Raytracing Basics

rendered in a matter of seconds with Travis Fischer’s ’09 raytracer
What “effects” do you see?

- red mirror
- green mirror
- "soft" illumination changes
- soft shadow?

No – due to complex scene and multiple light sources

rendered in a matter of seconds with Travis Fischer’s ’09 raytracer
Rendering with Raytracing (1/2)

Simple idea: instead of forward mapping infinite number of rays from light source to object to viewer, backmap finite number of rays from viewer through each sample to object to light source (or other object)

- Generalizing from Durer’s painting showing perspective projection

- Durer: Put eye at center of projection on wall and record string/plane intersection
- Raytracing: shoot rays from eye through sample point (e.g., a pixel center) of a virtual photo (the image or film plane/screen)
  - compute color/intensity at the ray/object intersection
Rendering with Raytracing (2/2)

Subproblems to solve

- Generate primary (‘eye’) ray
  - ray goes out from eye through a pixel center (or any other sample point on image plane)
- Find closest object along ray path
  - find first intersection between ray and an object in scene
- Calculate lighting
  - use illumination model to determine direct contribution from light sources
  - recursively generate secondary rays at equal and opposite angle to capture indirect contributions from inter-object reflections (specular components only) which themselves have direct and indirect contributions, etc
Raytracing versus scan conversion

Scan conversion
- After meshing and projection
  for each triangle in scene...

Raytracing
- For each sample in pixel image...
  • Avoid meshing
  • Explicit projection of triangles by working directly with analytical surfaces
Raytracing versus scan conversion

- How is raytracing different from what you’ve been doing in your assignments? In Shapes, Sceneview you did:
  for each object in scene
    for each triangular facet of object
      pass vertex geometry, colors to OpenGL, which paints all interior points of triangle in framebuffer using Gouraud shading

- This is fine if you’re just using the simple hardware lighting model, and just using triangles

- We’re trying to solve a different problem:
  for each sample in our image
    determine which object in scene is hit by ray through that sample;
    paint that sample the color of the object at that point (based on evaluating lighting model at point)
Generating Rays \((1/4)\)

**Ray origin**

- Let’s look at geometry of problem in \textit{un-transformed} world-space (i.e., with perspective view volume)
  - we’ll see why later
- Start a ray from an “eye point”: \(P\)
- Send it out in some direction \(d\) from eye toward a point on film plane (a rectangle in the U-V plane in the camera’s UVN space) whose color we want to know
- Points along ray have form \(P + td\) where
  - \(P\) is ray’s base point: the camera’s eye
  - \(d\) is unit vector direction of ray
  - \(t\) is a nonnegative real number
- “Eye point” is center of projection in perspective view volume (view frustum)
- Don’t use de-perspectivizing ‘unhinging’ step; to avoid dealing with inverse of perspective transformation later
**Ray direction**

- Start with screen-space 2D points (pixels). Need to find a point in 3D that defines corresponding ray
  - we’ll use ray to intersect with original objects in original, untransformed world coordinate system
- Transform 2D screen-space points into points on camera’s film plane located in 3D space
- Any plane orthogonal to look vector is a convenient film plane: $z = k$ in canonical view volume

Choose a plane to be the film plane and then create a function that maps screen-space points onto it
- what’s a convenient plane? Try the far plane :-)  
- to convert, we just have to scale integer screen-space coordinates into real values between -1 and 1
Generating Rays (3/4)

**Ray direction (cont.)**

- Transform film plane point into world-space point
  - we can make direction vector between eye (at CoP) and this world-space point
  - we need vector to be in world-space in order to intersect with original object in world coordinate system; intersection point of the world-space geometry is needed for the illumination model

- Normalizing transformation takes world-space points to points in canonical view volume
  - **translate** to origin; **orient** so *Look* pointing down $-Z$, *Up* along $Y$; **scale** $x$ and $y$ to adjust viewing angles to 45°, scale $z$: $[-1, 0]$; $x, y$: $[-1, 1]$

