3D Rendering and Ray Casting

Jason Lawrence

CS 4810: Graphics

Acknowledgment: slides by Misha Kazhdan, Allison Klein, Tom Funkhouser, Adam Finkelstein and David Dobkin
Rendering

- Generate an image from geometric primitives
Rendering

- Generate an image from geometric primitives
3D Rendering Example

What issues must be addressed by a 3D rendering system?
Overview

• 3D scene representation
• 3D viewer representation
• Ray Casting
Overview

- 3D scene representation
- 3D viewer representation
- Ray casting

How is the 3D scene described in a computer?
3D Scene Representation

- Scene is usually approximated by 3D primitives
  - Point
  - Line segment
  - Polygon
  - Polyhedron
  - Curved surface
  - Solid object
  - etc.
3D Point

- Specifies a location
3D Point

- Specifies a location
  - Represented by three coordinates
  - Infinitely small

```c
typedef struct {
    Coordinate x;
    Coordinate y;
    Coordinate z;
} Point;
```
3D Vector

- Specifies a direction and a magnitude
3D Vector

- Specifies a direction and a magnitude
  - Represented by three coordinates
  - Magnitude $|V| = \sqrt{dx \, dx + dy \, dy + dz \, dz}$
  - Has no location

```c
typedef struct {
    Coordinate dx;
    Coordinate dy;
    Coordinate dz;
} Vector;
```
Linear Algebra: a Little Review

- What is...?
- $V_1 \cdot V_1 = ?$
Linear Algebra: a Little Review

• What is...?

• $V_1 \cdot V_1 = dx \, dx + dy \, dy + dz \, dz$
Linear Algebra: a Little Review

• What is...?

• $V_1 \cdot V_1 = (\text{Magnitude})^2$
Linear Algebra: a Little Review

- $V_1 \cdot V_1 = (\text{Magnitude})^2$
- Now, let $V_1$ and $V_2$ both be unit-length vectors.
- What is…?
- $V_1 \cdot V_1 =$
Linear Algebra: a Little Review

- $V_1 \cdot V_1 = (\text{Magnitude})^2$
- Now, let $V_1$ and $V_2$ both be unit-length vectors.
- What is...?
- $V_1 \cdot V_1 = \|V_1\| \|V_1\| \cos(\Theta)$
Linear Algebra: a Little Review

• $V_1 \cdot V_1 = \text{Magnitude}^2$

• Now, let $V_1$ and $V_2$ both be unit-length vectors.

• What is...?

• $V_1 \cdot V_1 = ||V_1|| \cdot ||V_1|| \cos(\Theta) = \cos(\Theta)$
Linear Algebra: a Little Review

- \( V_1 \cdot V_1 = (\text{Magnitude})^2 \)
- Now, let \( V_1 \) and \( V_2 \) both be unit-length vectors.
- What is…?
- \( V_1 \cdot V_1 = \|V_1\| \|V_1\| \cos(\Theta) = \cos(\Theta) = \cos(0) \)
Linear Algebra: a Little Review

• $V_1 \cdot V_1 = (\text{Magnitude})^2$

• Now, let $V_1$ and $V_2$ both be unit-length vectors.

• What is...?

• $V_1 \cdot V_1 = 1$
Linear Algebra: a Little Review

- \( V_1 \cdot V_1 = (\text{Magnitude})^2 \)
- Now, let \( V_1 \) and \( V_2 \) both be unit-length vectors.
- What is…?
- \( V_1 \cdot V_1 = 1 \)
- \( V_1 \cdot V_2 = \)
Linear Algebra: a Little Review

- $V_1 \cdot V_1 = \text{(Magnitude)}^2$
- Now, let $V_1$ and $V_2$ both be unit-length vectors.
- What is...?
- $V_1 \cdot V_1 = 1$
- $V_1 \cdot V_2 = \|V_1\| \|V_2\| \cos(\Theta)$
Linear Algebra: a Little Review

• $V_1 \cdot V_1 = (\text{Magnitude})^2$

• Now, let $V_1$ and $V_2$ both be unit-length vectors.

