"Space Art, or Something": Generating 3D Voronoi Diagrams with a divide-and-conquer algorithm

by
Jonathan Emil Mogensen (jmog@itu.dk) &
Marcus Grosen Poulsen (mgpo@itu.dk)

Spring 2021

Course title: Algorithms for Games Development
Course code: KGALGAD1KU

Contents

1 Introduction

Voronoi diagrams are used in digital games as space partitioning structures, but also in other domains, such as biology and physics [Wikipedia, 2021e]. With their large range of use cases in several different domains, it is of great interest to design algorithms for the most effective calculation of the diagram. Many different algorithms have been implemented for this task, some in 3D, though most in 2D. A voronoi diagram partitions space according to a set of seed points, and marks the subareas of an overall space that is the closest to each point, as seen in Figure 1. These subareas are called cells. Discretized voronoi diagrams are voronoi diagrams, represented by a discrete number of points, with each point associated with a single cell in the diagram.

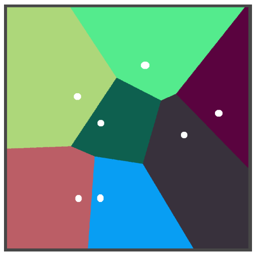


Figure 1: A 2D voronoi diagram with 7 seed points

In this project a 3D version of the divide-and-conquer algorithm from [Smith et al., 2020] has been implemented. The divide-and-conquer algorithm is in its base form an algorithm for generating discretized voronoi diagrams in 2D, with an asymptotic run-time complexity of $O(n * log(m^2))$ where n is the number of seed points, and m is the resolution of a square grid of points. This is already an improvement over the naive approach, where you check the distance between every seed and every point in the grid (which is $O(n * m^2)$), though in 2D it is still worse than Fortune's algorithm which is $n * log(n)$ [Wikipedia, 2021b].

To showcase the developed algorithm, a tech demo called "Space Art, or something" has been made, where the algorithm is utilized to generate a large block of stone, made up of many smaller parts, that each can be broken or recolored individually.

2 Description of the algorithm

The implemented algorithm consists of two overall phases which can be further categorized into four sequential steps. The first phase is the 'space partitioning', which consists of generating seed points for the divide-and-conquer algorithm (DAC) (step 1) and then the DAC algorithm generating the discretized voronoi diagram (or in our case a variant only containing the points that belong to edges and vertices connecting cells or on the rim of the overall partitioning area) (step 2). The second phase consists of converting the discretized voronoi diagram outputted by the DAC algorithm into a non-directed graph representing the voronoi cells (step 3), and then generating meshes from this graph (step 4).

The final output of the overall algorithm is a set of convex mesh objects, with each object representing a single 3D voronoi cell. A general overview of the algorithm is shown in Figure 2.

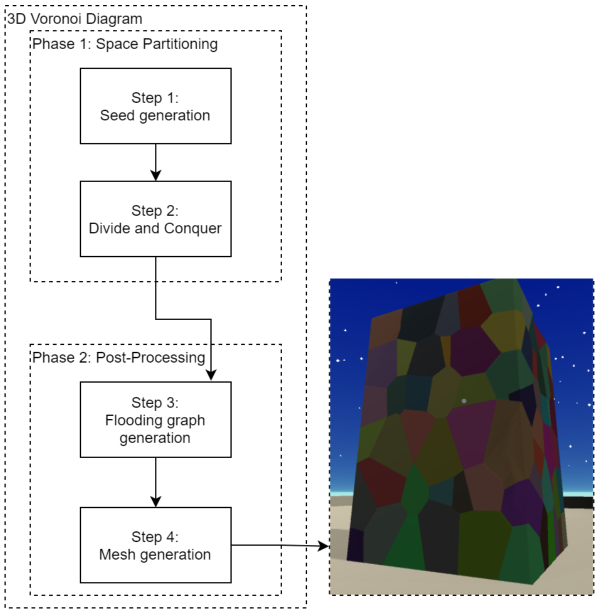


Figure 2: The four steps of the overall implemented algorithm.

2.1 Seed generation

The seed generation part of the algorithm is rather simple. It generates a grid of evenly distributed points that is at a resolution of $(rx*ry*rz)$ within the overall cube, which is of size (sx, sy, sz). It then changes the position of these points to a new random position such that each component c of the point adheres to Equation 1:

$$c = \left[c - \left(\frac{sc}{0.5*rc} - 0.1*\frac{sc}{rc}\right), c + \left(\frac{sc}{0.5*rc} - 0.1*\frac{sc}{rc}\right)\right] \quad (1)$$

This essentially moves the points, such that no two points can ever be in the exact same spot (since it adds a border-area to each "grid-cell" where the point cannot be placed), and that the points are not evenly distributed in a grid, yet still within their own "grid-cell" as to constrain the randomness.
To guarantee reproducibility we have implemented the use of a seed for the random number generator. An example of the seed-point generator can be seen in Figure 3.

