Developing Algorithms for Crack-Free Rendering of a Quad-Tree LOD Algorithm

By Gvidas Danielius

15/05/2024

Computer Science (Game Engineering) BSc

Supervisor: Gary Ushaw

Word Count: 10,015

Abstract

Video games have been using procedural terrain generation for decades, with one of the algorithms used for this being a Quad-Tree implementation of this. This implementation has the unfortunate downside of having T-Junctions among the mesh geometry, which cause cracks in the terrain. The aim of this dissertation is to remove these cracks in an efficient way via the implementation of three unique algorithms designed for this problem. The three algorithms are chosen to give a wide range of results when testing. The first algorithm deploys a skirt around the edge of a mesh to cover any cracks with a flat wall. This has been found to create a strong visual discontinuity, but to balance has the best performance and resource consumption. The second algorithm is similar but deploys an extra row of vertices around each edge to overlap with surrounding nodes. The visual discrepancies are much less noticeable, and only at a slightly larger resource cost compared to the first algorithm. The final algorithm is the most complex and involves displacing edge vertices to match the heights of the neighbour's edge. The implementation of this algorithm uses a binary coordinate system to find the path of a neighbour in any given direction. This final algorithm has seamless borders and as such is the most effective at covering the gap but comes at an extremely high cost especially with higher vertex counts within a mesh. Each algorithm has a specific preference to certain situations where they outperform other algorithms, as such no conclusive answer can be given to the question of which is best.

Acknowledgements

I firstly like to thank my supervisor Dr Gary Ushaw for their support throughout the project. I would also like to thank Dr Rich Davison and Dr Giacomo Bergami for helping me within weekly supervisor meetings and answering any questions about my project when my supervisor was unavailable.

Declaration

I declare that this dissertation is my own work except where otherwise stated.

Table of Contents

1. Introduction

In the ever-evolving landscape of video games, there has been a constant increase in difficulty in the design in terms of scope and scale, which also applies to world-building within open-world games. Hardware advances present a significant challenge with time and cost by increasing the potential that can be achieved, as such studios have employed the use of procedural content generation, a technique that automates content creation, to fill much of the repetitive work of designing an open-world environment. This technique brings many benefits to games that use it, including improved performance and significant time saving.

The need for procedural terrain generation also brings the need for level-of-detail (LOD) algorithms to reduce mesh complexity and improve rendering times. One of such algorithms is an implementation of a data structure called a Quad-Tree. This allows for discrete levels of detail to be used to remove unnecessary polygons at far distances efficiently. Unfortunately, this algorithm has an inherent problem in its geometry which I will refer to as the T-Junction problem.

A T-Junction is a geometric anomaly that occurs when the endpoint of one edge meets the middle of another edge, rather than a vertex. This can lead to visual artifacts and rendering issues by causing small cracks along these edges. The challenge comes from designing a Quad-Tree LOD implementation without the formation of any T-Junctions within the geometry.

Addressing this T-Junction problem brings many benefits, especially towards creating a more immersive gaming experience. T-Junctions cause noticeable gaps and flickering in the terrain, breaking the illusion of the game, and making it appear less polished and professional.

1.1 Aim and Objectives

The aim for this project is to develop and analyse three different algorithms of covering up cracks caused by T-Junctions within the geometry of the Quad-Tree mesh. These algorithms will be investigated and compared to each other in several different situations to find the optimal one for any given use case.

1.1.1 Objectives

1. **Identify and Research at least three algorithms for covering cracks formed by T-Junctions in geometry:** To accomplish my aim, algorithms must first be identified and ranked based on their potential in the departments of visual fidelity and performance.
2. **Implement three of these algorithms to the best of their capabilities:** The three best algorithms from the previous step are chosen and implemented to the best they can be. This last requirement is necessary to allow each algorithm to have a fair chance at comparison in the next step.
3. **Test and analyse the algorithms to find the most effective out of the three:** Each algorithm will undergo a series of tests to evaluate performance. These tests will include evaluating performance and resource consumption in several unique environments. An in-depth analysis will also be performed to determine which algorithms are the most effective in which situations.

1.1.2 Changes since Proposal

There have been several changes to the project since the proposal was first submitted. The main significant change has been a pivot in the overall project plan. Initially, the idea was to

create and analyse three unique LOD algorithms, each handling the problem of reducing mesh complexity at far distances using different methods. However, this plan proved to be too ambitious given the complexity of developing multiple complex algorithms from scratch, especially considering the time constraints.

After some consideration, the focus of the project was modified to the current plan of creating three unique algorithms for fixing cracks in a Quad-Tree LOD algorithm. While this change is quite significant, it keeps the structure similar as the requirements are to research and implement three unique algorithms to fix a problem. This new project plan is still sufficiently complex enough for a project of this scale but ensures that it can be completed effectively.

Another modification was the removal of the objective for turning the flat plane of the current terrain into a sphere to simulate a planet. This objective would have disrupted the overall goal of the project as I believe it is not relevant to the other objectives and is there for the sake of having another objective.

The ethics checklist has been reconsidered due to these substantial changes to the project and has not changed any ethical concerns. No part of the project is involved with potentially sensitive areas as it is still an implementation of software.

1.2 Dissertation Structure

Introduction

Introduces the project's goals and details the problem and possible solution.

- Context/Motivation
- Aim and Objectives
- Changes since Proposal
- Dissertation Structure

Implementation

Description and key details of the implementation of all three algorithms as well as the base algorithm.

- Unity
- Project Overview
- Implementing Quad-Tree Algorithm
- Algorithm One: Overlap Method
- Algorithm Two: Skirt Method
- Algorithm Three: Modified Skirt Method
- Algorithm Four: Neighbour Height Matching

Testing

Description of the tests implemented and used to evaluate all algorithms.

- Results Setup

- Measuring Mesh Generation Time
- Measuring Run-Time Performance
- Measuring Memory Consumption
- Test Results

Evaluation

Analysing of results and other information to deduce which algorithm is best for which situation.

- Offset Algorithm Evaluation
- Skirt Algorithm Evaluation
- Overlap Algorithm Evaluation
- Ease of Implementation

Conclusion

- Reflection of Objectives
- Personal Development
- Future Work

2. Background Research

Perlin Noise

Perlin Noise is a noise function developed by Ken Perlin in 1983 and is commonly used in computer graphics to create natural looking landscapes and textures. ”The purpose of the Noise function is to provide an efficiently implemented and repeatable, pseudo-random signal over R3 (three-dimensional space)” [1]. Unlike random number generators before it, Perlin noise produces smoother, more continuous variations which make it ideal for generating anything from landscapes to clouds. The algorithm accomplishes this by interpolating between gradients at the corners of a grid, creating seamless transitions between values. It’s ability to simulate such a versatile range of appearances has made it a staple in procedural content generation.

Procedural Terrain Generation

Procedural Terrain Generation either utilises some sort of noise field or other complex algorithms to create vast and varied landscapes algorithmically, with minimal need for manual input. There are many different techniques that can be employed to generate these immersive landscapes, but the most common by far is the use of Perlin Noise to generate this noise field as it is the simplest to implement and understand. A single layer of Perlin Noise is usually bland so many layers are added on top of each other as Xuexian et. al. mentions: “a fractal noise can be created from multiple octaves of Perlin noise” [5]. Each subsequent layer has its amplitude decreased and wavelength shortened to create higher levels of detail.

Level of Detail

To prevent the unnecessary rendering of small polygons at far-away distances to the camera, a Level-of-Detail system is introduced to reduce overall resource consumption. The fundamental idea behind LOD is to adjust the complexity of 3D models based on their distance to the camera. Objects closer to the viewer are rendered with more vertices, while those further away are reduced in quality. This approach significantly reduces the number of polygons that need to be rendered by the GPU.

The use of these algorithms is particularly important in open-world games and other large-scale simulations where the large scale of the game world can overwhelm the resources available to a player. By managing these objects well, frame rates and resource consumption can be improved.

Terrain Rendering using a Quad-Tree

Level of Detail can also be applied to terrain rendering instead of just 3D models within a world. This is done using a data structure called a Quad-Tree. This is essentially a binary tree but in two dimensions. The tree begins with a root node, which is a single quad/square. This quad can be split into four quadrants, each being a quarter of the original size. This split can be repeated as many times as necessary.

An LOD system can be created from this with a simple modification of rendering each quad with a consistent number of vertices. Smaller quads will inherently have more detail as the number of vertices per unit area is larger. To increase the detail level in each area a quad can be split up, with the opposite applying when merging meshes. This system is essentially a Multiresolution

Pyramid, where “with each level being subsampled to half the resolution of the level below it, which is similar to the concept of ‘mipmapping'” [2].

T-Junction Problem

The T-Junction is a common geometric issue in geometric graphics and occurs when a vertex is placed along the edge of another polygon. This is a problem in graphics as floating-point errors can render this vertex in a slightly different position, causing a miniscule but noticeable tear along this edge.

The issue of the T-Junction comes from a slightly different source. When two quads are touching each-other and have mismatched detail levels, their combined geometry causes these T-Junctions along the border. The granularity of the finer detailed mesh causes it to have a better approximation of the noise field compared to the less detailed mesh and as such cracks are formed along this boundary.

3. Implementation

3.1 Unity

I chose to use the Unity game engine for this project. It’s preferable to options such as Unreal Engine and OpenGL due to its ease of use and large market dominance. It also has useful features that other options don’t necessarily have, preventing me from scheduling time into implementing things that aren’t related to the goals of this project.

