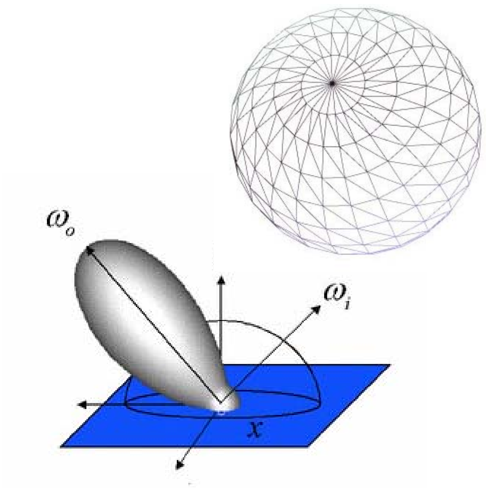
CS680: Scalable Global Illumination
Summary of Under. CG related to CS680

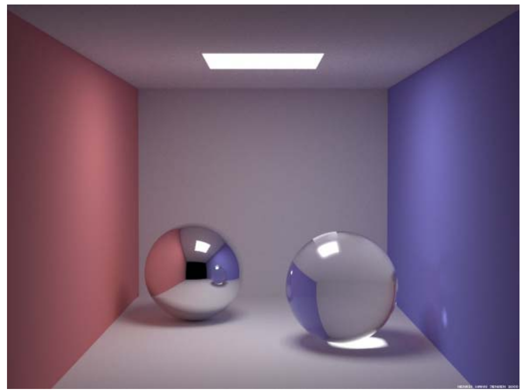
Sung-Eui Yoon
(윤성의)

Course URL:
http://jupiter.kaist.ac.kr/~sungeui/CG

Overview of Computer Graphics

- **We will discuss various parts of computer graphics**





Modelling **Simulation & Rendering** **Image**

Computer vision inverts the process
Image processing deals with images

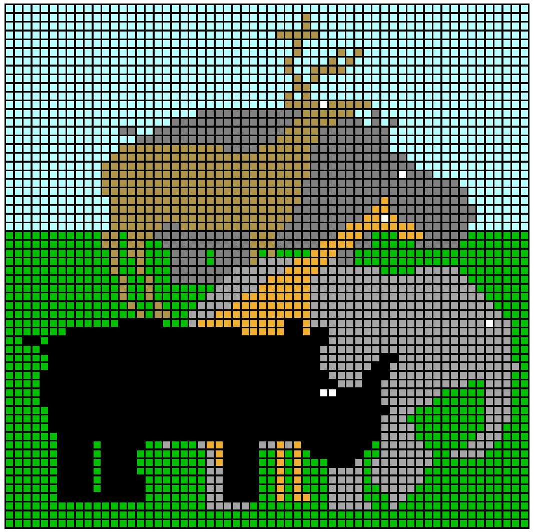
Lecture 2: Screen Space & World Space

Mapping from World to Screen

Screen Space

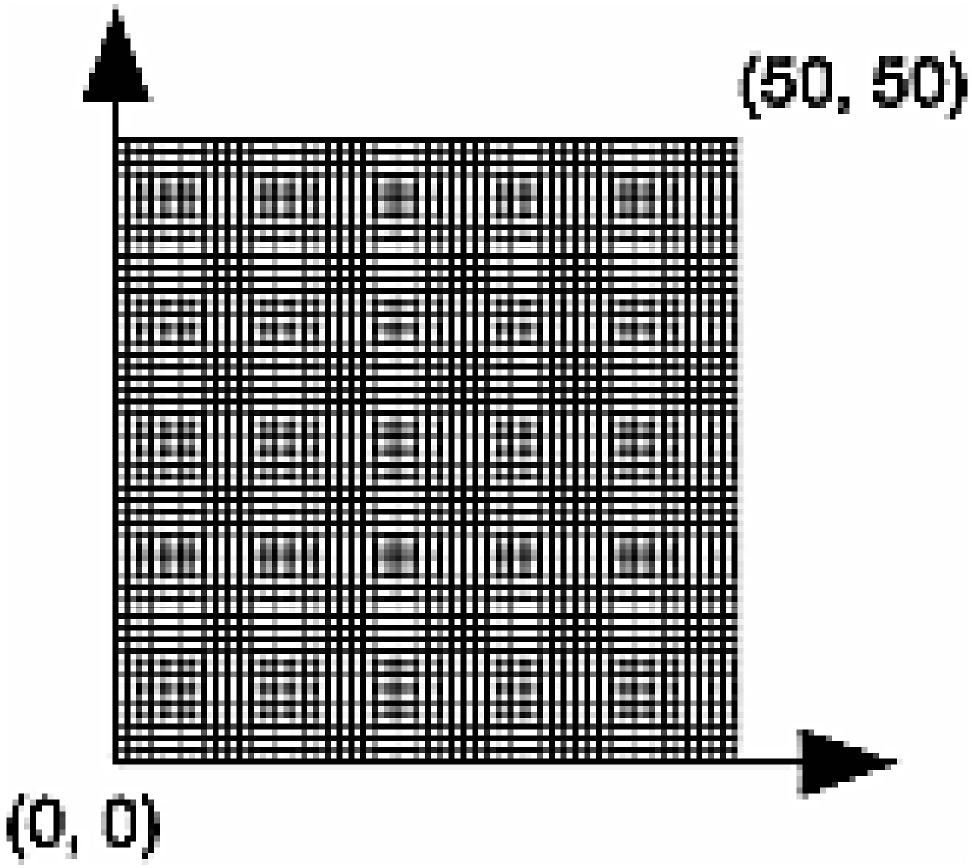
- **Graphical image is presented by setting colors for a set of discrete samples called "pixels"**
  - **Pixels displayed on screen in windows**
- **Pixels are addressed as 2D arrays**
  - **Indices are "screen-space" coordinates**

(0,0) **(width-1,0)**



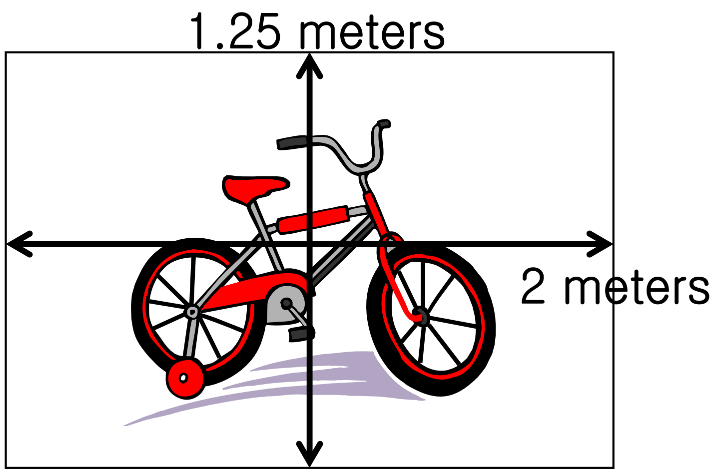
(0,height-1) **(width-1, height-1)**

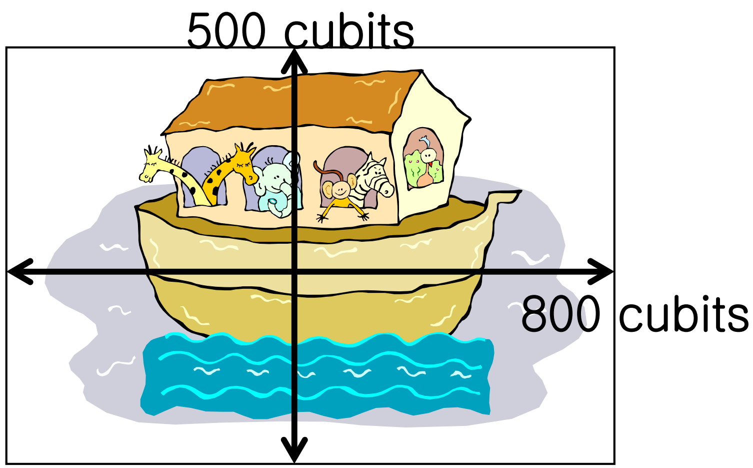
OpenGL Coordinate System



Pixel Independence

- Often easier to structure graphical objects independent of screen or window sizes
- Define graphical objects in “world-space”

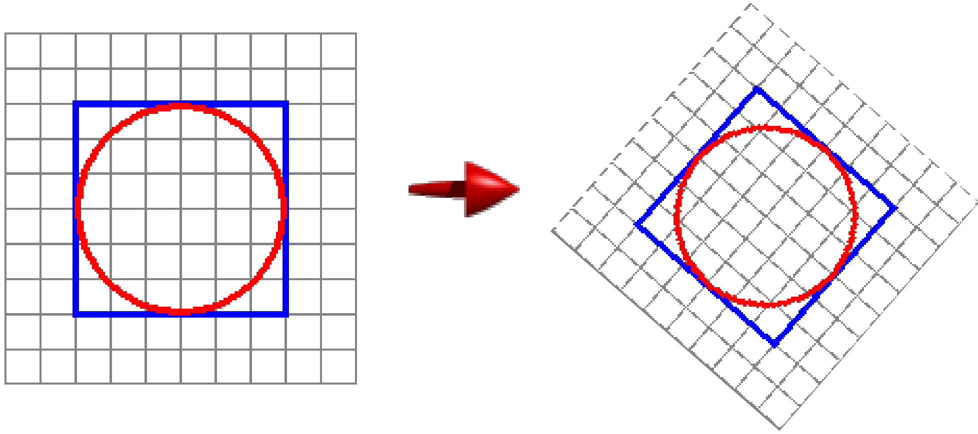


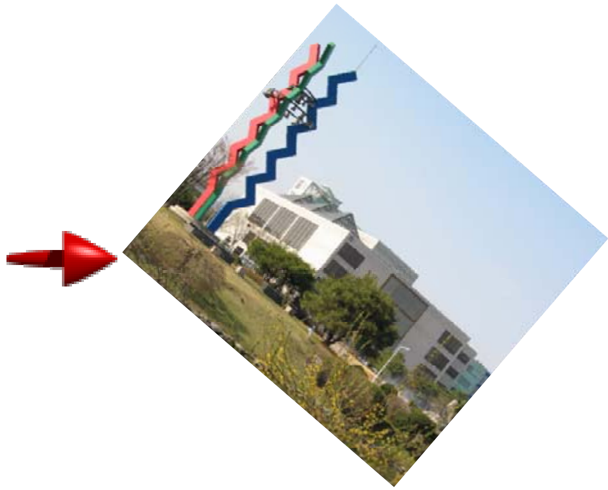


Lecture: 2D Transformation

2D Geometric Transforms

- **Functions to map points from one place to another**
- **Geometric transforms can be applied to**
  - **Drawing primitives (points, lines, conics, triangles)**
  - **Pixel coordinates of an image**





Demo

Translation

- **Translations have the following form:**

$$x' = x + t_x \qquad y' = y + t_y \qquad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

- ***inverse function:* undoes the translation:**

$$x = x' - t_x$$
$$y = y' - t_y$$

- ***identity*: leaves every point unchanged**

$$x' = x + 0$$
$$y' = y + 0$$

2D Rotations

- **Another group - rotation about the origin:**

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\Theta & -\sin\Theta \\ \sin\Theta & \cos\Theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = R \begin{bmatrix} x \\ y \end{bmatrix}$$

$$R^{-1} = \begin{bmatrix} \cos\Theta & \sin\Theta \\ -\sin\Theta & \cos\Theta \end{bmatrix}$$

$$R_{\theta=0} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Rotations in Series

- **We want to rotate the object 30 degree and, then, 60 degree**

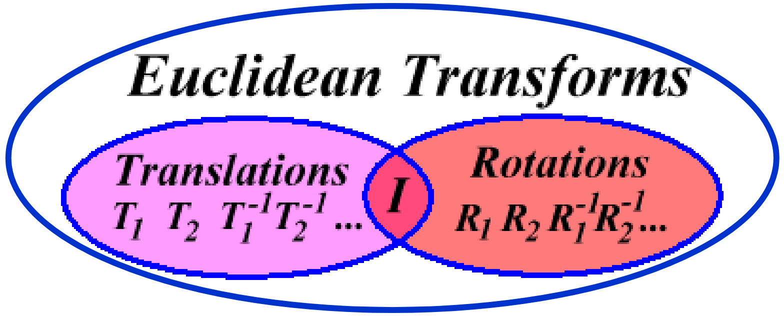
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \underbrace{\begin{bmatrix} \cos(60) & -\sin(60) \\ \sin(60) & \cos(60) \end{bmatrix} \begin{bmatrix} \cos(30) & -\sin(30) \\ \sin(30) & \cos(30) \end{bmatrix}} \begin{bmatrix} x \\ y \end{bmatrix}$$

