**[Collision detection (part 3): Benchmarks](https://0fps.net/2015/01/23/collision-detection-part-3-benchmarks/)**

Posted on [January 23, 2015](https://0fps.net/2015/01/23/collision-detection-part-3-benchmarks/)by [mikolalysenko](https://0fps.net/author/mikolalysenko/" \o "View all posts by mikolalysenko)

Previously in this series we covered the basics of collision detection and discussed some different approaches to finding intersections in sets of boxes:

* [Collision detection (part 1): Overview](https://0fps.net/2015/01/07/collision-detection-part-1/)
* [Collision detection (part 2): Box intersection](https://0fps.net/2015/01/18/collision-detection-part-2/)

Today, we’ll see how well this theory squares with reality and put as many algorithms as we can find to the test.  For those who want to follow along with some code, here is a link to the accompanying GitHub repo for this article:

[The Great JavaScript Box Intersection Benchmark](https://github.com/mikolalysenko/box-intersect-benchmark)

Finding modules

To get a survey of the different ways people solve this problem in practice, I searched GitHub, Google and npm, and also took several polls via IRC and twitter.  I hope that I managed to cover most of the popular libraries, but if there is anything here that I missed, please leave a comment and let me know.

While it is not an objective measurement, I have also tried to document my subjective experiences working each library in this benchmark.  In terms of effort, some libraries took far less time to install and configure than others.  I also took notes on libraries which I considered, but rejected for various reasons.  These generally could be lumped into 3 categories:

1. **Broken:**The library did not report correct results.
2. **Too much work**: Setting up the library took too long.  I was not as rigorous with enforcing a tight bound here, as I tended to give more generous effort to libraries which were popular or well documented.  Libraries with 0 stars and no README I generally skipped over.
3. **Irrelevant**: While the library may have at first looked like it was relevant, closer inspection revealed that it did not actually solve the problem of box intersection.  This usually happened because the library had a suspicious name, or because it was some framework whose domain appeared to include this problem.

A word on JavaScript

For the purpose of this benchmark, I limited myself to consider only JavaScript libraries. One major benefit of JavaScript is that it is easier to install and configure JavaScript libraries, which greatly simplifies the task of comparing a large number of systems.  Also, due to the ubiquity of JavaScript, it is easy for anyone to replicate these results or rerun these benchmarks on their own machine.  The disadvantage though is that there are not as many mature geometry processing libraries for JavaScript as there are for older languages like C++ or Java.  Still, JS is popular enough that I had no trouble finding plenty of things to test although the quality of these modules turned out to be wildly varying.

Implementations surveyed

Brute force

As a control I implemented a simple brute force O(n^2) algorithm in the obvious way.  While it is not efficient for large problem sizes, it performs pretty well up to few hundred points.

Bounding volume hierarchy modules

I found many modules implementing bounding volume hierarchies, usually in the form of R-trees.  Here is a short summary of the ones which I tested:

* [rbush](https://github.com/mourner/rbush):  This is one of the fastest libraries for box intersection detection.  It also has a very simple API, though it did take a bit of time tuning the branching factor to get best performance on my machine.  *I****highly recommend****checking this one out!*
* [rtree](https://github.com/leaflet-extras/RTree): An older rtree module, which appears to have largely been replaced by rbush.  It has more features, is more complicated, and runs a bit slower.  Still pretty easy to use though.
* [jsts-strtree](https://github.com/bjornharrtell/jsts):  The [jsts](https://github.com/bjornharrtell/jsts) library is a JavaScript port of the [Java Topology Suite](http://www.vividsolutions.com/jts/JTSHome.htm).  Many of the core implementations are solid, but the interfaces are not efficient.  For example, they use the [visitor pattern](http://en.wikipedia.org/wiki/Visitor_pattern) instead of just passing a closure to enumerate objects which requires lots of extra boilerplate.  Overall found it clumsy to use, but reasonably performant.
* [lazykdtree](https://github.com/0x0539/kdtree): I included this library because it was relatively easy to set up, even though it ended up being very slow.  Also notable for working in any dimension.