Unity uses a system of “GameObjects” and Components to be attached to them. This system works well with my project as each node of the quad-tree can be represented by a single GameObject. Each C# script can be modified to have outward-facing variables can be changed during run-time, which allows quick editing of parameters without the need to rebuild the project. Throughout this paper they are referred to as serialised variables.

3.2 Project Overview

To have the ability to fix the T-Junction problem, a base algorithm is designed to represent the underlying terrain height values with the three algorithms working to modify this base and fixing the crack problem in their own way. Each algorithm has a separate file to prevent interference with each other, although common functions such as the noise generator are combined for ease of use. I have structured the project like this as it becomes easier to develop, as well as having good modularity.

In terms of structure, each algorithm is assigned its own class which allows me to enable and disable features easily using Unity’s component system. The base algorithm is named “MeshGenBase” with subsequent algorithms being numbered using the name scheme of “MeshGenAlgX” where X is the algorithm number.

3.3 Implementing Quad-Tree Algorithm

3.3.1 Structure of Quad-Tree Algorithm

The quad-tree algorithm operates by recursively subdividing a single quad into four smaller segments. This structure can be visualised using a tree diagram, with a root node acting as the

parent for all other nodes. A given node is allowed either exactly zero or four children nodes. This structure is particularly useful for an LOD application as a node can be dynamically split to increase the quality of the mesh.

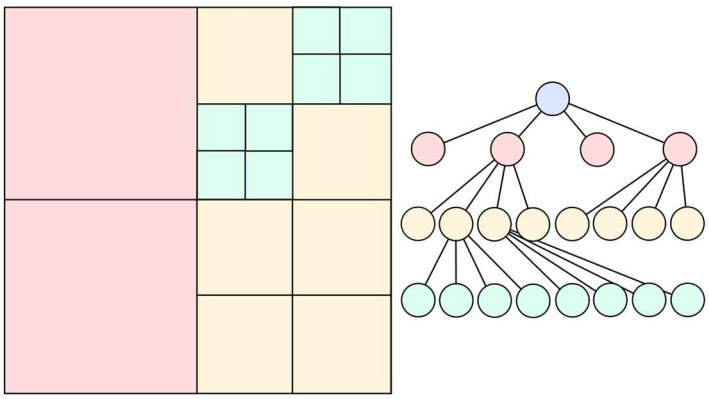

Figure 1: *Example of a Quad-Tree with accompanying data structure. Purple is the root node and has a detail level of zero.*

Each node is represented by a "MeshChunk" object, which stores information about the node such as the detail value and any connections to neighbours. The decision to use a class for this application allows for simple modification and good modularity when developing future algorithms. These nodes are numbered 0 – 3 depending on which quadrant they are in: South-West is 0, South-East is 1, North-West is 2 and North-East is 3. Numbering them unlocks several algorithms that rely on these indexes to work.

As I am using unity, I also need a way to represent each node within the game world. This is done by linking the MeshChunk object with its game world counterpart, allowing referencing at any point. The GameObject is responsible for rendering the mesh, with the anchor point being the south-west corner which means vertices in that corner are at index 0.

3.3.2 Mesh Chunk Object

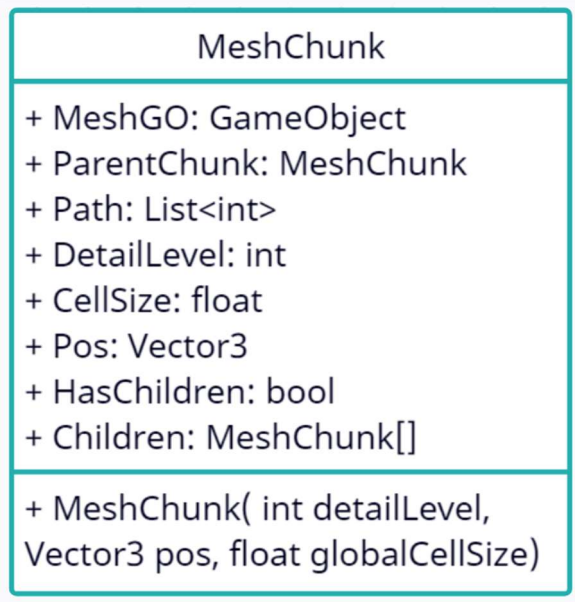

Figure 2: *Class diagram of MeshChunk class*

Above is the class diagram representing the structure of the MeshChunk class. As mentioned before, it stores necessary information about each node of the quad-tree.

- "MeshGameObject": a pointer to the GameObject within the game world responsible for storing and rendering the given node's mesh data. It allows for enabling, disabling, and refreshing the mesh by using Unity's built in methods.
- "ParentChunk" and "Children": assist with navigating the tree, containing references to their parent or children chunks accordingly. To help with this, the "HasChildren" Boolean variable is used to prevent needless computation.
- "Pos": stores the position of the node in world space, with the anchor being in the south-west corner.
- "DetailLevel" and "CellSize": related and store information relating to the current node's scale. CellSize is calculated from DetailLevel and represents the width of a cell in world space.
- "Path" stores a unique path to the current node. Starting with the root node, looping over the path array shows which child to index to get to the chosen node. Figure 12 demonstrates an example of the path variable.

3.3.3 Generating Height Field

To prevent the mesh from being a flat plane, a height field is required. This is done using Perlin Noise which is sampled using a vector acting as a position. The method of calculating the height involves creating a compounded value using multiple layers of noise with different scales and amplitudes to add additional detail. A single layer could not manage to do this.

3.3.4 Generating Mesh

To render a mesh, Unity requires two arrays that store the vertices and triangles of the mesh. The algorithm responsible for this is implemented in the "GenMesh" function.

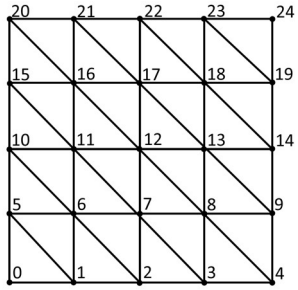

Figure 3: Example of mesh structure with a CellSize of four, not to be confused with a vertex width of five.

3.3.5 Generating Vertices

I have decided to create a square grid of vertices to represent each mesh. A nested for loop assigns each vertex a position based on the chunk's detail level and this position is used to sample the height field and given a height. As mentioned earlier, unity expects a single-dimensional array of position values, so this is inserted row by row. The serialised value "MeshCellCount" controls the width and height of the mesh, determining how many individual cells there are. Since this vertex array is an array, the language requires a pre-determined length which can be calculated using the cell count variable. The number of vertices along an edge will

always be one plus the cell count, and this new number is squared to find the total number of vertices. The resulting structure of the mesh is described in Figure 3.

3.3.6 Generating Triangles

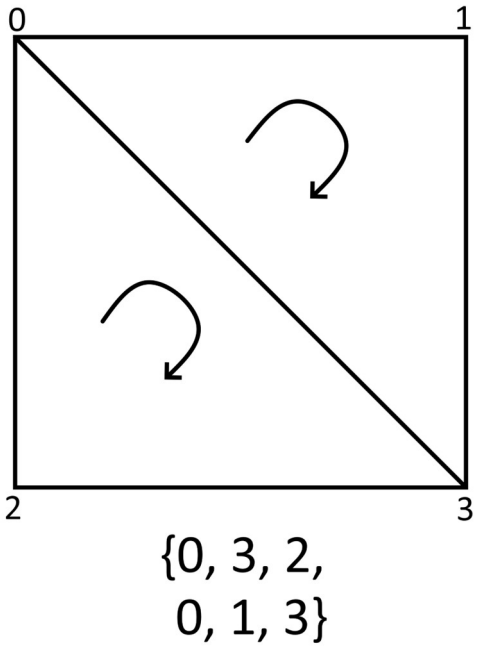

Figure 4: *Winding order of a given cell within a mesh. Accompanying array represents order of vertices added to triangle array.*

Similarly to vertices, a triangle array is also a single array but each value points to an index within the vertex array. A single triangle is represented by three contiguous values in this array. This system comes with a restriction, in that the vertices must be stored in a clockwise direction otherwise the triangle will face the opposite way. As demonstrated in Figure 4, each triangle is stored in this manner. As the nested loop passes over each cell, a running total is calculated to act as an offset within the array which is stored in "tileNum". A similar one is used to calculate the offset within the vertex array. The formulas for these variables can be seen in Figure 5. There are six vertices that must be stored for each cell, as such the offset is multiplied by six.

```
int[] triangles = new int[MeshCellCount * MeshCellCount * 6];
for (int y = 0; y < MeshCellCount; y++) {
    for (int x = 0; x < MeshCellCount; x++) {

        int tileNum = (MeshCellCount * y + x) * 6;
        int rootVert = (MeshCellCount + 1) * y + x;

        triangles[tileNum] = rootVert + MeshCellCount + 1;
        triangles[tileNum + 1] = rootVert + 1;
        triangles[tileNum + 2] = rootVert;

        triangles[tileNum + 3] = rootVert + MeshCellCount + 1;
        triangles[tileNum + 4] = rootVert + MeshCellCount + 2;
        triangles[tileNum + 5] = rootVert + 1;
    }
}
```

Figure 5: *Code snippet of triangle array implementation. Indices are calculated using the x and y values from the loop.*

There is an alternative method to storing the mesh called a Triangle Strip, however during my testing, I found problems with it adding unnecessary complexity. It functions by creating a new triangle from the previous two vertices, which is useful in specific use cases which unfortunately is not this one. The benefit of eliminating redundant vertices is countered by the additional complexity of generating the mesh and reduces flexibility.

