Binary Space Partitioning Trees

Computational Geometry

- We've covered lots of general purpose algorithms and data structures which are useful for a variety of applications
- “Computational geometry” is the study of algorithms/data structures used for geometric data
- Today we'll look at a data structure for organizing line segments in the plane
- We'll apply some of the ideas we've discussed to applications and see some new challenges that haven't come up so far

Problem 1: “The painter's algorithm”

- The painter's algorithm is a 3D rendering algorithm for drawing a 3D scene from a particular viewpoint (like taking a photo from a virtual camera)
- One of the challenges of rendering is handling “occlusion.” Some virtual objects are closer to the camera than others and will hide them
- The painter's algorithm is a simple algorithm for making only visible objects are drawn:
- For each object, sorted from farthest to nearest:
  - Draw it
- The algorithm is correct (though somewhat wasteful)

Note about about modern graphics

- Modern graphics hardware works sort of opposite to the painter's algorithm
- When we're about to draw an object over a given pixel, if we've already drawn something on that pixels that's closer, then we skip the current object
- The data structures used for that is called a “depth buffer” and is just a 2D array storing the “closest thing drawn so far” and it's not too interesting algorithmically
- Modern 3D engines do something like what we're discussing today to save the GPU some work, though

Challenge 1: We can't always sort objects!

- Even objects which don't overlap can't be sorted from “farthest to nearest” in general!
- Consider arranging 3 pencils so the tip end of one is on the eraser end of another for each of them
- Depending where you're looking, a different pencil is “closest” to you and occludes part of another pencil
- There's no way we could correctly draw this scene by drawing the pencils completely one at a time

Solution, 2D Simplicity

- One way we can solve this problem is by splitting the pencils into smaller pieces so we can draw the pieces in “far to near” order
- How should we split them? It's not obvious!
- In 2D things can't get quite so bad because we lose a dimension that we can orient things in, but we'll see that splitting our shapes (line segments) is still a useful trick for our algorithm

Challenge 2: View dependence

- Depending on where the viewer is, the order we want to draw the objects will be different!
- As an obvious example, if we look at the scene from one direction, then walk forward for a bit and turn around, the same objects will be “in frame” but their order will be reversed
- This means we can't just compute the object order once and use it for all viewpoints if the camera can move around

Solution “Shape”

- We need to somehow organize our line segments so that ANY viewpoint, we can quickly compute a front to back ordering
- Can we draw inspiration on other data structures we've seen?
- Hash tables are cool, but they don't provide any sort of ordering, which is what we need
- What about BSTs?

BST inspiration

- BSTs sort of point us in the right direction except that they assume there's a single way to order the elements
- How can we “order” line segments?
- There's not really an order, but we compute a relationship which is sort of like “less than” for line segments

Pseudo-comparator

- One way we can “compare” line segment A to B is to compute “which side of A” segment B is on
- If we imagine extending A infinitely we can determine which “side” of that line B's endpoints are on
- If B has one endpoint on each side, we split it into 2 segments, each of which is entirely on one side
- How does this help us with the painter's algorithm?
- If we're looking at segment “A” and know that “B” is on the “other side” of it, we must draw “B” before we draw “A”, or vise versa!
- For a given viewpoint, we can use this to tell whether to draw A or B first!

Binary Space Partitioning

- We call this “pseudo comparator” “binary space partitioning” because it partitions (splits) space into 2 regions, corresponding to the 2 “sides” of the infinitely extended line segment
- If we were to split space based on A, we could split the side that B's on in half by again paritioning space based on B
- If we have many segments we can cut space into smaller and smaller pieces by repeatedly performing partitioning
- Note, the first few partitions will extend out infinitely in some directions, but eventually we'll get finitely sized regions

The data structure: BSP Trees

- The BSP tree is a data structure similar to a BST
- Each node stores a segment, let's call it s
- All its left descendents contain segments which are one side of s and all of its right segments are on the other
- Which side is which is arbitrary since one side isn't “less than” or “greater than” the other, it's just 2, non-overlapping regions

