University of British Columbia
CPSC 314 Computer Graphics
Jan-Apr 2010

Tamara Munzner

Lighting/Shading IV, Advanced Rendering I

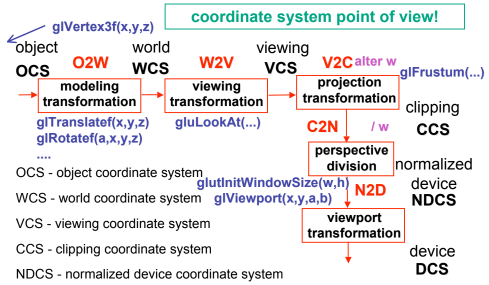
Week 7, Fri Mar 5

http://www.ugrad.cs.ubc.ca/~cs314/Vjan2010

News

- midterm is Monday, be on time!
- HW2 solutions out

Clarify: Projective Rendering Pipeline



OCS - object coordinate system

WCS - world coordinate system

VCS - viewing coordinate system

CCS - clipping coordinate system

NDCS - normalized device coordinate system

DCS - device coordinate system

Clarify: OpenGL Example

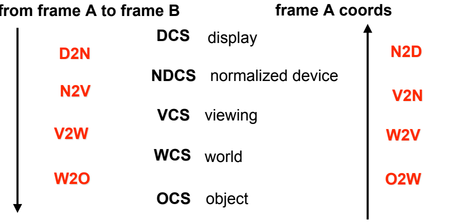

```
CCS   glMatrixMode( GL_PROJECTION );
      glLoadIdentity();
      gluPerspective( 45, 1.0, 0.1, 200.0 );
VCS   glMatrixMode( GL_MODELVIEW );
      glLoadIdentity();
      glTranslatef( 0.0, 0.0, -5.0 );   V2W
WCS   glPushMatrix()
      glTranslate( 4, 4, 0 );   W2O
OCS1  glutSolidTeapot(1);
      glPopMatrix();
      glTranslate( 2, 2, 0);   W2O
OCS2  glutSolidTeapot(1);
```

- transformations that are applied to object first are specified last

Coordinate Systems: Frame vs Point

read down: transforming between coordinate frames, from frame A to frame B

read up: transforming points, up from frame B coords to frame A coords



Coordinate Systems: Frame vs Point

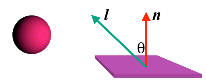
- is gluLookAt V2W or W2V? depends on which way you read!
  - coordinate frames: V2W
    - takes you from view to world coordinate frame
  - points/objects: W2V
    - transforms point from world to view coords

Homework

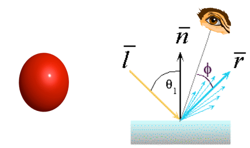
- most of my lecture slides use coordinate frame reading ("reading down")
  - same with my post to discussion group: said to use W2V, V2N, N2D
- homework questions asked you to compute for object/point coords ("reading up")
- correct matrix for question 1 is gluLookat
- enough confusion that we will not deduct marks if you used inverse of gluLookAt instead of gluLookAt!
  - same for Q2, Q3: no deduction if you used inverses of correct matices

Review: Reflection Equations

$$I_{diffuse} = k_d\, I_{light}\, (\mathbf{n} \bullet \mathbf{l})$$



$$I_{specular} = k_s I_{light} (\mathbf{v} \bullet \mathbf{r})^{n_{shiny}}$$



2 (**N** (**N** · **L**)) – **L** = **R**

Review: Phong Lighting Model

- combine ambient, diffuse, specular components

$$\mathbf{I}_{total} = \mathbf{k}_a \mathbf{I}_{ambient} + \sum_{i=1}^{\#\,lights} \mathbf{I}_i (\mathbf{k}_d (\mathbf{n} \bullet \mathbf{l}_i) + \mathbf{k}_s (\mathbf{v} \bullet \mathbf{r}_i)^{n_{shiny}})$$

