Ray-Tracing

CPSC 314

Electromagnetic Spectrum
Blackbody Radiation

Black body

- Dark material, so that reflection can be neglected
- Spectrum of emitted light changes with temperature
  - *This is the origin of the term “color temperature”*
    - E.g. when setting a white point for your monitor
  - Cold: mostly infrared
  - Hot: redish
  - Very hot: bluish
- Demo:
  
  http://www.mhhe.com/physsci/astronomy/applets/Blackbody/frame.html

Sunlight Spectrum

![Sunlight Spectrum](image)
Line Spectrum

Examples:
- Ionized gases
- Lasers
- Some fluorescent lamps

Physiology of Vision

The retina
- Rods
  - B/w, edges
- Cones
  - Color!
Physiology of Vision

Center of retina is densely packed region called the fovea.

• Cones much denser here than the periphery

Perceptual vs. Colorimetric Terms

<table>
<thead>
<tr>
<th>Perceptual</th>
<th>Colorimetric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hue</td>
<td>Dominant wavelength</td>
</tr>
<tr>
<td>Saturation</td>
<td>Excitation purity</td>
</tr>
<tr>
<td>Lightness</td>
<td>Luminance</td>
</tr>
</tbody>
</table>
  - Reflecting objects |          |
| Brightness          | Luminance             |
  - Light sources     |          |

© Wolfgang Heidrich
Color Matching Experiments

Performed in the 1930s

Idea: perceptually based measurement

- shine given wavelength ($\lambda$) on a screen
- User must control three pure lights producing three other wavelengths (say R=700nm, G=546nm, and B=436nm)
- Adjust intensity of RGB until colors are identical

Results

- It was found that any color $S(\lambda)$ could be matched with three suitable primaries $A(\lambda)$, $B(\lambda)$, and $C(\lambda)$
  - Used monochromatic light at 438, 546, and 700 nanometers
- Also found the space is linear, i.e. if $R(\lambda) \equiv S(\lambda)$
  
  then

\[
R(\lambda) + M(\lambda) \equiv S(\lambda) + M(\lambda)
\]

and

\[
k \cdot R(\lambda) \equiv k \cdot S(\lambda)
\]
Negative Lobes

Actually:

- Exact target match possible sometimes requires “negative light”
- Some red had to be added to target color to permit exact match using “knobs” on RGB intensity output
- Equivalently theoretically to removing red from RGB output
- Figure shows that red primary must remove some cyan for perfect match
- CRT phosphors cannot remove cyan, so 500 nm cannot be generated

CIE Color Space

- CIE defined three “imaginary” lights X, Y, and Z, any wavelength \( \lambda \) can be matched perceptually by positive combinations

Note that:

\[
\begin{align*}
X & \sim R \\
Y & \sim G \\
Z & \sim B
\end{align*}
\]
**Measured vs. CIE Color Spaces**

**Measured basis**
- Monochromatic lights
- Physical observations
- Negative lobes

**Transformed basis**
- "imaginary" lights
- All positive, unit area
- $Y$ is luminance, no hue
- $X,Z$ no luminance

**Facts about the CIE “Horseshoe” Diagram**
- All visible colors lie inside the horseshoe
  - Result from color matching experiments
- Spectral (monochromatic) colors lie around the border
  - The straight line between blue and red contains the purple tones
- Colors combine linearly (i.e. along lines), since the $xy$-plane is the projection of a linear space
Facts about the CIE “Horseshoe” Diagram (cont.)

A point C can be chosen as a white point corresponding to an illuminant

- Usually this point is of the curve swept out by the black body radiation spectra for different temperatures
- Relative to C, two colors are called complementary if they are located along a line segment through C, but on opposite sides (i.e., C is an affine combination of the two colors)
- The dominant wavelength of the color is found by extending the line from C through the color to the edge of the diagram
- Some colors (i.e., purples) do not have a dominant wavelength, but their complementary color does.

