Dungeon Procedural Generation in Video Games

Nathan Jovial Hartono - 13523032[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]nathjovi899@gmail.com, 13523032@std.stei.itb.ac.id

***Abstract*—Procedurally Content Generation (PCG) is a widely adopted technique in modern video games to create randomized unique content in an efficient manner. This study examines the dungeon PCG repository by "vazgriz" and explore the algorithms. The algorithms include Delaunay triangulation as a basis for creating the connection between rooms, Prim's MST to obtain a consistent path, and A* pathfinding to determine the generation of elements connecting each node. This study also explores the code implementation of each algorithm and how it represents a graph or node in the context of unity's C#**

***Keywords*—Procedural Content Generation, Delaunay Triangulation, Minimum Spanning Tree, Pathfinding**

I. Introduction

Procedural Content Generation (PCG) has been a popular method in modern video games, enabling the creation of diverse, unique, and replay-able environments, creating unique experiences exclusive to each player. PCG has been seen its implementation in dungeon generation, where the concept of "dungeon" in a video game is a set of rooms, connected to each other, that players can interact, which are uniquely instantiated by the algorithm. With PCG's implementation, it eliminates the labor work of manually designing what a dungeon should be like according to the game designers, instead PCG is capable of generating multiple rooms that can connect to each other with the algorithm defined for this specific notion of PCG. The complexity and variation of layouts will create a gameplay loop that's more engaging and filled with exciting nuances to be explored by the player [2][3].

This paper examines a GitHub repository owned by "vazgriz" [1], presenting the implementation of an intermediate but effective procedural dungeon generation algorithm, which is inspired by the discussion in reddit by user "phidinh6" [4], regarding it's own implementation techniques. The PCG implemented in vazgriz's repository [1] uses three distinct algorithms: Delaunay Triangulation, Prim's Minimum Spanning Three, and A* and or Dijkstra pathfinding. The algorithm ensures the random generation of rooms will always be connected to each other, creating a stable, unique, and random experience for players.

Delaunay Triangulation is a method to create a triangulation network of defined vertices, where it is known for its computational efficiency and robust property which guarantees no overlapping edges and proper triangulation among vertices [5]. Applying MST to the generated graph will yield a reduced graph with simplified connections between each vertex. After that, we finally apply the pathfinding algorithm, preferably A*, to provide path between each of the designated rooms in the dungeon system. This combination of algorithms creates a pipeline that suffices the aesthetics of the layout arrangements while providing proper functionality.

Procedural Content Generation has been widespread among various video games. Algorithms such as cellular automata were implemented for dungeon generation in the earlier days [6]. There also exists other techniques such as BST (Binary Space Partitioning) trees [7] which allowed more structured layouts.

The remainder of this paper is structured as follows. Section 2 explains the fundamental theorems of the algorithms used in the PCG pipeline. Section 3 describes the implementation of the algorithms in vazgriz's GitHub repository and presents the result. Section 4 discusses the potential applications in other sectors, followed by conclusion in section 5.

II. Prerequisites

A. Delaunay Triangulation

Delaunay Triangulation, in the context of computational geometry, is a network of triangles containing a set of vertices connected to each other where for every circumcircle of any triangles no additional vertex lies inside. The Delaunay triangulation maximizes the minimum angle, but it doesn't necessarily minimize the maximum angle or its length of the edge [10]. The circumcircle may overlap with each other as long as no more than 3 vertices are within the border or inside the circumcircle. The triangulation ensures no skinny triangles exist between vertices; this is the byproduct of the circumcircle property [7]. Important to take notice that the coordinates or position of the vertices must not be collinear, or triangulation is going to fail. Below is the defined triangulation [8] :

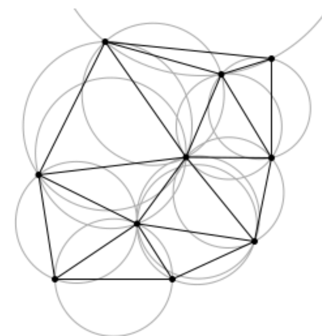

Fig 1. Delaunay Triangulation of Random Points [8]

