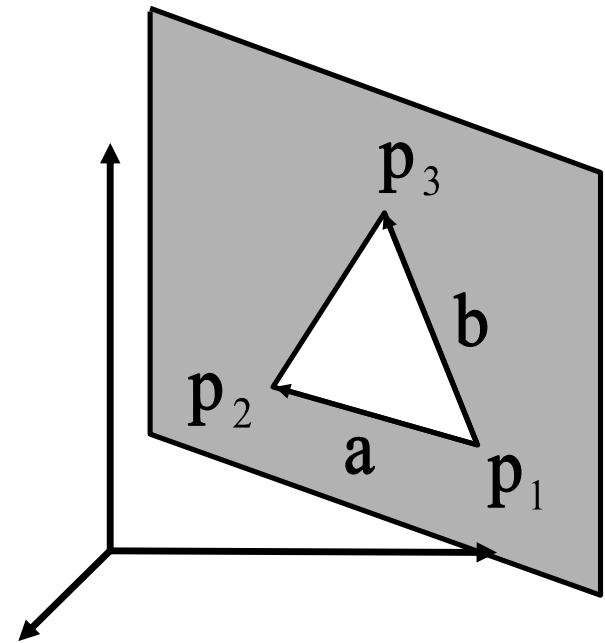


Graphics Lecture 11: Slide 21

Intersection calculations: Triangles

- To calculate intersections:
 - test whether triangle is front facing
 - test whether plane of triangle intersects ray
 - test whether intersection point is inside triangle
- If the triangle is front facing:

$$d \cdot n < 0$$



Graphics Lecture 11: Slide 22

Intersection calculations: Triangles

- To test whether plane of triangle intersects ray

– calculate equation of the plane using

$$\mathbf{p}_2 - \mathbf{p}_1 = \mathbf{a}$$

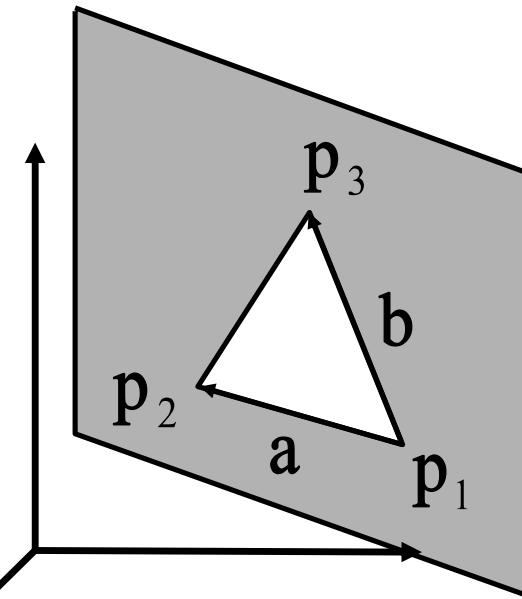
$$\mathbf{p}_3 - \mathbf{p}_1 = \mathbf{b}$$

– calculate intersections with plane as before

$$\mathbf{n} = \mathbf{a} \times \mathbf{b}$$

- To test whether intersection point is inside triangle:

$$\mathbf{q} = \alpha \mathbf{a} + \beta \mathbf{b}$$



Intersection calculations: Triangles

- A point is inside the triangle if

$$0 \leq \alpha \leq 1$$

$$0 \leq \beta \leq 1$$

$$\alpha + \beta \leq 1$$

- Calculate α and β by taking the dot product with a and b :

$$\alpha = \frac{(\mathbf{b} \cdot \mathbf{b})(\mathbf{q} \cdot \mathbf{a}) - (\mathbf{a} \cdot \mathbf{b})(\mathbf{q} \cdot \mathbf{b})}{(\mathbf{a} \cdot \mathbf{a})(\mathbf{b} \cdot \mathbf{b}) - (\mathbf{a} \cdot \mathbf{b})^2}$$

$$\beta = \frac{\mathbf{q} \cdot \mathbf{b} - \alpha(\mathbf{a} \cdot \mathbf{b})}{\mathbf{b} \cdot \mathbf{b}}$$

Graphics Lecture 11: Slide 24

Intersection calculations: Triangles

- algorithm by **Möller and Trumbore**:
- translate the origin of the ray and then change the base of that vector which yields parameter vector (t, u, v)
- t is the distance to the plane in which the triangle lies
- (u, v) are barycentric coordinates inside the triangle
- plane equation need not be computed on the fly nor be stored

Graphics Lecture 11: Slide 25

Intersection calculations: Triangles

```
1 bool triangle_intersection( vec3 v1,
2                             vec3 v2,
3                             vec3 v3, // Triangle vertices
4                             vec3 origin, //Ray origin
5                             vec3 ray_dir, //Ray direction
6                             float* out )
7
8 //Find vectors for two edges sharing v1
9 vec3 edge1 = v2-v1;
10 vec3 edge2 = v3-v1;
11
12 //Begin calculating determinant - also used to calculate u parameter
13 vec3 p = cross(ray_dir, edge2)
14
15 //if determinant is near zero, ray lies in plane of triangle or ray
16 //is parallel to plane of triangle
17 float det = dot(edge1, p);
18
19 if(det > -EPSILON && det < EPSILON)
20 return false;
21
22 float inv_det = 1.f / det;
23
24 //calculate distance from v1 to ray origin
25 vec3 t = origin-v1;
```

Intersection calculations: Triangles

```
27 //Calculate u parameter and test bound
28 float u = dot(t, p) * inv_det;
29
30 //The intersection lies outside of the triangle
31 if(u < 0.f || u > 1.f)
32 return false;
33
34 //Prepare to test v parameter
35 vec3 q = cross(t, edge1);
36
37 float v = dot(ray_dir, q) * inv_det;
38 //The intersection lies outside of the triangle
39 if(v < 0.f || u + v > 1.f)
40 return false;
41
42 float mu = dot(edge2, q) * inv_det;
43
44 if(t > EPSILON) { //ray intersection
45     *out = mu;
46     return true;
}
```

Ray tracing: Pros and cons

- Pros:
 - Easy to implement
 - Extends well to global illumination
 - shadows
 - reflections / refractions
 - multiple light bounces
 - atmospheric effects
- Cons:
 - Speed! (seconds per frame, not frames per second)

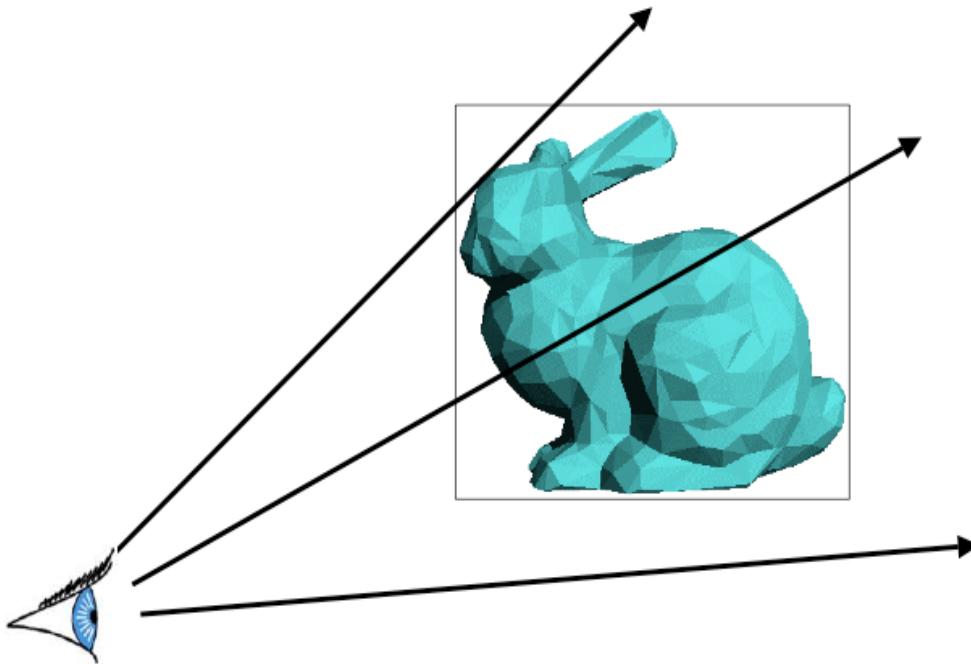
Speedup Techniques

- Why is ray tracing slow? How to improve?
 - Too many objects, too many rays
 - Reduce ray-object intersection tests
 - Many techniques!

Graphics Lecture 11: Slide 29

Acceleration of Ray Casting

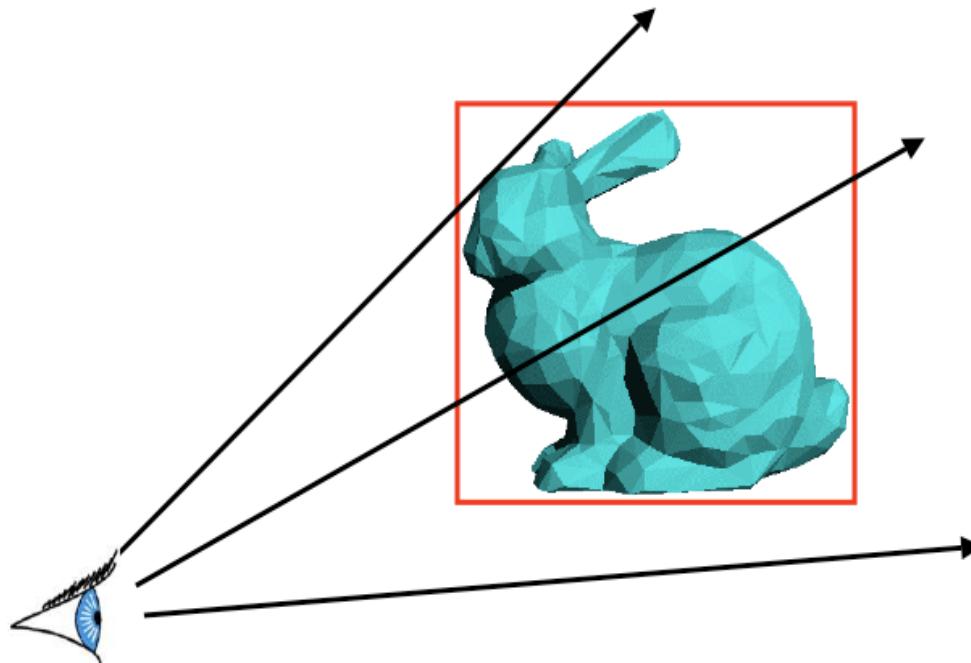
- Goal: Reduce the number of ray/primitive intersections



Graphics Lecture 11: Slide 30

Conservative Bounding Region

- First check for an intersection with a conservative bounding region
- Early reject



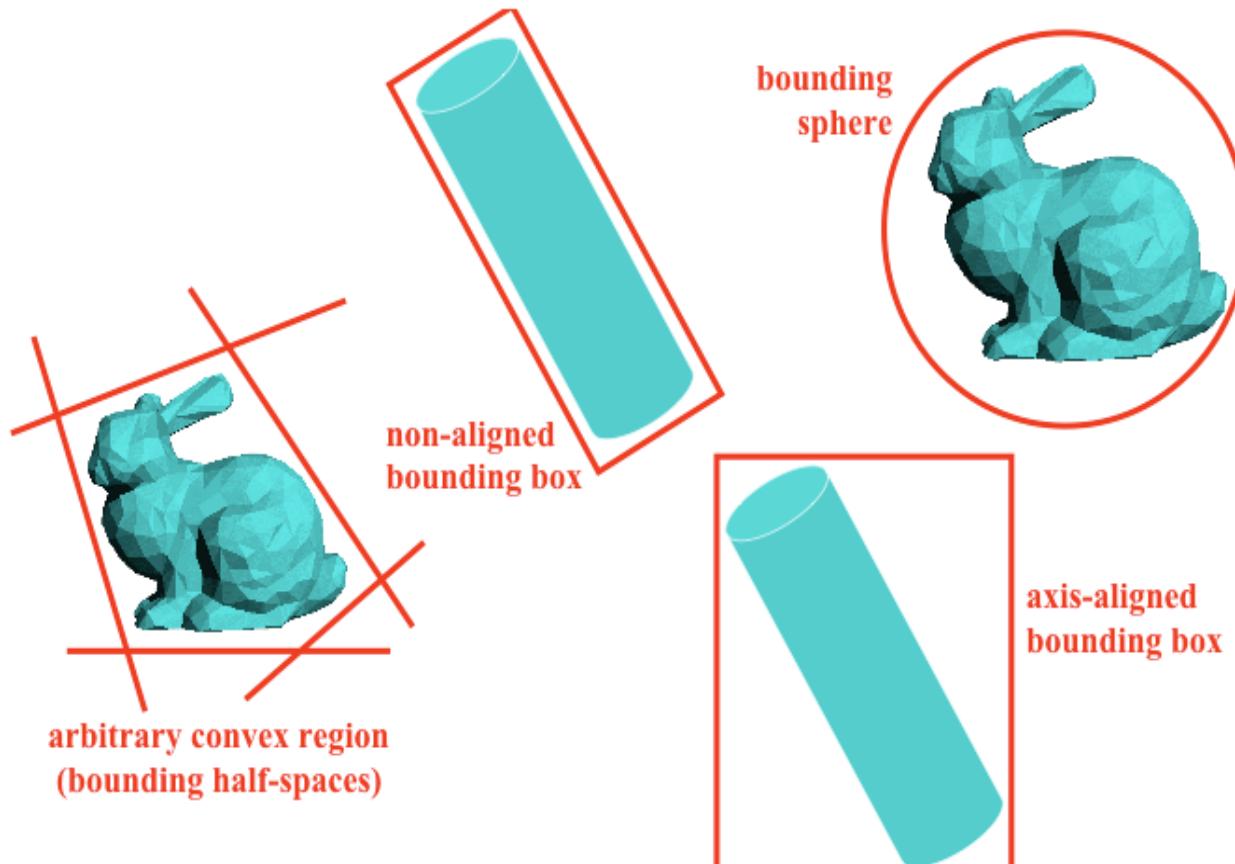
Graphics Lecture 11: Slide 31

Bounding Regions

- What makes a good bounding region?

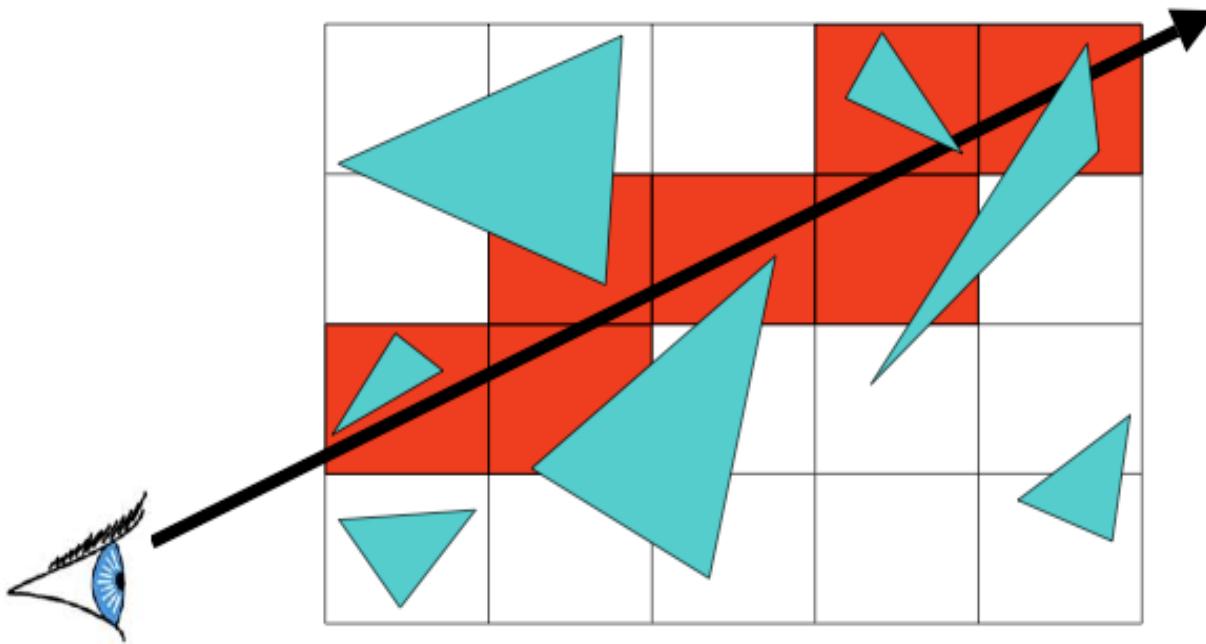
Graphics Lecture 11: Slide 32

Conservative Bounding Regions



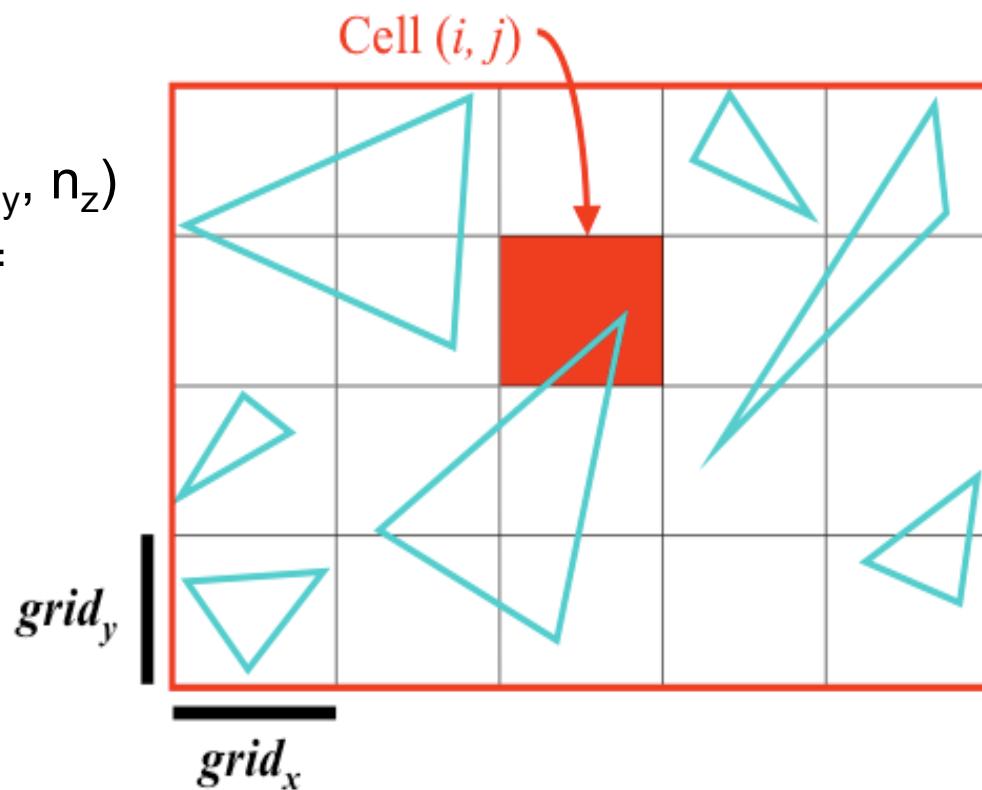
Graphics Lecture 11: Slide 33

Regular Grid



Create Grid

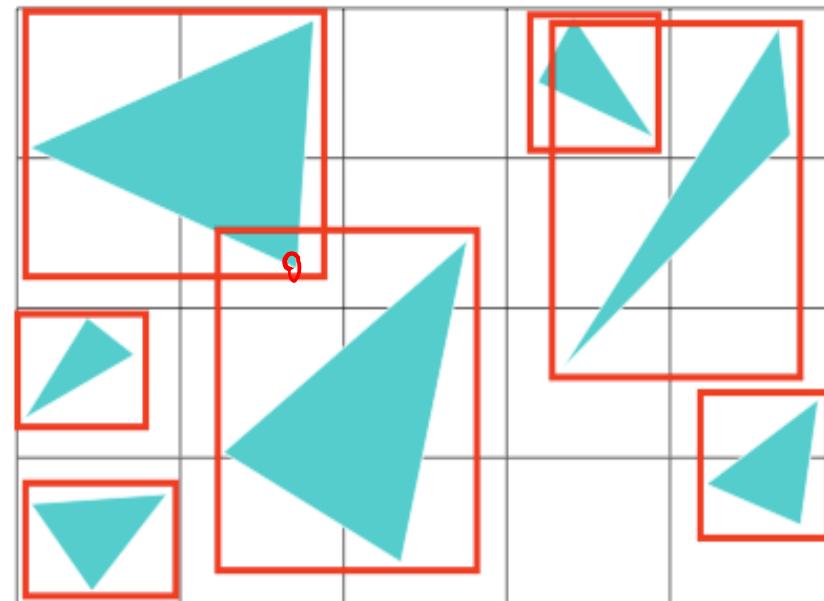
- Find bounding box of scene
- Choose grid resolution (n_x, n_y, n_z)
- $grid_x$ need not = $grid_y$



Graphics Lecture 11: Slide 35

Insert Primitives into Grid

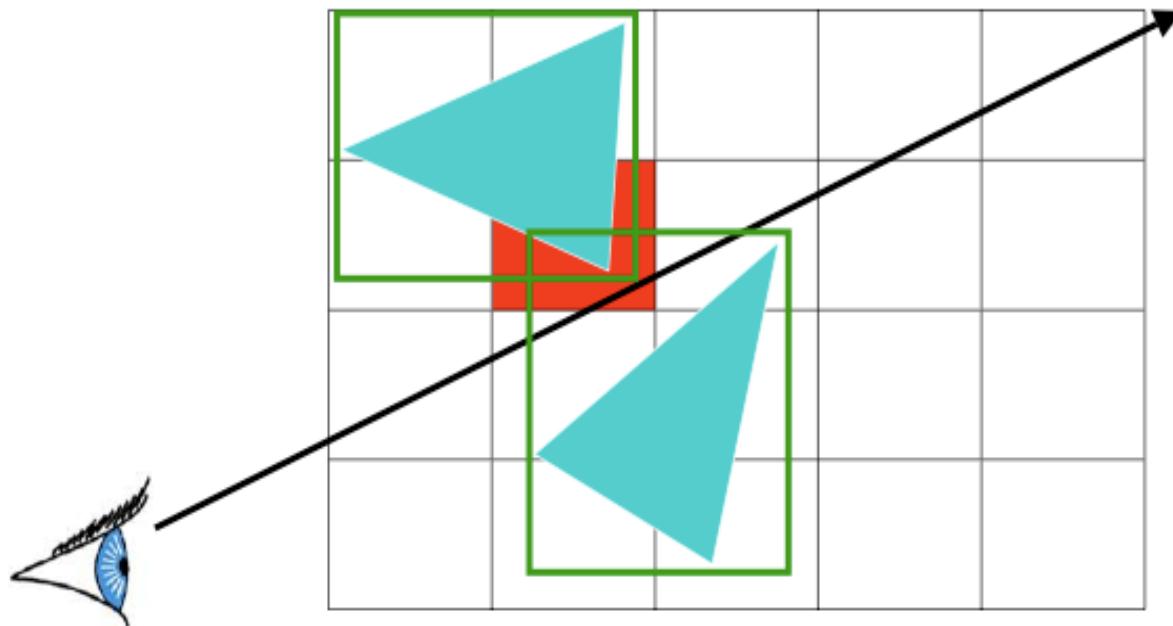
- Primitives that overlap multiple cells?
- Insert into multiple cells (use pointers)



Graphics Lecture 11: Slide 36

For Each Cell Along a Ray

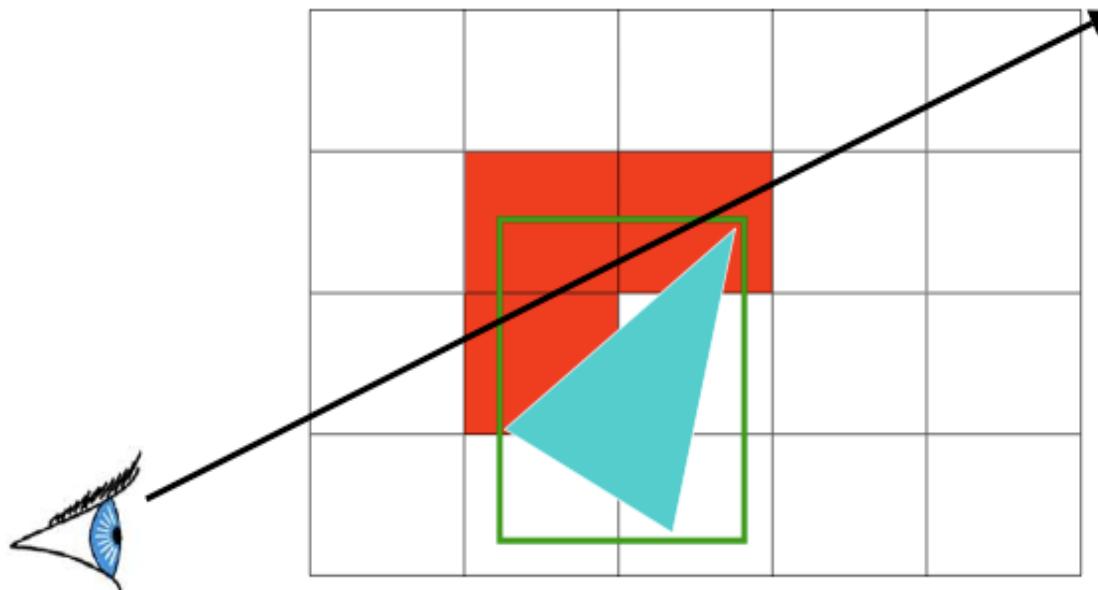
- Does the cell contain an intersection?
 - Yes: return closest intersection
 - No: continue



Graphics Lecture 11: Slide 37

Preventing Repeated Computation

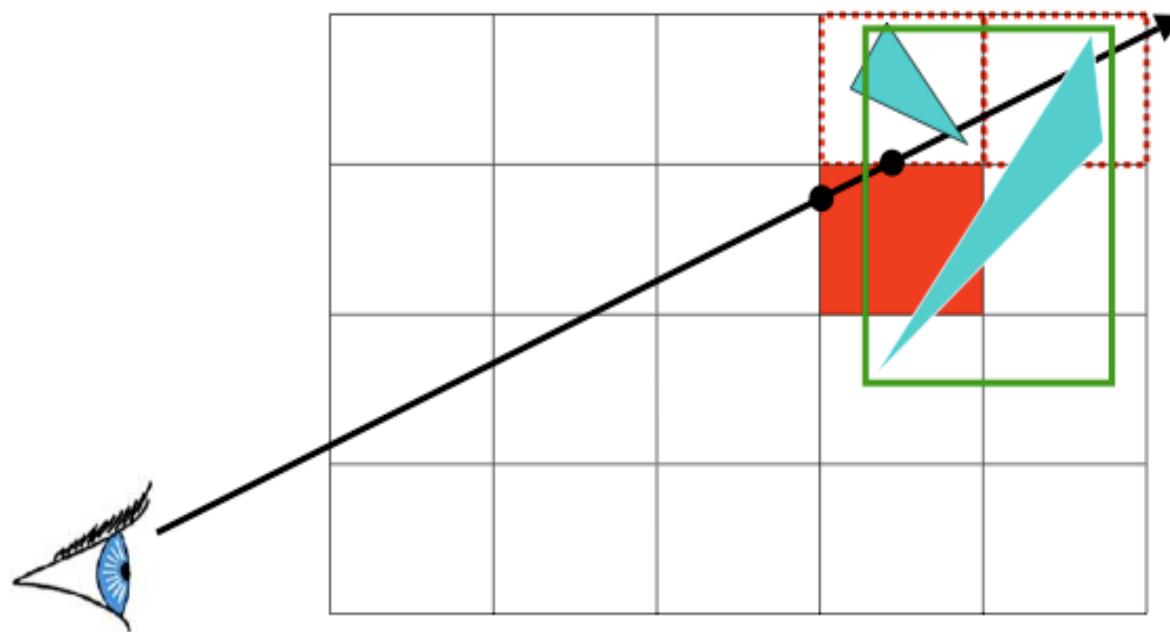
- Perform the computation once, "mark" the object
- Don't re-intersect marked objects



Graphics Lecture 11: Slide 38

Don't Return Distant Intersections

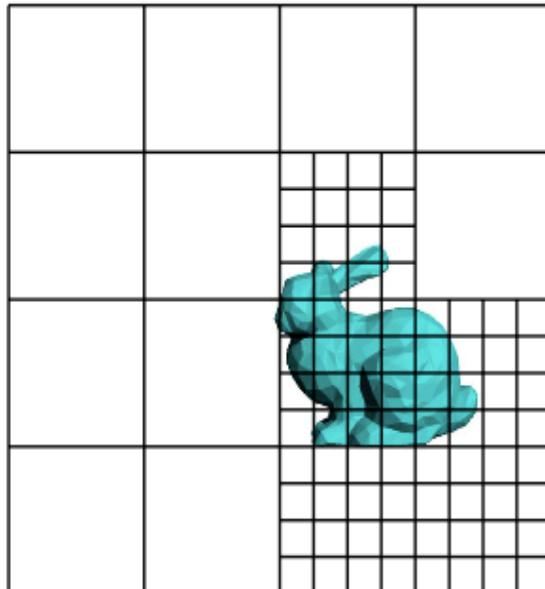
- If intersection t is not within the cell range, continue (there may be something closer)



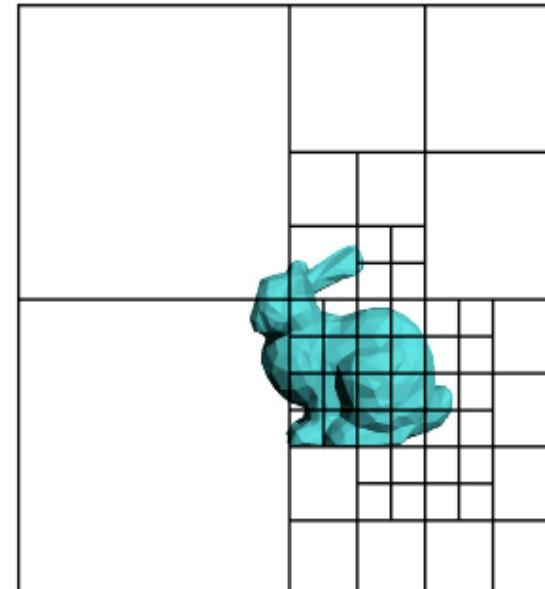
Graphics Lecture 11: Slide 39

Adaptive Grids

- Subdivide until each cell contains no more than n elements, or maximum depth d is reached



