CSE 167: Introduction to Computer Graphics Lecture #5: Rasterization

Jürgen P. Schulze, Ph.D.
University of California, San Diego
Spring Quarter 2015

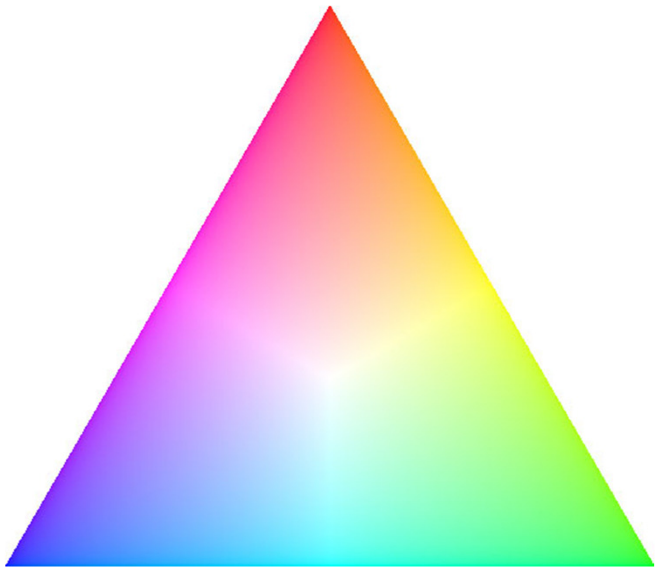
Announcements

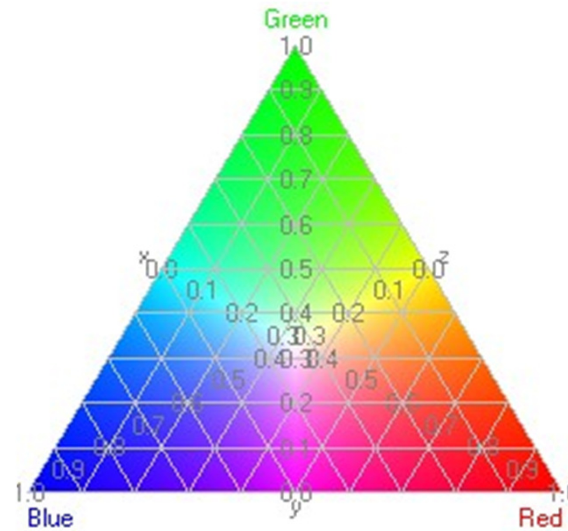
- Project 3 due this Friday at 1pm
- Grading starts at 12:15 in CSE labs 260+270

Lecture Overview

- Barycentric Coordinates

Color Interpolation



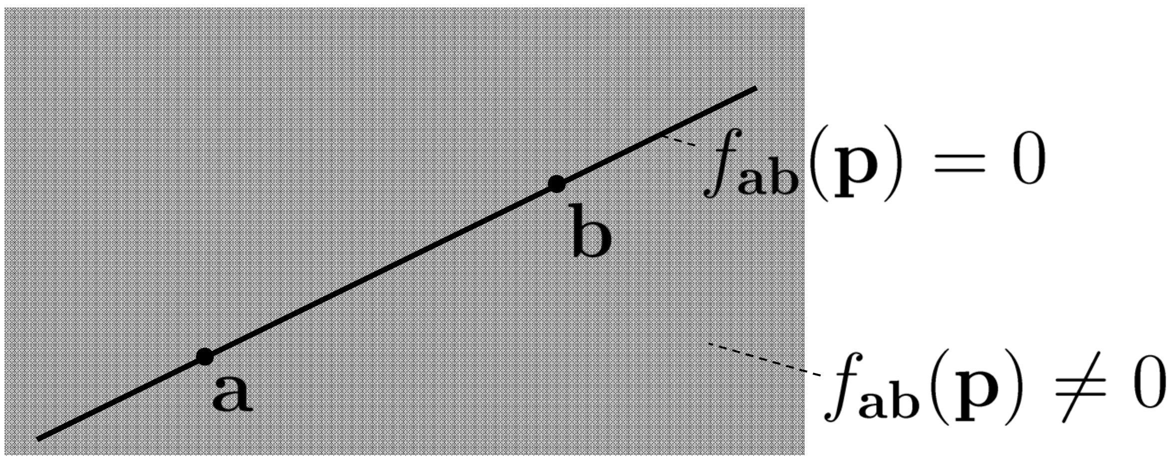


Source: efg's computer lab

- What if a triangle's vertex colors are different?
- Need to interpolate across triangle
  - How to calculate interpolation weights?

Implicit 2D Lines

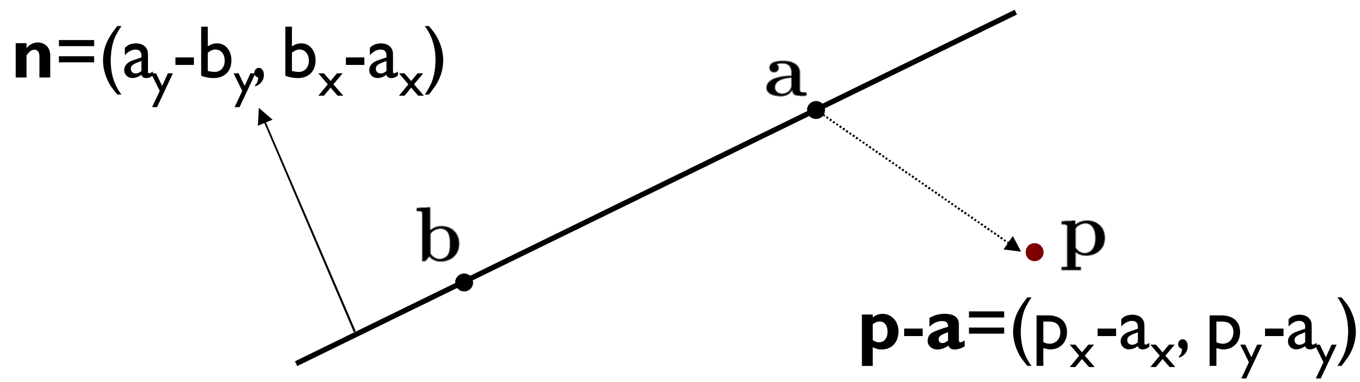
- Given two 2D points **a**, **b**
- Define function $f_{\mathbf{ab}}(\mathbf{p})$ such that $f_{\mathbf{ab}}(\mathbf{p}) = 0$ if **p** lies on the line defined by **a**, **b**



UCSD

Implicit 2D Lines

- Point **p** lies on the line, if **p-a** is perpendicular to the normal **n** of the line



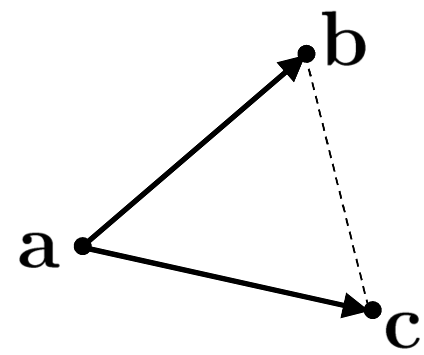
- Use dot product to determine on which side of the line **p** lies. If f(**p**)>0, **p** is on same side as normal, if f(**p**)<0 **p** is on opposite side. If dot product is 0, **p** lies on the line.

$$f_{\mathbf{ab}}(\mathbf{p}) = (a_y - b_y, b_x - a_x) \cdot (p_x - a_x, p_y - a_y)$$

UCSD

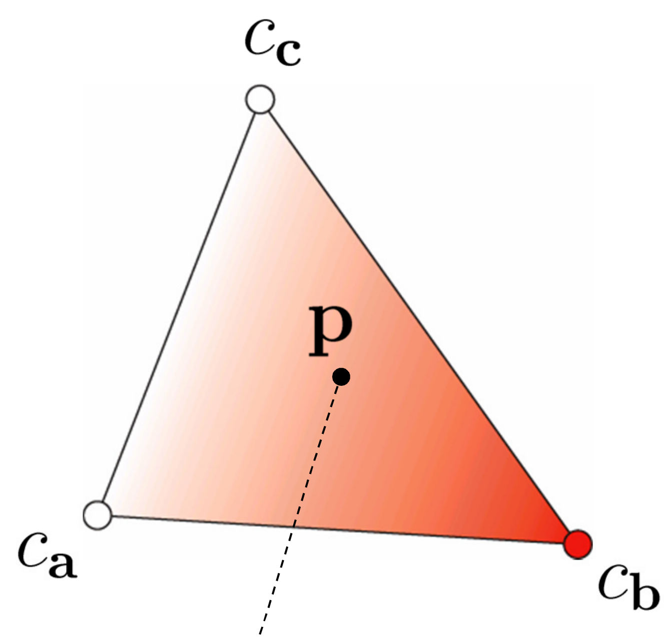
Barycentric Coordinates

- Coordinates for 2D plane defined by triangle vertices **a, b, c**
- Any point **p** in the plane defined by **a, b, c** is
  $\mathbf{p} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$
- Solved for a, b, c:
  $\mathbf{p} = (1 - \beta - \gamma)\,\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$
- We define $\alpha = 1 - \beta - \gamma$
  ➔ $\mathbf{p} = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$
- α, β, γ are called **barycentric** coordinates
- If we imagine masses equal to α, β, γ in the locations of the vertices of the triangle, the center of mass (the Barycenter) is then **p**. This is the origin of the term "barycentric" (introduced 1827 by Möbius)



Barycentric Interpolation

- Interpolate values across triangles, e.g., colors



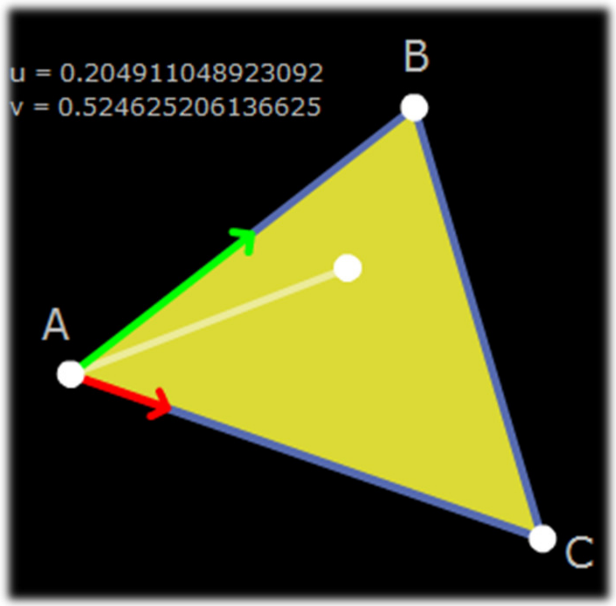
- Done by linear interpolation on triangle:

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

- Works well at common edges of neighboring triangles

Barycentric Coordinates

- Demo:
  - http://adrianboeing.blogspot.com/2010/01/barycentric-coordinates.html

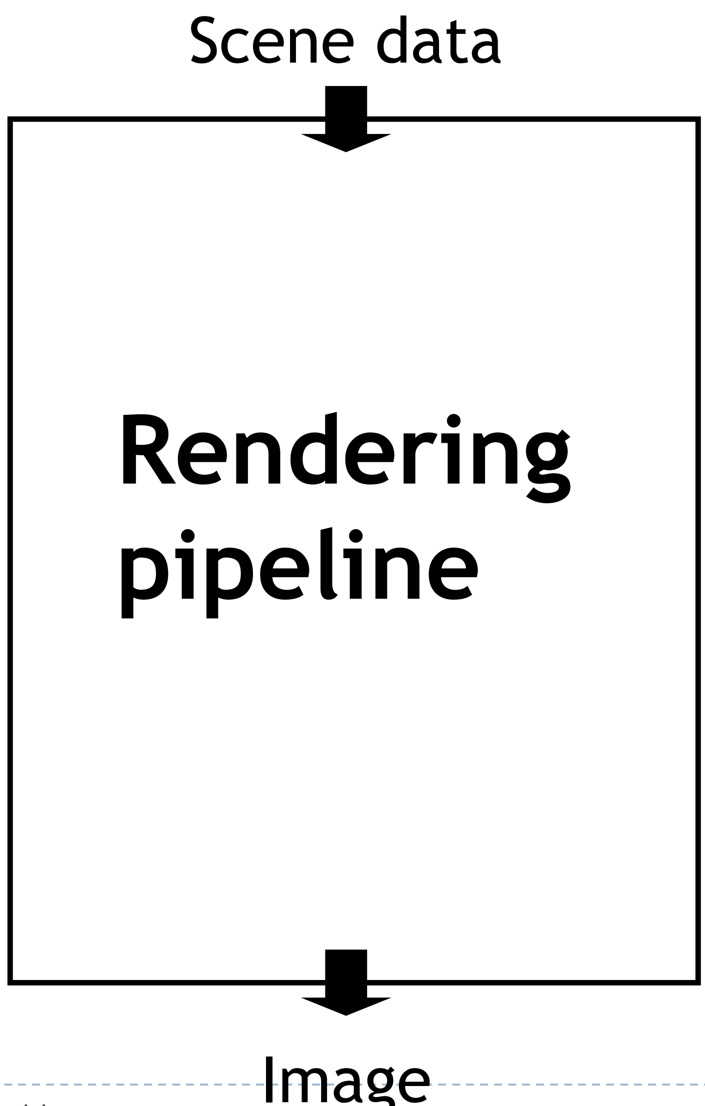


UCSD

Lecture Overview

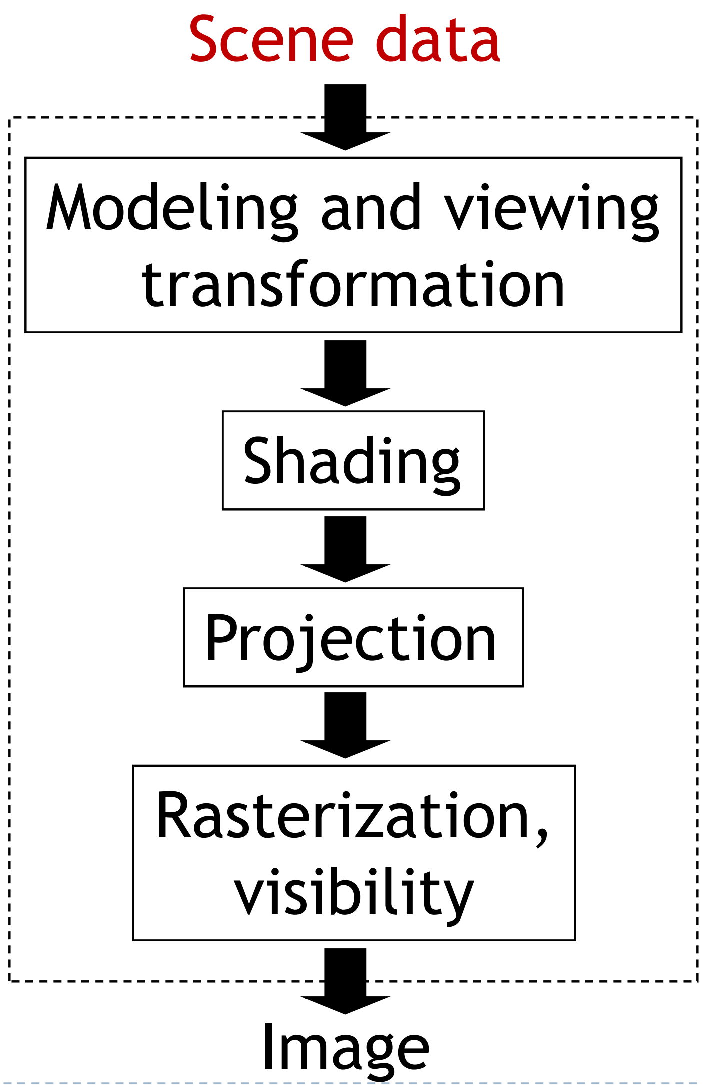
- Rendering Pipeline

Rendering Pipeline

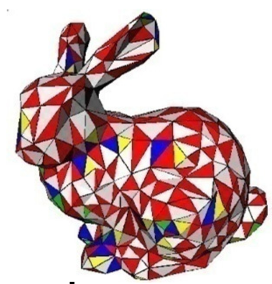


- Hardware and software which draws 3D scenes on the screen
- Consists of several stages
  - Simplified version here
- Most operations performed by specialized hardware (GPU)
- Access to hardware through low-level 3D API (OpenGL, DirectX)
- All scene data flows through the pipeline at least once for each frame