B. Validating Points in Delaunay Triangulation

Guibas & Stolfi described an interesting technique in determining if a point is within the circumcircle. The expression below determines the position of a point [7]:

$$\begin{bmatrix} A_x & A_y & A_x^2 + A_y^2 & 1 \\ B_x & B_y & B_x^2 + B_y^2 & 1 \\ C_x & C_y & C_x^2 + C_y^2 & 1 \\ D_x & D_y & D_x^2 + D_y^2 & 1 \end{bmatrix} \quad (1)$$

where A, B, C are points, sorted in counterclockwise order, that construct the circumcircle of the triangle and D is the point to be examined. If the determinant of the matrix above is more than zero, then point D lies within the circumcircle. If the determinant is less than zero, then point D lies outside of the circle. If the determinant is zero, then point D lies within the circumference of the circumcircle. Another way to define the expression is to write points A, B, C relative to point D which gives us the expression as below [7]:

$$\begin{bmatrix} A_x - D_x & A_y - D_y & (A_x - D_x)^2 + (A_y - D_y)^2 \\ B_x - D_x & B_y - D_y & (B_x - D_x)^2 + (B_y - D_y)^2 \\ C_x - D_x & C_y - D_y & (C_x - D_x)^2 + (C_y - D_y)^2 \end{bmatrix} \quad (2)$$

which yields a 3 x 3 matrix.

If a triangle is non-Delaunay (i.e. there exists a point inside the circumcircle of the triangle), then we can perform a flip operation to one of its edges. First, we determine the common edge of the triangles, examine the figurine below [11]:

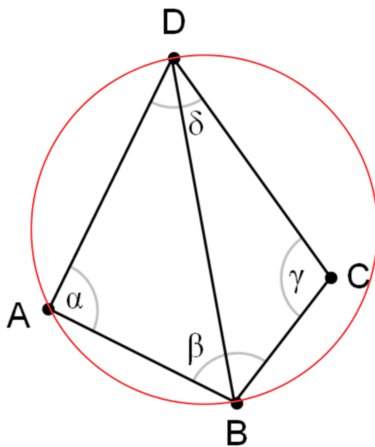

Fig 2. Non-Delaunay Triangle [11]

the edge $\overline{BD}$ represents the common edge of triangle ABC and BCD. We then determine another common edge that divides $ABCD$ into two triangles, in this case it's $\overline{AC}$. We then remove the edge on $\overline{BD}$ and instantiate an edge on $\overline{AC}$ [7]. This creates a valid Delaunay triangle for the four points, as follows [12]:

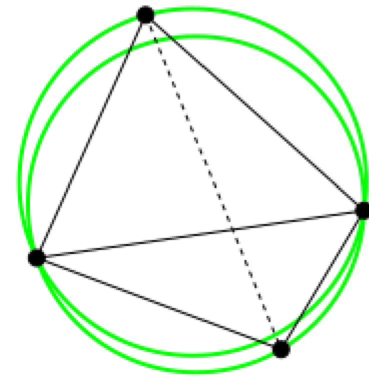

Fig 3. Delaunay Triangle After Flip Transformation [12]

With the ability to flip edges, it leads to a very straightforward algorithm for constructing a Delaunay triangulation, construct random triangulation among the set of points and flip the edges until no more non-Delaunay triangle exists. This approach can take up to $\Omega(n^2)$ edge flips and it is not guaranteed of its convergence between points [13].

C. Bowyer-Watson Algorithm

The Bowyer-Watson algorithm is an incremental algorithm for computing the Delaunay triangulation of finite sets of points in any number of dimensions [15][16]. Every insertion of points will validate the point's position and if the circumcircle of previous iteration contains the new points, then we delete that triangle and construct a new triangle based off the new point. The more detailed approach of this algorithm is as follows [14]:

1. Add a point to the triangulation.
2. Find all existing triangles where it's circumcircle contains the new point. The most optimal approach is to find the first triangle containing the new point then checking the validity of neighboring triangles.
3. Delete the identified triangles, this creates a convex cavity

