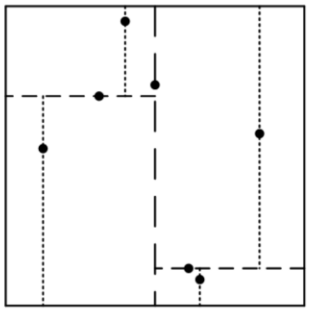
K-D Trees

jodavis42@gmail.com

K-D Tree

Generalization of oct/quad tree



A k-d tree, or k-dimensional tree is a generalization of oct and quad trees. It can also be thought of as a specialization of a bsp-tree. K-d trees are simply a tree composed of splitting hyper-planes, like bsp-trees, but only allowing axis aligned splits. That means the majority of bsp-tree topics that we've talked about also apply to k-d trees.

K-D Tree

Why use k-d tree over bsp?

So why do we care about k-d trees over bsp-trees? The constraint to only choose axis aligned splitting planes greatly simplifies a lot of logic. In particular we get 3 major beneifits:

1. Axis aligned splits are much cheaper to perform collision detection against
2. Axis aligned split planes are numerically robust
3. Axis aligned splits require less memory

K-D Tree

Why use k-d tree over bsp?

1. Axis aligned splits are computationally cheaper

Axis aligned splits are much cheaper to perform collision detection against than arbitrary aligned axes. This is one reason why Aabbs are used over Obbs. Instead of having to compare on a random axis which requires a projection we can directly check the x, y, or z-values which requires no additional computations.

K-D Tree

Why use k-d tree over bsp?

1. Axis aligned splits are computationally cheaper
2. Axis aligned split planes are numerically robust

The second reason to use k-d trees is that axis aligned splits are numerically robust. This is because we can directly compare a float's value without having to worry about the imprecision of floats. With a bsp-tree we can have a point accidentally classify as in-front, behind, or coplanar just due to how a float value projects onto a given normal. As there are no projections being computed this isn't a problem with k-d trees.

K-D Tree

Why use k-d tree over bsp?

1. Axis aligned splits are computationally cheaper
2. Axis aligned split planes are numerically robust
3. Axis aligned splits require less memory

And finally, an axis aligned split requires less memory than an arbitrary split.
An arbitrary split plane requires at minimum 4 floats for the plane normal and position (vector4 version of a plane). A k-d tree split only requires which axis and the float value, so 1 int and 1 float. If you go to the extreme, you can pack even more into the integer (as only 2 bits are needed for signifying which axis), but that's a topic for another time.

K-D Tree: Uses

Anywhere a bsp-tree is useful (spatial partition)

Nearest neighbor search
Store k-dimensional data
    Databases (age, height, etc…)

Focusing on point-based trees

K-d trees are useful in many of the same areas of bsp-trees, such as static broadphases. As these topics were already covered with bsp-trees I'll focus on some other useful applications of k-d trees, in particular nearest neighbor searches.

One common application of k-d trees outside of computational geometry is to store k-dimensional un-related data, such as height, age, etc… This data can be thought of as a point in k-dimensional space. Hence k-d trees often are constructed just from points. For simplicity's sake, we'll just look at 3d points, just keep in mind this can be extended for higher dimensions.

This can thought of as dividing space in half by a splitting hyperplane.

K-D Tree: Construction

Have to determine split axis and split value

Only 3 choices for split axis: x, y, or z

- Typically the axis is picked by the tree depth
- Heuristics can also be used

First let's look at how you construct a k-d tree of points. As with bsp-trees, there's 2 different criteria we have to consider: the split axis and the split position.

Our choices for split axis are much more limited with a k-d tree than a bsp-tree, as we just have to choose which of k-dimensions we're going to split. The canonical k-d tree just alternates the axis each time you move down the tree. For example, the root would split on the x-axis, then the next level would split on the y-axis, then the z, then the x, and so on.
As we're only sorting points, when the dimension of the tree is small (say 3) then this works fairly well.

Alternatively, a heuristic can be used to pick the axis. Heuristics aren't commonly used for k-d trees, but most of the typical ones could be used: largest spread, smallest spread, and so on.