Quad trees

Because of their popularity, I decided to make a special section for quad trees.  By a survey of modules on [npm](https://www.npmjs.com/search?q=quad+tree)/[GitHub](https://github.com/search?l=JavaScript&q=quad+tree&type=Repositories&utf8=%E2%9C%93), I would estimate that quad trees are one of the most commonly implemented data structures in JavaScript. However it is not clear to me how much of this effort is serious.  After much searching, I was only able to find a small handful of libraries that correctly implemented quad trees based rectangle stabbing:

* [simple-quadtree](https://www.npmjs.com/package/simple-quadtree): Simple interface, but sluggish performance.
* [jsts-quadtree](https://github.com/bjornharrtell/jsts):  Similar problems as jsts-strtree.  Unlike strtree, also requires you to filter out the boxes by further pruning them against your query window.  I do not know why it does this, but at least the behavior is well documented.

Beyond this, there many other libraries which I investigated and rejected for various reasons.  Here is a (non-exhaustive) list of some libraries that didn’t make the cut:

* [Google’s Closure Library](https://github.com/google/closure-library/blob/eeb8e18b4c9fd35c9ad38aded885ae0e629945ca/closure/goog/structs/quadtree.js): This library implements something called “quadtree”, but it doesn’t support any queries other than set membership.  I am still not sure what the point of this data structure is.
* [quadtree2](https://github.com/burninggramma/quadtree2.js): Only supports ball queries, not boxes
* [Mike Chamber’s QuadTree](https://github.com/mikechambers/ExamplesByMesh/tree/master/JavaScript/QuadTree): Returns incorrect results
* [quadtree-javascript](https://github.com/silflow/quadtree-javascript): Returns incorrect results
* [node-trees](https://www.npmjs.com/package/node-trees): Returns incorrect results
* [giant-quadtree](https://github.com/oztu/giant-quadtree): Returns incorrect results
* [generic-quadtree](https://www.npmjs.com/package/generic-quadtree):  Only implements point-in-rectangle queries, not rectangle-rectangle (stabbing) queries.
* [quadtree](https://www.npmjs.com/package/quadtree): I’m don’t know what this module does, but it is definitely not a quad tree.

Physics engines

Unfortunately, many of the most mature and robust collision detection codes for JavaScript are not available as independent modules.  Instead, they come bundled as part of some larger “physics framework” which implements multiple algorithms and a number of other features (like a scene graph, constraint solver, integrator, etc.).  This makes it difficult to extract just the collision detection component and apply it to other problems.  Still, in the interest of being comprehensive, I was able to get a couple of these engines working within the benchmark:

* [p2.js](https://schteppe.github.io/p2.js/):  A popular 2D physics engine written from the ground up in JavaScript.  Supports brute force, sweep and prune and grid based collision detection.  *If you are looking for a good 2D physics engine, check this one out!*
* [Box2D](http://box2d.org/): Probably the de-facto 2D physics engine, has been extremely influential in realtime physics and game development.  Unfortunately, the quality of the JS translations are much lower than the original C version.  Supports sweep-and-prune and brute force for broad phase collision detection.
* [oimo.js](https://github.com/lo-th/Oimo.js/):  This 3D physics engine is very popular in the THREE.js community.  It implements brute force, sweep and prune and bounding volume hierarchies for collision detection.  Its API is also very large and makes heavy use of object-oriented programming, which comes with some performance tradeoffs. However oimo does deserve credit for being one of the few libraries to tackle 3D collision detection in JavaScript.

