Costen, Cassette - c025180n

By Cassette Costen
October 10 in Games Technology and Programming Development Thread

Following 1

Start new thread Add to this thread

Cassette Costen

Posted October 10

Student

121

during this week and last week, i completed my project proposal and ethics forms, and started on my literature research. the outline for my project is as follows (per the proportionate review form):

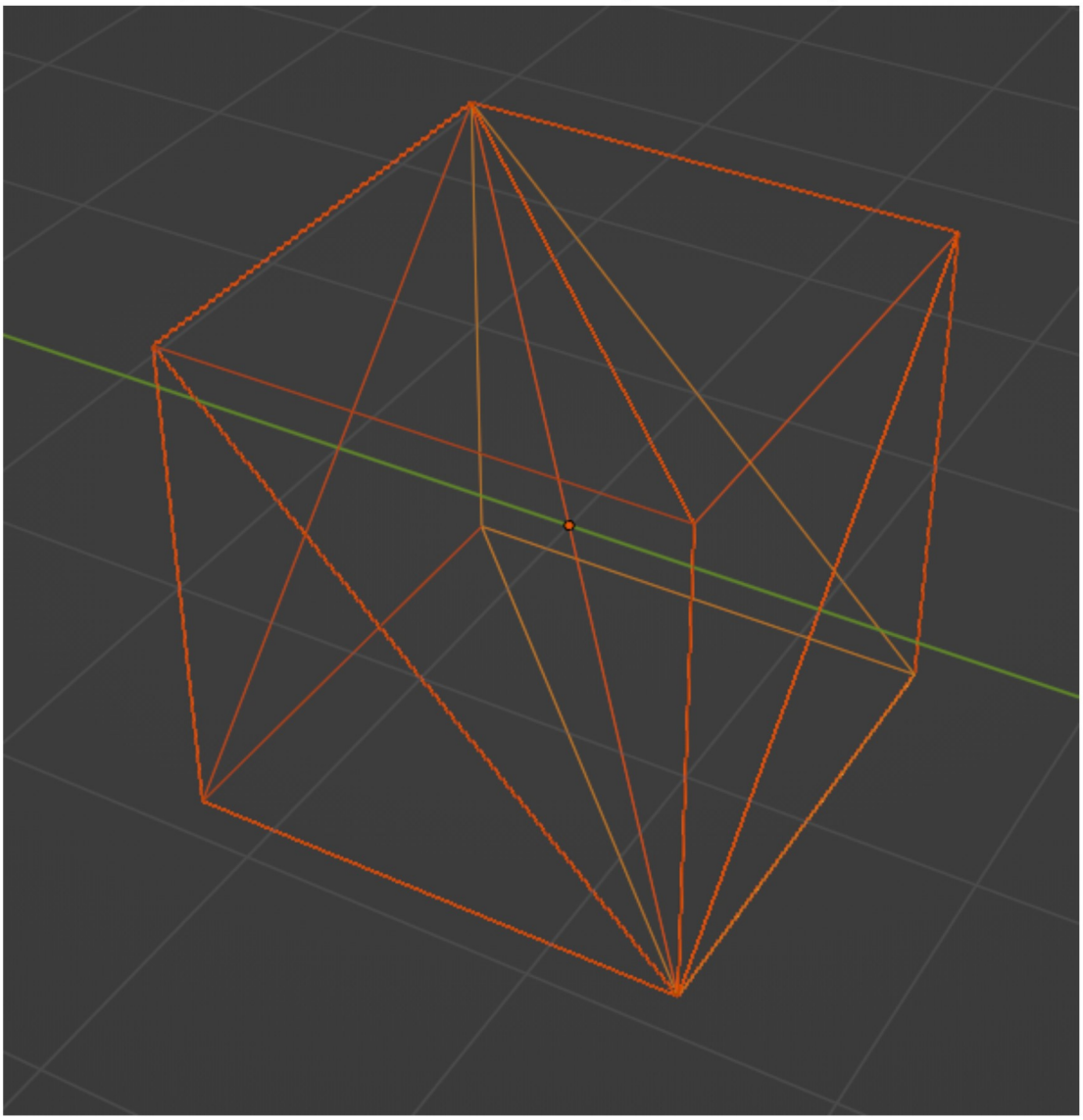
> **Quote**
>
> **The project tests several techniques/variations of the marching tetrahedra algorithm for isosurface extraction, to determine the relative visual and performance benefits of each. Literature research will be conducted to determine the background and implementation of these techniques, then several will be implemented. Two different tessellation methods and three different vertex clustering methods will be compared for perceived visual quality and computational performance; a variety of different voxel resolutions will have their performance analysed; the mean triangle aspect ratio and triangle area standard deviation will be analysed for both tesselation methods and all vertex clustering methods.**

i spent a long time understanding my key piece of literature for the project, Regularised marching tetrahedra, by Treece, Prager and Gee (1999), which aside from briefly explaining the different tetrahedral tesselation methods for cubic-lattice space, presents a very elegant solution for vertex clustering **before** generating geometry (thus avoiding a large amount of computation down the line), by taking advantage of the extra situational knowledge available during the space sampling stage of tetrahedron-marching.

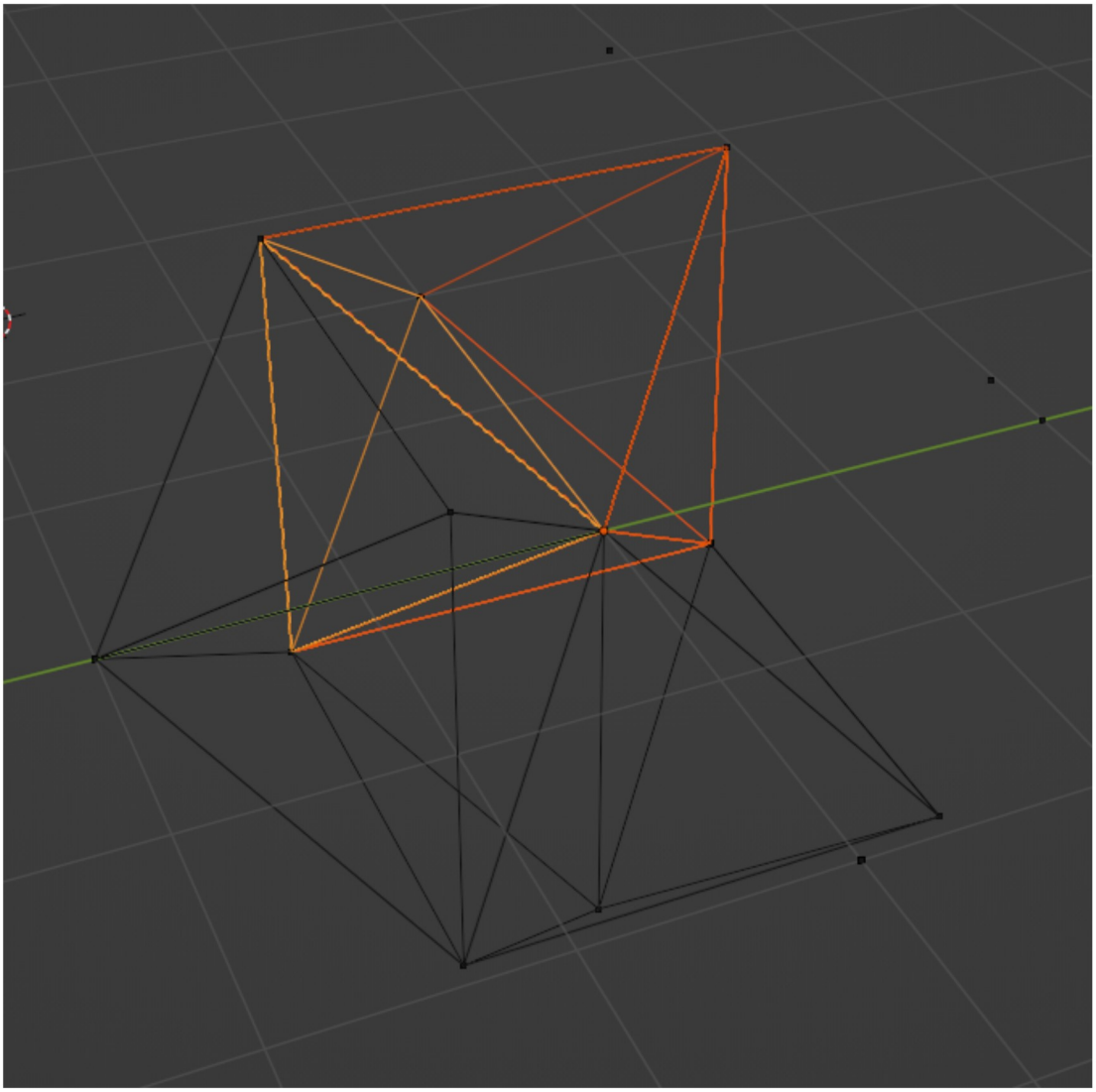
i also read A new tetrahedral tessellation scheme, by Chan and Purisima (1998), which explores the different ways of arranging tetrahedra in 3D space in more detail: the two key methods i intend to implement and test are 'simple cubic' and 'body centered' lattice

structures. the latter has better sampling efficiency, eliminates any ambiguity problems due to tesselation asymmetry, and produces tetrahedra which are closer to being mathematically regular (and where all tetrahedra are mathematically similar). however, the former is entirely contained within a cube shape (which could be advantageous for chunked sampling).

i used the diagrams in these papers to reconstruct the two tetrahedral arrangements in blender, as they're hard to visualise from static images or text descriptions.



simple cubic



and body centered models.

i also used description in the RMT paper to come up with pseudocode for the algorithm itself.

```
for each sample point:
    sample the value

for each sample point:
    for each connected edge:
        check if the value at the other end is opposite the threshold,
        and the computed intersection is closer to the current sample point:
            set a bit in the current sample point's bitfield for this edge

for each sample point:
    check if the near intersections can be merged,
    based on combinations of edge bitflags:
        compute a merged vertex position
        insert it into the list of vertices
        set the relevant edges (with near intersections)
        to reference this vertex
    else:
        compute vertex positions for each edge with a near intersection
        insert each into the list of vertices
        assign each edge a reference to the relevant vertex

for each tetrahedron:
    use the edge intersection flags (i.e. whether each edge has a vertex
    reference set) to determine which pattern to use
    use the lookup table to generate a triangulation, using the
    vertices specified by the references on the edges
    discard triangles with one or more of the same vertex reference.
```

based on my reading and experimentation so far, i believe i can make improvements both to the arrangement of the simple cubic tesselation, and to the efficiency of the generation algorithm.

firstly, the cubic tesselation as described by the 1998 paper has alternating diagonals on opposing faces of two axes of the cube, which introduces ambiguity requiring alternating lattice elements to be flipped. i rectified this in my blender experimentation by re-organising the tetrahedra to produce a tesselation with slightly less regular tetrahedra, but which has no ambiguity issues. this could potentially be implemented alongside the original simple cubic lattice to evaluate the result of this change.

secondly, the pseudocode provided by the 1999 RMT paper does not appear to take advantage of the potential for pre-allocation of data structures and pre-computation of sample point values, a change which could significantly improve performance.

+ Quote

Cassette Costen

Posted October 17

Author

Student

121

during this week i filled out my ethics forms (a proportionate review), as i need to collect user feedback to gauge the 'visual quality' of different algorithm outputs.