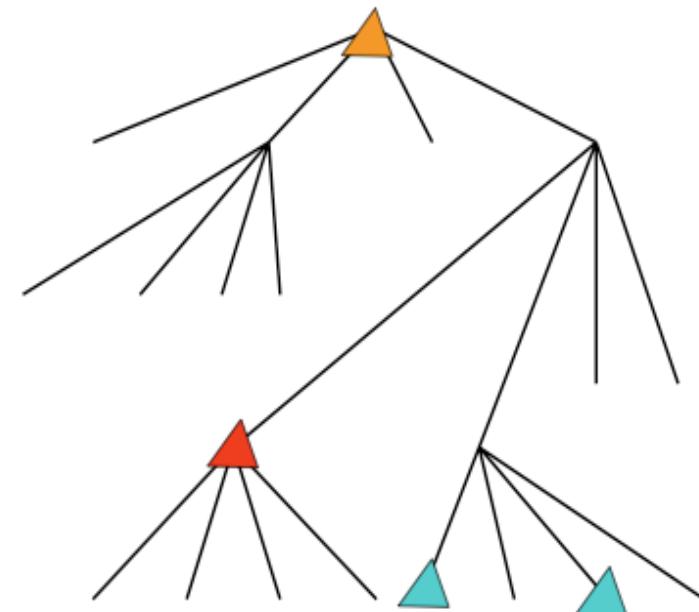
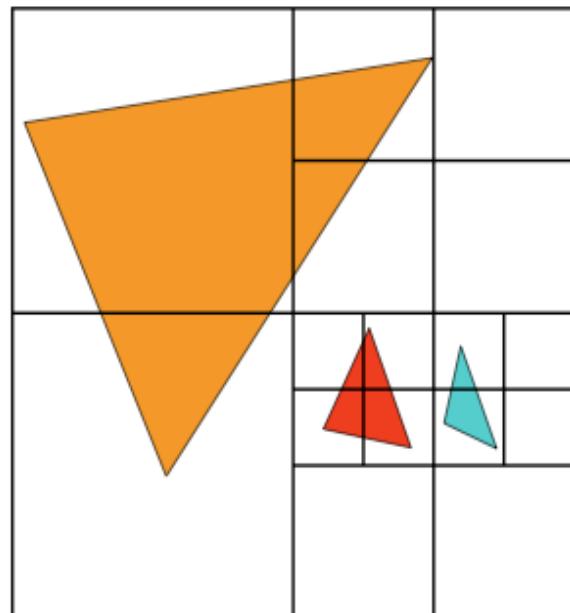
Nested Grids



Octree/(Quadtree)

Primitives in an Adaptive Grid

- Can live at intermediate levels, or be pushed to lowest level of grid

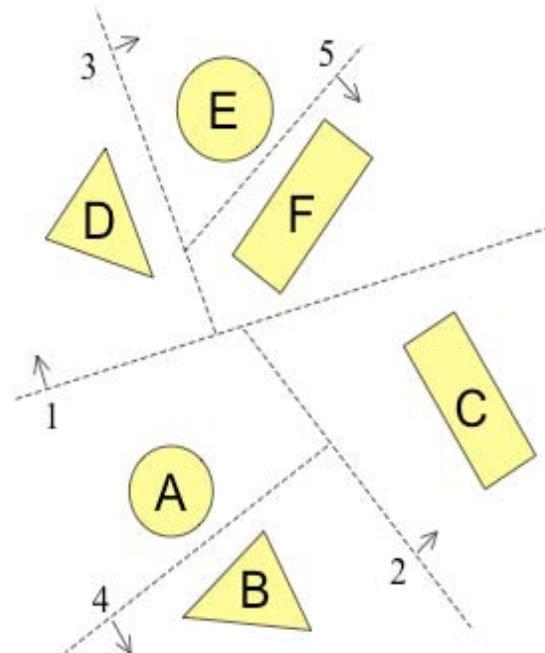
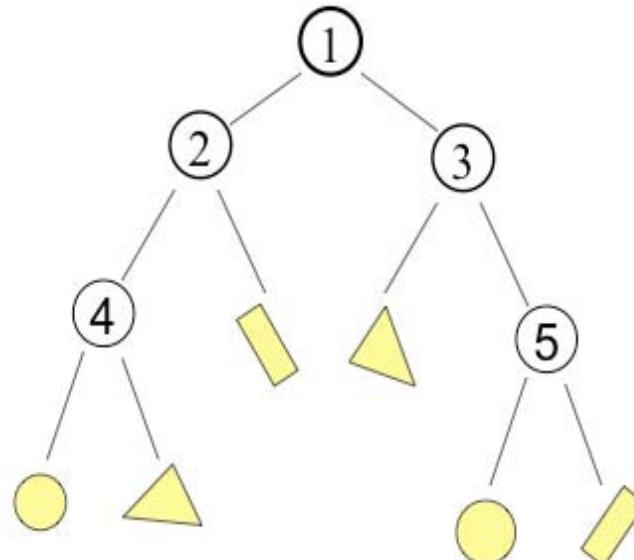


Octree/(Quadtree)

Graphics Lecture 11: Slide 41

Binary Space Partition (BSP) Tree

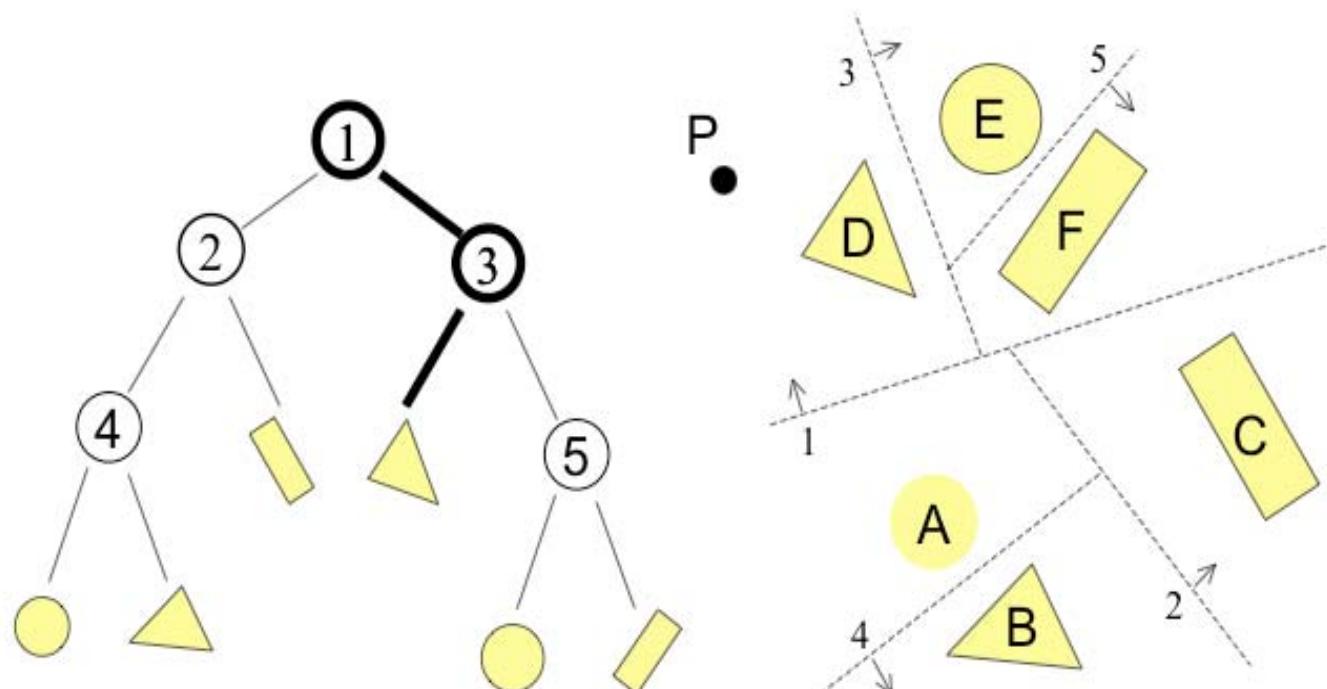
- Recursively partition space by planes
- Every cell is a convex polyhedron



Graphics Lecture 11: Slide 42

Binary Space Partition (BSP) Tree

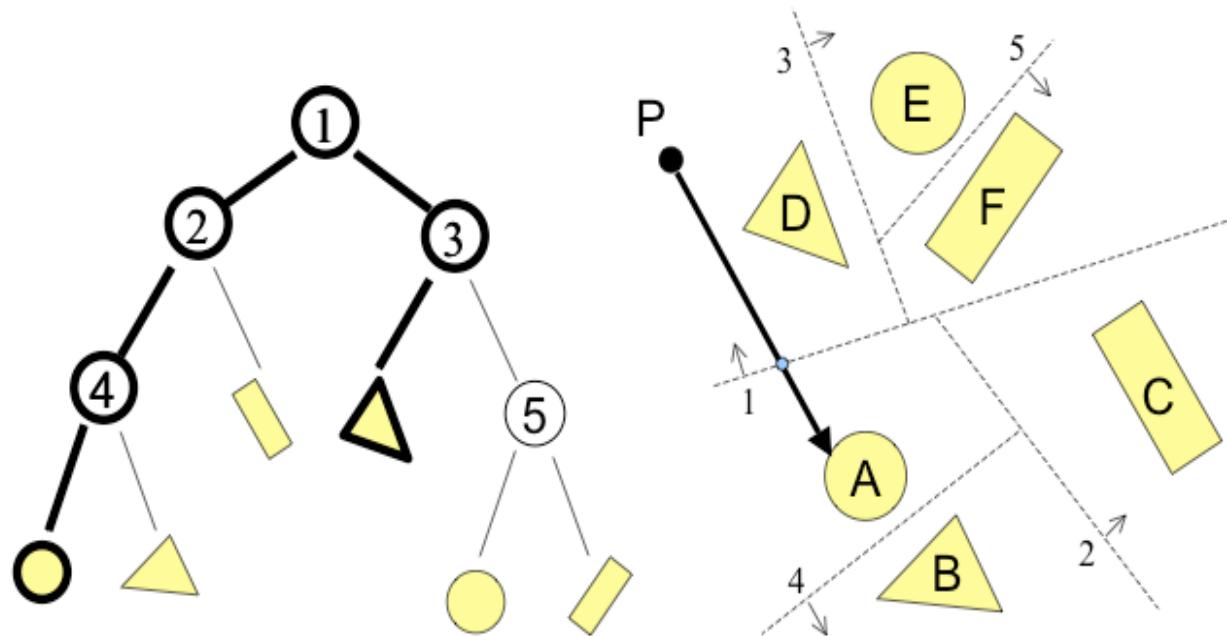
- Simple recursive algorithms
- Example: point finding



Graphics Lecture 11: Slide 43

Binary Space Partition (BSP) Tree

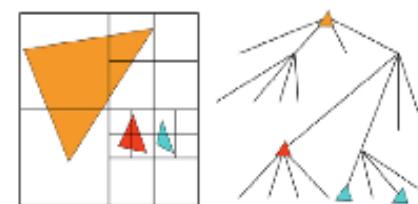
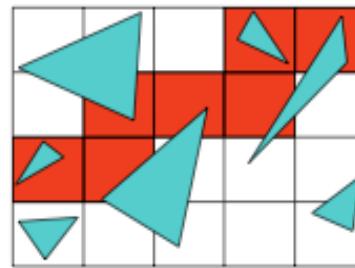
- Trace rays by recursion on tree
 - BSP construction enables simple front-to-back traversal



Graphics Lecture 11: Slide 44

Grid Discussion

- Regular
 - + easy to construct
 - + easy to traverse
 - may be only sparsely filled
 - geometry may still be clumped
- Adaptive
 - + grid complexity matches geometric density
 - more expensive to traverse (especially BSP tree)



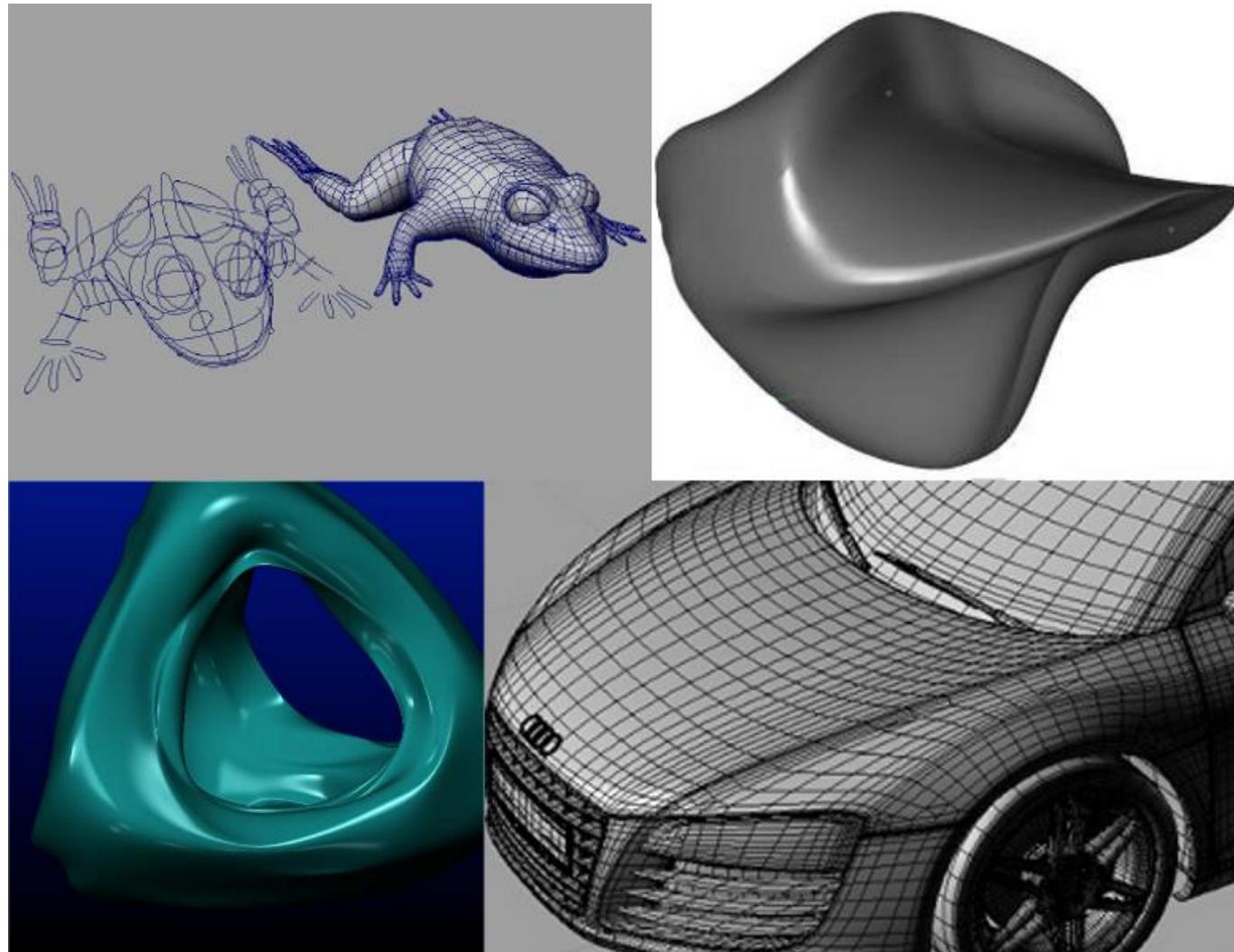
Example

- <https://www.youtube.com/watch?v=aKqxonOrI4Q>
- <https://www.youtube.com/watch?v=qlyzo9ll9Vw>
- <https://www.youtube.com/watch?v=AV279wThmVU>
-

Interactive Computer Graphics: *Lecture 12*

Introduction to Spline Curves

Splines



Graphics Lecture 11: Slide 2

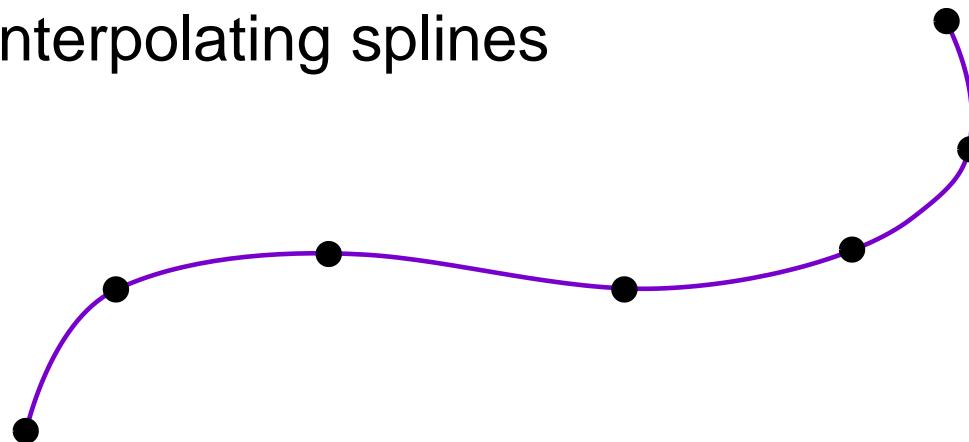
Splines

- The word spline comes from the ship building trade where planks were originally shaped by bending them round pegs fixed in the ground.
- Originally it was the pegs that were referred to as splines.



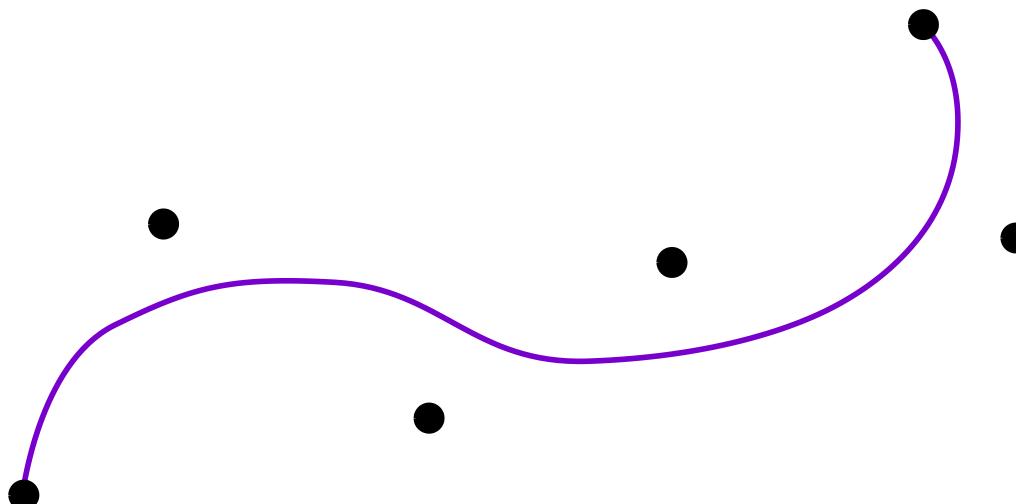
Interpolating Splines

- Modern splines are smooth curves defined from a small set of points often called *knots* or *control points*.
- In one main class of splines, the curve must pass through each point of the set.
- These are called interpolating splines



Approximating Splines

- In other cases the curves do not pass through the points.
- The points act as control points which the user can move to adjust the shape of the curve interactively



Non-Parametric Spline

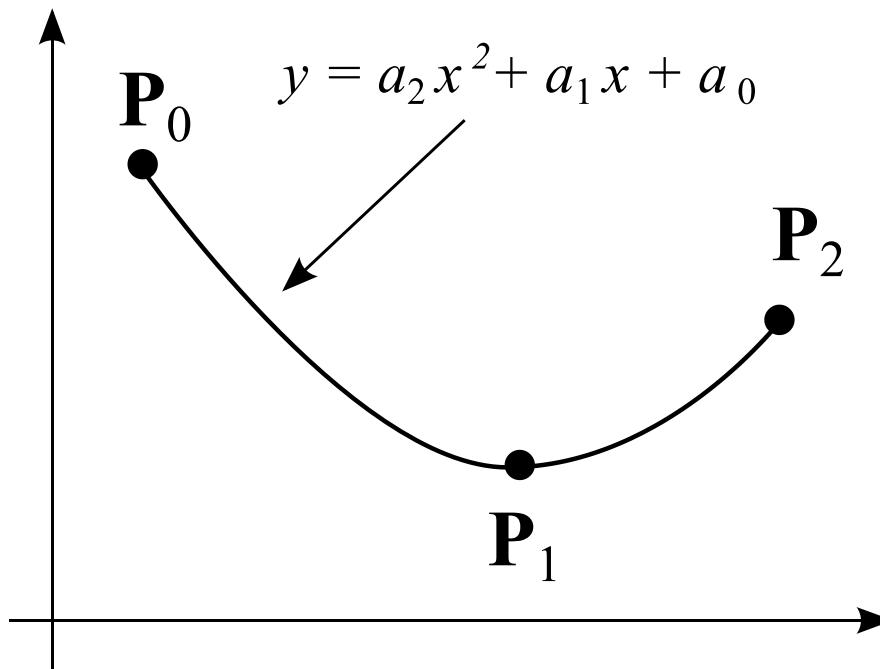
- The simplest splines are just equations in x and y (for two dimensions)
- The most common is the polynomial spline:

$$y = a_2x^2 + a_1x + a_0$$

- Given three points we can calculate a_2 , a_1 and a_0

A Non-Parametric (Parabolic) Spline

- Example of a degree 2 (parabolic) non-parametric spline:



- There is no control using non parametric splines. Only one curve (a parabola) fits the data.

Parametric Splines

- If we write our spline in a vector form we get:

$$\mathbf{P} = \mathbf{a}_2\mu^2 + \mathbf{a}_1\mu + \mathbf{a}_0$$

which has a parameter μ

- By convention, as μ ranges from 0 to 1 the point \mathbf{P} traces out a curve.

Calculating simple parametric splines

We can now solve for the vector constants \mathbf{a}_0 , \mathbf{a}_1 and \mathbf{a}_2 as follows:

- Suppose we want the curve to start at point \mathbf{P}_0

$$\mathbf{P}_0 = \mathbf{a}_2\mu^2 + \mathbf{a}_1\mu + \mathbf{a}_0$$

- We have $\mu = 0$ at the start so

$$\mathbf{P}_0 = \mathbf{a}_0$$

Calculating simple parametric splines

- Suppose we want the spline to end at \mathbf{P}_2
- We have that at the end $\mu = 1$
- Thus

$$\begin{aligned}\mathbf{P}_2 &= \mathbf{a}_2\mu^2 + \mathbf{a}_1\mu + \mathbf{a}_0 \\ &= \mathbf{a}_2 + \mathbf{a}_1 + \mathbf{a}_0 \\ \Rightarrow \mathbf{P}_2 &= \mathbf{a}_2 + \mathbf{a}_1 + \mathbf{P}_0\end{aligned}$$

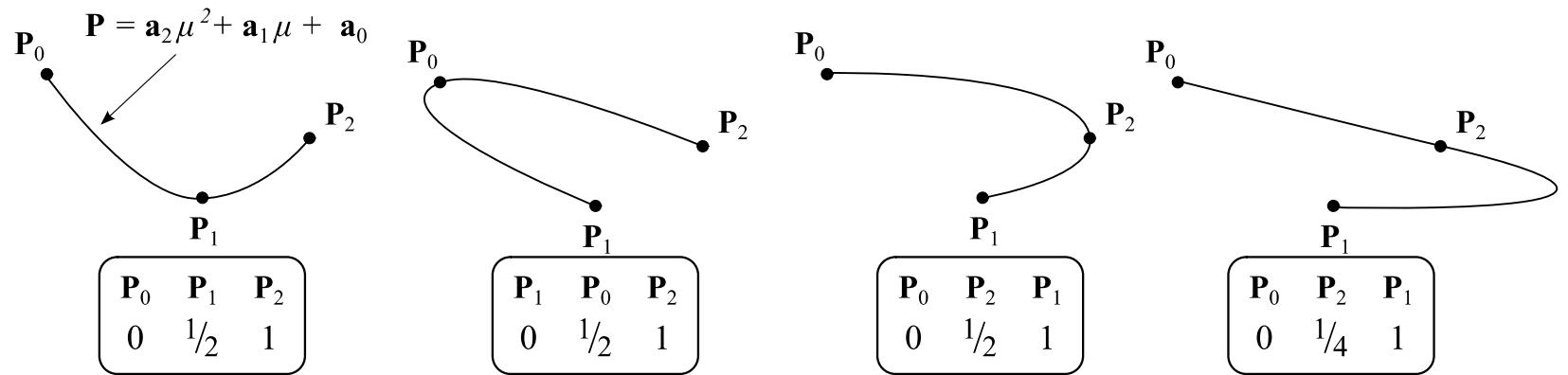
Calculating simple parametric splines

- And in the middle (say $\mu = 1/2$) we want it to pass through \mathbf{P}_1

$$\begin{aligned}\mathbf{P}_1 &= \mathbf{a}_2\mu^2 + \mathbf{a}_1\mu + \mathbf{a}_0 \\ \Rightarrow \mathbf{P}_1 &= \frac{1}{4}\mathbf{a}_2 + \frac{1}{2}\mathbf{a}_1 + \mathbf{P}_0\end{aligned}$$

- We have enough equations to solve for \mathbf{a}_1 and \mathbf{a}_2 .
- Notice that the method is the same whether we are working in 2 or 3 dimensions, we just have to solve separately for each of the ordinates in the vectors \mathbf{a}_1 and \mathbf{a}_2 .

Possibilities using parametric splines

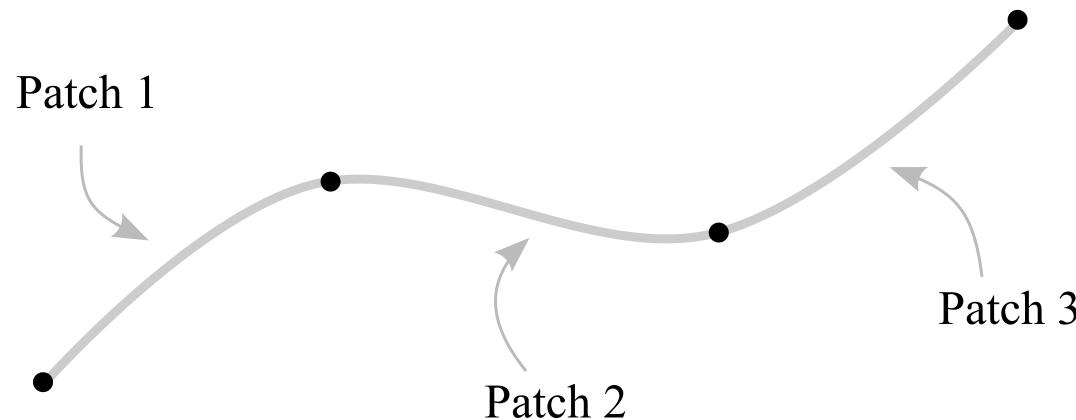


Higher order parametric splines

- Parametric polynomial splines must have an order to match the number of knots.
 - 3 knots - quadratic polynomial
 - 4 knots - cubic polynomial
 - etc.
- Higher order polynomials are undesirable since they tend to oscillate

Spline Patches

- To get round the problem, we can piece together a number of patches, each patch being a parametric spline.



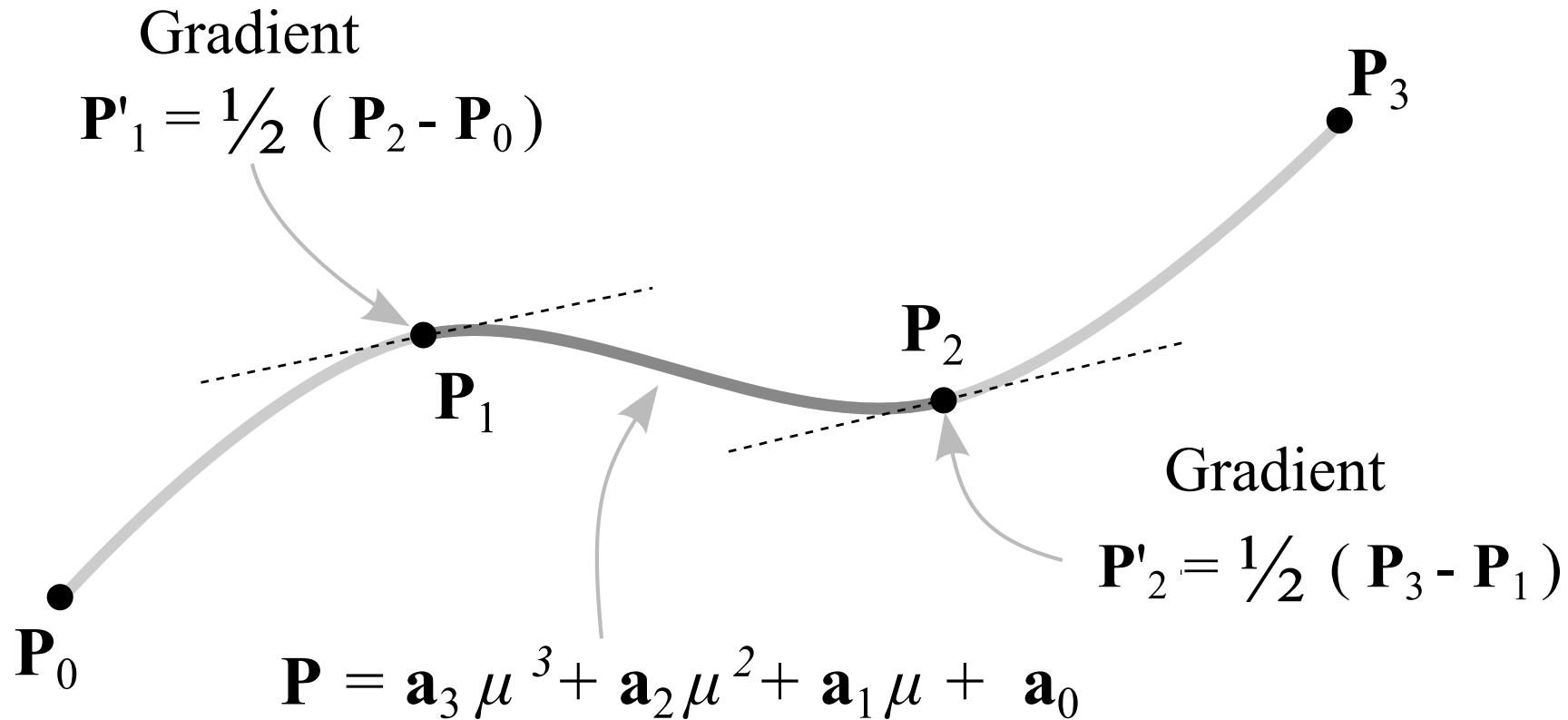
Cubic Spline Patches

- The simplest, and most effective way to calculate parametric spline patches is to use a cubic polynomial.

$$\mathbf{P} = \mathbf{a}_3\mu^3 + \mathbf{a}_2\mu^2 + \mathbf{a}_1\mu + \mathbf{a}_0$$

- This allows us to join the patches together smoothly

Choosing the gradients



Indices here are $\{0, 1, 2, 3\}$ but can be any successive set of four numbers taken from the available control points

Calculating a Cubic Spline Patch

- Each patch has the form: $\mathbf{P} = \mathbf{a}_3\mu^3 + \mathbf{a}_2\mu^2 + \mathbf{a}_1\mu + \mathbf{a}_0$
- For the patch which joins points \mathbf{P}_i and \mathbf{P}_{i+1} , we have

$$\mu = 0 \text{ at } \mathbf{P}_i$$

$$\mu = 1 \text{ at } \mathbf{P}_{i+1}$$

- Substituting these values we get

$$\mathbf{P}_i = \mathbf{a}_0$$

$$\mathbf{P}_{i+1} = \mathbf{a}_3 + \mathbf{a}_2 + \mathbf{a}_1 + \mathbf{a}_0$$

Calculating a Cubic Spline Patch

- Differentiating $P = a_3\mu^3 + a_2\mu^2 + a_1\mu + a_0$ we get

$$P' = 3a_3\mu^2 + 2a_2\mu + a_1$$

- Substituting for $\mu=0$ at P_i and $\mu=1$ at P_{i+1} we get

$$P'_i = a_1$$

$$P'_{i+1} = 3a_3 + 2a_2 + a_1$$

Calculating a Cubic Spline Patch

- Putting these four equations into matrix form we get:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} \mathbf{a}_0 \\ \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \end{pmatrix} = \begin{pmatrix} \mathbf{P}_i \\ \mathbf{P}'_i \\ \mathbf{P}_{i+1} \\ \mathbf{P}'_{i+1} \end{pmatrix}$$

- The initial matrix is always the same whether the points \mathbf{P} are in 2-D or in 3-D

Calculating a Cubic Spline Patch

- Finally, inverting the matrix gives us the values of $\mathbf{a}_0, \dots, \mathbf{a}_3$ that we want

$$\begin{pmatrix} \mathbf{a}_0 \\ \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3 & -2 & 3 & -1 \\ 2 & 1 & -2 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{P}_i \\ \mathbf{P}'_i \\ \mathbf{P}_{i+1} \\ \mathbf{P}'_{i+1} \end{pmatrix}$$

- Notice that the matrix is the same
 - for every patch
 - whether the data are 2-D, 3-D, ...

Parametric and Geometric Continuity

- We want to create smooth and realistic shapes.
 - What exactly do we mean by "smooth"?
 - How precisely do we determine if a given curve or surface is smooth?
- Recall that a parametric curve is defined as:
$$\mathbf{P}(\mu) = \begin{pmatrix} x(\mu) \\ y(\mu) \end{pmatrix}$$
- Generally a function is smooth if its derivatives are well-defined up to some order. There are actually two definitions for curves and surfaces, depending on whether the curve or surface is viewed as a function or purely a shape.

Parametric Continuity

- For parametric continuity, we view the curve or surface as a function rather than a shape.
 - A junction between two curves is said to be C^0 continuous if the (x, y) values of the two curves agree.
 - A junction between two curves is said to be C^1 continuous if the (x, y) values of the two curves agree, and all their first derivatives ($dx/ds, dy/ds$) agree at their junction.
 - A junction between two curves is said to be C^2 continuous if the (x, y) values of the two curves agree, and their first and second parametric derivatives all agree at their junction.

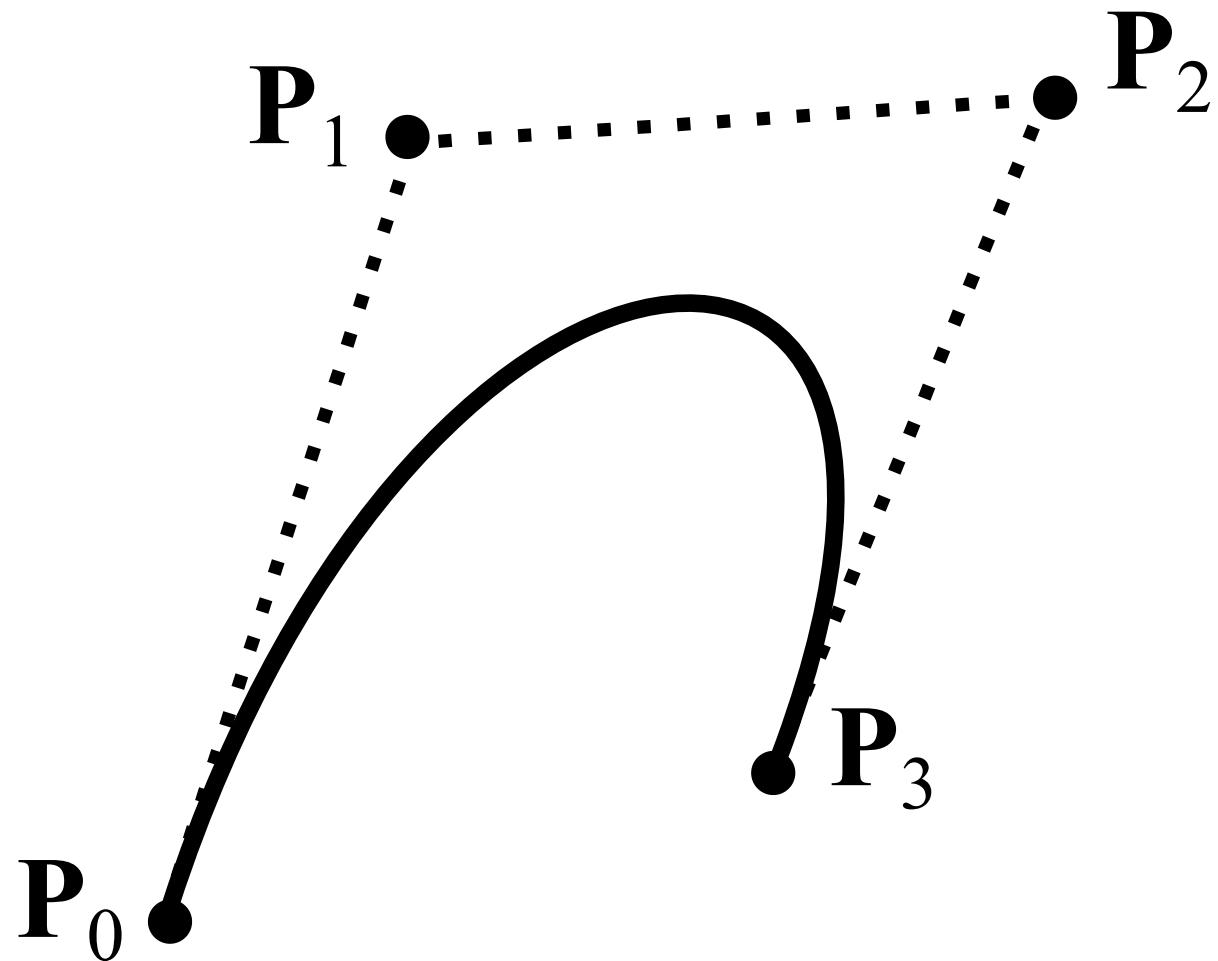
Geometric Continuity

- Geometric continuity can be defined using only the shape of the curve (parametrization does not affect the outcome):
 - A junction between two curves is said to be G^0 continuous if the (x, y) values of the two curves agree. Same as C^0 continuity.
 - A junction between two curves is said to be G^1 continuous if the (x, y) values of the two curves agree, and all their first derivatives ($dx/ds, dy/ds$) are proportional (the tangent vectors are parallel) at their junction.
 - Higher order geometric continuity is a bit tricky to define.

Bezier Curves

- Bezier curves were developed as a method for CAD design. They give very predictable results for small sets of knots, and so are useful as spline patches.
- The main characteristics of Bezier curves are
 - They interpolate the end points
 - The slope at an end is the same as the line joining the end point to its neighbour

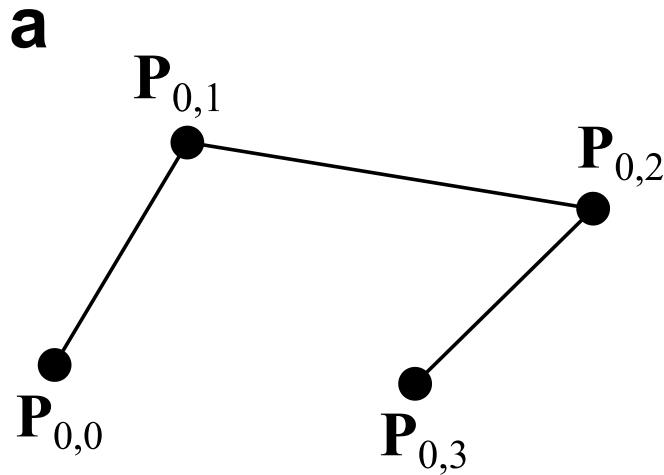
A typical Bezier Curve



Casteljau's Algorithm

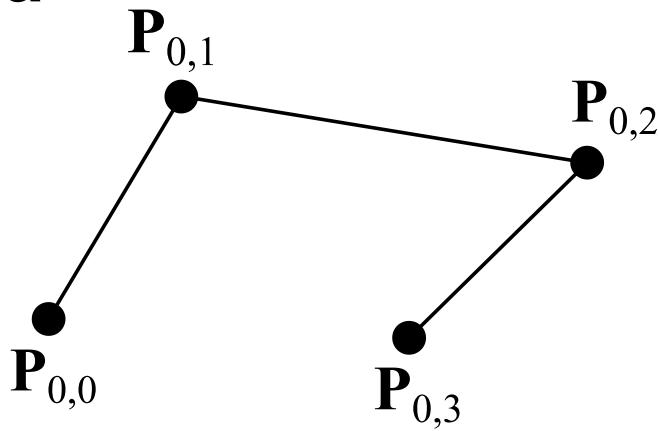
- Bezier curves may be computed and visualised using a geometric construction introduced by Paul de Casteljau.
- Like a cubic patch, we need a parameter μ which is
 - 0 at the start of the curve
 - 1 at the end.
- The construction
 - is recursive
 - can be made for any value of μ

Casteljau's Construction $\mu = 0.25$

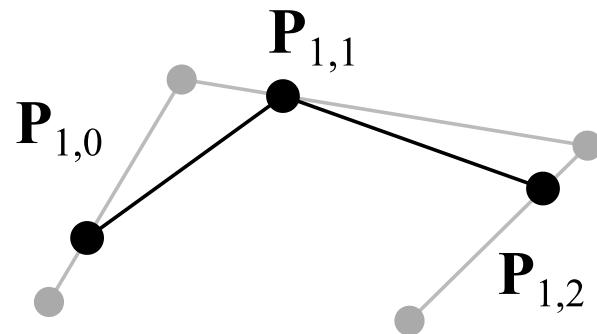


Casteljau's Construction $\mu = 0.25$

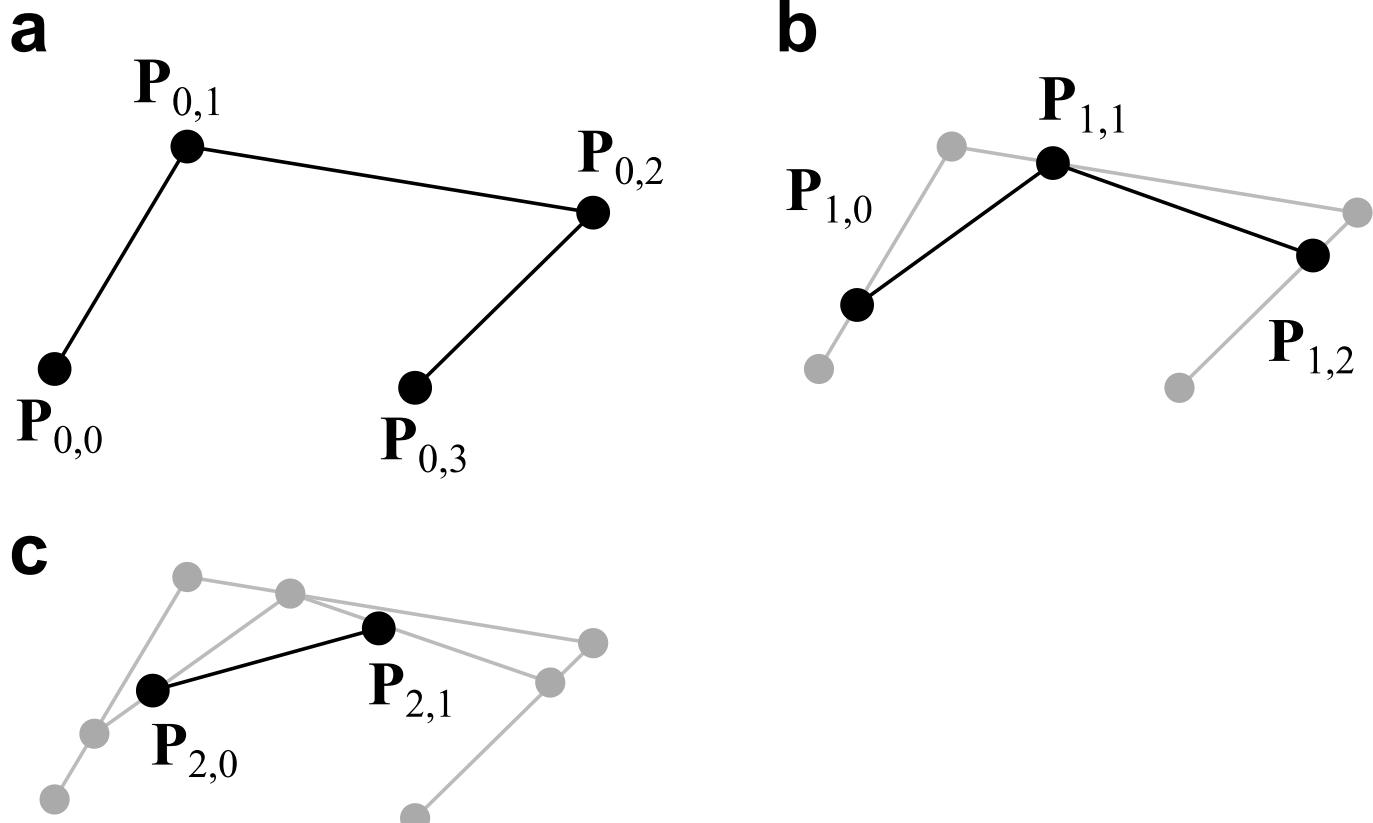
a



b

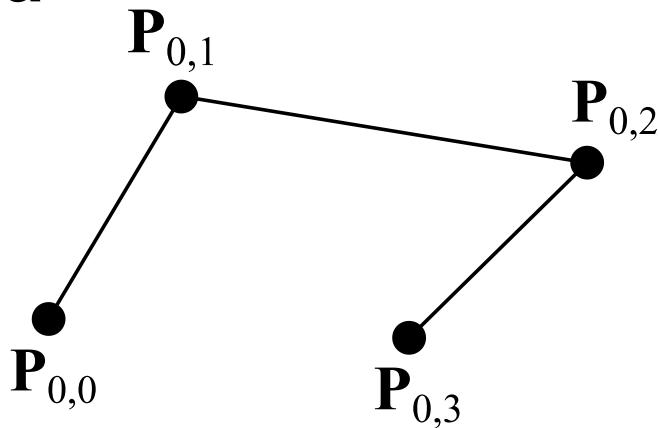


Casteljau's Construction $\mu = 0.25$

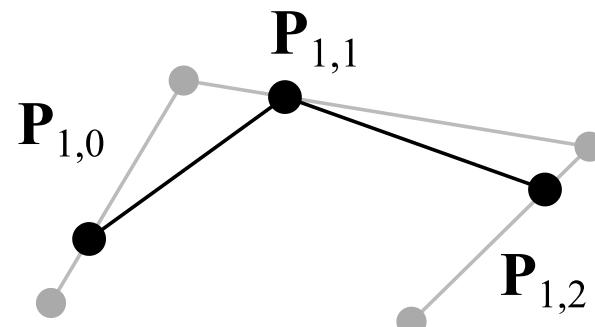


Casteljau's Construction $\mu = 0.25$

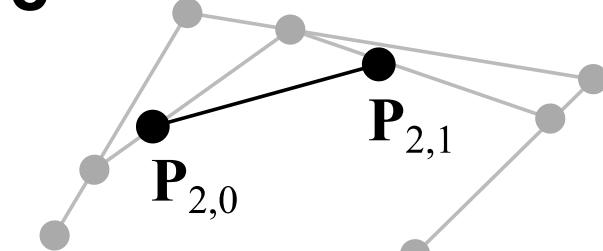
a



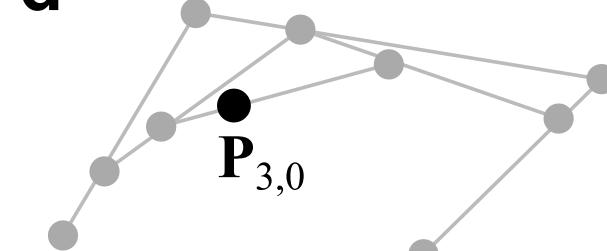
b



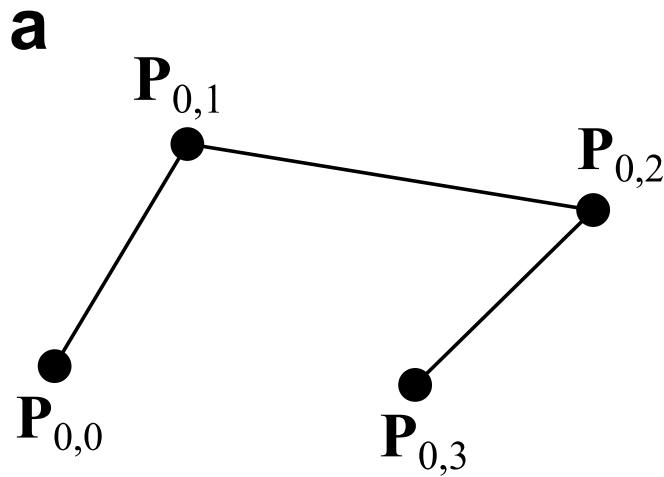
c



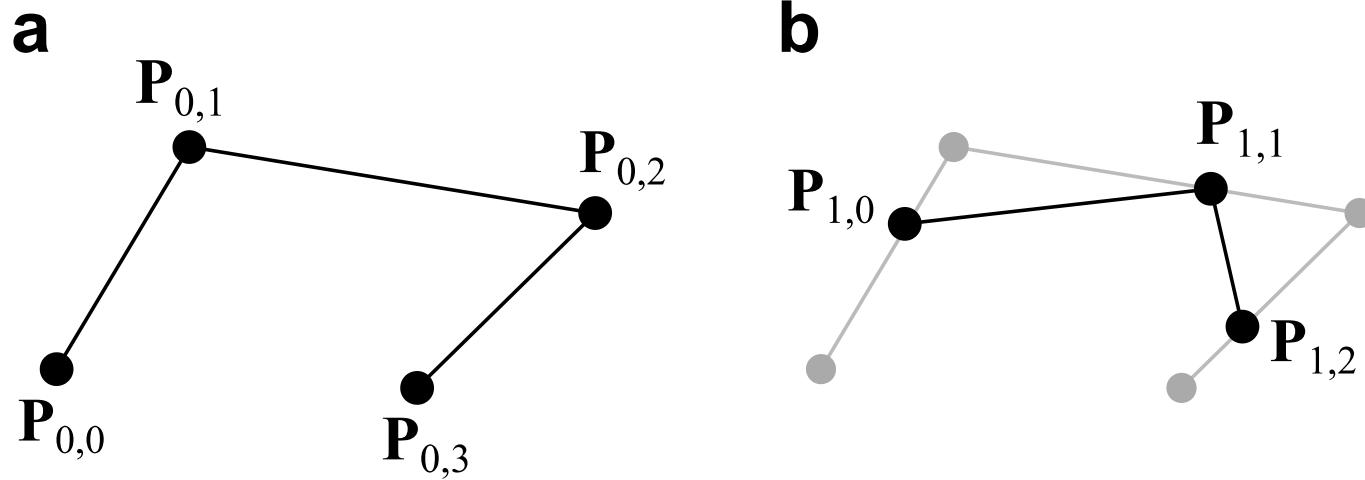
d



Casteljau's Construction $\mu = 0.6$

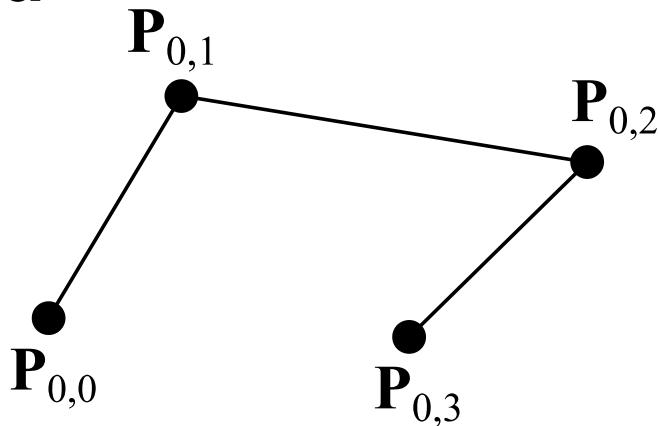


Casteljau's Construction $\mu = 0.6$

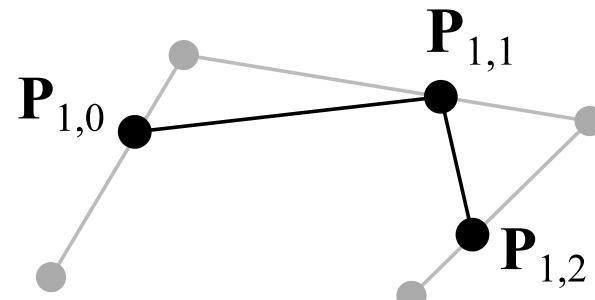


Casteljau's Construction $\mu = 0.6$

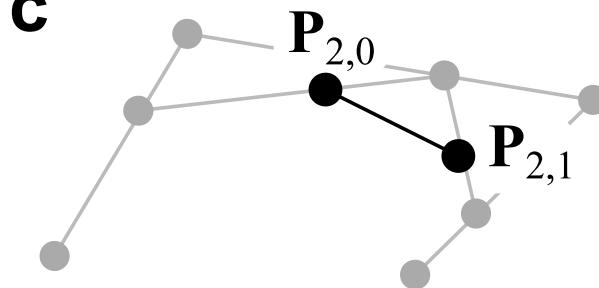
a



b

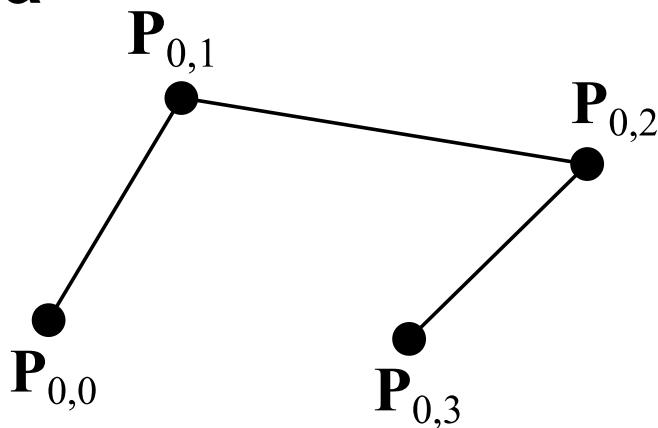


c

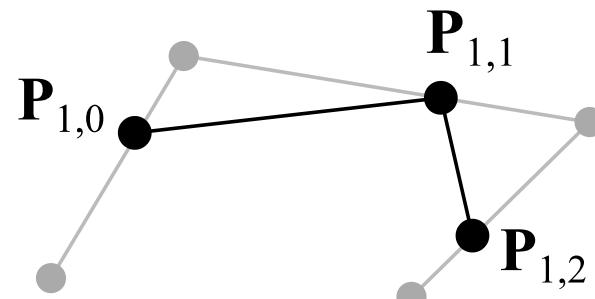


Casteljau's Construction $\mu = 0.6$

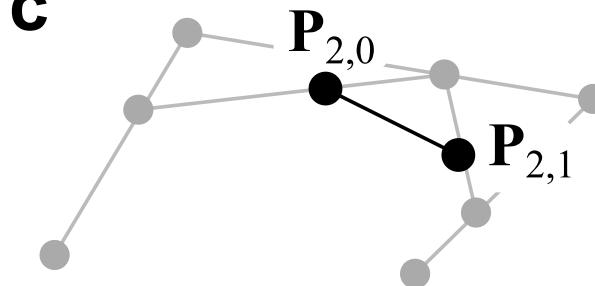
a



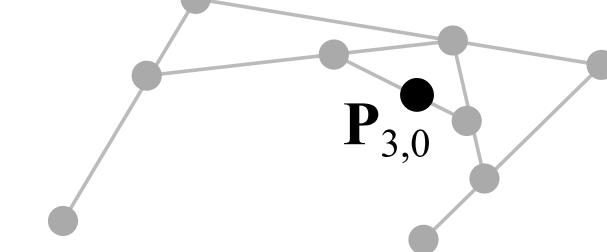
b



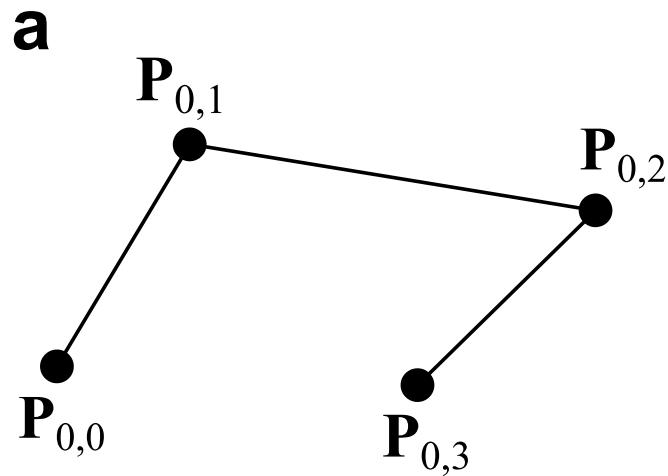
c



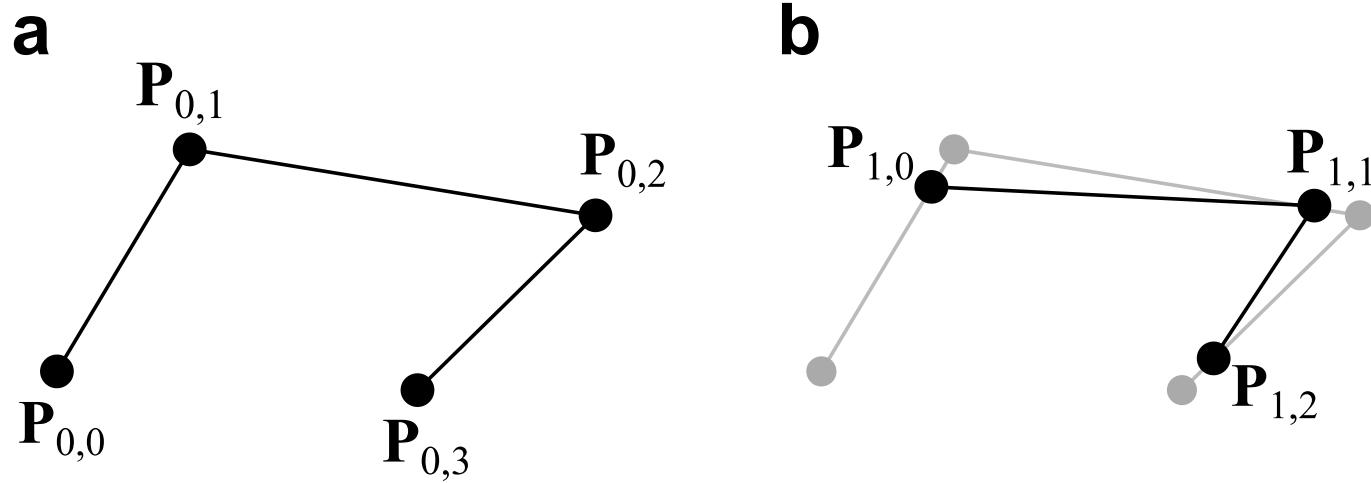
d



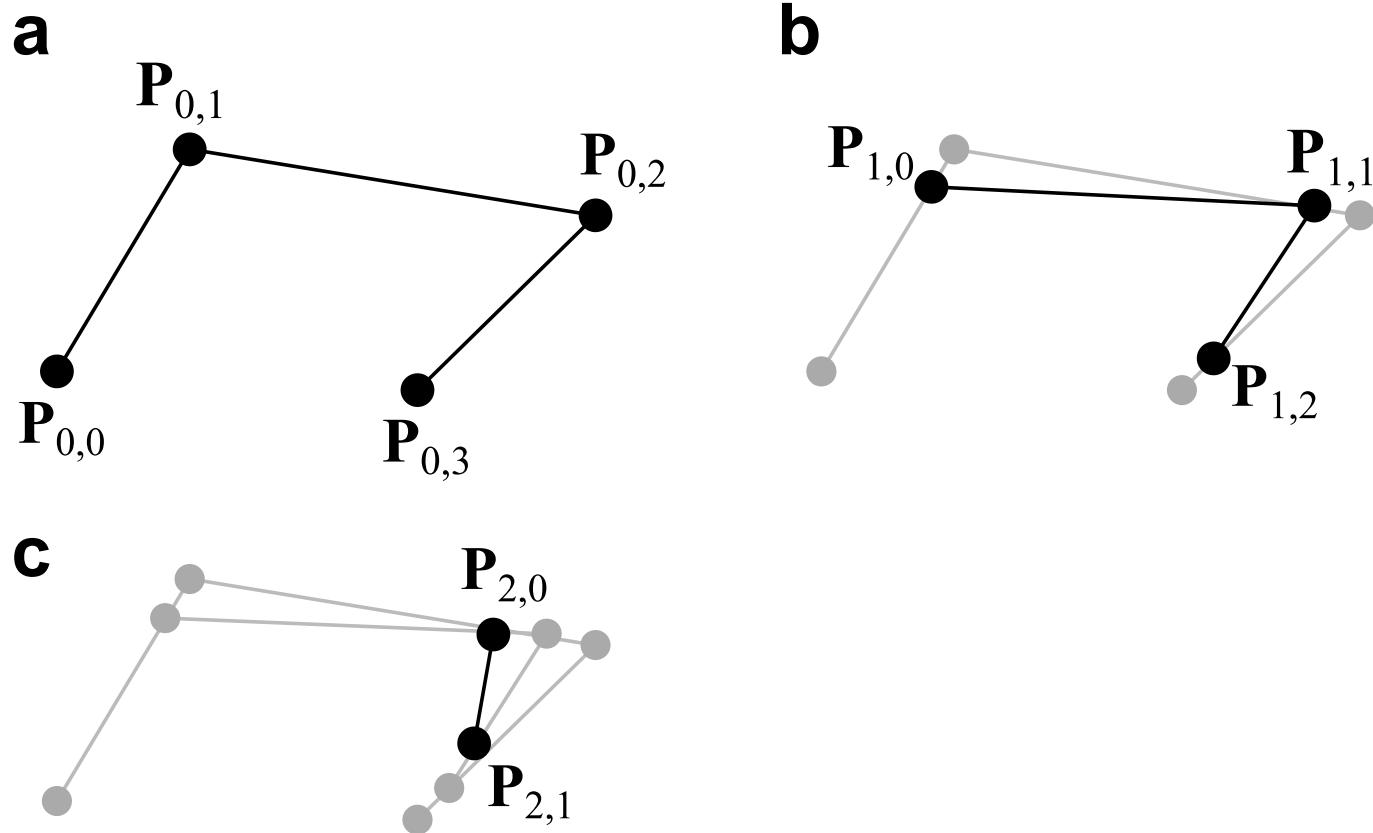
Casteljau's Construction $\mu = 0.9$



Casteljau's Construction $\mu = 0.9$

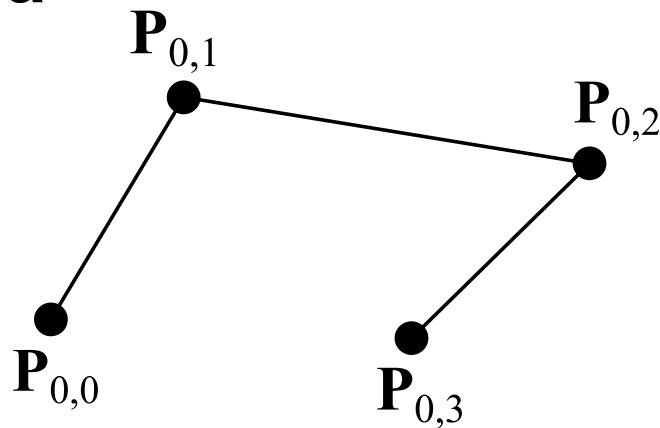


Casteljau's Construction $\mu = 0.9$

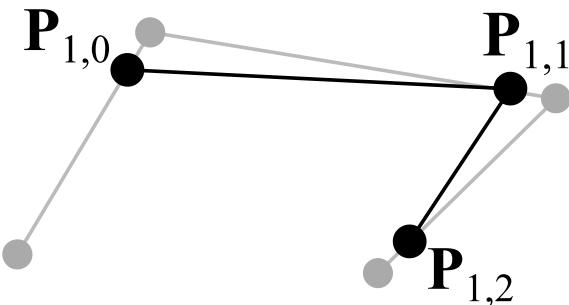


Casteljau's Construction $\mu = 0.9$

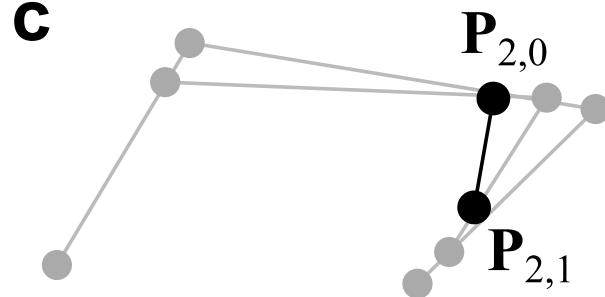
a



b



c



d



Bernstein Blending Function

- Splines (including Bezier curves) can be formulated as a blend of the knots.
- Consider the vector line equation

$$\mathbf{P} = (1 - \mu)\mathbf{P}_0 + \mu\mathbf{P}_1$$

- It is a linear ‘blend’ of two points, and could also be considered the two-point Bezier curve!

Blending Equation

- Any point on the spline is simply a blend of all the other points. For $N+1$ knots we have:

$$\mathbf{P}(\mu) = \sum_{i=0}^N W(N, i, \mu) \mathbf{P}_i$$

- where W is the Bernstein blending function

$$W(N, i, \mu) = \binom{N}{i} \mu^i (1 - \mu)^{N-i}$$

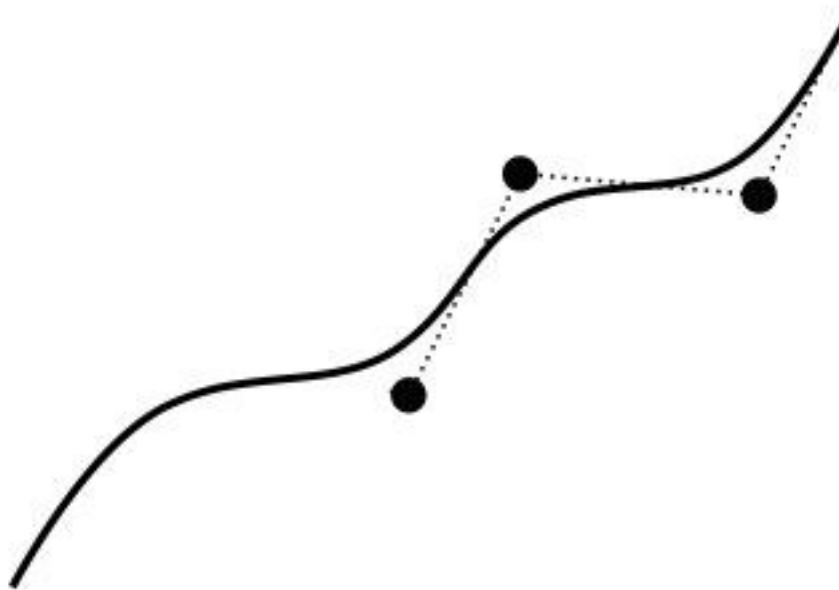
$$\binom{N}{i} = \frac{N!}{(N - i)!i!}$$

Blending Equation: Expansions for different N

N	Expansion
1	$(1 - \mu)\mathbf{P}_0 + \mu\mathbf{P}_1$
2	$(1 - \mu)^2\mathbf{P}_0 + 2\mu(1 - \mu)\mathbf{P}_1 + \mu^2\mathbf{P}_2$
3	$(1 - \mu)^3\mathbf{P}_0 + 3\mu(1 - \mu)^2\mathbf{P}_1 + 3\mu^2(1 - \mu)\mathbf{P}_2 + \mu^3\mathbf{P}_3$
:	:

Bezier Curves lack local control

- Since all the knots of the Bezier curve all appear in the blend they cannot be used for curves with fine detail.
- However they are very effective as spline patches.



Four point Bezier Curves and Cubic Patches

- We can show their equivalence:

Four point Bezier curve = Cubic patch going through the first and last knots (\mathbf{P}_0 and \mathbf{P}_3)

- It is possible to show their equivalence by
 - Expanding the iterative blending equation
 - Reversing the de Casteljau algorithm

Expanding the blending equation

- For the case of four knots we can expand the Bernstein blending function to get a polynomial in μ :

$$\begin{aligned}\mathbf{P}(\mu) &= \sum_{i=0}^3 W(3, i, \mu) \mathbf{P}_i \\ &= (1 - \mu)^3 \mathbf{P}_0 + 3\mu(1 - \mu)^2 \mathbf{P}_1 + 3\mu^2(1 - \mu) \mathbf{P}_2 + \mu^3 \mathbf{P}_3\end{aligned}$$

- This can be multiplied out to give an equation of the form:

$$\mathbf{P}(\mu) = \mathbf{a}_3 \mu^3 + \mathbf{a}_2 \mu^2 + \mathbf{a}_1 \mu + \mathbf{a}_0$$

where

$$\mathbf{a}_0 = \mathbf{P}_0$$

$$\mathbf{a}_1 = 3\mathbf{P}_1 - 3\mathbf{P}_0$$

$$\mathbf{a}_2 = 3\mathbf{P}_2 - 6\mathbf{P}_1 + 3\mathbf{P}_0$$

$$\mathbf{a}_3 = \mathbf{P}_3 - 3\mathbf{P}_2 + 3\mathbf{P}_1 - \mathbf{P}_0$$

Expanding the blending equation

- These equations are linear

$$\mathbf{a}_0 = \mathbf{P}_0$$

$$\mathbf{a}_1 = 3\mathbf{P}_1 - 3\mathbf{P}_0$$

$$\mathbf{a}_2 = 3\mathbf{P}_2 - 6\mathbf{P}_1 + 3\mathbf{P}_0$$

$$\mathbf{a}_3 = \mathbf{P}_3 - 3\mathbf{P}_2 + 3\mathbf{P}_1 - \mathbf{P}_0$$

- Note that \mathbf{P}_0 and \mathbf{P}_3 are the endpoints
- Recall the matrix form used for a cubic spline patch

$$\begin{pmatrix} \mathbf{a}_0 \\ \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3 & -2 & 3 & -1 \\ 2 & 1 & -2 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{P}_0 \\ \mathbf{P}'_0 \\ \mathbf{P}_3 \\ \mathbf{P}'_3 \end{pmatrix}$$

Expanding the blending equation

- These equations are linear

$$\mathbf{a}_0 = \mathbf{P}_0$$

$$\mathbf{a}_1 = 3\mathbf{P}_1 - 3\mathbf{P}_0$$

$$\mathbf{a}_2 = 3\mathbf{P}_2 - 6\mathbf{P}_1 + 3\mathbf{P}_0$$

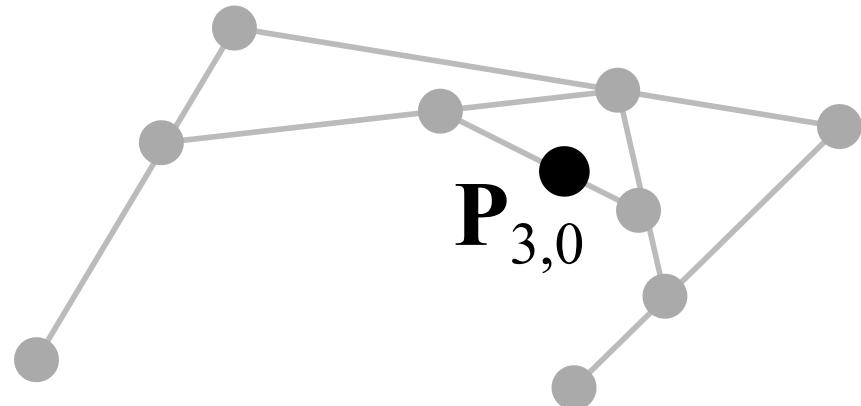
$$\mathbf{a}_3 = \mathbf{P}_3 - 3\mathbf{P}_2 + 3\mathbf{P}_1 - \mathbf{P}_0$$

- So we get the directions at the endpoints by using \mathbf{P}_1 and \mathbf{P}_2 .
- We have shown the blending equation is the same as a cubic patch

$$\begin{pmatrix} \mathbf{a}_0 \\ \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3 & -2 & 3 & -1 \\ 2 & 1 & -2 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{P}_0 \\ 3\mathbf{P}_1 - 3\mathbf{P}_0 \\ \mathbf{P}_3 \\ 3\mathbf{P}_3 - 3\mathbf{P}_2 \end{pmatrix}$$

Reversing the de Casteljau algorithm

We start from the point $\mathbf{P}_{3,0}$ and work in reverse to express it in terms of its construction line.



$$\begin{aligned}\mathbf{P}_{3,0} &= (1 - \mu)\mathbf{P}_{2,0} + \mu\mathbf{P}_{2,1} \\ &= (1 - \mu)\{(1 - \mu)\mathbf{P}_{1,0} + \mu\mathbf{P}_{1,1}\} + \mu\{(1 - \mu)\mathbf{P}_{1,1} + \mu\mathbf{P}_{1,2}\} \\ &= (1 - \mu)^2\mathbf{P}_{1,0} + 2\mu(1 - \mu)\mathbf{P}_{1,1} + \mu^2\mathbf{P}_{1,2} \\ &= (1 - \mu)^2\{(1 - \mu)\mathbf{P}_{0,0} + \mu\mathbf{P}_{0,1}\} \\ &\quad + 2\mu(1 - \mu)\{(1 - \mu)\mathbf{P}_{0,1} + \mu\mathbf{P}_{0,2}\} \\ &\quad + \mu^2\{(1 - \mu)\mathbf{P}_{0,2} + \mu\mathbf{P}_{0,3}\}\end{aligned}$$

Reversing the de Casteljau algorithm

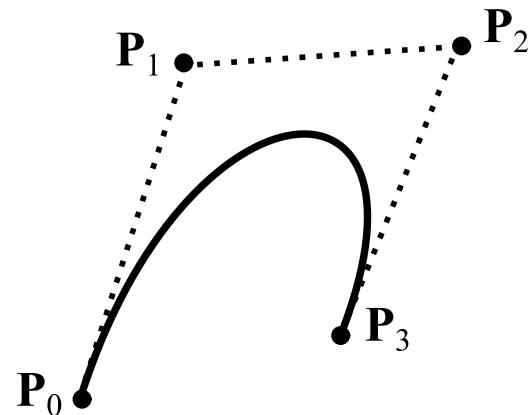
. . . continuing the expansion, we can drop the first subscript (which indicates the recursion level) to get:

$$\begin{aligned}\mathbf{P}(\mu) &= (1 - \mu)^2 \{(1 - \mu)\mathbf{P}_0 + \mu\mathbf{P}_1\} \\ &\quad + 2\mu(1 - \mu) \{(1 - \mu)\mathbf{P}_1 + \mu\mathbf{P}_2\} \\ &\quad + \mu^2 \{(1 - \mu)\mathbf{P}_2 + \mu\mathbf{P}_3\} \\ &= (1 - \mu)^3\mathbf{P}_0 + 3\mu(1 - \mu)^2\mathbf{P}_1 + 3\mu^2(1 - \mu)\mathbf{P}_2 + \mu^3\mathbf{P}_3\end{aligned}$$

This is the same as the expanded Bernstein blending polynomial which we have already shown is equivalent to a cubic spline patch

Control Points

- We can summarise the four point Bezier Curve by saying that it has
 - two points that are interpolated (P_0, P_3)
 - two control points (P_1, P_2)
- The curve starts at P_0 and ends at P_3 and its shape can be determined by moving control points P_1, P_2 .
- This could be done interactively using a mouse.



In summary ...

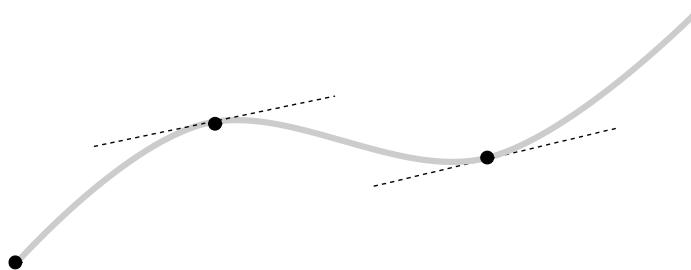
- The simplest and most effective way to draw a smooth curve through a set of points is to use a cubic patch.

No interaction needed?

setting the gradients by
the central difference

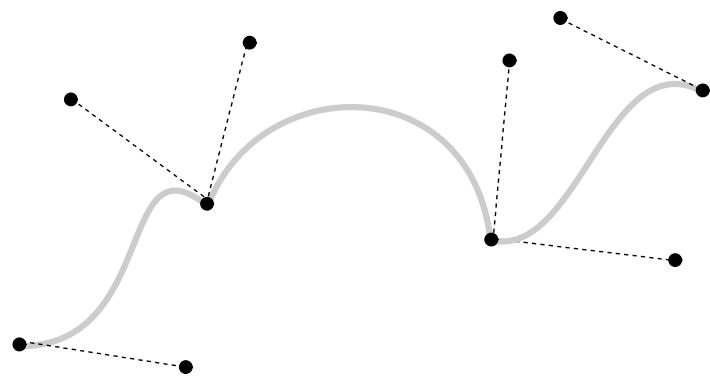
$$\frac{1}{2}(\mathbf{P}_{i+1} - \mathbf{P}_{i-1})$$

is effective.



User wants interactive
shape adjustment?

The four point Bezier
formulation is ideal



Interactive Computer Graphics: *Lecture 13*

Introduction to Surface Construction

Teapot Subdivision: Russ Fish



Non Parametric Surface

- Surfaces can be constructed from Cartesian equations directly, and this is acceptable for specific applications, usually involving interpolation.
- As before, using a simple polynomial surface is a quick and easy approach.

Non Parametric Polynomial Surface

$$(x \ y \ z \ 1) \begin{pmatrix} a & b & c & d \\ b & e & f & g \\ c & f & h & j \\ d & g & j & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = 0$$

- which multiplies out to:

$$ax^2 + ey^2 + hz^2 + 2bxy + 2cxz + 2fyz + 2dx + 2gy + 2jz + 1 = 0$$

- Because of the symmetry there are 9 scalar unknowns in the equation
- So we need to specify nine points through which the surface will pass

As Before

- This formulation suffers the same problems as the non-parametric spline curve. It is a fixed surface for a given set of nine points.
- We need more flexibility for the design of surfaces.

Simple Parametric surfaces

- We can extend the formulation to simple parametric surfaces using the vector equation:

$$\mathbf{P}(\mu, \nu) = (\mu, \nu, 1) \begin{pmatrix} \mathbf{a} & \mathbf{b} & \mathbf{c} \\ \mathbf{b} & \mathbf{d} & \mathbf{e} \\ \mathbf{c} & \mathbf{e} & \mathbf{f} \end{pmatrix} \begin{pmatrix} \mu \\ \nu \\ 1 \end{pmatrix}$$

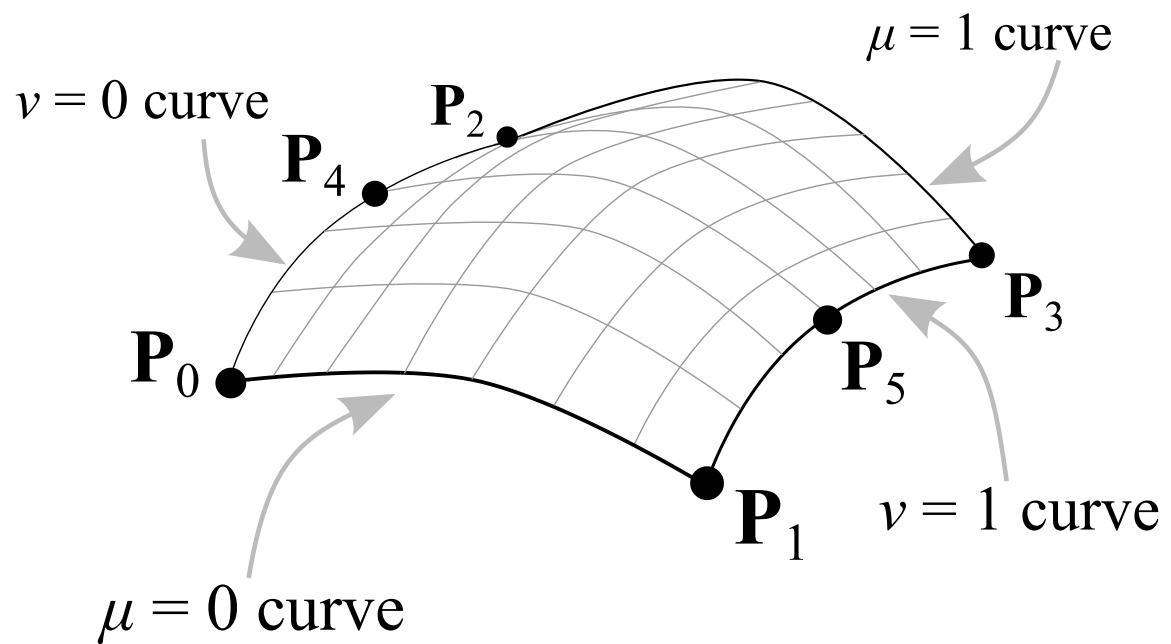
$$\mathbf{P}(\mu, \nu) = \mathbf{a}\mu^2 + \mathbf{d}\nu^2 + 2\mathbf{b}\mu\nu + 2\mathbf{c}\mu + 2\mathbf{e}\nu + \mathbf{f}$$

- There are six unknown parameter vectors $\{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{f}\}$

Associating points and parameters

- We can solve for the six vector unknowns by substituting in six points at known values of μ and ν .
- We might have an arrangement such as:

	μ	ν
P_0	0	0
P_1	0	1
P_2	1	0
P_3	1	1
P_4	1/2	0
P_5	1/2	1



Surface parameter equations

- Substituting these values of μ and ν into the patch equation gives us these six equations

$$\mathbf{P}_0 = \mathbf{f}$$

$$\mathbf{P}_1 = \mathbf{d} + 2\mathbf{e} + \mathbf{f}$$

$$\mathbf{P}_2 = \mathbf{a} + 2\mathbf{c} + \mathbf{f}$$

$$\mathbf{P}_3 = \mathbf{a} + 2\mathbf{b} + 2\mathbf{c} + \mathbf{d} + 2\mathbf{e} + \mathbf{f}$$

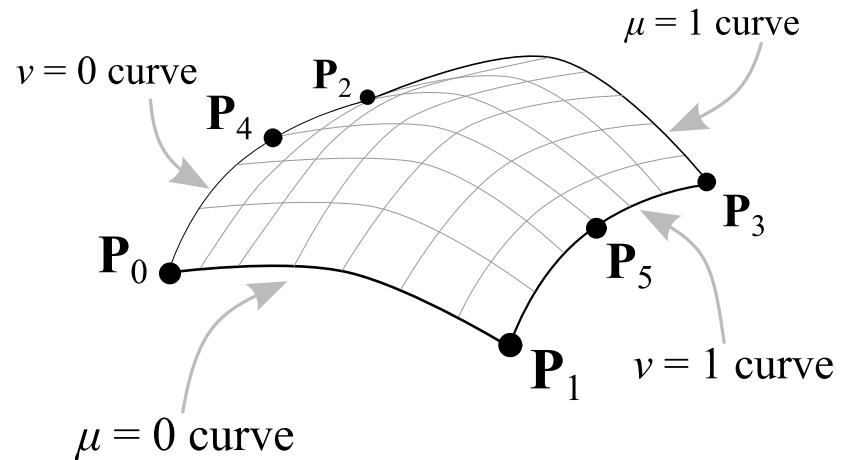
$$\mathbf{P}_4 = \mathbf{a}/4 + \mathbf{c} + \mathbf{f}$$

$$\mathbf{P}_5 = \mathbf{a}/4 + \mathbf{b} + \mathbf{c} + \mathbf{d} + 2\mathbf{e} + \mathbf{f}$$

- The \mathbf{P} 's are known and we can solve for the unknowns $\{\mathbf{a}, \dots, \mathbf{f}\}$ using standard methods

Getting the edges from the surface equation

μ and ν are in the range $[0, 1]$.
Thus the contours that bound the
patch can be found by
substituting 0 or 1 for one of μ or ν
in the patch equation.



$$\mathbf{P}(0, \nu) = \mathbf{d}\nu^2 + 2\mathbf{e}\nu + \mathbf{f}$$

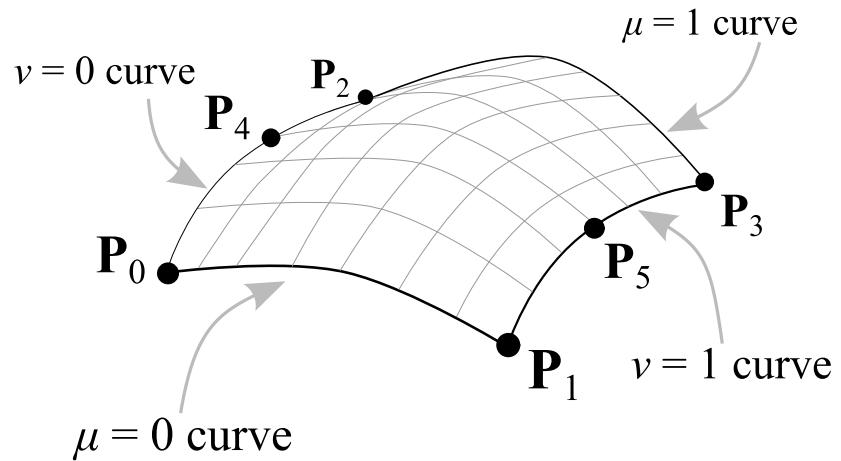
$$\mathbf{P}(1, \nu) = \mathbf{a} + 2(\mathbf{b} + \mathbf{e})\nu + 2\mathbf{c} + \mathbf{d}\nu^2 + \mathbf{f}$$

$$\mathbf{P}(\mu, 0) = \mathbf{a}\mu^2 + 2\mathbf{c}\mu + \mathbf{f}$$

$$\mathbf{P}(\mu, 1) = \mathbf{a}\mu^2 + 2(\mathbf{b} + \mathbf{c})\mu + \mathbf{d} + 2\mathbf{e} + \mathbf{f}$$

The resulting surface

The boundaries are all second order curves and so will be nice and smooth



There is quite a lot of flexibility in this formulation, but it is still only suitable for simple surfaces.

We can use higher orders

E.g. using the tensor product

$$\mathbf{P}(\mu, \nu) = (\mu^3 \quad \mu^2 \quad \mu \quad 1) \begin{pmatrix} \mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{d} \\ \mathbf{b} & \mathbf{e} & \mathbf{f} & \mathbf{g} \\ \mathbf{c} & \mathbf{f} & \mathbf{h} & \mathbf{j} \\ \mathbf{d} & \mathbf{g} & \mathbf{j} & \mathbf{k} \end{pmatrix} \begin{pmatrix} \nu^3 \\ \nu^2 \\ \nu \\ 1 \end{pmatrix}$$

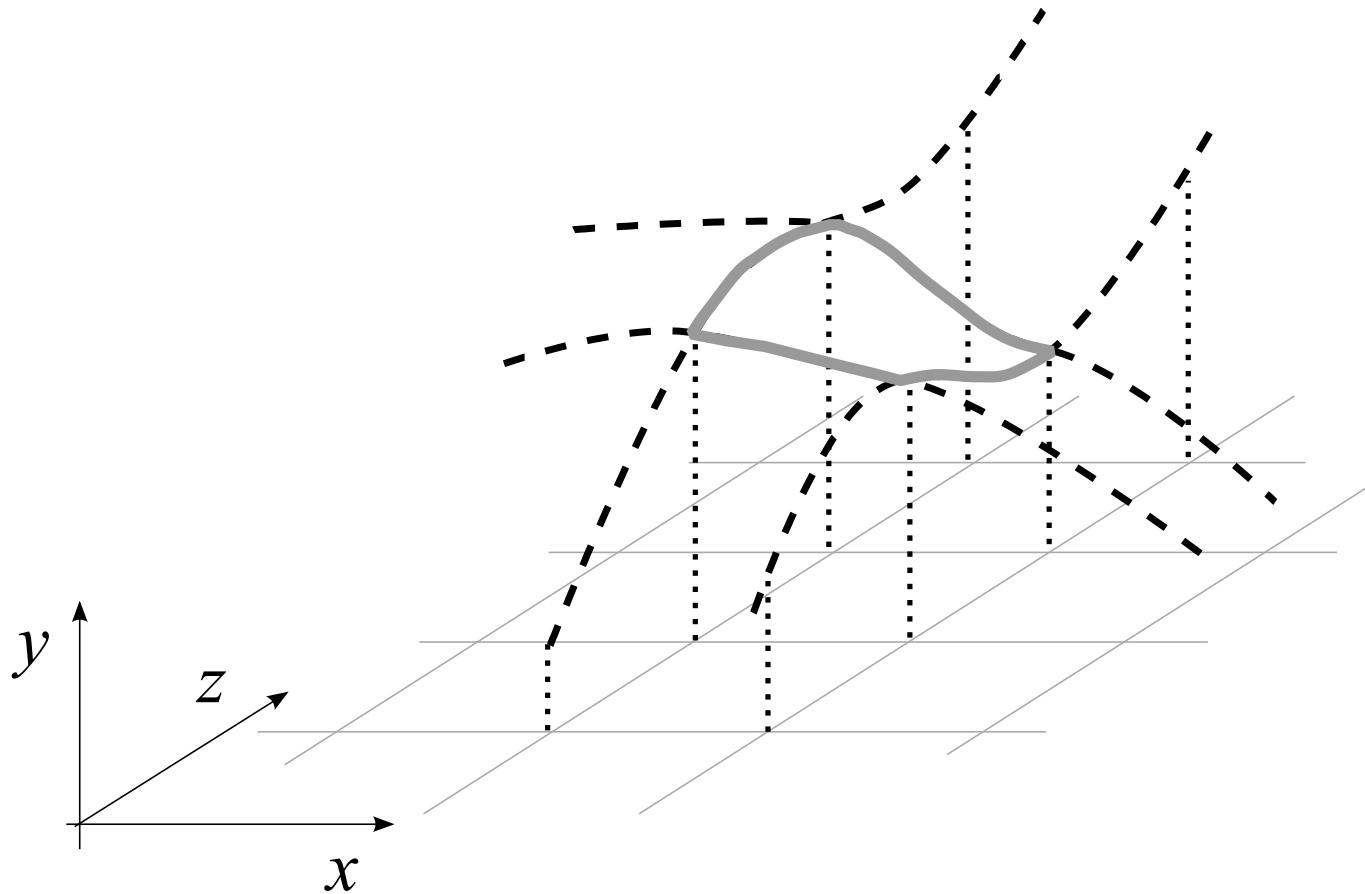
Using higher orders gives more variety in shape and better control

But the method is hard to apply and generalise, and so is not usually done

Cubic Spline Patches

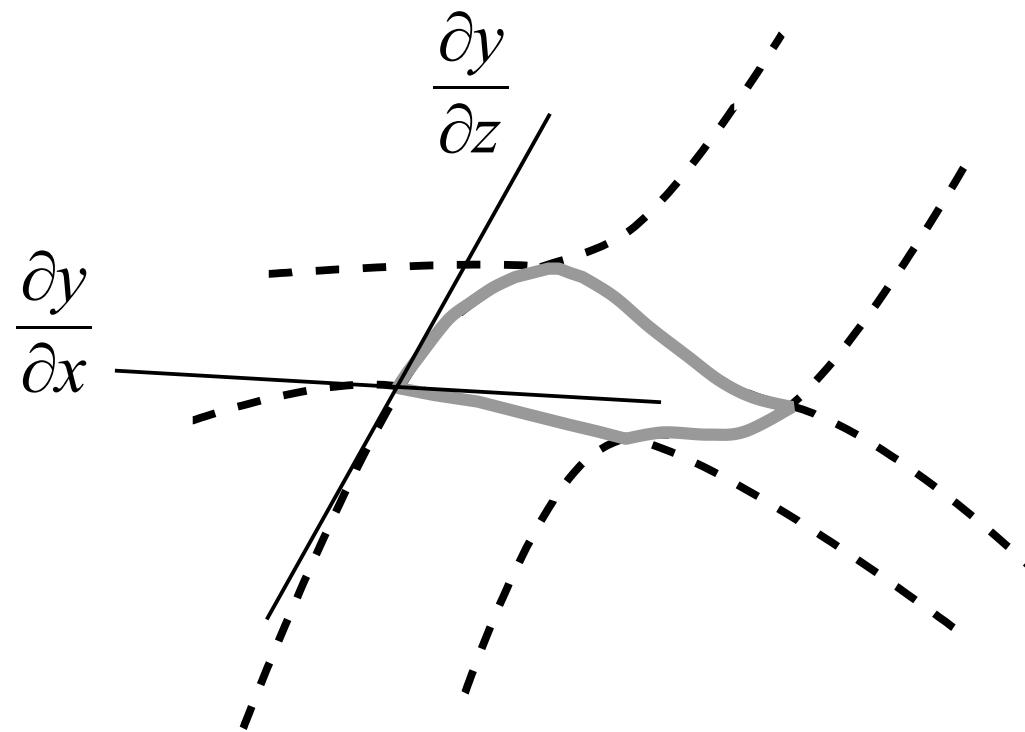
- The patch method is generally effective in creating more complex surfaces.
- The idea is, as in the case of the curves, to create a surface by joining a lot of simple surfaces continuously.