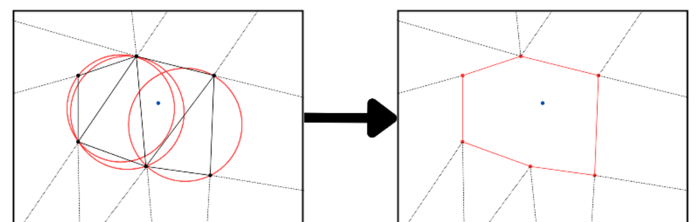

Fig 4. Convex cavity creation

4. Connect the new point to the points of the cavity's boundary.

The initial phase of the algorithm is to define a super-triangle, an infinitely large triangle that encapsulates the entire set [17]. Then we can proceed with the algorithm above. After iterating through every point, we remove the triangles that contain vertices from the super-triangle. In computers we cannot achieve an infinitely large scale of an object, so we identify the outermost points in the set and create the super-triangle vertices off a distance from the outermost points.

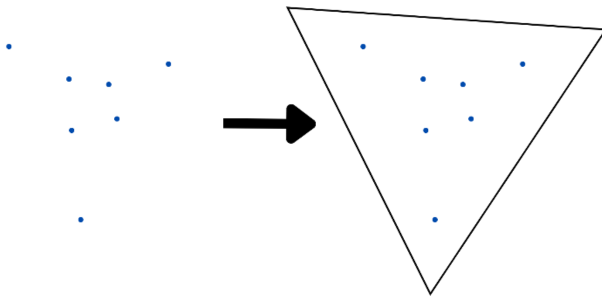

Fig 5. Instantiate super triangle on points

The following pseudocode perfectly describes the simple algorithm used in constructing the Delaunay triangulation from the explanation above, written by author “SupernovaPhoenix” from the Wikipedia page Bowyer-Watson algorithm [18], which in change became the golden standard for multiple repositories such as “Bl4ckb0ne” a.k.a. Simon Zeni’s repository [19]:

```
function BowyerWatson (pointList)
    // pointList is a set of coordinates defining the points to be triangulated
    triangulation := empty triangle mesh data structure
    add super-triangle to triangulation // must be large enough to completely contain all the points in pointList
    for each point in pointList do // add all the points one at a time to the triangulation
        badTriangles := empty set
        for each triangle in triangulation do // first find all the triangles that are no longer valid due to the insertion
            if point is inside circumcircle of triangle
                add triangle to badTriangles
        polygon := empty set
        for each triangle in badTriangles do // find the boundary of the polygonal hole
            for each edge in triangle do
                if edge is not shared by any other triangles in badTriangles
                    add edge to polygon
        for each triangle in badTriangles do // remove them from the data structure
            remove triangle from triangulation
        for each edge in polygon do // re-triangulate the polygonal hole
            newTri := form a triangle from edge to point
            add newTri to triangulation
    for each triangle in triangulation // done inserting points, now clean up
        if triangle contains a vertex from original super-triangle
            remove triangle from triangulation
    return triangulation
```

D. Minimum Spanning Tree

A minimum spanning tree (MST) is a subset of a graph or a spanning tree consisting of edges connecting all the connected nodes while minimizing the total sum of weight on its edges [20]. An MST will have $n-1$ edges for n vertices in the graph. MST has a uniqueness property where if all the edge has a distinct weight, then there will be only one MST created.

This ensures that all the vertices are connected to each other by an edge. In the context of our procedural dungeon generation, the MST will provide a solid foundation for generating rooms without redundant unnecessary corridors (edge) between all the rooms (vertices). There are multiple algorithms to create a MST from an existing graph, but we will be using Prim’s algorithm.

E. Prim’s Algorithm in MST

Prim’s algorithm is a greedy algorithm to find the MST of a connected weighted graph. The algorithm is simplified into three steps as follows [21]:

1. Select an edge with the minimum weight value from graph G and move it into graph T.
2. Select and edge (u, v) with the minimum weight value adjacent to the edges in T, with the exception (u, v) does not create a circuit in T. Move (u, v) into T.
3. Repeat the second step for $n-2$ times, where n is the number of vertices.

