Computer Graphics: Recap

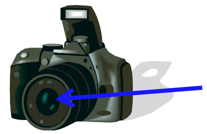
Part 2 – Lecture 15

The Camera Analogy

1. **Model Transformations**
   Arranging objects in a scene

2. **View Transformation**
   Positioning the camera

3. **Projection**
   Choosing a lens & taking a photo

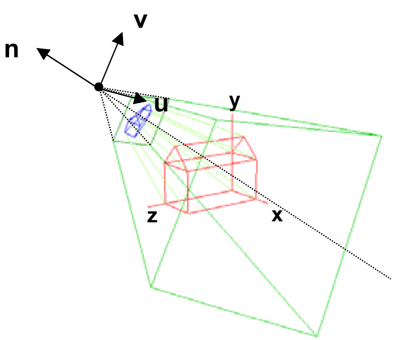


4. **Viewport Transformation**
   Printing a photo

The View Coordinate System

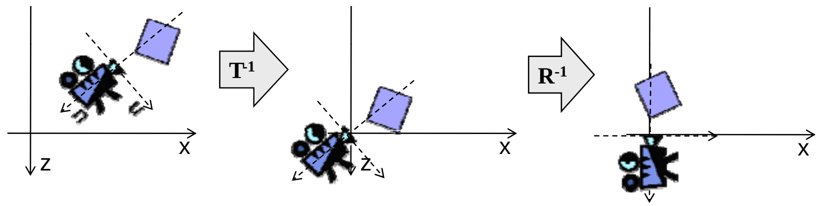
```
gluLookAt(
   eyeX, eyeY, eyeZ,
   lookAtX, lookAtY, lookAtZ,
   upX, upY, upZ
)
```

n = Normalised(Eye – LookAt)
u = Normalised(Cross(**Up**, **n**))
v = Cross(**n**, **u**)



View Transformation

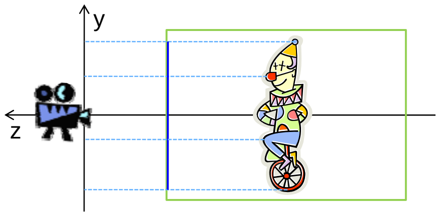
- Camera is at the origin looking down negative Z axis
- Could change camera position with translation $\mathbf{T}$ and rotation $\mathbf{R}$
- But instead of rotating and moving camera, transform our scene inversely so that the camera sees what we want it to see:



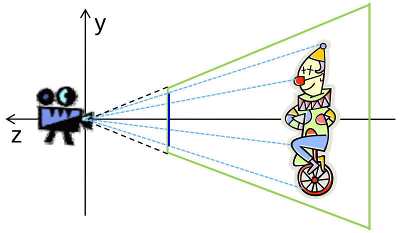
- In other words: we translate and rotate **view coordinate system** so that it is aligned with world coordinate system
- Viewing transform can be done as the last transform in $\mathbf{M}_{\mathbf{ModelView}}$ (i.e. must be set first in program)