Rendering Pipeline

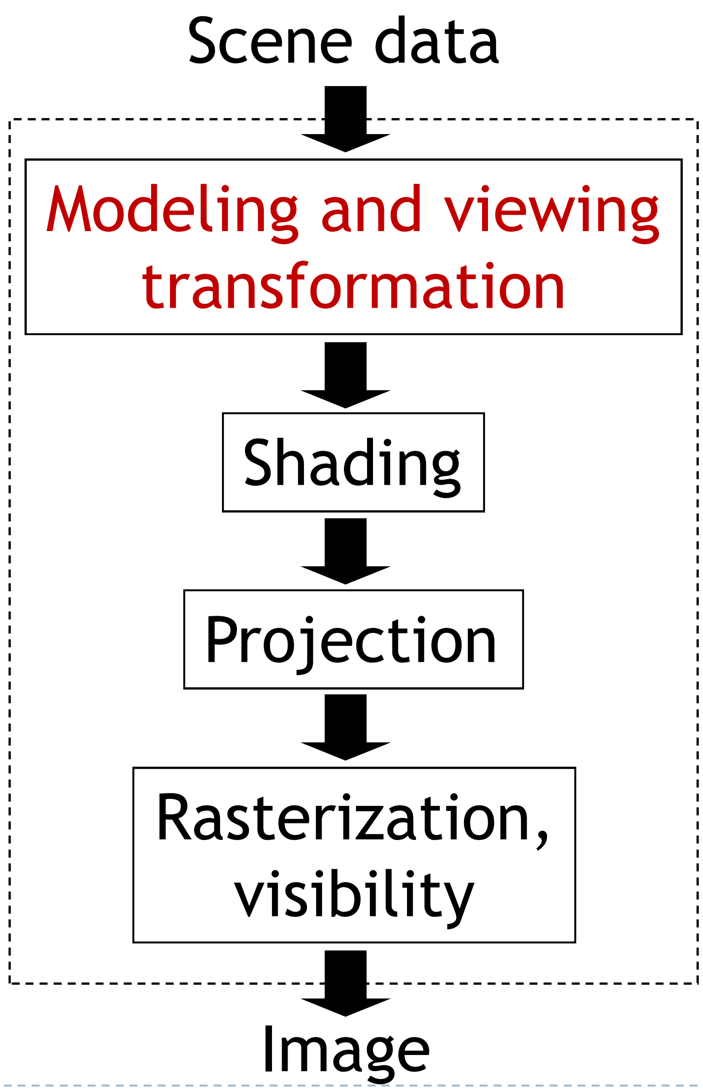


- Textures, lights, etc.
- Geometry
  - Vertices and how they are connected
  - Triangles, lines, points, triangle strips
  - Attributes such as color



- Specified in object coordinates
- Processed by the rendering pipeline one-by-one

Rendering Pipeline



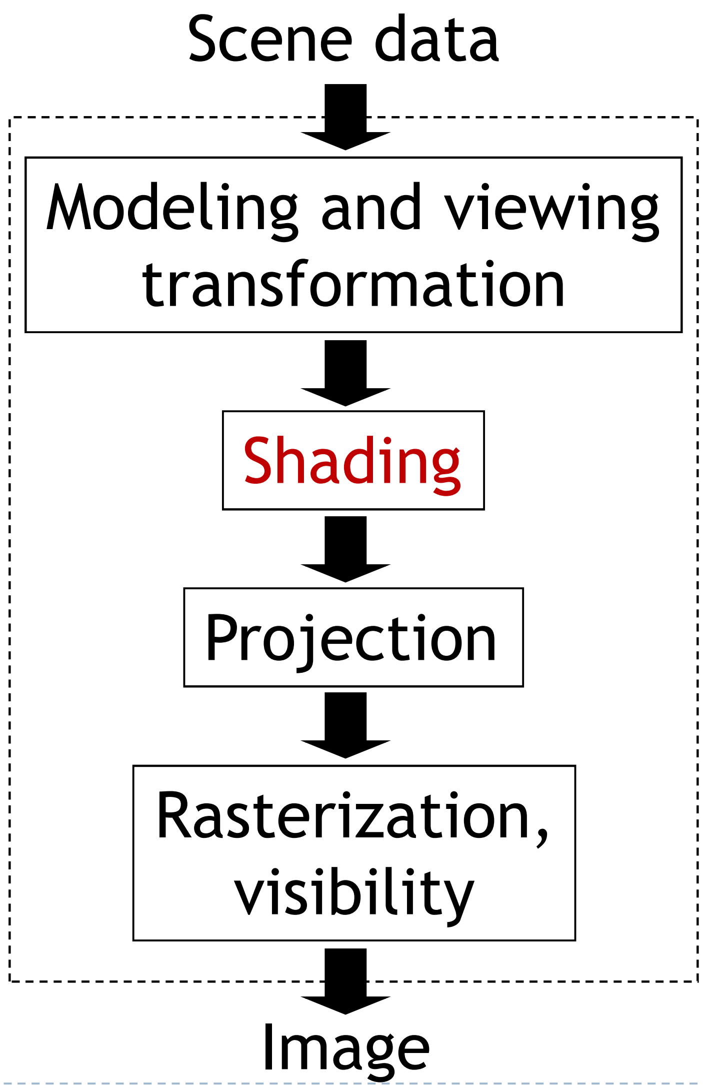
- Transform object to camera coordinates
- Specified by GL_MODELVIEW matrix in OpenGL
- User computes GL_MODELVIEW matrix as discussed

$$\mathbf{p}_{camera} = \mathbf{C}^{-1}\mathbf{M}\mathbf{p}_{object}$$

MODELVIEW matrix

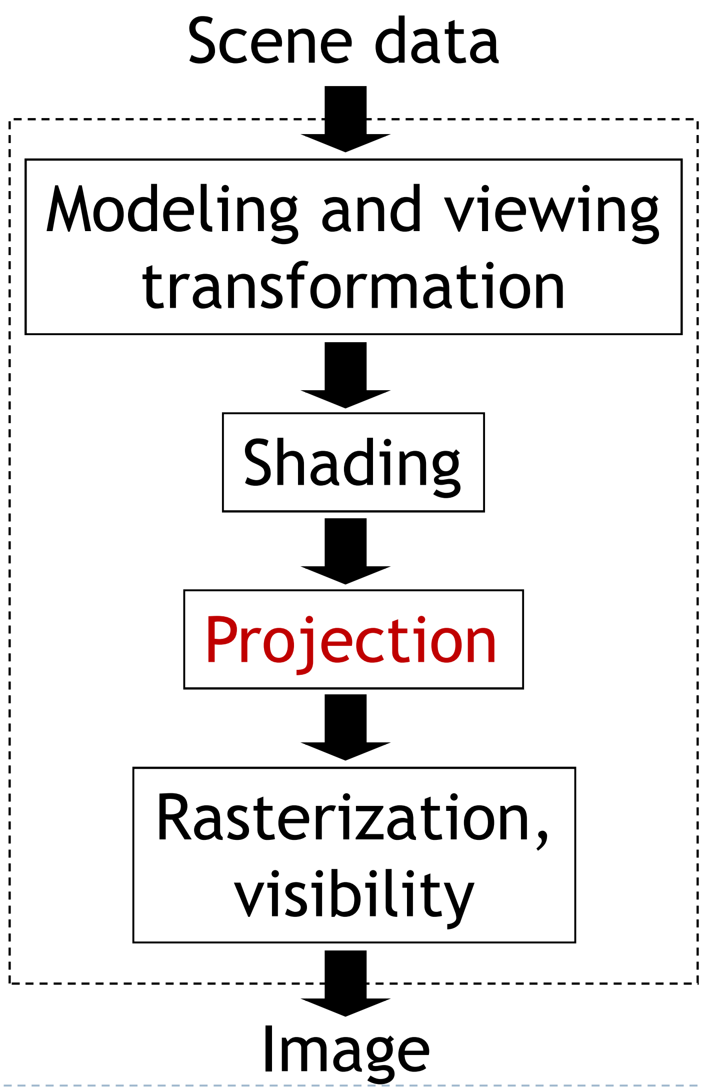
UCSD

Rendering Pipeline



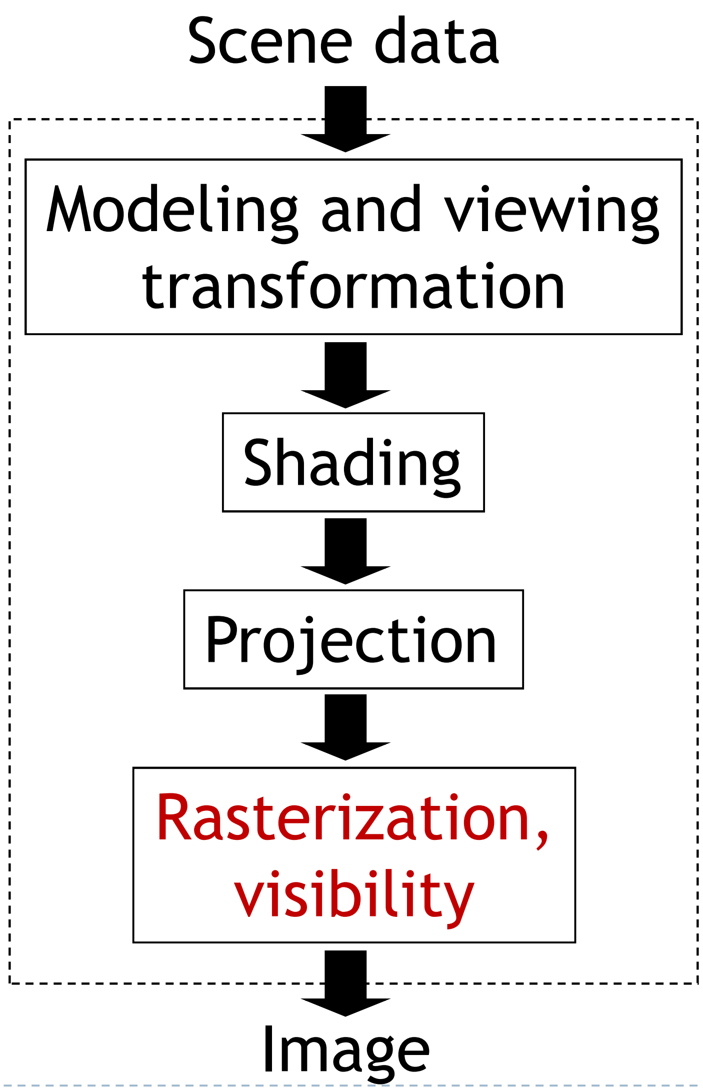
- Look up light sources
- Compute color for each vertex

