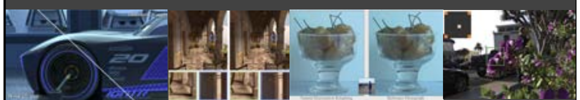
Sampling and Reconstruction of Visual Appearance

CSE 274 [Fall 2018], Special Lecture Ray Tracing

Ravi Ramamoorthi

http://www.cs.ucsd.edu/~ravir



Motivation

- Ray Tracing is a core aspect of both offline and real-time rendering today
- Basic topic which I cover in CSE 167
- But not everyone does, this class also covers 168 (ray tracing is a prelude to key path tracing algorithm)
- Background for some (most?) of you, but critical topic for those who haven't seen it before, go over it fast
- Important if you want to do the optional path tracer assignment (includes a ray tracing assignment)
- http://www.edx.org/course/computer-graphics

Effects needed for Realism

- (Soft) Shadows
- Reflections (Mirrors and Glossy)
- Transparency (Water, Glass)
- Interreflections (Color Bleeding)
- Complex Illumination (Natural, Area Light)
- Realistic Materials (Velvet, Paints, Glass)
- And many more

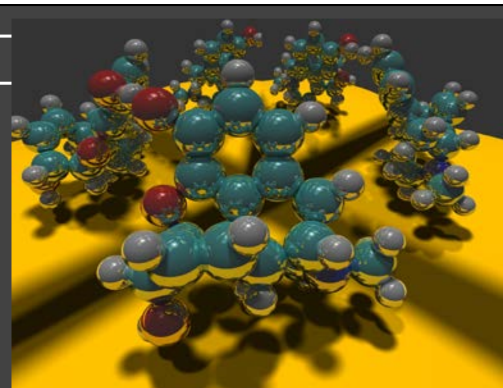


Image courtesy Paul Heckbert 1983

Ray Tracing

- Different Approach to Image Synthesis as compared to Hardware pipeline (OpenGL)
- Pixel by Pixel instead of Object by Object
- Easy to compute shadows/transparency/etc

Outline

- *History*
- Basic Ray Casting (instead of rasterization)
  - Comparison to hardware scan conversion
- Ray-Surface Intersection
- Shadows / Reflections (core algorithm)
- Optimizations
- Current Research

Ray Tracing: History

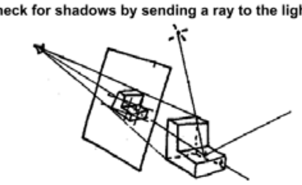
- Appel 68
- Whitted 80 [recursive ray tracing]
  - Landmark in computer graphics
- Lots of work on various geometric primitives
- Lots of work on accelerations
- Current Research
  - Real-Time raytracing (historically, slow technique)
  - Ray tracing architecture

Ray Tracing History

Ray Tracing in Computer Graphics

Appel 1968 - Ray casting

1. Generate an image by sending one ray per pixel
2. Check for shadows by sending a ray to the light



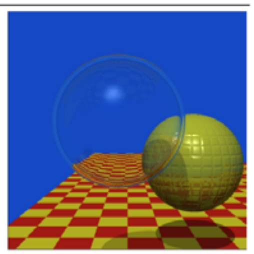
CS348B Lecture 2 Pat Hanrahan, Spring 2009

Ray Tracing History

Ray Tracing in Computer Graphics

"An improved Illumination model for shaded display," T. Whitted, CACM 1980

Resolution:
512 x 512
Time:
VAX 11/780 (1979)
74 min.
PC (2006)
6 sec.