K-D Tree: Construction

Have to determine split axis and split value

Split values have a lot more variation

To guarantee a balanced tree the spatial median is typically picked

Store the split point in the node

Once the split axis is chosen then split position can be chosen. The most common method is to divide the data set in half, that is sort along the split axis and choose the median point. This leads naturally to constructing a node-storing k-d tree where the split point is stored in the internal node.

Alternatively any method described for bsp-trees can work, even surface area heuristics.

Termination of tree construction typically continues until a node contains some max number of points, a max tree depth, or some other termination condition.

K-D Tree: Construction

When do we stop recursing? Some mixture of:

- Max number of points
- Max tree depth
- etc…

This just leaves the question of when to stop tree construction? The most obvious answer is to stop when we only have 1 point left, but we could choose an earlier termination condition. The two most typical are point count and tree depth. Other heuristics could be used where appropriate.

K-D Tree: Insertion

Recurse down the tree, classifying at each step

If input < node.Split

    Recurse down the leftside

Else

    Recurse down the right side

Insert as a child of the found leaf-node

New points can be added dynamically to a k-d tree. Simply traverse the tree classifying the input point at each step. If the point is to the left of the split plane then recurse down the left, otherwise down the right. When reaching a leaf node the input point is just inserted as the new left or right child of the leaf node.

K-D Tree: Removal

Re-build the whole thing?

Re-build the sub-tree of the removal point?

Replace the point with the closest child?

The absolute simplest way to remove a point from the tree is to just re-build the whole thing. This is obviously wasteful and can easily be improved upon.

The next simplest method is to find the point that's being removed and re-build its entire sub-tree.

One other method to remove a point is to replace it with a child in the sub-tree. The typical way to do this is to either replace it with largest value less than it or the smallest value larger than it. That is, replace it with 1 of its 2 closest values (on the split axis). The point that replaces has to also be removed from the tree to make this work though. This is performed by repeating this removal process on the replacement point until a leaf node is reached.

K-D Tree: Rebalance

No easy way to update

Can't rotate due to sorting on multiple dimensions

Both insertion and removal will leave the k-d tree unbalanced. Unfortunately, there's no easy way to re-balance a standard k-d tree. Rotations can't be performed as the tree is sorted among multiple dimensions. If a simple rotation was performed then the sorting of sub-trees would be broken with respect to the new root.

There are variations of k-d tree that allow balancing but they are outside the scope of this class. Just know that insertion and removal will unbalance the tree.

K-D Tree: Queries

Point in sphere is the starting point

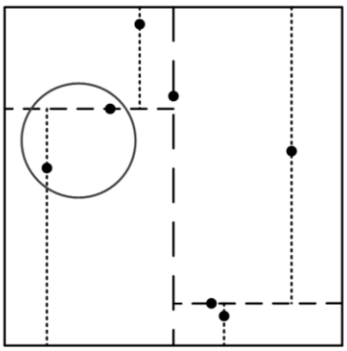
For each node:

- Check split node against sphere
- Recurse down the side the sphere center is on
- If the sphere hits the split plane recurse down the opposite side

The easiest traverse to start looking at is points in sphere, that is find all points that are within a query sphere.

At any given node, we always need to check if the point contained in the node is within the sphere. We also always need to traverse down the side that the sphere's center is on. The question is just do we need to traverse down the opposite side. This is a simple distance check to see if the sphere overlaps the split plane.

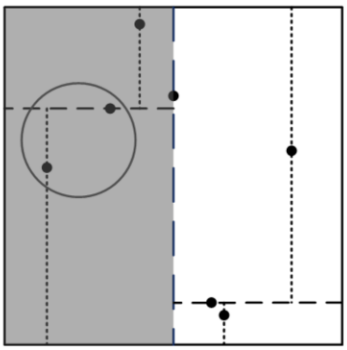
K-D Tree: Queries



Now for an example of checking this sphere against the given k-d tree.