Rendering Pipeline

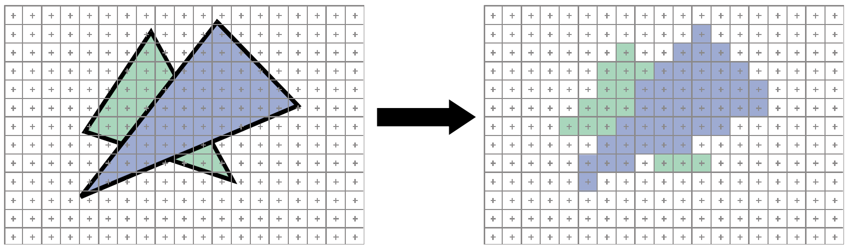


- Project 3D vertices to 2D image positions
- GL_PROJECTION matrix

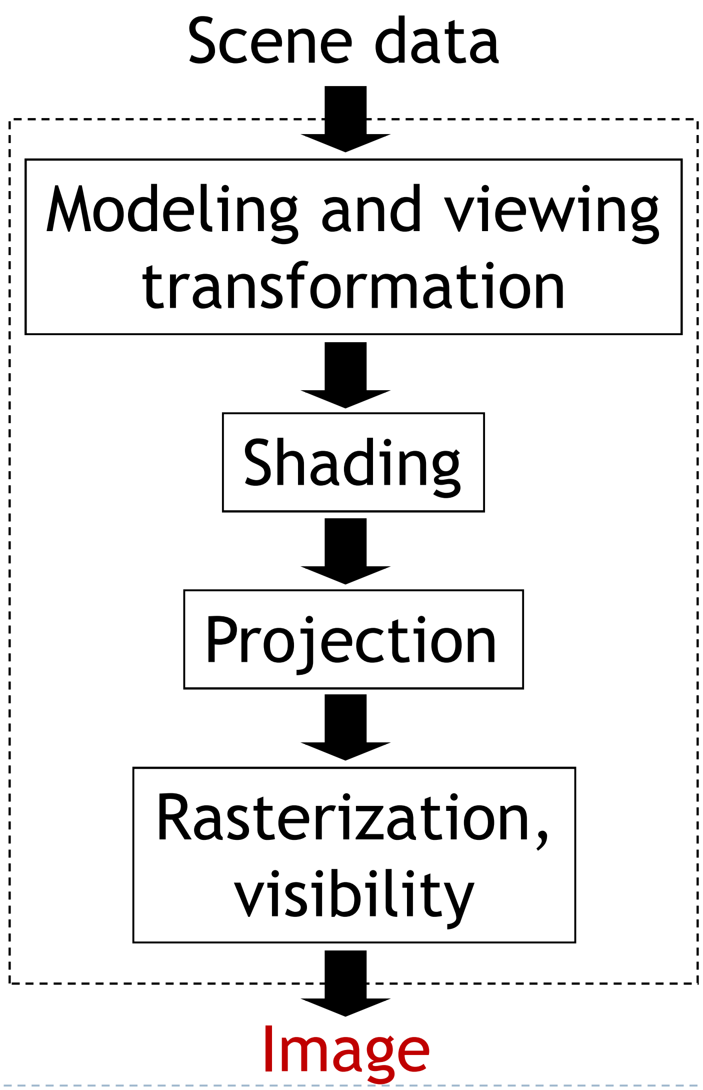
Rendering Pipeline



- Draw primitives (triangles, lines, etc.)
- Determine what is visible



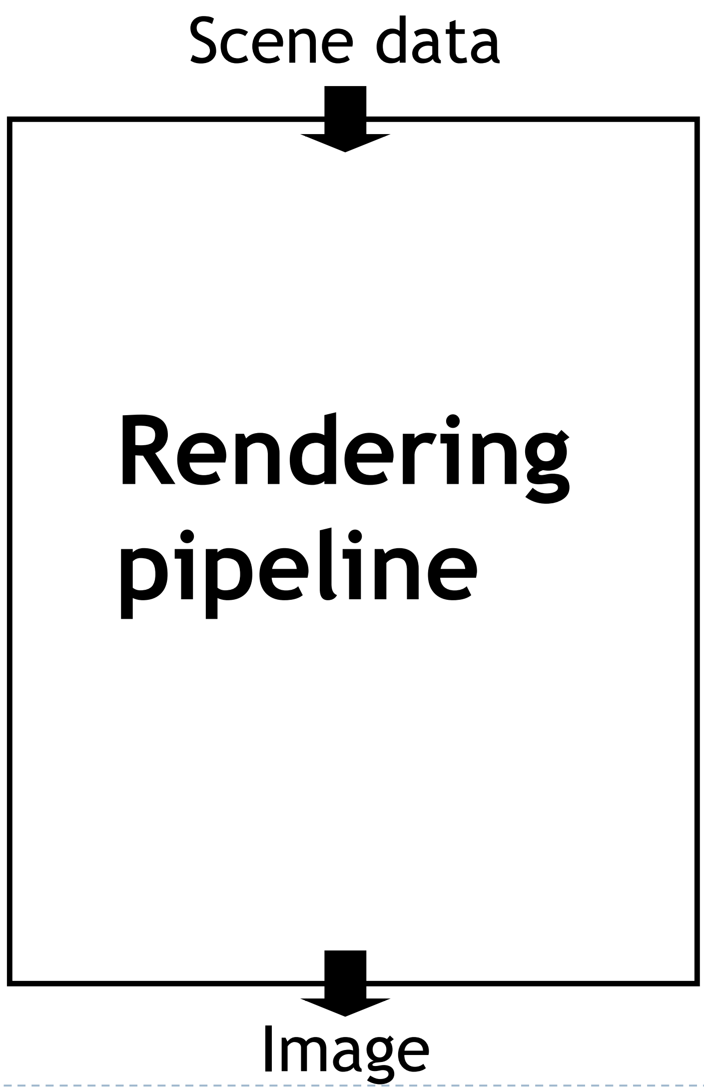
Rendering Pipeline



- Pixel colors

UCSD

Rendering Engine



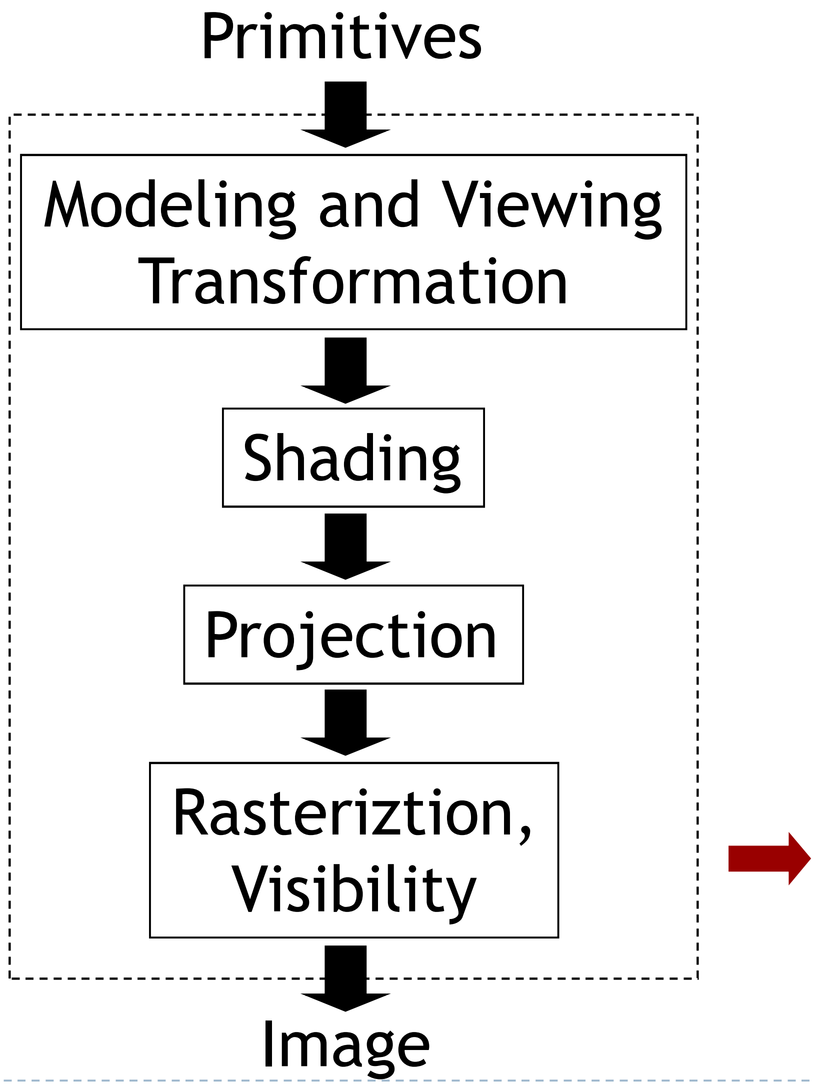
Rendering Engine:

- Additional software layer encapsulating low-level API
- Higher level functionality than OpenGL
- Platform independent
- Layered software architecture common in industry
  - Game engines
  - Graphics middleware

Lecture Overview

- Rasterization
- Visibility
- Shading

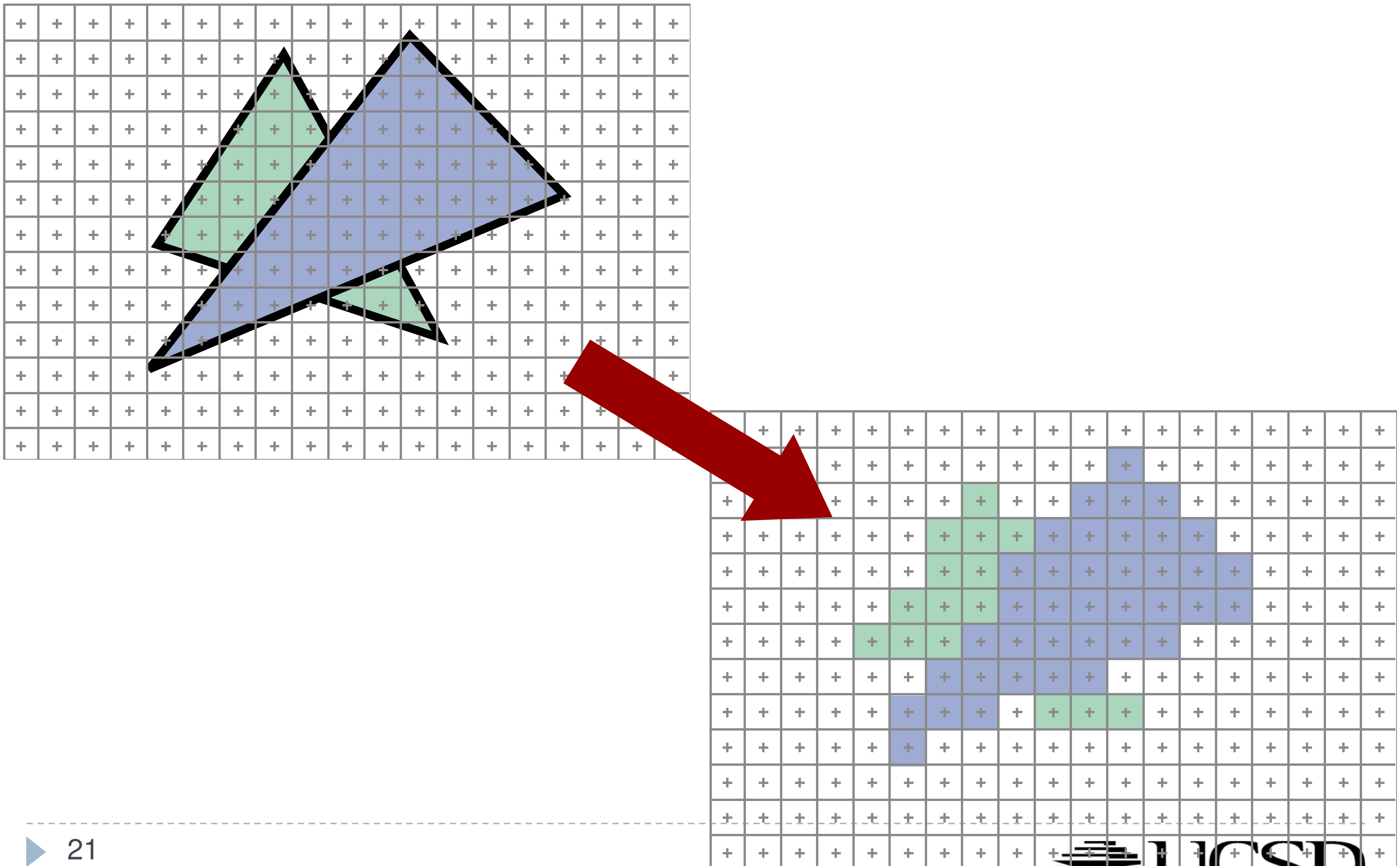
Rendering Pipeline



- Scan conversion and rasterization are synonyms
- One of the main operations performed by GPU
- Draw triangles, lines, points (squares)
- Focus on triangles in this lecture

UCSD

Rasterization



Rasterization

- Given vertices in pixel coordinates

$$\mathbf{p}' = \mathbf{D}\mathbf{P}\mathbf{C}^{-1}\mathbf{M}\mathbf{p}$$

World space

Camera space

Clip space

Image space

$$\mathbf{p}' = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix}$$

Pixel coordinates x'/w', y'/w'

Rasterization

- How many pixels can a modern graphics processor draw per second?

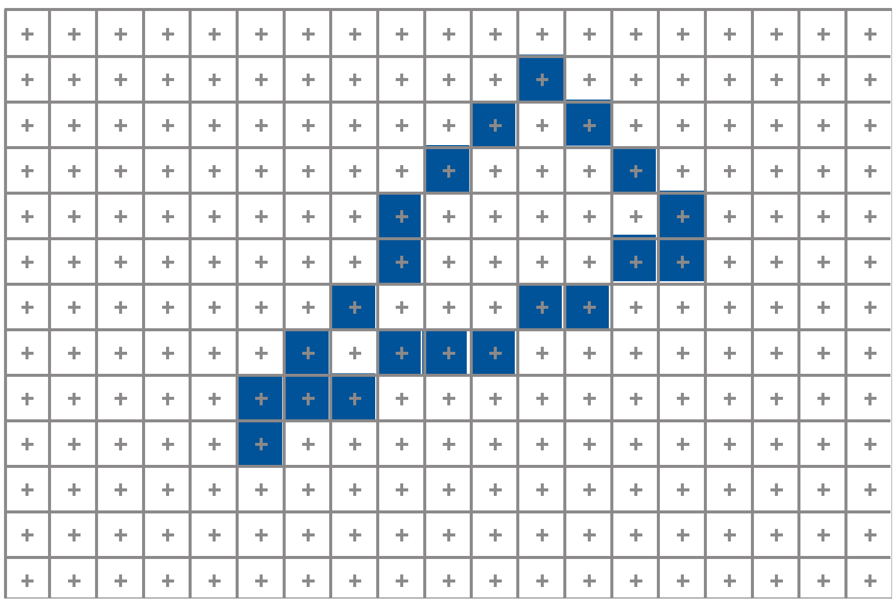
Rasterization

- How many pixels can a modern graphics processor draw per second?
- NVidia GeForce GTX 780
  - 160 billion pixels per second
  - Multiple of what the fastest CPU could do

UCSD

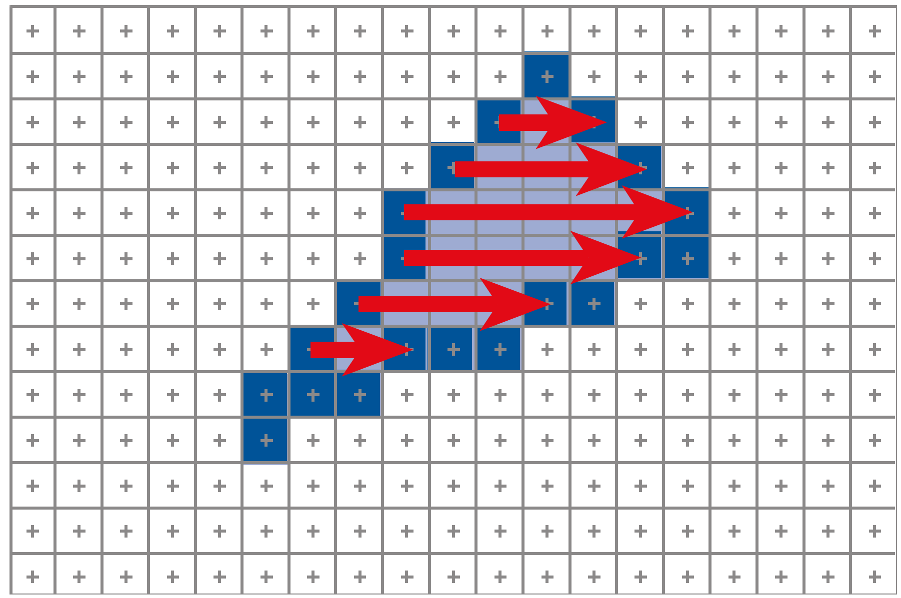
Rasterization

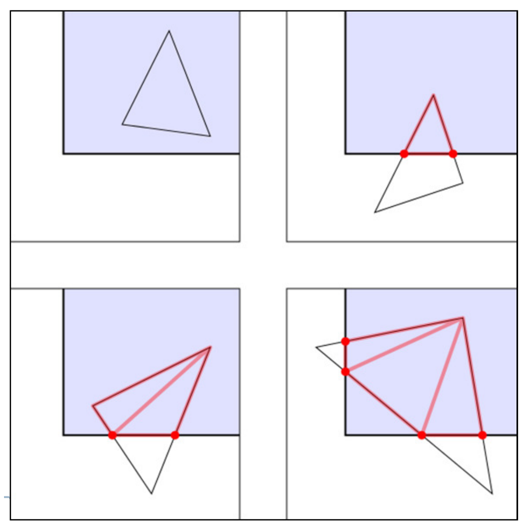
- Many different algorithms
- Old style
  - Rasterize edges first



Rasterization

- Many different algorithms
- Example:
  - Rasterize edges first
  - Fill the spans (scan lines)
- Disadvantage:
  - Requires clipping





Source: http://www.arcsynthesis.org

Rasterization

- GPU rasterization today based on “Homogeneous Rasterization”

  http://www.ece.unm.edu/course/ece595/docs/olano.pdf

  Olano, Marc and Trey Greer, "Triangle Scan Conversion Using 2D Homogeneous Coordinates", Proceedings of the 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware (Los Angeles, CA, August 2-4, 1997), ACM SIGGRAPH, New York, 1995.