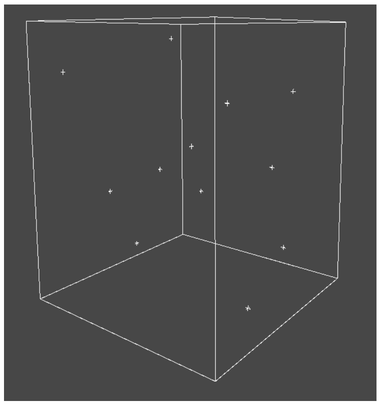


Figure 3: Seed-points generated with a resolution of $2*3*2$, a size of $8*10*8$, and a random seed of 142

2.2 Divide and conquer

This section describes the base version of the DAC algorithm, and further details the extensions that have been made to it during this project.

2.2.1 Base version

Conceptually the base version is rather simple, you start by having a grid of points, where non of the points are associated with a seed. This grid can be represented by a kD array (k for the number of dimensions the algorithm is being implemented for) of integers, where each integer either does not correspond to a seed-point-id (a negative integer) or it does (zero or positive integer). The array will be initialized such that points are not associated with a seed. The algorithm is then run and contains the following 3 steps [Smith et al., 2020, pg.4]:

1. **Calculate Points**: Find the seeds closest to each of the four corner points.

2. **Base Case**: If all four corners are closest to the same seed, then all of the points inside would also be, so they are all assigned to that seed.

3. **Subdivide Case**: If not all the corner points are closest to the same seed, subdivide the area into 4 smaller rectangles, consisting of the top-left, top-right, bottom-left, and bottom-right subrectangles of the original rectangle. These four subareas are then recursively processed according to Step 1.

The above described method has the limitation of only working on rectangles, where the implementation of this project has the further limitation of only working with squares. The asymptotic run-time of this algorithm is $O(n * log(m^k))$, in the average case, where n is the number of seed points, m is the resolution, and k is the number of dimensions. This is a lot better than the naive approach described in section 1, as it only checks $log(m^k)$ against each seed point (see figure 5). This is a best case run-time, as cases could still appear where every point in the grid would need to be checked, but in an average scenario this is not the case.

An example run of the algoritm can be seen in figure 4.

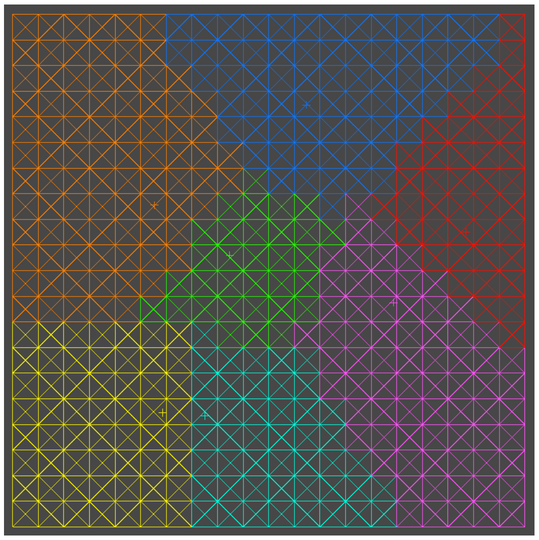


Figure 4: Example of Divide and Conquer in 2D when run on 7 seed-points on a resolution of 20. Each colored square with an X inside is a discretized point marked by the algorithm.

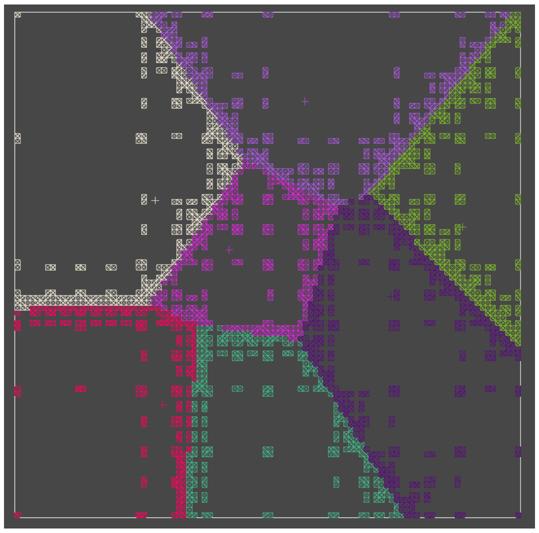


Figure 5: Example of Divide and Conquer in 2D when run on 7 seed-points on a resolution of 20. Each colored square with an X inside is a discretized point where the algorithm has checked the distance from that point to all seed points.