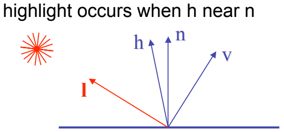
- commonly called *Phong lighting*
  - once per light
  - once per color component
- reminder: normalize your vectors when calculating!
  - normalize all vectors: n,l,r,v

Review: Blinn-Phong Model

- variation with better physical interpretation
  - Jim Blinn, 1977

$$I_{out}(\mathbf{x}) = \mathbf{k}_s (\mathbf{h} \bullet \mathbf{n})^{n_{shiny}} \bullet I_{in}(\mathbf{x}); \text{with } \mathbf{h} = (\mathbf{l} + \mathbf{v})/2$$

- ***h***: halfway vector
  - h must also be explicitly normalized: h / |h|
  - highlight occurs when h near n



Review: Lighting

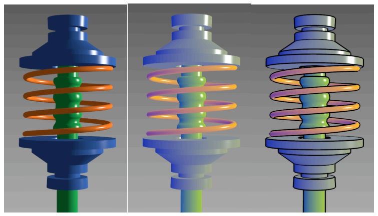
- lighting models
- ambient
  - normals don't matter
- Lambert/diffuse
  - angle between surface normal and light
- Phong/specular
  - surface normal, light, and viewpoint

Review: Shading Models Summary

- flat shading
  - compute Phong lighting once for entire polygon
- Gouraud shading
  - compute Phong lighting at the vertices
  - at each pixel across polygon, interpolate lighting values
- Phong shading
  - compute averaged vertex normals at the vertices
  - at each pixel across polygon, interpolate normals and compute Phong lighting

Non-Photorealistic Shading

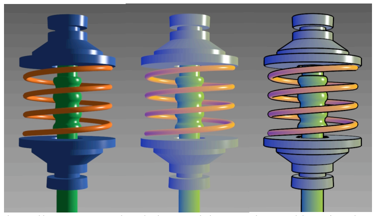
- cool-to-warm shading $k_w = \frac{1 + \mathbf{n} \cdot \mathbf{l}}{2}, c = k_w c_w + (1 - k_w) c_c$



http://www.cs.utah.edu/~gooch/SIG98/paper/drawing.html

Non-Photorealistic Shading

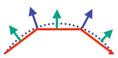
- draw silhouettes: if $(\mathbf{e} \cdot \mathbf{n_0})(\mathbf{e} \cdot \mathbf{n_1}) \le 0$, **e**=edge-eye vector
- draw creases: if $(\mathbf{n_0} \cdot \mathbf{n_1}) \le threshold$

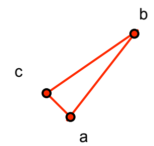


http://www.cs.utah.edu/~gooch/SIG98/paper/drawing.html

Computing Normals

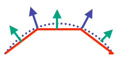
- per-vertex normals by interpolating per-facet normals
  - OpenGL supports both
- computing normal for a polygon

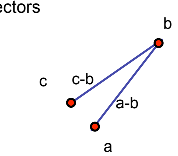




Computing Normals

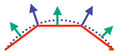
- per-vertex normals by interpolating per-facet normals
  - OpenGL supports both
- computing normal for a polygon
  - three points form two vectors

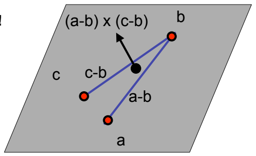




Computing Normals

- per-vertex normals by interpolating per-facet normals
  - OpenGL supports both
- computing normal for a polygon
  - three points form two vectors
  - cross: normal of plane gives direction
  - **normalize to unit length!**
  - which side is up?
    - convention: points in counterclockwise order





Specifying Normals

- OpenGL state machine
  - uses last normal specified
  - if no normals specified, assumes all identical
- per-vertex normals

```
glNormal3f(1,1,1);
glVertex3f(3,4,5);
glNormal3f(1,1,0);
glVertex3f(10,5,2);
```