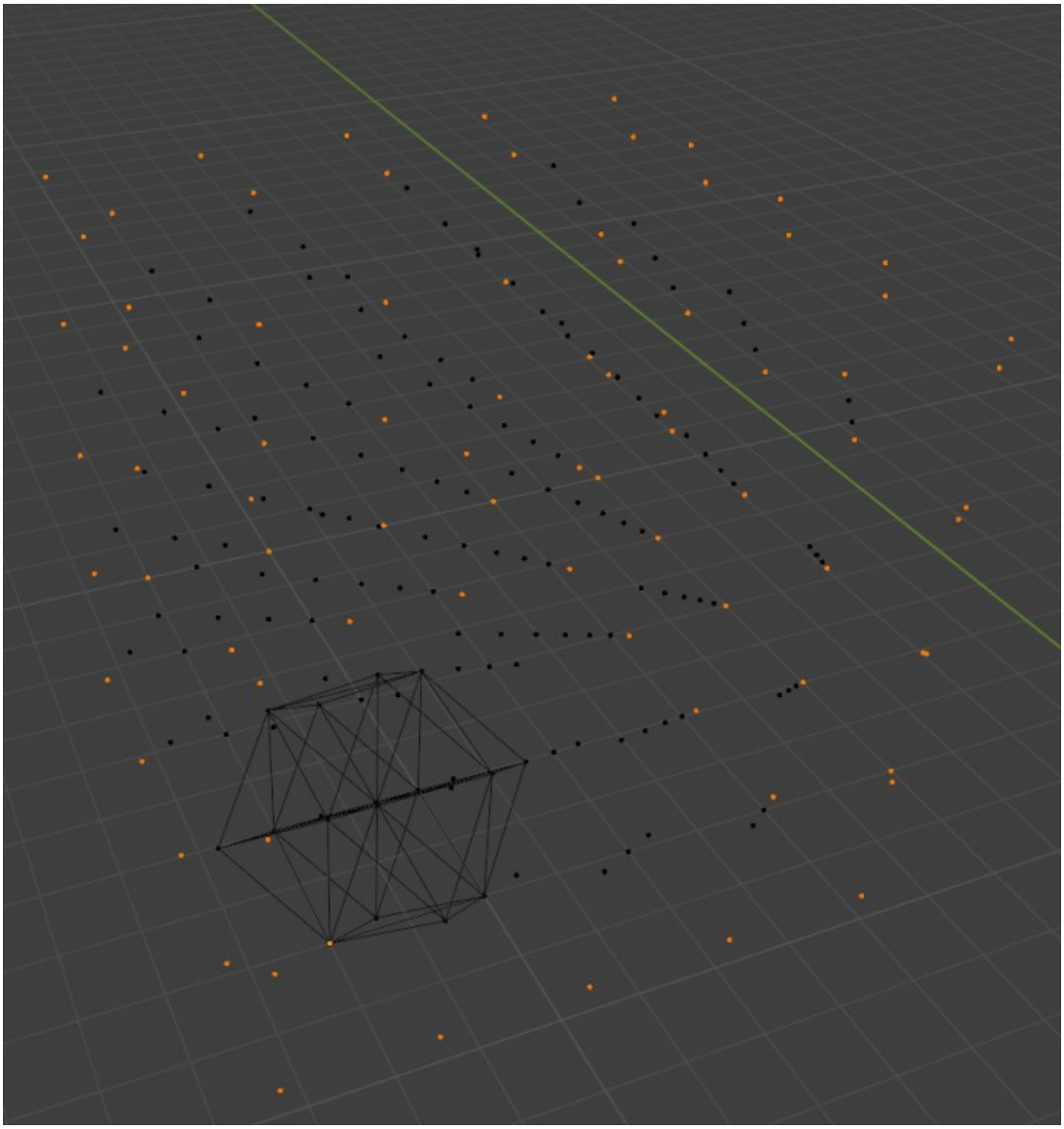
the more interesting thing i did this week was begin to implement the algorithm itself! based on the pseudocode i came up with last week, i implemented the first two passes (sampling and flagging) in C#. i even built a neat little test harness which i can customise to easily run whatever test function i want, at a configurable resolution,

multiple times, and automatically generates a nice little performance summary. see below:

```
-- summary ----------------------------
  sphere test (100 iterations)
  120x120x120 resolution
  results:
    sample points:  3616812
    edges:          0
    tetrahedra:     0
    vertices:       0
    indices:        0
  performance:
    allocation:     0.000004s
    sampling:       0.016863s
    flagging:       0.104101s
    vertex:         0.000000s
    geometry:       0.000000s
---------------------------------------
```

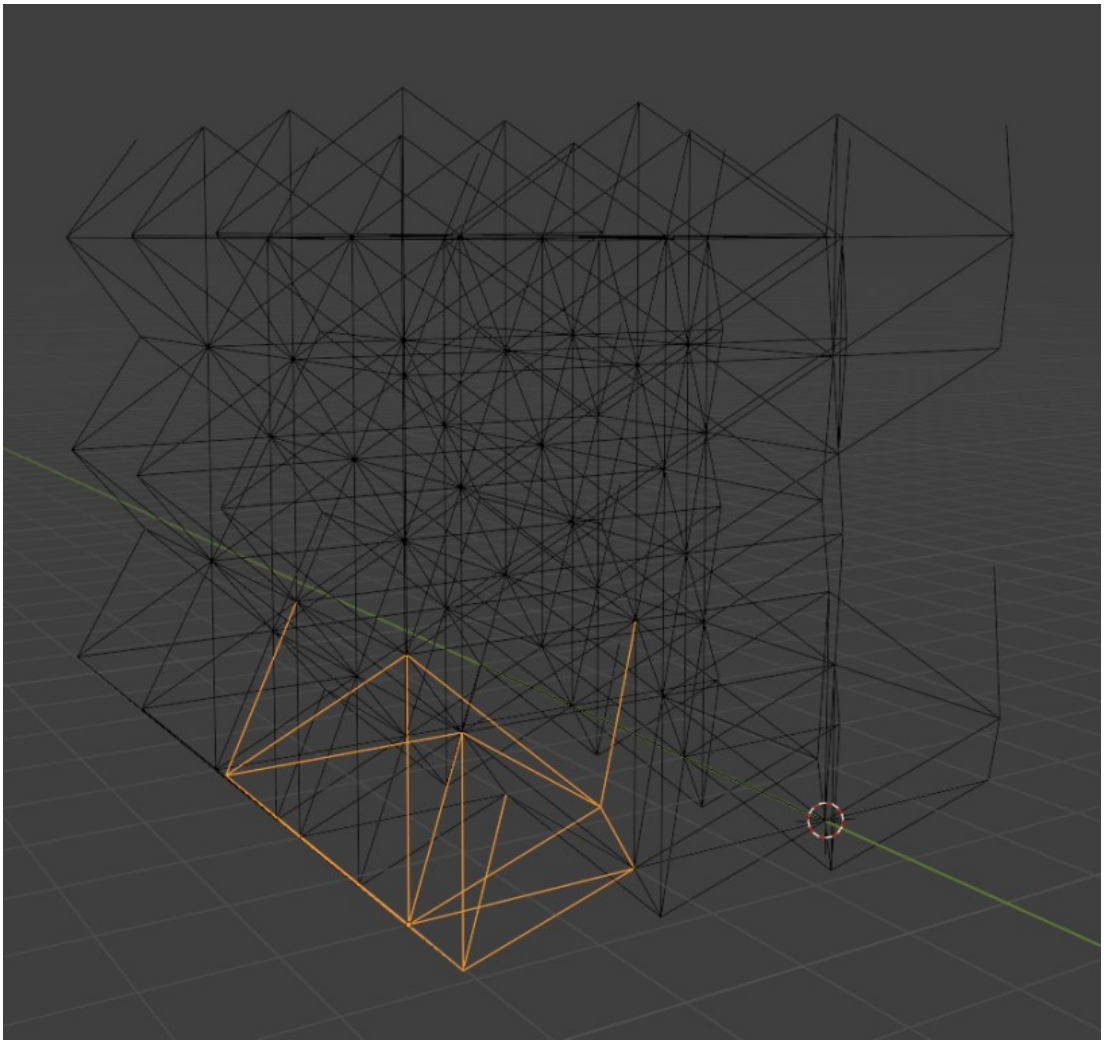
this has been super useful already for comparing individual code changes to test if they improve performance (e.g. changing how an if statement is formatted, adding extra checks, etc), and i'll definitely be using this later when i do my performance testing.

i encountered a few challenges while implementing this however. firstly i spent a while figuring out how to store the per-sample-point data, and after some experimentation including using blender to visualise the problem, i settled on a 3D array (although implemented as a simple linear array) with twice the number of Z layers, where every other Z layer's points are offset in 3D space by half a unit. this leaves some points on the table (all of the odd Z layers have (width + height) extra points which will never be used), but means that indexing into the grid doesn't require lots of extra conditional math for odd/even Z layers. below is a 3x3x3 lattice showing the sample points (as stored in the program), and the edges present in one lattice unit. points which are stored but not used are highlighted:



the second challenge that arose was whether to precompute and cache the positions of each sample point. this can be computed from the x/y/z indices into the lattice, but since these positions are used more than once, it could be worth saving them. i actually tested this and found that it was more efficient (at least, at this point in development) to recompute the positions later (this could change when more of the algorithm is implemented, so i've left it as an option). this is likely due to the position array not being CPU-cached successfully, leading to wasting time reading values back from main memory when recomputing them would be faster.

the third challenge i encountered was more difficult to overcome (although as of last night i believe i have a solution). the third pass of the algorithm requires storing data on the *edges* in the lattice, rather than the sample points. the third pass involves checking each sample point and trying to merge nearby edge intersections on its connected edges (which we flagged in the previous pass) into singular vertices. when this happens, we need to be able to tell each of these edges what vertex to use (which will then be used in the geometry generation pass after this), and hence we need to store per-edge data. however, since these edges aren't arranged in a simple regular grid which can be easily decomposed, i struggled to see how to store this data. i experimented with deleting individual edges in a lattice element until i had something which could be tiled in 3 dimensions but was unable to produce a single unit that could be repeated; i tried using a HashSet (which could be changed to a Dictionary later) but this slowed the program down by multiple orders of magnitude.

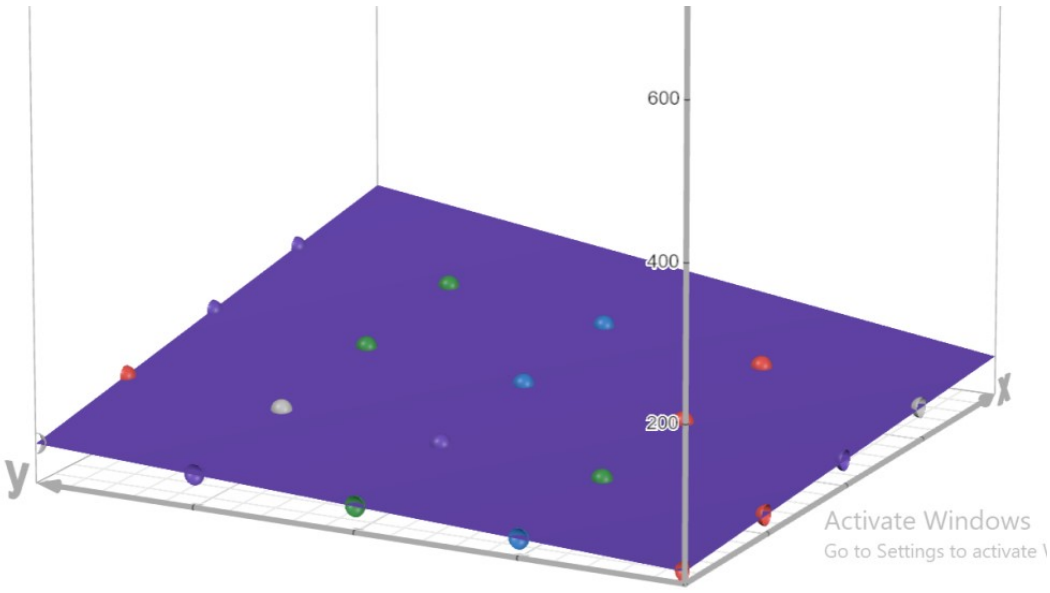


as part of this, i spent a great deal of time trying to understand how the number of sample points and edges relate to the size of the lattice (in integer units in each dimension). possibly (definitely) overkill, i made an array of different sized lattice grids in blender, merging duplicate vertices on each one to get an accurate count of the number of vertices (aka sample points) and edges. i then noted the numbers of sample points and edges in each, along with its dimensions, and then studied the change in SP count after increasing the X dimension by one unit for instance.

```
// 1x1                -  15 sp,  50 e
// 1x2                -  24 sp,  87 e
// 1x3                -  33 sp, 124 e
// 1x4                -  42 sp, 161 e
// 1x5                -  51 sp, 198 e

// sp grad =  9 (dSP/dY)
// e grad  = 37

// 2x1                -  24 sp,  87 e
// 2x2                -  38 sp, 149 e
// 2x3                -  52 sp, 211 e
// 2x4                -  66 sp, 273 e
// 2x5                -  80 sp, 335 e

// sp grad =  14 (dSP/dY)
// e grad  = 62
```

```
// sp grad =  24 (dSP/dY)
// e grad  = 112

// sp grad =  9 (dSP/dX)
// sp grad = 14 (dSP/dX)
// sp grad = 19 (dSP/dX)
// sp grad = 24 (dSP/dX)
// sp grad = 29 (dSP/dX)

// dSP/dX = 9y + 6
// dSP/dY = 9x + 6

// d(dSP/dX)/dY = 5x + 4

// 1d formula = 9x + 6

// 2d formula = 5xy + 4x + 4y + 2

// 1x4x2            -  66 sp, 273 e
// 2x4x2            - 101 sp, 452 e
// 3x4x2            - 136 sp, 631 e
// 4x4x2            - 171 sp, 810 e
// 5x4x2            - 206 sp, 989 e

// sp grad =  35 (dSP/dX)

// sp grad =  24 (dSP/dZ)
// sp grad =  35 (dSP/dZ)
// sp grad =  46 (dSP/dZ)
```

i then used the excellent tool Desmos to plot the points where Z = 1 (all points which don't have a Z dimension are Z = 1), and after some experimentation i was able to come up with the formula relating X and Y size, and number of sample points (seen in screenshots above in the 2d formula line), producing the graph below.