We can merge multiple rotations into one rotation matrix

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(90) & -\sin(90) \\ \sin(90) & \cos(90) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Euclidean Transforms

- **Euclidean Group**
  - **Translations + rotations**
  - **Rigid body transforms**



- **Properties:**
  - **Preserve distances**
  - **Preserve angles**
  - **How do you represent these functions?**

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

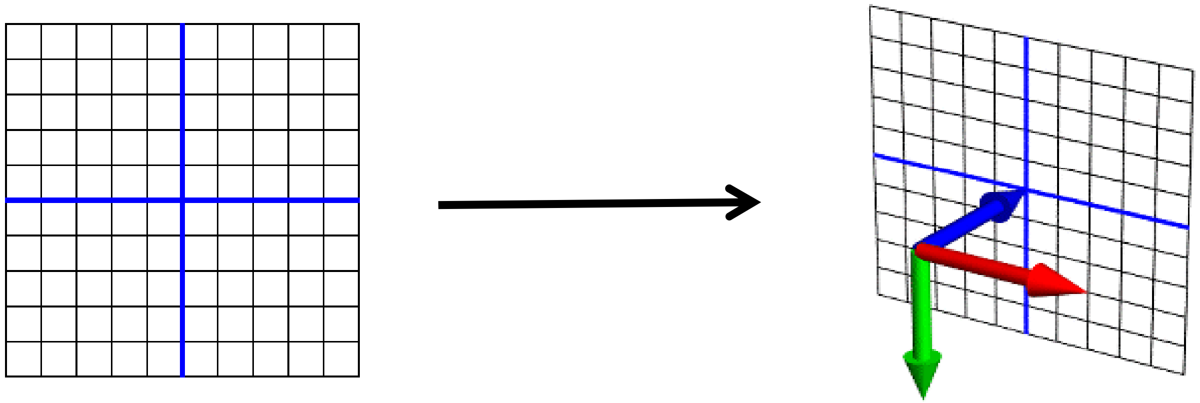
Problems with this Form

- Translation and rotation considered separately
  - Typically we perform a series of rotations and translations to place objects in world space
  - It's inconvenient and inefficient in the previous form
  - Inverse transform involves multiple steps

- How can we address it?
  - How can we represent the translation as a matrix multiplication?

Homogeneous Coordinates

- Consider our 2D plane as a subspace within 3D



(x, y)

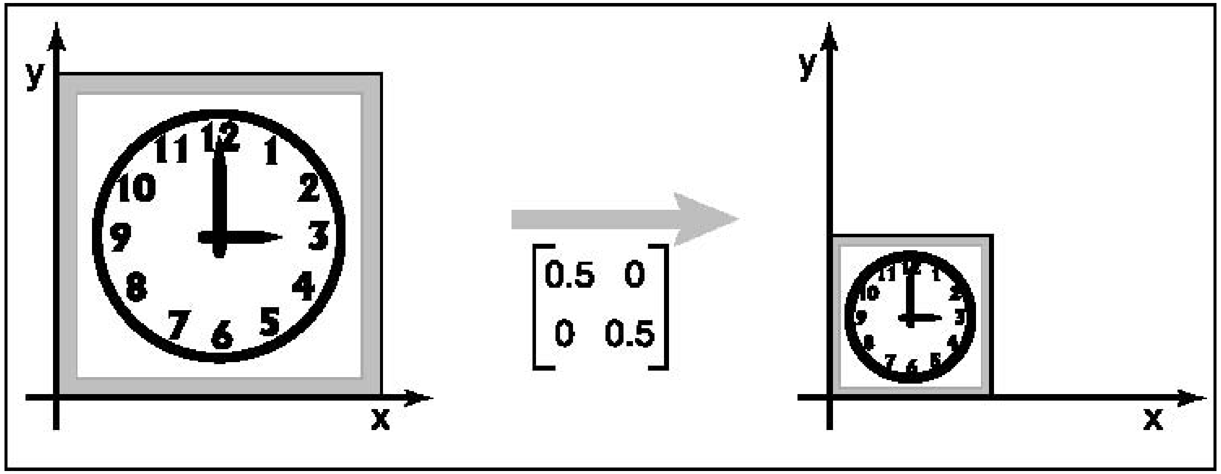
(x, y, z)

Matrix Multiplications and Homogeneous Coordinates

- **Can use any planar subspace that does not contain the origin**
- **Assume our 2D space lies on the 3D plane *z = 1***
  - **Now we can express all Euclidean transforms in matrix form:**

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & t_x \\ \sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Scaling



- **S is a scaling factor**

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

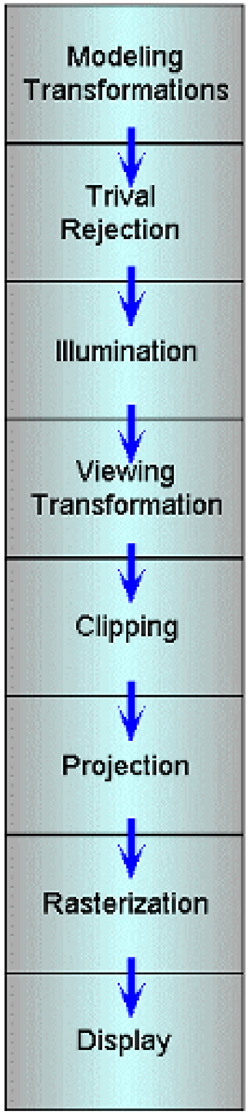
Frame Buffer

- Contains an image for the final visualization
- Color buffer, depth buffer, etc.

- Buffer initialization
  - glClear(GL_COLOR_BUFFER_BIT);
  - glClearColor (..);
- Buffer creation
  - glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
- Buffer swap
  - glutSwapBuffers();

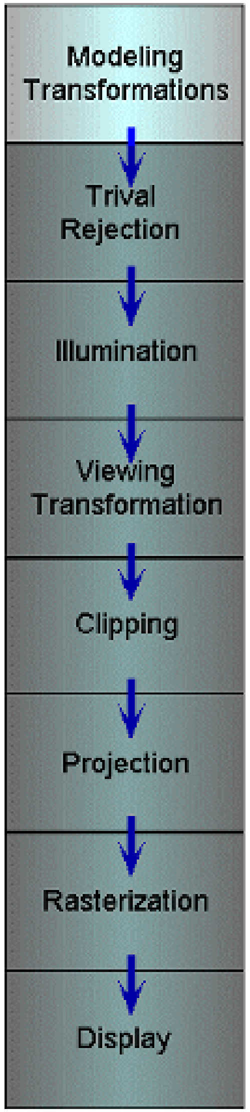
Lecture: Modeling Transformation

The Classic Rendering Pipeline

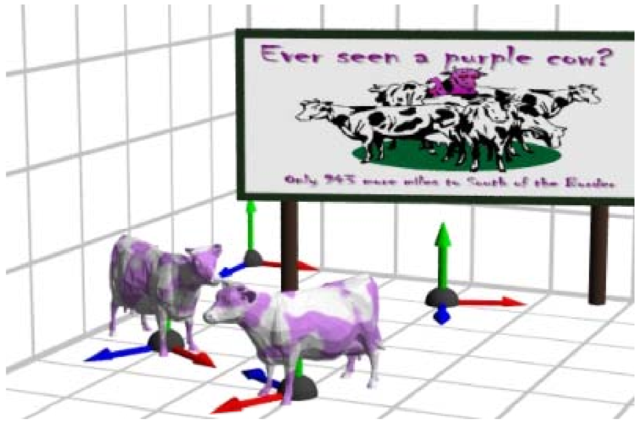


- Object primitives defined by vertices fed in at the top
- Pixels come out in the display at the bottom
- Commonly have multiple primitives in various stages of rendering

Modeling Transforms

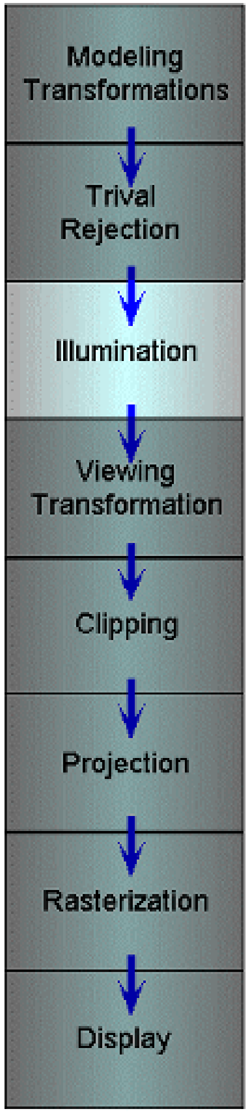


- **Start with 3D models defined in modeling spaces with their own modeling frames:** $\dot{m}_1^t, \dot{m}_2^t, \ldots, \dot{m}_n^t$
- **Modeling transformations orient models within a common coordinate frame called world space,** $\dot{w}^t$
  - **All objects, light sources, and the camera live in world space**

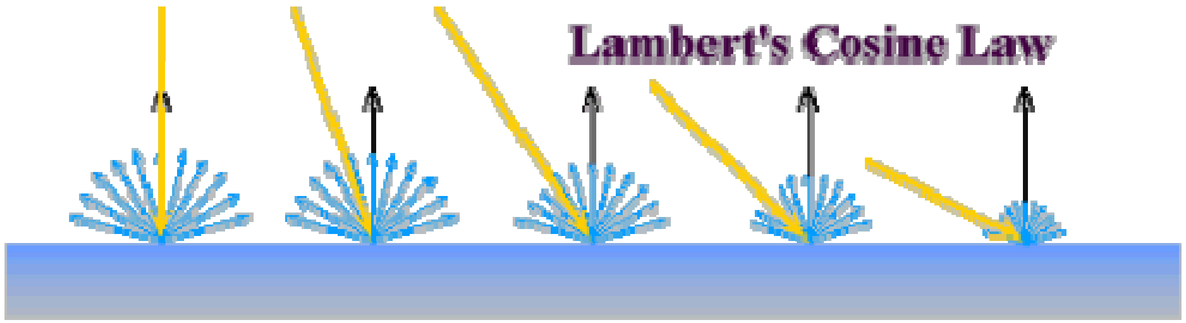


- **Trivial rejection attempts to eliminate objects that cannot possibly be seen**
  - **An optimization**

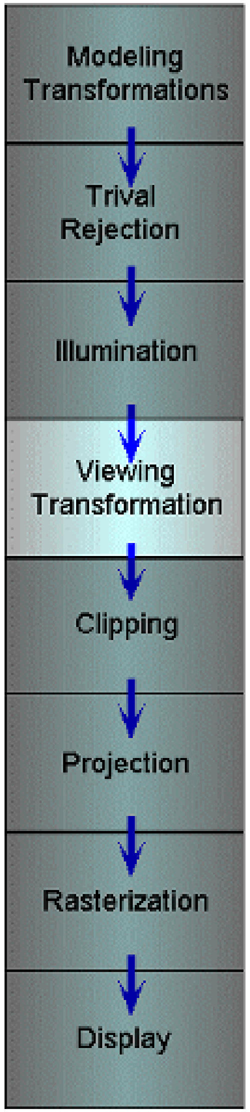
Illumination



- Illuminate potentially visible objects
- Final rendered color is determined by object's orientation, its material properties, and the light sources in the scene



