Questions from Last Week?
- Color models
- Light models
- Phong shading model
- Assignment 2

Ray/Plane Intersection
Ray is defined by \( R(t) = R_o + R_d \cdot t \) where \( t \geq 0 \)
- \( R_o \) = Origin of ray at \((x_o, y_o, z_o)\)
- \( R_d \) = Direction of ray \([x_d, y_d, z_d]\) (unit vector)

Plane is defined by \([A, B, C, D]\)
- \( Ax + By + Cz + D = 0 \) for a point in the plane
- Normal Vector, \( N = [A, B, C] \) (unit vector)
- \( A^2 + B^2 + C^2 = 1 \)
- \( D = -N \cdot P_o \) (\( P_o \) - point in plane)

Ray/Plane (cont.)
Substitute the ray equation into the plane equation:
\[
A(x_o + x_d \cdot t) + B(y_o + y_d \cdot t) + C(z_o + z_d \cdot t) + D = 0
\]
Solve for \( t \):
\[
t = -\frac{(Ax_o + By_o + Cz_o + D)}{(Ax_d + By_d + Cz_d)}
\]
\[
t = -\frac{(N \cdot R_o - N \cdot P_o)}{(N \cdot R_d)}
\]
What Can Happen?

Ray/Plane Summary

Intersection point:
\[(x_i, y_i, z_i) = (x_o + x_d t_i, y_o + y_d t_i, z_o + z_d t_i)\]

1. Calculate \(N \cdot R_d\) and compare it to zero.
2. Calculate \(t_i\) and compare it to zero.
3. Compute intersection point.
4. Flip normal if \(N \cdot R_d\) is positive

Ray-Parallelepiped Intersection

- Axis-aligned
- From \((X_1, Y_1, Z_1)\) to \((X_2, Y_2, Z_2)\)
- Ray \(P(t) = R_o + R_d t\)

Naïve ray-box Intersection

- Use 6 plane equations
- Compute all 6 intersection
- Check that points are inside box
  \[Ax + By + Cz + D \leq 0\]

Factoring out computation

- Pairs of planes have the same normal
- Normals have only one non-0 component
- Do computations one dimension at a time
- Maintain \(t_{near}\) and \(t_{far}\) (closest and farthest so far)

Test if parallel

- If \(R_o \times 0\), then ray is parallel
  - if \(R_o \times X_1\) or \(R_o \times x_2\) return false
If not parallel

- Calculate intersection distance $t_1$ and $t_2$
  - $t_1 = (X_1 - R_x)/R_d$
  - $t_2 = (X_2 - R_x)/R_d$

- $Y_1 = \text{Y}$
- $Y_2 = \text{Y}$
- $X_1 = \text{X}$
- $X_2 = \text{X}$

Test 1

- Maintain $t_{\text{near}}$ and $t_{\text{far}}$
  - If $t_1 > t_2$, swap
  - If $t_1 > t_{\text{near}}$, $t_{\text{near}} = t_1$
  - If $t_2 < t_{\text{far}}$, $t_{\text{far}} = t_2$
  - If $t_{\text{near}} > t_{\text{far}}$, box is missed

Test 2

- If $t_{\text{far}} < 0$, box is behind

Algorithm recap

- Do for all 3 axes
  - Calculate intersection distance $t_1$ and $t_2$
  - Maintain $t_{\text{near}}$ and $t_{\text{far}}$
    - If $t_{\text{near}} > 0$, box is missed; Done
    - If $t_{\text{far}} < 0$, box is behind; Done
  - If box survived tests, return intersection at $t_{\text{near}}$
  - If $t_{\text{near}}$ is negative, return $t_{\text{far}}$

Motivation

- Distributed Ray Tracing
- Soft shadows
- Anti-aliasing (getting rid of jaggies)
- Glossy reflection
- Motion blur
- Depth of field (focus)

Extra rays needed for these effects
Shadows
- one shadow ray per intersection per point light source

Soft Shadows
- multiple shadow rays to sample area light source

Antialiasing – Supersampling
- multiple rays per pixel

Reflection
- one reflection ray per intersection

Glossy Reflection
- multiple reflection rays

Motion Blur
- Sample objects temporally
Depth of Field
- multiple rays per pixel

Algorithm Analysis
- Ray casting
- Lots of primitives
- Recursive
- Distributed Ray Tracing Effects
  - Soft shadows
  - Anti-aliasing
  - Glossy reflection
  - Motion blur
  - Depth of field

Cost: \[ \text{height} \times \text{width} \times \text{num primitives} \times \text{intersection cost} \times \text{num shadow rays} \times \text{supersampling} \times \text{num glossy rays} \times \text{num temporal samples} \times \text{max recursion depth} \times \ldots \]

Bounding Regions

Acceleration of Ray Casting
- Goal: Reduce the number of ray/primitive intersection tests

Conservative Bounding Region
- First check for an intersection with a conservative bounding region
- Early reject