2.2.2 Extended version

The algorithm described in section 2.2.1 was chosen for this project, because it was conceptually easy to extend to 3 dimensions. Going from 2 to 3 dimensions, increases the number of points checked at each `Calculate Points` step from four to eight, and it increases the number of subareas of the `Subdivide Case` to 8 subareas (splitting also on the z-axis).

The original paper only implemented a version working on squares with the resolution being a multiple of 2, which is a restriction that has been loosened for this project, such that the extended implementation can work on any resolution as long as it is square.

This creates some edge-cases that must be handled. In the 2D version, this resulted in the edge-case of being able to end up with a 3x3 square, which cannot be subdivided evenly, and thus must be handled as a 2x2 (which the algorithm natively supports), but then also explicitly checking two 1x2 areas against the seed points, and one area of a single point. In 3D, this got extended with an additional dimension, adding more areas to explicitly handle, since the edge-case of 3x3x3, would end up with a 2x2x2 area, three 2x2x1 areas, three 1x1x2, and a single one point area.

Several different versions of the divide-and-conquer algorithm have been implemented in this project.

First a 2D version similar to the base version, then a memory-optimized one that only used integers in the point-grid-array (instead of the first version that used objects containing the positions and the identifying seed-point-id), after that a 3D-version was implemented. These version were all recursive in nature, but an iterative version of the 3D variant also got implemented, as it was assumed that it might be faster, due to a decreased overhead of having fewer method calls than the recursive variant. In practice no such performance benefit was observed, though this continued to be the version worked on, as the recursive version in theory can throw a stack overflow exception if the resolution is too high, and the iterative version cannot.

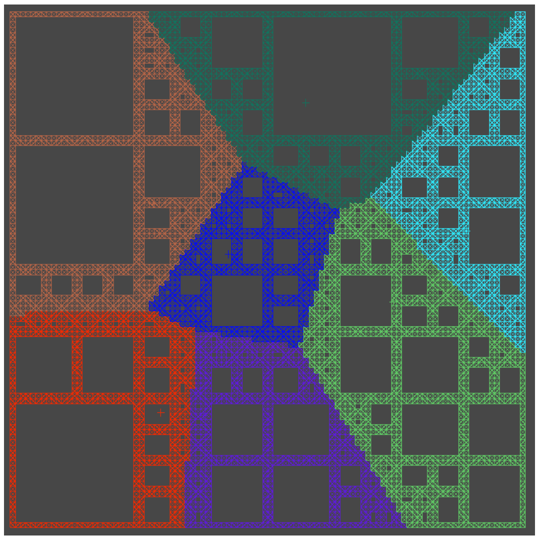


Figure 6: The points marked by this implementation of the DAC algorithm.

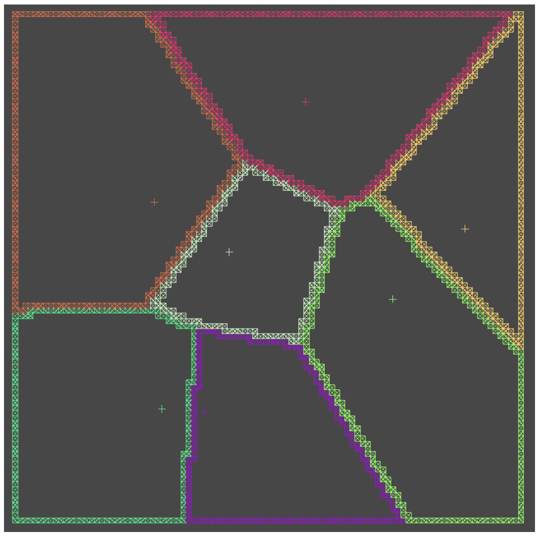


Figure 7: The points marked by this DAC algorithm after the `CullInnerPoints()` method has been called to cut the points that are not on edges of a voronoi cell.

(The figures are visualized in 2D, but the general concept of them still hold in 3D, as the 2D and 3D versions of the algorithm are conceptually similar.)

In Figure 6 and 7 some of the further modifications made to the algorithm, can be seen. These modifications were made so that the algorithm would better fit the use case of generating convex 3D meshes. For generating the meshes, only the points lying on the edges between the cells, and the points lying at the vertices on the corners of cells, are needed, thus the algorithm only adds the points searched and the points lying on the rim of the square that has these searched points as its corners (see Figure 6). Afterwards a method is called to remove all the points from this set, unless it has a neighbouring cell that belongs to another cell (see Figure 7).

k-d tree containing seed points

A further extension made to the divide-and-conquer algorithm, was to use a kd tree to hold the seed-points. This enables the algorithm to only calculate the distance between the gridpoints checked and $O(log(n))$ seed-points in the average case, and $O(n)$ in the worst case [Wikipedia, 2021c]. This makes the `Calculate Points` step much faster, as without the kd tree the search-time would be $O(n)$ in all cases.
This results in the final implementation of the divide-and-conquer algorithm in 3D having an asymptotic run-time of $O(log(n) * log(m^3))$ in the average case. An example run, with manually defined seed-points, can be seen in Figure 8.