- Apply inverse of normalizing transformation: **Viewing Transformation**
Generating Rays (4/4)

Summary

- Start ray at center of projection (eye point)

- Map 2D integer screen-space point onto 3D film plane
  - scale $x$, $y$ to fit between -1 and 1
  - set $z$ to -1 so points lie on far clip plane as film plane

- Transform 3D film plane point (mapped pixel) into untransformed world coordinate system point
  - need to undo normalizing transformation (i.e., viewing transformation)

- Construct direction vector
  - point minus point is a vector
  - world-space point (mapped pixel) minus eye point
Ray-Object Intersection (1/6)

**Implicit objects**

- If an object is defined implicitly by a function $f$ such that $f(Q) = 0$ IFF $Q$ is a point on surface of object, then ray-object intersection is relatively easy
  - can define many objects implicitly
  - implicit functions provide potentially infinite resolution
  - tessellating implicit functions is more difficult than using them directly

- For example, a circle of radius $R$ is an implicit object in the plane, and its equation is
  $$ f(x,y) = x^2 + y^2 - R^2 $$
  - points where $f(x,y) = 0$ are points on the circle

- An infinite plane is defined by the function:
  $$ f(x,y,z) = Ax + By + Cz + D $$

- A sphere of radius $R$ in 3-space:
  $$ f(x,y,z) = x^2 + y^2 + z^2 - R^2 $$
Ray-Object Intersection (2/6)

Implicit objects (cont.)

- At what points (if any) does ray intersect object?
- Points on ray have form \( P + td \)
  - \( t \) is any nonnegative real
- A point \( Q \) lying on object has property that \( f(Q) = 0 \)
- Combining, we want to know “For which values of \( t \) is \( f(P + td) = 0 \)” (if any)

We are solving a system of simultaneous equations in \( x, y \) (in 2D) or \( x, y, z \) (in 3D)
An Explicit Example (1/3)

2D ray-circle intersection example

• Consider the eye-point $P = (-3, 1)$, the direction vector $d = (.8, -.6)$ and the unit circle given by:

$$f(x,y) = x^2 + y^2 - R^2$$

• A typical point of the ray is:

$$Q = P + td = (-3,1) + t(.8,-.6) = (-3 + .8t,1 - .6t)$$

• Plugging this into the equation of the circle:

$$f(Q) = f(-3 + .8t,1 - .6t) = (-3+.8t)^2 + (1-.6t)^2 - 1$$

• Expanding, we get:

$$9 - 4.8t + .64t^2 + 1 - 1.2t + .36t^2 - 1$$

• Setting this to zero, we get:

$$t^2 - 6t + 9 = 0$$
An Explicit Example (2/3)

2D ray-circle intersection example (cont.)

• Using the quadratic formula:

\[
\text{roots} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\]

• We get:

\[
t = \frac{6 \pm \sqrt{36 - 36}}{2}, \quad t = 3, 3
\]

• Because we have a root of multiplicity 2, ray intersects circle at one point (i.e., it’s tangent to the circle)

• We can use discriminant \( D = b^2 - 4ac \) to quickly determine if a ray intersects a curve or not
  - if \( D < 0 \), imaginary roots; no intersection
  - if \( D = 0 \), double root; ray is tangent
  - if \( D > 0 \), two real roots; ray intersects circle at two points

• Smallest non-negative real \( t \) represents intersection nearest to eye-point
2D ray-circle intersection example (cont.)

- Generalizing: our approach will be to take an arbitrary implicit surface with equation \( f(Q) = 0 \), a ray \( P + td \), and plug the latter into the former:

\[
  f(P + td) = 0
\]

- This results, after some algebra, in an equation with \( t \) as unknown
- We then solve for \( t \), analytically or numerically
Implicit objects-multiple conditions

- For objects like cylinders, the equation
  \[ x^2 + z^2 - 1 = 0 \]
  in 3-space defines an infinite cylinder of unit radius, running along the y-axis.