Ray Tracing Acceleration Techniques
- Faster Intersections
- Fewer Rays
- Generalized Rays
  - Object bounding volumes
  - Efficient intersection routines
  - Adaptive tree depth
  - Statistical optimization
  - Beam tracing
  - Cone tracing
  - Pencil tracing

**Regions**
- Tight (avoid false positives)
- Fast to compute
- Fast to intersect

**Bounding Volumes**
- **What makes a “good” bounding volume?**
  - Tightness of fit (expressed how?)
  - Easy to compute
  - Simplicity of intersection
- **Total cost** = \( b \cdot \# \text{times volume tested for intersection} + i \cdot \# \text{times item is tested for intersection} + f \cdot \# \text{cost of ray-item intersection test} + S \cdot \# \text{cost to compute BV parameters} \)

**Bounding Volumes**
- Spheres
  - Cheap intersection test
  - Poor fit
  - Somewhat costly to fit to data

**Bounding Volumes**
- **Axis-aligned bounding boxes (AABBs)**
  - Relatively cheap intersection test
  - Usually better fit
  - Trivial to fit to data

**Bounding Volumes**
- **Arbitrary convex region (bounding half-spaces)**
- **Non-aligned bounding box**
- **Bounding sphere**

**Bounding Volumes**
- **Axis-aligned bounding box**
- **Bounding box**

**Bounding Volumes**
- **Really expensive object**
- **Ray through this pixel only tests the bounding box**

**Bounding Volumes**
- **Scene**
- **Bounding box**

**Bounding Volumes**
- **Image**
- **Ray through this pixel tests the bounding box and intersects the object**

**Bounding Volumes**
- **Bounding box slows down rendering**

**Bounding Volumes**
- **Bounding sphere**

**Bounding Volumes**
- **Axis-aligned bounding box**

**Bounding Volumes**
- **Image**

**Bounding Volumes**
- **Ray through this pixel only tests the bounding box**

**Bounding Volumes**
- **Really expensive object**

**Bounding Volumes**
- **Scene**

**Bounding Volumes**
- **Bounding box**

**Bounding Volumes**
- **Image**

**Bounding Volumes**
- **Ray through this pixel tests the bounding box and intersects the object**

**Bounding Volumes**
- **Bounding box slows down rendering**

**Bounding Volumes**
- **Bounding sphere**

**Bounding Volumes**
- **Axis-aligned bounding box**
Bounding Volumes
- Oriented bounding boxes (OBBs)
  - Medium-expensive intersection test
  - Very good fit (asymptotically better)
  - Medium-difficult to fit to data

Bounding Volumes
- Slabs (parallel planes)
  - Comparatively expensive
  - Very good fit
  - Very difficult to fit to data

Intersection with Axis-Aligned Box
- For all 3 axes, calculate the intersection distances \( t_1 \) and \( t_2 \)
- \( t_{\text{near}} = \max(t_{x1}, t_{x2}, t_{x3}) \)
- \( t_{\text{far}} = \min(t_{x1}, t_{x2}, t_{x3}) \)
- If \( t_{\text{near}} > t_{\text{far}} \), box is missed
- If \( t_{\text{far}} < 0 \), box is behind
- If box survived tests, report intersection at \( t_{\text{near}} \)

Bounding Box of a Triangle

Bounding Box of a Sphere

Bounding Box of a Group
Spatial Data Structures

- Spatial partitioning techniques classify all space into non-overlapping portions
- Easier to generate automatically
- Can “walk” ray from partition to partition
- Hierarchical bounding volumes surround objects in the scene with (possibly overlapping) volumes
- Often tightest fit

Spatial Partitioning

- Some spatial partitioning schemes:
  - Regular grid (2-D or 3-D)
  - Octree
  - k-D tree
  - BSP-tree

Regular Grid

- Find bounding box of scene
- Choose grid spacing
- grid, need not = grid,

Create grid

- Cell (i, j)
Insert primitives into grid
- Primitives that overlap multiple cells?
- Insert into multiple cells (use pointers)

50

For each cell along a ray
- Does the cell contain an intersection?
- Yes: return closest intersection
- No: continue

51

Preventing repeated computation
- Perform the computation once, "mark" the object
- Don't re-intersect marked objects

54

Don't return distant intersections
- If intersection $t$ is not within the cell range, continue (there may be something closer)

52

Where do we start?
- Intersect ray with scene bounding box
- Ray origin may be inside the scene bounding box

53

Is there a pattern to cell crossings?
- Yes, the horizontal and vertical crossings have regular spacing

54
What's the next cell?

```plaintext
if t_{next_v} < t_{next_h}:
    i += sign_x
    t_{next} = t_{next_v}
    t_{next_v} += dt_v
else:
    j += sign_y
    t_{next} = t_{next_h}
    t_{next_h} += dt_h
```

Pseudo-code

```plaintext
create grid
insert primitives into grid
for each ray r:
    find initial cell c(i,j), t_{min}, t_{next_v} & t_{next_h}
    compute dt_v, dt_h, sign_x and sign_y
    while c != NULL:
        for each primitive p in c:
            intersect r with p
            if intersection in range found:
                return
        c = find next cell
```