I also considered the following physics engines, but ended up rejecting them for various reasons:

* [cannon.js](http://schteppe.github.io/cannon.js/): To its credit, cannon.js has a very clear API and well documented code.  It is also by the [same author](https://github.com/schteppe) as p2.js, so it is probably good.  However, it uses spheres for broad phase collision detection, not boxes, and so it is not eligible for this test.
* [GoblinPhysics](https://github.com/chandlerprall/GoblinPhysics):  Still at very early stages.  Right now only supports brute force collision detection, but it seems to be progressing quickly.  Probably good to keep an eye on this one.
* [PhysicsJS](http://wellcaffeinated.net/PhysicsJS/):  I found this framework incredibly difficult to deal with.  I wasted 2 days trying to get it to work before eventually giving up.  The scant API documentation was inconsistent and incomplete.  Also, it did not want to play nice in node.js or with any other library in the browser, hooking event handlers into all nooks and crannies of the DOM, effectively making it impossible to run as a standalone program for benchmarking purposes.  Working with PhysicsJS made me upset.
* [Matter.js](http://brm.io/matter-js/):  After the fight with PhysicsJS, I didn’t have much patience for dealing with large broken libraries.  Matter.js seems to have many of the same problems, again trying to patch a bunch of weird stuff onto the window/DOM on load, though at least the documentation is better.  I spent about an hour with it before giving up.
* [ammo.js](https://github.com/kripken/ammo.js/)/[physijs](http://chandlerprall.github.io/Physijs/): This is an emscripten generated port of the popular [bullet](http://bulletphysics.org/wordpress/) library, however due to the translation process the API is quite mangled.  I couldn’t figure out how to access any of the collision detection methods or make it work in node, so I decided to pass on it.

Range-trees

Finally, I tried to find an implementation of segment tree based intersection for JavaScript, but searching turned up nothing.  As far as I know, the only widely used implementation of these techniques is contained in [CGAL](https://www.cgal.org/), which comes with some licensing restrictions and also only works in C++.  As a result, I decided to implement of Edelsbrunner & Zomorodian’s algorithm for streaming segment trees myself.  You can find the implementation here:

[box-intersect: Fast and robust d-dimensional box intersection](https://github.com/mikolalysenko/box-intersect)

The code is available under a liberal MIT license and easily installable via [npm](https://www.npmjs.com/).  It should work in any modern CommonJS environment including [browserify](http://browserify.org/), [iojs](https://iojs.org/) and [node.js](http://nodejs.org/).

Testing procedure

In each of these experiments, a set of n boxes was generated, and then sent to each library to compute a checksum of the set of pairs of intersections.  Because different libraries expect their inputs in different formats, as a preprocessing step the boxes are converted into whatever data type is expected by the library.  This conversion step is not counted towards the total running time.  Note that the preparation phase does not include any time required to build associated data structures like search trees or grids; these timings are counted toward the total run time.

Because algorithms for collision detection are output sensitive, care was taken to ensure that the total number of intersections in each distribution is at most O(n), in order to avoid measuring the reporting time for each method.

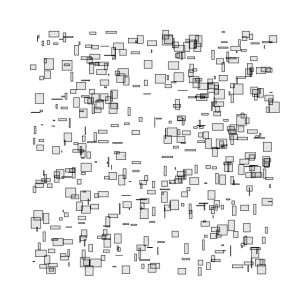
A limitation of this protocol is that it favors batched algorithms (as would be typically required in CAD applications), and so it may unfairly penalize iterative algorithms like those used in many physics engines.  To assess the performance of algorithms in the context of dynamic boxes more work would be needed.

Results

Here is a summary of the results from this benchmark.  For a more in depth analysis of the data, please see the associated GitHub repo.  All figures in this work were made with [plot.ly](http://plot.ly/), click on any of the images to get an interactive version.