Rasterization

- Given vertices in pixel coordinates

$$\mathbf{p}' = \mathbf{D}\mathbf{P}\mathbf{C}^{-1}\mathbf{M}\mathbf{p}$$

World space

Camera space

Clip space

Image space

$$\mathbf{p}' = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix}$$

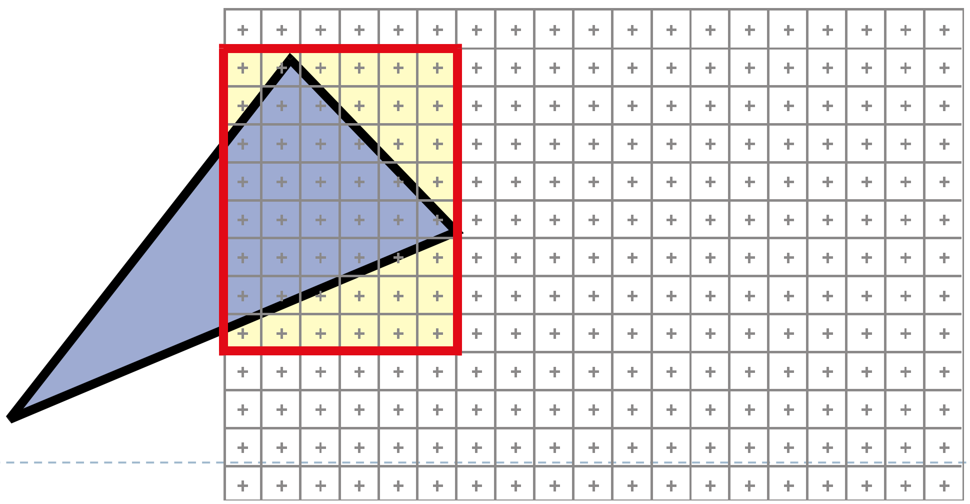
Pixel coordinates $\begin{matrix} x'/w' \\ y'/w' \end{matrix}$

Rasterization

- Simple algorithm

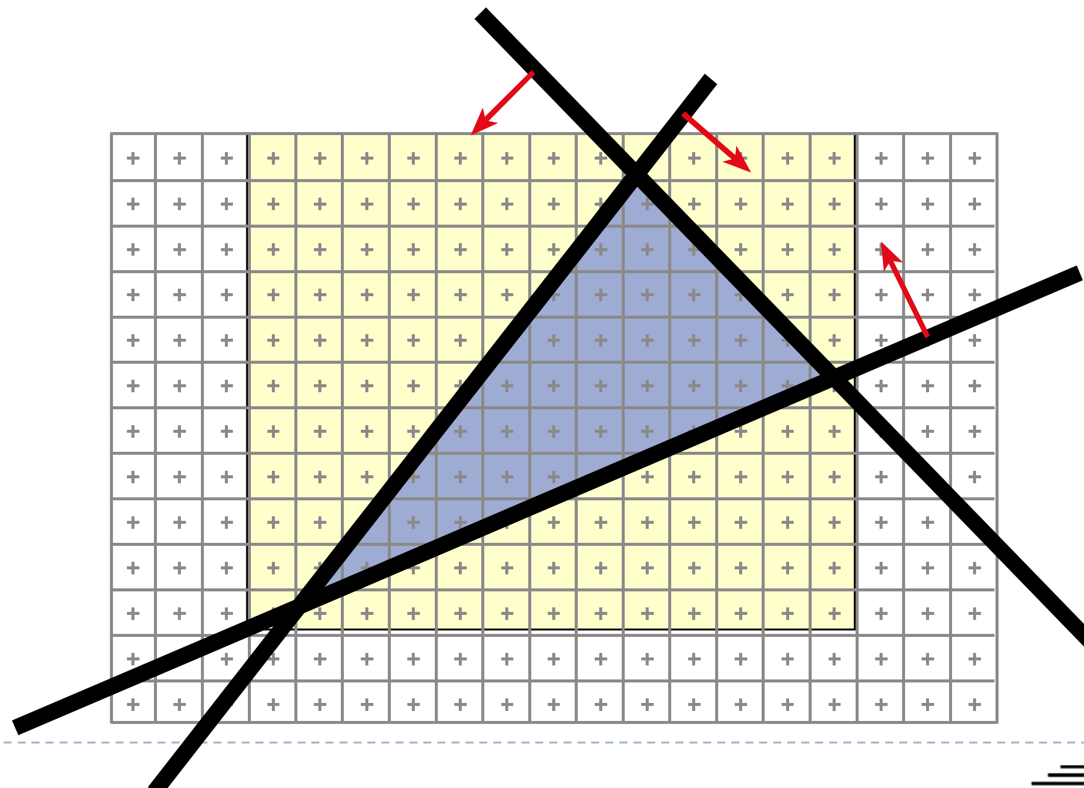
```
compute bbox
clip bbox to screen limits
for all pixels [x,y] in bbox
   compute barycentric coordinates alpha, beta, gamma
   if 0<alpha,beta,gamma<1 //pixel in triangle
         image[x,y]=triangleColor
```

- Bounding box clipping trivial



Rasterization

- So far, we compute barycentric coordinates of many useless pixels
- How can this be improved?

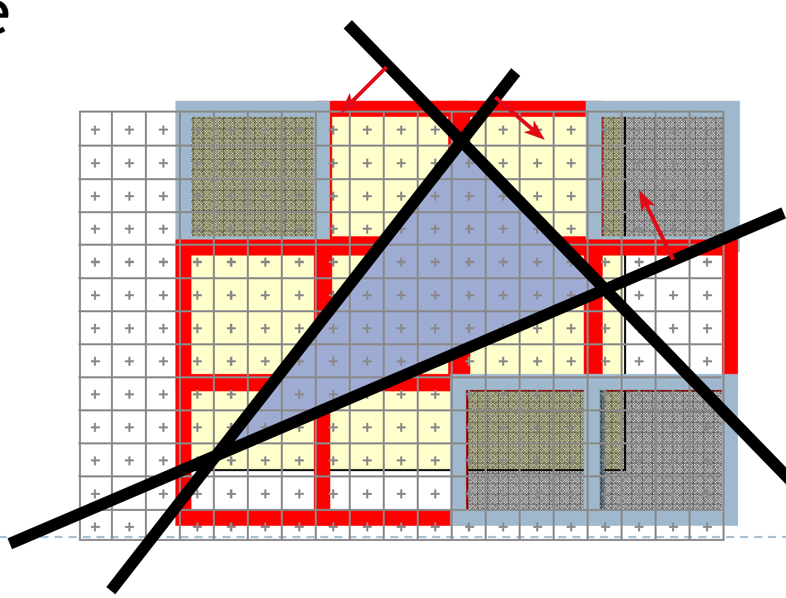


UCSD

Rasterization

Hierarchy

- If block of pixels is outside triangle, no need to test individual pixels
- Can have several levels, usually two-level
- Find right granularity and size of blocks for optimal performance



UCSD

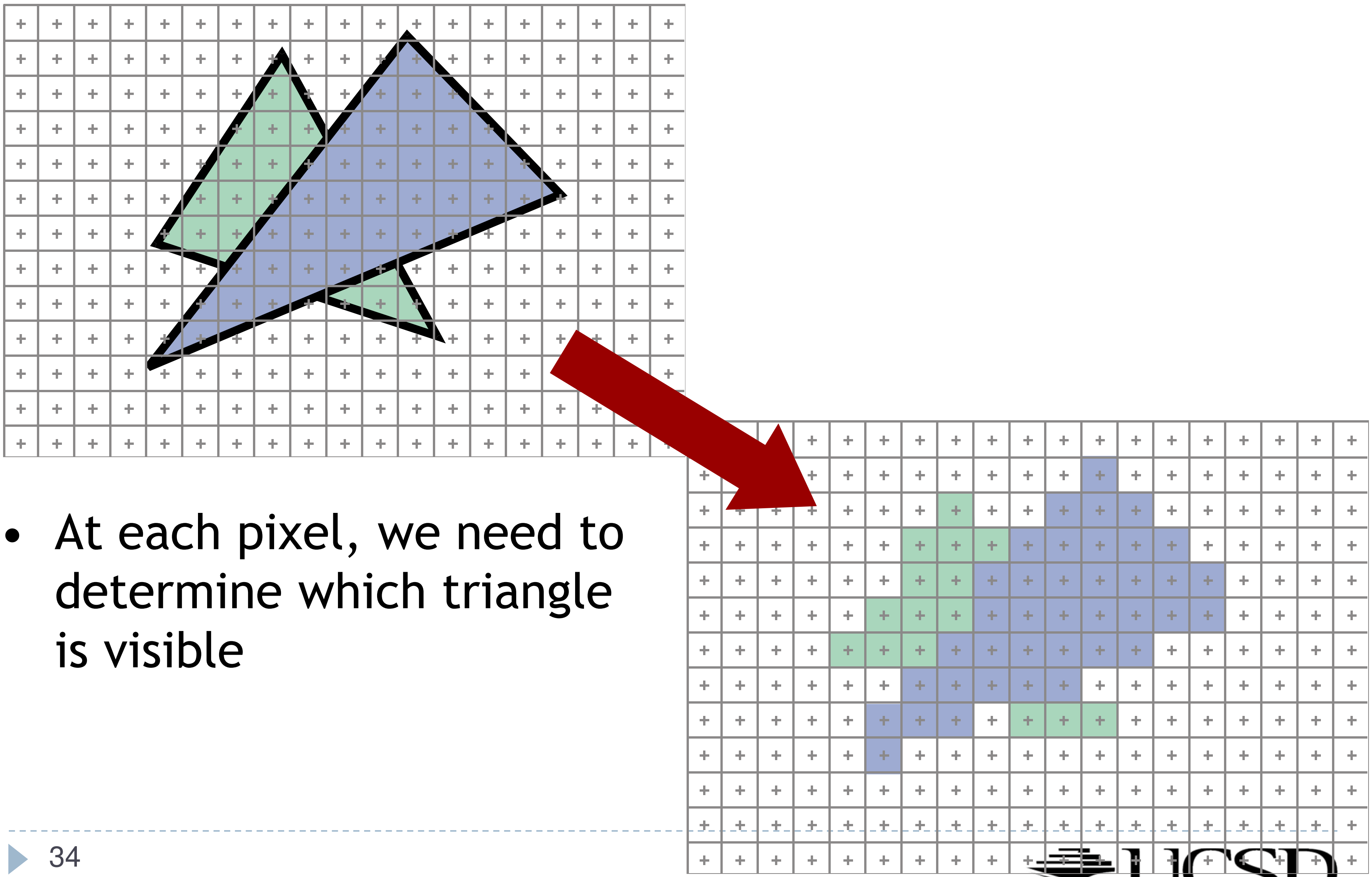
2D Triangle-Rectangle Intersection

- If one of the following tests returns true, the triangle intersects the rectangle:
  - Test if any of the triangle's vertices are inside the rectangle (e.g., by comparing the x/y coordinates to the min/max x/y coordinates of the rectangle)
  - Test if one of the quad's vertices is inside the triangle (e.g., using barycentric coordinates)
  - Intersect all edges of the triangle with all edges of the rectangle

Lecture Overview

- Rasterization
- Visibility
- Shading

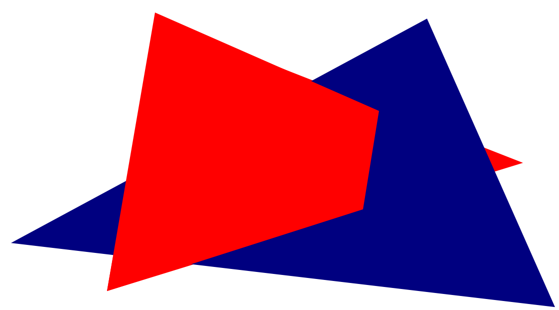
Visibility



- At each pixel, we need to determine which triangle is visible

Painter's Algorithm

- Paint from back to front
- Every new pixel always paints over previous pixel in frame buffer
- Need to sort geometry according to depth
- May need to split triangles if they intersect



- Outdated algorithm, created when memory was expensive

Z-Buffering

- Store z-value for each pixel
- Depth test
  - During rasterization, compare stored value to new value
  - Update pixel only if new value is smaller

```
setpixel(int x, int y, color c, float z)
if(z<zbuffer(x,y)) then
  zbuffer(x,y) = z
  color(x,y) = c
```

- z-buffer is dedicated memory reserved for GPU (graphics memory)
- Depth test is performed by GPU

Z-Buffering in OpenGL

- In your application:
  - Ask for a depth buffer when you create your window.
  - Place a call to glEnable (GL_DEPTH_TEST) in your program's initialization routine.
  - Ensure that your *zNear* and *zFar* clipping planes are set correctly (in glOrtho, glFrustum or gluPerspective) and in a way that provides adequate depth buffer precision.
  - Pass GL_DEPTH_BUFFER_BIT as a parameter to glClear.