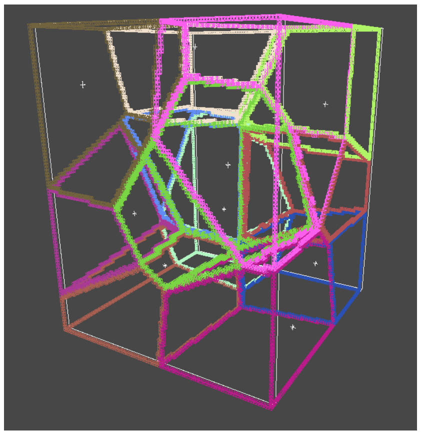


Figure 8: Example run of the final DAC algorithm in 3D.

2.3 Graph generation

The output from the divide and conquer step results in a `VCell` per seedpoint that contains their corresponding outline of `GridPoints`. This step is used to find where these outlines lie adjacent to one another, and generate vertex points on these spots. These vertices will then represent the corners of our polyhedrons, and the connections between these vertices then represents the edges of the polyhedrons. This graph generation is separated into two steps:

- The generation of the vertices
- Establishing the connections between the vertices.

2.3.1 Vertex Generation

The algorithm used for finding the vertices works as follows: For each `VCell`, the algorithm traverses along the outline points, and on each point checks how many `VCells` lie adjacent to it by checking the seedpoint-IDs marked on the surrounding points. The amount of different points surrounding the current point is then used to determine if a vertex should be created or not.

When creating a new vertex they are assigned a priority value that affects the outcome of combining vertices. The priority is determined by the number of points around the current point that belongs to other cells. An example can be seen in Figure 9, where for the point `X` it would have 2 surrounding cells, namely red and green, and a priority of 5 (`x2, x3, x4, x5 & x6`).
Whether a vertex should be created or not depends on where the point is located in the 3D-matrix. The different scenarios are listed below:

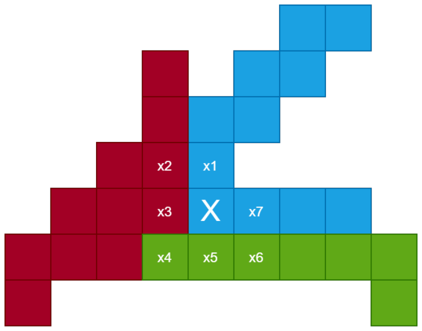


Figure 9: Scanning step principle in 2D. `X` is the current point, `x1-x7` is the surrounding points.

- **Corner point:** This point is always marked as a vertex point. It is given a priority of `Int.MaxValue`.
- **Side line point:** The point is located on the a side line of the cube, aligning with a corner point. It then needs to have at least 1 other cell in its surrounding points to create a vertex. It is given a priority boost of +20.

- **Side plane point:** The point is located on a side plane of the cube. It then needs to have at least 2 other cells in its surrounding points to create a vertex. It is given a priority boost of +10.
- **If none of the above, the point is in the middle:** Needs to have at least 3 other cells in its surrounding points to create a vertex. It is not given a priority boost.

Whenever a new vertex is created, the algorithm looks through all the existing vertices to see if any of them are within a certain range of the new vertex. If there are any, the existing nearby vertices are compared to the new vertex based on their priority. The vertex with the highest priority is then kept and absorbs the cellIDs from the vertex with less priority (this essentially merges the two vertices into one that is shared by both voronoi cells).
An Example: When flooding though the red in Figure 9 at some point the flood will reach `x3` which is a valid vertex candidate, as it has two surrounding cells, and a priority of 4. It will then detect the already existing vertex on `X` and see that `X` has a higher priority, which is 5. So the `x3` vertex will be combined into `X` meaning that `X` now represents both a corner for the red cell and the blue cell.
The reason for this, is to prevent gaps from forming between the final meshes. If the vertices were not combined, it would always result in a small gap between the cells.

2.3.2 Vertex Connecting

When it comes to connecting the vertices, the algorithm uses a similar approach to that of breadth first search [Wikipedia, 2021a]. It goes over each of the generated vertices and starts a search for other nearby vertices.
This search makes use of a set of visited GridPoints and a queue, where each queue-element contains three properties:

- **GridPoint:** The point to investigate when the queue-element is dequeued and analysed.
- **Found:** A boolean for keeping track of whether a vertex has been found or not in the given chain.
- **Chain:** A `LinkedList` of other queue elements that was searched before this.

