Examining Quadtrees, k-d Trees, and Tile Arrays

Matthew Shelley
School of Computer Science, Carleton University
mshelle1@connect.carleton.ca

Abstract

In the context of two-dimensional video games, it is often necessary to perform collision detection and viewport culling. Such operations may query a collection of objects for a subset which appears within a given range. Due to the frequency and cost of these operations, these range queries must be implemented efficiently.

In particular, three data structures—Quadtrees, k-d Trees, and Tile Arrays—answer range queries with efficient runtime.

This paper describes each of these three data structures including their implementation along with the results of rigorous testing and some further considerations.

1 Introduction

Within a video game, rendering occurs between 30 to 60 times per second [1] while collision detection can be even more frequent. Due to the cost of these operations, it is important to reduce their workload by ignoring unnecessary objects.

Problem Definition

Formally, for a given a set of objects—each represented as either a two-dimensional point $(o.x, o.y)$ or as a two-dimensional axis-aligned bounding box with center $(o.c.x, o.c.y)$ and half-dimensions $(o.hd.x, o.hd.y)$—return the subset of objects which are located within the range specified by the two-dimensional axis-aligned bounding box with center $(r.c.x, r.c.y)$ and half-dimensions $(r.hd.x, r.hd.y)$.

Assumptions

It is assumed that all data structures described in the paper store static objects; their positions will never change. Additional assumptions are provided on a per-data-structure basis.

Data Structures

Quadtrees, k-d Trees, and Tile Arrays provide efficient solutions for the range query problem.

A Quadtree is a tree whose internal nodes each have four children which subdivide their parent into four equal areas. Each node can contain a maximum number of objects before it must split.

A k-d (k-dimensional) Tree is similar to a binary tree in that it splits a set of values into two parts. Unlike a binary tree however, each level of the k-d tree splits by alternating axes.

A Tile Array maps every (x, y) where x and y are integers to a value indicating if an object appears at that location.

These data structures perform well despite their different, albeit somewhat similar techniques.

Testing

For each data structure, two categories of tests have been created: correctness and performance. Correctness tests verify the actual outcome of a data structure against its intended functionality. Performance tests gather time information for various operations to see how well they run. These tests thoroughly examine each solution.

2 Implementation

Each data structure has been coded in C++ using Microsoft Visual Studio 2008 Express. The associated C++ solution can be run to see the results of correctness and performance tests.

3 Quadtrees

A quadtree is a tree data structure whose internal nodes all have four children. Each node of the tree represents a unique range with its children (if any) containing equal-sized subdivisions. Each quadtree node can contain a maximum number of elements before it must be split [2].

Assumptions

The implementation used in this project makes the additional assumption that every object is an axis-aligned bounding box. This assumption is reasonable as every object can be contained within an axis-aligned bounding box.

Were more complex collision detection to be performed, the quadtree implementation could be extended to return more specific geometry.

Complexity

On average, insertion and query operations can be performed in $O(logn)$ time with $O(n)$ space [3]. At worst, query and remove operations can take $O(n)$ time when using a range [2].

Example

Below, a quadtree has been generated given a collection of objects (each indicated with an X).

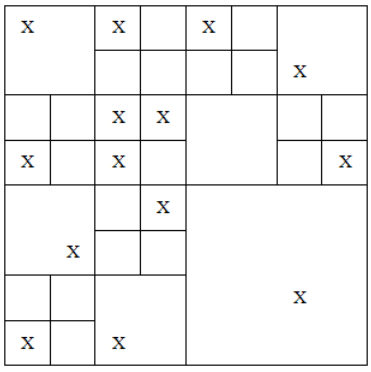

Figure 1. Objects placed in quadtree cells.

Methods Implemented

The following methods have been implemented with the help of [4].

bool insert(AABB *object) – Given a pointer to an object, add that object to the appropriate node within the quadtree. The method returns true if the object was added and false otherwise.