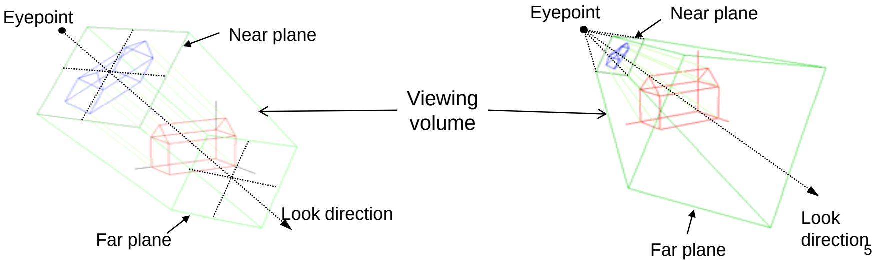
Orthographic vs. Perspective Projection

Orthographic Projection

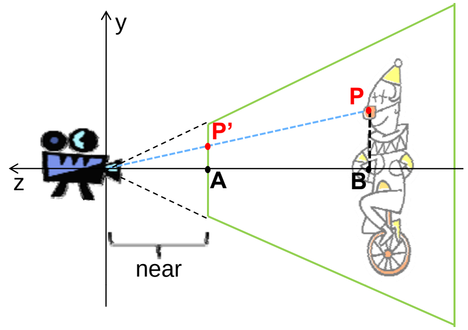


Perspective Projection





Perspective Projection of a Vertex



- What are the coordinates of **P'** ?
- Camera-**A**-**P'** and Camera-**B**-**P** are similar triangles
- Ratios of similar sides are equal:

$$\frac{P_y'}{near} = \frac{P_y}{-P_z} \Leftrightarrow P_y' = \frac{near}{-P_z} P_y$$

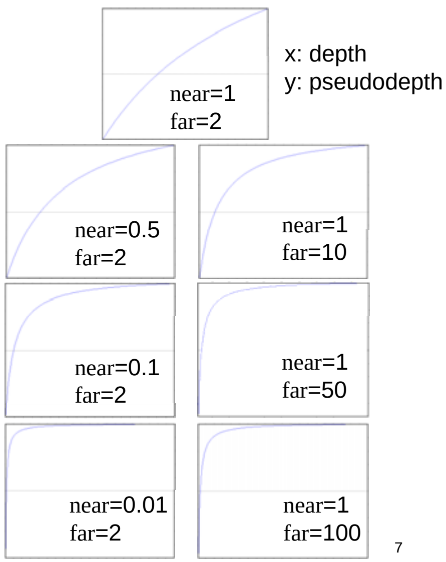
- When looking from the bottom, we get analogous calculations for the x-coordinate of **P'**:

$$\frac{P_x'}{near} = \frac{P_x}{-P_z} \Leftrightarrow P_x' = \frac{near}{-P_z} P_x$$

- Perspective scaling factor $s_{persp} = \frac{near}{-P_z}$

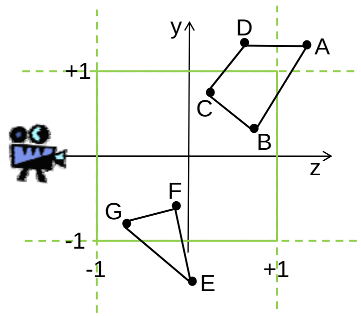
Pseudodepth

- Transformed z* not linear function of z $z^* = \frac{(far + near)z + 2\, far * near}{(far - near)z}$
- This is ok because
  1. z* monotonic increasing, and
  2. z* = -1 for z = -near
     z* = +1 for z = -far
- Avoid very small near and very large far
  → resolution too low for points that are further away



Clipping

- Determine which lines are in the canonical view volume (using NDC)
- Outside of the view volume is given by:
  $p_x < -1$, $p_x > +1$, $p_y < -1$, $p_y > +1$,
  $p_z < -1$, $p_z > +1$
  (→ **clip planes**)
- Each line is either…
  1. completely inside
     → **trivial accept**
  2. completely outside
     → **trivial reject**
  3. Partially in the view volume
     → need to find out which part is inside



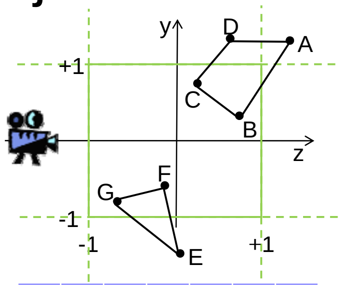
Trivial accept for:
CB and GF

Trivial reject for:
DA

Partially visible:
AB, CD, EF and EG

Trivial Accept and Reject Tests

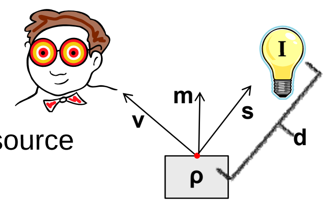
- For each point, check if it is outside of left (L), right (R), bottom (B), top (T), near (N) and far (F) clip plane
- Create table with **outcodes**:
  1 if point is outside, 0 if inside
- **Trivial reject** of a line PQ:
  = P and Q outside of the same clip plane
  = outcodes for same plane both 1
  = `(outcode P & outcode Q)!=0`
- **Trivial accept** of a line PQ:
  = both endpoints inside of all clip planes
  = all outcodes 0
  = `(outcode C | outcode D)==0`