- per-face normals

```
glNormal3f(1,1,1);
glVertex3f(3,4,5);
glVertex3f(10,5,2);
```

- normal interpreted as direction from vertex location
- can automatically normalize (computational cost)

```
glEnable(GL_NORMALIZE);
```

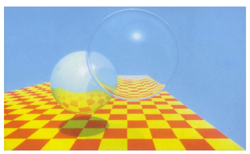

Advanced Rendering

Global Illumination Models

- simple lighting/shading methods simulate local illumination models
  - no object-object interaction
- global illumination models
  - more realism, more computation
  - leaving the pipeline for these two lectures!
- approaches
  - ray tracing
  - radiosity
  - photon mapping
  - subsurface scattering

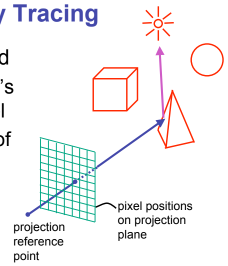
Ray Tracing

- simple basic algorithm
- well-suited for software rendering
- flexible, easy to incorporate new effects
  - Turner Whitted, 1990



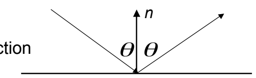
Simple Ray Tracing

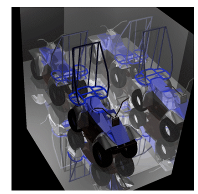
- view dependent method
  - cast a ray from viewer's eye through each pixel
  - compute intersection of ray with first object in scene
  - cast ray from intersection point on object to light sources

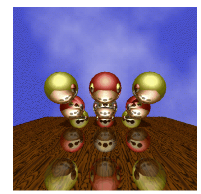


Reflection

- mirror effects
  - perfect specular reflection

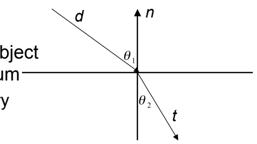


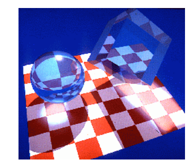




Refraction

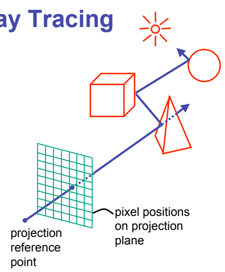
- happens at interface between transparent object and surrounding medium
  - e.g. glass/air boundary
- Snell's Law
  - $c_1 \sin\theta_1 = c_2 \sin\theta_2$
  - light ray bends based on refractive indices c_1, c_2





Recursive Ray Tracing

- ray tracing can handle
  - reflection (chrome/mirror)
  - refraction (glass)
  - shadows
- spawn secondary rays
  - reflection, refraction
    - if another object is hit, recurse to find its color
  - shadow
    - cast ray from intersection point to light source, check if intersects another object



Basic Algorithm

for every pixel p_i {
    generate ray r from camera position through pixel p_i
    for every object o in scene {
        if (r intersects o)
            compute lighting at intersection point, using local normal and material properties; store result in p_i
        else
            p_i= background color
    }
}

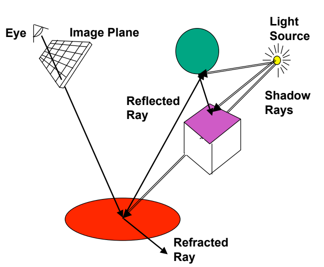
Basic Ray Tracing Algorithm

RayTrace(r,scene)
obj := **FirstIntersection**(r,scene)
if (no obj) return BackgroundColor;
else begin
    if (**Reflect**(obj)) then
        reflect_color := **RayTrace**(**ReflectRay**(r,obj));
    else
        reflect_color := Black;
    if (**Transparent**(obj)) then
        refract_color := **RayTrace**(**RefractRay**(r,obj));
    else
        refract_color := Black;
    return **Shade**(reflect_color,refract_color,obj);
end;

Algorithm Termination Criteria

- termination criteria
  - no intersection
  - reach maximal depth
    - number of bounces
  - contribution of secondary ray attenuated below threshold
    - each reflection/refraction attenuates ray