The following steps are presented in the figurines below [21]:

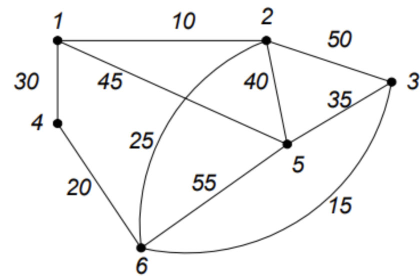

Fig 6. Closed Weighted Graph [21]

Langkah	Sisi	Bobot	Pohon rentang
1	(1, 2)	10	
2	(2, 6)	25	
3	(3, 6)	15	
4	(4, 6)	20	
5	(3, 5)	35	

Fig 7. Prim’s Algorithm Visualization [21]

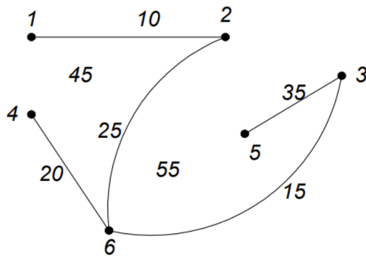

Fig 8. MST of Figurine 6 [21]

The code implementation of MST will may on a Priority Queue to keep track of edges with the smallest weights to ensure efficiency in selecting the edges, but the iterative implementation above explicitly is also a valid strategy.

F. Dijkstra Pathfinding

Dijkstra's algorithm is an iterative pathfinding algorithm for weighted graphs, where the algorithm explores all the shortest paths of the vertices from a source vertex. Defining the source vertex differentiates this algorithm from MST algorithms such as Prim's algorithm. The downside of this algorithm is that it won't work with negative distance values, but that won't be an issue in our case so we assume the distance is always positive.

The algorithm works by defining a source vertex, which is set to zero, while the rest of the vertices are initialized to infinity. A min-priority queue / min-prioqueue is then used to extract every vertex with the shortest distance and to be relaxed to its neighbors. This process loops until all vertices have been processed. Below is the pseudocode implementation using the min-priority queue concept [22]:

```
1   function Dijkstra(Graph, source):
2       create vertex priority queue Q
3
4       dist[source] ← 0                          //
Initialization
5       Q.add_with_priority(source, 0)            //
associated priority equals dist[·]
6
7       for each vertex v in Graph.Vertices:
8           if v ≠ source
9               prev[v] ← UNDEFINED               //
Predecessor of v
10              dist[v] ← INFINITY                //
Unknown distance from source to v
11              Q.add_with_priority(v, INFINITY)
12
13
14      while Q is not empty:                     //
The main loop
15          u ← Q.extract_min()                   //
Remove and return best vertex
16          for each neighbor v of u:             //
Go through all v neighbors of u
17              alt ← dist[u] + Graph.Edges(u, v)
18              if alt < dist[v]:
19                  prev[v] ← u
20                  dist[v] ← alt
21                  Q.decrease_priority(v, alt)
22
23      return dist, prev
```

First, we define the min-prioqueue with the source vertex's distance set to 0 and the rest set to infinity stored in the prioqueue. We also define a list of predecessor relations, prev, where every vertex such as {A: undefined, B: undefined, …}, which are now undefined. The algorithm then proceeds to extract the minimum vertex with the minimum value of the distance and iterate through its respective neighbors. Each iteration will "relax" the vertex where the distance is the value of the previously extracted vertex added with the edge distance between both vertices. If the resulting distance is less than the current value of the distance, then we update the distance at that vertex while also updating the relation list prev.

G. A Pathfinding*

A* algorithm combines Dijkstra's algorithm information, favoring vertices close to the starting point, and The Greedy Best-First-Search algorithm information, favoring vertices close to the goal. It's an iterative algorithm that, like Dijkstra's, requires a starting vertex. Each iteration of the loop examines the vertex with the lowest value of $f(n)$ where $f(n)$ is represented as follows [23]:

$$f(n) = g(n) + h(n) \quad (3)$$

$g(n)$ represents the exact cost path from the starting point to the assigned vertex n, the property of Dijkstra's algorithm, and $h(n)$ represents the estimated cost from the assigned vertex n to the goal, the heuristic property of Greedy Best-First-Search algorithm. The heuristic property of this approach suggests [24]:

1. If $h(n)$ is equal to zero, then only $f(n) = g(n)$ resulting in Dijkstra's algorithm.
2. If $h(n) \gg g(n)$, then A* turns into Greedy Best-First-Search.