- Usually, it’s more useful to work with finite objects, e.g. such a unit cylinder truncated with the limits
  \[
  y \leq 1 \\
  y \geq 1
  \]

- But how do we do the “caps?”
- The cap is the inside of the cylinder at the y extrema of the cylinder
  \[ x^2 + z^2 - 1 < 0, \ y = \pm 1 \]
Ray-Object Intersection (4/6)

Multiple conditions (cont.)

- We want intersections satisfying the implicit equation for a cylinder:
  \[ x^2 + z^2 - 1 = 0 \]
  \[-1 \leq y \leq 1 \]
  or top cap:
  \[ x^2 + z^2 - 1 \leq 0 \]
  \[ y = 1 \]
  or bottom cap:
  \[ x^2 + z^2 - 1 \leq 0 \]
  \[ y = -1 \]
Ray-Object Intersection (5/6)

Multiple conditions-cylinder pseudocode

- Solve in a case-by-case approach

```plaintext
Ray_inter_finite_cylinder(P, d):
  // Check for intersection with infinite cylinder
  t1, t2 = ray_inter_infinite_cylinder(P, d)
    compute P + t1*d, P + t2*d
  // If intersection, is it between “end caps”?
  if y > 1 or y < -1 for t1 or t2, toss it

  // Check for intersection with top end cap
  Compute ray_inter_plane(t3, plane y = 1)
  Compute P + t3*d
  // If it intersects, is it within cap circle?
  if x^2 + z^2 > 1, toss out t3

  // Check intersection with other end cap
  Compute ray_inter_plane(t4, plane y = -1)
  Compute P + t4*d
  // If it intersects, is it within cap circle?
  if x^2 + z^2 > 1, toss out t4

Among all the t’s that remain (1-4), select the smallest non-negative one
```
Implicit surface strategy summary

- Substitute ray \((P + td)\) into implicit surface equations and solve for \(t\)
  
  - surface you see “first” from eye point is at smallest non-negative \(t\)-value

- For complicated objects (not defined by a single equation), write out a set of equalities and inequalities and then code as cases...

- Latter approach can be generalized cleverly to handle all sorts of complex combinations of objects
  
  - Constructive Solid Geometry (CSG), where objects are stored as a hierarchy of primitives and 3-D set operations (union, intersection, difference)
  
  - “blobby objects”, which are implicit surfaces defined by sums of implicit equations \((F(x,y,z) = 0)\)

\[
F(x,y,z) = ((x^2(1-x^2)-y^2)^2+0.5z^2-f(1+b*(x^2+y^2+z^2))) = 0
\]
World Space Intersection

World space - a global view

- We need to intersect each object in world coordinates to compute illumination there. Need an analytical description of object in world space.
- Example: a unit sphere translated to \((3, 4, 5)\) after it was scaled by 2 in the x-direction has equation

\[
f(x, y, z) = \frac{(x-3)^2}{2^2} + (y-4)^2 + (z-5)^2 = 0.5^2
\]

- One can take ray \(P + td\) and plug it into the equation

\[
f(P + td) = 0
\]

- Solve resulting equation for \(t\)
- Start with untransformed object definition in its own coordinate system
- Try to derive what transformed version should be, given CTM (call it \(M\))
  - not easy for general transformations; furthermore, transformed version of equation is always more complicated and thus more expensive to compute
  - can we just work with the object in its own coordinate system? Yes, as follows
Object Space Intersection

Transform ray into object space

- Express world-space point of intersection as \( MQ \), where Q is some point in object-space:

\[
P + td = MQ
\]

\[
M^{-1} \cdot (P + td) = Q
\]

\[
M^{-1}P + tM^{-1}d = Q
\]

Let \( \tilde{P} = M^{-1}P \), \( \tilde{d} = M^{-1}d \)

- If \( \tilde{f}(x, y, z) \) is the equation of the untransformed object, we just have to solve

\[
\tilde{f}(\tilde{P} + t\tilde{d}) = 0
\]

- note \( \tilde{d} \) is probably not a unit vector

- the parameter \( t \) along this vector and its world space counterpart always have the same value.