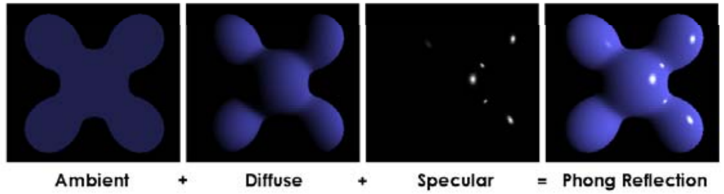


	L	R	B	T	N	F
A	0	0	0	**1**	0	**1**
B	0	0	0	0	0	0
C	0	0	0	0	0	0
D	0	0	0	**1**	0	0
E	0	0	**1**	0	0	0
F	0	0	0	0	0	0
G	0	0	0	0	0	0

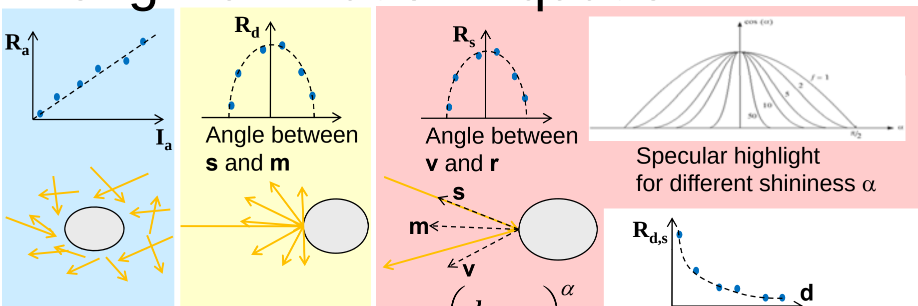
Phong Illumination Model

- **Idea**: calculate intensity **R** (and color) of visible light at a point as the sum of ambient, diffuse and specular reflection
- Variables taken into account:
  - Intensities $\mathbf{I}_a$, $\mathbf{I}_d$, $\mathbf{I}_s$ for incident light
  - Surface normal vector **m**
  - Vector **s** describing the direction to the light source
  - Distance **d** to light source
  - Vector **v** describing the direction to the viewer
  - Reflection coefficients of the surface material ρ_a, ρ_d, ρ_s

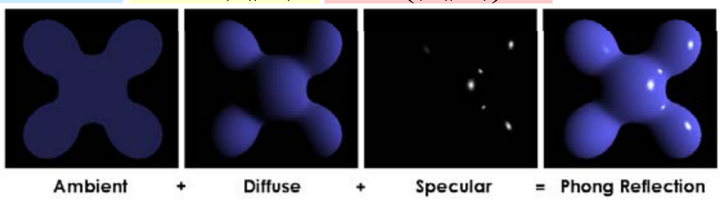




Phong Illumination Equation



$$\mathbf{R} = \mathbf{I_a}\,\rho_a + \left(\mathbf{I_d}\,\rho_d \frac{s \cdot m}{|s||m|} + \mathbf{I_s}\,\rho_s \left(\frac{h \cdot m}{|h||m|}\right)^{\alpha}\right) / \left(k_c + k_l d + k_q d^2\right)$$



Setting Up Lights

```
float lightPos0[] = {-1.0, 2.0, 3.0, 1.0}; // point source
glLightfv(GL_LIGHT0, GL_POSITION, lightPos0);
float lightPos1[] = {0.0, 1.0, 2.0, 0.0};  // directional
glLightfv(GL_LIGHT1, GL_POSITION, lightPos1);
glEnable(GL_LIGHTING);    // enable lighting in general
glEnable(GL_LIGHT0);      // enable light number 0
glEnable(GL_LIGHT1);      // enable light number 1
```