after some more experimentation with trying to fit in the Z contribution, i realised that increasing Z size should contribute in exactly the same way as increasing X or Y (since a 1x2x1 lattice should have the same number of SP as 2x1x1 and 1x1x2), and indeed the 2D equation above is symmetric (the coefficients of 4x and 4y are the same). as a result, i developed this general formula for all surfaces like this:

$$h(x,y,z) = ixyz - j(xy + xz + yz) + k(x + y + z) - l$$

where i, j, k, and l are all constant coefficients. i was then able to decompose my 5xy + 4x + 4y + 2 equation to give some constraints/relations between the coefficients, to the point where only i could be changed (and all other coefficients would change automatically). i then changed i until the equation gave the correct value for all of my data points with Z != 1.

$$g(x,y,z) = axyz - b(xy + xz + yz) + c(x + y + z) -$$

$$a = 2$$

-5 13

$$b = a - 5$$

$$= -3$$

$$c = b + 4$$

$$= 1$$

$$d = (a - 3b + 3c) - 15$$

$$= -1$$

$$g(x,y,1)$$

$$g(1,4,2) - 66$$

$$= 0$$

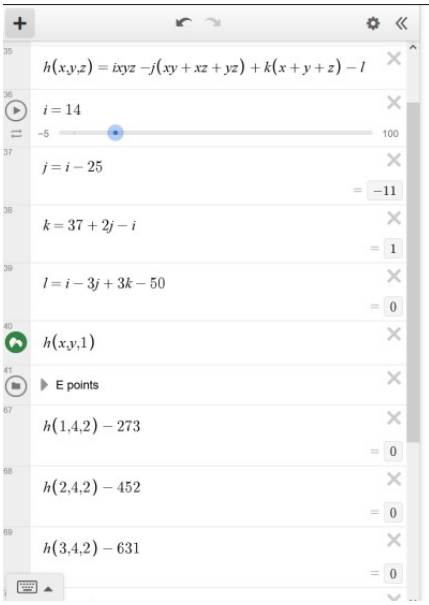
$$g(2,4,2) - 101$$

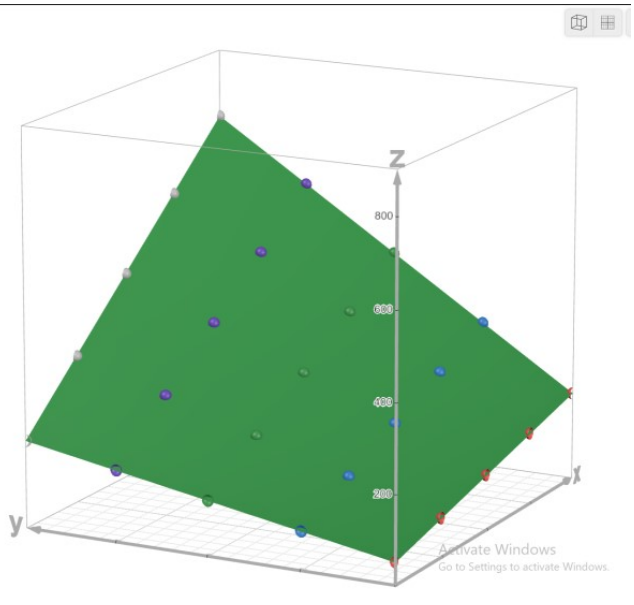
$$= 0$$

$$g(3,4,2) - 136$$

$$= 0$$

i was kinda surprised this worked as a technique but it did. i then did the same process for the edge counts, although instead of trial-and-erroring until i found a 2D equation, i just solved simultaneous equations for a few different observed data points until i had expressions to relations between the coefficients again, leaving me with only one coefficient to modify until all my Z != 1 test values came into line.





all of this gave us these two equations relating lattice size to number of edges/sample points:

```
// FINAL 3D equation (SP):
// sp(x,y,z) = 2xyz + 3(xy + xz + yz) + (x + y + z) + 1

// FINAL 3D equation (E):
// e(x,y,z) = 14xyz + 11(xy + xz + yz) + (x + y + z) + 0
```

overall, while this wasn't super useful to the implementation, it did help my understanding of the lattice. these numbers make sense after all, there are 14 edges which connect directly to each central sample point (one of which is shared with its neighbour, but there is always an extra one at the end of a row), and 36 edges (36 = (3 * 11) + 3) other edges per lattice unit, which this equation reflects pretty well. regardless, these equations should be useful tools for determining how much memory is being left on the table when storing edges or sample points (i.e. these equations reflect the number of each in the actual lattice that data structures represent).

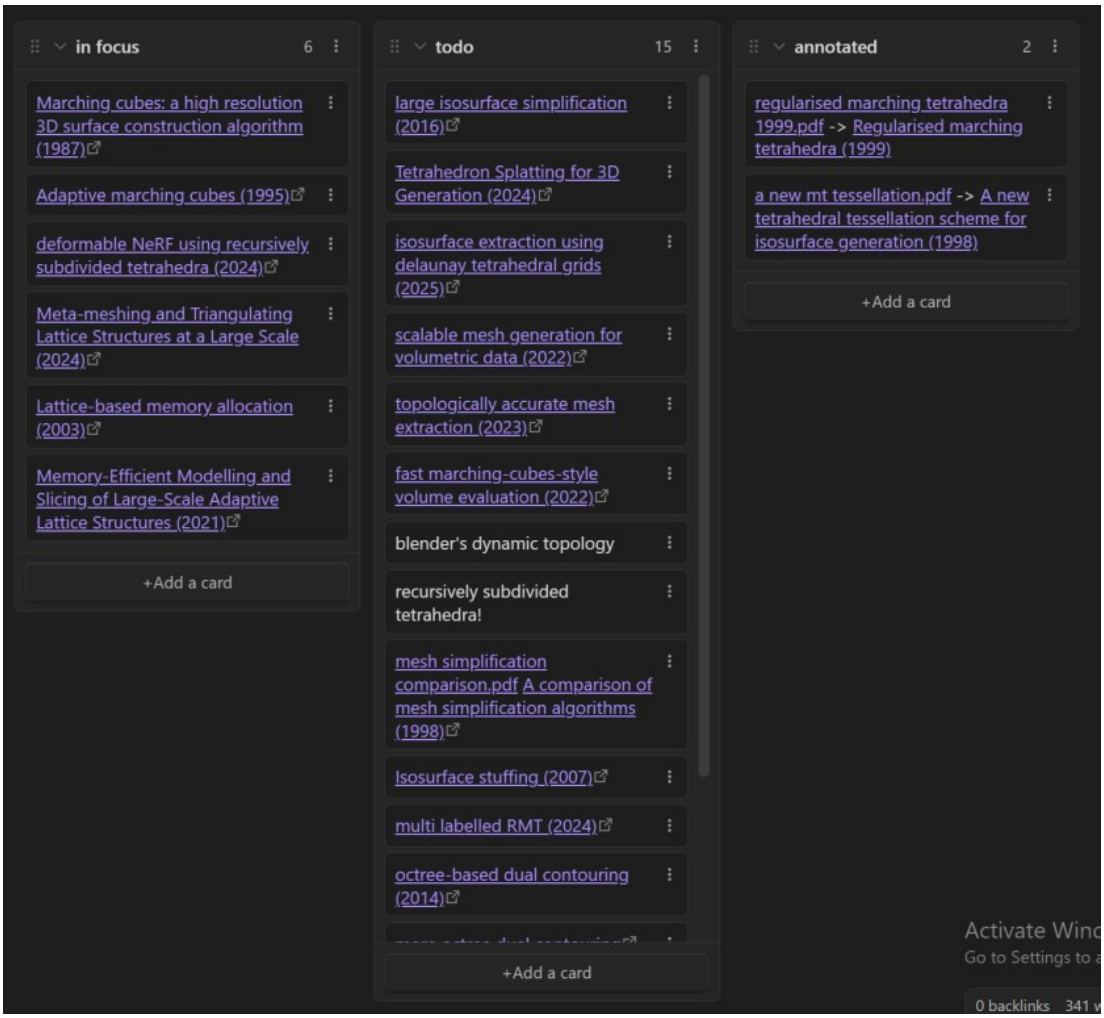
however, during this experimentation i came to the realisation that although each edge is shared between two sample points, only at most one of the sample points will ever store data on it at a time. this means that edge data can actually be stored per-sample-point (and which edge the data is referring to can be referenced with an index), since the final geometry pass will consider all the sample points anyway (i.e., if it looks at the edge between two sample points, either one of them could have the 'nearby intersection' flag set, or neither, but not both; if the flag is set then the program will simply take the edge data value from whichever sample point has the edge flag set.

although this solution is somewhat memory inefficient, it should be very fast (hopefully). i'll get to implementing it next week.

the fourth and final issue that i encountered this week was performance: even in release mode with (i think) fairly efficient coding, the sampling and flagging passes combined (for a 120x120x120 grid with a very simple sampling function, just a sphere) take about 0.11s. as far as realtime performance goes this is **checks notes** bad. i've made some improvements by marking various data arrays as 'unsafe' to skip bounds checking, and marking a temporary int array as 'stackalloc' to skip a slow 'new' call, and this resulted in

a 0.02s speedup. on Davin's advice i'm going to continue working in C# for now as a test implementation while i continue my research, but i may jump into C++ in a few weeks time to reimplement it in a hopefully faster language.

in addition to all of this stuff, i've done some more literature research (though i havent gone through any papers in detail this week, this will pick up over the weekend and next week) and acquired a bunch of new interesting papers. i also organised my research a bit into its own kanban board so i can actually see what i have left to do.



for whatever reason i couldn't access any Springer articles yesterday afternoon which was weird, hopefully that's not permanent. i also made a note to study blender's dynamic topology tool, as it seems like it could be really useful for understanding realtime deformation alternate approaches. octree-based dual contouring and recursively subdivided tetrahedra sound really cool as well, this research is making me want to implement all of them.

+ Quote

Libby Sedgwick 1

Cassette Costen

Posted Wednesday at 10:37 Author

wow okay so i did a lot of stuff last week and so far this week.

first of all, i completely pivoted to C++. while C# has a lot of benefits on the easy-to-integrate side, my C++ re-implementation of the first two passes (sampling and flagging)

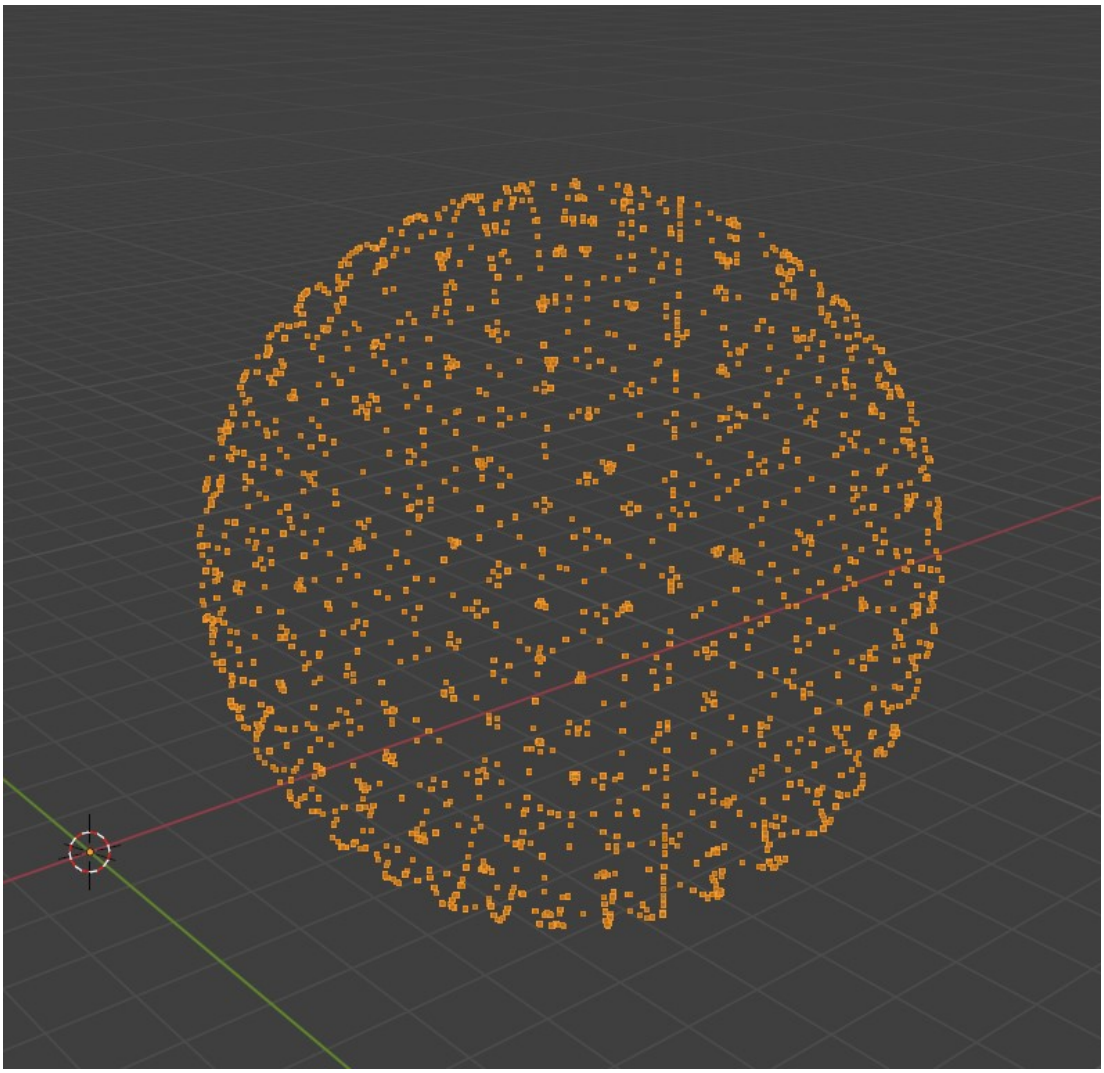
Student

121

by just lightly transcribing C# code and implementing a few (very minor) C++ only optimisations, i was achieving literally double the performance out of the gate. so i've continued with C++.