Ray Tracing Algorithm

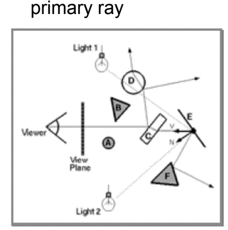


Ray-Tracing Terminology

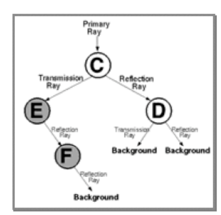
- terminology:
  - primary ray: ray starting at camera
  - shadow ray
  - reflected/refracted ray
  - ray tree: all rays directly or indirectly spawned off by a single primary ray
- note:
  - need to limit maximum depth of ray tree to ensure termination of ray-tracing process!

Ray Trees

- all rays directly or indirectly spawned off by a single primary ray



Ray traced through scene



Ray tree

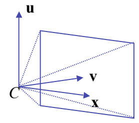
w.cs.virginia.edu/~gfx/Courses/2003/Intro.fall.03/slides/lighting_web/lighting.pdf

Ray Tracing

- issues:
  - generation of rays
  - intersection of rays with geometric primitives
  - geometric transformations
  - lighting and shading
  - efficient data structures so we don't have to test intersection with *every* object

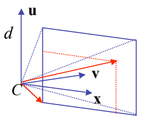
Ray Generation

- camera coordinate system
  - origin: C (camera position)
  - viewing direction: $\mathbf{v}$
  - up vector: $\mathbf{u}$
  - x direction: $\mathbf{x} = \mathbf{v} \times \mathbf{u}$
- note:
  - corresponds to viewing transformation in rendering pipeline
  - like gluLookAt



Ray Generation

- other parameters:
  - distance of camera from image plane: d
  - image resolution (in pixels): w, h
  - left, right, top, bottom boundaries in image plane: l, r, t, b
- then:
  - lower left corner of image: $O = C + d \cdot \mathbf{v} + l \cdot \mathbf{x} + b \cdot \mathbf{u}$
  - pixel at position i, j ($i=0..w-1, j=0..h-1$):



$$P_{i,j} = O + i \cdot \frac{r-l}{w-1} \cdot \mathbf{x} - j \cdot \frac{t-b}{h-1} \cdot \mathbf{u}$$
$$= O + i \cdot \Delta x \cdot \mathbf{x} - j \cdot \Delta y \cdot \mathbf{y}$$

Ray Generation

- ray in 3D space:

$$\mathrm{R}_{i,j}(t) = C + t \cdot (P_{i,j} - C) = C + t \cdot \mathbf{v}_{i,j}$$

where $t = 0 \ldots \infty$

Ray Tracing

- issues:
  - generation of rays
  - intersection of rays with geometric primitives
  - geometric transformations
  - lighting and shading
  - efficient data structures so we don't have to test intersection with *every* object

Ray - Object Intersections

- inner loop of ray-tracing
  - must be extremely efficient
- task: given an object o, find ray parameter t, such that $\mathbf{R}_{i,j}(t)$ is a point on the object
  - such a value for t may not exist
- solve a set of equations
- intersection test depends on geometric primitive
  - ray-sphere
  - ray-triangle
  - ray-polygon

Ray Intersections: Spheres

- spheres at origin
  - implicit function

$$S(x, y, z): x^2 + y^2 + z^2 = r^2$$

  - ray equation

$$\mathrm{R}_{i,j}(t) = C + t \cdot \mathbf{v}_{i,j} = \begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix} + t \cdot \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} c_x + t \cdot v_x \\ c_y + t \cdot v_y \\ c_z + t \cdot v_z \end{pmatrix}$$

Ray Intersections: Spheres

- to determine intersection:
  - insert ray $\mathbf{R}_{i,j}(t)$ into $S(x,y,z)$:

$$(c_x + t \cdot v_x)^2 + (c_y + t \cdot v_y)^2 + (c_z + t \cdot v_z)^2 = r^2$$

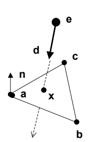
  - solve for t (find roots)
    - simple quadratic equation