• What is…?

• $V_1 \cdot V_1 = 1$

• $V_1 \cdot V_2 = \|V_1\| \|V_2\| \cos(\Theta) = \cos(\Theta)$
Linear Algebra: a Little Review

- $V_1 \cdot V_1 = \text{(Magnitude)}^2$
- Now, let $V_1$ and $V_2$ both be unit-length vectors.
- What is...?
- $V_1 \cdot V_1 = 1$
- $V_1 \cdot V_2 = \cos(\Theta) = \text{(adjacent / hyp)}$
Linear Algebra: a Little Review

• $V_1 \cdot V_1 = (\text{Magnitude})^2$

• Now, let $V_1$ and $V_2$ both be unit-length vectors.

• What is…?

• $V_1 \cdot V_1 = 1$

• $V_1 \cdot V_2 = (\text{adjacent} / 1)$
Linear Algebra: a Little Review

- $V_1 \cdot V_1 = (\text{Magnitude})^2$
- Now, let $V_1$ and $V_2$ both be unit-length vectors.
- What is...?
- $V_1 \cdot V_1 = 1$
- $V_1 \cdot V_2 = \text{length of } V_1 \text{ projected onto } V_2 \text{ (or vice-versa)}$
3D Vector

- Specifies a direction and a magnitude
  - Represented by three coordinates
  - Magnitude $||V|| = \sqrt{dx \cdot dx + dy \cdot dy + dz \cdot dz}$
  - Has no location

```c
typedef struct {
    Coordinate dx;
    Coordinate dy;
    Coordinate dz;
} Vector;
```

- Cross product of two 3D vectors
  - $V_1 \times V_2 = \text{Vector normal to plane } V_1, V_2$
  - $|| V_1 \times V_2 || = ||V_1|| \cdot ||V_2|| \cdot \sin(\Theta)$
Linear Algebra: More Review

• Let $C = A \times B$:
  - $Cx = AyBz - AzBy$
  - $Cy = AzBx - AxBz$
  - $Cz = AxBy - AyBx$

• $A \times B = -B \times A$ (remember “right-hand” rule)

• We can do similar derivations to show:
  - $V_1 \times V_2 = \|V_1\| \|V_2\| \sin(\Theta) n$, where $n$ is unit vector normal to $V_1$ and $V_2$
  - $\|V_1 \times V_1\| = 0$
  - $\|V_1 \times (-V_1)\| = 0$

• http://physics.syr.edu/courses/java-suite/crosspro.html
3D Line Segment

• Linear path between two points
3D Line Segment

• Use a linear combination of two points
  • Parametric representation:
    \[ P = P_1 + t (P_2 - P_1), \quad (0 \leq t \leq 1) \]

```c
typedef struct {
    Point P1;
    Point P2;
} Segment;
```
3D Ray

• Line segment with one endpoint at infinity
  • Parametric representation:
    \[ P = P_1 + t \mathbf{V}, \quad (0 \leq t < \infty) \]

```c
typedef struct {
    Point P1;
    Vector V;
} Ray;
```
3D Line

- Line segment with both endpoints at infinity
  - Parametric representation:
    \[ P = P_1 + t \cdot V, \quad (-\infty < t < \infty) \]

```c
typedef struct {
    Point P1;
    Vector V;
} Line;
```
3D Plane

• A linear combination of three points
3D Plane

- A linear combination of three points
  - Implicit representation:
    - \( P \cdot N + d = 0 \), or
    - \( ax + by + cz + d = 0 \)
  - \( N \) is the plane “normal”
    - Unit-length vector
    - Perpendicular to plane

```c
typedef struct {
    Vector N;
    Distance d;
} Plane;
```
3D Polygon