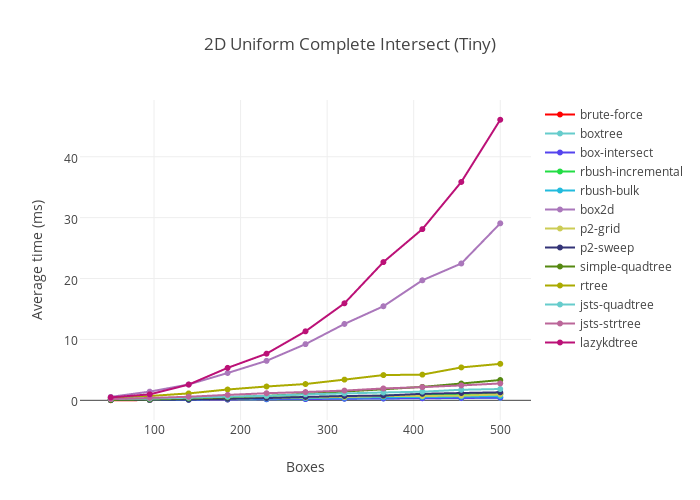
Uniform distribution

I began this study by testing each algorithm against the uniform distribution.  To ensure that the number of intersections is at most O(n), I borrowed a trick from Edelsbrunner and Zomorodian; and scaled the side length of each box to be O(n^{\frac{1}{d}-1}) while constraining the boxes to remain within the unit hypercube, [0,1]^d.  A typical sample from this distribution looks like this,

[](https://0fps.files.wordpress.com/2015/01/uniform.png)

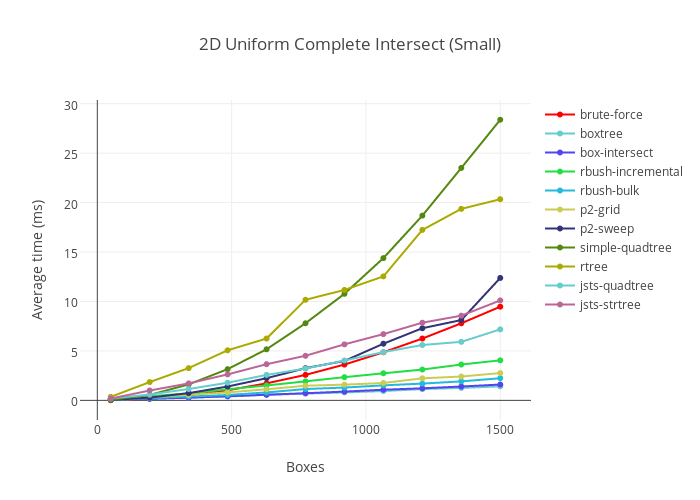
Uniformly distributed boxes

To save time, I split the trials into phases ordered by number of boxes; algorithms which performed took too long on smaller instances were not tested on larger problem sizes.  One of the first and most shocking results was a test instance I ran with just 500 boxes:

[](https://plot.ly/~MikolaLysenko/124/_2d-uniform-complete-intersect-tiny/)

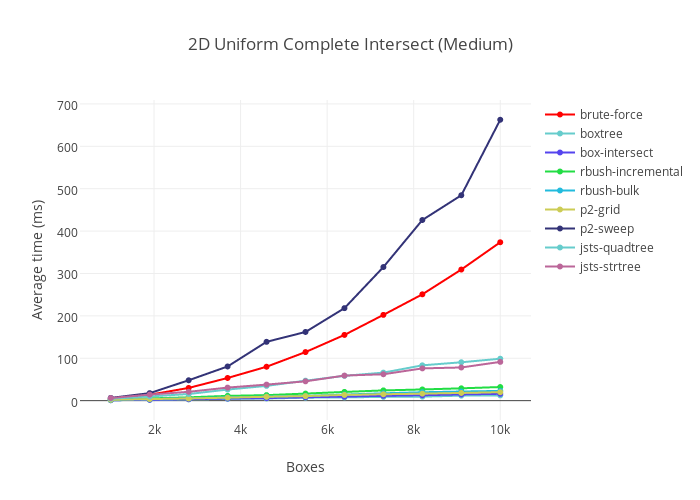
[Click to inspect](https://plot.ly/~MikolaLysenko/124/_2d-uniform-complete-intersect-tiny/)

Here two libraries stand out for their incredibly bad performance: Box2D and lazykdtree. I am not sure why this situation is so bad, since I believe the C version of Box2D does not have these problems (though I need to verify this).  Scaling out to 1500 boxes without these two libraries gives the following results:

[](https://plot.ly/~MikolaLysenko/125/_2d-uniform-complete-intersect-small/)

