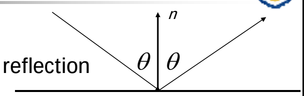
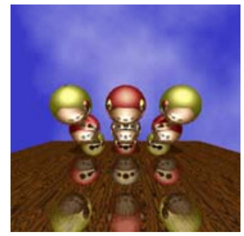
Chapter 11

Ray-Tracing

Reflection

- Mirror effects
  - Perfect specular reflection





Global Illumination Models

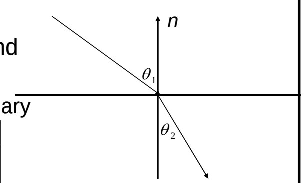
- Basic shading (rendering pipeline) = local illumination model
  - No object interaction
- Global illumination models require more sophisticated, computation-intensive algorithms
  - Ray Tracing
  - Global Illumination
- Ray-tracing
  - Usually offline (e.g. movies etc.)
    - research on making real-time
  - Flexible – can incorporate lots of phenomena

Refraction

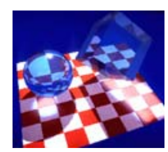
- Interface between transparent object and surrounding medium
  - E.g. glass/air boundary

$$c_2 \sin\theta_1 = c_1 \sin\theta_2$$

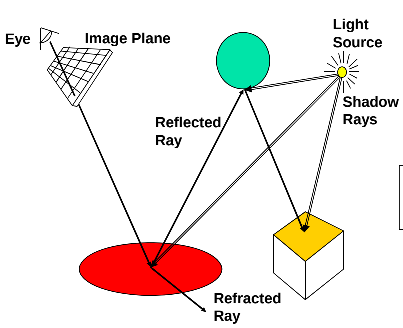
Snell's Law



- Light ray breaks (changes direction) based on refractive indices c_1, c_2



Ray-Tracing Algorithm



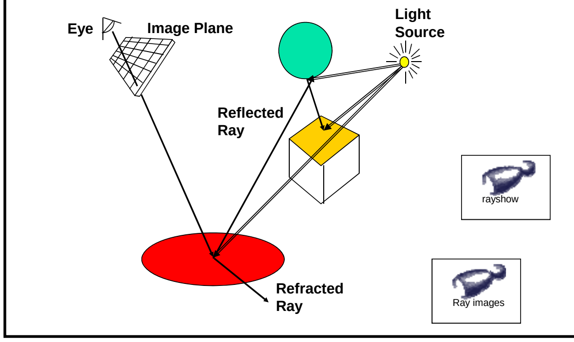
Basic Ray-Tracing Algorithm

```
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;
```

Sub-Routines

- ReflectRay(r,obj) – computes reflected ray (use obj normal at intersection)
- RefractRay(r,obj) - computes refracted ray
  - Note: ray is inside obj
- Shade(reflect_color,refract_color,obj) – compute illumination given three components

Ray-Tracing With Shadows



More About Ray-Tracing

- Algorithm above has a BUG....
- Does not terminate
- Termination Criteria
  - No intersection
  - Contribution of secondary ray attenuated below threshold – each reflection/refraction attenuates ray
  - Maximal depth is reached

Ray-Tracing: Practicalities

- Generation of rays
- Intersection of rays with geometric primitives
- Geometric transformations
- Lighting and shading
- Speed: Reducing number of intersection tests
  - E.g. use BSP trees or other types of space partitioning

Simulating Shadows

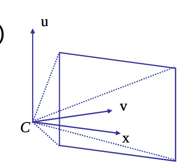
- Trace ray from each ray-object intersection point to light sources
  - If the ray intersects an object in between ⇒ point is shadowed from the light source

shadow = **RayTrace**(**LightRay**(obj,r,light));

return **Shade**(shadow,reflect_color,refract_color,obj);

Ray-Tracing: Generation of Rays

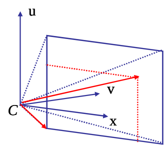
- Camera Coordinate System
  - Origin: C (camera position)
  - Viewing direction: $\mathbf{v}$
  - Up vector: $\mathbf{u}$
  - x direction: $\mathbf{x} = \mathbf{v} \times \mathbf{u}$



- Note:
  - Corresponds to viewing transformation in rendering pipeline!
  - See gluLookAt...

