Acceleration Structures
rasterize( vert v[3] )
{
    bbox b; bound3(v, b);
    line l0, l1, l2;
    makeline(v[0],v[1],l2);
    makeline(v[1],v[2],l0);
    makeline(v[2],v[0],l1);
    for( y=b.ymin; y<b.ymax, y++ ) for( x=b.xmin; x<b.xmax, x++ ){
        e0 = l0.A * x + l0.B * y + l0.C;
        e2 = l2.A * x + l2.B * y + l2.C;
        if( inside(e0,l0)&inside(e1,l1)&inside(e2,l2) )
            fragment(x,y);
    }
}
Bounding Volumes

• Enclose a complex object within simpler geometry:
  – Sphere
  – Box (axis-aligned, object-aligned)

• If a ray does not hit the bounding volume, it can not hit the enclosed complex object
  – Avoid expensive intersection tests with complex objects
Bounding Spheres
Bounding Sphere

• A sphere enclosing all of the vertices of the object.
• Fastest bounding volume for a ray intersection test
Minimize the Radius

• Straightforward method:
  – Compute the centroid $P_c$ of the vertices
  – Loop through vertices and compute the maximal $|P_i-P_c|^2$ to find the farthest vertex $P_{\text{farthest}}$ from $P_c$
  – $|P_{\text{farthest}}-P_c|$ is the minimal radius
  – A crowd of points with a single outlier gives a radius twice as big as the minimal one

• Hierarchical method:
  – Make a bounding sphere for each triangle
  – Recursively merge neighboring spheres into larger ones
  – The radius of the final sphere is near minimal
Pros and Cons

• Advantages:
  – Construction is fast
  – Ray-sphere intersection test is fast
  – Bounding sphere is still valid when the object rotates (rotating a sphere doesn’t change it)

• Disadvantages:
  – Cannot tightly enclose some objects, e.g. long/thin
    • leads to many false positives
Axis-Aligned Bounding Boxes
Axis-Aligned Bounding Box (AABB)

• Construction:
  – Loop through all vertices to find the min and max values in each dimension
  – The min/max values are the diagonal corners of the AABB
Ray-AABB Intersection

- A box is the intersection of interiors of 3 pairs of planes
- Detect the intersections between the ray and the 3 pairs of planes
  - For each pair of planes, two intersection points: $P_{\text{near}}$ and $P_{\text{far}}$
    - 6 intersection points in total
  - If the last $P_{\text{near}}$ is before the first $P_{\text{far}}$ on the ray (according to parameter $t$), the ray misses the box (otherwise it hits the box)
Pros and Cons

• Advantages
  – Construction is simple and efficient
  – Ray-box intersection test is fast

• Disadvantages
  – Need to recalculate when objects rotates
    • unless the ray is transformed into object space!
  – Cannot tightly enclose some objects (similar to spheres)
Oriented Bounding Boxes
Oriented Bounding Box (OBB)

- The orientation depends on the orientation of the object
- Don’t need to recompute the box when the object rotates
- Can pre-compute in object space, and transform to world space with the object
  - the same is true for spheres
Construction

• To compute the principle orientation of the object:
  – Singular Value Decomposition (SVD)
  – Compute the covariance matrix, and the eigenvalues and corresponding eigenvectors of this 3x3 matrix
  – Using these eigenvectors as the basis of the local coordinate system of the bounding box
  – Similar to moment of inertia calculations for rigid bodies
    • except with a homogeneous mass density
• Or just estimate the principal directions with some heuristics...
Ray-OBB Intersection

• Similar to Ray-AABB intersection:
  – Calculate the last $t_{\text{near}}$ and the first $t_{\text{far}}$
  – The planes are not axis aligned any more
  – Recall how to compute intersection between a ray and an arbitrary plane in 3D space (lecture 11)
Or transform the ray...