3.3.7 Managing the Quad-Tree Structure

To increase and decrease detail dynamically at runtime, there must be a mechanism to add/remove nodes from the quad-tree which is done with the "SplitMesh" and "MergeMesh" functions inside the MeshGen class. These functions must manage these nodes effectively.

3.3.8 Splitting a Mesh

The algorithm must create four new objects for each quadrant and give a position based on its child index, relative to the current position of the parent. This was done using a dictionary, with the index as a key. The values in the dictionary are described in Figure 6.

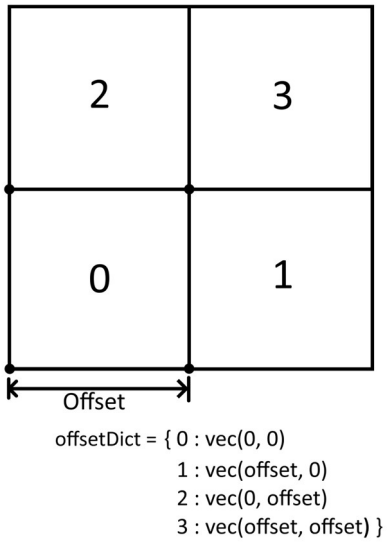

Figure 6: *Order of children of a mesh. Dictionary represents offset values when calculating new child positions. Bottom left is (0,0).*

To split a node, the following steps are taken:

1. Check if the input mesh already has children, if not then carry on
2. Assign the parent node's child array a new empty array of length four.
3. Calculate child offsets using current detail with the formula "2^(detail level + 1)"
4. Create a new child MeshChunk object for each quadrant and assign values such as position.
5. Disable the parent node, so it's mesh is not displayed anymore.

The use of a hierarchical structure within my Unity project has helped in managing these game objects effectively. As a child node is created, it is also assigned as a child of the parent node's game object. This groups related nodes together, allowing for easier management within the scene.

To prevent multiple layers of meshes stacking in the same area, the parent node is disabled to prevent interference. The height data is not deleted, as this would require recalculating if the child nodes ever needed to be merged.

3.3.9 Merging Meshes

Merging a mesh is simpler, with the method being to delete all children and enable the parent mesh. In practice this involves using the link between the MeshChunk object and the corresponding game object to remove the mesh from the game using Unity's Destroy() function.

3.3.10 Updating Mesh Dynamically

The algorithm must dynamically adjust the level of detail around the player, so the terrain stays a consistent quality. This is done during run-time using the "CheckChunkDistance" function. To determine the appropriate level of a given node, the centre point must be calculated using the formula "mp = pos + 1 / 2^(detail+1)" where pos is the south-west corner position of the node which is stored in the data structure.

A dictionary storing the upper end range of each detail level is utilised to simplify the modification of the program for later testing. Every frame, a depth-first algorithm compares the midpoint of any leaf node to its corresponding range within this dictionary, with any that do not align in this range being adjusted: those beyond the range are merged with their siblings to reduce detail, while those within a closer range are split into finer nodes to enhance detail. This method ensures that the terrain is consistent in visual quality, adapting as the player moves around the environment.

3.3.11 Analysing the T-Junction Problem

An inherent issue about this standard implementation of the Quad-Tree is the T-Junction problem, which is explained above. Many solutions implement vertices in some way to cover up this gap, either by moving them or creating new ones. The three solutions below are no exception to that, which have been proposed to focus on concealing this gap from the player's perspective. The idea of using certain shader techniques to cover these gaps has been thought about but the implementation is outside the scope of this project, which is mentioned in the limitations section.

3.4 Algorithm One: Overlap Method

The first solution involves creating an additional layer of cells surrounding the original grid for each mesh, which serves to obscure the seam with an overlap of the surrounding nodes. Although this approach doesn't eliminate them all, it conceals the vast majority from most camera angles while having a minimal performance impact.

The implementation of this algorithm required a series of adjustments to the original algorithm. Firstly, the vertex and triangle arrays were modified to allow for the increased number of cells. This was calculated using a new "width" value which is the new combined width of the cell, as the overlap value increases total width. The positions of each vertex also had an issue where it would not extend towards the negative side of each axis, preventing an overlap with the south and west neighbours, but this was fixed with an adjustment incorporating the overlap size into the position calculation. The structure of the vertex and triangle arrays are split in half, with the first half being the original grid and the second half being the new addition of geometry. The split

allows for easier manipulation of the data structure, as opposed to some mix of original and new geometry which overcomplicates the order.

There is a specific scenario between two neighbouring nodes of equal detail level, where the overlaps match the neighbour's vertex nodes exactly. This is a waste of resources as they are being drawn multiple times. A possible improvement to the algorithm can fix this issue, which is only creating an overlap with neighbouring nodes of differing detail levels. A screenshot of this implementation can be seen in the testing section.

3.5 Algorithm Two: Skirt Method

The basic idea of this method is the same; add vertices to hide terrain cracks from the player, although it does it in a different way. The method comes from the paper by Suárez et. al. [4] as they had a similar crack issue with their planetary LOD implementation. The previous method extended vertices away from the centre of the mesh, while this one creates a loop below the edge of the mesh. This ring of vertices is positioned a height "h" below the edge of the mesh grid, helps cover gaps effectively as this "h" value can be fine-tuned to find a compromise between length and render time.

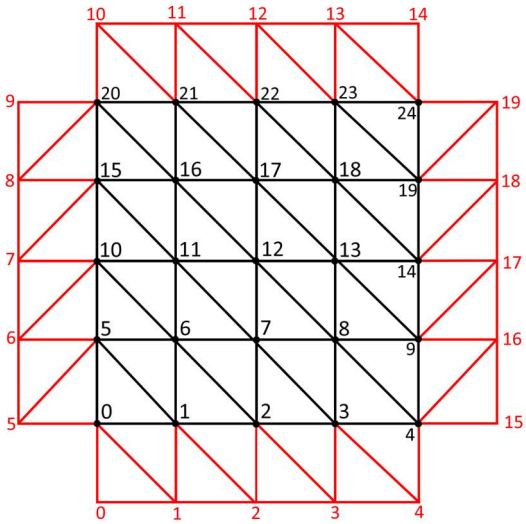

Figure 7: *Mesh structure of a mesh with a skirt. Red values are added after the black values within the vertex array therefore have an offset of 25 in the example.*

Modifying the structure of the mesh was necessary to accommodate this addition. Figure 7 visually describes the new structure, with a new ring of vertices created and attached to the edge of the original mesh.

Suárez et. al. has implemented textures in their solution, with "lower vertices of this strip will [using] the same texture coordinates of their upper neighbors in the tiles edge, matching their color." [4]. They have also analysed the memory requirement of a single mesh stored in both RAM and GPU.

Within the implementation there is a system to add vertices into the array in the right order. In Figure 7, the vertices coloured in red are added to the list after the black vertices. This is done to reduce code complexity, as it is an intuitive way to order them. Another way would be to order them as the nested loop reaches them, but this is confusing when finding the vertex indexes for triangles in the next step. The red vertices are also added row by row. I have chosen to use this order as opposed to other methods such as a clockwise order due to the use of a nested loop

within the code. making it simpler to work with in the next step regarding the triangle array. This nested loop generates vertices starting at the bottom row and going across.

Changes were also made to the lengths of the vertex and triangle arrays to include this addition. The vertex array follows the formula “(w+1)^2 + 4w”, which comes from the area of the original square with the addition of the skirt of four times the cell count, where “w” is the number of cells, not to be confused with the number of vertices. The triangle array is calculated by the number of cells multiplied by the 6 vertices used per cell which comes out to “6(w^2 + 4w)”.

When creating these vertices, a series of if/else statements were employed to ensure that they were only generated while on the edge of the mesh. The order of these statements is important as the indexes of these vertices are calculated depending on which edge they are in. The order is as follows: Bottom edge, top edge, left edge, right edge. These correspond to the zero and max values of each axis and are necessary so that any potential duplicated vertices are ignored. The indices of these vertices must also be calculated. This is done by first calculating an offset value which uses the formula “(w+1)^2” and is the number of vertices in the original grid. It gives the ability to index vertices in the second half without having filled the first half of the list first. Depending on the direction of the side, a different formula is used to calculate the index of a vertex.

To include the skirt in the mesh geometry, a series of if statements similarly to the vertex array were used. They calculate whether a given vertex in the nested loop is along an edge and more specifically which edge. Once an edge vertex is identified, a set of six indices for the triangle array and corresponding indices for the vertex array are calculated using a modified version of the base algorithm formula, derived from the example structure in Figure X. These formulas allow for the mesh skirt to be integrated seamlessly.

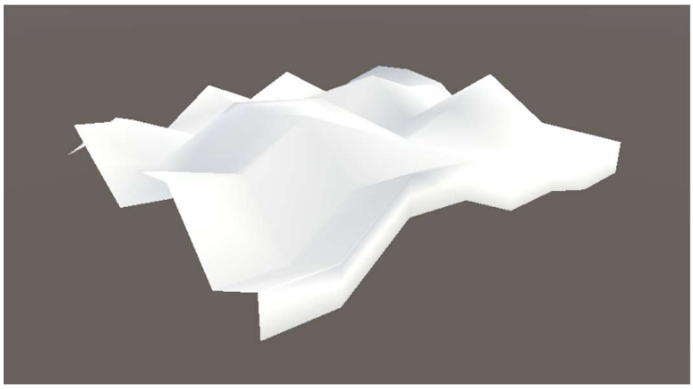

Figure 8: *Screenshot of skirt generated on one side of a mesh. Lightning is interpolated along the edge which is not ideal.*

Unfortunately, there was an issue identified while testing this method concerning normal values along the edge of the meshes. Normal values represent a vector pointing in some direction for each vertex of a mesh, and these normal vectors are interpolated within the fragment shader to find the value it should be effectively smoothing out a triangle. The vertices along an edge of a connected mesh can only store one normal value which will be rendered with a smoothed edge as seen In Figure 8. The smoothing of the edges is undesirable and leads to a seam issue along adjacent node borders.

3.6 Algorithm Three: Modified Skirt Method

To address the problem of the previous algorithm, a different approach must be taken when designing the mesh structure. The issue comes from the skirt being attached to the same vertices as the edge of the original grid, so the natural solution is to clone these edge vertices and give them different normal values corresponding to side that they will be attached to. The new geometry of the mesh is illustrated in Figure 9, but essentially involves splitting the four edges of the skirt into their own strips of cells.

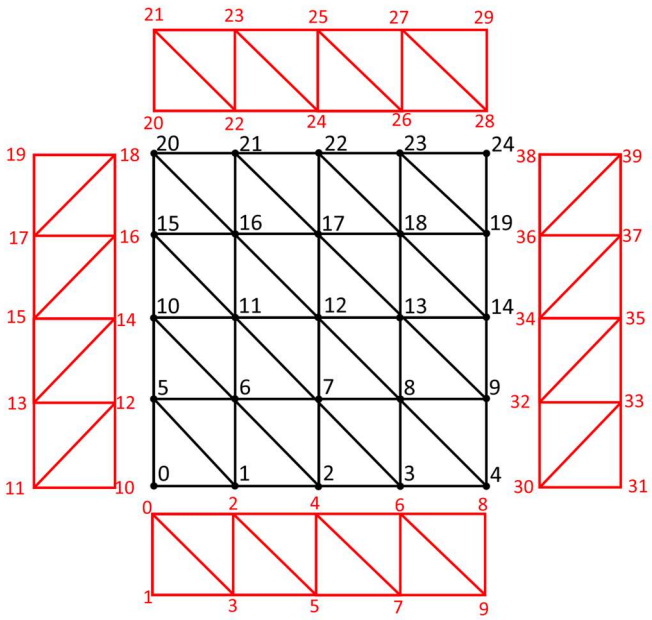

Figure 9: *New mesh geometry of skirt mesh, with separated skirt strips around the edge.*

This modification also affects the vertex and triangle array lengths, as the geometry modifies the number of cells and triangles overall. The vertex array length now follows the formula “(w+1)^2 + 8w”, an increase of 4w vertices from the duplication of the edge ones. The triangle array length formula is unchanged due to there being no cells removed or added.

The order of these strips are as follows: South, West, North, East. This order is important as it is used to further subdivide the red section of the vertex array by which side they sit on. South has an offset of 0 and East has an offset of “d(18 * w)”, where “d” is the index of the direction, i.e. West has a “d” value of 1. The offset is then used to displace the index by that value along the vertex array.

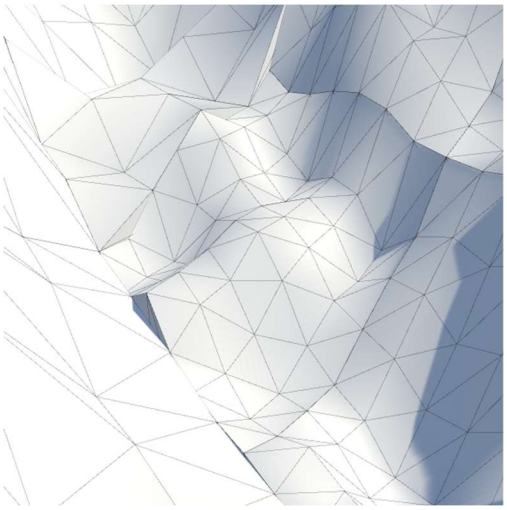

Figure 10: *Screenshot of new skirt algorithm. Dark edge near the left is an example of this skirt and the lighting issues it causes.*

3.7 Algorithm Four: Neighbour Height Matching

The previous algorithms involved generating additional vertices to conceal cracks in the terrain. However, these methods increased the computational load of the rendering process from the increase in vertices. In contrast, algorithm four focuses on adjusting the vertical position of vertices to align with the heights of neighbouring vertices.

A simplified breakdown of this algorithm is as follows:

1. Identify Neighbouring Nodes: The first step is to determine the neighbouring nodes for each cardinal direction surrounding the main node. These neighbours only need to be identified down to the same detail level as the main node as the method is designed to only act on the smaller of the two nodes. Any child nodes of a neighbour node of the same size can be ignored, simplifying the subsequent steps.
2. Compare Detail Levels: The detail level values of these neighbouring nodes are compared to the main node. Levels lower than the main node have been discarded in the previous step. This comparison is critical for determining whether adjustments are necessary for a given edge as the detail levels can be equal.
3. Adjust Vertex Heights: If it is determined that the main node is smaller than a neighbouring node, the bordering vertices of the main node require an adjustment of vertex height to match those of the neighbour. This is carried out using an interpolation algorithm which essentially calculates, for each main node vertex, an interpolated height of the neighbour's next and previous vertex heights.

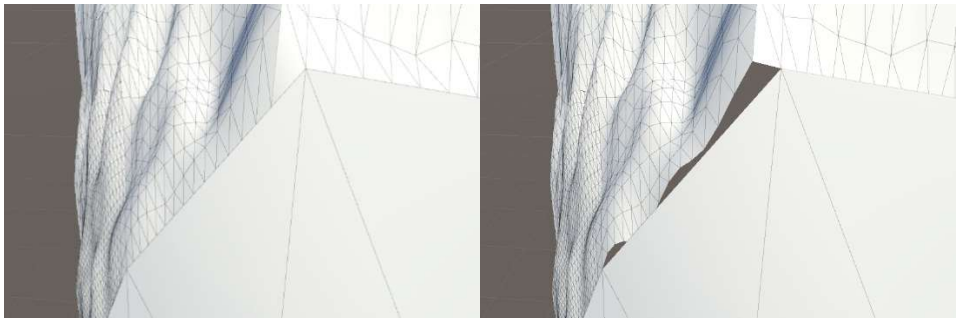

Figure 11: *After and Before of Offset algorithm. Vertices are moved towards the neighbour edge.*

The steps outlined above are abstracting many key details that needed to be figured out, which are all described in the upcoming sections, but eventually end up creating a cohesive solution to the T-Junction problem. The difference this algorithm makes can be seen in Figure 11, with the vertex height matching causing the more detailed node to attach its nodes to the neighbour with less vertices.

3.7.1 Finding Neighbour Nodes

The process of efficiently locating neighbouring nodes within the quad-tree structure for mesh adjustments initially presented several approaches, each with their own advantages and complexities.

The initial idea was to employ a recursive algorithm to traverse the tree until a desired neighbour was located. This method would travel up towards the root node, taking note of any neighbouring nodes along the way. While this approach offers the potential to find neighbours very quickly, it proved to be overly complicated to implement due to the numerous edge cases it introduced. This complexity made the method less appealing for this application, although can work in scenarios where corner neighbours are necessary.

A second method is to calculate the neighbours as the tree is expanded/shrunk, storing this value in each “MeshChunk” object. This exploits the fact that the tree is initialised as a single mesh called the root chunk and has no neighbours. Every time a mesh is split, the neighbours of that mesh can be passed down to the outside border of its child nodes. This method gives a trade-off between run-time performance and pre-processing, preventing the need for expensive tree traversal.

Ultimately, the chosen method was to calculate the paths of neighbouring nodes during run-time. This third option optimises a good balance between implementation complexity and performance. This method works by exploiting a feature of Quad-Trees where a given node’s path can be split into an X and Y coordinate and modified from there to find any cell, relative to the main node.

3.7.2 Binary Coordinate System

As explained briefly previously, each node within the tree has a path that essentially acts as directions to get to it while traversing the quad-tree. Each node also has a unique path, which can be seen in Figure 12, which allows the exploitation of an inherent coordinate system within a quad-tree representation.

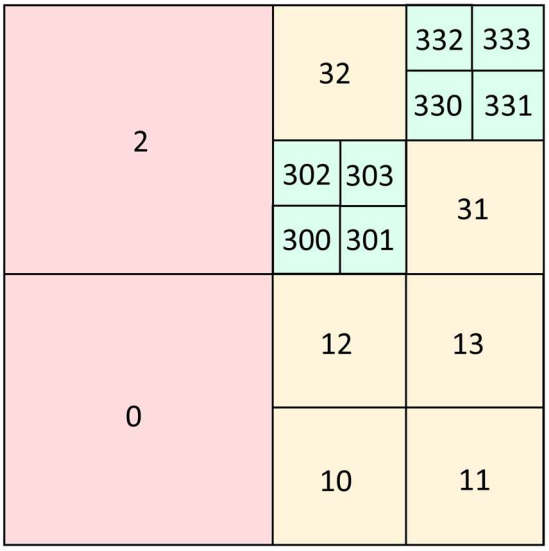

Figure 12: *Mesh structure with labelled paths of each quad. Length of path dictates detail level.*

Studying the figure, a pattern is uncovered where the path number of a given node is a combination of its X and Y coordinates in binary using the table in Figure X. For example, the path number “203” results in an X coordinate of “110” and a Y coordinate of “101”. If treating this binary string as an integer, the coordinate of node “203” is (6, 5).

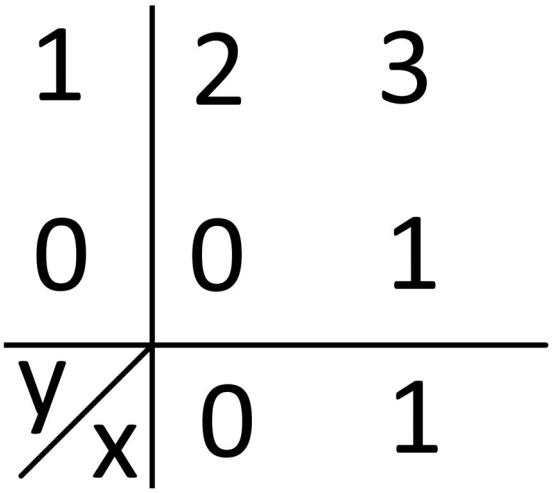

Figure 13: *Conversion table between binary and path numbers.*

This conversion to a coordinate system allows a different form of traversal of the grid, which conveniently makes finding neighbours much simpler. Modifying the resulting coordinate can find other nodes relative to the main node, as an example, adding (0,1) to the “203” example results in (6,6), which is the northern neighbour of the main node.

3.7.3 Implementation of Coordinate Algorithm

In terms of the implementation, it begins with a loop of the path number to iterate over each value, as well as two C# lists created to store the output coordinates in binary. For each value in the path the X coordinate is calculated with the formula “value % 2”, as only the least significant bit is responsible for horizontal traversal. The Y coordinate is calculated similarly, but only the second bit is responsible for vertical traversal, so the formula is “(value >> 1) % 2”. This formula involves the right bitwise shift “>>” which shifts the whole binary string rightwards, discarding the least significant bit.

There is a need to use arithmetic on these binary strings, but there is no built-in way to manipulate lists of binary digits and as such I would have to create custom functions for addition and subtraction on this list. Because of this, two functions “ListToInt” and “IntToList” were also created to facilitate the conversion between a binary list and an integer to allow for this arithmetic using built in C# syntax.

The situation that a node is along the edge of the tree can cause problems. Without any error catching, the neighbour algorithm will wrap around and assume that a node on the other side of the map is a neighbour. If a digit’s X coordinate is “111” and another digit is added, “1000” is the output. The program assumes a detail level of 2 so only takes the first 3 digits “...000”. This issue is fixed with a simple if statement checking if the digit is above the value of (2^detailLevel) - 1.

3.7.4 Converting Path Number to MeshChunk Object

The method to traverse the tree using a path list was described earlier, but essentially goes as follows: The path list is looped over, with each value representing the index of the child needed to get to the target node. A MeshChunk variable is used to store the furthest reached node, as the coordinate algorithm doesn’t know if a given node exists at that detail level. If at any point a node doesn’t have child nodes, the loop is broken and that furthest reached node is the output neighbour for that direction.

3.7.5 Interpolating Height Value

In the process of manipulating the height of a given vertex, there is a need to assess whether this vertex accomplishes two things: it must be on the border of the mesh, and it must be along a border with a neighbour of higher detail. An if/else block is utilised to determine the first objective due to the issue of corner vertices being along two edges at once because a simple if block would not be able to deal with this. It checks if a given neighbour exists, and then if that detail level is larger than the main node for each cardinal neighbour.

There is a nested loop within the base algorithm that iterates over each vertex and generates its corresponding height. This needs to be modified to allow for height adjustment. Another set of if/else statements are used within this nested loop to analyse each vertex to analyse whether the vertex is along the edge. If a vertex passes this test, it can then be interpolated.

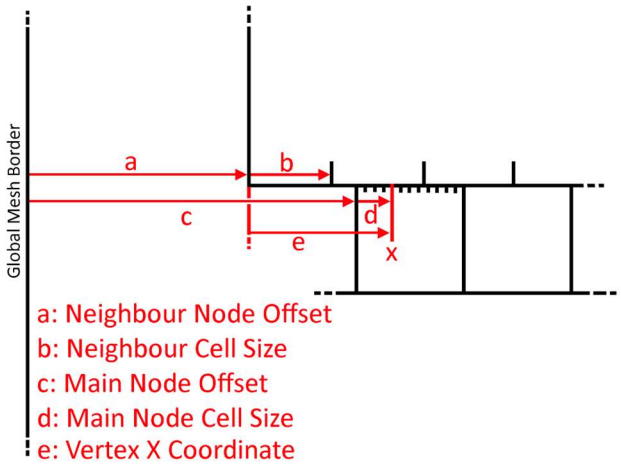

Figure 14: *Visualisation of interpolation algorithm. Top node is neighbour node.*

Figure 14 demonstrates the basics of this algorithm, but essentially the idea is as follows:

1. Convert the current index of the vertex into a position in the “mesh space” of the neighbour node.
2. Use the modulus operator on the mesh space position using the neighbour’s cell size as a dividend.
3. The output of this modulus operation represents the amount of the neighbour’s cells that can fit from the edge to that vertex.
4. Use this output to calculate the position of the previous and next neighbour vertex positions.
5. Interpolate between the heights of these vertices to find a height in the middle.

To help with explanation, everything talked about in this section will be focused primarily on the northern border of the main node although it is important to note that the principles apply equally to all other orthogonal borders.

3.7.6 Neighbour Mesh Space

In the mesh structure, each vertex is defined by an x and a y integer index, which pinpoint its position within the mesh. The northern edge of the main node and the southern edge of the neighbouring node share the same y value, which can be considered as a single line within the world. Along this line, each vertex along this edge from both the main node and neighbouring node can be resided.

The need for this conversion comes from a method requiring vertex positions to calculate a height value. Using the information about both nodes, such as the distances between vertices on both sides of this line, the positions of the previous and next vertex along this line can be calculated relative to a vertex within the main node.

3.7.7 Interpolation Algorithm Implementation

Within the structure of the GenMesh function, a nested loop iterates over each vertex and generates its height and position based on its x and y index within the loop. For each of these vertices that are along the north edge of the mesh, the x coordinate is calculated relative to the left most edge of the global mesh. This is done using the formula “x * ChunkCellSize + ChunkXPosition”, where X is the vertex’s x index within the nested loop. ChunkCellSize is the width of a cell within the chunk, and ChunkXPosition is position of the current chunk within the world from the left corner of the chunk.

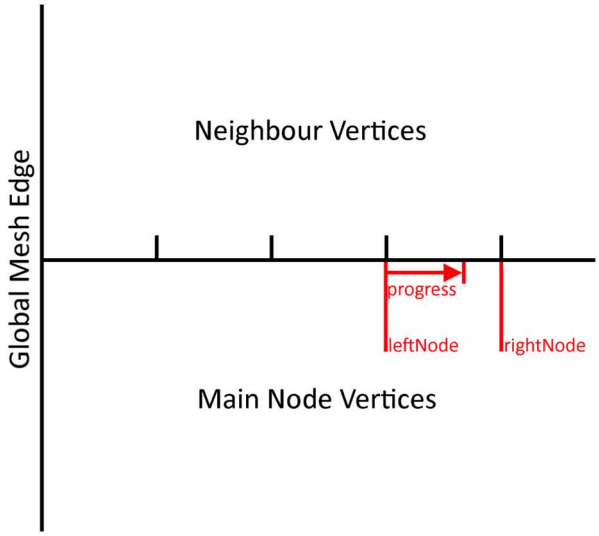

Figure 15: *Calculation of left and right nodes, with progress between them. The example will have a leftNode value of 3 and rightNode of 4. Progress will be around 0.7.*

This x coordinate is crucial in determining the heights of the left and right vertices relative to the current vertex. It is used to find out how many cells from the neighbour’s node can fit between this x coordinate and the left side of the neighbour’s mesh using integer division, with the neighbour’s cell size used as a divisor. The output of this is a single integer named “leftNode” that represents the index of the left vertex, which can be multiplied by the neighbour’s cell size again and plugged into the height generation function to get the height of this vertex. “leftNode” can be incremented by one, with the same steps repeated, to get the height of the right node as this offsets the position by one cell width.

To get a height between these two heights, linear interpolation must be performed using a blend factor. This blend factor is represented as a value from 0-1 and can be thought of as the proportional distance a vertex on the main node has travelled between the left and right vertices of the neighbour node. It is calculated using the “leftNode” variable from the previous calculation using the formula “chunkProgress – leftNode”, where chunkProgress is a standard division of the x coordinate instead of an integer division that was performed previously. The introduction of this new “chunkProgress” variable was needed to prevent a floating-point issue where the algorithm would assume it was on the other side of the boundary of a vertex in extreme cases. This formula allows this floating-point error to be insignificant as opposed to when using a modulus operation.

3.7.8 Refreshing Node

The algorithm works perfectly if all neighbouring nodes have also been unadjusted, but this is rarely the case. In reality, a neighbouring node has the possibility to have its edges adjusted to the current node before the current node has been adjusted itself. Once it receives the adjustment, the neighbouring node will have been adjusted to a wrong detail level and as such create more cracks.

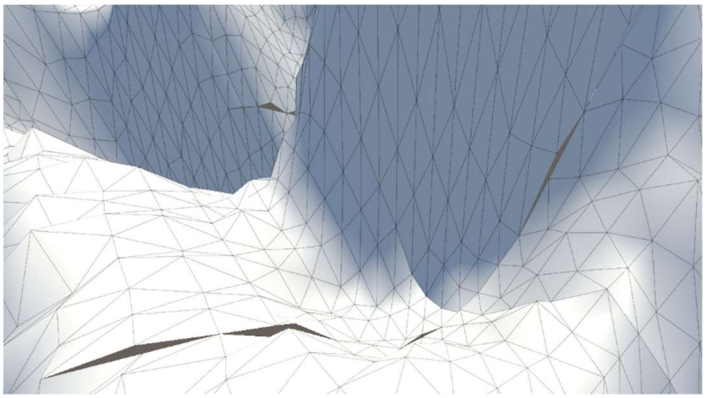

Figure 16: *Screenshot of meshes with same detail level having cracks due to old mesh vertex adjustments.*

The solution is to recalculate the adjustment of all neighbouring nodes when any given mesh is combined or split. This ensures that each neighbouring node correctly aligns with the new geometry of the current cell.

3.7.9 Calculating Neighbours Along a Border

The nodes along a border must be calculated to refresh their meshes. This is done by using the neighbour of the same detail as a stepping off point, as this is already known from previously explained functions. This neighbour, which is referred to as the root neighbour from now on, has the possibility to not be a leaf node. This wasn't an issue for the interpolation algorithm as it was always performed on the smaller node. Due to the design of the project, only leaf nodes are displayed within the game, so it is these that must be identified.

The procedure employs two lists to manage nodes. The first list, named "output" is initialised as empty and is designated to store all chunks that border the edge. The second list "listToCheck", holds all chunks that are situated on the edge but have not yet been confirmed as leaf nodes and is initialised with the root neighbour.

Every chunk within "listToCheck" is analysed to see whether it is a leaf node. If this chunk qualifies as a leaf node, it is added to the output. If not, all children of this node that border the relevant edge are added to "listToCheck". A while loop is used to repeat this algorithm until "listToCheck" is empty.

4. Testing

4.1 Results Setup

My project has the overall aim of analysing multiple T-Junction algorithms to find the best performant one, as such a comprehensive testing system has been developed in order to test each for efficiency and scalability in different scenarios.

Operating System	Windows 10 Home
Device	IdeaPad 5 Pro 16" (Laptop)
GPU	NVIDIA GeForce GTX 1650
CPU	AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz
Memory	16.0 GB (13.9 GB usable)

Figure 17: Spec of PC used to run tests.

Testing was done on a Windows 10 Laptop, with the parts described in figure 17 above. To ensure that each algorithm gets a fair representation, many precautions were taken to reduce any variance between tests. When running tests that require processing time, careful attention was given to having consistent background system cost. All background tasks that were not necessary were closed, as well as maximising the game view window to remove the scene view within Unity. The charger is also plugged in to remove the issue of battery saver fluctuating CPU speed.

4.2 Measuring Mesh Generation Time

Each of the four algorithms have their modifications within the SplitMesh and MergeMesh functions, with SplitMesh calling the GenMesh function. As these two functions are opposites of each other, a test was created which repeatedly splits a mesh then merges it back into one an X number of times. Once this loop completes, an average time is calculated for how long each loop takes and output in milliseconds.

```
public void TestGenMesh() {
    float startTime = Time.realtimeSinceStartup;
    SplitMesh(RootChunk);
    SplitMesh(RootChunk.Children[1]);
    SplitMesh(RootChunk.Children[1].Children[2]);
    SplitMesh(RootChunk.Children[1].Children[1]);

    for (int i = 0; i < 10000; i++) {
        SplitMesh(RootChunk.Children[1].Children[1].Children[0]);
        MergeMesh(RootChunk.Children[1].Children[1].Children[0]);
    }

    float timeTaken = Time.realtimeSinceStartup - startTime;
    print( message: $"{timeTaken/10000 * 1000}ms");

}
```

Figure 18: *Code snippet of mesh generation test. Initial SplitMesh calls are responsible for setting up neighbour nodes to recalculate.*

The loop can be modified to run any number of times, removing any variance within a single call. I have chosen a value of ten thousand repetitions as I believe this is a good trade-off. This test was also run using a different value for MeshCellSize, which controls the number of vertices within each mesh. Specifically, the values were 1, 4, 16, 32 and 64 cells wide as this gives a wide range of values which can be interpolated to find any middle points.

4.3 Measuring Run-Time Performance

To accurately assess performance during run-time, it is important to evaluate how effectively the terrain updates as the player navigates the environment. To simulate this, an object is moved around the scene in a circular motion which allows the mesh to update the mesh in real-time, with faster updates correlating with increased player speed.

```
void Update(){


    if (movePlayer) {
        float time = Time.time % playerSpeed / playerSpeed *2 * Mathf.PI;
        float posX = (Mathf.Cos(time) + 1) * 2048;
        float posY = (Mathf.Sin(time) + 1) * 2048;
        player.transform.position = new Vector3(posX, y:0, z:posY);
    }
```

Figure 19: *Code snippet of player movement simulation. Cos and Sin functions are used to move an object in a circular motion around the perimeter.*

To accurately monitor the frame rate, Unity's "unscaledDeltaTime" variable is used. It tracks the time elapsed since the last rendered frame, providing a good method to track the number of frames a second the build can output. The final FPS value has an inherent problem with it being very jittery. If one frame is rendered much quicker by accident, then it can affect the test results negatively. To fix this, the average FPS is calculated from the past X seconds, where X can be modified. The method can be seen in Figure 20, but essentially it uses a fixed-length array to track all FPS values within these past X seconds. A pointer is used as an index to replace the oldest value with the new incoming information.

```
private float maxfps;
private float fps;
private float updateTimer;
[SerializeField] private float updateInterval;

private float[] fpsHist;
private int histIndex;
private int maxHistIndex;
[SerializeField] private float avgTime;

void Update(){

    updateTimer -= Time.deltaTime;
    if (updateTimer <= 0) {
        fps = 1f / Time.unscaledDeltaTime;
        if (fps > maxfps) maxfps = fps;

        updateTimer = updateInterval;
        fpsHist[histIndex] = fps;

        histIndex++;
        if (histIndex > maxHistIndex) {
            histIndex = 0;
        }

        float total = 0;
        for (int i = 0; i < fpsHist.Length; i++) {
            total += fpsHist[i];
        }

        float avgfps = total / (maxHistIndex + 1);

        fpstext.text = $"FPS: {fps}, max: {maxfps}, avg: {avgfps}";
    }
}
```

Figure 20: *Code snippet of FPS and avgFPS calculations. fpsHist stores all the past framerate values to calculate an average from.*

A few different test cases were created to have a wide range of values for assessment. Mesh cell sizes of 1, 16 and 64 were used and each run three times to find an average FPS to hopefully remove any anomalous data. This test was repeated once more using a flat plane as a noise field, as opposed to the bumpy Perlin noise field.

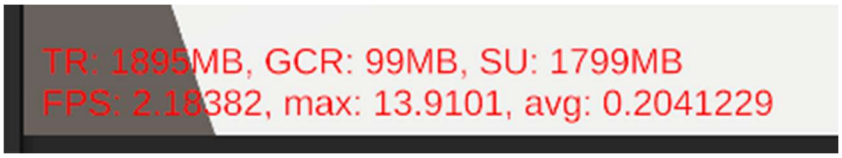

Figure 21: *Screenshot of FPS and memory usage values displayed on screen.*

4.4 Measuring Memory Consumption

The project employs Unity's ProfileRecorder class to capture key memory values, specifically used memory and total reserved memory. The measurement of used memory is important to find which algorithm is best at managing resources in this department. Mesh Cell sizes of 1, 16 and 64 are used to allow for the collection of data across a wide spectrum.

The test is designed to be simple, simply modifying variables to the required test and noting the value within the memory counter on screen. The base value of no algorithm is also tested to allow for the exact measurement of algorithm memory cost.

4.5 Test Results

4.5.1 Mesh Generation Time

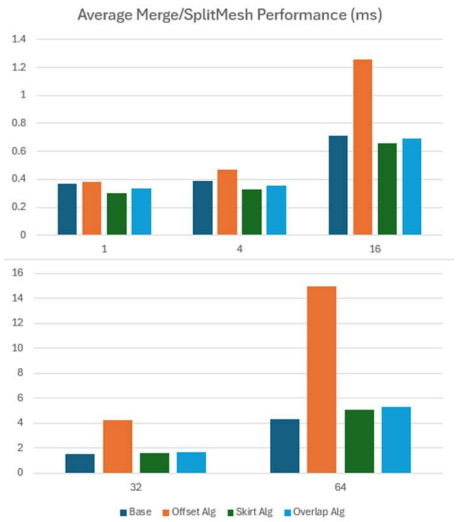

Figure 22: *Bar chart results of mesh generation time test. This is split into two sections as the 64 CellSize test was overwhelmingly large.*

The bar graph depicts the average performance of four different algorithms in terms of time taken to perform a Split/Merge loop and is measured in milliseconds. The tests are performed on mesh cell sizes of 1, 4, 16, 32 and 64, with the latter two being split into the lower section of the figure due to a higher range of values. All four algorithms begin with similar performance, but the offset algorithm becomes less efficient with higher mesh cell counts, while the others stay within the same range of performance.

4.5.2 Run-Time Performance

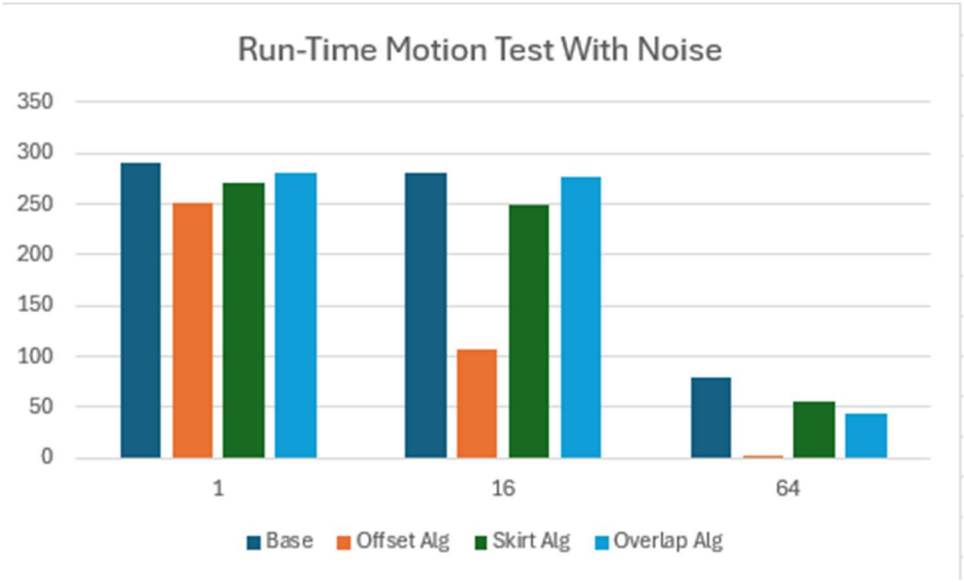

Figure 23: *Bar chart of results from motion test with large noise values with test cases of 1, 16 and 64 cells.*

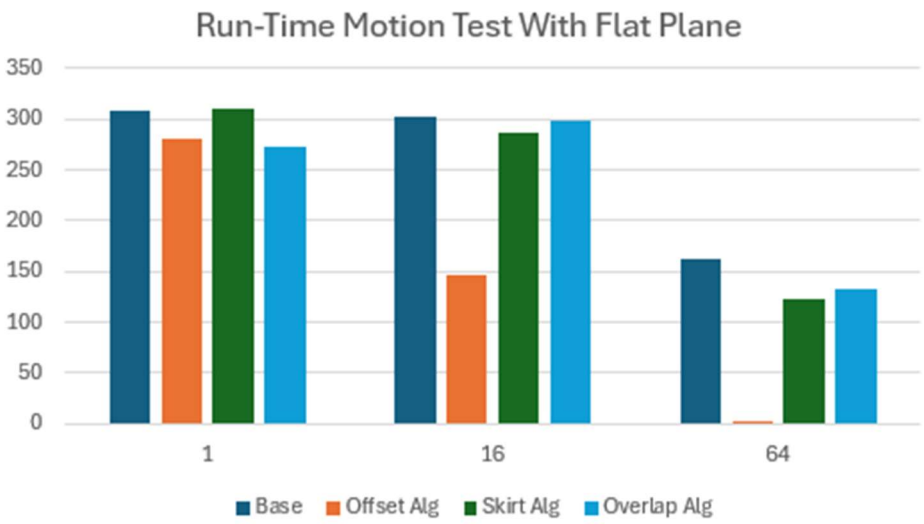

Figure 24: *Bar chart of results from motion test with flat plane with test cases of 1, 16 and 64 cells.*

The two bar graphs above display the results of simulating player movement during run-time using the two situations of a flat noise field and a highly irregular noise field. The tests are performed across the three cell sizes of 1, 16 and 64. The results show similar findings to the previous test, where the offset algorithm is by far the least performant while the others stay within a small range of each other.

4.5.3 Memory Consumption

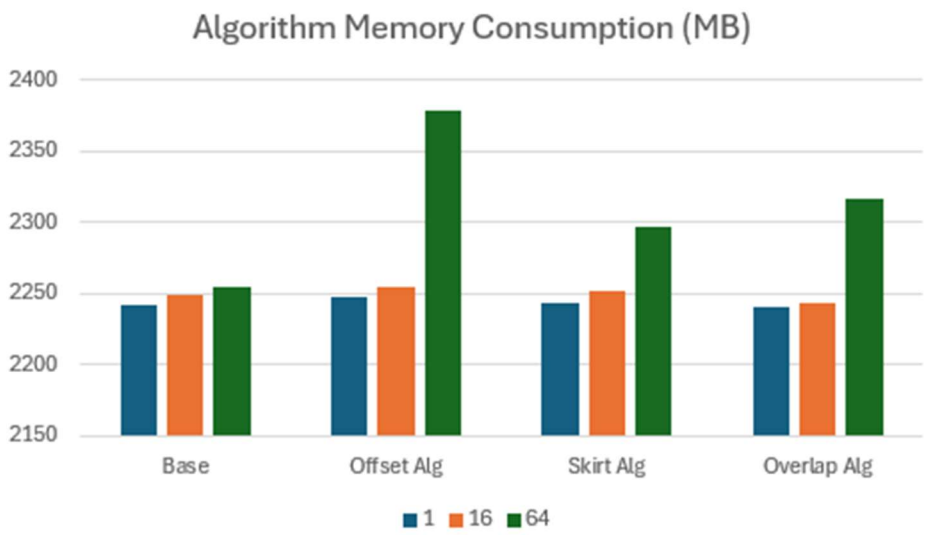

Figure 25: *Bar chart of results of memory consumption test, with test cases of 1, 16 and 64 cells. Results are in MB.*

The bar graph describes the memory consumption in megabytes for each algorithm across the same cell sizes of 1, 16 and 64. All algorithms have a large increase in memory consumption at a cell size of 64, with similar consumption at lower cell sizes.

5. Evaluation

5.1 Offset Algorithm Evaluation

The offset algorithm (Algorithm Four) stands out as the most visually appealing approach because it fully covers the gap caused by the T-Junction problem. However, it does come with many issues. One of these comes from when three meshes with unique detail levels come together on a corner causing a gap due to interpolating the wrong mesh. This can be fixed by creating an edge case within the code that interpolates using a 2D method instead of the single dimension of the current method. Another issue is the regeneration of all neighbouring meshes caused by the splitting of a single mesh. This is especially a problem with higher mesh cell counts as there are many more vertices to generate for every mesh. A good solution to this problem is to only recalculate the vertices along the edge of the mesh which reduces the recalculation from O(n^2) to O(n) complexity.

In terms of performance, this O(n^2) complexity is made visible by the 64 cell tests. When looking at memory consumption, all algorithms have increased usage at this higher cell count, but this algorithm specifically has a large increase. It is a similar story for the performance tests, with the offset algorithm being much slower than the others by a large margin. In the case of 64 cells in the Motion test, the algorithm seems to suffer extremely from the Big O complexity getting an average framerate of 1.40fps and 2.09fps with and without noise respectively.

The implementation of this algorithm took some shortcuts, but there are a few things that can be added to make it more performant. Firstly, the method of calculating left and right vertex heights in the interpolation algorithm can benefit from caching values to prevent multiple calculations of the same node. Another improvement comes from realising that the Split/MergeMesh functions can be used multiple times in a single frame on a single Mesh. This introduces unnecessary code repetition. A fix to this comes from the implementation of a buffer of jobs that can be processed to cancel any repetitive function calls out.

5.2 Skirt Algorithm Evaluation

The skirt algorithm is effective at covering gaps and does it in a way that is consistent. However, at further distances this implementation breaks down due to the constant skirt length within the code. Cracks in the terrain are larger at further distances so a constant length would eventually not fully cover this gap. This can be fixed by using a value dependant on the distance from the player, which would also create consistent skirt sizes on the screen.

There is also an issue with Algorithm Three with lighting caused by the hard edge of the skirt and the main mesh leading to a very noticeable visual discrepancy.

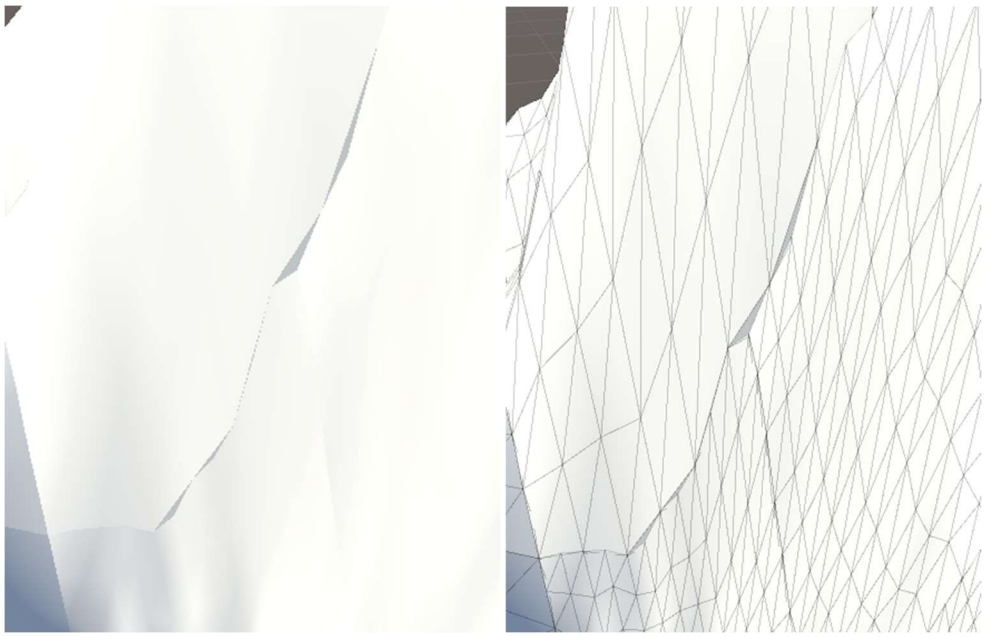

Figure 26: *Screenshot of skirt algorithm with polygon edges rendered and not rendered. The border of two meshes can be clearly seen due to the hard border of skirt*

Algorithm Two can be a good solution to this issue, but I believe it is an inherent problem with the skirt algorithm as the skirt is always perpendicular to the terrain that the mesh is displaying and thus causing this lighting issue. A possible fix can be to exploit normal values to point the same direction as the mesh vertices above it.

Performance-wise, it is the most efficient out of any of the modifications as it has the least Memory consumption and processing time. This allows it to be the most scalable for larger meshes representing massive landscapes compared to other algorithms.

5.3 Overlap Algorithm Evaluation

The overlap algorithm essentially blends two neighbouring meshes, offering a seamless boundary that covers most cracks caused by T-Junctions, and in the case that a crack is visible then the overlap size variable can be increased from the default of 1 to cover it up. There are also no obvious lighting issues caused by this overlap method, although at low mesh cell counts there are hard angles in the terrain that cause harsh lighting changes.

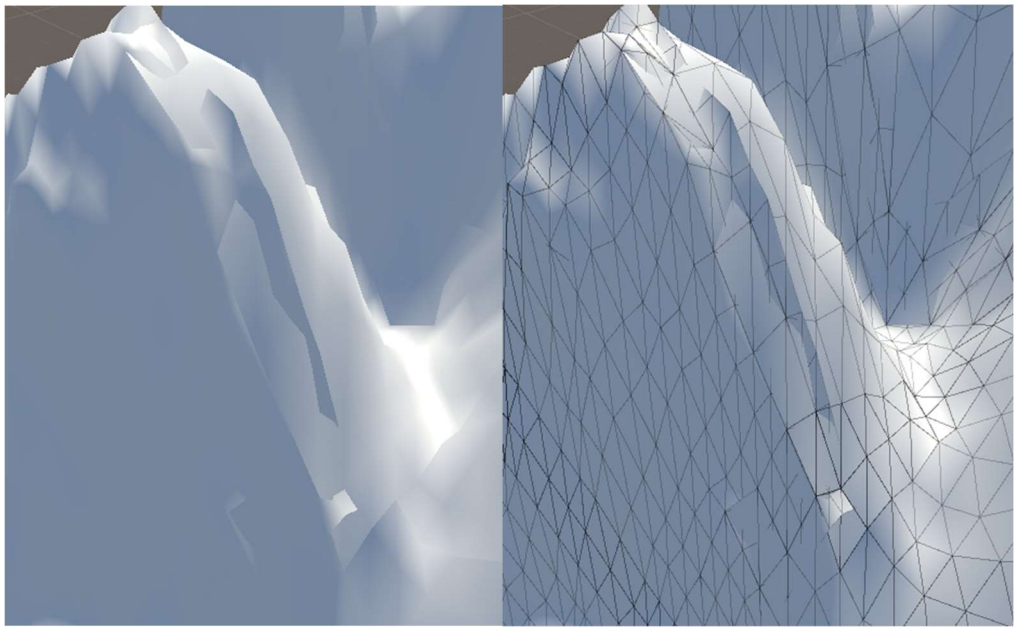

Figure 27: *Screenshot of overlap function with polygon edges turned off and on. Polygon overlap can be seen along this edge, causing hard boundaries in terms of lighting.*

The algorithm is also not as efficient as it can possibly be. One issue is that meshes are overlapped without considering their respective detail levels. In the situation that two meshes of the same detail level are bordering each other, there are no cracks, so it is a waste of resources to duplicate the vertices along this boundary. An improvement to the algorithm can take this into account and only overlap on boundaries with different detail levels.

5.4 Ease of Implementation

To accurately judge whether an algorithm is superior to another, I believe analysing the difficulty of implementation is important. In game development, a “good-enough” algorithm is often preferred over a perfect one that takes twice as long to implement, especially in a field where time constraints are common. Overcomplicating code can also lead to increased maintenance difficulty.

The offset algorithm was by far the most complex to implement. It required the understanding of several intricate algorithms for the seemingly small task of finding neighbouring vertices to interpolate. The process required a deeper understanding, making the implementation time-consuming and very prone to errors. This complexity may make this a bad fit for projects with time issues.

The overlap algorithm on the over hand was very simple to implement. It required a small change within the nested loop responsible for generating vertices of increasing the size of the loop and offsetting all vertices south-west. Despite its simplicity, the algorithm effectively blends meshes and as such makes it a versatile and practical choice for many use cases.

The skirt algorithm was also easy to implement but was quite more difficult than the overlap algorithm. It was important to find a mesh topology that worked well in any situation so required quite a bit of trial and error. This algorithm, although not perfect, offers a good balance between ease of implementation and effective gap coverage.

6. Conclusion

6.1 Reflection of Objectives

6.1.1 Objective One

During this portion of the project, I was able to identify five possible algorithms for implementation. The algorithms that were ultimately chosen were those I believed had the most potential to address the T-Junction problem, however I realise that having a better selection of algorithms might have provided a more comprehensive result.

As an example of this, the skirt and offset algorithms turned out to be quite similar in terms of their structure. This similarity limited the diversity of results within the project. One such algorithm I could have implemented adjusts the height of a mesh object within the game world based on the player's distance to this object. This approach would have been another unique perspective on the T-Junction problem.

Overall, I believe I successfully met the objective of researching and identifying appropriate potential algorithms however, I acknowledge that a more diverse selection of algorithms would have been more beneficial for the project enabling a broader evaluation of the overall aim.

6.1.2 Objective Two

Three algorithms were implemented as required from the objective, although not to their fullest potential. There were numerous improvements and optimisations for each algorithm that could have been incorporated to improve performance but unfortunately many of these improvements were cut due to time constraints. These enhancements would have likely improved their performance and provided an accurate representation of their potential.

Despite this, I believe the overall trends of each implementation were still captured during the testing stage, still allowing for a fair and meaningful comparison. Overall, this objective was a success but, as mentioned, the algorithms could have greatly benefited from additional optimisations to further refine them and achieve better results.

6.1.3 Objective Three

Each algorithm was tested in both departments of performance and resource consumption, with CPU consumption, framerate and memory consumption being tested. While these tests provided a good understanding of the algorithms, they represented a small selection compared to the full range of potential tests that could have been conducted. They were enough to draw conclusions from, but a wider range of results would have helped provide a better analysis of the algorithms.

The analysis of each algorithm was performed in the Evaluation section, and their best use cases were identified. There was no algorithm that could fit any use case, like first thought at the beginning of the project. As mentioned before, a wider range of results would have helped draw better insights about each algorithm, but the overall trends were displayed well enough.

Overall, the testing half of this objective was somewhat limited in terms of variety and range of tests conducted and an expansion of this selection would have offered richer results with a more thorough evaluation of each algorithm.

6.1.4 Personal Development

This project has allowed me to develop skills in many areas such as research of niche algorithms and time management such as learning key lessons in keeping a schedule and timetabling goals with enough time to account for unexpected problems. Some of these skills include:

- Scripting using the Unity engine: I have learned lots of techniques to extract as much as I possibly can from the engine into my work.
- Procedural Generation: I have learned the method Unity requires to generate procedural meshes and have utilised this in many ways to generate what I need it to.
- Quad-Trees: The under-the-hood mechanics of a quad-tree have always been a mystery to me, so to use this opportunity to implement and work with quad trees in such a unique scenario has allowed me to appreciate it more.

6.1.5 Future Work

There are quite a few possible improvements that can be done. A lot of these have already been mentioned during objective reflection, such as implementing new algorithms, but there are many other improvements that the project can take.

A feature that would have been useful in analysing implemented algorithms was the inclusion of textures on the terrain. This would allow a simulation of a real game scenario and open up some more possibilities in terms of new algorithms such as blending together textures along mesh edges using shader programs.

The current mesh can be scaled to be any size, but if a really large size is required then it may cause issues. Implementing support to extend this mesh to have the illusion of infinite distance would create a more immersive solution in a game situation.

Similarly, as mentioned in the Changes since Proposal section, there is a removed objective of converting the flat plane of the current implementation into a globe. This would most likely be done using 6 separate meshes connected in the shape of a cube and each vertex transformed to create a sphere. This allows the project to test the algorithms in different use cases such as space games that need to render planets at large scales.

7. References

[1] https://developer.nvidia.com/gpugems/gpugems/part-i-natural-effects/chapter-5-implementing-improved-perlin-noise

[2] Platings, M. & Day, A.M. (2004) 'Compression of Large-Scale Terrain Data for Real-Time Visualization Using a Tiled Quad Tree', Computer graphics forum, 23(4), pp. 741–759.

[3] Hu, Z., Gavriushin, S., Petoukhov, S. & He, M. (2022) 'Algorithms Optimization for Procedural Terrain Generation in Real Time Graphics', in Advances in Intelligent Systems, Computer Science and Digital Economics III. [Online]. Switzerland: Springer International Publishing AG. pp. 125–137.

[4] José P. Suárez, Agustín Trujillo, José M. Santana, Manuel de la Calle, Diego Gómez-Deck, "An efficient terrain Level of Detail implementation for mobile devices and performance study", Computers, Environment and Urban Systems, Volume 52,2015,Pages 21-33, ISSN 0198-9715, https://doi.org/10.1016/j.compenvurbsys.2015.02.004.

[5] Pi, X., Song, J., Zeng, L. & Li, S. (2006) 'Procedural Terrain Detail Based on Patch-LOD Algorithm', in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. [Online]. 2006 Berlin, Heidelberg: Springer Berlin Heidelberg. pp. 913–920.