Ray-Tracing: Generation of Rays

- Other parameters:
  - Distance to 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):

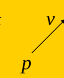
$$P_{i,j} = O + i \cdot \frac{r-l}{w-1} \cdot \mathbf{x} - j \cdot \frac{t-b}{h-1} \cdot \mathbf{u}$$
$$= O + i \cdot \Delta x \cdot \mathbf{x} - j \cdot \Delta y \cdot \mathbf{y}$$

Ray-Object Intersections

- Kernel of ray-tracing $\Rightarrow$ must be extremely efficient
- Usually involves solving a set of equations
  - Using implicit formulas for primitives

Example: Ray-Sphere intersection

ray: $x(t) = p_x + v_x t,\ y(t) = p_y + v_y t,\ z(t) = p_z + v_z t$
(unit) sphere: $x^2 + y^2 + z^2 = 1$
quadratic equation in t:



$$0 = (p_x + v_x t)^2 + (p_y + v_y t)^2 + (p_z + v_z t)^2 - 1$$
$$= t^2(v_x^2 + v_y^2 + v_z^2) + 2t(p_x v_x + p_y v_y + p_z v_z)$$
$$+ (p_x^2 + p_y^2 + p_z^2) - 1$$

Ray-Tracing: Generation of Rays

- 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…∞

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!)
    - Higher order functions (e.g. tori and other quartic functions)
      - In principle the same
      - But root-finding difficult
      - Net to resolve to numerical methods

Ray-Tracing: Practicalities

- Generation of rays
- **Intersection of rays with geometric primitives**
- Geometric transformations
- Lighting and shading
- Speed: Reducing number of intersection tests
  - E.g. use BSP trees or other types of space partitioning

Ray Intersections

- Other Primitives (cont)
  - 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 right side of every boundary edge
      - Similar to computation of outcodes in line clipping

Ray-Tracing: Practicalities

- Generation of rays
- Intersection of rays with geometric primitives
- **Geometric transformations**
- Lighting and shading
- Speed: Reducing number of intersection tests
  - E.g. use BSP trees or other types of space partitioning

Ray-Tracing: 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: Transformations

- Note: rays replace perspective transformation
- Geometric Transformations:
  - Similar goal as in rendering pipeline:
    - Modeling scenes convenient using different coordinate systems for individual objects
  - Problem:
    - Not all object representations are easy to transform
      - This problem is fixed in rendering pipeline by restriction to polygons (affine invariance!)

Ray-Tracing: Practicalities

- Generation of rays
- Intersection of rays with geometric primitives
- Geometric transformations
- **Lighting and shading**
- Speed: Reducing number of intersection tests
  - E.g. use BSP trees or other types of space partitioning

Ray-Tracing: Transformations

- Geometric Transformations:
  - Similar goal as in rendering pipeline:
    - Modeling scenes convenient using different coordinate systems for individual objects
  - Problem:
    - Not all object representations are easy to transform
      - This problem is fixed in rendering pipeline by restriction to polygons (affine invariance!)
    - Ray-Tracing has different solution:
      - The ray itself is always affine invariant!
      - Thus: transform ray into object coordinates!

Ray-Tracing: Local Lighting

- Light sources:
  - For the moment: point and directional lights
  - Later: are light sources
  - More complex lights are possible
    - Area lights
    - Global illumination
      - Other objects in the scene reflect light
      - Everything is a light source!
      - Talk about this on Monday

Ray-Tracing: 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:

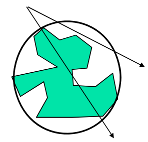
$$F(x,y,z)=x^2+y^2+z^2-r^2$$

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

Needs to be normalized!