my second step was to implement the next pass, the vertex generation pass. as a reminder, this pass runs over each sample point again, looks at the edge-proximity flags generated by the flagging pass, and creates vertices on the relevant connected edges (relative to each sample point), storing the index of the resulting vertex on that edge for the geometry pass to use later. however, it quickly became obvious that a lot of the traversal and memory access work performed in the flagging pass was simply being repeated in the vertex pass, so i made the decision to merge the two, which took us **from 0.025s for flagging and 0.044s for vertex generation to a combined 0.052s for both the combined flagging/vertex pass (a 25% improvement)**.

this allowed me to write a super simple OBJ file generator for the vertices generated by the algorithm and... voila:

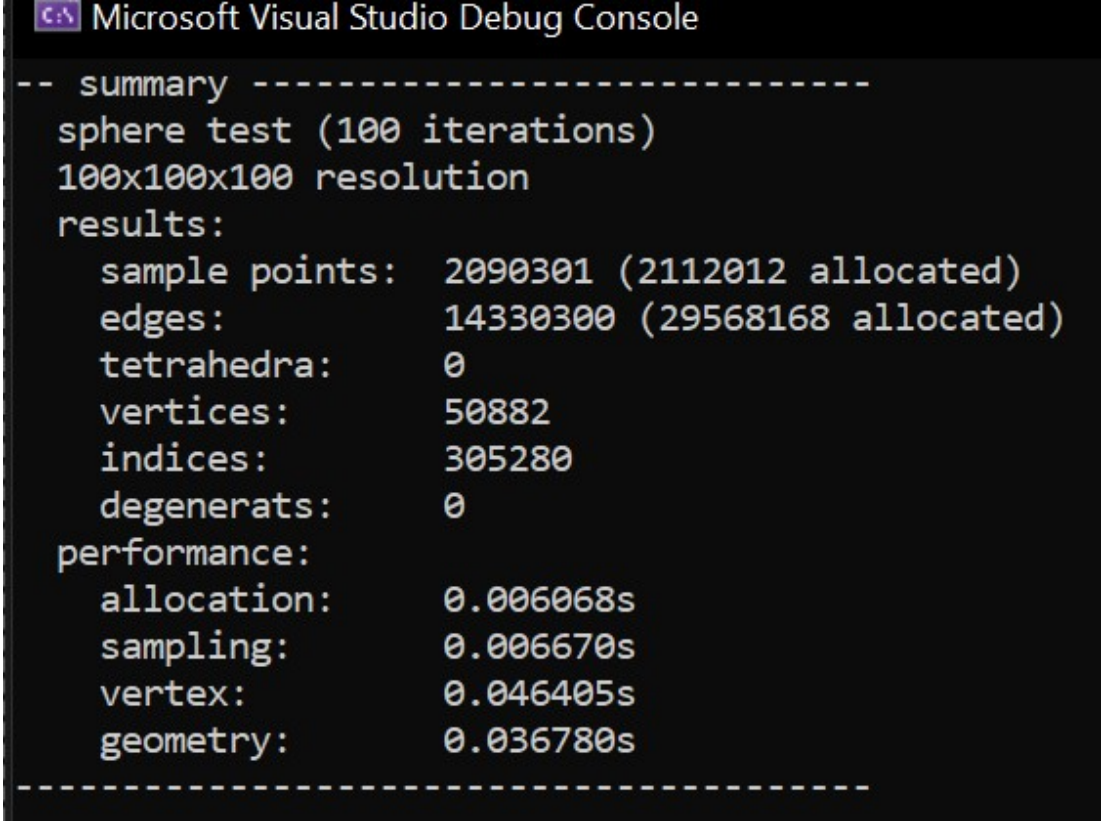


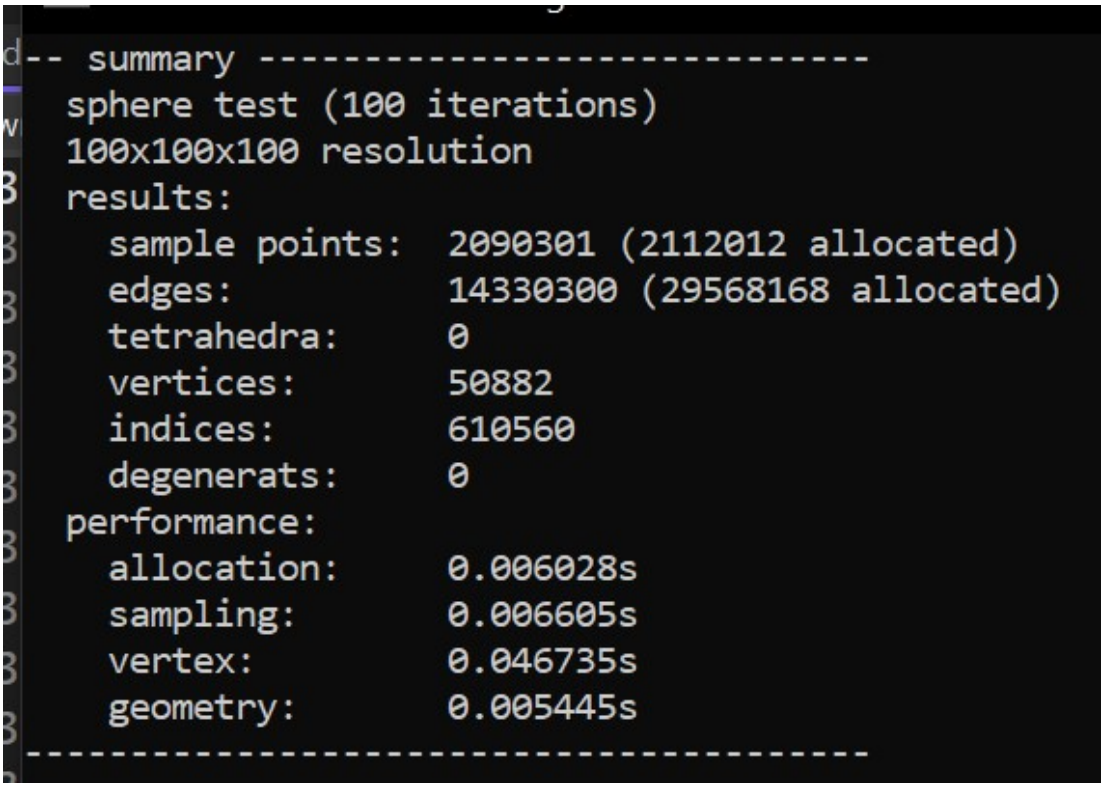
it was quite satisfying actually getting to see something at last.

in the next couple of days i grinded through implementing the final pass, the geometry pass. in this pass, instead of iterating over sample points, we iterate over *lattice cubes*. for each one, we gather the relevant information about which sample points are used (we can do this by calculating the SP at the center of the cube using a bit of math, and then using the index offset table to find the other 14 neighbouring SPs).
then, we iterate through the 24 tetrahedra per LC - 4 around each cardinal direction (so 4 on the +X axis from the center SP, 4 on the -X, +Y, -Y, etc). for each one, we use some **painstakingly-written-out** tables, to work out *for tetrahedron X, which 4 of the 15*

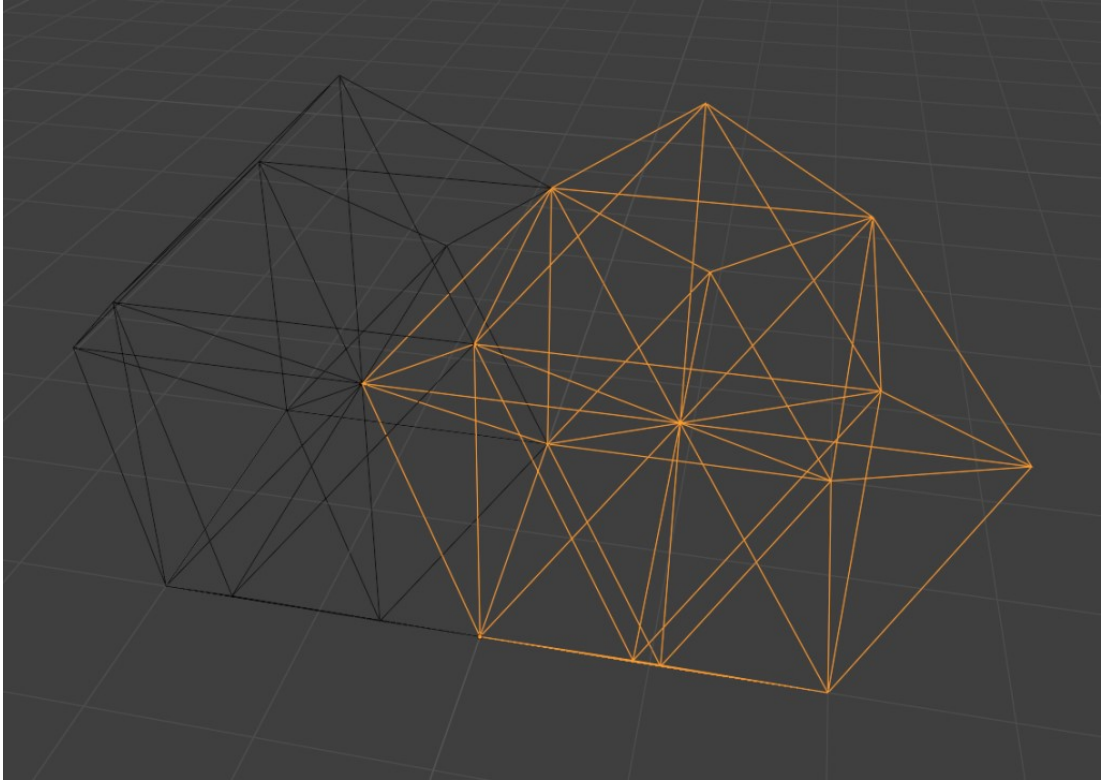
sample points (1 center and 14 neighbours) are used, and in what order?, and then we use that to check the sample values for each of those points against the threshold (i.e. to find out which of the 4 corners of the tetrahedron are inside/outside the isosurface). this then gives us a 4-byte pattern, which indexes into another table which tells us *how many triangles should be present in this tetrahedron? and which edges should we look at for each triangle corner?*. we then look at one more table which tells us *for each of the 6 edges in the tetrahedron, which of our 4 sample points in the tetrahedron, and which edge address (0-13) stores the data for that edge?*. there are actually 12 edges, where every other edge is the inverted form (i.e. center -> pyramid point @ +X edge, followed by pyramid -> center point @ -X edge are technically the same edge, but two possible places for the vertex reference to be stored, depending which sample point the isosurface intersection was closest to). finally, we can use all of this information to pull out 3 or 6 vertex references which represent the corners of 1 or 2 triangles, and we add them to the triangle array.