Normalizing \( \tilde{d} \) would alter this relationship.

Do NOT normalize \( \tilde{d} \).
World Space vs. Object Space

- To compute world space intersections, have to transform implicit equation of canonical object defined in object space - *often difficult*
- To compute intersections in object space, need only apply a matrix \((M^{-1})\) to \(P\) and \(d\) - *much simpler*
  - does \(M^{-1}\) exist?
  - \(M\) was composed from two parts: the cumulative modeling transformation that positions the object in world-space, and the camera’s normalizing transformation
  - modeling transformations are just translations, rotations, and scales (all invertible)
  - normalizing transformation also consists of translations, rotations and scales (also invertible); but the perspective transformation is not invertible! (Now you see why we used the perspective view volume directly rather than de-perspectivizing it to the canonical cuboid: slide 9)

- When you’re done, you get a \(t\)-value
- This \(t\) can be used in two ways:
  - \(P + td\) is the world-space location of the intersection between ray and transformed object
  - \(\tilde{P} + t\tilde{d}\) is the corresponding point on untransformed object (in object space)
Normal Vectors at Intersection Points (1/4)

Normal vector to implicit surfaces

- For illumination, you need the normal at the point of intersection in world space.
- Instead we’ll start by solving for point of intersection in the object's own space and computing normal there; then transform the object space normal to the world space.

- If a surface bounds a solid whose interior is given by
  \[ f(x, y, z) < 0 \]
  then can find normal vector at point \((x, y, z)\) via gradient at that point:
  \[ n = \nabla f(x, y, z) \]

- Recall that the gradient is a vector with three components, the partial derivatives:
  \[ \nabla f(x, y, z) = \left( \frac{\partial f}{\partial x}(x, y, z), \frac{\partial f}{\partial y}(x, y, z), \frac{\partial f}{\partial z}(x, y, z) \right) \]
Normal Vectors at Intersection Points (2/4)

**Sphere normal vector example**

- For the sphere, the equation is
  \[ f(x, y, z) = x^2 + y^2 + z^2 - 1 \]

- The partial derivatives are
  \[ \frac{\partial f}{\partial x}(x, y, z) = 2x \]
  \[ \frac{\partial f}{\partial y}(x, y, z) = 2y \]
  \[ \frac{\partial f}{\partial z}(x, y, z) = 2z \]

- So the gradient is
  \[ \mathbf{n} = \nabla f(x, y, z) = (2x, 2y, 2z) \]

- Normalize \( \mathbf{n} \) before using in dot products!

- In some degenerate cases, the gradient may be zero, and this method fails...use nearby gradient as a cheap hack.
Normal Vectors at Intersection Points (3/4)

**Transforming back to world space**

- Have an object space normal vector
- Want a world space normal vector for lighting equation
- To transform object to world coordinates, we just multiplied its vertices by $M$, the object’s CTM
- Can we treat the normal vector the same way? Answer: NO

\[ \mathbf{n}_{\text{world}} \neq M \mathbf{n}_{\text{object}} \]

- Example: say $M$ scales in $x$ by .5 and $y$ by 2

Wrong!

- Normal must be perpendicular

- See the normal scaling applets in *Exploratories → Lighting and Shading*
Normal Vectors at Intersection Points (4/4)

- Why doesn’t multiplying the normal work?
- For translation and rotation, which are rigid body transformations, it actually does work
- Scale, however, distorts the normal in exactly the opposite sense of the scale applied to the surface
  - scaling y by 2 causes normal to scale by \( \frac{1}{2} \):

\[
\begin{align*}
y = 2x & \quad \quad \quad \quad \quad \quad y = 4x \\
\vec{n} = (1, -\frac{1}{2}) & \quad \quad \quad \quad \quad \quad \vec{n} = (1, -\frac{1}{4})
\end{align*}
\]

- We’ll see this algebraically in the next slides
Transforming Normals \((1/4)\)