[Click to inspect](https://plot.ly/~MikolaLysenko/125/_2d-uniform-complete-intersect-small/)

The next two worst performing libraries were simple-quadtree and rtree.  simple-quadtree appears to have worse than quadratic growth, suggesting fundamental algorithmic flaws.  rtree’s growth is closer to O(n \:\mathrm{polylog}(n)), but due to the constants involved is still far too slow.  As it took too long and there were 2 other representatives of the r-tree data structure in the benchmark, I decided to drop it from larger tests.  Moving on to 10k boxes,

[](https://plot.ly/~MikolaLysenko/127/_2d-uniform-complete-intersect-medium/)

[Click to inspect](https://plot.ly/~MikolaLysenko/127/_2d-uniform-complete-intersect-medium/)

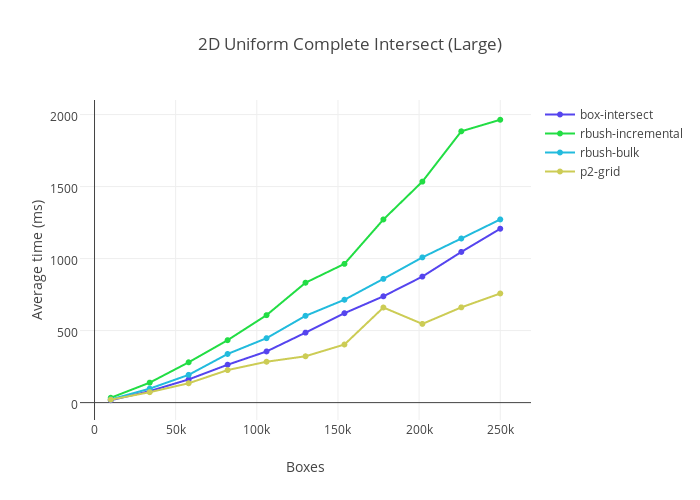
At this point, the performance of brute force begins to grow too large, and so it was omitted from the large brute force tests.

Because p2’s sweep method uses an insertion sort, it takes O(n^2) time with a cold start.  This causes it to perform close to brute force in almost all cases which were considered.  I suspect that the results would be better if p2 was used from a warm start or in a dynamic environment.

Both of jsts’ data structures continue to trend at about O(n \log(n)) growth, however because the constants involved are so large they were also dropped from the large problem size.

10000 boxes appears to be the cut off for real time simulation (or 60fps) at least on my machine.  Beyond, this point, even the fastest collision detection algorithm (in this case box-intersect) takes at least 16ms.  It is unlikely that any purely JavaScript collision detection library would be able to handle significantly more boxes than this, barring substantial improvements in either CPU hardware or VM technology.

One surprise is that within this regime, box-intersect is consistently 25-50% faster than p2-grid, even though one would expect the O(n) complexity of the grid algorithm to beat the O(n \log^2(n)) time of box-intersect.  An explanation for this phenomenon would be that box-intersect enjoys better cache locality (scaling linearly with block size), while hashing causes O(1) indirect main memory accesses for each box.  If the size of a cache line is B, then the cross over point should occur when n \approx \Theta(2^{\sqrt{B}}) in 2D.  This is illustrated in the following chart which carries out the experiment to 250k boxes,

[](https://plot.ly/~MikolaLysenko/129/_2d-uniform-complete-intersect-large/)

[Click to inspect](https://plot.ly/~MikolaLysenko/129/_2d-uniform-complete-intersect-large/)

As expected, somewhere between 10000 and 20000 boxes, p2-grid finally surpasses box-intersect.  This is expected as grids realize O(n) complexity for uniform distributions, which is ultimately faster than O(n \log^2(n)) for sufficiently large n.