The algorithm starts the search by analyzing the six adjacent points to the root/starting vertex, each with a new initialized chain. The search then goes in the different directions and adds any adjacent points that has a cellID that is not -1 and is not in the visited set, to the queue. Unless any of the adjacent points is a vertex point, then the search is done and a connection is created from the root vertex to the found vertex. Then all the elements in the chain gets `Found` set to true, so the elements know not to continue the search down this chain.

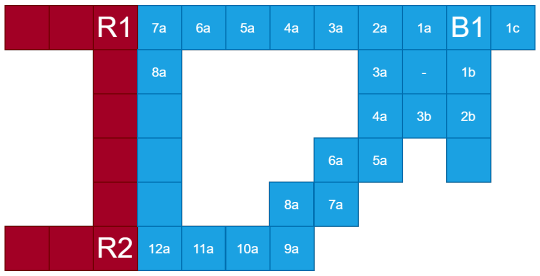


Figure 10: The case of one chain going down two paths. Search starts from `B1`.

A problem arose with the chain logic, because of the breadth first search nature of the algorithm, the chain sometimes went down "multiple paths". An example of this is shown in figure 10 where the numbers in the squares represents the depth of the search and the letters represents what chain the `GridPoint` is a part of. As can be seen the `a` chain goes down towards both `R1` and `R2`. But when the chain reaches `R1` the whole chain gets marked as done, so the search going towards `R2` also gets stopped, so a connection between `B1 & R2` is never established.

A couple different approaches were experimented with to combat this issue.
The first solution tried was having each chain associated with its own visited set, instead of a global set for the whole search. This then resulted in a new problem. Whenever there was a very close vertex, all chains would always end at that vertex. An example of this would be if `R1` was located at `3a`, then both the `a & b` chain would end there.
The second was to check that whenever a new queue element is created, that the element is near the

current head of the chain. If not the chain is assumed to have split down two paths, so the current element initializes a new chain. But this also sometimes caused issues.

In some cases, the search manages to travel past a vertex, before it is found. This results in the chain split occurring at an unwanted time. As can be seen in Figure 11 the search going from `vertex 4` towards `vertex 1` ends up going along the path towards `vertex 2` before `vertex 1` is found. This triggers a chain split, which allows the new chain to traverse all the way to `vertex 2` and make a connection between `vertex 2 & 4`. As can be seen in Figure 12, when splitting is disabled this problem does not occur.

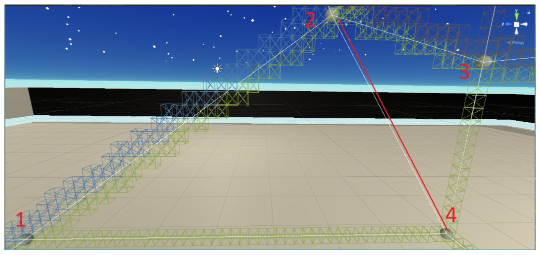


Figure 11: Connection error happening with chain splitting enabled. Wrong connection marked with a red line.

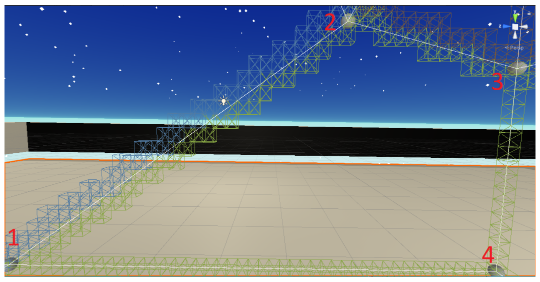


Figure 12: Connection error not present when splitting is disabled.

From these two solutions the second one was chosen. The reason for this was that it is better for there to be too many connections compared to missing connections. This of cause has an impact on the meshes that are generated.
The resulting graph from the the vertex generation and connecting steps can be seen in Figure 13. Where each sphere represents a vertex and the white lines represents the connections between them.

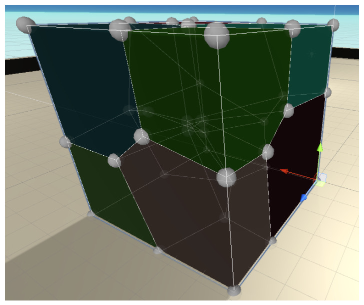


Figure 13: 3D-voronoi diagram with the vertices and connections drawn on top of the generated meshes.

