Ray-Tracing

**Overview**

**So far**
- Real-time/HW rendering w/ Rendering Pipeline
- Rendering algorithms using the Rendering Pipeline

**Today**
- Ray-Tracing
  - Simple algorithm for software rendering
    - Usually offline (e.g., movies etc.)
  - Extremely flexible (new effects can easily be incorporated)

---

**Ray-Tracing**

**Basic Algorithm (Whithead):**

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: $v$
- Up vector: $u$
- $x$ direction: $x = v \times u$

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

---

**Ray-Tracing**

**Other parameters:**
- Distance of Camera from image plane, $d$
- Image resolution (in pixels): $w$, $h$
- Left, right, top, bottom boundaries in image plane: $i$, $r$, $t$, $b$

**Then:**
- Lower left corner of image: $O = C + d \cdot v + i \cdot x + j \cdot b \cdot u$
- Pixel at position $i, j$ ($i=0..w-1$, $j=0..h-1$):
  $P_{ij} = O + \frac{r - l}{w - 1} \cdot x - j \cdot \frac{t - b}{h - 1} \cdot u$
  $= O + i \cdot \Delta x \cdot x - j \cdot \Delta y \cdot y$
Ray-Tracing – Generation of Rays

**Ray in 3D Space:**

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

where \( t \geq 0 \ldots \infty \)

Ray Intersections

**Task:**

- Given an object \( o \), find ray parameter \( t_i \), such that \( R_{i}(t) \) is a point on the object
  - Such a value for \( t_i \) may not exist
- Intersection test depends on geometric primitive

Ray Intersections

**To determine intersection:**

- Insert ray \( R_{i}(t) \) into \( S(x, y, z) \):
  \[
  (c_i + t \cdot v_i)^2 + (c_j + t \cdot v_j)^2 + (c_k + t \cdot v_k)^2 = r^2
  \]
- Solve for \( t \) (find roots)
  - Simple quadratic equation

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

**Spheres at origin:**

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

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

© Wolfgang Heidrich
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**
- 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
  \[
  \mathbf{n}(x,y,z) = \begin{pmatrix}
  \frac{\partial F(x,y,z)}{\partial x} \\
  \frac{\partial F(x,y,z)}{\partial y} \\
  \frac{\partial F(x,y,z)}{\partial z}
  \end{pmatrix}
  \]
- Example: \( F(x,y,z) = x^2 + y^2 + z^2 - r^2 \)
  \[
  \mathbf{n}(x,y,z) = \begin{pmatrix}
  2x \\
  2y \\
  2z
  \end{pmatrix}
  \]

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:**
- Sand 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 Algorithm

```cpp
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

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 \( c_1, c_2 \)

**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**

- Image 1: Ray tracing through a transparent object
- Image 2: Shading and lighting effects

---

**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

### Spatial Subdivision Data Structures

**BoundingVolumes:**
- 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):

### 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
**Traversals**

*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

**Adaptive Grid Discussion**

*Advantages*
- Grid complexity matches geometric density

*Disadvantages*
- More expensive to traverse than regular grid

**Coming Up...**

*Thursday:*
- Global illumination

*Tuesday:*
- Color