Cartesian surface patches - terrain map



Points and Gradients

- At each corner of the patch we need to interpolate the points and set the gradients to match the adjacent patch.
- There are two gradients



Parametric patches

- In practice we use the more general parametric patch formulation with two parameters μ and ν .
- The terrain map can be modelled with parametric patches.
- We need to match three values at each corner

$$\begin{array}{ccc} \mathbf{P}(\mu, \nu) & \frac{\partial \mathbf{P}(\mu, \nu)}{\partial \mu} & \frac{\partial \mathbf{P}(\mu, \nu)}{\partial \nu} \end{array}$$

Corners

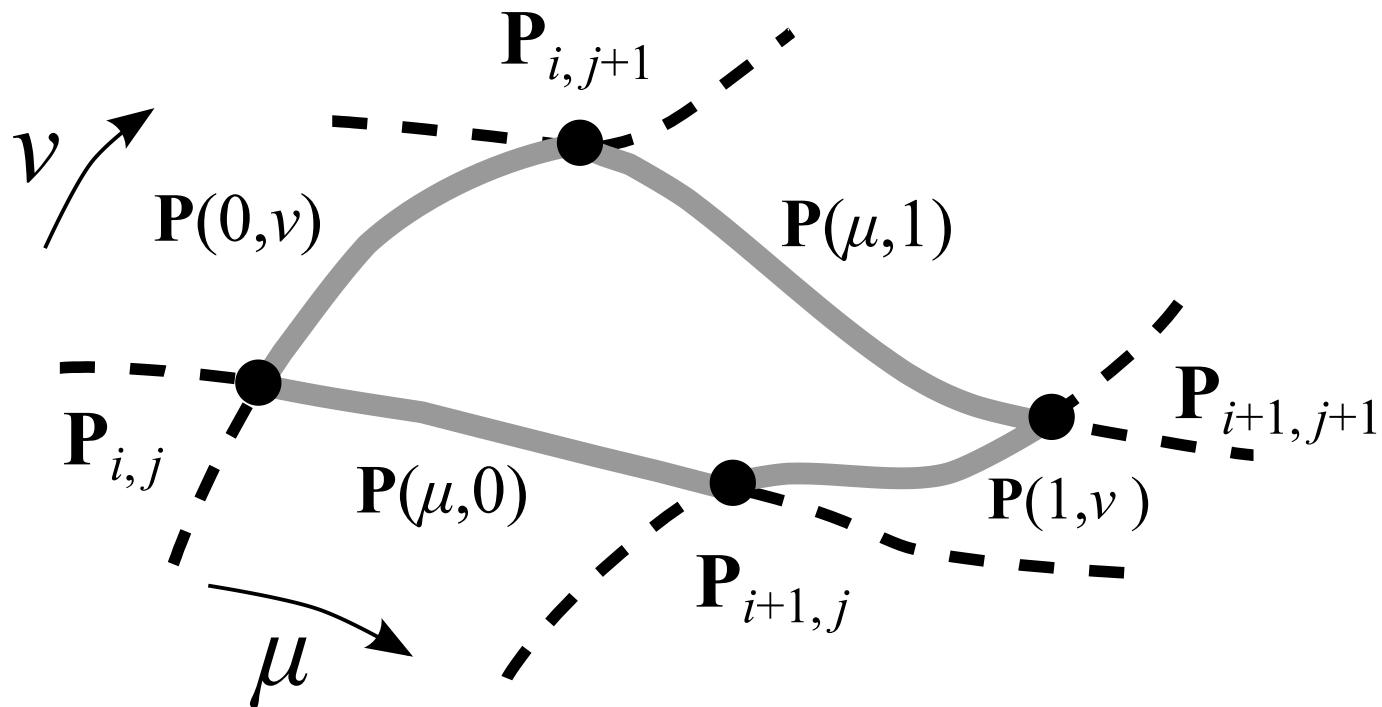
- As usual we adopt the convention that the corners are at parameter values $(0, 0)$, $(0, 1)$, $(1, 0)$ and $(1, 1)$
- We need to ensure that the patch joins its neighbours exactly at the edges.
- Hence we ensure that the edge contours are the same on adjacent patches

Edges

- We do this by designing the edge curves in an identical manner to the cubic spline curve patch.

Edge curve	Points joined	
$\mathbf{P}(0, \nu)$	$\mathbf{P}_{i,j}$	$\mathbf{P}_{i,j+1}$
$\mathbf{P}(1, \nu)$	$\mathbf{P}_{i+1,j}$	$\mathbf{P}_{i+1,j+1}$
$\mathbf{P}(\mu, 0)$	$\mathbf{P}_{i,j}$	$\mathbf{P}_{i+1,j}$
$\mathbf{P}(\mu, 1)$	$\mathbf{P}_{i,j+1}$	$\mathbf{P}_{i+1,j+1}$

A parametric spline patch



As long as the gradients are the same for the four patches that meet at a point the surface will join seamlessly

The Coons patch

To define the internal points we linearly interpolate the edge curves:

$$\begin{aligned}\mathbf{P}(\mu, \nu) = & \mathbf{P}(\mu, 0)(1 - \nu) + \mathbf{P}(\mu, 1)\nu + \\ & \mathbf{P}(0, \nu)(1 - \mu) + \mathbf{P}(1, \nu)\mu - \\ & \mathbf{P}(0, 1)(1 - \mu)\nu - \mathbf{P}(1, 0)\mu(1 - \nu) - \\ & \mathbf{P}(0, 0)(1 - \mu)(1 - \nu) - \mathbf{P}(1, 1)\mu\nu\end{aligned}$$

Substituting values of 0 or 1 for μ and/or ν we can easily verify that the equation fits the edge curves.

Rendering a patch: Polygonisation

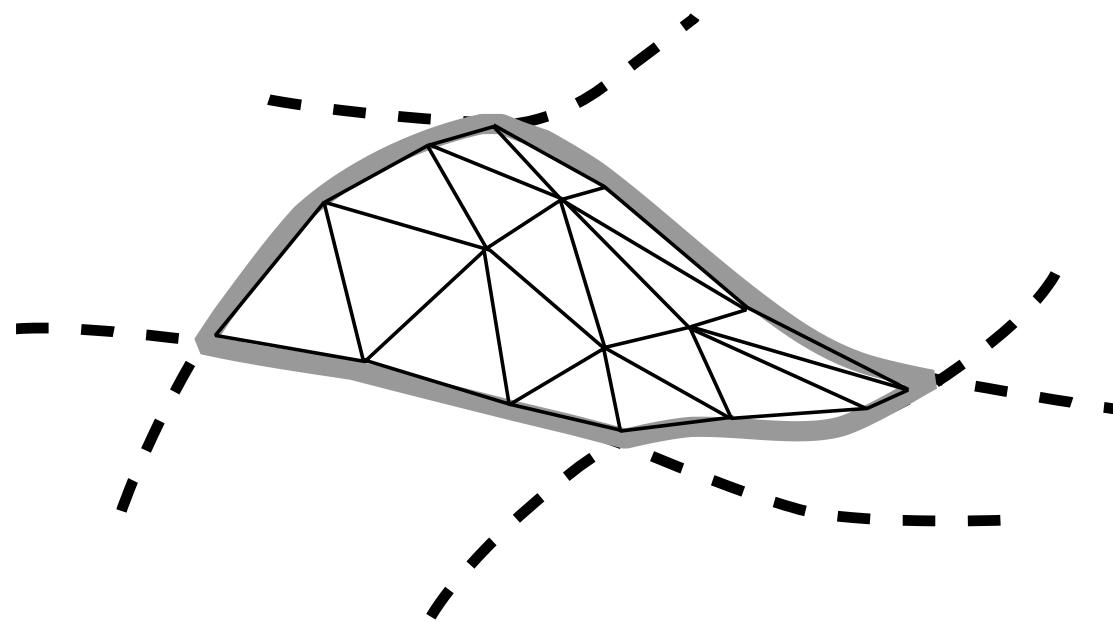
To render (draw) a spline patch we can simply transform it into polygons.

We select a grid of points, e.g.:

$$\begin{aligned}\mu &= \{0.0, 0.1, 0.2, \dots, 1.0\} \\ \nu &= \{0.0, 0.1, 0.2, \dots, 1.0\}\end{aligned}$$

and triangulate to that grid.

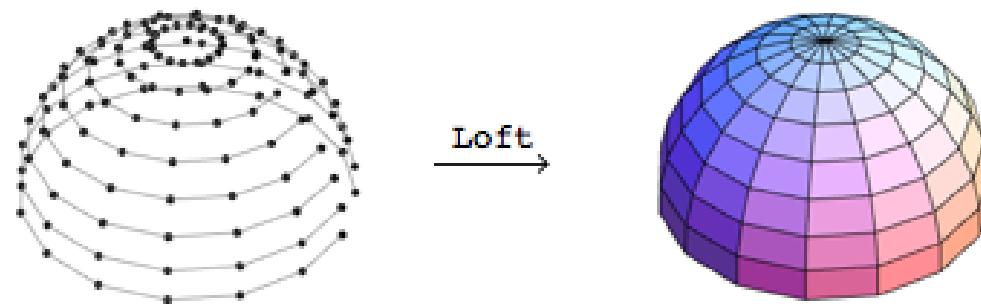
Rendering a patch: Polygonisation



- For speed we can use large polygons with Gouraud or Phong shading.
- For accuracy we use small polygons, chosen to match the pixel size.

Rendering a patch: Lofting

- Surfaces can also be drawn by a technique called lofting (now really obsolete).
- This means drawing contours of constant μ and/or of constant ν
- Algorithms for eliminating the hidden parts have been devised.



Rendering a patch: Ray tracing

- The patch equation is fourth order

$$\begin{aligned}\mathbf{P}(\mu, \nu) = & \mathbf{P}(\mu, 0)(1 - \nu) + \mathbf{P}(\mu, 1)\nu + \\ & \mathbf{P}(0, \nu)(1 - \mu) + \mathbf{P}(1, \nu)\mu - \\ & \mathbf{P}(0, 1)(1 - \mu)\nu - \mathbf{P}(1, 0)\mu(1 - \nu) - \\ & \mathbf{P}(0, 0)(1 - \mu)(1 - \nu) - \mathbf{P}(1, 1)\mu\nu\end{aligned}$$

- Hence no closed form solution exists for a ray patch intersection
- Can use numeric algorithms but computation can be costly

Rendering a patch: Ray tracing

- Numerical Ray-Patch algorithm
 1. Polygonise the patch at a low resolution (say 4×4)
 2. Calculate the ray intersection with the 32 triangles and find the nearest intersection.
 3. Polygonise the immediate area of the intersection and calculate a better estimate of the intersection
 4. Continue until the best estimate is found

Rendering a patch: Ray tracing

- Numerical Ray-Patch algorithm
 - May be multiple intersections between the ray and the surface
 - Algorithm will find an intersection, but not necessarily the nearest.
 - If the object is relatively smooth it should work well in most cases.
 - Note that it will be necessary to do a ray intersection with each patch of the object to find the nearest intersection.

Example of Using a Coons Patch

- Part of a terrain map defined on a regular x - y grid is as follows:

		$y, \nu \rightarrow$					
		2	3	4	5	6	7
		7
		8	.	.	10	9	.
x, μ		9	.	14	12	11	10
\downarrow		10	.	15	13	14	10
		11	.	.	10	11	.

- Find the Coons patch on the centre four points

Corners

- The corners at $\mu, \nu = 0, 1$ are defined directly in the question:

$$\mathbf{P}(0, 0) = (9, 4, 12)$$

$$\mathbf{P}(0, 1) = (9, 5, 11)$$

$$\mathbf{P}(1, 0) = (10, 4, 13)$$

$$\mathbf{P}(1, 1) = (10, 5, 14)$$

		$y, \nu \rightarrow$						
		2	3	4	5	6	7	
		7
		8	.	.	10	9	.	.
x, μ		9	.	14	12	11	10	.
\downarrow		10	.	15	13	14	10	.
		11	.	.	10	11	.	.

Gradients in the x / μ direction

Example

$$\frac{\partial \mathbf{P}}{\partial \mu} \Big|_{(0,0)} = \frac{(10, 4, 13)^T - (8, 4, 10)^T}{2} = \begin{pmatrix} 1 \\ 0 \\ 1.5 \end{pmatrix}$$

		$y, \nu \rightarrow$					
		2	3	4	5	6	7
		7
		8	.	.	10	9	.
x, μ		9	.	14	12	11	10
↓		10	.	15	13	14	10
		11	.	.	10	11	.

Gradients in the x / μ direction

$$\frac{\partial \mathbf{P}}{\partial \mu} \Big|_{(0,0)} = \frac{(10, 4, 13)^T - (8, 4, 10)^T}{2} = (1, 0, 1.5)^T$$

$$\frac{\partial \mathbf{P}}{\partial \mu} \Big|_{(1,0)} = \frac{(11, 4, 10)^T - (9, 4, 12)^T}{2} = (1, 0, -1)^T$$

$$\frac{\partial \mathbf{P}}{\partial \mu} \Big|_{(0,1)} = \frac{(10, 5, 14)^T - (8, 5, 9)^T}{2} = (1, 0, 2.5)^T$$

$$\frac{\partial \mathbf{P}}{\partial \mu} \Big|_{(1,1)} = \frac{(11, 5, 11)^T - (9, 5, 11)^T}{2} = (1, 0, 0)^T$$

Gradients in the y / v direction

$$\frac{\partial \mathbf{P}}{\partial \nu} \Big|_{(0,0)} = \frac{(9, 5, 11)^T - (9, 3, 14)^T}{2} = (0, 1, -1.5)^T$$

$$\frac{\partial \mathbf{P}}{\partial \nu} \Big|_{(1,0)} = \frac{(10, 5, 14)^T - (10, 3, 15)^T}{2} = (0, 1, -0.5)^T$$

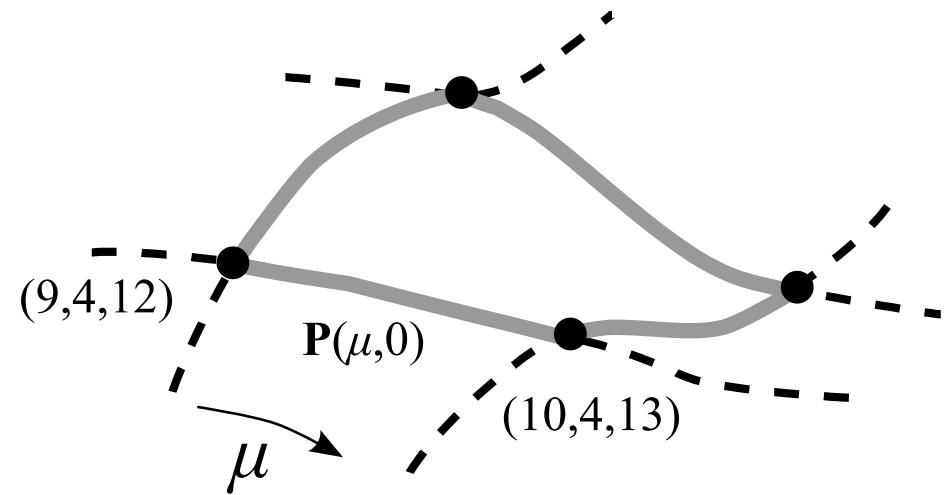
$$\frac{\partial \mathbf{P}}{\partial \nu} \Big|_{(0,1)} = \frac{(9, 6, 10)^T - (9, 4, 12)^T}{2} = (0, 1, -1)^T$$

$$\frac{\partial \mathbf{P}}{\partial \nu} \Big|_{(1,1)} = \frac{(10, 6, 10)^T - (10, 4, 13)^T}{2} = (0, 1, -1.5)^T$$

Finding the boundary curves

E.g. Finding curve $\mathbf{P}(\mu, 0)$

$$\mathbf{P}(\mu, 0) = \mathbf{a}_3\mu^3 + \mathbf{a}_2\mu^2 + \mathbf{a}_1\mu + \mathbf{a}_0$$



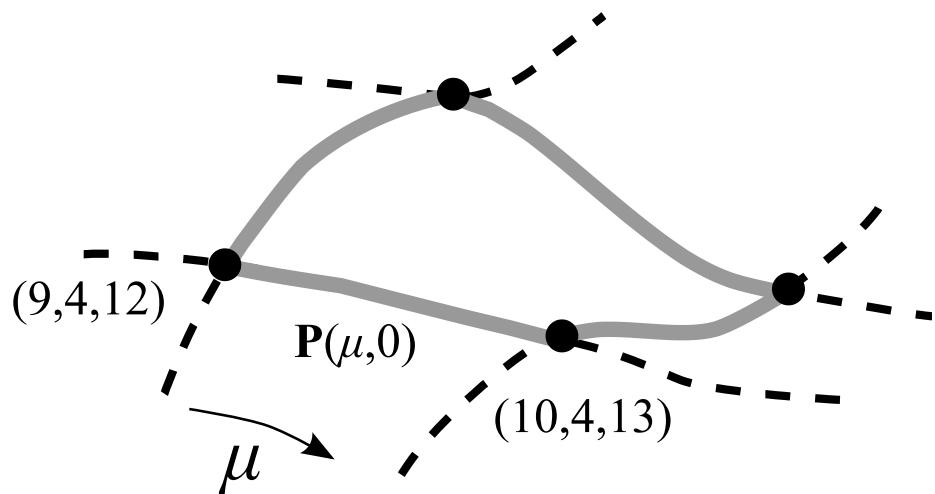
$$\begin{pmatrix} \mathbf{a}_0 \\ \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3 & -2 & 3 & -1 \\ 2 & 1 & -2 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{P}_0 \\ \mathbf{P}'_0 \\ \mathbf{P}_1 \\ \mathbf{P}'_1 \end{pmatrix}$$

- see cubic spline patch equation (previous lecture)

Finding the boundary curves

E.g. Finding curve $\mathbf{P}(\mu, 0)$

$$\mathbf{P}(\mu, 0) = \mathbf{a}_3\mu^3 + \mathbf{a}_2\mu^2 + \mathbf{a}_1\mu + \mathbf{a}_0$$



$$\begin{pmatrix} \mathbf{a}_0 \\ \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3 & -2 & 3 & -1 \\ 2 & 1 & -2 & 1 \end{pmatrix} \begin{pmatrix} 9 & 4 & 12 \\ 1 & 0 & 1.5 \\ 10 & 4 & 13 \\ 1 & 0 & -1 \end{pmatrix}$$

After substituting in $\mathbf{P}(0, 0)$, $\left. \frac{\partial \mathbf{P}}{\partial \mu} \right|_{(0,0)}$, $\mathbf{P}(1, 0)$, $\left. \frac{\partial \mathbf{P}}{\partial \mu} \right|_{(1,0)}$

Finding the boundary curve $\mathbf{P}(\mu, 0)$

- Calculating the constant vectors $\mathbf{a}_3, \mathbf{a}_2, \mathbf{a}_1$ and \mathbf{a}_0

$$\mathbf{a}_0 = \mathbf{P}_0 = (9, 4, 12)$$

$$\mathbf{a}_1 = \mathbf{P}'_0 = (1, 0, 1.5)$$

$$\mathbf{a}_2 = -3\mathbf{P}_0 - 2\mathbf{P}'_0 - 3\mathbf{P}_1 - \mathbf{P}'_1$$

$$\begin{aligned} &= -3 \times (9, 4, 12) - 2 \times (1, 0, 1.5) + 3 \times (10, 4, 13) - (1, 0, 1) \\ &= (0, 0, 1) \end{aligned}$$

$$\mathbf{a}_3 = 2\mathbf{P}_0 + \mathbf{P}'_0 - 2\mathbf{P}_1 + \mathbf{P}'_1$$

$$\begin{aligned} &= 2 \times (9, 4, 12) + (1, 0, 1.5) - 2 \times (10, 4, 13) + (1, 0, 1) \\ &= (0, 0, 0.5) \end{aligned}$$

Finding the boundary curves $\mathbf{P}(\mu, 1)$, $\mathbf{P}(0, \nu)$, $\mathbf{P}(1, \nu)$

- These curves are found identically to $\mathbf{P}(\mu, 0)$.
- We now have all the individual bits:

$\mathbf{P}(\mu, 0)$: a cubic polynomial in μ

$\mathbf{P}(\mu, 1)$: a cubic polynomial in μ

$\mathbf{P}(0, \nu)$: a cubic polynomial in ν

$\mathbf{P}(1, \nu)$: a cubic polynomial in ν

$\mathbf{P}(0, 0)$, $\mathbf{P}(0, 1)$, $\mathbf{P}(1, 0)$ and $\mathbf{P}(1, 1)$: the corner points

- Given values of μ and ν , we can calculate each of these eight points

So, for any given value for μ and ν ...

... we can evaluate the coordinate on the Coons patch:

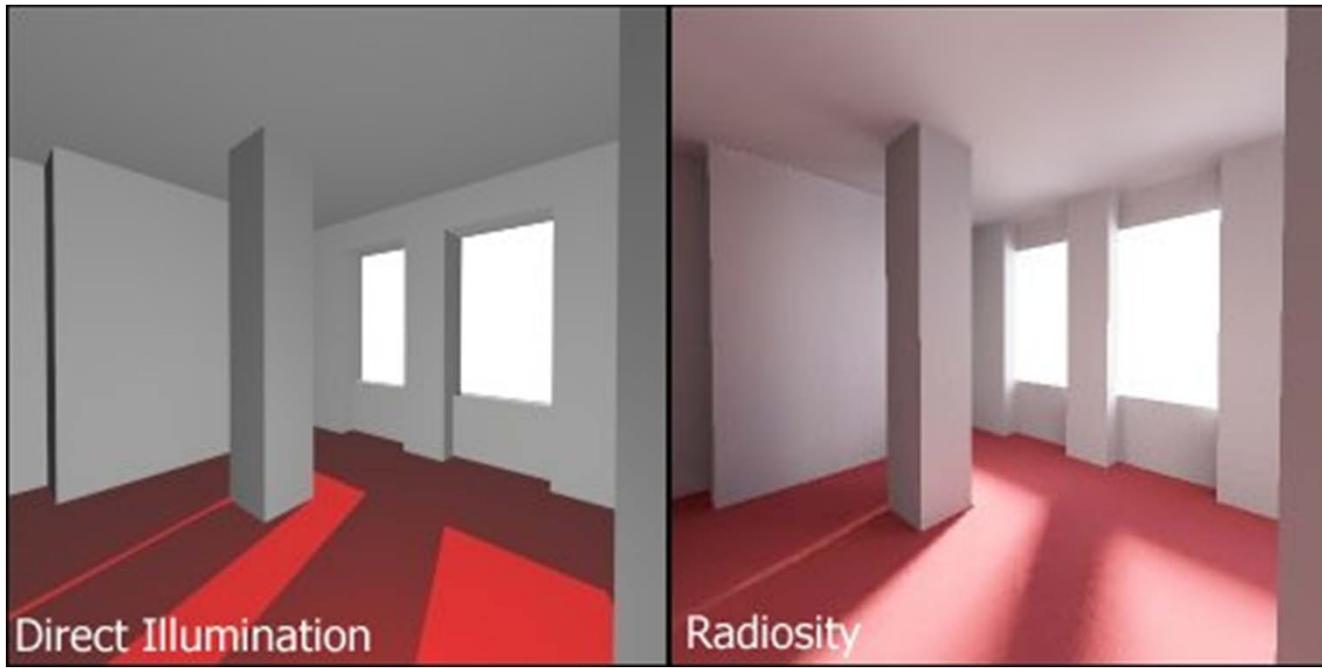
$$\begin{aligned}\mathbf{P}(\mu, \nu) = & \mathbf{P}(\mu, 0)(1 - \nu) + \mathbf{P}(\mu, 1)\nu + \\ & \mathbf{P}(0, \nu)(1 - \mu) + \mathbf{P}(1, \nu)\mu - \\ & \mathbf{P}(0, 1)(1 - \mu)\nu - \mathbf{P}(1, 0)\mu(1 - \nu) - \\ & \mathbf{P}(0, 0)(1 - \mu)(1 - \nu) - \mathbf{P}(1, 1)\mu\nu\end{aligned}$$

Interactive Computer Graphics: Lecture 14

Radiosity



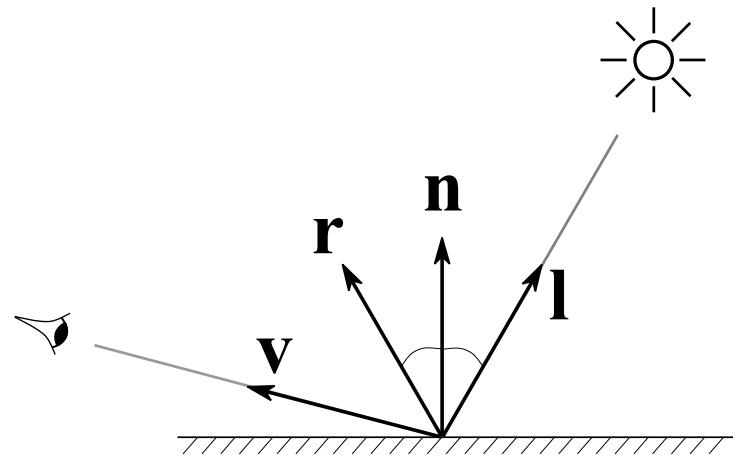
Direct Illumination



The reflectance equation

- Recall the reflectance equation: models reflected light from a surface:

$$I_{reflected} = k_a + k_d (\mathbf{n} \cdot \mathbf{l}) I_{incident} + k_s (\mathbf{r} \cdot \mathbf{v})^q I_{incident}$$

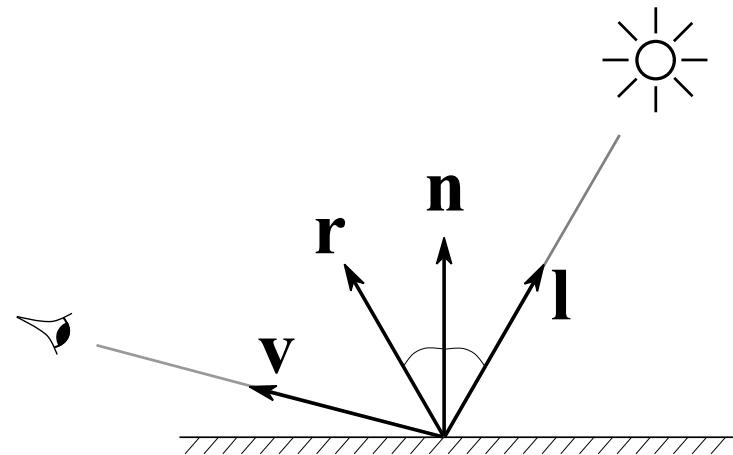


The reflectance equation

- Recall the reflectance equation: models reflected light from a surface:

$$I_{reflected} = k_a + k_d (\mathbf{n} \cdot \mathbf{l}) I_{incident} + k_s (\mathbf{r} \cdot \mathbf{v})^q I_{incident}$$

- $I_{incident}$: light intensity

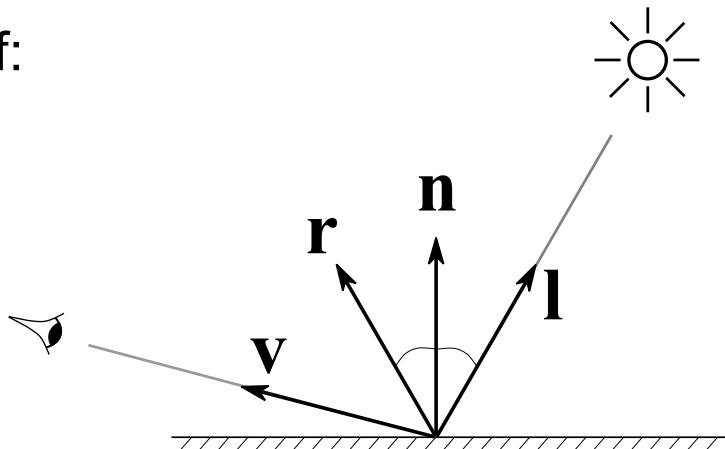


The reflectance equation

- Recall the reflectance equation: models reflected light from a surface:

$$I_{reflected} = k_a + k_d (\mathbf{n} \cdot \mathbf{l}) I_{incident} + k_s (\mathbf{r} \cdot \mathbf{v})^q I_{incident}$$

- $I_{incident}$: light intensity
- $k_{\{a,d,s\}}$: control amounts of:
 - k_a : ambient light
 - k_d : diffuse light
 - k_s, q : specular reflection



Lighting model for ray tracing

For ray tracing we assumed that there were a small number of point light sources.

However, according to the reflectance equation, every surface is reflecting light, and so should also be considered a light source.

So rather than use a constant for ambient light, shouldn't we sum the light received from all other surfaces in the scene?

Ambient light

- A better approximation to the reflectance equation is to make the ambient light term a function of the incident light as well

$$I_{reflected} = k_a I_{incident} + k_d (\mathbf{n} \cdot \mathbf{l}) I_{incident} + k_s (\mathbf{r} \cdot \mathbf{v})^q I_{incident}$$

- Or, more simply, to write (for a given viewpoint)

$$I_{reflected} = R I_{incident}$$

- where R is the (viewpoint dependent) reflectance function.

Radiosity

- Radiosity is defined as the energy per unit area leaving a surface.
- It is the sum of
 - the emitted energy per unit area (if any)
 - the reflected energy.
- For a small area of the surface (patch) dA (where the emitted energy can be regarded as constant) we have:

$$BdA = E dA + R I$$

- Notice that we now treat light sources as distributed

Radiosity

- Divide the scene into patches $i = 1, \dots, n$
- For the i -th patch, let:
 - B_i = total energy leaving the patch
 - E_i = total energy emitted by patch itself
 - R_i = reflectance value
 - I_i = incident light energy arriving at the patch
- With this notation, the above equation can be re-written

$$B_i = E_i + R_i I_i$$

- Radiosity computation a finite element method!