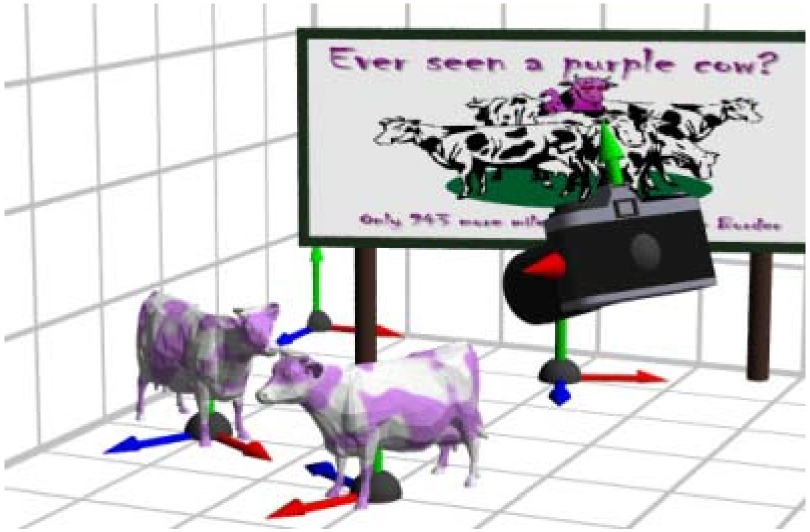
Viewing Transformations



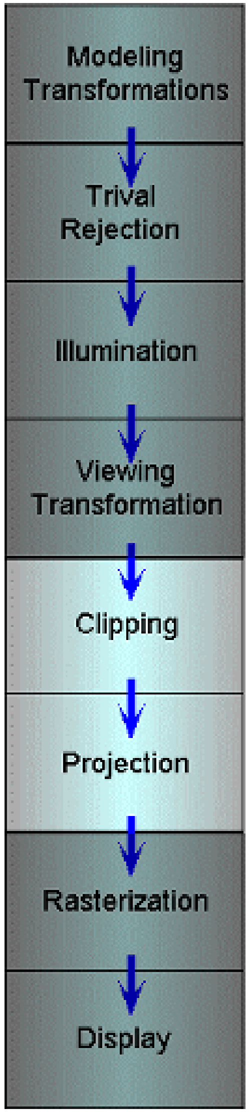
- Maps points from world space to **eye space**:

$$\vec{e}^{\,t} = \vec{w}^{\,t}\mathbf{V}$$

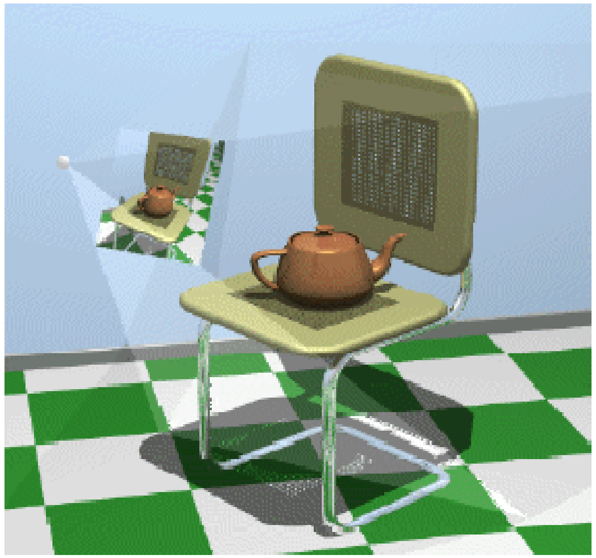
  - Viewing position is transformed to the origin
  - Viewing direction is oriented along some axis



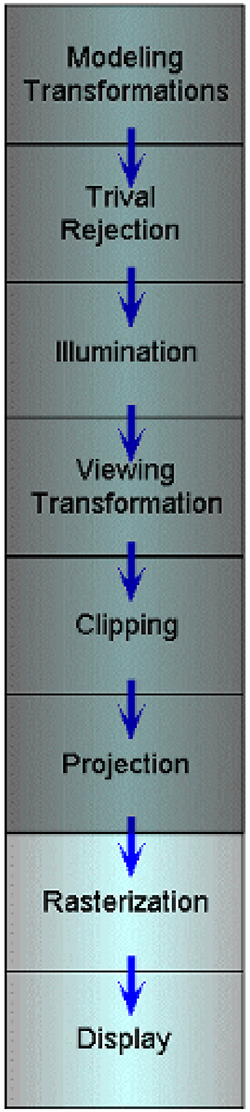
Clipping and Projection



- We specify a volume called a *viewing frustum*
- Map the view frustum to the unit cube
- Clip objects against the view volume, thereby eliminating geometry not visible in the image
- Project objects into two-dimensions
- Transform from eye space to normalized device coordinates



Rasterization and Display



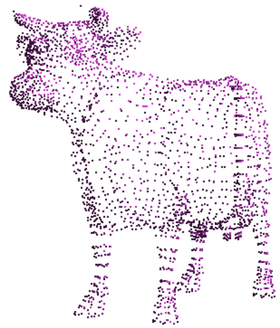
- **Transform normalized device coordinates to screen space**
- **Rasterization converts objects pixels**

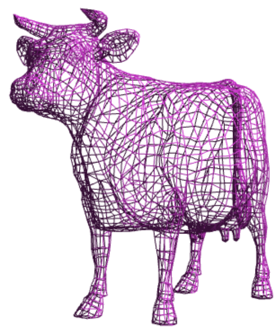
- Almost every step in the rendering pipeline involves a change of coordinate systems!
- Transformations are central to understanding 3D computer graphics

Lecture: Interaction

Primitive 3D

- How do we specify 3D objects?
  - Simple mathematical functions, $z = f(x,y)$
  - Parametric functions, $(x(u,v), y(u,v), z(u,v))$
  - Implicit functions, $f(x,y,z) = 0$





- Build up from simple primitives
  - Point – nothing really to see
  - Lines – nearly see through
  - Planes – a surface

Simple Planes

- Surfaces modeled as connected planar facets
  - N (>3) vertices, each with 3 coordinates
  - Minimally a triangle

Specifying a Face

- **Face or facet**

Face [v0.x, v0.y, v0.z] [v1.x, v1.y, v1.z] … [vN.x, vN.y, vN.z]

- **Sharing vertices via indirection**

Vertex[0] = [v0.x, v0.y, v0.z]

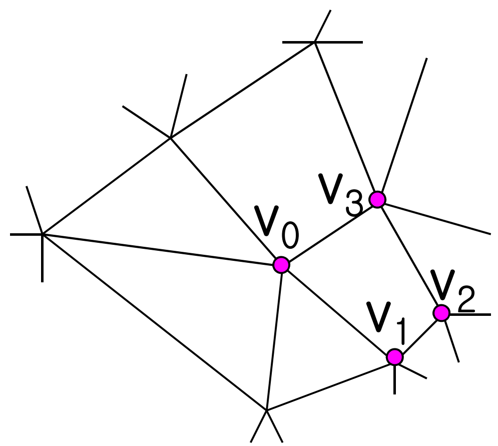
Vertex[1] = [v1.x, v1.y, v1.z]

Vertex[2] = [v2.x, v2.y, v2.z]

:

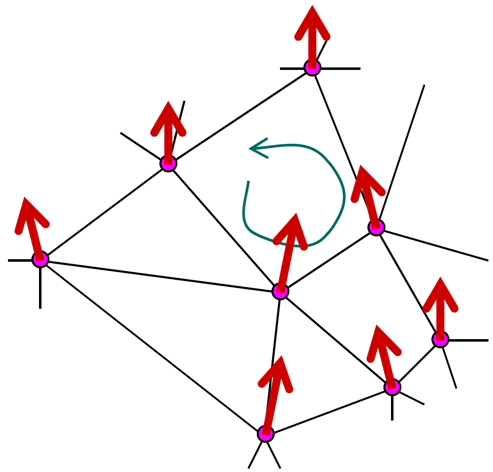
Vertex[N] = [vN.x, vN.y, vN.z]

Face v0, v1, v2, … vN



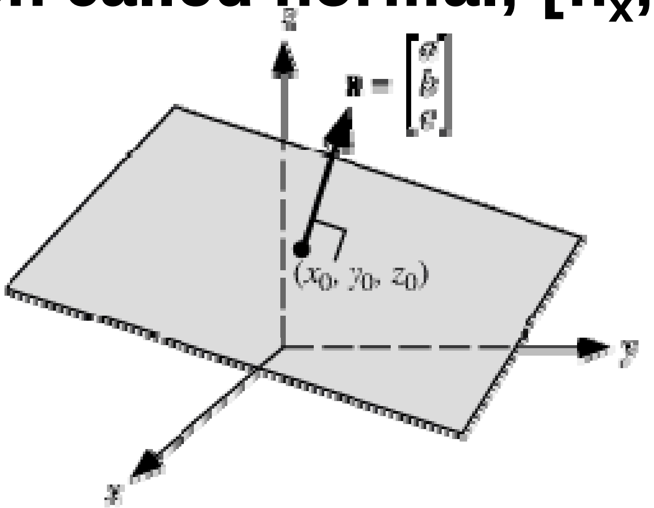
Vertex Specification

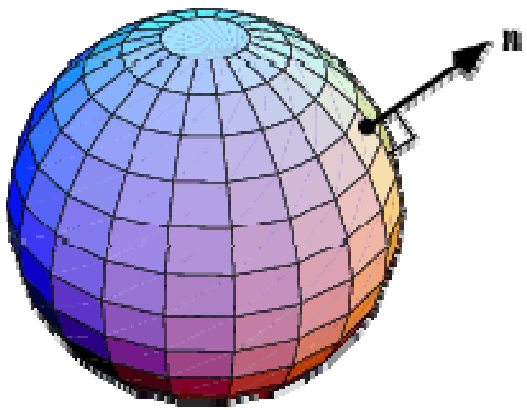
- **Where**
  - **Geometric coordinates [x, y, z]**
- **Attributes**
  - **Color values [r, g, b]**
  - **Texture Coordinates [u, v]**
- **Orientation**
  - **Inside vs. Outside**
  - **Encoded implicitly in ordering**
- **Geometry nearby**
  - **Often we'd like to "fake" a more complex shape than our true faceted (piecewise-planar) model**
  - **Required for lighting and shading in OpenGL**



Normal Vector

- **Often called normal, $[n_x, n_y, n_z]$**





- **Normal to a surface is a vector perpendicular to the surface**
  - **Will be used in illumination**
- **Normalized:** $$\hat{n} = \frac{[n_x, n_y, n_z]}{\sqrt{n_x^2 + n_y^2 + n_z^2}}$$

Drawing Faces in OpenGL

```
glBegin(GL_POLYGON);
foreach (Vertex v in Face) {
  glColor4d(v.red, v.green, v.blue, v.alpha);
  glNormal3d(v.norm.x, v.norm.y, v.norm.z);
  glTexCoord2d(v.texture.u, v.texture.v);
  glVertex3d(v.x, v.y, v.z);
}
glEnd();
```

- Heavy-weight model
  - Attributes specified for every vertex
- Redundant
  - Vertex positions often shared by at least 3 faces
  - Vertex attributes are often face attributes (e.g. face normal)

KAIST

3D File Formats

- MAX – Studio Max
- DXF – AutoCAD
  - Supports 2-D and 3-D; binary
- 3DS – 3D studio
  - Flexible; binary
- VRML – Virtual reality modeling language
  - ASCII – Human readable (and writeable)
- OBJ – Wavefront OBJ format
  - ASCII
  - Extremely simple
  - Widely supported

KAIST

OBJ File Tokens

- **File tokens are listed below**

 # some text

Rest of line is a comment

v *float float float*

A single vertex's geometric position in space

vn *float float float*

A normal

vt *float float*

A texture coordinate

OBJ Face Varieties

f *int int int* ... **(vertex only)**

or

f *int/int int/int int/int* . . . **(vertex & texture)**

or

f *int/int/int int/int/int int/int/int* ... **(vertex, texture, & normal)**

- **The arguments are 1-based indices into the arrays**
  - **Vertex positions**
  - **Texture coordinates**
  - **Normals, respectively**

KAIST

OBJ Example

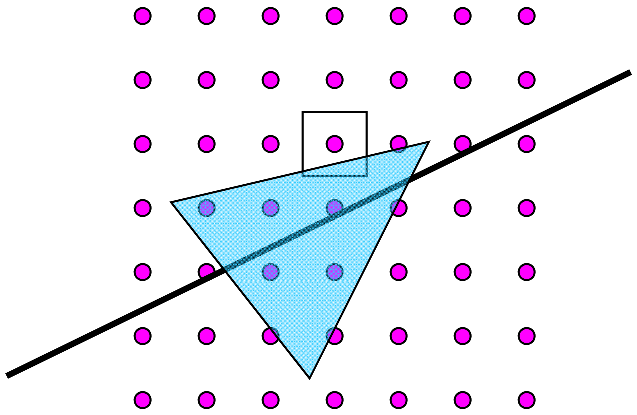
- Vertices followed by faces
  - Faces reference previous vertices by integer index
  - 1-based

```
# A simple cube
v 1 1 1
v 1 1 -1
v 1 -1 1
v 1 -1 -1
v -1 1 1
v -1 1 -1
v -1 -1 1
v -1 -1 -1
f 1 3 4
f 5 6 8
f 1 2 6
f 3 7 8
f 1 5 7
f 2 4 8
```

