Spatial Data Structures

- Spatial data structures store data indexed in some way by their spatial location
  - For instance, store points according to their location, or polygons, ...
  - Before graphics, used for queries like “Where is the nearest hotel?” or “Which stars are strong enough to influence the sun?”

- Multitude of uses in computer games
  - Visibility - What can I see?
  - Ray intersections - What did the player just shoot?
  - Collision detection - Did the player just hit a wall?
  - Proximity queries - Where is the nearest power-up?
Focus on spatial data structures that partition space into regions, or *cells*, of some type

- Generally, cut up space with planes that separate regions
- Almost always based on tree structures (surprise, huh?)

**Octrees (Quadtrees):** Axis aligned, regularly spaced planes cut space into cubes (squares)

**Kd-trees:** Axis aligned planes, in alternating directions, cut space into rectilinear regions

**BSP Trees:** Arbitrarily aligned planes cut space into convex regions
Many geometric queries are expensive to answer precisely. The best way to reduce the cost is with fast, approximate queries that eliminate most objects quickly. Trees with a containment property allow us to do this. The cell of a parent completely contains all the cells of its children. If a query fails for the cell, we know it will fail for all its children. If the query succeeds, we try it for the children. If we get to a leaf, we do the expensive query for things in the cell.

Spatial decompositions are most frequently used in this way. For example, if we cannot see any part of a cell, we cannot see its children. If we see a leaf, use the Z-buffer to draw the contents.
Octree

- Root node represents a cube containing the entire world
- Then, recursively, the eight children of each node represent the eight sub-cubes of the parent
- Quadtree is for 2D decompositions - root is square and four children are sub-squares
  - What sorts of games might use quadtrees instead of octrees?
- Objects can be assigned to nodes in one of two common ways:
  - All objects are in leaf nodes
  - Each object is in the smallest node that fully contains it
  - What are the benefits and problems with each approach?
Octree Node Data Structure

What needs to be stored in a node?
- Children pointers (at most eight)
- Parent pointer - useful for moving about the tree
- Extents of cube - can be inferred from tree structure, but easier to just store it
- Data associated with the contents of the cube
  - Contents might be whole objects or individual polygons, or even something else
- Neighbors are useful in some algorithms (but not all)
Define a function, `buildNode`, that:
- Takes a node with its cube set and a list of its contents
- Creates the children nodes, divides the objects among the children, and recurses on the children, or
- Sets the node to be a leaf node

Find the root cube (how?), create the root node and call `buildNode` with all the objects

When do we choose to stop creating children?
- Is the tree necessarily balanced?

What is the hard part in all this? Hint: It depends on how we store objects in the tree
Example Construction
Assignment of Objects to Cells

- Basic operation is to intersect an object with a cell
  - What can we exploit to make it faster for octrees?
- Fast algorithm for polygons (Graphics Gems V):
  - Test for trivial accept/reject with each cell face plane
    - Look at which side of which planes the polygon vertices lie
    - Note speedups: Vertices outside one plane must be inside the opposite plane
  - Test for trivial reject with edge and vertex planes
    - Planes through edges/vertices with normals like (1,1,1) and (0,1,1)
  - Test polygon edges against cell faces
  - Test a particular cell diagonal for intersection with the polygon
  - Information from one test informs the later tests. Code available online
Approximate Assignment

Recall, we typically use spatial decompositions to answer approximate queries

- Conservative approximation: We will sometimes answer yes for something that should be no, but we will never answer no for something that should be yes

- Observation 1: If one polygon of an object is inside a cell, most of its other polygons probably are also
  - Should we store lists of objects or polygons?

- Observation 2: If a bounding volume for an object intersects the cell, the object probably also does
  - Should we test objects or their bounding volumes? (There is more than one answer to this - the reasons are more interesting)
Objects in Multiple Cells

- Assume an object intersects more than one cell
- Typically store pointers to it in all the cells it intersects
  - Why can’t we store it in just one cell? Consider the ray intersection test
- But it might be considered twice for some tests, and this might be a problem
  - One solution is to flag an object when it has been tested, and not consider it again until the next round of testing
    - Why is this inefficient?
  - Better solution is to tag it with the frame number it was last tested
    - Subtle point: How long before the frame counter overflows?
Sometimes it helps if a cell knows its neighbors

- How far away might they be in the tree? (How many links to reach them?)
- How does neighbor information help with ray intersection?