the next step was optimising all this. and also fixing two big problems. firstly, i realised that the sample proximity array, which stores flags for each sample point stating *which connected edges have nearby intersections with the isosurface?* (i.e., which connected edges should we have created vertices on?). this was never actually being used in the vertex pass, so i just removed it, slightly speeding up the vertex pass by reducing memory writes. this lead me to a further optimisation (via some profiling), that the most expensive part of the geometry pass was the part where we test each sample point involved in a given tetrahedron against the threshold. this is not only 4 memory fetches (and probably cache misses too) per tetrahedron, per lattice cube, but is something we already did earlier! as such, i added a new data array (per sample-point) which is filled out in the vertex pass which stores *for a given sample point, which connected edges intersect the isosurface?*; now i know this sounds the same as the sample proximity array but this one includes *all* intersecting edges, not just the ones which are nearby. this can be used to massively speed up the geometry pass, in three ways: 1. there is only one fetch per lattice cube (and it's only 2 bytes instead of 4 * 4); 2. the math to pull out booleans representing which sample points are less than the threshold is much simpler (a bitwise AND instead of greater than followed by AND); 3. we can check the edge-crossing flags value for the center point right at the start of the lattice cube, and if it's zero (i.e. none of the edges radiating from the center point intersect the isosurface, and hence none of the tetrahedra can possibly have any geometry in them) we simply skip onto the next lattice cube, saving a ton of traversal, math, and checking. this (as shown below) netted a **massive speedup of around 85%!!**

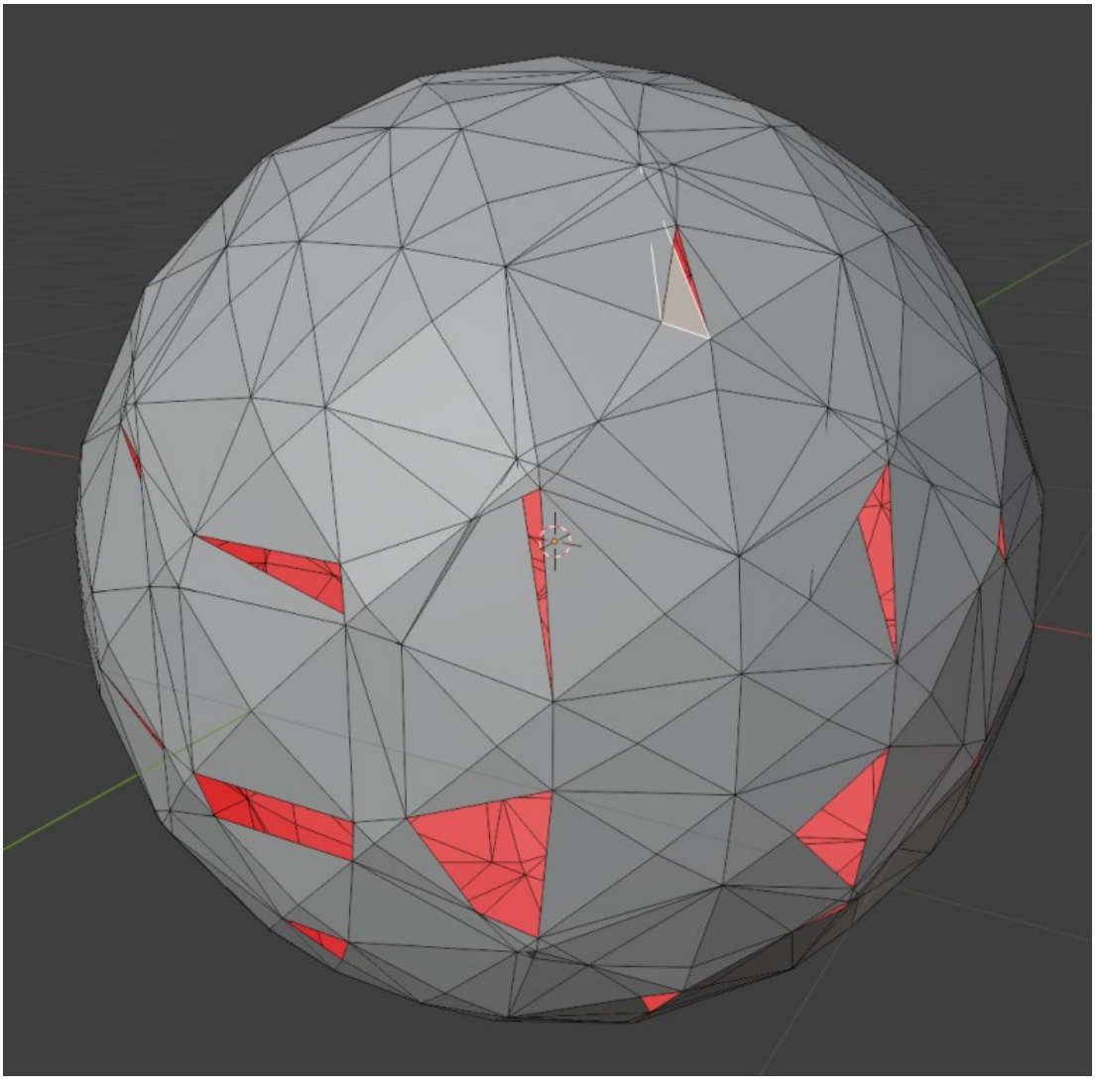




however, after making this optimisation, i had those two problems to deal with. the first of these was that almost all of the tetrahedra were being computed twice over.



if you look at the image above, hopefully you can see that the 4 tetrahedra around the +X axis from the leftmost lattice cube are **identical** to the 4 tetrahedra around the -X axis from the rightmost lattice cube (and likewise for the Y and Z axes for a 3D lattice grid). in order to solve this problem, i added a little bit of logic to skip over the -X tetrahedra if the current lattice cube is anything but the first LC on the X axis (and likewise for the other axes). this almost halved the workload! however, fixing this problem revealed the second.

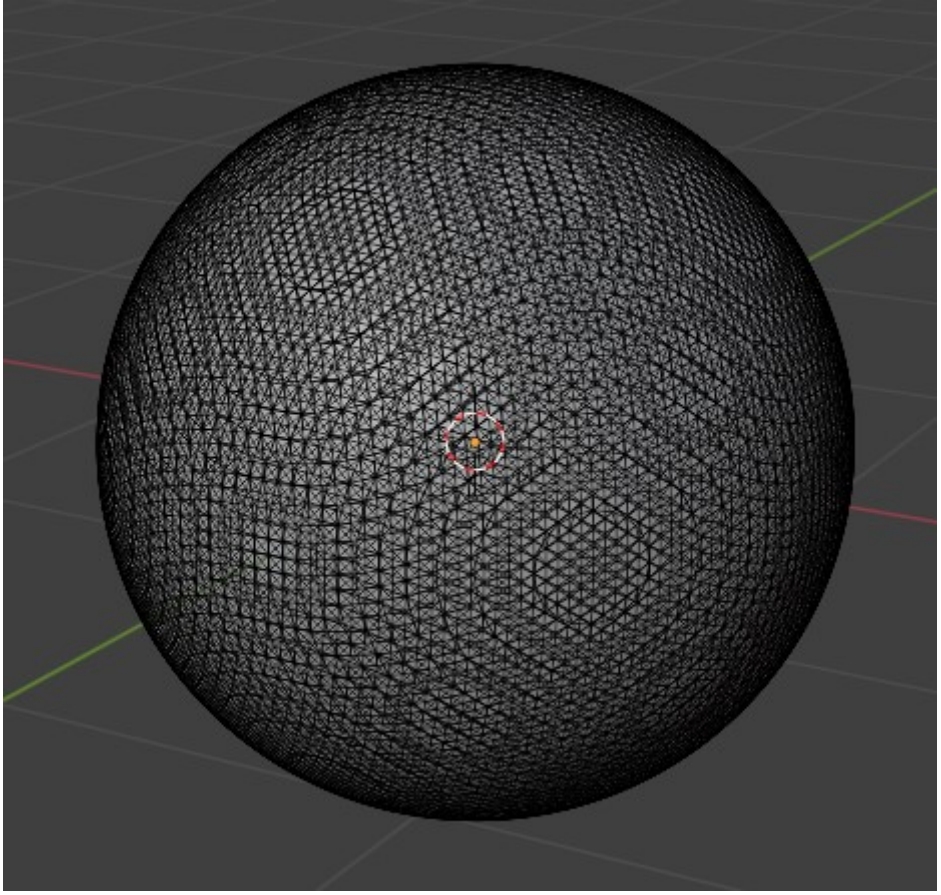


notice anything wrong with this sphere? yeah, thats right. it's fucked. the two visual issues are **missing triangles**, and **triangles connected to the wrong vertices**. in addition (or possibly, indicatively) the OBJ file had triangle indices which were clearly '-1' (expressed as 65536, since i'm using uint16_t and OBJs index from 1). after beating my head against this for a while, wondering why it was only apparent when the code to skip duplicate tetrahedra was enabled, and debugging when those pesky '-1' indices were appearing (which should never happen by the way, if this happens it means that the geometry pass is looking for a vertex on a particular edge, but the vertex pass didn't create one, which shouldn't be possible), i finally discovered the issue. one of my **painstakingly-written-out** tables? one single value was wrong. this meant that for one of the tetrahedra, one of the edge addresses was wrong, (imagine instead of a nice closed tetrahedra with 6 edges, one of the edges is just pointing off into space instead of closing the shape), leading to the wrong vertex reference data being read when creating the triangles (either reading the wrong vertex index for a different isosurface intersection, or reading '-1' if the incorrectly-referenced edge didn't have an intersection), resulting in the weird mix of missing, invalid, or incorrect triangles.

```
455 455     // hence, entries represent the 6 edges in the tetrahedron and are ordered
456 456     //     CP, CU, CL, PU, PL, UL
457 457     static constexpr uint8_t tetrahedra_edge_address_templates[24][6] =
458 458     {
459 459         // +x side
460 460         { PX, PXNYPZ, PXNYNZ, NXNYPZ, NXNYNZ, NZ },
461     -       { PX, PXNYNZ, PXPYNZ, NXNYNZ, PXNYNZ, PY },
    461 +       { PX, PXNYNZ, PXPYNZ, NXNYNZ, NXPYNZ, PY },
462 462         { PX, PXPYNZ, PXPYPZ, NXPYNZ, NXPYPZ, PZ },
463 463         { PX, PXPYPZ, PXNYPZ, NXPYPZ, NXNYPZ, NY },
464 464         // -x side
```

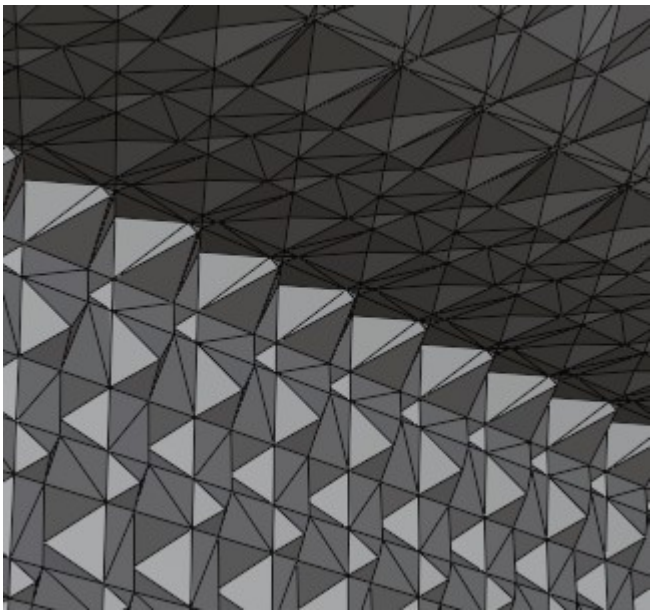
the reason this became apparent only when the skip-duplicate-tetrahedra code was enabled was because the inverted version of that tetrahedron (i.e. when it computed a second time by the next lattice cube) did not have an error, and thus the holes were being covered up by correctly-generated triangles the second time around. here's the result of all that:

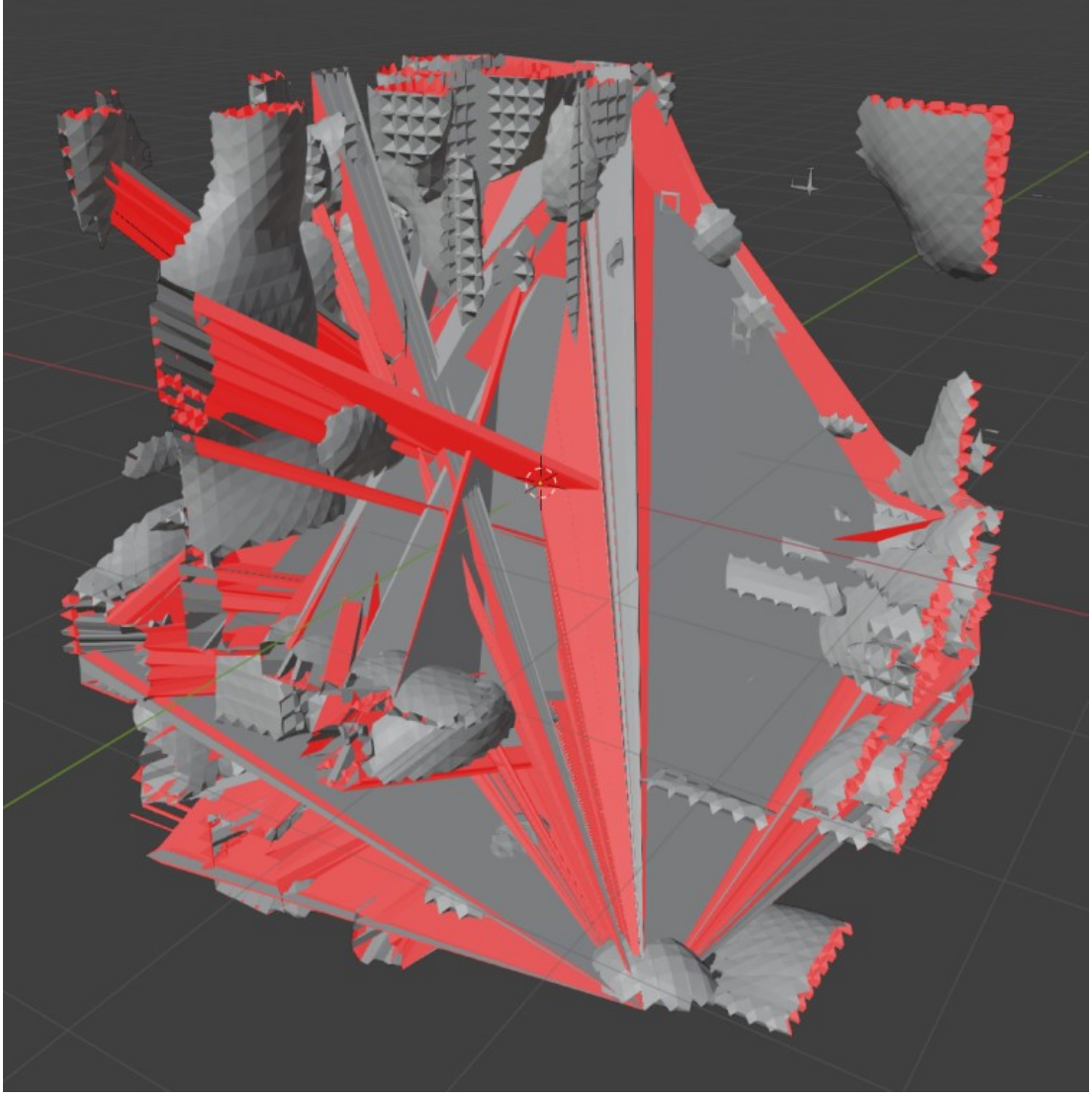
```
-- summary ----------------------------
  sphere test (100 iterations)
  100x100x100 resolution
  results:
    sample points:  2090301 (2112012 allocated)
    edges:          14330300 (29568168 allocated)
    tetrahedra:     163968
    vertices:       50882
    indices:        305280
    degenerates:    0
  performance:      0.055293s
    allocation:     0.005826s
    sampling:       0.006017s
    vertex:         0.039412s
    geometry:       0.004038s
-------------------------------------
```



beautiful really, isn't it. this made me very very happy.

after this, i wanted to better demonstrate the now-mostly-working algorithm. i ported a previously-written fractal brownian motion noise algorithm into the project and used it as the sampling function for another benchmark. this resulted in some nicely curved areas mixed with a lot of jagged flat areas. these baffled me and it took me a while to solve this (i'll explain a bit further down), but this test also highlighted an issue where edges on the outer sides of the sample volume were skipped in the vertex pass, resulting in '-1' vertex references. i fixed this by adjusting my early rejection logic in the vertex pass and this resolved the issue. the images below are the weird jaggedy-ness, and the broken edge skipping.





obviously not working as intended.

however, next it was back to optimisation. i was able to eliminate the array of sample positions, as the only time they were used outside the sampling pass was during the vertex pass. my vertex generation math initially was fetching the neighbouring sample positions from memory (stored in the sample position array), but in the shower on Sunday morning (yeah i know, literal shower thought) i realised that all the math needed was the *relative vector* between the current sample point and the neighbour. and since we're in a regular lattice, this is always the same, no matter which point you're currently at. so i turned this into a table, storing the vector offset for each possible edge address (0-13). this eliminated a full 15 random memory fetches from the vertex pass (i also added logic to track the current sample position in the vertex pass), and allowed me to completely eliminate the sample position array, saving a bunch of memory too.

at this point i also added some extra information to the benchmarking summary (in fact, i was incrementally adding stuff the whole time), including how many tetrahedra were evaluated out of the total number in the lattice, the percentage of total time spent in each pass, and the memory efficiency of the program relative to the theoretical minimum.

now we need to return to the FBM benchmark issue. i recreated my FBM math precisely in blender's geometry nodes (god bless blender as usual), and it worked perfectly... and looked totally different to my result. i realised the sudden discontinuities with the geometry were actually *due to the way i was performing rounding* in my vector math library.

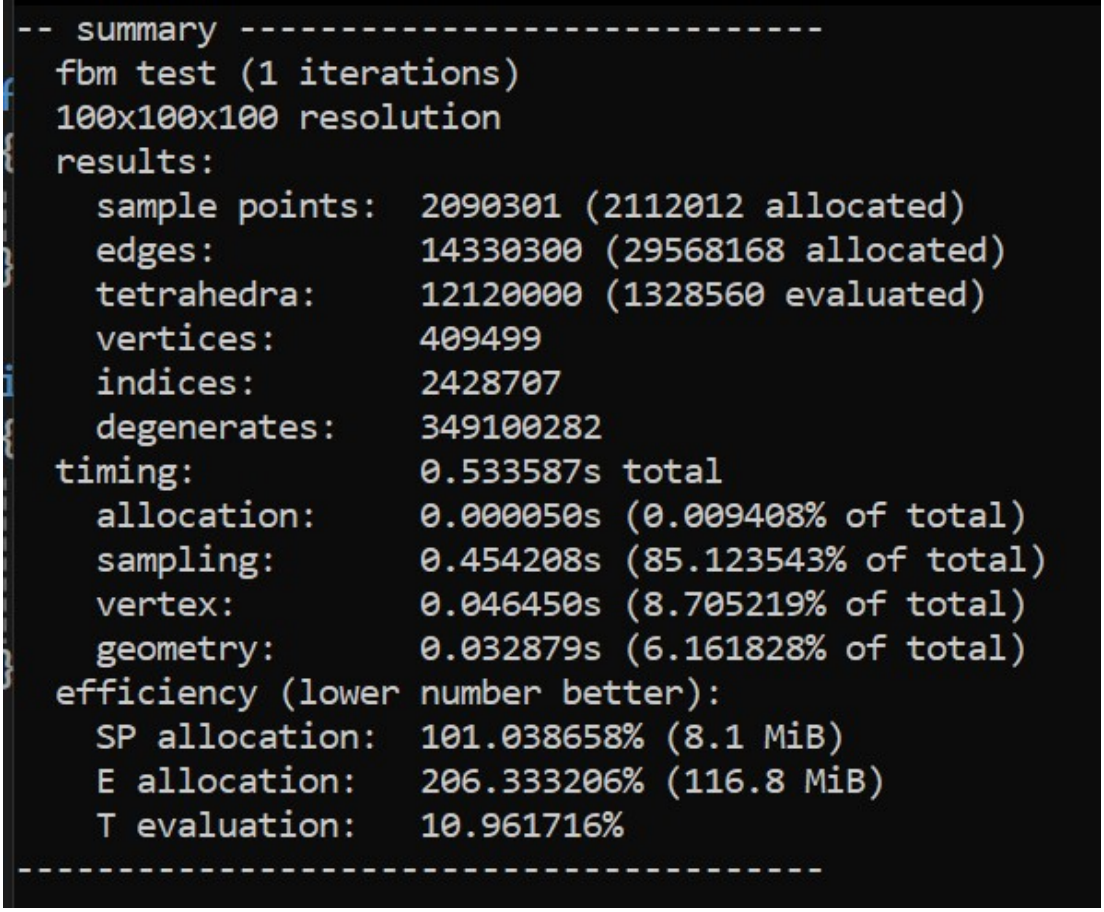
```
47      - inline Vector3 fract(const Vector3& a) { return Vector3{ a.x - (long)a.x, a.y - (long)a.y, a.z - (long)a.z }; }
47      + inline Vector3 fract(const Vector3& a) { return Vector3{ a.x - floor(a.x), a.y - floor(a.y), a.z - floor(a.z) }; }
```

my `fract` function was rounding towards zero. NOT towards negative infinity, as the algorithm was expecting. this resulted in huge discontinuities across the X/Y/Z = 0 boundary, and because it was a discontinuity (and not just a transition from big value to small value), the linear interpolation performed when generating vertices was not smart enough to reconstruct the instantaneous jump in the surface correctly. so i wasted a lot of time trying to understand what was wrong with my algorithm when actually it was a bit of math used in the benchmark itself. oh well.

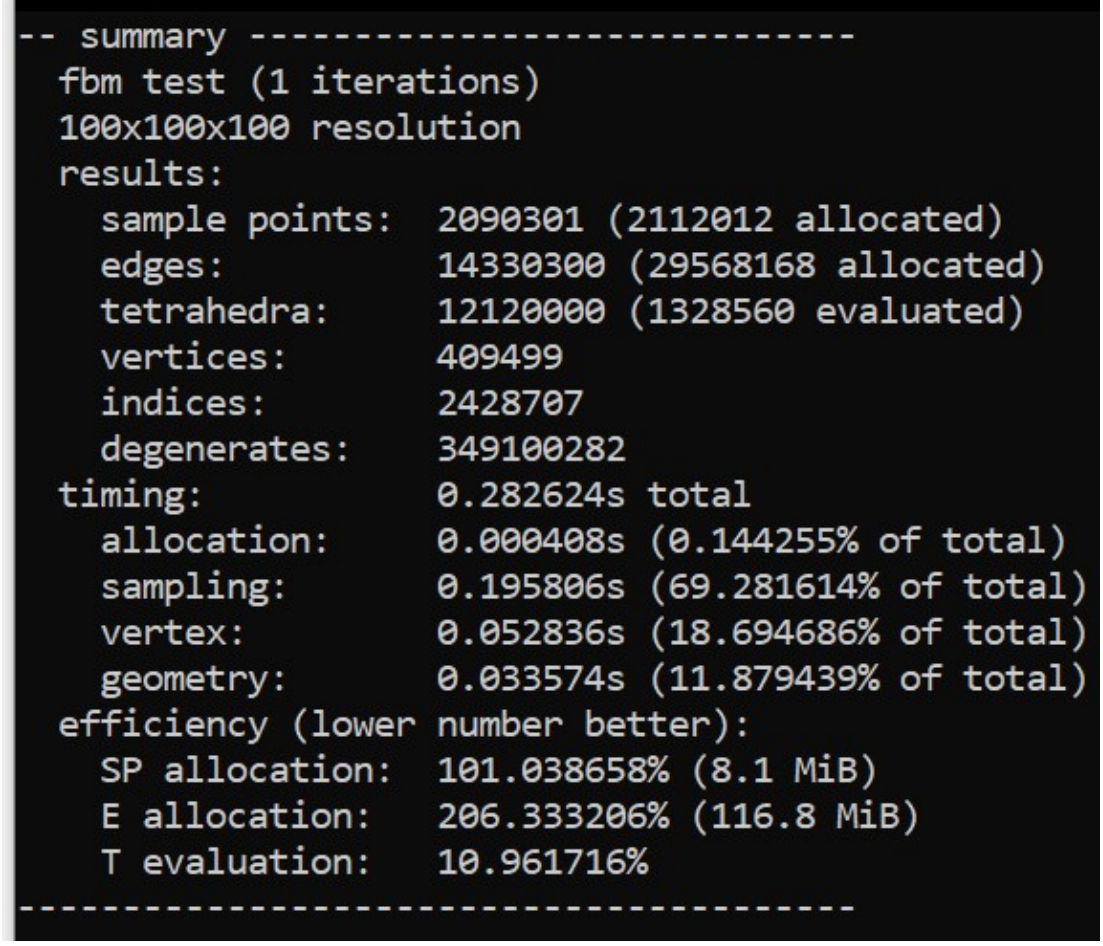
after that i treated myself to some cleanup work:
- moved all my algorithm code into an `MTVT` namespace (i named the VS solution this initially but i don't actually know what i meant it to stand for. Marching Tetrahedra.... Visualisation.... Tool?)
- turned a bunch of types into typedefs, to make code clearer and more maintainable (i replaced instances of uint16_t when used as a vertex index with VertexRef; when uint16_t is used as a 14-bit collection of edge flags it becomes the EdgeFlags typedef; an index into the sample point arrays becomes an Index (previously size_t); and an 8-bit value ranging from 0-13 used to express which index edge to use (0-13 out of a 14 bit flags value, etc) became an EdgeAddr)
- added #defines for each of these (e.g. VERTEX_NULL is defined as (VertexRef)-1, to make code clearer)
- tested one or two minor possible performance improvements i had TODO-ed in my code to look at later, though neither turned out to bring a benefit
- added a bunch of checks to the `configure` and `generate` functions to throw errors if the user tries to initialise a builder with parameters which would exceed the limits of the relevant data types (e.g. if size_t can't index the entire sample value array due to the dimensions being too large)

as part of this i also fixed an issue where garbage meshes were generated when the resolution was too high. this was due to more than 65535 vertices being generated (the limit of uint16_t), and so i switched to using uint32_t for vertex references (this can easily be changed now that i'm using typdefs, and there is also a handler in the `generate` function for if the number of vertices exceeds the maximum value of whatever datatype is being used for VertexRef).

i promise we're near the end now! my last actually important change so far was to parallelise the sampling pass. in the sphere benchmarks you've seen so far, the majority of the time is spent in the vertex pass, since this is kinda where the magic (math and memory fetching) happens. however, it turns out that when you use a sample function which is more complex than like... vector length, suddenly most of your time is spent in the sampling pass!

