Program 2

- Run Program 1 in simhwrt
- Also run Program 2 and include that output
- Lab time?

- Inheritance probably doesn’t work
Boxes

- Axis aligned boxes
- Parallelepiped
- 12 triangles?
- 6 planes with squares?
Ray-box intersection

\[
\vec{N} \cdot \vec{P} - d = 0
\]

\[
t = \frac{d - \vec{N} \cdot \vec{O}}{\vec{N} \cdot \vec{V}}
\]

x plane: \( \vec{N} = [1 \ 0 \ 0] \)

\[
t = \frac{d - O_x}{V_x}
\]

\( d_1 = P_{1x}, d_2 = P_{2x} \)

\[
t_{x1} = \frac{P_{1x} - O_x}{V_x}, t_{x2} = \frac{P_{2x} - O_x}{V_x}
\]

Same for y, z planes
Intersection of intervals

Intersection occurs where x interval overlaps y interval

x interval: \( \min(t_{x1}, t_{x2}) \leq t \leq \max(t_{x1}, t_{x2}) \)

y interval: \( \min(t_{y1}, t_{y2}) \leq t \leq \max(t_{y1}, t_{y2}) \)

intersection: \( \max(\min_x, \min_y) \leq t \leq \min(\max_x, \max_y) \)

In 3D also check z interval
Improved Box

- http://www.cs.utah.edu/~awilliam/box/
Ray tracing optimization

- Faster rays
- CPU optimization techniques (CS6620 lecture)
- Fewer rays
  - Adaptive supersampling
  - Ray tree pruning
- Faster ray-object intersection tests
- High-order surfaces
- Fewer ray-object intersection tests
- Acceleration structures

Adapted from Arvo/Kirk “A survey of ray tracing acceleration techniques”
Acceleration structures

- Current ray tracer is $O(\# \text{ objects})$
- Large # of objects can be SLOW

- Solution: intelligently determine which objects to intersect
- Good news: $<<O(N)$ achievable!
- Real-time ray tracer exploits this to do very large models:
  - 262144x262144 heightfield (10+ billion patches)
  - 35 million spheres
  - (1.1 million spheres + volume rendering) * 170 timesteps
Acceleration structures

• Three main types

• Tree-based:
  • Binary Space Partitioning (BSP) tree
  • Bounding Volume Hierarchy

• Grid-based:
  • Uniform grid
  • Hierarchical grid
  • Octree

• Directional
Uniform grid

- Split space up in a grid
- A heightfield is a specialized acceleration structure
- A grid is more general
Grid traversal

- Just like a 2D Grid but in 3D
Heightfield traversal

- Step 1: Compute a few derived values
  - Diagonal: $D = P_{\text{max}} - P_{\text{min}}$
  - Cell Size: $\text{cellsize} = D / (nx, ny, 1)$
  - Data min, max: $Z_{\text{min}}, Z_{\text{max}}$
- Grid:
  - Add nz
  - Cell size 3D
  - Don’t need $Z_{\text{min}}/Z_{\text{max}}$
  - Data located in cells, not at corners
Grid traversal

- Step 2-8: straightforward extension to 3D
Heightfield traversal

- Step 2: Compute $t_{\text{near}}$
- Use ray-box intersection
- Be careful with rays that begin inside (set $t_{\text{near}}=0$)

\[ \text{Pmin} \rightarrow t_{\text{near}} \rightarrow \text{Pmax} \]
Heightfield traversal

- Step 3: Compute lattice coordinates of near point
  - World space: $P = O + t_{near} V$
  - Lattice space: $L = \text{int}((P - P_{\text{min}}) / \text{cellsize})$
- Be careful of
  - Roundoff error
Heightfield traversal

• Step 4: Determine how ray marching changes index
  • \( \text{diy} = \text{D.y} \times \text{V.y} > 0?1:-1 \)
  • \( \text{stopy} = \text{D.y} \times \text{V.y} > 0?\text{yres}:-1 \)
  • similar for \( x \)
Heightfield traversal

- Step 5: Determine how t value changes with ray marching:
  - \( dtdx = \text{Abs}(\text{cellsize}.x/V.x) \)
  - \( dtdy = \text{Abs}(\text{cellsize}.y/V.y) \)
Heightfield traversal