Z-Buffering

- Problem: translucent geometry
  - Storage of multiple depth and color values per pixel (not practical in real-time graphics)
  - Or back to front rendering of translucent geometry, after rendering opaque geometry
    - Does not always work correctly: programmer has to weight rendering correctness against computational effort

UCSD

Lecture Overview

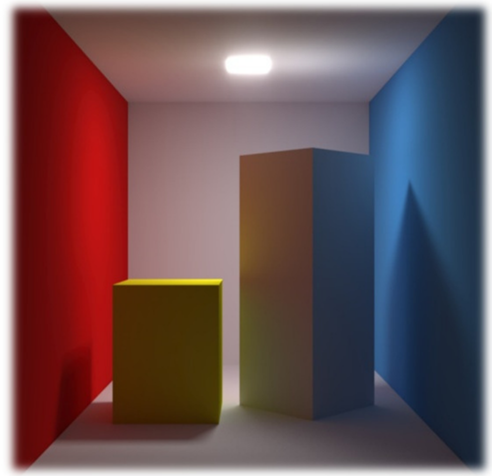
- Rasterization
- Visibility
- Shading

UCSD

Shading

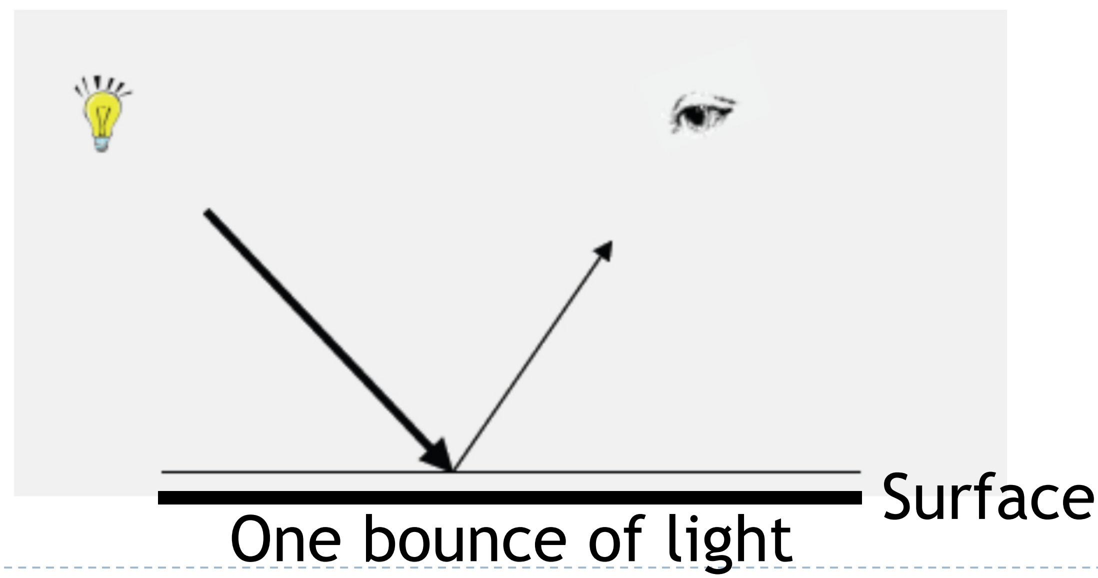
- Compute interaction of light with surfaces
- Requires simulation of physics
- "Global illumination"
  - Multiple bounces of light
  - Computationally expensive, minutes per image
  - Used in movies, architectural design, etc.

Global Illumination



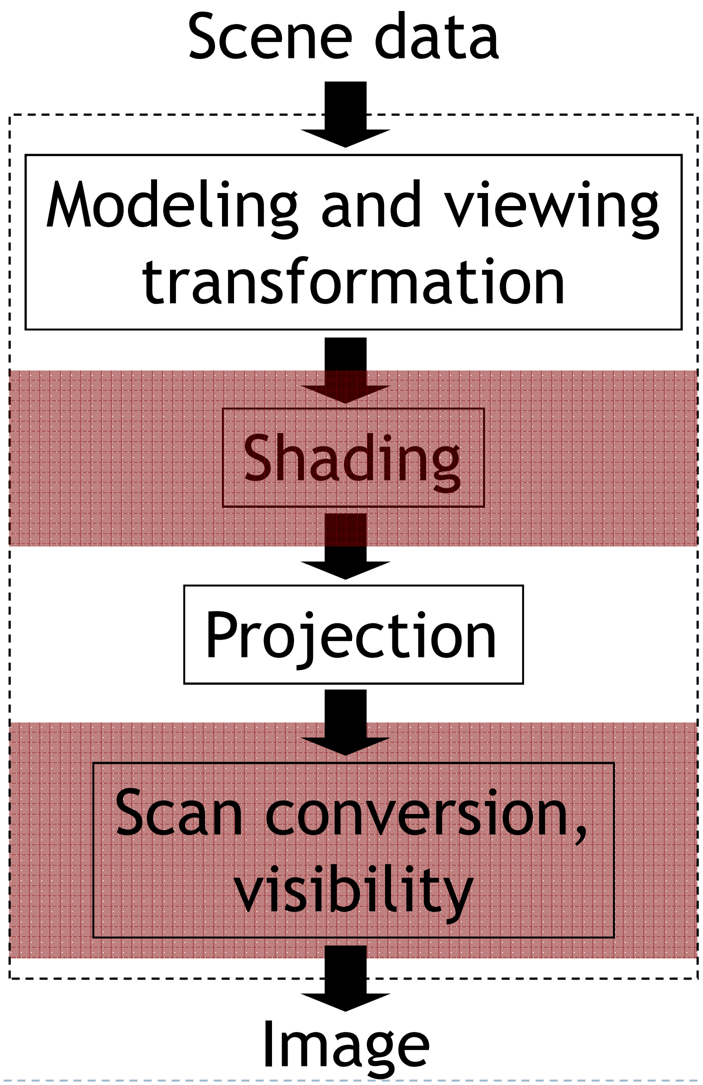
Interactive Applications

- No physics-based simulation
- Simplified models
- Reproduce perceptually most important effects
- Local illumination
  - Only one bounce of light between light source and viewer



UCSD

Rendering Pipeline



- Position object in 3D
- Determine colors of vertices
  - Per vertex shading
- Map triangles to 2D
- Draw triangles
  - Per pixel shading

UCSD

Lecture Overview

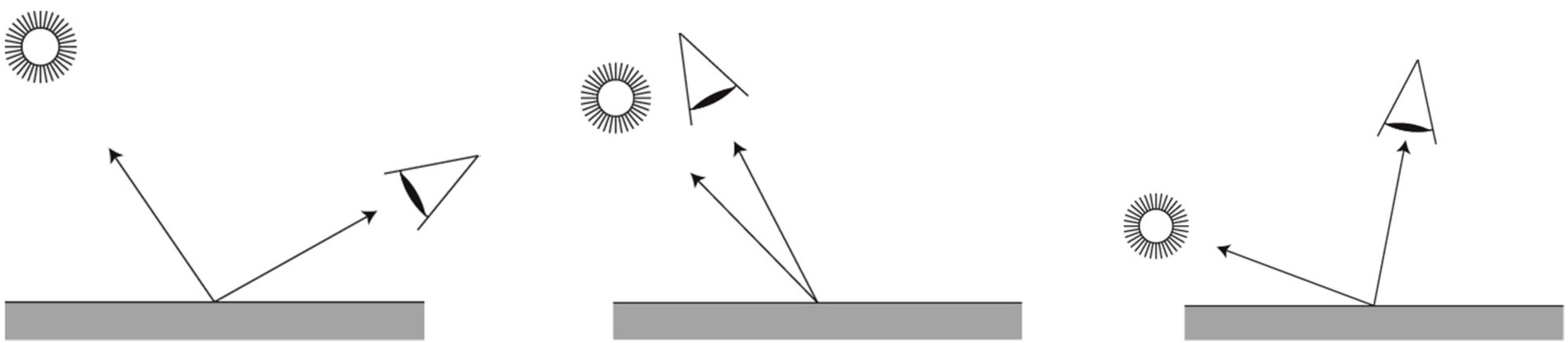
- OpenGL's local shading model

Local Illumination

- What gives a material its color?
- How is light reflected by a
  - Mirror
  - White sheet of paper
  - Blue sheet of paper
  - Glossy metal

Local Illumination

- Model reflection of light at surfaces
  - Assumption: no subsurface scattering
- Bidirectional reflectance distribution function (BRDF)
  - Given light direction, viewing direction, how much light is reflected towards the viewer
  - For any pair of light/viewing directions!

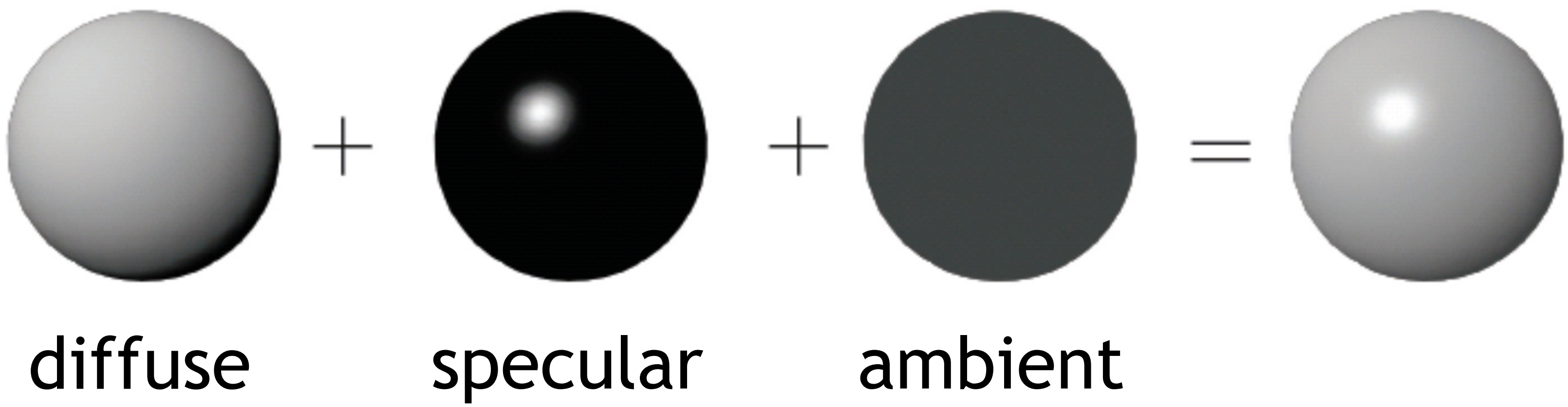


UCSD

Local Illumination

Simplified model

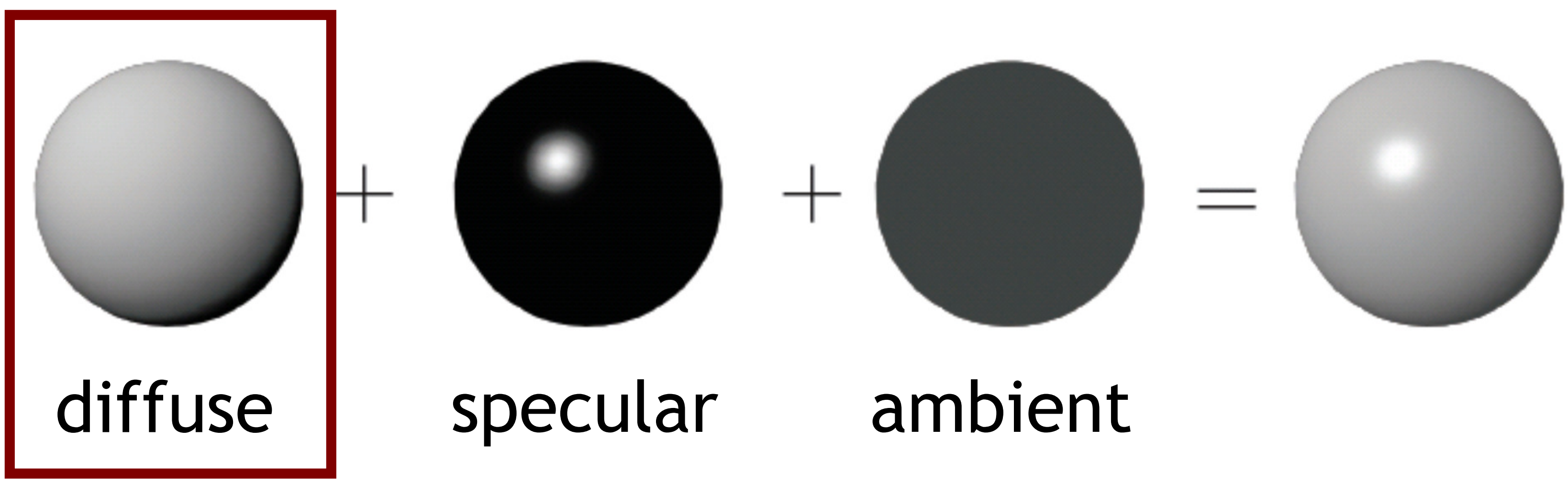
- Sum of 3 components
- Covers a large class of real surfaces



Local Illumination

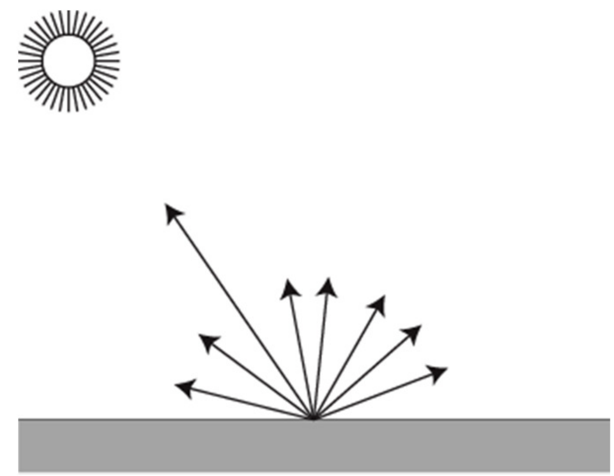
Simplified model

- Sum of 3 components
- Covers a large class of real surfaces



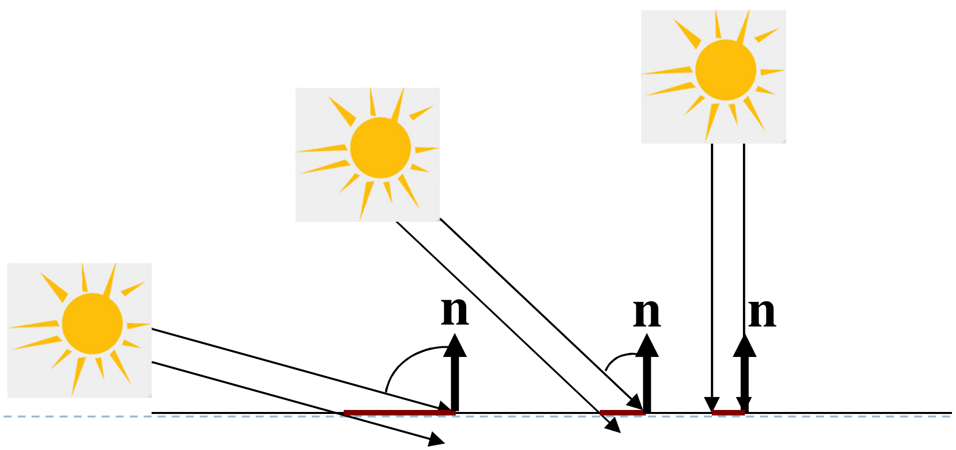
Diffuse Reflection

- Ideal diffuse material reflects light equally in all directions
- View-independent
- Matte, not shiny materials
  - Paper
  - Unfinished wood
  - Unpolished stone