An object may not be added to the quadtree for two reasons. The primary reason is that the object simply falls outside of the boundary. The secondary reason is that the node which should contain the object has no more space and its children cannot contain the object because it is too large. An object which is not added will not influence any range or collision query as the quadtree will simply have no knowledge of it.

void subdivide() – Whenever an object is to be added to a leaf node which is full, this method will split that leaf into four quadrants. Any objects which can be fully contained within one of the new children will be moved to that child to free up space at the current node. Combined, the new quadrants can hold four times the number of objects as the original leaf.

bool removeExact(AABB *object) – Given a pointer to an object, the quadtree is traversed to find the node containing the object. Then that node deletes the object from its list. The method returns true if the object was found and deleted, otherwise false.

bool remove(AABB range, bool requireFullContainment) – Given a range, all objects within that range are removed from the quadtree. If the requireFullContainment option is set to true, then an object must appear fully within the range – otherwise, any overlap will result in the object being deleted. The method returns whether or not any objects were deleted.

vector<AABB*> queryRange(AABB range) – Given a range, all objects appearing within that range (with any overlap) are returned.

bool queryCollision(AABB object) – This method works similar to queryRange except that it will return true as soon as another object is found to collide with the given object. If no objects are found to collide, false is returned.

4 k-d Trees

A k-d tree is a binary tree which partitions space over k dimensions, alternating its splitting axis at each level of the tree. Each node splits a set of points by finding the median (where applicable) for the current axis [5].

Assumptions

The implementation used in the associated project assumes that all objects within the k-d tree are points with just x and y coordinates.

Furthermore, within the implementation this structure is considered static in the sense that it can only be built and then queried upon. The structure cannot be modified after construction.

These strong assumptions apply to the implementation provided and are thus not representative of all k-d trees. It is certainly possible to make a dynamic structure which supports more complex objects.

Complexity

A k-d tree can be constructed in $O(nlogn)$ time using $O(n)$ space, such that range queries can be solved in $O(logn + k)$ time where k is the number of results [5].

Example

The image to follow shows a range query (the dark grey box) over a collection of points which have been stored in a k-d tree.

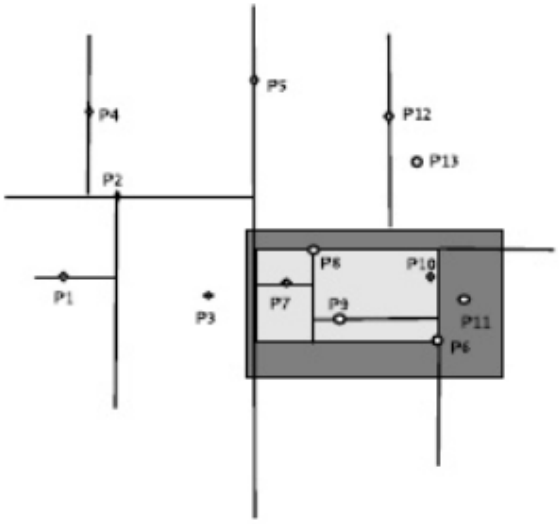

Figure 2. A collection of points with axes split to form a k-d tree [5].

Such a collection would appear as the following k-d tree. Notice that that resulting nodes are highlighted in dark grey.

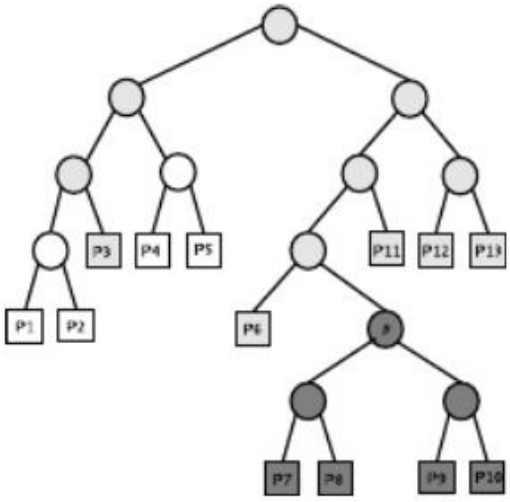

Figure 3. The same collection of points shown as a tree with highlighted query results [5].

It is important to note that the example above places every point at a leaf. The implementation for the associated project places points at all nodes, not just the leaves.

Methods Implemented

As mentioned previously, this data structure has been implemented in a static sense. It can be built once then queried upon, but never changed.

The following methods were implemented with the help of [5] and [6].

construct(AABB _boundary, vector<Vector2f> points, int depth) – This method is called in two ways: either from a newly constructed k-d tree or from a new node of an existing k-d tree. The first version requires just a collection of points as the boundary can be considered infinite by default; in this case, depth defaults to 0. With each newly created k-d tree node, the boundary is set to its appropriate contained area and the depth is incremented by 1. A k-d tree node is constructed for every point that must be split.

vector<Vector2f> queryRange(AABB range) – All points found within a range are returned.