For setting the properties of lights, use one of

`glLightfv(GLenum light, GLenum pname, float* params)`
`glLightf(GLenum light, GLenum pname, float param)`

- `light` selects a light `GL_LIGHTi` with 0 < i < `GL_MAX_LIGHTS` (8)
- `pname` selects a property to set (e.g. `GL_POSITION`)

- For point sources: set position to (x, y, z, 1)
- For directional light sources: set position to (x, y, z, 0)
  (x,y,z) points towards the light source

Using Materials

```
float ambient[] = {0.1, 0.1, 0.1, 1.0};         // ρar, ρag, ρab, 1
float diffuse[] = {0.4, 0.4, 0.6, 1.0};         // ρdr, ρdg, ρdb, 1
float specular[] = {0.8, 0.8, 1.0, 1.0};        // ρsr, ρsg, ρsb, 1

glMaterialfv(GL_FRONT, GL_AMBIENT, ambient);
glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, specular);

glMaterialf(GL_FRONT, GL_SHININESS, 40.0);      // α=40
```

Set the current material, then draw primitives (they will use the material)

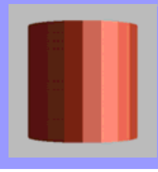
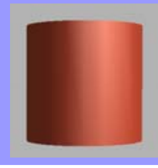
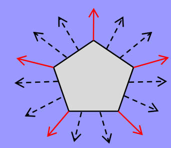
`glMaterialfv(GLenum face, GLenum pname, float* params)`

`glMaterialf(GLenum face, GLenum pname, float param)`

- face selects side to use material on (`GL_FRONT`, `GL_BACK` or `GL_FRONT_AND_BACK`)
- pname selects a property to set (e.g. `GL_AMBIENT`, `GL_EMISSION`, `GL_AMBIENT_AND_DIFFUSE`, `GL_SHININESS`, …)

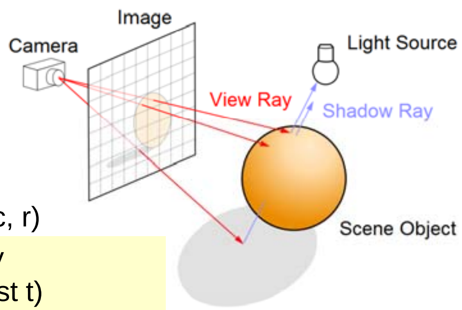
- Set coefficients as RGBA: A (alpha) for color blending, is usually 1

Shading Algorithms

Flat Shading	Gouraud Shading	Phong Shading
		
Simple and fast Phong equation only once per face	Still fast Phong equation at each vertex No 0th-order color discontinuities	Crisp highlights with few vertices
Mach Bands	Slight mach bands, Color invariance with quadrilaterals, Problems with highlights	Slow Phong calculation for every Pixel

Ray Casting Algorithm

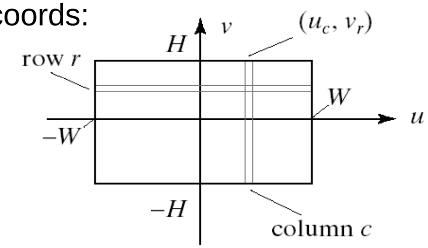
define scene = ({ objects }, { lights })
define camera (eye, u, v, n)
for (int r = 0; r < nRows; r++) {
    for (int c = 0; c < nCols; c++) {
        construct ray going through (c, r)
        find closest intersection of ray with an object (smallest t)
        find intersection point P **intersect**
        get the surface normal at P
        get the color at the intersection **shade**
        pixel(c, r) = color
} }



Constructing Rays

Wanted: ray (***startPoint***, ***direction***) from eye through every pixel

- Corners of the view plane in world coords:
  *bottomLeft = centre + (-W**u**, -H**v**)*
  *bottomRight = centre + (W**u**, -H**v**)*
  *topLeft = centre + (-W**u**, H**v**)*
  *topRight = centre + (W**u**, H**v**)*