Diffuse Reflection

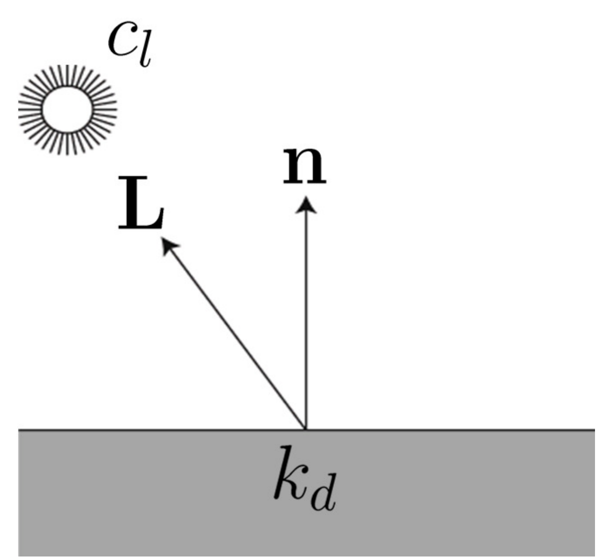
- Beam of parallel rays shining on a surface
  - Area covered by beam varies with the angle between the beam and the normal
  - The larger the area, the less incident light per area
  - Incident light per unit area is proportional to the cosine of the angle between the normal and the light rays
- Object darkens as normal turns away from light
- Lambert's cosine law (Johann Heinrich Lambert, 1760)
- Diffuse surfaces are also called Lambertian surfaces



UCSD

Diffuse Reflection

- Given
  - Unit surface normal **n**
  - Unit light direction **L**
  - Material diffuse reflectance (material color) k_d
  - Light color (intensity) c_l



- Diffuse color c_d is:

$$c_d = c_l k_d \underbrace{(\mathbf{n} \cdot \mathbf{L})}$$

Proportional to cosine between normal and light

Diffuse Reflection

Notes

- Parameters k_d, c_l are r,g,b vectors
- Need to compute r,g,b values of diffuse color c_d separately
- Parameters in this model have no precise physical meaning
  - c_l: strength, color of light source
  - k_d: fraction of reflected light, material color

Diffuse Reflection

- Provides visual cues
  - Surface curvature
  - Depth variation

Lambertian (diffuse) sphere under different lighting directions

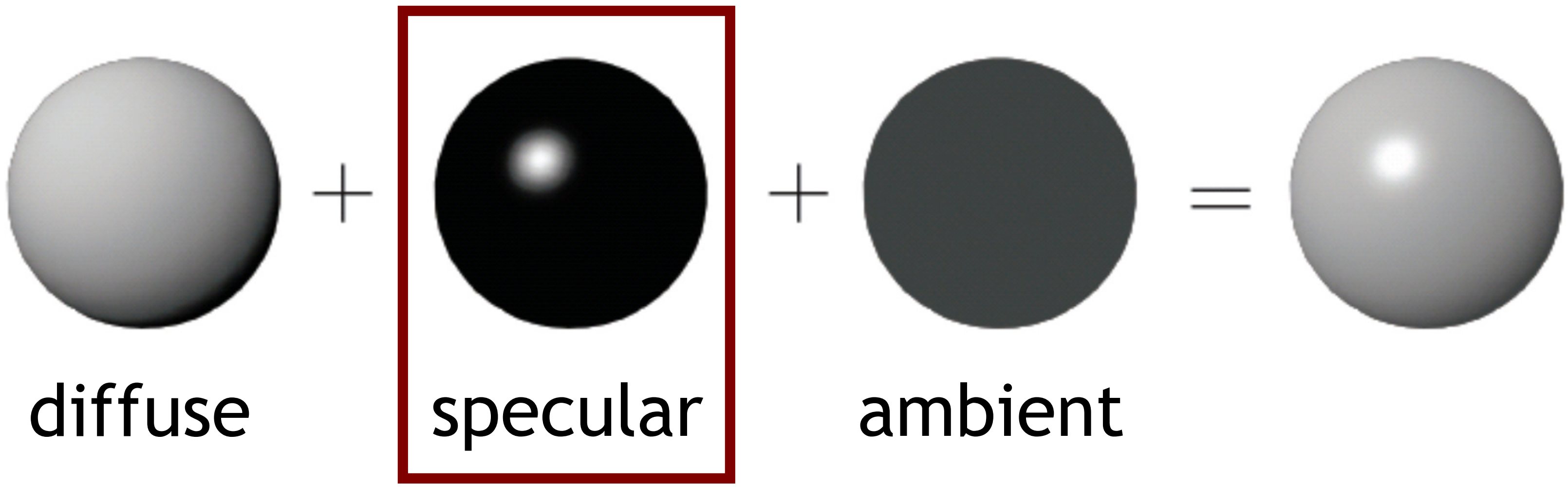
OpenGL

- Lights (glLight*)
  - Values for light: $(0,0,0) \leq c_l \leq (1,1,1)$
  - Definition: (0,0,0) is black, (1,1,1) is white
- OpenGL
  - Values for diffuse reflection
  - Fraction of reflected light: $(0,0,0) \leq k_d \leq (1,1,1)$
- Consult OpenGL Programming Guide (Red Book)
  - See course web site

Local Illumination

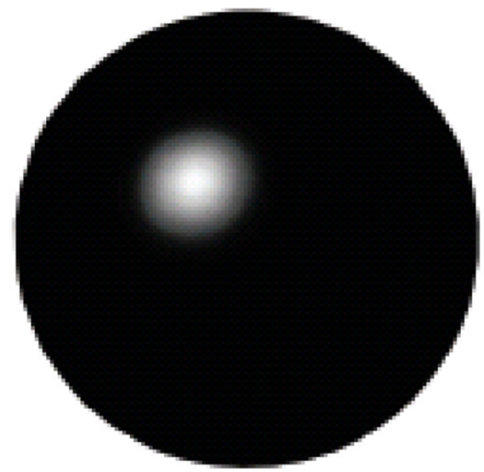
Simplified model

- Sum of 3 components
- Covers a large class of real surfaces



Specular Reflection

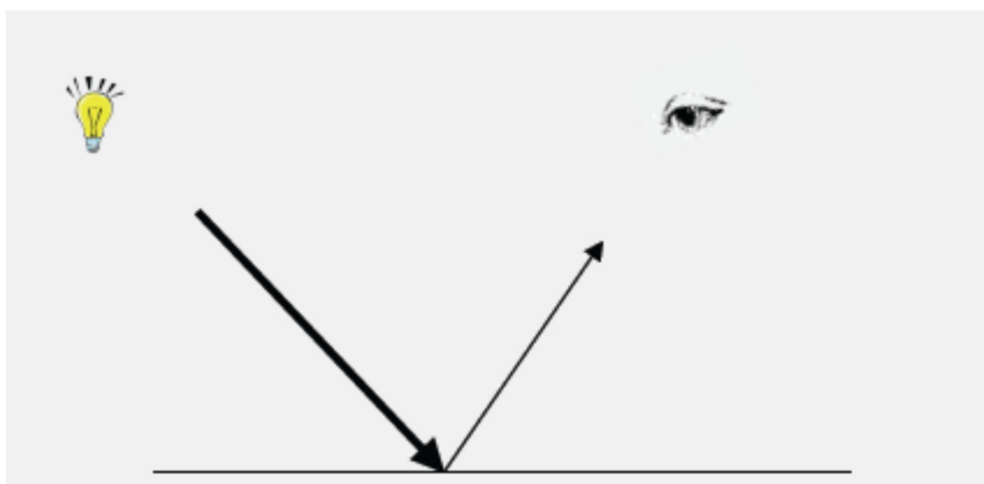
- Shiny surfaces
  - Polished metal
  - Glossy car finish
  - Plastics
- Specular highlight
  - Blurred reflection of the light source
  - Position of highlight depends on viewing direction



Specular highlight

Specular Reflection

- Ideal specular reflection is mirror reflection
  - Perfectly smooth surface
  - Incoming light ray is bounced in single direction
  - Angle of incidence equals angle of reflection

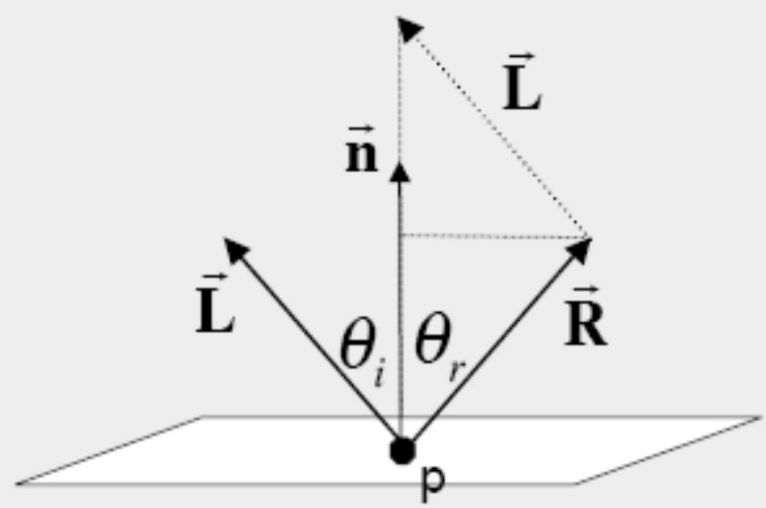


Law of Reflection

- Angle of incidence equals angle of reflection

$$\vec{\mathbf{R}} + \vec{\mathbf{L}} = 2\cos\theta\ \vec{\mathbf{n}} = 2\left(\vec{\mathbf{L}} \cdot \vec{\mathbf{n}}\right)\vec{\mathbf{n}}$$

$$\vec{\mathbf{R}} = 2\left(\vec{\mathbf{L}} \cdot \vec{\mathbf{n}}\right)\vec{\mathbf{n}} - \vec{\mathbf{L}}$$

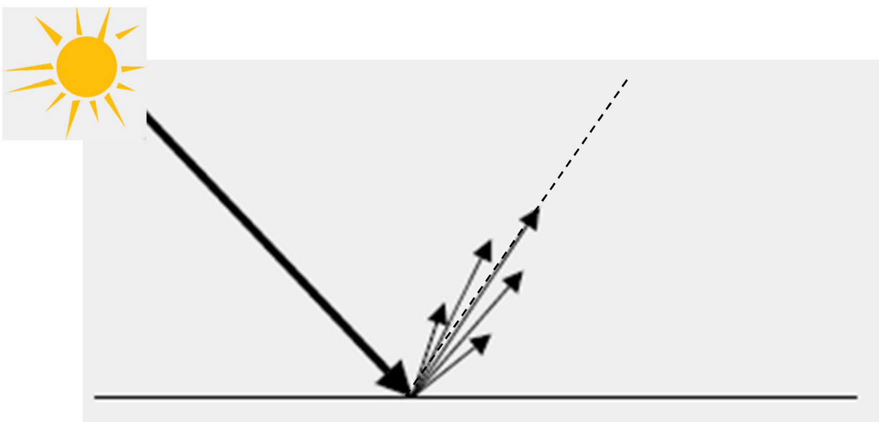


$$\theta_r = \theta_i$$

UCSD

Specular Reflection

- Many materials are not perfect mirrors
  - Glossy materials



Glossy teapot

UCSD

Glossy Materials

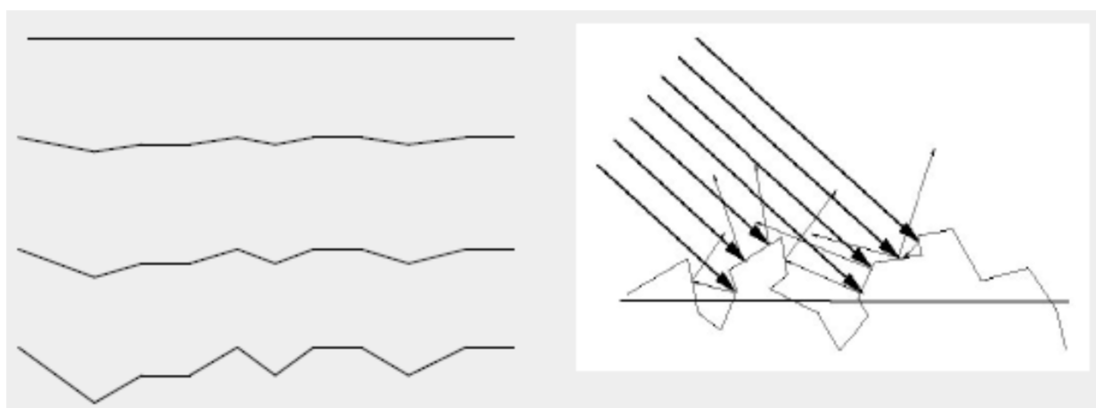
- Assume surface composed of small mirrors with random orientation (micro-facets)
- Smooth surfaces
  - Micro-facet normals close to surface normal
  - Sharp highlights
- Rough surfaces
  - Micro-facet normals vary strongly
  - Blurry highlight