K-D Tree: Queries

Sphere doesn't contain root and doesn't intersect other side

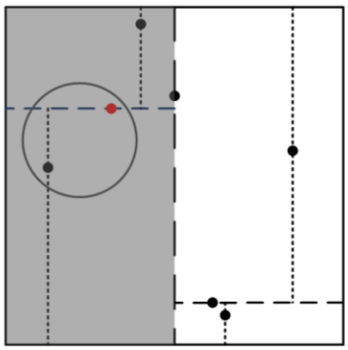


To start with we check the sphere against the root level of the tree. The point contained in the root leaf isn't inside the sphere so we don't add it to our list. Similarly the sphere doesn't intersect the split plane so we only need to recurse down the left side of the tree.

Note here that I'm shading the portions of space we actually check against.

K-D Tree: Queries

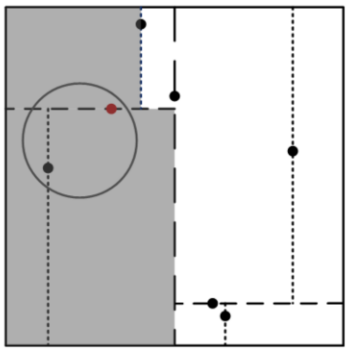
Sphere contains the point and intersects both sides



Now we check the next level of the tree. The sphere contains the point in this node (marked in red). We obviously need to recurse down the side of the tree the sphere center is on (the bottom) but since the sphere intersects the split plane we also need to recurse down the top side.

K-D Tree: Queries

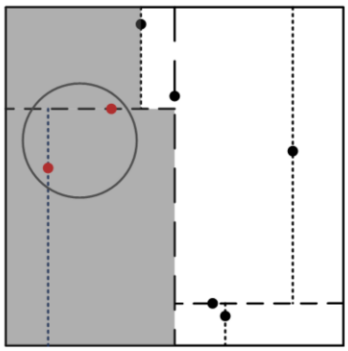
Sphere doesn't contain the point and doesn't intersect the other side



Now we traverse down the top side first (this order doesn't matter). In this case the sphere doesn't contain the point on the split plane and it doesn't intersect the plane. Since this is also a leaf node there's nothing more to do and iteration stops.

K-D Tree: Queries

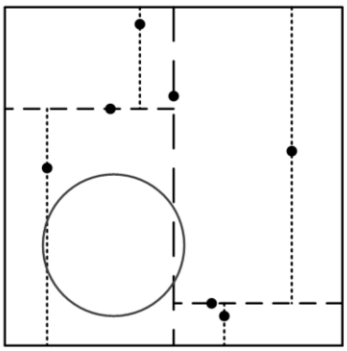
Sphere contains the point and intersects both sides



Now we can go back to the other side of the tree. In this case the sphere contains the point and intersects both sides. However, as this is a leaf node there's nothing more to do.
We now stop with the sphere query returning 2 points.

K-D Tree: Queries

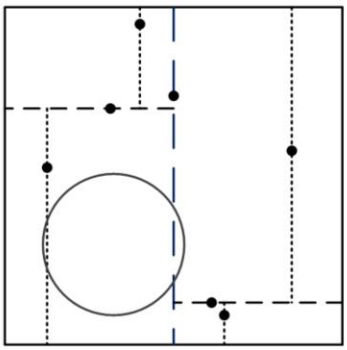
Unfortunately the sphere vs. plane test is insufficient



Unfortunately this test isn't quite sufficient. It'll never miss any results but it will spend more time traversing down the tree than it should. To see why we can look at another example.

K-D Tree: Queries

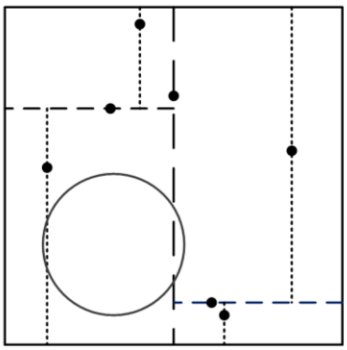
Test the root. Recurse down both sides (let's look at the right first)



To start with the sphere doesn't contain the point on the plane, but it does intersect both sides. To make my point quicker I'm going to recurse into the right side first.