- Step 6: Determine the far edges of the cell
  - if dix == 1:
    - far.x = (L.x + 1) * cellsize.x + Pmin
  - if dix == -1:
    - far.x = L.x * cellsize.x + Pmin
- Similar for y
Heightfield traversal

- Step 7: Determine $t$ value of far slabs
  - $t_{next \_x} = (\text{far} \_x - O \_x)/V \_x$
  - $t_{next \_y} = (\text{far} \_y - O \_y)/V \_y$
Heightfield traversal

- Step 8: Beginning of loop Compute range of Z values
  - zenter = O.z + tnear * V.z
  - texit = Min(next_x, next_y)
  - zexit = O.z + texit * V.z
Heightfield traversal

- Step 8: Beginning of loop Compute range of Z values
  - $z_{\text{center}} = O.z + t_{\text{near}} \cdot V.z$
  - $t_{\text{exit}} = \text{Min}(\text{next}_x, \text{next}_y)$
  - $z_{\text{exit}} = O.z + t_{\text{exit}} \cdot V.z$

- Grid: not needed
Heightfield traversal

- Step 9: Determine overlap of z range
  - datamin = Min(data[L.x][L.y],
    data[L.x+1][L.y],
    data[L.x][L.y+1],
    data[L.x+1][L.y+1])
  - zmin = Min(zenter, zexit)
  - Similar for max
  - if zmin > datamax || zmax<datamin
    - skip to step 11
  - Grid: not needed (skip cell if no objects in cell)
Step 10:

- Intersect ray with cell
- 2 triangles, 4 triangles, Bilinear patch, Bicubic patch

- Grid: intersect with list of objects that partially overlap cell
Heightfield traversal

- Step 11: March to next cell
- if $\text{tnext}_x < \text{tnext}_y$
  - $\text{tnear} = \text{tnext}_x$
  - $\text{tnext}_x += \text{dtdx}$
  - $L.x += \text{dix}$
- else
  - Similar for $y$
- Grid: 3 way minimum
Heightfield traversal

- Step 12: Decide if it is time to stop
  - Stop if hit the surface at \( t > \epsilon \)
  - Stop if \( L.x = stop_x \) or \( L.y = stop_y \)
  - Stop if \( t_{near} > t_{far} \)
    - Otherwise, back to step 8

- Grid:
  - Cannot stop if you find a hit!
  - Stop if \( hit.minT < new t_{near} \)
  - Stop if \( L.x = stop_x \) or \( L.y = stop_y \) or \( L.z = stop_z \)
  - \( t_{near} > t_{far} \) condition is redundant
Stopping example

- When intersecting cell 1, ray finds yellow triangle, t outside of cell

- Proceed to next cell and intersect again, ray finds blue circle with smaller t

- Second example shows the opposite
Extra work

- This example highlights a common grid problem: redundant intersections
- Ray can intersect yellow triangle up to three times!
- Worst case: ground polygon
- Gets worse with increased grid size
Avoiding extra work

- Don’t worry about it? (not always effective)
- Mailbox: store unique ray ID with each object, don’t call intersect if ray ID == last ray ID for object (NOT thread safe!)
- Remember last N intersected objects, don’t re-intersect (extra overhead)
- Hierarchical grid
Building a Grid
Building a grid

• Add two methods to your object class:
  – Required:
    // Get the bounding box for this object
    // Expand the input bounding box
    void getBounds(BoundingBox& bbox);
  – Optional:
    // Does the object intersect this box?
    // always return true if you aren’t sure
    bool intersects(BoundingBox& cell_bbox);

• Methods to find these later
Building a grid

Foreach object:
- Compute bounding box
- Transform extents to index space (careful with rounding)

Rounding:
- Lower: round down
- Upper: round up
Building a grid

Foreach object:
  Compute bounding box
  Transform extents to index space (careful with rounding)
  3D loop lx, ly, lz to hx, hy, hz:
    Add object to each cell

Loop over green area
More efficient

Foreach object:
  Compute bounding box
  Transform extents to index space (careful with rounding)
  3D loop lx, ly, lz to hx, hy, hz:
    If(object intersects cell boundary)
      Add object to each cell

Blue cells won’t get added with an additional check
Still more efficient

• Two pass algorithm:
  – First pass: count objects in each cell
  – Allocate memory all at once
  – Second pass: insert objects into list