The scale measurements of both values must be the same, as in if $h(n)$ is measured in meter then $g(n)$ must be measured in meter as well. The pseudocode below, provided by Patel [25], displays the similarity of Dijkstra's algorithm to A* that properly expresses the theory above:

```
OPEN = priority queue containing START
CLOSED = empty set
while lowest rank in OPEN is not the GOAL:
  current = remove lowest rank item from OPEN
  add current to CLOSED
  for neighbors of current:
    cost = g(current) + movementcost(current,
neighbor)
    if neighbor in OPEN and cost less than
g(neighbor):
      remove neighbor from OPEN, because new path is
better
    if neighbor in CLOSED and cost less than
g(neighbor): ⁽²⁾
      remove neighbor from CLOSED
    if neighbor not in OPEN and neighbor not in
CLOSED:
      set g(neighbor) to cost
      add neighbor to OPEN
      set priority queue rank to g(neighbor) +
h(neighbor)
      set neighbor's parent to current
```

III. Implementation and Result of Vazgriz's Repository

A. Representing a graph in code

The author of the repository represents a graph's vertex and edge in the namespace Graphs. The Vertex class consists of the attribute Position of unity's data type Vector3 with methods of checking equality with other Vertex objects. The Vertex extends to associate with a generic T item, a way to store additional data. The Edge class consists of the attributes U and V of Vertex with methods of checking equality with other Edge objects [1].

B. Generate random rooms

The author creates a class Room containing the attributes of bounds as RectInt with the method Intersect to validate if two Room objects are intersecting according to its definition. The author generates rooms based off the determined roomCount as integer, size as Vector2Int, roomMaxSize as Vector2Int. The process iterates with roomCount as the loop count. The loop begins by determining a random location of the room according to the X and Y boundaries of size. The roomsize as Vector2Int is initialized with random values between 1 and roomMaxSize. The process creates a new object newRoom to with it's new location and roomSize. The process also creates a buffer Room, acting like an invincible border around newRoom [1].

```
Room newRoom = new Room(location, roomSize);
Room buffer = new Room(location + new Vector2Int(-1, -1),
                       roomSize + new Vector2Int(2, 2));
```

Fig 9. Buffer Room Generation [1]

The figure above explains how buffer encapsulates newRoom. This ensures that when running validation tests, no rooms are placed directly side by side. If the size and location of roomSize are valid, we add newRoom to the list rooms while also instantiating it in the scene view. The figure below displays the result of the random generation of the algorithm [1]:

Fig 10. Random Room Generation [1]

C. Defining the Minimum Tree

The process begins by defining *vertices* as a list of Vertex. For every room that has been generated, find the center point of each room and store it in *vertices*. After obtaining all of the room's data, perform the Delaunay triangulation for every room utilizing the Bowyer-Watson algorithm.

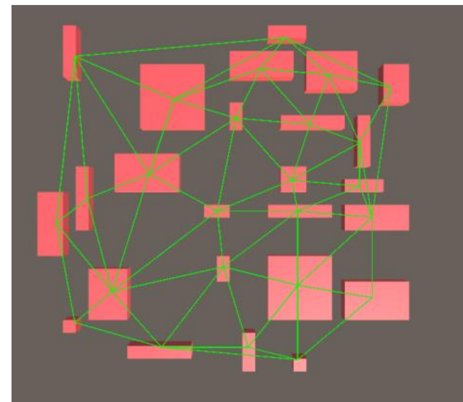

Fig 11. Room Triangulation [1]

This yields a graph of properly defined triangulations between each room. Prim's MST is applied to the graph above, with each edge's weight considered as the Euclidean distance between two Vertex objects. The process initializes on defining openSet as a list of Vertex that stores all of the edge data, while closedSet as a list of Vertex initially stores the starting point. The process will iterate through every single point in openSet. Each edge will be validated for its connectivity to the MST and its cycle property. The process finds the smallest weighted edge of the MST and moves it from openSet to closedSet. If there are no edge, then the process will break the loop iterating every point, an edge case detection to disconnected graphs. The MST result is as follows [1]:

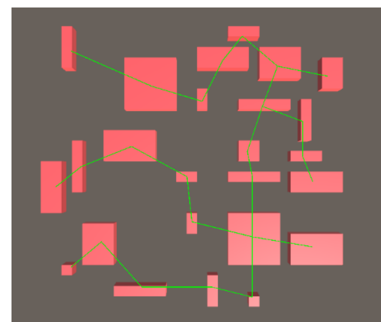

Fig 12. MST of Triangulation [1]

The process also adds extra edges by chance (12.5%) after the MST process, an example is an edge connecting the top most left corner room to the top most right corner room at Fig 12.