Lecture: Rasterization

Primitive Rasterization

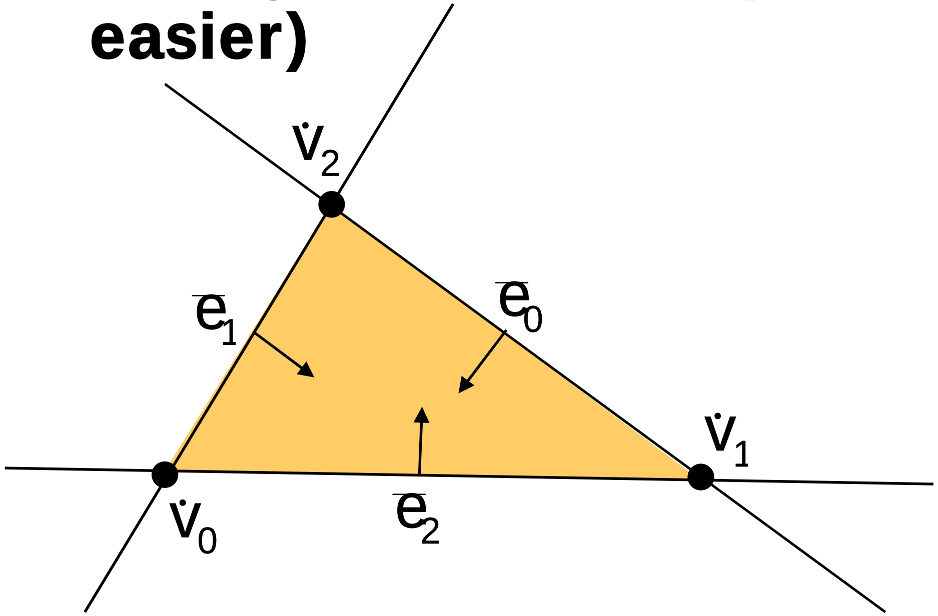
- Rasterization converts vertex representation to pixel representation



- Coverage determination
  - Computes which pixels (samples) belong to a primitive
- Parameter interpolation
  - Computes parameters at covered pixels from parameters associated with primitive vertices

Why Triangles?

- **Triangles are simple**
  - **Simple representation for a surface element (3 points or 3 edge equations)**
  - **Triangles are linear (makes computations easier)**

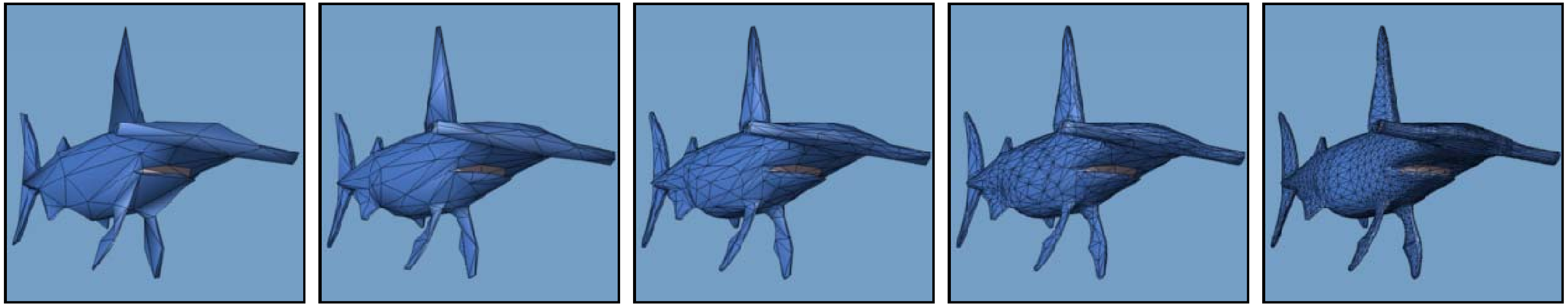


$$T = (\dot{v}_0, \dot{v}_1, \dot{v}_2)$$

$$T = (\bar{e}_0, \bar{e}_1, \bar{e}_2)$$

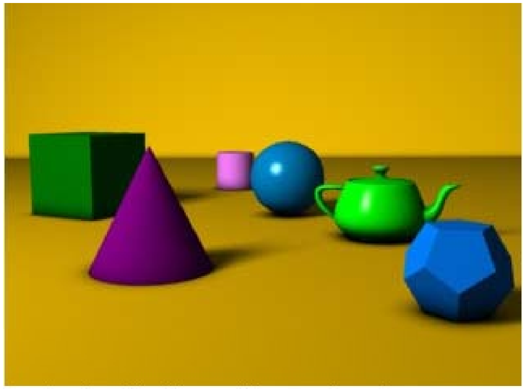
Why Triangles?

- **Triangles can approximate any 2-dimensional shape (or 3D surface)**
  - **Polygons are a locally linear (planar) approximation**
- **Improve the quality of fit by increasing the number edges or faces**



Z-Buffering

- When rendering multiple triangles we need to determine which triangles are visible
- Use z-buffer to resolve visibility
  - Stores the depth at each pixel
- Initialize z-buffer to 1
  - Post-perspective z values lie between 0 and 1
- Linearly interpolate depth (z_{tri}) across triangles
- If $z_{tri}(x,y)$ < zBuffer[x][y]
  write to pixel at (x,y)
  zBuffer[x][y] = $z_{tri}(x,y)$



A simple three dimensional scene

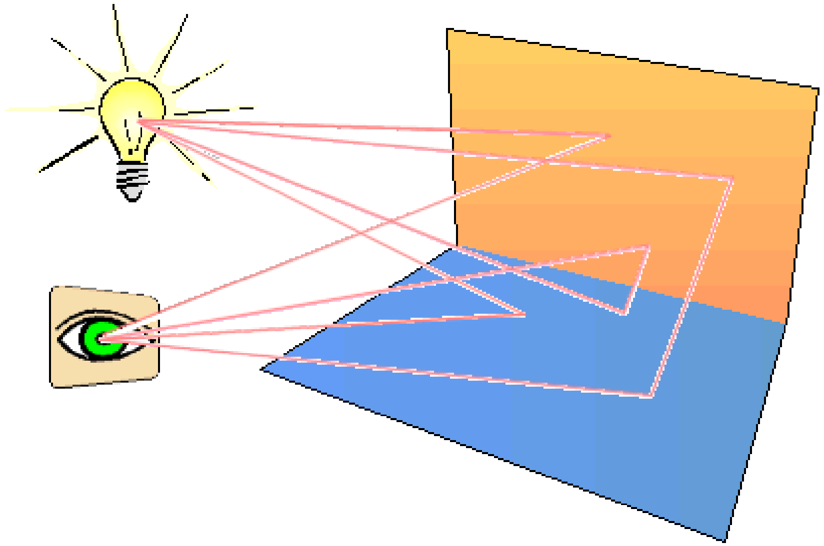
Z-buffer representation

image from wikipedia.com

Lecture: Illumination

Illumination Models

- Illumination
  - Light energy transport from light sources between surfaces via direct and indirect paths
- Shading
  - Process of assigning colors to pixels



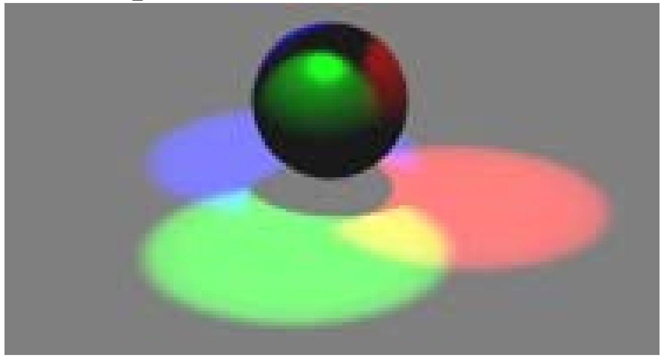
KAIST

Illumination Models

- Physically-based
  - Models based on the actual physics of light's interactions with matter
- Empirical
  - Simple formulations that approximate observed phenomenon

Two Components of Illumination

- **Light sources:**
  - **Emittance spectrum (color)**
  - **Geometry (position and direction)**
  - **Directional attenuation**

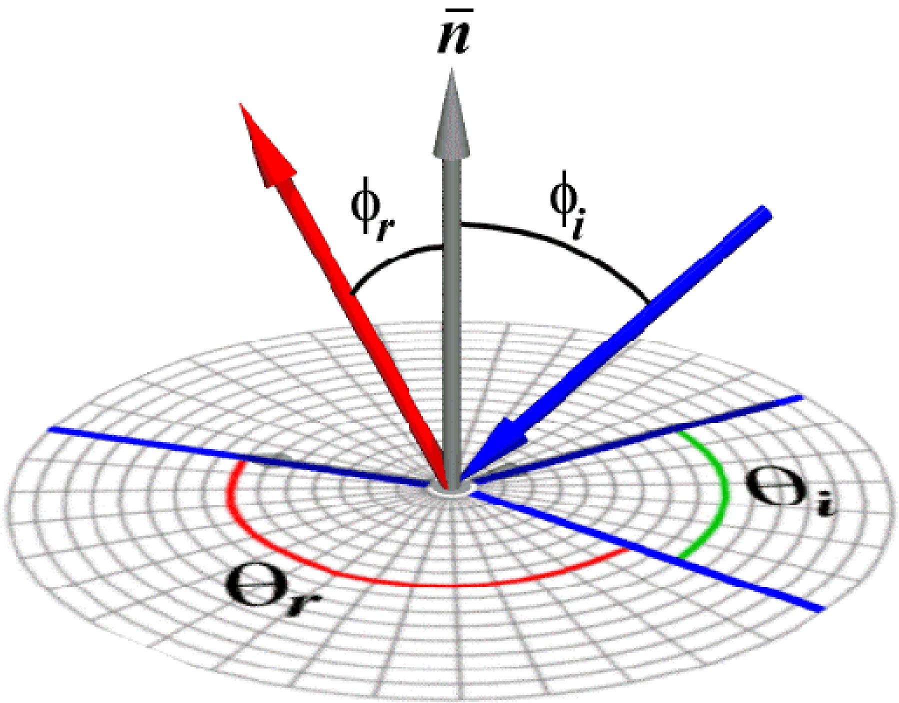


- **Surface properties:**
  - **Reflectance spectrum (color)**
  - **Geometry (position, orientation, and micro-structure)**
  - **Absorption**

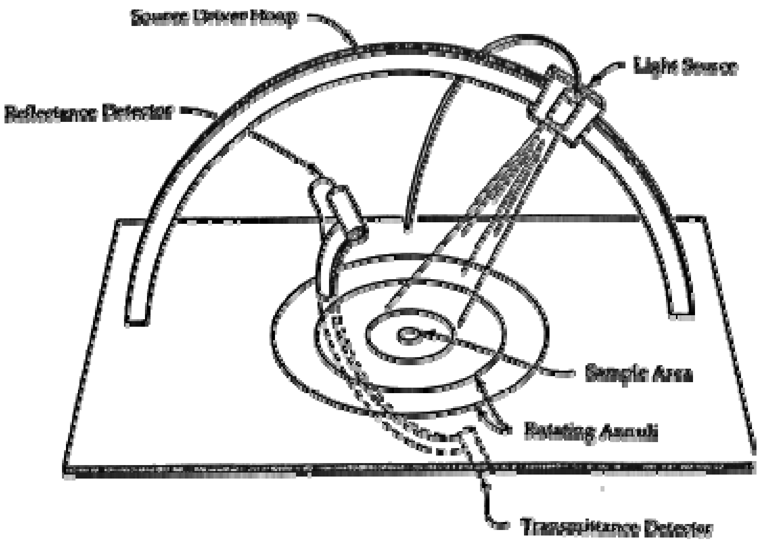
Bi-Directional Reflectance Distribution Function (BRDF)

- **Describes the transport of irradiance to radiance**

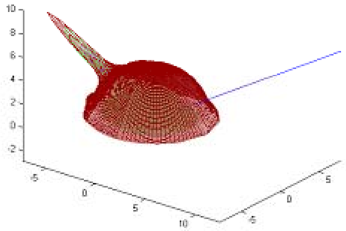
$$\rho(\theta_r, \phi_r, \theta_i, \phi_i)$$



Measuring BRDFs



- **Goniophotometer**
  - **One 4D measurement at a time (slow)**



How to use BRDF Data?

