[**An Analysis of Minecraft-like Engines**](https://0fps.net/2012/01/14/an-analysis-of-minecraft-like-engines/)

Posted on [January 14, 2012](https://0fps.net/2012/01/14/an-analysis-of-minecraft-like-engines/)by [mikolalysenko](https://0fps.net/author/mikolalysenko/)

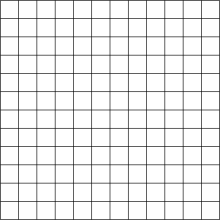
Voxel engines are everywhere…

…and given the huge success of Minecraft, I think they are going to become a permanent fixture in the landscape of game engines.  Minecraft (and Infiniminer) live in a unique space somewhere between high-resolution voxels and static prefabricated environments.  Modifying a Minecraft world is both intuitive and elegant; you simply pick up and place blocks.  In many ways, it is the natural 3D generalization of older 2D tile based games.  Of course creating a Minecraft engine is much more challenging than making a 2D engine for a number of reasons.  First of all, it is in 3D and second it is generally expected that the environments must be much more dynamic, with interactive lighting, physics and so on.

When building a voxel game, it is important to choose a data structure for representing the world early on.  This decision more than any other has the greatest impact on the overall performance, flexibility, and scale of the game.  This post discusses some of the possible choices that can be made along these lines, and hopefully give a better explanation of what sort of technical considerations are involved.  In fact, I posit that some commonly held performance assumptions about Minecraft-type engines are probably wrong. In particular, the importance of random-access reads/writes for modifications is often overestimated, while the relative cost of iteration is greatly underestimated.  In concluding this article, I introduce a new data structure that exploits this observation, and ultimately gives much faster iteration (for typical Minecraft-esque maps) at the expense of slightly slower random accesses.

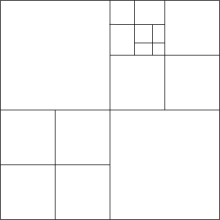
Conventional data structures

But before getting ahead of ourselves, let’s review some of the approaches were are currently being used to solve this problem.  We start with the absolute simplest choice, which is the “flat array”.  In the flat array model, you simply store all of your level data in one gigantic array:

[](https://0fps.files.wordpress.com/2012/01/grid.png)

As Minecraft and Infiniminer haver shown us this strategy can be viable — providing you limit the number of voxel types and make the world small enough.  Storing blocks in a big array has many advantages, like constant time random access reads and writes, but it is far from perfect. The most obvious downside is that the amount of memory consumed grows with the cube of the linear dimension, and that they are ultimately fixed in size.  These properties make them suitable for small, sand-box worlds (like the old-school Minecraft creative mode), but prevent them from being used in an infinite world (like Minecraft survival mode).

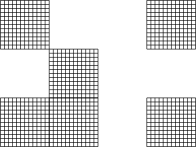
The most naive solution to the perceived disadvantages of flat arrays (and I say this because it is almost invariably the first thing that anyone suggests) is to try putting all the voxels in an [octree](http://www.sea-of-memes.com/LetsCode1/LetsCode1.html).  An octree is a tree that recursively subdivides space into equal sized octants:

[](https://0fps.files.wordpress.com/2012/01/octree.png)

On the face of it, this does sound like a good idea.  Minecraft maps are mostly empty space (or at least piecewise constant regions), and so removing all those empty voxels ought to simplify things quite a bit.  Unfortunately, for those who have tried octrees, the results are typically [not so impressive](http://www.sea-of-memes.com/LetsCode32/LetsCode32.html):  in many situations, they turn out to be orders of magnitude slower than flat arrays.  One commonly cited explanation is that access for neighborhood queries times in octrees are too slow, causing needless cache misses and that (re)allocating nodes leads to memory fragmentation.

However, to understand why octrees are slower for Minecraft games, it isn’t really necessary to invoke such exotic explanations.  The underlying reason for the worse performance is purely algorithmic: each time an arbitrary voxel is touched, either when iterating or performing a random access, it is necessary to traverse to the entire height of the octree.  Since octrees are not necessarily balanced, this can be as much as log(maximum size of game world)!  Assuming the coordinate system is say 32 bit, this brings an added overhead of 32 additional non-coherent memory accesses per each operation that touches the map (as opposed to the flat array, where everything is simply constant time).

A more successful method — and even less sophisticated — method is to use paging or “virtual memory” to expand the size of the flat array.  To do this one breaks down the flat 3D array into a collection of pages or “chunks”, and then uses a hash table to sparsely store only a subset of the pages which are currently required by the game:

[](https://0fps.files.wordpress.com/2012/01/virtual.png)

Pages/chunks are typically assigned in the same way as any virtual memory system: by reserving the upper k-bits of the coordinate for the chunk ID.  Using a virtual  memory system has the additional advantage that chunks can be lazily initialized, which is great for games with procedural content.  The same concept also allows chunks to be mapped back to disk when they are not needed (that is when there are no players nearby).  Using a hash map to store the chunks allows one to maintain constant time random access, while simultaneously taking advantage of sparsity as in an octree.