Spheres and Checkerboard, T. Whitted, 1979

CS348B Lecture 2 Pat Hanrahan, Spring 2009

Outline

- History
- *Basic Ray Casting (instead of rasterization)*
  - Comparison to hardware scan conversion
- Ray-Surface Intersection
- Shadows / Reflections (core algorithm)
- Optimizations
- Current Research

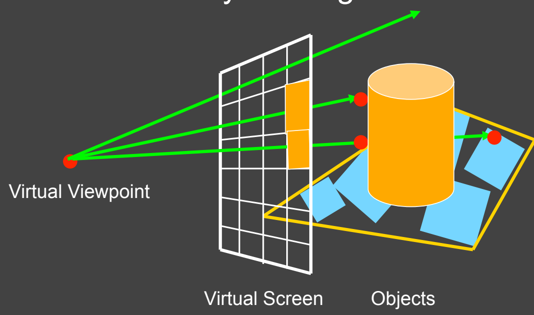
Outline in Code

```
Image Raytrace (Camera cam, Scene scene, int width, int height)
{
   Image image = new Image (width, height) ;
   for (int i = 0 ; i < height ; i++)
        for (int j = 0 ; j < width ; j++) {
                Ray ray = RayThruPixel (cam, i, j) ;
                Intersection hit = Intersect (ray, scene) ;
                image[i][j] = FindColor (hit) ;
                }
   return image ;
}
```

Ray Casting

Produce same images as with OpenGL
- Visibility per pixel instead of Z-buffer
- Find nearest object by shooting rays into scene
- Shade it as in standard OpenGL

Ray Casting



Multiple intersections: Use closest one (as does OpenGL)

Comparison to hardware scan-line

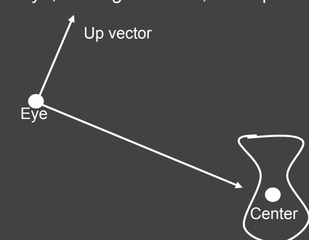
- Per-pixel evaluation, per-pixel rays (not scan-convert each object). On face of it, costly
- But good for walkthroughs of extremely large models (amortize preprocessing, low complexity)
- More complex shading, lighting effects possible

Finding Ray Direction

- Goal is to find ray direction for given pixel i and j
- Many ways to approach problem
  - Objects in world coord, find dirn of each ray (we do this)
  - Camera in canonical frame, transform objects (OpenGL)
- Basic idea
  - Ray has origin (camera center) and direction
  - Find direction given camera params and i and j
- Camera params as in gluLookAt
  - Lookfrom[3], LookAt[3], up[3], fov

Similar to gluLookAt derivation

- gluLookAt(eyex, eyey, eyez, centerx, centery, centerz, upx, upy, upz)
- Camera at eye, looking at center, with up direction being up



From OpenGL lecture on deriving gluLookAt (see edX MOOC)

Constructing a coordinate frame?

We want to associate **w** with **a**, and **v** with **b**

- But **a** and **b** are neither orthogonal nor unit norm
- And we also need to find **u**

$$w = \frac{a}{\|a\|}$$

$$u = \frac{b \times w}{\|b \times w\|}$$

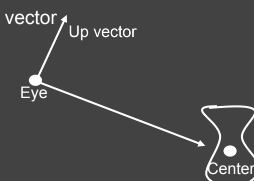
$$v = w \times u$$

From basic math lecture - Vectors: Orthonormal Basis Frames

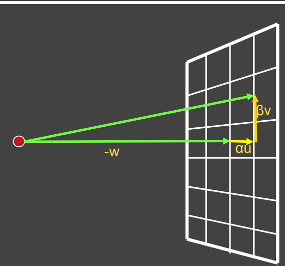
Camera coordinate frame

$$w = \frac{a}{\|a\|} \qquad u = \frac{b \times w}{\|b \times w\|} \qquad v = w \times u$$

- We want to position camera at origin, looking down –Z dirn
- Hence, vector **a** is given by **eye** – **center**
- The vector **b** is simply the **up** vector



Canonical viewing geometry



$$ray = eye + \frac{\alpha u + \beta v - w}{|\alpha u + \beta v - w|}$$

$$\alpha = \tan\left(\frac{fovx}{2}\right) \times \left(\frac{j - (width/2)}{width/2}\right) \qquad \beta = \tan\left(\frac{fovy}{2}\right) \times \left(\frac{(height/2) - i}{height/2}\right)$$

Outline

- History
- Basic Ray Casting (instead of rasterization)
  - Comparison to hardware scan conversion
- *Ray-Surface Intersection*
- Shadows / Reflections (core algorithm)
- Optimizations
- Current Research