Nickel

Hematite

Gold Paint

Pink Felt

One can make direct use of acquired BRDFs in a renderer

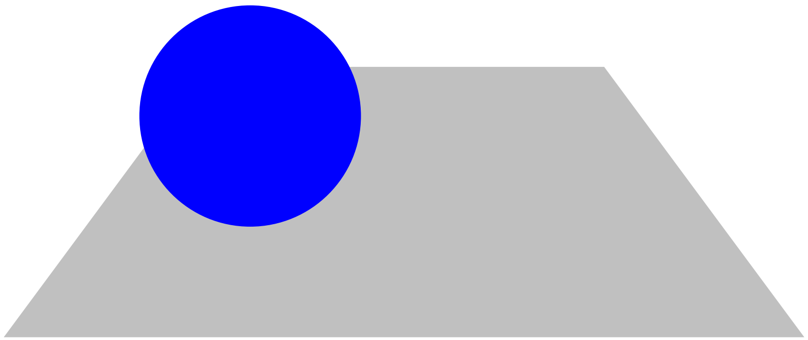
KAIST

Two Components of Illumination

- Simplifications used by most computer graphics systems:
  - Compute only direct illumination from the emitters to the reflectors of the scene
  - Ignore the geometry of light emitters, and consider only the geometry of reflectors

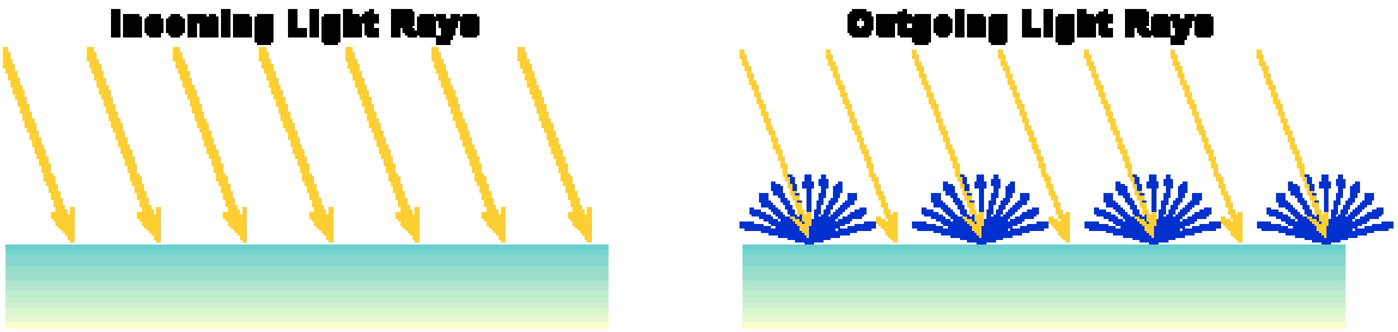
Ambient Light Source

- A simple <u>hack</u> for indirect illumination
  - Incoming ambient illumination ($I_{i,a}$) is constant for all surfaces in the scene
  - Reflected ambient illumination ($I_{r,a}$) depends only on the surface's ambient reflection coefficient (k_a) and not its position or orientation $I_{r,a} = k_a I_{i,a}$
  - These quantities typically specified as (R, G, B) triples



Ideal Diffuse Reflection

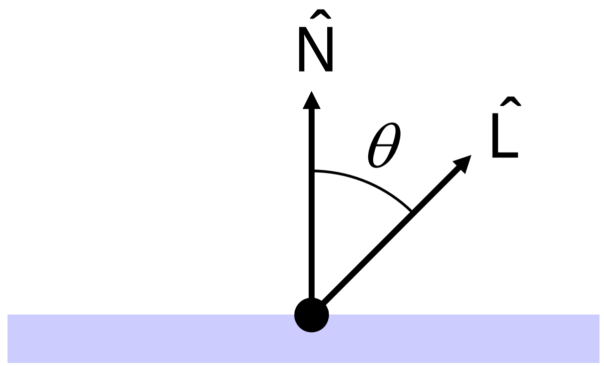
- Ideal diffuse reflectors (e.g., chalk)
  - Reflect uniformly over the hemisphere
  - Reflection is view-independent
  - Very rough at the microscopic level
- Follow Lambert's cosine law

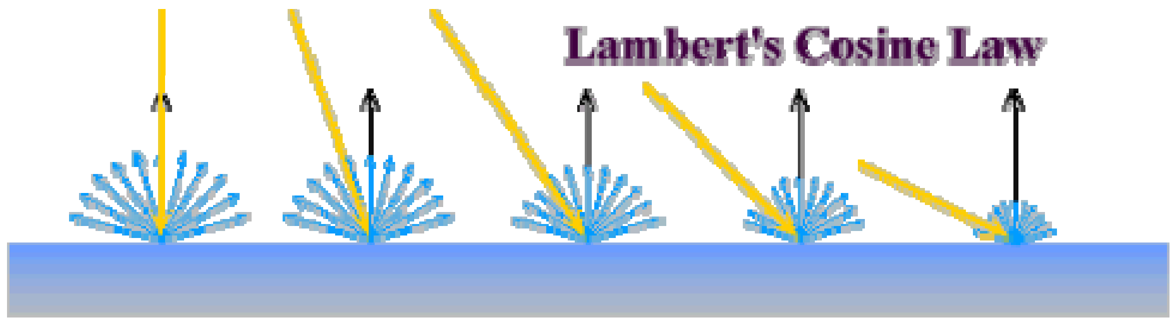


Lambert's Cosine Law

- **The reflected energy from a small surface area from illumination arriving from direction $\hat{L}$ is proportional to the cosine of the angle between $\hat{L}$ and the surface normal**

$$I_r \approx I_i \cos\theta$$
$$\approx I_i(\hat{N} \bullet \hat{L})$$





Specular Reflection

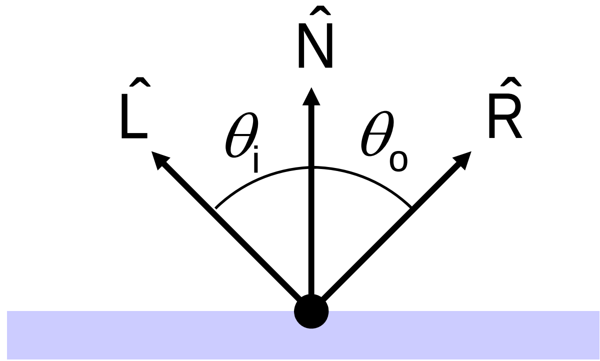
- Specular reflectors have a bright, view dependent highlight
  - E.g., polished metal, glossy car finish, a mirror
  - At the microscopic level a specular reflecting surface is very smooth
  - Specular reflection obeys Snell's law

Image source: astochimp.com and wiki

Snell's Law

- **The relationship between the angles of the incoming and reflected rays with the normal is given by:**

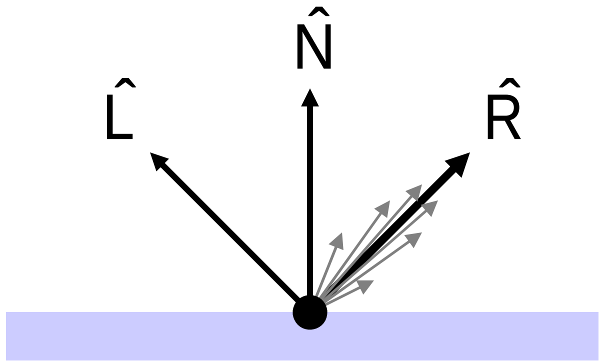
$$n_i \sin\theta_i = n_o \sin\theta_o$$

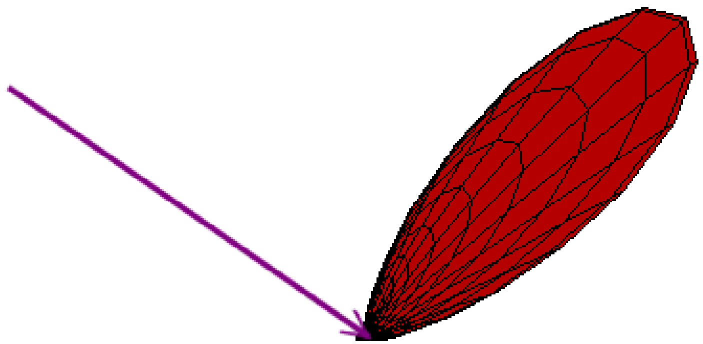


  - **n_i and n_o are the indices of refraction for the incoming and outgoing ray, respectively**
  - **Reflection is a special case where $n_i = n_o$ so $\theta_o = \theta_i$**
  - **The incoming ray, the surface normal, and the reflected ray all lie in a common plane**

Non-Ideal Reflectors

- **Snell's law applies only to *ideal* specular reflectors**
  - **Roughness of surfaces causes highlight to "spread out"**
  - **Empirical models try to simulate the appearance of this effect, without trying to capture the physics of it**

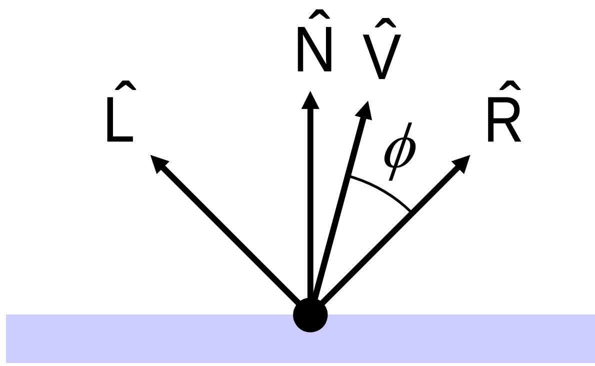




Phong Illumination

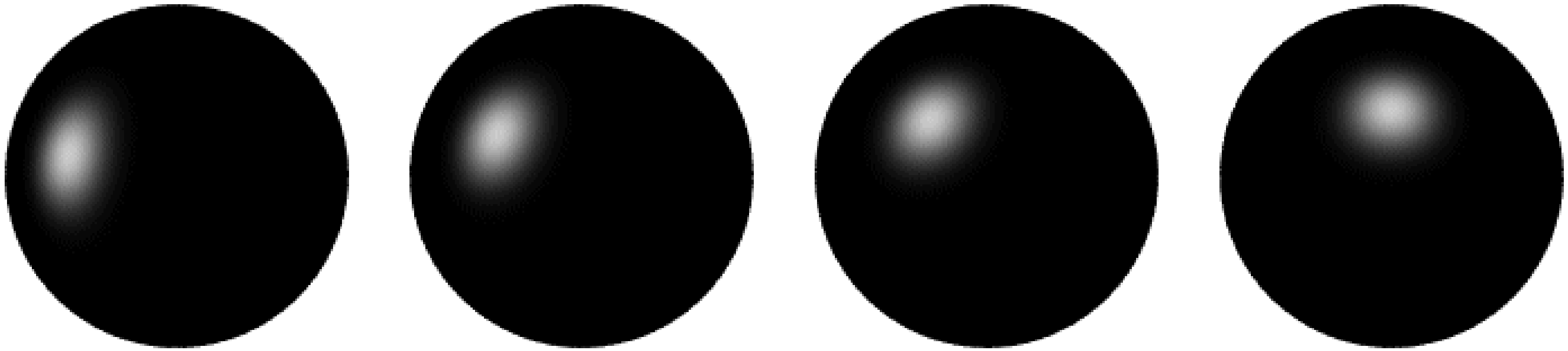
- **One of the most commonly used illumination models in computer graphics**
  - **Empirical model and does not have no physical basis**

$$I_r = k_s I_i (\cos\phi)^{n_s}$$
$$= k_s I_i (\hat{V} \bullet \hat{R})^{n_s}$$

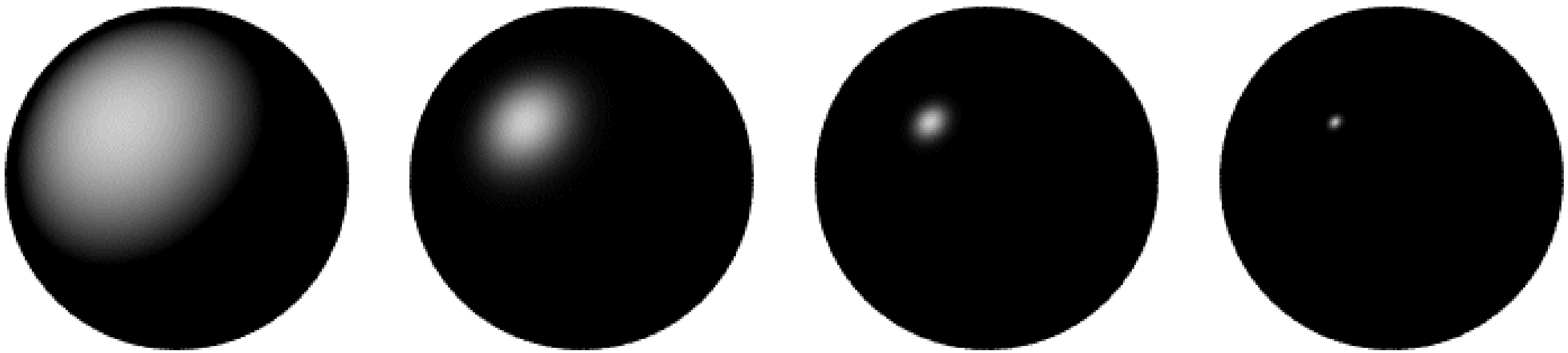


- $(\hat{V})$ **is the direction to the viewer**
  - $(\hat{V} \bullet \hat{R})$ **is clamped to [0,1]**
  - **The specular exponent n_s controls how quickly the highlight falls off**