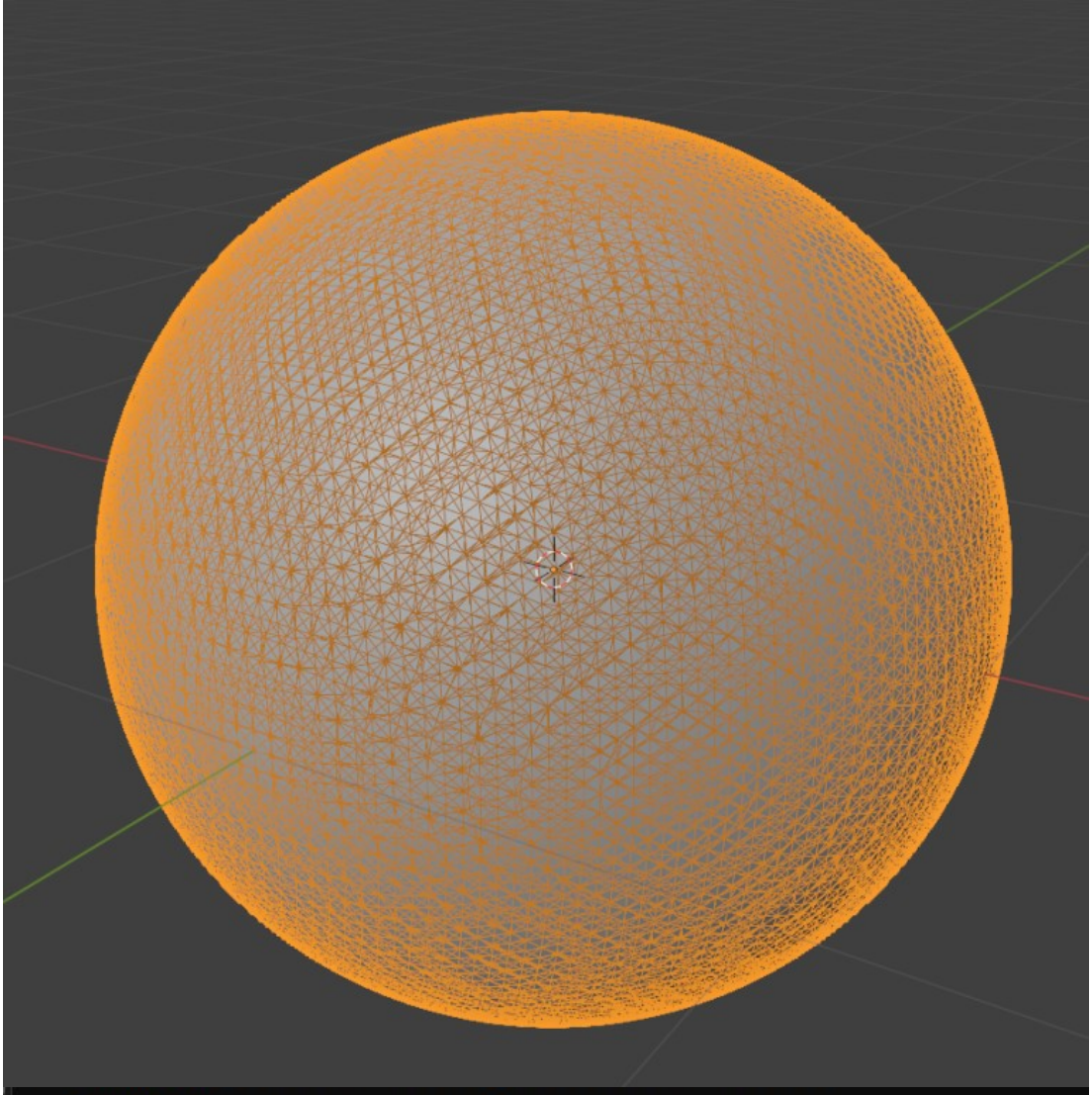


even thought we're marching a 100x100x100 volume, literally 85% of the time is just sampling. in response to this, i decided to experiment with parallelising this pass. initially i split this up so that each thread would compute alternate Z-layers (so thread 0 would compute layer 0, 2, 4, etc, while thread 1 would compute 1, 3, 5, etc). however, this only pushed the sampling pass down to **75%** of the total time. not exactly the double-performance expected. talking to my supervisor Kieran, we theorised this could be due to the alternating-layers strategy resulting in a fuck-ton of cache misses due to skipping forward in the array. and, whether causation or just correlation, changing the approach to process the Z-layers in blocks (so thread 0 computes layers 0 up to n/2, while thread 1 computes (n/2)+1 up to n-1) significantly improved the results, pulling sampling down to around **65%** of total time. i made the number of threads controllable by the user. this benchmark is a little unfair since my laptop is running out of battery.

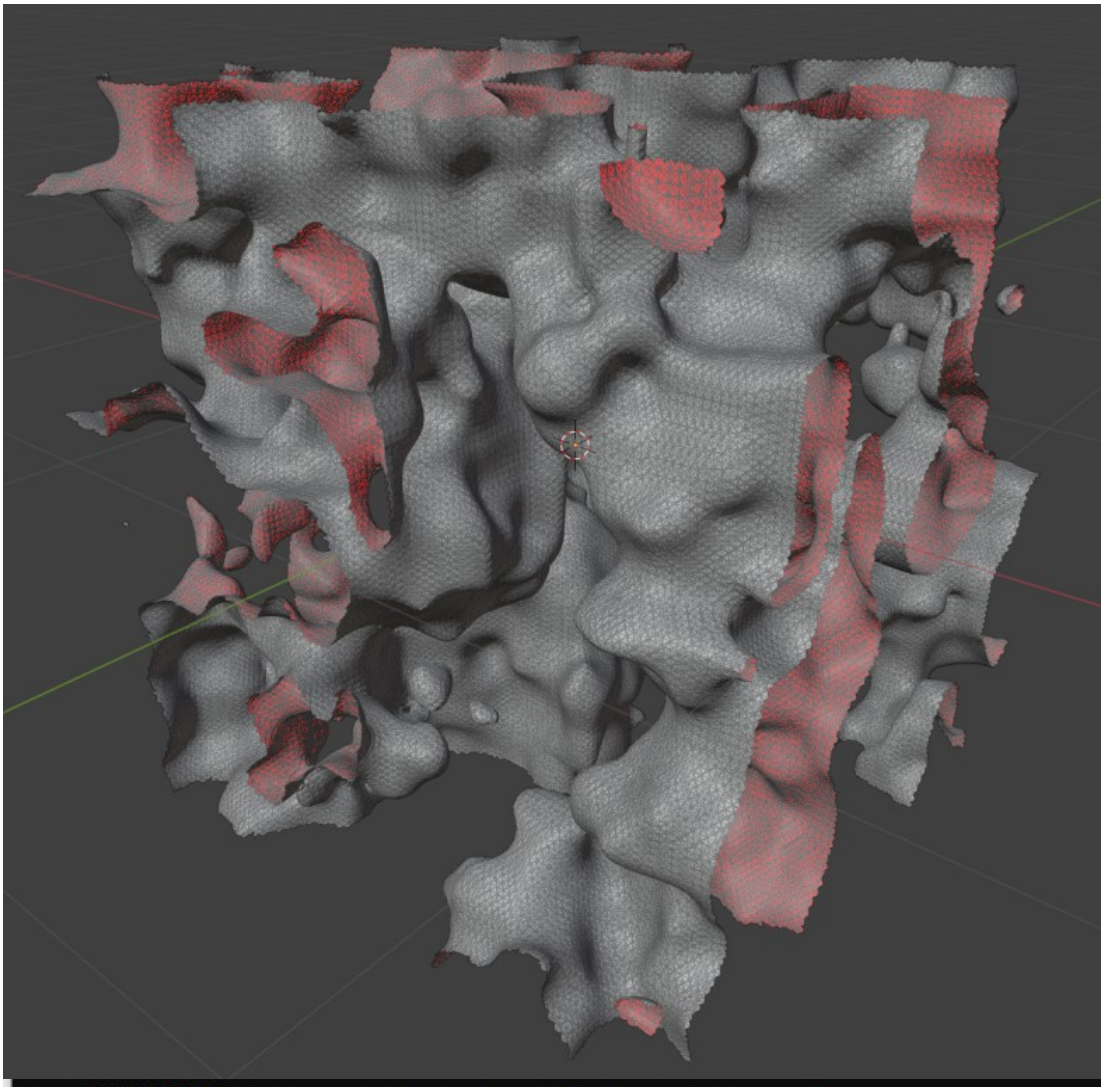


in addition to this, me and Kieran discussed parallelising the vertex pass, as it's the next most significant time-wise. however, i explained the issue of vertex ordering and the need for a shared indexing of vertices - basically, the vertices need to end up in the right order to that the vertex references (indices) assigned to each edge are correct. this means that either you need to put a lock around the vertex array, or provide threads with their own vertex arrays that get merged later. but in the second case, how do you merge things and ensure the vertex references are correct? are you going to update all of the relevant (how do we know which ones??) vertex references when you combine the two (or more!) vertex arrays? at what point does the merging logic take longer than the actual vertex pass itself? with this in mind we decided this might be more trouble than it was worth to parallelise, until literally yesterday morning when i had a thought about how this could be implemented. based on Kieran's suggestion to use an ammortised counter (a concurrency-safe integer variable basically), my proposal was simply to generate vertices alongside a unique index (from the ammortised counter), and then store the vertices along with their unique index in a thread-local vertex array. when merged, rather than merging the arrays sequentially, we copy vertices into the combined array at the positions specified by their accompanying unique index (since these are guaranteed to saturate and be limited to the total size of the arrays), and we should end up with a single vertex array where the unqiue index we used as a vertex reference for edges does actually refer to the same vertex when used as an index into the master vertex array. i haven't implemented this yet, but i likely will in future.

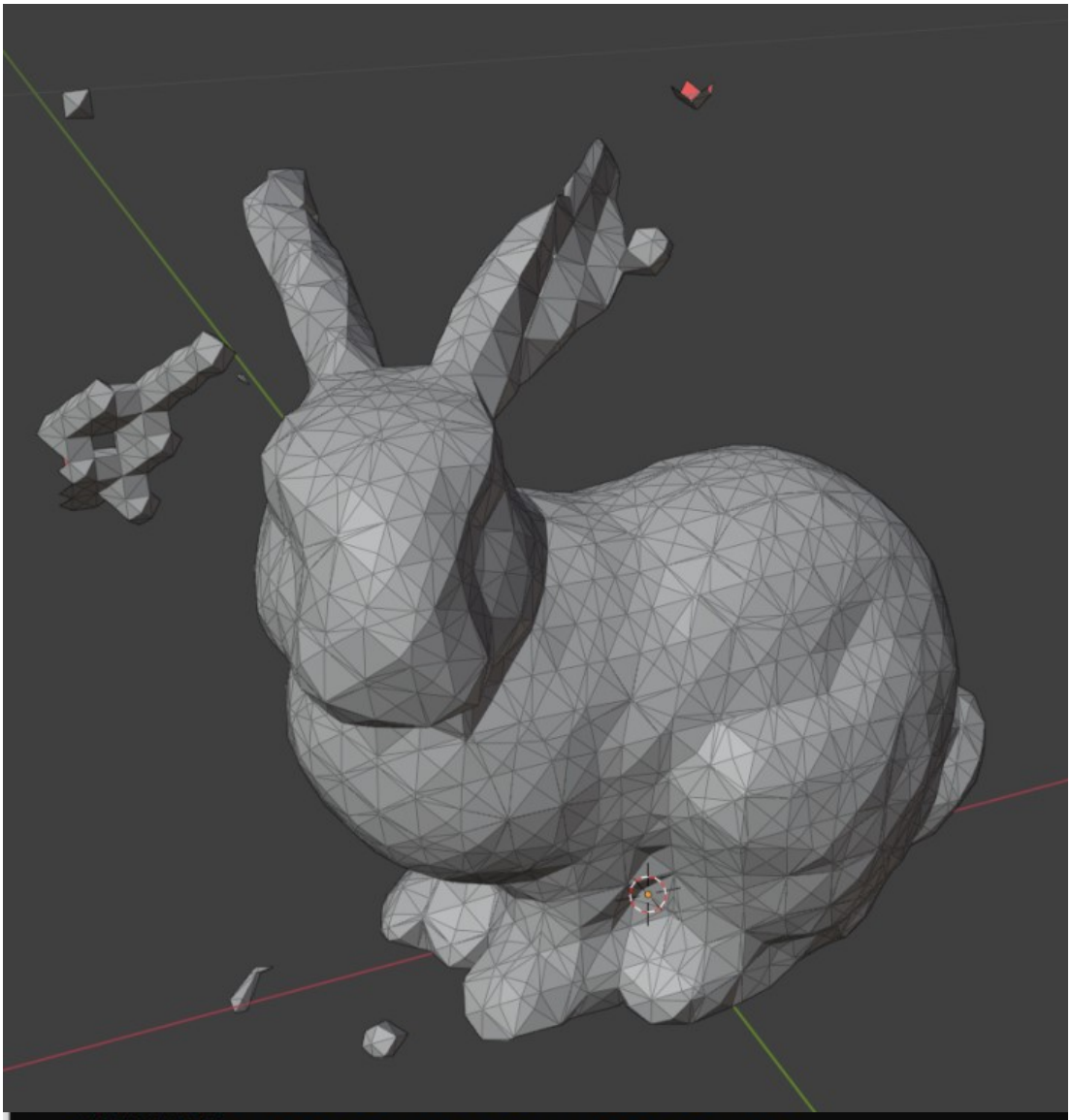
the final, final thing i want to show off is another benchmark. all of them in fact, since im quite pleased with them. enjoy.