Collecting energy

We can estimate the incident energy for patch i as:

$$I_i = \sum_{j=1}^n B_j F_{ij}$$

where the sum is taken over all surface patches of the scene

The B_j 's in the sum represent the energy leaving all the other patches in the scene

F_{ij} is a constant that links surface patch i with patch j and is called the *form factor*

We can assume that $F_{ii} = 0$

Final formulation

- We can substitute this expression for incident light into the previous equation
- We obtain a discrete approximation for the energy leaving the i -th patch:

$$B_i = E_i + R_i I_i \quad \text{becomes} \quad B_i = E_i + R_i \sum_j B_j F_{ij}$$

- The form factors F_{ij} take into account
 - Patch areas
 - The angle at which they ‘face’ each other
- They control the amount of energy leaving patch j that reaches patch i

In matrix form

- Re-write the equation for the i -th patch

$$B_i - \sum_j R_i B_j F_{ij} = E_i$$

- Joining the equations for all patches, we can formulate the matrix equation:

$$\begin{pmatrix} 1 & -R_1 F_{12} & -R_1 F_{13} & \cdot & \cdot & -R_1 F_{1n} \\ -R_2 F_{21} & 1 & -R_2 F_{23} & \cdot & \cdot & -R_2 F_{2n} \\ -R_3 F_{31} & -R_3 F_{32} & 1 & \cdot & \cdot & -R_3 F_{3n} \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ -R_n F_{n1} & -R_n F_{n2} & -R_n F_{n3} & \cdot & \cdot & 1 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{pmatrix}$$

In matrix form

$$\begin{pmatrix} 1 & -R_1 F_{12} & -R_1 F_{13} & \cdot & \cdot & -R_1 F_{1n} \\ -R_2 F_{21} & 1 & -R_2 F_{23} & \cdot & \cdot & -R_2 F_{2n} \\ -R_3 F_{31} & -R_3 F_{32} & 1 & \cdot & \cdot & -R_3 F_{3n} \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ -R_n F_{n1} & -R_n F_{n2} & -R_n F_{n3} & \cdot & \cdot & 1 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{pmatrix}$$

- If we can solve this for every B_i then we will be able to render each patch with a correct light model.
- However, this is not so easy to do since
 - the form factors need to be found
 - the full reflectance equation is insolvable
 - the matrix is big - minimum 10000 by 10000

Wavelengths

- The radiosity values are wavelength dependent, hence we will need to compute a radiosity value for R , G and B .
- Each patch will require a separate set of parameters for R , G and B .
- The three radiosity values are the values that the rendered pixels will receive.

Back to the reflectance function

$$I_{reflected} = k_a I_{incident} + k_d (\mathbf{n} \cdot \mathbf{l}) I_{incident} + k_s (\mathbf{r} \cdot \mathbf{v})^q I_{incident}$$

Note that the specular term depends on the relative positions of the viewpoint and each light source \mathbf{v} .

But now, every patch is a light source!

Specular reflections

- Moreover our light sources are no longer points, so we need to collect the incident light in a specular cone to determine the specular reflection.
- This is computationally infeasible.
- We will consider only diffuse radiosity.

Patching Problems

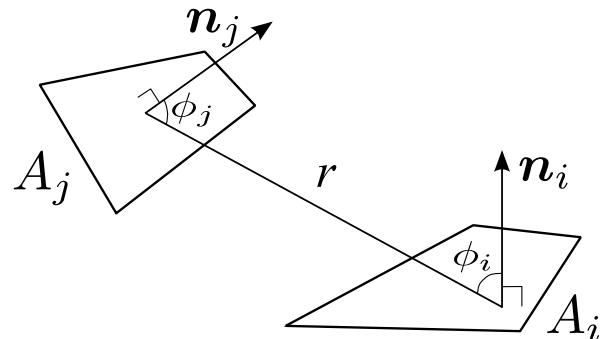
- We need to divide our graphics scene into patches for computing the radiosity.
- For small polygons we can perhaps use the polygon map, but for large polygons we need to subdivide them.
- Since the emitted light will not be constant across a large polygon we will see the subdivisions

Large Polygons

- Each patch will have a different but constant illumination.
- Thus we will see the patch boundaries unless either:
 - Patches project to (sub) pixel size or
 - We smooth the results (eg by interpolation)

Form Factors

- The form factors couple every pair of patches, determining the proportion of radiated energy from one that strikes the other.

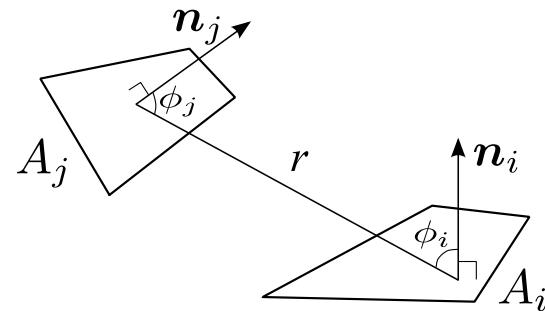


$$F_{ij} = \frac{1}{|A_i|} \int_{A_i} \int_{A_j} \frac{\cos \phi_i \cos \phi_j}{\pi r^2} dA_j dA_i$$

Form Factors - the definition

- The cos terms compute the projection of each patch in the direction to the other.
- If the patches are in the same plane, facing the same way, there is no coupling. If they directly face each other they are maximally coupled.

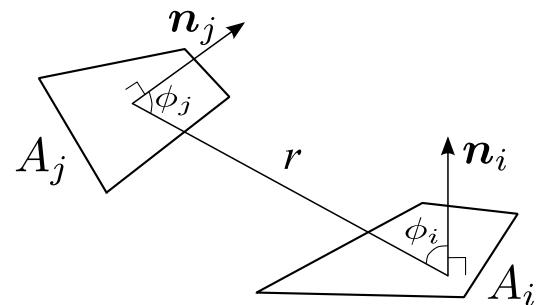
$$F_{ij} = \frac{1}{|A_i|} \int_{A_i} \int_{A_j} \frac{\cos \phi_i \cos \phi_j}{\pi r^2} dA_j dA_i$$



Form Factors - simplifying the computation

- The equation can be simplified if we assume that A_i is small compared with r .
- If this is the case, then we can treat the inner integral as constant over the surface of A_i .

$$F_{ij} = \frac{1}{|A_i|} \int_{A_i} \int_{A_j} \frac{\cos \phi_i \cos \phi_j}{\pi r^2} dA_j dA_i$$



Simplifying form factors

- With this assumption the outer integral evaluates to $|A_i|$ (i.e. the area of A_i).
- Hence we can write the integral as:

$$F_{ij} = \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r^2} dA_j$$

Further simplifying

We assumed that the radius is large compared with patch A_i . Should also be reasonable to assume it is large compared to the size of A_j .

Hence the integrand of

$$F_{ij} = \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r^2} dA_j$$

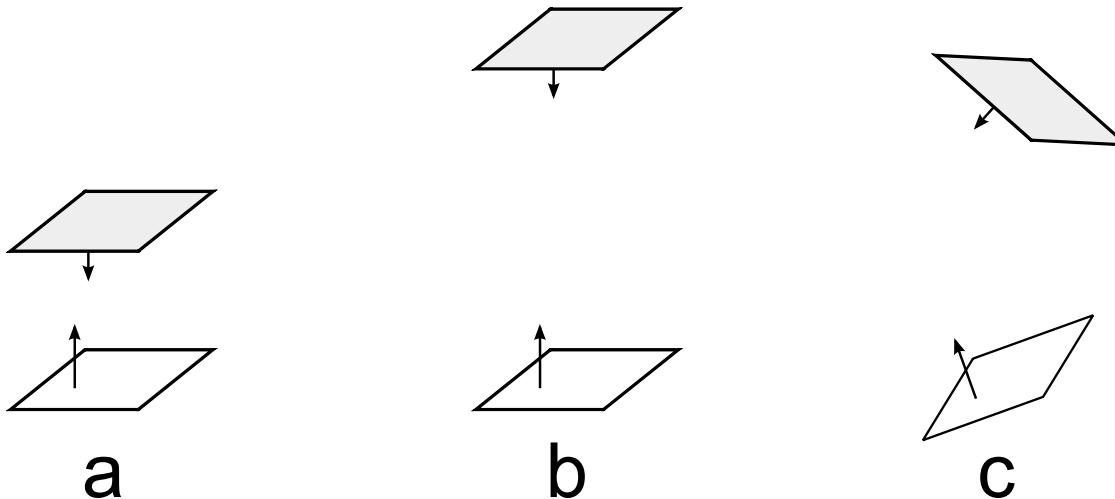
can similarly be considered constant over A_j

So we get the approximation

$$F_{ij} = \frac{\cos\phi_i \cos\phi_j |A_j|}{\pi r^2}$$

Form factors

$$F_{ij} = \frac{\cos \phi_i \cos \phi_j |A_j|}{\pi r^2}$$



(a) Big form factor perhaps 0.5

(b) Further away, smaller form factor, perhaps 0.25

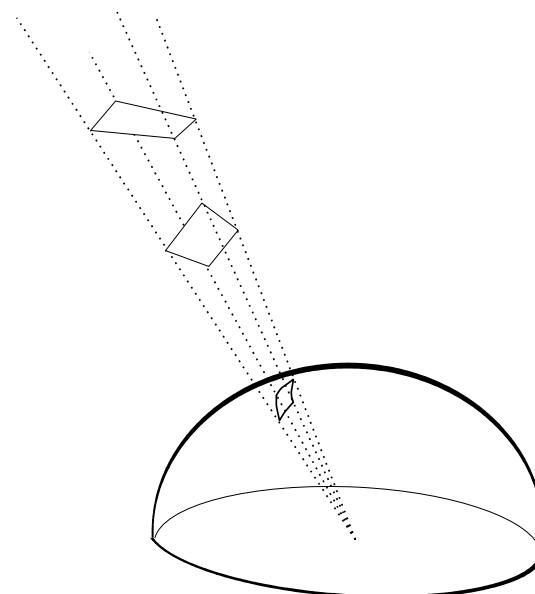
(c) Not really facing each other, even smaller form factor perhaps 0.1

The Hemicube method

Using a bounding hemisphere it can be shown that all patches that project onto the same area of the hemisphere have the same form factor

Direct computation of the approximate form factor equation for every pair of patches will be expensive to compute.

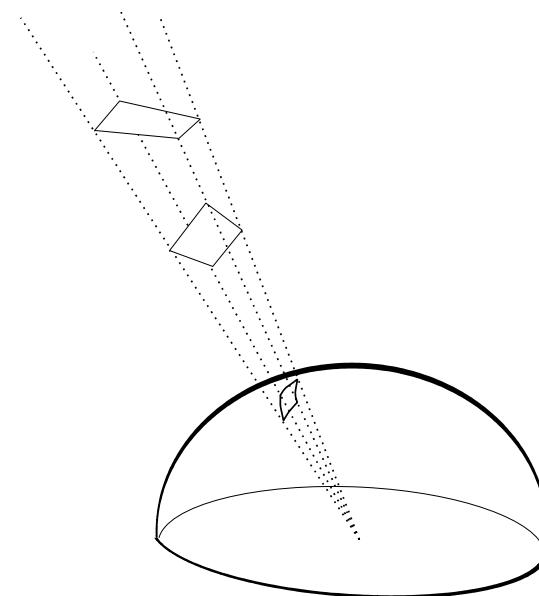
So an approximate computation method based on this observation (the hemi-cube) was developed.



The Hemicube method

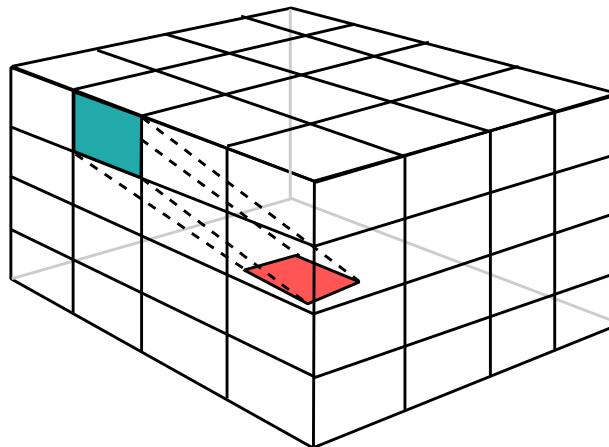
So, all patches that project onto the same area of a surrounding hemicube have approximately the same form factor.

The hemicube is preferred to the hemisphere since computing intersections with planes is computationally less demanding



Delta form factors

The hemicube is divided into small pixel areas and form factors are computed for each.



The resulting form factors can be used for every patch in the scene. We just ‘place’ the same hemicube over each.

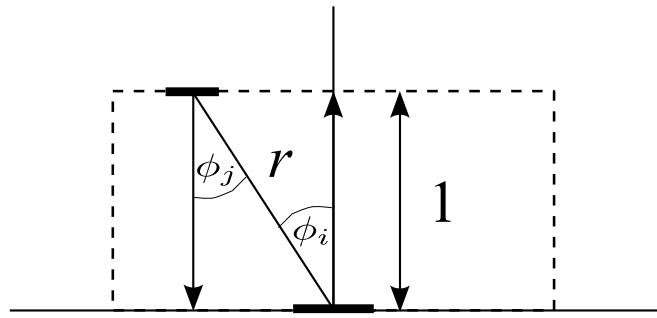
Delta form factors

- If the area of a hemicube pixel is $|A|$, its form factor is:

$$\frac{\cos \phi_i \cos \phi_j |A|}{\pi r^2}$$

- These delta form factors can be computed and stored in a look up table.
- They can then be applied to every patch without the need for further form factor calculations.

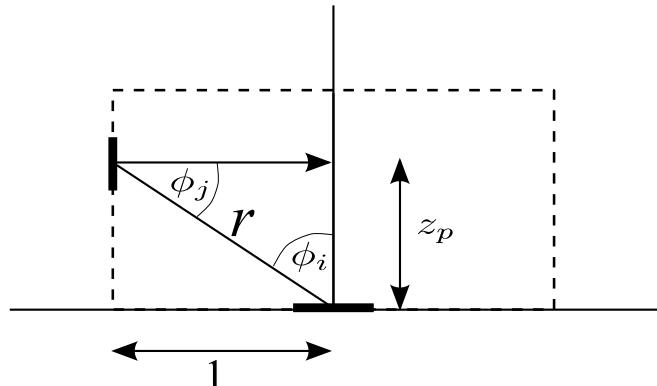
Computing the delta form factors



For a top face we have:

$$\cos \phi_i = \cos \phi_j = \frac{1}{r}$$

so the form factor is $\frac{|A|}{\pi r^4}$



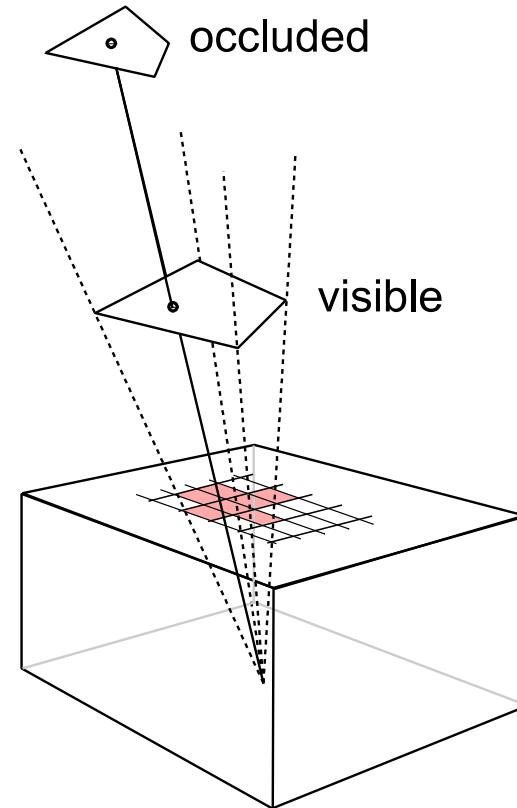
For a side face we have:

$$\cos \phi_i = \frac{1}{r} \quad \cos \phi_j = \frac{z_p}{r}$$

so the form factor is $\frac{z_p |A|}{\pi r^4}$

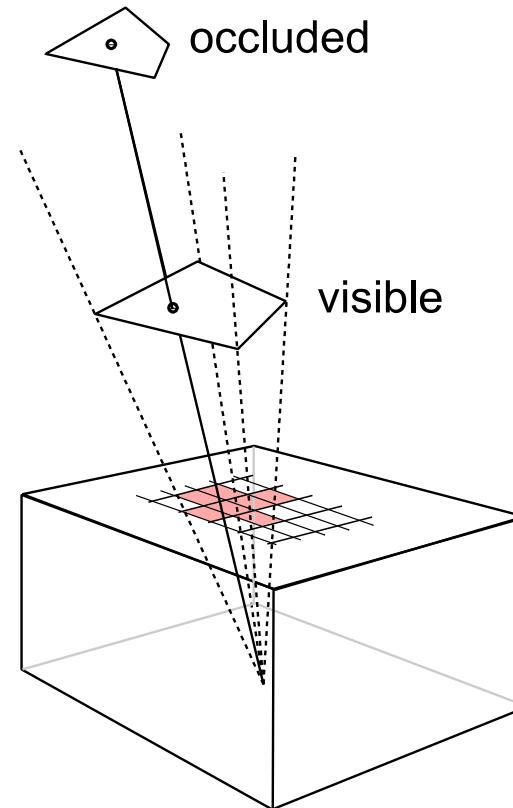
Projection of patches onto the hemicube

- We now need to know which patch is visible from each hemicube pixel.
- This could be done by ray tracing (casting), or projection.
- Ray tracing neatly solves the occlusion problem.
- Projection would require z-buffering.



Sum the pixels per patch

- Notice that all we need to determine is the nearest visible patch at each hemicube pixel.
- Once this is found we calculate the form factor for each patch by summing the delta form factors of the hemicube pixels to which it projects.





Summary of Radiosity method

1. Divide the graphics world into discrete patches
2. Compute form factors by the hemicube method
3. Solve the matrix equation for the radiosity of each patch.
4. Average the radiosity values at the corners of each patch
5. Compute a texture map of each point or render directly
 - https://www.youtube.com/watch?v=0-0aMo_qkGo

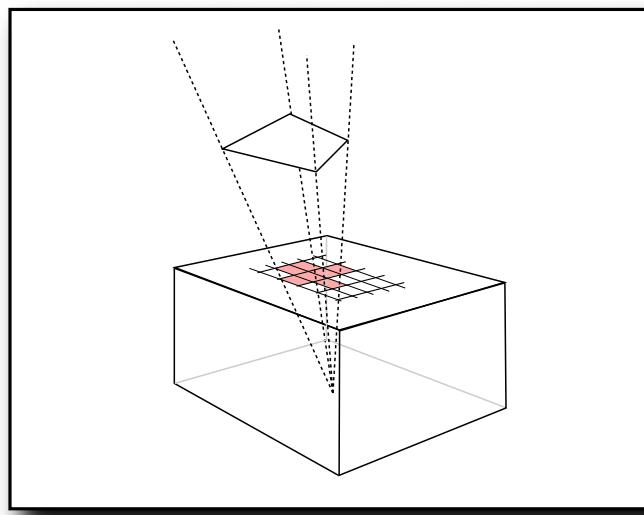
Summary of Radiosity method

1. Divide the graphics world into discrete patches
Meshing strategies, meshing errors
2. Compute form factors by the hemicube method
Alias errors
3. Solve the matrix equation for the radiosity of each patch.
Computational strategies
4. Average the radiosity values at the corners of each patch
Interpolation approximations
5. Compute a texture map of each point or render directly
At least this stage is relatively easy

Now read on ...

Alias Errors

- Computation of the form factors will involve alias errors.
- Equivalent to errors in texture mapping, due to discrete sampling of a continuous environment.
- However, as the alias errors are averaged over a large number of pixels the errors will not be significant.



Form Factor reciprocity

- Form factors have a reciprocal relationship:

$$F_{ij} = \frac{\cos \phi_i \cos \phi_j |A_j|}{\pi r^2} \quad F_{ji} = \frac{\cos \phi_i \cos \phi_j |A_i|}{\pi r^2}$$

$$\Rightarrow \quad F_{ji} = \frac{F_{ij}|A_i|}{|A_j|}$$

- So form factors for only half the patches need be computed.

The number of form factors

There will be a large number of form factors:

For 60,000 patches, there are 3,600,000,000 form factors.

We only need store half of these (reciprocity), but we will need four bytes for each, hence 7 GB are needed.

As many of them are zero we can save space by using an indexing scheme (e.g. use one bit per form factor, bit = 0 implies form factor zero and not stored)

Inverting the matrix

- Inverting the matrix can be done by the Gauss Seidel method:

$$\begin{pmatrix} 1 & -R_1F_{12} & -R_1F_{13} & \dots & -R_1F_{1n} \\ -R_2F_{21} & 1 & -R_2F_{23} & \dots & -R_2F_{2n} \\ -R_3F_{31} & -R_3F_{32} & 1 & \dots & -R_3F_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -R_nF_{n1} & -R_nF_{n2} & -R_nF_{n3} & \dots & 1 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{pmatrix}$$

- Each row of the matrix gives an equation of the form:

$$B_i = E_i + R_i \sum_j B_j F_{ij}$$

Inverting the matrix

- The Gauss Seidel method is iterative and uses the equation of each row
- Given:

$$B_i = E_i + R_i \sum_j B_j F_{ij}$$

- We use the iteration:

$$B_i^k = E_i + R_i \sum_j B_j^{k-1} F_{ij}$$

- To give successive estimates B_i^0, B_i^1, \dots
- Can set initial values $B_i^0 = 0$

Gauss-Seidel method for solving equations

- Given a scene with three patches, we can write the iterations as *update* equations:

$$B_0 \leftarrow E_0 + R_0(F_{01} B_1 + F_{02} B_2)$$

$$B_1 \leftarrow E_1 + R_1(F_{10} B_0 + F_{12} B_2)$$

$$B_2 \leftarrow E_2 + R_2(F_{20} B_0 + F_{21} B_1)$$

- Assume we know numeric the values for $E_0, E_1, E_2, R_0, R1, R_2, F_{01}, F_{02}, F_{10}, F_{12}, F_{20}, F_{21}$:

$$B_0 \leftarrow 0 + 0.5(0.2 B_1 + 0.1 B_2)$$

$$B_1 \leftarrow 5 + 0.5(0.2 B_0 + 0.3 B_2)$$

$$B_2 \leftarrow 0 + 0.2(0.1 B_0 + 0.3 B_1)$$

Gauss-Seidel method for solving equations

Simplify:

$$B_0 \leftarrow 0.1 B_1 + 0.05 B_2$$

$$B_1 \leftarrow 5 + 0.1 B_0 + 0.15 B_2$$

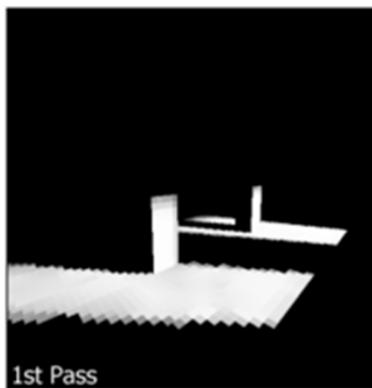
$$B_2 \leftarrow 0.02 B_0 + 0.06 B_1$$

Step	B_0	B_1	B_2
0	0	0	0
1	0	5	0
2	0.5	5	0.3
3	0.515	5.095	0.31
:	:	:	:

The process eventually converges to 0.53, 5.07 and 0.31 in this case

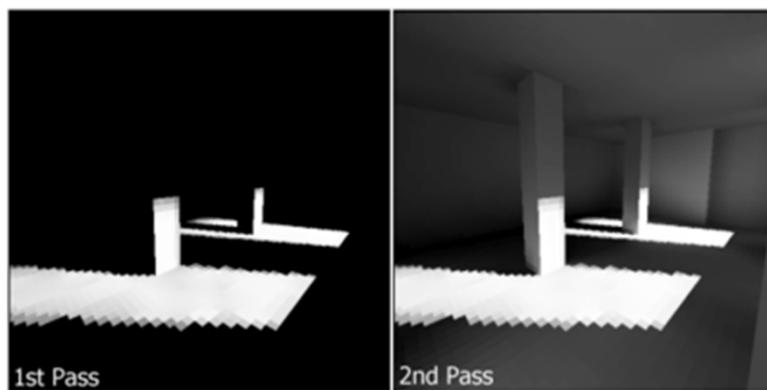
Gauss-Seidel method for solving equations

- The Gauss-Seidel method is stable and converges
- It can be shown that the radiosity matrix is ‘diagonally dominant’ (a sufficient condition to guarantee convergence).
- At the first iteration the emitted light energy is distributed to those patches that are illuminated
- In the next cycle, those patches illuminate others and so on.
- The image will start dark and progressively illuminate as the iteration proceeds



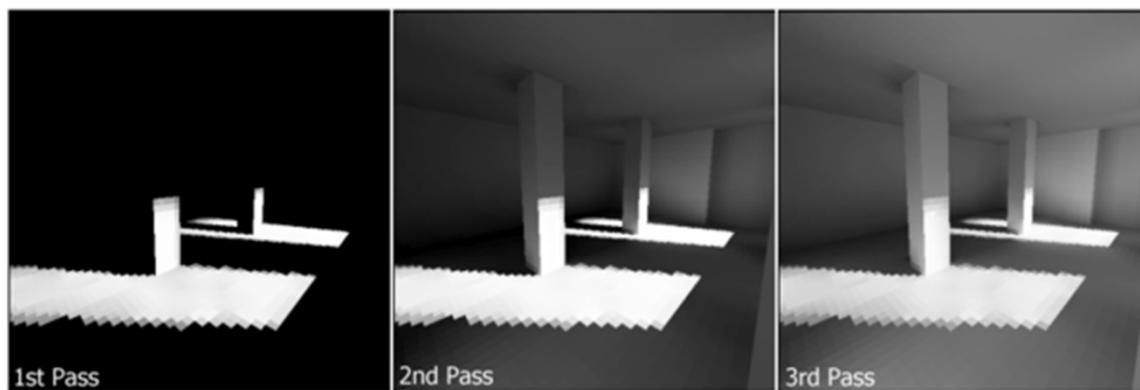
Gauss-Seidel method for solving equations

- The Gauss-Seidel method is stable and converges
- It can be shown that the radiosity matrix is ‘diagonally dominant’ (a sufficient condition to guarantee convergence).
- At the first iteration the emitted light energy is distributed to those patches that are illuminated
- In the next cycle, those patches illuminate others and so on.
- The image will start dark and progressively illuminate as the iteration proceeds



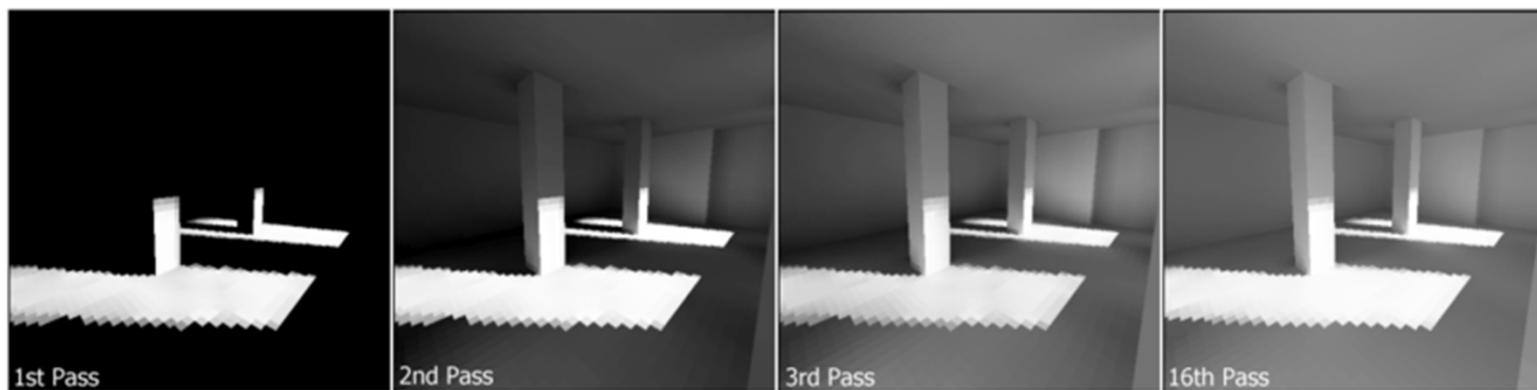
Gauss-Seidel method for solving equations

- The Gauss-Seidel method is stable and converges
- It can be shown that the radiosity matrix is ‘diagonally dominant’ (a sufficient condition to guarantee convergence).
- At the first iteration the emitted light energy is distributed to those patches that are illuminated
- In the next cycle, those patches illuminate others and so on.
- The image will start dark and progressively illuminate as the iteration proceeds



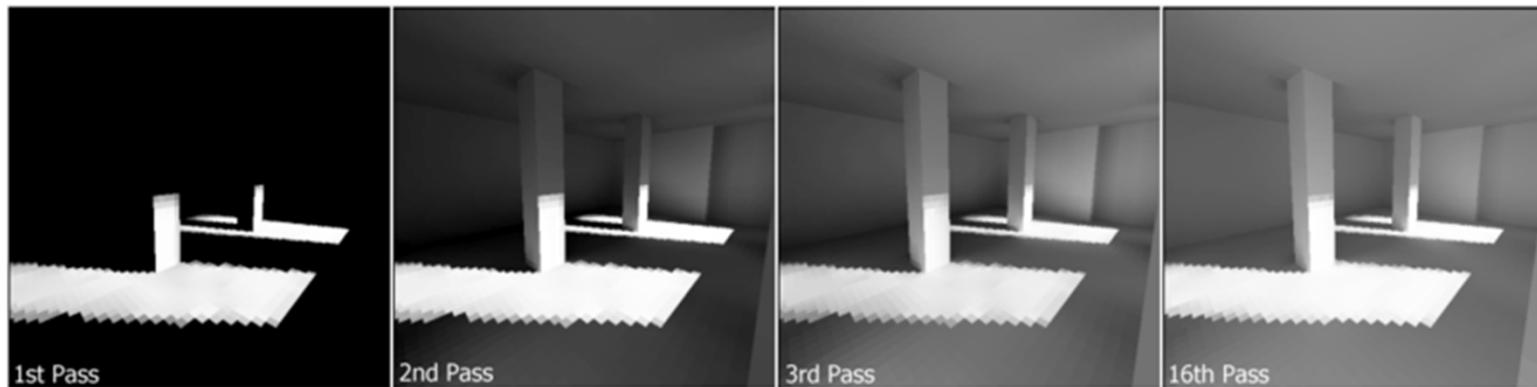
Gauss-Seidel method for solving equations

- The Gauss-Seidel method is stable and converges
- It can be shown that the radiosity matrix is ‘diagonally dominant’ (a sufficient condition to guarantee convergence).
- At the first iteration the emitted light energy is distributed to those patches that are illuminated
- In the next cycle, those patches illuminate others and so on.
- The image will start dark and progressively illuminate as the iteration proceeds



Progressive Refinement

- The nature of the Gauss Seidel method allows a partial solution to be rendered as the computation proceeds.
- Without altering the method we could render the image after each iteration, allowing the designer to stop the process and make corrections quickly.
- This may be particularly important if the scene is so large that we need to re-calculate the form factors every time we need them.