Ray/Object Intersections

- Heart of Ray Tracer
  - One of the main initial research areas
  - Optimized routines for wide variety of primitives
- Various types of info
  - Shadow rays: Intersection/No Intersection
  - Primary rays: Point of intersection, material, normals
  - Texture coordinates
- Work out examples
  - Triangle, sphere, polygon, general implicit surface

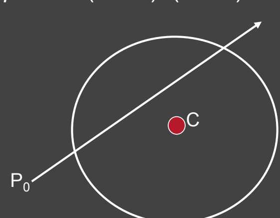
Outline in Code

```
Image Raytrace (Camera cam, Scene scene, int width, int height)
{
   Image image = new Image (width, height) ;
   for (int i = 0 ; i < height ; i++)
        for (int j = 0 ; j < width ; j++) {
                Ray ray = RayThruPixel (cam, i, j) ;
                Intersection hit = Intersect (ray, scene) ;
                image[i][j] = FindColor (hit) ;
                }
   return image ;
}
```

Ray-Sphere Intersection

$$ray \equiv \vec{P} = \vec{P}_0 + \vec{P}_1 t$$

$$sphere \equiv (\vec{P} - \vec{C}) \cdot (\vec{P} - \vec{C}) - r^2 = 0$$



Ray-Sphere Intersection

$$ray \equiv \vec{P} = \vec{P}_0 + \vec{P}_1 t$$

$$sphere \equiv (\vec{P} - \vec{C}) \cdot (\vec{P} - \vec{C}) - r^2 = 0$$

Substitute

$$ray \equiv \vec{P} = \vec{P}_0 + \vec{P}_1 t$$

$$sphere \equiv (\vec{P}_0 + \vec{P}_1 t - \vec{C}) \cdot (\vec{P}_0 + \vec{P}_1 t - \vec{C}) - r^2 = 0$$

Simplify

$$t^2(\vec{P}_1 \cdot \vec{P}_1) + 2t\, \vec{P}_1 \cdot (\vec{P}_0 - \vec{C}) + (\vec{P}_0 - \vec{C}) \cdot (\vec{P}_0 - \vec{C}) - r^2 = 0$$

Ray-Sphere Intersection

$$t^2(\vec{P}_1\cdot\vec{P}_1)+2t\,\vec{P}_1\cdot(\vec{P}_0-\vec{C})+(\vec{P}_0-\vec{C})\cdot(\vec{P}_0-\vec{C})-r^2=0$$

Solve quadratic equations for t

- 2 real positive roots: pick smaller root
- Both roots same: tangent to sphere
- One positive, one negative root: ray origin inside sphere (pick + root)
- Complex roots: no intersection (check discriminant of equation first)

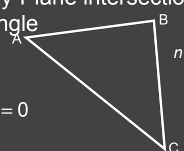
Ray-Sphere Intersection

- Intersection point: $ray \equiv \vec{P}=\vec{P}_0+\vec{P}_1 t$
- Normal (for sphere, this is same as coordinates in sphere frame of reference, useful other tasks)

$$normal = \frac{\vec{P}-\vec{C}}{\left|\vec{P}-\vec{C}\right|}$$

Ray-Triangle Intersection

- One approach: Ray-Plane intersection, then check if inside triangle
- Plane equation:

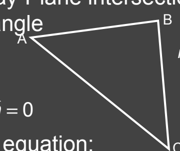


$$n=\frac{(C-A)\times(B-A)}{\left|(C-A)\times(B-A)\right|}$$

$$plane \equiv \vec{P}\cdot\vec{n}-\vec{A}\cdot\vec{n}=0$$

Ray-Triangle Intersection

- One approach: Ray-Plane intersection, then check if inside triangle
- Plane equation:



$$n=\frac{(C-A)\times(B-A)}{\left|(C-A)\times(B-A)\right|}$$

$$plane \equiv \vec{P}\cdot\vec{n}-\vec{A}\cdot\vec{n}=0$$

- Combine with ray equation:

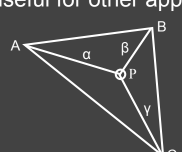
$$ray \equiv \vec{P}=\vec{P}_0+\vec{P}_1 t$$

$$(\vec{P}_0+\vec{P}_1 t)\cdot\vec{n}=\vec{A}\cdot\vec{n}$$

$$t=\frac{\vec{A}\cdot\vec{n}-\vec{P}_0\cdot\vec{n}}{\vec{P}_1\cdot\vec{n}}$$

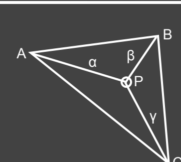
Ray inside Triangle

- Once intersect with plane, still need to find if in triangle
- Many possibilities for triangles, general polygons (point in polygon tests)
- We find parametrically [barycentric coordinates]. Also useful for other applications (texture mapping)



$$P=\alpha A+\beta B+\gamma C$$
$$\alpha\ge 0,\beta\ge 0,\gamma\ge 0$$
$$\alpha+\beta+\gamma=1$$

Ray inside Triangle



$$P=\alpha A+\beta B+\gamma C$$
$$\alpha\ge 0,\beta\ge 0,\gamma\ge 0$$
$$\alpha+\beta+\gamma=1$$

$$P-A=\beta(B-A)+\gamma(C-A)$$

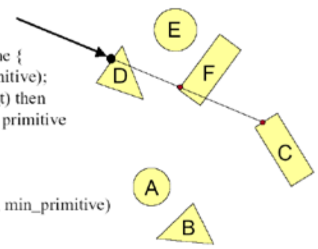
$$0\le\beta\le 1\ ,\ 0\le\gamma\le 1$$
$$\beta+\gamma\le 1$$

Other primitives

- Much early work in ray tracing focused on ray-primitive intersection tests
- Cones, cylinders, ellipsoids
- Boxes (especially useful for bounding boxes)
- General planar polygons
- Many more

Ray Scene Intersection

```
Intersection FindIntersection(Ray ray, Scene scene)
{
    min_t = infinity
    min_primitive = NULL
    For each primitive in scene {
        t = Intersect(ray, primitive);
        if (t > 0 && t < min_t) then
            min_primitive = primitive
            min_t = t
        }
    }
    return Intersection(min_t, min_primitive)
}
```



Transformed Objects

- E.g. transform sphere into ellipsoid
- Could develop routine to trace ellipsoid (compute parameters after transformation)
- May be useful for triangles, since triangle after transformation is still a triangle in any case
- But can also use original optimized routines

Ray-Tracing Transformed Objects

We have an optimized ray-sphere test
- But we want to ray trace an ellipsoid…

Solution: Ellipsoid transforms sphere
- Apply inverse transform to ray, use ray-sphere
- Allows for instancing (traffic jam of cars)
- Same idea for other primitives

Transformed Objects

- Consider a general 4x4 transform M
  - Will need to implement matrix stacks like in OpenGL
- Apply inverse transform M^{-1} to ray
  - Locations stored and transform in homogeneous coordinates
  - Vectors (ray directions) have homogeneous coordinate set to 0 [so there is no action because of translations]
- Do standard ray-surface intersection as modified
- Transform intersection back to actual coordinates
  - Intersection point p transforms as Mp
  - Distance to intersection if used may need recalculation
  - Normals n transform as $M^{-t}n$. Do all this before lighting

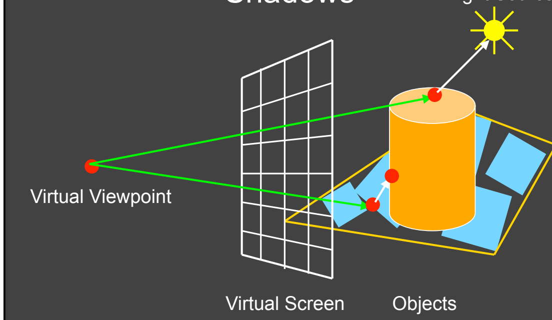
Outline

- History
- Basic Ray Casting (instead of rasterization)
  - Comparison to hardware scan conversion
- Ray-Surface Intersection
- *Shadows / Reflections (core algorithm)*
- Optimizations
- Current Research

Outline in Code

```
Image Raytrace (Camera cam, Scene scene, int width, int height)
{
    Image image = new Image (width, height) ;
    for (int i = 0 ; i < height ; i++)
        for (int j = 0 ; j < width ; j++) {
                Ray ray = RayThruPixel (cam, i, j) ;
                Intersection hit = Intersect (ray, scene) ;
                image[i][j] = FindColor (hit) ;
                }
    return image ;
}
```

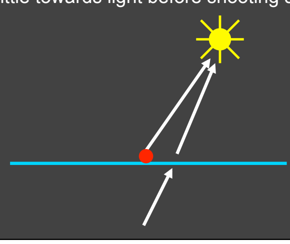
Shadows



Shadow ray to light is unblocked: object visible
Shadow ray to light is blocked: object in shadow

Shadows: Numerical Issues

- Numerical inaccuracy may cause intersection to be below surface (effect exaggerated in figure)
- Causing surface to incorrectly shadow itself
- Move a little towards light before shooting shadow ray

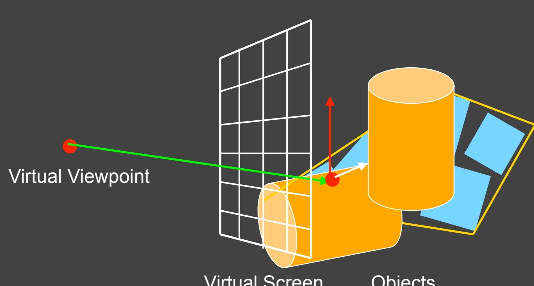


Shading Model

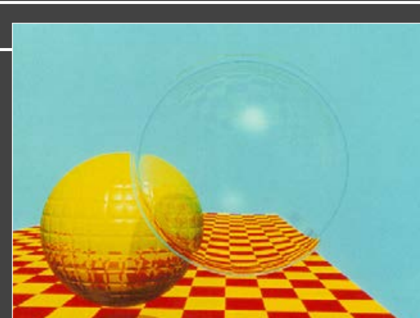
$$I = K_a + K_e + \sum_{i=1}^{n} V_i L_i (K_d \max\,(l_i \bullet n, 0) + K_s (\max(h_i \bullet n, 0))^s)$$

- Global ambient term, emission from material
- For each light, diffuse specular terms
- Note visibility/shadowing for each light (not in OpenGL)
- Evaluated per pixel per light (not per vertex)

Mirror Reflections/Refractions



Generate reflected ray in mirror direction,
Get reflections and refractions of objects



Turner Whitted 1980

Recursive Ray Tracing

For each pixel

- Trace Primary Eye Ray, find intersection
- Trace Secondary Shadow Ray(s) to all light(s)
  - Color = Visible ? Illumination Model : 0 ;
- Trace Reflected Ray
  - Color += reflectivity * Color of reflected ray

Recursive Shading Model

$$I = K_a + K_e + \sum_{i=1}^{n} V_i L_i (K_d \max(l_i \cdot n, 0) + K_s(\max(h_i \cdot n, 0))^s) + K_s I_R + K_T I_T$$

- Highlighted terms are recursive specularities [mirror reflections] and transmission (latter is extra credit)
- Trace secondary rays for mirror reflections and refractions, include contribution in lighting model
- GetColor calls RayTrace recursively (the I values in equation above of secondary rays are obtained by recursive calls)

Problems with Recursion

- Reflection rays may be traced forever
- Generally, set maximum recursion depth
- Same for transmitted rays (take refraction into account)

Effects needed for Realism

- (Soft) Shadows
- Reflections (Mirrors and Glossy)
- Transparency (Water, Glass)
- Interreflections (Color Bleeding)
- Complex Illumination (Natural, Area Light)
- Realistic Materials (Velvet, Paints, Glass)

Discussed in this lecture
Not discussed but possible with distribution ray tracing
Hard (but not impossible) with ray tracing; path tracing next time