Object-space to world-space

- Take polygonal case, for example
- Let’s compute relationship between object-space normal \(n_{obj}\) to polygon \(H\) and world-space normal \(n_{world}\) to transformed version of \(H\), called \(MH\)
- For any vector \(v\) in world space that lies in polygon (e.g., one of edge vectors), normal is perpendicular to \(v\):
  \[ n_{world} \cdot v_{world} = 0 \]
- But \(v_{world}\) is just a transformed version of some vector in object space, \(v_{obj}\). So we could write
  \[ n_{world} \cdot Mv_{obj} = 0 \]
- Recall that since vectors have no position they are unaffected by translations (they have \(w = 0\))
  - so to make things easier, we can consider only
    \[ M_3 = \text{upper left 3 x 3 of } M \] (rotation/scale component)
    \[ n_{world} \cdot M_3 v_{obj} = 0 \]
Transforming Normals (2/4)

Object-space to world-space (cont.)

- So we want a vector $n_{\text{world}}$ such that for any $v_{obj}$ in the plane of the polygon
  \[ n_{\text{world}} \cdot M_3 v_{obj} = 0 \]

- We will show on next slide that this equation can be rewritten as
  \[ M_3^t n_{\text{world}} \cdot v_{obj} = 0 \]

- We also already have
  \[ n_{obj} \cdot v_{obj} = 0 \]

- Therefore
  \[ M_3^t n_{\text{world}} = n_{obj} \]

- Left-multiplying by $(M_3^t)^{-1}$,
  \[ n_{\text{world}} = (M_3^t)^{t} n_{obj} \]
Transforming Normals \((3/4)\)

*Object-space to world-space*(cont.)

- So how did we rewrite this:
  \[
  n_{\text{world}} \cdot M_3v_{\text{obj}} = 0
  \]
- As this:
  \[
  (M_3^t n_{\text{world}}) \cdot v_{\text{obj}} = 0
  \]
- Recall that if we think of vector as \(n \times 1\) matrices, then switching notation,
  \[
  a \cdot b = a^t b
  \]
- Rewriting our original formula, we thus have
  \[
  n^t_{\text{world}} M_3 v_{\text{obj}} = 0
  \]
- Writing \(M = M^{tt}\), we get
  \[
  n^t_{\text{world}} M_3^{tt} v_{\text{obj}} = 0
  \]
- Recalling that \((AB)^t = B^tA^t\), we can write this:
  \[
  (M_3^t n_{\text{world}})^t v_{\text{obj}} = 0
  \]
- Switching back to dot product notation, our result:
  \[
  (M_3^t n_{\text{world}}) \cdot v_{\text{obj}} = 0
  \]
Transforming Normals (4/4)

Applying inverse-transpose of $M$ to normals

- So we ended up with
  
  $$ n_{world} = \left( M_3^t \right)^{-1} n_{obj} $$

- “Invert” and “transpose” can be swapped, to get our final form:
  
  $$ n_{world} = \left( M_3^{-1} \right)^t n_{obj} $$

- Why do we do this? It’s easier!
  - Instead of inverting composite matrix, accumulate composite of inverses which are easy to take for each individual transformation

- A hand-waving interpretation of $(M_3^{-1})^t$
  - $M_3$ is composition of rotations and scales, $R$ and $S$ (why no translates?). Therefore
    
    $$ ((RS\ldots)^{-1})^t = (\ldots S^{-1} R^{-1})^t = ((R^{-1})^t (S^{-1})^t \ldots) $$
  
  - so we’re applying transformed (inverted, transposed) versions of each individual matrix in original order
  
  - for rotation matrix, transformed version equal to original rotation, i.e., normal rotates with object
    - $(R^{-1})^t = R$; inverse reverses rotation, and transpose reverses it back
  
  - for scale matrix, inverse inverts scale, while transpose does nothing:
    - $(S(x,y,z)^{-1})^t = S(x,y,z)^{-1} = S(1/x,1/y,1/z)$
Summary