D. Connecting the Nodes

The process involves finding the most optimal path along the edge of the MST and then reconstructing it immediately after reaching the final destination. The author initializes a class Node with attributes: Position as Vector2Int, Previous as Node, storing the reference of the previous Node, and Cost as float, a struct PathCost with attributes: traversable as bool and cost as float, neighbors as list of Vector2Int which determines the movement, grid as Grid2D of Node, queue as PriorityQueue, closed as Hashset of Node, and stack as Stack. The object grid is defined with size of room generation and every x and y coordinate of grid is declared a new Node. The implementation of A* by the author is similar to the pseudocode provided above. The process begins by resetting every Node, setting the cost to positive infinite except the starting Node, which is set to 0. The

newCost is the sum of its respective $h(n)$ and $g(n)$ values. After finding the most optimal path, the process will backtrack from the ending node to the starting node storing in result. The process pushes the current Node into stack and traceback to its previous Node repeating the process until no reference to the previous Node is found. The process reverses the order of the Nodes by pushing it to stack and then popping out to result. The resulting process is as follows [1]:

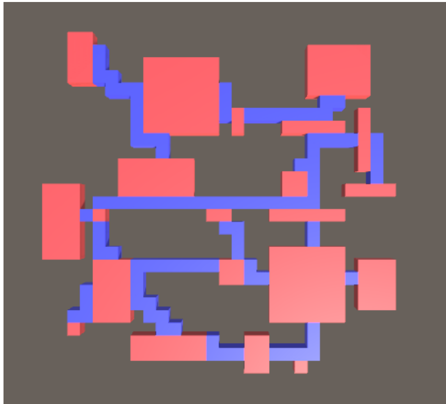

Fig 13. Result of Dungeon PCG

V. CONCLUSION

The pipeline method introduced in this paper for dungeon PCG is deemed to be effective for generating random room and hallways that connect to each other. We explored the PCG method by the user "vazgriz" [1] and found the relation between Delaunay triangulation, Prim's algorithm, and pathfinding algorithms such as A*. Delaunay triangulation is useful for determining a graph that represents the connectivity of each rooms in an organized way, Prim's MST algorithm creates a path following the algorithm, and A* connects each room with the determined path from the MST. The success of this approach suggest the applicability of this pipeline in various other sectors such as path planning in robotics. Future works will explore the strategy and methods implemented for 3D PCG generation, optimizing Delaunay triangulation generation, and implementing complex triangulation algorithms to create more complex structures. Overall, this paper highlights the effectiveness of this pipeline to achieve consistent generation.

VI. ACKNOWLEDGMENT

I would like to thank my family and friends for supporting me throughout the period of me researching this paper. I would also like to thank my professor Ir. Rila Mandala, M.Eng., Ph.D. for providing lectures about Discrete Mathematics and building a foundation of the subject to kickstart this research.

REFERENCES

[1] Vazgriz, "Dungeon Generator," GitHub repository, [Online]. Available: https://github.com/vazgriz/DungeonGenerator. [Accessed: Jan. 3 - 8, 2025].

[2] J. Togelius, N. Shaker, and M. Nelson, Procedural Content Generation in Games: A Textbook and an Overview, Springer, 2016.

[3] K. Compton and M. Mateas, "Procedural content generation in games: A textbook overview," in IEEE Transactions on Computational Intelligence and AI in Games, vol. 6, no. 1, pp. 1–14, Mar. 2014.

[4] 6. [Username], "Procedural dungeon generation algorithm explained," Reddit. [Online]. Available: https://www.reddit.com/r/gamedev/comments/1dlwc4/procedural_dungeon_generation_algorithm_explained/. [Accessed: Jan. 3, 2025].

