Ray Tracing 1: The Basics
Today’s topics

- **pbrt overview**
- Basic algorithms
- Ray-surface intersection
- Accelerating ray tracing of large numbers of geometric primitives

- Next time: more advanced primitives, incremental acceleration techniques, and practical floating point issues
Three ideas about light rays

1. Light travels in straight lines (mostly)
2. Light rays do not interfere with each other if they cross (light is invisible!)
3. Light rays travel from the light sources to the eye (but the physics is invariant under path reversal - reciprocity).
Ray Tracing in Computer Graphics

Appel 1968 - Ray casting

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

Camera → Image → View Ray → Shadow Ray → Scene Object

Light Source
Ray Tracing

Shooting rays to determine what is visible to camera at each pixel

Ray represented by its origin and direction:

\[ r(t) = o + t \vec{d} \]
Ray Tracing

Shooting rays to determine whether a surface is visible from a light source.
Ray Tracing in Computer Graphics

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

1. Always send ray to the light source (unless glass or mirror)

2. Recursively generate reflected rays (mirror) and transmitted rays (glass)

Time:
- VAX 11/780 (1979) 74m
- PC (2006) 6s
- GPU (2012) 1/30s

Spheres and Checkerboard
T. Whitted, 1979
Ray Tracing

Shooting rays determine what light reaches a surface
Mirror, depth 1
Mirror, depth 2
Mirror, depth 3
Mirror, depth 10
Glass, depth 1
Glass, depth 2
Glass, depth 3
Glass, depth 10
Bidirectional paths
The key primitive to make the pictures above is ray intersection with scene geometry

Let’s start with intersection of a ray and a single simple surface
Implicit Equation for Plane

Plane is defined by:

- Its normal: \( n \)
- A point \( p \) on the plane

The plane is all points \( p' \), where \( p' - p \) is orthogonal to \( n \).

\[
(p' - p) \cdot \overrightarrow{n} = 0
\]

\( (n, p, p' \) on this slide are 3-vectors)
Ray-Plane Intersection

Ray: \( r(t) = o + t \overrightarrow{d} \)

Plane: \((p' - p) \cdot \overrightarrow{n} = 0\)

Want \( t \) where the ray intersects the plane...

Substitute ray equation into plane equation:

\[
(r(t) - p) \cdot \overrightarrow{n} = ((o + t \overrightarrow{d}) - p) \cdot \overrightarrow{n} = 0
\]

\[
t = \frac{(p - o) \cdot \overrightarrow{n}}{\overrightarrow{d} \cdot \overrightarrow{n}}
\]
Finding The Closest Intersection

\[ r(0) \quad \overrightarrow{d} \quad r(t) \]

Ray

Plane 1

Plane 2

Plane 3

\[ r(t_0) \]

\[ r(t_1) \]

\[ r(t_2) \]
Consider intersection with \( x=c \) plane... \( N=[1 \ 0 \ 0]^T \)

\[
t = \frac{(p - o) \cdot \vec{n}}{\vec{d} \cdot \vec{n}} \quad \Rightarrow \quad t = \frac{p_x - o_x}{d_x} \quad \Rightarrow \quad t = a p_x + b
\]

Where \( a \) and \( b \) can be precomputed from the ray (box independent): \( a=1/d_x \) and \( b=-o_x/d_x \)
What About Rays Parallel to a Plane?

**Math says:**
\[ t = \frac{p_x - o_x}{d_x} \]

**IEEE Floating Point standard says:**
1. positive num / 0 = +Inf
   negative num / 0 = -Inf
2. +Inf > all other floats
   -Inf < all other floats
Ray-Sphere Intersection

Ray: \[ r(t) = \mathbf{o} + t \mathbf{d} \]

Sphere: \[ \| \mathbf{p} - \mathbf{c} \|^2 - r^2 = 0 \]
\[ (\mathbf{o} + t \mathbf{d} - \mathbf{c})^2 - r^2 = 0 \]

\[ a = \mathbf{d} \cdot \mathbf{d} \]
\[ b = 2(\mathbf{o} - \mathbf{c}) \cdot \mathbf{d} \]
\[ c = ( (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) ) - r^2 \]
Ray-Triangle Intersection
Ray-Triangle Intersection

1. Find ray-plane intersection point using the methods developed previously
2. Test whether the intersection point is inside the triangle
The signed area of the parallelogram given by the vectors \( v_1 = (x_1, y_1) \) and \( v_2 = (x_2, y_2) \) is given by

\[
\begin{vmatrix}
  x_1 & x_2 \\
  y_1 & y_2
\end{vmatrix}
= (x_1y_2) - (x_2y_1)
\]

Half of this area is the area of the triangle determined by the 3 points.
Area of a triangle with vertices: \( p_i = (x_i, y_i) \)

\[
\frac{1}{2} \begin{vmatrix}
  x_1 - x_0 & x_2 - x_0 \\
  y_1 - y_0 & y_2 - y_0
\end{vmatrix} = \frac{1}{2}((x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0))
\]
Barycentric Coordinates

\[ a_0(p) = \text{Area}(p_1, p_2, p) \]
\[ a_1(p) = \text{Area}(p_2, p_0, p) \]
\[ a_2(p) = \text{Area}(p_0, p_1, p) \]

Define barycentric coordinates:

\[ b_i = \frac{a_i}{\text{Area}(p_0, p_1, p_2)}, \quad 0 \leq i \leq 2 \]

\( p \) is inside the triangle if \( b_0 > 0, b_1 > 0, b_2 > 0 \)
Ray-Triangle Intersection

**Points on a plane:** \( \mathbf{p} = b_0 \mathbf{p}_0 + b_1 \mathbf{p}_1 + b_2 \mathbf{p}_2 \)

\[
\begin{bmatrix}
\mathbf{p}_0 & \mathbf{p}_1 & \mathbf{p}_2 \\
\end{bmatrix}
\begin{bmatrix}
b_0 \\
b_1 \\
b_2 \\
\end{bmatrix}
= \begin{bmatrix}
\mathbf{p} \\
\end{bmatrix}
\]

1. **Find ray-plane intersection point**

2. **Test whether that point is inside the triangle**

\[
\begin{bmatrix}
b_0 \\
b_1 \\
b_2 \\
\end{bmatrix}
= \begin{bmatrix}
\mathbf{p}_0 & \mathbf{p}_1 & \mathbf{p}_2 \\
\end{bmatrix}^{-1}
\begin{bmatrix}
\mathbf{p} \\
\end{bmatrix}
\]

**Inside iff** \( b_0 > 0, b_1 > 0, b_2 > 0 \)
Ray Triangle Intersection

\[ \mathbf{o} + t \mathbf{d} = (1 - b_1 - b_2)\mathbf{p}_0 + b_1\mathbf{p}_1 + b_2\mathbf{p}_2 \]

3 equations, 3 unknowns \((t, b_1, b_2)\)

\[
\begin{bmatrix}
  t \\
  b_1 \\
  b_2 \\
\end{bmatrix}
= \frac{1}{s_1 \cdot e_1}
\begin{bmatrix}
  \mathbf{s}_2 \cdot \mathbf{e}_2 \\
  \mathbf{s}_1 \cdot \mathbf{s} \\
  \mathbf{s}_2 \cdot \mathbf{d} \\
\end{bmatrix}
\]

\[
\begin{aligned}
\mathbf{e}_1 &= \mathbf{p}_1 - \mathbf{p}_0 \\
\mathbf{e}_2 &= \mathbf{p}_2 - \mathbf{p}_0 \\
\mathbf{s} &= \mathbf{o} - \mathbf{p}_0 \\
\mathbf{s}_1 &= \mathbf{d} \times \mathbf{e}_2 \\
\mathbf{s}_2 &= \mathbf{s} \times \mathbf{e}_1
\end{aligned}
\]

[Möller and Trumbore 1997]
Ray-Implicit Surface Intersection

Implicit surface

\[ f(x, y, z) = 0 \]

Substitute ray equation

\[ x = o_x + t d_x \]
\[ y = o_y + t d_y \]
\[ z = o_z + t d_z \]

Univariate root finding

\[ f^*(t) = 0 \]
Acceleration Structures
Complex scenes

3.1B tris
Disney Moana scene

Released for rendering research purposes in 2018. 
15 billion primitives in scene (more than 90M unique geometric primitives)
Disney Moana scene
Disney Moana scene
How do we find closest ray-scene intersection without individually performing ray-primitive intersection for all scene primitives?
Uniform grid

- Partition space into equal sized volumes (volume-elements or “voxels”)
- Each grid cell contains primitives that overlap the voxel.
  - Cheap to construct
- Walk ray through volume in order
  - Efficient implementation possible (think: 3D line rasterization)
  - Only consider intersection with primitives in voxels the ray intersects
Objects Overlapping Multiple Cells

**Mistake: Output first intersection found**

![Diagram showing objects overlapping multiple cells]
Objects Overlapping Multiple Cells

Solution: Check whether intersection is inside cell
Objects Overlapping Multiple Cells

Solution: Check whether intersection is inside cell
Mailboxes

Solution: Check whether intersection is inside cell

Problem: Objects tested for intersection multiple times
Mailboxes

Solution: Check whether intersection is inside cell

Problem: Objects tested for intersection multiple times
Solution: Mailboxes
- Assign each ray an increasing number
- Primitive intersection cache (mailbox)
  - Give each ray a number $N$
  - Store intersection point and ray $N$ with each primitive
  - Only re-intersect if ray $N$ is greater than last ray $N$
- This solution creates problems for parallel tracing.
What should the grid resolution be?

Too few grids cell: degenerates to brute-force approach

Too many grid cells: incur significant cost traversing through cells with empty space
Grid size heuristic

Choose number of cells \(\sim\) total number of primitives

Intersection cost: \(O\left(\frac{3}{\sqrt{N}}\right)\)

(assuming 3D grid)

(yields constant prims per cell for any scene size — assuming uniform distribution of primitives)
When uniform grids work well: uniform distribution of primitives in scene

[Image credit: www.kevinboulanger.net/grass.html]
Uniform grids cannot adapt to non-uniform distribution of geometry in scene

“Teapot in a stadium problem”

Scene has large spatial extent.

Contains a high-resolution object that has small spatial extent (ends up in one grid cell)
When uniform grids do not work well:
non-uniform distribution of geometric detail
Quad-tree / octree

Quad-tree: nodes have 4 children (partitions 2D space)

Octree: nodes have 8 children (partitions 3D space)

Like uniform grid: easy to build (don’t have to choose partition planes)

Has greater ability to adapt to location of scene geometry than uniform grid.

But less ability than a K-D tree where partitioning planes can adapt to location of geometry (next slide)
K-D tree

- Recursively partition space via axis-aligned partitioning planes
- Interior nodes correspond to spatial splits
- Ability to put spatial splits anywhere gives greater adaptability than octree
Space-partitioning (e.g. grid, octrees, K-D tree) partitions space into disjoint regions (primitives may be contained in multiple regions of space)

Primitive partitioning (e.g. bounding volume hierarchy): partitions primitives into disjoint sets (but sets of primitives may overlap in space)
One simple idea

“Early out” — Skip ray-primitive test if it is computationally easy to determine that ray does not intersect primitives

E.g., A ray cannot intersect a primitive if it doesn’t intersect the bounding box containing it!

Does not change asymptotic complexity of ray-scene intersection. But reduces cost by a constant if ray is far away from most triangles.
Bounding volume hierarchy (BVH)
Bounding volume hierarchy (BVH)
Bounding volume hierarchy (BVH)
Bounding volume hierarchy (BVH)

- **Leaf nodes:**
  - Contain small list of primitives

- **Interior nodes:**
  - Proxy for a large subset of primitives
  - Stores bounding box for all primitives in subtree
Ray-scene intersection using a BVH

```c
struct Node {
    bool leaf; // true if node is a leaf
    BBox bbox; // min/max coords of enclosed primitives
    Node* child1; // "left" child (could be NULL)
    Node* child2; // "right" child (could be NULL)
    Primitive* primList; // for leaves, stores primitives
};

struct HitInfo {
    Primitive* prim; // which primitive did the ray hit?
    float t; // at what t value along ray?
};

void find_closest_hit(Ray* ray, Node* node, HitInfo* closest) {
    HitInfo hit = ray_box_intersect(ray, node->bbox); // test ray against node's bounding box
    if (hit.t > closest.t) // don't update the hit record
        return;

    if (node->leaf) {
        for (each primitive p in node->primList) {
            hit = ray_prim_intersect(ray, p);
            if (hit.prim != NULL && hit.t < closest.t) {
                closest.prim = p;
                closest.t = t;
            }
        }
    } else {
        find_closest_hit(ray, node->child1, closest);
        find_closest_hit(ray, node->child2, closest);
    }
}
```