Simple, non-recursive raytracer

\[ P = \text{eyePt} \]

\textbf{for} each sample of image

Compute d

\textbf{for} each object

Intersect ray \( P + td \) with object

Select object with smallest non-negative t-value (visible object)

For this object, find object space intersection point

Compute normal at that point

Transform normal to world space

Use world space normal for lighting computations
Shadows

• Each light in scene makes contribution to color and intensity of a surface element...

\[ \text{objectIntensity}_\lambda = \text{ambient} + \sum_{\text{light} = 1}^{\text{numLights}} \text{attenuation} \cdot \text{lightIntensity}_\lambda \cdot (\text{diffuse} + \text{specular}) \]

• IFF it reaches the object!
  - could be occluded by other objects in scene
  - could be self-occluding

• Construct a ray from surface to each light

• Check to see if ray intersects other objects before it gets to light
  - if ray does not intersect that same object or another object on its way to light, count full contribution of light
  - if ray does intersect an object on its way to light, then light does not contribute
  - causes hard shadows; soft shadows are harder to compute (must sample)

• what about transparent or specularly reflective objects? Such contributions are beginning of global illumination => need recursive raytracing
Recursive Raytracing (1/3)

Simulating global lighting effects (Whitted, 1979)

- By recursively casting new rays into scene, we can look for more information
- Start from point of intersection
- We’d like to send rays in all directions, but that’s too hard/computationally taxing
- Send rays in directions likely to contribute most:
  - toward lights (blockers to lights create shadows for those lights)
  - specular bounce off other objects to capture specular inter-object reflections
  - use ambient hack to capture diffuse inter-object reflection
  - through object (transparency)
- For more info on recursive ray tracing, see section 16.12 of the old textbook
Recursive Raytracing (2/3)

- Trace ‘secondary’ rays at intersections:
  - light: trace ray to each light source. If light source blocked by opaque object, it does not contribute to lighting
  - specular reflection: trace reflecting ray in mirror direction
  - refractive transmission/transparency: trace ray in refraction direction by Snell’s law
  - recursively spawn new light, reflection, refraction rays at each intersection until contribution negligible / max recursion depth reached

- Your new lighting equation (Phong lighting)...

\[
I_\lambda = I_{a\lambda} k_a O_{d\lambda} + \sum_m f_{\text{att}} I_{p\lambda} [k_d O_{d\lambda} \hat{N} \cdot \hat{L}] + k_s O_{s\lambda} (\hat{R} \cdot \hat{V})^n + k_t O_{t\lambda} + I_{t\lambda}
\]

- note: intensity from recursive rays calculated with same lighting eqn
- light sources contribute specular and diffuse lighting

- Limitations
  - recursive inter-object reflection is strictly specular
  - diffuse inter-object reflection handled differently

- Oldie-but-goody videos
  - Silent Movie on raytracing,
  - A Long Ray’s Journey into Light
Recursive Raytracing (3/3)

**Light-ray Trees**

![Diagram of light-ray trees showing indirect illumination and recursive raytracing](image)

- $\vec{N}_i$: Surface normal
- $\vec{R}_i$: Reflected ray
- $\vec{L}_{ia}$: Light ray (ray to light a)
- $\vec{T}_i$: Transmitted ray
- $P_i$: Intersection Point

**Light List**

```
[ ] a
[ ] b
[ ] c
[ ] ...
[ ] n
```

Light rays cast to each light in list to determine contribution; if a light ray intersects an object in front of a light, that light does not contribute.
• A ray traced image with recursive ray tracing, transparency and refractions

Whitted 1980
For a partially transparent polygon

\[ I_\lambda = (1 - k_{t1})I_{\lambda 1} + k_{t1}I_{\lambda 2} \]

- \( k_{t1} = \) transmittance of polygon 1
  - (0 = opaque; 1 = transparent)
- \( I_{\lambda 1} = \) intensity calculated for polygon 1
- \( I_{\lambda 2} = \) intensity calculated for polygon 2
Transparent Surfaces (2/2)