CIE Diagram

- Blackbody curve
- Illumination:
  - Candle 2000K
  - Light bulb 3000K (A)
  - Sunset/sunrise 3200K
  - Day light 6500K (D)
  - Overcast day 7000K
  - Lightning >20000K
Color Interpolation, Dominant & Opponent Wavelength

RGB Color Space (Color Cube)

Define colors with \((r, g, b)\) amounts of red, green, and blue
- Used by OpenGL
- Hardware-centric
- Describes the colors that can be generated with specific RGB light sources

*RGB color cube sits within CIE color space*
- Subset of perceivable colors
- Scaled, rotated, sheared cube
Additive vs. Subtractive Colors

**Additive: light**

- Monitors, LCDs
- RGB model

\[
\begin{bmatrix}
C \\
M \\
Y
\end{bmatrix} = \begin{bmatrix}
1 \\
1 \\
1
\end{bmatrix} \begin{bmatrix}
R \\
G \\
B
\end{bmatrix}
\]

**Subtractive: pigment**

- Printers
- CMY(K) model

---

Ray-Tracing

*CPSC 314*
Overview

So far
- Rendering Pipeline
- Rendering algorithms using the Rendering Pipeline

Today
- Ray-Tracing
  - Simple algorithm for software rendering
  - Extremely flexible (new effects can easily be incorporated)

Ray-Tracing

**Basic Algorithm (Whithead):**

```plaintext
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
    }
}
```
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-Tracing – Generation of Rays

**Camera Coordinate System**
- Origin: \( C \) (camera position)
- Viewing direction: \( \mathbf{v} \)
- Up vector: \( \mathbf{u} \)
- \( \mathbf{x} \) 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 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: $O = C + d \cdot \mathbf{v} + l \cdot \mathbf{u}$
- 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-Tracing – Generation of Rays

Ray in 3D Space:

\[
R_{i,j}(t) = C + t \cdot (P_{i,j} - C) = C + t \cdot \mathbf{v}_{i,j}
\]

where $t = 0...\infty$
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 Intersections

Task:
- Given an object \( o \), find ray parameter \( t \), such that \( \mathbf{R}_{i_d}(t) \) is a point on the object
  - Such a value for \( t \) may not exist
- Intersection test depends on geometric primitive
Ray Intersections

Spheres at origin:
- Implicit function:
  \[ S(x, y, z) : x^2 + y^2 + z^2 = r^2 \]
- Ray equation:
  \[
  R_{i,j}(t) = C + t \cdot v_{i,j} = \begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix} + t \cdot \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} c_x + t \cdot v_x \\ c_y + t \cdot v_y \\ c_z + t \cdot v_z \end{pmatrix}
  \]

To determine intersection:
- Insert ray \( R_{i,j}(t) \) 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!)
  - 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 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

**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-Tracing – Geometric Transformations

**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*
    - 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 – 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
Ray-Tracing
Lighting and Shading

Local Effects:
- Local Lighting
  - Any reflection model possible
  - Have to talk about light sources, normals…
- Texture mapping
  - Color textures
  - Bump maps
  - Environment maps
  - Shadow maps

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 $n(x,y,z)$ can be easily computed at every intersection point using the gradient

$$n(x,y,z) = \left( \begin{array}{c} \frac{\partial F(x,y,z)}{\partial x} \\ \frac{\partial F(x,y,z)}{\partial y} \\ \frac{\partial F(x,y,z)}{\partial z} \end{array} \right)$$

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

$$n(x,y,z) = \left( \begin{array}{c} 2x \\ 2y \\ 2z \end{array} \right) \quad \text{Needs to be normalized!}$$

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 Texture Mapping

**Approach:**
- Works in principle like in rendering pipeline
  - Given $s$, $t$ parameter values, perform texture lookup
  - Magnification, minification just as discussed
- Problem: how to get $s$, $t$
  - Implicit surfaces often don’t have parameterization
  - For special cases (spheres, other conic sections), can use parametric representation
  - Triangles/meshes: use interpolation from vertices

Ray-Tracing Lighting and Shading

**Global Effects**
- Shadows
- Reflections/refractions
Ray-Tracing
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

Ray-Tracing
Reflections/Refractions

Approach:
• Send rays out in reflected and refracted direction to gather incoming light
• That light is multiplied by local surface color and Fresnel term, and added to result of local shading
Recursive Ray Tracing