Regular Grid Discussion

- Advantages?
  - easy to construct
  - easy to traverse

- Disadvantages?
  - may be only sparsely filled
  - geometry may still be clumped

Acceleration Spatial Data Structures

- Adaptive Grids
  - Subdivide until each cell contains no more than n elements, or maximum depth d is reached

Adaptive Grids

- Nested Grids
- Octree/Quadtree
Primitives in an Adaptive Grid
- Can live at intermediate levels, or be pushed to lowest level of grid

Octree/(Quadtree)

Adaptive Grid Discussion
- Advantages?
  - grid complexity matches geometric density
- Disadvantages?
  - more expensive to traverse (especially octree)

k-D Trees
- k-D tree pros:
  - Moderately simple to generate
  - More adaptive than octrees
- k-D tree cons:
  - Less efficient to trace rays across
  - Moderately complex data structure

k-D tree

BSP Trees
- BSP tree pros:
  - Extremely adaptive
  - Simple & elegant data structure
- BSP tree cons:
  - Very hard to create optimal BSP
  - Splitting planes can explode storage
  - Simple but slow to trace rays across

BSP Tree

Acceleration Spatial Data Structures

Bounding Volume Hierarchy

Bounding Volume Hierarchy
- What makes a “good” bounding volume hierarchy?
  - Grouped objects (or volumes) should be near each other
  - Volume should be minimal
  - Sum of all volumes should be minimal
  - Top of the tree is most critical
  - Constructing the hierarchy should pay for itself!
Bounding Volume Hierarchy

- Find bounding box of objects
- Split objects into two groups
- Recurse

Where to split objects?

- At midpoint OR
- Sort, and put half of the objects on each side OR
- Use modeling hierarchy
**Data Structure Pseudo-code**

```c
sort_axis = 0;
Make_BVH(object_list, sort_axis, ptr);
struct.bbox = BoundingBox(object_list);
If # of objects < Threshold
struct.obj_list = object_list
Else
If (sort_axis % 3) == 0 Sort object centroids in X
ElseIf (sort_axis % 3) == 1 Sort object centroids in Y
Else Sort object centroids in Z
Split sorted list into two halves
Make_BVH(left_obj_list, sort_axis++, lptr)
Make_BVH(right_obj_list, sort_axis++, rptr)
struct.lptr = lptr; struct.rptr = rptr;
ptr = &struct;
Return
```

**Intersection with BVH**

- Check subvolume with closer intersection first

**Intersection with BVH**

- Don't return intersection immediately if the other subvolume may have a closer intersection

**Intersection Pseudo-code**

```c
Does ray intersect box?
intersect_BVH(box, ray, xsect_pt, t)
If no more subboxes
Intersect geometry and return nearest xsect_pt & t
Intersect ray with both subboxes
No hits: return xsect_pt = Null, t
Sort t's
Call subbox of nearest t subbox
intersect_BVH(subbox, ray, xsect_pt, t)
If hit subbox?
If xsect_pt == Null || tnew < t old
intersect_BVH(subbox2, ray, xsect_pt, t)
Set nearest xsect_pt and t
Return
```

**Bounding Volume Hierarchy Discussion**

- **Advantages**
  - easy to construct
  - easy to traverse
  - binary

- **Disadvantages**
  - may be difficult to choose a good split for a node
  - poor split may result in minimal spatial pruning

- **Hint**
  - Alternate sorting in X, Y & Z

**Transformation Hierarchy**

- Group & Transformation hierarchy may not be a good spatial hierarchy

---

1. Data Structure Pseudo-code
2. Intersection with BVH
3. Intersection with BVH
4. Intersection Pseudo-code
5. Bounding Volume Hierarchy Discussion
6. Transformation Hierarchy
What’s the best method?

- What kind of scene are you rendering?
  - Teapot in a stadium vs. uniform distribution
  - Impact on surface tessellation on distribution
- Parameter values are critical

Shoot Fewer Rays

- Adaptive depth control
  - Naïve ray tracer: spawn 2 rays per intersection until max recursion limit
  - In practice, few surfaces are transparent or reflective
  - Stop shadow ray at first intersection between start and light source
  - Just shoot the rays you need
  - Determine contribution of ray
    - Don’t shoot rays w/ contribution near 0%

Shoot Fewer Rays

- Adaptive sampling
  - Shoot rays coarsely, interpolating their values across pixels
  - Where adjacent rays differ greatly in value, sample more finely
  - Stop when some maximum resolution is reached

Generalized Rays

- Beams, cones, pencils
- Area sampling, rather than point sampling
- Geometric computations are tricky (expensive?)
- Problems with reflection/refractions

Wrap Up

- Discuss next programming assignment
  - Add an acceleration technique
    - Adaptive grid
    - Bounding volume hierarchy
  - Supersample image
- Discuss status/problems/issues with this week’s programming assignment