• Huge memory savings over linked lists (2X to 8X or more)

• Huge memory coherence improvement
Other improvements

• Memory tiling (or bricking in 3D)
• Pseudo-tiling of contiguous object lists
• Hierarchical:
  – Objects only at bottom levels
  – Objects mixed throughout the tree
  – Lots of variations
• Octree:
  – Theoretically optimal
  – BUT traversal across grid is much faster than up/down grid
Grid summary

- Grids work very well for objects of uniform size
- Should be easy if you understood the heightfield traversal
- Build is straightforward

- Possibly large memory requirements
- Grid spacing requires tuning (tradeoff memory consumption and redundant intersections vs. efficiency)
Bounding primitives

- Optimize intersections
- Enclose expensive objects in a simpler primitive
- If a ray misses the bounds, it misses the object
Inner bounds

- Can also be used to know that a ray hits a particular object
- Only good for shadows
- Rarely used
Bounding primitives

- Tradeoff: tight bounds vs. intersection speed
- Box
- Spheres
- Ellipsoids
- Slabs

- Box is most common
- Depends on the relative costs of intersection
Bounding boxes

- **Box:** easy
- **bounds** = Min(p1, p2), Max(p1, p2)

- **Sphere:**
- **bounds** = C-Vector(radius, radius, radius)
- C+Vector(radius, radius, radius)
Bounding Boxes

- Triangle:
  - bounds = Min(p1, p2, p3), Max(p1, p2, p3)

- Plane
  - infinite bounds
  - You may want to consider removing planes from your renderer at this point
  - Or keep them outside of your accel structures
Bounding boxes

- Disc/Ring:
  
  \[ C_x \pm rad \sqrt{N_y^2 + N_z^2} \]

  \[ \| \vec{N} \| = 1 \]

  Similar for y/z

- Group:

- Union of object bounding boxes

- min of mins

- max of maxs
Uses for bounding boxes

- Quick reject for expensive primitives
- To fill in grid cells
- Directly in Bounding Volume Hierarchy or similar
Bounding Volume Hierarchy

- Observe that intersection is like a search
- We know that searching is $O(n)$ for unsorted lists but $O(\log n)$ for sorted lists
- How do we sort objects from all directions simultaneously?
Bounding volume hierarchy

- Organize objects into a tree
- Group objects in the tree based on spatial relationships
- Each node in the tree contains a bounding box of all the objects below it
BVH traversal

- At each level of the tree, intersect the ray with the bounding box
- miss: ray misses the entire subtree
- hit: recurse to both children
BVH traversal

- At each level of the tree, intersect the ray with the bounding box
  - miss: ray misses the entire subtree
  - hit: recurse to both children
BVH traversal

- At each level of the tree, intersect the ray with the bounding box
  - miss: ray misses the entire subtree
  - hit: recurse to both children
BVH traversal

- At each level of the tree, intersect the ray with the bounding box
  - miss: ray misses the entire subtree
  - hit: recurse to both children
BVH optimizations

- Stop if the current T value is closer than the BVH node
BVH optimizations

- Stop if the current T value is closer than the BVH node
- Traverse down side of tree that is closer to origin of the ray first
• Stop if the current T value is closer than the BVH node

• Traverse down side of tree that is closer to origin of the ray first

• Three or more way split
Building a BVH

- Determining optimal BVH structure is NP-hard problem
- Heuristic approaches:
  - Cost models (minimize volume or surface area)
  - Spatial models
- Categories of approaches:
  - Top down
  - Bottom up
Median cut BVH construction

- Top down approach:
  - Sort objects by position on axis
    - cycle through x,y,z
    - use center of bounding box
  - Insert tree node with half of objects on left and half on right
Weghorst BVH construction

- Bottom up construction
- Add objects one at a time to tree
- Insert to subtree that would cause smallest increase to area
Weghorst BVH construction

- Bottom up construction
- Add objects one at a time to tree
- Insert to subtree that would cause smallest increase to area
Weghorst BVH construction

- Bottom up construction
- Add objects one at a time to tree
- Insert to subtree that would cause smallest increase in area
Weghorst BVH construction

- Bottom up construction
- Add objects one at a time to tree
- Insert to subtree that would cause smallest increase to area
**k-d tree**

- Recursively divide space in half
- Alternate coordinate axes
- Cycle through axes or store axis split in each node

- What do you do with objects split by the plane?
- Hard part: where do you split in each dimension?
BSP tree

- Like a k-d tree where splitting planes can be arbitrarily located

- Hard part: where do you split in each dimension?