Neighbors of cell A are cells that:

- Share a face plane with A
- Have all of A’s vertices contained within the neighbor’s part of the common plane
- Have no child with the same property
Finding Neighbors

- Your right neighbor in a binary tree is the leftmost node of the first sub-tree on your right
  - Go up to find first rightmost sub-tree
  - Go down and left to find leftmost node (but don’t go down further than you went up)
  - Symmetric case for left neighbor

- Find all neighbors for all nodes with an in-order traversal

- Natural extensions for quadtrees and octrees
Frustum Culling With Octrees

- We wish to eliminate objects that do not intersect the view frustum
- Which node/cell do we test first? What is the test?
- If the test succeeds, what do we know?
- If the test fails, what do we know? What do we do?
We wish to eliminate objects that do not intersect the view frustum.

Have a test that succeeds if a cell may be visible:
- Test the corners of the cell against each clip plane. If all the corners are outside one clip plane, the cell is not visible.
- Otherwise, is the cell itself definitely visible?

Starting with the root node cell, perform the test:
- If it fails, nothing inside the cell is visible.
- If it succeeds, something inside the cell might be visible.
- Recurse for each of the children of a visible cell.

This algorithm with quadtrees is particularly effective for a certain style of game. What style?
Interference Testing

- Consider the problem of finding out which cells an object interferes with (collides with)
  - When do we need to answer such questions?
- Consider a sphere and an octree
  - Which octree node do we start at?
  - What question do we ask at each node?
  - What action do we take at each node?
Ray Intersection Testing

- Consider the problem of finding which cells a ray intersects
  - Why might we care?
- Consider a ray (start and direction) and an octree
  - Which cell do we start at?
  - How does the algorithm proceed?
  - What information must be readily accessible for this algorithm?
Octree Problems

- Octrees become very unbalanced if the objects are far from a uniform distribution
  - Many nodes could contain no objects
- The problem is the requirement that cube always be equally split amongst children

A bad octree case
Kd-trees

- A kd-tree is a tree with the following properties
  - Each node represents a rectilinear region (faces aligned with axes)
  - Each node is associated with an axis aligned plane that cuts its region into two, and it has a child for each sub-region
  - The directions of the cutting planes alternate with depth – height 0 cuts on x, height 1 cuts on y, height 2 cuts on z, height 3 cuts on x, …

- Kd-trees generalize octrees by allowing splitting planes at variable positions
  - Note that cut planes in different sub-trees at the same level need not be the same
Kd-tree Example
Kd-tree Node Data Structure

- What needs to be stored in a node?
  - Children pointers (always two)
  - Parent pointer - useful for moving about the tree
  - Extents of cell - \( x_{\text{max}}, x_{\text{min}}, y_{\text{max}}, y_{\text{min}}, z_{\text{max}}, z_{\text{min}} \)
  - List of pointers to the contents of the cell
  - Neighbors are complicated in kd-trees, so typically not stored
Building a Kd-tree

- Define a function, `buildNode`, that:
  - Takes a node with its cell defined and a list of its contents
  - Sets the splitting plane, creates the children nodes, divides the objects among the children, and recurses on the children, or
  - Sets the node to be a leaf node
- Find the root cell (how?), create the root node and call `buildNode` with all the objects
- When do we choose to stop creating children?
- What is the hard part?
Kd-tree
Kd-tree
Kd-tree
Kd-tree
Kd-tree
Kd-tree
Kd-tree
Kd-tree
Choosing a Split Plane

- Two common goals in selecting a splitting plane for each cell
  - Minimize the number of objects cut by the plane
  - Balance the tree: Use the plane that equally divides the objects into two sets (the *median cut* plane)
  - One possible global goal is to minimize the number of objects cut throughout the entire tree (intractable)
Choosing Split Planes

Generally NP-complete
• Use some heuristic
• Minimize split objects?
• Balance - median split?
For many operations, it’s convenient to simplify objects to “fat points”
- Compute axis-aligned bounding box
- For each coordinate, compute the mean of the bounds
Candidate Split Planes

Infinite number of choices:
- Which planes to choose from?
- Axis-aligned?
- Use AABBs to define candidate planes
Choosing Split Planes

Generally NP-complete
- Use some heuristic
- Minimize split objects?
- Balance?
- Just split at midpoint of range?
The probability of a ray hitting a convex shape that is completely inside a convex cell equals

\[
\Pr[r \cap S_o | r \cap S_c] = \frac{S_o}{S_c}
\]
Surface Area Heuristic

Intersection time

\[ t_i \]