Can this occur if ray hits the box? (assume hit.t is INF if ray misses box)
void find_closest_hit(Ray* ray, Node* node, HitInfo* closest) {
    if (node->leaf) {
        for (each primitive p in node->primList) {
            hit = ray_prim_intersect(ray, p);
            if (hit.prim != NULL && t < closest.t) {
                closest.prim = p;
                closest.t = t;
            }
        }
    } else {
        HitInfo hit1 = ray_box_intersect(ray, node->child1->bbox);
        HitInfo hit2 = ray_box_intersect(ray, node->child2->bbox);
        if (hit1.t <= hit2.t) {
            find_closest_hit(ray, node->child1, closest);
            if (second child’s t is closer than closest.t) {
                find_closest_hit(ray, node->child2, closest); // why might we still need to do this?
            }
        } else {
            find_closest_hit(ray, node->child2, closest);
        }
    }
}
Aside: another type of query: any hit

Sometimes it is useful to know if the ray hits ANY primitive in the scene at all (don’t care about distance to first hit)

```c
bool find_any_hit(Ray* ray, Node* node) {
    if (!ray_box_intersect(ray, node->bbox))
        return false;

    if (node->leaf) {
        for (each primitive p in node->primList) {
            hit = ray_prim_intersect(ray, p);
            if (hit.prim)
                return true;
        }
    } else {
        return ( find_closest_hit(ray, node->child1, closest) ||
                 find_closest_hit(ray, node->child2, closest) );
    }
}
```

Interesting question of which child to enter first. How might you make a good decision?
Why “any hit” queries?

Shadow computations!
class Shape {
    public:
        Bounds3f ObjectBound() const;
        Bounds3f WorldBound() const;
        bool Intersect(const Ray &ray, Float *tHit, SurfaceInteraction *isect, bool testAlphaTexture) const;
        bool IntersectP(const Ray &ray, bool testAlphaTexture);
        Float Area() const;
        // …
};
Bounding volume hierarchy (BVH)

Left: two different BVH organizations of the same scene containing 22 primitives.

Is one BVH better than the other?
For a given set of primitives, there are many possible BVHs

\(2^N \text{ ways to partition } N \text{ primitives into two groups}\)
How would you partition these triangles into two groups?
What about these?
Intuition about a “good” partition?

Partition into child nodes with equal numbers of primitives

Better partition
Intuition: avoid bboxes with significant empty space
Which partition is fastest?

What is the cost of tracing a ray?

\[
\text{Cost}(\text{node}) = \text{C}_{\text{trav}} + \text{Prob} (\text{hit L}) \ast \text{Cost}(\text{L}) + \text{Prob} (\text{hit R}) \ast \text{Cost}(\text{R})
\]

\text{C}_{\text{trav}} = \text{cost of traversing a cell}

\text{Cost}(\text{L}) = \text{cost of traversing left child}

\text{Cost}(\text{R}) = \text{cost of traversing right child}
Intuition about a “good” partition?

Two equal sized groups:
child costs equal but probabilities unequal

Two unequal sized groups:
child costs unequal but sum of probabilities now much lower
Projected Area and Surface Area

The probability of ray in a given direction hitting an object with surface area $S$ is proportional to its projected area $A$ in the direction of the ray.
Average Proj Area and Surface Area

The probability of ray in a any direction hitting an object with surface area $S$ is proportional to its average projected area

**Average projected area:**

$$\bar{A} = \frac{1}{4\pi} \int A(\omega) \, d\omega$$

**Crofton’s theorem:**

$$\bar{A} = \frac{S}{4} \quad \text{(Convex shapes only)}$$
Ray Intersection Probability

The probability of a random ray hitting a convex shape A enclosed by another convex shape B is:

\[ P(\text{hit} A | \text{hit} B) = \frac{S_A}{S_B} \]

Leads to surface area heuristic:

\[ \text{Cost(node)} = C_{\text{trav}} + \text{SA}(L) \times \text{Cost}(L) + \text{SA}(R) \times \text{Cost}(R) \]

- \( C_{\text{trav}} \): the ratio of cost to traverse to cost to intersect (\( C_{\text{trav}} = 1:5 - 1:1.5 \) is typical)

Assumptions of the SAH (which may not hold in practice!):
- Rays are randomly distributed
- Rays are not occluded
Cost

Need the probabilities
  ■ Proportional to surface area

Need the child cell costs
  ■ Triangle count is a good approximation

Cost(node) = C_{trav} + SA(L) \times \text{TriCount}(L) + SA(R) \times \text{TriCount}(R)
Termination Criteria

When should we stop splitting?

■ Bad: depth limit, number of triangles
■ Good: When the split does not lower the cost

Threshold of cell size

■ Absolute probability \( \frac{SA(\text{node})}{SA(\text{scene})} \) small
Basic Top-down SAH-based Build

Partition(list of prims) {

    if (termination criteria reached) {
        // make leaf node
    }

    (prim_list_1, prim_list2) = // perform SAH split

    // recursive calls can execute in parallel
    left_child = Partition(prim_list_1)
    right_child = Partition(prim_list_2)
}
Finding a good partition

- Constrain search for good partitions to axis-aligned spatial partitions
- Choose an axis; choose a split plane on that axis
- Partition primitives by the side of splitting plane their centroid lies
- SAH changes only when split plane moves past triangle boundary
- Have to consider large number of possible split planes... (2N possibilities)
Efficient modern approximation: split spatial extent of primitives into $B$ buckets ($B$ is typically small: $B < 32$)

For each axis: $x, y, z$:
- Initialize bucket counts to 0, per-bucket bboxes to empty
- For each primitive $p$ in node:
  - $b = \text{compute\_bucket}(p.\text{centroid})$
  - $b.\text{bbox}.\text{union}(p.\text{bbox})$
  - $b.\text{prim\_count}++$
- For each of the $B-1$ possible partitioning planes evaluate SAH
- Use lowest cost partition found (or make node a leaf)
300k tris
Time: Accel 27.4%, Ray-Triangle 10.2%
**Avg. 5.8 tri isect tests / ray**
Memory: Accel 19 MB, Tris 21 MB
5.3M tris
Time: Accel 42.7%, Ray-Triangle 13.3%
Avg. 4.5 tri isect tests / ray
Memory: Accel 339 MB, Tris 901 MB
3.1B tris
Time: Accel 74.7%, Ray-Triangle 13.3%
Avg. 32.9 tri isect tests / ray
Memory: Accel 1.47 GB, Tris 4.58 GB
Recall Moana:
Moana costs

Number of nodes visited

Num ray-triangle tests
Another example
Another example: diagonal geometry

Number of nodes visited

Num ray-triangle tests
Axis-alignment and performance

Wall and its bounding box

Rotated wall and its bounding box
Original Scene

Rendering time: 27m 38s
Transformed (Rotated in World Space) Scene

Rendering time: 1h 55m 45s
Axis-alignment and performance

Rotated wall and its bounding box

Work-around: refine bounding boxes

Note: this introduces back the idea of partitioning space!
(Recall octree, KD-tree)
Top down “greedy” build

“greedy” algorithm: not guaranteed to give global optimum

Partition(list of prims) {

    if (termination criteria reached) {
        // make leaf node
    }

    (prim_list_1, prim_list2) = // perform SAH split

    // recursive calls can execute in parallel
    left_child = Partition(prim_list_1)
    right_child = Partition(prim_list_2)
}

Recall SAH cost estimate:
Cost(node) = C_trav + SA(L)*TriCount(L) + SA(R)*TriCount(R)
Modern, fast and high quality BVH construction schemes

Combine “top-down” divide-and-conquer build with “bottom up” construction techniques

Step 1: build low-quality BVH quickly

Step 2: Use initial BVH to accelerate construction of high-quality BVH
Building a low-quality BVH quickly

1. Discretize each dimension of scene into $2^B$ cells
2. Compute index of centroid of bounding box of each primitive: $(c_i, c_j, c_k)$
3. Interleave bits of $c_i$, $c_j$, $c_k$ to get $3B$ bit-Morton code
4. Sort primitives by Morton code (primitives now ordered with high locality in 3D space: in a space-filling curve!)
   - $O(N)$ parallel radix sort

Leads to simple, highly parallelizable BVH build:

```
Partition(int i, primitives):
    node.bbox = bbox(primitives)
    (left, right) = partition prims by bit i
    if there are more bits:
        Partition(left, i+1);
        Partition(right, i+1);
    else:
        make a leaf node
```

[2D Morton Order]

[Lauterbach 09, Pantaleoni 10]
Karras 2013 bottom up treelet-based construction

Step 1: (top down) build low quality BVH quickly using Morton codes
Step 2: (bottom up) walk from leaves toward root forming small treeless

For each treelet, exhaustively try all possible combinations to find optimal (SAH) treelet
- Brute force search implemented using dynamic programming method

Shaded region:
- treelet with 7 leaf nodes

After optimization: this is the optimal treelet for these nodes (minimal SAH cost)
Can afford to build a better BVH if you are shooting many rays (can amortize cost)

The graph below plots effective ray throughput (Mrays/sec) as a function of the number of rays traced per BVH build
- More rays = can amortize costs of BVH build across many ray trace operations
PBRT Overview
PHYSICALLY BASED RENDERING
From Theory to Implementation
Third Edition

http://www.pbr-book.org/
**Table 1.1: Main Interface Types.** Most of pbrt is implemented in terms of 10 key abstract base classes, listed here. Implementations of each of these can easily be added to the system to extend its functionality.