- Area “inside” a sequence of coplanar points
  - Triangle
  - Quadrilateral
  - Convex
  - Star-shaped
  - Concave
  - Self-intersecting

```c
typedef struct {
    Point *points;
    int npoints;
} Polygon;
```

Points are in counter-clockwise order

- Holes (use > 1 polygon struct)
3D Sphere

- All points at distance “r” from point “(c_x, c_y, c_z)”
  - Implicit representation:
    » \((x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = r^2\)
  - Parametric representation:
    » \(x = r \cos(\phi) \cos(\Theta) + c_x\)
    » \(y = r \cos(\phi) \sin(\Theta) + c_y\)
    » \(z = r \sin(\phi) + c_z\)

```c
typedef struct {
    Point center;
    Distance radius;
} Sphere;
```
Other 3D primitives

- Cone
- Cylinder
- Ellipsoid
- Box
- Etc.
3D Geometric Primitives

- More detail on 3D modeling later in course
  - Point
  - Line segment
  - Polygon
  - Polyhedron
  - Curved surface
  - Solid object
  - etc.
Overview

• 3D scene representation
• 3D viewer representation
• Visible surface determination
• Lighting simulation

How is the viewing device described in a computer?
Camera Models

- The most common model is pin-hole camera
  - All captured light rays arrive along paths toward focal point without lens distortion (everything is in focus)

Other models consider...
- Depth of field
- Motion blur
- Lens distortion
Camera Parameters

- What are the parameters of a camera?
Camera Parameters

- **Position**
  - Eye position (px, py, pz)

- **Orientation**
  - View direction (dx, dy, dz)
  - Up direction (ux, uy, uz)

- **Aperture**
  - Field of view (xfov, yfov)

- **Film plane**
  - "Look at" point
  - View plane normal
Other Models: Depth of Field

Close Focused  Distance Focused

P. Haeberli
Other Models: Motion Blur

- Mimics effect of open camera shutter
- Gives perceptual effect of high-speed motion
- Generally involves temporal super-sampling

Brostow & Essa
Other Models: Lens Distortion

- Camera lens bends light, especially at edges
- Common types are barrel and pincushion

Barrel Distortion

Pincushion Distortion
Other Models: Lens Distortion

- Camera lens bends light, especially at edges
- Common types are barrel and pincushion

Barrel Distortion

No Distortion
Traditional Pinhole Camera

- The film sits behind the pinhole of the camera.
Traditional Pinhole Camera

• The film sits behind the pinhole of the camera.
• Rays come in from the outside, pass through the pinhole, and hit the film plane.
Traditional Pinhole Camera

- The film sits behind the pinhole of the camera.
- Rays come in from the outside, pass through the pinhole, and hit the film plane.
Virtual Camera

- The film sits in front of the pinhole of the camera.
Virtual Camera

- The film sits in front of the pinhole of the camera.
- Rays come in from the outside, pass through the film plane, and hit the pinhole.
Virtual Camera

- The film sits in front of the pinhole of the camera.
- Rays come in from the outside, pass through the film plane, and hit the pinhole.

*Photograph is right side up*
Overview

• 3D scene representation

• 3D viewer representation

• Ray Casting
  • What do we see?
  • How does it look?
Ray Casting

- Rendering model
- Intersections with geometric primitives
  - Sphere
  - Triangle
- Acceleration techniques
  - Bounding volume hierarchies
  - Spatial partitions
    - Uniform grids
    - Octrees
    - BSP trees
Ray Casting

- We invert the process of image generation by sending rays **out** from the pinhole, and then we find the first intersection of the ray with the scene.
Ray Casting

- The color of each pixel on the view plane depends on the radiance emanating from visible surfaces.
Ray Casting

- For each sample …
  - Construct ray from eye position through view plane
  - Find first surface intersected by ray through pixel
  - Compute color sample based on surface radiance
Ray Casting