[5] F. Aurenhammer, "Voronoi diagrams—a survey of a fundamental geometric data structure," in *ACM Computing Surveys*, vol. 23, no. 3, pp. 345–405, Sep. 1991.

[6] F. Blomqvist and P. Wang, "Cellular automata for dungeon generation," in International Game Developers Association Conference Proceedings, 2018.

[7] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, Computational

[8] Geometry: Algorithms and Applications, 3rd ed. Berlin, Germany: Springer, 2008.

[9] By Gjacquenot - Own work, File:Delaunay circumcircles.png (Nü es), Public Domain, https://commons.wikimedia.org/w/index.php?curid=30370476. [Accessed: Jan. 6, 2025].

[10] [9] L. J. Guibas and J. Stolfi, "Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams," *ACM Transactions on Graphics (TOG)*, vol. 4, no. 2, pp. 74–123, Apr. 1985.

[11] H. Edelsbrunner, T. S. Tan, and R. Waupotitsch, "An $O(n^2 \log n)$ time algorithm for the minmax angle triangulation," SIAM Journal on Scientific and Statistical Computing, vol. 13, no. 4, pp. 994–1008, 1992.

[12] By Nü es - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=703333. [Accessed: Jan. 6, 2025].

[13] By Jespa - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=56658046. [Accessed: Jan. 6, 2025].

[14] R. Seidel, "Constrained Delaunay Triangulations and Voronoi Diagrams," in Proceedings of the 3rd Annual Symposium on Computational Geometry (SoCG), Waterloo, Canada, 1987, pp. 178–191.

[15] C. A. Arens, "A note on the Bowyer-Watson algorithm for constructing Delaunay triangulations," Department of Geodesy, Faculty of Civil Engineering and Geosciences, Delft University of Technology, GDMC Publications, 2002. [Online]. Available: https://www.gdmc.nl/publications/2002/Bowyer_Watson_algorithm.pdf. [Accessed: 05-Jan-2025].

[16] A. Bowyer, "Computing Dirichlet tessellations," The Computer Journal, vol. 24, no. 2, pp. 162–166, 1981.

[17] D. Watson, "Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes," The Computer Journal, vol. 24, no. 2, pp. 167–172, 1981.

[18] J. R. Shewchuk, "Delaunay Refinement Algorithms for Triangular Mesh Generation," Computational Geometry: Theory and Applications, vol. 22, no. 1–3, pp. 21–74, 2002.

[19] SupernovaPhoenix, "Bowyer–Watson algorithm," Wikipedia, 7-Aug-2014. [Online]. Available: https://en.wikipedia.org/wiki/Bowyer%E2%80%93Watson_algorithm. [Accessed: 05-Jan-2025].

[20] Bl4ckb0ne, "Delaunay Triangulation," GitHub repository, 2018. [Online]. Available: https://github.com/Bl4ckb0ne/delaunay-triangulation. [Accessed: 05-Jan-2025].

[21] "scipy.sparse.csgraph.minimum_spanning_tree - SciPy v1.7.1 Manual," Numpy and Scipy Documentation — Numpy and Scipy Documentation. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csgraph.minimum_spanning_tree.html. [Accessed: Jan. 05, 2025].

[22] R. Munir, "Pohon Bagian 1," Discrete Mathematics Lecture Notes, School of Electrical Engineering and Informatics, Institut Teknologi Bandung, 2024-2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/23-Pohon-Bag1-2024.pdf. [Accessed: Jan. 5, 2025].

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed. Cambridge, MA: MIT Press, 2009, pp. 658–664.

[24] A. Patel, "Amit's Thoughts on Pathfinding: A* Algorithm Comparison," Stanford University, [Online]. Available: https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html. [Accessed: 07-Jan-2025].

[25] A. Patel, "Amit's Thoughts on Pathfinding: Heuristics," Stanford University, [Online]. Available: https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html. [Accessed: 07-Jan-2025].

[26] A. Patel, "Implementation Notes," Amit's Thoughts on Pathfinding, 2025. [Online]. Available: https://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html. [Accessed: 07-Jan-2025].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 8 Januari 2024

Nathan Jovial Hartono - 13523032