2.4 Mesh generation

The fourth step of the algorithm generates the meshes and game objects that represent each voronoi cell in the diagram. As input it takes the undirected graph generated by step three (see section 2.3), and generates the meshes from this graph. To generate the meshes, this step needs to find which vertices make up each of the individual faces of a cell. Once it has found all of the faces of a single cell, it will generate the triangles of that face, by finding the centroid of each vertex of the face, and then generate polygons in a fan-like pattern, as shown in Figure 14. It should be noted that a method for detecting which side of a given face is facing outwards and which is facing inwards, has not been implemented, and thus the polygons for every face is generated both as front and backwards facing.

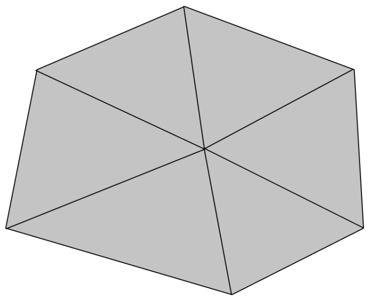


Figure 14: Illustration of the polygon-generation pattern of a single face of a voronoi cell.

To find the vertices of each face, a variant of breadth first search was implemented to search the undirected graph. This variant does not terminate early, and it is utilized to detect cycles in the graph, as each face is represented as a cycle in the graph. The frontier `Queue` of the search algorithm thus contains `LinkedLists`, such that while running the algorithm it can keep track of every link of a chain of searched vertices, and once it reaches a vertex where it detects a cycle, it will create a `List` of the vertices that are a part of that cycle, with the proper order of them to represent the cycle.

This is then run for every vertex of the cell, and then that process is repeated for every cell. Breadth first search is used, as it was assumed that all of the faces of a cell, would be equal to the shortest cycles that could be generated from the subgraph representing the cell. This turned out to not necesserily be the case for every face, and thus the search is not stopped early, but is only stopped after it has searched all vertices in the subgraph, as this exhaustive approach guarantees that it will find all the cycles, but then it also finds cycles that are located internally in the cell.

It also generates some cycles multiple times, but these can be detected and discarded. It also produces some cycles that are not feasible (e.g. $3 \to 2 \to 1 \to 2$) as they contain the same vertex several times, and these cycles are discarded as well.

3 Results

For this project there are two sets of results. One being the experimental measurement's of the algorithms performance, and the other being the implemented tech demo, utilizing this algorithm.

3.1 Run-time experiments

The run-time performance of the algorithm has been measured through Unity's Performance Profiler, to empirically test whether the implementation lives up to the theoretical run-time performance. The results are found in Figure 15, 16, 18 & 19.

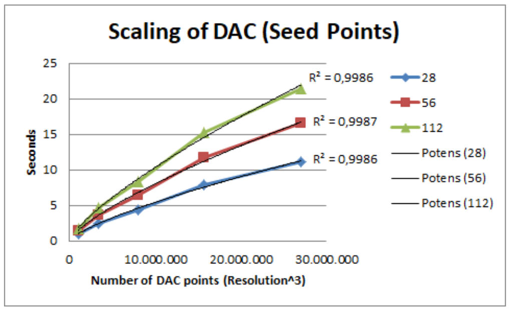


Figure 15: Run time of Phase 1 of the algorithm, if the number of seed-points are fixed, and the resolution varies. Potens means it is a power function.

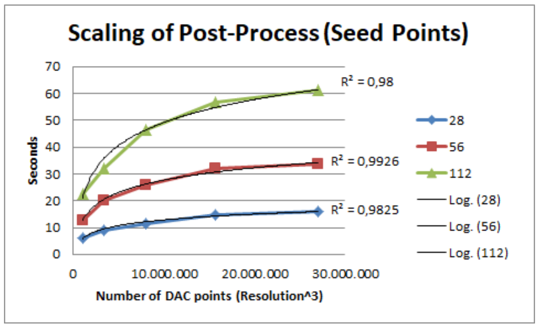
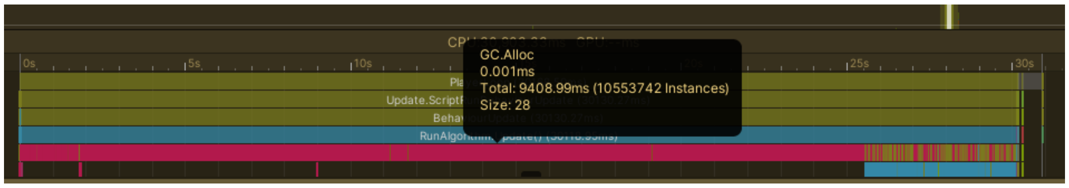


Figure 16: Run time of Phase 2 of the algorithm, if the number of seed-points are fixed, and the resolution varies.



GC.Alloc
0.001ms
Total: 9408.99ms (10553742 Instances)
Size: 28

Figure 17: Screenshot of the Unity Performance Profiler. All the purple/pink areas are related to garbage collector memory allocations.