- Simple implementation:

```java
Image RayCast(Camera camera, Scene scene, int width, int height)
{
    Image image = new Image(width, height);
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            Ray ray = ConstructRayThroughPixel(camera, i, j);
            Intersection hit = FindIntersection(ray, scene);
            image[i][j] = GetColor(hit);
        }
    }
    return image;
}
```

- Where are we looking?
- What are we seeing?
- What does it look like?
Constructing a Ray Through a Pixel
Constructing a Ray Through a Pixel

The ray has to originate at $P_0$, the position of the camera. So the equation for the ray is of the form:

$$\text{Ray} = P_0 + tV$$
If the ray passes through the point \( P \), then the solution for \( V \) is:

\[
V = (P - P_0) / ||P - P_0||
\]
If $P$ represents the $(i,j)$-th pixel of the image, what is the position of $P$?
Constructing Ray Through a Pixel

• 2D Example: Side view of camera at $P_0$
  - What is the position of the $i$-th pixel $P[i]$?

$\theta$ = frustum half-angle (given), or field of view
$d$ = distance to view plane (arbitrary = you pick)
Constructing Ray Through a Pixel

- 2D Example: Side view of camera at $P_0$
  - What is the position of the $i$-th pixel $P[i]$?
    - $\theta =$ frustum half-angle (given), or field of view
    - $d =$ distance to view plane (arbitrary = you pick)

$$P_1 = P_0 + d*\text{towards} - d*\tan(\theta)*\text{up}$$
$$P_2 = P_0 + d*\text{towards} + d*\tan(\theta)*\text{up}$$
Constructing Ray Through a Pixel

- 2D Example: Side view of camera at $P_0$

  - What is the position of the $i$-th pixel?

  $\theta =$ frustum half-angle (given), or field of view
  $d =$ distance to view plane (arbitrary = you pick)

  \[ P_1 = P_0 + d \cdot \text{towards} - d \cdot \tan(\theta) \cdot \text{up} \]
  \[ P_2 = P_0 + d \cdot \text{towards} + d \cdot \tan(\theta) \cdot \text{up} \]
  \[ P[i] = P_1 + ((i+0.5)/\text{height}) \cdot (P_2 - P_1) \]
  \[ = P_1 + ((i+0.5)/\text{height}) \cdot 2 \cdot d \cdot \tan(\theta) \cdot \text{up} \]
Constructing Ray Through a Pixel

- 2D Example:
  - The ray passing through the $i$-th pixel is defined by:
  \[
  \text{Ray} = P_0 + tV
  \]

- Where:
  - $P_0$ is the camera position
  - $V$ is the direction to the $i$-th pixel:
    \[
    V = (P[i] - P_0) / \|P[i] - P_0\| 
    \]
  - $P[i]$ is the $i$-th pixel location:
    \[
    P[i] = P_1 + ((i+0.5)/\text{height})*(P_2 - P_1)
    \]
  - $P_1$ and $P_2$ are the endpoints of the view plane:
    \[
    P_1 = P_0 + d\text{towards} - d\text{tan}(\theta)\text{up} \\
    P_2 = P_0 + d\text{towards} + d\text{tan}(\theta)\text{up}
    \]
Ray Casting

- **2D implementation:**

```java
Image RayCast(Camera camera, Scene scene, int width, int height)
{
    Image image = new Image(width, height);
    for (int i = 0; i < height; i++) {
        Ray ray = ConstructRayThroughPixel(camera, i, height);
        Intersection hit = FindIntersection(ray, scene);
        image[i][height] = GetColor(hit);
    }
    return image;
}
```
Constructing Ray Through a Pixel

- Figuring out how to do this in 3D is assignment 2

\[
2d \tan(\theta)
\]
Ray Casting

- Simple implementation:

```java
Image RayCast(Camera camera, Scene scene, int width, int height) {
    Image image = new Image(width, height);
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            Ray ray = ConstructRayThroughPixel(camera, i, j);
            Intersection hit = FindIntersection(ray, scene);
            image[i][j] = GetColor(hit);
        }
    }
    return image;
}
```
Ray Casting