Traversal time

\[ t_t \]

\[ t_i = 80t_t \]

\[ C = t_t + p_a N_a t_i + p_b N_b t_i \]
Surface Area Heuristic

\[ p_a = \frac{S_a}{S} \quad \quad p_b = \frac{S_b}{S} \]
Kd-tree Applications

- Kd-trees work well when axis aligned planes cut things into meaningful cells
  - What are some common environments with rectilinear cells?
- View frustum culling extents trivially to kd-trees
- Kd-trees are frequently used as data structures for other algorithms – particularly in visibility
- Specific applications:
  - Soda Hall Walkthrough project (Teller and Sequin)
    - Splitting planes came from large walls and floors
  - Real-time Pedestrian Rendering (University College London)
BSP Trees

- **Binary Space Partition** trees
  - A sequence of cuts that divide a region of space into two

- Cutting planes can be of any orientation
  - A generalization of kd-trees, and sometimes a kd-tree is called an axis-aligned BSP tree

- Divides space into convex cells

- The industry standard for spatial subdivision in game environments
  - General enough to handle most common environments
  - Easy enough to manage and understand
  - Big performance gains
Notes:
- Splitting planes end when they intersect their parent node’s planes
- Internal node labeled with planes, leaf nodes with regions
What needs to be stored in a node?
- Children pointers (always two)
- Parent pointer - useful for moving about the tree
- If a leaf node: Extents of cell
  - How might we store it?
- If an internal node: The split plane
- List of pointers to the contents of the cell
- Neighbors are useful in many algorithms
  - Typically only store neighbors at leaf nodes
  - Cells can have many neighboring cells
- Portals are also useful - holes that see into neighbors
Define a function, `buildNode`, that:
- Takes a node with its cell defined and a list of its contents
- Sets the splitting plane, creates the children nodes, divides the objects among the children, and recurses on the children, or
- Sets the node to be a leaf node

Create the root node and call `buildNode` with all the objects
- Do we need the root node’s cell? What do we set it to?

When do we choose to stop creating children?

What is the hard part?
Choosing Splitting Planes

- **Goals:**
  - Trees with few cells
  - Planes that are mostly opaque (best for visibility calculations)
  - Objects not split across cells

- **Some heuristics:**
  - Choose planes that are also polygon planes
  - Choose large polygons first
  - Choose planes that don’t split many polygons
  - Try to choose planes that evenly divide the data
  - Let the user select or otherwise guide the splitting process
  - Random choice of splitting planes doesn’t do too badly
BSP trees can be used to order polygons from back to front, or visa-versa.

- Descend tree with viewpoint
- Things on the same side of a splitting plane as the viewpoint are always in front of things on the far side

Can draw from back to front
- Removes need for z-buffer, but few people care any more
- Gives the correct order for rendering transparent objects with a z-buffer, and by far the best way to do it

Can draw front to back
- Use info from front polygons to avoid drawing back ones
- Useful in software renderers
BSP in Games

- Use a BSP tree to partition space as you would with an octree or kd-tree
  - Leaf nodes are cells with lists of objects
  - Cells typically roughly correspond to “rooms”, but don’t have to
- The polygons to use in the partitioning are defined by the level designer as they build the space
  - A *brush* is a region of space that contributes planes to the BSP
  - Artists lay out brushes, then populate them with objects
  - Additional planes may also be specified
    - Sky planes for outdoor scenes, that dip down to touch the tops of trees and block off visibility
    - Planes specifically defined to block sight-lines, but not themselves visible
Dynamic Lights and BSPs

- Dynamic lights usually have a limited radius of influence to reduce the number of objects they light.
- The problem is to find, using the BSP tree, the set of objects lit by the light (intersecting a sphere center \((x,y,z)\) radius \(r\)).
- Solution: Find the distance of the center of the sphere from each split plane.
  - What do we do if it’s greater than \(r\) distance on the positive side of the plane?
  - What do we do if it’s greater than \(r\) distance on the negative side of the plane?
  - What do we do if it’s within distance \(r\) of the plane?
  - Any leaf nodes reached contain objects that might be lit.
BSP and Frustum Culling

- You have a BSP tree, and a view frustum
  - With near and far clip planes
- At each splitting plane:
  - Test the boundaries of the frustum against the split plane
  - What if the entire frustum is on one side of the split plane?
  - What if the frustum intersects the split plane?