Optimized Ray-Tracing

- Basic algorithm simple but VERY expensive
- Optimize…
  - Reduce number of rays traced
  - Reduce number of ray-object intersection calculations
- Methods
  - Bounding Boxes
  - Spatial Subdivision
    - Visibility & Intersection
  - Tree Pruning



Ray-Tracing: Local Lighting

- Local surface information
  - Alternatively: can interpolate per-vertex information for triangles/meshes as in rendering pipeline
    - Phong shading!
    - Same as discussed for rendering pipeline
  - Difference to rendering pipeline:
    - Interpolation cannot be done incrementally
    - Have to compute Barycentric coordinates for every intersection point (e.g plane equation for triangles)

Ray Tracing

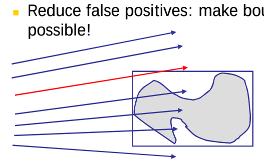
- Data Structures
  - Goal: reduce number of intersection tests per ray
  - Lots of different approaches:
    - (Hierarchical) bounding volumes
    - Hierarchical space subdivision
      - Octree, k-D tree, BSP tree

Ray-Tracing: Practicalities

- Generation of rays
- Intersection of rays with geometric primitives
- **Geometric transformations**
- Lighting and shading
- **Speed:** Reducing number of intersection tests
  - E.g. use BSP trees or other types of space partitioning

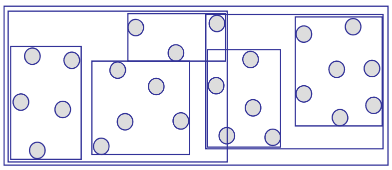
Bounding Volumes

- Idea:
  - Rather than test every ray against a potentially very complex object (e.g. triangle mesh), do a quick _*conservative*_ test first which eliminates most rays
    - Surround complex object by simple, easy to test geometry (typically sphere or axis-aligned box)
      - Reduce false positives: make bounding volume as tight as possible!



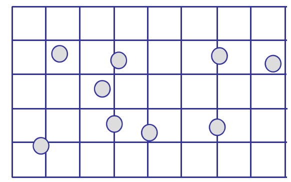
Hierarchical Bounding Volumes

- Extension of previous idea:
  - Use bounding volumes for groups of objects



Creating a Regular Grid

- Steps:
  - Find bounding box of scene
  - Choose grid resolution in x, y, z
  - Insert objects
  - Objects that overlap multiple cells get referenced by all cells they overlap

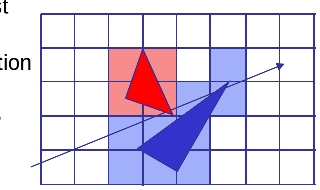


Spatial Subdivision Data Structures

- Bounding Volumes:
  - Find simple object completely enclosing complicated objects
    - Boxes, spheres
  - Hierarchically combine into larger bounding volumes
- Spatial subdivision data structure:
  - Partition the whole space into cells
    - Grids, octrees, (BSP trees)
  - Simplifies and accelerates traversal
  - Performance less dependent on order in which objects are inserted

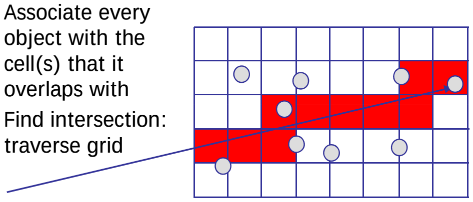
Grid Traversal

- Start at ray origin
- While no intersection found
  - Go to next grid cell along ray
  - Compute intersection of ray with all objects in the cell
  - Determine closest such intersection
  - Check if intersection is inside the cell
    - If so, terminate search

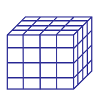


Regular Grid

- Subdivide space into rectangular grid:
  - Associate every object with the cell(s) that it overlaps with
  - Find intersection: traverse grid



In 3D: regular grid of cubes (voxels):



Traversal

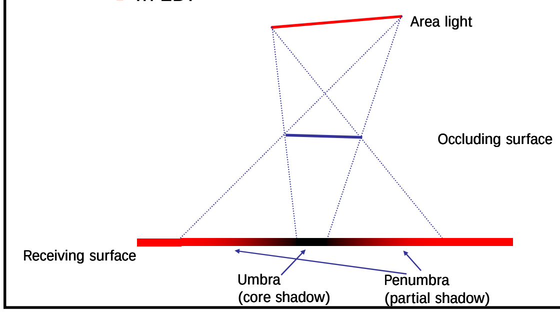
- Note:
  - This algorithm calls for computing the intersection points multiple times (once per grid cell)
  - In practice: store intersections for a (ray, object) pair once computed, reuse for future cells

Regular Grid Discussion

- Advantages?
  - Easy to construct
  - Easy to traverse
- Disadvantages?
  - May be only sparsely filled
  - Geometry may still be clumped

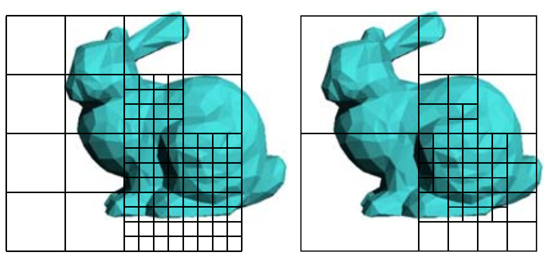
Area Light Sources

- Area lights produce soft shadows:
  - In 2D:



Adaptive Grids

- Subdivide until each cell contains no more than *n* elements, or maximum depth *d* is reached



Nested Grids | Octree/(Quadtree)

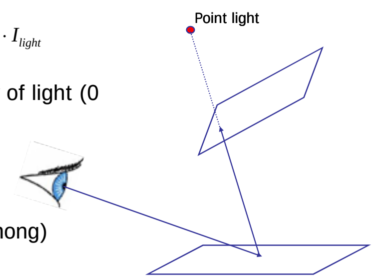
- This slide is curtsey of Fredo Durand at MIT

Area Light Sources

- Point lights:
  - Only one light direction:

$$I_{reflected} = \rho \cdot V \cdot I_{light}$$

  - V is visibility of light (0 or 1)
  - ρ is lighting model (e.g. diffuse or Phong)



Soft Shadows: Area Light Sources

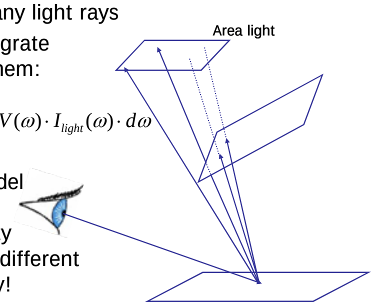
- So far:
  - All lights were either point-shaped or directional
    - Both for ray-tracing and the rendering pipeline
  - Thus, at every point, we only need to compute lighting formula and shadowing for ONE direction per light
- In reality:
  - All lights have a finite area
  - Instead of just dealing with one direction, we now have to integrate over all directions that go to the light source

Are Light Sources

- Area Lights:
  - Infinitely many light rays
  - Need to integrate over all of them:

$$I_{reflected} = \int_{\substack{light\\directions}} \rho(\omega) \cdot V(\omega) \cdot I_{light}(\omega) \cdot d\omega$$

  - Lighting model visibility and light intensity can now be different for every ray!