The results show that the DAC part of the algorithm does in fact scale logarithmically with the number of seed points after the k-d tree has been implemented. Though it also shows that logarithmic scaling has not been achieved in relation to the resolution. This is most likely due to poor memory management, as the standard implementation of array-lists, linked-lists, queues, and sets have been used from the `System.Collections` library, instead of creating custom managed types, which have more optimal memory allocation patterns. The C# garbage collector was found to be a large part of poor performance, as can be seen on Figure 17.
It was also found that Phase 2 scaled linearly when increasing the number of seed points, yet only scaling logarithmically when increasing the resolution.

This results in the run-time of the overall algorithm in practice scaling linearly with the amount of seed-points, and scaling according to a power-function in relation to the resolution, and thus the implementation has not achieved the theoretical run-time scaling.

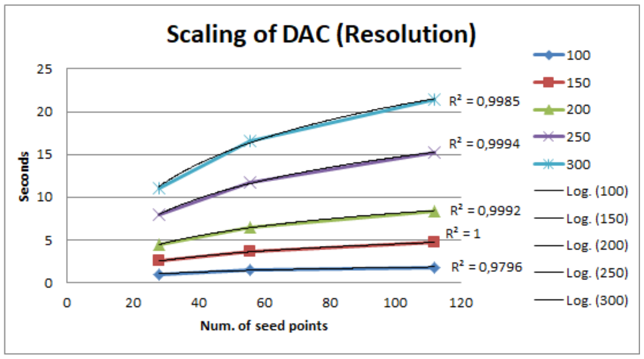


Figure 18: Run time of Phase 1 of the algorithm, if the resolution is fixed, and number of seed-points varies.

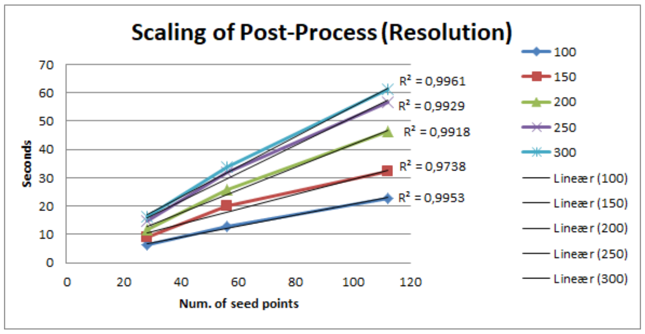


Figure 19: Run time of Phase 2 of the algorithm, if the resolution is fixed, and number of seed-points varies. Lineær means linear.

3.2 The game built on top

With the algorithms in place for generating the 3D-voronoi diagram, a basic game that utilizes the generated meshes was implemented. Its a first-person game, where the player has access to two tools, the hammer and the paint gun. The hammer can be used to destroy segments of the "space rock" (Figure 20), whereas the paint gun can be used to change the color of the rock segments (Figure 21). This resulted in the game "Space Art, or something", which serves as a tech demo for the algorithm.

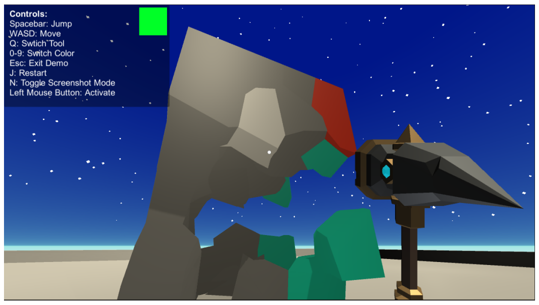


Figure 20: The hammer tool. Used to destroy a segment of the structure.

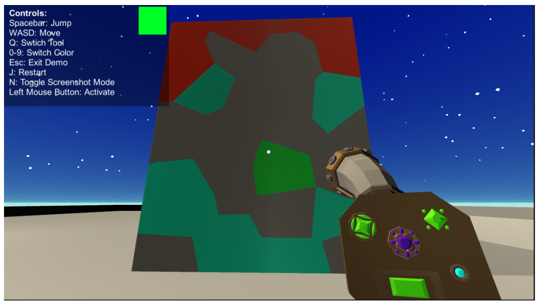


Figure 21: The paint gun tool, used to color a segment of the structure.

4 Discussion

4.1 Shortcomings

4.1.1 Vertex connections

There are still some shortcomings related to the connecting of the vertices, that can have quite a significant impact on the mesh generation. As mention in section 2.3.2 different solutions were explored regarding how to handle the termination of the search, but none of the presented solutions handles all the problems. So there still occurs errors from the graph generation that can impact the mesh generation, leaving either missing or overlapping faces, as can be seen in Figure 22.