**Ray tracing can handle**
- Reflection (chrome)
- 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

Recursive Ray-Tracing

[Image of ray tracing diagram showing light source, eye, image plane, reflected ray, refracted ray, and shadow rays.]
Recursive Ray-Tracing Algorithm

\[ \text{RayTrace}(r, \text{scene}) \]
\[ \text{obj := FirstIntersection}(r, \text{scene}) \]
\[ \text{if (no obj) return BackgroundColor; else begin} \]
\[ \text{if ( Reflect(obj) ) then} \]
\[ \quad \text{reflect\_color := RayTrace(ReflectRay}(r, \text{obj}); \]
\[ \text{else} \]
\[ \quad \text{reflect\_color := Black;} \]
\[ \text{if ( Transparent(obj) ) then} \]
\[ \quad \text{refract\_color := RayTrace(RefractRay}(r, \text{obj}); \]
\[ \text{else} \]
\[ \quad \text{refract\_color := Black;} \]
\[ \text{return Shade(reflect\_color, refract\_color, obj);} \]
\[ \text{end;} \]

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
Reflection

**Mirror effects**
- Perfect specular reflection

Refrraction

*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 \( c_1 \), \( c_2 \)
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.

Ray-Tracing Example Images
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-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 Tracing

Data Structures

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

Bounding Volumes

Idea:

- Rather than testing every ray against a potentially very complex object (e.g. triangle mesh), do a quick conservative test first which eliminates most of the rays
  - Surround complex object by very simple, easy to test geometry (typically sphere or axis-aligned box)
  - Want to make bounding volume as tight as possible!
Hierarchical Bounding Volumes

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

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, oct-trees, (BSP trees)
• Simplifies and accelerates traversal
• Performance less dependent on order in which objects are inserted
**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):

---

**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
Grid Traversal

**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
  - Find closest intersection
  - Check if that intersection is inside the cell
  - If so, terminate search

**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

Adaptive Grids

- Subdivide until each cell contains no more than \( n \) elements, or maximum depth \( d \) is reached
Primitives in an Adaptive Grid

- Can live at intermediate levels, or be pushed to lowest level of grid

Octree/(Quadtree)

Adaptive Grid Discussion

**Advantages**
- Grid complexity matches geometric density

**Disadvantages**
- More expensive to traverse than regular grid
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 light direction

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
Area Light Sources

**Area lights produce soft shadows:**

![Diagram of area light sources with umbra and penumbra](image)

**Point lights:**
- Only one light direction:

\[ I_{\text{reflected}} = \rho \cdot V \cdot I_{\text{light}} \]
- \( V \) is visibility of light (0 or 1)
- \( \rho \) is lighting model (e.g. diffuse or Phong)
Are Light Sources

Area Lights:
• Infinitely many light rays
• Need to integrate over all of them:
  \[ I_{\text{reflected}} = \int \rho(\omega) \cdot V(\omega) \cdot I_{\text{light}}(\omega) \cdot d\omega \]
• 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
  \[ I_{\text{reflected}} = \int \rho(\omega) \cdot V(\omega) \cdot I_{\text{light}}(\omega) \cdot d\omega \]
  we can integrate over points on the light source
  \[ I_{\text{reflected}}(q) = \int_{s,t} \frac{\rho(p - q) \cdot V(p - q)}{|p - q|^2} \cdot I_{\text{light}}(p) \cdot ds \cdot dt \]
  where q: point on reflecting surface, p = F(s,t) is a point on the area light
  − We are integrating over p
  − Denominator: quadratic falloff!
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

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

**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

---

Monte Carlo Integration

- one shadow ray
- lots of shadow rays
Monte Carlo Integration

Formally:

- Approximate integral with finite sum

\[
I_{\text{reflected}}(q) = \int_{p,q} \rho(p-q) \cdot V(p-q) \cdot I_{\text{light}}(p) \cdot ds \cdot dt
\]

\[
\approx A \sum_{i=1}^{N} \rho(p_i - q) \cdot V(p_i - q) \cdot I_{\text{light}}(p_i)
\]

where

- The \( p_i \) 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)

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
- More on this next lecture!
Coming Up...

**Monday:**
- Global illumination / curves and surfaces
- Last topics to be on the final

**Wednesday:**
- Current research topics in graphics
- Advanced graphics courses at UBC
- Lab tour

**Friday:**
- Final