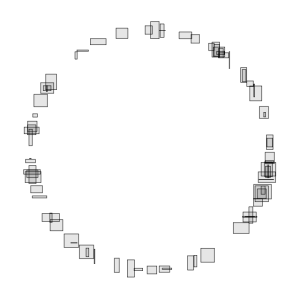
The complexity of rbush on this distribution is more subtle.  For the bulk insertion method, rbush uses the “overlap minimizing tree” (OMT) heuristic of Lee and Lee,

T. Lee, S. Lee. (2003) “[OMT: Overlap minimizing top-down bulk loading algorithm for R-Trees](http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-74/files/FORUM_18.pdf)” CAiSE

The OMT heuristic partitions the space into an adaptive R \times R grid at each level of the tree along the quantiles.  In the case of a uniform distribution of boxes, this reduces to uniform grid giving a query time of O( R^2 \log_{R^2}(n) + k), which for finite R gives means that rbush will find all intersections in O(n \log(n) + k) time.  As a result, we can expect that once n \approx \Theta(2^B), rbush-bulk should eventually surpass box-intersect in performance (though this did not occur in my benchmarks).  This also suggests another way to interpret the OMT heuristic:  it is basically a hedged version of the uniform grid.  While not quite as fast in the uniform case, it is more adaptive to sparse data.

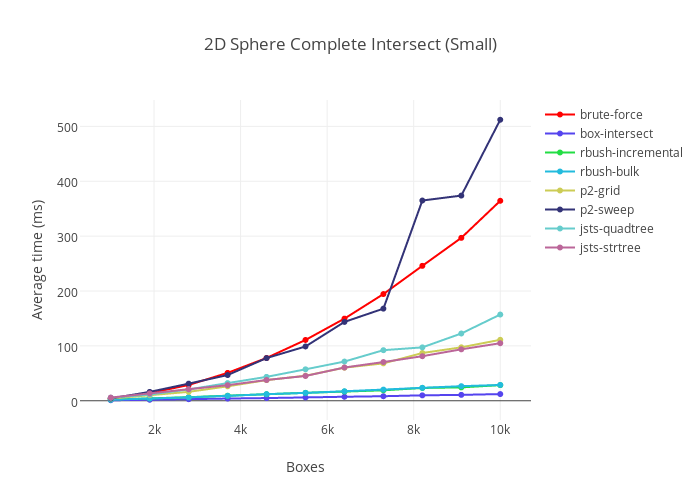
Sphere

Of course realistic data is hardly ever uniformly distributed.  In CAD applications, most boxes tend to be concentrated in a lower dimensional sub-manifold, typically on the boundary of some region.  As an example of this case, we consider boxes which are distributed over the surface of a sphere (again with the side lengths scaled to ensure that the expected number of collisions remains at most O(n)).

[](https://0fps.files.wordpress.com/2015/01/sphere.png)

A collection of boxes distributed over the boundary of ball.

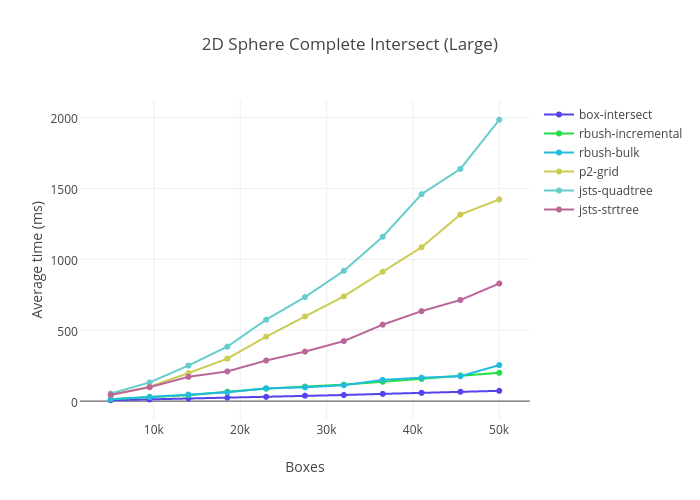
To streamline these benchmarks, I excluded some of the worst performing libraries from the comparison.  Starting with a small case, here are the results:

[](https://plot.ly/~MikolaLysenko/130/_2d-sphere-complete-intersect-small/)

[Click to inspect](https://plot.ly/~MikolaLysenko/130/_2d-sphere-complete-intersect-small/)

Again, brute force and p2’s sweep reach similar O(n^2) performance.