Inverting the matrix

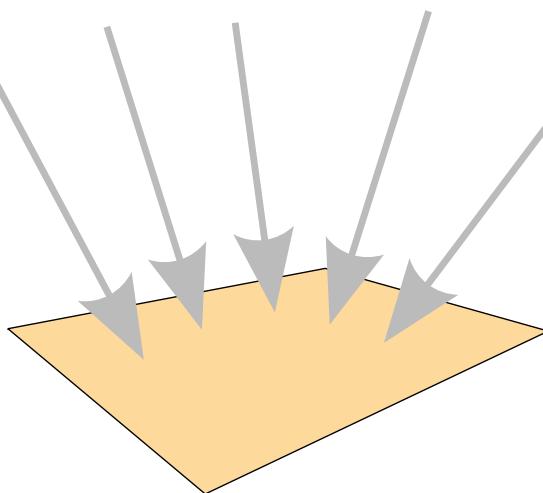
- The Gauss Seidel inversion can be modified to make it faster by making use of the fact that it is essentially distributing energy around the scene.
- The method is based on the idea of “shooting and gathering”, and also provides visual enhancement of the partial solution.

Gathering Patches

- Evaluation of one B_i value using one line of the matrix:

$$B_i^k = E_i + R_i \sum_j B_j^{k-1} F_{ij}$$

is the process of *gathering*.

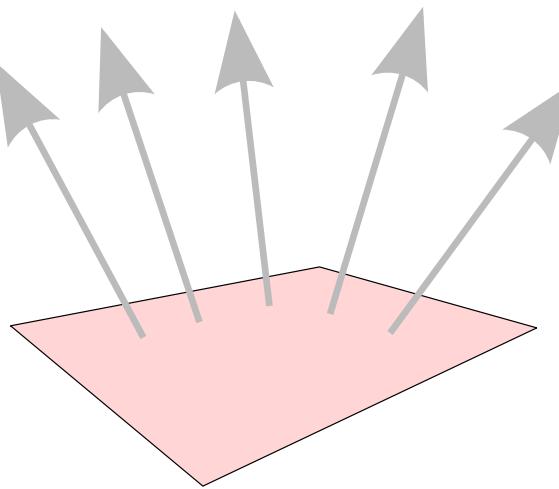


Shooting Patches

- Suppose in an iteration B_i changes by ΔB_i
- The change to every other patch can be found using:

$$B_j^k = B_j^{k-1} + R_j F_{ji} \Delta B_i^{k-1}$$

- This is the process of shooting, and is evaluating the matrix column wise.



Evaluation Order

- The idea of gathering and shooting allows us to choose an evaluation order that ensures fastest convergence.
- The patches with the largest change ΔB (called the unshot radiosity) are evaluated first.
- The process starts by initialising all unshot radiosity to zero except emitting patches where $\Delta B_i = E_i$

Processing unshot radiosity

- Choose patch with largest unshot radiosity ΔB_i

Patch	Unshot radiosity
B_0	ΔB_0
B_1	ΔB_1
B_2	ΔB_2
:	:
B_N	ΔB_N

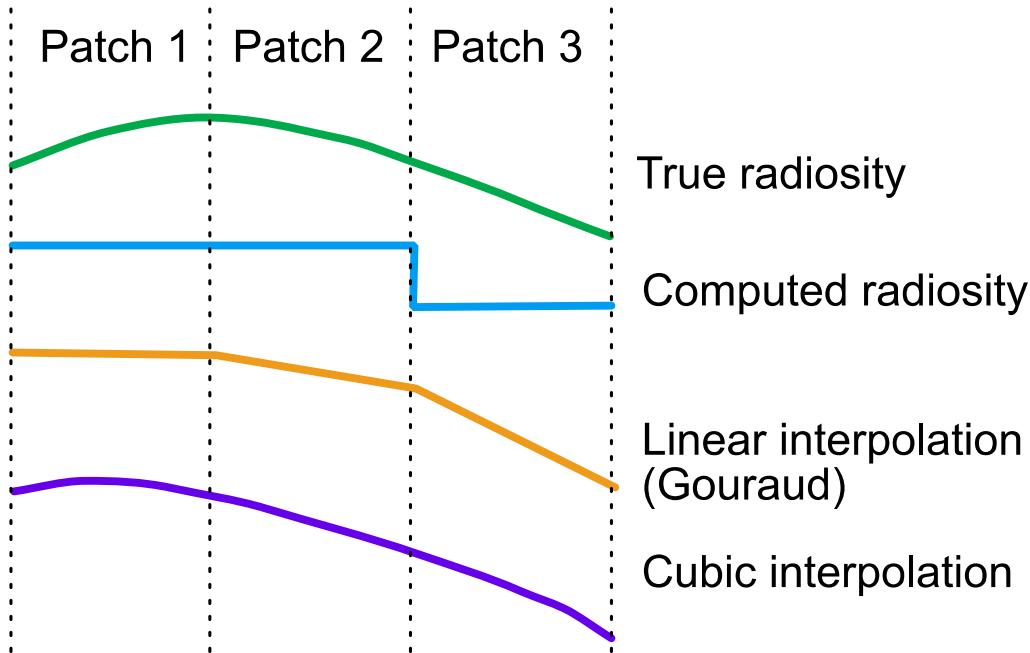
- Shoot the radiosity for the chosen patch, i.e. for all other patches update

$$\Delta B_j = R_j F_{ji} \Delta B_i$$

- and add it to their radiosity
- Set $\Delta B_i = 0$ and iterate

Interpolation Strategies

- Visual artefacts do occur with interpolation strategies, but may not be significant for small patches

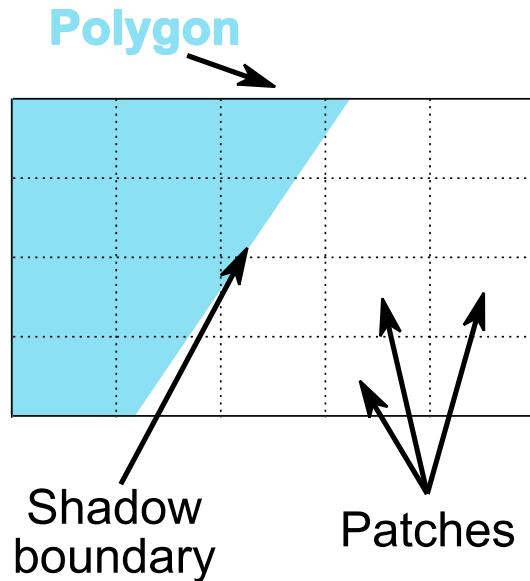


Meshing

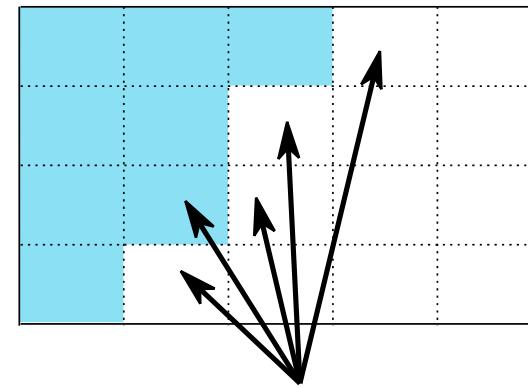
- Meshing is the process of dividing the scene into patches.
- Meshing artifacts are scene dependent.
- The most obvious are called D^0 artifacts, caused by discontinuities in the radiosity function

D^0 artifacts

- Discontinuities in the radiosity are exacerbated by bad patching



Computed radiosity



Incorrectly rendered patches
(even after interpolation)

Discontinuity Meshing (a-priori)

- The idea is to compute discontinuities in advance:
 - Object boundaries
 - Albedo/reflectivity discontinuities
 - Shadows (requires pre-processing by ray tracing)
 - etc.
- Place patches in advance so that they align with the discontinuities
- Then calculate radiosity

Adaptive Meshing (a posteriori)

The idea is to re-compute the mesh during the radiosity calculation

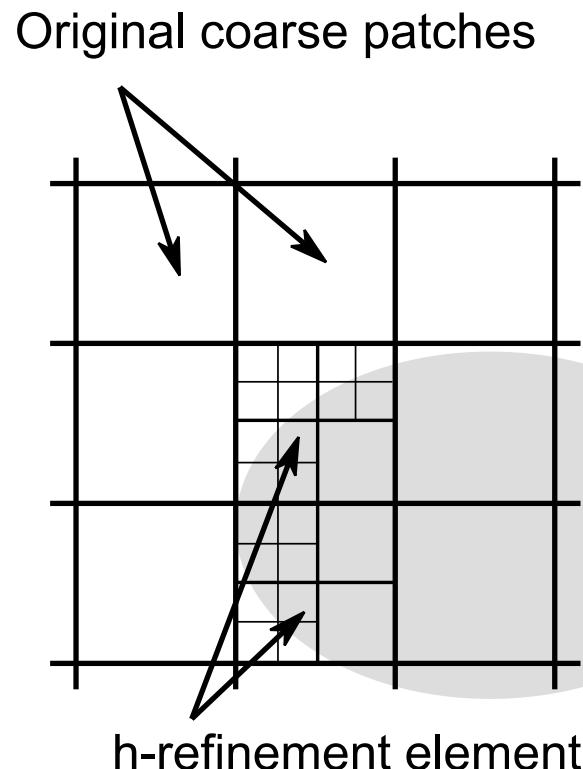
If two adjacent patches have a strong discontinuity in radiosity value, we can

1. Put more patches (elements) into that area, or
2. Move the mesh boundary to coincide with the greatest change

Subdivision of Patches (r/h-refinement)

Compute the radiosity at the vertices of the coarse grid.

Subdivide into elements if the discontinuities exceed a threshold





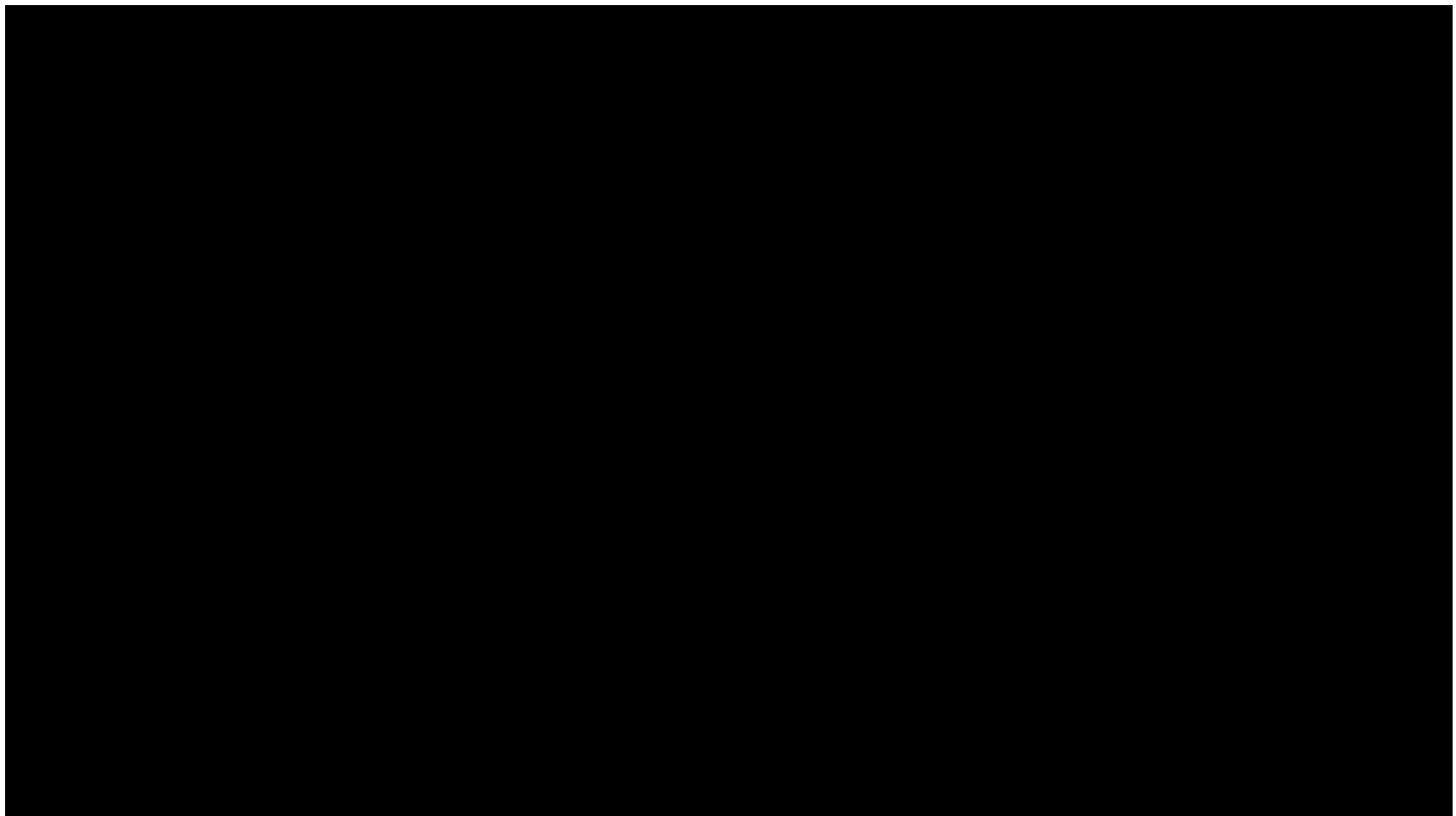
By Dhatfield - Own work, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=4303243>

Interactive Computer Graphics: Lecture 15

Special effects

Some slides adopted from
Daniel Wagner, Michael Kenzel, TU-Graz

Motivation



<https://www.youtube.com/watch?v=MnQLjZSX7xM>

Motivation

- Add special effects in image space after rendering
 - Independent of geometric scene complexity
 - Often uses image processing techniques
- Irregular objects: billboards, particle systems
- Distance to camera: fog, depth of field
- Camera exposure: motion blur
- Camera aperture: bokeh, lens flare
- Non-photorealistic rendering

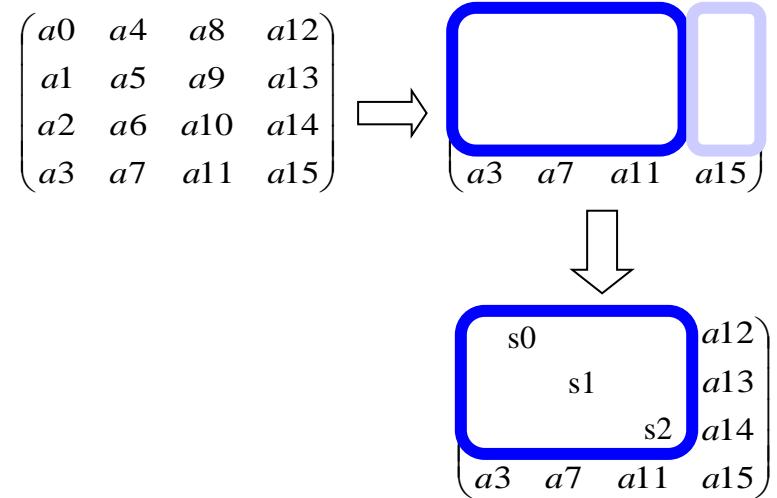
Billboards

- Prerequisite for many effects
- Synonyms: impostors, sprites
- Textured rectangles which either
 - Face the viewer, or
 - Are aligned with some axis
- Can be used in large quantities
 - Simple, only 2 textured triangles
- Low memory footprint
- Rendered using graphics hardware
- Only look good at a distance or when small
- Example: clouds in a game



Billboards

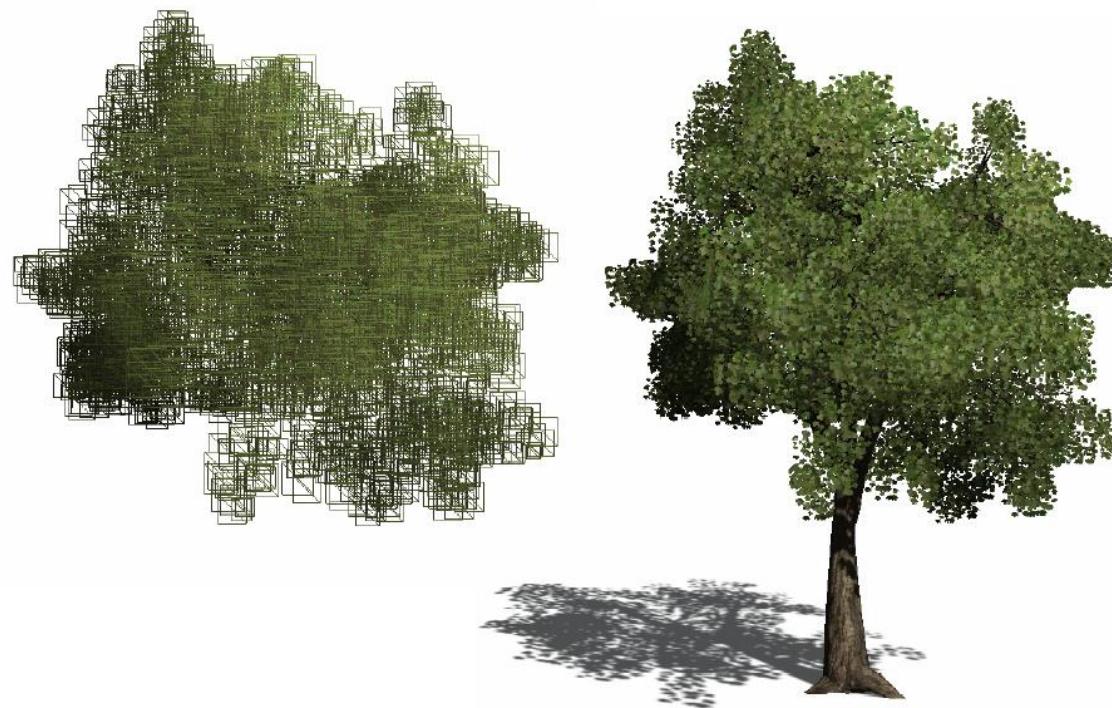
- How: modify the ModelView matrix
(remove rotation)



- Maintain scale!
- Result: BB will appear at the right position and distance, but will face camera

Billboards

- A set of billboards with different size/orientation
- Created procedurally (from 3D model or rule set)
- Can be animated by physical simulation



Particle systems

- a system to control collection of a number of individual elements over time (points, line, triangle or Sprit texture), which act independently but share some common attributes:
 - position (3D)
 - velocity (vector: speed and direction)
 - color +(transparency)
 - lifetime
 - size, shape

Particle systems

- The first CG paper about particle systems by William T. Reeves: Particle Systems A Technique for Modeling a Class of Fuzzy Objects. Computer Graphics, vol. 17-3, July 1983
- in “Star Trek II: The Wrath of Kahn” 1983

Particle systems



- “Star Trek II: The Wrath of Kahn” 1983

Particle systems

- Modeling of natural phenomena:
 - Rain, snow, clouds
 - Explosions, fireworks, smoke, fire
 - Sprays, waterfalls



Particle systems

- All particles of a system use the same update method (share the same properties)
- The particle system handles
 - Initializing
 - Updating
 - Randomness
 - Rendering
- Particle parameters change
 - Location, Speed, lifetime
- Particles are emitted somewhere and “die” after some time

```
struct particle
{
    float t;           // life time
    float v;           // speed
    float x, y, z;    // coordinates
    float xd, yd, zd; // direction
    float alpha;       // fade alpha
};
```

Particle systems / Physics

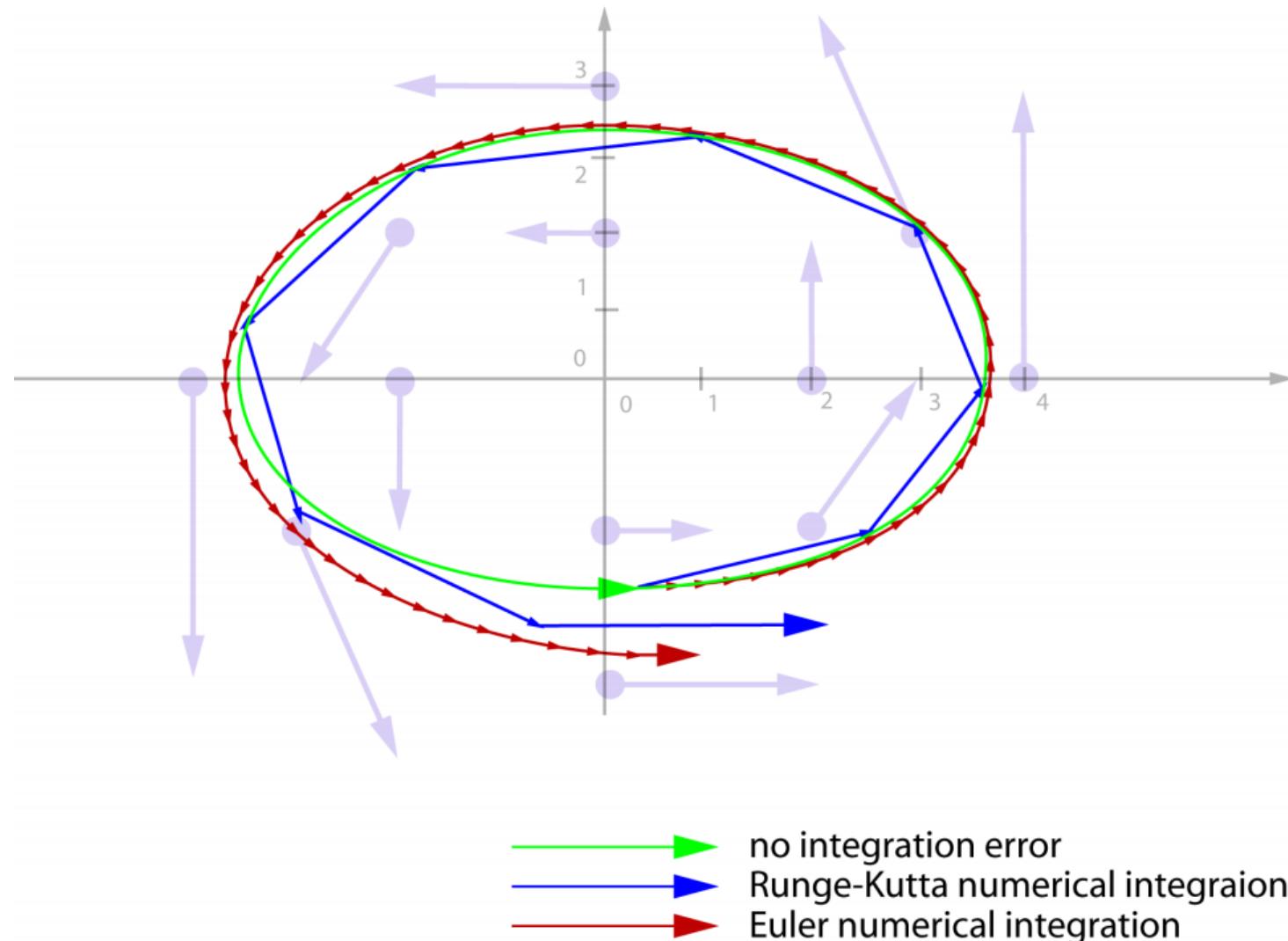
- Motion may be controlled by external forces
 - E.g., gravity, collision, vector field
- Particles can interfere with other particles
- Causes a more entropic movement, e.g., sprays of liquids



Particle systems / Physics



Particle systems / Integration



Particle systems / Integration / Euler

- the continuous movement of a massless particle under the influence of an evenly varying vector field

$$\frac{\partial x}{\partial t} = v(x(t), \tau), \quad x(t_0) = x_0, \quad x : \mathbb{R} \rightarrow \mathbb{R}^n$$

- v is a sampled vector field whose sampled values depend on the current position of a particle $x(t)$
- The simplest form to solve the initial value problem is the standard explicit Euler-approach
- Step size $\Delta t = h > 0$
$$t_{k+1} = t_k + h,$$

$$x_{k+1} = x_k + h v(x_k, t_k, \tau).$$
- accuracy depends on the selected step size Δt

Particle systems / Integration / Runge-Kutta

- Reduce integration error or computational effort with intermediate steps:

$$x_{k+1} = x_k + h \sum_{j=1}^n b_j c_j,$$

- With coefficients b_j and intermediate steps c_j . Each c_j is a basic Euler integration step. E.g., $n = 4$ (Runge-Kutta fourth order, RK4)

$$x_{k+1} = x_k + \frac{h}{6}(c_1 + 2c_2 + 2c_3 + c_4), \text{ where}$$

$$c_1 = v(x_k, t_k, \tau),$$

$$c_2 = v\left(x_k + \frac{h}{2}c_1, t_k + \frac{h}{2}, \tau\right),$$

$$c_3 = v\left(x_k + \frac{h}{2}c_2, t_k + \frac{h}{2}, \tau\right) \text{ and}$$

$$c_4 = v(x_k + hc_3, t_k + h, \tau).$$

Particle systems

- Interactive animation:

<http://demonstrations.wolfram.com/UnderstandingRungeKutta/>

Fog

- Atmospheric effect (scattering of light)
 - Stylistic element
 - Depth cue
 - Hide artifacts
 - Limited viewing range/clipping at far plane
 - Billboard updates
 - ...
- Fog intensity scales with distance to camera
 - Distance Fog

Fog

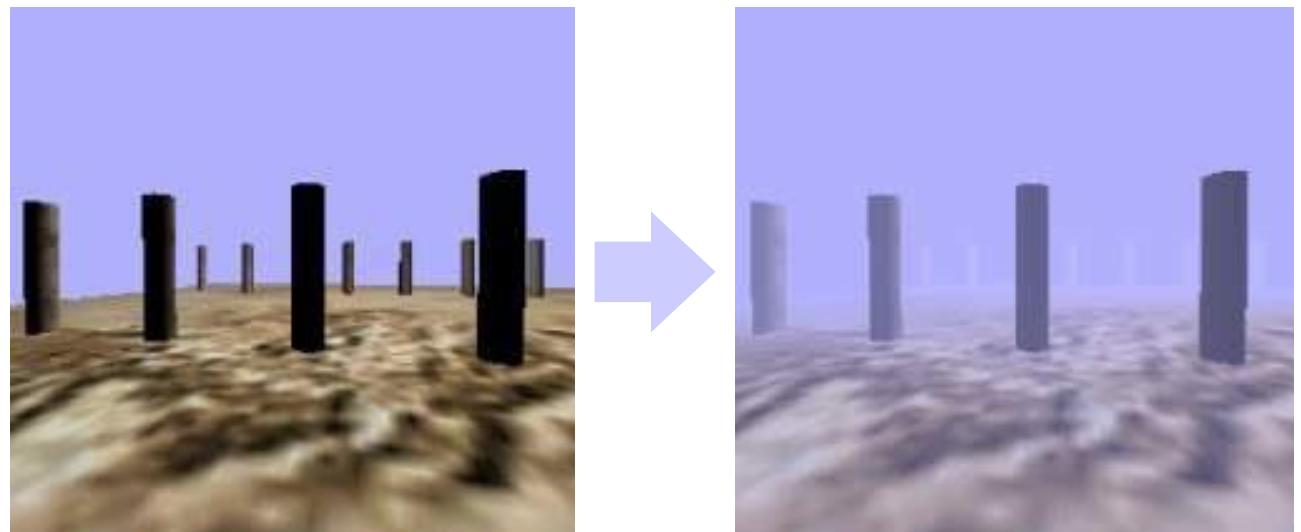
- Blend surface color with fog color

$$\mathbf{c} = f\mathbf{c}_s + (1 - f)\mathbf{c}_f$$

\mathbf{c}_s surface color

\mathbf{c}_f fog color

f fog factor



Fog

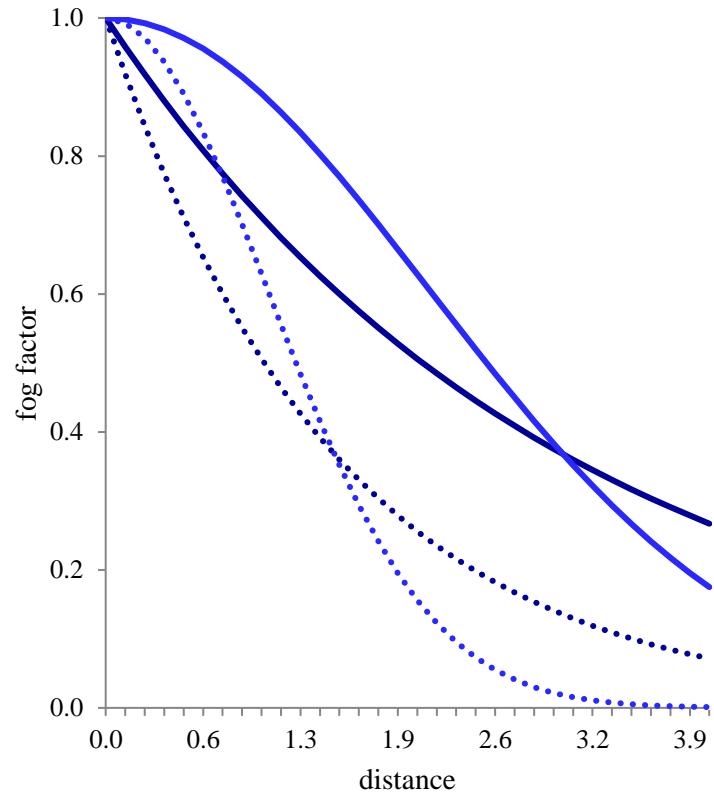
- Linear fog:
- Exponential fog:
- Squared exponential fog:

$$f = \frac{d_{end} - d}{d_{end} - d_{start}}$$

$$f = e^{-d_f \cdot d}$$

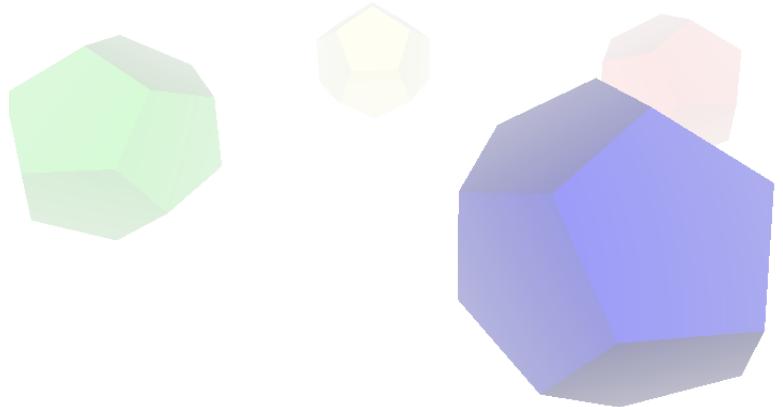
d fragment distance
 d_{start} fog start
 d_{end} fog end
 d_f fog density

linear
— exp $d=0.33$
···· exp $d=0.66$
— exp2 $d=0.33$
···· exp2 $d=0.66$



Fog

```
#version 330
#include <framework/utils/GLSL/camera>
uniform vec3 c_d;
uniform vec3 c_f;
uniform float d_f;
in vec3 p;
in vec3 normal;
layout(location = 0) out vec4 color;
void main()
{
    vec3 v = camera.position - p;
    float d = length(v);
    vec3 c_s = ...;
    float f = exp(-d_f * d);
    color = vec4(f * c_s + (1.0f - f) * c_f, 1.0f);
}
```

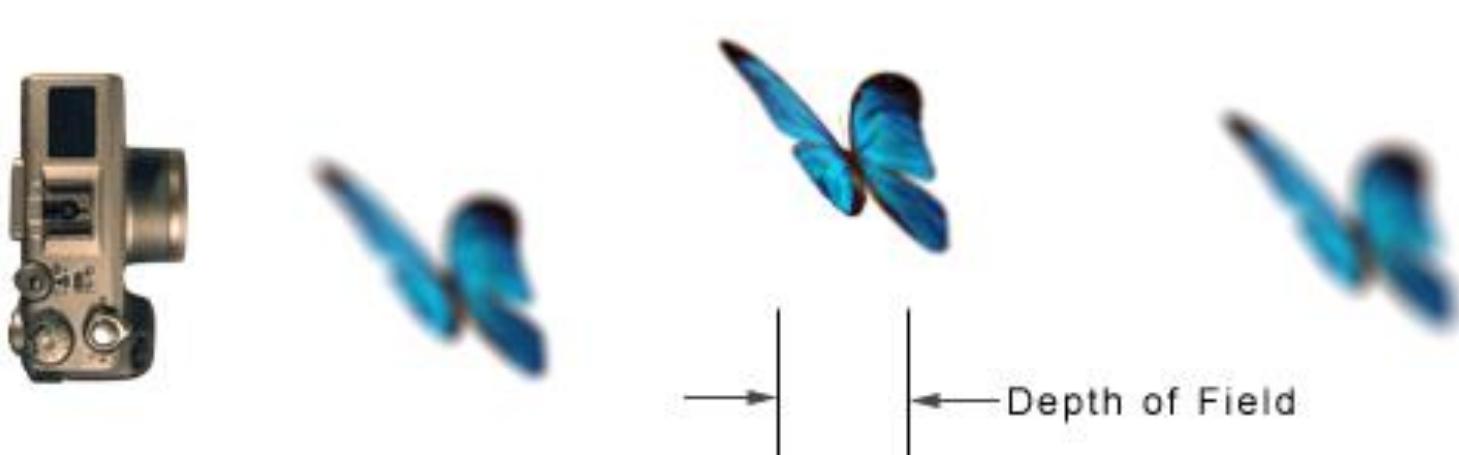


Post Processing Effects

1. Render scene into textures
 - Color
 - Depth
 - ...
2. Render screen-filling primitives
 - Fragment shader samples rendered textures
 - Can implement
 - Image filters
 - Color transformations
 - ...