Ray Intersections: Other Primitives

- implicit functions
  - spheres at arbitrary positions
    - same thing
  - conic sections (hyperboloids, ellipsoids, paraboloids, cones, cylinders)
    - same thing (all are quadratic functions!)
- polygons
  - first intersect ray with plane
    - linear implicit function
  - then test whether point is inside or outside of polygon (2D test)
  - for convex polygons
    - suffices to test whether point in on the correct side of every boundary edge
    - similar to computation of outcodes in line clipping (upcoming)

Ray-Triangle Intersection

- method in book is elegant but a bit complex
- easier approach: triangle is just a polygon
  - intersect ray with plane



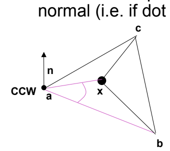
$$\text{normal}: \mathbf{n} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})$$
$$\text{ray}: \mathbf{x} = \mathbf{e} + t\mathbf{d}$$
$$\text{plane}: (\mathbf{p} - \mathbf{x}) \cdot \mathbf{n} = 0 \Rightarrow \mathbf{x} = \frac{\mathbf{p} \cdot \mathbf{n}}{\mathbf{n}}$$
$$\frac{\mathbf{p} \cdot \mathbf{n}}{\mathbf{n}} = \mathbf{e} + t\mathbf{d} \Rightarrow t = -\frac{(\mathbf{e} - \mathbf{p}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$

$\mathbf{p}$ is $\mathbf{a}$ or $\mathbf{b}$ or $\mathbf{c}$

  - check if ray inside triangle

Ray-Triangle Intersection

- check if ray inside triangle
  - check if point counterclockwise from each edge (to its left)
  - check if cross product points in same direction as normal (i.e. if dot is positive)



$$(\mathbf{b} - \mathbf{a}) \times (\mathbf{x} - \mathbf{a}) \cdot \mathbf{n} \geq 0$$
$$(\mathbf{c} - \mathbf{b}) \times (\mathbf{x} - \mathbf{b}) \cdot \mathbf{n} \geq 0$$
$$(\mathbf{a} - \mathbf{c}) \times (\mathbf{x} - \mathbf{c}) \cdot \mathbf{n} \geq 0$$

  - more details at http://www.cs.cornell.edu/courses/cs465/2003fa/homeworks/raytri.pdf

Ray Tracing

- issues:
  - generation of rays
  - intersection of rays with geometric primitives
  - geometric transformations
  - lighting and shading
  - efficient data structures so we don't have to test intersection with *every* object

Geometric Transformations

- similar goal as in rendering pipeline:
  - modeling scenes more convenient using different coordinate systems for individual objects
- problem
  - not all object representations are easy to transform
    - problem is fixed in rendering pipeline by restriction to polygons, which are affine invariant
  - ray tracing has different solution
    - ray itself is always affine invariant
    - thus: transform ray into object coordinates!

Geometric Transformations

- ray transformation
  - for intersection test, it is only important that ray is in same coordinate system as object representation
  - transform all rays into object coordinates
    - transform camera point and ray direction by inverse of model/view matrix
  - shading has to be done in world coordinates (where light sources are given)
    - transform object space intersection point to world coordinates
    - thus have to keep both world and object-space ray

Ray Tracing

- issues:
  - generation of rays
  - intersection of rays with geometric primitives
  - geometric transformations
  - lighting and shading
  - efficient data structures so we don't have to test intersection with *every* object

Local Lighting

- local surface information (normal…)
  - for implicit surfaces $F(x,y,z)=0$: normal $\mathbf{n}(x,y,z)$ can be easily computed at every intersection point using the gradient

$$\mathbf{n}(x, y, z) = \begin{pmatrix} \partial F(x, y, z) / \partial x \\ \partial F(x, y, z) / \partial y \\ \partial F(x, y, z) / \partial z \end{pmatrix}$$