Inserting into a BSP tree

- To insert a segment into a BSP tree, we modify the BST search algorithm
- Starting at the root, if both endpoints of the new segment are on the same side of the segment at that node, go to the corresponding child and continue searching
- Just like with a BST, we'll always insert new segments as leaf nodes
- What happens if the endpoints are on opposite sides of a segment?
- We find the intersection point and split the segment we're inserting into 2 segments
- We insert one segment into the left subtree and the other into the right subtree

One-shot construction

- If we have our whole list of segments when we want to build the tree we can use that to simplify + hopefully speed up tree construction
- We'll build the tree recursively, in a top down manner so we won't have to keep searching from the root over and over again.

Build tree

```
buildNode(segments){
    s = Pick a segment to store at this node
    Partition the array so all segments "left" of s at the beginning
    s.left = buildNode(left part of the array)
    s.right = buildNode(right part of the arary)
}
```

- Base case omitted
- Note “left” is arbitrary, see next slide

Segment math

- How can we compute which side of a line a point is on?
- We'll look at 2 vectors
  - the vector between endpoints of the vector that's defining the split (computed as segment.end - segment.begin)
  - the vector between one segment point and the point we're “classifying” (computed as point - segment.begin)
- A point is “left” of the segment if the dot product between those 2 vectors is positive (dot product is `v1.x*v2.x + v1.y*v2.y`)
- If the dot product is exactly 0 then the point is exactly on the line
- Note, doing this stuff in real applications using floats can lead to all sorts of weird bugs!

Traversal for Painter's Algorithm

- To draw the scene, we need to visit all the nodes in the tree in “far to near” order, which is of course dependent on where the viewer is
- We can define a recursive traversal which will give us a valid ordering (there are many) and we can just draw the segments in the order the traveral visits them

Far to near traversal

```
traverse(node, viewPos){
    determine which side of node.segment viewPos is
    If it's on the "left" side:
        traverse(node.right, viewPos)
        visit(node.segment)
        traverse(node.left, viewPost)
    else:
        ...
}
```

Analysis

- What's the runtime of the various operations in Big O?
- Insert(Segment) will traverse the tree from root to leaf, possibly multiple times if it gets split!
- Assuming O(1) splits as we go down, we get O(height) runtime for the insert
- Since we have a binary tree, we know the hight is O(lg N) if the tree is balanced, but could be O(N) otherwise
- Note that this tree doesn't do any balancing operations so the height is dependent on the order segments are inserted!

Bulk construction analysis

- What about constructing a tree from an array of elements?
- The analysis is similar to quicksort! At each recursive call we partition the segments into left/right halves, the recurse
- If the partitions are roughly equal, then we'll get O(lg N) height and O(N lg N) runtime
- The fact that segments might get split complicates the analysis a bit, so we'll mostly ignore that
- How can we choose good “pivot” elements?
- It's hard, so don't!

Randomized algorithms

- If we choose a bad segment to split at we can end up with O(N) height in our tree, even if we have all the segments when we start construction!
- If we choose a split node randomly, then we can assume that we won't keep making the worst choice at each level
- This is a common strategy for avoiding worst case behavior in algorithms where making a “good” choice is difficult/impossible

Investing up front

- For many applications of BSP trees, you only build a tree once and then perform many traversals, so extra time spent during construction is likely well spent
- An extreme example is that DOOM (the video game) stored maps as BSP trees, so they were built by id software before the game was shipped
- Even if it took overnight to build a tree because they spent a long time choosing good pivots, that's fine! The game would run faster/better for all users, so that's a great tradeoff!

Application 2: Collision detection

- If we have a user moving around a scene that we'd like to draw, we probably want to prevent them from going through walls
- We can use BSP trees for collision detection too
- For this application we'll compute a segment which goes from the player's current position to where they want to go
- If that segment collides with any walls, it means they'd be going through walls so we won't let them

Traversal algorithm

```
traverse(Node, query)
    if both query points are on the same side of node.segment:
        recuse to that side
    else:
        check to see if query intersects node.segment
        if not, recurse to both sides
```

- Base cases omitted
- Disclaimer, I'm still working on my implementation!