Outline

- History
- Basic Ray Casting (instead of rasterization)
  - Comparison to hardware scan conversion
- Ray-Surface Intersection
- Shadows / Reflections (core algorithm)
- *Optimizations*
- Current Research

Some basic add ons

- Area light sources and soft shadows: break into grid of n x n point lights
  - Use jittering: Randomize direction of shadow ray within small box for given light source direction
  - Jittering also useful for antialiasing shadows when shooting primary rays
- More complex reflectance models
  - Simply update shading model
  - But at present, we can handle only mirror global illumination calculations

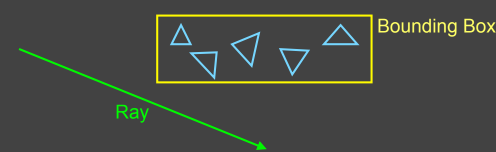
Acceleration

Testing each object for each ray is slow
- Fewer Rays
  - Adaptive sampling, depth control
- Generalized Rays
  - Beam tracing, cone tracing, pencil tracing etc.
- Faster Intersections
  - Optimized Ray-Object Intersections
  - ***Fewer Intersections***

We just discuss some approaches at high level

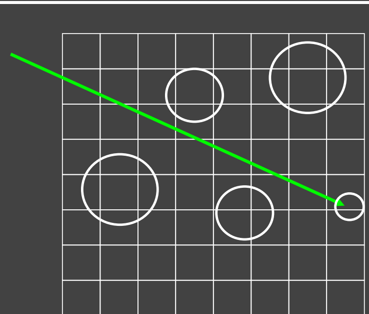
Acceleration Structures

Bounding boxes (possibly hierarchical)
If no intersection bounding box, needn't check objects



Spatial Hierarchies (Oct-trees, kd trees, BSP trees)

Acceleration Structures: Grids



Acceleration and Regular Grids

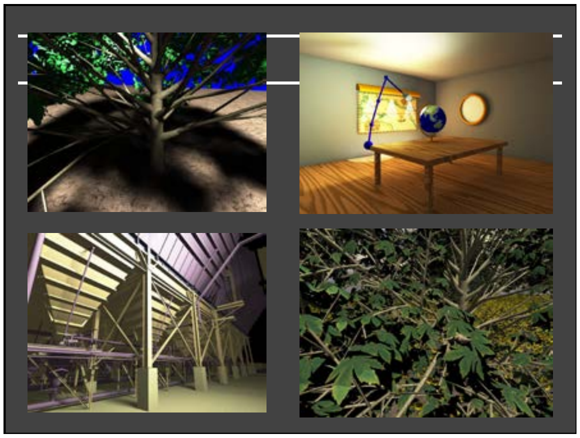
- Simplest acceleration, for example 5x5x5 grid
- For each grid cell, store overlapping triangles
- March ray along grid (need to be careful with this), test against each triangle in grid cell
- More sophisticated: kd-tree, oct-tree bsp-tree
- Or use (hierarchical) bounding boxes
- Some acceleration is critical for path tracing

Outline

- History
- Basic Ray Casting (instead of rasterization)
  - Comparison to hardware scan conversion
- Ray-Surface Intersection
- Shadows / Reflections (core algorithm)
- Optimizations
- *Current Research*

Interactive Raytracing

- Ray tracing historically slow
- Now viable alternative for complex scenes
  - Key is sublinear complexity with acceleration; need not process all triangles in scene
- Allows many effects hard in hardware
- NVIDIA OptiX ray-tracing API like OpenGL
- Recent NVIDIA OptiX release major advance
  - Ray tracing now practical for games
  - Integration with Microsoft's DirectX
  - Dedicated Hardware
  - Machine Learning for denoising (later in course)



Raytracing on Graphics Hardware

- Modern Programmable Hardware general streaming architecture
- Can map various elements of ray tracing
- Kernels like eye rays, intersect etc.
- In vertex or fragment programs
- Convergence between hardware, ray tracing

[Purcell et al. 2002, 2003]

http://graphics.stanford.edu/papers/photongfx