**Refractive transparency**

- We model the way light bends at interfaces with Snell’s Law

\[
\sin \theta_r = \frac{\sin \theta_i \eta_{i\lambda}}{\eta_{r\lambda}}
\]

\(\eta_{i\lambda}\) = index of refraction of medium 1

\(\eta_{r\lambda}\) = index of refraction of medium 2
Choosing Samples

• In the examples and in your assignment we sample once per pixel and get images similar to the one below.
• We have a case of the jaggies.
• Can we do better?
Choosing Samples

- In the simplest case, choose our sample points at pixel centers.
- For better results, can *supersample*:
  - e.g., at corners and at center.
- Even better techniques do *adaptive sampling*: increase sample density in areas of rapid change (in geometry or lighting).
- With *stochastic sampling*, samples are taken probabilistically; as mentioned in image processing lecture, converges faster than regularly spaced sampling.
- For fast results, can *subsample*: fewer samples than pixels:
  - take as many samples as time permits.
  - *beam tracing*: track a bundle of neighboring rays together.
- How to convert samples to pixels? Filter to get weighted average of samples.

Instead of shooting one ray, you can sample within a region to create a better approximation.
Super Samples

With SS  
Without SS
Raytracing Pipeline

**Pipeline**

- Raytracer produces visible samples from model
  - samples convolved with filter to form pixel image
- Additional pre-processing
  - pre-process database to speed up per-sample calculations
  - e.g., organize by spatial partitioning via bins and/or bounding boxes (see Visible Surface Determination lecture)
  - done once, sometimes must be redone if objects transformed (resize, translate)

For each desired sample:
(some (u,v) on film plane)

- Traverse model
- Accumulate CMTM
- Spatially organize objects

Object database suitable for ray-tracing

Loop over objects
- Intersect each with ray
- Keep track of smallest t

Closest point

Light the sample

Generate secondary rays

All samples

Filter

Pixels of final image

Andries van Dam November 3, 2009 Raytracing 42/46
Real-time Raytracing (1/3)

- Traditionally computationally impossible to do in real-time
  - “Embarrassing” parallelism due to the independence of each ray
  - Hard to make hardware optimized for raytracing:
    - large amount of floating point calculations
    - complex control flow structure
    - complex memory access for scene data

- One solution: software-based, highly optimized raytracer using cluster with multiple CPUs
  - Hard to have widespread adoption because of size and cost
  - May be possible with the advance of commercially available multi-core CPUs
  - Example: OpenRT project (http://www.openrt.de)

OpenRT rendering of five maple trees and 28,000 sunflowers (35,000 triangles) on 48 CPUs
Real-time Raytracing (2/3)

- Another solution: Use regular GPU’s to speed up raytracing
  - Difficult: GPU traditionally specialized for rasterization
  - NVIDIA at SIGGRAPH 2008: demos HD resolution raytracing at 30fps
  - Uses four (unreleased, next-gen) GPUs in parallel
    - running specialized multithreading software
  - For full details, see http://developer.nvidia.com/object/nvision08-IRT.html.

Frame from 1920x1080 sequence ray-traced at 30fps (recursion depth: 3)
Real-time Raytracing (3/3)

• Yet another solution: custom hardware for doing raytracing
  – Custom chip with sole purpose of performing raytracing
  – Example: RPU, a custom chip to do ray tracing proposed at SIGGRAPH 2005. (http://graphics.cs.uni-b.de/~woop/rpu/rpu.html)
  – The 66 MHz prototype outperforms OpenRT running on a 2.66 GHz Intel Pentium 4

A scene (52,470 triangles) taken from the game UT2003 rendered in RPU
POV-Ray: Pretty Pictures

Free Advanced Raytracer

- Full-featured raytracer available online: povray.org
- Obligatory pretty pictures (see irtc.org):

Image credits: Jochen Diehl (top), Juergen Ahl (bottom left)