5 Tile Arrays

With a tile-based approach, it is relatively easy to implement an efficient data structure for range queries over a two-dimensional grid using a one-dimensional array. Each element in the array corresponds to exactly one tile, while its value determines whether or not an object is present.

Assumptions

In order to use this solution, some additional assumptions must be made.

1. All objects stored in the array must align with and fill an entire tile.
2. Large objects which do not fill an entire tile must be broken into smaller objects which do fill an entire tile.
3. The dimensions of the grid must be known (referred to as width and height).

Additionally, it is necessary to have a means of converting from a 2D integer coordinate into an index of the 1D array and vice versa.

To convert a 2D coordinate (tx, ty) into an index of the 1D array, we use the formula

$$index = (ty \times width) + tx$$

Conversely, to revert from an index of the 1D array to a 2D coordinate (tx, ty) we set

$$tx = index \,\%\, width$$
$$ty = \lfloor index / width \rfloor$$

Complexity

Insert and remove operations can be performed in $O(1)$ time using $O(width \times height)$ space.

Query operations take $O(\lceil rw \rceil \times \lceil rh \rceil)$ time where *rw* and *rh* correspond to the width and height of the range being queried, respectively. It is worthwhile to note that if *rw* and *rh* are constants, such as the dimensions of the screen or the size of each character, then this method could also be considered constant.

Example

As an example, the images below—which are from my own work—display a tile-based map in full and how it appears when used in a real game. In the second image, the map has been culled by the window to display only the tiles which need to be visible to the player.

Figure 4. An example tile map.

Figure 5. An in-game screenshot using the above tile map to display a range of tiles.

Methods Implemented

The following methods have been implemented.

bool insert(int tx, int ty) – Update the array such that the index representing a tile at (tx, ty) contains the value true. The method returns true if the tile was previously empty, false otherwise.

bool insertFromFile(char* filename) – A file consisting of the characters {0, 1} is read then stored in the array such that all '0' values correspond to an empty tile and all '1' values a filled tile. The method returns false if the file cannot be read, otherwise true.

bool remove(int tx, ty) – This method works exactly like insert, except that the value false is stored at the element representing the position. Conversely, if no object was stored at this tile then false is returned, otherwise true.

bool queryCollision(AABB object) – Given an object represented as an axis-aligned bounding box, all overlapped tiles are examined until another object is found. The method returns whether such an object exists.

vector<TilePosition> queryRange(AABB range) – Given an axis-aligned bounding box which represents a range, all tile positions—stored as two integers (tx, ty)—which contain an object are returned. This method filters out all tiles which do not contain an object.

As a disclaimer, I wish to point out that this solution was independently created by me, but I make no claims that this solution is unique. I believe this solution would be obvious to anyone who has ever seriously considered a tile-based map. For this reason, I have avoided referencing other literature so I could focus specifically on my solution.

6 Testing

For each data structure, tests have been divided into correctness and performance categories.

Testing Correctness

As each data structure has its own specific cases which need to be handled to verify correctness, personalized code has been developed for each structure. The goal of these tests is to find which operations may be incorrect.

It is assumed that an operation is correct if its correctness test passes. Should a test fail, it is unnecessary to record the performance of the faulty associated operation.

Testing Performance

Each performance test has been instrumented with a timer to record the duration of each applicable operation therein.

Insertion – Objects are generated and then inserted into each data structure in quantities of 1, 10, 100, 1000, and 10000.

Removal – All objects which have been inserted from the previous test are deleted from the structure either by the removal of an all-encompassing range or by removing objects on an individual basis.

Do note that k-d trees have not been tested for removal as this aspect was not implemented.

Query – Various range queries are performed on the data structure with the inserted objects.

Recording Time

Time is recorded via code by having a timer start and end immediately before and after each operation block. For instance, an individual insertion would be setup as *{Begin Timer, Insert Objects, End Timer}* while a series of insertions would be setup as *{Begin Timer, Insert Object, …, Insert Object, End Timer}*. The timer used was found at [7].