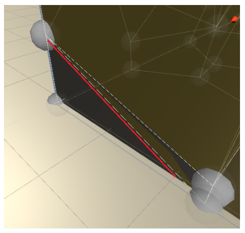


Figure 22: Two mesh faces overlapping, caused by a wrong connection between two vertices.

4.1.2 Mesh generation

Due to the way the meshes are being generated, many extraneous polygons are also being created as part of the mesh. These polygons are the inwards facing polygons of the faces of the cells, and the inner planes that have been generated. On the scale that the algorithm has been run at, this has not caused any particular performance issues of the tech demo at runtime, but if the use case was at a much larger scale, then these excessive polygons could have an impact on the runtime performance of the application.

4.1.3 Memory management

As discussed in section 3.1 the garbage collector ends up taking a lot of time because of the all the overhead that exists int `Systems.Collections` implementation of array-lists. This overheard is associated with managing the size and content of the list, and particularly in this case, the performance degradation comes from having many array-list with elements added and removed frequently. To improve memory management custom implementations of the basic collections from `System.Collections` would have to be implemented, such that they would not resize as often as is currently the case.

4.2 Future work and improvements

4.2.1 Octree for point-grid

Since the divide-and-conquer step of the algorithm stores information about every grid-point in a 3D array, it utilizes $O(n^3)$ memory but has a $O(1)$ time to access each point. A possible improvement could be to use an octree [Wikipedia, 2021d] to hold these points, and incrementally inputting areas of points while running the algorithm, and thus that way utilize the divide-and-conquer algorithm to construct the octree. The octree would thus hold areas marked by discrete points, instead of the discrete points themselves. This would improve the memory utilization, though it would also increase the time needed to access each point, and thus might decrease the runtime performance of the algorithm. There might thus occur a space/speed trade-off in this scenario.

4.2.2 Other game use cases

Cave generation: When doing the connection of the different vertices, the delaunay tetrahedralization could be calculated. This would result in having an overall graph representing the adjacency of polyhedrons. From this another algorithm could be applied to remove certain cells, resulting in an algorithm for generating 3D caves.
3D puzzle pieces: Another game could be to random shuffle the polyhedrons out of the partitioned area, and then having the player drag and drop the polyhedrons back to their original positions. A simple way of knowing if the pieces are back in their correct position is simple, if the centroids of the polyhedron are within a certain range of their original positions, lock it in place.

5 Conclusion

This project has managed to take the divide and conquer space partitioning strategy, originally designed for 2D, implemented it in Unity, and extended it to work in three-dimensional space. Furthermore there has been worked on developing methods to take the result of this space partitioning algorithm, and transform it into a graph that could then be used to generate meshes representing the individual cells of a 3D voronoi diagram. While some of the algorithm steps are not fully optimized, and can produce incorrect results in some edge-cases, it still has ended up being sufficient for the tech demo. The tech demo is a small game utilizing the implemented algorithm, to generate stone blocks with unique formations, and is a sort of space themed stone sculptor simulator. This is just one use case for the algorithm, with more also having been presented.

Software and assets

- Unity Version 2020.2.2f1
- POLYGON Dungeon Realms - Synty Studios https://assetstore.unity.com/packages/3d/environments/dungeons/polygon-dungeon-realms-low-poly-3d-art-by-synty-189093
- GDC2015 + GDC2020 - Sonniss Archive https://sonniss.com/gameaudiogdc
- Made - Dark Fantasy Studio http://darkfantasystudio.com/music/made/
- Polyverse Skies - BOXOPHOBIC https://assetstore.unity.com/packages/vfx/shaders/polyverse-skies-low-poly-skybox-shaders-and-textures-104017

References

[Smith et al., 2020] Smith, E., Trefftz, C., and DeVries, B. (2020). A divide-and-conquer algorithm for computing voronoi diagrams. In *2020 IEEE International Conference on Electro Information Technology (EIT)*. IEEE.

[Wikipedia, 2021a] Wikipedia (2021a). Breadth-first search. https://en.wikipedia.org/wiki/Breadth-first_search. Accessed: 28-05-2021.

[Wikipedia, 2021b] Wikipedia (2021b). Fortune's algorithm. https://en.wikipedia.org/wiki/Fortune%27s_algorithm. Accessed: 29-05-2021.

[Wikipedia, 2021c] Wikipedia (2021c). K-d tree. https://en.wikipedia.org/wiki/K-d_tree. Accessed: 29-05-2021.

[Wikipedia, 2021d] Wikipedia (2021d). Octree. https://en.wikipedia.org/wiki/Octree. Accessed: 29-05-2021.

[Wikipedia, 2021e] Wikipedia (2021e). Voronoi diagram. https://en.wikipedia.org/wiki/Voronoi_diagram. Accessed: 29-05-2021.