Examples of Phong



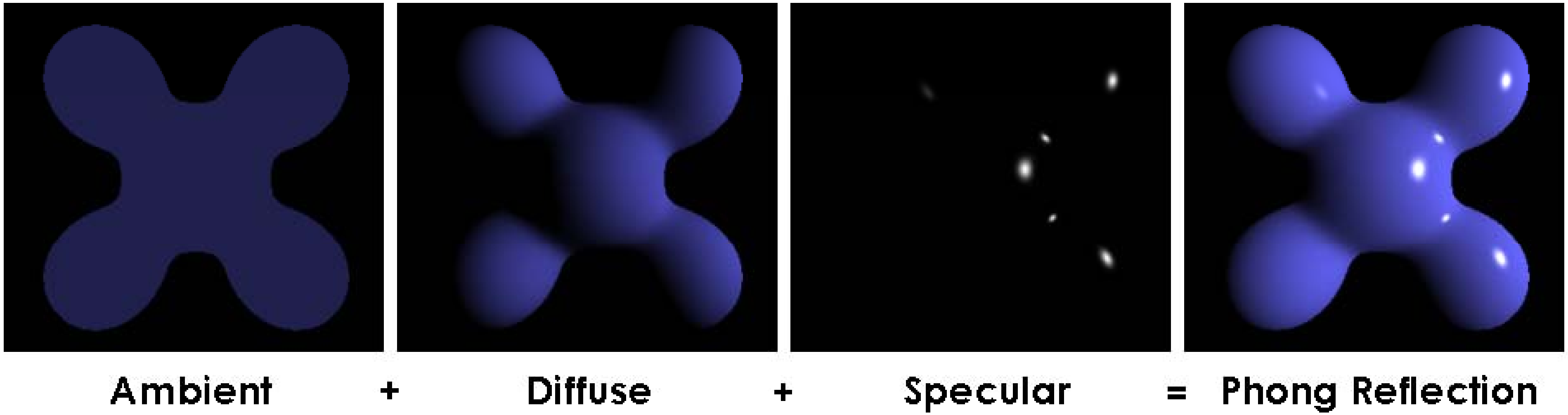
varying light direction



varying specular exponent

Putting it All Together

$$I_r = \sum_{j=1}^{numLights} \left(k_a^j I_a^j + k_d^j I_d^j \max((\hat{N} \bullet \hat{L}_j), 0) + k_s^j I_s^j \max((\hat{V} \bullet \hat{R})^{n_s}, 0)\right)$$



From Wikipedia

OpenGL's Illumination Model

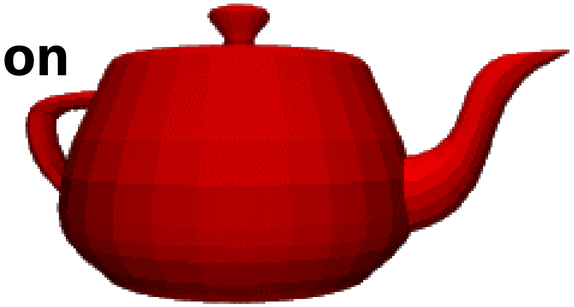
$$I_r = \sum_{j=1}^{numLights} (k_a^j I_a^j + k_d^j I_d^j \max((\hat{N} \bullet \hat{L}_j), 0) + k_s^j I_s^j \max((\hat{V} \bullet \hat{R})^{n_s}, 0))$$

- **Problems with empirical models:**
  - **What are the coefficients for copper?**
  - **What are k_a, k_s, and n_s?**
    Are they measurable quantities?
  - **Is my picture accurate? Is energy conserved?**

KAIST

Flat Shading

- **The simplest shading method**
  - **Applies only one illumination calculation per face**



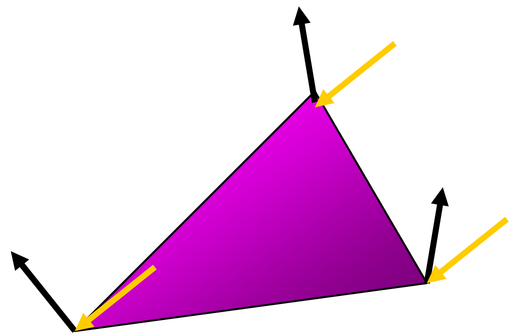
- **Illumination usually computed at the centroid of the face:**

$$centroid = \frac{1}{n}\sum_{i=1}^{n} p_i$$

- **Issues:**
  - **For point light sources the light direction varies over the face**
  - **For specular reflections the viewer direction varies over the facet**

Gouraud Shading

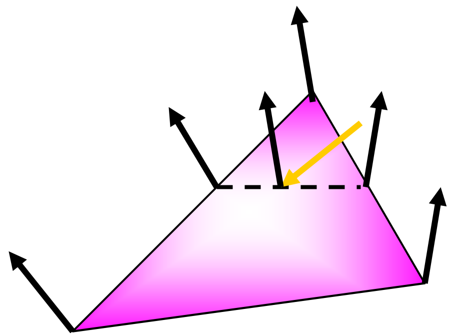
- **Performs the illumination model on vertices and interpolates the intensity of the remaining points on the surface**



Notice that facet artifacts are still visible

Phong Shading

- Surface normal is linearly interpolated across polygonal facets, and the illumination model is applied at every point
  - Not to be confused with Phong's illumination model



- Phong shading will usually result in a very smooth appearance
  - However, evidence of the polygonal model can usually be seen along silhouettes

Local Illumination

- Local illumination models compute the colors of points on surfaces by considering only local properties:
  - Position of the point
  - Surface properties
  - Properties of any light affect it
- No other objects in the scene are considered neither as light blockers nor as reflectors
- Typical of immediate-mode renders, such as OpenGL

KAIST

Global Illumination

- **In the real world, light takes indirect paths**
  - **Light reflects off of other materials (possibly multiple objects)**
  - **Light is blocked by other objects**
  - **Light can be scattered**
  - **Light can be focused**
  - **Light can bend**
- **Harder to model**
  - **At each point we must consider not only every light source, but and other point that might have reflected light toward it**

Lecture: Texture Mapping

Texture Mapping

- Requires lots of geometry to fully represent complex shapes of models
- Add details with image representations

Excerpted from MIT EECS 6.837,
Durand and Cutler

KAIST

The Quest for Visual Realism

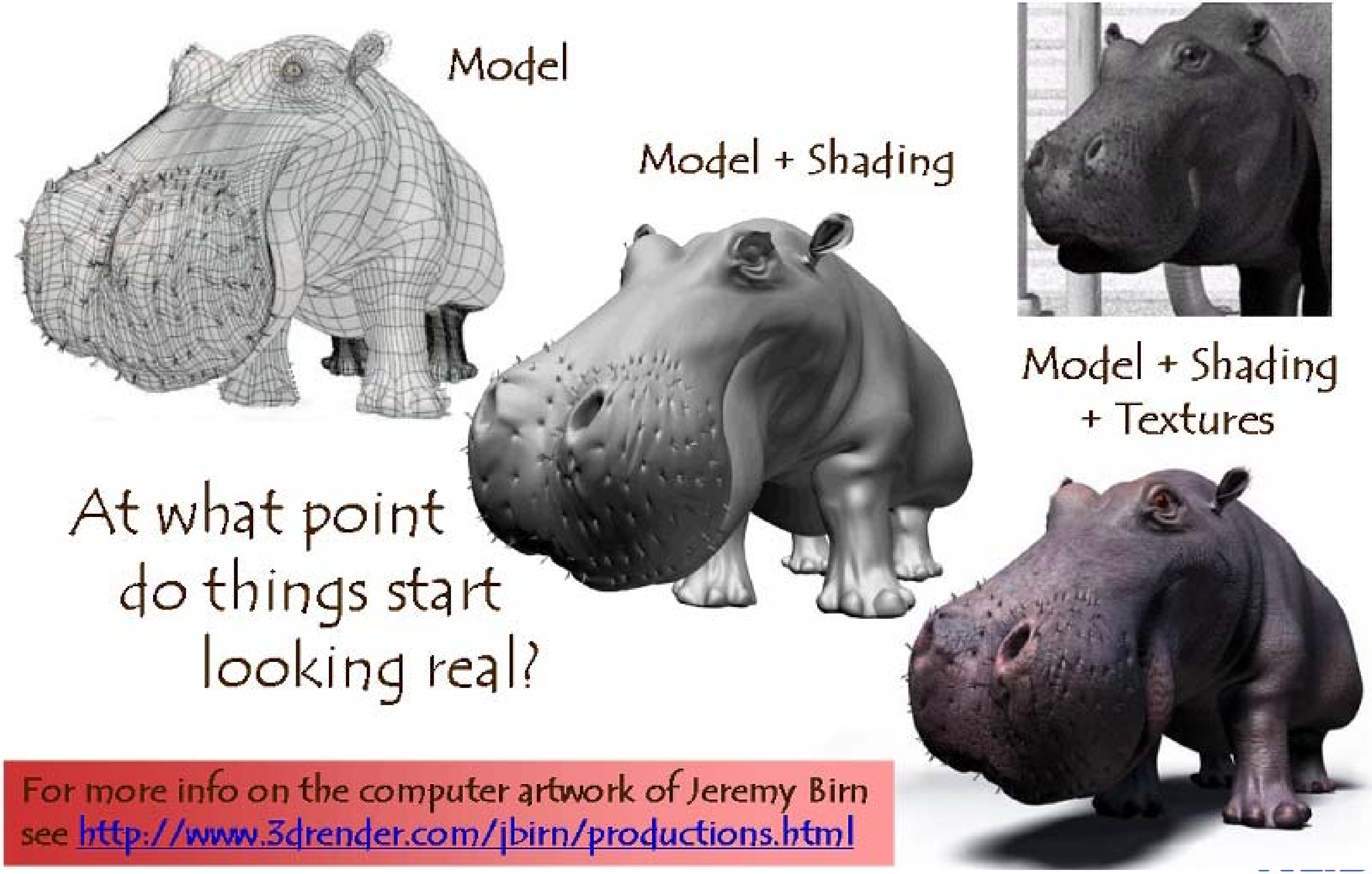
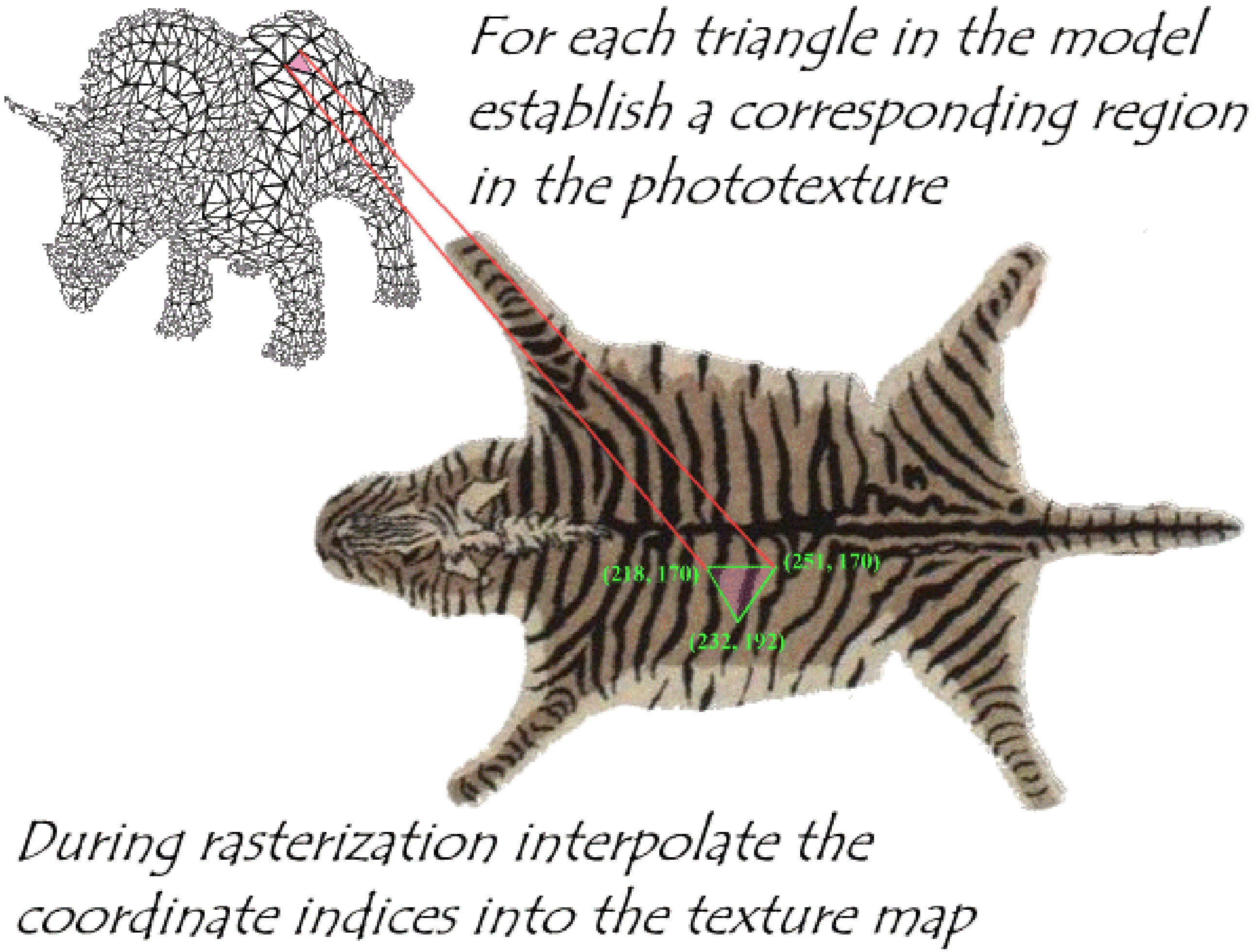
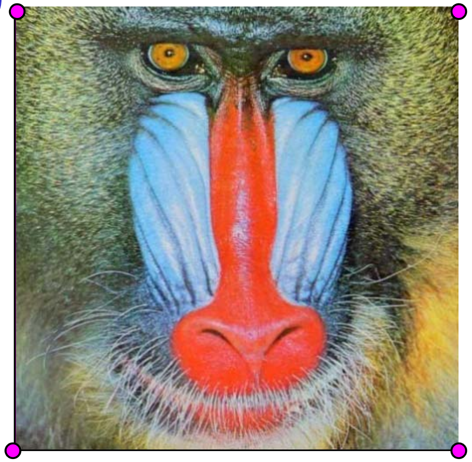