Polished

Smooth

Rough

Very rough

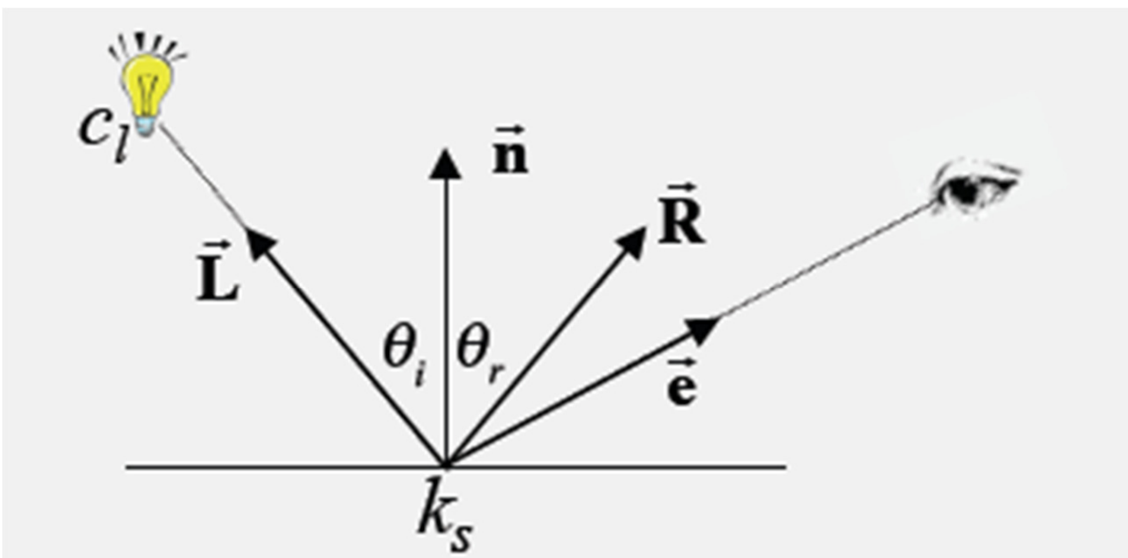


Glossy Surfaces

- Expect most light to be reflected in mirror direction
- Because of micro-facets, some light is reflected slightly off ideal reflection direction
- Reflection
  - Brightest when view vector is aligned with reflection
  - Decreases as angle between view vector and reflection direction increases

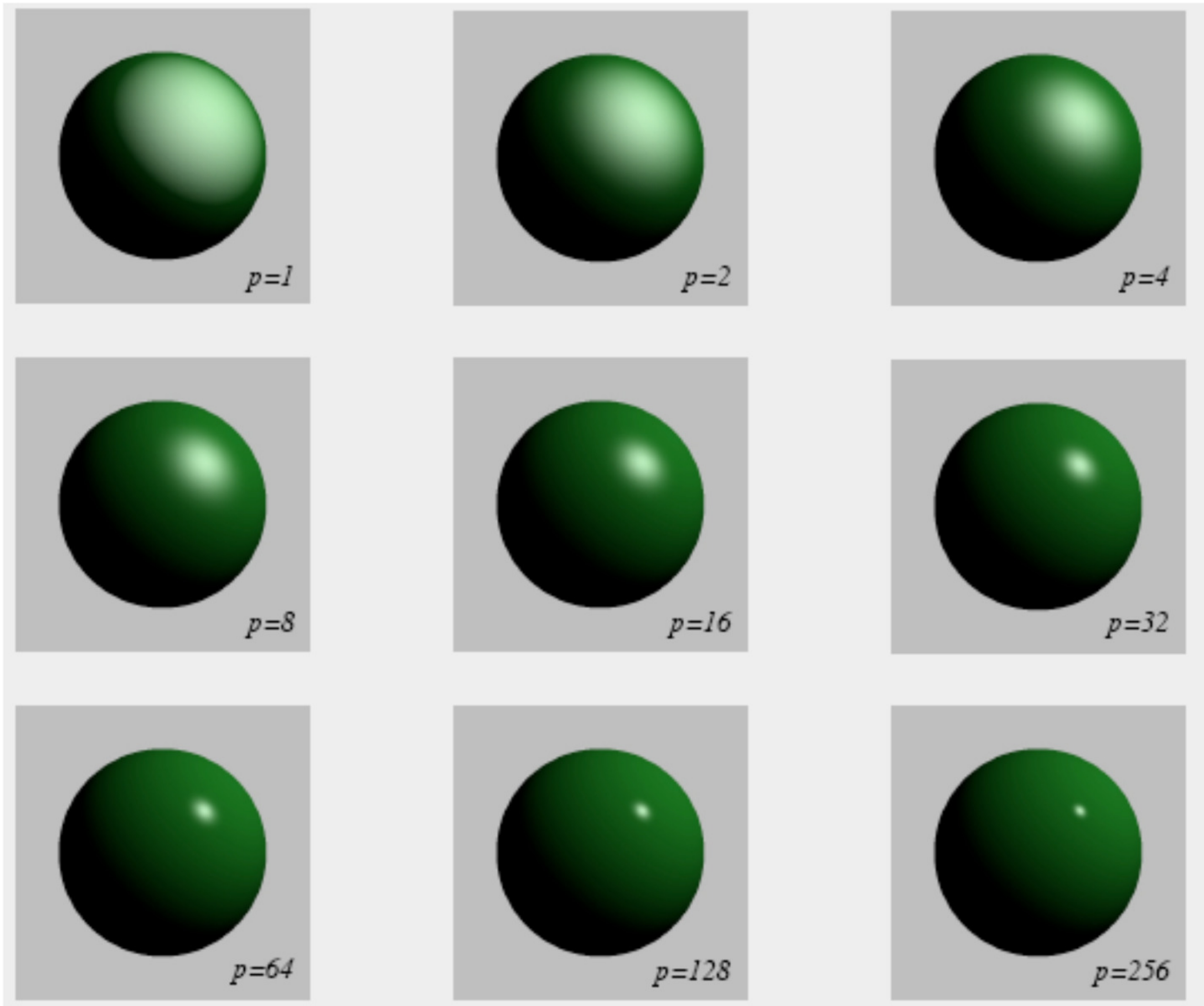
Phong Shading Model

- Developed by Bui Tuong Phong in1973
- Specular reflectance coefficient k_s
- Phong exponent p
  - Greater p means smaller (sharper) highlight



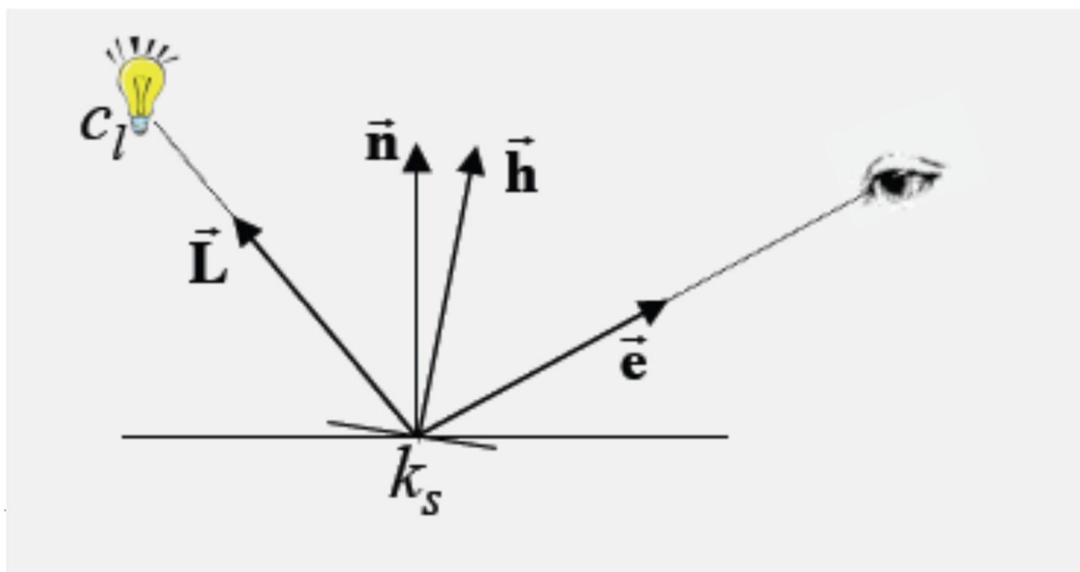
$$c = k_s c_l \left(\mathbf{R} \cdot \mathbf{e}\right)^p$$

Phong Shading Model



Blinn Shading Model (Jim Blinn, 1977)

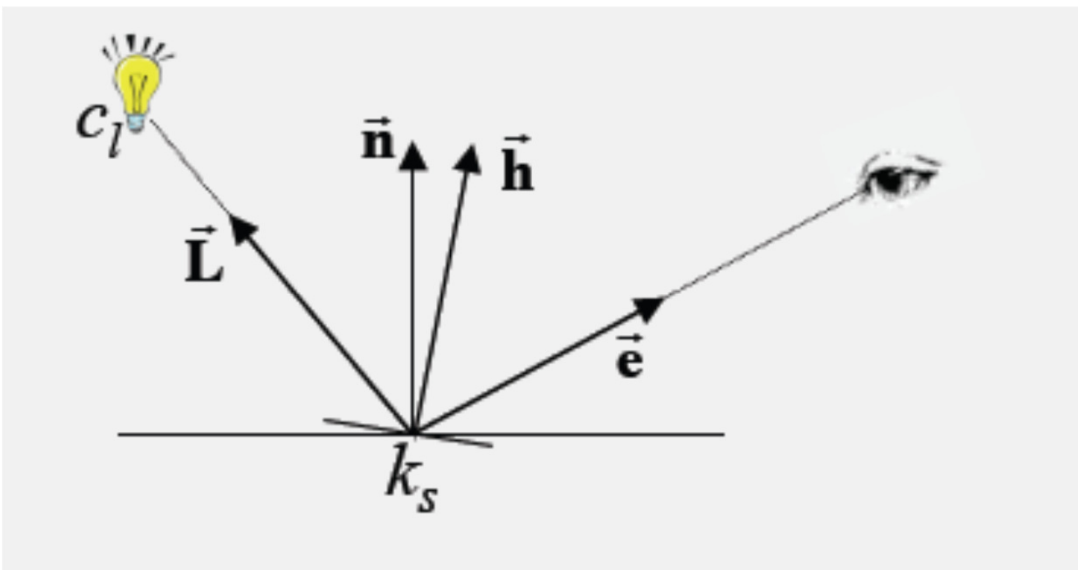
- Modification of Phong Shading Model
- Defines unit halfway vector $$\mathbf{h} = \frac{\mathbf{L} + \mathbf{e}}{\|\mathbf{L} + \mathbf{e}\|}$$
- Halfway vector represents normal of micro-facet that would lead to mirror reflection to the eye



UCSD

Blinn Shading Model

- The larger the angle between micro-facet orientation and normal, the less likely
- Use cosine of angle between them
- Shininess parameter *s*
- Very similar to Phong Model

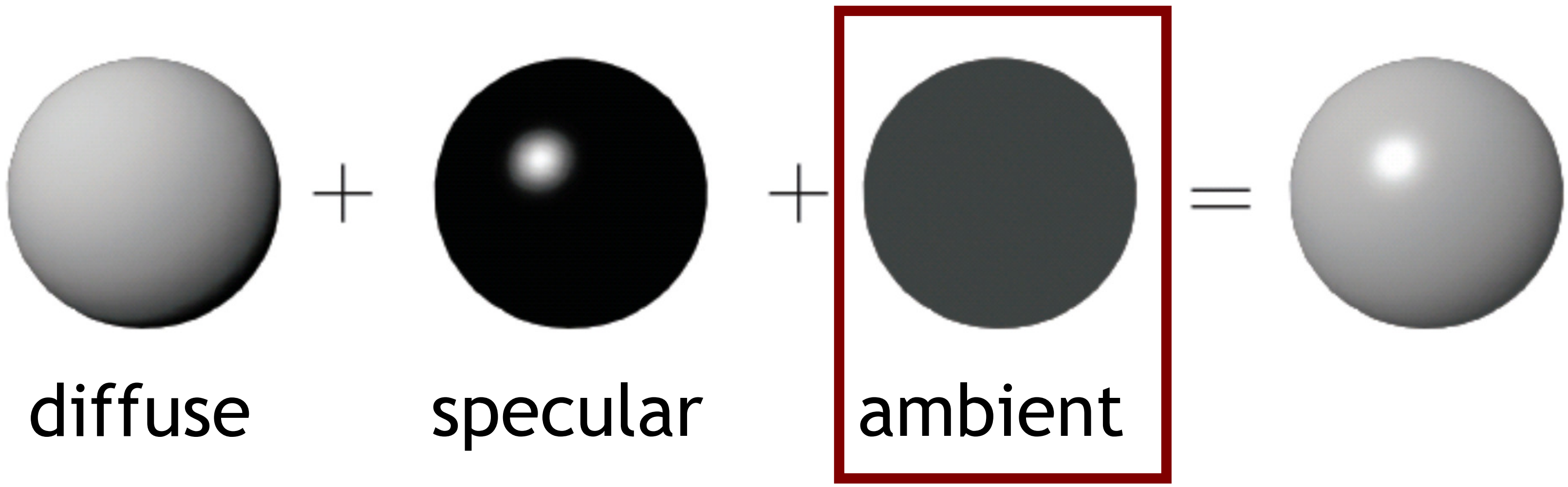


$$c = k_s c_l \left(\mathbf{h} \cdot \mathbf{n}\right)^s$$

Local Illumination

Simplified model

- Sum of 3 components
- Covers a large class of real surfaces



UCSD

Ambient Light

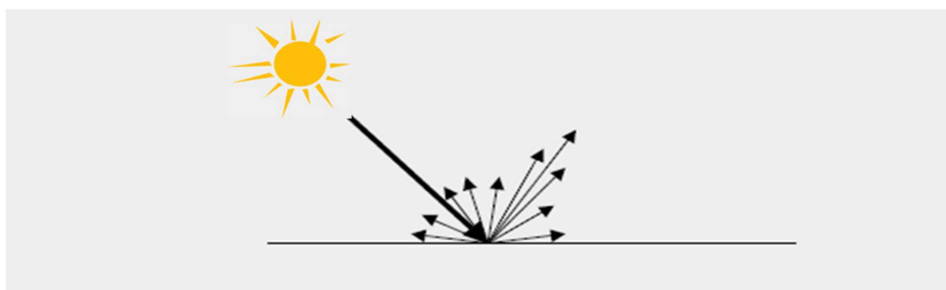
- In real world, light is bounced all around scene
- Could use global illumination techniques to simulate
- Simple approximation
  - Add constant ambient light at each point: $k_a c_a$
  - Ambient light color: c_a
  - Ambient reflection coefficient: k_a
- Areas with no direct illumination are not completely dark