Integrating over Light Source

- Rewrite the integration
  - Instead of integrating over directions

$$I_{reflected} = \int_{\substack{light\\directions}} \rho(\omega) \cdot V(\omega) \cdot I_{light}(\omega) \cdot d\omega$$

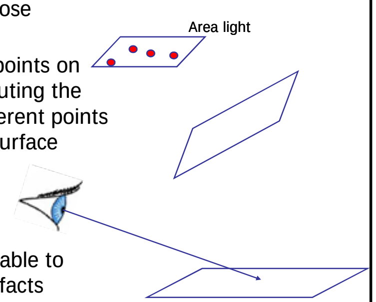
  integrate over points on the light source

$$I_{reflected}(q) = \int_{s,t} \frac{\rho(p-q) \cdot V(p-q)}{|p-q|^2} \cdot I_{light}(p) \cdot ds \cdot dt$$

  where: q point on reflecting surface & p= F(s,t) point on the area light
    - We are integrating over p
    - Denominator: quadratic falloff!

Monte Carlo Integration

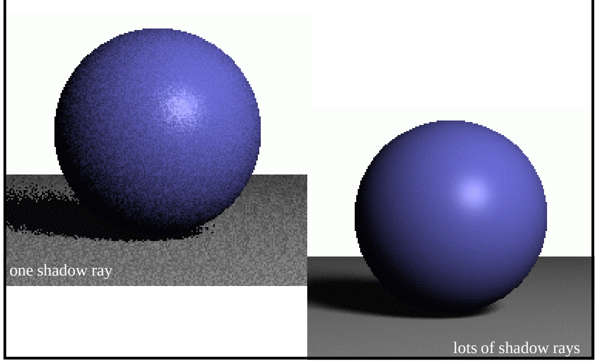
- Better:
  - Randomly choose the points
  - Use different points on light for computing the lighting in different points on reflecting surface
  - This produces random noise
  - Visually preferable to structured artifacts



Integration

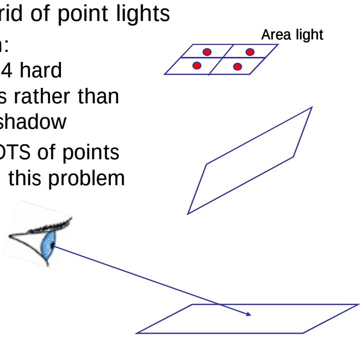
- Problem:
  - Except for the simplest of scenes, either integral is not solvable analytically!
  - This is mostly due to the visibility term, which could be arbitrarily complex depending on the scene
- So:
  - Use numerical integration
  - Effectively: approximate the light with a whole number of point lights

Monte Carlo Integration



Numerical Integration

- Regular grid of point lights
  - Problem: will see 4 hard shadows rather than as soft shadow
  - Need LOTS of points to avoid this problem



Monte Carlo Integration

- Formally:
  - Approximate integral with finite sum

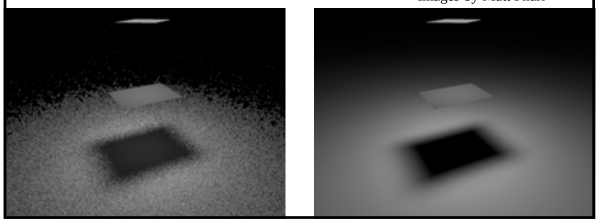
$$I_{reflected}(q) = \int_{s,t} \frac{\rho(p-q) \cdot V(p-q)}{|p-q|^2} \cdot I_{light}(p) \cdot ds \cdot dt$$
$$\approx \frac{A}{N} \sum_{i=1}^{N} \frac{\rho(p_i - q) \cdot V(p_i - q)}{|p_i - q|^2} \cdot I_{light}(p_i)$$

  where
    - The pi are randomly chosen on the light source
      - With equal probability!
    - A is the total area of the light
    - N is the number of samples (rays)

Sampling

- Sample directions vs. sample light source
  - Most directions do not correspond to points on the light source
    - Thus, variance will be higher than sampling light directly

Images by Matt Pharr



Monte Carlo Integration

- Note:
  - This approach of approximating lighting integrals with sums over randomly chosen points is much more flexible than this!
  - In particular, it can be used for global illumination
    - Light bouncing off multiple surfaces before hitting the eye