Photo-Textures



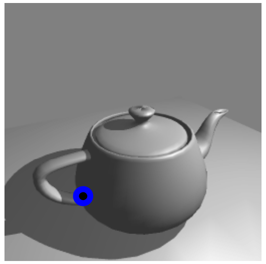
Texture Maps in OpenGL



- **Specify normalized texture coordinates at each of the vertices (u, v)**
- **Texel indices (s,t) = (u, v) · (width, height)**

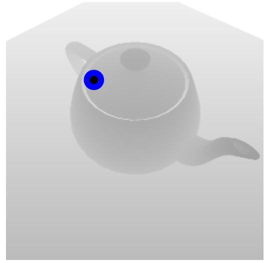
```
glBindTexture(GL_TEXTURE_2D, texID)
glBegin(GL_POLYGON)
  glTexCoord2d(0,1); glVertex2d(-1,-1);
  glTexCoord2d(1,1); glVertex2d( 1,-1);
  glTexCoord2d(1,0); glVertex2d( 1, 1);
  glTexCoord2d(0,0); glVertex2d(-1, 1);
glEnd()
```

Shadow Maps

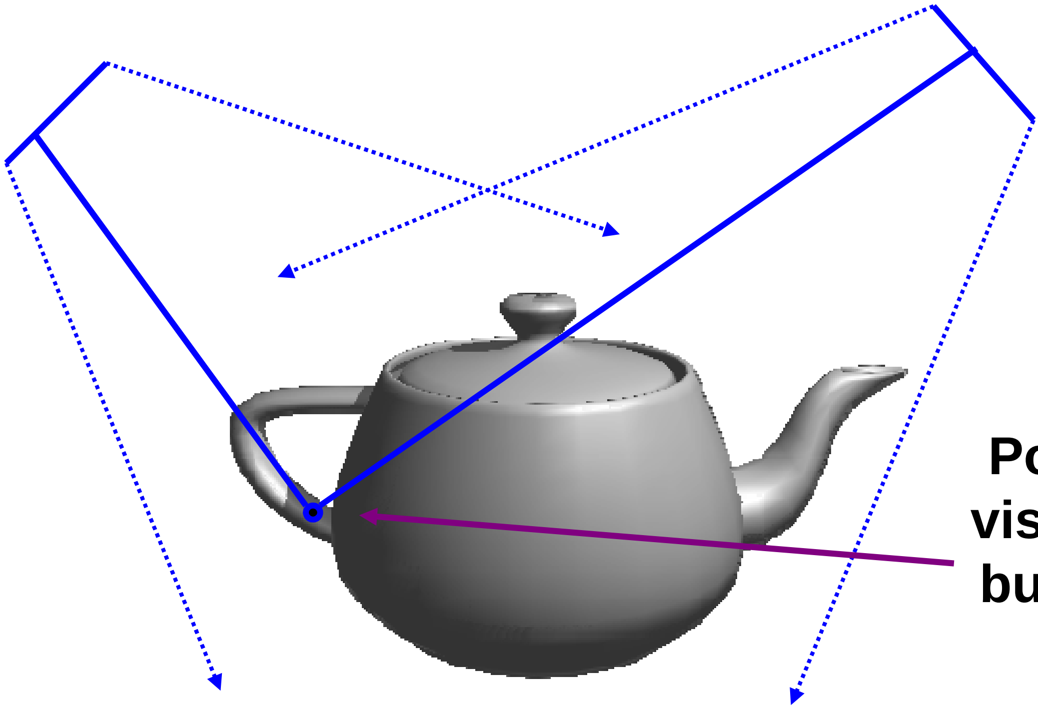


Eye

Use the depth map in the light view to determine if sample point is visible



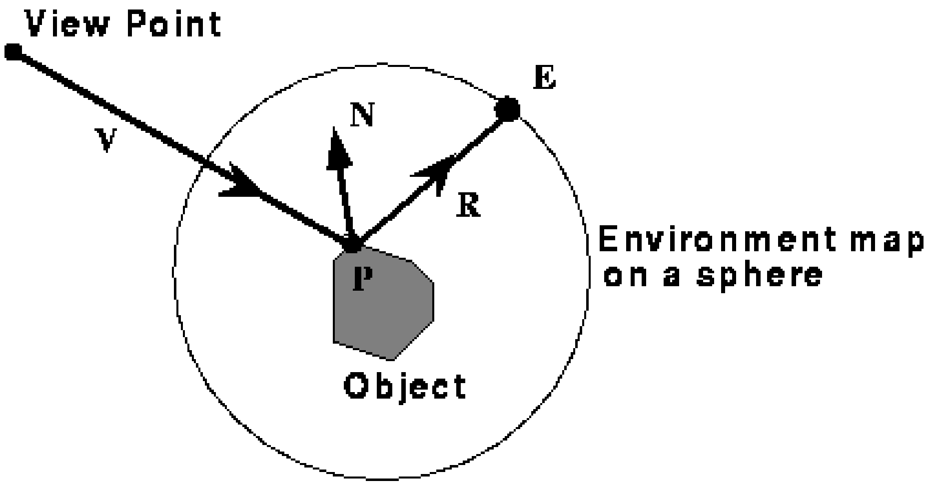
Light



Point in shadow visible to the eye, but not visible to the light

Environment Maps

- **Simulate complex mirror-like objects**
  - **Use textures to capture environment of objects**
  - **Use surface normal to compute texture coordinates**

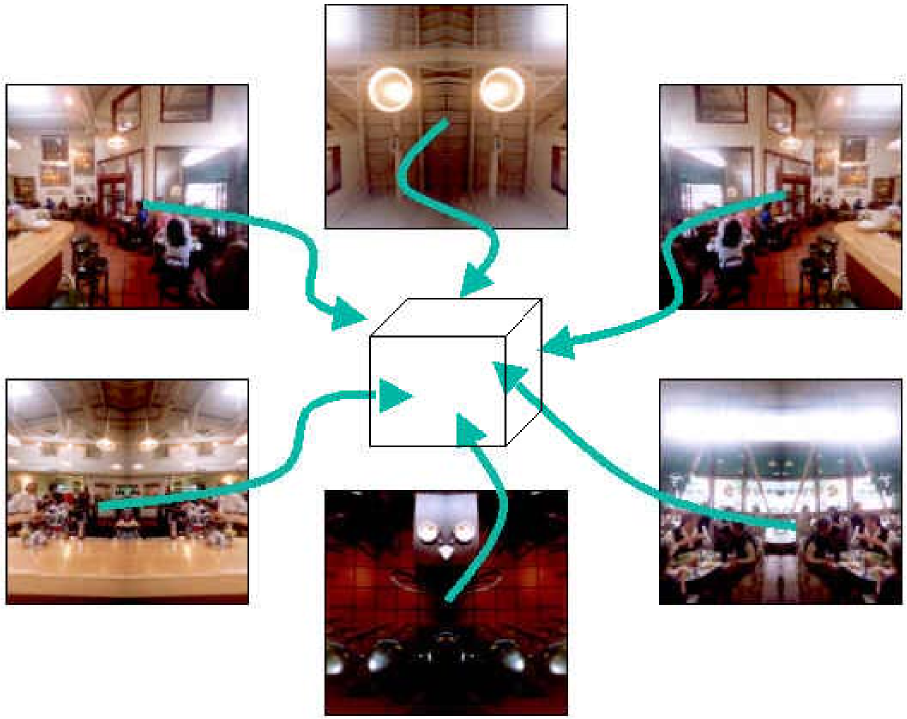


Environment Maps - Example

T1000 in Terminator 2 from Industrial Light and Magic

Cube Maps

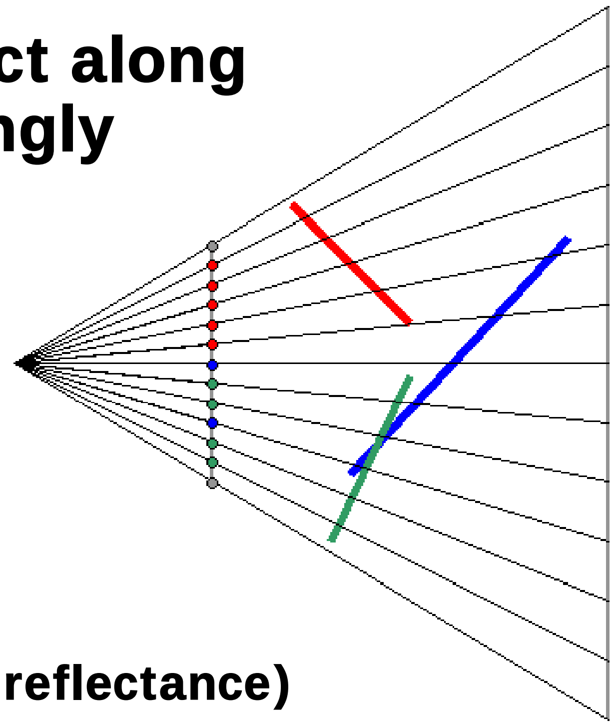
- Maps a viewing direction b and returns an RGB color
  - Use stored texture maps