Depth of Field

- Simulate camera property: lens can only focus on one depth level
- Objects around that depth level appear sharp
- Rest is blurred, depending on distance to focal plane



Depth of Field

- Guide the user's attention towards something



Depth of Field

- Effect does not occur with small apertures
- CG mostly uses pinhole cameras
 - Infinitely small aperture
- Simulating depth of field (DoF):
 - Adapt camera model
 - Not possible using standard OpenGL pipeline
 - Approximate DoF by blurring image based on depth buffer values

Depth of Field

1. Render scene to texture
2. Draw fullscreen quad
 - Compute the circle of confusion (CoC)
 - Based on the scene depth buffer
 - Blur the image using convolution or random sampling
 - Window size depends on the CoC

Depth of Field -- Artifacts

- Color bleeding
- Discontinuities at silhouettes
- Solutions:
 - Use bilateral filter
 - Advanced techniques
 - Diffusion based methods
 - ...



Motion Blur

- Fast moving objects appear blurry
- Property of the human eye and cameras
- Cameras: too long exposure
- Humans: moving the eye causes blur
- Advantages:
 - Looks good/realistic
 - Can cover performance problems

Motion Blur

Blurry, moves fast relative to camera:



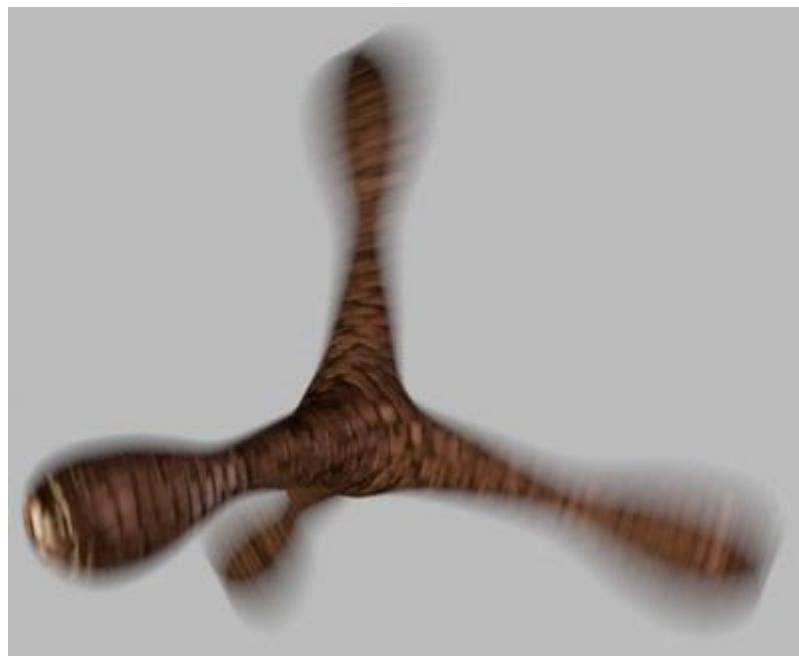
No blur, does not move relative to camera

Motion Blur

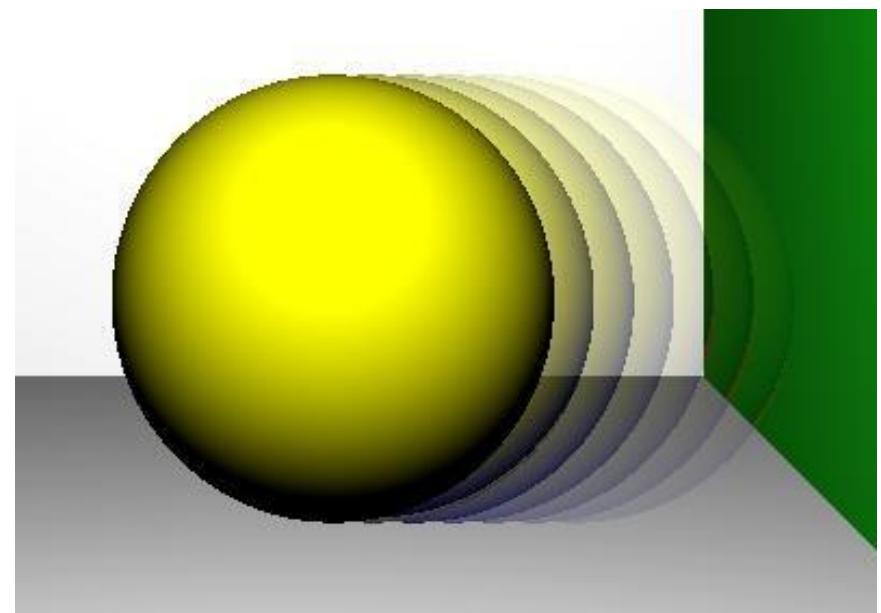


Motion Blur

- Continuous vs Discrete



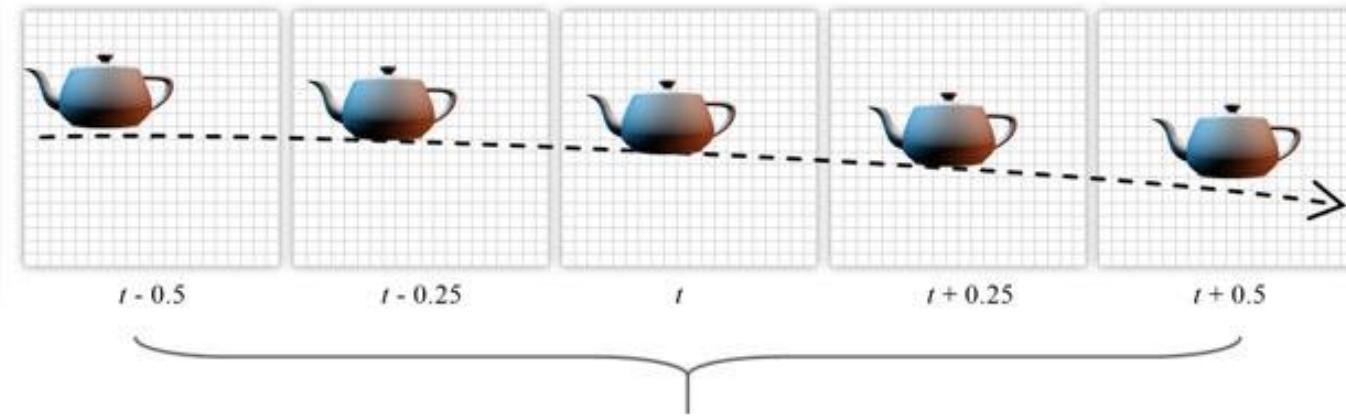
Correct, continuous MB



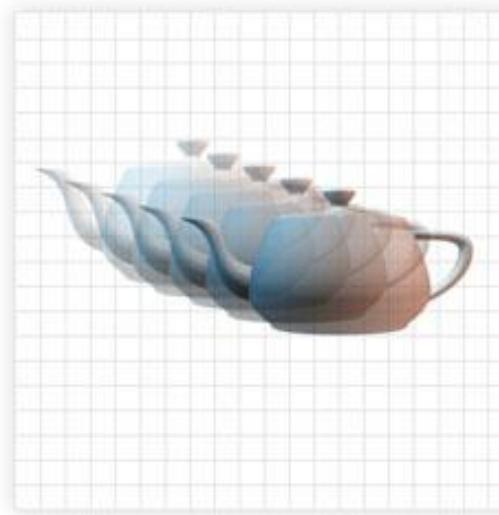
Approximated, discrete MB

Motion Blur

– Discrete Methods



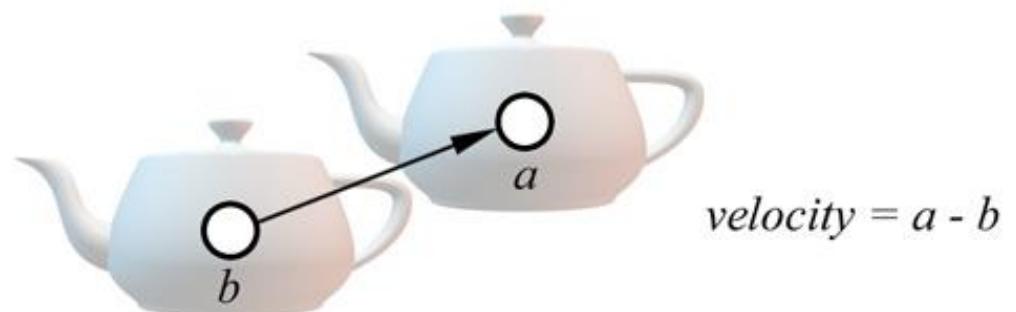
- Simplest method
 - Render object at past positions with varying transparency
 - Object needs to be rendered multiple times
- Image Space Motion Blur
 - Render object to buffer
 - Copy buffer with varying transparency
 - More efficient



Continuous Motion Blur

For each pixel:

- Compute how pixel moves over time
- Current and previous model-view projection matrix form *velocity buffer*
- Sample line along that direction
- Accumulate color values



Continuous Motion Blur – Examples



Image courtesy of Epic Games, Inc.

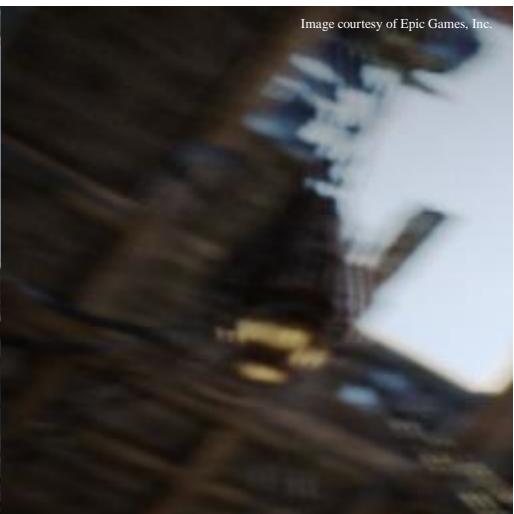
Continuous Motion Blur – Examples



Image courtesy of Epic Games, Inc.



Image courtesy of Epic Games, Inc.



Continuous Motion Blur – Examples



Image courtesy of Epic Games, Inc.



Image courtesy of Epic Games, Inc.

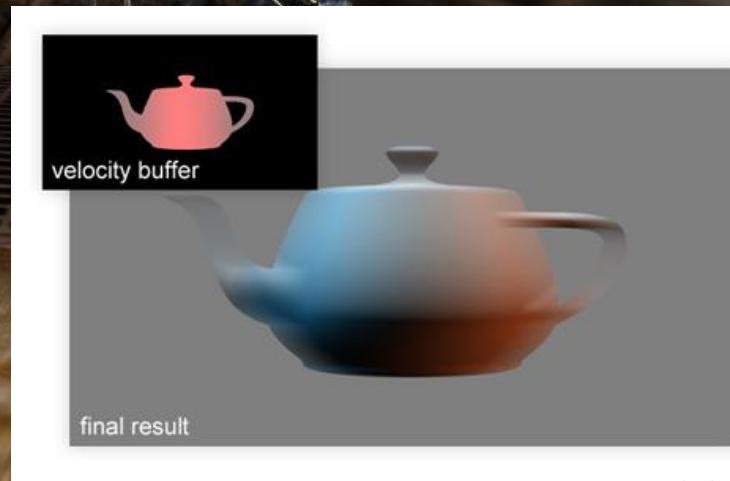
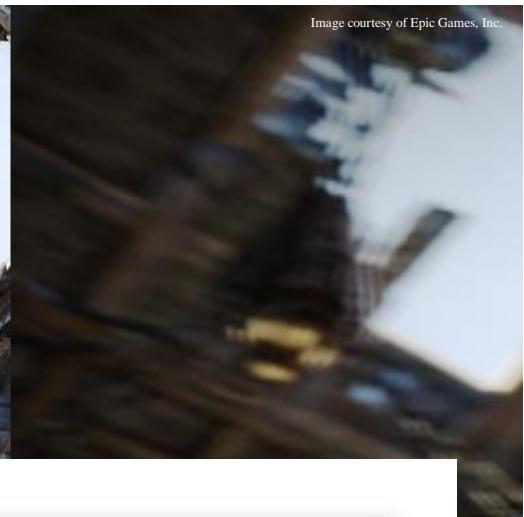


Image by John Chapman

Continuous Motion Blur – Artifacts

- Color bleeding
 - Slow foreground objects bleed into fast background objects
- Discontinuities at silhouettes



Blur not centred



Blur centred

Lens Flare

- A shortcoming of cameras that photographers try to avoid
- However: looks realistic and fancy
- Effect occurs inside lens system
 - Always on top
- Happens when light source inside image
- Star, ring or hexagonal shapes



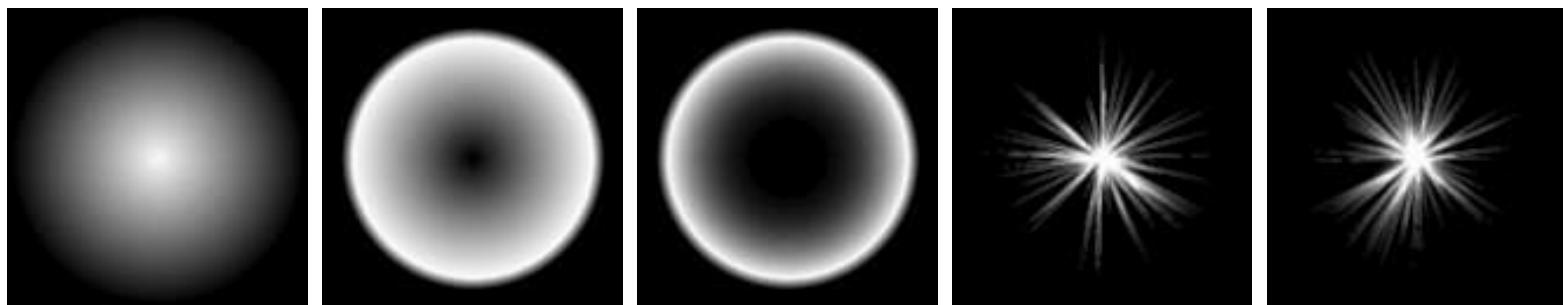
Lens Flare



Graphics Lecture 16: Slide 40

Lens Flare Rendering

- Choose a lens flare texture
- All lens flares lie on the line between light source and image center
- Rendered with differently sized textured quads and alpha blending



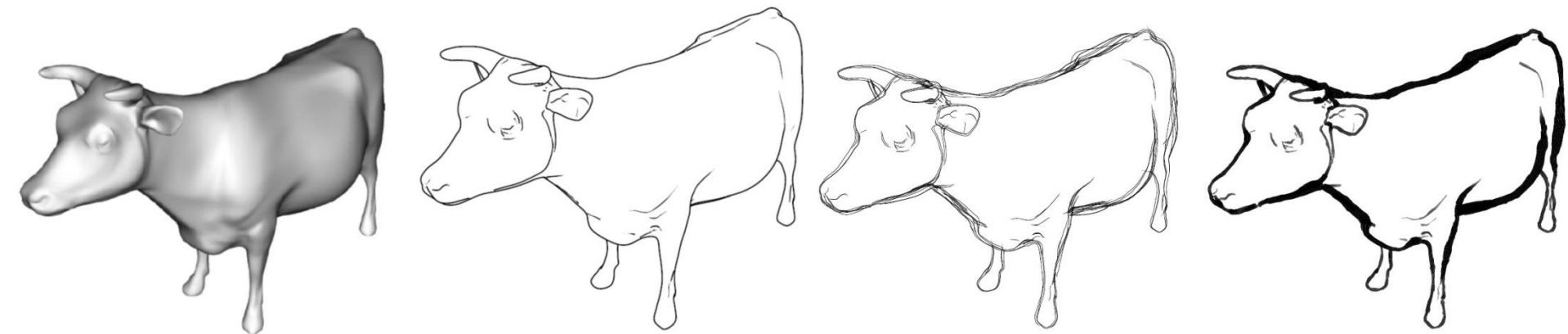
Lens Flare Rendering

- Don't overdo it!

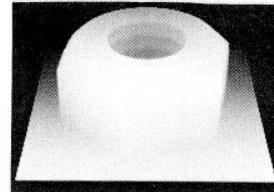


Non-Photorealistic Rendering

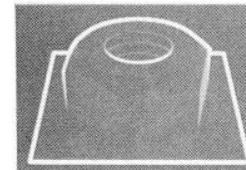
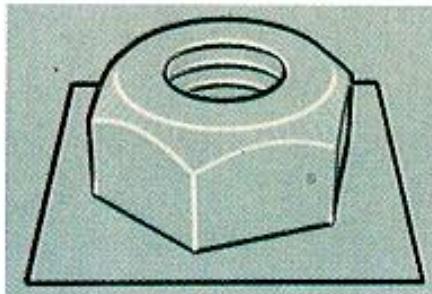
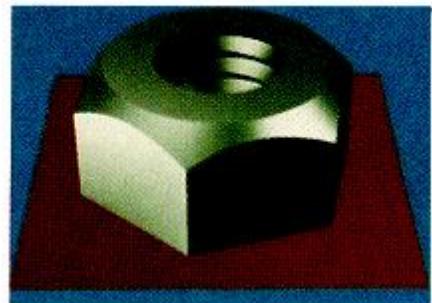
- Emphasizes object edges and silhouettes



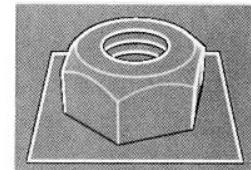
- Either from z-buffer or in object space
- Profile: 1st order differential operator (e.g., Sobel)
- Internal: 2nd order differential operator (e.g., Laplace)
- ...



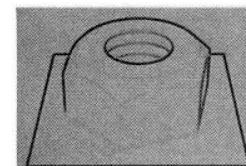
depth image



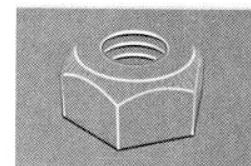
1st order differential



2nd order differential



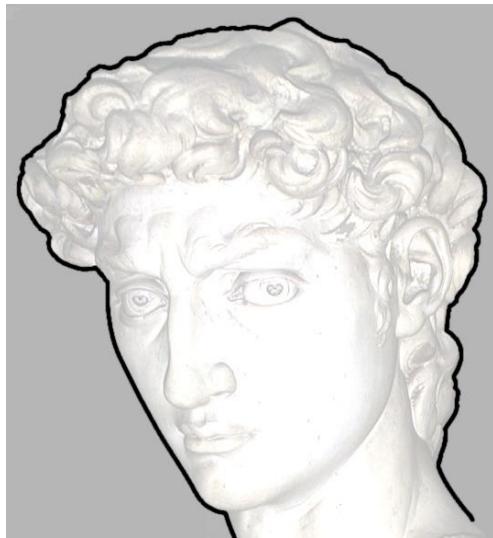
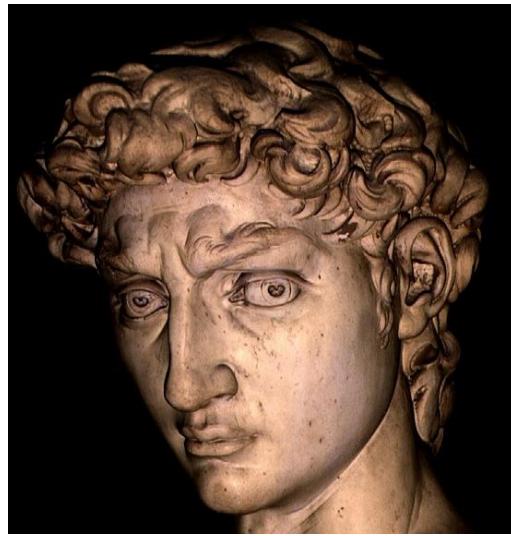
profile image



internal edge image

Line Classification

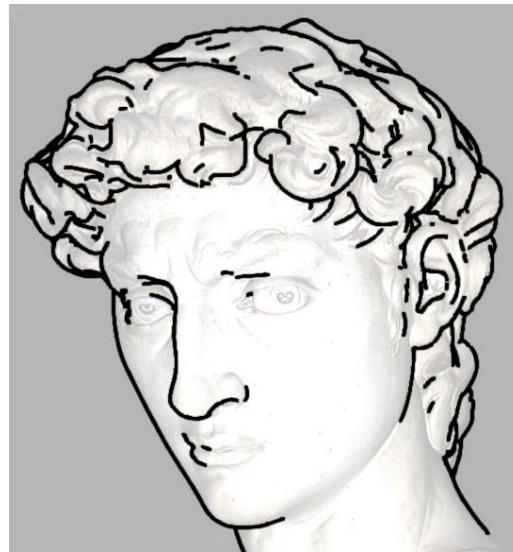
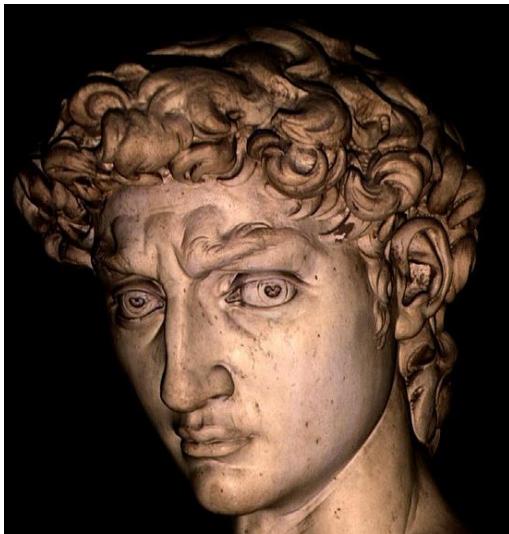
- Silhouette
 - Contour (Outer Silhouette)



Rusinkiewicz 05

Line Classification

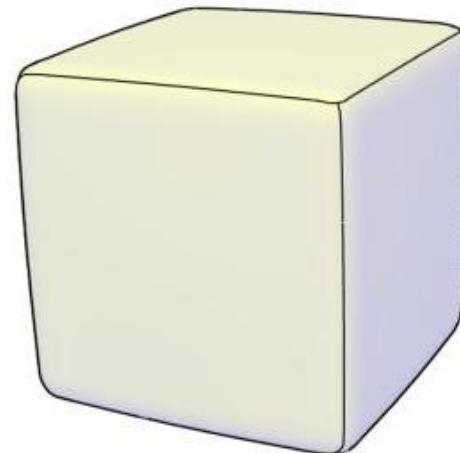
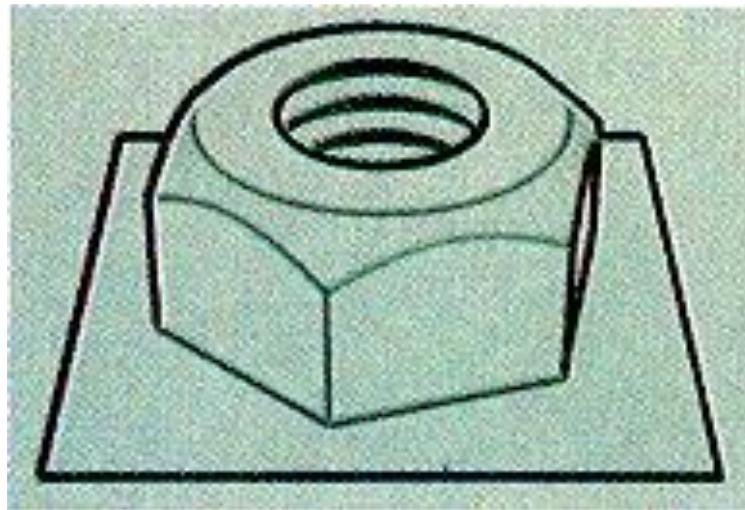
- Silhouette
 - Contour (Outer Silhouette)
 - **Occluding** contour (Inner Silhouette)



Rusinkiewicz 05

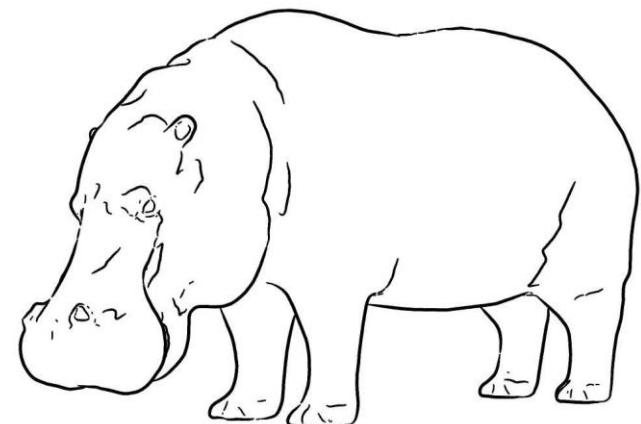
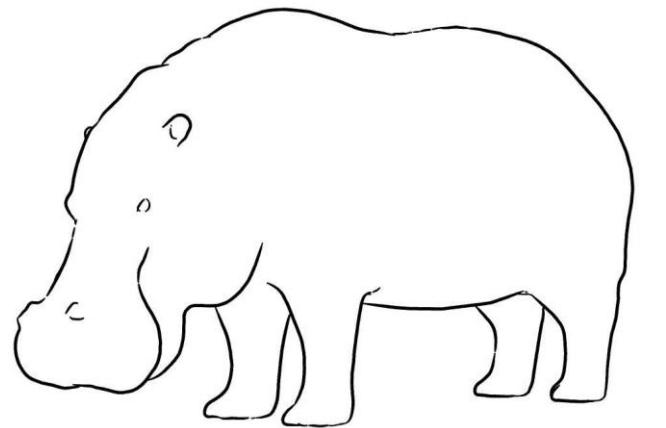
Line Classification

- Creases
 - Local maxima and minima of curvature
 - Ridges / Valleys



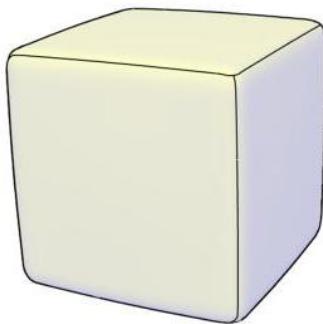
Line Classification

- Suggestive Contours
 - “Almost contours”
 - Points that become contours in nearby views

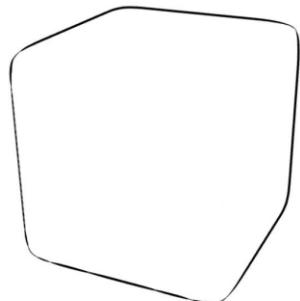


Line Classification

- Which Lines to Draw?
- Some objects do not have suggestive contours



← Creases →



← Suggestive Contours →

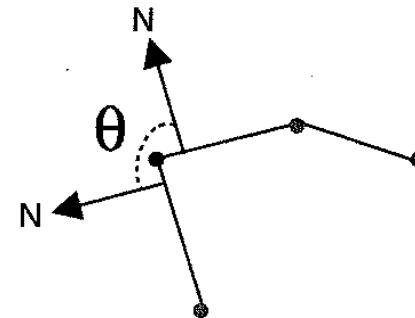
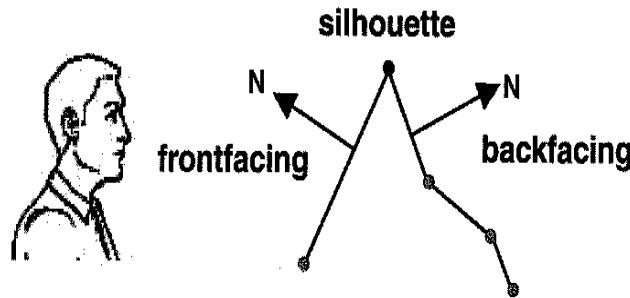


Rusinkiewicz 05

=> No universal rule which lines to draw <=

Line Detection in Object Space

- Silhouette
 - Points at which $n \cdot v = 0$
- Creases
 - Points at which angle > threshold



Gooche 01

Line Detection in Object Space

- Silhouette
 - View dependent
 - Online computation
- Creases
 - View independent
 - Pre-processing



Rusinkiewicz 05

Questions?