Bachelor Thesis

Implementing Static Mesh Hole Filling in Unreal Engine 4.27

Felix Wallquist
Computer Science

Studies from the School of Science and Technology at Örebro University
Örebro 2024

Studies from the School of Science and Technology at Örebro University

Felix Wallquist

Implementing Static Mesh Hole Filling in Unreal Engine 4.27

Supervisors: **Victor Björkgren, Piktiv AB**
Jasmin Grosinger, Örebro University
Examiner: **Andrey Kiselev, Örebro University**

Abstract

This project, completed in collaboration with Piktiv AB, aimed to develop an automated surface hole-filling feature for static meshes in Unreal Engine 4.27, with the goal of making repaired surfaces visually indistinguishable from their surrounding areas. The solution was primarily designed to address holes that arose from, but were not limited to, the use of Reduction Settings within Unreal Engine on static meshes. The functionality encompassed four key stages: boundary detection, where all holes on the mesh were identified; triangulation, which involved patching the hole using vertices from the boundary; refinement, entailing the addition of vertices and triangles to the patched area to mimic the density of the surrounding surface; and fairing, which smoothed the patched surface. Additionally, the project introduced a straightforward method for determining the texture coordinates of newly added vertices and a technique for ensuring that triangle normals correctly faced outward from the mesh.

The Static Mesh Hole Filler, as implemented, demonstrates efficiency in filling an arbitrary amount of small, planar holes, which commonly result from polygon reduction using Reduction Settings in Unreal Engine. However, this function falls short in preserving unique texture details and maintaining the curvature of surfaces when dealing with larger holes. This limitation necessitates users to seek alternative methods for effectively repairing the mesh.

Keywords

BSc Thesis, Computer Science, Computer Engineering, Unreal Engine, Static Mesh, Mesh Hole Filling

Contents

Chapter 1
Introduction

The realm of video game development has continually evolved, leveraging advanced technologies to create immersive and complex virtual worlds. At the heart of this evolution lies Unreal Engine, a state-of-the-art game engine developed by Epic Games. A game engine, in its essence, is a software framework used for the creation and development of video games. It provides game developers with a platform to build upon, offering tools for rendering graphics, simulating physics, scripting, animation, and much more. These engines are crucial in simplifying the game development process, enabling developers to focus more on the creative aspects of game creation rather than the technical intricacies.

The focus of this thesis is the implementation of a static mesh hole filling tool within Unreal Engine. A 'static mesh' in game development refers to a 3D model used within a game scene, which does not change its shape or form during gameplay. A detailed description of static meshes within the context of Unreal Engine can be found in section 1.1. These meshes are fundamental in constructing the visual aspects of a game's environment. However, a prevalent issue in game development, especially in the context of porting games to mobile platforms, is the need to optimize these static meshes. This optimization often involves reducing the polygon count of these meshes to enhance performance on less powerful devices. Unfortunately, this process can inadvertently create 'holes' or missing parts in the mesh, disrupting the game's visual integrity and player immersion.

The significance of this thesis lies in addressing this issue. By implementing a tool for automatic mesh hole filling, the process of porting games to mobile platforms becomes more efficient and less prone to visual degradation. The proposed tool aims to intelligently fill these gaps, preserving the original aesthetics of the game while ensuring optimal performance on mobile devices. This advancement not only streamlines the development process but also enhances the overall gaming experience on mobile platforms, where the demand for high-quality games continues to grow.

In summary, this project aims to contribute to the field of game development by introducing a solution that balances the visual fidelity and performance requirements of modern mobile gaming, all within the versatile environment of Unreal Engine.

1.1 Static Mesh

In the development of world geometry for different levels in games designed using Unreal Engine, most of the renderable elements within these levels consist of Static Meshes[6]. A Static Mesh is a three-dimensional polygon mesh comprising a collection of vertices, edges, and polygons with fixed geometry. The vertices are connected by edges, forming two-dimensional closed shapes bounded by these edges, referred to as polygons. In Unreal Engine, since all polygons in a Static Mesh are triangles, the term 'polygons' will henceforth be referred to as 'triangles'.

1.1.1 Manifold Triangle Mesh

In a manifold 3D triangle mesh, every edge connects exactly two triangles, forming a watertight structure. Triangles do not intersect or overlap with each other and there exists a distinct inside and outside of the mesh.

1.1.2 Non-manifold Triangle Mesh

Non-manifold 3D triangle meshes are meshes that don't conform to the previously described structure of a manifold 3D triangle mesh. The shape of non-manifold meshes can be of various complexity and irregularity, and some examples of non-manifold meshes are:

Internal triangles Meshes that include triangles within the volume of the distinct inside of the mesh.

Intersecting triangles Meshes where triangles intersect or cross with each other.

Overlapping edges or vertices Meshes were two or more edges or vertices occupy the same space but are not connected.

Meshes with holes Meshes with missing triangles. This results in holes in the mesh that are bounded by a loop of connected half-edges that are only connected to one triangle each.

Meshes with non-uniform normals Meshes with triangle normals pointing in towards to the distinct inside of the mesh.

1.2 Problem Formulation

Piktiv AB is a consulting company based in Örebro, Skövde and Stockholm, offering services within game and software development, along with digital marketing. As part of their game development services, they regularly perform porting of games for mobile using Unreal Engine. One part of the process of porting games for mobile involves reusing the same static meshes used in the original platform version but with each mesh's triangle amount reduced in order to optimize performance using Unreal Engine's built in Level Of Detail (LOD) Reduction Settings[2].

One recent such project involved porting a game for mobile using Unreal Engine 4.27 and within that project, it was noted that some static meshes became non-manifold as the amount of triangles were reduced past a certain threshold, see Figure 1. To combat this, two options exist; either increase the overall amount of triangles until the mesh is visibly manifold or spend valuable time manually fixing the mesh within a third party software. None of these options are practical as the first option diminishes the improvement on performance while the second option costs time and given that thousands of static meshes need to be reduced, and deadlines need to be met, that time is better spent elsewhere.

As neither of the two initial options is viable, a third alternative becomes necessary. The aim of this third option, and the overarching objective of this project, is to devise a method that not only conserves time but also retains the reduced triangle count in the static mesh undergoing modification. To achieve this, the strategy will involve developing specialized functionality to address the specific areas (non-manifold edges or vertices) causing the mesh's non-manifold status. This focused approach will correct these issues while maintaining the mesh's overall geometric structure. The project's end goal is to seamlessly integrate surfaces and textures into the repaired areas, making them visually indistinguishable from the rest of the mesh. Additionally, the implementation will be exclusively developed for Unreal Engine 4.27, ensuring it is precisely tailored to the needs and specifications of the project described. The function should preferably be made

callable from the details panel of the static mesh editor UI[7] by customizing the Unreal Engine editor.

1.3 Outline

The rest of this thesis is organised as follows:

Chapter 2 gives an overview and explanation for the various methods the constitute the basis of the implementation in this project.

Chapter 3 contains the implementation of the function Static Mesh Hole Filler, along with solutions for the various problems that were detected during development.

Chapter 4 contains the results as a visual comparison between the implemented function and Unreal Engine 5's native mesh filling functionality.

Chapter 5 contains conclusions drawn from the results along with a discussion on ways to improve the function.

Chapter 2
Related Works

2.1 Liepa's Method

In 2003, at the Eurographics Symposium on Geometry Processing, Peter Liepa introduced an innovative method for filling holes in 3D triangular meshes, as detailed in his publication[9]. Liepa's technique consists of four distinct stages: identifying the holes, triangulating these holes, refining the resulting structure, and then applying a fairing process for smooth integration. This method makes some assumptions regarding the mesh being processed, namely that the mesh is oriented (contains no non-uniform normals), manifold and connected. This implies that the method is not equipped to manage scenarios involving singular vertices, defined as vertices connected to more than two boundary edges. Additionally, it is not capable of addressing situations where the hole being processed encompasses islands, which are distinct connected meshes within the hole that do not connect to the rest of the mesh.

2.1.1 Hole Identification

Liepa's approach to identifying holes in a mesh is straightforward and efficient in the presence of simple holes[9]. It begins with the selection of a seed boundary vertex, which is one of the two vertices connected to a boundary edge. From this starting point, the method involves traversing along the connected boundary edges in a continuous loop until returning to the seed vertex. This process effectively outlines the boundary loop that defines the hole.

2.1.2 Hole Triangulation

For triangulating an identified hole, Liepa uses an already existing algorithm[1] for yielding a minimum area triangulation, see Algorithm 1. The algorithm performs a triangulation on the given hole which minimizes the total area of the triangles and utilizes a weightset:

$$L_{area} = [0, \infty), \text{ with } 0_{L_{area}} = 0,$$

which represents a range of weights from 0 to infinity, starting initially at 0. This set is used to evaluate different triangulation options based on their associated weights.

Along with this, a weighting function is used:

$$\Omega_{area}(v_i, v_m, v_k) = \text{Area of } (v_i, v_m, v_k).$$

This function calculates the area of a triangle formed by vertices v_i, v_m and v_k. It is used to assign a weight to each triangulation option, with the algorithm favoring options that minimize the total area of the triangles.

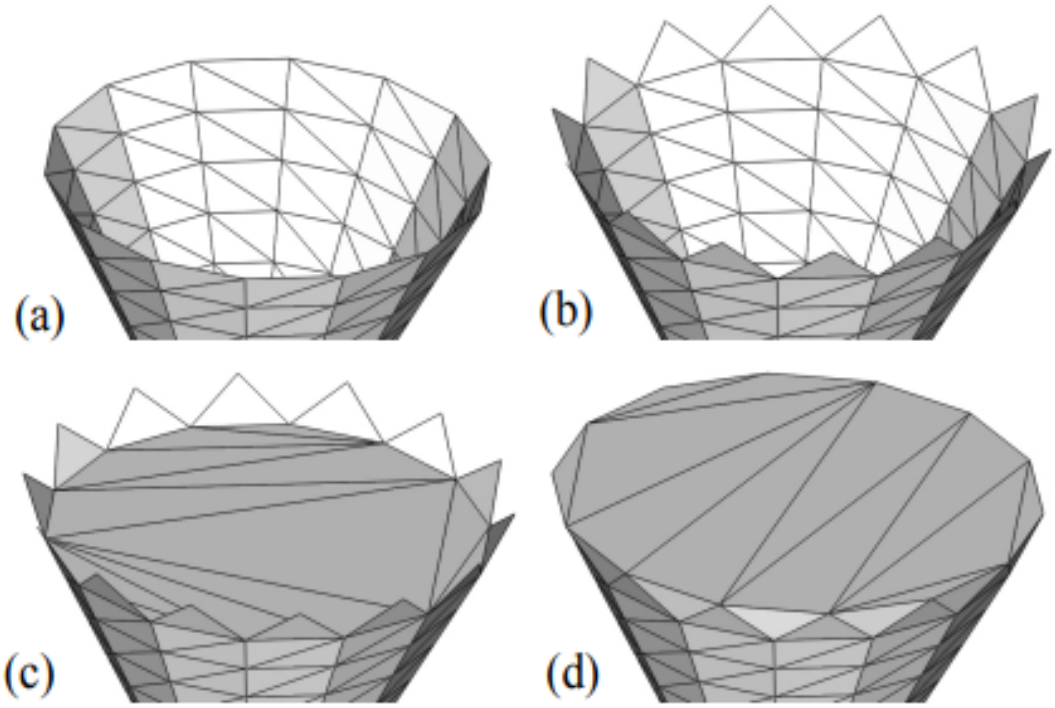


Figure 2.1: (a) Hole with no crenellations. (b) Hole with crenellations. (c) Hole triangulated using minimum-area triangulation. Leaves behind non-manifold edges. (d) Hole triangulated using Liepa's min-max dihedral angle triangulation. Figure is from [9].

Liepa also goes on to extend the algorithm by introducing a new weighting on the triangles that takes into account their dihedral angles with adjacent triangles. The reasoning behind this extension lies within the minimum-area triangulation's shortcomings when the plane of the triangulation is orthogonal to features in the boundary. These features are called *crenellations* (Figure 2.1(b)) and, when using a minimum-area triangulation, they tend to be left out of the triangulated plane, leaving behind non-manifold edges, as can be seen in Figure 2.1(c).

Liepa's proposed weightset L_{angle}[9] is defined as:

$$L_{angle} = [0, \pi] \times [0, \infty), \text{ with } 0_{L_{angle}} = (0, 0).$$

This weightset combines dihedral angles (ranging from 0 to π) and areas (ranging from 0 to ∞), starting with an initial weight of (0,0). The ordering of the weightset gives precedence to dihedral angles over areas with the aim to penalize large dihedral angles:

$$(a, b) < (c, d) \iff (a < c \text{ or } (a = c \text{ and } b < d)).$$

The addition operator for Liepa's weightset sums the areas of the weights, as does the weightset for minimum-area triangulation. However, it retains the largest dihedral angle:

$$(a, b) + (c, d) := (\max(a, c), b + d).$$

The weighting function Ω_{angle} is expressed as:

$$\Omega_{angle}(v_i, v_m, v_k) = (\mu(v_i, v_m, v_k), \Omega_{area}(v_i, v_m, v_k))$$

where $\mu(v_i, v_m, v_k)$ is the maximum dihedral angle between the triangle (v_i, v_m, v_k) and existing adjacent triangles. By replacing the minimum-area weighting in Algorithm 1 with

Algorithm 1 Minimum Weight Triangulation

Require: A list of vertices V of the polygon to triangulate.
Ensure: The minimum weight $W_{0,n-1}$ of the triangulated polygon.

1: **for** $i = 0$ to $n-2$ **do**
2: $W_{i,i+1} \leftarrow 0$
3: **end for**
4: **for** $i = 0$ to $n-3$ **do**
5: $W_{i,i+2} \leftarrow \Omega(v_i, v_{i+1}, v_{i+2})$
6: **end for**
7: $j \leftarrow 2$.
8: **while** $j < n-1$ **do**
9: $j \leftarrow j+1$
10: **for** $i = 0$ to $n-j-1$ **do**
11: $k = i+j$
12: $W_{i,k} \leftarrow min_{i<m<k}(W_{i,m} + W_{m,k} + \Omega(v_i, v_m, v_k))$
13: Record the index m where the minimum is achieved as $\lambda_{i,k}$
14: **end for**
15: **end while**
16: $S \leftarrow$ Trace(0,$n-1$) // **Recovers the triangulation**

Algorithm 2 Trace

1: Function Trace(i, k):
2: **if** $i+2 = k$ **then**
3: $S \leftarrow S \cup \Delta v_i, v_{i+1}, v_k$ // **Adds triangle with vertices** v_i, v_{i+1}, v_k **to** S
4: **else**
5: $o \leftarrow \lambda_{i,k}$
6: **if** $o \neq i+1$ **then**
7: Trace(i, o)
8: **end if**
9: $S \leftarrow S \cup \Delta v_i, v_o, v_k$ // **Adds triangle with vertices** v_i, v_o, v_k **to** S
10: **if** $o \neq k-1$ **then**
11: Trace(o, k)
12: **end if**
13: **end if**

this weighting, the algorithm yields a triangulation that minimizes the maximum dihedral angle and the results can be seen in Figure 2.1(d).

2.1.3 Refinement

For refining the triangulation computed in the previous step, Liepa made an adaptation[9] of an algorithm by Pfeifle and Seidel[10], see Algorithm 3. The algorithm introduces a scale attribute for each vertex on the hole boundary, which is computed as the average length of each edge adjacent to the vertex. These values are then diffused into the interior of the patching mesh, leading to the subdivision of triangles in order to reduce edge lengths. The algorithm also employs a relaxation process on the interior edges of the patching mesh. Relaxation of an edge involves verifying whether the vertices opposite this edge, within the two adjacent triangles, lie outside the circumscribing sphere of the counterpart triangle. If either vertex lie inside of the opposite triangles circumsphere, the edge is swapped, as can seen in Figure 2.2(b). The algorithm additionally utilizes a density control factor α, as can be seen on line 9 in Algorithm 3. Liepa found that a density control factor of $\alpha = \sqrt{2}$ yields the best visual results in regards to matching the density of the surrounding mesh.

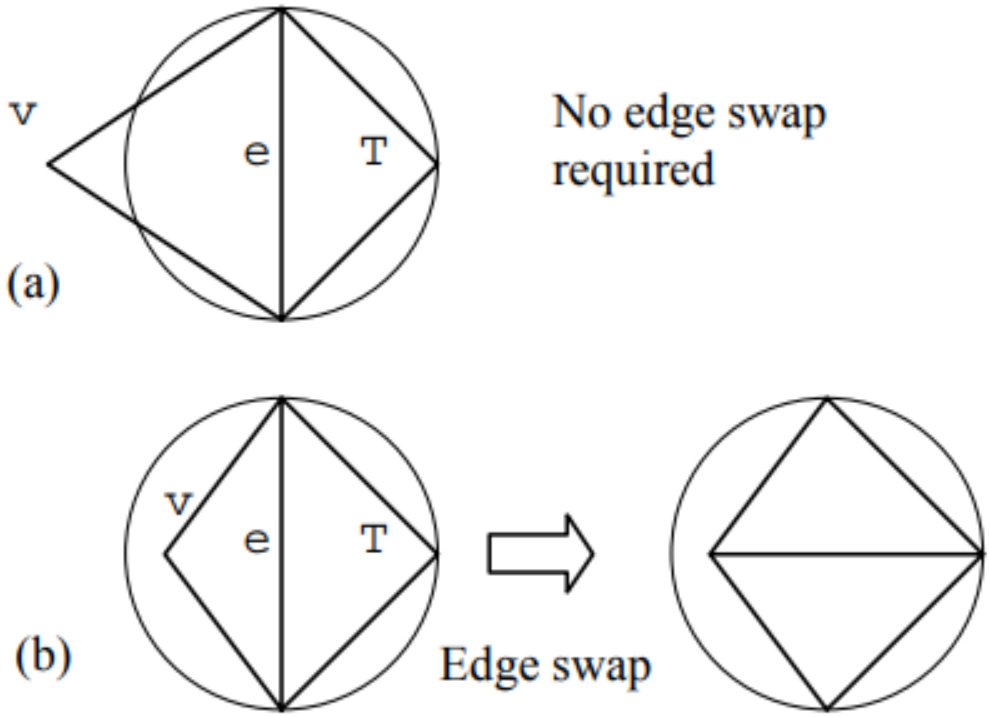


Figure 2.2: (a) Vertex v lies outside of the circumsphere of opposite triangle T. (b) Vertex v lies inside of the circumsphere of opposite triangle T, resulting in the edge being swapped. Figure is from [9].

Algorithm 3 Refinement Algorithm

1: **for** each vertex v_i on the hole boundary **do**
2: Compute the scale attribute $\sigma(v_i)$ as the average length of the edges adjacent to v_i in the surrounding mesh
3: **end for**
4: Initialize the patching mesh as the given hole triangulation
5: **for** each triangle (v_i, v_j, v_k) in the patching mesh **do**
6: Compute the centroid v_c of the triangle
7: Compute the scale attribute for the centroid: $\sigma(v_c) \leftarrow (\sigma(v_i) + \sigma(v_j) + \sigma(v_k))/3$
8: **for** m in {i, j, k} **do**
9: **if** $\alpha\|v_c - v_m\| > \sigma(v_c)$ and $\alpha\|v_c - v_m\| > \sigma(v_m)$ **then**
10: Replace triangle (v_i, v_j, v_k) with triangles (v_c, v_j, v_k), (v_i, v_c, v_k), and (v_i, v_j, v_c) in the patching mesh
11: Relax the edges (v_i, v_j), (v_i, v_k), and (v_j, v_k)
12: **end if**
13: **end for**
14: **end for**
15: **if** no new triangles were created in the previous step **then**
16: The patching mesh is complete
17: **else**
18: Relax all interior edges of the patching mesh
19: **if** no edges were swapped in the previous step **then**
20: Go to line 5
21: **else**
22: Go to line 18
23: **end if**
24: **end if**

2.1.4 Fairing

For the process of fairing, Liepa[9] makes use of a generalized version of the weighted umbrella-operator. The goal of this operator is to smooth out the surface by modifying the position of vertices based on the weighted average of their direct neighbors.

The weighted umbrella-operator:

$$U_\omega(v) = -v + \frac{1}{\omega(v)} \sum_i \omega(v, v_i) v_i,$$

where $\omega : V^2 \to \mathbb{R}$ is a weighting function that assigns a weight to every edge (v_i, v_j). The term $\omega(v)$ is the sum of weights of edges connected to vertex v, where v_i are the direct neighbors of v:

$$\omega(v) = \sum_i \omega(v, v_i).$$

The operator $U_\omega(v)$ essentially calculates a displacement vector for each vertex v, based on the weighted average position of its neighbors.

Different versions of the umbrella-operator can be obtained by modifying the weighting function for edges. Liepa gives three examples in his method[9]:

The uniform umbrella-operator

$$\omega(v_i, v_j) = 1,$$

assigns a uniform weight of 1 to every edge. This means each neighbor contributes equally to the new position of the vertex, regardless of the geometric properties of the mesh.

The scale-dependent umbrella-operator

$$\omega(v_i, v_j) = \|v_i - v_j\|,$$

assigns weights based on the Euclidean distance between vertices v_i and v_j. In this case, longer edges have more weight, thus greater influence in the repositioning of the vertex.

The harmonic umbrella-operator

$$\omega(v_i, v_j) = \cot(\angle(v_i, v_{k_1}, v_j)) + \cot(\angle(v_i, v_{k_2}, v_j)),$$

uses the cotangent of angles opposite the edge in the two adjacent triangles to assign weights. Here, edges with smaller opposing angles have higher weights. This operator is sensitive to the angles in the mesh, helping to preserve geometric details..

Additionally, the umbrella-operator can be applied recursively, resulting in *the second-order weighted umbrella-operator*

$$U^2_\omega(v) = -U_\omega(v) + \frac{1}{\omega(v)} \sum_i \omega(v, v_i) U_\omega(v_i),$$

which involves applying the umbrella-operator on each vertex v along with each of v's direct neighbors v_i in order to take into account the broader neighborhood of v, which can lead to a smoother and more globally fair surface.

2.2 Yip, Stahl, and Schellewald's Hole-Detection Technique

Yip, Stahl, and Schellewald[11] introduce a robust method for hole detection in 3D triangular meshes, capable of identifying all simple holes present in any edge-manifold triangular mesh, even in the presence of complex holes. The defining characteristic of a complex hole is the presence of singular vertices on the boundary loop, an example of which can be seen in Figure something. Yip, Stahl, and Schellewald divide their method into three algorithms which will be explained below.

The purpose of Algorithm 4[11] is to find the next half-edge when traversing the half-edges on a boundary. If the boundary contains singular vertices, traversing it is not entirely trivial. Singular vertices are connected to more than two half-edges, so when traversing one of them and passing the singular vertex, it is not quite as simple as traversing the next connected half-edge. A choice needs to be made, and if we make the right choice in every scenario, we are able to identify the entirety of the complex hole. Algorithm 4 makes the right choice by utilizing the adjacent triangle of the current half-edge to find the second edge within that triangle that is connected to the vertex being passed, known as the transition edge. This is repeated for every transition edge until the correct next half-edge is found. This method is used when passing both non-singular and singular vertices, as can be seen in Figure 2.3(a)(b).

Yip, Stahl, and Schellewald's second algorithm[11], Algorithm 5, constructs the boundaries on the mesh. These boundaries can be either simple or complex. The algorithm makes use of a set of all half-edges present within the mesh, from which it selects a random half-edge as a start half-edge. The boundary that includes the starting half-edge is traversed using Algorithm 4 until the algorithm loops back to the starting half-edge, forming a closed loop. Once a closed loop is confirmed, the half-edges on the boundary are removed from the half-edge set, making sure each half-edge in the half-edge set is present in only one boundary.

Once all boundaries present on the mesh have been detected and constructed, the next step is to divide all and any complex boundaries into simple boundaries. Yip, Stahl, and Schellewald's third algorithm[11], Algorithm 6, has the objective of breaking down a

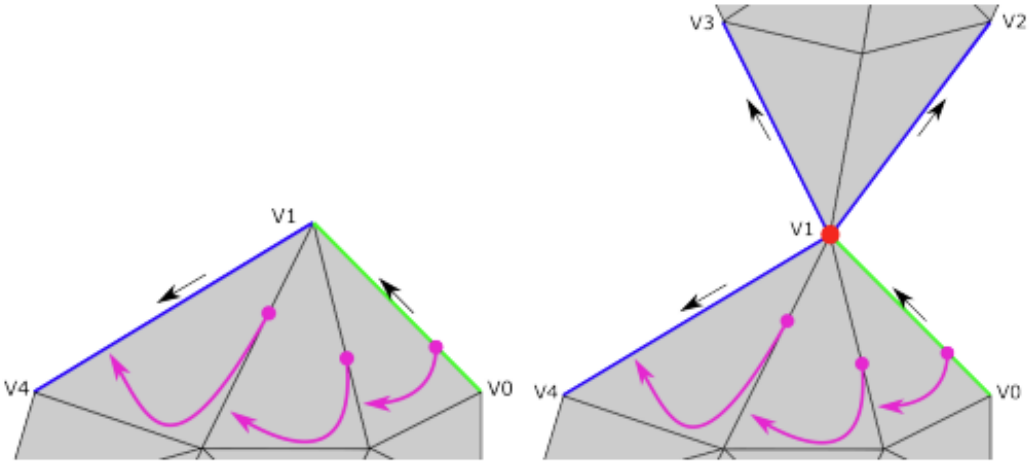


Figure 2.3: (a) Traversing a non-singular vertex v_1 (b) Traversing a singular vertex v_1. Figure is from [11].

Algorithm 4 Calculate the next half-edge

Require: Current half-edge $h_{current}$, set of half-edges H, and triangulation T
Ensure: Next half-edge h_{next}

$t \leftarrow$ Find_triangle_containing_half_edge$(h_{current}, T)$
$e_{current} \leftarrow$ Find_transition_edge$(h_{current}, t)$
if $e_{current} \in H$ **then**
  $h_{next} \leftarrow e_{current}$
  return h_{next}
else
  $e_{previous} \leftarrow h_{current}$
  while true **do**
    $t \leftarrow$ Find_triangle_containing_first_edge_but_not_second$(e_{current}, e_{previous}, T)$
    $e_{next} \leftarrow$ Find_transition_edge$(e_{current}, t)$
    if $e_{next} \in H$ **then**
      $h_{next} \leftarrow e_{next}$
      return h_{next}
    else
      $e_{previous} \leftarrow e_{current}$
      $e_{current} \leftarrow e_{next}$
    end if
  end while
end if

Algorithm 5 Construct Boundaries

```
1: Initialize B as an empty set
2: while H is not empty do
3:    h_start ← Random_select(H)
4:    Initialize an ordered array b with h_start
5:    h_current ← h_start
6:    repeat
7:       h_next ← next_half_edge(h_current, H, T)
8:       Reorient h_next to h_current if necessary
9:       if h_next is the same as h_start then
10:         Break from the loop
11:      else
12:         Add h_next to b
13:         h_current ← h_next
14:      end if
15:   until the loop is exited
16:   Add b to the set B
17:   Remove the edges in b from H
18: end while
19: return B
```

single complex boundary into multiple simple ones. This is done by identifying repeated vertices (the result of the presence of singular vertices) on the boundary and recursively decomposing the complex boundary by breaking out simple boundaries. The result of Yip, Stahl, and Schellewald's method is the identification and construction of all simple boundaries present on the mesh, regardless of the presence of singular vertices.

Algorithm 6 Decompose a Complex Boundary into Simples

```
1: S ← ∅
2: if has_repeated_vertex(b) then
3:    [index_1, index_2] ← Find_repeated_index(b)
4:    b1 ← b[1 : index_1] + b[index_2 : end]
5:    b2 ← b[index_1 : index_2]
6:    S1 ← decompose_complex_to_simples(b1)
7:    S2 ← decompose_complex_to_simples(b2)
8:    S ← S1 ∪ S2
9: else
10:   S ← {b}
11: end if
12: return S
```

Chapter 3
Implementation

3.1 Unreal Engine 4.27

Unreal Engine 4.27 is a premier 3D creation tool developed by Epic Games, renowned for its advanced graphical rendering, including dynamic lighting and global illumination, which contribute to its high degree of realism. This versatile platform supports VR and AR development, offering a comprehensive suite of animation, particle systems, and physics simulation tools. With enhancements for improved performance and material creation, Unreal Engine 4.27 serves a wide array of creators, from game developers to architects, providing a user-friendly interface and powerful Blueprint scripting system for efficient project development across various platforms.

This project was implemented as a C++ project within Unreal Engine 4.27, specifically tailored to address the specified problem section 1.2. A blueprint callable UFunction[8] was defined using Unreal Engine's UFUNCTION macro with the added BlueprintCallable function specifier, exposing the function to Unreal's Blueprint Visual Scripting system.

3.2 Blender 4.0

Blender 4.0 is the latest iteration of the open-source 3D creation suite, widely recognized for its robust capabilities in modeling, animation, simulation, rendering, and more. With an array of advanced features and a user-friendly interface, Blender 4.0 serves as an invaluable tool in the 3D graphics and animation industry.

In the context of this project, Blender 4.0 was employed primarily for manual mesh modification and result visualization. Specific tasks performed using Blender included the deliberate removal of vertices to create holes within meshes. This process was vital for generating test cases that mimic scenarios where meshes might be incomplete or damaged, thereby providing a realistic set of data to assess the effectiveness of the custom static mesh hole functionality developed in Unreal Engine 4.27.

3.3 Static Mesh Hole Filler

This section of the thesis is dedicated to an in-depth explanation of the primary function implemented, known as the Static Mesh Hole Filler. It is structured into several subsections, each elucidating a different aspect of the function.

In subsection 3.3.1, a concise summary is provided, detailing the function's objectives and its significance within the broader context of the project. This part sets the stage by giving a high-level understanding of what the Static Mesh Hole Filler is designed to accomplish.

Next, subsection 3.3.2 delves into the specifics of the function. Here, each parameter of the Static Mesh Hole Filler is explained, outlining its role and how it influences the function's behavior. This section is crucial for understanding the customizable aspects of the function and how it can be adapted to various scenarios.

Subsection 3.3.3 dissects the function into its constituent stages. Each stage is detailed, explaining its purpose and how it contributes to the overall functionality of the Static Mesh Hole Filler. This breakdown conceptualizes the function's internal workings, offering insight into the logical flow and the decision-making process based on the arguments provided.

Finally, two specialized subsections, 3.3.4 and 3.3.5 address specific challenges encountered during the function's development. Subsection 3.3.4 discusses the issue of non-uniform normals, which can lead to triangles not rendering correctly, and presents the strategies implemented to overcome this problem. Meanwhile, subsection 3.3.5 focuses on the technicalities of assigning correct texture coordinates to new vertices, ensuring that the mesh's visual integrity is maintained after the hole-filling process.

Overall, this section is a comprehensive exploration of the Static Mesh Hole Filler function, offering readers a detailed understanding of its purpose, design, and the technical challenges it addresses.

3.3.1 Function Overview

The **Static Mesh Hole Filler** is a designated `UFunction` meticulously developed for the purpose of modifying static meshes within the Unreal Engine environment. This function is integral to the process of programmatically repairing and enhancing mesh structures by identifying and filling both simple and complex holes on the surface of static meshes. It is characterized by a set of six parameters, each serving a distinct function in the mesh modification process.

3.3.2 Parameters

1. **StaticMesh (UStaticMesh*):** This parameter is a pointer to the `UStaticMesh` object that is subject to modification. It provides the function with direct access to the mesh data, enabling targeted alterations.

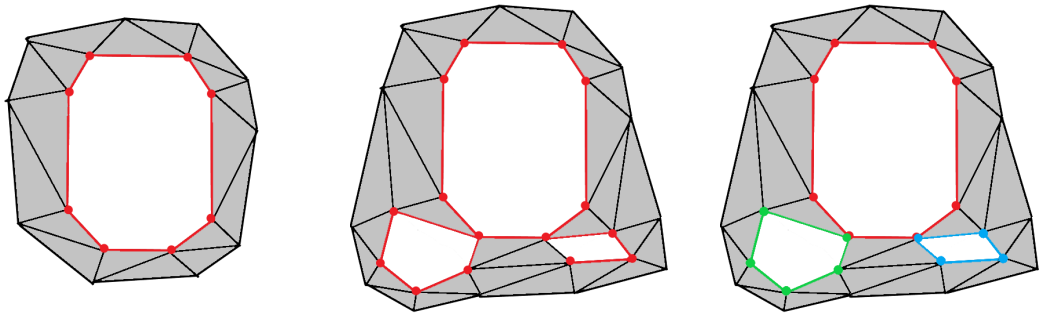
2. **LodIndex (int):** Represents the LOD index of the Static Mesh that is to be targeted. This integer parameter allows the function to precisely identify and operate on the specific LOD layer of the mesh, facilitating focused and efficient mesh refinement.

3. **Method (Enum):** An enumerated type that determines the triangulation method to be employed. This parameter allows for the selection of a triangulation strategy, each option catering to different mesh topologies and hole complexities, thereby offering tailored modification approaches.

4. **RMethod (Enum):** Another enumerated type that dictates whether the mesh should undergo refinement. This decision parameter is crucial for determining the level of detail and smoothness of the resulting mesh after the hole-filling process.

5. **FMethod (Enum):** This parameter decides the fairing method to be applied. Fairing is a critical post-processing step that smoothens and improves the visual continuity of the mesh. The chosen method directly impacts the aesthetic and structural quality of the final output.

6. **NumFairIt (int):** Specifies the number of fairing iterations to be conducted. This integer parameter allows the user to define the intensity and depth of the fairing process, directly influencing the smoothness and quality of the mesh surface post-modification.



(a) (b) (c)

Figure 3.1: Illustration of different types of holes in 3D mesh models. (a) Shows a simple hole where each vertex on the boundary connects to exactly two half-edges, forming a clean gap without any irregular vertices. (b) Depicts a complex hole that includes two singular vertices, which are points on the mesh where the usual edge connectivity is disrupted, resulting in an irregular boundary. (c) Presents a decomposed complex hole that has been divided into three simpler holes, each delineated by a different color (red, green, and blue), to demonstrate how a complex hole can be broken down into simpler components for easier processing and reconstruction.

3.3.3 Breakdown of Function Stages

Boundary Detection

The initial segment of the function is dedicated to identifying all simple boundaries on the static mesh. A simple boundary can be seen in Figure 3.1(a). This process is critical, yet Liepa's method for hole detection[9] demonstrates limitations. Specifically, it naively assumes all mesh boundaries are simple, an oversimplification which is particularly inadequate when considering Unreal Engine's polygon reduction settings. These settings can introduce complex containing singular vertices as can be seen in Figure 3.1(b), rendering Liepa's approach insufficient for comprehensive boundary detection in this context.

For this very reason, the choice was made to replace Liepa's boundary detection with that of Yip, Stahl and Schellewald[11]. The implementation for boundary detection involves finding all half-edges on the mesh and adding it to a set. A random half-edge is then chosen from the set and adjacent half-edges are traversed using Algorithm 4 until a simple or complex boundary loop is formed, at which point the half-edges on the boundary are removed from the half-edge set. This is repeated using Algorithm 5 until the half-edge set is empty. All boundaries found are then decomposed into simple boundaries using Algorithm 6, resulting in the detection of all simple boundaries present on the surface of the mesh, seen in Figure 3.1(c). Each boundary is then processed and patched using the methods provided by the arguments used for the function.

Hole Triangulation

Once a boundary loop is detected, triangulation is carried out using Algorithm 1, accompanied by the preferred weighting function determined by the **Method** argument. This enumeration includes two options: **AREA** for minimum area triangulation and **ANGLE** for min-max dihedral angle triangulation. The **Method** argument specifies the triangulation technique applied to all processed boundaries.

The **AREA** method is adept at optimizing the triangulation of a polygonal area defined by a boundary loop in a mesh, aiming to minimize the total area of the resulting triangles. It works by iteratively considering segments of the boundary loop, determining the best way to divide them into triangles to ensure the smallest possible total area. Central to its operation is a dynamic programming approach that stores and reuses intermediate

results, enhancing efficiency and effectiveness, especially for complex meshes with numerous vertices. This method ensures quick determination of optimal triangulation by maintaining a record of minimal areas and the corresponding vertex indices that lead to these configurations as it progresses through the boundary loop.

The **ANGLE** function, similar to the **AREA**, optimizes the triangulation of a polygonal area in a mesh, but with a nuanced focus on minimizing angles between triangles as well as the total area. However, it's important to note that the **ANGLE** doesn't always function flawlessly; at times, it can lead to an index out of range error, resulting in the crashing of the editor. This issue underscores a critical area for further investigation and refinement. Despite this, the function operates by iteratively examining segments of the boundary loop, aiming to strike a balance between the smallest area and the smoothest surface by maximizing the dot product of normals between adjacent triangles. This approach ensures that the mesh surface is not only compact in area but also aesthetically smooth and aligned. Employing a dynamic programming strategy, the function efficiently stores and utilizes intermediate results, calculating both the area and dot product for potential triangles to determine the most optimal configuration in terms of area and angular harmony.

Refinement

Once the triangulation is complete, the actual triangles are created and stored in a separate array of triangles as the patching mesh, not connected to the original mesh. This mesh then undergoes either refinement or no refinement based on the **RMethod** argument. This argument is an enumeration consisting of **NO_REFINEMENT** and **REFINE**. If **REFINE** is selected, the patching mesh is refined according to Algorithm 3. This involves iteratively passing through all the triangles in the mesh and dividing those that meet the criteria specified in line 9 of Algorithm 3. The refinement process is considered complete once a pass through all the triangles is made without any division occurring. However, during the refinement process, a bug was identified where the mesh occasionally becomes non-manifold; specifically, edges may end up with more than two adjacent triangles. This leads to an infinite loop. As a temporary solution, the refinement process is halted once the mesh becomes non-manifold. All changes made are then discarded, and the mesh is left unprocessed. This means that in some cases, the user might need to run the function multiple times until there is no non-manifoldness detected.

Connecting the patching mesh

After refining the patching mesh, the function connects it to the original mesh. This is done by iterating through the triangles of the patching mesh and creating new triangles in the original mesh. It achieves this by either reusing vertices on the boundary or creating new vertices as needed.

Fairing

The final phase of the Static Mesh Hole Filler process is to smooth out the patching mesh. This is achieved by using the **FMethod** parameter to determine the fairing technique to be applied, or to decide if fairing should be skipped altogether. The available options are **NO_FAIR**, **UNIFORM**, **SCALE**, and **HARMONIC**. The last three represent the three weighting functions detailed in section 2.1.4. If one of these three options is selected, the fairing function begins by establishing a set of processed vertices to ensure that the position of any single vertex is modified only once. The process involves all the vertices of the triangles within the patching mesh. Initially, it calculates the total weight of all edges connected to a vertex using the chosen weighting function. Then, it computes the sum of each edge's weight multiplied by the position of the opposing vertex. The umbrella-operator is derived by dividing this second sum by the first and then subtracting the current vertex's position. After determining the umbrella-operator, the new vertex position

is obtained by adding the umbrella-operator to its existing position. After all vertices are processed, one iteration is completed. The total number of fairing iterations to be conducted is specified by the **NumFairIt** parameter.

3.3.4 Solution For Non-Uniform Normals

As the implementation mentioned above, along with its underlying algorithms, does not consider triangle normals, the addition of new triangles to the original mesh resulted in some triangles exhibiting non-uniform normals, meaning they were not rendered in Unreal Engine. A solution to this issue involves a method to determine whether the orientation of an added triangle's face should be reversed. This method entails calculating the average normal of all triangles adjacent to the boundary. Once a new triangle is added, the dot product of the new triangle's normal and the average normal is calculated. If the dot product yields a result less than zero, the orientation of the new triangle's face is reversed, effectively ensuring the triangle aligns uniformly with the normals.

3.3.5 Determining Texture Coordinates

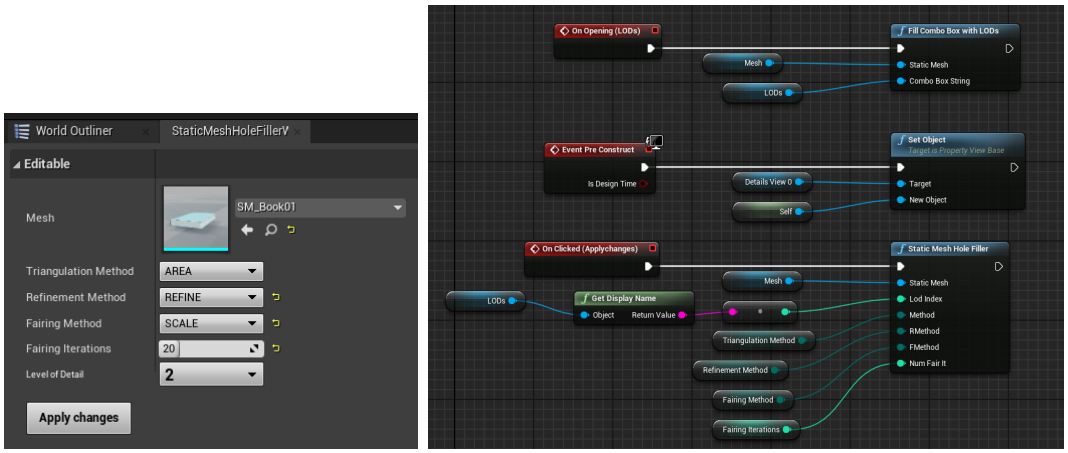
In Unreal Engine, embedded texture coordinates within each vertex are crucial for determining the surface texture of the triangles they form. When these coordinates are coupled with a material, they establish the definitive look of the triangle's surface. However, when a new vertex is created in Unreal, its texture coordinates default to zero. This leads to an inaccurate rendering of the surface for the newly created triangle, as the default coordinates do not typically correspond to the correct texture. To attain a visually appealing surface texture, Algorithm 7 is introduced. This algorithm activates whenever a triangle is subdivided and its centroid becomes a vertex in the mesh. It allocates weights to each vertex of the original triangle based on their Euclidean distance to the centroid, assigning larger weights to vertices closer to the centroid. These weights are then utilized to compute the weighted sum of the UV coordinates for each vertex, which is subsequently assigned to the newly formed vertex at the centroid. This ensures a seamless and aesthetically consistent texture mapping across the mesh.

Algorithm 7 Calculate Weighted UV Coordinates

Require: Vertex positions A, B, C, Point P, UV coordinates UV_A, UV_B, UV_C
Ensure: Weighted UV coordinates $WeightedUV$
1: FUNCTION CalculateWeights(P, A, B, C)
2: $AWeight \leftarrow 1.0/\text{Distance}(A, P)$
3: $BWeight \leftarrow 1.0/\text{Distance}(B, P)$
4: $CWeight \leftarrow 1.0/\text{Distance}(C, P)$
5: $TotalWeight \leftarrow AWeight + BWeight + CWeight$
6: **return** $(AWeight/TotalWeight, BWeight/TotalWeight, CWeight/TotalWeight)$
7: $Centroid \leftarrow$ [Define or obtain the centroid]
8: $Weights \leftarrow CalculateWeights(Centroid, A, B, C)$
9: $WeightedUV \leftarrow Weights.X \cdot UV_A + Weights.Y \cdot UV_B + Weights.Z \cdot UV_C$

3.4 User Interface Design for the Static Mesh Hole Filler Functionality

An Editor Utility Widget[3] was developed and employed for the purposes of initiating the exposed UFunction. The widget, along with its blueprint, can be seen in Figure 3.2(a) and (b) respectively. The widget contains six editable properties corresponding to the six



(a) (b)

Figure 3.2: Interface and Blueprint Visual Scripting of the Static Mesh Hole Filler in Unreal Engine. (a) The user interface within the Unreal Engine editor, designed as a Utility Widget, which allows users to invoke the hole filling process on static meshes. It provides options to select a mesh and set parameters for triangulation method, refinement method, fairing method, number of fairing iterations, and level of detail before applying changes. (b) The underlying blueprint network, constructed using Unreal Engine's Blueprint Visual Scripting system, which outlines the logic and flow of data that drive the Static Mesh Hole Filler functionality. It illustrates event-driven programming paradigms and interaction with the user interface elements, showcasing how the tool responds to user input and executes the hole filling process.

parameters of the Static Mesh Hole Filler function. The user begins by selecting a target static mesh from the content browser for modification. Subsequently, they choose preferred methods for triangulation, refinement, and fairing via respective dropdown menus. A slider allows the user to specify the number of fairing iterations, adjustable between 0 and 100. Should **NO_FAIR** be chosen as the fairing method, the iteration setting is disregarded. The final step involves selecting the Level of Detail (LOD) index for the modifications on the static mesh. This dropdown menu refreshes upon each access to accommodate varying LOD counts across different meshes. Once all settings have been set, the user simply presses the **Apply changes** button to perform the modifications to the static mesh.

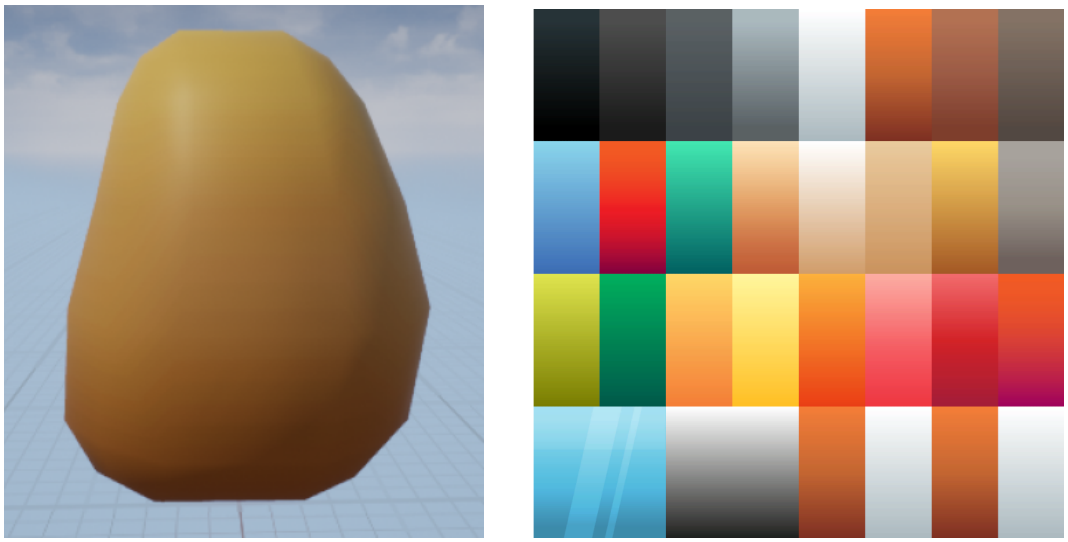
Chapter 4
Results

Considering the issue of holes in static meshes and the requirement for meshes in mobile games to appear visually manifold, the outcomes of this project will be assessed primarily on a visual basis. The objective of the Static Mesh Hole Filler function is to identify and fill both simple holes, which contain no singular vertices, and complex holes, which may contain any number of singular vertices, in order for the mesh to appear visually manifold and watertight. To ascertain the effectiveness of the Static Mesh Hole Filler, a comparative analysis will be conducted between the implemented function and Unreal Engine 5's 'Fill All Mesh Holes'[4] feature, accessible through the Geometry Script library[5]. The comparison will encompass triangulating both simple (section 4.1) and complex holes (section 4.2) using the two methods and comparing the resulting patched mesh to the original mesh, evaluating the visual outcomes of each method.

This comparison will be made on two separate static meshes. The first is a relatively simple oval object (a potato), initially containing just 80 vertices before the introduction of holes. This design choice aims to mimic the simplicity of polygon-reduced objects typically utilized in ported mobile games. The mesh, along with the texture used for it's surface, can be seen in figure 4.1.

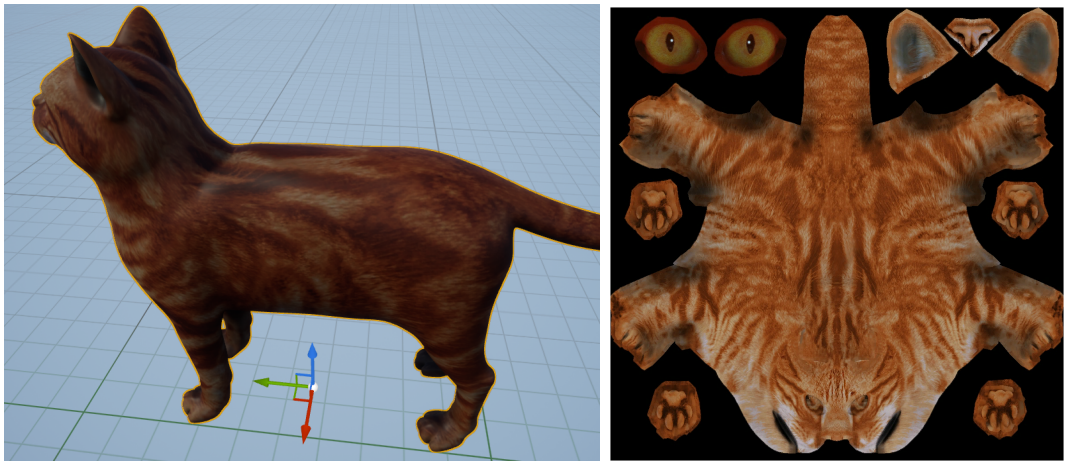
The second static mesh (a cat), seen in Figure 4.2, features a significantly higher level of detail, boasting a total of 35,290 vertices. Beyond the substantial increase in vertex count, this mesh employs a complex texture material that assigns distinct portions of the image to corresponding parts of the 3D model, adding to its realism and visual intricacy. In contrast, the first mesh utilizes a simpler gradient map for texturing, which, while less complex, offers a more streamlined and less detailed appearance.

Additionally, an analysis will be made (section 4.3) of the Static Mesh Hole Filler's ability to refine and fair non-planar surfaces with the objective of making the repaired hole indistinguishable from the surrounding area.



(a) (b)

Figure 4.1: Illustrates the initial state of a static mesh and its associated texture mapping. (a) presents a complete static mesh designed for surface rendering without any missing elements. (b) displays the texture atlas used for the mesh, highlighting the specific texture coordinates utilized to map the surface details of the mesh.

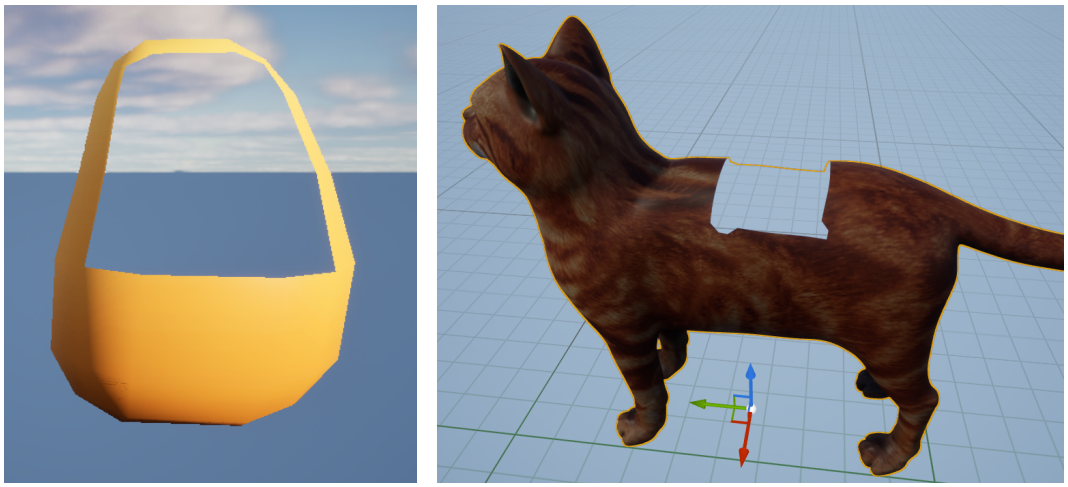


(a) (b)

Figure 4.2: Showcases a textured static mesh model and its corresponding texture layout. (a) shows the model fully textured without any missing sections or defects. (b) displays the detailed texture map applied to the model, which includes all the necessary surface information for realistic visual representation.

4.1 Filling Simple Holes

Firstly, a comparison was made between the two functions on their ability to fill simple holes on the two different objects in order to determine how distinguishable the filled areas are compared to their surrounding mesh. The two meshes containing introduced simple holes can be seen in Figure 4.3.



(a) (b)

Figure 4.3: Display of static meshes with the introduction of a simple hole. (a) A potato-shaped mesh with a singular, clean-cut hole that interrupts the continuity of the surface. (b) A cat-shaped mesh also featuring a single, simple hole, which showcases the impact of such a defect on a complex, textured 3D model. These examples serve to illustrate the initial state of meshes prior to the application of hole-filling algorithms.

4.1.1 Fill All Mesh Holes

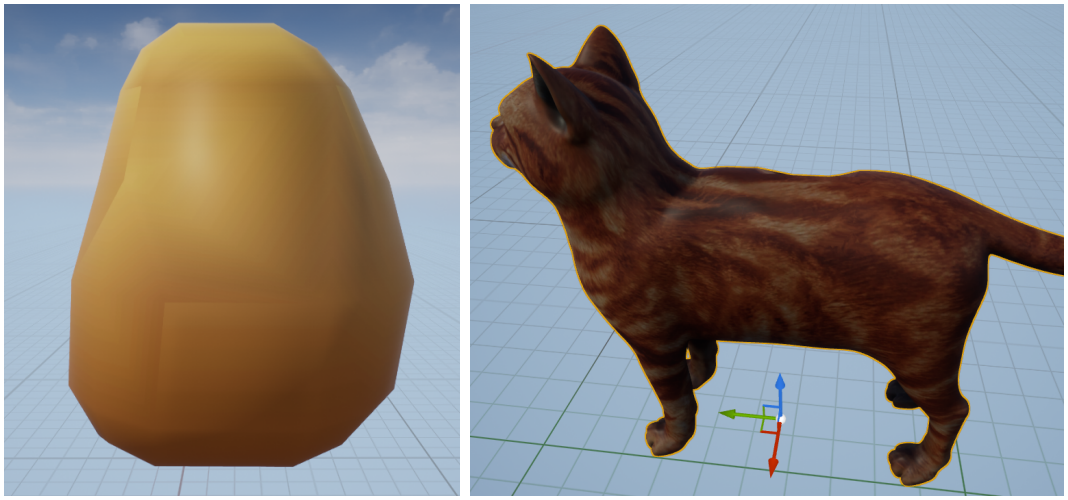
As illustrated in Figure 4.4, the Fill All Mesh Holes function successfully fills the simple holes, leaving no gaps and transforming both meshes into watertight structures. However, the patched holes are highly distinguishable from their surrounding mesh, necessitating a second step where the user must manually assign texture coordinates.

(a) (b)

Figure 4.4: Post-application visualization of the Fill All Mesh Holes function on the static meshes. (a) Shows the potato mesh after the hole has been filled, and (b) depicts the cat mesh with the hole similarly patched. The images clearly indicate where the function has acted, marking the starting point for subsequent texture coordination.

4.1.2 Static Mesh Hole Filler

Figure 4.5 shows the results of triangulation using Static Mesh Hole Filler. Both holes are filled, successfully making the meshes watertight. The patched areas are clearly less distinguishable in regards to their surrounding mesh. However, both objects exhibits features that does not fully correspond with the originals. In Figure 4.5(a), edges can be seen along the boundary of the patched mesh. Figure 4.5(b) displays what looks like a slight curvature along the patched surface as the triangulation is unsuccessful in taking the curved surface into account. The patched surface is instead a flat mesh.



(a) (b)

Figure 4.5: Completed hole filling in static meshes using the Static Mesh Hole Filler. (a) Exhibits the potato mesh after processing, with the filled area distinct from the original mesh structure. (b) Shows the cat mesh, where the patched area is noticeable and lacks the curvature of the surrounding mesh, resulting in a flat fill rather than one that integrates with the existing surface contours.

4.2 Filling Complex Holes

A comparison was also made between the two functions on their ability to find and fill complex holes. Figure 4.6(a) shows the potato with one complex hole containing three singular vertices which, after decomposition from complex to simple holes, consists of four simple holes. Figure 4.6(b) shows the cat with one complex hole containing three singular vertices which, after decomposition from complex to simple holes, consists of three simple holes.

(a) (b)

Figure 4.6: Visualization of complex holes in mesh objects. (a) Displays a potato mesh with a complex hole characterized by three singular vertices, and (b) shows a cat mesh with a similarly complex hole, highlighting the challenges such irregularities present before applying advanced hole-filling techniques.

4.2.1 Fill All Mesh Holes

As can be seen in Figure 4.7, Unreal Engine 5's method is once again able to patch both surfaces, transforming the meshes into watertight structures. The textures of the patched surfaces however, are still clearly distinguishable from their surrounding mesh.

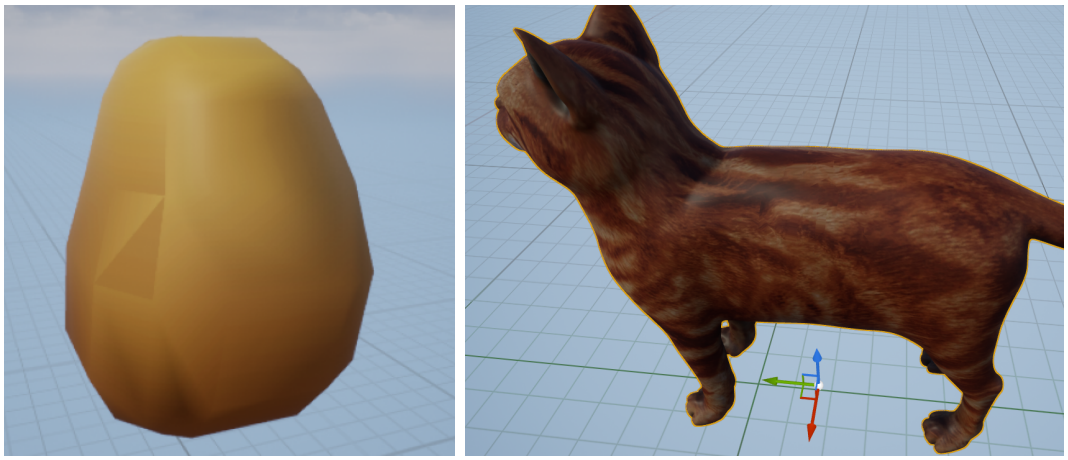
(a) (b)

Figure 4.7: (a) A potato mesh and (b) a cat mesh after undergoing the hole-filling process in Unreal Engine 5, showcasing the closed gaps. The patchwork is visible, indicating the areas where the mesh has been repaired to create watertight models.

4.2.2 Static Mesh Hole Filler

Figure 4.7 shows the result of Static Mesh Hole Filler. Both meshes have become watertight, leaving zero gaps or holes. The patched surface of the potato (Figure 4.7(a)) is very close to being indistinguishable from the surrounding mesh, although, some lines along

the boundary of the patched surface of the big hole can be seen. The patched surface of the cat (Figure 4.7(b)) is however, indistinguishable from its surrounding mesh.

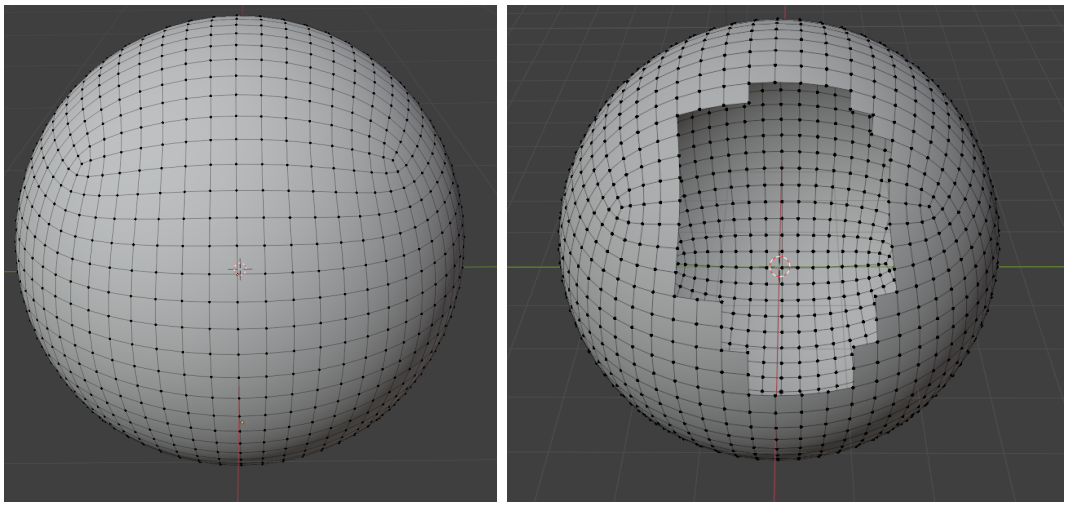


(a) (b)

Figure 4.8: (a) A potato mesh with a large hole seamlessly filled to match the surrounding surface. (b) A cat mesh with a filled section, flawlessly blending into the existing texture and structure.

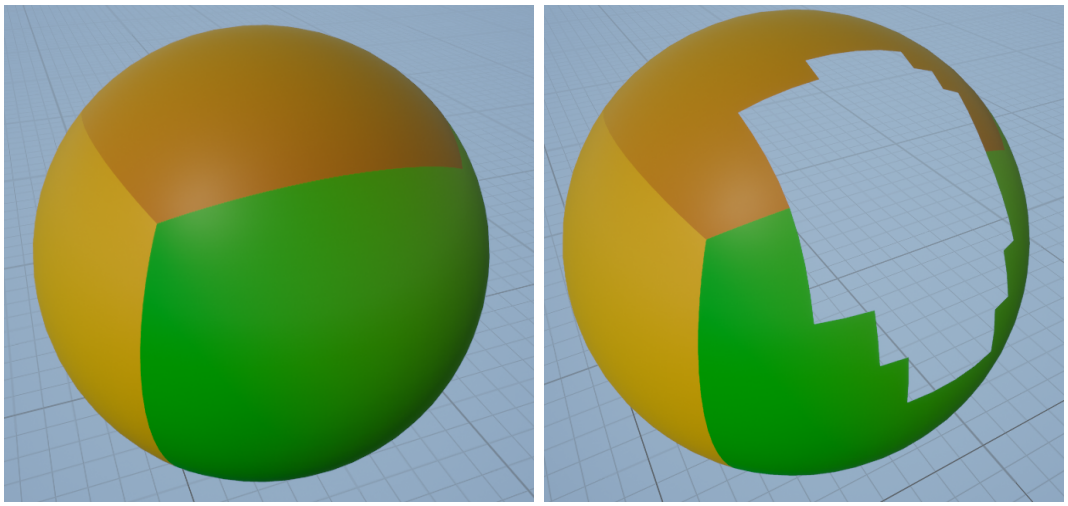
4.3 Filling A Large Hole On A Sphere

This section involves analysing the efficiency of the various stages of Static Mesh Hole Filler when used on a large non-planar hole on a sphere. Additionally, the solutions for non-manifold normals and texture coordinate assignment is also visualized and analyzed. The original sphere, which can be seen in Figure 4.9, has a total of 1,538 vertices before the hole is introduced and 1,464 after introduction. The sphere was given a colorful gradient map for its texture and the hole is present across the seam between two distinguishable colors, as can be seen in Figure 4.10.



(a) (b)

Figure 4.9: Sphere mesh structure visualized using Blender. (a) Original sphere containing 1,538 vertices. (b) Sphere after the removal of 74 vertices, containing a total of 1,464 vertices.



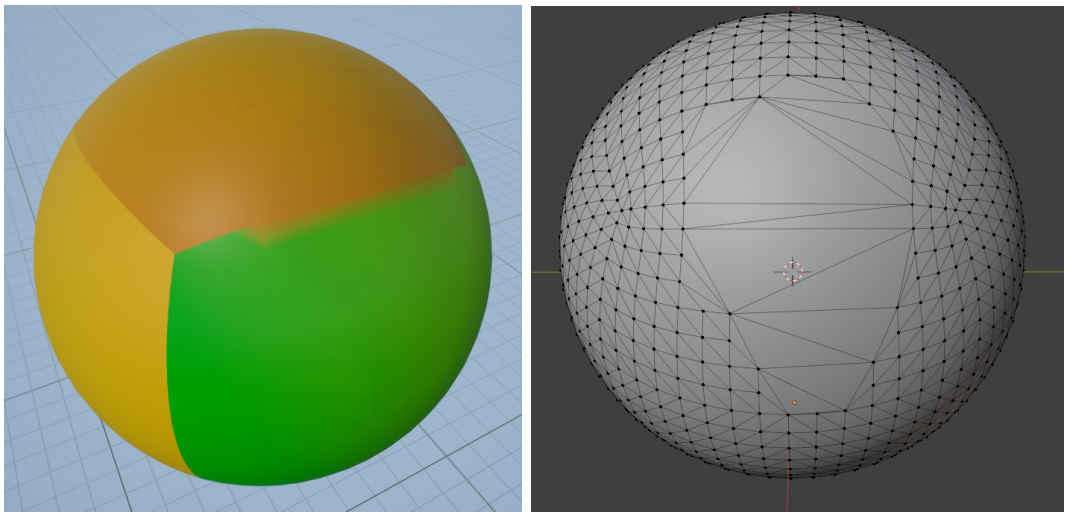
(a) (b)

Figure 4.10: Sphere visualized using Unreal Engine 4.27. (a) The original sphere with added texture. (b) The sphere containing the hole which has been placed across the seam of the orange and green colors.

4.3.1 Triangulating The Sphere

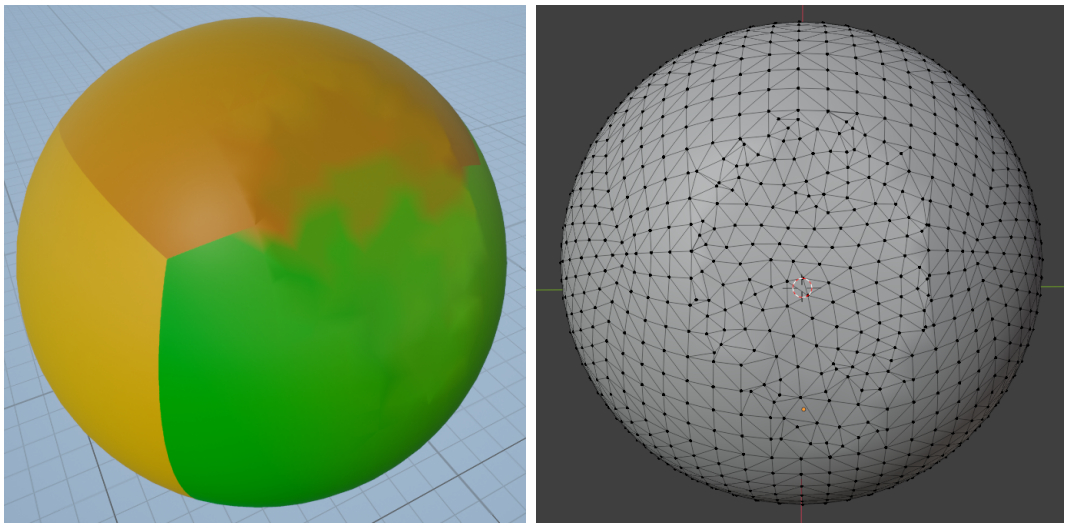
The sphere underwent triangulation via the **Area** method, which resulted in a planar patch filling the void left by the hole. This outcome is illustrated in Figure 4.11(a) for Unreal Engine and Figure 4.11(b) for Blender. Although the seam where two colors meet remains visible, it is noticeably blurred compared to its original state. The texture mapping for each triangle is derived from the vertices involved in the triangulation. Since the triangulation process does not interpolate new vertices but instead relies on the existing ones along the hole's boundary, these vertices retain their original texture coordinates from the mesh. As depicted in Figure 4.11(b), two notable triangles stretch horizontally across the middle of

the patched area, aligning with the seam. These triangles' vertices are located on opposite sides of the seam, resulting in a smudged appearance in the texture at this junction.



(a) (b)

Figure 4.11: (a) The triangulated sphere in Unreal Engine. (b) The triangulated sphere in Blender.



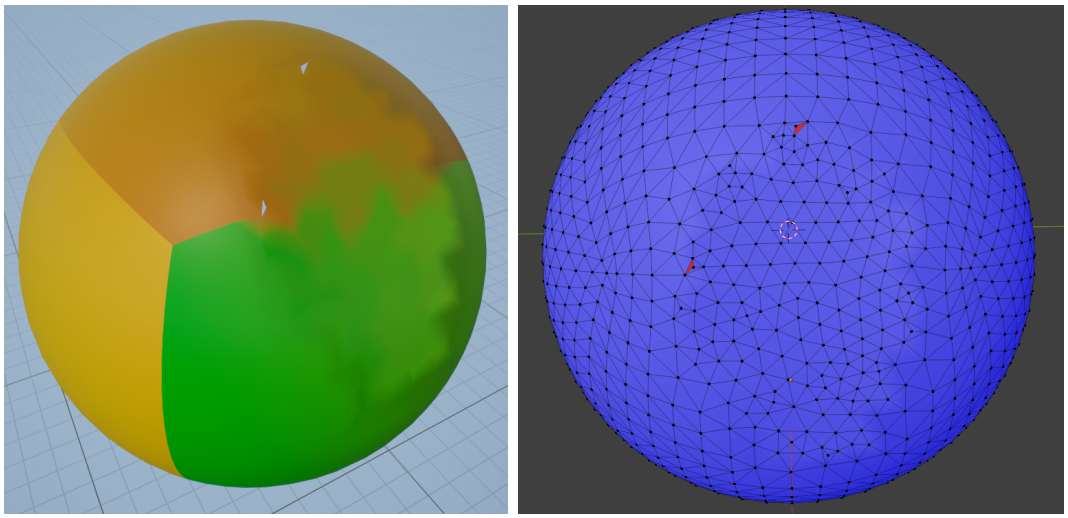
(a) (b)

Figure 4.12: (a) The refined sphere in Unreal Engine. (b) The refined sphere in Blender.

4.3.2 Refining The Sphere

Subsequent to the triangulation, the sphere's patched area underwent a refinement process. The outcomes are depicted in Figure 4.12(a) for the texture and 4.12(b) for the mesh structure. The vertex density on the refined patch appears to be consistent with the surrounding mesh, as observed in Figure 4.12(b). However, the original structure and patterning of the mesh have not been preserved. The texture seam, on the other hand, has undergone a significant blending effect, rendering it virtually indistinct, as evident in Figure 4.12(a). The introduction of new vertices along the seam facilitated the progressive

diffusion of the texture, effectively softening and dispersing any harsh transitions across the patch's surface.



(a) (b)

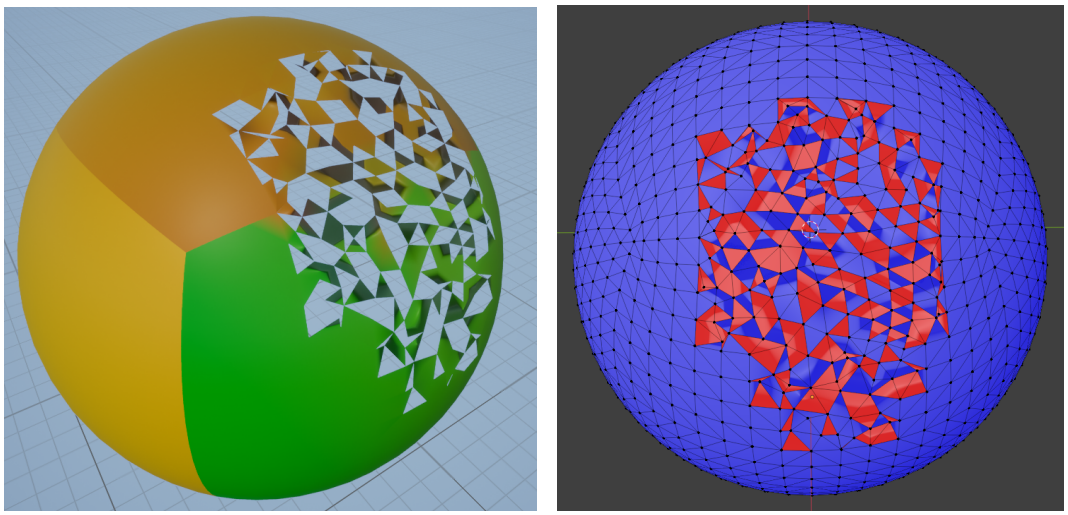
Figure 4.13: (a) The faired sphere in Unreal Engine. (b) The faired sphere in Blender, including face orientation. Blue triangles contain normals facing outwards from the mesh, red triangle normals face inwards.

4.3.3 Fairing The Sphere

The refinement of the sphere's surface was followed by a fairing step aimed at harmonizing the patched area with the sphere's overall curvature. Employing the **Scale** method across 50 iterations, the fairing sought to imbue the patch with the same curved quality as the rest of the mesh. The results of this process are presented in Figure 4.13. Unfortunately, as Figure 4.13(a) reveals, the outcome was less than ideal. The boundary vertices of the patch appear to have been displaced outward, deviating from the mesh's interior, while the inner vertices of the patch did not seem to be significantly adjusted, resulting in a patch that remains conspicuously flatter than its adjacent regions. The objective to seamlessly smooth the patched surface was not met. Moreover, the fairing procedure led to an anomaly with two triangles exhibiting non-manifold normals, as indicated in Figure 4.13(a), where these triangles are not rendered, and Figure 4.13(b), which displays the mesh with face orientation highlighting the problematic triangles in red.

4.3.4 Non-Manifold Normal Solution

Figure 4.14 illustrates the challenge of non-manifold normals encountered in the development process. The resolution for this, detailed in Section 3.3.4, effectively ensures all triangle normals are manifold, resulting in a fully rendered triangle surface. This successful outcome is showcased in Figure 4.12(a).



(a) (b)

Figure 4.14: (a) The refined sphere with non-manifold normals in Unreal Engine. (b) The refined sphere in Blender, visualized with face orientation, where red triangles exhibit non-manifold normals that point towards the distinct inside of the mesh structure.

4.3.5 Texture Coordinate Assignment

Figure 4.15 demonstrates the outcome of refinement when Algorithm 7 is not applied, underscoring its necessity. New vertices introduced into the mesh default to texture coordinates of (0, 0), resulting in the appearance of a black texture on the patched surface. The effectiveness of employing Algorithm 7 to correctly assign texture coordinates is evident in the improved results shown in Figure 4.12(a).

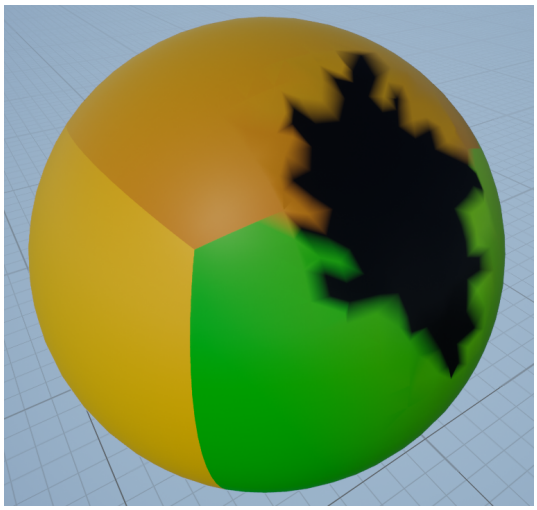


Figure 4.15: The result of refinement without the use of weighted texture coordinate assignment.

Chapter 5
Conclusions

5.1 Review of Project Goals

The primary objective of this project was to enhance Unreal Engine 4.27 by introducing a feature capable of addressing non-manifold issues in static meshes. This specifically targeted the challenge of surface holes and inward-facing triangle normals, which were causing rendering issues in the game engine. The ultimate aim was to seamlessly integrate repaired or modified surfaces, making them visually consistent with their adjacent areas.

Throughout the testing phase, where various meshes with either simple or complex holes were subjected to repair, the efficiency of the Static Mesh Hole Filler became evident. It demonstrated a solid ability in filling arbitrary amounts of smaller holes on relatively planar surfaces. Furthermore, it effectively maintained manifold normals for all newly added triangles on the repaired surface following refinement. However, a notable limitation was observed in the final stage of the process, known as fairing. As detailed in Section 4.3.3 and illustrated in Figure 4.13, fairing occasionally reintroduced non-manifold normals post-refinement. This revelation highlighted a critical challenge in the process, undermining the effort to resolve non-manifold normal issues effectively.

The function also falls short in preserving the unique characteristics of the original mesh following the hole-filling process. It struggles to adequately smooth curved surfaces, often resulting in flat, patched areas on regions that were originally non-planar, especially in the case of larger holes. Moreover, the introduction of new vertices during the refinement process tends to blur distinct texture features. This issue can lead to the patched surface becoming noticeably distinct from its surrounding area, rather than blending seamlessly as intended.

In summary, the Static Mesh Hole Filler function excels in repairing small, planar holes, yet it falls short in preserving the unique characteristics of larger holes.

5.2 Context

In the context of filling holes produced by the use of Reduction Settings in Unreal Engine, which often produces smaller holes scattered throughout the mesh, the Static Mesh Hole Filler function might be sufficient enough to avoid warranting the manual modification of hole-riddled meshes in third party software. This capability offers the potential to save considerable time typically spent on modifying static meshes in porting projects. Furthermore, it may help in avoiding the need to compromise performance for the sake of visual integrity.

5.3 Future Works

Given that the function is not yet fully developed and contains bugs, the initial step in any potential continuation of this work would involve addressing the identified issues in the refinement method, as well as improving the min-max dihedral angle triangulation process. Furthermore, considering the subpar results of the fairing process, it becomes imperative to investigate different strategies for enhancing surface smoothing techniques.

Moreover, there is considerable room for enhancement in the user experience aspect. At present, users are not notified about the success or failure of operations. Incorporating information about the processed mesh, such as the number of detected holes, the number of successfully filled holes, and the count of added vertices, edges and triangles, would significantly improve functionality. Additionally, separating the hole detection stage from the main function and executing it earlier could inform users about existing holes, thereby offering them the option to choose which ones to address. This approach would be particularly beneficial in cases where the various holes present on the mesh may necessitate distinct methods of repair, allowing for a more tailored and effective operation process.

Beyond refining the current functionality, there is significant potential to expand the function by developing solutions for additional causes of non-manifold status in static meshes. A straightforward application of this could involve addressing issues like intersecting triangles, removing those that contribute to this problem. However, the complexity in this scenario arises from the need to discern which triangles are the culprits, and which are essential to maintaining the structural integrity of the mesh.

Acronyms

LOD Level Of Detail

References

[1] Gill Barequet and Micha Sharir. Filling gaps in the boundary of a polyhedron. *Comput. Aided Geom. Des.*, 12:207–229, 1995. (Cited on page 7.)

[2] Epic Games. Unreal engine automatic lod generation documentation. `https://docs.unrealengine.com/4.27/en-US/WorkingWithContent/Types/StaticMeshes/HowTo/AutomaticLODGeneration/`, 2023. Accessed: 2023-12-14. (Cited on page 5.)

[3] Epic Games. Unreal engine editor utility widget. `https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/UMG/UserGuide/EditorUtilityWidgets/`, 2023. Accessed: 2024-01-03. (Cited on page 19.)

[4] Epic Games. Unreal engine fill all mesh holes. `https://docs.unrealengine.com/5.0/en-US/BlueprintAPI/GeometryScript/Repair/FillAllMeshHoles/`, 2023. Accessed: 2023-12-27. (Cited on page 21.)

[5] Epic Games. Unreal engine geometry script. `https://docs.unrealengine.com/5.0/en-US/geometry-script-reference-in-unreal-engine/`, 2023. Accessed: 2024-01-03. (Cited on page 21.)

[6] Epic Games. Unreal engine static meshes documentation. `https://docs.unrealengine.com/4.27/en-US/WorkingWithContent/Types/StaticMeshes/`, 2023. Accessed: 2023-12-14. (Cited on page 4.)

[7] Epic Games. Unreal engine static meshes editor ui. `https://docs.unrealengine.com/4.27/en-US/WorkingWithContent/Types/StaticMeshes/Editor/`, 2023. Accessed: 2024-01-03. (Cited on page 6.)

[8] Epic Games. Unreal engine ufunctions. `https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/GameplayArchitecture/Functions/`, 2023. Accessed: 2023-12-20. (Cited on page 15.)

[9] Peter Liepa. Filling Holes in Meshes. In Leif Kobbelt, Peter Schroeder, and Hugues Hoppe, editors, *Eurographics Symposium on Geometry Processing*. The Eurographics Association, 2003. (Cited on pages 7, 8, 9, 10, 11, and 17.)

[10] Ron Pfeifle and Hans-Peter Seidel. Triangular b-splines for blending and filling of polygonal holes. In *Proceedings of the Conference on Graphics Interface '96*, GI '96, page 186â193, CAN, 1996. Canadian Information Processing Society. (Cited on page 9.)

[11] Mauhing Yip, Annette Stahl, and Christian Schellewald. Robust hole-detection in triangular meshes irrespective of the presence of singular vertices. *ArXiv*, abs/2311.12466, 2023. (Cited on pages 12, 13, and 17.)