Lecture: Ray Tracing

Ray Casting

- For each pixel, find closest object along the ray and shade pixel accordingly

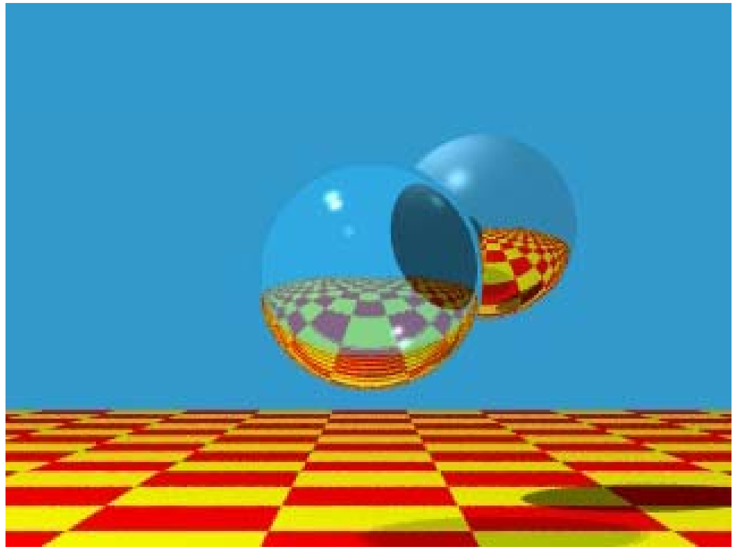


- Advantages
  - Conceptually simple
  - Can support CSG
  - Can take advantage of spatial coherence in scene
  - Can be extended to handle global illumination effects (ex: shadows and reflectance)
- Disadvantages
  - Renderer must have access to entire retained model
  - Hard to map to special-purpose hardware
  - Visibility computation is a function of resolution

Recursive Ray Casting

- **Ray casting generally dismissed early on:**
  - **Takes no advantage of screen space coherence**
  - **Requires costly visibility computation**
  - **Only works for solids**
  - **Forces per pixel illumination evaluations**

- **Gained popularity in when Turner Whitted (1980) recognized that *recursive* ray casting could be used for global illumination effects**



Overall Algorithm of Ray Tracing

- Per each pixel, compute a ray, R

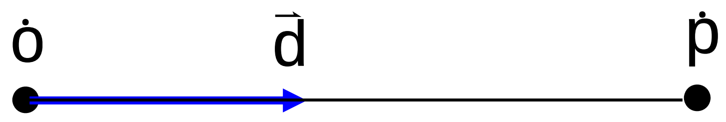
function RayTracing (R)

- Compute an intersection against objects
- If no hit,
  - Return the background color
- Otherwise,
  - Compute shading, c
  - General secondary ray, R'
  - Perform c' = RayTracing (R')
  - Return c+c'

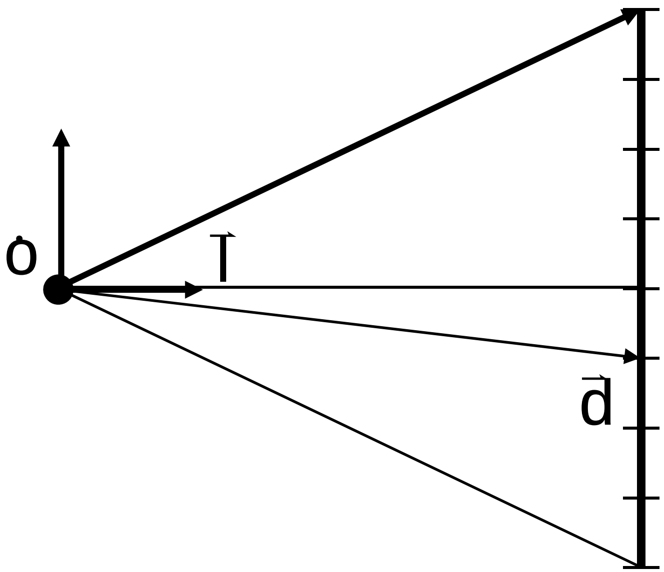
Ray Representation

- We need to compute the first surface hit along a ray
  - Represent ray with origin and direction
  - Compute intersections of objects with ray
  - Return closest object

$$\dot{p}(t) = \dot{o} + t\vec{d}$$

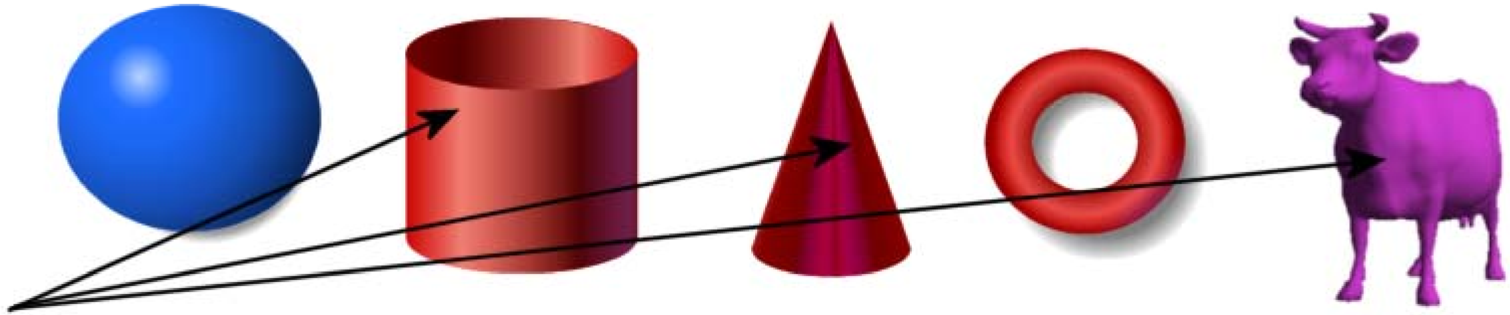


Generating Primary Rays



Intersection Tests

Go through all of the objects in the scene to determine the one closest to the origin of



Strategy: Solve of the intersection of the Ray with a mathematical description of the object

KAIST

Simple Strategy

- **Parametric ray equation**
  - **Gives all points along the ray as a function of the parameter**

$$\dot{p}(t) = \dot{o} + t\vec{d}$$

- **Implicit surface equation**
  - **Describes all points on the surface as the zero set of a function**

$$f(\dot{p}) = 0$$

- **Substitute ray equation into surface function and solve for t**

$$f(\dot{o} + t\vec{d}) = 0$$

KAIST

Ray-Plane Intersection

- **Implicit equation of a plane:**

$$\vec{n} \cdot \vec{p} - d = 0$$

- **Substitute ray equation:**

$$\vec{n} \cdot (\vec{o} + t\vec{d}) - d = 0$$

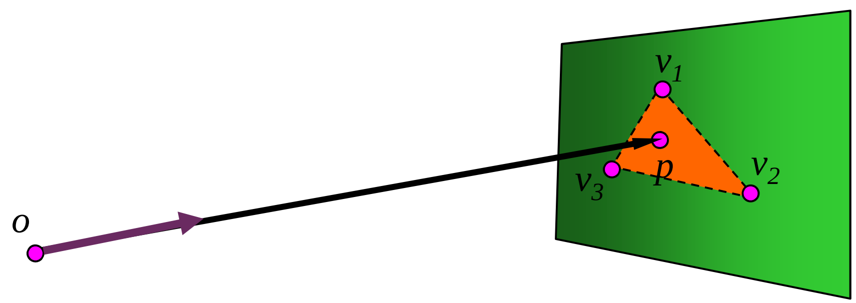
- **Solve for t:**

$$t(\vec{n} \cdot \vec{d}) = d - \vec{n} \cdot \vec{o}$$

$$t = \frac{d - \vec{n} \cdot \vec{o}}{\vec{n} \cdot \vec{d}}$$

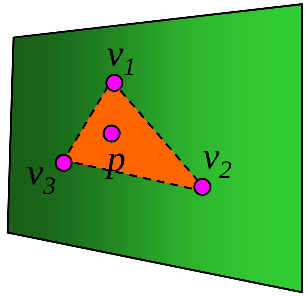
Generalizing to Triangles

- Find of the point of intersection on the plane containing the triangle
- Determine if the point is inside the triangle
  - Barycentric coordinate method
  - Many other methods

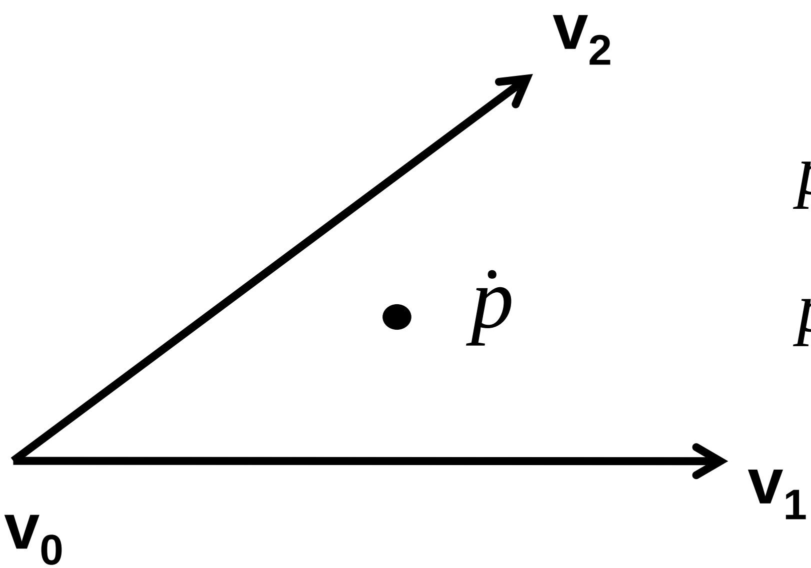


Barycentric Coordinates

- **Points in a triangle have positive barycentric coordinates:**



$$\dot{p} = \alpha\dot{v}_0 + \beta\dot{v}_1 + \gamma\dot{v}_2$$ **,where** $$\alpha + \beta + \gamma = 1$$



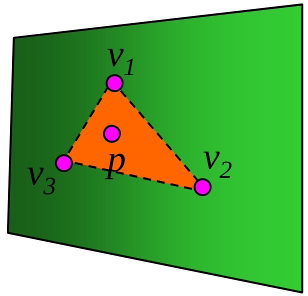
$$\dot{p} = \dot{v}_0 + \beta(\dot{v}_1 - \dot{v}_0) + \gamma(\dot{v}_2 - \dot{v}_0)$$

$$\dot{p} = (1 - \beta - \gamma)\dot{v}_0 + \beta\dot{v}_1 + \gamma\dot{v}_2$$

Barycentric Coordinates

- **Points in a triangle have positive barycentric coordinates:**

$$\dot{p} = \alpha\dot{v}_0 + \beta\dot{v}_1 + \gamma\dot{v}_2 \quad \text{,where } \alpha + \beta + \gamma = 1$$

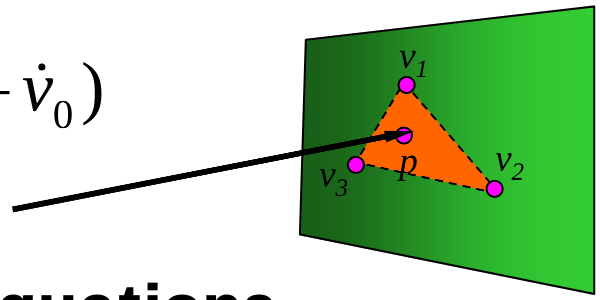


- **Benefits:**
  - **Barycentric coordinates can be used for interpolating vertex parameters (e.g., normals, colors, texture coordinates, etc)**

Ray-Triangle Intersection

- A point in a ray intersects with a triangle

$$\dot{p}(t) = \dot{v}_0 + \beta(\dot{v}_1 - \dot{v}_0) + \gamma(\dot{v}_2 - \dot{v}_0)$$



  - Three unknowns, but three equations
  - Compute the point based on t
  - Then, check whether the point is on the triangle
  - Refer to Sec. 9.3.2 in the textbook for the detail equations