- Harder part: how do you orient each plane?

- More storage/computation, tighter bounds
BSP/k-d tree tradeoffs

- Build is NP-hard problem
- Heuristic approaches are always used
- Traversal is quick
- Storage can be lower than BVH or grid
K-d traversal

- Keep track of intervals on ray:
  - min: 0
  - max: $\infty$

- Compute $t$ of split plane
- If $t > \text{max}$: goto near child
- If $t < \text{min}$: goto far child
- Otherwise: goto both children with updated intervals
K-d traversal

- Keep track of intervals on ray:
  - Compute t of split plane
  - If t > max: goto near child
  - If t < min: goto far child
  - Otherwise: goto both children with updated intervals

Min: 0
Max: ∞
K-d traversal

- Keep track of intervals on ray:
  - Compute $t$ of split plane
  - If $t > \text{max}$: goto near child
  - If $t < \text{min}$: goto far child
  - Otherwise: goto both children with updated intervals

Min: 0
Max: $\infty$
Split: 1.0
Stack: 1
New min: 0
New max: 1.0
K-d traversal

- Keep track of intervals on ray:
  - Compute $t$ of split plane
  - If $t > \text{max}$: goto near child
  - If $t < \text{min}$: goto far child
  - Otherwise: goto both children with updated intervals

Min: 0
Max: 1.0
Split: 1.3
Stack: 1
New min: 0
New max: 1.0
K-d traversal

- Keep track of intervals on ray:
  - Compute $t$ of split plane
  - If $t > \text{max}$: goto near child
  - If $t < \text{min}$: goto far child
  - Otherwise: goto both children with updated intervals

Min: 0
Max: 1.0
Split: 0.4
Stack: 1 5
New min: 0
New max: 0.4
K-d traversal

- Keep track of intervals on ray:
  - Compute t of split plane
  - If t > max: goto near child
  - If t < min: goto far child
  - Otherwise: goto both children with updated intervals

Min: 0.4
Max: 1.0
Split: 0.7
Stack: 1 8
New min: 0.4
New max: 0.6
K-d traversal

- Keep track of intervals on ray:
  - Compute $t$ of split plane
  - If $t > \text{max}$: goto near child
  - If $t < \text{min}$: goto far child
  - Otherwise: goto both children with updated intervals

Min: 0.7
Max: 1.0
Split: 1
Stack: 1
New min: 0.4
New max: 0.6
K-d traversal

- Keep track of intervals on ray:
  - Compute \( t \) of split plane
  - If \( t > \) max: goto near child
  - If \( t < \) min: goto far child
  - Otherwise: goto both children with updated intervals

Min: 1.0
Max: \( \infty \)
Split: 1.5
Stack: 10
New min: 1.0
New max: 1.5
K-d traversal

- Keep track of intervals on ray:
  - Compute $t$ of split plane
  - If $t > \text{max}$: goto near child
  - If $t < \text{min}$: goto far child
  - Otherwise: goto both children with updated intervals

Min: 1.0
Max: 1.5
Split: 1.4
Stack: 10 11
New min: 1.0
New max: 1.4
K-d tree build

- Optimal solution is NP-hard
- Spatial median split (a little like octree)
- Object median split (makes balanced trees)
- Cost model based (better)
  - Cost of traversal
  - Cost of intersection
  - Probability of hit
K-d tree build

- Havran ‘01:
  - Start with bounding box of scene
  - Select split plane along each axis
    - Start with spatial median
    - Move toward bounding box of child nodes
  - Recurse
    - Stop if node contains 2-3 objects
    - Stop if depth > max (20-30)
  - Backtrack
    - Combine children with identical content
Acceleration Structure Summary

- Most common: Grid, Hierarchical Grid, k-d tree, BVH
- Grid based:
  - P time deterministic build
  - Grid resolution tradeoff
- Tree based:
  - NP hard build
  - Heuristic approaches