  - example: $F(x, y, z) = x^2 + y^2 + z^2 - r^2$

$$\mathbf{n}(x, y, z) = \begin{pmatrix} 2x \\ 2y \\ 2z \end{pmatrix}$$

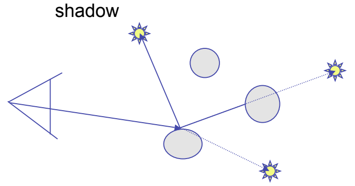
needs to be normalized!

Local Lighting

- local surface information
  - alternatively: can interpolate per-vertex information for triangles/meshes as in rendering pipeline
    - now easy to use Phong shading!
      - as discussed for rendering pipeline
  - difference with rendering pipeline:
    - interpolation cannot be done incrementally
    - have to compute barycentric coordinates for every intersection point (e.g plane equation for triangles)

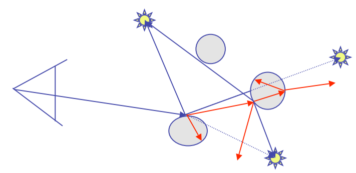
Global Shadows

- approach
  - to test whether point is in shadow, send out **shadow rays** to all light sources
    - if ray hits another object, the point lies in shadow



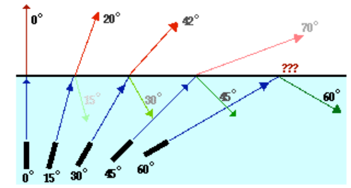
Global Reflections/Refractions

- approach
  - send rays out in reflected and refracted direction to gather incoming light
  - that light is multiplied by local surface color and added to result of local shading



Total Internal Reflection

As the angle of incidence increases from 0 to greater angles ...



...the refracted ray becomes dimmer (there is less refraction)
...the reflected ray becomes brighter (there is more reflection)
...the angle of refraction approaches 90 degrees until finally a refracted ray can no longer be seen.

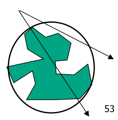
http://www.physicsclassroom.com/Class/refrn/U14L3b.html

Ray Tracing

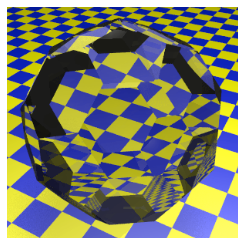
- issues:
  - generation of rays
  - intersection of rays with geometric primitives
  - geometric transformations
  - lighting and shading
  - efficient data structures so we don't have to test intersection with every object

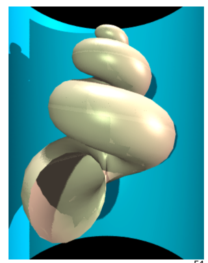
Optimized Ray-Tracing

- basic algorithm simple but very expensive
- optimize by reducing:
  - number of rays traced
  - number of ray-object intersection calculations
- methods
  - bounding volumes: boxes, spheres
  - spatial subdivision
    - uniform
    - BSP trees
- (more on this later with collision)



Example Images



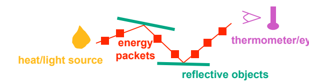


Radiosity

- radiosity definition
  - rate at which energy emitted or reflected by a surface
- radiosity methods
  - capture diffuse-diffuse bouncing of light
    - indirect effects difficult to handle with raytracing

Radiosity

- illumination as radiative heat transfer

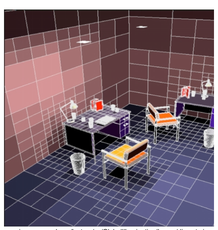


- conserve light energy in a volume
- model light transport as packet flow until convergence
- solution captures diffuse-diffuse bouncing of light

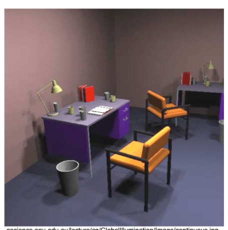
- view-independent technique
  - calculate solution for entire scene offline
  - browse from any viewpoint in realtime