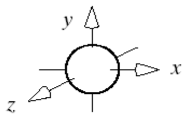


- Go through all pixels, with column 0 and row 0 at *bottomLeft*
- Ray direction ***d*** = ***pixelPos*** - ***eye***

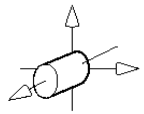
$$\mathbf{d} = -N\mathbf{n} + W\left(\frac{2c}{nCols} - 1\right)\mathbf{u} + H\left(\frac{2r}{nRows} - 1\right)\mathbf{v}$$

Ray-Object Intersection

- Define each object as an implicit function f:
  f(**p**) = 0 for every point **p** on the surface of the object
  (if **p** is not on surface, then f(**p**) ≠ 0)
- Examples for simple objects ("primitives"):
  - Sphere (center at origin, radius 1)
    $f(\mathbf{p}) = x^2 + y^2 + z^2 - 1 = |\mathbf{p}|^2 - 1$



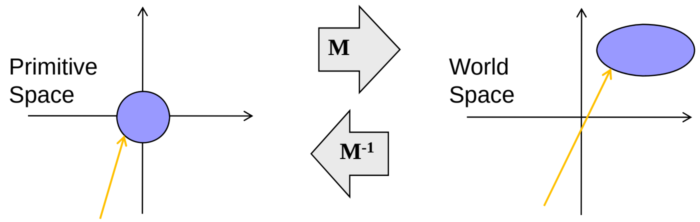
  - Cylinder (around z-axis, radius 1)
    $f(\mathbf{p}) = x^2 + y^2 - 1$



- Where a ray (**eye** + **d** *t*) meets the object:
  f(**eye** + **d** *t*) = 0
  → solve for *t* and get intersection point **eye** + **d** *t*

Transformed Primitives

Problem: How to intersect with transformed primitives?
(e.g. scaled and translated unit sphere)



Solution: intersection of ray with transformed primitive is the same as intersection with inversely transformed ray and primitive

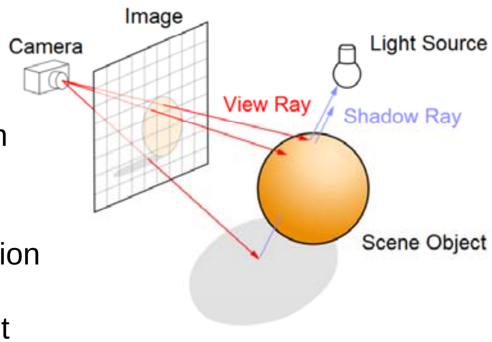
- Intersect with transformed ray ($\mathbf{eye}_t + \mathbf{d}_t\, t$)
  i.e. $\mathbf{eye}_t = \mathbf{M}^{-1}\,\mathbf{eye}$ and $\mathbf{d}_t = \mathbf{M}^{-1}\,\mathbf{d}$
- *t* for the intersection is the same in world and primitive space

Shadow Feelers

Problem: How do we know if a point **p** is in shadow of a light **l** ?

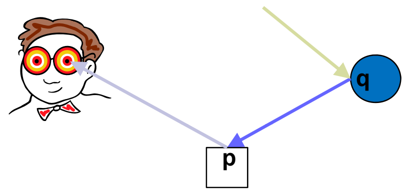
Solution: Check if there is something between **p** and **l**

1. Calculate (**source**, **d**) for a ray that starts at **p** and goes to **l** (a "shadow feeler")
2. Check if there is an intersection with any scene object (→ use `intersect`)
3. If there is a ray-object intersection between **p** and **l** then: do not illuminate **p** with the light i.e. do not add R_d and R_s
   Otherwise: normal illumination



Ray Tracing Reflections

Idea: the color of a point is influenced by the color that the ray carries over from the previous reflection



Ray is reflected at **q** (blue sphere) before being reflected at **p** (white box)
→ ray has bluish color when it hits the box