• Simple implementation:

```java
Image RayCast(Camera camera, Scene scene, int width, int height) {
    Image image = new Image(width, height);
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            Ray ray = ConstructRayThroughPixel(camera, i, j);
            Intersection hit = FindIntersection(ray, scene);
            image[i][j] = GetColor(hit);
        }
    }
    return image;
}
```
Ray-Scene Intersection

• Intersections with geometric primitives
  o Sphere
  o Triangle

• Acceleration techniques
  o Bounding volume hierarchies
  o Spatial partitions
    » Uniform (Voxel) grids
    » Octrees
    » BSP trees
Ray-Sphere Intersection

Ray: \( P = P_0 + tV \)
Sphere: \( |P - O|^2 - r^2 = 0 \)
Ray-Sphere Intersection I

Ray: \( P = P_0 + tV \)
Sphere: \(|P - O|^2 - r^2 = 0\)

Substituting for \( P \), we get:
\[ |P_0 + tV - O|^2 - r^2 = 0 \]
Ray-Sphere Intersection I

Ray: $P = P_0 + tV$
Sphere: $|P - O|^2 - r^2 = 0$

Substituting for $P$, we get:
$|P_0 + tV - O|^2 = r^2 - 0$

Solve quadratic equation:
$at^2 + bt + c = 0$

where:
$$a = 1$$
$$b = 2 \, V \cdot (P_0 - O)$$
$$c = |P_0 - O|^2 - r^2 = 0$$
Ray: \( P = P_0 + tV \)
Sphere: \(|P - O|^2 - r^2 = 0\)

Substituting for \( P \), we get:
\(|P_0 + tV - O|^2 - r^2 = 0\)

Solve quadratic equation:
\[at^2 + bt + c = 0\]
where:
\(a = 1\)
\(b = 2 V \cdot (P_0 - O)\)
\(c = |P_0 - O|^2 - r^2 = 0\)

Algebraic Method

Generally, there are two solutions to the quadratic equation, giving rise to points \( P \) and \( P' \).
You want to return the first hit.
Ray-Sphere Intersection II

Ray: \( P = P_0 + tV \)

Sphere: \( |P - O|^2 - r^2 = 0 \)

\( L = O - P_0 \)
Ray-Sphere Intersection II

Ray: $P = P_0 + tV$
Sphere: $|P - O|^2 - r^2 = 0$

$L = O - P_0$

$t_{ca} = L \cdot V$ (assumes $V$ is unit length)
Ray-Sphere Intersection II

Ray: \( P = P_0 + tV \)
Sphere: \( |P - O|^2 - r^2 = 0 \)

\( L = O - P_0 \)

\( t_{ca} = L \cdot V \) (assumes \( V \) is unit length)

\[ d^2 = L \cdot L - t_{ca}^2 \]

if \( (d^2 > r^2) \) return 0
Ray-Sphere Intersection II

Ray: \( P = P_0 + tV \)
Sphere: \( |P - O|^2 - r^2 = 0 \)

\( L = O - P_0 \)

\( t_{ca} = L \cdot V \) (assumes \( V \) is unit length)

\( d^2 = L \cdot L - t_{ca}^2 \)
if \( (d^2 > r^2) \) return 0

\( t_{hc} = \sqrt{r^2 - d^2} \)

\( t = t_{ca} - t_{hc} \) and \( t_{ca} + t_{hc} \)
Ray-Sphere Intersection

- Need normal vector at intersection for lighting calculations

\[ N = \frac{(P - O)}{||P - O||} \]
Ray-Scene Intersection

• Intersections with geometric primitives
  o Sphere
    » Triangle

• Acceleration techniques
  o Bounding volume hierarchies
  o Spatial partitions
    » Uniform grids
    » Octrees
    » BSP trees
Ray-Triangle Intersection