Environment

All tests were run on an HP HDX X16 1160US laptop with a 2.4GHz Intel Core 2 Duo P8600 processor and 6GB RAM using Windows 7 Home Premium 64-bit edition.

7 Results

Tests for correctness and performance have been instrumented on a per-data-structure basis. The results of these personalized tests are as follows.

Quadtrees

For quadtrees, three tests have been used: correctness, insertion, and removal. Query time was not recorded due to an error in correctness.

Correctness – The following tests have been implemented in the order they appear. While some tests may seem identical or very similar the state of the data structure would be different.

Test	Pass / Fail
Construct	Pass
Insert	Pass
For-Sure Collision	Pass
For-Sure-Not Collision	Pass
For-Sure Query Range	Pass
For-Sure-Not Query Range	Pass
Remove Exact	Pass
Invalid Remove Exact	Pass
After Remove Exact Range Query (with no results)	Pass
Insert Again	Pass
Remove Range	Pass
Invalid Remove Range	Pass
Insert [0]	Pass
Insert [1]	Pass
Insert [2]	Pass
Insert [3]	Pass
Insert with Subdivision	Pass
Query Range for All	Fail
Remove Range	Pass
Query Range for All	Pass

With the exception of "Query Range for All" after a subdivision, all tests passed.

Insertion – A collection of 10000 randomized objects of half dimensions (1.0, 1.0) within the bounds of (-100.0, -100.0) and (100.0, 100.0) were generated then inserted into five different quad trees in the amounts of 1, 10, 100, 1000, and 10000. Rejections and time were recorded.

For mass insertion, it was found that the number of rejections appeared to increase exponentially while the total time increased linearly.

Objects	Actual Inserted	Total Time (ms)
1	1	0.008555
10	10	0.080840
100	95	0.723711
1000	881	9.231168
10000	5537	83.748619

The insertion time of each object was also recorded to find the average and maximum. The minimum time is not worth mentioning as it is likely associated with the first object inserted.

# Objects	Avg Time (ms)	Max Time (ms)
1	0.005133	0.005133
10	0.008084	0.012832
100	0.007618	0.016254
1000	0.010478	0.034218
10000	0.015125	1.198914

Discrepancies in time (most noticeable with one object) are most likely the result of the for-loop needed to traverse the set of objects.

Removal – After inserting all of the objects from the previous test, remove was called using a range which covered the entire quadtree boundary. Only the total time was recorded.

# Objects	Actual Removed	Time (ms)
1	1	0.002994
10	10	0.016681
100	95	0.050899
1000	881	0.351590
10000	5537	2.587738

Of interest, removal performed far better than insertion despite removing the same number of objects as had been inserted. The reason for this difference is that remove needs to only explore every node once where as a single insertion may explore $O(logn)$ nodes alone.

Query – Due to the correctness test for queries failing, no performance data is available as the results would be inaccurate. It is expected that query time would have been similar to removal time as both algorithms are nearly identical.

k-d Trees

Due to k-d trees being implemented as a static data structure, only correctness, construction, and query tests have been created.

Correctness – The following tests have been implemented in the order they appear.

Test	Pass / Fail
Construct with 1 point	Pass
For-Sure Query Range	Pass
For-Sure-Not Query Range	Pass
Construct with 10 points	Pass
For-Sure Query Range	Fail
For-Sure-Not Query Range	Pass

It was determined that a query range would fail when given a larger data set because it would return duplicate results. Despite this problem, a query test as still used as the operation was considered accurate enough to be worth testing.

Construction – A collection of 10000 randomly generated points having positions residing between (-100.0, -100.0) and (100.0, 100.0) were inserted into five k-d trees in the amounts of 1, 10, 100, 1000, and 10000. The total time for each mass insertion was recorded.

Points	Avg Time (ms)	Total Time (ms)
1	0.039778	0.039778
10	0.100344	1.003444
100	0.130893	13.089250
1000	0.194125	194.125278
10000	0.307843	3078.429427

Query – A single query with a range intended to cover the entire k-d tree was run after each construction. It was found that some results were duplicated in larger data sets. Despite this problem, these times were recorded.

# Objects	Time (ms)
1	0.043628
10	0.519259
100	6.511263
1000	75.470423
10000	774.185408

The time for each query appears to grow linearly in *n*, with a higher cost compared to quadtrees.