- What do you test in situations with no far plane?
- What do you do when you get to a leaf?
So far, we have had subdivisions that break the world into cell

General Bounding Volume Hierarchies (BVHs) start with a bounding volume for each object
  - Many possibilities: Spheres, AABBs, OBBs, k-dops, …
  - More on these later

Parents have a bound that bounds their children’s bounds
  - Typically, parent’s bound is of the same type as the children’s
  - Can use fixed or variable number of children per node

No notion of cells in this structure
BVH Example
BVH Construction

- Simplest to build top-down
  - Bound everything
  - Choose a split plane (or more), divide objects into sets
  - Recurse on child sets
- Can also be built incrementally
  - Insert one bound at a time, growing as required
  - Good for environments where things are created dynamically
- Can also build bottom up
  - Bound individual objects, group them into sets, create parent, recurse
  - What's the hardest part about this?
Some of the operations we’ve looked at so far work with BVHs
- Frustum culling
- Collision detection

BVHs are good for moving objects
- Updating the tree is easier than for other methods
- Incremental construction also helps (don’t need to rebuild the whole tree if something changes)

But, BVHs lack some convenient properties
- For example, not all space is filled, so algorithms that “walk” through cells won’t work
Visibility

- Visibility algorithms aim to identify everything that will be visible, and not much more
- Trade off: Application-side time on visibility, vs. hardware time on processing invisible stuff
- *Conservative Visibility*: Identify more than what is visible and clean up remaining with a z-buffer
- The simplest are view-frustum algorithms that eliminate objects outside the view frustum
  - These algorithms don’t do very well on scenes with *high depth complexity*, or many objects behind a single pixel
  - Buildings are a classic case of high depth complexity
Point-based vs. Cell-based

- Point-based algorithms compute visibility from a specific point
  - Which point?
  - How often must you compute visibility?
- Cell-based algorithms compute visibility from an entire cell
  - Union of the stuff visible from each point in the cell
  - How often must you compute visibility?
- Which method has a smaller visible set?
- Which method is suitable for pre-computation?
Cell-Portal Structures

- Cell-Portal data structures dispense with the hierarchy and just store neighbor information
  - This make them graphs, not trees
- Cells are described by bounding polygons
- Portals are polygonal openings between cells
- Good for visibility culling algorithms, OK for collision detection and ray-casting
- Several ways to construct
  - By hand, as part of an authoring process
  - Automatically, starting with a BSP tree or kd-tree and extracting cells and portals
  - Explicitly, as part of an automated modeling process
Cell Portal Example

- Portals can be one way (directed edges)
- Graph is normally stored in adjacency list format
  - Each cell stores the edges (portals) out of it
Cell-Portal Visibility

- Keep track of which cell the viewer is in
- Somehow walk the graph to enumerate all the visible regions
- Cell-based: Preprocess to identify the potentially visible set (PVS) for each cell
  - Set may contain whole cells or individual objects
- Point-based: Traverse the graph at runtime
  - Granularity can be whole cells, regions, or objects
- Trend is toward point-based, but cell-based is still very common
  - Why choose one over the other?
Potentially Visible Sets

- **PVS**: The set of cells/regions/objects/polygons that can be seen from a particular cell
  - Generally, choose to identify objects that can be seen
  - Trade-off is memory consumption vs. accurate visibility
- **Computed as a pre-process**
  - Have to have a strategy to manage dynamic objects
- **Used in various ways:**
  - As the only visibility computation - render everything in the PVS for the viewer’s current cell
  - As a first step - identify regions that are of interest for more accurate run-time algorithms
Cell A is in cell B’s PVS if there exist a *stabbing line* that originates on a portal of B and reaches a portal of A

- A *stabbing line* is a line segment intersecting only portals
- Neighbor cells are trivially in the PVS

PVS for I contains: B, C, E, F, H, J
Finding Stabbing Lines

- In 2D, have to find a line that separates the left edges of the portals from the right edges
  - A linearly separable set problem solvable in $O(n)$ where $n$ is the number of portals
- In 3D, more complex because portals are now a sequence of arbitrarily aligned polygons
  - Put rectangular bounding boxes around each portal and stab those
  - $O(n \log n)$ algorithm
Cell-To-Region PVS

- Identify which *regions* are visible from a cell
  - Add objects within region to PVS for the cell
- Key idea is separating planes (or lines in 2D):
  - Lines going through left edge of one portal and right edge of the other, and vice versa
  - Potentially visible region is bounded by planes
  - In 3D, have to find maximal planes (those that make region biggest)

This picture should remind you of something (Hint: Think of the left portal as a light source)
Cell-To-Region (More)

- If the sequence has multiple portals, find maximal separating lines.

![Diagram of portals and separating lines]

- This work originates from many sources, including shadow computations and mesh generation for radiosity.
- More applications of separating and supporting planes later.
  - Is it OK to use portals that are larger than the actual opening?
  - Is it OK to use portals that are smaller than the actual opening?
Properties of PVSs?