K-D Tree: Queries

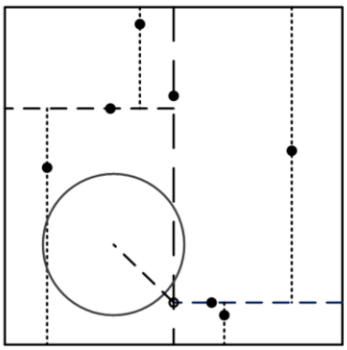
What does this plane test return?



Now we test the sphere against the highlighted plane. Unfortunately when performing this test the sphere does intersect the plane. So in this case we'll recurse down the bottom side even though the sphere can't possible intersect anything inside.

K-D Tree: Queries

Instead test against point on the plane closest to the sphere center.



One solution to fix this (as mentioned by the orange book) is to find the point on the plane closest to the sphere center when traversing down the side the sphere is not on.

This is simply performed by clamping the sphere center at each iteration to the plane. This point is passed down each time and clamped against the new plane. In so doing we'll end up with the displayed point when testing. Now by performing a simple point-in-sphere test instead of sphere intersecting plane we will determine to not recurse down the bottom side of the tree.

K-D Tree: Queries – Nearest Neighbor Search

Basically a point-in-sphere query where we shrink the sphere

Initialize the sphere to the query point with radius to the root point

For each node

1. If the node's point is closer then store that
2. Recurse down the side the center is on
3. If the sphere intersects the plane then recurse down that side

Perhaps a more common operation with k-d trees is the nearest neighbor search. That is, given a point find what other point is closer. Implemented naively this would be an $O(n)$ search but with the k-d tree this becomes $O(\log_2(n))$.

The reason why we started with querying a sphere instead of nearest neighbor search is that how we query a sphere is basically how we perform nearest neighbor search. The basic idea is to construct a sphere centered at our query point with a radius equal to the distance between this point and the root's point.

At any given node we now have the following steps:

1. If the node's point is closer than the old closest point then we update the closest point (and our implicit sphere).
2. Recurse down the side of the tree that the query point is on
3. If the distance to the other plane is closer than our current closest point then recurse down that side.

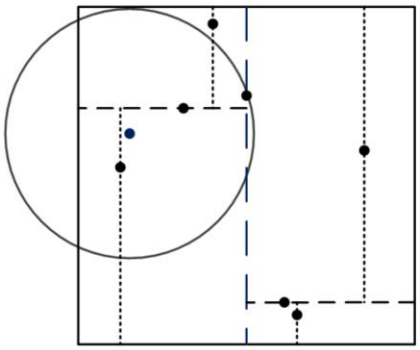
One way to look at this is that we send a sphere down the tree and as we find closer points we shrink the sphere.

Do note that you still have to be careful of checking a recursing down a plane where the sphere hits the infinite plane that should've been clipped by the parent nodes. Also note that these steps can be slightly re-ordered by switching steps 1 and 2 around (not 3). Basically do we update before recursing or after? In either case it's whichever we think will shrink the sphere faster.

K-D Tree: Queries – Nearest Neighbor Search

Initialize the sphere center and radius

Recurse down the side the sphere center is on



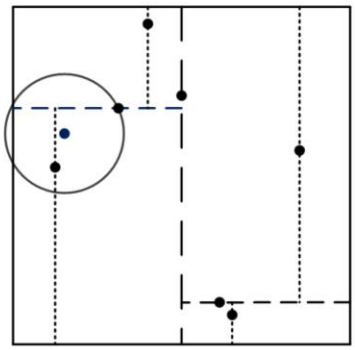
Now for an example. As said before we start by initializing the sphere to be centered at our query point (pictured in dark blue) with a radius equal to the distance between the center and the point in the plane. Not that this is not the same as the distance to the plane.

Now we start by recursing down the side the sphere center is on, in this case the left side.

K-D Tree: Queries – Nearest Neighbor Search

The split point is closer than our previous one, update it.

Recurse down the side the sphere is on

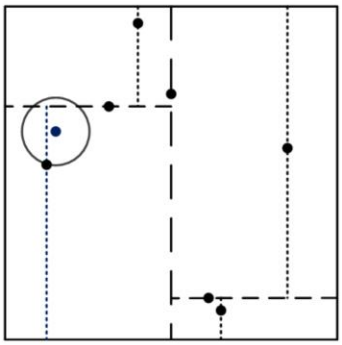


Now at this level the split point is closer than the previous point so we update our sphere to be smaller. Then we continue to recurse down the side the sphere center is on, in this case the bottom.

