Ray-Tracing

Overview

So far
- projective rendering (hardware)
- radiosity

Ray-Tracing
- simple algorithm for software rendering
- extremely flexible
- well suited to transparent and specular objects
- global illumination (*)
- partly physics-based: geometric optics

```
raytrace( ray ) {
    find closest intersection
    cast shadow ray, calculate colour_local
    colour_reflect = raytrace( reflected_ray )
    colour_refract = raytrace( refracted_ray )
    colour = k1*colour_local + k2*colour_reflect + k3*colour_refract
    return( colour )
}
```

“raycasting” : only cast first ray from eye
Ray-Tracing

Ray Termination Criteria:
- ray hits a diffuse surface
- ray exits the scene
- threshold on contrib. towards final pixel colour
- maximum recursion depth

Camera Coordinate System
- Origin: C (camera position)
- Viewing direction: v
- Up vector: u
- x direction: x = v × u

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

Ray in 3D Space:
\[ R_{i,j}(t) = C + t \cdot (P_{i,j} - C) = C + t \cdot v_{i,j} \]
where \( t = 0 \ldots \infty \)
Ray Intersections

Task:
- Given an object o, find ray parameter \( t \), such that \( R_{i,j}(t) \) 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:
  \[ R_{i,j}(t) = C + t \cdot v_{i,j} = \begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix} + t \cdot \begin{pmatrix} v_{x} \\ v_{y} \\ v_{z} \end{pmatrix} = \begin{pmatrix} c_x + t \cdot v_{x} \\ c_y + t \cdot v_{y} \\ c_z + t \cdot v_{z} \end{pmatrix} \]

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)
    - Root-finding more difficult
    - Resort 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 is on the right side of every boundary edge
    - Similar to computation of outcodes in line clipping

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.*