- Transform the ray into the OBB coordinate system
- Then perform the ray-AABB intersection test
Pros and Cons

• Advantages
  – Fits the object tighter than an AABB or sphere
    • so it prunes out more rays

• Disadvantages
  – Extra cost for ray intersections
  – Calculating the OBB is computationally expensive
    • but can usually be done as a pre-computation
  – Finding the minimal OBB can be difficult
    • but an approximation is usually good enough
Bounding Volume Hierarchies
Bounding Volume Hierarchies (BVHs)

- Subparts of objects can be contained in their own bounding boxes in a hierarchical fashion
- E.g., sub-OBBs can be constructed by recursively splitting an OBB using center points along the longest axis

[Gottschalk et.al. 96] [Kamat and Martinez, 2007]
BVH Tree Structure

- Each leaf represents a bounding volume at the lowest level of the hierarchy.
- Nodes at higher levels of the tree represent bounding volumes that enclose all the bounding volumes of that node’s children.
Constructing BVHs

- **Top down**
  - Begin with the bounding volume for the entire object
  - Recursively split the object into smaller pieces according to some criteria, and construct bounding volumes for these sub-pieces
    - E.g., split primitives w.r.t the longest coordinate axis
  - Stop when some tolerance is reached for leaf nodes

- **Bottom up**
  - Begin with a bounding volume for each subpart of an object
  - Recursively merge them into larger volumes according to some criteria
    - E.g., merge nearest neighbors
  - Stop when there is only a single bounding volume at the root
Hierarchical Ray Traversal

- Intersect child bounding volumes only if intersections are found with the parent
- Intersect low level geometry only if a leaf node is reached and the ray intersects its bounding volume
Early Termination

• Here, every node and triangle, except for F, needs to be checked

• Sorting intersections based on the ray parameter, and finding the intersection with the triangle in G before checking the triangles in H and I allows those intersection tests to be skipped
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)

- Traversing all objects in the scene for each ray is inefficient
- Need auxiliary structures to accelerate this process

Each object should have a BV or BVH
Two Uses for BVHs

• Create a BVH for an individual object to serve as its bounding volume (**1st use - we already considered this**)  
  – The BVH is in object space  
  – As long as we transform the rays into object space, the BVH can be precomputed and does not need to be updated  

• Create a BVH for unioning together all the bounding volumes of all the objects in the scene (**2nd use**)  
  – If an object has its own object BVH, use the root BV of the BVH  
  – This newly constructed global (scene) BVH is in world space  
  – Must be updated whenever something moves in the scene  

• Do both!
Global (Scene) BVHs don’t always work well

• If many similar sized objects are approximately uniformly distributed in space, the global BVH hierarchy is vertically traversed quite a bit looking for the object the ray intersects
• One can do better...

http://hof.povray.org/shorebir.html
Uniform Spatial Partition
Uniform Spatial Partition

- Partition all of space and register objects with the cells that their bounding volumes overlap
Uniform Grids

• Divide 3D space into $n_x \times n_y \times n_z$ axis-aligned grid cells
• Perform ray-object intersection tests only when the ray hits a grid cell that contains an object
• The grid cell size is crucial to performance
  – No speed-up if the cell size is too big; no pruning, if everything is in one cell
  – A lot of empty cells, if the cell size is too small; the same object may be in too many different cells
  – A practical way to choose cell size is to average the edge lengths of the bounding volumes of all the objects
Constructing a Uniform Grid

• Find a bounding box for the entire scene

• Initialize the grid with that bounding box and a properly chosen cell size
  – Each cell maintains a list (or an array) to store the objects whose bounding volumes overlap that cell

• Insert objects into cells
  – The same object may be inserted into multiple cells
Ray Traversal

- Traverse all the cells pierced by the ray until the ray intersects with a primitive or reaches the boundary of the uniform grid.
Ray Traversal

- An incremental algorithm similar to line rasterization
- From the current intersection point $P$ on the face of cell $(i, j, k)$, perform ray-plane intersection tests with the next 3 grid planes along the ray direction to get the 3 candidate intersection points for the next intersection
- The next intersection point is the nearest one among the 3 candidates
- Update the cell index according to the new intersection point. Perform ray-primitive intersection tests in the new cell
- Repeat the process until the ray intersects a primitive or reaches the boundary
Ray Traversal Optimization