Complete Blinn-Phong Shading Model

- Blinn-Phong model with several light sources I
- All colors and reflection coefficients are vectors with 3 components for red, green, blue

$$c = \sum_i c_{l_i} \left(k_d \left(\mathbf{L}_i \cdot \mathbf{n}\right) + k_s \left(\mathbf{h}_i \cdot \mathbf{n}\right)^s\right) + k_a c_a$$

diffuse specular ambient



UCSD

Lecture Overview

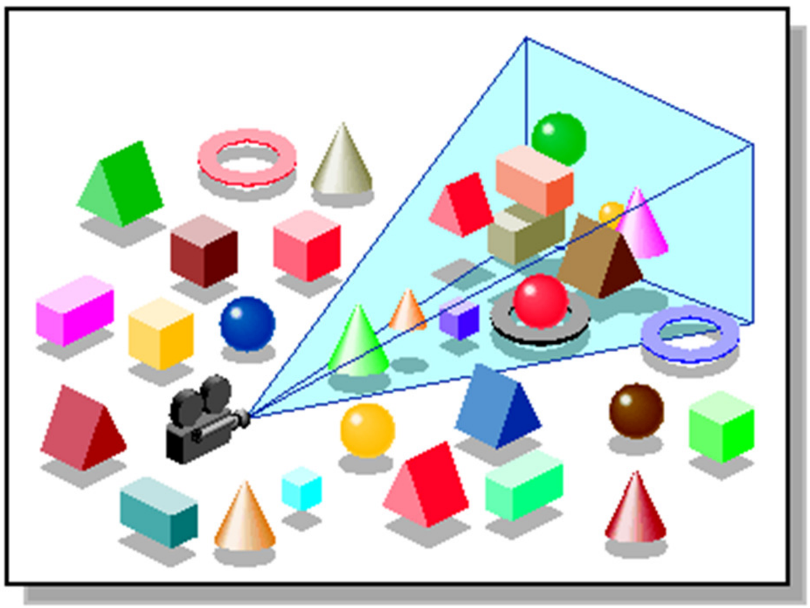
- Culling

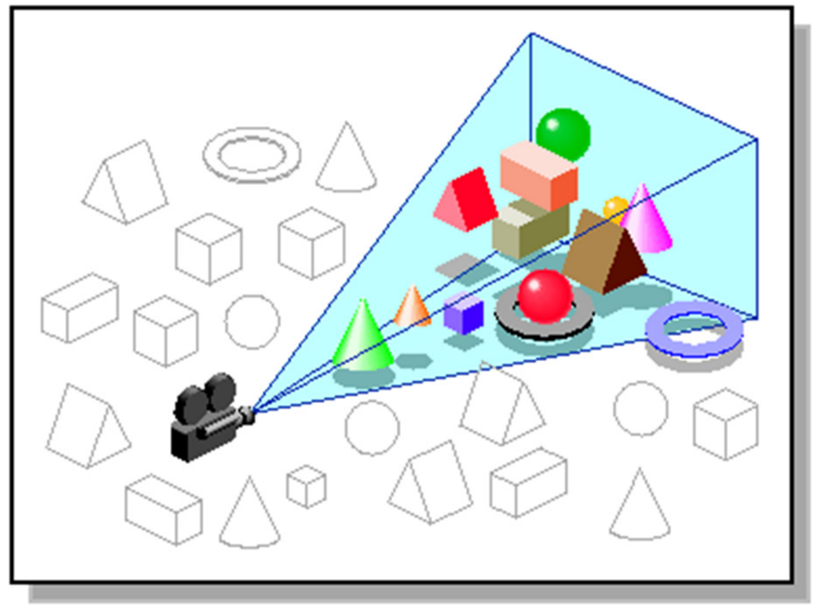
Culling

- Goal:
  Discard geometry that does not need to be drawn to speed up rendering
- Types of culling:
  - View frustum culling
  - Occlusion culling
  - Small object culling
  - Backface culling
  - Degenerate culling

View Frustum Culling

- Triangles outside of view frustum are off-screen
  - Done on canonical view volume





Images: SGI OpenGL Optimizer Programmer's Guide

Videos

- Rendering Optimizations - Frustum Culling
  - http://www.youtube.com/watch?v=kvVHp9wMAO8
- View Frustum Culling Demo
  - http://www.youtube.com/watch?v=bJrYTBGpwic

Bounding Box

- How to cull objects consisting of may polygons?
- Cull bounding box
  - Rectangular box, parallel to object space coordinate planes
  - Box is smallest box containing the entire object

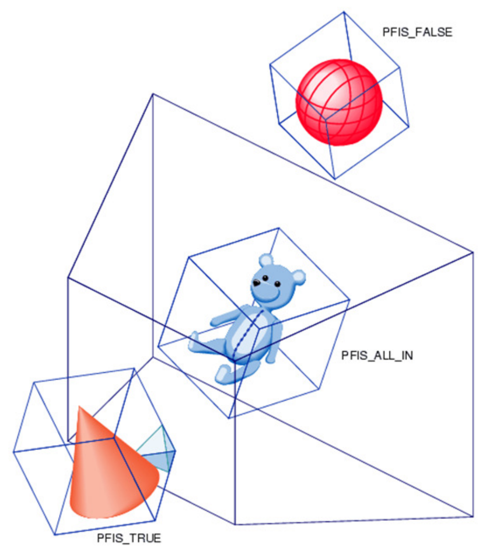
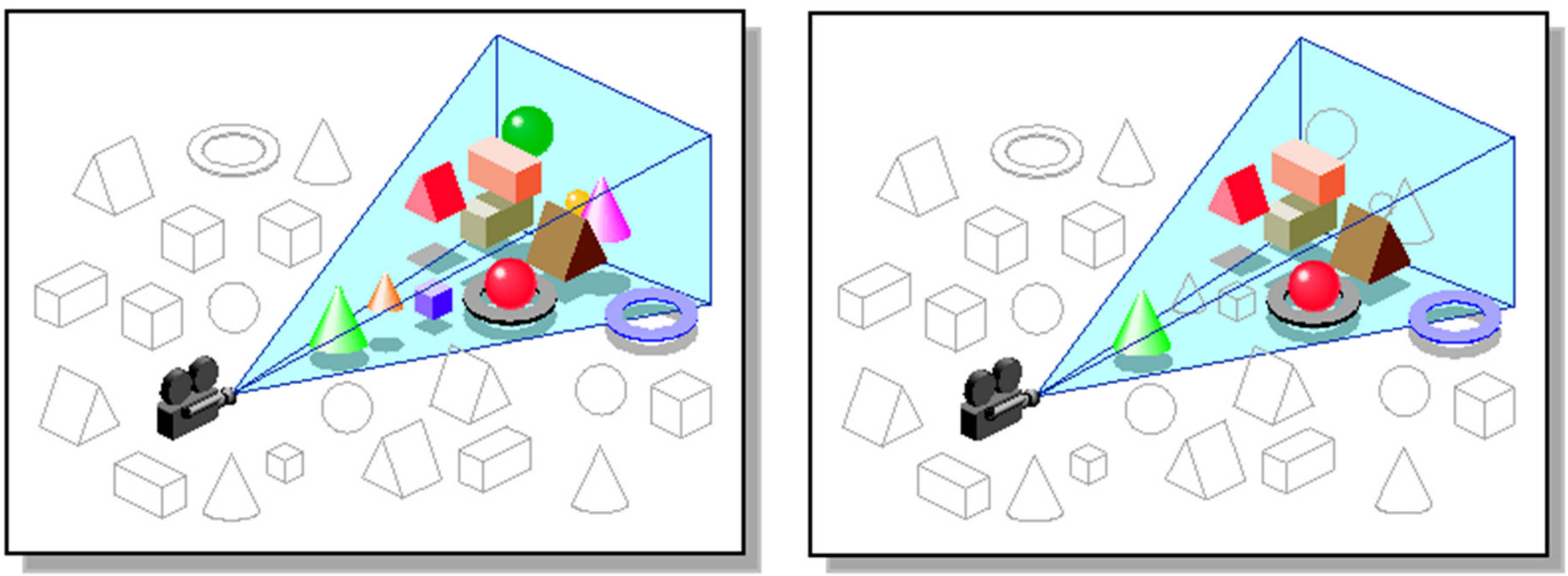


Image: SGI OpenGL Optimizer Programmer's Guide

Occlusion Culling

- Geometry hidden behind occluder cannot be seen
  - Many complex algorithms exist to identify occluded geometry



Images: SGI OpenGL Optimizer Programmer's Guide

Video

- Umbra 3 Occlusion Culling explained
  - http://www.youtube.com/watch?v=5h4QgDBwQhc

Small Object Culling

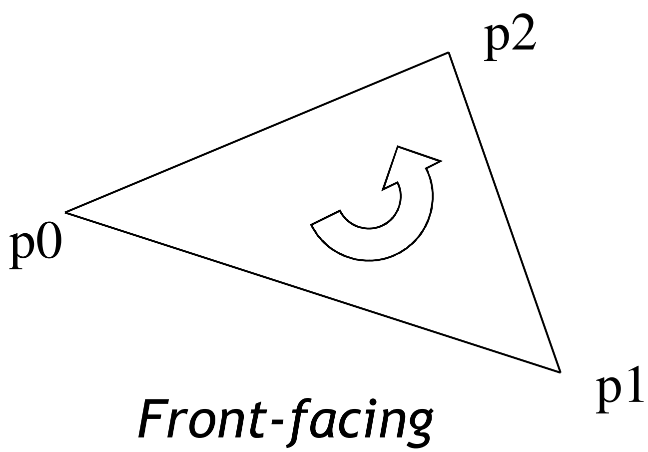
- Object projects to less than a specified size
  - Cull objects whose screen-space bounding box is less than a threshold number of pixels

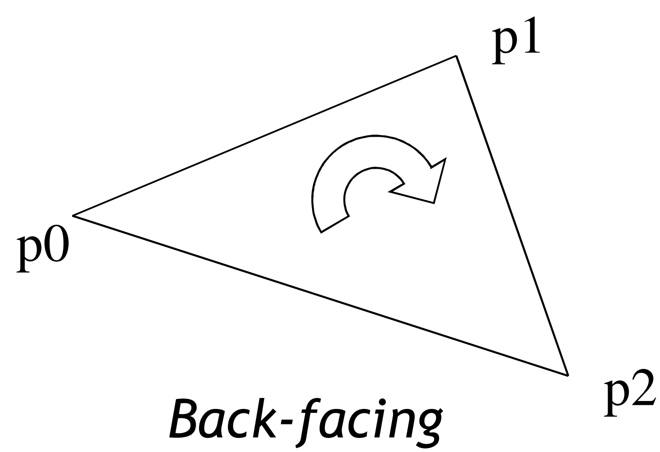
Backface Culling

- Consider triangles as "one-sided", i.e., only visible from the "front"
- Closed objects
  - If the "back" of the triangle is facing the camera, it is not visible
  - Gain efficiency by not drawing it (culling)
  - Roughly 50% of triangles in a scene are back facing

Backface Culling

- Convention:
  Triangle is front facing if vertices are ordered counterclockwise





- OpenGL allows one- or two-sided triangles
  - One-sided triangles:
    glEnable(GL_CULL_FACE); glCullFace(GL_BACK)
  - Two-sided triangles (no backface culling):
    glDisable(GL_CULL_FACE)

Backface Culling

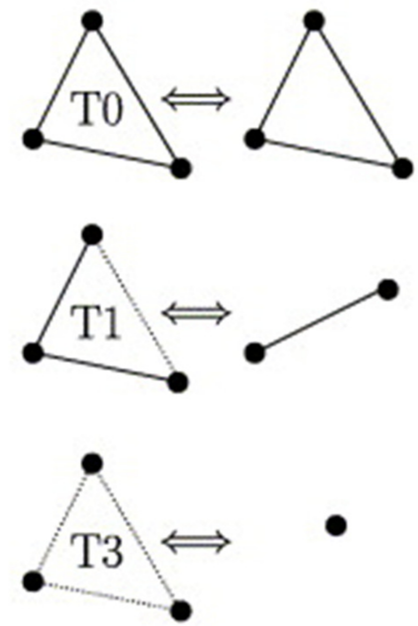
- Compute triangle normal after projection (homogeneous division)

$$\mathbf{n} = (\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)$$

- Third component of $\mathbf{n}$ negative: front-facing, otherwise back-facing
  - Remember: projection matrix is such that homogeneous division flips sign of third component

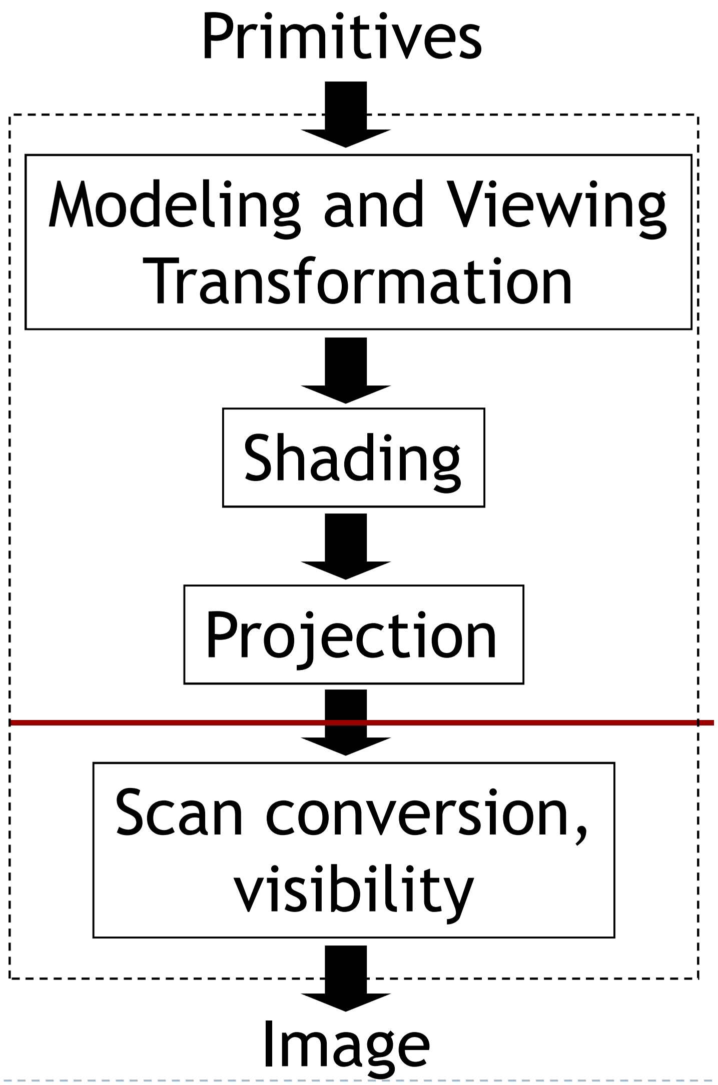
Degenerate Culling

- Degenerate triangle has no area
  - Vertices lie in a straight line
  - Vertices at the exact same place
  - Normal **n**=0



Source: Computer Methods in Applied Mechanics and Engineering, Volume 194, Issues 48–49

Rendering Pipeline



Culling, Clipping

- Discard geometry that will not be visible

UCSD