More significantly, p2-grid did not perform as well as in the uniform case, running an order of magnitude slower.  This is as theory would predict, so no real surprises.  Continuing the trend out to 50k boxes gives the following results,

[](https://plot.ly/~MikolaLysenko/131/_2d-sphere-complete-intersect-large/)

[Click to inspect](https://plot.ly/~MikolaLysenko/131/_2d-sphere-complete-intersect-large/)

Both p2-grid and jsts-quadtree diverge towards O(n^2), as grid based partitioning fails for sparse data.

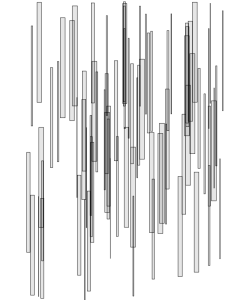
Both rbush and jsts-strtree show similar growth rates, as they implement nearly identical algorithms, however the constant factors in rbush are much better.  The large gap in their performance can probably be explained by the fact that jsts uses Java style object oriented programming which does not translate to JavaScript well.

One way to understand the OMT heuristic in rbush, is that it is something like a grid, only hedged against sparse cases (like this circle).  In the case of a uniform distribution, it is not quite as fast as a grid, suffering a O(\log(n)) penalty, while adding robustness against sparse data.

Again box-intersect delivers consistent O(n \log^2(n) + k) performance, as predicted.

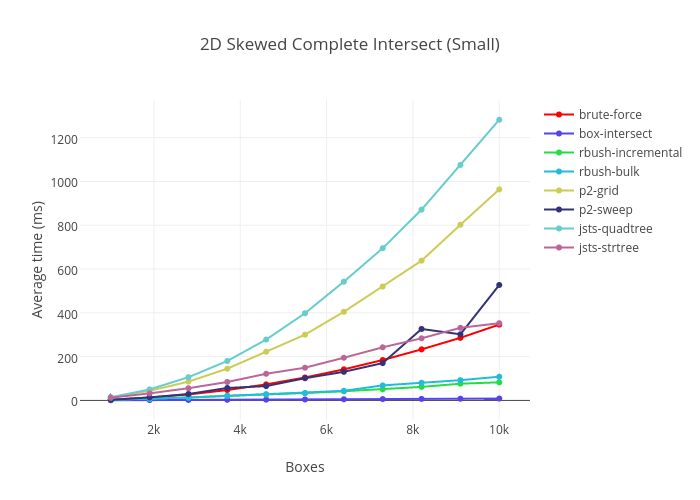
High aspect ratio

Finally, we come to the challenging case of high aspect ratio boxes.  Here we generate boxes which are uniformly distributed in x and and stretched along the y-axis by a factor of O(n):

[](https://0fps.files.wordpress.com/2015/01/skewed.png)

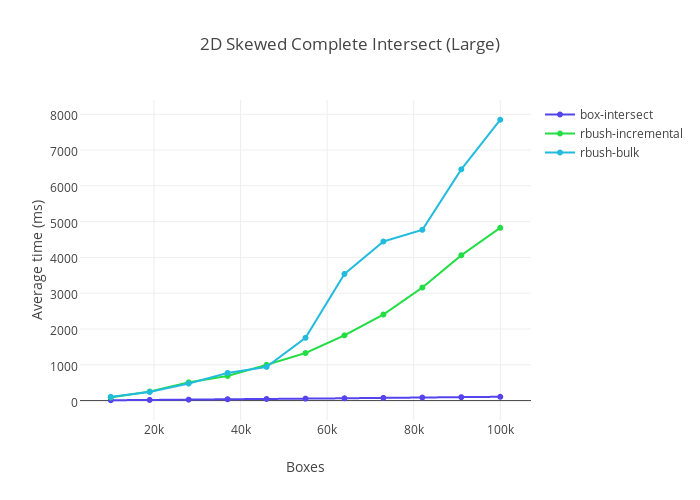
A set of boxes with a high aspect ratio

High aspect ratio boxes absolutely destroy the performance of grids and quad trees, causing exponential divergence in complexity based on the aspect ratio.  Here are some results,