- First, intersect ray with plane
- Then, check if point is inside triangle
Ray-Plane Intersection

Ray: \( P = P_0 + tV \)

Plane: \( P \cdot N + d = 0 \)

Substituting for \( P \), we get:

\[
(P_0 + tV) \cdot N + d = 0
\]

Solution:

\[
t = -\frac{(P_0 \cdot N + d)}{(V \cdot N)}
\]
Ray-Triangle Intersection I

• Check if point is inside triangle algebraically

For each side of triangle
\[ V_1 = T_1 - P_0 \]
\[ V_2 = T_2 - P_0 \]
\[ N_1 = V_2 \times V_1 \]

if \((P - P_0) \cdot N_1 < 0\)
   return FALSE;
end
Ray-Triangle Intersection II

• Check if point is inside triangle parametrically

Every point $P$ inside the triangle can be expressed as:

$$P = T_1 + \alpha (T_2 - T_1) + \beta (T_3 - T_1)$$

where:

$$0 \leq \alpha \leq 1 \quad \text{and} \quad 0 \leq \beta \leq 1$$

$$\alpha + \beta \leq 1$$
Ray-Triangle Intersection II

• Check if point is inside triangle parametrically

Solve for $\alpha$, $\beta$ such that:

$P = T_1 + \alpha (T_2-T_1) + \beta (T_3-T_1)$

Check if point inside triangle.

$0 \leq \alpha \leq 1$ and $0 \leq \beta \leq 1$

$\alpha + \beta \leq 1$
Other Ray-Primitive Intersections

- Cone, cylinder, ellipsoid:
  - Similar to sphere

- Box
  - Intersect 3 front-facing planes, return closest

- Convex polygon
  - Same as triangle (check point-in-polygon algebraically)

- Concave polygon
  - Same plane intersection
  - More complex point-in-polygon test
Ray-Scene Intersection

• Intersections with geometric primitives
  o Sphere
  o Triangle

• Acceleration techniques
  o Bounding volume hierarchies
  o Spatial partitions
    » Uniform grids
    » Octrees
    » BSP trees
Ray-Scene Intersection

- Find intersection with front-most primitive in group

Intersection FindIntersection(Ray ray, Scene scene) {
    min_t = \infty
    min_shape = NULL
    For each primitive in scene {
        t = Intersect(ray, primitive);
        if (t > 0 and t < min_t) then
            min_shape = primitive
            min_t = t
    }
    return Intersection(min_t, min_shape)
}
Ray-Scene Intersection

• Intersections with geometric primitives
  o Sphere
  o Triangle

  » Acceleration techniques
    o Bounding volume hierarchies
    o Spatial partitions
      » Uniform grids
      » Octrees
      » BSP trees
Acceleration Techniques

• A direct approach tests for an intersection of every ray with every primitive in the scene.

• Acceleration techniques:
  - Grouping:
    Group primitives together and test if the ray intersects the group. If it doesn’t, don’t test individual primitives.
  - Ordering:
    Test primitives/groups based on their distance along the ray. If you find a close hit, don’t test distant primitives/groups.
Bounding Volumes

- Check for intersection with the bounding volume:
  - Bounding cubes
  - Bounding boxes
  - Bounding spheres
  - Etc.

Stuff that’s easy to intersect
Bounding Volumes

• Check for intersection with the bounding volume
Bounding Volumes

- Check for intersection with the bounding volume
  - If ray doesn’t intersect bounding volume, then it doesn’t intersect its contents
Bounding Volumes

• Check for intersection with the bounding volume
  - If ray doesn’t intersect bounding volume, then it doesn’t intersect its contents

Still need to check for intersections with shape.
Bounding Volume Hierarchies

- Build hierarchy of bounding volumes
  - Bounding volume of interior node contains all children