Tile Arrays

For tile arrays, correctness, insertion, and query tests have been used. It was not worthwhile to test removal as the method works exactly the same as insertion save for one trivial difference.

Correctness – As with the other two data structures, all correctness tests were run in the order they appear. Every test passed.

Test	Pass / Fail
Construct	Pass
Insert	Pass
Insert and Reject	Pass
Remove	Pass
Invalid Remove	Pass

Insert Previously Removed	Pass
For-Sure Collision	Pass
For-Sure-Not Collision	Pass
Insert [1]	Pass
Insert [2]	Pass
For-Sure Query Range	Pass
For-Sure-Not Range Query	Pass
Insert from File	Pass

Insertion – A tile array of dimensions 100 by 100 was created to be filled with 10000 objects. Each object was inserted in order as the position of a given tile does not affect runtime; each insertion runs in $O(1)$ time. The time to insert 10000 objects was recorded then modified to show the time for the intervals 1, 10, 100, 1000, and 10000. Total insertion time is affected only by the number of insertions and not the number of objects already stored in the tile array.

# Objects	Time (ms)
1	0.000008
10	0.000083
100	0.000830
1000	0.008302
10000	0.830215

Removal – As the remove method works exactly the same as the insertion method, save for changing a value to false instead of true, it is sufficient to refer to the times above.

Query – Using a fully populated tile array of dimensions 100x100, square ranges with half dimensions $\{1, 5, 10, 25, 50\}$ were queried.

Range Half-Dimensions	Time (ms)
1x1	0.103082
5x5	0.319511
10x10	0.866999
25x25	4.365792
50x50	16.621406

Additionally, a special range of 20x15 was queried which ran in 2.894417ms. This range represents the width and height in tiles of a video game screen using 640x480 pixels at 32x32 pixels per tile.

8 Discussion and Conclusion

This paper has examined three data structures which solve range queries for the purpose of reducing the workload of collision detection and rendering in two-dimensional video games by filtering out unnecessary objects. These data structures were implemented in C++ and then tested for both correctness and performance.

Having performed rigorous correctness and performance tests, it is evident that both the quadtree and k-d tree implementations could use significant improvement. For one, neither truly had a successful query method. Two, quadtrees rejected many objects due to their size. Three, k-d trees were only implemented in a static sense. Finally, the performance of k-d trees was poor when compared to quadtrees. While the data collected is only preliminary, quadtrees may be a better solution in the realm of video games.

The tile array data structure performed very well as was expected given its simple nature. For a typical game screen, it could take just 2.9 milliseconds to determine exactly which tiles needed to be drawn regardless of the map size. Culling for collision detection against a small object could be performed very quickly as only the immediate tiles around the object would be checked. There are three downsides to this data structure, however: the additional assumptions, the storage requirements, and the filtering of empty tiles. For such a simple approach, this data structure is efficient.

My suggestion to solving range queries for two-dimensional video games, having created these three data structures and tested them thoroughly, would be to use tile arrays for static tiles and perhaps quadtrees for dynamic entities.

References

[1] Andreev, D. 2010. Real-time framerate up-conversion for video games. In SIGGRAPH 2010 Talks.

http://dl.acm.org/citation.cfm?id=1837047

[2] Quadtrees – Hierarchical Grids. Seminar in Algorithms, Spring 2012.

http://www.cs.tau.ac.il/~haimk/seminar12b/Quadtrees.pdf

[3] R. A. Finkel and J. L. Bentley. 1974. Quad Trees: A Data Structure for Retrieval on Composite Keys.

http://www.springerlink.com/content/x7147683u3241843/fulltext.pdf

[4] BYTES. "'Generic' QuadTree implementation." Accessed on April 1, 2012.

http://bytes.com/topic/c-sharp/insights/880968-generic-quadtree-implementation

[5] Kakde, H. 2005. Range Searching using Kd Tree.

http://www.scribd.com/doc/26091526/Range-Searching-Using-Kd-Tree

[6] Wikipedia. "k-d Tree." Accessed on April 16, 2012.

http://en.wikipedia.org/wiki/K-d_tree#Operations_on_k-d_trees

[7] CodeGuru. "C++ Profiling: How do I determine the speed of a particular function or operation?" Accessed on April 19, 2012.

http://forums.codeguru.com/showthread.php?t=291294