Chapter 11

Ray-Tracing

Midterm 2

Average XXX%

In 2012 average was 74%
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/Radiosity
- Ray-tracing
  - Usually offline (e.g. movies etc.)
    - research on making real-time
  - Flexible - can incorporate lots of phenomena

Ray-Tracing Algorithm
Reflection

- Mirror effects
  - Perfect specular reflection

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 \( c_1, c_2 \)
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;
```

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
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

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 With Shadows

Eye → Image Plane

Light Source → Reflected Ray → Geometric Content

Refracted Ray

Ray images

Replaces Rendering Pipeline!!!

Geometry Processing

Geometric Content → Model/View Transform. → Lighting → Perspective Transform. → Clipping

Scan Conversion → Texturing → Depth Test → Blending → Frame-buffer

Rasterization → Fragment Processing
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: Generation of Rays

- Camera Coordinate System
  - Origin: C (camera position)
  - Viewing direction: \( \mathbf{w} \)
  - Up vector: \( \mathbf{v} \)
  - \( \mathbf{u} \) direction: \( \mathbf{u} = \mathbf{w} \times \mathbf{v} \)
- 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): $x, h$
  - Left, right, top, bottom boundaries in image plane: $l, r, t, b$

- Then:
  - Lower left corner of image: $O = C + d \cdot w + l \cdot u + b \cdot v$
  - Pixel at position $i, j$ ($i=0..x-1, j=0..h-1$):

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

$$\quad = O + i \cdot \Delta u \cdot u - j \cdot \Delta v \cdot v$$

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 $t = 0, \ldots, \infty$
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-Object Intersections

- Kernel of ray-tracing ⇒ 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 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

- 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

- 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: 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: 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
Light sources:
- For the moment: point and directional lights
- 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

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
  \[
  \mathbf{n}(x,y,z) = \begin{pmatrix}
  \frac{\partial F(x,y,z)}{\partial x} \\
  \frac{\partial F(x,y,z)}{\partial y} \\
  \frac{\partial F(x,y,z)}{\partial z}
  \end{pmatrix}
  \]
- Example:
  \[
  F(x,y,z) = x^2 + y^2 + z^2 - r^2
  \]
  \[
  \mathbf{n}(x,y,z) = \begin{pmatrix}
  2x \\
  2y \\
  2z
  \end{pmatrix}
  \]
  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:
    - Have to compute Barycentric coordinates for every intersection point (e.g. plane equation for triangles)

Ray-Tracing: Practicalities

- Generation of rays
- Intersection of rays with geometric primitives
- Geometric transformations
- Lighting and shading
  - **Speed:** Reducing number of intersection tests
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

- 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
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
BSP Trees: Idea

- For any plane (3D) objects on the same side of plane as viewer CANNOT be occluded by objects on other side => intersect closer side first/if don’t intersect plane can’t intersect other side

- Idea:
  - Recursively split space by planes
  - Traverse resulting tree to establish rendering/intersection order
    - Test eye location w.r.t. each plane

Creating BSP Trees: Objects
Creating BSP Trees: Objects
Splitting Objects

- No bunnies were harmed in previous example
- But what if a splitting plane passes through an object?
  - Split the object; give half to each node

Traversing BSP Trees

- Tree creation independent of viewpoint
  - Preprocessing step
- Tree traversal uses viewpoint
  - Runtime, happens for many different viewpoints
BSP Trees: Viewpoint A

- Decide independently at each tree vertex
- Not just left or right child!
BSP Trees: Viewpoint A

1
2

F
N
N
F
F
1
2

F
N
N
F
F
N
N
F
1
2
BSP Trees: Viewpoint A

- Each plane divides world into near and far
  - For given viewpoint, decide which side is near and which is far
    - Check which side of plane viewpoint is on independently for each tree vertex
    - Tree traversal differs depending on viewpoint!
  - Recursive algorithm
    - Recurse on far side
    - Draw object
    - Recurse on near side

Traversing BSP Trees
renderBSP(BSPtree *T)
    BSPtree *near, *far;
    if (eye on left side of T->plane)
        near = T->left; far = T->right;
    else
        near = T->right; far = T->left;
    renderBSP(far);
    if (T is a leaf node)
        renderObject(T)
    renderBSP(near);
BSP Trees: Viewpoint B

- Split along the plane defined by any polygon from scene
- Classify all polygons into positive or negative half-space of the plane
  - If a polygon intersects plane, split polygon into two and classify them both
- Recurse down the negative half-space
- Recurse down the positive half-space

BSP Tree Traversal: Polygons
BSP Demo

- Useful demo:
  - http://symbolcraft.com/graphics/bsp

Summary: BSP Trees

- Pros:
  - Simple, elegant scheme
  - Correct version of painter’s algorithm back-to-front rendering approach
  - Still very popular for video games

- Cons:
  - Slow(ish) to construct tree: $O(n \log n)$ to split, sort
  - Splitting increases polygon count: $O(n^2)$ worst-case
  - Computationally intense preprocessing stage restricts algorithm to static scenes
**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

---

**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
Area Light Sources

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

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

\[
I_{\text{reflected}} = \int \rho(\omega) \cdot V(\omega) \cdot I_{\text{light}}(\omega) \cdot d\omega
\]

  - Integrate over points on the light source

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

- \(q\) point on reflecting surface
- \(p = F(s,t)\) point on the area light
- We are integrating over \(p\)
**Integration**

- **Problem:**
  - Except for basic case *not solvable analytically!*
    - Largely due to the visibility term
  - So:
    - Use numerical integration = approximate light with lots of point lights

**Numerical Integration**

- Regular grid of point lights
  - Problem: Too regular see 4 hard shadows
  - 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
- Produces random noise
  - Visually preferable to structured artifacts !!!

Monte Carlo Integration

one shadow ray

lots of shadow 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