Radiosity

- divide surfaces into small patches
- loop: check for light exchange between all pairs
  - form factor: orientation of one patch wrt other patch (n x n matrix)



escience.anu.edu.au/lecture/cg/GlobalIllumination/Image/discrete.jpg

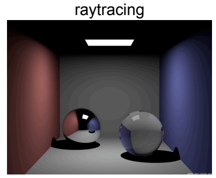


escience.anu.edu.au/lecture/cg/GlobalIllumination/Image/continuous.jpg

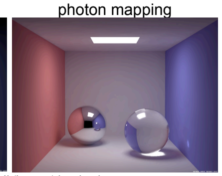
Better Global Illumination

- ray-tracing: great specular, approx. diffuse
  - view dependent
- radiosity: great diffuse, specular ignored
  - view independent, mostly-enclosed volumes
- photon mapping: superset of raytracing and radiosity
  - view dependent, handles both diffuse and specular well

raytracing



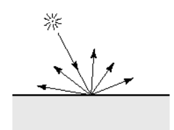
photon mapping

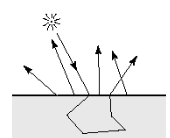


graphics.ucsd.edu/~henrik/images/cbox.html

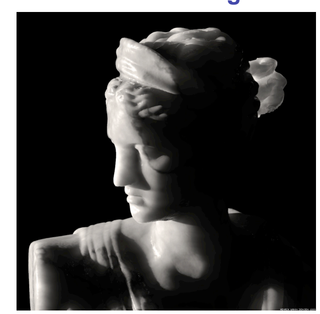
Subsurface Scattering: Translucency

- light enters and leaves at *different* locations on the surface
  - bounces around inside
- technical Academy Award, 2003
  - Jensen, Marschner, Hanrahan

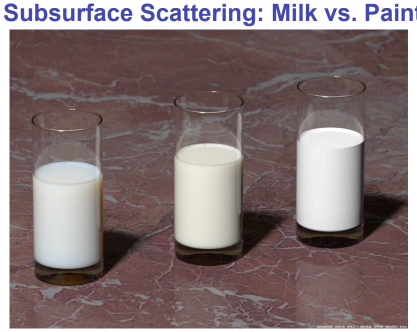




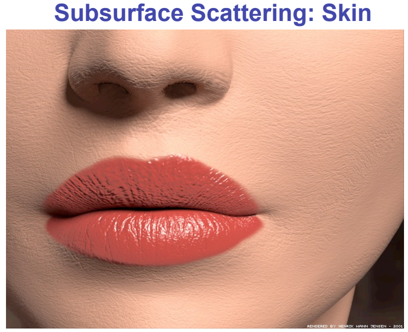
Subsurface Scattering: Marble



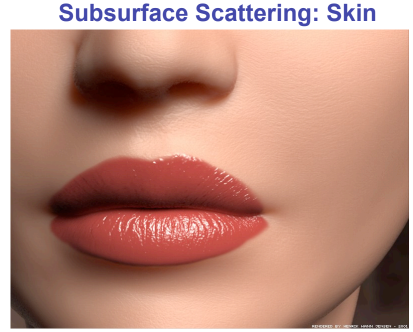
Subsurface Scattering: Milk vs. Paint



Subsurface Scattering: Skin

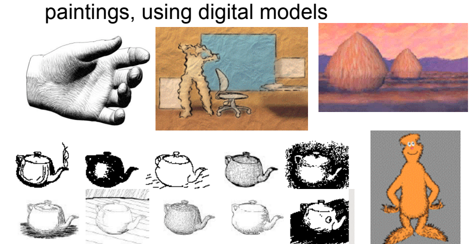


Subsurface Scattering: Skin



Non-Photorealistic Rendering

- simulate look of hand-drawn sketches or paintings, using digital models



www.red3d.com/cwr/npr/