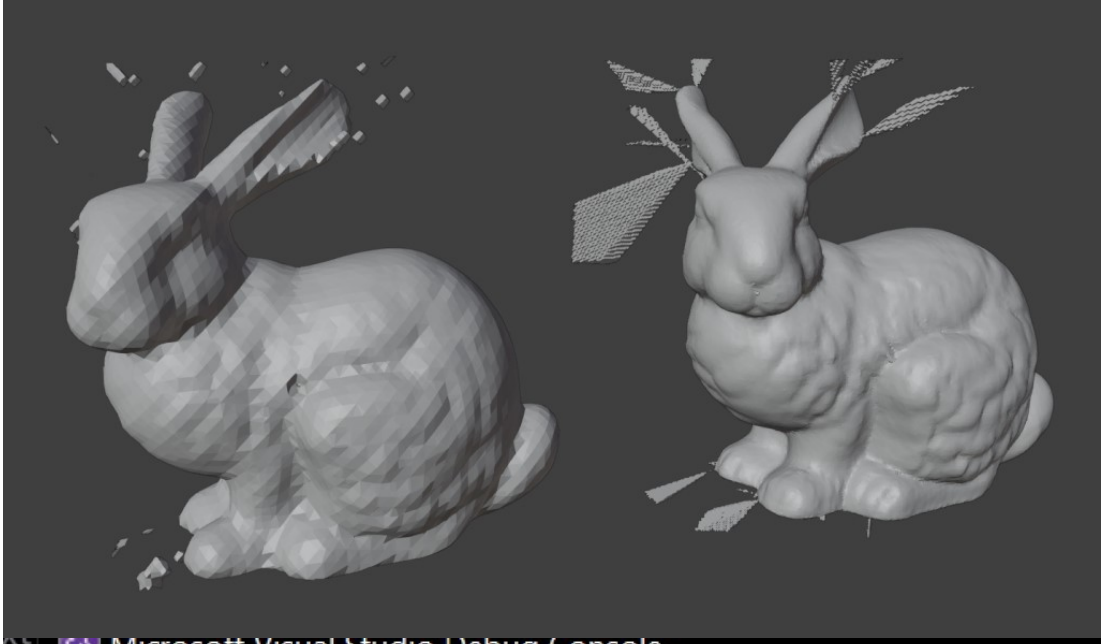
```
-- summary ------------------------------
  sphere test (100 iterations)
  100x100x100 resolution
  results:
    sample points:  2090301 (2112012 allocated)
    edges:          14330300 (29568168 allocated)
    tetrahedra:     12120000 (163968 evaluated)
    vertices:       50890
    indices:        305328
    degenerates:    0
  timing:           0.054907s total
    allocation:     0.000026s (0.046852% of total)
    sampling:       0.002823s (5.142055% of total)
    vertex:         0.047861s (87.167253% of total)
    geometry:       0.004197s (7.643839% of total)
  efficiency (lower number better):
    SP allocation:  101.038658% (8.1 MiB)
    E allocation:   206.333206% (116.8 MiB)
    T evaluation:   1.352871%
---------------------------------------
```



```
-- summary ----------------------------
  fbm test (10 iterations)
  100x100x100 resolution
  results:
    sample points:   2090301 (2112012 allocated)
    edges:           14330300 (29568168 allocated)
    tetrahedra:      12120000 (1328560 evaluated)
    vertices:        409499
    indices:         2428707
    degenerates:     349100282
  timing:            0.228421s total
    allocation:      0.000026s (0.011531% of total)
    sampling:        0.153478s (67.190706% of total)
    vertex:          0.046653s (20.424247% of total)
    geometry:        0.028264s (12.373521% of total)
  efficiency (lower number better):
    SP allocation:   101.038658% (8.1 MiB)
    E allocation:    206.333206% (116.8 MiB)
    T evaluation:    10.961716%
-------------------------------------
```



```
-- summary ----------------------------
  bunny test (1 iterations)
  20x14x17 resolution
  results:
    sample points:  12146 (13024 allocated)
    edges:          76129 (182336 allocated)
    tetrahedra:     60552 (12464 evaluated)
    vertices:       3672
    indices:        21807
    degenerates:    2000002
  timing:           0.585077s total
    allocation:     0.000011s (0.001812% of total)
    sampling:       0.584403s (99.884800% of total)
    vertex:         0.000469s (0.080246% of total)
    geometry:       0.000194s (0.033141% of total)
  efficiency (lower number better):
    SP allocation:  107.228722% (50.9 KiB)
    E allocation:   239.509247% (737.7 KiB)
    T evaluation:   20.583960%
-------------------------------------------
```



Microsoft Visual Studio Debug Console

```
-- summary ---------------------------
  bunny test (1 iterations)
  200x140x170 resolution
  results:
    sample points:  9777911 (9838612 allocated)
    edges:          67584310 (137740568 allocated)
    tetrahedra:     57463200 (1380696 evaluated)
    vertices:       421305
    indices:        2524605
    degenerates:    48400129
  timing:           350.212769s total
    allocation:     0.000024s (0.000007% of total)
    sampling:       349.901428s (99.911100% of total)
    vertex:         0.240084s (0.068554% of total)
    geometry:       0.071241s (0.020342% of total)
  efficiency (lower number better):
    SP allocation:  100.620796% (37.5 MiB)
    E allocation:   203.805542% (544.2 MiB)
    T evaluation:   2.402748%
-----------------------------------------
```

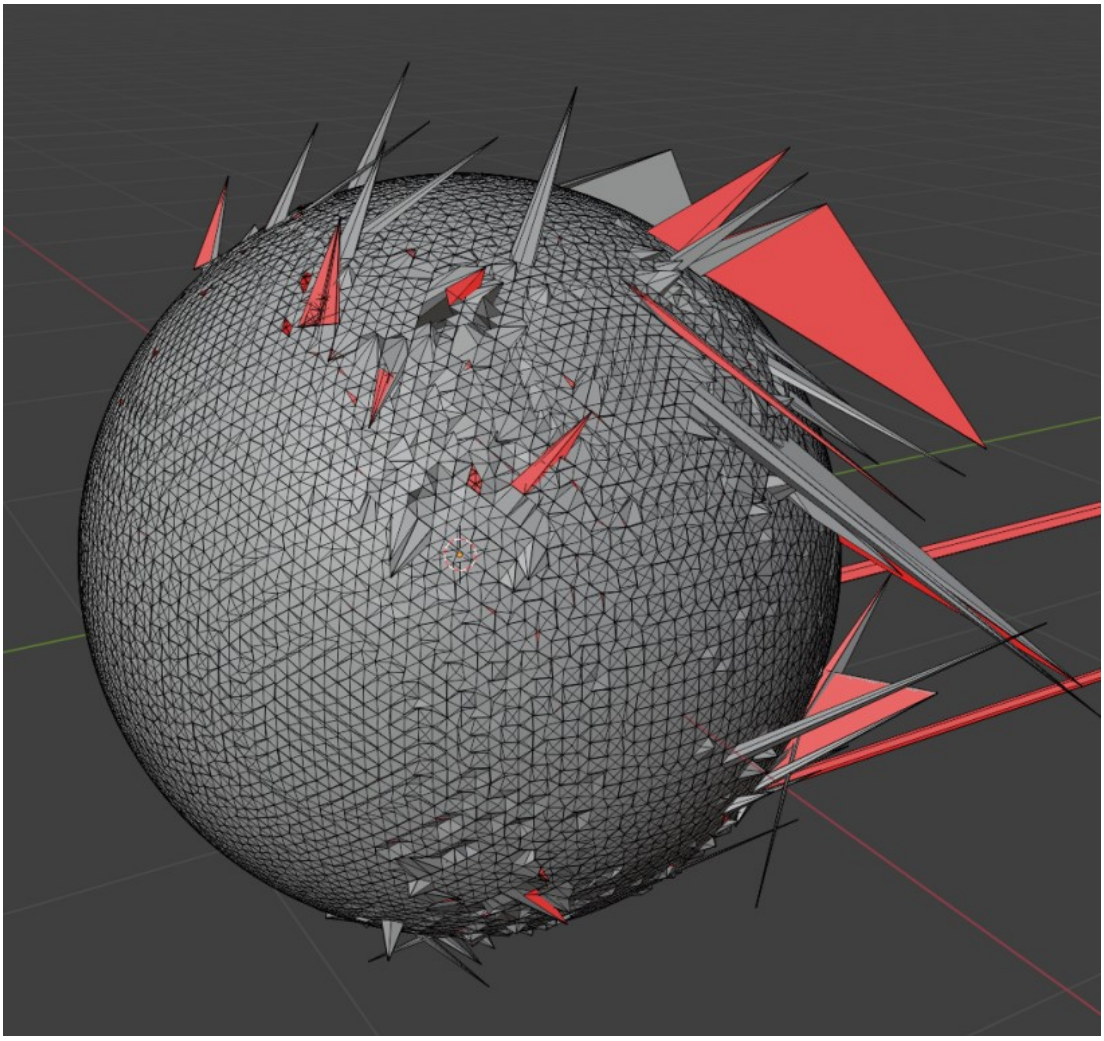
the last two of these benchmarks you may recognise as the infamous Stanford Bunny. you may be wondering what the weird fucking glitched shapes sticking out of the rabbit are, please rest assured no bunnies were harmed in the making of this. this benchmark is implemented via one of the jankiest mesh-nearest-point-signed-distance-field algorithms you have ever seen, which is where the weird artefacts come from. but aside from those, bunny!! as you can see i ran the bunny in two different resolutions, one at a sensible 20x14x17 which took 0.58 seconds (still slow af but that's my terrible SDF algorithm for you), and one at a completely unreasonable 200x140x170 which took a good SIX WHOLE MINUTES to complete, and resulted in a 841-thousand triangle mesh. still, its a cool comparison.

so yeah, that's what ive done last week and this week. i'm finally getting on top of reading papers later this week (today. i will.) after spending a lot of time on the

implementation, though working on the implementation has given me a lot of topics to do more research on. who would have thought!

my remaining implementation tasks are pretty much as follows:
- there are still some degenerate triangles. not sure why, as the mesh itself has no obvious problems, need to look into that
- fix the bunny benchmark, obviously. the weird bugged out patches don't look great to put in a dissertation writeup
- implement the merging algorithm from Treece, Gee, and Prager 1999 in the vertex pass. i actually tried this yesterday and it absolutely did not work at all. more consideration required
- implement the alternative (post-processed) merging algorithm. need to do a bit of research before this
- implement mesh normal generation
- implement behaviour for handling vertices on the outer edge of the sample cube, to ensure square-edged geometry to make chunking possible
- implement the alternative lattice structure (simple cubic). hopefully this won't require me to completely rewrite the vertex/geometry passes and i'll just need to invent some alternative tables. idk....

also i need to start drafting the literature review!! anyway that's it! bye!



merging algorithm status : **NOT WORKING**

Add to this thread...

Go to topic listing

Next unread topic