It seems like the current direction of Minecraft engines has begun to converge on the “virtualization” approach.  (And I base this claim on [numerous](http://www.gavanw.com/) [blog](http://bigbadwofl.blogspot.com/2012/01/infinite-terrain-test.html) [posts](http://techcraft.codeplex.com/)).  But is this really the best solution?  After all, before we jump in and try to solve a problem we should at least try to quantitatively understand what is going on first; otherwise we might as well be doing literary criticism instead of computer science.  So I ask the following basic question:

What factors really determine the performance of a voxel engine?

There is a superificial answer to this question, which is “read/write access times”; after all every operation on a Minecraft map can be reduced to simple voxel queries.  If you accept this claim, then a virtualization is optimal — end of story.  However for reasons which I shall soon explain, this assertion is false.  In fact, I hope to persuade you that really it is the following that is true:

Claim:  The main bottleneck in voxel engines is iteration — not random access reads and writes!

The basic premise hinges on the fact that the vast majority of accesses to the voxel database come in the form of iterations over 3D ranges of voxels.  To demonstrate this point let us first consider some of the high level tasks involving the world map that a voxel engine needs to deal with:

* Placing and removing blocks
* Collision detection
* Picking/raytracing
* Lighting updates
* Physics updates
* Mesh generation
* Disk serialization
* Network serialization

The last two items here could be considered optional, but I shall leave them in for the sake of completeness (since most voxel games require some level of persistence, and many aspire at least to one day be multiplayer games).  We shall classify each task in terms of whatever operations are required to implement it within the game loop, which are either read/write and random access/iteration.  To measure cost, we count the number of raw voxel operations (read-writes) that occur over some arbitrary interval of gameplay.  The idea here is that we wait some fixed unit of time (t) and count up how many times we hit the voxels (ie the number of bytes we had to read/write).  Under the assumption that the world is much larger than cache (which is invariably true), these memory operations are effectively the limiting factor in any computation involving the game world.

At a fairly coarse level, we can estimate this complexity in terms of  quantities in the game state, like the number of MOBs and players in the world, or in terms of the number of atomic events that occur in the interval, like elapsed frames, chunk updates etc.  This gives an asymptotic estimate for the cost of each task in terms of intelligible units:

|  |  |  |
| --- | --- | --- |
| Task | Operation | Cost (in voxel ops) |
| Block placement | Write | number of clicks |
| Collision check | Read | (number of MOBs) \* (frame count) |
| Picking | Read\* | (pick distance) \* (frame count) |
| Lighting | Iteration (RW) | (torch radius)^3 \* (number of torches) + (active world size) \* (sun position changes) |
| Physics update | Iteration (RW) | (active world size) \* (frame count) |
| Mesh generation | Iteration (R) | (chunk size) \* (chunk updates) |
| Disk serialization | Iteration (R) | (active world size) \* (save events) |
| Network IO | Iteration (R) | (active world size) \* (number of players) + (chunk size) \* (chunk updates) |

*\* Assuming that picking is implemented by ray-marching.*

*Disclaimer:  The entries in this chart are estimates, and I do not claim that the dimensions for each task are 100% accurate.  If you disagree with anything here, please leave a comment.*

Our goal is to pinpoint which operation(s) in this chart is the bottleneck, and to do so we need dimensional analysis to convert all our units into a common set of dimensions.  Somewhat arbitrarily, we choose frame count and number players, and so our units will be in *(voxel ops \* frames \* players)*.  Going through the list of each item, we now try to estimate some plausible bounds for each dimension:

1. *(number of clicks)* is the amount of times the user clicked the mouse in the interval.  Since a user can click at most once per frame (and in practice is likely to click much much less), this value is bounded above by:
   * *(number of clicks) < (frame count) \* (number of players)*
2. *(number of MOBs)*is the number of active entities, or movable objects in the game.  Assuming that they are not moving too fast (ie there is a maximum speed of a constant number of blocks per frame), then the number of reads due to a collision check is going to be proportional to the number of MOBs and the number of frames.  We assume that number of MOBs in the world is proportional to the number of players, according to some constant k_{MOB},
   * *(number of MOBs) =* k_{MOB}*\* (number of players)*
3. *(pick distance)* is the maximum range a block can be selected by a player, and we shall assume that it is just a small constant.
4. Again, *(torch radius)* is a small number.  In Minecraft, this could theoretically be as high as 16, but we will simply leave it as a tiny constant for now.
5. *(number of torches)*is the number of torches placed / removed during the interval.  Torch placement is generally an irregular event, and so we estimate that a player will place a torch somewhat infrequently, at a rate goverened by some constant k_{TORCH},
   * *(number of torches) = (number of players) \* (frame count) \** k_{TORCH}
6. *(sun position changes)*is the number of times that the sun changes.  This happens regularly every few frames, and so we estimate that,
   * *(sun position changes) = (frame count) \** k_{SUN}
7. *(active world size)*is the size of the world.  It is basically proportional to the number of chunks visible by each player.  Assuming a Minecraft style world with a height cap, this would vary quadratically with the visible radius, r_V, or cubically for a world with unlimited z-value.
   * *(active world size) = (number of players) \* (chunk size) \* r_V^{2}*
8. *(chunk updates)* occur whenever a chunk changes.  Assuming this is random, but proportional to the size of the active world, we get
   * *(chunk updates) = (active chunk size)  / (chunk size) \* k_{UPDATE}*
9. *(save events)*occur at regular intervals and save the map to disk.  Again, because they are regular we can just bound them by a constant times the number of frames.
   * *(save events) = (frame count) \** k_{SAVE}

Plugging all that in, we get the following we get the following expression for the total amount of random access voxel ops:

*(random voxel ops) = C \* (number of players) \* (frame\_count)*

And for the sequential or iteration voxel ops, we get:

*(sequential voxel ops) = C \* (number of players) \* (frame count) \* (chunk size) \* r_V^2*

In general, the last quantity is much larger, by a factor of *(chunk size) \**r_V^2, or *(chunk size) \**r_V^3 if you have unlimited height.  So, if you believe these estimates are reasonable, then you should be convinced that iteration *by far*dominates the performance of a Minecraft style game.  In fact, this estimate explains a few other things, like why visible radius is such an important performance factor in Minecraft.  Increasing the draw radius linearly, degrades the performance quadratically!

Can we make iteration faster?

The main conclusion to draw from the above estimate is that we should really focus our optimization efforts on iteration.  If we can bring that cost down measurably, then we can expect to see large improvements in the game’s performance.  But the question then becomes: is this even possible?  To show that at least in principle it is, we make use of a simple observation:  In any situation where we are iterating, we only need to consider local neighborhoods around each cell.  For example, if you are computing a physics update (in minecraft) you only consider the cells within your [Moore neighborhood](http://mathworld.wolfram.com/MooreNeighborhood.html).  Moreover, these updates are deterministic:  If you have two voxels with the same Moore neighborhood, then the result of applying a physics update to each cell will be the same.  As a result, if we can iterate over the cells in groups which all have the same neighborhood, then we can apply the same update to all cells in that group at once!

This is the basic idea behind our strategy for speeding up iteration.  We will now try to explain this principle in more detail.  Let B be the set of block types, and let G : \mathbb{Z}^3 \to B be a map representing our world (ie an assignment of block types to integral coordinates on a regular grid).  A radius r*-Moore neighborhood* is an element of the set B^{2r+1} \times B^{2r + 1} \times B^{2r + 1} \cong \{ f : \mathbb{Z}_{2r + 1}^3 \to B \}.  The Moore neighborhood of a coordinate (i,j,k) in G is determined by a map, M_{r} : \mathbb{Z}^3 \to (\mathbb{Z}_{2 r + 1}^3 \to B) such that,

M_r(i,j,k)(o, p, q) = G(i + o - r, j + p - r, k + q - r)

Then an *update rule* is a map sending a Moore neighborhood to a block, P : B^{2r+1} \times B^{2r + 1} \times B^{2r + 1} \to B, and the application of P to G is the rule sending,

G \mapsto P \circ M_r

Then the fact that we can group cells corresponds to the observation that:

M_r(i_1, j_1, k_1) = M_r(i_2, j_2, k_2) \implies P(M_r(i_1, j_1, k_1)) = P(M_r(i_2, j_2, k_2)) 

So, if our world data allows us to group up similar Moore neighborhoods, then we can expect that on average we should be able to reduce the complexity of iteration by a substantial amount.

In fact, this same idea also applies to meshing:  If you have a sequence of adjacent cubes that all have the same 3x3x3 neighborhood, then you can fuse their faces together to make a smaller mesh.  For example, instead of generating a mesh for each cube one at a time:

[](https://0fps.files.wordpress.com/2012/01/mesh1.png)

We can fuse cells with the same Moore neighborhood together:

[](https://0fps.files.wordpress.com/2012/01/mesh2.png)

So if we can iterate over the cells in batches, we should expect that not only will our physics loop get faster, but we should also expect that mesh quality should improve as well!

Run length encoding and interval trees

At this point, we now have some pretty good intuition that we can do iteration faster, so the obvious next step is to figure out exactly how this works.  Here is one strategy that I have used in a simple javascript based Minecraft game (which [you can get here](https://github.com/mikolalysenko/NodeMinecraftThing)).  The idea follows from the simple observation that Minecraft style worlds can be easily run-length-encoded.  In run length encoding, one first flattens the 3D array describing a chunk down to a 1D string.  Then, substrings of repeated characters or “runs” are grouped together.  For example, the string:

    aaaaabbbbacaa

Would become:

     5a4b1a1c2a

Which is a size reduction of about 25%.  You can imagine replacing the characters in the string with voxel types, and then compressing them down this way.  Now, applying run-length-encoding to chunks is by no means a new idea.  In fact, it is even reasonable to do so as a preprocessing step before g-zipping the chunks and sending them over the network/spooling to disk.  The fact that run-length encoding does compress chunks works to our advantage:  We can use this fact to iterate over the voxel set in units of runs instead of units of pixels, and can easily be extended to walk runs of Moore neighborhoods as well.

This sounds great for iterations, but what about doing random-access reads and writes?  After all, we still need to handle collision detection, block placement and raytracing somehow.  If we were dumb about it, just accessing a random block from a run-length-encoded chunk could take up to linear time.  It is obvious that we shouldn’t be happy with this performance, and indeed there is a simple solution.  The key observation is that a run-length encoding of a string is formally equivalent to an [interval tree](http://en.wikipedia.org/wiki/Interval_tree) representation.

An interval tree is just an ordinary binary tree, where the key of each node is the start of a run and the value is the coordinate of the run.  Finding a greatest lower bound for a coordinate is equivalent to finding the interval containing a node.  To do insertion is a bit trickier, but not by much.  It does involve working through a few special cases by hand, but it is nothing that cannot be done without taking a bit of time and a pad of paper.  If we implement this data structure using some type of self-balancing binary tree (for example a red-black tree or a splay tree), then we can perform random reads and writes in logarithmic time on *the number of runs*.  Best of all: we also get improved mesh generation and in-memory compression for free!

Now in practice, you would want to combine this idea with virtualization.  Basically, you would store each chunk as an interval tree, then represent the entire world as a hash map.  The reason for doing this would be to integrate paging and chunk generation more seamlessly.  In fact, this is the method I took in the last two minecraft type games that I wrote, and you can find some example code illustrating this data structure [right here](https://github.com/mikolalysenko/NodeMinecraftThing/blob/master/client/voxels.js).

Comparisons and conclusion

Ok, now that we’ve gone through the details, let’s break down our options by data structure and time complexity:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data structure | Random access time | Iteration time | Size | Notes |
| Flat array | O(1) | O(n) | O(n) |  |
| Virtualized array | O(1) | O(n) | O(v) | v is the number of occupied (virtual) chunks |
| Octree | O(h) | O(n h) | O( v h ) | h is the height of the octree |
| Interval tree + Hash table | O( \log(r_C) ) | O( r ) | O( v r_C ) | r is the number of runs in a region, r_C is the number of runs a single chunk. |

Under our previous estimates, we can suppose quite reasonably that an interval tree should outperform a virtualized array for typical maps.  (Of course this claim goes out the window if your level geometry is really chaotic and not amenable to run-length encoding).  To test this hypothesis, I coded up a pretty simple benchmark.  At the outset the map is initialized to a 256x256x256 array of volume, with 5 different layers of blocks.  Randomly, 2^16 blocks from 32 different types are sprinkled around the domain.  To assess performance, we measure both how long an average random read takes and how long a sequential iteration requires (for Moore radii of 0 and 1).  The results of the benchmark are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| Data structure | Avg. random read | Avg. sequential read (Moore radius = 0) | Avg. sequential read (Moore radius = 1) |
| Flat array | **0.224 μs/read** | 0.178 μs/read | 0.060 μs/read |
| Virtual array | 0.278 μs/read | 0.210 μs/read | 0.107 μs/read |
| Octree | 2.05 μs/read | 0.981 μs/read | 0.933 μs/read |
| Interval tree + hashing | 0.571 μs/read | **0.003 μs/read** | **0.006 μs/read** |

In each column, the best result is underlined in bold.  There are a few things to take home from this.  First, for random access reads and writes, nothing beats a flat array.  Second, octrees are not a very good idea for voxel worlds.  On all accounts, a virtual array is far superior.  Finally, while the access time for random reads is noticeably slower in an interval tree, the speed benefits of improved sequential iteration make them more than worthwhile in practice.  [The code for the benchmark can be found here](https://github.com/mikolalysenko/0fpsBlog/tree/master/MineBench).

Of course, interval trees will not perform well in worlds which are more-or-less random, or if there are not many runs of similar blocks that can be grouped together.  But when your world can be described in this way (which is certainly true of a Minecraft like game), switching to intervals can offer a huge improvement in performance.  I also firmly believe that there is still quite a bit of room for improvement.  If you have any suggestions or ideas, please leave a comment!