- Almost all of the work is done as a preprocess
  - At run-time, simply traverse PVS and render contents
  - Can pre-compute display lists for each cell – fast rendering

- Most algorithms go further than just Cell-to-Cell PVS
  - It overestimates by quite a lot – PVS removes 90% of the model, 99.6% is actually invisible, and better visibility gets 98% (Teller 91)

- Cell-to-Cell PVS is good for dynamic objects
  - Associate moving objects with the cell they currently occupy
  - Draw a moving object if the cell it is in is visible
PVS Problems?

- Does not take into account the viewer’s location, so reports things that the viewer cannot possibly see
- Not good at managing dynamic cells/portals
  - What do you do for doors that can be open or closed?
- Pre-processing time can be huge
  - Impacts development of game – turnaround time for changes is large
- Other algorithms address these things
Enhancing Cell-to-Anything

- If the viewer cannot go everywhere in the cell, then cell-based visibility will be too pessimistic.
- One solution is to add special cells that the viewer can see into, but can’t see out of:
  - Put them in places that the viewer cannot go, but can still see
    - Above a certain altitude in outdoor games
    - Below the player’s minimum eye level
  - Basically implemented as one-way portals
    - The portals only exist in the direction into the cell
- Note, doesn’t work if the player should be able to see through a special cell into another cell beyond – why not?
Define a procedure `renderCell`:
- Takes a view frustum and a cell
  - Viewer not necessarily in the cell
- Draws the contents of the cell that are in the frustum
- For each portal out of the cell, clips the frustum to that portal and recurse with the new frustum and the cell beyond the portal
  - Make sure not to go to the cell you entered
- Start in the cell containing the viewer, with the full viewing frustum
- Stop when no more portals intersect the view frustum
Eye-to-Region Example (1)
Eye-to-Region Example (2)
Implementation

- Each portal that is passed through contributes some clipping planes to the frustum
  - If the hardware has enough planes, add them as hardware clipping planes
  - Or, clip object bounding volumes against them to determine which objects to draw

- Mirrors are reasonably easy to deal with
  - Flip the view frustum about the mirror
  - Add appropriate clipping planes to make sure the right things are drawn

- A very effective algorithm if the portals are simple
  - More complex portals can be bounded with screen-space rectangles
No Cell or Portals?

- Many scenes do not admit a good cell and portal structure
  - Scenes without large co-planar polygons to act as blockers or cell walls
  - Canonical example is a forest – you can’t see through it, but no one leaf is responsible
- What can we do?
  - Find *occluders* and use them to cull geometry
  - Inverse of cells as portals: Assume all space is open and explicitly look at places where it is blocked
Using Occluders

- Assume the occluder is a polygon
- Form clipping planes using the eye point and the polygon edges
  - Supporting planes
- Objects inside all of the occluder’s clipping planes are NOT visible
  - Occluder itself is a clipping plane
  - Can use tests similar to view frustum culling, but note that now we trivially accept as soon as the object is outside a clipping plane
Simple Occluder Finding

- Cell based approach
- Find good sets of occluders for each cell in a preprocess
  - At run time, use occluders from the viewer’s region
- What makes a good occluder?
  - Things that occlude lots of stuff
  - What properties will a good occluder have?
Simple Occluder Issues

- Works best when there are large polygons close to the viewer
  - Dashboards are a good example
- For objects, how do you choose their “occlusion shape”?
- Level designers can add special polygons just to act as occluders
  - In what situation would you do this?
  - But should they be drawn?
- Cell size is clearly important
- Problem: If an object is partially hidden by one occluder, and partially by another, it is hard to determine whether the entire object is occluded
Occluder Fusion

- Small occluders can be merged to generate larger occluders
  - Level editors are essentially doing this by hand when they place special occluders

- Key insight: If a potential occluder intersects the occluded region of another, they can be fused
  - Depth of fused occluder is farthest depth of fused occluders
Algorithms for Combining Occluders

- **Occlusion Horizons** work for 2.5D scenes
  - Great for cities and the like
  - An extension exists for relatively simple 3D scenes (e.g., bridges)
- Green’s **Hierarchical Z-Buffer** builds occluders in screen space and does occlusion tests in screen space
  - Requires special hardware or a software renderer
- **Zhang et.al. Hierarchical Occlusion Maps** render occluders into a texture map, then compare objects to the map
  - Uses existing hardware, but pay for texture creation operations at every frame
  - Allows for approximate visibility if desired (sometimes don’t draw things that should be)
- **Schaufler et.al. Occluder Fusion** builds a spatial data structure of occluded regions