[](https://plot.ly/~MikolaLysenko/132/_2d-skewed-complete-intersect-small/)

[Click to inspect](https://plot.ly/~MikolaLysenko/132/_2d-skewed-complete-intersect-small/)

Even for very small n, **grids take almost a second**to process all intersections for just 10000 boxes.  This number can be made arbitrarily high by choosing as extreme an aspect ratio as one likes.  Here I select an aspect ratio of O(n), forcing the asymptotic growth of both jsts-quadtree and p2-grid to be O(n^2).  If I had selected the aspect ratio as O(2^n) or O(\infty), then it would have been possible to increase their running times to some arbitrarily large value.  In this benchmark, rbush also grows though by a much slower O(n^\frac{3}{2}).  Continuing out to 100k boxes, eventually rbush also fails,

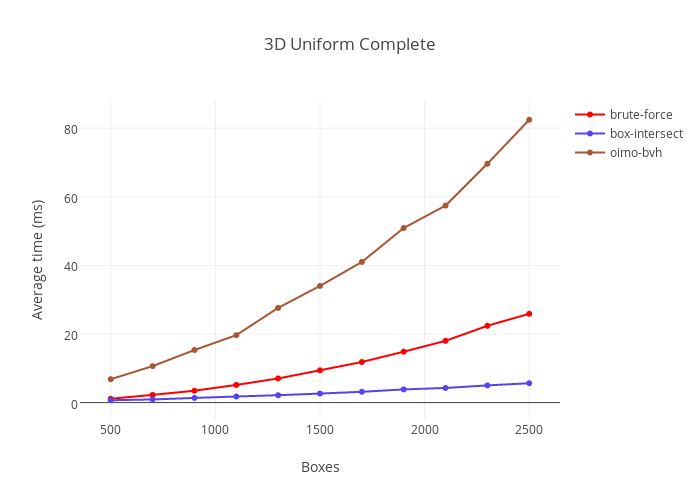
[](https://plot.ly/~MikolaLysenko/139/_2d-skewed-complete-intersect-large/)

[Click to inspect](https://plot.ly/~MikolaLysenko/139/_2d-skewed-complete-intersect-large/)

In this case rbush-bulk takes more than **40x slower** and rbush-bulk more than **70x.** Unlike in the case of grids however, these numbers are only realized by scaling in the number of boxes and so they cannot be made arbitrarily large.  However, it does illustrate that for certain inputs rbush will fail catastrophically.  box-intersect again continues to grow very slowly.

3D

The only libraries which I found that implemented 3D box intersection detection were lazykdtree and oimo.js.  As it is very popular, I decided to test out oimo’s implementation on a few small problem sizes.  Results like the following are typical:

[](https://plot.ly/~MikolaLysenko/152/_3d-uniform-complete/)

[Click here to inspect](https://plot.ly/~MikolaLysenko/152/_3d-uniform-complete/)

Conclusions

For large, uniform distributions of boxes, grids are still the best.  For everything else, use segment trees.

* Brute force is a good idea up to maybe 500 boxes or so.
* Grids succeed spectacularly for large, uniform distributions, but fail catastrophically for anything more structured.
* Quad trees (at least when properly implemented) realize similar performance as grids.  While not as fast for uniform data, they are slightly hedged against boxes of wildly variable size.
* RTrees with a tuned heuristic can give good performance in many practical cases, but due to theoretical limitations (see the previous post), they will always fail catastrophically in at least some cases, typically when dealing with boxes having a high aspect ratio.
* Zomorodian & Edelsbrunner’s streaming segment tree algorithm gives robust worst case O(n \log^d(n) + k) performance no matter what type of input you throw at it.  It is even faster than grids for uniform distributions at small problem sizes(<10k) due to superior cache performance.

Overall, streaming segment trees are probably the safest option to select as they are fastest in almost every case.  The one exception is if you have a large number of uniformly sized boxes, in which case you might consider using a grid.