![Bounding Volume Hierarchies Diagram]
Bounding Volume Hierarchies

- Grouping acceleration

```java
FindIntersection(Ray ray, Node node) {
    min_t = ∞
    min_shape = NULL
    // Test if you intersect the bounding volume
    if( !intersect ( node.boundingVolume ) ) {
        return (min_t, min_shape);
    }
    // Test the children
    for each child {
        (t, shape) = FindIntersection(ray, child)
        if (t < min_t) {min_shape=shape}
    }
    return (min_t, min_shape);
}
```
Bounding Volume Hierarchies

- Use hierarchy to accelerate ray intersections
  - Intersect node contents only if hit bounding volume
Bounding Volume Hierarchies

• Use hierarchy to accelerate ray intersections
  - Intersect node contents only if hit bounding volume

• Don’t need to test shapes A or B
• Need to test groups 1, 2, and 3
• Need to test shapes C, D, E, and F
Bounding Volume Hierarchies

- Grouping + Ordering acceleration

```c
FindIntersection(Ray ray, Node node) {
    // Find intersections with child node bounding volumes
    ...
    // Sort intersections front to back
    ...
    // Process intersections (checking for early termination)
    min_t = ∞
    min_shape = NULL
    for each intersected child {
        if (min_t < bv_t[child]) break;
        (t, shape) = FindIntersection(ray, child);
        if (t < min_t) {
            min_t = t
            min_shape = shape
        }
    }
    return (min_t, min_shape);
}
```
Bounding Volume Hierarchies

- Use hierarchy to accelerate ray intersections
  - Intersect nodes only if you haven’t hit anything closer
Bounding Volume Hierarchies

- Use hierarchy to accelerate ray intersections
  - Intersect nodes only if you haven’t hit anything closer

- Don’t need to test shapes A, B, D, E, or F
- Need to test groups 1, 2, and 3
- Need to test shape C
Ray-Scene Intersection

• Intersections with geometric primitives
  o Sphere
  o Triangle

» Acceleration techniques
  o Bounding volume hierarchies
    o Spatial partitions
      » Uniform (Voxel) grids
      » Octrees
      » BSP trees
Uniform (Voxel) Grid

- Construct uniform grid over scene
  - Index primitives according to overlaps with grid cells

- A primitive may belong to multiple cells
- A cell may have multiple primitives
Uniform (Voxel) Grid

• Trace rays through grid cells
  o Fast
  o Incremental

Only check primitives in intersected grid cells
Uniform (Voxel) Grid

• Potential problem:
  - How choose suitable grid resolution?

Too much cost if grid is too fine

Too little benefit if grid is too coarse
Ray-Scene Intersection

- Intersections with geometric primitives
  - Sphere
  - Triangle

  - Acceleration techniques
    - Bounding volume hierarchies
    - Spatial partitions
      - Uniform (Voxel) grids
      - Octrees
      - BSP trees
Octrees

- We can think of a voxel grid as a tree.
  - The root node is the entire region.
  - Each node has eight children obtained by subdividing the parent into eight equal regions.
Octrees

- We can think of a voxel grid as a tree.
  - The root node is the entire region
  - Each node has eight children obtained by subdividing the parent into eight equal regions
Octrees

- We can think of a voxel grid as a tree.
  - The root node is the entire region
  - Each node has eight children obtained by subdividing the parent into eight equal regions
Octrees

- We can think of a voxel grid as a tree.
  - The root node is the entire region
  - Each node has eight children obtained by subdividing the parent into eight equal regions
Octrees

• In an octree, we only subdivide regions that contain more than one shape.
Octrees

- In an octree, we only subdivide regions that contain more than one shape.
Octrees

- In an octree, we only subdivide regions that contain more than one shape.
Octrees

• In an octree, we only subdivide regions that contain more than one shape.
Octrees

• In an octree, we only subdivide regions that contain more than one shape.
Octrees

• In an octree, we only subdivide regions that contain more than one shape.