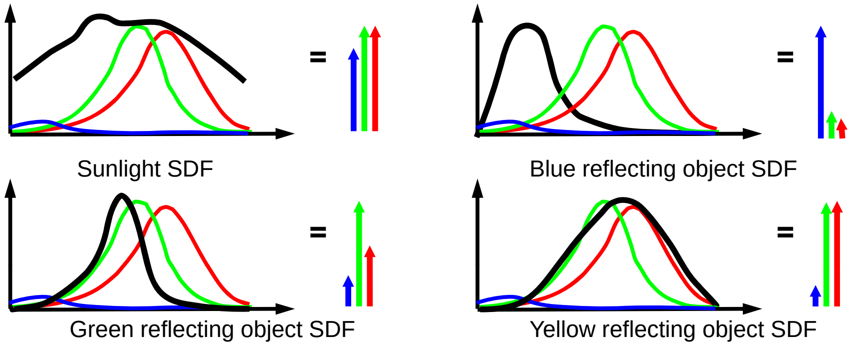
Reflectivity: fraction of incident radiation reflected by a surface (between 0 and 1)

Add the fraction of light reflected from q to the reflection at p:

$$R_p = R_{ambient,p} + R_{diffuse,p} + R_{specular,p} + reflectivity_p\ R_q$$

Seeing Red, Green, Blue (cont'd)

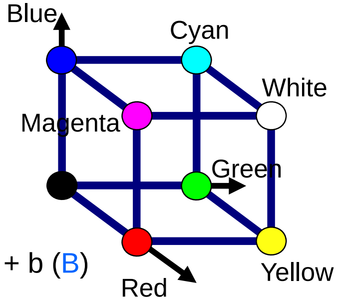
- Example L, M, S responses for various SDF's

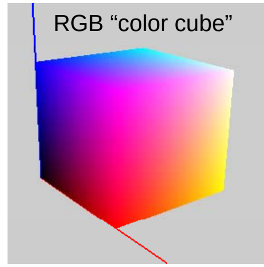


- Resulting L, M, and S SRF responses are independent values
- The 3 SRF response values are interpreted as hues by our brain, e.g. red + green = yellow, red + green + blue = white

Color Coordinate Space

- Defines 3 SRFs (**color matching functions**) for some sensing system
- One dimension for each SRF (→ **tristimulus color space**)
  - Each dimension represents a **primary color P**
  - Coordinate value = resulting SDF integral normalized to (0, 1)
- Color triple is 3D point defined by **chromaticity values** (c_0, c_1, c_2)
- Example: RGB color space
  - Primaries:
    Red, Green, Blue
    with basis vectors
    R = (0,0,1)
    G = (1,0,0)
    B = (0,1,0)
  - Chromaticity values:
    (r,g,b) = r (R) + g (G) + b (B)

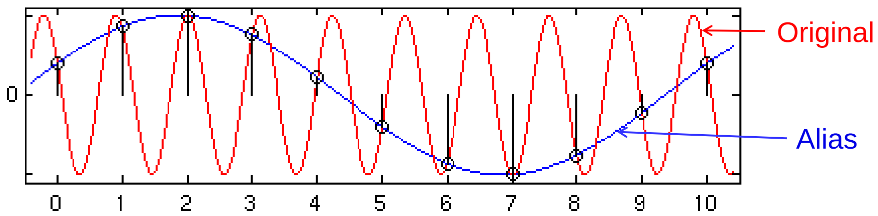




Aliasing

A signal looks like another signal (the "alias") after sampling

- Not a problem if the signals are still very similar
- But is a problem if the alias looks really different (→ aliasing artifacts)
- Happens particularly when sampling a high-frequency signal with a low sample frequency



Exam

- Multiple-choice only
- Closed book
- Question types in my part:
  - A few calculations (involving matrices)
  - Which formula is correct?
  - Which of the statements is false?
  - Given some code:
    - "What needs to be changed to achieve X?"
    - "What happens if you change X?"