<table>
<thead>
<tr>
<th>Base class</th>
<th>Directory</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shape</td>
<td>shapes/</td>
<td>3.1</td>
</tr>
<tr>
<td>Aggregate</td>
<td>accelerators/</td>
<td>4.2</td>
</tr>
<tr>
<td>Camera</td>
<td>cameras/</td>
<td>6.1</td>
</tr>
<tr>
<td>Sampler</td>
<td>samplers/</td>
<td>7.2</td>
</tr>
<tr>
<td>Filter</td>
<td>filters/</td>
<td>7.8</td>
</tr>
<tr>
<td>Material</td>
<td>materials/</td>
<td>9.2</td>
</tr>
<tr>
<td>Texture</td>
<td>textures/</td>
<td>10.3</td>
</tr>
<tr>
<td>Medium</td>
<td>media/</td>
<td>11.3</td>
</tr>
<tr>
<td>Light</td>
<td>lights/</td>
<td>12.2</td>
</tr>
<tr>
<td>Integrator</td>
<td>integrators/</td>
<td>1.3.3</td>
</tr>
</tbody>
</table>
Figure 1.17: Class Relationships for the Main Rendering Loop in the SamplerIntegrator::Render() Method in core/integrator.cpp. The Sampler provides a sequence of sample values, one for each image sample to be taken. The Camera turns a sample into a corresponding ray from the film plane, and the Li() method implementation computes the radiance along that ray arriving at the film. The sample and its radiance are given to the Film, which stores their contribution in an image. This process repeats until the Sampler has provided as many samples as are necessary to generate the final image.
Figure 1.19: Class Relationships for Surface Integration. The main rendering loop in the SamplerIntegrator computes a camera ray and passes it to the Li() method, which returns the radiance along that ray arriving at the ray’s origin. After finding the closest intersection, it computes the material properties at the intersection point, representing them in the form of a BSDF. It then uses the Lights in the Scene to determine the illumination there. Together, these give the information needed to compute the radiance reflected back along the ray at the intersection point.
Shape Interface (Simplified)

class Shape {
    public:
        Bounds3f ObjectBound() const;
        Bounds3f WorldBound() const;
        bool Intersect(const Ray &ray, Float *tHit,
                        SurfaceInteraction *isect,
                        bool testAlphaTexture) const;
        bool IntersectP(const Ray &ray,
                        bool testAlphaTexture);
        Float Area() const;
        // ... 
};
**Surface Interaction (Simplified)**

Information about the surface point hit by a ray.

class SurfaceInteraction {
    Point3f p;
    Normal3f n;

    Point2f uv;
    Vector3f dpdu, dpdv;
    Normal3f dndu, dndv;

    struct {
        Normal3f n;
        Vector3f dpdu, dpdv;
        Normal3f dndu, dndv;
    } shading;

    // ...
};
Primitives in PBRT

pbtrt Primitive base class

- Shape
- Material (for a later class)

class Primitive {
  public:
      virtual Bounds3f WorldBound() const = 0;
      virtual bool Intersect(const Ray &r,
                              SurfaceInteraction *) const = 0;
      virtual bool IntersectP(const Ray &r) const = 0;
      virtual const AreaLight *GetAreaLight() const = 0;
      virtual const Material *GetMaterial() const = 0;
      virtual void ComputeScatteringFunctions(...) const = 0;
};
Primitives

Collections

- TransformedPrimitive: Transformation + primitive
- Aggregate
  - Treat acceleration data structures as primitives
  - Two types of accelerators: kdtree.cpp, and bvh.cpp
  - May nest accelerators of different types

```c++
class Scene {
    // ...
    bool Intersect(const Ray &ray,
                   SurfaceInteraction *isect) const {
        return aggregate->Intersect(ray, isect);
    }
    std::shared_ptr<Primitive> aggregate;
};
```
Two-level Acceleration Structures

Top-level acceleration structure (e.g., BVH)

Bottom-level acceleration structures
(Just primitives with boxes and intersect() methods from the perspective of the top level acceleration structure)