K-D Tree: Queries – Nearest Neighbor Search

The split point is closer than our previous one, update it.

This is a leaf so stop recursing

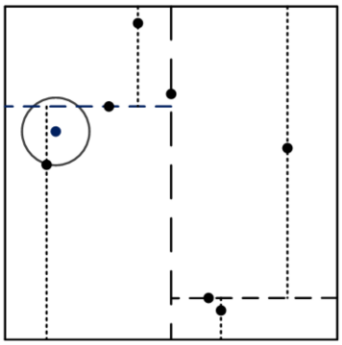


Once again the split point is closer so shrink the sphere. This time however we're at a leaf node so recursion stops and we return back up to the parent node.

K-D Tree: Queries – Nearest Neighbor Search

Check if the sphere intersects the other side

It does so recurse



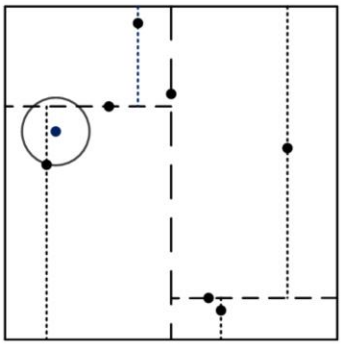
Now we perform the remaining step in our query: checking to see if the sphere intersects the other side. Just because the point on the plane is not closer doesn't mean that we can't find a closer point.

Since the sphere does intersect the split plane still then we need to recurse into the other side.

K-D Tree: Queries – Nearest Neighbor Search

The point isn't closer

The sphere doesn't intersect the plane

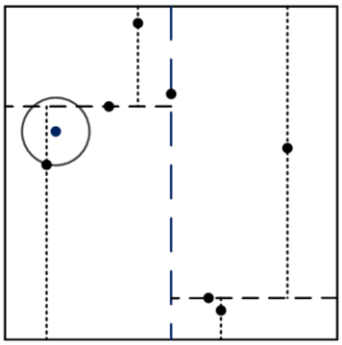


Now we perform all of the steps on this new sub-tree. In this case however note that no more work is necessary and we just return back up the tree.

K-D Tree: Queries – Nearest Neighbor Search

Check if the sphere intersects the split plane

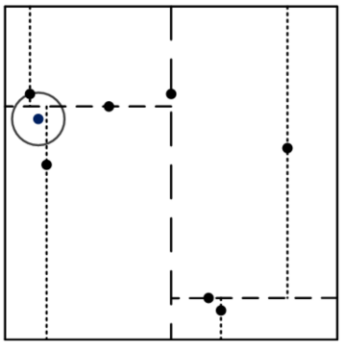
It doesn't so return up



Finally we end back up at the root. Once again we perform the remaining step and check if the sphere intersects the split plane. It does not so we return. Since this is the root we then terminate with the closest point.

K-D Tree: Queries – Nearest Neighbor Search

Note: The sphere center doesn't have to be on the side with the closest point!



Just as a quick example, it's easy to construct a scenario where we must traverse down the other side of the plane to find the closest point.

K-D Tree: Queries – N-Nearest Neighbors

Can find the nearest n-neighbors with almost the same algorithm

Only a few small changes:

- Use a Priority Queue
- Check the distance of the n-th nearest neighbor against the split plane
- If the queue doesn't have n items yet then also recurse down the other side

With a small alteration we can also perform an n-nearest neighbor search, that is find the closest n-points to our query point.

There's only a few things to alter here. First we have to store a priority queue of our closest n-points sorted by distance (or squared distance) from the query point. Now when we check the hyper sphere to see if it crosses the split plane we have to check the largest hyper sphere. That is, if we have 2 distances for closest points in a 2-nearest neighbor search of 1.0 and 2.0, we have to traverse the other side if the plane is within a distance of 2.

The other small alteration that has to be performed is to always traverse the other side if we don't yet have our max number of query points. That is, even if the largest hypersphere doesn't intersect the split plane, we still allow more points so we have to check the other side.

Questions?