• Adaptively determines grid resolution.
Ray-Scene Intersection

- Intersections with geometric primitives
  - Sphere
  - Triangle

- Acceleration techniques
  - Bounding volume hierarchies
  - Spatial partitions
    - Uniform (Voxel) grids
    - Octrees
    - BSP trees
Binary Space Partition (BSP) Tree

- Recursively partition space by planes
Binary Space Partition (BSP) Tree

- Recursively partition space by planes
  - Generate a tree structure where the leaves store the shapes.
Binary Space Partition (BSP) Tree

- Recursively partition space by planes
  - Generate a tree structure where the leaves store the shapes.
Binary Space Partition (BSP) Tree

• Recursively partition space by planes
  - Generate a tree structure where the leaves store the shapes.
Binary Space Partition (BSP) Tree

- Recursively partition space by planes
  - Generate a tree structure where the leaves store the shapes.
Binary Space Partition (BSP) Tree

- Recursively partition space by planes
  - Generate a tree structure where the leaves store the shapes.
Binary Space Partition (BSP) Tree

• Recursively partition space by planes
  o Every cell is a convex polyhedron
Binary Space Partition (BSP) Tree

- Example: Point Intersection
Binary Space Partition (BSP) Tree

- Example: Point Intersection
  - Recursively test what side we are on
Binary Space Partition (BSP) Tree

- Example: Point Intersection
  - Recursively test what side we are on
    » Left of 1 (root) → 2
Binary Space Partition (BSP) Tree

- Example: Point Intersection
  - Recursively test what side we are on
    » Left of 2 → 4
Binary Space Partition (BSP) Tree

- Example: Point Intersection
  - Recursively test what side we are on
    - Right of 4 → Test B
Binary Space Partition (BSP) Tree

- Example: Point Intersection
  - Recursively test what side we are on
    » Missed B. No intersection!
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 1
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 1
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    » Test half to the left of 1
Binary Space Partition (BSP) Tree

• Example: Ray Intersection 1
  o Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    » Test half to the right of 2
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 1
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    » Intersection with C. Done!

```
A
B
C
D
E
F
1
2
3
4
5
```
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    » Test half to the left of 1
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    » Test half to the right of 2
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    - Missed C. Recurse!
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    » Test half to left of 2
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    » Test half to left of 4
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    » Missed A. Recurse!
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    - No half to right of 4.

```
  1
 / \
2   3
\   \\
   4
   \
   5

A   B
  \\
D   E
  \\
F
  \\
C
```
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    » Test half to right of 1
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    » Test half to left of 3
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    » Intersection with D. Done!
RayTreeIntersect(Ray ray, Node node, double min, double max) {
    if (Node is a leaf)
        return intersection of closest primitive in cell, or NULL if none
    else
        // Find splitting point
        dist = distance along the ray point to split plane of node

        // Find near and far children
        near_child = child of node that contains the origin of Ray
        far_child = other child of node

        // Recurse down near child first
        if the interval to look is on near side {
            isect = RayTreeIntersect(ray, near_child, min, max)
            if(isect) return isect  // If there’s a hit, we are done
        }

        // If there’s no hit, test the far child
        if the interval to look is on far side
            return RayTreeIntersect(ray, far_child, min, max)
}
Acceleration

• Intersection acceleration techniques are important
  o Bounding volume hierarchies
  o Spatial partitions

• General concepts
  o Sort objects spatially
  o Make trivial rejections quick

Expected time is sub-linear in number of primitives
Summary

- Writing a simple ray casting renderer is easy
  - Generate rays
  - Intersection tests
  - Lighting calculations

```java
Image RayCast(Camera camera, Scene scene, int width, int height)
{
    Image image = new Image(width, height);
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            Ray ray = ConstructRayThroughPixel(camera, i, j);
            Intersection hit = FindIntersection(ray, scene);
            image[i][j] = GetColor(hit);
        }
    }
    return image;
}
```
Next Time is Illumination!

Without Illumination

With Illumination