- The intersections with the grid planes have the same spacing in each independent dimension
- Use precomputed $\delta t_x$, $\delta t_y$, $\delta t_z$ to avoid ray-plane intersection tests
- $\delta t_x = C_x / D_x$, $\delta t_y = C_y / D_y$, $\delta t_z = C_z / D_z$
  - where $(C_x, C_y, C_z)$ is the cell size and $(D_x, D_y, D_z)$ is the ray direction
Avoid Redundant Intersection Tests

• A primitive may be stored in multiple cells
• An intersection test on that primitive may accidently be performed repeatedly for every cell that it is in
• Store a bool with each primitive
  – If the primitive has already been determined to not intersect the ray, or is not the closest intersecting primitive, store false
  – Before a ray-primitive intersection test, check the bool to see if the test was already performed
Viewing Frustrums
Viewing Frustums

• Send rays in “camera space”
  – Create a “uniform” grid in the frustum space
  – Avoids traversal steps and plane intersection tests
  – Cache coherent, all the cells are aligned along the ray!
  – Can traverse a bundle of rays at the same time
Optimizations
Optimizing the Uniform Grid

• Optimizing storage
  – Spatial Hashing: use a hash table instead of a 3D array
  – Avoid extra storage for a large number of empty cells

• Optimizing performance
  – Adaptive grids: rectilinear grid, grid hierarchy, octree
Octrees

PathTracer
Octree

- Each node has 0 or 8 children
  - Each node can equally subdivide its space (an AABB) into 8 subboxes by 3 midplanes
  - Children of a node are contained within the box of the node itself
Constructing an Octree

• Top-down:
  – Global bounding box that contains all the primitives is the root
  – Recursively partition a node into 8 octants by 3 midplanes
    • If a primitive belongs to multiple octants, put it in all of them
  – Recursion stops when the termination criteria are satisfied
    • e.g., maximum depth, minimum number of primitives in a node, etc.
Ray Traversal

- Traverse all leaf nodes in the octree passed through by the ray
- Perform intersection tests for the primitives inside those leaves
- Similar to (same as) BVH...
Optimization

- For a given ray and parent box, the subboxes that intersect the ray and the order of intersection are determined by the intersection points of the parent box.
Find the Next Subnode

- Build an automaton with states corresponding to boxes and transitions associated with movements between neighboring sub-nodes visited sequentially by the ray.
Example...
Uniform/Octree Hybrid
Hybrid Structure

- Start with a uniform grid and subdivide each node
  - Each node of a uniform grid can contain its own octree
  - Improves performance over that of a standard octree
    - No need to traverse the tree from a single root every time

[Losasso et.al. 2006]
K-d Trees
K-d Tree

- Use a hyperplane translated in one dimension
K-d Tree

- A binary tree for space searching
- Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides space into two parts
- Points to the left of this hyperplane are represented by the left subtree of that node and points to the right of this hyperplane are represented by the right subtree
Constructing a K-d Tree

• Top-down:
  – Begin with the global bounding box containing all primitives.
  – Choose an axis and a splitting plane perpendicular to that axis
  – Subdivide the primitives on both sides of the plane into two groups
  – Stop when the number of primitives in each single group is below a threshold
Ray Traversal

- Traverse all leaf nodes in the k-d tree passed through by the ray
Recursive Traversal

• Similar to recursive traversal in an octree
  – Each interior node has two children, there are 3 cases for subnode intersections:
    • left child only, the right child only, and both
  – The number of intersected subboxes and its sequence can be determined by the intersection points
    • Notice the difference with an octree; the splitting plane is not necessarily in the middle of the box
Non-Recursive Traversal

- Recursion is expensive on the GPU for real-time ray tracing
- Kd-Restart
  - Restart the traversal at the root every time it reaches a leaf
- Kd-Backtrack
  - Return to parent nodes to enable traversing other unvisited nodes
- Neighbor-link
  - Each leaf stores ropes that directly link it to the adjacent node via its 6 faces
  - Octrees typically have these neighbor structures too

[Popov et.al. 07]
Your Scene...
Question 1 (short/long)

Describe a smart use of acceleration structures for your scene...
Question 2 (long only)

Read about k-d trees; write about 500 words summarizing important aspects: