Advanced Computer Graphics
Introduction to
Ray-Tracing and
Physically-Based
Rendering

G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de
Claude Monet's Haystacks
Effects Needed for Realistic Rendering

- Remember one of the local lighting models from CG1?
- All *local* lighting models fail to render one or more of the following effects:
  - Soft Shadows (Halbschatten)
    - Hard shadows (Schlagschatten) can be done using multi-pass OpenGL rendering (see CG1)
  - Indirect lighting (sometimes in the form of "color bleeding")
  - Reflection on glossy surfaces, e.g., mirrors
  - Refraction, e.g., on water or glass surfaces
  - Diffraction (Beugung)
  - ...

- Global Illumination
The Rendering Equation

- **Goal:** photorealistic rendering
- **The "solution":** the rendering equation

\[ L_r(x, \omega_r) = L_e(x, \omega_r) + \int_{\Omega} \rho(x, \omega_r, \omega_i) L_i(x, \omega_i) \cos(\theta_i) \, d\omega_i \]

- \( L_i \) = the "amount" of light incident on \( x \) from direction \( \omega_i \)
- \( L_e \) = the "amount" of light emitted (i.e., "produced") from \( x \) into direction \( \omega_r \)
- \( L_r \) = the "amount" of light reflected from \( x \) into direction \( \omega_r \)
- \( \rho \) = function of the reflectance coefficient = BRDF (see CG1)
- \( \omega = (\theta, \varphi) \) = a direction (two polar angles)
- \( \Omega \) = hemisphere around the normal
\[ L_r = L_e + \int_{\Omega} \rho \cdot L_i \cdot \cos(\theta) \, d\omega \]
Approximations to the Rendering Equation

- Solving the rendering equation is impossible!
- Observation: the rendering equation is a recursive function
- Consequently, a number of approximation methods have been developed that are based on the idea of following rays:
  - **Radiosity** [Goral et. al, Siggraph 1984, "Modeling the Interaction of Light between diffuse Surface"]
- Current state of the art:
  - Ray-tracing, combined with photon tracing, combined with Monte Carlo methods

Turner Whitted, Microsoft Research
The Simple "Whitted-style" Ray-Tracing

- Synthetic camera = viewpoint + image plane in world space
1. Shoot rays from camera through every pixel into scene (primary rays)
2. If the ray hits more than one object, then consider only the first hit
3. From there, shoot rays to all light sources (shadow feelers)
4. If a shadow feeler hits another obj → point is in shadow w.r.t. that light source. Otherwise, evaluate a lighting model (e.g., Phong [see CG1])
5. If the hit obj is glossy, then shoot reflected rays into scene (secondary rays) → recursion
6. If the hit object is transparent, then shoot refracted ray → more recursion
- Visualizing the ray tree can be very helpful for debugging

**diagram:**
- reflected ray
- shadow ray
- transmitted (refracted) ray
The Ray Tree

- Basic idea of ray-tracing: construct ray paths from the light sources to the eye, but follow those paths "backwards"
- Leads (conceptually!) to a tree, the ray tree:

E1 = primary ray  
Ri = reflected rays  
Ti = transmitted rays  
Si = shadow rays
- Each recursive algorithm needs a criterion for stopping:
  - If the maximum recursion depth is reached (fail-safe criterion)
  - If the contribution to a pixel's color is too small (decreases with $\text{depth}^n$)

Ray depth: 128 (!)

https://renderman.pixar.com/stories/piper
Excerpt from "Piper", Pixar 2017
One of the First Ray-Traced Images

Turner Whitted 1980
The principle of ray-tracing is so easy that you can write a "complete" ray-tracer on the back of a business card:

typedef struct {double x,y,z} vec; vec U, black, amb = {.02, .02, .02}; struct sphere{
  vec cen, color; double rad, kd, ks, kl, ir, it, kt, kl, ir;
  struct sphere* s, * best, sph[] = {0., 6., 1., 1., 1., 9, .05, .25, .5, 1.7, .8, .15, .5, .21, .7, 0., 0., 0., 6.1, 3.5, -.3, .12, .81, 1., .5, 0., 0., 0., 5, 1.5};
  vec double u, b, tmin, sqrt(), tan();
  double vdot(A, B) vec A, B; {return A.x * B.x + A.y * B.y + A.z * B.z;}
  vec vunit(A) vec A; {return vcomb(1. / sqrt(vdot(A, A)), A, black);} struct sphere* intersect(P, D) vec P, D; {
    best = 0; tmin = 1e30; s = sph + 5; while (s-- > sph) b = vdot(D, U = vcomb(-1., P - s > cen)), u = b - b * vdot(D, U) + s > rad * s > rad, u > 07? sqrt(u) : 1e31, u = b - b > 1e-7? b - u : b + u, tmin = u > 1e-7 & & u < tmin? best = s, u: tmin; return best; }
  vec trace(level, P, D) vec P, D; {double d, eta, e;
    struct sphere* s, * l;
    if (! level--)
      return black; if (s = intersect(P, D)); else return amb; color = amb; eta = s > ir; d = vdot(D, N = vunit(vcomb(-1., P = vcomb(tmin, D, P), s > cen)));
    if (d < 0) N = vcomb(-1., N, black), eta = 1 / eta, d = -d, l = sph + 5; while (l --> sph) if (e = 1 - kl * vdot(N, U = vunit(vcomb(-1., P, l > cen)))) > 0 & & intersect(P, U) == l) color = vcomb(e, 1 > color, color, & s > color, color); U = s > color, color, y * = U, y, color, z * = U, z; e = 1 - eta * eta * (1 - d); return vcomb(s < -kl, e > 0? trace(level, P, vcomb(eta, D, vcomb(eta * d - sqrt(e), N, black))) : black, vcomb(s > ks, trace(level, P, vcomb(2 * d, N, D)), vcomb(s > kd, color, vcomb(s > -kl, U, black)))); main() {printf("%d %d\n", 32, 32); while (yx < 32 * 32) U.x = yx * 32 - 32 * 2, U.z = 322 / yx * 32 + 32, U.y = 322 / tan(25 / 114.591590261), U = vcomb(255., trace(3, black, vunit(U)), black), printf("%.0f %.0f %.0f\n", U)); /*minray*/

(Also won the International Obfuscated C Code Contest) [Paul Heckbert, ca. 1994]
Interactive Demo

http://www.siggraph.org/education/materials/HyperGraph/raytrace/rt_java/raytrace.html
Digression

- The ancient explanation for our capability of seeing: "seeing rays"
Albrecht Dürer's "Ray Casting Machines" [16th century]
Examples of Ray-Traced Images
Ray Tracing in the Animation Industry

*Doc Hudson’s chrome bumper with two levels of ray-traced reflection.*  
(Copyright 2006 Disney/Pixar)

Ray-traced wine glasses from *Ratatouille.*  
(Copyright 2007 Disney/Pixar)
Fake or Real?

The Challenge

Take a look at the ten images below. Some of them are photographs of real objects or scenes, others are created by computer graphics (CG) artists. Test your ability to tell which among the array of images are real, and which are CG. If you want a closer look, click the image to see a larger view of the picture. Once you've decided what's what, click either CG or REAL to begin the tally of your score. Work through each of the ten images. When you've finished, you'll be prompted to get your score.
The Principle of Ray-Tracing vs. Principle of Polygonal Rendering

**Polygonal (OpenGL) rendering** is a "forward-mapping" approach

**Raytracing** can be considered an "inverse-mapping" approach

for each polygon:
  for each pixel:
  ...
- Scan conversion = *object-order rendering*: start with triangles, project each vertex = send ray through each vertex

![Diagram of scan conversion]

- Raytracing = *image-order rendering*: start with pixels, send ray through each pixel

![Diagram of raytracing]
- For rendering a complete scene using scan conversion ...

... scan-convert each triangle

- For rendering a complete scene using raytracing ...

... trace a ray through each pixel